## 基础语法

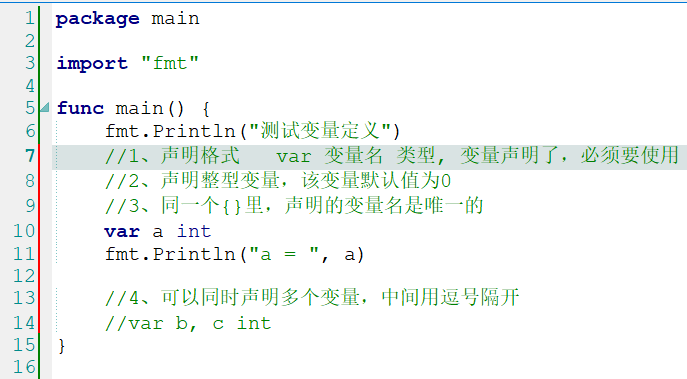
### 数据类型

| **类型** | **名称** | **长度** | **零值** | **说明** |
| --- | --- | --- | --- | --- |
| bool | 布尔类型 | 1 | false | 其值不为真即为假，不可以用数字代表true或false |
| byte | 字节型 | 1 | 0 | uint8别名 |
| rune | 字符类型 | 4 | 0 | 专用于存储unicode编码，等价于uint32 |
| int, uint | 整型 | 4或8 | 0 | 有符号32位或无符号64位 |
| int8 | 整型 | 1 | 0 | -128 ~ 127, |
| uint8 | 整型 | 1 | 0 | 0 ~ 255 |
| int16 | 整型 | 2 | 0 | -32768 ~ 32767, |
| uint16 | 整型 | 2 | 0 | 0 ~ 65535 |
| int32 | 整型 | 4 | 0 | -2147483648 到 2147483647 |
| uint32 | 整型 | 4 | 0 | 0 到 4294967295(42亿) |
| int64 | 整型 | 8 | 0 | 0 到 18446744073709551615（1844京） |
| uint64 | 整型 | 8 | 0 | -9223372036854775808到 9223372036854775807 |
| float32 | 浮点型 | 4 | 0.0 | 小数位精确到7位 |
| float64 | 浮点型 | 8 | 0.0 | 小数位精确到15位 |
| complex64 | 复数类型 | 8 |  |  |
| complex128 | 复数类型 | 16 |  | 64 位实数和虚数 |
| uintptr | 整型 | 4或8 |  | ⾜以存储指针的uint32或uint64整数 |
| string | 字符串 |  | "" | utf-8字符串 |

### 变量

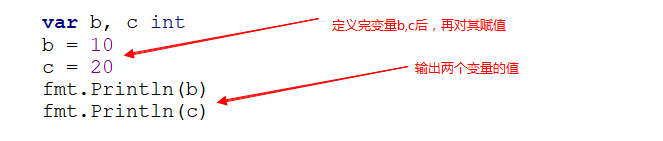
#### 变量声明

Go语言引入了关键字var，而类型信息放在变量名之后，示例如下：



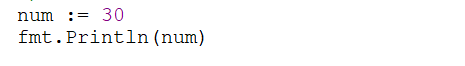
#### 变量初始化

示例如下：



#### 自动推导类型

Go语言可以使用自动推导类型直接创建变量并赋值，具体示例如下：



自动推导类型，不用通过var声明变量，不用指定类型，直接在变量名后面跟”:”号，同时完成赋值。那么GO会根据所赋的值自动推导出变量的类型。如果给num变量赋值为小数，那么该变量的类型为小数类型（浮点）。

通过比较，我们发现这种方式比前面的赋值方式要简单方便。这种方式，在Go开发过程中最常用的方式。

#### 多重赋值 匿名变量

（1）多重赋值

C:\Users\wangc\AppData\Local\Temp\8e17cfb6-1a19-410b-a97b-b1f18fb50dce.tmp

（2）匿名变量

\_匿名变量，丢弃数据不进行处理, \_匿名变量配合函数返回值使用才有价值，目前大家只需要知道其语法，后面学到函数时，我们在强调匿名变量的使用场景。

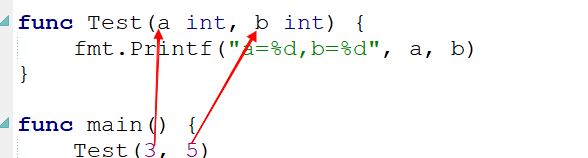
变量的语法：



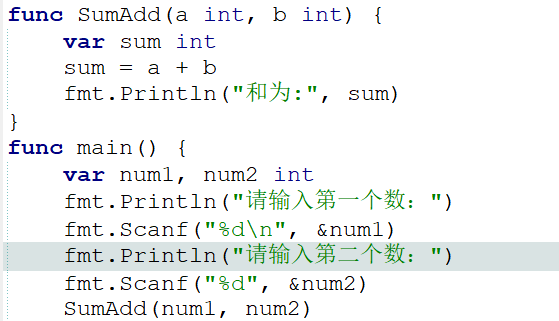
### 函数

#### 普通参数列表

基本语法如下：

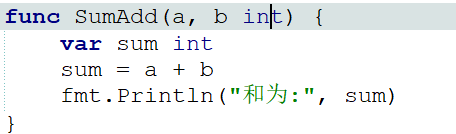


函数定义和使用



**注意：与C语言不同的是， 类型声明在变量名之后。**

也可以写成如下的形式：



但是，不建议这样定义，因为不够清晰。

请看如下方法的定义：

func MyFunc05(a, b string, c float64, d, e int) {

}

推荐方法定义成如下形式，非常清晰。

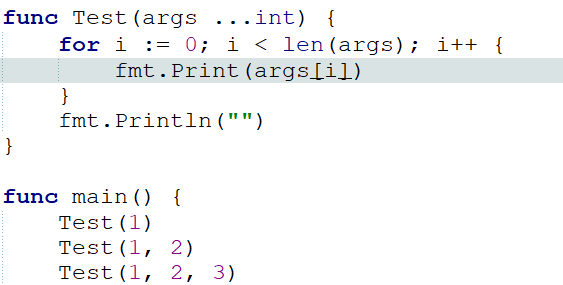
func MyFunc06(a string, b string, c float64, d int, e int) {

}

#### 不定参数列表

无法确定参数的个数时Go语言中也可以使用不定参函数（变参函数）

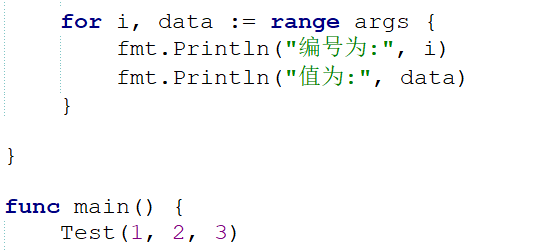
可以通过如下的方式来定义函数:



Test()函数是变参函数，可以传多个值。

在Test()函数内，我们可以通过len()函数来计算出args集合中存储了多少个数，通过for循环将该集合中的数全部输出，在输出时我们通过下标的方式将args集合中的值输出的。

除了可以使用len()函数计算出集合中存储的数据的个数外，还有另外一种输出的方式可以使用range关键字。如下所示：

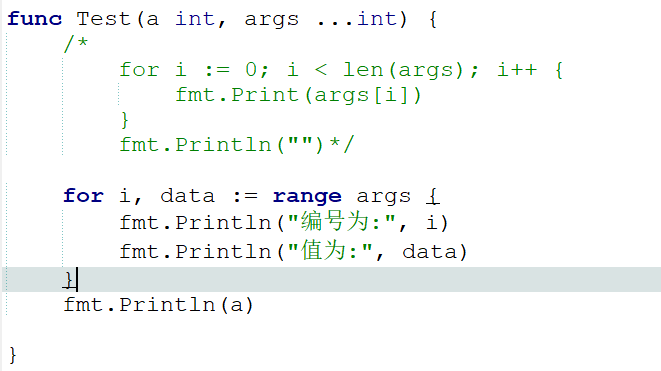


range会从集合中返回两个数，第一个是对应的坐标，赋值给了变量i，第二个就是对应的值，赋值了变量data

以上两种输出集合的方式，在Go开发过程中都可以使用。

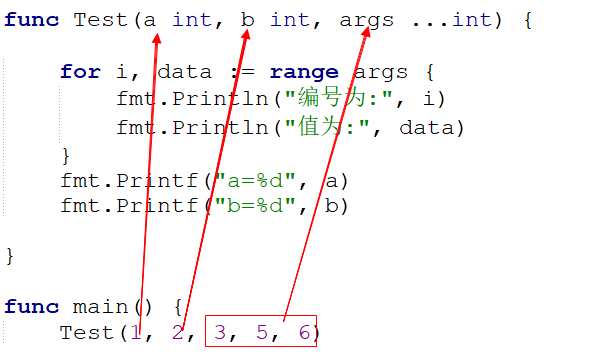
在使用不定参数时，要注意几个问题：

第一：不定参**只能**放在形参中的最后一个参数。例如：



上面我们定义了一个Test()函数，该函数第一个参数是一个普通的整型类型，第二个参数是不定参数。那么不定参数args，必须放在后面，整型类型的参数a必须放在前面。**如果两者的位置进行互换，在编译的时候就会出错**：

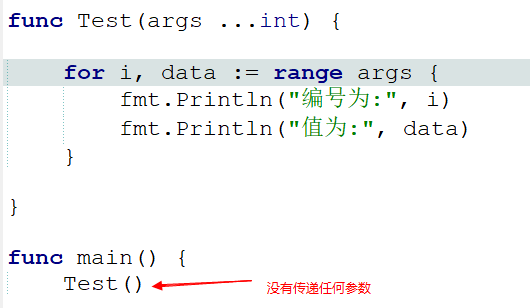
第二：在函数调用时，固定参数必须传值，不定参可根据需要来决定是否传值。



定义Test()函数，有两个固定参数，和不定参数，在main()函数中，对Test函数进行调用，实参1,2这两个值分别传递给形参中的a和b,实参中的3,5,6传递给了不定参数。

但是如果我们在调用Test()函数时，一个参数都不传递，会报错。

如果Test()函数只有不定参数，没有固定参数，那么在调用时，可以根据需要来决定是否进行传值。例如：

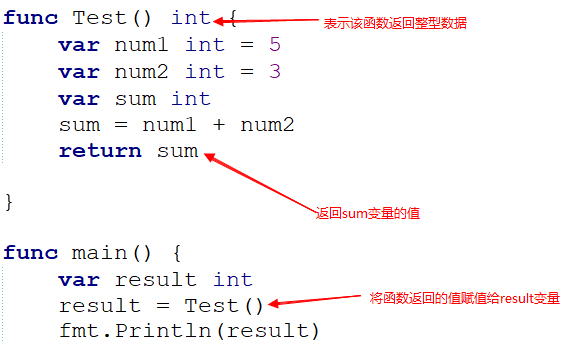


我们在调用Test()函数时，没有传递任何的参数，程序并没有出现任何的错误，当然也没有输出结果。

#### 函数返回值

1. **返回值函数基本定义**

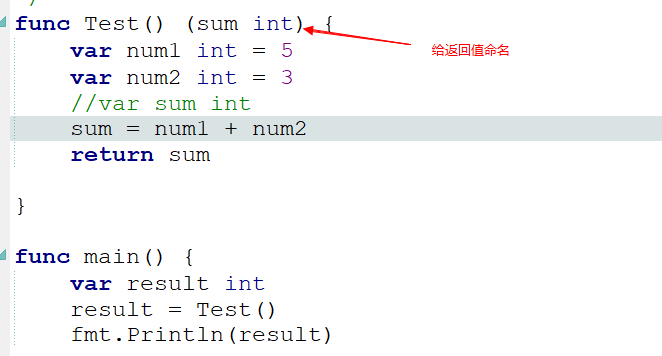
基本语法如下：



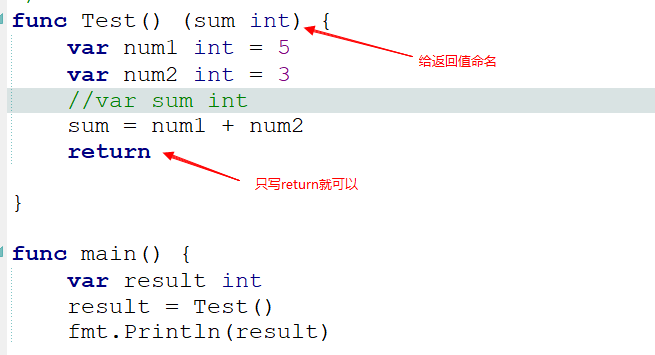
Go语言在定义函数Test( )时，返回值int加在形参列表后面。

接收Test()函数返回的值方法与C语言同。

GO语言也提供了另外一种语法定义具有返回值的函数，如：

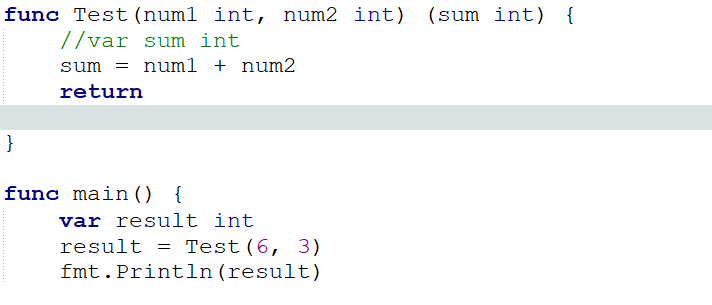


第三种写法：



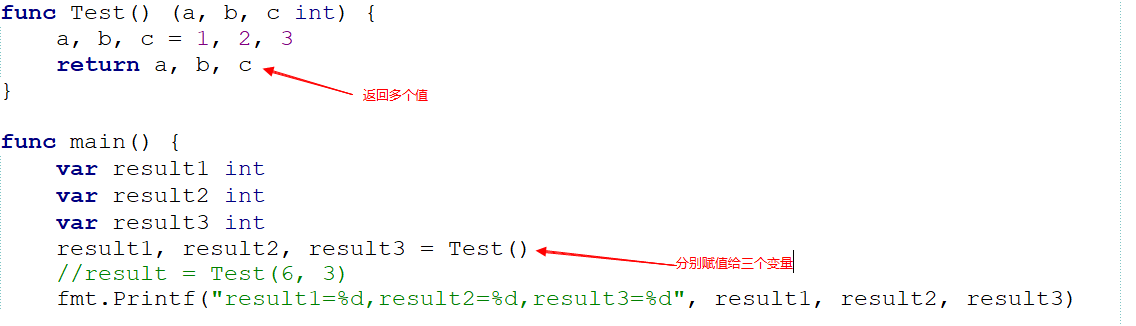
以上几种写法，都可以。

案例演示如下：

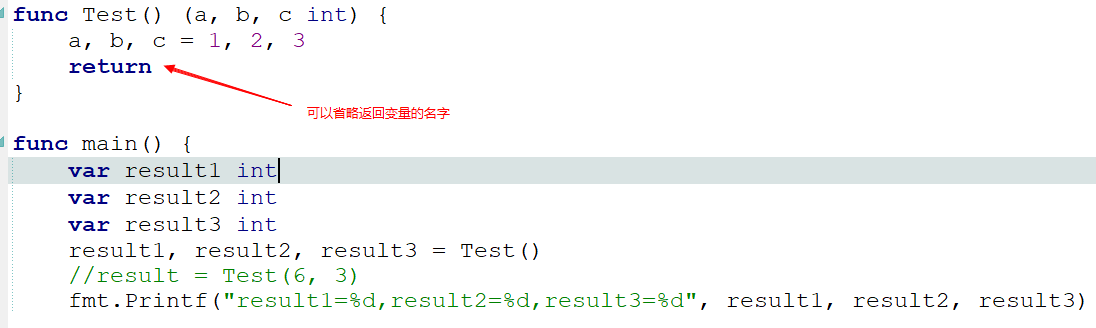


1. **返回多个值**

**Go语言的函数可以返回多个值呢！ 如：**



第二种写法：



**看一个用户注册的示例：**

****

#### **延迟调用defer**

1. **defer基本使用**

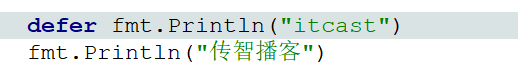
函数定义完成后，只有调用函数才能够执行，并且一经调用立即执行。例如：

C:\Users\wangc\AppData\Local\Temp\34943793-28e0-457b-aa7d-8ce21cda41dd.tmp

先输出“itcast”，然后再输出“传智播客”

关键字 defer ⽤于延迟一个函数的执行。注意，defer语句只能出现在函数的内部。

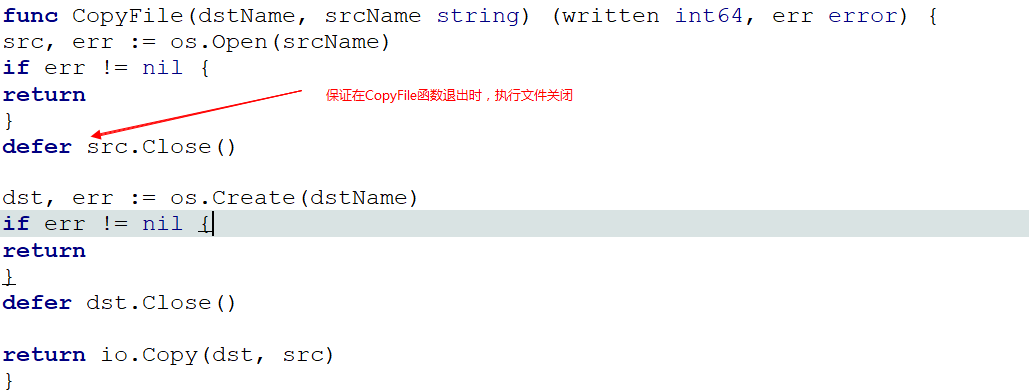
基本用法如下：



输出的结果为，先输出“传智播客”，然后输出”itcast”.

defer的应用场景：文件操作，先打开文件，执行读写操作，最后关闭文件。为了保证文件的关闭能够正确执行，可以使用defer.

看一下文件操作的伪代码，来体会一下关于defer的 场景。

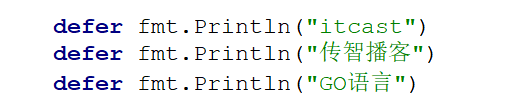


通过在文件关闭函数之前加defer，不管什么情况下都会执行文件关闭的操作。代码逻辑越复杂，defer使用越重要。

同理，进行网络编程时，关闭网络链接时，也会用到defer。

1. **defer执行顺序**

**先看如下程序执行结果是：**



**执行的结果是：**

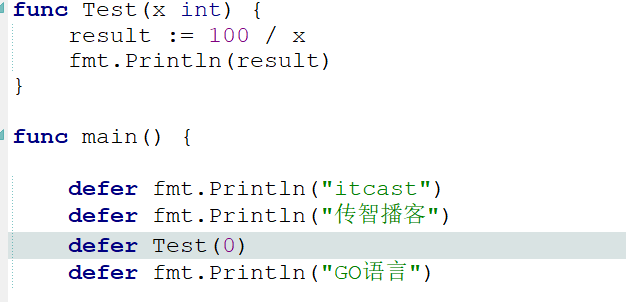
**GO语言**

**传智播客**

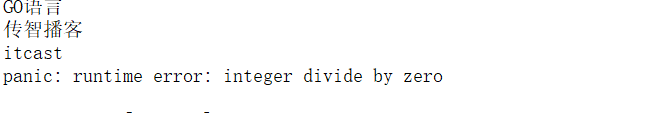
**Itcast**

总结：如果一个函数中有多个defer语句，它们会以LIFO（后进先出）的顺序执行。

**执行如下程序：**



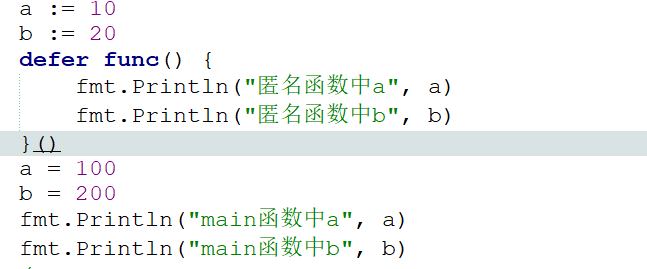
**执行结果：**



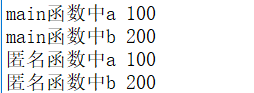
**即使函数或某个延迟调用发生错误，这些调用依旧会被执⾏。**

1. **defer与匿名函数结合使用**

**看以下程序：**

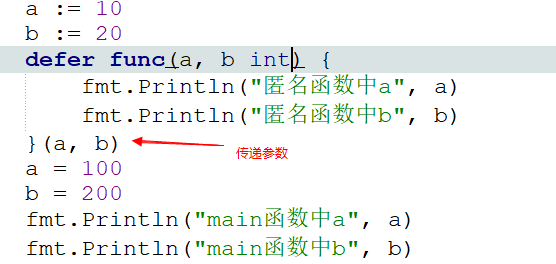


**执行的结果如下：**

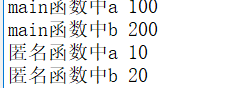


**defer会延迟函数的执行，虽然立即调用了匿名函数，但是该匿名函数不会执行，等整个main( )函数结束之前在去调用执行匿名函数，所以输出结果如上所示。**

**将程序做如下修改：**



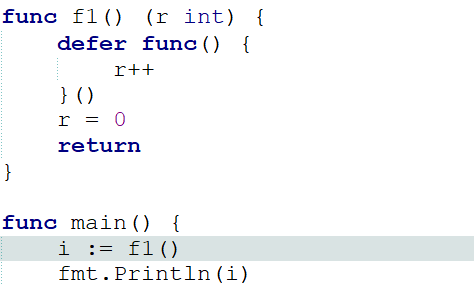
**该程序的执行结果如下：**



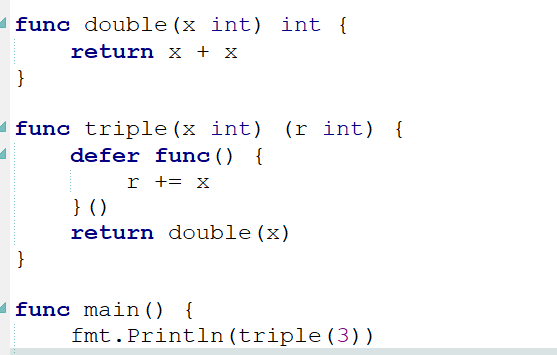
**由于匿名函数前面加上了defer。 所以，匿名函数没有立即执行。但程序从上开始执行当执行到匿名函数时，虽然没有立即调用执行匿名函数，已经完成了参数的传递。**

**思考：以下程序的输出结果：**

**（1）阅读程序，分析结果**



**（2）阅读程序，分析结果**



## 工程管理

### 工作区介绍

通过前面函数的学习，我们能够体会到函数的优势，就是可以将不同的功能放在不同的函数中实现，主函数（main( )）可以直接调用。这样结构非常的清晰，也非常方面代码的管理。如果我们把所有的代码都写在main( )函数中，会出现什么样的情况呢？

代码混乱，非常不容易管理。但是现在我们面临了另外一个问题就是：我们所有自己定义的函数都写在了一个文件中，

如果我们做的项目代码量越来越多，那么该文件会变的非常臃肿，代码也会变得非常难管理。所以，我们在开发中，除了要定义函数，同时还要将代码放在不同的文件中。例如：我们定义了一个UserInfo.go文件，里面包含了用户的添加函数，修改函数，删除函数等操作。

这就涉及到项目的工程管理也就是怎样对项目中的文件进行管理。

为了更好的管理项目中的文件，要求将文件都要放在相应的文件夹中。GO语言规定如下的文件夹如下：

 （1）src目录：用于以代码包的形式组织并保存Go源码文件。（比如：.go .c .h .s等）

 （2）pkg目录：用于存放经由go install命令构建安装后的代码包（包含Go库源码文件）的“.a”归档文件。

 （3）bin目录：与pkg目录类似，在通过go install命令完成安装后，保存由Go命令源码文件生成的可执行文件。

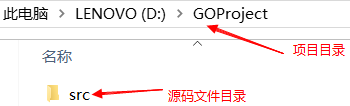
以上目录称为工作区，工作区其实就是一个对应于特定工程的目录。

目录src用于包含所有的源代码，是Go命令行工具一个强制的规则，而pkg和bin则无需手动创建，如果必要Go命令行工具在构建过程中会自动创建这些目录

### 创建同级目录

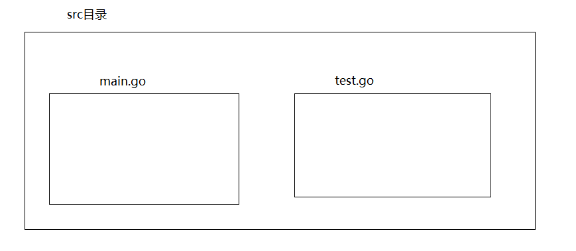
（2.1）创建src目录，在改目录下创建go源码文件

（2.1.1）在项目文件夹下新建src目录，如下图所示：



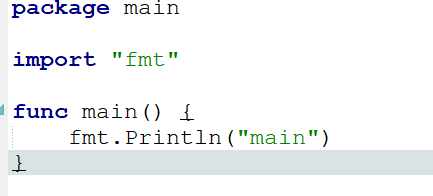
在D盘GoProject目录下创建了src目录。

（2.1.2）在改目录下创建不同的go源码文件，如下图所示：

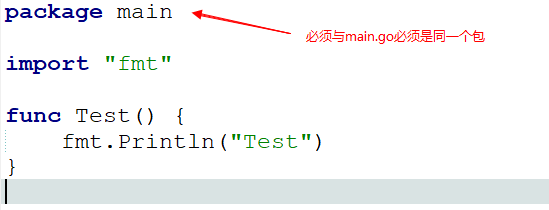


在src目录下创建main.go文件和test.go文件（注意：这个两个文件是在同一个目录下面，都是在src目录下面）

main.go 文件下的代码如下所示：



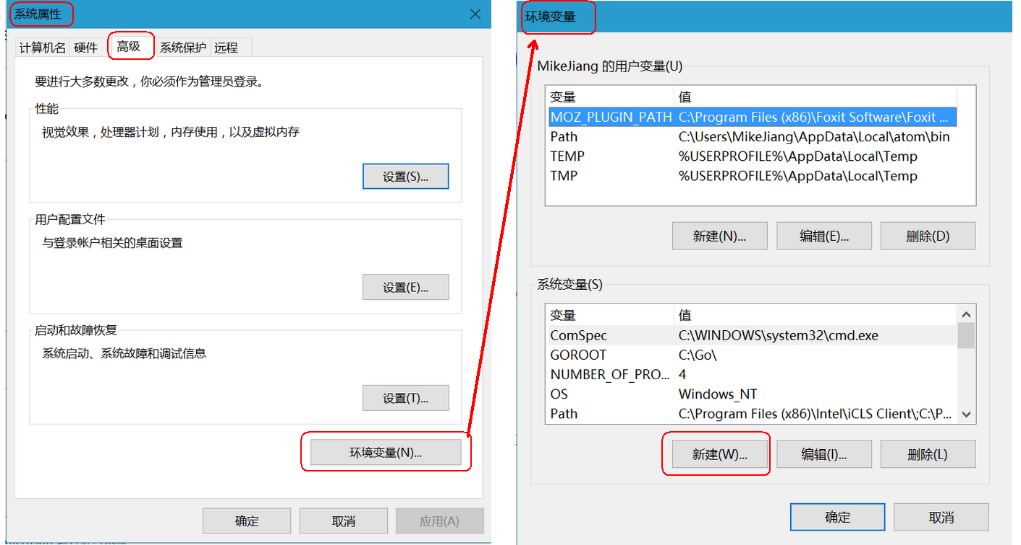
test.go 文件下的代码如下所示：

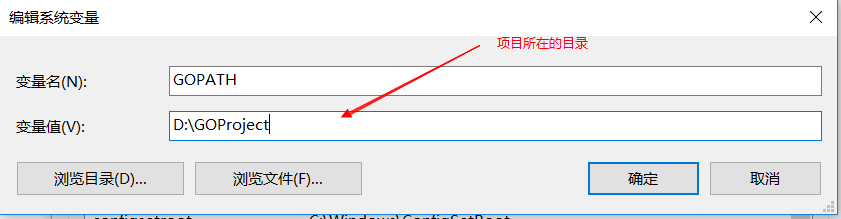


现在已经完成两个文件代码的编写，接下来的问题是，我们怎样在main.go文件中的入口函数main( )中调用test.go文件中的Test( )函数呢？这就需要设置环境变量GOPATH属性。如果要完成不同文件中函数的调用，必须设置GOPATH，否则，即使文件处于同一工作目录（工作区）下，也是无法完成调用的。

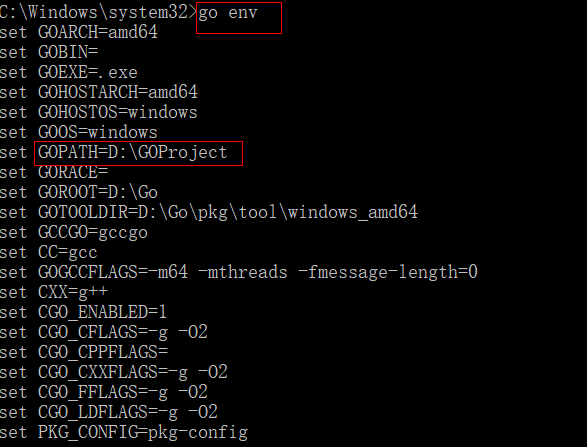
（2.2）GOPATH设置

GOPATH设置的具体步骤如下：

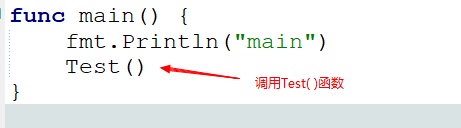




配置完成后，可以测试一下是否配置成功



在main.go文件中完成对test.go文件中函数的调用



最后编译执行。

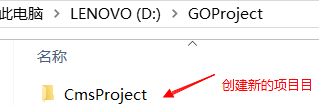
注意：同一个目录下不能定义不同的package。

### 创建不同级目录

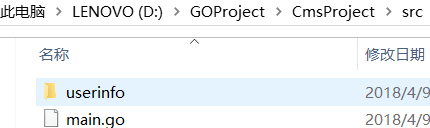
在上一小节中，将不同的go源代码文件都放在了同一个目录下面，如果将go源代码文件放在不同的目录下面应该怎样进行处理呢？

具体的步骤如下：

（3.1）新建项目目录，如下图所示

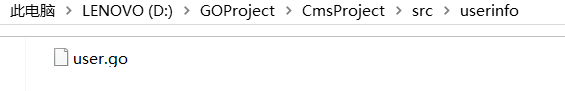


在“CmsProject”目录下面，创建src目录，在src目录下面创建如下文件目录与文件.

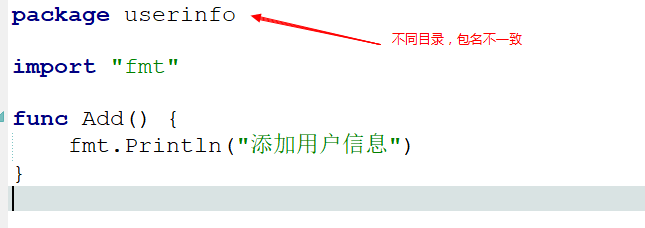


main.go 定义的是入口函数main( ).

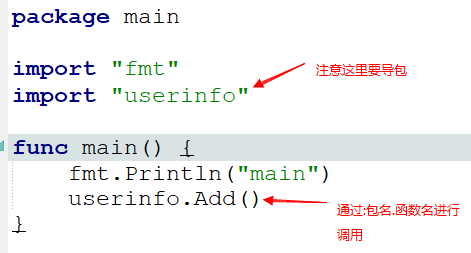
userinfo 文件夹下定义的是user.go文件。



user.go 文件中的代码如下：



main.go 文件中的代码如下：

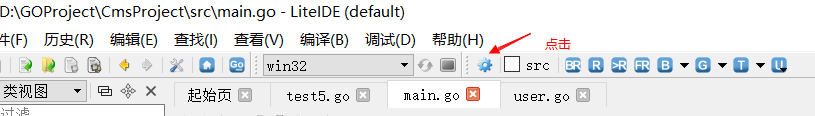


通过以上两个文件中的代码，可以总结出如下几点:

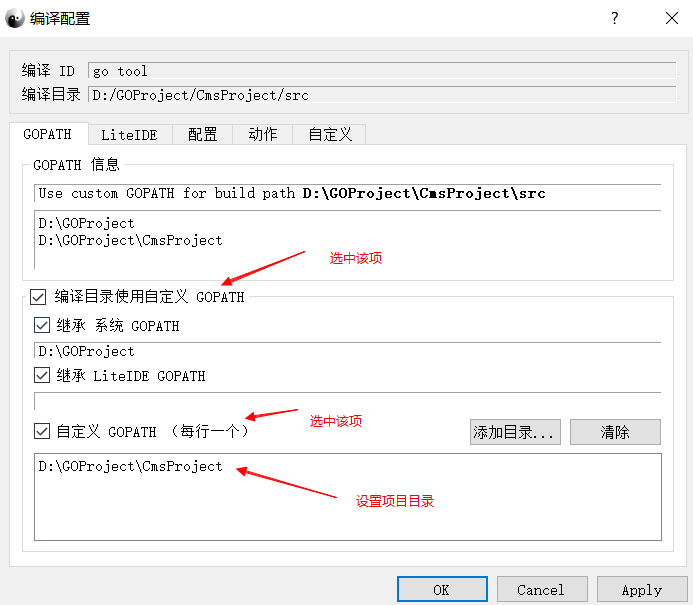
第一：不同目录，包名不一致（自定义包）。

第二：main.go中调用user.go中的方法时，一定要导包，并且调用的方式是：包名.函数名 的方式

（3.2）要想运行以上代码，还需要设置GOPATH环境变量的值，这样设置比较麻烦，可以通过以下方法进行快速设置：

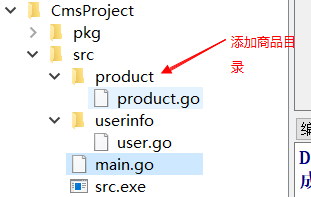


会打开如下的窗口，然后进行设置。

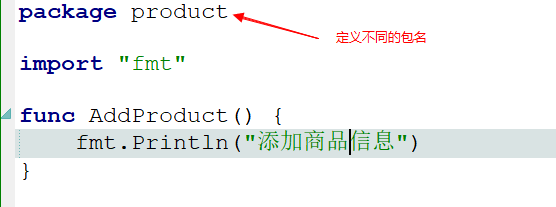


（3.3）注意：user.go文件中的函数名首字母必须大写，如果改成小写在main.go中无法进行调用

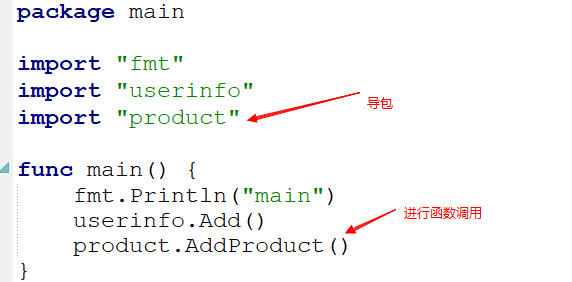
这种不同级目录应用，在以后的项目开发中使用频率非常高。例如：上面我们的案例中，可以将用户管理的操作放在userinfo目录下，商品管理模块可以再定义一个目录,例如:product.如下图所示：



product.go中的代码如下：



main.go中的代码如下：

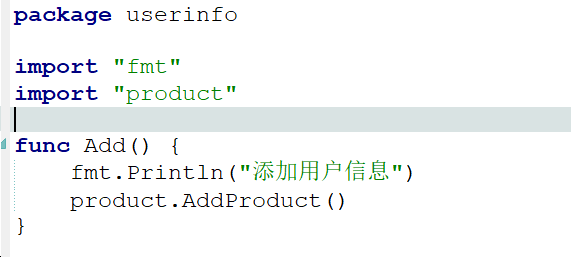


（3.4）关于包的问题

在第一天的课程中，我们已经讲解了关于包的概念，包就是一个标识，标志着代码是来自哪儿，对代码进行管理。

所以，在main( )函数中要使用相应的函数，必须进行导包，然后根据包名去调用相应的函数。

通过上面的代码，我们也能够体会出“包”的优势，就是可以在userinfo包中定义名叫Add( )方法，在product包中也可以定义Add( )方法，但是在main( )函数中进行调用时，通过包名进行调用，就可以很清楚Add( )方法来自哪个包，不会造成混乱，和名称的冲突。并且相关的功能代码，放在一个包中，可以很好的被复用。例如：可以在userinfo包中使用product,如下图所示：



但是我们创建的的自定义包最好放在GOPATH的src目录下，在Go语言中，代码包中的源码文件名可以是任意的。但是，这些任意名称的源码文件都必须以包声明语句作为文件中的第一行，每个包都对应一个独立的名字空间。

包中成员以名称⾸字母⼤⼩写决定访问权限：

 public: ⾸字母⼤写，可被包外访问

 private: ⾸字母⼩写，仅包内成员可以访问

注意：同一个目录下不能定义不同的package。

（3.5）导包的问题

在上面的案例中，要使用包，必须要进行导入，可以通过关键字进行import进行导入，它会告诉编译器你想引用该包内的代码。

如果导入的是标准库中的包（GO语言自带，例如:”fmt”包）会在安装 Go 的位置找到。 Go 开发者创建的包会在 GOPATH 环境变量指定的目录里查找。所以，import关键字的作用就是查找包所在的位置。

如果编译器查遍 GOPATH 也没有找到要导入的包，那么在试图对程序执行 run 或者 build

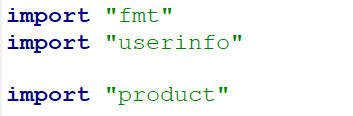
的时候就会出错。

注意：如果导入包之后，未调用其中的函数或者类型将会报出编译错误。

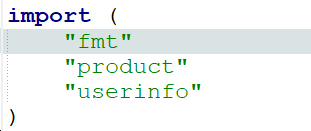
2017-09-24_150149

我们常规的导包方式是用import关键子一个个导入。

例如：



表示导入三个包，有GO语言自带的包，也有我们自定义的包。但是，这种写法可能比较麻烦，所以为了简化也可以采用如下的方式进行导包：



这种方式，使用的频率是非常高的。

## 切片

### 切片简述

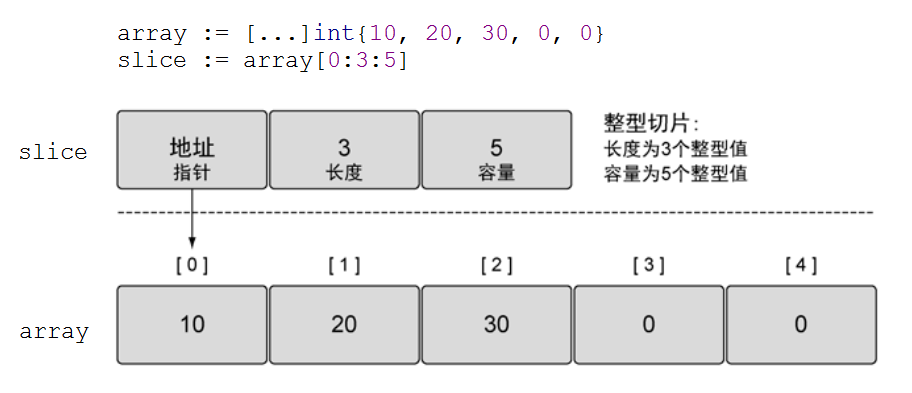
数组的长度在定义之后无法再次修改；数组是值类型，每次传递都将产生一份副本。显然这种数据结构无法完全满足开发者的真实需求。Go语言提供了数组切片（slice）来弥补数组的不足。

Slice（切片）代表变长的序列，序列中每个元素都有相同的类型。一个slice类型一般写作[]T，其中T代表slice中元素的类型；slice的语法和数组很像，只是没有固定长度而已。

数组和slice之间有着紧密的联系。一个slice是一个轻量级的数据结构，提供了访问数组子序列（或者全部）元素的功能，而且slice的底层确实引用一个数组对象。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。

**切片并不是数组或数组指针，它通过内部指针和相关属性引⽤数组⽚段，以实现变⻓⽅案。**

slice并不是真正意义上的动态数组，而是一个引用类型。slice总是指向一个底层array，slice的声明也可以像array一样，只是不需要长度。



### 创建切片

slice和数组的区别：声明**数组**时，[ ]**内**写明了数组的**长度**或使用**...**自动计算长度，而声明**slice**时，**[ ]内**没有任何字符。经常使用的切片创建方法：

1. 自动推导类型创建slice

s1 := [] int {1, 2, 3, 4} 创建 有 4 个元素的切片，分别为：1234

1. 借助**make**创建 slice，格式：make(切片类型，长度，容量)

s2 := make([]int, 5, 10) len(s2) = 5, cap(s2) = 10

1. **make时，没有指定容量，那么 长度==容量**

s3 := make([]int, 5) len(s3) = 5, cap(s3) = 5

**func** main() {  
 s1 := [] int {1, 2, 3, 4} // 创建 有4个元素的切片  
 fmt.Println(**"s1="**, s1)  
  
 s2 := make([]int, 5, 10) // 借助make创建 slice，格式：make(切片类型，长度，容量)  
 s2[4] = 7  
 //s2[5] = 9 // 报错：panic: runtime error: index out of range  
 fmt.Println(**"s2="**, s2)  
 fmt.Printf(**"len(s2)=%d, cap(s2)=%d\n"**, len(s2), cap(s2))  
  
 s3 := make([]int, 5) // make时，没指定容量，那么 长度 == 容量  
 s3[2] = 3  
 fmt.Println(**"s3="**, s3)  
 fmt.Printf(**"len(s2)=%d, cap(s2)=%d\n"**, len(s3), cap(s3))  
}

注意：make只能创建slice、map和channel，并且返回一个有初始值(非零)的对象。

### 切片操作

切片截取

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

截取可表示为s[low：high：max]。low：表示下标的起点。 high：表示下标的终点（左闭右开，不包括此下标）。 长度 len = high – low。容量 cap = max – low。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。内置的len()和cap() 函数分别返回slice的长度和容量。

示例说明：

    array := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

| **操作** | **结果** | **len** | **cap** | **说明** |
| --- | --- | --- | --- | --- |
| **array[:6:8]** | [0 1 2 3 4 5] | 6 | 8 | 省略 low |
| **array[5:]** | [5 6 7 8 9] | 5 | 5 | 省略 high、 max |
| **array[:3]** | [0 1 2] | 3 | 10 | 省略 high、 max |
| **array[:]** | [0 1 2 3 4 5 6 7 8 9] | 10 | 10 | 全部省略 |

切片和底层数组关系

**func** main() {  
 arr := [] int {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
 s1 := arr[2:5] // 从arr[2]开始，取 5-2 个元素，组成切片s1。  
 fmt.Println(**"s1="**, s1) // s1= [2 3 4]  
  
 s1[1] = 666 // 这样将arr数组中 3 --> 666。  
 fmt.Println(**"arr="**, arr) // arr= [0 1 2 666 4 5 6 7 8 9]  
  
 s2 := s1[2:7] // 从s1[2]开始， 取 7-2 个元素，组成 s2。  
 fmt.Println(**"s2="**, s2) // 实际上还是取的 数组arr。 s2= [4 5 6 7 8]  
  
 s2[2] = 777 // 这会将arr中的 6 --> 777  
 fmt.Println(**"arr="**, arr) // arr= [0 1 2 666 4 5 777 7 8 9]  
}

利用数组创建切片。切片在操作过程中，是直接操作原数组。因此，在**go语言中，我们常常使用切片代替数组。**

### Append函数

append() 函数可以向 slice **尾部**添加数据，可以自动为切片扩容。常常会返回新的 slice 对象：

**var** s1 []int //创建nil切片，或者：s1 := make([]int, 0)

    s1 = append(s1, 1) //追加1个元素

    s1 = append(s1, 2, 3) //追加2个元素

    s1 = append(s1, 4, 5, 6) //追加3个元素

    fmt.Println(s1) //[1 2 3 4 5 6]

    s2 := make([]int, 5)

    s2 = append(s2, 6)

    fmt.Println(s2) //[0 0 0 0 0 6]

    s3 := []int{1, 2, 3}

    s3 = append(s3, 4, 5)

    fmt.Println(s3) //[1 2 3 4 5]

append函数会智能的将底层数组的容量增长，一旦超过原底层数组容量，通常以2倍（1024以下）容量重新分配底层数组，并复制原来的数据。因此，使用append 给切片做扩充时，切片的地址可能发生变化。但，数据都被重新保存了，不影响使用。

**func** main() {  
 s := make([]int, 0, 1)  
 c := cap(s)  
 **for** i := 0; i < 100; i++ {  
 s = append(s, i)  
 **if** n := cap(s); n > c {  
 fmt.Printf(**"cap: %d -> %d\n"**, c, n)  
 c = n  
 }  
 }}

输出结果如下：

cap: 1 -> 2

cap: 2 -> 4

cap: 4 -> 8

cap: 8 -> 16

cap: 16 -> 32

cap: 32 -> 64

cap: 64 -> 128

### 切片做函数参数

切片作为函数参数时，是怎样传递呢？是传值，还是传引用呢？

**func** testFunc(s []int) { // 切片做函数参数  
 s[0] = -1 // 直接修改 main中的 slice  
}  
  
**func** main() {  
 slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
 fmt.Println(slice)  
  
 testFunc(slice) // 传引用  
 fmt.Println(slice)  
}

从上面的示例，粗略分析，好似切片在作为函数参数时，直接传递的是“引用”。真的是这样吗？我们再看下面一个demo：

**func** myAppend( a []int) {  
 a = append(a, 11)  
 fmt.Printf(**"in Append -- len:%d, cap:%d, data:%v\n"**, len(a), cap(a), a)  
}  
  
**func** main() {  
 a := []int{6, 7, 8}  
  
 fmt.Printf(**"1 -- len:%d, cap:%d, data:%v\n"**, len(a), cap(a), a)  
 myAppend(a)  
  
 fmt.Printf(**"2 -- len:%d, cap:%d, data:%v\n"**, len(a), cap(a), a)  
}

结合我们前面强调的切片本质，尝试分析上述代码执行结果。

### 切片练习

**练习1：写一个函数，给定一个字符串列表，返回不包含空字符串的列表， 如：**

{**"red"**, **""**, **"black"**, **""**, **""**, **"pink"**, **"blue"**}

——> {**"red"**, **"black"**, **"pink"**, **"blue"**}

**练习2：写一个函数，消除[]string中重复字符串，如：**

{**"red"**, **"black"**, **"red"**, **"pink"**, **"blue"**, **"pink"**, **"red"**, **"blue"**}

——> {**"red"**, **"black"**, **"pink"**, **"blue"**}

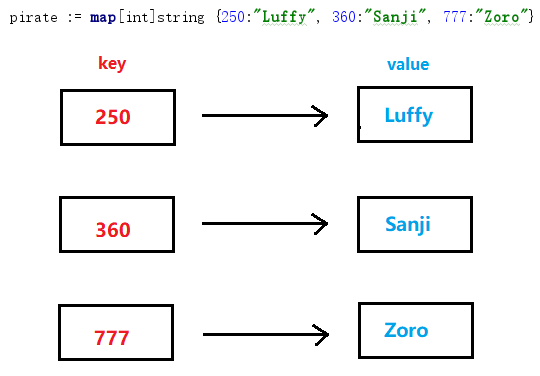
**练习3：要删除slice中间的某个元素并保存原有的元素顺序， 如：**

{5, 6, **7**, 8, 9} ——> {5, 6, 8, 9}

## Map

### map简述

Go语言中的map(映射、字典)是一种内置的数据结构，它是一个无序的key-value对的集合，比如以身份证号作为唯一键来标识一个人的信息。Go语言中并没有提供一个set类型，但是map中的key也是不相同的，可以用map实现类似set的功能。



map格式为：

**map**[keyType]valueType

在一个map里所有的键都是唯一的，而且必须是支持==和!=操作符的类型，**切片、函数**以及包含切片的结构类型这些类型由于具有**引用**语义，**不能作为映射的键**，使用这些类型会造成编译错误：

    dict := **map**[ []string ]int{} //err, invalid map key type []string

map值可以是任意类型，没有限制。map里所有键的数据类型必须是相同的，值也必须如此，但键和值的数据类型可以不相同。

**注意：**map是无序的，我们无法决定它的返回顺序，所以，每次打印结果的顺序有可能不同。

### 创建、初始化map

创建map

**var** m1 **map**[int]string //只是声明一个map，没有初始化, 为空(nil)map

    fmt.Println(m1 == nil) //true

    //m1[1] = "Luffy" //nil的map不能使用err, panic: assignment to entry in nil map

    m2 := **map**[int]string{} //m2, m3的创建方法是等价的

    m3 := make(**map**[int]string)

    fmt.Println(m2, m3) //map[] map[]

    m4 := make(**map**[int]string, 10) //第2个参数指定容量

    fmt.Println(m4) //map[]

创建m4的方法指定了map的初始创建容量。 与slice类似，后期在使用过程中，map可以自动扩容。只不过map更方便一些，不用借助类似append的函数，直接赋值即可。如，m1[17] = "Nami"。赋值过程中，key如果与已有map中key重复，会将原有map中key对应的value覆盖。

但是！对于map而言，可以使用len()函数，但**不能使用cap()函数**。

初始化map

也可以直接指定初值，要保证key不重复。

    //1、定义同时初始化

**var** m1 **map**[int]string = **map**[int]string{1: "Luffy", 2: "Sanji"}

    fmt.Println(m1) //map[1:Luffy 2:Sanji]

    //2、自动推导类型 :=

    m2 := **map**[int]string{1: "Luffy", 2: "Sanji"}

    fmt.Println(m2)

### 常用操作

赋值

    m1 := **map**[int]string{1: "Luffy", 2: "Sanji"}

    m1[1] = "Nami" //修改

    m1[3] = "Zoro" //追加， go底层会自动为map分配空间

    fmt.Println(m1) //map[1:Nami 2:Sanji 3:Zoro]

    m2 := make(**map**[int]string, 10) //创建map

    m2[0] = "aaa"

    m2[1] = "bbb"

    fmt.Println(m2) //map[0:aaa 1:bbb]

    fmt.Println(m2[0], m2[1]) //aaa bbb

遍历

Map的迭代顺序是不确定的，并且不同的哈希函数实现可能导致不同的遍历顺序。在实践中，遍历的顺序是随机的，每一次遍历的顺序都不相同。这是故意的，每次都使用随机的遍历顺序可以强制要求程序不会依赖具体的哈希函数实现。

    m1 := **map**[int]string{1: "Luffy", 2: "Sanji"}

    //遍历1，第一个返回值是key，第二个返回值是value

**for** k, v := **range** m1 {

        fmt.Printf("%d ----> %s\n", k, v)

        //1 ----> Luffy

        //2 ----> yoyo

    }

    //遍历2，第一个返回值是key，第二个返回值是value（可省略）

**for** k := **range** m1 {

        fmt.Printf("%d ----> %s\n", k, m1[k])

        //1 ----> Luffy

        //2 ----> Sanji

    }

有时候可能需要知道对应的元素是否真的是在map之中。可以使用下标语法判断某个key是否存在。map的下标语法将产生两个值，其中第二个是一个布尔值，用于报告元素是否真的存在。

如果key存在，第一个返回值返回value的值。第二个返回值为 true。

    value, ok := m1[1]

    fmt.Println("value = ", value, ", ok = ", ok) //value = mike , ok = true

如果key不存在，第一个返回值为空，第二个返回值为false。

    value2, has := m1[3]

    fmt.Println("value2 = ", value2, ", has = ", has) //value2 = , has = false

删除

使用delete()函数，指定key值可以方便的删除map中的k-v映射。

m1 := **map**[int]string{1: "Luffy", 2: "Sanji", 3: "Zoro"}

**for** k, v := **range** m1 { //遍历，第一个返回值是key，第二个返回值是value

        fmt.Printf("%d ----> %s\n", k, v)

    }

    //1 ----> Sanji

    //2 ----> Sanji

    //3 ----> Zoro

    delete(m1, 2) //删除key值为2的map

**for** k, v := **range** m1 {

        fmt.Printf("%d ----> %s\n", k, v)

}

    //1 ----> Luffy

    //3 ----> Zoro

delete()操作是安全的，即使元素不在map中也没有关系；如果查找删除失败将返回value类型对应的零值。

如：

    delete(m1, 5) //删除key值为5的map

**for** k, v := **range** m1 {

        fmt.Printf("%d ----> %s\n", k, v)

}

    //1 ----> Luffy

    //3 ----> Zoro

map输出结果依然是原来的样子，且不会有任何错误提示。

### map做函数参数

与slice 相似，在函数间传递映射并不会制造出该映射的一个副本，不是值传递，而是**引用传递**：

**func** DeleteMap(m **map**[int]string, key int) {

    delete(m, key) //删除key值为2的map

**for** k, v := **range** m {

        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)

}

    //len(m)=2, 1 ----> Luffy

    //len(m)=2, 3 ----> Zoro

}

**func** main() {

    m := **map**[int]string{1: "Luffy", 2: "Sanji", 3: "Zoro"}

    DeleteMap(m, 2) //删除key值为2的map

**for** k, v := **range** m {

        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)

}

    //len(m)=2, 1 ----> Luffy

    //len(m)=2, 3 ----> Zoro

}

### map做函数返回值

值传递

返回的依然是**引用**：

**func** test() **map**[int]string {

// m1 := map[int]string{1: "Luffy", 2: "Sanji", 3: "Zoro"}  
 m1 := make(**map**[int]string, 1) // 创建一个初创容量为1的map  
 m1[1] = **"Luffy"** m1[2] = **"Sanji"** // 自动扩容  
 m1[67] = **"Zoro"** m1[2] = **"Nami"** // 覆盖 key值为2 的map  
 fmt.Println(**"m1 = "**, m1)  
 **return** m1  
}

**func** main() {  
 m2 := test() // 返回值 —— 传引用  
 fmt.Println(**"m2 = "**, m2)  
}

输出：

m1 = map[1:Luffy 2:Nami 67:Zoro]

m2 = map[2:Nami 67:Zoro 1:Luffy]

**练习3： 封装 wcFunc() 函数。接收一段英文字符串str。返回一个map，记录str中每个“词”出现次数的。**

**如："I love my work and I love my family too"**

**输出：**

**family : 1**

**too : 1**

**I : 2**

**love : 2**

**my : 2**

**work : 1**

**and : 1**

**提示：使用 strings.Fields() 函数可提高效率。**

## 指针

### Go语言中的指针

Go语言保留了指针，但与C语言指针有所不同。主要体现在：

* 默认值 nil
* 操作符 "&" 取变量地址， " \* " 通过指针访问目标对象
* 不支持指针运算，不支持 "->" 运算符，直接⽤ "." 访问目标成员

**func** main() {  
 **var** a int = 100 // 声明 int 变量 a  
 fmt.Printf(**"&a = %p\n"**, &a) // "&" 取 a 地址  
  
 **var** p \*int = nil // 声明变量p, 类型为 \*int  
 p = &a // p指向a  
 fmt.Printf(**"p = %p\n"**, p)  
 fmt.Printf(**"a = %d, \*p = %d\n"**, a, \*p)  
 \*p = 324 //\*p操作指针所指向的内存，即为a  
 fmt.Printf(**"a = %d, \*p = %d\n"**, a, \*p)  
}

### 函数new

表达式new(T)将创建一个T类型的匿名变量，所做的是为T类型的新值分配并清零一块内存空间，然后将这块内存空间的地址作为结果返回，而这个结果就是指向这个新的T类型值的指针值，返回的指针类型为\*T。

new创建的内存空间默认值为数据类型默认值。如：new(int) 则 \*p为0，new(bool) 则 \*p为false

**func** main() {  
 **var** p1 \*int  
 p1 = new(int) //p1为\*int 类型, 指向匿名的int变量  
 fmt.Println(**"\*p1 = "**, \*p1) //\*p1 = 0  
  
 p2 := new(bool) //p2为\*bool 类型, 指向匿名的bool变量  
 fmt.Println(**"\*p2 = "**, \*p2) //\*p2 = false  
 \*p2 = **true** fmt.Println(**"\*p2 = "**, \*p2) //\*p1 = true  
}

我们只需使用new()函数，无需担心其内存的生命周期或怎样将其删除，因为Go语言的内存管理系统会帮我们打理一切。

### 指针做函数参数

**func** swap01(a, b int) {

    a, b = b, a

    fmt.Printf("swap01 a = %d, b = %d\n", a, b)

}

**func** swap02(x, y \*int) {

    \*x, \*y = \*y, \*x

}

**func** main() {

    a := 10

    b := 20

    //swap01(a, b) //值传递（传值）

    swap02(&a, &b) //地址传递（传引用）

    fmt.Printf("a = %d, b = %d\n", a, b)

}

### 内存逃逸（引用计数）

变量的生命周期指的是在程序运行期间变量有效存在的时间间隔。对于在包一级声明的变量来说，它们的生命周期和整个程序的运行周期是一致的。而相比之下，局部变量的声明周期则是动态的：每次从创建一个新变量的声明语句开始，直到该变量不再被引用为止，然后变量的存储空间可能被回收。函数的参数变量和返回值变量都是局部变量。它们在函数每次被调用的时候创建。

Go语言的自动垃圾收集器是如何知道一个变量是何时可以被回收的呢？基本的实现思路是，从每个包级的变量和每个当前运行函数的每一个局部变量开始，通过指针或引用的访问路径遍历，是否可以找到该变量。如果不存在这样的访问路径，那么说明该变量是不可达的，也就是说它是否存在并不会影响程序后续的计算结果。

因为一个变量的有效周期只取决于是否可达，因此一个循环迭代内部的局部变量的生命周期可能超出其局部作用域。同时，局部变量可能在函数返回之后依然存在。

编译器会自动选择在栈上还是在堆上分配局部变量的存储空间，但可能令人惊讶的是，这个选择并不是由用var还是new声明变量的方式决定的。

var global \*int

func f() {

var x int

x = 1

global = &x

}

func g() {

y := new(int)

\*y = 1

}

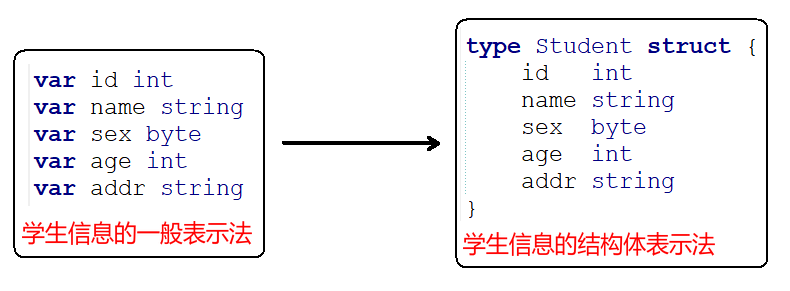
f函数里的x变量必须在堆上分配，因为它在函数退出后依然可以通过包一级的global变量找到，虽然它是在函数内部定义的；用Go语言的术语说，这个x局部变量从函数f中逃逸了。相反，当g函数返回时，变量\*y将是不可达的，也就是说可以马上被回收的。因此，\*y并没有从函数g中逃逸，编译器可以选择在栈上分配\*y的存储空间（译注：也可以选择在堆上分配，然后由Go语言的GC回收这个变量的内存空间），虽然这里用的是new方式。其实在任何时候，你并不需为了编写正确的代码而要考虑变量的逃逸行为，要记住的是，逃逸的变量需要额外分配内存，同时对性能的优化可能会产生细微的影响。

Go语言的自动垃圾收集器对编写正确的代码是一个巨大的帮助，但也并不是说你完全不用考虑内存了。你虽然不需要显式地分配和释放内存，但是要编写高效的程序你依然需要了解变量的生命周期。例如，如果将指向短生命周期对象的指针保存到具有长生命周期的对象中，特别是保存到全局变量时，会阻止对短生命周期对象的垃圾回收（从而可能影响程序的性能）。

## 结构体

### 结构体类型

有时我们需要将不同类型的数据组合成一个有机的整体，如：一个学生有学号/姓名/性别/年龄/地址等属性。显然单独定义以上变量比较繁琐，数据不便于管理。



结构体是一种聚合的数据类型，它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。每个数据称为结构体的成员。

### 结构体初始化

普通变量

**type** Student **struct** {

    id int

    name string

    sex byte

    age int

    addr string

}

**func** main() {

    //1、顺序初始化，必须每个成员都初始化

**var** s1 Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    s2 := Student{2, "Sanji", 'f', 20, "EastSea"}

    //s3 := Student{2, "Nami", 'm', 20} //err, too few values in struct initializer

    //2、指定初始化某个成员，没有初始化的成员为零值

    s4 := Student{id: 2, name: "Zoro"}

}

指针变量

**type** Student **struct** {

    id int

    name string

    sex byte

    age int

    addr string

}

**func** main() {

**var** s5 \*Student = &Student{3, "Nami", 'm', 16, "EastSea"}

    s6 := &Student{4, "ro", 'm', 3, "NorthSea"}

}

### 使用结构体成员

普通变量

    //===============结构体变量为普通变量

    //1、打印成员

**var** s1 Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    //结果：id = 1, name = Luffy, sex = m, age = 18, addr = EastSea

    fmt.Printf("id = %d, name = %s, sex = %c, age = %d, addr = %s\n", s1.id, s1.name, s1.sex, s1.age, s1.addr)

    //2、成员变量赋值

**var** s2 Student

    s2.id = 2

    s2.name = "Sanji"

    s2.sex = 'f'

    s2.age = 16

    s2.addr = "EastSea"

    fmt.Println(s2) //{2 yoyo 102 16 EastSea}

指针变量

    //===============结构体变量为指针变量

    //3、先分配空间，再赋值

    s3 := new(Student)

    s3.id = 3

    s3.name = "Nami"

    fmt.Println(s3) //&{3 Nami 0 0 }

    //4、普通变量和指针变量类型打印

**var** s4 Student = Student{4, "Sanji", 'm', 18, "EastSea"}

    fmt.Printf("s4 = %v, &s4 = %v\n", s4, &s4) //s4 = {4 Sanji 109 18 sz}, &s4 = &{4 Sanji 109 18 EastSea}

**var** p \*Student = &s4

    //p.成员 和(\*p).成员 操作是等价的

    p.id = 5

    (\*p).name = "ro"

    fmt.Println(p, \*p, s4) //&{5 ro 109 18 EastSea} {5 ro 109 18 EastSea} {5 ro 109 18 EastSea}

在Go语言中，普通结构体变量 和 结构体指针变量发访问成员的方法一致。不需要加以区分。

### 结构体比较

如果结构体的全部成员都是可以比较的，那么结构体也是可以比较的，那样的话两个结构体将可以使用 == 或 != 运算符进行比较，但不支持 > 或 < 。

**func** main() {

    s1 := Student{1, "Luffy", 'm', 18, "EastSea"}

    s2 := Student{1, "Luffy", 'm', 18, "EastSea"}

    fmt.Println("s1 == s2", s1 == s2) //s1 == s2 true

    fmt.Println("s1 != s2", s1 != s2) //s1 != s2 false

}

### 作函数参数

传值

**func** printValue(stu Student) {

    stu.id = 250

    //printValue stu = {250 Luffy 109 18 s EastSea}

    fmt.Println("printValue stu = ", stu)

}

**func** main() {

**var** s Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    printValue(s) //值传递，形参修改不会影响到实参值

    fmt.Println("main s = ", s) //main s = {1 Luffy 109 18 EastSea}

}

传参过程中，实参会将自己的值拷贝一份给形参。因此结构体“传值”操作几乎不会在实际开发中被使用到。近乎100%的使用都采用“传址”的方式，将结构体的**引用**传递给所需函数。

传引用

**func** printPointer(p \*Student) {

    p.id = 250

    //printPointer p = &{250 Luffy 109 18 EastSea}

    fmt.Println("printPointer p = ", p)

}

**func** main() {

**var** s Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    printPointer(&s) //传引用(地址)，形参修改会影响到实参值

    fmt.Println("main s = ", s) //main s = {250 Luffy 109 18 EastSea}

}

**练习： 封装一个有 string、bool、int、[]string 类型的结构体 Person。**

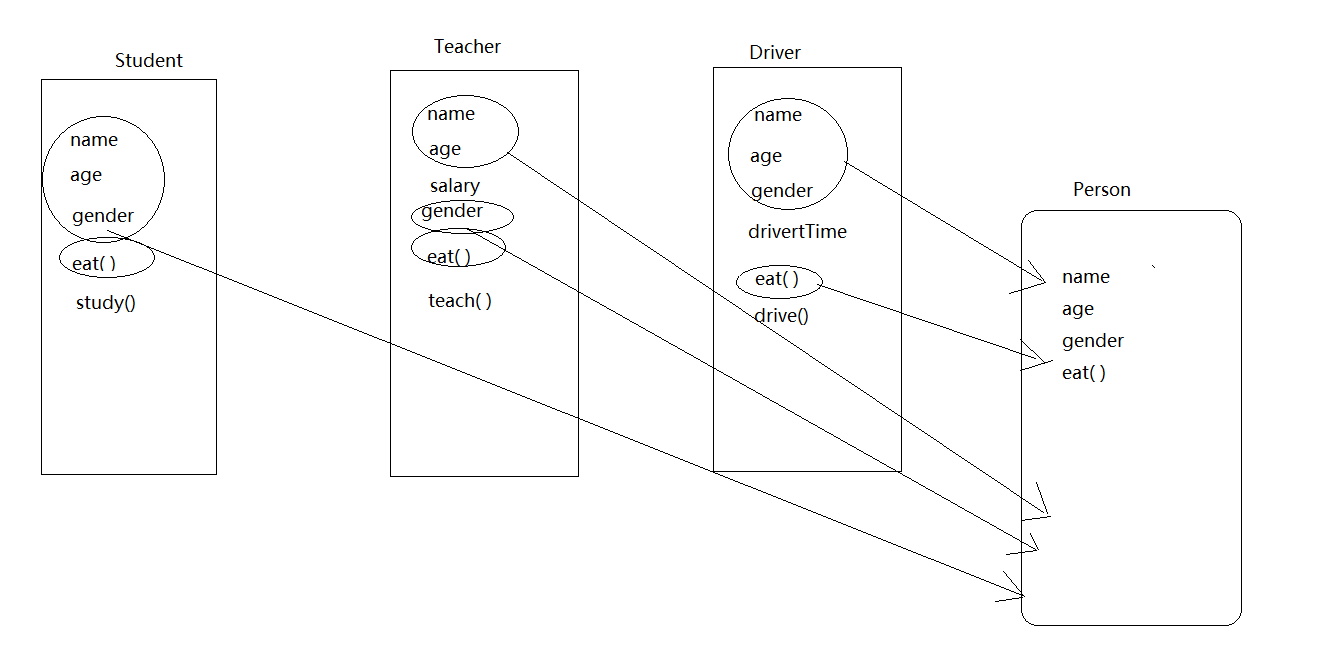
**main函数中声明结构体变量，调用initPerson函数完成结构体赋值，并在main中查看赋值结果。**

## 类

GO语言中的面向对象在某些概念上和其它的编程语言还是有差别的。GO语言中没有类(class)的概念,但是我们可以将结构体比作为类，因为在结构体中可以添加属性（成员），方法（函数）。

先来说一下“继承”，

### 继承



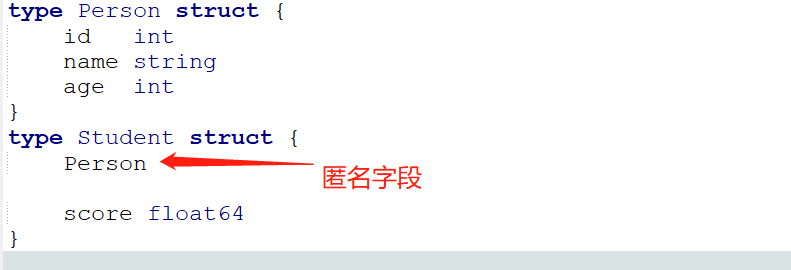
根据上面的图，我们发现学生类(结构体)，讲师类(结构体)等都有共同的成员（属性和方法），这样就存在重复，所以我们把这些重复的成员封装到一个父类（结构体）中。然后让学生类(结构体)和讲师类(结构体)继承父类(结构体)

接下来，我们可以先将公共的属性，封装到父类（结构体）中实现继承,关于方法(函数)的继承后面再讲。

#### **匿名字段创建与初始化**

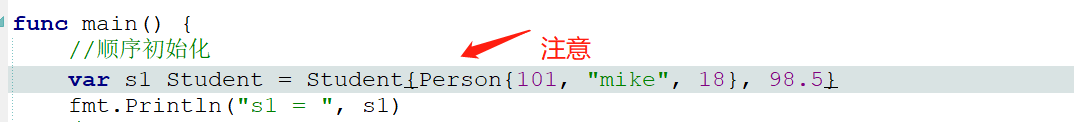
那么怎样实现属性的继承呢？

可以通过匿名字段(也叫匿名组合)来实现，什么是匿名字段呢？通过如下使用，大家就明白了。



以上代码通过匿名字段实现了继承，将公共的属性封装在Person中，在Student中直接包含Person,那么Student中就有了Person中所有的成员，Person就是匿名字段。注意：Person匿名字段，只有类型，没有名字。

那么接下来说我们就可以给Student赋值了，具体初始化的方式如下：

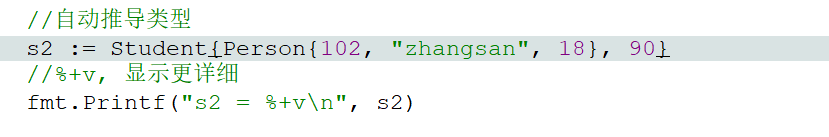


以上代码中创建了一个结构体变量s1, 这个s1我们可以理解为就是Student对象，但是要注意语法格式，以下的写法是错误的：

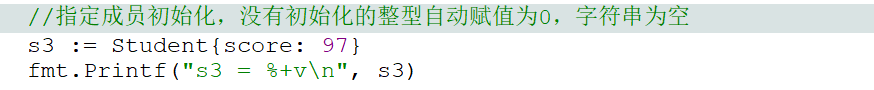
var s2 Student=Student{101,”mike”,18,98.5}

其它初始化方式如下：

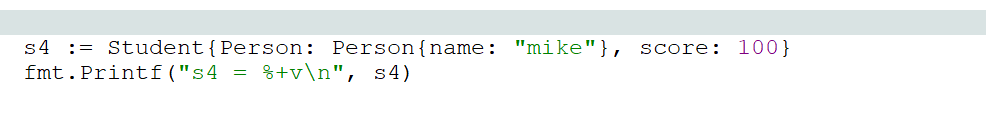
(1)自动推导类型



（2）指定初始化成员

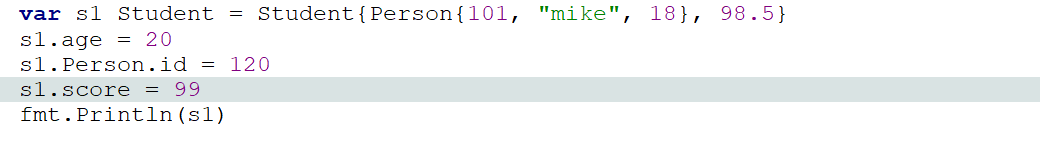


接下来还可以对Person中指定的成员进行初始化。



#### **成员操作**

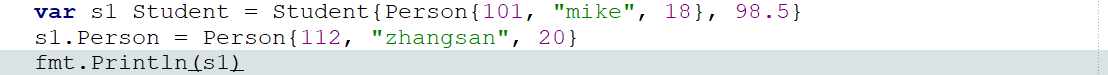
**创建完成对象后，可以根据对象来操作对应成员属性，是通过“.”运算符来完成操作的。具体案例如下：**



**由于Student继承了Person，所以Person具有的成员，Student也有，所以根据Student创建出的对象可以直接对age成员项进行修改。**

**由于在Student中添加了匿名字段Person，所以对象s1,也可以通过匿名字段Person来获取age，进行修改。**

**当然也可以进行如下修改：**

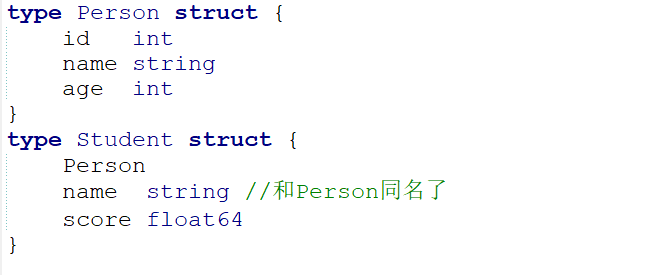


**直接给对象s1中的Person成员（匿名字段）赋值**

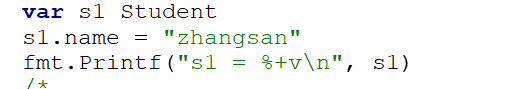
**通过以上案例我们可以总结出，根据类（结构体）可以创建出很多的对象，这些对象的成员（属性）是一样的，但是成员（属性）的值是可以完全不一样的。**

#### **同名字段**

**现在将Student结构体与Person结构体，进行如下的修改：**



**在Student中也加入了一个成员name,这样与Person重名了，那么如下代码是给Student中name赋值还是给Person中的name 进行赋值？**



**输出的结果如下：**

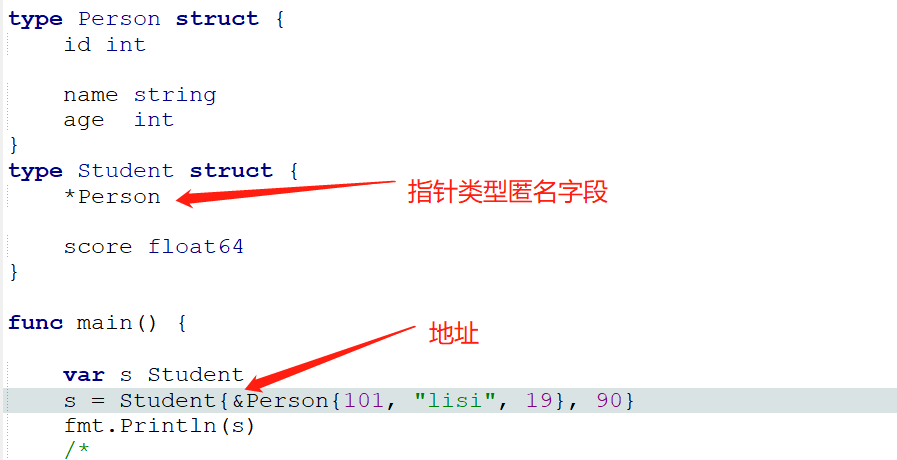
C:\Users\wangc\AppData\Local\Temp\1524110102(1).png

**通过结果发现是对Student中的name进行赋值，所以在操作同名字段时，有一个基本的原则：如果能够在自己对象所属的类（结构体）中找到对应的成员，那么直接进行操作，如果找不到就去对应的父类（结构体）中查找。这就是所谓的就近原则。**

#### **指针类型匿名字段**

**结构体（类）中的匿名字段的类型，也可以是指针。**

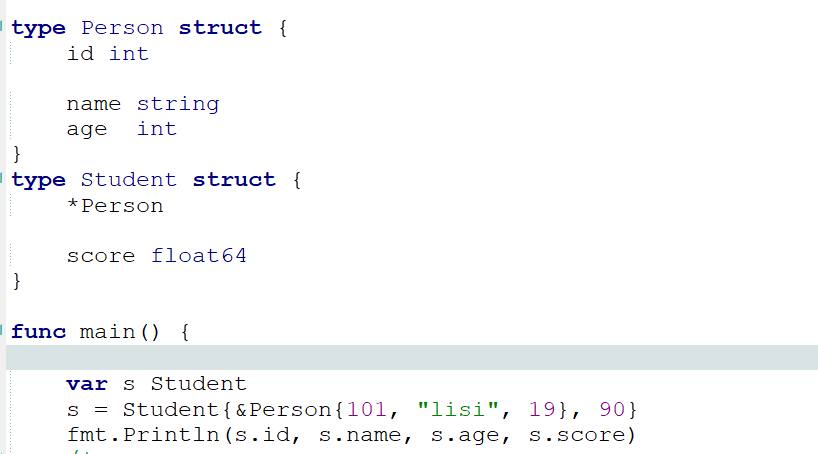
**例如：**



**输出结果：**

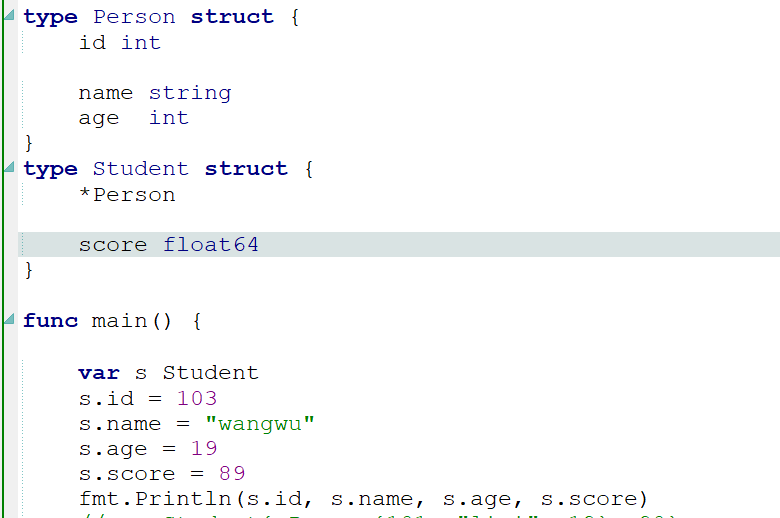
C:\Users\wangc\AppData\Local\Temp\1524122320(1).png

**输出了结构体的地址。如果要取值，可以进行如下操作：**



**在定义对象s时，完成初始化，然后通过”.”的操作完成成员的操作。**

**但是，注意以下的写法是错误的：**



**大家可以思考一下，以上代码为什么会出错？**

**会出错，错误信息如下：**

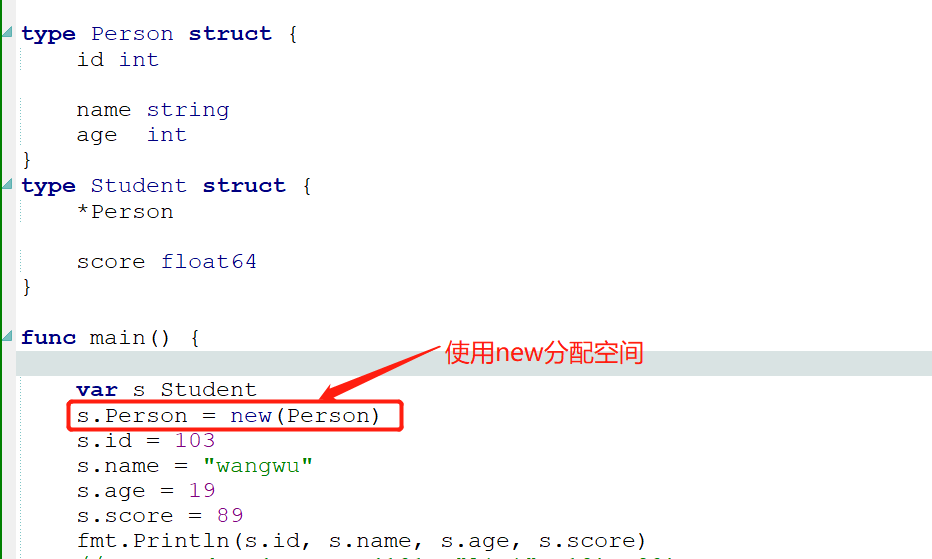
**invalid memory address or nil pointer dereference**

**翻译成中文：无效的内存地址或nil指针引用**

**意思是\*Person没有指向任何的内存地址，那么其默认值为nil.**

**也就是指针类型匿名字段\*Person没有指向任何一个结构体，所以对象s也就无法操作Person中的成员。**

**具体的解决办法如下：**



**new( )的作用是分配空间，new( )函数的参数是一个类型，这里为Person结构体类型，返回值为指针类型，所以赋值给\*Person,**

**这样\*Person也就指向了结构体Person的内存**

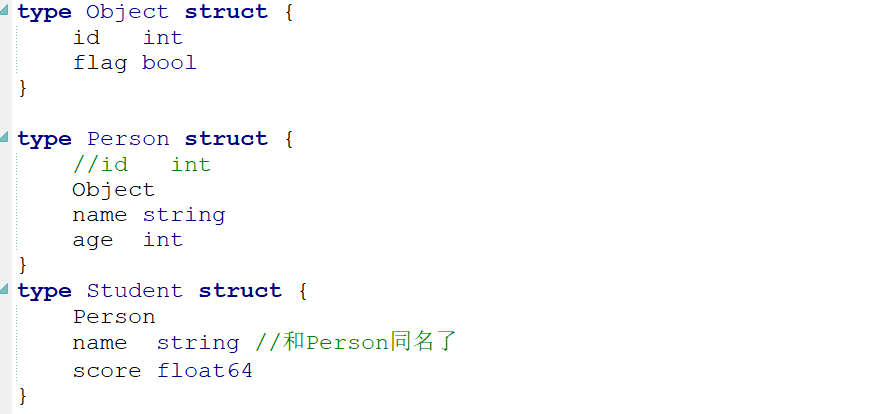
#### **多重继承**

**在上面的案例，Student类（结构体）继承了Person类（结构体），那么Person是否可以在继承别的类(结构体)呢？**

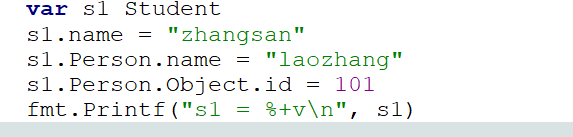
**可以，这就是多重继承。**

**多重继承指的是一个类可以继承另外一个类，而另外一个类又可以继承别的类，比如A类继承B类，而B类又可以继承C类，这就是多重继承。**

**具体案例如下：**



**接下来，看一下怎样对多重继承中的成员进行操作：**



**操作的方式，与前面讲解的是一样的。**

**注意：尽量在程序中，减少多重继承，否则会增加程序的复杂度。**

### **方法**

**通过以上内容的讲解，大家能够体会出面向对象编程中继承的优势了，接下来会给大家介绍**

**面向对象编程中另外的特性：封装性，其实关于封装性，在前面的编程中，大家也已经能够体会到了，就是通过函数来实现封装性。**

**大家仔细回忆一下，当初在讲解函数时，重点强调了函数的作用，就是将重复的代码封装来，用的时候，直接调用就可以了，不需要每次都写一遍，这就是封装的优势。（超级玛丽案例）**

**在面向对象编程中，也有封装的特性。面向对象中是通过方法来实现。下面，将详细的给大家讲解一下方法的内容。**

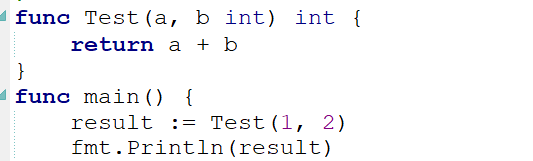
#### **方法创建**

在介绍面向对象时，讲过可以通过属性和方法（函数）来描述对象。

什么是方法呢？

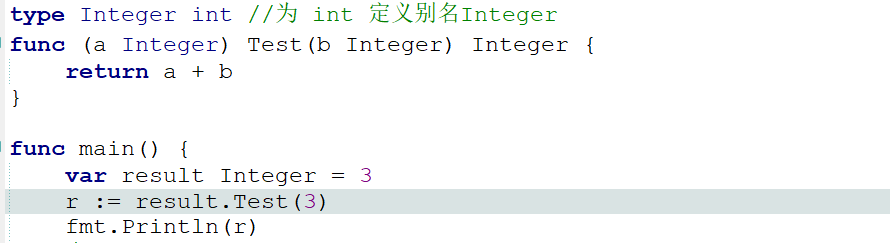
方法，大家可以理解成就是函数，但是在定义使用方面与前面讲解的函数还是有区别的。

我们先定义一个传统的函数：



这个函数非常简单，下面定义一个方法，看一下在语法与传统的函数有什么区别：

方法的定义：



type Integer int :表示的意思是给int类型指定了一个别名叫Integer,别名可以随便起，只要符合GO语言的命名规则就可以。

指定别名后，后面可以用Integer来代替int 来使用。

func (a Integer) Test(b Integer) Integer{  
}

表示定义了一个方法，方法的定义与函数的区别

第一：在关键字后面加上( a Integer), 这个在方法中称之为接收者，所谓的接受者就是接收传递过来的第一个参数，然后复制a， a的类型是Integer ,由于Integer是int的别名，所以a的类型为int

第二：在表示参数的类型时，都使用了对应的别名。

通过方法的定义，可以看出方法其实就是给某个类型绑定的函数。在该案例中，是为整型绑定的函数，只不过在给整型绑定函数(方法)时，一定要通过type来指定一个别名，因为int类型是系统已经规定好了，无法直接绑定函数，所以只能通过别名的方式。

第三:调用方式不同

var result Interger=3

表示定义一个整型变量result，并赋值为3.

result.Test( 3)

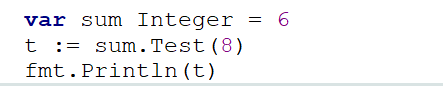
通过result变量，完成方法的调用。因为，Test( )方法，是为int类型绑定的函数，而result变量为int类型。所以可以调用Test( )方法。result变量的值会传递给Test( )方法的接受者，也就是参数a, 而实参Test( 3),会传递形参b.

当然，我们也可以将Test( )方法，理解成是为int类型扩展了，追加了的方法。因为系统在int类型时，是没有改方法的。

通过以上的定义，发现方法其实方法就是函数的语法糖。

在以上案例中，Test( )方法是为int类型绑定的函数，所以任何一个整型变量，都可以调用该方法。

例如：

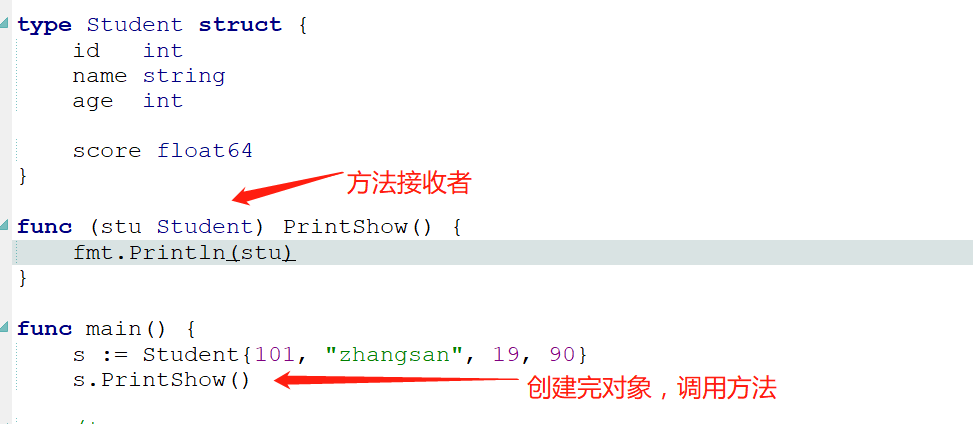


#### **给类添加类方法**

**上面给整型创建了一个方法，那么直接通过整型变量加上“点”，就可以调用该方法了。**

**大家想一下，如果给结构体(类)加上了方法，那么根据结构体（类）创建完成对象后，是不是就可以通过对象加上“点”，就可以完成方法的调用，这与调用类中定义的属性的方式是完全一样的。这样就完成了通过方法与属性来描述一个对象的操作。（大家自己回想一下，前面在讲解对象的概念时，一直在说其具有的属性与行为（方法））**

**给结构体添加方法，语法如下：**

****

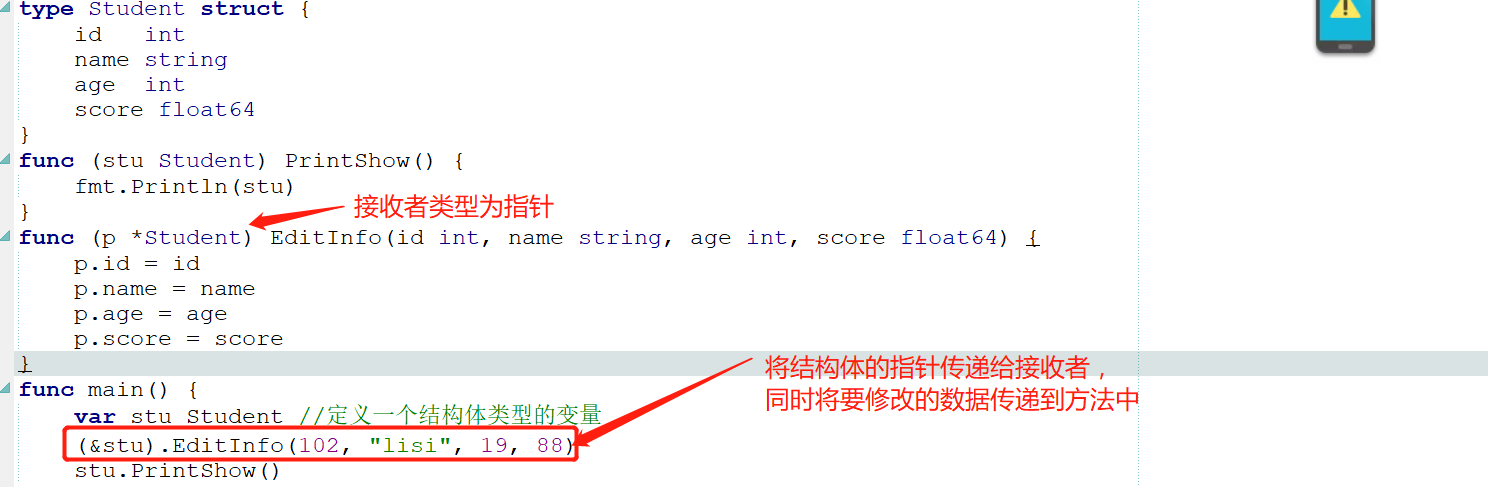
**给结构体添加方法的方式与前面给int类型添加方法的方式，基本一致。唯一不同的是，不需要给结构体指定别名，因为结构体Student就是相当于其所有成员属性的别名（id,name,score）,所以这里不要在给结构体Student创建别名。**

**调用方式：根据结构体(类)创建的对象，完成了方法的调用。**

**PrintShow( )方法的作用，只是将结构体的成员（属性）值打印出来，如果要修改其对应的值，应该怎么做呢？**

**这时，大家肯定相当了指针，将方法的接收者，修改成对应的指针类型。**

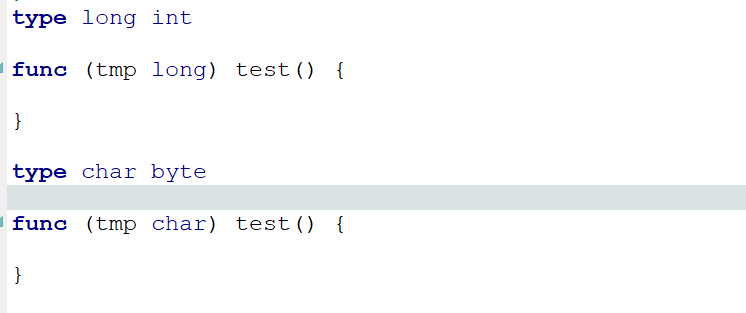
**具体修改如下：**



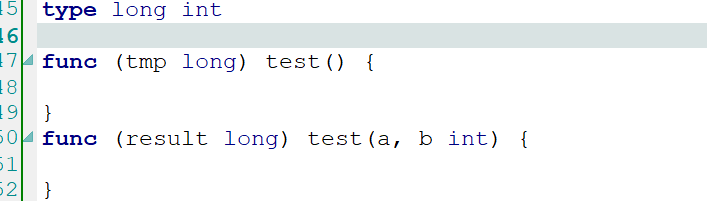
**在创建方法时，接收者类型为指针类型，所以在调用方法时，创建一个结构体变量，同时将结构体变量的地址，传递给方法的接收者，然后调用EditInfo( )方法，完成要修改的数据传递。**

**在使用方法是，要注意如下几个问题：**

**第一：只要接收者类型不一样，这个方法就算同名，也是不同方法，不会出现重复定义函数的错误**

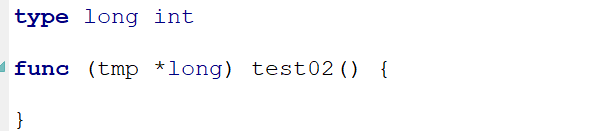


**但是，如果接收者类型一样，但是方法的参数不一样，是会出现错误的。**

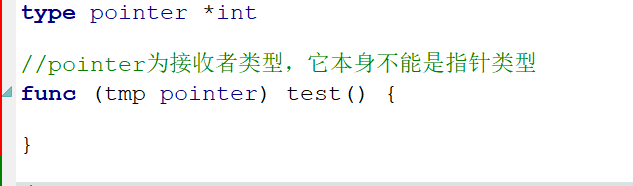


**也就是，在GO中没有方法重载(所谓重载，指的是方法名称一致，参数类型，个数不一致)。**

**第二：关于接收者不能为指针类型。**

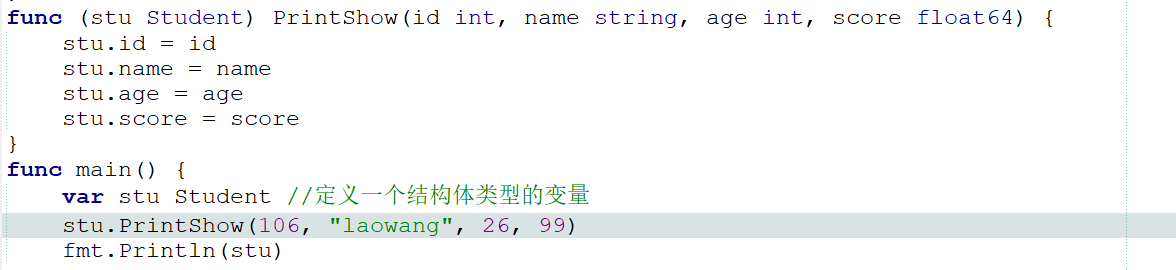


**以上定义正确**

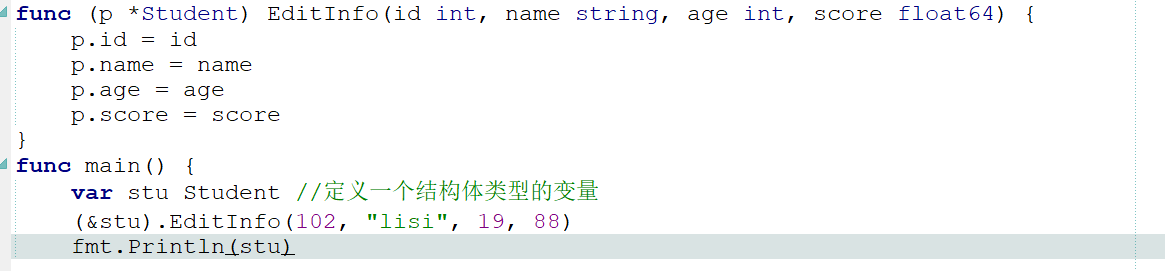


#### **指针变量的方法值**

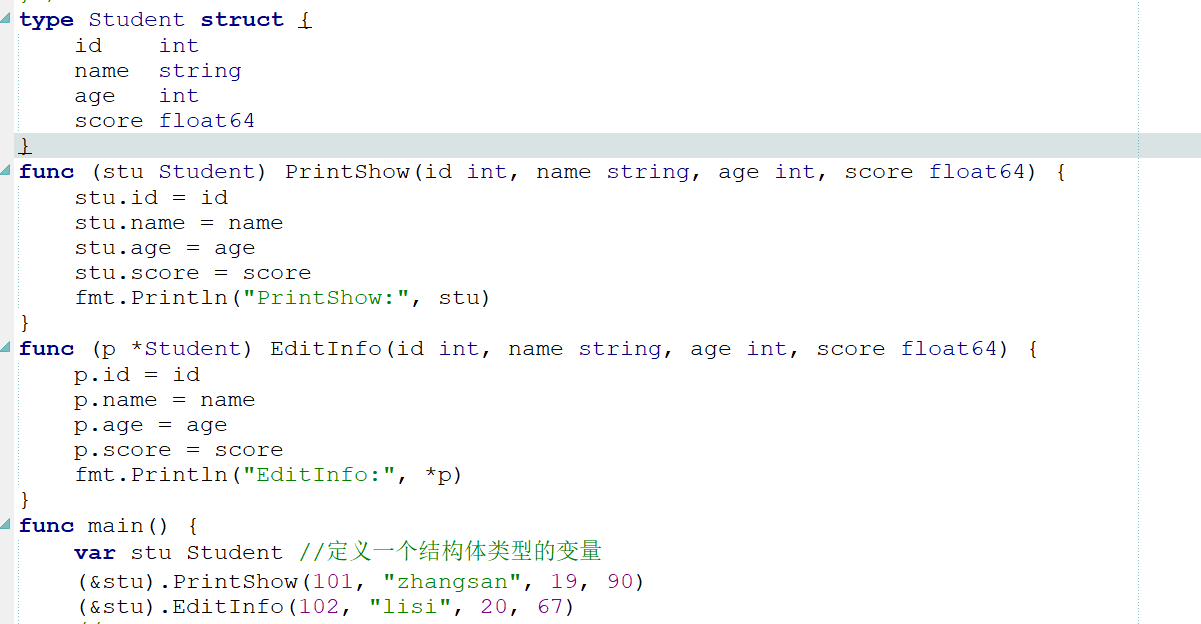
**接收者为普通变量，非指针，值传递**



**接收者为指针变量，引用传递**



**在上面的案例中，我们定义了两个方法，一个是PrintShow( ), 该方法的接收者为普通方法，一个EditInfo( )方法，该方法的接收者为指针变量，那么大家思考这么一个问题：定义一个结构体指针变量，能否调用PrintShow( )方法呢？如下所示：**



**通过测试，发现是可以调用的。**

**为什么结构体指针变量，可以调用PrintShow( )方法呢？**

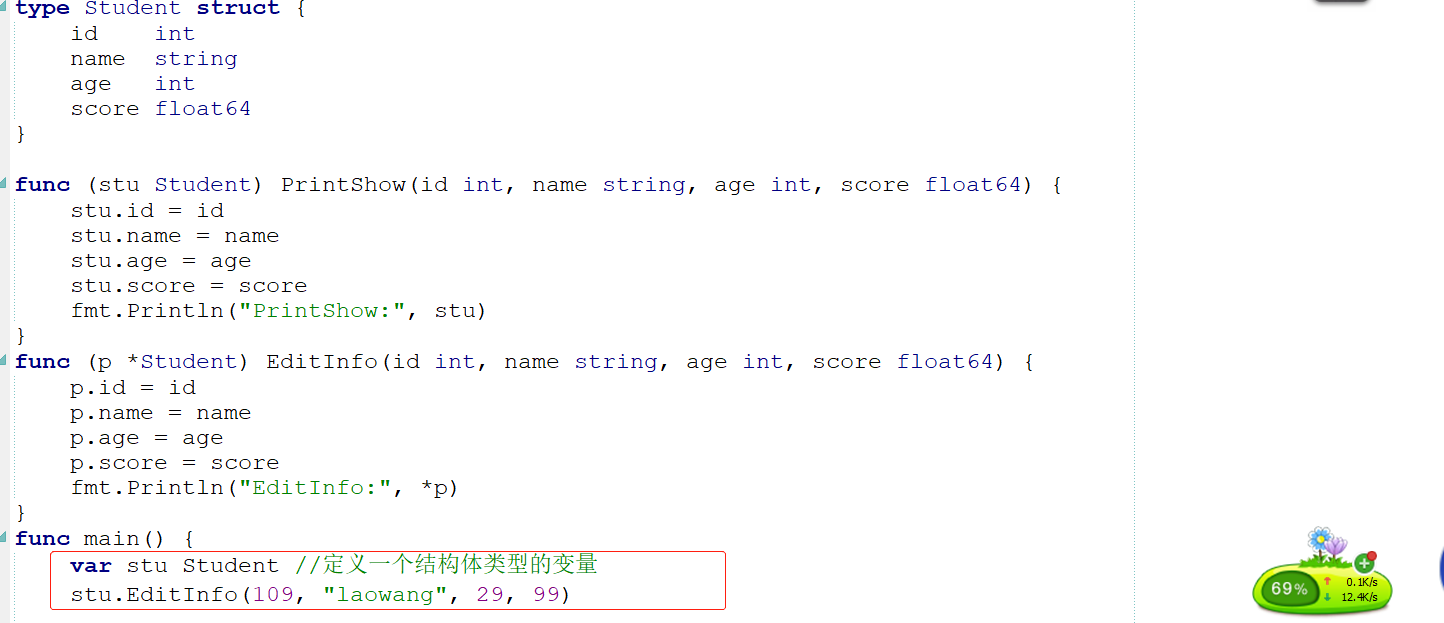
**原因是：先将指针stu, 转换成\*stu在调用。**

**等价如下代码：**

C:\Users\wangc\AppData\Local\Temp\1524209355(1).png

**所以，如果结构体变量是一个指针变量，它能够调用哪些方法，这些方法就是一个集合，简称方法集**

**如果是普通的结构体变量能否调用EditInfo( )方法。**

****

**是可以调用的，原因是：将普通的结构体类型的变量转换成(&stu)在调用EditInfo( )方法。**

**这样的好处是非常灵活，创建完对应的对象后，可以随意调用方法，不需要考虑太多指针的问题。**

下面进行面向对象编程的练习

练习：

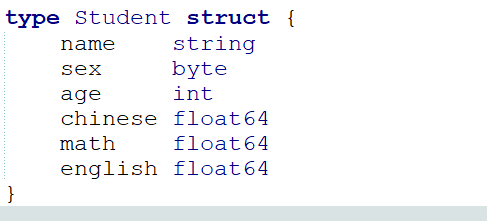
定义一个学生类,有六个属性,分别为姓名、性别、年龄、语文、数学、英语成绩。

有2个方法：

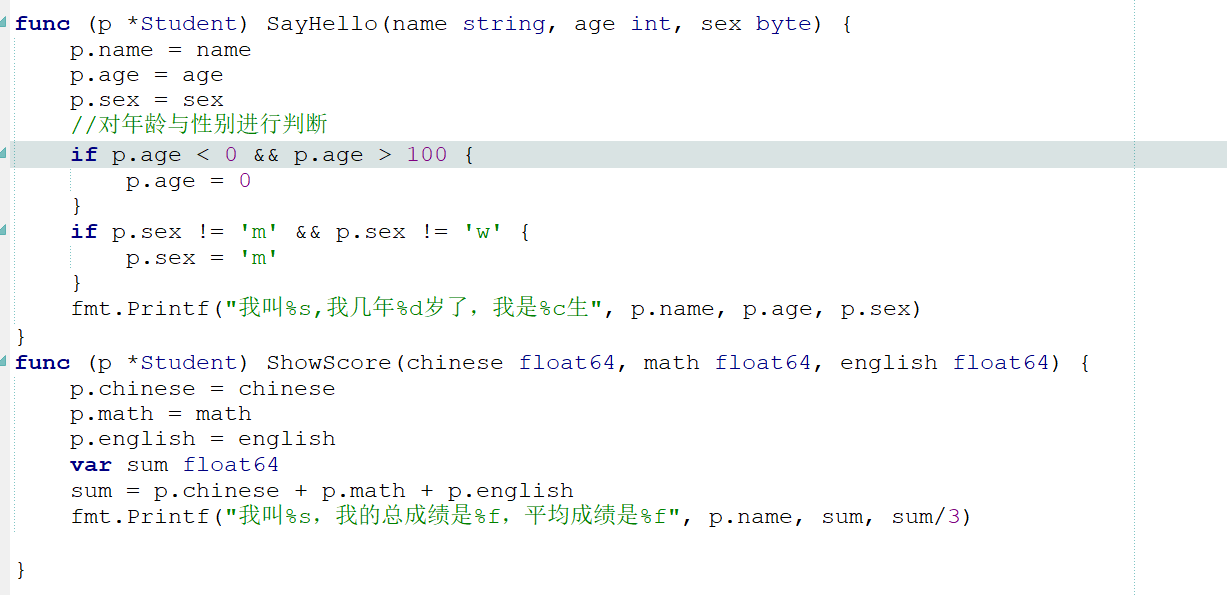
一个打招呼的方法：介绍自己叫XX，今年几岁了。是男同学还是女同学。

两个计算自己总分数和平均分的方法。{显示:我叫XX,这次考试总成绩为X分,平均成绩为X分}

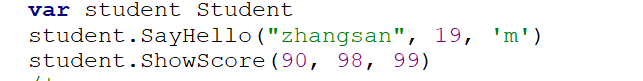
1：结构体定义如下：



2：为结构体定义相应的方法，并且在方法中可以完成对传递过来的数据的校验



3：完成方法的调用



在以上的案例中，SayHello()方法中已经完成了name属性的赋值，所以在ShowScore( )方法中，可以直接使用，因为我们使用指针指向了同一个结构体内存。

在调用的过程中，也能体会出确实很方便，**，不需要考虑太多指针的问题**

**练习2：写一个Ticket类,有一个距离属性,**

**不能为负数,有一个价格属性 ,**

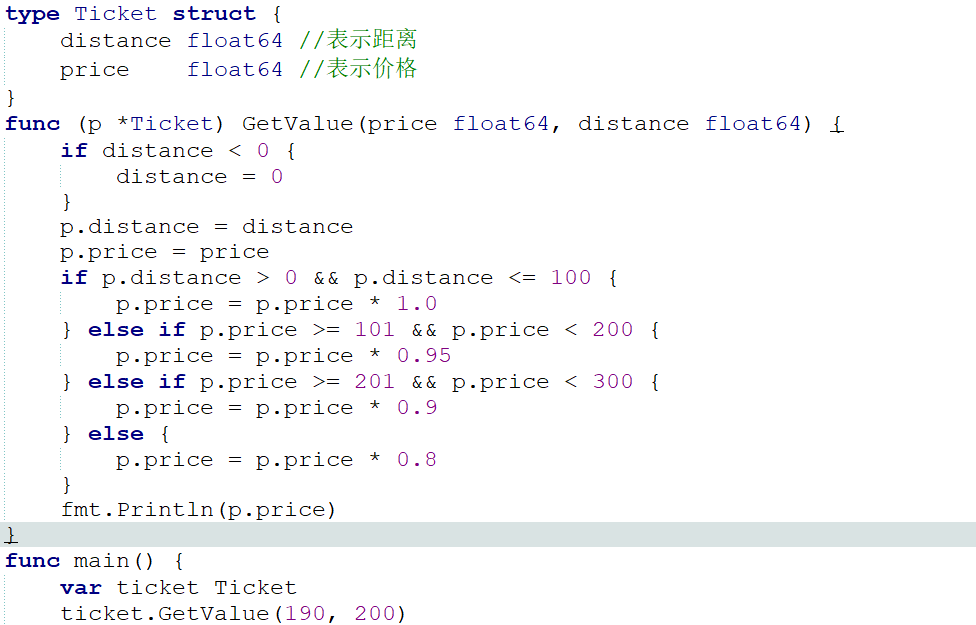
**并且根据距离distance计算价格Price (1元/公里):**

**0-100公里 票价不打折**

**101-200公里 总额打9.5折**

**201-300公里 总额打9折**

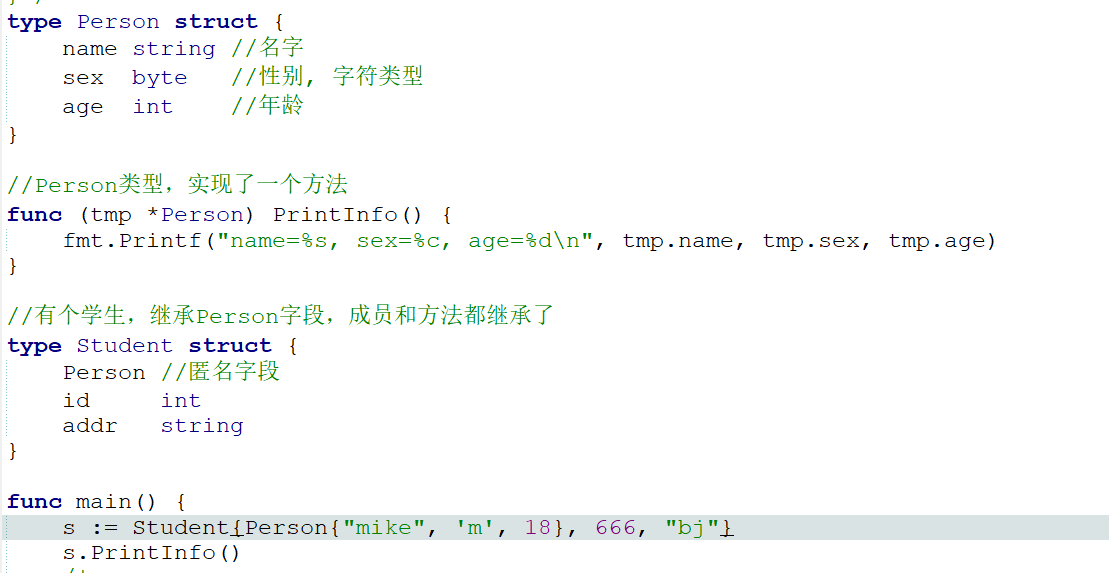
**300公里以上 总额打8折**



### **方法继承**

现在我们已经实现了为结构体添加成员(属性)，和方法，并且实现了成员属性的继承，那么方法能否继承呢？

具体如下：

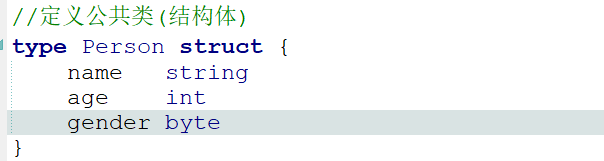


练习1：根据以下信息，实现对应的继承关系

记者：我是记者 我的爱好是偷拍 我的年龄是34 我是一个男狗仔

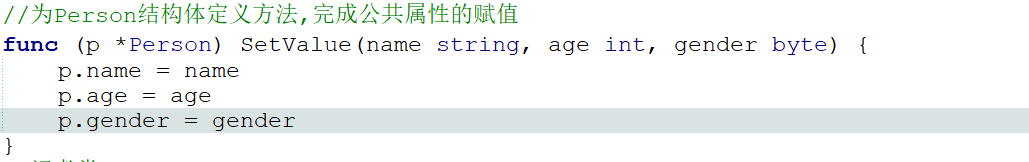
程序员：我叫孙权 我的年龄是23 我是男生 我的工作年限是 3年

思路：1.找出公共的属性，定义父类(结构体)



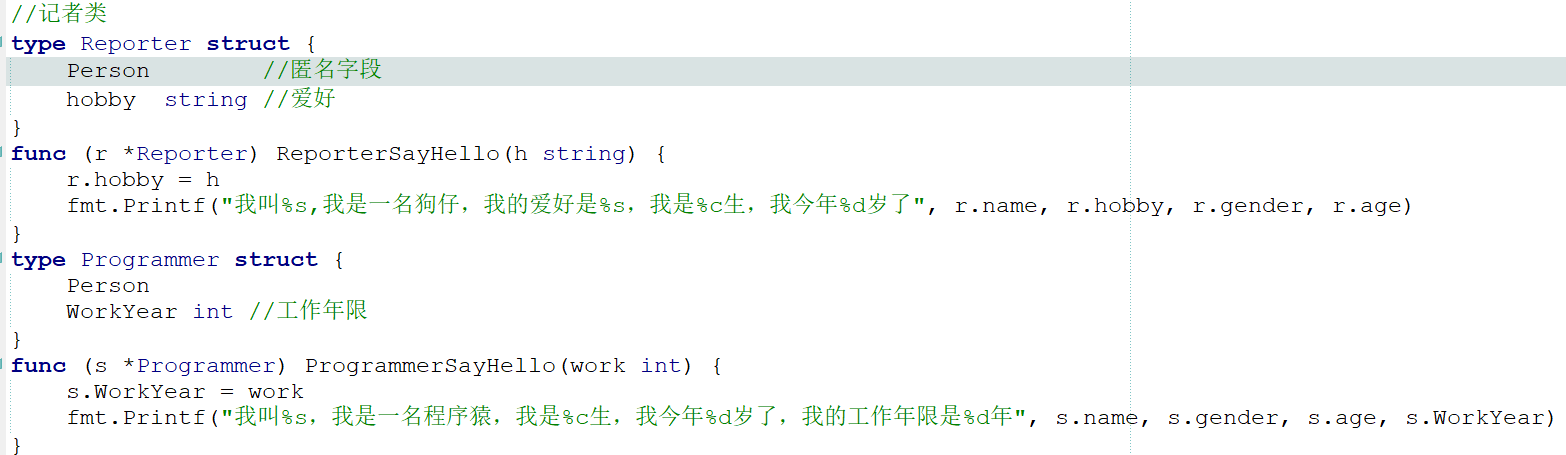
姓名，年龄，性别 是公共的成员

2：找出公共的方法，定义在父类(结构体)

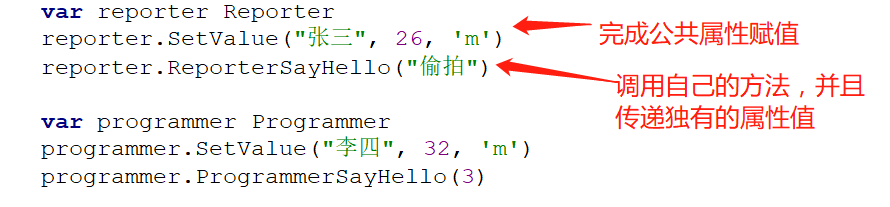


3：找出独有的属性，定义在自己的结构体(类)中。

4: 找出独有的方法，定义在自己的结构体(类)中



完成调用：



### **方法重写**

**在前面的案例中，子类(结构体)可以继承父类中的方法，但是，如果父类中的方法与子类的方法是重名方法会怎样呢？**

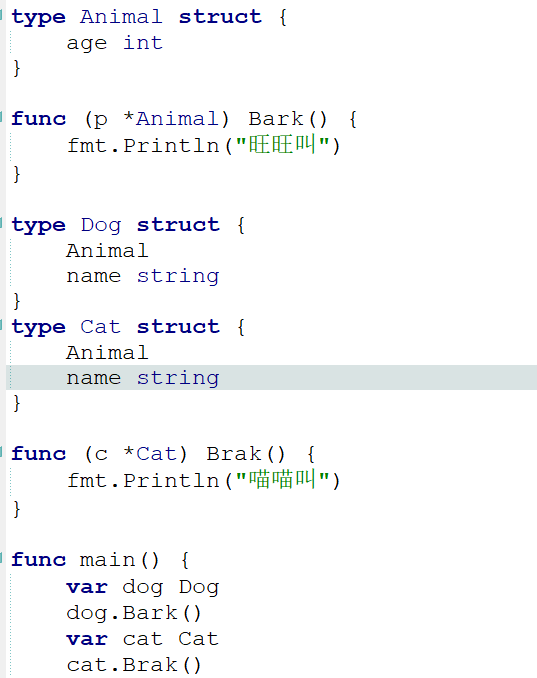


**如果子类(结构体)中的方法名与父类(结构体)中的方法名同名，在调用的时候是先调用子类(结构体)中的方法，这就方法的重写。**所谓**的重写：就是子类(结构体)中的方法，将父类中的相同名称的方法的功能重新给改写了。**

**为什么要重写父类(结构体)的方法呢？  
通常，子类(结构体)继承父类(结构体)的方法，在调用对象继承方法的时候，调用和执行的是父类的实现。但是，有时候需要**

**对子类中的继承方法有不同的实现方式。例如，假设动物存在“跑”的方法，从中继承有狗类和马类两个子类，但是它们的跑是不一样的。**

**例如以下案例：**



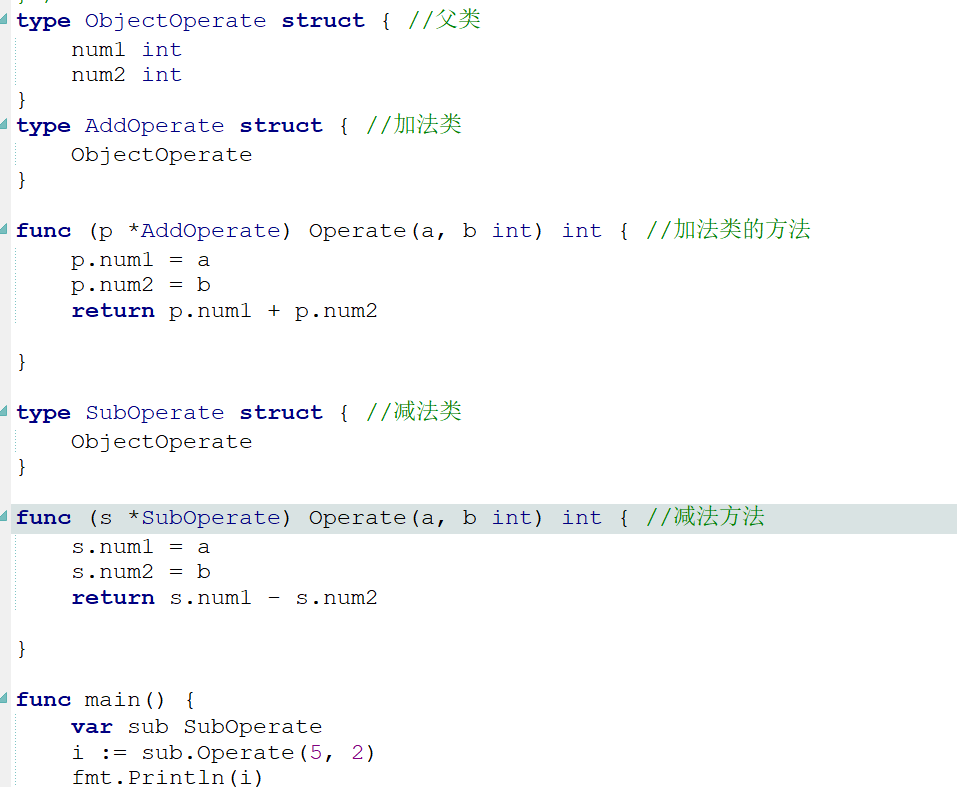
**在改案例中，定义了一个动物类(结构体)，并且有一个叫的方法，接下来小狗的类(结构体)继承动物类，小猫的类继承动物类，它们都有了叫的方法，但是动物类中的叫的方法无法满足小猫的叫的要求，只能重写。**

## Interface

在讲解具体的接口之前，先看如下问题。

使用面向对象的方式，设计一个加减的计算器

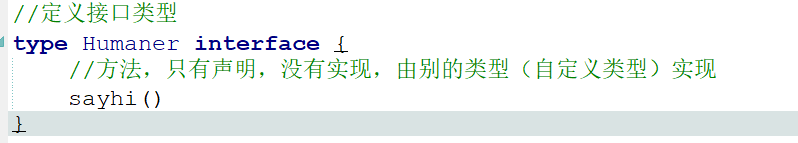
代码如下：



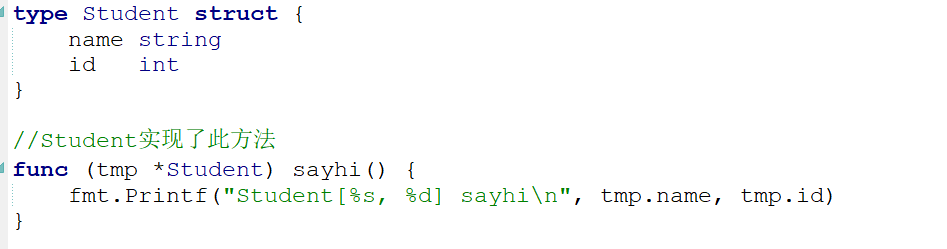
以上实现非常简单，但是有个问题，在main( )函数中，当我们想使用减法操作时，创建减法类的对象，调用其对应的减法的方法。但是，有一天，系统需求发生了变化，要求使用加法，不再使用减法，那么需要对main( )函数中的代码，做大量的修改。将原有的代码注释掉，创建加法的类对象，调用其对应的加法的方法。有没有一种方法，让main( )函数，只修改很少的代码就可以解决该问题呢？有，要用到接下来给大家讲解的接口的知识点。

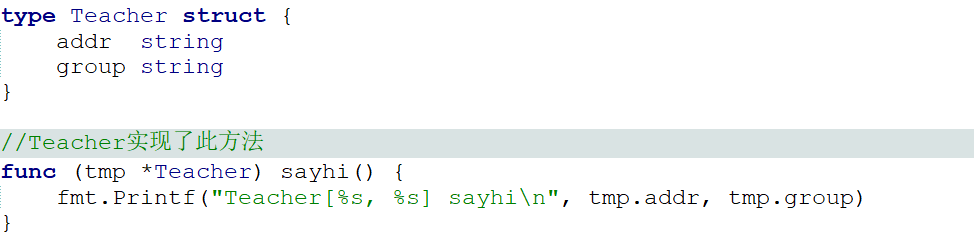
### **接口定义**

**接口定义的语法如下：**

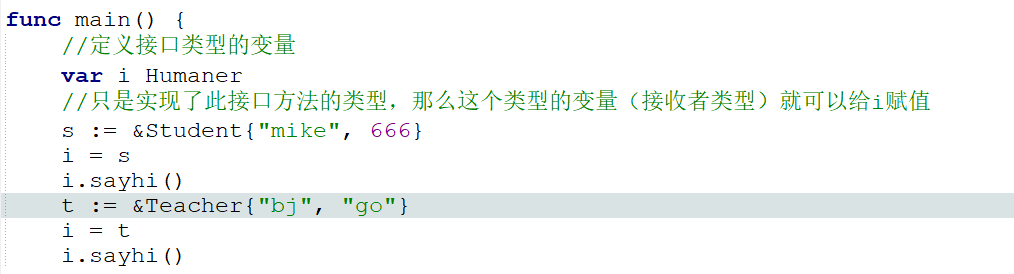


**怎样具体实现接口中定义的方法呢？**





**具体的调用如下：**



**只要类(结构体)实现对应的接口，那么根据该类创建的对象，可以赋值给对应的接口类型。**

**接口的命名习惯以er结尾。**

### **多态**

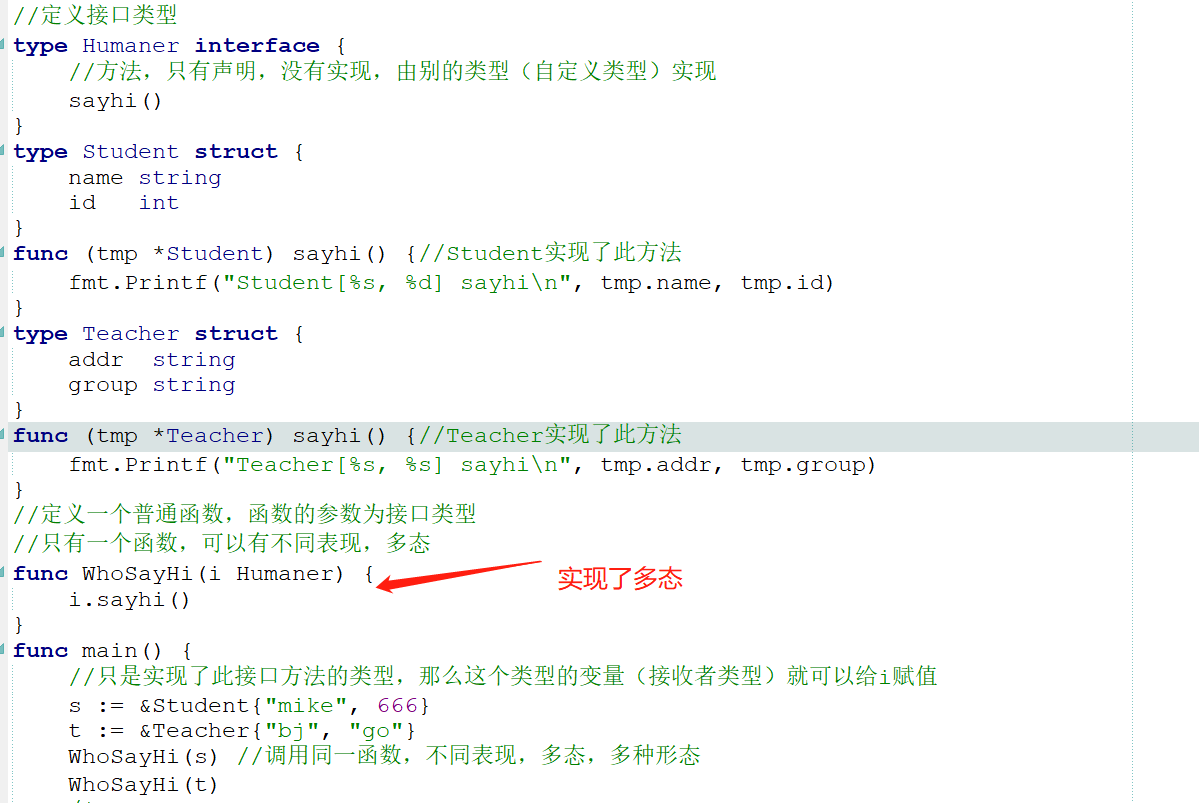
**接口有什么好处呢？实现多态。**

**所谓多态指的是多种表现形式，如下图所示：**



**该拖拉机既可以扫地又可以当风扇。功能非常强大。**

**使用接口实现多态的方式如下：**



**练习：**

**type** Person **struct** {

    name string

}

**func** (p \*Person) SayHello() {

    fmt.Println("我是人类,我叫:" + p.name)

}

**type** Chinese **struct** {

    Person

}

**func** (c \*Chinese) SayHello() {

    fmt.Println("我是中国人,我叫:" + c.name)

}

**type** English **struct** {

    Person

}

**func** (e \*English) SayHello() {

    fmt.Println("我是英国人,我叫:" + e.name)

}

**type** Personer **interface** {

    SayHello()

}

**func** main() {

    c := &Chinese{Person{"zhangsan"}}

    e := &English{Person{"MrZhang"}}

    x := make([]Personer, 2)

    x[0] = c

    x[1] = e

**for** i := 0; i < len(x); i++ {

        x[i].SayHello()

}

上面的案例与第一个案例，基本上是一样的，不同之处是在输出方面，通过一个循环获取切片中存储的所有对象，然后分别调用SayHello( )方法。

如果没有接口，那么只能一个一个的调用其方法。

练习2：用多态来实现 将 移动硬盘或者U盘或者MP3插到电脑上进行读写数据（分析类，接口，方法）

**type** MobileStorager **interface** {

    Read()

    Write()

}

**type** MobileDisk **struct** { //移动硬盘

}

**func** (m \*MobileDisk) Read() {

    fmt.Println("移动硬盘在读取数据")

}

**func** (m \*MobileDisk) Write() {

    fmt.Println("移动硬盘在写入数据")

}

**type** UDisk **struct** {

}

**func** (u \*UDisk) Read() {

    fmt.Println("U盘在读取数据")

}

**func** (u \*UDisk) Write() {

    fmt.Println("U盘在写入数据")

}

**type** Computer **struct** {

}

**func** (c \*Computer) CpuRead(i MobileStorager) {

    i.Read()

}

**func** (c \*Computer) CpuWrite(i MobileStorager) {

    i.Write()

}

**func** main() {

    m := &MobileDisk{}

    c := &Computer{}

    c.CpuRead(m)

    c.CpuWrite(m)

**练习3：麻雀会飞 鹦鹉会飞，直升飞机会飞**

**type** Flyabler **interface** {

    Fly()

}

**type** Bird **struct** {

}

**func** (b \*Bird) EatAndDrink() { //为Bird定义方法

    fmt.Println("鸟儿吃喝")

}

**type** MaQue **struct** { //麻雀

    Bird

}

**func** (m \*MaQue) Fly() {

    fmt.Println("麻雀会飞")

}

**type** YingWu **struct** {

    Bird

}

**func** (y \*YingWu) Fly() {

    fmt.Println("鹦鹉飞")

}

**type** Plane **struct** {

}

**func** (p \*Plane) Fly() {

    fmt.Println("飞机飞")

}

**func** WhoFly(i Flyabler) {

    i.Fly()

}

**func** main() {

    m := &MaQue{}

    m.EatAndDrink()

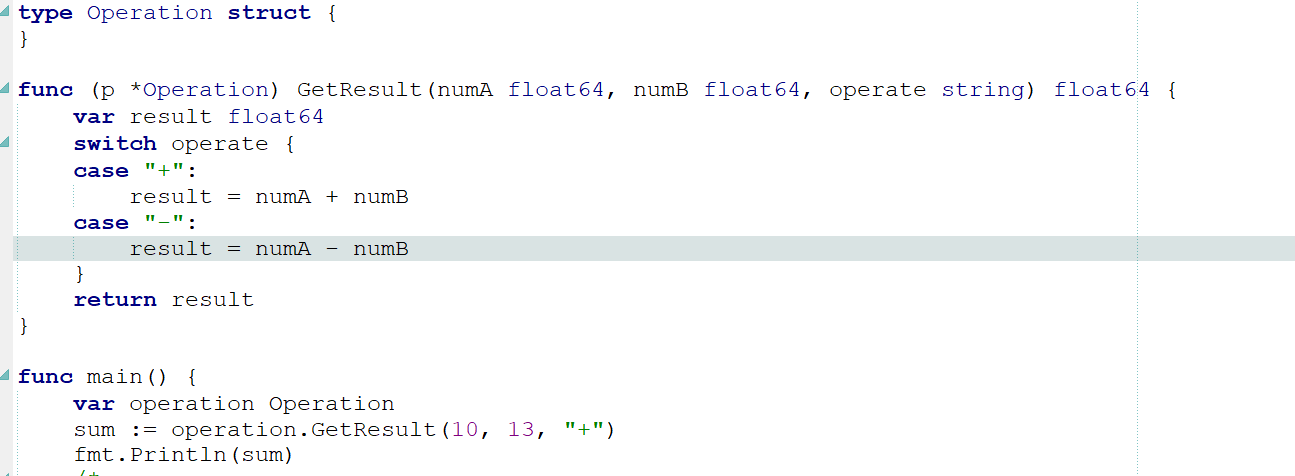
    WhoFly(m)

    plane := &Plane{}

WhoFly(plane)

关于接口的定义，以及使用接口实现多态，大家都比较熟悉了，但是多态有什么好处呢？现在还是以开始提出的计算器案例给大家讲解一下，在开始我们已经实现了一个加减功能的计算器，但是有同学感觉太麻烦了，因为实现加法，就要定义加法操作的类（结构体）,实现减法就要定义减法的类(结构体)，所以该同学实现了一个比较简单的加减法的计算器，如下所示：

使用面向对象的思想实现一个加减功能的计算器，可能有同学感觉非常简单，代码如下：



我们定义了一个类(结构体)，然后为该类创建了一个方法，封装了整个计算器功能，以后要使用直接使用该类(结构体)创建对象就可以了。这就是面向对象总的封装性。

也就是说，当你写完这个计算器后，交给你的同事，你的同事要用，直接创建对象，然后调用GetResult()方法就可以， 根本不需要关心该方法是怎样实现的.这不是我们前面在讲解面向对象概念时说到的，找个对象来干活吗？不需要自己去实现该功能。

大家仔细观察上面的代码，有什么问题吗？

现在让你在改计算器中，再增加一个功能，例如乘法，应该怎么办呢？你可能会说很简单啊，直接在GetResult( )方法的switch中添加一个case分支就可以了。

问题是：在这个过程中，如果你不小心将加法修改成了减法怎么办？或者说，对加法运算的规则做了修改怎么办？举例子说明：

你可以把该程序方法想象成公司中的薪资管理系统。如果公司决定对薪资的运算规则做修改，由于所有的运算规则都在Operation类中的GetResult()方法中，所以公司只能将该类的代码全部给你，你才能进行修改。这时，你一看自己作为开发人员工资这么低，心想“TMD,老子累死累活才给这么点工资，这下有机会了”。直接在自己工资后面加了3000

numA+numB+3000

所以说，我们应该将 加减等运算分开，不应该全部糅合在一起，这样你修改加的时候，不会影响其它的运算规则：

具体实现如下：

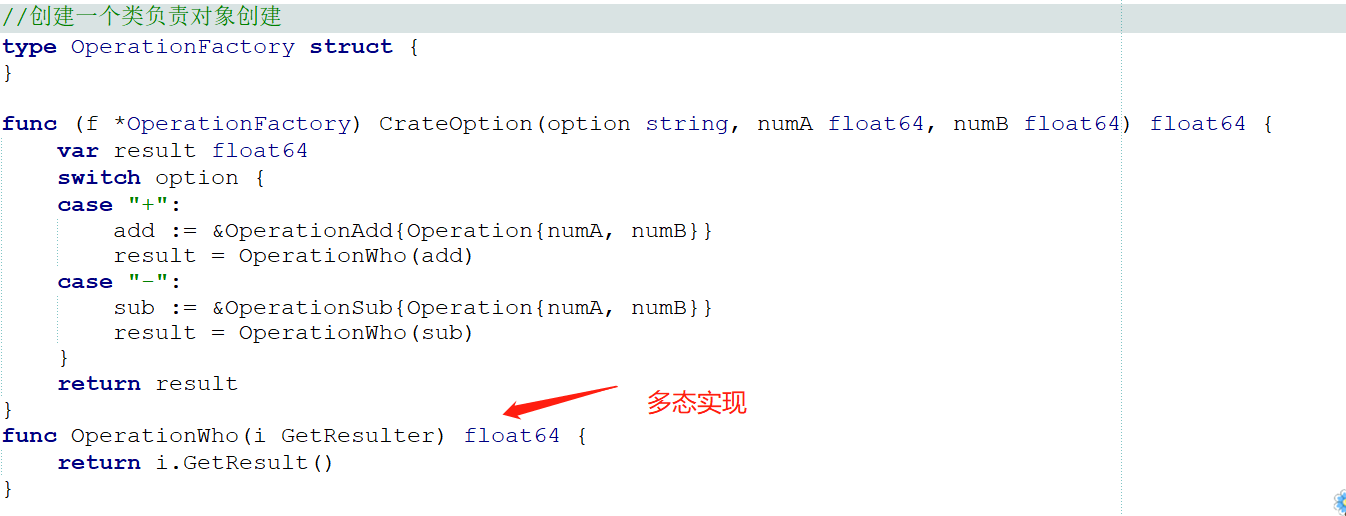


现在已经将各个操作分开了，并且这里我们还定义了一个父类(结构体)，将公共的成员放在该父类中。如果现在要修改某项运算规则，只需将对应的类和方法发给你，进行修改就可以了。

这里的实现虽然将各个运算分开了，但是与我们第一次实现的还是有点区别。我们第一次实现的加减计算器也是将各个运算分开了，但是没有定义接口。那么该接口的意义是什么呢?继续看下面的问题。

3：现在怎样调用呢？

这就是我们一开始给大家提出的问题，如果调用的时候，直接创建加法操作的对象，调用对应的方法，那么后期要改成减法呢？需要做大量的修改，所以问题解决的方法如下：



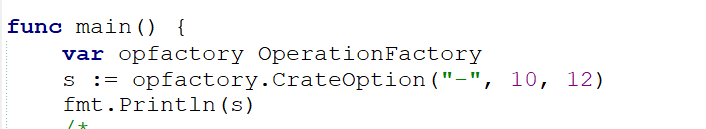
**创建了一个类OperationFactory，在改类中添加了一个方法CreateOption( )负责创建对象，如果输入的是“+”，创建**

**OperationAdd的对象，然后调用OperationWho( )方法，将对象的地址传递到该方法中，所以变量i指的就是OperationAdd，接下来在调用GetResult( )方法，实际上调用的是OperationAdd类实现的GetResult( )方法。**

**同理如果传递过来的是“-”，流程也是一样的。**

**所以，通过该程序，大家能够体会出多态带来的好处。**

**4：最后调用**

****

**这时会发现调用，非常简单，如果现在想计算加法，只要将”-”,修改成”+”就可以。也就是说,除去了main( )函数与具体运算类的依赖。**

**当然程序经过这样设计以后：如果现在修改加法的运算规则，只需要修改OperationAdd类中对应的方法，**

**不需要关心其它的类，如果现在要增加“乘法” 功能，应该怎样进行修改呢？第一：定义乘法的类，完成乘法运算。**

**第二：在OperationFactory类中CrateOption( )方法中添加相应的分支。但是这样做并不会影响到其它的任何运算。**

**大家可以自己尝试实现“乘法”与“除法”的运算。**

**在使用面向对象思想解决问题时，一定要先分析，定义哪些类，哪些接口，哪些方法。把这些分析定义出来，然后在考虑具体实现。**

**最后完整代码如下：**

**type** Operation **struct** { //定义操作父类

    numA float64

    numB float64

}

**type** GetResulter **interface** { //定义接口

    GetResult() float64

}

**type** OperationAdd **struct** { //加法

    Operation

}

**func** (a \*OperationAdd) GetResult() float64 { //实现接口

**return** a.numA + a.numB

}

**type** OperationSub **struct** { //减法

    Operation

}

**func** (s \*OperationSub) GetResult() float64 {

**return** s.numA - s.numB

}

//创建一个类负责对象创建

**type** OperationFactory **struct** {

}

**func** (f \*OperationFactory) CrateOption(option string, numA float64, numB float64) float64 {

**var** result float64

**switch** option {

**case** "+":

        add := &OperationAdd{Operation{numA, numB}}

        result = OperationWho(add)

**case** "-":

        sub := &OperationSub{Operation{numA, numB}}

        result = OperationWho(sub)

    }

**return** result

}

**func** OperationWho(i GetResulter) float64 {

**return** i.GetResult()

}

**func** main() {

**var** opfactory OperationFactory

    s := opfactory.CrateOption("-", 10, 12)

fmt.Println(s)

**}**

**通过以上案例，大家应该能够体会出多态的好处。**

**下面我们再通过一个练习，体会一下接口和多态的应用。**

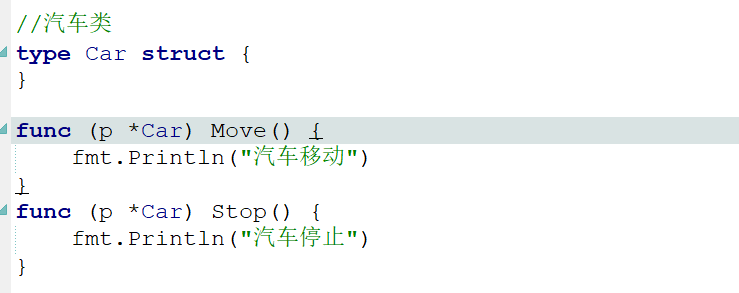
**练习：设计一个4s店卖车的程序。(分析要设计多个类，多个方法，接口)**

**1:首先我们思考一下，该程序需要设计几个类(结构体)**

**大家想到的有汽车类，还有4s店类。**

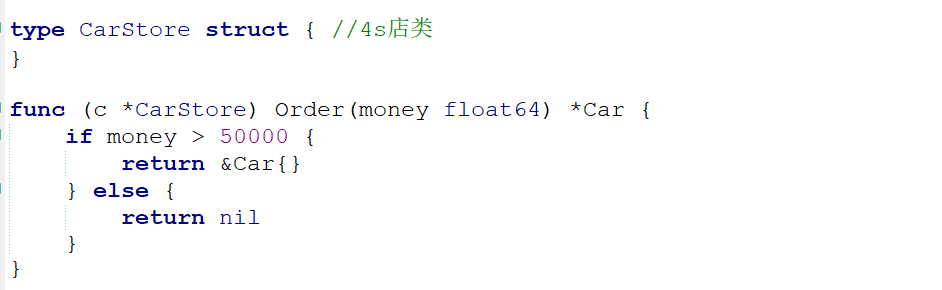
**所以基本设计的代码：**

**(1):创建汽车类(结构体)**

****

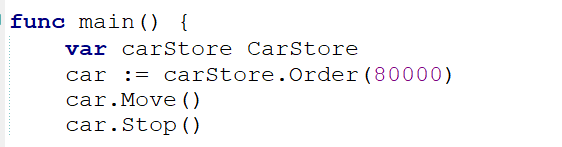
**并在改结构体中定义了两个方法。**

**(2):创建4S店类(结构体)**

****

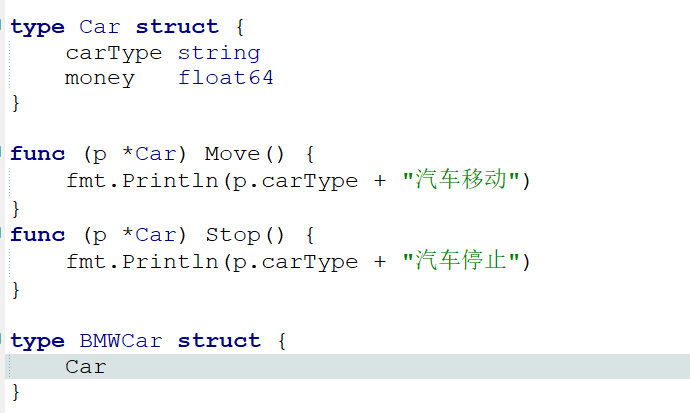
**为该类(结构体)添加Order( )方法，该方法的作用就是卖车，所以需要给该方法传递“钱”，然后进行判断，如果条件满足，就返回Car地址，所以返回类型为\*Car**

**(3):下面进行调用**

****

**(4):如果在增加不同品牌的车，应该怎样处理呢？**

**代码如下：**

****

**在以上的代码中，定义了“宝马车”类,让其继承Car类，并且在Car类中定义了两个成员 。**

**定义接口**

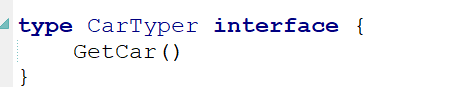
**在定义接口前，又定义了“奥迪车”类，也让其继承Car类**

**type** AudiCar **struct** {

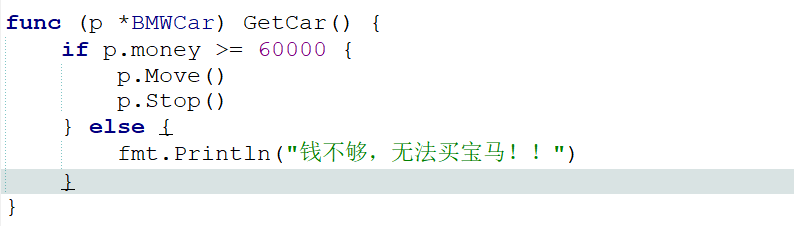
    Car

}

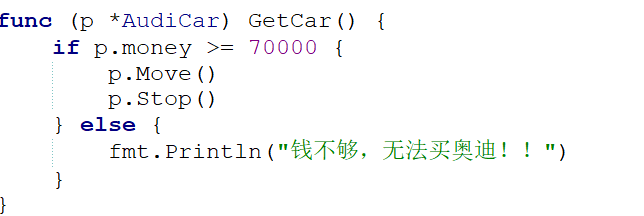
**然后定义一个接口：**

****

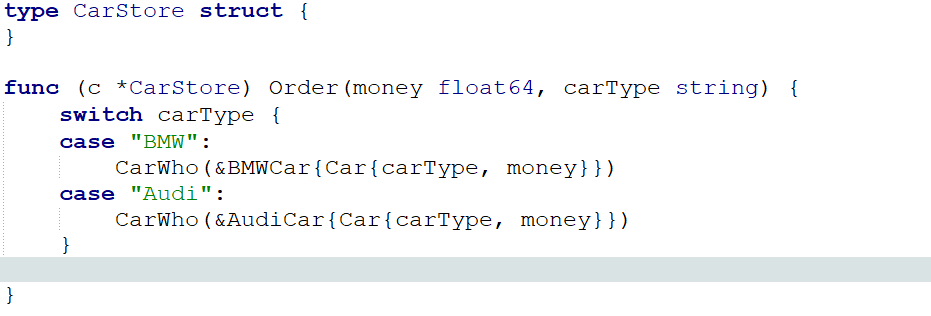
**下面实现该接口中定义的方法，**

****

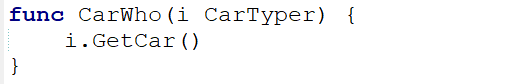
**该方法的作用就是判断钱是否够了，如果钱够了，就可以调用车具有的方法。**

****

**(7):对Order( )的改造**

****

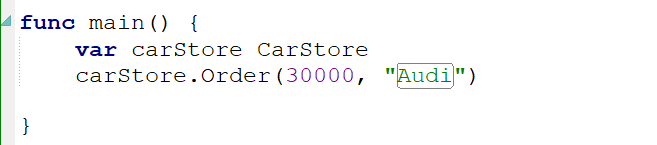
**根据传递过来的车的类型，进行判断，然后调用CarWho( )方法，该方法是一个多态的方法，定义如下：**

****

**如果传递过来的类型是”BMW”，在调用CarWho( )方法时，将BMWCar{ }类(结构体)的地址传递到该方法中(同时完成了父类Car中两个属性的赋值)。由于CarWho( )方法参数的类型是一个接口，但是BMWCar{ }类(结构体)实现了该接口，所以是完全可以BMWCar{ }类的地址传递过来。这时参数i指的就是BMWCar{ },调用GetCar( )方法，指的就是 BMWCar{ }实现的方法。在改方法中完成钱数的判断。**

**同理如果传递的过来的类型是“Audi”，那么过程也是一样的。**

1. **main( )函数的调用**

****

**(9)完整代码如下：**

**package** main

**import** "fmt"

**type** CarTyper **interface** {

    GetCar()

}

**type** CarStore **struct** {

}

**func** (c \*CarStore) Order(money float64, carType string) {

**switch** carType {

**case** "BMW":

        CarWho(&BMWCar{Car{carType, money}})

**case** "Audi":

        CarWho(&AudiCar{Car{carType, money}})

    }

}

**type** Car **struct** {

    carType string

    money float64

}

**func** (p \*Car) Move() {

    fmt.Println(p.carType + "汽车移动")

}

**func** (p \*Car) Stop() {

    fmt.Println(p.carType + "汽车停止")

}

**type** BMWCar **struct** {

    Car

}

**func** (p \*BMWCar) GetCar() {

**if** p.money >= 60000 {

        p.Move()

        p.Stop()

    } **else** {

        fmt.Println("钱不够，无法买宝马！！")

    }

}

**type** AudiCar **struct** {

    Car

}

**func** (p \*AudiCar) GetCar() {

**if** p.money >= 70000 {

        p.Move()

        p.Stop()

    } **else** {

        fmt.Println("钱不够，无法买奥迪！！")

    }

}

**func** CarWho(i CarTyper) {

    i.GetCar()

}

**func** main() {

**var** carStore CarStore

    carStore.Order(30000, "Audi")

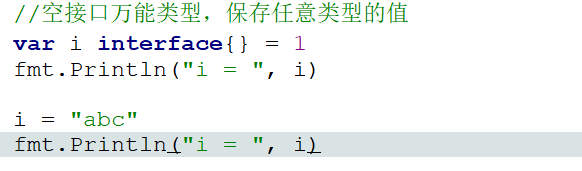
}

下面我们将接口其它的知识点再给大家说一下：

### **空接口**

**空接口(interface{})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的数值。**

**例如：**



**当函数可以接受任意的对象实例时，我们会将其声明为interface{}，最典型的例子是标准库fmt中PrintXXX系列的函数，例如：**

**func Printf(fmt string, args ...interface{})**

**func Println(args ...interface{})**

**如果自己定义函数，可以如下：**

**func Test(arg ...interface{}) {**

**}**

**Test( )函数可以接收任意个数，任意类型的参数。**

### **类型断言**

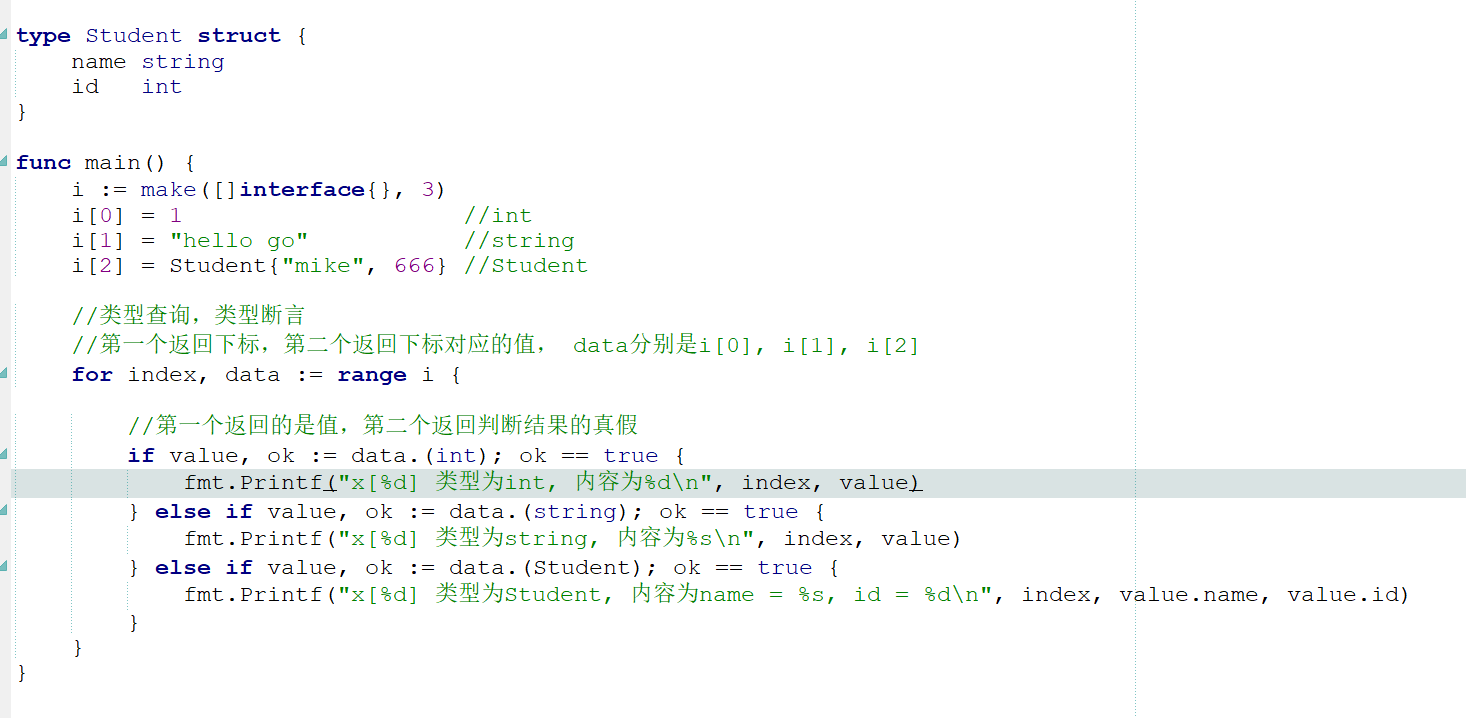
我们知道interface的变量里面可以存储任意类型的数值(该类型实现了interface)。那么我们怎么反向知道这个变量里面实际保存了的是哪个类型的对象呢？目前常用的有两种方法：

 comma-ok断言

Go语言里面有一个语法，可以直接判断是否是该类型的变量： value, ok = element.(T)，这里value就是变量的值，ok是一个bool类型，element是interface变量，T是断言的类型。

如果element里面确实存储了T类型的数值，那么ok返回true，否则返回false

具体案例如下：



switch测试

type Student struct {

name string

id int

}

func main() {

i := make([]interface{}, 3)

i[0] = 1 //int

i[1] = "hello go" //string

i[2] = Student{"mike", 666} //Student

//类型查询，类型断言

for index, data := range i {

switch value := data.(type) {

case int:

fmt.Printf("x[%d] 类型为int, 内容为%d\n", index, value)

case string:

fmt.Printf("x[%d] 类型为string, 内容为%s\n", index, value)

case Student:

fmt.Printf("x[%d] 类型为Student, 内容为name = %s, id = %d\n", index, value.name, value.id)

}

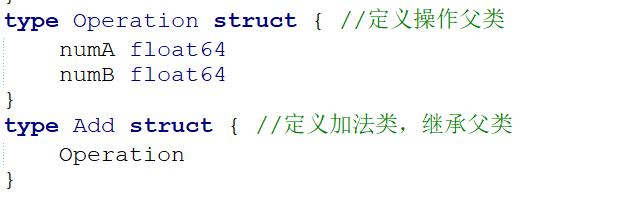
}

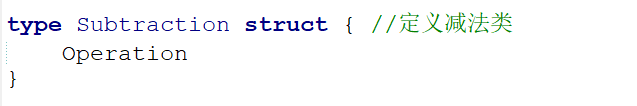
}

现在我们已经将空接口与类型断言的基本语法给大家讲解完毕了，那么在实际的开发中，我们应该怎样应用这方面的知识呢？下面我们将前面我们写的计算器这个案例，结合空接口与类型断言，在给大家写一遍。

具体的实现如下：（大家可以思考一下）

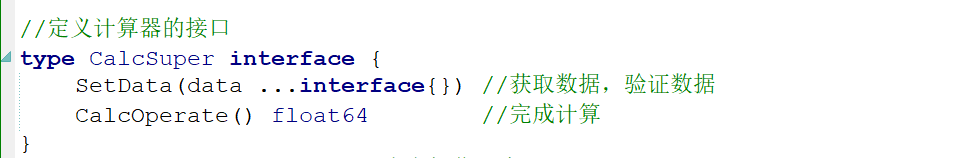
1：定义父类(结构体)，完成公共成员定义





现在父类已经定义完成，并且定义了加法类与减法类，继承父类

2：定义接口

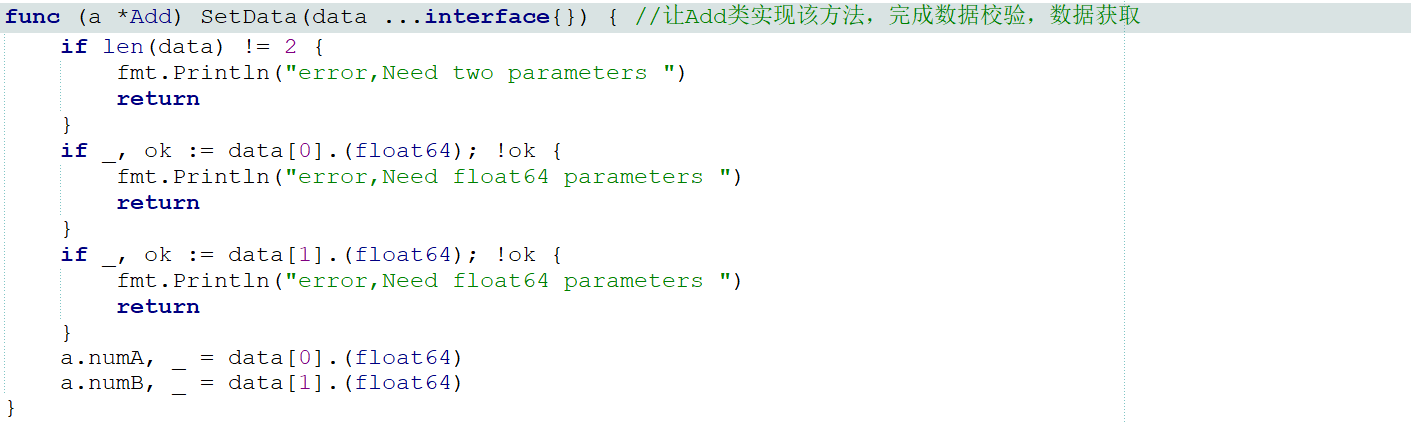


这个接口的定义与我们前面定义的接口不同之处，就是在这里我们又加入了一个方法SetData( ),该方法的作用主要是对传递过来的数据进行校验，例如，我们要求对float64类型的数据进行运算，那么只能传递小数，如果传递过来的是int类型，那么会给出相应的错误提示。该方法的参数是：不定参数，同时也是空接口，表示可以传递各种类型的数据。

3：实现接口

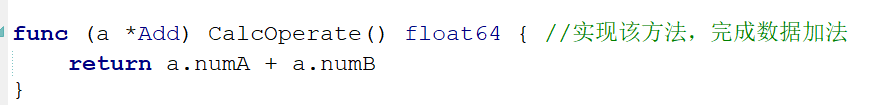
以下是加法类实现对应的接口中声明的方法。

实现SetData( )方法

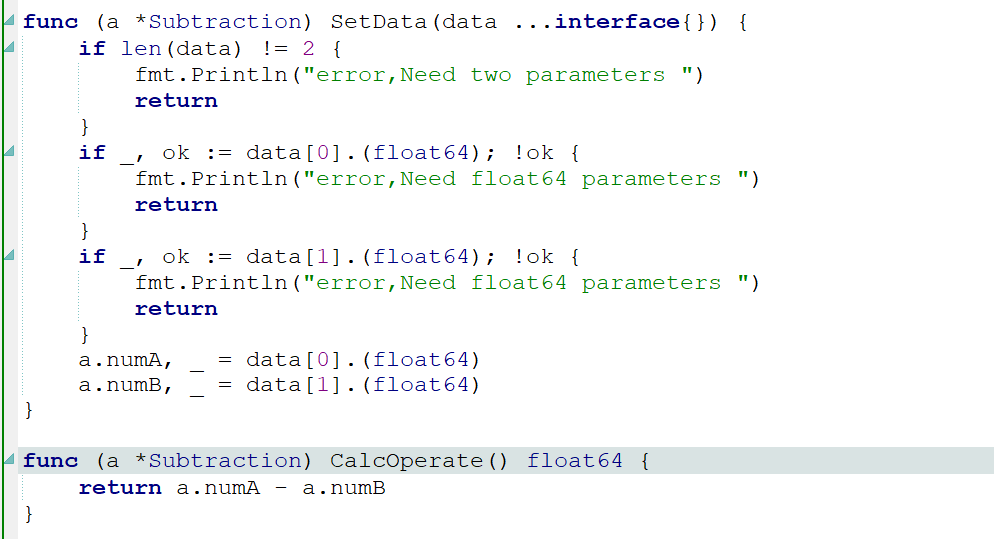


在改方法中首先对传递过来的数据的长度进行校验，然后对类型进行校验。

实现CalcOperate( )方法



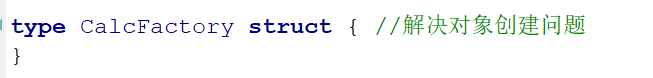
同理减法类的实现如下：



4：对象创建的封装

为了在main( )函数中，更方面的创建加法类对象，与减法类对象，所以将对象的创建封装一下。

为了解决这个问题，我们前面定义了一个OperationFactory类(结构体)，并且为该类创建了一个CreateOption( )方法，该方法完成对象创建，并且该方法返回的类型是一个float64，表示的运算结果。但是，如果我想返回一个对象，应该怎样做呢？可以将返回的类型改成接口类型，因为加法类与减法类都实现了该接口，所以定义如下：

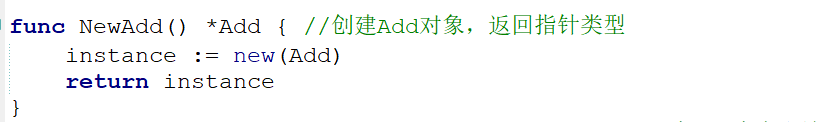


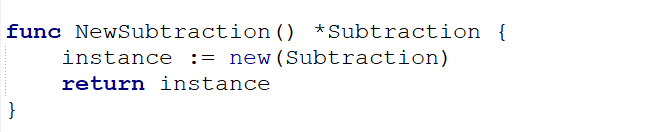


5：完善CreateOperate( )方法

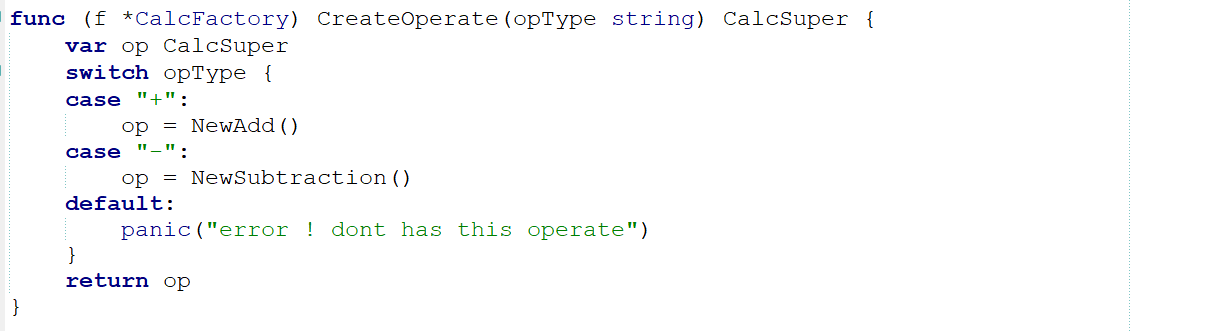
该方法主要是根据传递过来的参数opType进行判断，创建出不同的对象。

为了不让改方法代码量过于庞大，过于复杂，我们将对象的创建又单独的放在了不同的方法中。如下所示：



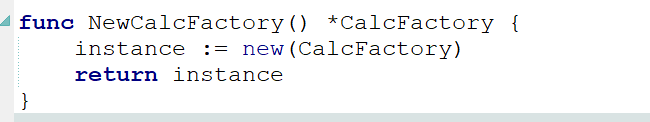


接下来在CreateOperate( )方法中完成以上两个方法的调用，具体如下所示：



6：main( )函数中完成调用

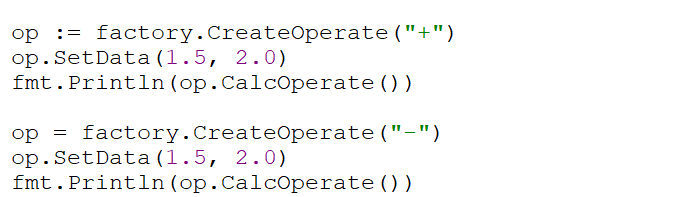
首先完成CalcFactory类对象的创建，也是单独封装在一个方法中。



在main( )函数中完成NewCalcFactory( )方法的调用，获取CalcFactory的对象



完成后续方法的调用



在这个程序中，大家要体会总结出与前面程序的不同之处。

完整代码如下：

//定义计算器的接口

**type**CalcSuper**interface**{

    SetData(data...**interface**{})//获取数据，验证数据

    CalcOperate()float64//完成计算

}

**type**Operation**struct**{//定义操作父类

    numAfloat64

    numBfloat64

}

**type**Add**struct**{//定义加法类，继承父类

    Operation

}

**func**NewAdd()\*Add{//创建Add对象，返回指针类型

    instance:=new(Add)

**return**instance

}

**func**(a\*Add)SetData(data...**interface**{}){//让Add类实现该方法，完成数据校验，数据获取

**if**len(data)!=2{

        fmt.Println("error,Needtwoparameters")

**return**

    }

**if**\_,ok:=data[0].(float64);!ok{

        fmt.Println("error,Needfloat64parameters")

**return**

    }

**if**\_,ok:=data[1].(float64);!ok{

        fmt.Println("error,Needfloat64parameters")

**return**

    }

    a.numA,\_=data[0].(float64)

    a.numB,\_=data[1].(float64)

}

**func**(a\*Add)CalcOperate()float64{//实现该方法，完成数据加法

**return**a.numA+a.numB

}

**type**Subtraction**struct**{//定义减法类

    Operation

}

**func**NewSubtraction()\*Subtraction{

    instance:=new(Subtraction)

**return**instance

}

**func**(a\*Subtraction)SetData(data...**interface**{}){

**if**len(data)!=2{

        fmt.Println("error,Needtwoparameters")

**return**

    }

**if**\_,ok:=data[0].(float64);!ok{

        fmt.Println("error,Needfloat64parameters")

**return**

    }

**if**\_,ok:=data[1].(float64);!ok{

        fmt.Println("error,Needfloat64parameters")

**return**

    }

    a.numA,\_=data[0].(float64)

    a.numB,\_=data[1].(float64)

}

**func**(a\*Subtraction)CalcOperate()float64{

**return**a.numA-a.numB

}

**type**CalcFactory**struct**{//解决对象创建问题

}

**func**NewCalcFactory()\*CalcFactory{

    instance:=new(CalcFactory)

**return**instance

}

**func**(f\*CalcFactory)CreateOperate(opTypestring)CalcSuper{

**var**opCalcSuper

**switch**opType{

**case**"+":

        op=NewAdd()

**case**"-":

        op=NewSubtraction()

**default**:

        panic("error!donthasthisoperate")

    }

**return**op

}

**func**main(){

    //调用

    factory:=NewCalcFactory()

    op:=factory.CreateOperate("+")

    op.SetData(1.5,2.0)

    fmt.Println(op.CalcOperate())

    op=factory.CreateOperate("-")

    op.SetData(1.5,2.0)

    fmt.Println(op.CalcOperate())

}

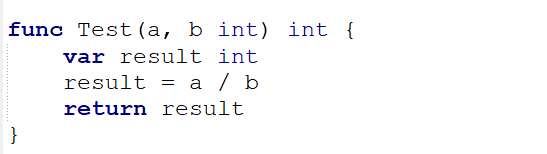
作业：使用面向对象的思想设计超市促销活动，如“满一百减20”，“打八折”等。（一定要先分析类，接口，方法）

在面向对象的编程中，重点理解面向对象编程思想，同时能够理解继承，封装和多态的应用。

### 异常处理

所谓的异常：当GO检测到一个错误时，程序就无法继续执行了，出现了一些错误的提示，这就是所谓的"异常"。

所以为了保证程序的健壮性，要对异常的信息进行处理。例如,如下程序，定义一个函数实现整除操作，这个程序对大家来说已经很简单了，实现如下：



但是，大家仔细考虑一下，该方法是否有问题？

如果b的值为0，会出现什么情况？

程序会出现以下的异常信息：

C:\Users\wangc\AppData\Local\Temp\1524466813(1).png

并且整个程序停止运行。

那么出现这种情况，应该怎样进行处理呢？这时就要用到异常处理方法的内容。

#### error接口

Go语言引入了一个关于错误处理的标准模式，即error接口，它是Go语言内建的接口类型，该接口的定义如下：

type error interface {

Error() string

}

Go语言的标准库代码包errors为用户提供如下方法：



通过以上代码，可以发现error接口的使用是非常简单的（error是一个接口，该接口只声明了一个方法Error()，返回值是string类型，用以描述错误

）。下面看一下基本使用,

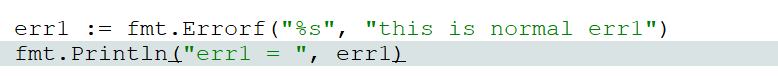
首先导包：

C:\Users\wangc\AppData\Local\Temp\1524475312(1).png

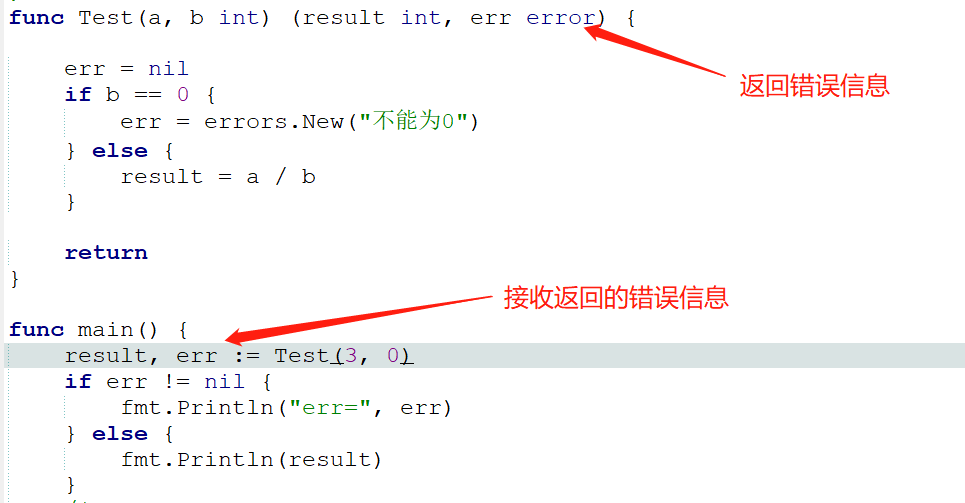
然后调用其对应的方法：

C:\Users\wangc\AppData\Local\Temp\1524475400(1).png

当然fmt包中也封装了一个专门输出错误信息的方法，如下所示：



了解完基本的语法以后，接下来使用error接口解决Test( )函数被0整除的问题，如下所示：



在Test( )函数中，判断变量b的取值，如果有误，返回错误信息。并且在main( )中接收返回的错误信息，并打印出来。

这种用法是非常常见的，例如，后面讲解到文件操作时，涉及到文件的打开，如下：

C:\Users\wangc\AppData\Local\Temp\1524477456(1).png

在打开文件时，如果文件不存在，或者文件在磁盘上存储的路径写错了，都会出现异常，这时可以使用error记录相应的错误信息。

#### panic函数

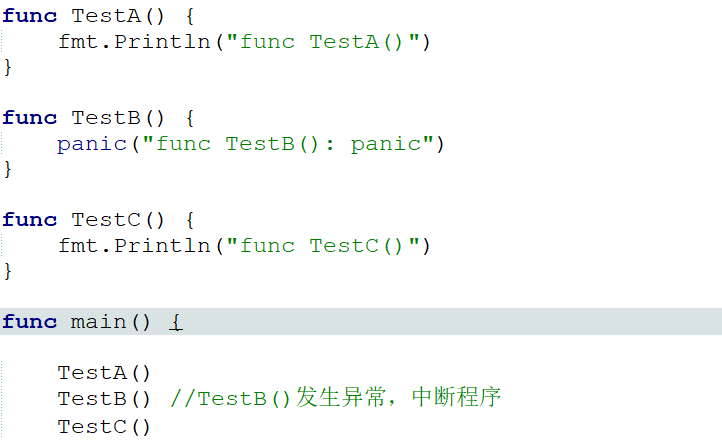
error返回的是一般性的错误，但是panic函数返回的是让程序崩溃的错误。

也就是当遇到不可恢复的错误状态的时候，如数组访问越界、空指针引用等，这些运行时错误会引起painc异常，在一般情况下，我们不应通过调用panic函数来报告普通的错误，而应该只把它作为报告致命错误的一种方式。当某些不应该发生的场景发生时，我们就应该调用panic。

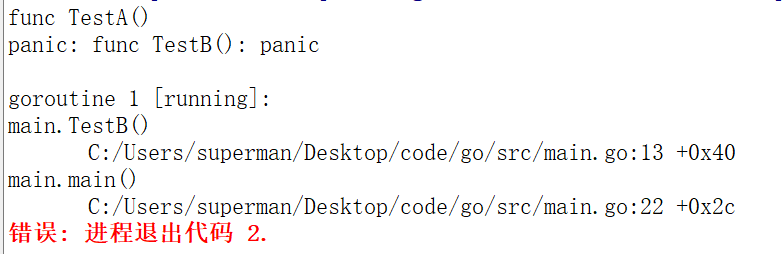
一般而言，当panic异常发生时，程序会中断运行。随后，程序崩溃并输出日志信息。日志信息包括panic value和函数调用的堆栈跟踪信息。

当然，如果直接调用内置的panic函数也会引发panic异常；panic函数接受任何值作为参数。

下面给大家演示一下，直接调用panic函数，是否会导致程序的崩溃。

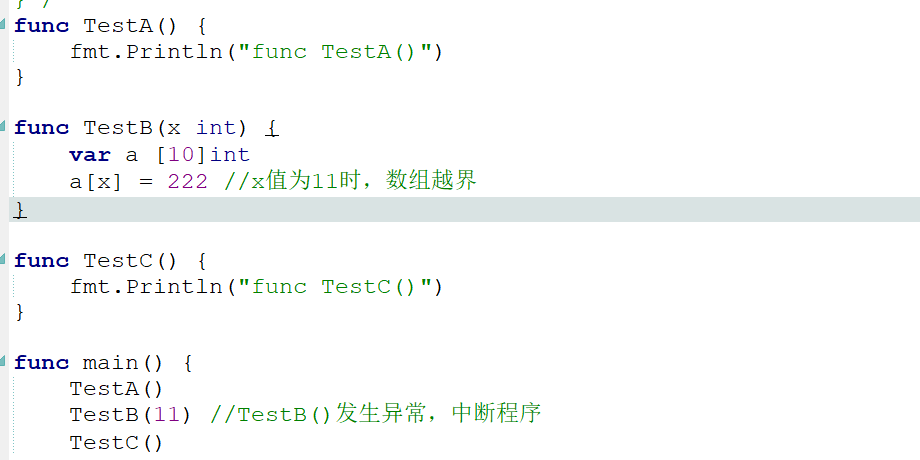


错误信息如下：

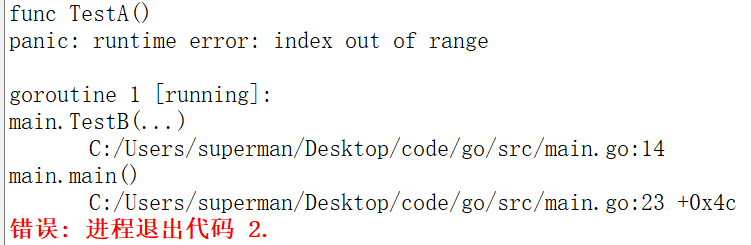


所以，我们在实际的开发过程中并不会直接调用panic( )函数，但是当我们编程的程序遇到致命错误时，系统会自动调用该函数来终止整个程序的运行，也就是系统内置了panic函数。

下面给大家演示一个数组下标越界的问题：



错误信息如下：



通过观察错误信息，发现确实是panic异常，导致了整个程序崩溃。

## Goroutine

### 什么是Goroutine

goroutine是Go语言并行设计的核心，有人称之为go程。 Goroutine从量级上看很像协程，它比线程更小，十几个goroutine可能体现在底层就是五六个线程，Go语言内部帮你实现了这些goroutine之间的内存共享。执行goroutine只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

一般情况下，一个普通计算机跑几十个线程就有点负载过大了，但是同样的机器却可以轻松地让成百上千个goroutine进行资源竞争。

### 创建Goroutine

只需在函数调⽤语句前添加 **go** 关键字，就可创建并发执⾏单元。开发⼈员无需了解任何执⾏细节，调度器会自动将其安排到合适的系统线程上执行。

在并发编程中，我们通常想将一个过程切分成几块，然后让每个goroutine各自负责一块工作，当一个程序启动时，主函数在一个单独的goroutine中运行，我们叫它main goroutine。新的goroutine会用go语句来创建。而go语言的并发设计，让我们很轻松就可以达成这一目的。

示例代码：

**package** main

**import** (

    "fmt"

    "time"

)

**func** newTask() {

    i := 0

**for** {

        i++

        fmt.Printf("new goroutine: i = %d\n", i)

        time.Sleep(1\*time.Second) //延时1s

    }

}

**func** main() {

    //创建一个 goroutine，启动另外一个任务

**go** newTask()

    i := 0

    //main goroutine 循环打印

**for** {

        i++

        fmt.Printf("main goroutine: i = %d\n", i)

        time.Sleep(1 \* time.Second) //延时1s

    }

}

程序运行结果：



### Goroutine特性

**主goroutine退出后，其它的工作goroutine也会自动退出：**

**package main**

**import (**

**"fmt"**

**"time"**

**)**

**func** newTask() {

    i := 0

**for** {

        i++

        fmt.Printf("new goroutine: i = %d\n", i)

        time.Sleep(1 \* time.Second) //延时1s

    }

}

**func** main() {

    //创建一个 goroutine，启动另外一个任务

**go** newTask()

    fmt.Println("main goroutine exit")

}

程序运行结果：

2018-01-15_100253

### Goexit函数

调用 runtime.Goexit() 将立即终止当前 goroutine 执⾏，调度器确保所有已注册 defer 延迟调用被执行。

示例代码：

**package main**

**import (**

**"fmt"**

**"runtime"**

**)**

**func** main() {

**go** **func**() {

**defer** fmt.Println("A.defer")

**func**() {

**defer** fmt.Println("B.defer")

            runtime.Goexit() // 终止当前 goroutine, import "runtime"

            fmt.Println("B") // 不会执行

        }()

        fmt.Println("A") // 不会执行

    }() //不要忘记()

    //死循环，目的不让主goroutine结束

**for** {

    }

}

程序运行结果：

2018-01-15_104908

替换使用return、Goexit() 和 os.Exit() 对比看看它们的作用一样吗？？？

### 练习题

有一个程序，每隔 3 秒，向屏幕打印：“大家好！我是计算机！我很嗨皮!”

当用户键入数据时，该程序不再打印这句话，而打印用户键入的数据。

无用户数据键入时，它继续打印“大家好！我是计算机！我很嗨皮!”

提示： os.Stdin.Read(str) 可以从键盘读取带空格的数据。

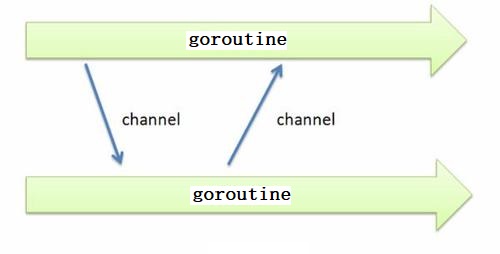
## Channel

channel是Go语言中的一个**核心类型**，可以把它看成管道（pipe）。并发核心单元通过它就可以发送或者接收数据进行通讯，这在一定程度上又进一步降低了编程的难度。

channel是一个数据类型，主要用来解决go程的同步问题以及go程之间数据共享（数据传递）的问题。

goroutine运行在相同的地址空间，因此访问共享内存必须做好同步。goroutine 奉行通过通信来共享内存，而不是共享内存来通信。

引⽤类型 channel可用于多个 goroutine 通讯。其内部实现了同步，确保并发安全。



### 定义channel变量

和map类似，channel也是一个对应make创建的底层数据结构的**引用**。

当我们复制一个channel或用于函数参数传递时，我们只是拷贝了一个channel引用，因此调用者和被调用者将引用同一个channel对象。和其它的引用类型一样，channel的零值也是nil。

定义一个channel时，也需要定义发送到channel的值的类型。channel可以使用内置的make()函数来创建：

**chan**是创建channel所需使用的关键字。Type 代表指定channel收发数据的类型。

    make(**chan** Type) //等价于make(chan Type, 0)

    make(**chan** Type, capacity)

当我们复制一个channel或用于函数参数传递时，我们只是拷贝了一个channel引用，因此调用者和被调用者将引用同一个channel对象。和其它的引用类型一样，channel的零值也是nil。

当 参数capacity= 0 时，channel 是无缓冲阻塞读写的；当capacity > 0 时，channel 有缓冲、是非阻塞的，直到写满 capacity个元素才阻塞写入。

channel非常像生活中的管道，一边可以存放东西，另一边可以取出东西。channel通过操作符 <- 来接收和发送数据，发送和接收数据语法：

    channel <- value      //发送value到channel

    <-channel             //接收并将其丢弃

    x := <-channel        //从channel中接收数据，并赋值给x

    x, ok := <-channel    //功能同上，同时检查通道是否已关闭或者是否为空

默认情况下，channel接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得goroutine同步变的更加的简单，而**不需要显式的lock。**

示例代码：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**defer** fmt.Println("子go程结束")

        fmt.Println("子go程正在运行……")

        c <- 666 //666发送到c

    }()

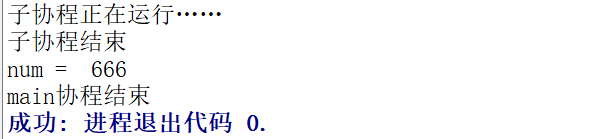
    num := <-c //从c中接收数据，并赋值给num

    fmt.Println("num = ", num)

    fmt.Println("main go程结束")

}

程序运行结果：



### 无缓冲channel

无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何数据值的通道。

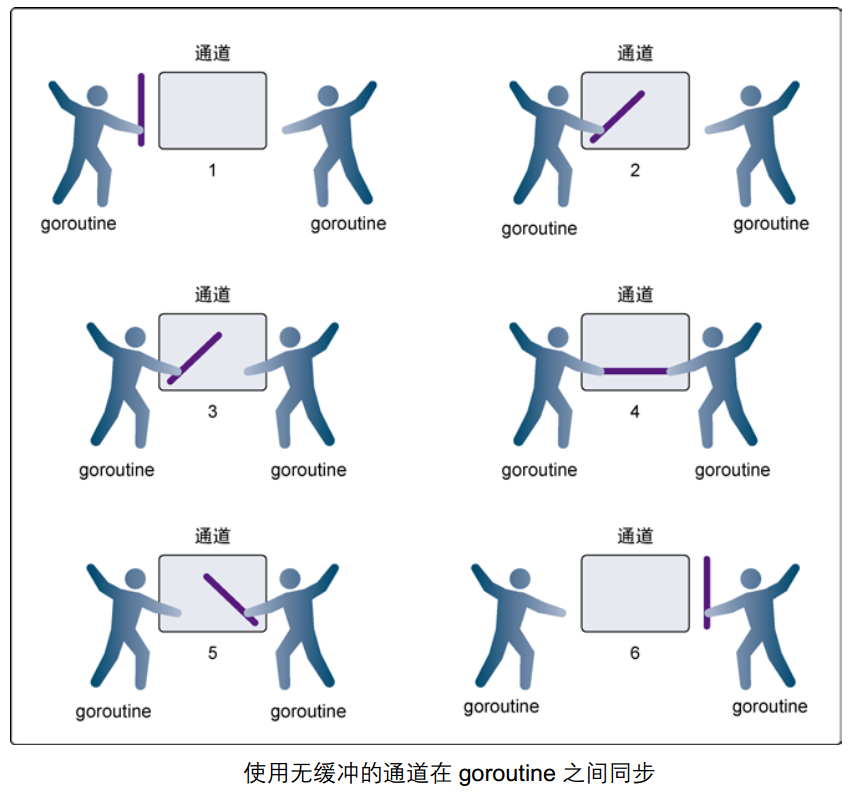
这种类型的通道要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。否则，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。

这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。

**阻塞：**由于某种原因数据没有到达，当前go程（线程）持续处于等待状态，直到条件满足，才解除阻塞。

**同步：**在两个或多个go程（线程）间，保持数据内容一致性的机制。

下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值：



* 在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。
* 在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。
* 在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。
* 在第 4 步和第 5 步，进行交换，并最终，在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做其他事情了。

无缓冲的channel创建格式：

    make(**chan** Type) //等价于make(chan Type, 0)

如果没有指定缓冲区容量，那么该通道就是同步的，因此会阻塞到发送者准备好发送和接收者准备好接收。

示例代码：

**package main**

**import (**

**"fmt"**

**"time"**

**)**

**func** main() {

    c := make(**chan** int, 0) //创建无缓冲的通道 c

    //内置函数 len 返回未被读取的缓冲元素数量，cap 返回缓冲区大小

    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

**go** **func**() {

**defer** fmt.Println("子go程结束")

**for** i := 0; i < 3; i++ {

            c <- i

            fmt.Printf("子go程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))

        }

    }()

    time.Sleep(2 \* time.Second) //延时2s

**for** i := 0; i < 3; i++ {

        num := <-c //从c中接收数据，并赋值给num

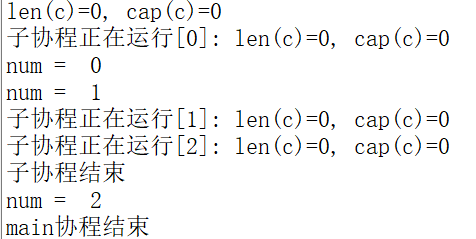
        fmt.Println("num = ", num)

    }

    fmt.Println("main进程结束")

}

程序运行结果：



### 有缓冲channel

有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个数据值的通道。

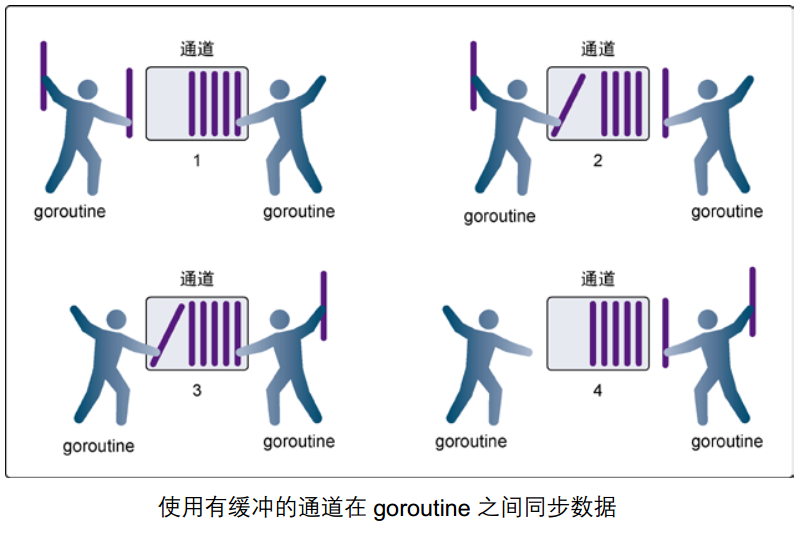
这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也不同。

只有通道中没有要接收的值时，接收动作才会阻塞。

只有通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。

示例图如下：



* 在第 1 步，右侧的 goroutine 正在从通道接收一个值。
* 在第 2 步，右侧的这个 goroutine独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。
* 在第 3 步，左侧的goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。
* 最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓冲的channel创建格式：

    make(**chan** Type, capacity)

如果给定了一个缓冲区容量，通道就是异步的。只要缓冲区有未使用空间用于发送数据，或还包含可以接收的数据，那么其通信就会无阻塞地进行。

借助函数 **len(ch)** 求取缓冲区中剩余元素个数， **cap(ch)** 求取缓冲区元素容量大小。

示例代码：

**func** main() {

    c := make(**chan** int, 3) //带缓冲的通道

    //内置函数 len 返回未被读取的缓冲元素数量， cap 返回缓冲区大小

    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

**go** **func**() {

**defer** fmt.Println("子go程结束")

**for** i := 0; i < 3; i++ {

            c <- i

            fmt.Printf("子go程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))

        }

    }()

    time.Sleep(2 \* time.Second) //延时2s

**for** i := 0; i < 3; i++ {

        num := <-c //从c中接收数据，并赋值给num

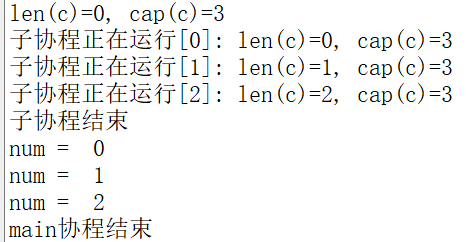
        fmt.Println("num = ", num)

    }

    fmt.Println("main进程结束")

}

程序运行结果：



### 关闭channel

如果发送者知道，没有更多的值需要发送到channel的话，那么让接收者也能及时知道没有多余的值可接收将是有用的，因为接收者可以停止不必要的接收等待。这可以通过内置的close函数来关闭channel实现。

示例代码：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 5; i++ {

            c <- i

        }

        close(c)

    }()

**for** {

        //ok为true说明channel没有关闭，为false说明管道已经关闭

**if** data, ok := <-c; ok {

            fmt.Println(data)

        } **else** {

**break**

        }

    }

    fmt.Println("Finished")

}

程序运行结果：



**注意：**

* channel不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的，才去关闭channel；
* 关闭channel后，无法向channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；
* 关闭channel后，可以继续从channel接收数据；
* 对于nil channel，无论收发都会被阻塞。

可以使用 **range** 来迭代不断操作channel：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 5; i++ {

            c <- i

        }

        close(c)

    }()

**for** data := **range** c {

        fmt.Println(data)

    }

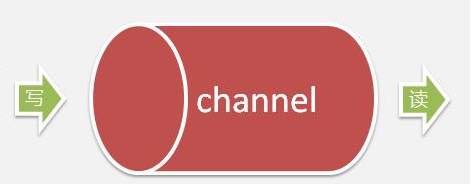
    fmt.Println("Finished")

}

### 单向channel

默认情况下，通道channel是双向的，也就是，既可以往里面发送数据也可以同里面接收数据。

但是，我们经常见一个通道作为参数进行传递而只希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这时候我们可以指定通道的方向。



单向channel变量的声明非常简单，如下：

**var** ch1 **chan** int // ch1是一个正常的channel，是双向的

**var** ch2 **chan**<- float64 // ch2是单向channel，只用于**写**float64数据

**var** ch3 <-**chan** int // ch3是单向channel，只用于**读**int数据

* chan<- 表示数据进入管道，要把数据写进管道，对于调用者就是输出。
* <-chan 表示数据从管道出来，对于调用者就是得到管道的数据，当然就是输入。

可以将 channel 隐式转换为单向队列，只收或只发，不能将单向 channel 转换为普通 channel：

    c := make(**chan** int, 3)

**var** send **chan**<- int = c // send-only

**var** recv <-**chan** int = c // receive-only

    send <- 1

    //<-send //invalid operation: <-send (receive from send-only type chan<- int)

    <-recv

    //recv <- 2 //invalid operation: recv <- 2 (send to receive-only type <-chan int)

    //不能将单向 channel 转换为普通 channel

    d1 := (**chan** int)(send) //cannot convert send (type chan<- int) to type chan int

    d2 := (**chan** int)(recv) //cannot convert recv (type <-chan int) to type chan int

示例代码：

// chan<- //只写

**func** counter(out **chan**<- int) {

**defer** close(out)

**for** i := 0; i < 5; i++ {

        out <- i //如果对方不读 会阻塞

    }

}

// <-chan //只读

**func** printer(in <-**chan** int) {

**for** num := **range** in {

        fmt.Println(num)

    }

}

**func** main() {

    c := make(**chan** int) // chan //读写

**go** counter(c) //生产者

    printer(c) //消费者

    fmt.Println("done")

}

生产者消费者模型

单向channel最典型的应用是“生产者消费者模型”

所谓“生产者消费者模型”: 某个模块（函数等）负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、go程、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。

单单抽象出生产者和消费者，还够不上是生产者／消费者模型。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据。大概的结构如下图：



举一个寄信的例子来辅助理解一下，假设你要寄一封平信，大致过程如下：

1．把信写好——相当于生产者制造数据

2．把信放入邮筒——相当于生产者把数据放入缓冲区

3．邮递员把信从邮筒取出——相当于消费者把数据取出缓冲区

4．邮递员把信拿去邮局做相应的处理——相当于消费者处理数据

那么，这个缓冲区有什么用呢？为什么不让生产者直接调用消费者的某个函数，直接把数据传递过去，而画蛇添足般的设置一个缓冲区呢？

缓冲区的好处大概如下：

**1：解耦**

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产生依赖（也就是耦合）。将来如果消费者的代码发生变化，可能会直接影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合度也就相应降低了。

接着上述的例子，如果不使用邮筒（缓冲区），须得把信直接交给邮递员。那你就必须要认识谁是邮递员。这就产生和你和邮递员之间的依赖（相当于生产者和消费者的强耦合）。万一哪天邮递员换人了，你还要重新认识下一个邮递员（相当于消费者变化导致修改生产者代码）。 而邮筒相对来说比较固定，你依赖它的成本也比较低（相当于和缓冲区之间的弱耦合）。

**2：处理并发**

生产者直接调用消费者的某个方法，还有另一个弊端。由于函数调用是同步的（或者叫阻塞的），在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者只能无端浪费时间。

使用了生产者／消费者模式之后，生产者和消费者可以是两个独立的并发主体。生产者把制造出来的数据往缓冲区一丢，就可以再去生产下一个数据。基本上不用依赖消费者的处理速度。

其实最当初这个生产者消费者模式，主要就是用来处理并发问题的。

从寄信的例子来看。如果没有邮筒，你得拿着信傻站在路口等邮递员过来收（相当于生产者阻塞）；又或者邮递员得挨家挨户问，谁要寄信（相当于消费者轮询）。

**3：缓存**

如果生产者制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

假设邮递员一次只能带走1000封信。万一某次碰上情人节送贺卡，需要寄出去的信超过1000封，这时候邮筒这个缓冲区就派上用场了。邮递员把来不及带走的信暂存在邮筒中，等下次过来时再拿走。

**示例代码：**

**package** main  
  
**import "fmt"**// 此通道只能写，不能读。  
**func** producer(out **chan**<- int) {  
 **for** i:= 0; i < 10; i++ {  
 out <- i\*i // 将 i\*i 结果写入到只写channel  
}  
 close(out)  
}  
  
// 此通道只能读，不能写  
**func** consumer(in <-**chan** int) {  
 **for** num := **range** in { // 从只读channel中获取数据  
 fmt.Println(**"num ="**, num)  
 }  
}  
  
**func** main() {  
ch := make(**chan** int) // 创建一个双向channel  
  
 // 新建一个groutine， 模拟生产者，产生数据，写入 channel  
**go** producer(ch) // channel传参， 传递的是引用。 // 主go程，模拟消费者，从channel读数据，打印到屏幕  
consumer(ch) // 与 producer 传递的是同一个 channel  
}

简单说明：首先创建一个双向的channel，然后开启一个新的goroutine，把双向通道作为参数传递到producer方法中，同时转成只写通道。子go程开始执行循环，向只写通道中添加数据，这就是生产者。主go程，直接调用consumer方法，该方法将双向通道转成只读通道，通过循环每次从通道中读取数据，这就是消费者。

注意：channel作为参数传递，是**引用传递**。

### 定时器

Timer是一个定时器。代表未来的一个单一事件，你可以告诉timer你要等待多长时间。

**type** Timer **struct** {  
 C <-**chan** Time  
 r runtimeTimer  
}

它提供一个channel，在定时时间到达之前，没有数据写入timer.C会一直阻塞。直到定时时间到，系统会自动向timer.C 这个channel中写入**当前时间**，阻塞即被解除。

示例代码：

**package main**

**import (**

"fmt"

"time"

**)**

**func** main() {

    //创建定时器，2秒后，定时器就会向自己的C字节发送一个time.Time类型的元素值

    timer1 := time.NewTimer(time.Second \* 2)

    t1 := time.Now() //当前时间

    fmt.Printf("t1: %v\n", t1)

    t2 := <-timer1.C

    fmt.Printf("t2: %v\n", t2)

    //如果只是想单纯的等待的话，可以使用 time.Sleep 来实现

    timer2 := time.NewTimer(time.Second \* 2)

    <-timer2.C

    fmt.Println("2s后")

    time.Sleep(time.Second \* 2)

    fmt.Println("再一次2s后")

    <-time.After(time.Second \* 2)

    fmt.Println("再再一次2s后")

    timer3 := time.NewTimer(time.Second)

**go** **func**() {

        <-timer3.C

        fmt.Println("Timer 3 expired")

    }()

    stop := timer3.Stop() //停止定时器

**if** stop {

        fmt.Println("Timer 3 stopped")

    }

    fmt.Println("before")

    timer4 := time.NewTimer(time.Second \* 5) //原来设置3s

    timer4.Reset(time.Second \* 1) //重新设置时间

    <-timer4.C

    fmt.Println("after")

}

定时器的常用操作：

1. 实现延迟功能
2. <-time.After(2 \* time.Second) //定时2s，阻塞2s,2s后产生一个事件，往channel写内容

fmt.Println("时间到")

1. time.Sleep(2 \* time.Second)

fmt.Println("时间到")

1. 延时2s后打印一句话

timer := time.NewTimer(2 \* time.Second)

<- timer.C

fmt.Println("时间到")

1. 定时器停止

timer := time.NewTimer(3 \* time.Second)

   go func() {

        <-timer.C

        fmt.Println("子go程可以打印了，因为定时器的时间到")

    }()

   timer.Stop() //停止定时器

    for {

}

1. 定时器重置

timer := time.NewTimer(3 \* time.Second)

ok := timer.Reset(1 \* time.Second) //重新设置为1s

fmt.Println("ok = ", ok)

<-timer.C

fmt.Println("时间到")

### 练习题

编程实现，主、子go程交替数数。

假设有全局变量 glb = 1 初值为1。

子go程打印1，主go程打印 2，子go程打印3， 主go程打印 4， 子go程打印 5， 主go程 打印 6 ......

以此类推。要求不使用 sleep函数，完成正常打印。

### select

select作用

Go里面提供了一个关键字select，通过select可以监听channel上的数据流动。

有时候我们希望能够借助channel发送或接收数据，并避免因为发送或者接收导致的阻塞，尤其是当channel没有准备好写或者读时。select语句就可以实现这样的功能。

select的用法与switch语言非常类似，由select开始一个新的选择块，每个选择条件由case语句来描述。

与switch语句相比，select有比较多的限制，其中最大的一条限制就是每个**case语句里必须是一个IO操作**，大致的结构如下：

**select** {

**case** <- chan1:

        // 如果chan1成功读到数据，则进行该case处理语句

**case** chan2 <- 1:

        // 如果成功向chan2写入数据，则进行该case处理语句

**default**:

        // 如果上面都没有成功，则进入default处理流程

    }

在一个select语句中，Go语言会按顺序从头至尾评估每一个发送和接收的语句。

如果其中的任意一语句可以继续执行(即没有被阻塞)，那么就从那些可以执行的语句中任意选择一条来使用。

如果没有任意一条语句可以执行(即所有的通道都被阻塞)，那么有两种可能的情况：

* 如果给出了default语句，那么就会执行default语句，同时程序的执行会从select语句后的语句中恢复。
* 如果没有default语句，那么select语句将被阻塞，直到至少有一个通信可以进行下去。

示例代码：

**package main**

**import (**

"fmt"

**)**

**func** fibonacci(c, quit **chan** int) {

    x, y := 1, 1

**for** {

**select** {

**case** c <- x:

            x, y = y, x+y

**case** <-quit:

            fmt.Println("quit")

**return**

        }

    }

}

**func** main() {

    c := make(**chan** int)

    quit := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 6; i++ {

            fmt.Println(<-c)

        }

        quit <- 0

    }()

    fibonacci(c, quit)

}

运行结果如下：



超时

有时候会出现goroutine阻塞的情况，那么我们如何避免整个程序进入阻塞的情况呢？我们可以利用select来设置超时，通过如下的方式实现：

**func** main() {

    c := make(**chan** int)

    out := make(**chan** bool)

**go** **func**() {

**for** {

**select** {

**case** v := <-c:

                fmt.Println(v)

**case** <-time.After(5 \* time.Second):

                fmt.Println("timeout")

                out <- true

**return**

            }

        }

    }()

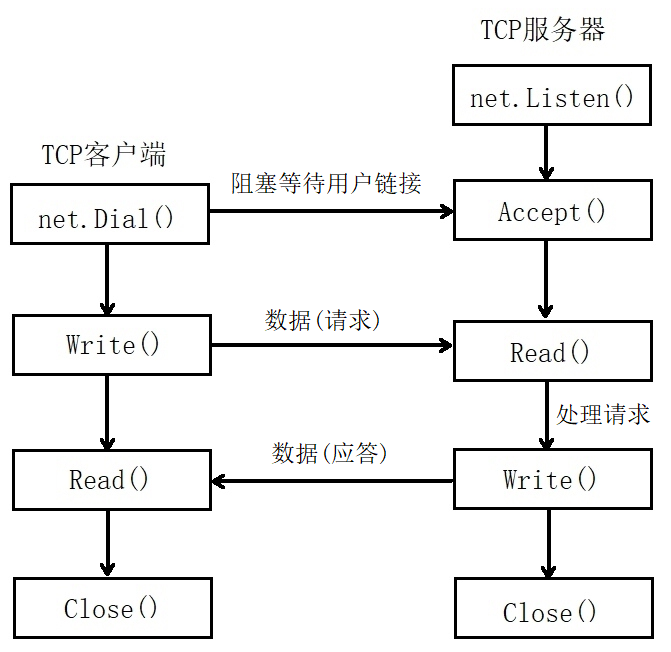
    //c <- 666 // 注释掉，引发 timeout

    <-out

}

## Socket

### 简单C/S模型通信



**TCP的C/S架构**

Server端：

Listen函数：

**func Listen(network, address** string**) (Listener, error)**

network：选用的协议：TCP、UDP， 如：“tcp”**注意**：只支持小写字母

address：IP地址+端口号, 如：“127.0.0.1:8000”或 “:8000”

Listener 接口：

type Listener interface {

**Accept**() (**Conn**, error)

**Close**() error

**Addr**() Addr

}

**Conn** 接口：

type Conn interface {

**Read**(b []byte) (n int, err error)

**Write**(b []byte) (n int, err error)

**Close**() error

LocalAddr() Addr

RemoteAddr() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

参看 <https://studygolang.com/pkgdoc> 中文帮助文档中的demo：

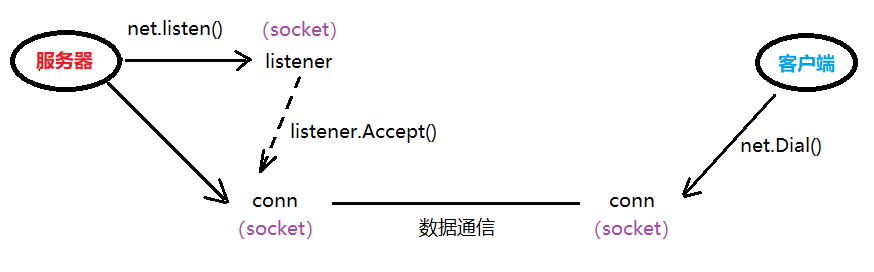
示例代码：

TCP服务器.go

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 // 创建监听  
listener, err:= net.Listen(**"tcp"**, **":8000"**) // tcp 不能使用大写  
 **if** err != nil {  
 fmt.Println(**"listen err:"**, err)  
 **return** }  
 **defer** listener.Close() // 主go程结束时，关闭listener  
fmt.Println(**"服务器等待客户端建立连接..."**)  
 // 等待客户端连接请求  
conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"accept err:"**, err)  
 **return** }

**defer** conn.Close() // 使用结束，断开与客户端链接  
 fmt.Println(**"客户端与服务器连接建立成功..."**)  
  
 // 接收客户端数据  
buf := make([]byte, 1024) // 创建1024大小的缓冲区，用于read  
n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"read err:"**, err)  
 **return** }  
 fmt.Println(**"服务器读到:"**, string(buf[:n])) // 读多少，打印多少。}

如图，在整个通信过程中，服务器端有两个socket参与进来，但用于通信的只有 conn 这个socket。它是由 listener创建的。隶属于服务器端。



Client 端：

Dial函数：

**func Dial(network, address** string**) (Conn, error)**

network：选用的协议：TCP、UDP，如：“tcp”或 “udp”

address：**服务器**IP地址+端口号, 如：“121.36.108.11:8000”或 “www.itcast.cn:8000”

Conn 接口：

type Conn interface {

**Read**(b []byte) (n int, err error)

**Write**(b []byte) (n int, err error)

**Close**() error

LocalAddr() Addr

RemoteAddr() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 // 主动发起连接请求  
conn, err := net.Dial(**"tcp"**, **"127.0.0.1:8000"**) // tcp 不能使用大写  
 **if** err != nil {  
 fmt.Println(**"Dial err:"**, err)  
 **return** }  
 **defer** conn.Close() // 结束时，关闭连接  
 // 发送数据  
\_, err = conn.Write([]byte(**"Are u ready?"**))   
 **if** err != nil {  
 fmt.Println(**"Write err:"**, err)  
 **return** }  
}

### 并发C/S模型通信

并发Server

现在已经完成了客户端与服务端的通信，但是服务端只能接收一个用户发送过来的数据，怎样接收多个客户端发送过来的数据，实现一个高效的并发服务器呢？

Accept()函数的作用是等待客户端的链接，如果客户端没有链接，该方法会阻塞。如果有客户端链接，那么该方法返回一个Socket负责与客户端进行通信。所以，每来一个客户端，该方法就应该返回一个Socket与其通信，因此，可以使用一个死循环，将Accep8t()调用过程包裹起来。

需要注意的是，实现并发处理多个客户端数据的服务器，就需要**针对每一个客户端连接，单独产生一个Socket，并创建一个单独的goroutine与之完成通信。**

//监听

    listener, err := net.Listen("tcp", "127.0.0.1:8001") // tcp 不能使用大写

**if** err != nil {

         fmt.Println("err = ", err)

**return**

    }

**defer** listener.Close()

    //接收多个用户

**for** {

         conn, err := listener.Accept()

**if** err != nil {

           fmt.Println("err = ", err)

**return**

         }

         //处理用户请求, 新建一个go程

**go** HandleConn(conn)

}

将客户端的数据处理工作封装到HandleConn方法中，需将Accept()返回的Socket传递给该方法，变量conn的类型为：net.Conn。可以使用conn.RemoteAddr()来获取成功与服务器建立连接的客户端IP地址和端口号：

Conn 接口：

type Conn interface {

Read(b []byte) (n int, err error)

Write(b []byte) (n int, err error)

Close() error

LocalAddr() Addr

**RemoteAddr**() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

    //获取客户端的网络地址信息

    addr := conn.RemoteAddr().String()

    fmt.Println(addr, " conncet sucessful")

客户端可能持续不断的发送数据，因此接收数据的过程可以放在for循环中，服务端也持续不断的向客户端返回处理后的数据。

添加一个限定，如果客户端发送一个“exit”字符串，表示客户端通知服务器不再向服务端发送数据，此时应该结束HandleConn方法，同时关闭与该客户端关联的Socket。

buf := make([]byte, 2048)  //创建一个切片，存储客户端发送的数据

**for** {

        //读取用户数据

        n, err := conn.Read(buf)

**if** err != nil {

            fmt.Println("err = ", err)

**return**

        }

        fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))

**if** "exit" == string(buf[:n-2]) { //自己写的客户端测试, 发送时，多了2个字符, "\r\n"

            fmt.Println(addr, " exit")

**return**

        }

        //服务器处理数据：把客户端数据转大写，再写回给client

        conn.Write([]byte(strings.ToUpper(string(buf[:n]))))

 }

在上面的代码中，Read()方法获取客户端发送过来的数据，填充到切片buf中，返回的是实际填充的数据的长度，所以将客户端发送过来的数据进行打印，打印的是实际接收到的数据。

fmt.Printf("[%s]: %s\n", addr, string(buf[:n])).同时也可以将客户端的网络地址信息打印出来。

在判断客户端数据是否为“exit”字符串时，要注意，客户端会自动的多发送2个字符：“\r\n”（这在windows系统下代表回车、换行）

Server使用Write方法将数据写回给客户端，参数类型是 []byte，需使用strings包下的ToUpper函数来完成大小写转换。转换的对象即为string(buf[:n])

综上，HandleConn方法完整定义如下：

//处理用户请求

**func** HandleConn(conn net.Conn) {

//函数调用完毕，自动关闭conn

**defer** conn.Close()

     //获取客户端的网络地址信息

     addr := conn.RemoteAddr().String()

      fmt.Println(addr, " conncet sucessful")

     buf := make([]byte, 2048)

**for** {

         //读取用户数据

         n, err := conn.Read(buf)

**if** err != nil {

             fmt.Println("err = ", err)

**return**

         }

          fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))

          fmt.Println("len = ", len(string(buf[:n])))

         //if "exit" == string(buf[:n-1]) { // nc测试，发送时，只有 \n

**if** "exit" == string(buf[:n-2]) { // 自己写的客户端测试, 发送时，多了2个字符, "\r\n"

             fmt.Println(addr, " exit")

**return**

         }

         //把数据转换为大写，再给用户发送

         conn.Write([]byte(strings.ToUpper(string(buf[:n]))))

     }

}

### 练习题

实现类似 nc 命令的客户端程序（IP地址和端口号可设为固定），能动态获取用户键盘输入数据，

并将用户输入的实际数据发送给服务器。获取到服务器回发的数据（如大写）后，显示到屏幕。

当用户键入“exit”时，退出当前程序，并关闭与服务器的连接。

### 网络文件传输

#### 获取文件属性和命令行参数

首先获取文件名。借助os包中的stat()函数来获取文件属性信息。在函数返回的文件属性中包含文件名和文件大小。Stat参数name传入的是文件访问的绝对路径。FileInfo中的Name()函数可以将不含路径的文件名单独提取出来。

**func** Stat(name string) (**FileInfo**, error) //格式校验 size

**type FileInfo** **interface** {  
 **Name()** string

**Size()** int64

Mode() FileMode ModTime() time.Time IsDir() bool Sys() **interface**{} }

获取文件属性示例：

**package** main  
  
**import** (  
 **"os"  
 "fmt"**)  
  
**func** main() {  
 list := os.Args *// 获取命令行参数，存入list中* **if** len(list) != 2 {  *// 确保用户输入了一个命令行参数* fmt.Println(**"格式为：xxx.go 文件名"**)  
 **return** }  
 fileName := list[1] *// 从命令行保存文件名(含路径)* fileInfo, err := os.Stat(fileName) *//根据文件名获取文件属性信息 fileInfo* **if** err != nil {  
 fmt.Println(**"os.Stat err:"**, err)  
 **return** }  
 fmt.Println(**"文件name为："**, fileInfo.Name()) *// 得到文件名(不含路径)* fmt.Println(**"文件size为："**, fileInfo.Size()) *// 得到文件大小。单位字节*}

#### 流程简析

借助TCP完成文件的传输，基本思路如下：

1：发送方（客户端）向服务端发送文件名，服务端保存该文件名。

2：接收方（服务端）向客户端返回一个消息ok，确认文件名保存成功。

3：发送方（客户端）收到消息后，开始向服务端发送文件数据。

4：接收方（服务端）读取文件内容，写入到之前保存好的文件中。

/home/itcast/4go13-按行读文件.avi /home/itcast/4go/home/itcast/4go

发送端实现

实现流程大致如下：

1. 提示用户输入文件名。接收文件名path（含访问路径）
2. 使用os.Stat()获取文件属性，得到纯文件名（去除访问路径）
3. 主动连接服务器，结束时关闭连接
4. 给接收端（服务器）发送文件名conn.Write()
5. 读取接收端回发的确认数据conn.Read()
6. 判断是否为“ok”。如果是，封装函数SendFile() 发送文件内容。传参path和conn
7. 只读Open文件, 结束时Close文件
8. 循环读文件，读到EOF终止文件读取
9. 将读到的内容原封不动Write给接收端（服务器）

代码实现：

**package** main  
  
**import** (  
 **"fmt"  
 "os"  
 "net"  
 "io"**)  
  
**func** SendFile(path string, conn net.Conn) {  
 *// 以只读方式打开文件* f, err := os.Open(path)  
 **if** err != nil {  
 fmt.Println(**"os.Open err:"**, err)  
 **return** }  
 **defer** f.Close() *// 发送结束关闭文件。  
  
 // 循环读取文件，原封不动的写给服务器* buf := make([]byte, 4096)  
 **for** {  
 n, err := f.Read(buf) *// 读取文件内容到切片缓冲中* **if** err != nil {  
 **if** err == io.EOF {  
 fmt.Println(**"文件发送完毕"**)  
 } **else** {  
 fmt.Println(**"f.Read err:"**, err)  
 }  
 **return** }  
 conn.Write(buf[:n]) *// 原封不动写给服务器* }  
}  
  
**func** main() {  
 *// 提示输入文件名* fmt.Println(**"请输入需要传输的文件："**)  
 **var** path string  
 fmt.Scan(&path)  
  
 *// 获取文件名 fileInfo.Name()* fileInfo, err := os.Stat(path)  
 **if** err != nil {  
 fmt.Println(**"os.Stat err:"**, err)  
 **return** }  
  
 *// 主动连接服务器* conn, err := net.Dial(**"tcp"**, **"127.0.0.1:8005"**)  
 **if** err != nil {  
 fmt.Println(**"net.Dial err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 *// 给接收端，先发送文件名* \_, err = conn.Write([]byte(fileInfo.Name()))  
 **if** err != nil {  
 fmt.Println(**"conn.Write err:"**, err)  
 **return** }  
  
 *// 读取接收端回发确认数据 —— ok* buf := make([]byte, 1024)  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"conn.Read err:"**, err)  
 **return** }  
  
 *// 判断如果是ok，则发送文件内容* **if "ok"** == string(buf[:n]) {  
 SendFile(path, conn) *// 封装函数读文件，发送给服务器，需要path、conn* }  
}

接收端实现

实现流程大致如下：

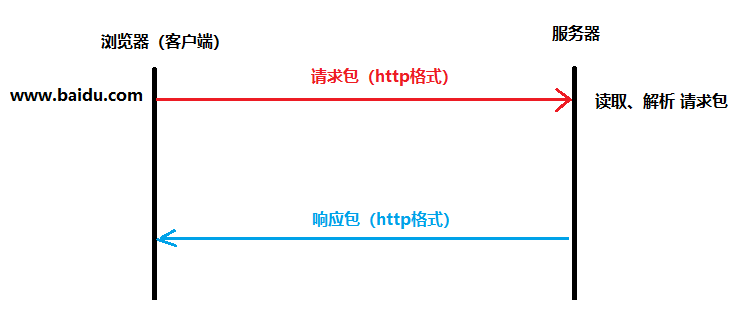
1. 创建监听listener，程序结束时关闭。
2. 阻塞等待客户端连接，程序结束时关闭conn。
3. 读取客户端发送文件名。保存fileName。
4. 回发“ok”给客户端做应答
5. 封装函数 RecvFile接收客户端发送的文件内容。传参fileName 和conn
6. 按文件名Create文件，结束时Close
7. 循环Read客户端发送的文件内容，当读到EOF说明文件读取完毕。
8. 将读到的内容原封不动Write到创建的文件中

代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"  
 "os"  
 "io"**)  
  
**func** RecvFile(fileName string, conn net.Conn) {  
 *// 创建新文件* f, err := os.Create(fileName)  
 **if** err != nil {  
 fmt.Println(**"Create err:"**, err)  
 **return** }  
 **defer** f.Close()  
  
 *// 接收客户端发送文件内容，原封不动写入文件* buf := make([]byte, 4096)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 **if** err == io.EOF {  
 fmt.Println(**"文件接收完毕"**)  
 } **else** {  
 fmt.Println(**"Read err:"**, err)  
 }  
 **return** }  
 f.Write(buf[:n]) *// 写入文件，读多少写多少* }  
}  
  
**func** main() {  
 *// 创建监听* listener, err := net.Listen(**"tcp"**, **"127.0.0.1:8005"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listener.Close()  
  
 *// 阻塞等待客户端连接* conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 *// 读取客户端发送的文件名* buf := make([]byte, 1024)  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 fileName := string(buf[:n]) *// 保存文件名  
  
 // 回复 0k 给发送端* conn.Write([]byte(**"ok"**))  
  
 *// 接收文件内容* RecvFile(fileName, conn) *// 封装函数接收文件内容， 传fileName 和 conn*}

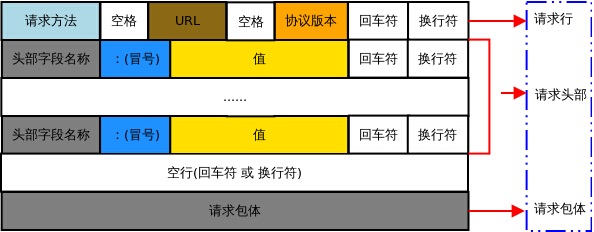
## http

### HTTP协议



### 请求报文格式说明

HTTP 请求报文由请求行、请求头部、空行、请求包体4个部分组成，如下图所示：



1. **请求行**

请求行由方法字段、URL字段 和HTTP 协议版本字段 3个部分组成，他们之间使用空格隔开。常用的 HTTP 请求方法有 GET、POST。

GET：

* 当客户端要从服务器中读取某个资源时，使用GET 方法。GET 方法要求服务器将URL 定位的资源放在响应报文的数据部分，回送给客户端，即向服务器请求某个资源。
* 使用GET方法时，请求参数和对应的值附加在 URL 后面，利用一个问号(“?”)代表URL 的结尾与请求参数的开始，传递参数长度受限制，因此GET方法不适合用于上传数据。
* 通过GET方法来获取网页时，参数会显示在浏览器地址栏上，因此保密性很差。

POST：

* 当客户端给服务器提供信息较多时可以使用POST 方法，POST 方法向服务器提交数据，比如完成表单数据的提交，将数据提交给服务器处理。
* GET 一般用于获取/查询资源信息，POST 会附带用户数据，一般用于更新资源信息。POST 方法将请求参数封装在HTTP 请求数据中，而且长度没有限制，因为POST携带的数据，在HTTP的请求正文中，以名称/值的形式出现，可以传输大量数据。

**Put,delete,head,patch**

1. **请求头部**

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

| **请求头** | **含义** |
| --- | --- |
| User-Agent | 请求的浏览器类型 |
| Accept | 客户端可识别的响应内容类型列表，星号“ \* ”用于按范围将类型分组，用“ \*/\* ”指示可接受全部类型，用“ type/\* ”指示可接受 type 类型的所有子类型 |
| Accept-Language | 客户端可接受的自然语言 |
| Accept-Encoding | 客户端可接受的编码压缩格式 |
| Accept-Charset | 可接受的应答的字符集 |
| Host | 请求的主机名，允许多个域名同处一个IP 地址，即虚拟主机 |
| connection | 连接方式(close或keepalive) |
| Cookie | 存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie |

1. **空行**

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

1. **请求包体**

请求包体不在GET方法中使用，而在POST方法中使用。POST方法适用于需要客户填写表单的场合。与请求包体相关的最常使用的是包体类型Content-Type和包体长度Content-Length。

#### 获取请求报文

为了更直观的看到浏览器发送的请求包，我们借助前面学习的TCP通信模型，编写一个简单的web服务器，只接收浏览器发送的内容，打印查看。

服务器测试代码：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 *//创建、监听socket* listenner, err := net.Listen(**"tcp"**, **"127.0.0.1:8000"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listenner.Close()  
  
 *//阻塞等待客户端连接* conn, err := listenner.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 fmt.Println(conn.RemoteAddr().String(), **"连接成功"**) *//连接客户端的网络地址* buf := make([]byte, 4096) *//切片缓冲区，接收客户端发送数据* n, err := conn.Read(buf) *//n 接收数据的长度* **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 result := buf[:n] *//切片截取* fmt.Printf(**"#\n%s#"**, string(result))  
}

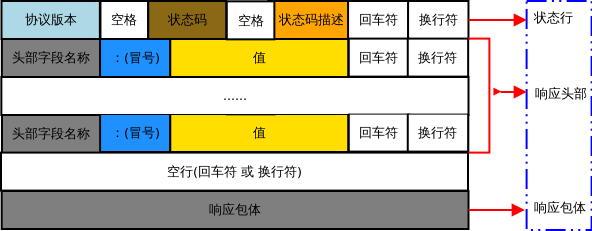
在浏览器中输入url地址： 127.0.0.1:8000

服务器端运行打印结果如下：



### 响应报文格式说明

HTTP 响应报文由状态行、响应头部、空行、响应包体4个部分组成，如下图所示：



1. 状态行

状态行由 HTTP 协议版本字段、状态码和状态码的描述文本3个部分组成，他们之间使用空格隔开。

状态码：状态码由三位数字组成，第一位数字表示响应的类型，常用的状态码有五大类如下所示：

| **状态码** | **含义** |
| --- | --- |
| 1xx | 表示服务器已接收了客户端请求，客户端可继续发送请求 |
| 2xx | 表示服务器已成功接收到请求并进行处理 |
| 3xx | 表示服务器要求客户端重定向 |
| 4xx | 表示客户端的请求有非法内容 |
| 5xx | 表示服务器未能正常处理客户端的请求而出现意外错误 |

常见的状态码举例：

| **状态码** | **含义** |
| --- | --- |
| 200 OK | 客户端请求成功 |
| 400 Bad Request | 请求报文有语法错误 |
| 401 Unauthorized | 未授权 |
| 403 Forbidden | 服务器拒绝服务 |
| 404 Not Found | 请求的资源不存在 |
| 500 Internal Server Error | 服务器内部错误 |
| 503 Server Unavailable | 服务器临时不能处理客户端请求(稍后可能可以) |

1. 响应头部

响应头可能包括：

| **响应头** | **含义** |
| --- | --- |
| Location | Location响应报头域用于重定向接受者到一个新的位置 |
| Server | Server 响应报头域包含了服务器用来处理请求的软件信息及其版本 |
| Vary | 指示不可缓存的请求头列表 |
| Connection | 连接方式 |

1. 空行

最后一个响应头部之后是一个空行，发送回车符和换行符，通知服务器以下不再有响应头部。

1. 响应包体

服务器返回给客户端的文本信息。

### HTTP服务端

Go语言标准库内建提供了net/http包，涵盖了HTTP客户端和服务端的具体实现。使用net/http包，我们可以很方便地编写HTTP客户端或服务端的程序。

示例代码：

**package** main

**import** (

    "fmt"

    "net/http"

)

//服务端编写的业务逻辑处理程序 —— 回调函数

**func** myHandler(w http.ResponseWriter, r \*http.Request) {

    fmt.Println("method = ", r.Method) //请求方法

    fmt.Println("URL = ", r.URL) // 浏览器发送请求文件路径

    fmt.Println("header = ", r.Header) // 请求头

    fmt.Println("body = ", r.Body) // 请求包体

    fmt.Println(r.RemoteAddr, "连接成功") //客户端网络地址

    w.Write([]byte("hello http")) //给客户端回复数据

}

**func** main() {

    http.HandleFunc("/hello", myHandler) // 注册处理函数

    //该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理传入的连接请求。

    //该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为nil

    //当参2为nil时，服务端调用 http.DefaultServeMux 进行处理

    http.ListenAndServe("127.0.0.1:8000", nil)

}

浏览器输入url地址：127.0.0.1:8000/hello

回调函数myHandler的函数原型固定。**func** myHandler(w http.ResponseWriter, r \*http.Request)

参数：w http.ResponseWriter用来“给客户端回发数据”。它是一个interface：

**type** ResponseWriter **interface** {Header() Header **Write([]byte) (int, error)** WriteHeader(int)   
}

参数：r \*http.Request 用来“接收客户端发送的数据”。浏览器发送给服务器的http请求包的内容可以借助r来查看。它对应一个结构体：

**type** Request **struct** {

**Method string** // 浏览器请求方法 GET、POST…

**URL \*url.URL** // 浏览器请求的访问路径

……

**Header Header** // 请求头部

**Body io.ReadCloser** // 请求包体

**RemoteAddr string** // 浏览器地址

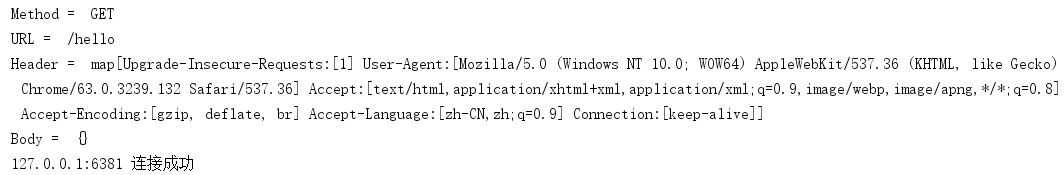
……

ctx context.Context  
}

查看一下结构体成员：

fmt.Println(**"Method = "**, r.Method)  
fmt.Println(**"URL = "**, r.URL)  
fmt.Println(**"Header = "**, r.Header)  
fmt.Println(**"Body = "**, r.Body)  
fmt.Println(r.RemoteAddr, **"连接成功"**)

查看到如下内容：



**练习：**

在计算机中选定一个目录，存放jpg、png、txt、mp3、gif、m4a等类型文件。编写一个服务器程序，可以给浏览器提供该目录下文件的访问服务。

如：目录中存有图片文件：lf.jpg。用户在浏览器中输入：127.0.0.1:8000**/**lf.jpg 可以查看该图片。

### HTTP客户端

客户端访问web服务器数据，主要使用**func Get**(url string) (resp \*Response, err error)函数来完成。读到的响应报文数据被保存在 Response 结构体中。

**type** Response **struct** {  
 Status string *// e.g. "200 OK"* StatusCode int *// e.g. 200* Proto string *// e.g. "HTTP/1.0"* ……Header HeaderBody io.ReadCloser ……  
}

服务器发送的响应包体被保存在Body中。可以使用它提供的Read方法来获取数据内容。保存至切片缓冲区中，拼接成一个完整的字符串来查看。

结束的时候，需要调用Body中的Close()方法关闭io。

示例代码：

**package** main  
  
**import** (  
 **"net/http"  
 "fmt"  
 "io"**)  
  
**func** main() {  
 *// 使用Get方法获取服务器响应包数据  
 //resp, err := http.Get("http://www.baidu.com")* resp, err := http.Get(**"http://127.0.0.1:8000/hello"**)  
 **if** err != nil {  
 fmt.Println(**"Get err:"**, err)  
 **return** }  
 **defer** resp.Body.Close()  
  
 *// 获取服务器端读到的数据* fmt.Println(**"Status = "**, resp.Status) *// 状态* fmt.Println(**"StatusCode = "**, resp.StatusCode) *// 状态码* fmt.Println(**"Header = "**, resp.Header) *// 响应头部* fmt.Println(**"Body = "**, resp.Body) *// 响应包体* buf := make([]byte, 4096) *// 定义切片缓冲区，存读到的内容* **var** result string  
 *// 获取服务器发送的数据包内容* **for** {  
 n, err := resp.Body.Read(buf) *// 读body中的内容。* **if** n == 0 {  
 fmt.Println(**"--Read finish!"**)  
 **break** }  
 **if** err != nil && err != io.EOF {  
 fmt.Println(**"resp.Body.Read err:"**, err)  
 **return** }  
 result += string(buf[:n]) *// 累加读到的数据内容* }  
 *// 打印从body中读到的所有内容* fmt.Println(**"result = "**, result)  
}

## Json

### Json概述

Xml

JSON ：JavaScript对象表示法（ JavaScript Object Notation ）。

是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。 JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C、C++、C#、Java、JavaScript、Perl、Python、go等）。这些特性使 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成 ( 一般用于提升网络传输速率 ) 。目前，json 已经成为主流的数据格式。

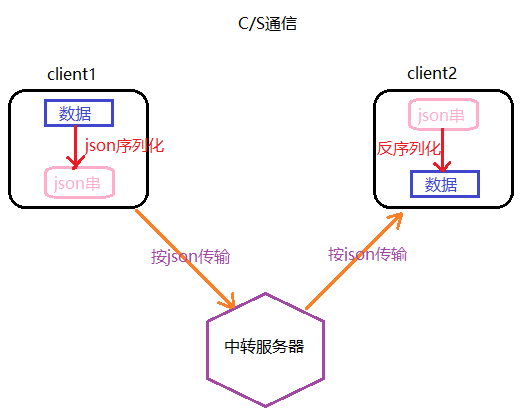
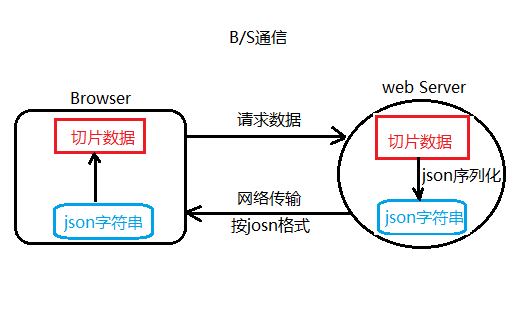
JSON 解析器和 JSON 库支持许多不同的编程语言。 JSON 文本格式在语法上与创建 JavaScript 对象的代码相同。由于这种相似性，无需解析器，JavaScript 程序能够使用内建的 eval() 函数，用 JSON 数据来生成原生的 JavaScript 对象。JSON 是存储和交换文本信息的语法。比 XML 更小、更快，更易解析。

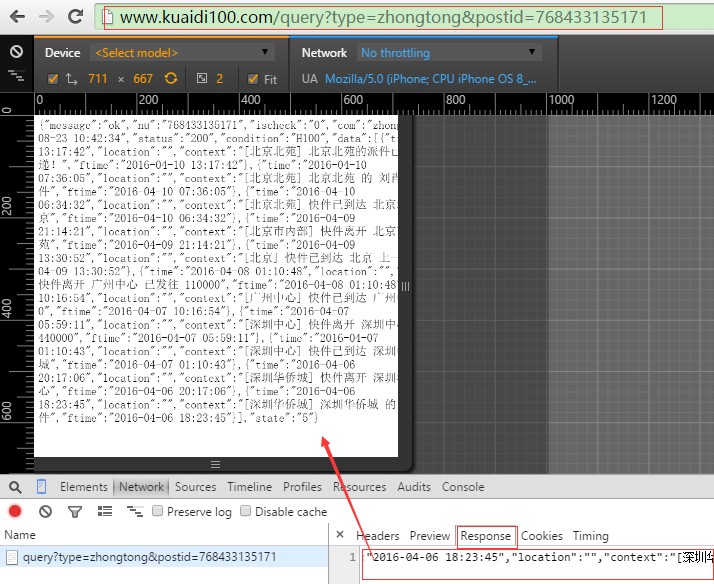
JSON 具有自我描述性，语法简洁，易于理解。

JSON数据主要有两种数据结构，一种是键/值，另一种是数组的形式来表示。

可登陆 json 中文官网，获取json相关资料： <http://www.json.org.cn/>

#### Json的用途





#### Json语法

首先，**任何数据类型都可以通过json来表示**。如：字符串、数字、对象、map、结构体、切片……

先来看一个简单的 JSON

{

"stars": [

{ "name":"Faye" ,"address":" 北京 " },

{ "name":"andy" ,"address":" 香港 " },

{ "name":"eddie","address":" 台湾 " },

]

}

JSON语法是JavaScript对象表示法语法的子集。

**数据在键 / 值对中。数据由逗号分隔。花括号保存对象**，也称一个文档对象。

**方括号保存数组，每个数组成员用逗号隔开，并且每个数组成员可以是文档对象或者数组或者键值对。**

JSON基于两种结构：“名称/值”对的集合（A collection of name/value pairs）。

不同的编程语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hashtable），有键列表（keyed list），或者关联数组 （associative array）。值的有序列表（An ordered list of values）。在大部分语言中，它被实现为数组（array），矢量（vector），列表（list），序列（sequence）。

上述描述可归纳为JSON三种语法：

1. 键/值对。 key **:** value，冒号分割。

如："name":"Faye"

key 必须是字符串类型，键/值对儿的“值”可以是：

数字（整数或浮点数）

字符串（在双引号中）

逻辑值（true 或 false）

null (空值 未设置)

对象（在花括号中）

数组（在方括号中）

key 是字符串类型，标准写法需加双引号，以免有可能报错。

1. Json对象。写在 { } 中，可包含多个key、value对。用“，”分隔。

如： { "name"**:**"Faye" ,"address"**:**"北京" }

1. Json数组。在 [ ] 中书写, 数组成员可以是对象、值，也可以是数组。

如： { "stars":[ {"name":"Faye" ,"address":"北京" },

{ "name":"andy" ,"address":"香港" },

{ "name":"eddie","address":"台湾"}, ]

}

#### json格式化

可以使用在线格式化工具，验证json数据的正确性。

1. [www.json.cn](http://www.json.cn)
2. <http://www.json.org.cn/tools/index.htm>

{

"name":"矮大紧",

"sex":"man",

"age":131,

"girls":["金莲","凤姐","码蓉","春哥"],

"成绩": [2, 14, 9, 78, 96],

"家电":{"彩电":"海尔", "洗衣机":"三星"}

}

### 序列化

在前面讲解json用途时，我们有介绍到，数据在网络传输前后要进行序列化和反序列化。目的是将复杂的数据类型按照统一、简单且高效的形式转储，以达到网络传输的目的。

在go语言中，较为复杂的数据存储形式无外乎结构体、map、切片。因此这里我们以这三种数据类型序列化为例进行讲解。其他简单数据类型请同学们融会贯通，自主学习。

所谓“序列化”(Serialize)，简单理解就是将 结构体、map、切片 这三种类型的数据相应的转换为json字符串的过程。

#### 结构体序列化

首先，创建结构体类型。由于之后要对该结构体进行序列化，而序列化函数是在 encoding/json包中，因此内部字段应指定为首字母大写，以便导出作用域。

**type** Student **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}

序列化函数：

import "encoding/json"

**func Marshal(v interface{}) ([]**[**byte**](https://studygolang.com/static/pkgdoc/pkg/builtin.htm#byte)**,** [**error**](https://studygolang.com/static/pkgdoc/pkg/builtin.htm#error)**)**

参数：空接口类型。意味着任何数据类型（int、float、结构体…）都可以使用该函数进行序列化。

返回值：字符切片，err

示例代码：

**import** (  
 **"encoding/json"  
 "fmt"**)  
  
*// 定义结构体***type** Student **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}  
*// 封装结构体序列化测试函数***func** structSerial() {  
 *// 定义结构体变量* stu := Student{  
 Name:**"瞎么海"**,  
 Id:110,  
 Age:33,  
 Addr:**"天津市海河"**,  
 }  
 *// 使用 encoding/json 包中的 Marshal() 函数将结构体变量进行序列化* data, err := json.Marshal(&stu)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"stu序列化后 = "**, string(data))  
}  
  
**func** main() {

// 测试结构体序列化

structSerial()  
}

输出结果为：

stu序列化后 = {"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}

Go 语言标准库的"encoding/json"包还提供了另外一个方法：MarshalIndent。该方法的作用与Marshall作用相同，只是可以通过方法参数，设置前缀、缩进等，对Json 多了一些格式处理，打印出来比较好看。如：

json.MarshalIndent(stu, "\t", "")

直接使用fmt.Println打印，能换行显示 stu的成员并带有 \t 缩进对齐效果。

#### Map序列化

Map序列化的大致流程与结构体一致。首先也要先创建map变量。如：

var m map[string]interface{}

key值为string，value 可以是任意数据。但此m不能直接存储数据！必须使用make开辟空间再使用。

m = make(map[string]interface{})  
// 赋值  
m["name"] = "大晕头"  
m["sal"] = 3141.59  
m["age"] = 27  
m["addr"] = "北京朝阳"

同样，使用序列化函数json.Marshal() 可以将m完成序列化。由于m是引用可以直接传递。

示例代码：

*// 封装 map 序列化测试函数***func** mapSerial() {  
 *// 定义map变量* **var** m **map**[string]**interface**{}  
 *// 初始化map,获取空间* m = make(**map**[string]**interface**{})  
 *// 赋值* m[**"name"**] = **"大晕头"** m[**"sal"**] = 3141.59  
 m[**"age"**] = 27  
 m[**"addr"**] = [2]string{**"北京朝阳", "天津南开"**}*// 将 map 使用 Marshal() 函数进行序列化* data, err := json.Marshal(m) *// map本身为引用。* **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"map序列化后 = "**, string(data))  
}  
  
**func** main() {  
 *// 测试map序列化* mapSerial()  
}

输入结果为：

map序列化后 = {"addr":["北京朝阳","天津南开"],"age":27,"name":"大晕头","sal":3141.59}

由于map是无序存储，因此查询序列化后的结果，可以与定义的顺序不一致。

#### Slice序列化

首先，创建上面map类型的切片mySlice。

**var** mySlice []**map**[string]**interface**{}

依次创建map，并初始化、赋值，追加到切片中

*// 创建map1，初始化，赋值*m1 := make(**map**[string]**interface**{})  
m1[**"name"**] = **"李白"**m1[**"age"**] = 20  
m1[**"addr"**] = **"莫高窟"**mySlice = append(mySlice, m1)  
  
*// 创建map2，初始化，赋值*m2 := make(**map**[string]**interface**{})  
m2[**"name"**] = **"杜子腾"**m2[**"age"**] = 29  
m2[**"addr"**] = **[3]string{"毛纺厂","方便村","洗手街",}**mySlice = append(mySlice, m2)

由于map是**map**[string]**interface**{}类型，因此value可以是任意数据类型，数组也可以。

同样，使用相同的方法进行序列化并查看。

示例代码：

*// 封装 slice 序列化测试函数***func** sliceSerial() {  
 *// 创建一个 map[string]interface{}类型的 切片*

……  
 *// 创建map，初始化，赋值*

……  
 *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(mySlice)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"slice序列化后 = "**, string(data))  
}  
**func** main() {  
 *// 测试slice序列化* sliceSerial()  
}

输入结果为：slice序列化后 =

[{"addr":"莫高窟","age":20,"name":"李白"},{"addr":["毛纺厂","方便村","洗手街"],"age":29,"name":"杜子腾"}]

可以将这个结果，导入到json在线格式化验证网站中查验。数据正确！

#### 基础类型序列化

前面讲json的时候有提到，**任何数据类型都可以通过json来表示**。可以递推理解为：**任何数据类型都可以进行序列化**。

因此，普通的int数组、切片、普通的 string 变量等基础数据类型也都可以完成序列化。

示例代码：

*// int 类型切片序列化测试函数***func** sliceSerial2() {  
 **var** s = []int{1,2,3,4,5,6}  
 *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(s)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"普通slice序列化后 = "**, string(data))  
}  
  
*// string 类型变量序列化函数***func** varSerial() {  
 **var** str string = **"hoho haha"** *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(str)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"var序列化后 = "**, string(data))  
}  
  
**func** main() {  
 *// 测试int类型切片序列化* sliceSerial2()  
 *// 测试普通变量序列化* varSerial()  
}

可以看到，基础数据类型由于数据本身没有key-value这种存储形式，因此序列化后，得到的仅仅是字符串类型数据，实际使用时无太大意义，因此很少使用。

### 结构体标签

结构体的字段除了名字和类型外，还可以有一个可选的标签（tag）：它是一个附属于字段的字符串，可以是文档或其他的重要标记。比如在我们解析json或生成json文件时，常用到encoding/json包，它提供一些默认标签，例如：omitempty标签可以在序列化的时候忽略0值或者空值；而“-”标签的作用是不进行序列化，其效果和直接将结构体中的字段写成小写的效果一样。

type Info struct {

Name string

Age int `json:"age,omitempty"`

Sex string

}

在序列化和反序列化的时候，也支持类型转化等操作。如

type Info struct {

Name string

Age int `json:"age,string"`

//这样生成的json对象中，age就为字符串

Sex string

}

定义结构体时，可以通过这些默认标签来设定结构体成员变量，使之在序列化后得到特殊的输出。

示例代码：

**type** Student **struct** {  
 Name string **`json:"-"`** *//“-”作用是不进行序列化，效果和将结构体字段写成小写一样。* Age int **`json:"age,omitempty"`** *// “omitempty”作用是在序列化的时候忽略0值或空值。* Id int **`json:"idx,string"`** *// 序列化时，类型转化为string* Sex string **`json:"sex"`**}  
**func** main() {  
 stu := Student{Name:**"Andy"**, Age:0, Id:100}ret, err:= json.Marshal(test)  
 **if** err != nil {  
 panic(err)  
 }  
 fmt.Println(**"json info1 ret="**, string(ret))

}

#### 序列化应用tag

通过前面序列化章节编写的测试代码，我们可以看到。生成的序列化json字符串是将struct 结构体的字段名作为json字符串的key，将字段值作为value的。

这在实际应用过程中容易出现问题。为了方便导出包作用域，我们在定义结构体时，要将字段名的首字母大写，而实际应用中，当我们用json和外部API进行交互的时候, 其他语言并不像Go这样用大写和小写来控制变量的作用域。所以借助结构体标签,可以得到小写字母开头的字串。

很自然的，可以想到，将struct 结构体中的字段名改为小写字母不就可以解决这个冲突吗？但实际测试发现，如果转为小写，则丢失了包作用域，Marshal() 函数不能正常完成序列化，得到的往往是空值。后续再来测试。

这里，就需要用到我们上节提到的结构体标签tag来解决这一问题了。从新定义结构体类型。

**注意语法：stu\_name一定要使用 " " 包裹起来。**

**type** Student **struct** {  
 Name string `json: "stu\_name"`  
 Id int `json: "stu\_id"`  
 Age int // 剩余两个字段不添加tag，用以比较  
 Addr string  
}

序列化方法与之前的序列化方法一致。使用json.Marshal() 可以直接得到json字符串，输出结果为：

stu序列化后 = {"stu\_name":"瞎么海","stu\_id":110,"Age":33,"Addr":"天津市海河"}

可以看到，此时json字符串的key值为“stu\_name”和“stu\_id”，是我们定义结构体时指定的tag。而没有指定tag的Age和Addr，key值依然是原来字段的名字。

如果改为小写会怎样呢？测试：

**type** Student **struct** {  
 Name string `json: "stu\_name"`  
 Id int `json: "stu\_id"`  
 age int // 将两个不添加tag的字段名改为小写  
 addr string  
}

当然，结构体变量赋值时也要指定小写的字段名进行赋值。

输出结果为：

stu序列化后 = {"stu\_name":"瞎么海","stu\_id":110}

可以看到，程序没有报错。但指定为小写字段名的两个字段age、addr没有成功转换为json字串。

### 反序列化

既然序列化是将go语言的某种类型的数据转换为josn字符串，那么 “反序列化”(Deserialize)，就是将josn字符串转换回序列化之前数据的过程。简单理解就是序列化的逆操作。

这里按我们前面章节序列化后得到的json串依次进行反序列化。

#### 结构体反序列化

首先，依然需要创建结构体类型。要求该类型必须与之前序列化时指定的结构体类型完全一致（个数、类型）。

**type Teacher struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}

定义结构体变量

**var** tea Teacher

准备待反序列化的json字符串，由于字串中包含 **" "**，使用反引号包裹该字串。当然，实际开发过程中，这个数据大多情况下是借助网络传输获取的。

str := **`{"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}`**

序列化函数：

import "encoding/json"

**func Unmarshal(data []**[**byte**](https://studygolang.com/static/pkgdoc/pkg/builtin.htm#byte)**, v interface{})** [**error**](https://studygolang.com/static/pkgdoc/pkg/builtin.htm#error)

参数1：待解析的 json编码字符串

参数2：解析后传出的结果

返回值：err

示例代码：

**import** (  
 **"encoding/json"  
 "fmt"**)  
*// 定义结构体***type** Teacher **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}

*// 封装函数测试结构体反序列化***func** structDeserial() {  
 *// 准备待反序列化的json字符串* str := **`{"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}`** *// 定义Teacher类型变量* **var** tea Teacher  
 *// 使用 encoding/json 包中的 Unmarshal() 函数，反序列化* err := json.Unmarshal([]byte(str), &tea)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }  
 *// 查看反序列化结果* fmt.Println(**"反序列化后：tea="**, tea, **"tea.Name="**, tea.Name)  
}  
**func** main() {  
 *// 测试结构体反序列化* structDeserial()  
}

输入结果为：

反序列化后：tea= {瞎么海 110 33 天津市海河} tea.Name= 瞎么海

#### Map反序列化

准备待反序列化的json字符串。

str := **`{"addr":"北京朝阳","age":27,"name":"大晕头","sal":3141.59}`**

定义map变量，类型必须与之前序列化的类型完全一致。

**var** m **map**[string]**interface**{}

不需要使用make函数给m初始化，开辟空间。这是因为，在反序列化函数 Unmarshal() 中会判断传入的参数2，如果是map类型数据，会自动开辟空间。相当于是Unmarshal() 函数可以帮助我们做make操作。

但！传参时需要注意，**Unmarshal**的第二个参数，是用作传出，返回结果的。因此必须传m的地址值。

json.Unmarshal([]byte(str), &m)

示例代码：

*// 封装函数测试map反序列化***func** mapDeserial() {  
 *// 准备反序列化的json字符串* str := **`{"addr":"北京朝阳","age":27,"name":"大晕头","sal":3141.59}`** *// 定义map变量* **var** m **map**[string]**interface**{}  
 err := json.Unmarshal([]byte(str), &m)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }  
 *// 查看反序列化结果* fmt.Println(**"反序列化后：m ="**, m)  
}  
**func** main() {  
 *// 测试map反序列化* mapDeserial()  
}

输出结果为：

反序列化后：m = map[addr:北京朝阳 age:27 name:大晕头 sal:3141.59]

#### Slice反序列化

实现思路与前面两种的实现完全一致，这里不再赘述。直接参考代码：

*// 封装函数测试slice反序列化***func** sliceDeserial() {

*// 准备反序列化的json字符串* str := **`[{"addr":"莫高窟","age":20,"name":"李白"},{"addr":["毛纺厂","方便村","洗手街"],"age":29,"name":"杜子腾"}]`**

*// 定义slice变量* **var** slice []**map**[string]**interface**{}  
 err := json.Unmarshal([]byte(str), &slice)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }

*// 查看反序列化结果* fmt.Println(**"反序列化后：slice ="**, slice)  
}

**func** main() {  
 *// 测试slice反序列化* sliceDeserial()  
}

输出结果为：

slice = [map[addr:莫高窟 age:20 name:李白] map[addr:[毛纺厂 方便村 洗手街] age:29 name:杜子腾]]

**特别强调：**

反序列化json字符串时，务必确保反序列化传出的数据类型，与之前序列化的数据类型完全一致。

Gob序列化

## 网络聊天室

并发编程和网络编程是现今行业开发中常用的技术。Go语言强大的语法设定使得并发和网络编程都变的简洁而高效。

下面我们利用前面学到的知识，使用并发和网络实现一个简单的网络在线聊天室。体会下这两种技术的实际应用。在整个聊天室的项目中，充分利用了go程并发，处理不同任务。

整个聊天室程序可简单划分为如下模块，都分别使用go程来实现：

**主go程（服务器）：**

负责监听、接收用户（客户端）连接请求，建立通信关系。同时启动相应的go程处理任务。

**处理连接用户数据go程：HandleConnect**

负责新上线用户的存储，用户消息读取、发送，用户改名、下线处理及超时处理。

为了提高并发效率，同时给一个用户维护多个go程来并行处理上述任务。

**用户消息广播go程：Manager**

负责在线用户遍历，用户消息广播发送。需要与**HandleConnect** go程及用户子go程协作完成。

**go程间应用数据及通信：**

map：存储所有登录聊天室的用户信息， key：用户的ip+port。Value：Client结构体。

Client结构体：包含成员：用户名Name，网络地址Addr（ip+port），发送消息的通道C（channel）

通道message：协调并发go程间消息的传递。

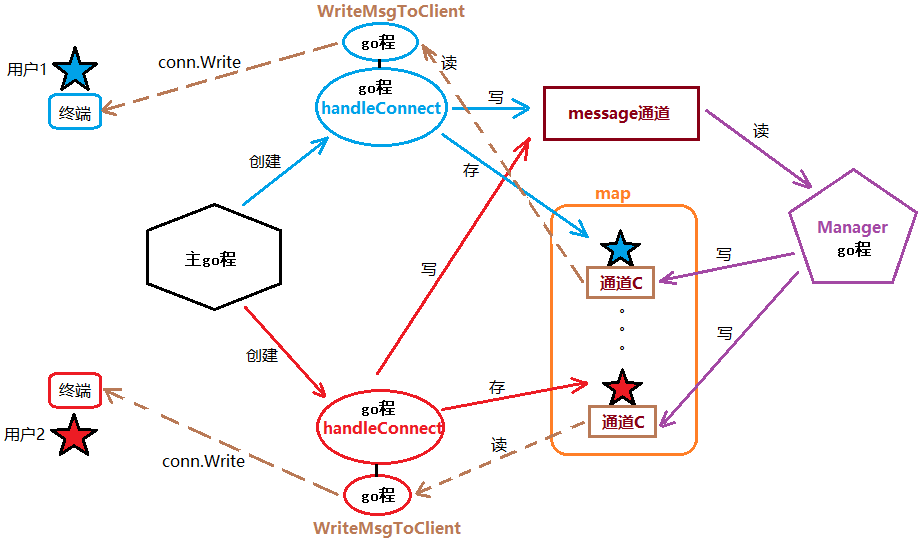
### 广播用户上线

首先，服务器启动，等待用户建立通信连接。当有用户连接上来，将其存储到map中，这样就维护了一个“在线用户”的列表。当再有新用户连接上来时，应向该列表中所有用户进行广播通知，提示xxx用户上线。

当然，简单实现手法可以循环读取列表中的用户，依次向其发送消息通知新用户上线。但这种方式无疑是一种串行的通信手段，实现简单，但执行效率较低。

在go语言中，我们利用go程轻便、高效、并发性好的特性，给每个登录用户维护多个go程来进行数据通信，借助channel不需要使用同步锁，就可以实现高效的并发通信。

下图充分利用goroutine和channel实现了新用户登录，向所有在线用户进行广播通知：



分析上图，主要分为几大模块。

全局位置定义用户结构体类型 Client，存储登录用户信息。成员包含channel、Name、Addr

type Client struct {

C chan string

Name string

Addr string

}

定义全局通道message处理消息。

定义全局map 存储在线用户信息。Key为用户网络地址。Value为用户结构体。

主go程，监听客户端连接请求，当有新的客户端连接，创建新go程handleConnet处理用户连接。

handleConnet go程，获取用户网络地址（Ip+port），创建新用户结构体，包含成员C、Name、Addr。新用户的Name和Addr初值都是用户网络地址（Ip+port）。将用户结构体存入map中。并创建WriteMsgToClient go程，专门负责给当前用户发送消息。组织新用户上线广播消息内容，写入全局通道message中。

WriteMsgToClient go程，读取用户结构体C中的数据，没有则阻塞等待，有数据写出给登录用户。

Manager go程，给map分配空间。循环读取 message 通道中是否有数据。没有，阻塞等待。有则解除阻塞，将message通道中读到的数据写到用户结构体中的C通道。

代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
*// 定义用户结构体类型***type** Client **struct** {  
 C **chan** string  
 Name string  
 Addr string  
}  
*// 定义全局 map 存储在线用户 key:IP+port, value:Client***var** onlineMap **map**[string]Client  
  
*// 定义全局 channel 处理消息***var** message = make(**chan** string)  
  
**func** WriteMsgToClient(clnt Client, conn net.Conn) {  
 *// 循环跟踪 clnt.C，有消息则读走，Write 给客户端* **for** msg := **range** clnt.C {  
 conn.Write([]byte(msg + **"\n"**)) *// 发送消息 给客户端* }  
}  
  
**func** MakeMsg(clnt Client, msg string) (buf string) {  
 buf = **"["** + clnt.Addr + **"]"** + clnt.Name + **": "** + msg  
 **return**}  
  
**func** HandleConnect(conn net.Conn) {  
 **defer** conn.Close()  
 *// 获取新连接上来的用户的网络地址(IP+port)* netAddr := conn.RemoteAddr().String()  
 *// 给新用户创建结构体。用户名、网络地址一样* clnt := Client{make(**chan** string), netAddr, netAddr}  
 *// 将新创建的结构体，添加到 map 中，key值为获取到的网络地址（IP+port）* onlineMap[netAddr] = clnt  
  
 *// 新创建一个go程，专门给当前客户端发送消息。* **go** WriteMsgToClient(clnt, conn)  
  
 *// 广播新用户上线  
 // message <- "[" + clnt.Addr + "]" + clnt.Name + ": login"* message <- MakeMsg(clnt, **"login"**)  
  
 **for** { *// 防止当前go程结束。* runtime.GC()  
 }  
}  
  
**func** Manager() {  
 *// 给map分配空间* onlineMap = make(**map**[string]Client)  
  
 *// 循环读取 message 通道中的数据* **for** {  
 *// 通道 message 中有数据读到 msg 中。 没有，则阻塞* msg := <-message  
  
 *// 一旦执行到这里，说明message中有数据了，解除阻塞。 遍历 map* **for** \_, clnt := **range** onlineMap {  
 clnt.C <- msg *// 把从Message通道中读到的数据，写到 client 的 C 通道中。* }  
 }  
}  
  
**func** main() {  
 *// 创建监听 socket* listener, err := net.Listen(**"tcp"**, **"127.0.0.1: 8000"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listener.Close()  
  
 *// 创建go程 处理消息* **go** Manager()  
  
 *// 循环接收客户端连接请求* **for** {  
 conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **continue** *// 失败，监听其他客户端连接* }  
  
 *// 给新连接的客户端，单独创建一个go程，处理客户端连接请求* **go** HandleConnect(conn)  
 }  
}

### 广播用户消息

当某个客户端向服务端发送消息后，服务端应将该消息广播给其它的客户端，达到聊天室的群聊效果。

开启一个新的go程，为方便传参，可以选择匿名go程。专门负责接收从客户端传递过来的数据，然后将接收到的数据写到messaage通道中。

在实现“广播用户上线”时，我已经完成：Manager go程会阻塞读message通道，一旦有数据，则遍历map中的在线用户。将数据写到结构体成员的C通道中。WriteMsgToClient go程会迭代C这个channel，最终将数据发送给客户端。

综上，实际上我们想完成“广播用户消息”给所有在线用户的功能，只需要将读到的数据写到message通道即可达到目的。

相关代码：

**func** HandleConnect(conn net.Conn) {

……

……  
 *// 广播新用户上线* message <- MakeMsg(clnt, **"login"**)  
  
 *// 创建一个新go程，循环读取用户发送的消息，广播给在线用户* **go func**() {buf := make([]byte, 2048)  *// 定义切片缓冲区，存储读到的用户消息*  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 { *// 用户退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 msg := string(buf[:n]) *// 保存用户写来的消息内容* message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户* }  
 }()  
  
 **for** { *// 不能让当前go程结束。* ;  
 }  
}

### 展示在线用户

因为nc工具默认会添加‘\n’， 所以conn.Read()读取用户消息后，修改保存用户消息内容实现语句：

msg := string(buf[:n-1]) 重新读取用户消息。

读到后，对消息内容进行判断：如果用户发送了“who”，则当成一个查询指令处理。遍历map中所有在线用户，取出每个用户的相关描述信息，组成提示消息，写给当前用户即可。

由于这里客户端我们使用nc工具模拟，该工具对中文支持较差，所以我们组织的用户描述信息中不要包含中文字符。

代码片段如下：

msg := string(buf[:n-1]) *// 保存用户写来的消息内容, nc 工具默认添加‘\n’***if** msg == **"who"** && len(msg) == 3 { *// 判断用户发送了 who 指令* conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap { *// 遍历map获取在线用户* userInfo := user.Addr + **":"** + user.Name + **"\n"** *// 组织在线用户信息* conn.Write([]byte(userInfo)) *// 写给当前用户* }  
} **else** {  
 message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户*}

### 修改用户名

前面我们查看用户信息时，用户名都是与用户网络地址相同的内容。主要由于用户登录时，创建该用户名不是用户自己完成的，无法洞悉用户的意图。当用户成功登录上来可以通过给服务器发送消息，来修改自己的用户名。

设定，如果用户发送“rename | Iron man”指令，既是想修改自己的用户名为“Iron man”。判断用户消息，是否包含“rename|”关键字：if len(msg) >= 8 && msg[:6] == "rename" {。如果是，那么拆分用户意欲修改的用户名保存。**strings.Split()**函数可以完成拆分字符串操作。

将该用户名替换当前用户的Name。使用用户的Addr作为key，找到map中当前用户，覆盖即可达到改名的目的。操作结束提示用户改名成功。

代码片段如下：

msg := string(buf[:n-1])  
**if** msg == **"who"** && len(msg) == 3 {  
 conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap {  
 userInfo := user.Addr + **":"** + user.Name + **"\n"** conn.Write([]byte(userInfo))  
 }  
 *// 判断用户输入的前6个字符是否为 rename*} **else if** len(msg) >= 8 && msg[:6] == **"rename"** { *// rename | Iron man* newName := strings.Split(msg, **"|"**)[1] *// 按"|"拆分，rename为[0], Iron man为[1]* clnt.Name = newName *// 替换掉当前用户原始Name* onlineMap[netAddr] = clnt *// 使用netAddr为key找到map中当前用户。覆盖* conn.Write([]byte(**"rename successful\n"**))  
} **else** {message <- MakeMsg(clnt, msg)  
}

### 用户退出

前面在“广播用户消息”时，当conn.Read() 读到0时，我们在服务器端，简单打印了“用户xxx退出登录”的提示。

但实际上，在聊天室中，有在线用户离开，我们应该将这一事件广播给所有用户知晓，并且将该用户从map在线用列表中移除。需要实时的监看在线用户的状态。可以创建channel来检测用户退出状态，并使用select来监听channel上的数据流动。

当channel上有数据时，select对应阻塞case语句得以执行。将用户从map中移除。同时通知所有在线用户。

代码片段：

**func** HandleConnect(conn net.Conn) {

……

message <- MakeMsg(clnt, **"login"**)  
**isQuit := make(chan bool)** *// 检测用户主动退出*  
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 {  
 **isQuit <- *true*** *// 用户主动退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 ……

}  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit: *// 用户不主动退出，阻塞*

close(clnt.C)   
delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"logout"**) *// 广播给在线用户，谁退出了* **return** *// 结束当前退出用户对应go程* }  
 }  
}

### 超时处理

如果客户端没有主动退出，并且长时间没有发送消息，会一直占用服务端的资源。服务器通常针对这种用户添加“超时强踢”机制，强制将该客户端与服务器连接断开。

可以借助并发编程时我们所学的select超时机制来实现。Select监听time.After(60 \* time.Second) 通道上的数据流动。如果在计时期间一直没有数据，通道中会被写入当前系统时间，select 的case满足读条件，不再阻塞。但，有一个问题，用户如果持续在输入数据，这个计时器依然在计时，时间到，依然会强制踢出用户。

因此，我们另外创建一个通道hasData来检测用户是否有数据发送，让Select也来监听这个channel。这样，当用户有数据输入时，select监听的这个hasData通道会满足case条件得以执行，但我们不做任何处理。目的是使得监听在select中的计时器被重新计时。

只有当真正持续60s没有数据发送时，select 中用于计时的case才满足条件，将用户与服务器连接断开。

代码片段：

**func** HandleConnect(conn net.Conn) {  
 ……

……  
isQuit := make(**chan** bool)  
 **hasData := make(chan bool)** *// 检测用户是否有消息发送*  
   
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 ……  
 msg := string(buf[:n-1]) if {

……

} else if {

……

} else {

……

}  
 hasData <- ***true*** *// 只要执行到这里，就说明用户有数据发送* }  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit:

close(clnt.C)delete(onlineMap, netAddr) message <- MakeMsg(clnt, **"logout"**) **return** **case** <-hasData:  
 *// 什么都不做，目的是让计时器归零* **case** <-time.After(60\*time.***Second***):

close(clnt.C)  
 delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"time out leave"**) *// 广播给在线用户，超时退出* **return** *// 结束当前退出用户对应go程* }  
 }  
}

这里需要注意的是，每循环一次，第三个case后面的时间都会重新计算。（例如:执行完case<-hasData后，紧跟着执行第三个case，发现时间是10秒，不到60秒，条件不成立，不会执行该case后面的代码，进入下次循环，这时时间重新计算）

当hasData没有数据，isQuit没有数据，60s时间没有到，这时三个case都阻塞等待。直到60秒后，前两个case条件依然不成立，第三个case满足，执行后面代码，断开客户端连接，踢下线。