# Go快速入门-下篇

## 一、错误处理

### 1.1 Error处理

- 大部分的内置包或者外部包，都有自己的报错处理机制。因此我们使用的任何函数可能报错，这些报错都不应该被忽略，而是在调用函数的地方，优雅地处理报错。

```
package main

import (
  "fmt"
    "net/http"
)

func main(){
    resp, err := http.Get("http://example.com/")
    if err ≠ nil {
        fmt.Println(err)  //打印报错
        return
    }
    fmt.Println(resp)
}
```

### 1.2 Panic异常

- 当程序在运行过程中，突然遇到了未处理的报错，就会导致panic。在Go中，更推荐使用error对象，而不是panic来处理异常。发生panic后，程序会停止运行，但会运行defer语句代码，Defer适用于 需要在函数最后执行某些操作的场景，比如关闭文件。

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r ≠ nil {
            fmt.Println("Recovered in f", r)
        }
    }()
}
```

```
fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}
```

# 二、内置运算符

## 1.1 算数运算符

| 运算符 | 描述 |
| --- | --- |
| + | 相加 |
| - | 相减 |
| * | 相乘 |
| / | 相除 |
| % | 求余=被除数-（被除数/除数）*除数 |

### 1. 算数运算符使用

```
package main
import (
  "fmt"
)
func main() {
```

```
    fmt.Println("10+3=", 10+3)  //10+3= 13
    fmt.Println("10-3=", 10-3)  //10-3= 7
    fmt.Println("10*3=", 10*3)  //10*3= 30
    //除法注意：如果运算的数都是整数，那么除后，去掉小数部分，保留整数部分
    fmt.Println("10/3=", 10/3)    //10/3= 3
    fmt.Println("10.0/3=", 10.0/3)  //3.3333333333333335
    // 取余注意 余数=被除数-（被除数/除数）*除数
    fmt.Println("10%3=", 10%3)   //10%3= 1
    fmt.Println("-10%3=", -10%3)   // -10%3= -1
    fmt.Println("10%-3=", 10%-3)   // 10%-3= 1
    fmt.Println("-10%-3=", -10%-3)  // -10%-3= -1
}
```

### 2. i++

```
package main
import (
  "fmt"
)
func main() {
    var i int = 1
    i++
    fmt.Println("i=", i)  // i= 2
}
```

## 1.2 关系运算符

| 运算符 | 描述 |
| --- | --- |
| == | 检查两个值是否相等，如果相等返回True，否则返回False |
| ! = | 检查两个值是否不相等，如果不相等返回True，否则返回False |
| > | 检查左边值是否大于右边值，如果是返回True，否则返回Flase |
| ⩾ | 检查左边值是否大于等于右边值，如果是返回True，否则返回False |
| < | 检查左边值是否小于右边值，如果是返回True，否则返回False |
| ⩽ | 检查左边值是否小于等于右边值，如果是返回True，否则返回False |

```
package main
import (
  "fmt"
)
func main() {
    var n1 int = 9
    var n2 int = 8
    fmt.Println(n1 == n2)  //false
    fmt.Println(n1 ≠ n2)  //true
```

```
    fmt.Println(n1 > n2)   //true
    fmt.Println(n1 ⩾ n2)  //true
    fmt.Println(n1 < n2)   //flase
    fmt.Println(n1 ⩽ n2)  //flase
    flag := n1 > n2
    fmt.Println("flag=", flag)  //flag= true
}
```

## 1.3 逻辑运算符

| 运算符 | 描述 |
| --- | --- |
| && | 逻辑AND运算符。如果两边的操作数都是True，则为True，否则为False |
| \|\| | 逻辑OR运算符。如果两边的操作数有一个True，则为True，否则False |
| ! | 逻辑NOT运算符。如果条件为True，则为False，否则为True |

```
package main
import (
  "fmt"
)
func main() {
    //演示逻辑运算符的使用 &&
    var age int = 40
    if age > 30 && age < 50 {
      fmt.Println("ok1")
    }
    if age > 30 && age < 40 {
      fmt.Println("ok2")
    }
    //演示逻辑运算符的使用 ||
    if age > 30 || age < 50 {
      fmt.Println("ok3")
    }
    if age > 30 || age < 40 {
      fmt.Println("ok4")
    }
    //演示逻辑运算符的使用
    if age > 30 {
      fmt.Println("ok5")
    }
    if !(age > 30) {
      fmt.Println("ok6")
    }
}
```

### 1.4 赋值运算符

| 运算符 | 描述 |
| --- | --- |
| = | 简单的赋值运算符，将一个表达式的值赋给左边的变量 |
| += | 相加后再赋值 |
| -= | 相减后再赋值 |
| *= | 相乘后再赋值 |
| /= | 相除后再赋值 |
| %= | 求余后再赋值 |

```
package main
import (
  "fmt"
)
func main() {
    d := 8 + 2*8    //赋值运算从右向左
    fmt.Println(d)  //24
    x := 10
    x += 5      //x=x+5
    fmt.Println("x += 5 的值:", x)  //24
}
```

# 三、条件循环

## 2.1 if else(分支结构)

#### 1. if条件判断基本写法

```
package main
import (
  "fmt"
)
func main() {
    score := 65
    if score ≥ 90 {
      fmt.Println("A")
    } else if score > 75 {
      fmt.Println("B")
    } else {
      fmt.Println("C")
    }
}
```

### 2. if条件判断特殊写法

- if 条件判断还有一种特殊的写法，可以在 if 表达式之前添加一个执行语句，再根据变量值进行判断

```
package main
import "fmt"
func main() {
    //这里的 score 是局部作用域
    if score := 56; score ⩾ 90 {
      fmt.Println("A")
    } else if score > 75 {
      fmt.Println("B")
    }else {
      fmt.Println("C")
      fmt.Println(score) // 只能在函数内部打印 score
    }
    // fmt.Println(score) //undefined: score
}
```

## 2.2 for循环结构

### 1. for循环

- 普通for循环

```
package main
import "fmt"
func main() {
    // 打印:  0 ~ 9 的数字
    for i := 0; i < 10; i++ {
      fmt.Println(i)
    }
}
```

- 外部定义i

```
package main
import "fmt"
func main() {
    i := 0
    for i < 10 {
      fmt.Println(i)
      i++
    }
}
```

### 2. 模拟while循环

- Go 语言中是没有 while 语句的，我们可以通过 for 代替

```
package main
import "fmt"
func main() {
    k := 1
    for { // 这里也等价 for ; ; {
      if k <= 10 {
            fmt.Println("ok~~", k)
        } else {
            break  //break 就是跳出这个 for 循环
        }
        k++
    }
}
```

### 3. for range键值循环

```
package main
import "fmt"
func main() {
    str := "abc上海"
    for index, val := range str {
      fmt.Printf("索引=%d, 值=%c \n", index, val)
    }
}
/*
索引=0, 值=a
索引=1, 值=b
索引=2, 值=c
索引=3, 值=上
索引=6, 值=海
*/
```

## 2.3 switch case

- 使用 switch 语句可方便地对大量的值进行条件判断

```
package main
import "fmt"
func main() {
    score := "B"
    switch score {
        case "A":
          fmt.Println("非常棒")
        case "B":
          fmt.Println("优秀")
```

```
        case "C":
          fmt.Println("及格")
        default:
          fmt.Println("不及格")
    }
}
```

## 2.4 break、continue、return

- break跳出整个循环
- continue跳出本次循环
- return跳出整个函数，并返回响应的值

### 1. break

```
package main
import "fmt"
func main() {
    k := 1
    for { // 这里也等价 for ; ; {
        if k <= 10 {
          fmt.Println("ok~~", k)
        } else {
          break  //break 就是跳出这个 for 循环
        }
        k++
    }
}
```

### 2. continue

```
package main
func main() {
    for i := 0; i < 10; i++ {
        if i%2 == 0 {
          continue
        }
      println(i)
    }
} /*1 3 5 7 9*/
```

### 3. return

```go
package main
import "fmt"
func main() {
  for i := 0; i < 10; i++ {
    fmt.Println(i)
    if i == 5 {
      return
    }
  }
  fmt.Println("最后执行的打印")
}
```

# 四、函数

## 3.1 函数基础

### 1. 函数定义

- 函数语法：func关键字 函数名（接收参数）（返回值） {函数体}
- 函数是组织好的、可重复使用的、用于执行指定任务的代码块。

```go
package main
import "fmt"
func main() {
    ret := intSum(1, 2)
    fmt.Println(ret) // 3
}
func intSum(x, y int) int {
  return x + y
}
```

### 2. 函数返回值

- Go 语言中通过 return 关键字向外输出返回值。
- 函数多返回值，Go 语言中函数支持多返回值，函数如果有多个返回值时必须用()将所有返回值包裹起来

```go
package main
import "fmt"
func main() {
    plus, sub := calc(4, 5)
    fmt.Println(plus) // 和为: 9
    fmt.Println(sub)  // 差为: -1
}
func calc(x, y int) (int, int) {
    sum := x + y
    sub := x - y
    return sum, sub
}
```

## 3.2 匿名函数

- 匿名函数就是一个没有名字的函数，当我们在一个函数内部重复调用一段代码， 并且这段代码不想被别的函数使用，我们就可以定义一个匿名函数，或者在一段代码需要并发的时候使用匿名函数。

```go
package main
import (
    "fmt"
)
func main(){
    //将匿名函数fun 赋给变量test_fun
    //则test_fun的数据类型是函数类型，可以通过test_fun完成调用
    test_fun := func (n1 int, n2 int) int {
        return n1 - n2
    }

    res2 := test_fun(10, 30)
    res3 := test_fun(50, 30)
    fmt.Println("res2=", res2)
    fmt.Println("res3=", res3)
    fmt.Printf("%T", test_fun)
}
/*
res2= -20
res3= 20
func(int, int) int
 */
```

## 3.3 闭包

- 在函数外部访问函数内部变量成为可能
- 闭包函数的返回值是匿名函数
- 这种用法主要场景就是未免函数内部的环境（变量等）被外部污染，如gin中间件

```go
package main
import (
```

```
    "fmt"
)
func AddInt() func(i int) int{
    var num int
    return func(i int) int {
        num += i
        return num
    }
}

func main() {
    fn := AddInt()

    fmt.Println(fn(1))  // 1
    fmt.Println(fn(1))  // 2
    fmt.Println(fn(1))  // 3
    fmt.Println(fn(1))  // 4
}
```

- 函数内部变量离开其作用域后始终保持在内存中而不被销毁

```
$ go build -gcflags=-m main.go
# command-line-arguments
./main.go:7:9: can inline AddInt.func1
./main.go:16:13: inlining call to fmt.Println
./main.go:17:13: inlining call to fmt.Println
./main.go:18:13: inlining call to fmt.Println
./main.go:19:13: inlining call to fmt.Println
./main.go:6:6: moved to heap: num
./main.go:7:9: func literal escapes to heap
./main.go:16:16: fn(1) escapes to heap
./main.go:16:13: main []interface {} literal does not escape
./main.go:16:13: io.Writer(os.Stdout) escapes to heap
./main.go:17:16: fn(1) escapes to heap
./main.go:17:13: main []interface {} literal does not escape
./main.go:17:13: io.Writer(os.Stdout) escapes to heap
./main.go:18:16: fn(1) escapes to heap
./main.go:18:13: main []interface {} literal does not escape
./main.go:18:13: io.Writer(os.Stdout) escapes to heap
./main.go:19:16: fn(1) escapes to heap
./main.go:19:13: main []interface {} literal does not escape
./main.go:19:13: io.Writer(os.Stdout) escapes to heap
<autogenerated>:1: (*File).close .this does not escape
```

## 3.4 函数变量作用域

### 1. 全局变量

- 全局变量是定义在函数外部的变量，它在程序整个运行周期内都有效。
- 在函数中可以访问到全局变量

```
package main
import "fmt"

//定义全局变量 num
var num int64 = 10

func main() {
    fmt.Printf("num=%d\n", num) //num=10
}
```

### 2. 局部变量

- 局部变量是函数内部定义的变量， 函数内定义的变量无法在该函数外使用
- 例如下面的示例代码 main 函数中无法使用 test 函数中定义的变量 name

```
package main
import "fmt"

func main() {
    // 这里name是函数test的局部变量，在其他函数无法访问
    //fmt.Println(name)
    test()
}
func test() {
    name := "zhangsan"
    fmt.Println(name)
}
```

### 3. for 循环语句中定义的变量

- 我们之前讲过的 for 循环语句中定义的变量，也是只在 for 语句块中生效

```go
package main
import "fmt"

func main() {
  testLocalVar3()
}
func testLocalVar3() {
    for i := 0; i < 10; i++ {
        fmt.Println(i) //变量 i 只在当前 for 语句块中生效
    }
  // fmt.Println(i) //此处无法使用变量 i
}
```

# 五、结构体

## 1.1 什么是结构体

- Go语言中没有“类”的概念，也不支持“类”的继承等面向对象的概念。
- Go语言中通过结构体的内嵌再配合接口比面向对象具有更高的扩展性和灵活性。

## 1.2 结构体定义

### 1. 基本实例化（方法1）

- 只有当结构体实例化时，才会真正地分配内存，也就是必须实例化后才能使用结构体的字段。
- 结构体本身也是一种类型，我们可以像声明内置类型一样使用 var 关键字声明结构体类型。

```go
package main
import "fmt"

type person struct {
    name string
    city string
    age  int
}

func main() {
    var p1 person
    p1.name = "张三"
    p1.city = "北京"
    p1.age = 18
    fmt.Printf("p1=%v\n", p1)  // p1={张三 北京 18}
    fmt.Printf("p1=%#v\n", p1) // p1=main.person{name:"张三", city:"北京", age:18}
}
```

### 2. new实例化（方法2）

- 我们还可以通过使用 new 关键字对结构体进行实例化，得到的是结构体的地址
- 从打印的结果中我们可以看出 p2 是一个结构体指针。
- 注意：在 Golang 中支持对结构体指针直接使用.来访问结构体的成员。
- p2.name = "张三" 其实在底层是 (*p2).name = "张三"

```
package main
import "fmt"
type person struct {
    name string
    city string
    age  int
}
func main() {
    var p2 = new(person)
    p2.name = "张三"
    p2.age = 20
    p2.city = "北京"
    fmt.Printf("p2=%#v \n", p2) //p2=&main.person{name:"张三", city:"北京", age:20}
}
```

### 3. 键值对初始化（方法3）

```
package main
import "fmt"

type person struct {
    name string
    city string
    age int
}
func main() {
    p4 := &person{
        name: "zhangsan",
        city: "北京",
        age: 18,
    }
    // p4=&main.person{name:"zhangsan", city:"北京", age:18}
    fmt.Printf("p4=%#v\n", p4)
}
```

## 1.3 结构体方法和接收者

## 1. 结构体说明

- 在 go 语言中，没有类的概念但是可以给类型（结构体，自定义类型）定义方法。
- 所谓方法就是定义了接收者的函数。
  - Go语言中的方法（Method）是一种作用于特定类型变量的函数。
  - 这种特定类型变量叫做接收者（Receiver）。
  - 接收者的概念就类似于其他语言中的this或者 self。
- 方法的定义格式如下

```
func（接收者变量 接收者类型）方法名(参数列表)（返回值）{
  函数体
}
```

## 2. 结构体方法和接收者

- 给结构体 Person 定义一个方法打印 Person 的信息

```
package main
import "fmt"

type Person struct {
    name string
    age  int8
}

func (p Person) printInfo() {
  fmt.Printf("姓名:%v 年龄:%v", p.name, p.age) // 姓名:小王子 年龄:25
}

func main() {
    p1 := Person{
        name: "小王子",
        age:  25,
    }
  p1.printInfo() // 姓名:小王子 年龄:25
}
```

## 3. 值类型和指针类型接收者

- 实例1：给结构体 Person 定义一个方法打印 Person 的信息
- 值类型的接收者
  - 当方法作用于值类型接收者时，Go 语言会在代码运行时 将接收者的值复制一份 。
  - 在值类型接收者的方法中可以获取接收者的成员值，但 修改操作只是针对副本 ，无法修改接收者变量本身。
- 指针类型的接收者
  - 指针类型的接收者由一个结构体的指针组成
  - 由于指针的特性，调用方法时修改接收者指针的任意成员变量，在方法结束后，修改都是有效的。
  - 这种方式就十分接近于其他语言中面向对象中的 this 或者 self

```go
package main
import "fmt"

type Person struct {
    name string
    age  int
}
//值类型接受者
func (p Person) printInfo() {
  fmt.Printf("姓名:%v 年龄:%v\n", p.name, p.age) // 姓名:小王子 年龄:25
}
//指针类型接收者
func (p *Person) setInfo(name string, age int) {
    p.name = name
    p.age = age
}
func main() {
    p1 := Person{
        name: "小王子",
        age:  25,
    }
    p1.printInfo() // 姓名:小王子 年龄:25
    p1.setInfo("张三", 20)
    p1.printInfo() // 姓名:张三 年龄:20
}
```

## 1.4 encoding-json包

### 1. struct与json

- 比如我们 Golang 要给 App 或者小程序提供 Api 接口数据，这个时候就需要涉及到结构体和Json 之间的相互转换
- GolangJSON 序列化是指把结构体数据转化成 JSON 格式的字符串
- Golang JSON 的反序列化是指把 JSON 数据转化成 Golang 中的结构体对象
- Golang 中的序列化和反序列化主要通过 "encoding/json" 包中的 json.Marshal() 和 json.Unmarshal()方法实现

1)struct转Json字符串

```go
package main
import (
    "encoding/json"
    "fmt"
)

type Student struct {
    ID int
    Gender string
    name string   //私有属性不能被 json 包访问
    Sno string
}
```

```
func main() {
    var s1 = Student{
        ID: 1,
        Gender: "男",
        name: "李四",
        Sno: "s0001",
    }
    fmt.Printf("%#v\n", s1)  // main.Student{ID:1, Gender:"男", name:"李四",Sno:"s0001"}
    var s, _ = json.Marshal(s1)
    jsonStr := string(s)
    fmt.Println(jsonStr)  // {"ID":1,"Gender":"男","Sno":"s0001"}
}
```

2)Json字符串转struct

```
package main
import (
    "encoding/json"
    "fmt"
)
type Student struct {
    ID int
    Gender string
    Name string
    Sno string
}
func main() {
    var jsonStr = `{"ID":1,"Gender":"男","Name":"李四","Sno":"s0001"}`
    var student Student  //定义一个 Monster 实例
    err := json.Unmarshal([]byte(jsonStr), &student)
    if err != nil {
      fmt.Printf("unmarshal err=%v\n", err)
    }
    // 反序列化后 student=main.Student{ID:1, Gender:"男", Name:"李四", Sno:"s0001"}
    student.Name="张三"
    fmt.Printf("反序列化后 student=%#v student.Name=%v \n", student, student.Name)
}
```

## 2. struct tag

1)Tag标签说明

- Tag 是结构体的元信息，可以在运行的时候通过反射的机制读取出来。
- Tag 在结构体字段的后方定义，由一对反引号包裹起来
- 具体的格式如下:

```
key1:"value1" key2:"value2"
```

- 结构体 tag 由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。

- 同一个结构体字段可以设置多个键值对 tag，不同的键值对之间使用空格分隔。
- 注意事项:
  - 为结构体编写 Tag 时，必须严格遵守键值对的规则。
  - 结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，通过反射也无法正确取值。
  - 例如不要在 key 和 value 之间添加空格。

2)Tag结构体转化Json字符串

```
package main
import (
    "encoding/json"
    "fmt"
)

type Student struct {
    ID int `json:"id"`    //通过指定 tag 实现 json 序列化该字段时的 key
    Gender string `json:"gender"`
    Name string
    Sno string
}

func main() {
    var s1 = Student{
        ID: 1,
        Gender: "男",
        Name: "李四",
        Sno: "s0001",
    }
    // main.Student{ID:1, Gender:"男", Name:"李四", Sno:"s0001"}
    fmt.Printf("%#v\n", s1)
    var s, _ = json.Marshal(s1)
    jsonStr := string(s)
    fmt.Println(jsonStr)  // {"id":1,"gender":"男","Name":"李四","Sno":"s0001"}
}
```

3)Json字符串转成Tag结构体

```
package main
import (
    "encoding/json"
    "fmt"
)

type Student struct {
    ID int `json:"id"` //通过指定 tag 实现 json 序列化该字段时的 key
    Gender string `json:"gender"`
    Name string
    Sno string
}
```

```go
func main() {
    var s2 Student
    var str = `{"id":1,"gender":"男","Name":"李四","Sno":"s0001"}`
    err := json.Unmarshal([]byte(str), &s2)
    if err != nil {
      fmt.Println(err)
    }
    // main.Student{ID:1, Gender:"男", Name:"李四", Sno:"s0001"}
    fmt.Printf("%#v", s2)
}
```

# 六、面向对象

## 5.1 封装-工厂模式

- golang的结构体没有构造函数，通常可以使用工厂模式来解决这个问题。
- 如果包里面的结构体变量首字母小写，引入后，不能直接使用，可以工厂模式解决。
- 只关心结果，不关心实现过程，因为过程是由封闭的工厂来实现的。

```go
package demo

//定义一个结构体
type student struct{
    Name string
    score float64
}
//*student返回结构体的指针
func NewStudent(name string, score float64) *student {
    //外包引用不了，但本包是可以引用的
    stu := student{
        Name : name,
        score : score,
    }
    return &stu
}
//结构体中的score字段也是小写所以需要写一个方法返回
func (s *student) GetScore() float64 {
    return s.score
}

func (s *student) SetScore(score float64) {
    s.score = score
}
```

```go
package main
```

```
import (
    "fmt"
    "go-demo/demo"
)

func main() {
    s := demo.NewStudent("Tony", 60.5)
    fmt.Println(s)
    fmt.Println(s.GetScore())
    s.SetScore(80)
    fmt.Println(s)
    //&{Tony 60.5}
    //60.5
    //&{Tony 80}
}
```

## 5.2 继承-struct嵌套

- 在golang中，采用匿名结构体字段来模拟继承关系。
- 这个时候，可以说`Student`是继承自`Person`.

```
package main

import (
  "fmt"
)

type Person struct {
  name string
  age int
  sex string
}
func (Person) SayHello(){
  fmt.Println("this is from Person")
}

type Student struct {
  Person
  school string
}


func main() {
  stu := Student{school:"middle"}
  stu.name = "leo"
  stu.age = 30
  fmt.Println(stu.name)
```

```
  stu.SayHello()
}
```

## 5.3 多态-Golang接口的定义

### 1. Golang中的接口

- 在Go语言中接口（interface）是一种类型，一种抽象的类型。
- 接口（interface）定义了一个对象的行为规范， 只定义规范不实现 ，由具体的对象来实现规范的细节 。
- 实现接口的条件：
  - 一个对象只要全部实现了接口中的方法 ，那么就实现了这个接口。
  - 换句话说，接口就是一个需要实现的方法列表。

### 2. 为什么要使用接口

- 下面的代码中定义了猫和狗，然后它们都会叫，你会发现main函数中明显有重复的代码
- 如果我们后续再加上猪、青蛙等动物的话，我们的代码还会一直重复下去
- 那我们能不能把它们当成“能叫的动物”来处理呢？

```
package main
import (
    "fmt"
)

type Cat struct {
  Name string
}
func (c Cat) Say() string { return c.Name + ": 喵喵喵" }
type Dog struct {
  Name string
}
func (d Dog) Say() string { return d.Name + ": 汪汪汪" }
func main() {
    c := Cat{Name: "小白猫"} // 小白猫: 喵喵喵
    fmt.Println(c.Say())
    d := Dog{"阿黄"}
    fmt.Println(d.Say()) // 阿黄: 汪汪汪
}
/*
小白猫: 喵喵喵
阿黄: 汪汪汪
*/
```

### 3. 定义一个Usber接口

- 定义一个 Usber 接口让 Phone 和 Computer 结构体实现这个接口

```
package main

import "fmt"

//1.接口是一个规范
type Usber interface {
  getName() string
}
//2.如果接口里面有方法的话，必要要通过结构体或者通过自定义类型实现这个接口
type Phone struct {
  Name string
}

type Computer struct {
  Brand string
}
func (c *Computer) getName() string {
  return c.Brand
}
//3.手机要实现usb接口的话必须得实现usb接口中的所有方法
func (p *Phone) getName() string {
  return p.Name
}
func main() {
  p := &Phone{
    Name: "华为手机",
  }
  c := &Computer{
    Brand: "联想电脑",
  }
  var p1 Usber // golang中接口就是一个数据类型
  p1 = p // 表示手机实现Usb接口
  fmt.Println(p1.getName())

  //接口使用场景，处理相同类型的数据
  newName := transData(p)
  newName1 := transData(c)
  fmt.Println(newName, newName1)
}
func transData(usber Usber) string {
  name := usber.getName()
  return fmt.Sprintf("%s%s", name, "处理后")
}
```

## 5.4 空接口

### 1. 空接口说明

- Golang中空接口也可以直接当做类型来使用，可以表示任意类型（泛型概念）
- Golang 中的接口可以不定义任何方法，没有定义任何方法的接口就是空接口。
- 空接口表示没有任何约束，因此任何类型变量都可以实现空接口。
- 空接口在实际项目中用的是非常多的，用空接口可以表示任意数据类型

### 2. 空接口作为函数的参数

```
package main
import "fmt"

//空接口作为函数的参数
func show(a interface{}) {
  fmt.Printf("值:%v 类型:%T\n", a, a)
}
func main() {
    show(20)            // 值:20 类型:int
    show("你好golang")       // 值:你好golang 类型:string
    slice := []int{1, 2, 34, 4}
    show(slice)           // 值:[1 2 34 4] 类型:[]int
}
```

### 3. 切片实现空接口

```
package main
import "fmt"

func main() {
  var slice = []interface{}{"张三", 20, true, 32.2}
  fmt.Println(slice)  // [张三 20 true 32.2]
}
```

### 4. map 的值实现空接口

```go
package main
import "fmt"

func main() {
    // 空接口作为 map 值
    var studentInfo = make(map[string]interface{})
    studentInfo["name"] = "张三"
    studentInfo["age"] = 18
    studentInfo["married"] = false
    fmt.Println(studentInfo)
    // [age:18 married:false name:张三]
}
```

## 5.5 类型断言

- 一个接口的值（简称接口值）是由一个具体类型和具体类型的值两部分组成的。
- 这两部分分别称为接口的动态类型和动态值。
- 如果我们想要判断空接口中值的类型，那么这个时候就可以使用类型断言
- 其语法格式： x.(T)
    - x ： 表示类型为 interface{}的变量
    - T ： 表示断言 x 可能是的类型

```go
package main
import "fmt"

func main() {
    var x interface{}
    x = "Hello golnag"
    v, ok := x.(string)
    if ok {
      fmt.Println(v)
    }else {
      fmt.Println("非字符串类型")
    }
}
```

## 5.6 值接收者和指针接收者

### 1. 值接收者

- 当方法作用于值类型接收者时，Go语言会在代码运行时将接收者的值复制一份。
- 在值类型接收者的方法中可以获取接收者的成员值，但修改操作只是针对副本，无法修改接收者变量本身。

```go
package main
import "fmt"

type Usb interface {
  Start()
```

```
  Stop()
}
type Phone struct {
  Name string
}
func (p Phone) Start() {
  fmt.Println(p.Name, "开始工作")
}
func (p Phone) Stop() {
  fmt.Println("phone 停止")
}
func main() {
    phone1 := Phone{     // 一：实例化值类型
      Name: "小米手机",
    }
    var p1 Usb = phone1     //phone1 实现了 Usb 接口 phone1 是 Phone 类型
    p1.Start()
    phone2 := &Phone{    // 二：实例化指针类型
      Name: "苹果手机",
    }
    var p2 Usb = phone2     //phone2 实现了 Usb 接口 phone2 是 *Phone 类型
    p2.Start()        //苹果手机 开始工作
}
```

### 2. 指针接收者

- 指针类型的接收者由一个结构体的指针组成
- 由于指针的特性，调用方法时修改接收者指针的任意成员变量，在方法结束后，修改都是有效的。
- 这种方式就十分接近于其他语言中面向对象中的 this 或者 self 。
- 例如我们为 Person 添加一个 SetAge 方法，来修改实例变量的年龄。

### 3. 指针类型接收者 使用时机

注：并不是所有情况下都希望修改数据

- 需要修改接收者中的值
- 接收者是拷贝代价比较大的大对象
- 保证一致性，如果有某个方法使用了指针接收者，那么其他的方法也应该使用指针接收者。

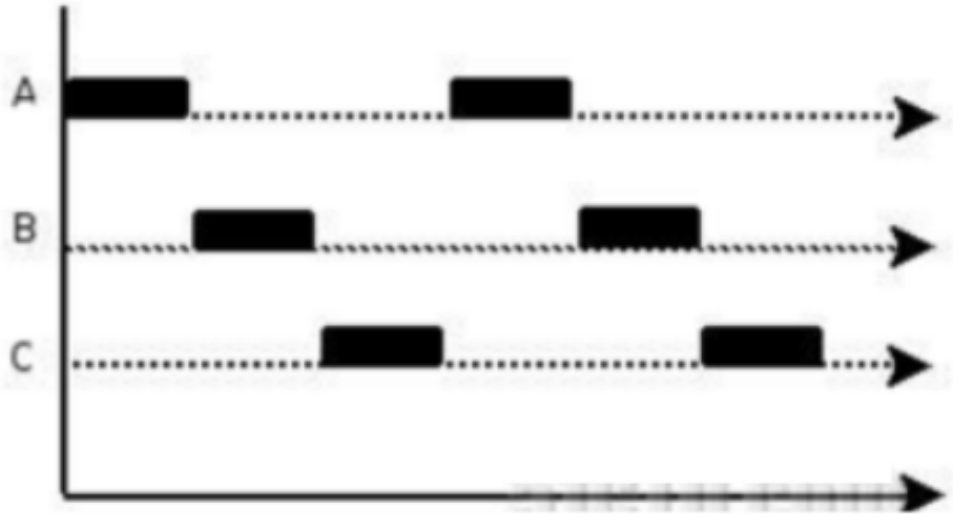
# 七、并发编程

## 6.1 并发介绍

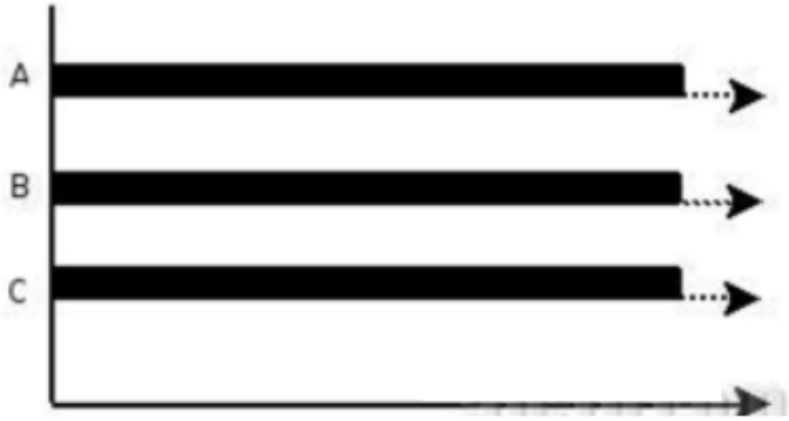
## 1. 并发和并行

```
A. 多线程程序在一个核的cpu上运行，就是并发。
B. 多线程程序在多个核的cpu上运行，就是并行。
```

- 并发：本质还是串行
  - 食堂窗口一个大妈（同一时间类只能给一个人打饭）
  - python本质没有并行的线程




- 并行：任务分布在不同CPU上，同一时间点同时执行
  - 并行就是有多个食堂大妈，同事给不同人打饭




## 2. 协程和线程

- 协程：独立的栈空间，共享堆空间，调度由用户自己控制，本质上有点类似于用户级线程，这些用户级线程的调度也是自己实现的
- 线程：一个线程上可以跑多个协程，协程是轻量级的线程。
- 线程和协程最大的区别
  - 开启一个线程需要大概2M空间，而且需要cpu调度才能执行，线程会强cpu
  - 开启一个协程大概只需要2K的空间，而且是有go解释器自己实现的GPM调度，主动退出
  - 所以我们可以同时启动成千上万个goroutine二不会过大的占用内存
  - 相反如果我们开启成千上万个线程，第一，会大量的占用内存，甚至导致机器奔溃，第二，操作系统调度线程，本身

也需要耗费大量时间

- 协程如果需要用CPU才会去使用CPU，如果没有使用CPU的需求，他就会主动把cpu让给其他协程执行
- 线程在时间片内，即使不使用CPU，比如当前正在从磁盘读数据，他也不会让出cpu

## 6.2 goroutine

### 1. 多线程编程缺点

- 在java/c++中我们要实现并发编程的时候，我们通常需要自己维护一个线程池
- 并且需要自己去包装一个又一个的任务，同时需要自己去调度线程执行任务并维护上下文切换

### 2. goroutine

- Go语言中的goroutine就是这样一种机制，goroutine的概念类似于线程，但 goroutine是由Go的运行时（runtime）调度和管理的。
- Go程序会智能地将 goroutine 中的任务合理地分配给每个CPU。
- Go语言之所以被称为现代化的编程语言，就是因为它在语言层面已经 内置了调度和上下文切换的机制 。
- 在Go语言编程中你不需要去自己写进程、线程、协程，你的技能包里只有一个技能-goroutine
- 当你需要让某个任务并发执行的时候，你只需要把这个任务包装成一个函数，开启一个goroutine去执行这个函数就可以了，就是这么简单粗暴。

## 6.3 协程基本使用

### 1. 启动一个协程

- 主线程中每个100毫秒打印一次，总共打印2次，另外开启一个协程，打印10次
- 情况一：打印是交替，证明是并行的
- 情况二：开启的协程打印两次，就退出了（因为主线程退出了）

```
package main
import (
    "fmt"
    "time"
)

func test() {
    for i := 0; i < 10; i++ {
        fmt.Println("test() 你好golang")
        time.Sleep(time.Millisecond * 100)
    }
}

func main() {
    go test() //表示开启一个协程
    for i := 0; i < 2; i++ {
        fmt.Println("main() 你好golang")
        time.Sleep(time.Millisecond * 100)
    }
}
/*
```

```
main() 你好golang
test() 你好golang
main() 你好golang
test() 你好golang
test() 你好golang
*/
```

## 2. WaitGroup

- 主线程退出后所有的协程无论有没有执行完毕都会退出
- 所以我们在主进程中可以通过WaitGroup等待协程执行完毕
  - sync.WaitGroup内部维护着一个计数器，计数器的值可以增加和减少。
  - 例如当我们启动了N 个并发任务时，就将计数器值增加N。
  - 每个任务完成时通过调用Done()方法将计数器减1。
  - 通过调用Wait()来等待并发任务执行完，当计数器值为0时，表示所有并发任务已经完成。

```
var wg sync.WaitGroup     // 第一步：定义一个计数器
wg.Add(1)         // 第二步：开启一个协程计数器+1
wg.Done()         // 第三步：协程执行完毕，计数器-1
wg.Wait()         // 第四步：计数器为0时推出
```

```
package main
import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup       // 第一步：定义一个计数器
func test1() {
    for i := 0; i < 10; i++ {
        fmt.Println("test1() 你好golang-", i)
        time.Sleep(time.Millisecond * 100)
    }
  wg.Done() //协程计数器-1   // 第三步：协程执行完毕，计数器-1
}
func test2() {
    for i := 0; i < 2; i++ {
        fmt.Println("test2() 你好golang-", i)
        time.Sleep(time.Millisecond * 100)
    }
    wg.Done() //协程计数器-1
}
func main() {
    wg.Add(1)  //协程计数器+1     第二步：开启一个协程计数器+1
    go test1() //表示开启一个协程
    wg.Add(1)  //协程计数器+1
    go test2() //表示开启一个协程
    wg.Wait() //等待协程执行完毕...   第四步：计数器为0时推出
```

```
    fmt.Println("主线程退出...")
}
/*
test2() 你好golang- 0
test1() 你好golang- 0
.....
test1() 你好golang- 8
test1() 你好golang- 9
主线程退出...
*/
```

### 3. 开启多个协程

- 在 Go 语言中实现并发就是这样简单，我们还可以启动多个 goroutine。
- 这里使用了 sync.WaitGroup 来实现等待 goroutine 执行完毕
- 多次执行上面的代码，会发现每次打印的数字的顺序都不一致。
- 这是因为 10 个 goroutine是并发执行的，而 goroutine 的调度是随机的

```go
package main
import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup
func hello(i int) {
    defer wg.Done() // goroutine结束就登记-1
    fmt.Println("Hello Goroutine!", i)
}
func main() {
    for i := 0; i < 10; i++ {
        wg.Add(1) // 启动一个goroutine就登记+1
        go hello(i)
    }
    wg.Wait() // 等待所有登记的goroutine都结束
}
/*
Hello Goroutine! 0
Hello Goroutine! 8
Hello Goroutine! 2
Hello Goroutine! 7
Hello Goroutine! 6
Hello Goroutine! 1
Hello Goroutine! 5
Hello Goroutine! 9
Hello Goroutine! 3
Hello Goroutine! 4
*/
```

# 6.4 channel

## 1. Channel说明

- 共享内存交互数据弊端
  - 单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。
  - 虽然可以使用共享内存进行数据交换，但是共享内存在不同的goroutine中容易发生竞态问题。
  - 为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。
- channel好处
  - Go 语言中的通道（channel）是一种特殊的类型。
  - 通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。
  - 每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。
  - 如果说goroutine是Go程序并发的执行体，channel就是它们之间的连接。
  - channel是可以让一个goroutine发送特定值到另一个goroutine的通信机制。

## 2. channel类型

- channel 是一种类型，一种引用类型
- 声明管道类型的格式如下:

```
var 变量 chan 元素类型
var ch1 chan int   // 声明一个传递整型的管道
var ch2 chan bool   // 声明一个传递布尔型的管道
var ch3 chan []int   // 声明一个传递 int 切片的管道
```

## 3. 创建channel

- 声明的管道后需要使用 make 函数初始化之后才能使用。
- 创建 channel 的格式如下:  make(chan 元素类型, 容量)

```
// 创建一个能存储 10 个 int 类型数据的管道
ch1 := make(chan int, 10)
// 创建一个能存储 4 个 bool 类型数据的管道
ch2 := make(chan bool, 4)
// 创建一个能存储 3 个[]int 切片类型数据的管道
ch3 := make(chan []int, 3)
```

## 4. channel操作

```
package main
import "fmt"

func main() {
    // 1、创建channel
    ch := make(chan int, 5)
    // 2、向channel放入数据
    ch <- 10
    ch <- 12
    fmt.Println("发送成功", ch)
    // 3、向channel取值
```

```
    v1 := <-ch
    fmt.Println(v1)
    v2 := <-ch
    fmt.Println(v2)
    // 4、空channel取值报错
    v3 := <-ch
    fmt.Println("v3", v3)
}
```

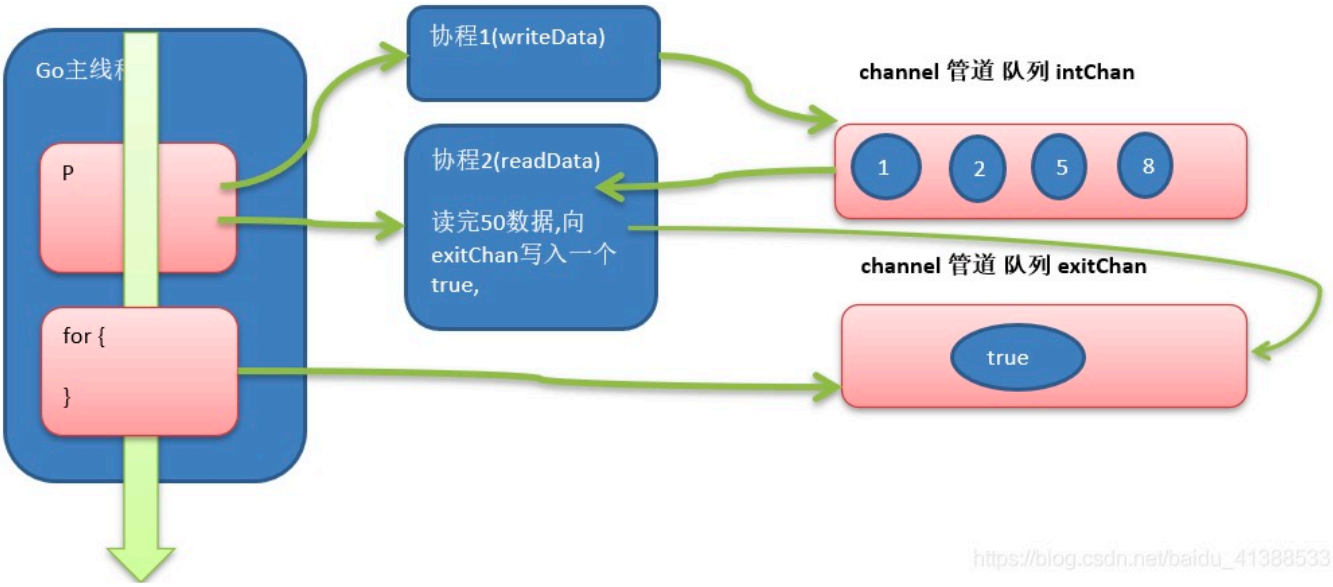
## 5. 优雅的从channel取值

- 当通过通道发送有限的数据时，我们可以通过close函数关闭通道来告知从该通道接收值的goroutine停止等待。
- 当通道被关闭时，往该通道发送值会引发panic，从该通道里接收的值一直都是类型零值。
- 那如何判断一个通道是否被关闭了呢?
- for range的方式判断通道关闭

```
package main
import "fmt"

func main() {
    ch := make(chan int, 5)
    ch <- 10
    ch <- 12
    ch <- 14
    ch <- 16
    ch <- 18
    close(ch)
    for i := range ch { // 通道关闭后会退出for range循环
    fmt.Println(i)
    }
}
```

## 6. goroutine和chanel小案例

- 请完成goroutine和channel协同工作的案例，具体要求:
  1）开启一个writeData协程，向管道intChan中写入50个整数.
  2）开启一个readData协程，从管道intChan中读取writeData写入的数据。
  3）注意: writeData和readDate操作的是同一个管道
  4）主线程需要等待writeData和readDate协程都完成工作才能退出





```go
package main
import (
    "fmt"
)

//write Data
func writeData(intChan chan int) {
    for i := 1; i ⩽ 50; i++ {
        //放入数据
        intChan<- i //
        fmt.Println("writeData ", i)
    }
    close(intChan) //关闭
}

//read data
func readData(intChan chan int, exitChan chan bool) {
    for {
        v, ok := <-intChan
        if !ok {
            break
        }
        fmt.Printf("readData 读到数据=%v\n", v)
    }
    //readData 读取完数据后，即任务完成
    exitChan<- true
    close(exitChan)

}

func main() {
    //创建两个管道
    intChan := make(chan int, 10)
    exitChan := make(chan bool, 1)
```

```
    go writeData(intChan)
    go readData(intChan, exitChan)

    for {
        _, ok := <-exitChan
        if !ok {
            break
        }
    }
}
```

## 6.5 select 多路复用

### 1. select说明

- 传统的方法在遍历管道时，如果不关闭会阻塞而导致 deadlock，在实际开发中，可能我们不好确定什么时候关闭该管道。
- 这种 for range 方式虽然可以实现从多个管道接收值的需求，但是运行性能会差很多。为了应对这种场景，Go 内置了 select 关键字，可以同时响应多个管道的操作。
- select 的使用类似于 switch 语句，它有一系列 case 分支和一个默认的分支。
- 每个 case 会对应一个管道的通信（接收或发送）过程。
- select 会一直等待，直到某个 case 的通信操作完成时，就会执行 case 分支对应的语句。
- 具体格式如下:

```
  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
```

### 2. select的使用

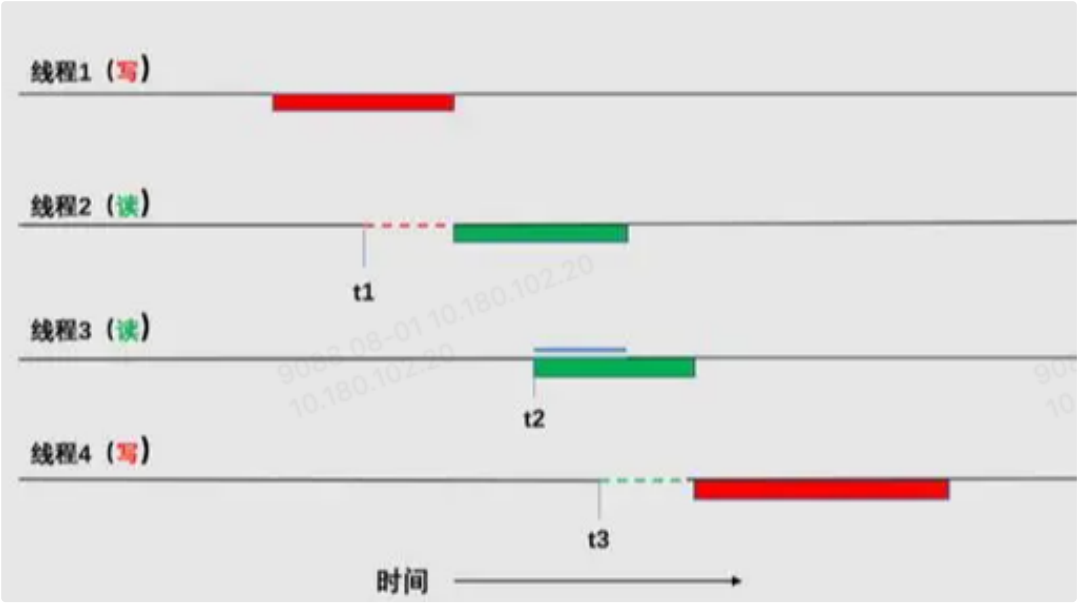
- 使用 select 语句能提高代码的可读性。
- 可处理一个或多个 channel 的发送/接收操作。
- 如果多个 case 同时满足，select 会随机选择一个。
- 对于没有 case 的 select{}会一直等待，可用于阻塞 main 函数。

```
package main

import (
  "fmt"
)

func main() {
  // 在某些场景下我们需要同时从多个通道接收数据,这个时候就可以用到golang中给我们提供的select多路复用
```

```
  //1.定义一个管道 10个数据int
  intChan := make(chan int, 10)
  for i := 0; i < 10; i++ {
    intChan <- i
  }
  //2.定义一个管道 5个数据string
  stringChan := make(chan string, 5)
  for i := 0; i < 5; i++ {
    stringChan <- "hello" + fmt.Sprintf("%d", i)
  }
  //使用select来获取channel里面的数据的时候不需要关闭channel
  for {
    select {
    case v := <-intChan:
      fmt.Printf("从 intChan 读取的数据%d\n", v)
    case v := <-stringChan:
      fmt.Printf("从 stringChan 读取的数据%v\n", v)
    default:
      fmt.Printf("数据获取完毕")
      return //注意退出...
    }
  }
}
```

## 6.6 互斥锁

- 互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个 goroutine 可以访问共享资源。




- Go语言中使用 sync 包的 Mutex 类型来实现互斥锁。
- 在多核的系统中，这1000个协程被分配到多个线程里面运行，那么就有可能是并行运行的，比如当前的num=987,后面两个线程同时执行那么结果就是988，但是这不是我们想要的结果，我们想要的是999。那如何解决了，其实小伙伴们应该很容易就会想到，就是我们让num +=1 这个代码同时只让一个线程（这里表现为协程）运行就好了

```
package main
```

```go
import (
  "fmt"
  "sync"
)

var num int

var mtx sync.Mutex
var wg sync.WaitGroup

func add() {
  defer wg.Done()
  mtx.Lock()
  num += 1
  mtx.Unlock()
}

func main() {
  for i := 0; i < 1000; i++ {
    wg.Add(1)
    go add()
  }
  wg.Wait()
  fmt.Println("num:", num)
}
```