问题

1.conn.read([]byte)

当conn不管是客户端还是服务端关闭后，都不会再阻塞，会继续往下执行

2.redis数据库中的key一定要与代码中对象的属性一样可以用tag改变在json中的名字

golang安装 www.jetbrains.com

破解 www.javatiku.cn/golang/11273.html

一、基础

1.go语言基础:

- 包声明，编写源文件时，必须在非注释的第一行指明这个文件属于哪个包，如 package main。
- 引入包，其实就是告诉Go 编译器这个程序需要使用的包，如import "fmt"其实就是引入了fmt包。
- 函数，和c语言相同，即是一个可以实现某一个功能的函数体，每一个可执行程序中必须拥有一个main函数。
- 变量，Go 语言变量名由字母、数字、下划线组成，其中首个字符不能为数字。
- 语句/表达式， 在Go程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号;结尾， 因为这些工作都将由 Go 编译器自动完成。
- 注释，和c语言中的注释方式相同，可以在任何地方使用以 // 开头的单行注释。以 /* 开头，并以 */ 结尾来进行多行注释，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。
- 一个变量分配在堆上还是栈上，与语法无关，主要依靠golang的逃逸分析，所以很多内存问题不必担心

1.1 输入输出

```
fmt.Println()    //打印并换行
fmt.Printf()     //格式化输出
fmt.Print()      //打印输出

//遇空格或回车结束
fmt.Scanf()
fmt.Scanln()
fmt.Scan()
```

Println输出可以接收多个变量，或者直接输出数组

```
a := 1
b := 2
arr := [3]int{1,2,3}
println(a, b)// 1 2
println(arr) //1 2 3
```

格式化输出

```
%v      仅输出该变量的值
%T      输出变量类型
%d      整数
%.1f    保留一位小数输出浮点数
%s      字符串
%c      字符
%p      地址
```

2.变量

2.1 变量定义

变量类型总是放在变量名后面面，但可以省略（根据初始化值进行行类型推断）

```
package main
import "fmt"
func main() {
    //var(关键字) 变量名 变量类型
    var a int = 27
    //省略变量类型
    var b = "hello"
    //使用:=
    c := "123"
    //同时声明多个类型相同的变量
    var a, b int
    var x, y = 1, "sdf"
    c, d := 1, 2
    //
    var(
        a int
        b int
    )
    fmt.Println(a, b, c);
}
```

指定变量类型，如果没有初始化，则变量默认为零值。**零值就是变量没有做初始化时系统默认设置的值。**

- bool: false
- integers: 0
- floats: 0.0
- string: ""
- pointers, functions, interfaces, slices, channels, maps: nil

2.2 匿名变量

匿名变量的特点是一个下画线_，这本身就是一个特殊的标识符，被称为空白标识符。它可以像其他标识符那样用于变量的声明或赋值（任何类型都可以赋值给它），但任何赋给这个标识符的值都将被抛弃，因此这些值不能在后续的代码中使用，也不可以使用这个标识符作为变量对其它变量进行赋值或运算。

```
//使用匿名变量时，只需要在变量声明的地方使用下画线替换即可。
func GetData() (int, int) {
    return 10, 20
}
func main(){
    a, _ := GetData()
    _, b := GetData()
    fmt.Println(a, b)
}
```

需要注意的是匿名变量不占用内存空间，不会分配内存。**匿名变量与匿名变量之间也不会因为多次声明而无法使用。**

2.3 变量作用域

作用域指的是已声明的标识符所表示的常量、类型、函数或者包在源代码中的作用范围，在此我们主要看一下go中变量的作用域，根据变量定义位置的不同，可以分为一下三个类型：

- 函数内定义的变量为局部变量，这种局部变量的作用域只在函数体内，函数的参数和返回值变量都属于局部变量。这种变量在存在于函数被调用时，销毁于函数调用结束后。
- 函数外定义的变量为全局变量，全局变量只需要在一个源文件中定义，就可以在所有源文件中使用，甚至可以使用import引入外部包来使用。全局变量声明必须以 var 关键字开头，如果想要在外部包中使用全局变量的首字母必须大写。
- 函数定义中的变量成为形式参数，定义函数时函数名后面括号中的变量叫做形式参数（简称形参）。形式参数只在函数调用时才会生效，函数调用结束后就会被销毁，在函数未被调用时，函数的形参并不占用实际的存储单元，也没有实际值。形式参数会作为函数的局部变量来使用。

2.4 注意

- 编译器会认为一个未被使用的变量和导入包是个错误。
- 在函数内部，甚至至可以省略var关键字，**用:=这种更短的表达式完成变量类型推断和初始化。** 函数外部不能使用赋值语句，故不能使用

```
a := 1 //等价入戏
var a int
a = 1; //赋值语句
```

3.常量

3.1 使用

使用const关键字，必须初始化且不能被修改

```
const x uint32 = 123
const y = "Hello"
const a, b, c = "meat", 2, "veg" // 同样支支持一一次定义多个常量。
const (
z = false
a = 123
)
```

在定义常量组时，如不提供初始化值，**则表示示与上行行常量的类型、值表达式完全相同**（是表达式相同而而非非结果相同）。

```
func main() {
    const (
        a = "abc"
        b
    )
    println(a, b)    //abc abc
}
```

常量值还可以是 len()、cap()，unsafe.Sizeof() 常量计算表达式的值。

```
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)
```

如果目标类型足足以存储常量值，不会导致溢出（overflow），可不做显式转换。

```
var million int = 1e6   // float syntax OK here
var b byte = 'a'        // int to byte
```

3.2 枚举

可以用用 iota 生生成从 0 开始的自自动增长长枚举值。按行行递增，可以省略后续行行的 iota 关键字。

```
const (
    Sunday = iota   // 0
    Monday          // 1
    Tuesday         // 2
    Wednesday       // 3
)
const (
    a = iota        //0
    b               //1
    c = "haha"      //haha   iota 2
    d               //haha   iota 3
    e = iota        //4
)
```

4.基本数据类型

类型	描述
bool	true,false。默认false,不能把非非零值当作 true。
字符串	string
uint8 / uint16 / uint32 / uint64	无符号 8 / 16 / 32 / 64位整型
int8 / int16 / int32 / int64	有符号8 / 16 / 32 / 64位整型
float32 / float64	IEEE-754 32 / 64 位浮点型数
complex64 / complex128	32 / 64 位实数和虚数
byte	类似 uint8
rune	类似 int32
uintptr	无符号整型，用于存放一个指针

Go 语言变量名由字母、数字、下划线组成，其中首个字符不能为数字。

可以用用 八八进制（071）、十十六进制（0xFF）或 科学计数法（1e2）对整数类型进行行赋值。math 包中包含了 Min/Max 开头的各类型最小、最大值常量。

4.1 类型转换

不支持隐式转换 ，必须进行行显式类型转换 "(expression)"。

```
a := 3
b := 5.0
c := float64(a)
d := int(b)
e := []byte("sdf");
```

类型转换只能在取值范围较小的类型转换到取值范围较大的类型，当取值范围较大的转换为取值范围较小的类型时会发生精度丢失。

5.运算符

Go运算符都是从左到右结合

算术运算符：+ - * / % ++ --

关系运算符：== ≠ > < ⩾ ⩽

逻辑运算符：&& || !

位运算符：& | ^ &^ << >>

赋值运算符：= += ...

6.流程控制

6.1 if

if语句只需记住：

- 条件表达式没有括号；
- 支持一个初始化表达式（可以是多变量初始化语句）；
- 左大括号必须和条件语句在同一行，不然会编译错误。

```
func main() {
    a := 10
    if a > 0 {  //左大括号必须写在这，否则被解释为 "if a > 0;"导致编译出错。
        a += 100
    } else if a == 0 { // 注意左大括号位置。
        a = 0
    } else {         // 注意左大括号位置。
        a -= 100
    }
    println(a)
}
```

6.2 switch

switch case 表达式可以使用用任意类型或表达式。也不必写 break，自自动终止止。如希望继续下一case 处理，则须显式执行fallthrough。

```
switch score {
    case 90:
        fmt.println("123")
    case 79:
        a := 10
        fallthrough
    case 80:
}
//默认是有一个true或false的匹配，且默认是去匹配true
switch {
    case true:
    case false:
}
```

6.3 for

没有while循环，for有三种形式

```
for i:=0; i<10; i++ {}
for i < 10 {}
for {}  //默认true一直循环
```

6.4 break continue

同c/c++

7.指针

与C相同，取地址符同样是&，指针变量其实就是用于存放某一个对象的内存地址。

7.1 指针声明和初始化

和基础类型数据相同，在使用指针变量之前我们首先需要申明指针，声明格式如下：

```
var ip *int        /* 指向整型*/
var fp *float32    /* 指向浮点型 */
```

指针的初始化就是取出相对应的变量地址对指针进行赋值，具体如下：

```
var a int = 20   /* 声明实际变量 */
var ip *int        /* 声明指针变量 */
*p = &a  /* 指针变量的存储地址 */
```

7.2 空指针

当一个指针被定义后没有分配到任何变量时，它的值为 `nil` ，也称为空指针。它概念上和其它语言的null、NULL一样，都指代零值或空值。

8.字符串

string是不可变值类型，Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用 UTF-8 编码标识 Unicode文本

8.1 注意

- 字符串一旦赋值了，字符串就不能修改了
- 字符串拼接直接用‘+’
- 可以用用索引号访问某个字节，如 s[i]。
- 不能通过序号获取字节元素指针，&s[i] 是非非法的。

8.2 遍历方法

```
for i := 0; i < len(s); i++ {
    fmt.Printf("%c", s[i])
}
//快捷输入 forr
for i, v := range str {
    fmt.Print(i)
    fmt.Printf("%c", v)
}
```

二、函数、包和错误处理

- 函数是基本的代码块，用于执行一个任务
- Go语言最少有个main函数
- **不支持重载和默认参数**

1.函数定义和使用

- fun 函数名(参数，参数...) 返回值类型 {函数体}
- 如果函数返回值只有一个可以不写()
- 函数返回值可以有多个，在接收是，如果希望忽略某个返回值可以**使用匿名变量_占位**

```
func add(a, b int) int {
    c := a + b
    return c
}
//可以有多个返回值
func f(a int, b int) (int, int) {
    return 1, 2
}
a, b := f(1, 2)
//可以自动返回参数
func f() (a int, b int) {
    a = 10
    b = 20
    return
}
//可变参数，必须要是最后一个参数，一个函数最多只有一个可变参数
func f(a int, nums ...int) int {
    var res int = 0;
```

```
    for i := 0; i < len(nums); i++ {
        res += nums[i]
    }
    return res
}
f(1,2,3,4)  //9
```

2.包

2.1 基本概念

go的每一个文件都是属于一个包的，也就是说go是以包的形式来管理文件和项目目录结构的。通过包可以区分相同名字的函数、变量等标识符，当程序文件很多时,可以很好的管理项目控制函数、变量等访问范围，即作用域。

2.2 使用

打包 package 包名

引入包　import"包的路径"
import"project1/..."
import (
"包名"
"包名"
别名 “包名”
)

访问其他包函数或变量　　包名.函数/变量

2.3 注意与细节

- package 指令在 文件第一行，然后是 import 指令。
- 一个程序中一个函数或变量要被其他包使用时，名称首字母要大写，类似public
- 在 import 包时，路径从 $GOPATH 的 src 下开始，不用带 src ，编译器会自动从 src 下开始引入
- 如果包名较长，Go支持给包取别名(直接在包名前面写)，**取别名后，原来的包名就不能使用了**
- 如果你要编译成一个可执行程序文件，就需要将这个包声明为main，即package main，这是一个语法规范，如果你是写一个库，包名可以自定义

3.init函数

3.1 介绍

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

3.2 注意

- 如果一个文件同时包含全局变量定义，init函数和main函数，则执行的流程为全局变量定义→init函数→main函数
- init 函数最主要的作用，就是完成一些**初始化的工作**，类似java静态代码块
- 如果两个包都含有全局变量、init函数、则执行流程如下

4.匿名函数

在函数内部不能定义函数，但匿名函数可以，匿名函数就是没有名字的函数，如果我们某个函数只是希望使用一次，可以考虑使用匿名函数，匿名函数也可以实现多次调用。

4.1 使用

```
func main() {
    //在定义匿名函数是就直接调用
    res1 := func(a, b int) int {
        return a + b
    }(1,2)
}
```

4.2 函数变量

函数其实也是一个变量，其类型为 func(参数类型) 返回值

```
//将匿名函数赋给一个变量
a := func(a, b int) int {
    return a + b
}
res1 = a(1, 2)
res2 = a(2, 3)
```

5.闭包

闭包就是**一个函数**和与**其相关的引用环境**组合的**一个整体**(实体)

5.1 认识其结构

```
//累加器
func AddUpper() func(int)int {
    var n int = 10
    return func (x int) int {
        n = n + x
        return n
    }
}

func main() {
    f := AddUpper()
    fmt.Println(f(1)) //n=11
    fmt.Println(f(1)) //n=12
    fmt.Println(f(1)) //n=13
}
```

可以看到，main函数中的变量在接收到AddUpper返回的匿名函数后，f调用时操作了AddUpper函数中的n，每次对n的修改都会保留。**类似面向对象中的类。**

闭包就是返回的函数和操作的外部函数变量构成的

5.2 案例

编写一个函数 makeSuffix(suffix string),可以接收一个文件后缀名(比如.jpg)，并返回一个闭包。调用闭包，可以传入一个文件名，如果该文件名没有指定的后缀(比如.jpg) ,则返回 文件名.jpg ， 如果已经有.jpg 后缀，则返回原文件名。

```
func makeSuffix(suffix string) func(string)string {
    return func (name string) string {
        if !strings.HasSuffix(name, suffix) {
            return name + suffix
        }
        return name
    }
}

func main() {
    f := makeSuffix(".jpg")
    fmt.Println(f("winter")) //winter.jpg
```

```
    fmt.Println(f("bird.jpg"))//bird
}
```

返回的匿名函数和 makeSuffix (suffix string)的 suffix 变量 组合成一个闭包。如果使用传统的方法，也可以轻松实现这个功能，但是传统方法需要每次都传入 后缀名，比如 .jpg ,而闭包因为可以保留上次引用的某个值，所以我们传入一次就可以反复使用

6.函数的defer

在函数中，程序员经常需要创建资源(比如：数据库连接、文件句柄、锁等），为了在**函数执行完毕后，及时的释放资源**，Go 的设计者提供 defer （延时机制）。

6.1 案例

```
func sum(a, b int) int {
    defer fmt.Println(a) //3.a=1
    defer fmt.Println(b) //2.b=2

    res := a + b
    fmt.Println(res) //1.res=3
    return res
}
func main() {
    res := sum(1, 2)
    fmt.Println(res) //4.res=3
}
```

6.2 注意

- 当函数执行到defer时，暂不执行，会将defer后面的语句压入defer栈中
- 当函数执行完毕后，在从defer栈按照先入后出的顺序执行在 defer 将语句放入到栈时，也会将相关的值拷贝同时入栈
- 在 defer 将语句放入到栈时，也会将相关的**值拷贝**同时入栈，其值不会改变。

```
func sum(a, b int) int {
    defer fmt.Println(a) //3.a=1
    defer fmt.Println(b) //2.b=2
    //增加一句话
    a++ //2
    b++ //3
    res := a + b    //5
    fmt.Println(res) //1.res=3
    return res
}
```

```
func main() {
    res := sum(1, 2)
    fmt.Println(res) //4.res=3
}
```

6.3 defer的最佳实践

```
func test() {
    //关闭文件资源
    file = openfile(文件名)
    defer file.close()
    //其他代码
}
func test() {
    //释放数据库资源
    connect = openDatabase()
    defer connect.close()
    //其他代码
}
```

这种机制，非常简洁，程序员不用再为在什么时机关闭资源而烦心。

7.值传递和引用传递

值传递的参数是基本数据类型**int系列、float系列、bool、string、数组和结构体**

引用传递的参数是**指针、slice切片、map、管道chan、interface等**

特点：...

8.错误处理

在默认情况下，当发生错误后(panic) ,程序就会退出（崩溃）。如果我们希望：当发生错误后，可以捕获到错误，并进行处理，保证程序可以继续执行。还可以在捕获到错误后，给管理员一个提示(邮件,短信。。。)

8.1 基本说明

Go 语言追求简洁优雅，所以，Go 语言不支持传统的 try…catch…finally 这种处理。Go 中引入的处理方式为：**defer, panic, recover**

```
func test() {
    //使用defer+recover来捕获异常
    defer func() {
        err := recover() //recover()是内置函数，可以捕获到异常
        if err ≠ nil {
            fim.Println("err=", err)
```

```
            //这里可以发送错误信息给管理员
            ...
        }
    }()
    a := 10
    b := 0
    res := a/b
    fmt.Println("res=", res)
}
func main() {
    test()
    fmt.Println("sdfdsdf")
}
```

8.2 自定义错误

Go 程序中，也支持自定义错误， 使用 errors.New 和 panic 内置函数。

- errors.New("错误说明") ,会返回一个**error类型**的值，表示一个错误
- panic内置函数 ,接收一个interface{}类型的值（也就是任何值）作为参数。可以接收 error 类型的变量，**输出错误信息**，**并退出程序**.

```
// 函数去读取配置文件init.conf的信息
// 如果文件名传入不正确，我们就返回一个自定义的错误
func readConf(name string) error {
    if name == "config.ini" {
        //读取...
        return nil
    } else {
        //返回一个自定义错误
        return errors.New("读取文件错误...")
    }
}

func main() {
    err := readConf("config2.ini")
    if err != nil {
        //如果读取文件错误，就输出这个错误，并终止程序
        panic(err)
    }
    fmt.Println("继续执行")
}
```

三、常用库函数

1.字符串常用函数

(1)len：按字节统计字符串的长度

```
s := "北京"
s2 := "err"
fmt.Println(len(s))  //6
fmt.Println(len(s2)) //3
```

(2)输出字符串中的中文的方法：r := []rune(s)

```
s := "北京"
r := []rune(s)
for i := 0; i < len(r); i++ {
    fmt.Printf("%c", r[i])
}
```

(3)字符串转整数：n, err := strconv.Atoi("12")

```
//需导包 strconv
n, err := strconv.Atoi("hello")
if err ≠ nil {
    fmt.Println("转换错误", err)
} else {
    fmt.Println("转成的结果是",  n)
}
```

(4)整数转字符串：str = strconv.Itoa

```
str = strconv.Itoa(12345)
fmt.Printf("%v", str)
```

(5)字符串转[]byte：var arr = []byte("hello go")

```
var bytes = []byte("hello go")
fmt.Print(bytes)
```

(6)[]byte转字符串：str = string([]byte{97,98,99})

```
str = string([]byte{97,98,99})
fmt.Printf("%v", str)
```

(7)10进制转2,8,16进制： str = strconv.FormatInt(123,2) //2→8, 16

```
str = strconv.FormatInt(123, 2)
fmt.Printf("123对应的二进制是%v", str)
str = strconv.FormatInt(123, 16)
fmt.Printf("123对应但是16进制是%v", str)
```

(8)查找子串是否在指定的字符串中：strings.Contains("seafood", "food")

```
fmt.Println(strings.Contains("seafood", "food")) //true
```

(9)统计一个字符串有几个指定的子串：strings.Count("ceheese", "e")

```
fmt.Printfln(strings.Count("ceheese", "e")) //4
```

(10)不区分大小写的字符串比较：fmt.Println(strings.EqualFold("abc", "Abc"))

```
fmt.Println(strings.EqualFold("abc", "Abc")) // true
```

(11)返回子串在字符串第一次出现的下标值，如果没有返回-1

```
strings.Index("NLT_abc", "abc") //4
```

(12)返回子串在字符串最后一次出现的 index，如没有返回-1

```
strings.LastIndex("go golang", "go")
```

(13)将指定的子串替换成另外一个子串： n可以指定你希望替换几个，如果n=-1表示全部替换

```
strings.Replace("go go hello", "go", "go 语言", n)
```

(14)按照指定的某个字符，为分割标识，将一个字符串拆分成字符串数组

```
strings.Split("hello,wrold,ok", ",")
```

(15)将字符串的字母进行大小写的转换：strings.ToLower("Go")

2.时间和日期相关函数

时间与日期函数需要导入time包，其中的**结构体类型time.Time表示时间**

(1)获取当前时间

```
now := time.Now()
fmt.Print(now)  //2024-02-29 20:42:12.1323412
```

(2)获取其他日期信息

```
now := time.Now()
now.Year() now.Month() now.Day() now.Hour() now.Minute()
now.Second()
```

(3)格式化日期时间

```
now := time.Now()
fmt.Printf(now.Format("2006/01/02 15:04:05"))  //字符串固定
```

(4)时间常量

```
const (
Nanosecond  Duration = 1 //纳秒
Microsecond = 1000 * Nanosecond //微秒
Millisecond = 1000 * Microsecond //毫秒Second=1000*Millisecond//
秒
Minute  = 60 * Second //分钟
Hour    = 60 * Minute //小时
)
```

(5)time 的 Unix 和 UnixNano 的方法

```
//Unix函数将t表示为Unix时间，即从时间点January1,1970 UTC到时间点t经过
的时间
func(t Time) Unix() int64
//UnixNano将t表示为Unix时间，即从时间点January1,1970 UTC到时间点t经过
的时间(单位/纳秒)，如果纳秒为单位的unix时间超出了int64能表示的范围，结果
是未定义的
func(t Time) UnixNano() int64

fmt.Print(now.Unix(), now.UnixNano())
//1312345344 12312435456465645767
```

3.内置函数

Golang 设计者为了编程方便，提供了一些函数，这些函数可以直接使用，我们称为 Go 的内置函数https://studygolang.com/pkgdoc

```
//len 用来求元素个数，比如 string、array、slice、map、channel
len(str)

//new 用来分配内存，主要用来分配值类型，比如 int、float32,struct...返回的是指针
p := new(int)

//make make: 用来分配内存，主要用来分配引用类型，比如channel、map、slice。
make(type, size)
```

四、数组与切片

4.1 数组定义和使用

```
//声明
var arr[10]int
//直接初始化
var arr = [3]int{1,2,3}
//省略var使用:=
arr := [3]int{1,2,3}
//数组长度不确定，在[]中输入...让编译器自行推导，也可以不写
var arr = [...]int{1,2,3}
//指定下标初始化
var arr = [5]int{1:1, 3:3}

//二维数组
var arr[2][3]int = [2][3]int{{1,2,3},{4,5,6}}
var arr[2][3]int = [...][3]int{{1,2,3},{4,5,6}}
```

4.2 数组的内存布局

- 数组的地址可以通过数组名来获取 `&intArr`
- 数组的第一个元素的地址，就是数组的首地址
- 数组的各个元素的地址间隔是依据数组的类型决定，比如 `int64 → 8 int32->4...`
- 二维数组第二行的元素是接着第一行元素的
- **数组也会默认初始化**

4.3 数组的遍历

使用Go语言独有的结构`for-range`遍历

```
func main() {
    //一维数组
    arr := [...]int{1,2,3}
    for i, v := range arr {
        fmt.Println(i, v)
    }
    //二维数组
    arr2 := [...][3]int{{1,2,3}, {4,5,6}}
    for i, v := range arr2 {
        for j, v2 := range v {
            fmt.Print(v2)
        }
        fmt.Println()
    }
}
```

4.4 切片slice基本使用

- 切片是数组的一个引用，因此切片是引用类型 ，在进行传递时，遵守引用传递的机制。
- 切片的**使用和数组类似**，遍历切片、访问切片的元素和求切片长度 `len(slice)`都一样。
- 切片的长度是可以变化的，因此切片是一个**可以动态变化数组**。

(1)切片的创建如下

```
//方式一：先定义一个数组，再定义一个切片去引用
var arr = [...]int{1,2,3,4,5}
var slice []int = arr[1:3]
fmt.Println("arr=", arr)
fmt.Println("slice=", slice)
fmt.Println("slice的元素个数是", len(slice))  //2
fmt.Println("slice的容量是", cap(slice))    //动态变化
//方式二：通过make来创建切片
var slice []int = make([]int, 2, 4(可不写))  // 创建了一个长度为2，容量为4的切片
//方式三：定义一个切片，直接就指定具体数组
var slice []int = []int{1,2,3}
```

注意：

- var slice = arr[0:end] 可以简写成var slice = arr[:end]
  var slice = arr[start:len(arr)] 可以简写成var slice = arr[start:]
  var slice = arr[0:len(arr)] 可以简写成var slice = arr[:]
- cap 是一个内置函数，用于统计切片的容量，即最大可以存放多少个元素。
- 切片定义完后，还不能使用，因为本身是一个空的，需要让其引用到一个数组，或者 make 一个空间供切片来使用
- 切片的遍历同数组

(2)**切片可以继续切片**

```
slice := []int{1,2,3}
slice2 = slice[:]
```

(3)**append函数**：可以对切片进行动态追加

```
slice := []int{1,2,3}
//通过append直接给slice追加元素
slice = append(slice, 4, 5, 6)
//通过append将切片追加给slice
slice = append(slice, slice2, slice3)
```

在底层slice扩容时，底层会创建一个新的数组，将所有元素拷贝过去，然后slice重新引用到这个新的数组。地址会改变

(4)**copy拷贝函数**

```
slice1 := []int{1,2,3,4,5}
slice2 := make([]int, 10)
copy(slice2, slice1)
fmt.Println(slice1)      //1 2 3 4 5
fmt.Println(slice2)      //1 2 3 4 5 0 0 0 0 0
```

注意：将长度大的slice拷贝给长度小的slice也不会出错

4.5 切片在内存中的形式

slice 的确是一个引用类型

slice 从底层来说，其实就是一个数据结构(struct 结构体)

```
type slice struct {
    ptr *[2]int //引用数组的地址
    len int     //长度
    cap int     //容量
}
```

4.6 slice与string

string 底层是一个 byte 数组，因此 string 也可以进行切片处理

```
str := "qweq"
slice := str[:]
```

因为string是不可变的，也就说不能通过切片方式来修改字符串

如果需要修改字符串，方法如下

```
//先将string→[]byte字节数组，修改完在转换为string
str := "asdf"
arr1 := []byte(str)
arr1[0] = 's'
str = string(arr1)
//先将string→[]rune字符数组，修改完在转换为string
//[]rune按字符处理，可以处理中文
arr1 := []rune(str)
arr1[0] = '你'
str = string(arr1)
```

五、map

5.1 map基本语法

5.1.1 map定义

```
var a map[string]string
```

5.1.2 map初始化方式

```
//方式一：直接初始化
var a map[string]int = map[string]int{
    "sa":   1,
    "sdf":  2,
    "sdfg": 3,
}
//方式二：make
var a map[string]int = make(map[string]int, 10)
var a map[string]int = make(map[string]int)
```

注意：

- map 的 key 是不能重复，如果重复了，则以最后这个 key-value 为准
- map 的 value 是可以相同的.
- map 的 key-value 是无序
- make函数的size可以省略，因为返回的是一个映射

5.1.3 map的增删改查

- map增加和更新：
  map["key"] = value //**如果key还没有，就是增加，如果key存在就是修改。**
- map删除：
  delete(map, "key") ，delete是一个内置函数，如果key存在，就删除该key-value,**如果 key不存在就不操作，但是也不会报错**
- 查找：
  val, ok := a["sa"]
  val是返回的值，没有就是默认值，ok是是否存在true/flase

5.1.4 map的遍历

```
var a map[string]int = map[string]int{
    "sa":   1,
    "sdf":  2,
    "sdfg": 3,
}
for k, v := range a {

}
```

5.2 map排序输出方法

先取出map所有key放入切片中，再将切片排序，然后按切片的顺序输出

```
map1 := map[int]int{
    2: 1, 3:4, 1: 6, 4: 3,
}
var keys []int
for i, _ := range map1 {
    keys = append(keys, i)
}
//排序
sort.Ints(keys)
//输出
for _, i := range keys {
    fmt.Println(map1[i])
}
```

5.3 注意细节

- **map是引用数据类型，** 故使用前必须指向一个地址，make或者初始化
- map 的容量达到后，再想 map 增加元素，会自动扩容，并不会发生 panic，也就是说 map 能动态的增长 键值对(key-value)
- map 的 **value 也经常使用 struct 类型**，更适合管理复杂的数据(比前面 value 是一个 map 更好)
- map，包括所有引用类型使用前必须初始化

六、面向对象编程

Golang 也支持面向对象编程(OOP)，但是和传统的面向对象编程有区别，并不是纯粹的面向对象语言。所以我们说 Golang 支持面向对象编程特性是比较准确的。

Golang 没有类(class)，Go 语言的结构体(struct)和其它编程语言的类(class)有同等的地位，你可以理解 Golang 是基于 struct 来实现 OOP 特性的。

Golang 面向对象编程非常简洁，去掉了传统 OOP 语言的继承、方法重载、构造函数和析构函数、隐藏的 this 指针等等

Golang 仍然有面向对象编程的**继承，封装和多态**的特性，只是实现的方式和其它 OOP 语言不一样，比如继承 ：Golang 没有 extends 关键字，继承是通过匿名字段来实现。

Golang 面向对象(OOP)很优雅，OOP 本身就是语言类型系统(type system)的一部分，通过接口(interface)关联，耦合性低，也非常灵活。后面同学们会充分体会到这个特点。也就是说在Golang 中面向接口编程是非常重要的特性。

6.1 结构体

6.1.1 声明

变量不用var，函数不用func

```
type cat struct {
    name string
    age int
    color string
}
```

6.1.2 初始化

```
//方式一
var person Person
//方式二
var person Person = Person{}
//该方式需要全部初始化
var person Person = Person{"afd", 13}
//该方式可以只写部分参数
var person Person = Person{name:"afd", age:13}
//方式三
var p *Person = new(Person)
//方式四
var p *Persono = &Person{}
```

6.1.3 注意细节

- 结构体是值类型
- 结构体的所有字段在**内存中是连续**的
- **结构体是用户单独定义的类型，和其它类型可以转换，但进行转换时需要有完全相同的字段(名字、个数和类型)**
- 结构体进行 type 重新定义(相当于取别名)，Golang 认为是新的数据类型，但是相互间可以强转

- struct 的每个字段上，可以写上一个 **tag**，该 tag 可以通过反射机制获取，常见的使用场景就是序列化和反序列化。

6.2 方法

Golang中的方法是**作用在指定的数据类型上的**(即：和指定的数据类型绑定)，因此**自定义类型， 都可以有方法**，而不仅仅是 **struct**。

6.2.1 格式

```
func (p Person) f1() int {
    return 1+2
}
func (p *Person) f1() int {
    return 1+2
}
```

多了一个与特定数据类型的参数，其实就相当于this

6.2.2 注意细节

- 自定义类型，都可以有方法，而不仅仅是struct，比如int,float32等都可以有方法
- **方法不能直接访问，只能有对应类型的对象访问**
- 如果一个类型实现了 String()这个方法，那么 fmt.Println 默认会调用这个变量的 String()进行输出
- p是一个结构体指针，p.name等价于(*p).name
- 若接收者为值类型，那么无论用值还是指针调用该方法，方法操作的都是对象的副本。若接收者为指针类型，那么无论用值还是指针调用该方法，方法操作的都是对象的指针

6.3 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。使用工厂模式实现跨包创建结构体实例(变量)的案例

```go
package model

type student stuct {
    Name string
    Score float64
}
//不能直接获取就定义一个函数返回结构体对象
func NewStudent(n string, s float64) *student {
    return &student {
        Name:n
        Score:s
    }
}
```

```go
package main
import "fmt"

func main() {
    var stu = model.NewStudent("tom~", 88.8)
}
```

如果 model 包的 **结构体变量首字母大写，引入后，直接使用**,没有问题,如果 model 包的 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**

类似构造函数，如果结构体中的变量也无法直接访问，则可以为结构体定义一个get方法

6.4 封装

封装的优点：

- 隐藏实现细节
- 提可以对**数据进行验证**，保证安全合理(Age)

如何封装：

- 通过**方法，包**实现
- 对结构体中的属性进行封装

6.4.1 实现步骤

1. 将结构体、字段(属性)的首字母小写(不能导出了，其它包不能使用，类似 private)
2. 给结构体所在包提供一个工厂模式的函数，首字母大写。类似一个构造函数
3. 提供一个首字母大写的Set方法(类似其它语言的public)，用于对属性判断并赋值
4. 提供一个首字母大写的Get方法(类似其它语言的 public)，用于获取属性的值

特别说明：在 Golang开发中并没有特别强调封装，这点并不像 Java. 所以提醒学过 java 的朋友，不用总是用java的语法特性来看待Golang,Golang本身对面向对象的特性做了简化的.

```
package entity

type person struct {
    name string
    age  int
    sal  float64
}

func NewPerson(s string) *person {
    return &person{
        name: s,
    }
}

func (p *person) SetAge(age int) {
    p.age = age
}
func (p *person) GetAge() int {
    return p.age
}
```

```
package test

import (
    "awesomeProject/entity"
    "fmt"
)

func main() {
    p := entity.NewPerson("zs")
    p.SetAge(10)
    fmt.Println(p.GetAge())
}
```

6.5 继承

6.5.1 嵌套匿名匿名结构体

在 Golang 中，如果一个struct嵌套了另一个匿名结构体，那么这个结构体可以直接访问匿名结构体的字段和方法，从而实现了继承特性。

```
type Goods struct {
    Name string
    Price int
}
type Book struct {
    Goods    //这里就是嵌套匿名结构体 Goods Writer string
    num int
}
```

6.5.2 深入讨论

(1)结构体可以**使用嵌套匿名结构体所有的字段和方法**

```
var b Book
b.Goods.Name = "adf"
b.Goods.Price = 12.4
```

(2)匿名结构体字段访问可以简化

```
b.Goods.Name → b.Name
b.Goods.Price → b.Price
```

(3)**当我们直接通过 b 访问字段或方法时，其执行流程如下**比如b.Name

- 编译器会先看 b 对应的类型有没有Name，如果有，则直接调用B类型的 Name 字段
- 如果没有就去看 B 中嵌入的匿名结构体 A 有没有声明 Name 字段，如果有就调用,如果没有继续查找..如果都找不到就报错.

(4)当**结构体**和**匿名结构体**有相同的字段或者方法时，**编译器采用就近访问原则访问**

(5)嵌套匿名结构体后，也可以在创建结构体时，直接指定各个**匿名结构体字段的**值

```
b := Book{Goods{"adf", "13.3"}, 20}
```

6.5.3 多重继承

如**一个 struct 嵌套了多个匿名结构体**，那么该结构体可以访问多个嵌套的匿名结构体的字段和方法，**从而实现了多重继承。**

- 如嵌入的匿名结构体有相同的字段名或者方法名，则在访问时，需要通过匿名结构体类型名来区分

- 为了保证代码的简洁性，建议大家尽量不使用多重继承

6.6 接口和多态

多态特征是通过接口实现的。可以按照统一的接口来调用不同的实现。这时接口变量就呈现不同的形态。

6.6.1 基本语法

```
type 接口名 interface {
    方法名(参数列表)返回值列表
    方法名(参数列表)返回值列表
}
```

```
type usb interface {
    start()
    stop()
}
//结构体phone
type phone struct {

}
func (p phone) start() {}
func (p phone) stop() {}
//结构体phone
type camera struct {

}
func (c camera) start() {}
func (c camera) stop() {}
//working函数
func working(u usb) {
    u.start()
    u.stop()
}

func main() {
    phone := phone{}
    camera := camera{}
    working(phone)
    working(camera)
}
```

6.6.2 注意细节

- 接口是引用类型
- 接口本身**不能创建实例**,但是**可以指向一个实现了该接口的自定义类型的变量**(实例)

```
var p phone
var u usb = p
```

- **接口没有变量，所有方法都没有方法体，方法不需要实现**
- 在Golang中，一个自定义类型需要**将某个接口的所有方法都实现**，我们说这个自定义类型实现了该接口。
- 一个自定义类型只有实现了某个接口，才能将该自定义类型的实例(变量)赋给接口类型
- 只要是自定义数据类型，就可以实现接口，不仅仅是结构体类型。
- 一个自定义类型可以实现多个接口
- **一个接口(比如 A 接口)可以继承多个别的接口(比如 B,C 接口)，这时如果要实现 A 接口，也必须将 B,C 接口的方法也全部实现。**
- **空接口interface{}没有任何方法，所以所有类型都实现了空接口， 即我们可以把任何一个变量 赋给空接口，可以类似java的Object**

6.6.3 接口实践

对结构体切片的排序

```
type Person struct {
    name string
    Age  int
}
//重命名切片，实现interface{}接口函数
type PersonSlice []Person

//实现三个函数
func (p PersonSlice) Len() int {
    return len(p)
}

func (p PersonSlice) Less(i, j int) bool {
    return p[i].Age < p[j].Age
}

func (p PersonSlice) Swap(i, j int) {
    temp := p[i]
    p[i] = p[j]
    p[j] = temp
```

```
}

func main() {
    p := PersonSlice{{"sd", 5}, {"d", 2}, {"df", 3}, {"sf", 1}}
    sort.Sort(p)
    for _, person := range p {
        fmt.Println(person.name, person.Age)
    }
}
```

6.7 类型断言

类型断言，由于接口是一般类型，不知道具体类型，如果要转成具体类型，就需要使用类型断言，

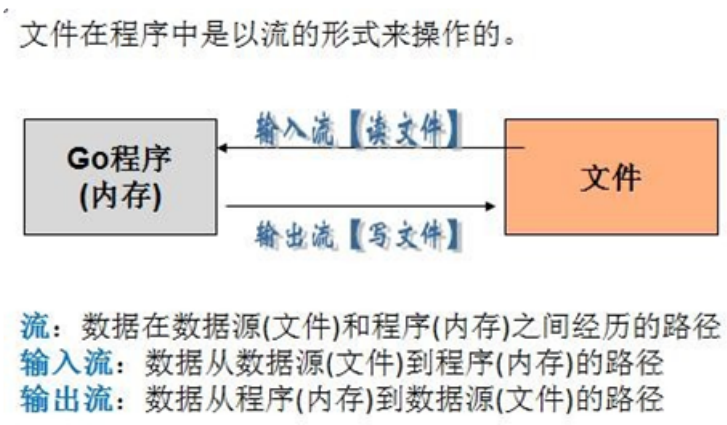
```
var x interface{}
var a int
a = x.(int)
```

如果接口变量指向某已实现类型的变量，则可以转换，否则报 panic

在进行断言时，带上检测机制，如果成功就 ok,否则也不要报 panic

```
var b float32 = 1.1
var x interface{} = b
//不会报panic
a, ok := x.(float64);
```

七、文件操作



os.File 封装所有文件相关操作，File 是一个结构体

7.1 打开和关闭文件

```
//打开文件
file, err := os.Open("D:/1.txt")
if err ≠ nil {
    fmt.Println("open file err:", err)
}
//关闭文件
err = file.close()
if err ≠ nil {
    fmt.Println("close err:", err)
}
```

7.2 读文件

7.2.1 带缓冲区的方式

```
import (
"fmt"
"os" "bufio" "io"
)

//file是*File类型
file, err := os.Open("d:/test.txt")
if err ≠ nil {
    fmt.Println("open file err:", err)
}
//当函数退出是要及时关闭文件
defer file.Close();
//创建 *Reader
reader := bufio.NewReader(file)
for {
```

```
    //读到一个换行就结束
    str, err := reader.ReadString('\n')
    if err == io.EOF {
        break
    }
    fmt.Print(str)
}
```

reader带缓冲区，缓冲区默认大小为4096

7.2.2 使用 ioutil 一次将整个文件读入到内存中

```
import (
"fmt"
"io/ioutil"
)
//使用ioutil.ReadFile一次性将文件读取到位
content, err := ioutile.Readfile("d:/test.txt")
if err ≠ nil {
    fmt.Println("read file err", err)
}
fmt.Print(string(content))
```

- 文件的open和close都封装到了Readfile内部
- content是一个[]byte切片

7.3 写文件

7.3.1 普通方式

```
file, err := os.OpenFile(file, os.O_WRONLY|os.O_CREATE, 0666)
if err ≠ nil {
    fmt.Println("open file err", err)
}
defer file.close()
//使用*Writer写文件
writer := bufio.NewWriter(file)
writer.writeString("sdf")

writer.Flush()
```

OpenFile函数参数：**文件位置， 打开模式，权限控制**

writerString会先写到缓存中，所以需要使用Flush方法将数据真正写到文件中

7.3.2 拷贝文件

```
file1, err1 := os.Open("D:\\E\\学习\\Go\\test.txt")
file2, err2 := os.OpenFile("D:\\E\\学习\\Go\\test2.txt",
os.O_WRONLY|os.O_APPEND, 111)
if err1 ≠ nil {
    fmt.Println("err1")
}
if err2 ≠ nil {
    fmt.Println("err2")
}

defer file1.Close()
defer file2.Close()

reader := bufio.NewReader(file1)
writer := bufio.NewWriter(file2)
io.Copy(writer, reader)
```

也可以通过io.Copy(writer, reader)实现

7.4 判断文件是否存在

golang判断文件或文件夹是否存在的方法为使用os.Stat()函数返回的错误值进行判断:
1) 如果返回的错误为nil,说明文件或文件夹存在
2) 如果返回的错误类型使用os.IsNotExist()判断为true,说明文件或文件夹不存在
3) 如果返回的错误为其它类型,则不确定是否在存在
//自己写了一个函数

```
func PathExists(path string) (bool, error) {
        _, err := os.Stat(path)
        if err == nil {// 文件或者目录存在
                return true, nil
        }
        if os.IsNotExist(err){
                return false, nil
        }
        return false, err
}
```

7.5 命令行参数

7.5.1 普通方法获取

`os.Args` 是一个 `string` 的切片，用来存储所有的命令行参数，遍历即可

7.5.2 flag 包用来解析命令行参数

```
//定义几个变量，用于接收命令行的参数值
var user string
var pwd string
var host string
var port int

//&user 就是接收用户命令行中输入的 -u 后面的参数值
//"u", 就是-u指定参数
//"", 默认参数
//"用户名，默认为空"，说明
flag.StringVar(&user, "u", "", "用户名,默认为空")
flag.StringVar(&pwd, "pwd", "", "密码，默认为空")
flag.StringVar(&hose, "h", "localhost", "主机名")
flag.StringVar(&port, "port", 3306, "端口号，默认为3306")

//非常重要的一个操作，必须调用该方法
flag.Parse()
//输出结果
fmt.Prinln(user, pwd, host, port)
```

7.6 json基本介绍

JSON是一种轻量级的**数据交换格式**。易于人阅读和编写。同时也易于及其解析和生成。JSON是在2001年开始推广使用的数据格式，目前已经成为 主流的数据格式

JSON易于机器解析和生成，通常程序在网络传输时会将数据(结构体、map、切片等) 序列化 成json字符串，到接收方得到json字符串后，再 反序列化 恢复为原来的数据。

7.6.1 json格式

在js语言中，一切都是对象。因此，任何的数据类型都可以通过json来表示，如字符串、数字、对象、数组、map、结构体等

json保存数据的方式是 键值对 ，键名用""包裹，冒号后接值

```
{"key1":val1, "key2":[val2, val3]},
```

可以通过网站https://www.json.cn/查询json语句格式是否正确

7.6.2 json的序列化

使用encoding/json包中的Marshal函数

```
//结构体序列化
type Person struct{
    Name string
    Age int
}

func main(){
    p := {"dsf", 234}
    //返回[]byte
    data, err := json.Marshal(&p)
}
```

注意:

- Marshal函数接收指针类型
- 为了在其他包中使用结构体的属性，结构体的属性必须大写
- 对于结构体的序列化，如果我们希望序列化后的key 的名字，又我们自己重新制定，那么可以给struct指定一个 tag 标签.

```
type Person struct{
    Name string `json:"person_name"`
    Age int     `json:"person_age"`
}
```

7.6.3 json的反序列化

json 反序列化是指，将 json 字符串反序列化成对应的数据类型(比如结构体、map、切片)的操作

通过encoding/json中的Unmashal函数实现

```
//反序列化结构体

//说明 str 在项目开发中，是通过网络传输获取到..  或者是读取文件获取到
str := "{\"Name\":\"牛魔王\",\"Age\":500,\"Birthday\":\"2011-11-11\",\"Sal\":8000,\"Skill\":\"牛魔拳\"}"
err := json.Unmarshal([]byte(str), &monster)
if err != nil {
    fmt.Println(err)
}
```

- 在反序列化一个json 字符串时，要确保**反序列化后的数据类型**和原来**序列化前的数据类型**一致。
- 如果 json 字符串是通过程序获取到的，则不需要再对 “ 转义处理。

八、单元测试

Go 语言中自带有一个轻量级的测试框架 testing 和自带的 go test 命令来实现单元测试和性能测试，testing 框架和其他语言中的测试框架类似，可以基于这个框架写针对相应函数的测试用例，也可以基于该框架写相应的压力测试用例。通过单元测试，可以解决如下问题：

- **确保每个函数是可运行，并且运行结果是正确的**
- 确保写出来的代码**性能是好**的，
- 单元测试能及时的发现程序设计或实现的**逻辑错误**，使问题及早暴露，便于问题的定位解决， 而**性能测试**的重点在于发现程序设计上的一些问题，让程序能够在高并发的情况下还能保持稳定

\1）测试用例文件名必须以 _test.go 结尾。 比如 cal_test.go , cal 不是固定的。

\2）测试用例函数必须以 Test 开头，一般来说就是 Test+被测试的函数名，比如 TestAddUpper

\3）TestAddUpper(t *tesing.T) 的形参类型必须是 *testing.T 【看一下手册】

\4）一个测试用例文件中，可以有多个测试用例函数，比如 TestAddUpper、TestSub

\5）运行测试用例指令
cmd>go test [如果运行正确，无日志，错误时，会输出日志]
cmd>go test -v [运行正确或是错误，都输出日志]

\6）当出现错误时，可以使用 t.Fatalf 来格式化输出错误信息，并退出程序

\7）t.Logf 方法可以输出相应的日志

\8）测试用例函数，并没有放在 main 函数中，也执行了，这就是测试用例的方便之处[原理图].

\9）PASS 表示测试用例运行成功，FAIL 表示测试用例运行失败

\10） 测试单个文件，一定要带上被测试的原文件

```
go test -v cal_test.go cal.go
```

\11）测试单个方法

```
go test -v -test.run  TestAddUpper
```

九、goroutine和channel

Go主线程(有程序员直接称为线程/也可以理解成进程)：一个 Go 线程上，可以起多个goroutine协程，你可以这样理解，协程是轻量级的线程[编译器做优化]。

Go协程的特点

- 有独立的栈空间
- 共享程序堆空间
- 调度由用户控制
- 协程是轻量级的线程

9.1 goroutine快速入门

```
func test() {
    for i := 0; i < 10; i++ {
        fmt.Println("hello, Go")
        time.Sleep(time.Second)
    }
}

func main() {
    go test()
    for i := 0; i < 10; i++ {
        fmt.Println("sdfsdf")
        time.Sleep(time.Second)
    }
}
```

- 主线程是一个物理线程，直接作用在 cpu 上的。是重量级的，非常耗费 cpu 资源。
- 协程从主线程开启的，是轻量级的线程，是逻辑态。对资源消耗相对小。
- Golang 的协程机制是重要的特点，可以轻松的**开启上万个协程**。其它编程语言的并发机制是一般基于线程的，开启过多的线程，资源耗费大，这里就突显 Golang 在并发上的优势了
- 主线程结束后其所有协程也会结束

9.2 goroutine调度模型(MPG)

M：操作系统的主线程

P：协程执行需要的上下文

G：协程

9.3 goroutine加锁

```
// 声明一个全局的互斥锁
var lock sync.Mutex
var a int = 0

func test() {
    lock.Lock()
    a++
    lock.Unlock()
}
func main() {
    for i := 0; i < 10; i++ {
        go test()
    }
    time.Sleep(time.Second*10)
}
```

9.4 channel

前面使用全局变量加锁同步来解决 goroutine 的通讯，但不完美，主线程在等待所有 goroutine 全部完成的时间很难确定，我们这里设置 10 秒，仅仅是估算。如果主线程休眠时间长了，会加长等待时间，如果等待时间短了，可能还有 goroutine 处于工作状态，这时也会随主线程的退出而销毁通过全局变量加锁同步来实现通讯，也并不利用多个协程对全局变量的读写操作。

上面种种分析都在呼唤一个新的通讯机制-channel

9.4.1 定义/声明和使用

channel本质是队列。线程安全，多协程访问时，不需要加锁。channel有类型，对应类型的channel存放对应类型的数据

```
var 变量名 chan 数据类型
var intChan chan int
//使用前必须make
intChan = make(chan int, 3)
intChan<-10
intChan<-11
num1 := <- intChan
fmt.Println(num1)
```

注意

- channel是引用数据类型， channel使用前必须make

- 在没有使用协程的情况下，如果 channel 数据取完了，再取，就会报 dead lock

9.4.2 channel的遍历和关闭

channel关闭后可读不可写，遍历时一般用for range前遍历前要关闭channel，如果写完后不关闭，去读会死锁

```
var intChan chan int
intChan = make(chan int, 100)
for i := 0; i < 100; i++ {
    intChan <- i
}
//只能用forrange遍历，遍历前如果不关闭channel,则会报错
close(intChan)
for v := range intChan {
    fmt.Println(v)
}
```

9.4.3 案例

writeData协程，向管道写50个整数

readData协程，向管道读数据

```
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
}

func readData(intChan chan int, exitChan chan bool) {
    for {
        time.Sleep(time.Second)
        v, ok := <-intChan
        if !ok {
            break
        }
        //time.Sleep(time.Second)
        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() {
    intChan := make(chan int, 50)
    exitChan := make(chan bool, 1)
    go writeData(intChan)
    go readData(intChan, exitChan)

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

- 当channel已满时继续写会阻塞，当channel为空时继续读也会阻塞，所以容易死锁。chan关闭后再读不会阻塞
- num, ok := <-inChan, 当inChan关闭后且intChan没有数据读后，ok为false，intChan没关闭读会阻塞

9.4.4 注意细节

1. channel 可以声明为只读，或者只写性质

```
//只写
var chan1 chan<- int
//只读
var chan2 <-chan int
```

2. 使用 select 可以解决从管道取数据的阻塞问题

```
intChan := make(chan int, 1)
//如果intChan一直没有关闭，不会一直阻塞而 deadlock,会自动到下一个case匹配
select {
    case v := <-intChan:
    fmt.Println(v)
    default:
}
```

3. goroutine 中使用 recover，解决协程中出现 panic，导致程序崩溃问题

十、反射

\1) 反射可以在运行时**动态获取变量的各种信息**，比如变量的类型(type)，类别(kind)

\2) 如果是结构体变量，还可以获取到结构体本身的信息(包括结构体的**字段**、**方法**)

\3) 通过反射，可以修改变量的值，可以调用关联的方法。

\4) 使用反射，需要 import (“**reflect**”)

10.1 快速入门

```
func reflectTest(b interface{}) {
    //通过反射获取传入的变量的type,kind
    //先获取reflect.Type
    rT := reflect.TypeOf(b)
    //获取reflect.Value
    rV := reflect.ValueOf(b)
    //rv转换为interface{}
    iv := rV.Interface()
    //iv通过类型断言转换
    stu, ok := iv.(Student)
}
```

10.2 注意细节

- reflect.Value.Kind，获取变量的类别，返回的是一个常量
- Type 是类型，Kind 是类别， Type 和 Kind 可能是相同的，也可能是不同的
  var num int = 10 num 的 Type 是 int , Kind 也是 int
  var stu Student stu 的 Type 是 pkg1.Student , Kind 是 struct
- 通过反射可以让变量在interface{}和reflect.Value之间转换
- 通过反射的来修改变量，注意当使用 SetXxx 方法来设置需要通过对应的指针类型来完成，这样才能改变传入的变量的值，同时需要使用到
  reflect.Value.Elem()方

```
var num int = 100
rv := reflect.ValueOf(&num) //需要传入内存地址
rv.Elem().SetInt(200)
fmt.Println(num)
```

10.3 案例

使用**反射来遍历结构体的字段**，**调用结构体的方法**，并**获取结构体标签**的值

```
//定义了一个 Monster 结构体
```

```go
type Monster struct {
    Name        string `json:"name"`
    Age int `json:"monster_age"`
    Score float32 `json:"成绩"`
    Sex string
}

//方法，返回两个数的和
func (s Monster) GetSum(n1, n2 int) int {
    return n1 + n2
}
//方法， 接收四个值，给 s 赋值
func (s Monster) Set(name string, age int, score float32, sex
string) {
    s.Name = name
    s.Age = age
    s.Score = score
    s.Sex = sex
}

//方法，显示 s 的值
func (s Monster) Print() {
    fmt.Println("---start~  ")
    fmt.Println(s)
    fmt.Println("---end~    ")
}
func TestStruct(a interface{}) {
    //获取 reflect.Type 类型
    typ := reflect.TypeOf(a)
    //获取 reflect.Value 类型
    val := reflect.ValueOf(a)
    //获取到 a 对应的类别
    kd := val.Kind()
    //如果传入的不是 struct，就退出if kd ≠
    reflect.Struct {
        fmt.Println("expect struct") return
    }
    //获取到该结构体有几个字段
    num := val.NumField()
    fmt.Printf("struct has %d fields\n", num) //4
    //变量结构体的所有字段
    for i := 0; i < num; i++ {
        fmt.Printf("Field %d:  值为=%v\n", i, val.Field(i))
```

```
        //获取到struct标签，注意需要通过 reflect.Type 来获取 tag 标
签的值
        tagVal := typ.Field(i).Tag.Get("json")
        //如果该字段于 tag 标签就显示，否则就不显示
        if tagVal != "" {
            fmt.Printf("Field %d: tag 为=%v\n", i, tagVal)
        }
    }
    //获取到该结构体有多少个方法
    numOfMethod := val.NumMethod()
    fmt.Printf("struct has %d methods\n", numOfMethod)
    //var params []reflect.Value
    //方法的排序默认是按照 函数名的排序（ASCII 码）
    val.Method(1).Call(nil) //获取到第二个方法。调用它

    //调用结构体的第 1 个方法 Method(0)
    var params []reflect.Value  //声明了 []reflect.Value
    params = append(params, reflect.ValueOf(10))
    params = append(params, reflect.ValueOf(40))
    //传入的参数是 []reflect.Value，返回[]reflect.Value
    res := val.Method(0).Call(params)
    //返回结果，返回的结果是 []reflect.Value*/
    fmt.Println("res=", res[0].Int())
}
func main() {
    //创建了一个 Monster 实例
    var a Monster = Monster{ Name: "黄鼠狼精",
                             Age: 400,
                             Score: 30.8,}
    //将 Monster 实例传递给 TestStruct 函数
    TestStruct(a)
}
```

十一、TCP编程

使用netstat -an查看本机有哪些端口在监听

使用netstat -anb查看监听端口的 pid,在结合任务管理器关闭不安全的端口

使用telnet测试服务器连接:telnet 127.0.0.1 8888

取消一个端口的监听：先查找这个端口的 pid>netstat -anb | findstr 8889
再taskkill /F /PID 23212

11.1 快速入门

server.go

```
package main

import (
    "fmt"
    "net"
)
//协程与客户端通信
func process(conn net.Conn) {
    defer conn.Close()
    //循环接收
    for {
        buf := make([]byte, 1024)
        //如果客户端没有发送就会一直阻塞
        n, err := conn.Read(buf[0:1024]})
        if err ≠ nil {
            fmt.Println("服务器端read出错: ", err)
            return
        } else {
            //在终端显示
            fmt.Println(string(buf[:n]))
        }
    }
}

func main() {
    fmt.Println("服务期开始监听...")
    //使用tcp协议，在本地监听8888端口
    listen, err := net.Listen("tcp", "0.0.0.0:8888")
    if err ≠ nil {
        fmt.Println("listen err:", err)
    }
    //延时关闭
    defer listen.Close()
    //循环等待客户端来连接
    for {
        //等待客户端连接
        fmt.Println("等待连接...")
        conn, err := listen.Accept()
        if err ≠ nil {
```

```
            fmt.Println("Accept err:", err)
        } else {
            fmt.Println("连接成功，客户端ip:",
conn.RemoteAddr().String())
            go process(conn)
        }
    }
}
```

client.go

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)

func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:8888")
    if err ≠ nil {
        fmt.Println("client dial err:", err)
    } else {
        fmt.Println("连接成功:", conn)
    }
    for {
        //从终端读取一行
        reader := bufio.NewReader(os.Stdin)
        line, err := reader.ReadString('\n')
        //去除换行符
        line = strings.Trim(line, "\n")
        if err ≠ nil {
            fmt.Println(err)
        }

        n, err := conn.Write([]byte(line))
        if err ≠ nil {
            fmt.Println("write err:", err)
        }
        fmt.Println("已发送", n, "字节")
```

```
    }
}
```

11.2 海量用户即时通讯系统

十二、Redis

12.1 基本介绍

。。。

12.2 操作

Redis 安装好后，默认有 16 个数据库，初始默认使用 0 号库，编号是 0...15

12.2.1 基本操作

keys *：查看redis当前所有key

select index：切换redis数据库

dbsize：查看当前数据库的 key-val 数量

flushdb/flushall：清空当前数据库的key-val/清空所有数据库的key-val

12.2.2 String

set/get/mset/mget/del

```
set name zs
set name 10 zs
get name

del name

mset name zs age 18
mget name age
```

set[如果存在就相当于修改，不存在就是添加]

12.2.3 Hash

hset/hget/hmset/hmget/hgetall/hdel/hlen/hexist

```
hset user1 name zs
hset user1 age 10
hget user1 name

hmset user1 name zs age 10
hmget user1 name age
hgetall user1
hdel user1
hlen user1
hexist user1 name
```

12.2.4 List

lpush/rpush/lrange/lpop/rpop/del/llen

```
lpush city a b c
rpush city d e f

lrange city 0 -1

lpop city
rpop city

del city
```

lrange使用：左边第一个元素下标为0，右边第一个元素下标为-1

12.2.5 Set

go连接RedisRedis 的 Set 是 string 类型的无序集合，不可重复

sadd/smembers/sismember/serm

```
sadd emails a b c

smembers emails

sismember emails a

serm emails a
```

12.3 go操作Redis

要安装第三方开源Redis库github.com/garyburd/redigo/redis

```
import (
    "fmt"
    "github.com/gomodule/redigo/redis"
)

func main() {
    //连接到redis
    conn, err := redis.Dial("tcp", "127.0.0.1:6379")
    if err ≠ nil {
        fmt.Println("redis.Dial err:", err)
        return
    }
    defer conn.Close()
    //set
    _, err := conn.Do("Set", "name", "zs")
    //get
    r, err2 :=  redis.String(conn.Do("get", "name"))
    //批量存入
    conn.Do("hmset", "m1", "name", m1.name, "age", m1.age,
"skill", m1.skill)
    //批量获取
    r, _ := redis.Strings(conn.Do("hgetall", "m1"))
}
```

12.4 连接池

\1） 事先初始化一定数量的链接，放入到链接池

\2） 当 Go 需要操作 Redis 时，**直接从 Redis 链接池取出链接**即可。

\3） 这样可以节省临时**获取 Redis 链接**的时间，从而提高效率.

```
var pool *redis.Pool

func init() {
    pool = &redis.Pool{
        MaxIdle:     8,    //最大空闲链接数
        MaxActive:   0,    //  表示和数据库的最大链接数，  0  表示没有
限制
        IdleTimeout: 100, //  最大空闲时间
```

```
        Dial: func() (redis.Conn, error) { // 初始化链接的代码，
链接哪个 ip 的 redis
            return redis.Dial("tcp", "localhost:6379")
        },
    }
}

func main() {
    conn := pool.Get()
    defer conn.Close()
}
```

连接池关闭后无法再取出连接

十三、go mysql

Go语言中的`database/sql`包提供了保证SQL或类SQL数据库的泛用接口，并不提供具体的数据库驱动。使用`database/sql`包时必须注入（至少）一个数据库驱动。

```
import (
    "database/sql"
    //注意必须手动导这个包
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    dsn := "root:123@tcp(127.0.0.1:3306)/mybatis"
    //指定数据库名和数据源
    //必须是mysql数据库
    db, err := sql.Open("mysql", dsn)
    if err ≠ nil {
        panic(err)
    }
    defer db.Close()
}
```

注意这行代码要写在上面err判断的下面，因为db可能为nil调用db.Close()会出错

13.1 初始化连接

Open函数可能只是验证其参数格式是否正确，实际上并不创建与数据库的连接。如果要检查数据源的名称是否真实有效，应该调用Ping方法。

返回的DB对象可以安全地被多个goroutine并发使用，并且维护其自己的空闲连接池。因此，Open函数应该仅被调用一次，很少需要关闭这个DB对象。

接下来，我们定义一个全局变量`db`，用来保存数据库连接对象。将上面的示例代码拆分出一个独立的`initDB`函数，只需要在程序启动时调用一次该函数完成全局变量db的初始化，其他函数中就可以直接使用全局变量`db`了。（**注意下方的注意**）

```
import (
    "database/sql"
    //注意必须手动导这个包
    _ "github.com/go-sql-driver/mysql"
)

// 定义一个全局对象db
var db *sql.DB

// 定义一个初始化数据库的函数
func initDB() (err error) {
    //dsn: data source name
    dsn := "user:password@tcp(127.0.0.1:3306)/sql_test?
charset=utf8mb4&parseTime=True"
    // 不会校验账号密码是否正确
    // 注意！！！这里不要使用:=，我们是给全局变量赋值，然后在main函数中使
用全局变量db
    db, err = sql.Open("mysql", dsn)
    if err ≠ nil {
        return err
    }
    // 尝试与数据库建立连接（校验dsn是否正确）
    err = db.Ping()
    if err ≠ nil {
        return err
    }
    return nil
}

func main() {
    err := initDB() // 调用输出化数据库的函数
    if err ≠ nil {
        fmt.Printf("init db failed,err:%v\n", err)
        return
    }
}
```

其中sql.DB是表示连接的数据库对象（结构体实例），它保存了连接数据库相关的所有信息。它内部维护着一个具有零到多个底层连接的连接池，它可以安全地被多个goroutine同时使用。

SetMaxOpenConns

```
func (db *DB) SetMaxOpenConns(n int)
```

SetMaxOpenConns设置与数据库建立连接的最大数目。 如果n大于0且小于最大闲置连接数，会将最大闲置连接数减小到匹配最大开启连接数的限制。 如果n≤0，不会限制最大开启连接数，默认为0（无限制）。

SetMaxIdleConns

```
func (db *DB) SetMaxIdleConns(n int)
```

SetMaxIdleConns设置连接池中的最大闲置连接数。 如果n大于最大开启连接数，则新的最大闲置连接数会减小到匹配最大开启连接数的限制。 如果n≤0，不会保留闲置连接。

13.2 crud操作

```
//单行查询
func (db *DB) QueryRow(query string, args ...interface{}) *Row
//多行查询
func (db *DB) Query(query string, args ...interface{}) (*Rows,
error)
//增删改查
func (db *DB) Exec(query string, args ...interface{}) (Result,
error)
```

单行查询

db.QueryRow()执行一次查询，并期望返回最多一行结果（即Row）。QueryRow总是返回非nil的值，直到返回值的Scan方法被调用时，才会返回被延迟的错误。（如：未找到结果）

```go
// 查询单条数据示例
func queryRowDemo() {
    sqlStr := "select id, name, age from user where id=?"
    var u user
    // 非常重要：确保QueryRow之后调用Scan方法，否则持有的数据库链接不会被释放
    err := db.QueryRow(sqlStr, 1).Scan(&u.id, &u.name, &u.age)
    if err != nil {
        fmt.Printf("scan failed, err:%v\n", err)
        return
    }
    fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name, u.age)
}
```

多行查询

多行查询`db.Query()`执行一次查询，返回多行结果（即Rows），一般用于执行select命令。参数args表示query中的占位参数。

```go
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

具体示例代码：

```go
// 查询多条数据示例
func queryMultiRowDemo() {
    sqlStr := "select id, name, age from user where id > ?"
    rows, err := db.Query(sqlStr, 0)
    if err != nil {
        fmt.Printf("query failed, err:%v\n", err)
        return
    }
    // 非常重要：关闭rows释放持有的数据库链接
    defer rows.Close()

    // 循环读取结果集中的数据
    for rows.Next() {
        var u user
        err := rows.Scan(&u.id, &u.name, &u.age)
        if err != nil {
            fmt.Printf("scan failed, err:%v\n", err)
            return
        }
```

```
        fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name,
u.age)
    }
}
```

插入数据

插入、更新和删除操作都使用`Exec`方法。

```
func (db *DB) Exec(query string, args ...interface{}) (Result,
error)
```

Exec执行一次命令（包括查询、删除、更新、插入等），返回的Result是对已执行的SQL命令的总结。参数args表示query中的占位参数。

具体插入数据示例代码如下：

```
// 插入数据
func insertRowDemo() {
    sqlStr := "insert into user(name, age) values (?,?)"
    ret, err := db.Exec(sqlStr, "王五", 38)
    if err ≠ nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    theID, err := ret.LastInsertId() // 新插入数据的id
    if err ≠ nil {
        fmt.Printf("get lastinsert ID failed, err:%v\n", err)
        return
    }
    fmt.Printf("insert success, the id is %d.\n", theID)
}
```

更新数据

具体更新数据示例代码如下：

```
// 更新数据
func updateRowDemo() {
    sqlStr := "update user set age=? where id = ?"
    ret, err := db.Exec(sqlStr, 39, 3)
    if err ≠ nil {
        fmt.Printf("update failed, err:%v\n", err)
        return
    }
```

```
    n, err := ret.RowsAffected() // 操作影响的行数
    if err ≠ nil {
        fmt.Printf("get RowsAffected failed, err:%v\n", err)
        return
    }
    fmt.Printf("update success, affected rows:%d\n", n)
}
```

删除数据

具体删除数据的示例代码如下：

```
// 删除数据
func deleteRowDemo() {
    sqlStr := "delete from user where id = ?"
    ret, err := db.Exec(sqlStr, 3)
    if err ≠ nil {
        fmt.Printf("delete failed, err:%v\n", err)
        return
    }
    n, err := ret.RowsAffected() // 操作影响的行数
    if err ≠ nil {
        fmt.Printf("get RowsAffected failed, err:%v\n", err)
        return
    }
    fmt.Printf("delete success, affected rows:%d\n", n)
}
```

13.3 mysql预处理

优化MySQL服务器重复执行SQL的方法，可以提升服务器性能，提前让服务器编译，一次编译多次执行，节省后续编译的成本。

避免SQL注入问题。

database/sql中使用下面的Prepare`方法来实现预处理操作。

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare方法会先将sql语句发送给MySQL服务端，返回一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

查询操作的预处理示例代码如下：

```
// 预处理查询示例
func prepareQueryDemo() {
```

```
    sqlStr := "select id, name, age from user where id > ?"
    stmt, err := db.Prepare(sqlStr)
    if err ≠ nil {
        fmt.Printf("prepare failed, err:%v\n", err)
        return
    }
    defer stmt.Close()
    rows, err := stmt.Query(0)
    if err ≠ nil {
        fmt.Printf("query failed, err:%v\n", err)
        return
    }
    defer rows.Close()
    // 循环读取结果集中的数据
    for rows.Next() {
        var u user
        err := rows.Scan(&u.id, &u.name, &u.age)
        if err ≠ nil {
            fmt.Printf("scan failed, err:%v\n", err)
            return
        }
        fmt.Printf("id:%d name:%s age:%d\n", u.id, u.name,
u.age)
    }
}
```

插入、更新和删除操作的预处理十分类似，这里以插入操作的预处理为例：

```
// 预处理插入示例
func prepareInsertDemo() {
    sqlStr := "insert into user(name, age) values (?,?)"
    stmt, err := db.Prepare(sqlStr)
    if err ≠ nil {
        fmt.Printf("prepare failed, err:%v\n", err)
        return
    }
    defer stmt.Close()
    _, err = stmt.Exec("小王子", 18)
    if err ≠ nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    _, err = stmt.Exec("沙河娜扎", 18)
```

```
    if err != nil {
        fmt.Printf("insert failed, err:%v\n", err)
        return
    }
    fmt.Println("insert success.")
}
```

我们任何时候都不应该自己拼接SQL语句！

这里我们演示一个自行拼接SQL语句的示例，编写一个根据name字段查询user表的函数如下：

```
// sql注入示例
func sqlInjectDemo(name string) {
    sqlStr := fmt.Sprintf("select id, name, age from user where name='%s'", name)
    fmt.Printf("SQL:%s\n", sqlStr)
    var u user
    err := db.QueryRow(sqlStr).Scan(&u.id, &u.name, &u.age)
    if err != nil {
        fmt.Printf("exec failed, err:%v\n", err)
        return
    }
    fmt.Printf("user:%#v\n", u)
}
```

此时以下输入字符串都可以引发SQL注入问题：

```
sqlInjectDemo("xxx' or 1=1#")
sqlInjectDemo("xxx' union select * from user #")
sqlInjectDemo("xxx' and (select count(*) from user) <10 #")
```

补充：不同的数据库中，SQL语句使用的占位符语法不尽相同。

数据库	占位符语法
MySQL	?
PostgreSQL	$1, $2等
SQLite	? 和$1
Oracle	:name

13.4 Go实现MySQL事务

什么是事务?

事务：一个最小的不可再分的工作单元；通常一个事务对应一个完整的业务(例如银行账户转账业务，该业务就是一个最小的工作单元)，同时这个完整的业务需要执行多次的DML(insert、update、delete)语句共同联合完成。A转账给B，这里面就需要执行两次update操作。

在MySQL中只有使用了Innodb数据库引擎的数据库或表才支持事务。事务处理可以用来维护数据库的完整性，保证成批的SQL语句要么全部执行，要么全部不执行。

事务的ACID

通常事务必须满足4个条件（ACID）：原子性（Atomicity，或称不可分割性）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。

条件	解释
原子性	一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
一致性	在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
隔离性	数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
持久性	事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

事务相关方法

Go语言中使用以下三个方法实现MySQL中的事务操作。 开始事务

```
func (db *DB) Begin() (*Tx, error)
```

提交事务

```
func (tx *Tx) Commit() error
```

回滚事务

```
func (tx *Tx) Rollback() error
```

事务示例

下面的代码演示了一个简单的事务操作，该事物操作能够确保两次更新操作要么同时成功要么同时失败，不会存在中间状态。

```
// 事务操作示例
func transactionDemo() {
    tx, err := db.Begin() // 开启事务
    if err ≠ nil {
        if tx ≠ nil {
            tx.Rollback() // 回滚
        }
        fmt.Printf("begin trans failed, err:%v\n", err)
        return
    }
    sqlStr1 := "Update user set age=30 where id=?"
    ret1, err := tx.Exec(sqlStr1, 2)
    if err ≠ nil {
        tx.Rollback() // 回滚
        fmt.Printf("exec sql1 failed, err:%v\n", err)
        return
    }
    affRow1, err := ret1.RowsAffected()
    if err ≠ nil {
        tx.Rollback() // 回滚
        fmt.Printf("exec ret1.RowsAffected() failed, err:%v\n",
err)
        return
    }

    sqlStr2 := "Update user set age=40 where id=?"
    ret2, err := tx.Exec(sqlStr2, 3)
    if err ≠ nil {
        tx.Rollback() // 回滚
        fmt.Printf("exec sql2 failed, err:%v\n", err)
        return
    }
    affRow2, err := ret2.RowsAffected()
    if err ≠ nil {
        tx.Rollback() // 回滚
```

```
        fmt.Printf("exec ret1.RowsAffected() failed, err:%v\n",
err)
        return
    }

    fmt.Println(affRow1, affRow2)
    if affRow1 == 1 && affRow2 == 1 {
        fmt.Println("事务提交啦...")
        tx.Commit() // 提交事务
    } else {
        tx.Rollback()
        fmt.Println("事务回滚啦...")
    }

    fmt.Println("exec trans success!")
}
```