

1. 第一个go程序

hello.go

```
// 1.go语言以包作为管理单位  
// 2.每个文件必须先声明包  
// 3.程序必须有一个main包(重要)  
package main  
  
import "fmt" // 导入包后必须要使用, 否则编译不通过  
  
func main() { // 这个{"不能换行  
    fmt.Println("Hello World!") // 结尾不需要";"  
}
```

2. 数据类型

2.1 变量声明

```
// 格式: var 变量名 类型, 变量声明了 必须要使用  
var a int  
// 可以同时声明多个变量  
var b, c int
```

2.2 变量初始化

```
// 声明的同时赋值  
var a int = 10  
// 自动推导类型, 必须初始化, 通过初始化的值确定类型 (常用)  
c := 30  
fmt.Printf("c type is %T\n", c) // 输出: c type is int
```

2.3 自动推导类型和赋值的区别

```
var a int
a = 10
a = 20
a = 30 // 赋值可以使用n次

b := 20
//b := 30 自动推导只能使用一次
```

2.4 多重赋值和匿名变量

```
// 多重赋值
a, b := 10, 20
// 交换两个变量的值
i, j := 10, 20
i, j = j, i

// _ 匿名变量，丢弃数据不处理，匿名变量配合函数返回值使用才有优势
var c, d, e int
c, d, e = test() // go可以返回多个值
_, d, _ = test() // 只返回d
```

2.5 常量的使用

```
const b = 20 // 不能使用:= 可以省略类型
```

2.6 多个变量或常量的定义

```
// 多个变量
var (
    a int = 10 // 可以省略类型
    b float64 = 2.0
)
// 多个常量
const (
    i int = 10 // 可以省略类型
    j float64 = 3.14
)
```

2.7 iota枚举

```
// 1.iota常量自动生成器，每隔一行，自动累加1
// 2.iota给常量赋值使用
const (
    a = iota
    b = iota
    c = iota
)
// 3.iota遇到const，重置为0
const d = iota // d为0
// 4.可以只写一个iota
const (
    a1 = iota // 0
    b1
    c1
)
// 5.如果是同一行，值都一样
const (
    i = iota // 值为0
    j1, j2, j3 = iota, iota, iota // 值都为1
    k = iota // 值为2
)
```

2.8 基础数据类型

2.8.1 字符类型

```
// go语言声明字符类型用byte
var ch byte
```

2.8.2 字符串类型

```
// 字符串都是隐藏了一个结束符，'\0'
var str string
str = "a" // 由'a'和'\0'组成了一个字符串
str = "hello go"
// 只想操作字符串的某个字符，从0开始操作
fmt.Printf("str[0] = %c, str[1] = %c\n", str[0], str[1]) // 输出h,e
```

2.9 变量的输入

```
var a int
fmt.Printf("请输入变量a: ")

// 阻塞等待用户输入
// fmt.Scanf("%d", &a) // 别忘了&
fmt.Scan(&a)
fmt.Println("a = ", a)
```

2.10 类型转换

```
var ch byte
ch = 'a' // 字符串类型本质上就是整型
var t int
t = int(ch) // 类型转换，把ch的值取出来后，转成int再给t赋值
fmt.Println("t = ", t)
```

2.11 类型别名

```
// 给int64起一个别名叫bigint
type bigint int64

var a bigint // 等价于var a int64

type(
    long int64
    char byte
)
var a long = 11
var ch char = 'a'
fmt.Printf("b = %d, ch = %c\n", b, ch)
```

3. 流程控制

3.1 if使用

```
s := "王思聪"
if s == "王思聪" { // if后面不需要括号
}

// if支持1个初始化语句， 初始化语句和判断条件以分号分隔
if a := 10; a == 10 {
}
```

3.2 switch使用

```
num := 1
switch num { // 不需要括号 支持一个默认初始化语句 switch num := 1; num {
    case 1:
        fmt.Println("1楼")
        break // 可以不写break, 默认包含了
    case 2, 3, 4:
        fmt.Println("2楼")
        fallthrough // 不跳出switch语句，后面的无条件执行
    default:
        fmt.Println("x楼")
}

// 另类写法
score := 85
switch { // 可以没有条件
    case score > 90: // case后面可以放条件
        fmt.Println("优秀")
    case score > 80:
        fmt.Println("良好")
    default:
        fmt.Println("其它")
}
```

3.3 for使用

```
sum := 0
for i := 1; i <= 100 i++ { // 不需要括号
    sum = sum + i
}
```

3.4 range使用

```
str := "abc"
// 通过for打印每个字符
for i := 0; i < len(str); i++ {
    fmt.Println("str[%d]=%c\n", i, str[i])
}

// 迭代打印每个元素， 默认返回2个值：一个是元素的位置，一个是元素本身
for i, data := range str {
    fmt.Println("str[%d]=%c\n", i, data)
}

// 简洁写法
for i := range str {
    fmt.Println("str[%d]=%c\n", i, str[i])
}

// 等价于
for i, _ := range str {
    fmt.Println("str[%d]=%c\n", i, str[i])
}
```

3.5 break和continue的区别

```
package main

import "fmt"
import "time"

func main() {
    i := 0

    for { // for后面不写任何东西， 表示死循环
        i++
        time.Sleep(time.Second) // 延时1s
        if i == 5 {
            break // 跳出循环 如果嵌套了多个循环 跳出最近的那个内循环
            // continue
        }
        fmt.Println("i = ", i)
    }
}
```

3.6 goto的使用

```
// goto可以用在任何地方，但是不能跨函数使用
fmt.Println("1")

goto End // goto是关键字 End是自定义名字

fmt.Println("2")

End:
fmt.Println("3")
```

4. 函数

4.1 有参无返回值函数

```
func MyFunc01(a int) {
    fmt.Println("a= ", a)
}

func MyFunc02(a int, b int) {
    fmt.Println("a= %d, b = %d", a, b)
}

// 简洁写法
func MyFunc03(a, b int) {
    fmt.Println("a= %d, b = %d", a, b)
}

// 如果每个类型都不同，需要写明类型
func MyFunc04(a int, b string, c float64) {
```

4.2 不定参数类型

```
func MyFunc02(args ...int) {
    fmt.Println("len(args) = ", len(args))
    for i := 0; i < len(args); i++ {
        fmt.Println("args[%d] = %d\n", i, args[i])
    }
}

// 返回2个值，第一个是下标，第二个是下标所对应的数
```

```

for i, data := range args {
    fmt.Println("args[%d] = %d\n", i, data)
}
}

func main() {
    MyFunc02()
    MyFunc02(1)
    MyFunc02(1, 2, 3)
}

func MyFun3(a int, args ...int) { // 不定参数一定要放在最后一个参数
}

```

4.3 不定参数传递

```

func test(args ...int) {
    // 全部元素传递给myfunc
    // MyFunc02(args...)

    // 只想2个参数传递给另外一个函数使用
    MyFunc02(args[:2]...) // 传递前2个参数
    MyFunc02(args[2:]...) // 传递后2个参数
}

func main() {
    test(1, 2, 3, 4)
}

```

4.4 一个返回值

```

// 无参有返回值：只有一个返回值
func myFunc01() int {
    return 666
}

// 给返回值取一个变量名，go推荐写法
func myFunc02() (result int) {
    result = 666
    return
}

```

```
func main() {
    // 无参有返回值调用
    var a int
    a = myFunc01()
    fmt.Println("a = ", a)

    b := myFunc01()
    fmt.Println("b = ", b)
}
```

4.5 多个返回值

```
// 多个返回值
func myfunc01(int, int, int) {
    return 1, 2, 3
}

// go官方推荐写法
func myfunc02(a int, b int, c int) { // 或者func myfunc02(a, b, c int)
    a, b, c = 111, 222, 333
    return
}

func main() {
    // 函数调用
    a, b, c := myfunc02()
    fmt.Println("a = %d, b = %d, c = %d\n", a, b, c)
}
```

4.6 有参有返回

```
func MaxAndMin(a, b int) (max, min int) {
    if a > b {
        max = a
        min = b
    } else {
        max = b
        min = a
    }
    return
}
```

```
func main() {
    max, min := MaxAndMin(10, 20)
    fmt.Println("max = %d, min = %d\n", max, min)

    // 通过匿名函数变量丢弃某个返回值
    a, _ := MaxAndMin(10, 20)
    fmt.Println("a = %d\n", a)
}
```

4.7 函数类型

```
// 函数也是一种数据类型，通过type给一个函数类型起名
// FuncType它是一个函数类型
type FuncType func(int, int) int // 没有函数名字，没有{}

func main() {
    var result int
    result = Add(1, 1) // 传统方式调用
    fmt.Println("result = ", result)

    // 声明一个函数类型的变量，变量名叫fTest
    var fTest FuncType
    fTest = Add // 是变量就可以赋值
    result = fTest(10, 20) // 等价于Add(10, 20)
}
```

4.8 回调函数

```
func Add(a, b int) int {
    return a + b
}

func Minus(a, b int) int {
    return a - b
}

// 回调函数，函数有一个参数是函数类型，这个函数就是回调函数
// 计算器，可以进行四则运算
// 先有想法，后面再实现功能
func Calc(a, b int, fTest FuncType) (result int) {
    fmt.Println("Calc")
```

```
result = fTest(a, b) // 这个函数还没有实现
// result = Add(a, b) // Add()必须先定义后，才能实现
return
}

func main() {
    a := Calc(1, 1, Add)
    b := Calc(1, 1, Minus)
}
```

4.9 匿名函数和闭包

```
func main() {
    a := 10
    str := "mike"

    // 匿名函数，没有函数名字，函数定义，还没有调用
    f1 := func() { // := 自动推导类型
        fmt.Println("a = ", a)
        fmt.Println("str = ", str)
    }

    f1()

    // -----
    // 给一个函数类型起别名（这种写法比较少用）
    type FuncType func() // 函数没有参数，没有返回值
    // 声明变量
    var f2 FuncType
    f2 = f1
    f2()

    // -----
    // 定义匿名函数，同时调用
    func() {
        fmt.Println("a = %d, str = %s\n", a, str)
    }()
    // 后面的()表示调用此匿名函数

    // -----
    // 带参数的匿名函数
    f3 := func(i, j int) {
```

```
    fmt.Println("i = %d, j = %d\n", i, j)
}

f3(1, 2)

// -----



// 定义匿名函数，同时调用
func() {
    fmt.Println("a = %d, str = %s\n", a, str)
}() // 后面的()表示调用此匿名函数

// -----



// 定义匿名函数，同时调用
func(i, j int) {
    fmt.Println("a = %d, str = %s\n", a, str)
}(10, 20)

// -----



// 匿名函数，有参有返回值
x, y := func(i, j int) (max, min int) {
    if i > j {
        max = i
        min = j
    } else {
        max = j
        min = i
    }
    return
}(10, 20)
fmt.Println("x = %d, y = %d\n", x, y)
}
```

4.10 闭包捕获外部变量的特点

```
func main() {
    a := 10
    str := "mike"

    func() {
        // 闭包以引用方式捕获外部变量
        a = 666
        str = "go"
        fmt.Println("内部: a = %d, str = %s\n", a, str) // 输出666, go
    }()
}

fmt.Println("外部: a = %d, str = %s\n", a, str) // 输出666, go
```

4.11 闭包的特点

```
// 函数的返回值是一个匿名函数，返回一个函数类型
func test02() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}

func main() {
    // 返回值为一个匿名函数，返回一个函数类型，通过f来调用返回的匿名函数，f来调用闭包函数
    // 它不关心这些捕获了的变量和常量是否已经超出了作用域
    // 所以只要闭包还在使用它，这些变量就还会存在
    f := test02()
    fmt.Println(f()) // 1
    fmt.Println(f()) // 4
    fmt.Println(f()) // 9
    fmt.Println(f()) // 16
    fmt.Println(f()) // 25
}
```

4.12 defer的使用

```
func main() {
    // defer延迟使用, main函数结束前调用
    defer fmt.Println("bbbbbb")
    fmt.Println("aaaaaa") // 先输出aaaaaa
}
```

4.13 多个defer执行顺序

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。哪怕函数或某个延迟调用发生错误，这些调用依旧会执行

```
func test(x int) {
    fmt.Println(100 / x)
}

func main() {
    defer fmt.Println("aaa")
    defer fmt.Println("bbb")

    defer test(0)

    defer fmt.Println("ccc")
}

// 运行结果
ccc
bbb
aaa
```

4.14 defer和匿名函数结合使用

```
func main() {
    a := 10
    b := 20

    defer func() {
        fmt.Printf("a = %d, b = %d\n", a, b)
    }()
}

a = 111
```

```
b = 222
fmt.Println("外部: a = %d, b = %d\n", a, b)
}
```

输出:

```
a = 111, b = 222
a = 111, b = 222
```

```
func main() {
    a := 10
    b := 20

    defer func(a, b int) {
        fmt.Printf("a = %d, b = %d\n", a, b)
    }(a, b) // ()代表调用此匿名函数，把参数传递过去，已经先传递参数，只是没有调用

    a = 111
    b = 222
    fmt.Println("外部: a = %d, b = %d\n", a, b)
}
```

输出:

```
a = 111, b = 222
a = 10, b = 20
```

5. 获取命令行参数

```
package main

import "fmt"
import "os"

func main() {
    // 接收用户传递的参数，都是以字符串方式传递
    list := os.Args

    n := len(list)
    fmt.Println("n = ", n)

    for i := 0; i < n; i++ {
```

```
    fmt.Println("list[%d] = %s\n", i, list[i])  
}  
  
for i, data := range list {  
    fmt.Println("list[%d] = %s\n", i, data)  
}  
}
```

6. 变量

6.1 全局变量

```
package main  
  
import "fmt"  
  
func test() {  
    fmt.Println("test = a", a)  
}  
  
var a int
```

```
func main() {  
    a = 10  
    fmt.Println("a = ", a)  
  
    test()  
}
```

输出：

```
a = 10  
test a = 10
```

6.2 不同作用域同名变量

```
package main

import "fmt"

var a byte // 全局变量

func main() {
    var a int // 局部变量

    // 1、不同作用域，允许定义同名变量
    // 2、使用变量的原则，就近原则
    fmt.Println("%T\n", a)
}
```

7. 工程管理

1. 通过go env查看go相关的环境路径
2. 同一个目录包名必须一样，不同目录包名不一样
3. 设置GOPATH环境变量
4. 同一个目录，调用别的文件的函数，直接调用即可，无需包名引用
5. 调用不同包的，格式：包名.函数名
6. 调用别的包的函数，函数名必须大写

7.1 main函数和init函数

先执行导入文件的init函数，然后执行main函数所在文件的init函数，最后执行main函数

7.2 _操作

```
import (
    _ "fmt" // 表示导入包，而不是直接使用包里面的函数，而是调用了该包里面的init函数
)
```

7.3 pkg和bin目录手动生成

配置GOBIN环境变量，指向src同级bin目录，进入src目录用go install命令即可生成

bin: 存放可执行程序

pkg: 放平台相关的库

8. 复合数据类型

8.1 指针

变量的内存和变量的地址

```
func main() {
    var a int = 10
    // 每个变量有2层含义：变量的内存，变量的地址
    fmt.Printf("a = %d\n", a) // 变量的内存
    fmt.Printf("&a = %v\n", &a) // 变量的地址

    // 内存：====>教室
    // &a: ===》教室外面的门牌号

    // 保存某个变量的地址，需要指针类型 *int 保存int的地址， **int保存*int地址
    var p *int
    p = &a

    *p = 666 // *p操作的不是p的内存，是p所指向的内存（就是a）
    fmt.Printf("*p = %v, a = %v\n", *p, a)
}
```

8.1.1 不要操作没有合法指向的内存

```
func main() {
    var p *int
    p = nil
    fmt.Println("p = ", p)

    *p = 666 // err，因为p没有合法指向
}
```

8.1.2 new函数的使用

```
func main() {
    var p *int
    p = new(int)
    *p = 666
    fmt.Println("*p = ", *p)

    q := new(int) // 自动推导类型
    *q = 777
    fmt.Println("*q = ", *q)
}
```

8.1.3 指针做函数参数

```
func swap(p1, p2 *int) {
    *p1, *p2 = *p2, *p1
}

func main() {
    a, b := 10, 20
    // 通过一个函数交换a和b的内容
    swap(&a, &b) // 地址传递
    fmt.Printf("main: a = %d, b = %d\n", a, b)
}
```

8.2 数组

```
func main() {
    var id [50]int
    for i := 0; i < len(id); i++ {
        fmt.Printf("id[%d] = %d\n", i, id[i])
    }

    for i, data := range id {
        fmt.Printf("a[%d] = %d\n", i, data)
    }

    // 指定某个元素初始化 下标为2的值是10 下标为4的值是20 其它为0
    d := [5]int{2: 10, 4: 20}

    // 二维数组指定元素初始化
    e := [3][4]int{{5, 6, 7, 8}}
```

```
// 支持比较, 只支持 == 或 !=, 比较是不是每个元素都一样, 2个数组比较, 数组类型要一样
a := [5]int{1, 2, 3, 4, 5}
b := [5]int{1, 2, 3, 4, 5}
c := [5]int{1, 2, 3}
fmt.Println("a == b", a == b)
fmt.Println("a == c", a == c)

// 同类型的数组可以赋值
var d [5]int
d = a
fmt.Println("d = ", d)
}
```

8.2.1 数组做函数参数

```
// 数组做函数参数, 它是值传递
// 实参数组的每个元素给形参数组拷贝一份
func modify(a [5]int) {
    a[0] = 666
    fmt.Println("modify a = ", a)
}

func main() {
    a := [5]int{1, 2, 3, 4, 5} // 初始化

    modify(a) // 数组传递过去
    fmt.Println("main: a = ", a)
}
```

8.2.2 数组指针做函数参数

```
// p指向实参数组a， 它是指向数组，它是数组指针
// *p代表指针所指向的内存，就是实参a
func modify(p *[5]int) {
    (*p)[0] = 666
    fmt.Println("modify *a = ", *p)
}

func main() {
    a := [5]int{1, 2, 3, 4, 5} // 初始化

    modify(&a) // 地址传递
    fmt.Println("main: a = ", a)
}
```

8.3 随机数的使用

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func main() {
    // 设置种子，只需一次
    // 如果种子参数一样，每次运行程序产生的随机数都一样
    rand.Seed(time.Now().UnixNano()) // 以当前系统时间作为种子参数
    //fmt.Println("rand = ", rand.Int()) // 随机很大的数
    fmt.Println("rand = ", rand.Intn(100)) // 限制在100以内的数
}
```

8.4 切片

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，以实现变长方案

8.4.1 切片的容量和长度

```
func main() {
    a := []int{1, 2, 3, 0, 0}
    s := a[0:3:5]
    fmt.Println("s = ", s)
    fmt.Println("len(s) = ", len(s) // 长度 3-0
    fmt.Println("cap(s) = ", cap(s) // 容量 5-0
}
```

```
package main

import "fmt"

func main() {
    // 切片
    s := []int{}
    fmt.Printf("len = %d, cap = %d\n", len(s), cap(s))

    s = append(s, 11)
    fmt.Printf("len = %d, cap = %d\n", len(s), cap(s))
}
```

输出：

```
len = 0, cap = 0
len = 1, cap = 1
```

8.4.2 切片的创建

```
package main

import "fmt"

func main() {
    // 自动推导类型，同时初始化
    s1 := []int{1, 2, 3, 4}
    fmt.Println("s1 = ", s1)

    // 借助make函数，格式make(切片类型, len, cap)
    s2 := make([]int, 5, 10)
    fmt.Printf("len = %d, cap = %d\n", len(s2), cap(s2))
```

```
// 没有指定容量，容量和长度一样
s3 := make([]int, 5)
fmt.Printf("len = %d, cap = %d\n", len(s3), cap(s3))
}
```

8.4.3 切片的截取

```
package main

import "fmt"

func main() {
    array := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    // [low:high:max] 取下标从low开始的元素, len = high - low, cap = max - low
    s1 := array[:] // [0:len(array):len(array)] 不指定容量和长度
    fmt.Println("s1 = ", s1)
    fmt.Printf("len = %d, cap = %d\n", len(s1), cap(s1)) // 输出10 10

    // 操作某个元素, 和数组操作方式一样
    data := array[1]
    fmt.Println("data = ", data)

    s2 := array[3:6] // a[3] a[4] a[5] len = 6 - 3 = 3, cap = 7 - 3 = 4
    fmt.Println("s2 = ", s2) // s2 = [3 4 5]
    fmt.Printf("len = %d, cap = %d\n", len(s2), cap(s2)) // len = 3, cap = 4

    s3 := array[:6] // 从0开始, 取6个元素, 容量10
    fmt.Println("s3 = ", s3) // s3 = [0 1 2 3 4 5]
    fmt.Printf("len = %d, cap = %d\n", len(s3), cap(s3)) // len = 6, cap = 10

    s4 := array[3:] // 从下标3开始 到结尾
    fmt.Println("s4 = ", s4) // s4 = [3 4 5 6 7 8 9]
    fmt.Printf("len = %d, cap = %d\n", len(s4), cap(s4)) // len = 7, cap = 7
}
```

8.4.4 切片与数组的关系

```
package main

import "fmt"

func main() {
    a := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```

// 新切片
s1 := a[2:5] // 从a[2]开始, 取3个元素
s1[1] = 666
fmt.Println("s1 = ", s1) // s1 = [2 666 4]
fmt.Println("a = ", a) // a = [0 1 2 666 4 5 6 7 8 9]

// 另外新切片
s2 := s1[2:7] // 此时指针已经移动到了下标为2的位置, 新的起始位置为2
s2[2] = 777
fmt.Println("s2 = ", s2) // s2 = [4 5 777 7 8]
fmt.Println("a = ", a) // a = [0 1 2 666 4 5 777 7 8 9]
}

```

8.5 map

8.5.1 map的基本使用

```

package main

import "fmt"

func main() {
    // 定义一个map
    var m map[int]string
    fmt.Println("m = ", m)
    // 对于map只有len, 没有cap
    fmt.Println("len = ", len(m))

    // 可以通过make创建
    m1 := make(map[int]string)
    fmt.Println("m1 = ", m1)
    fmt.Println("len = ", len(m1))

    // 可以通过make创建, 可以指定长度, 只是指定了容量, 但是里面一个数据也没有
    m2 := make(map[int]string, 2) // 自动扩容
    m2[1] = "java"
    m2[2] = "go"
    m2[3] = "c++"

    fmt.Println("m2 = ", m2)
    fmt.Println("len = ", len(m2))
}

```

```
// 初始化赋值
m3 := map[int]string{1: "java", 2: "go", 3: "c++"}
fmt.Println("m3 = ", m3)
}
```

8.5.2 map的遍历

```
package main

import "fmt"

func main() {
    m := map[int]string{1: "java", 2: "go", 3: "c++"}
    for key, value := range m {
        fmt.Printf("%d=====>%s\n", key, value)
    }

    // 判断一个key值是否存在
    // 第一个返回值为key所对应的value， 第二个返回值为key是否存在的条件， 存在ok为true
    value, ok := m[1]
    if ok == true {
        fmt.Println("m[1] = ", value)
    } else {
        fmt.Println("key不存在")
    }
}
```

8.5.3 map的删除

```
package main

import "fmt"

func main() {
    m := map[int]string{1: "java", 2: "go", 3: "c++"}
    delete(m, 1) // 删除key为1的内容
    fmt.Println("m = ", m)
}
```

8.5.4 map做函数参数

```
package main

import "fmt"

func test(m map[int]string) {
    delete(m, 1)
}

func main() {
    m := map[int]string{1: "java", 2: "go", 3: "c++"}

    test(m) // 在函数内部删除某个key

    fmt.Println("m = ", m)
}
```

8.6 结构体

结构体是值类型

8.6.1 结构体普通变量初始化

```
package main

import "fmt"

// 定义一个结构体类型
type Student struct {
    id int
    name string
    sex byte // 字符类型
    age int
    addr string
}

func main() {
    // 顺序初始化，每个成员必须初始化
    var s1 Student = Student{1, "jeffrey", 'm', 18, "深圳"}
    fmt.Println("s1 = ", s1)

    // 指定成员初始化，没有初始化的成员，自动赋值为0
```

```
s2 := Student{id: 1, name: "jeff"}  
fmt.Println("s2 = ", s2)  
}
```

8.6.2 结构体指针变量初始化

```
package main  
  
import "fmt"  
  
// 定义一个结构体类型  
type Student struct {  
    id int  
    name string  
    sex byte // 字符类型  
    age int  
    addr string  
}  
  
func main() {  
    // 顺序初始化， 每个成员必须初始化  
    var p1 *Student = &Student{id: 1, name: "jeffrey", 'm', 18, "深圳"}  
    fmt.Println("p1 = ", p1)  
  
    // 指定成员初始化， 没有初始化的成员， 自动赋值为0  
    p2 := &Student{id: 1, name: "jeff"}  
    fmt.Printf("p2 type is %T\n", p2)  
    fmt.Println("p2 = ", p2)  
}
```

8.6.3 结构体成员的使用：普通变量

```
package main  
  
import "fmt"  
  
// 定义一个结构体类型  
type Student struct {  
    id int  
    name string  
    sex byte // 字符类型  
    age int  
    addr string
```

```
}

func main() {
    // 定义一个结构体
    var s Student

    // 操作成员，需要使用点（.）运算符
    s.id = 1
    s.name = "jack"
    s.sex = 'm'
    s.age = 20
    s.addr = "bj"
    fmt.Println("s = ", s)
}
```

8.6.3 结构体成员的使用：指针变量

```
package main

import "fmt"

// 定义一个结构体类型
type Student struct {
    id int
    name string
    sex byte // 字符类型
    age int
    addr string
}

func main() {
    // 1、指针有合法指向后，才操作成员
    // 先定义一个普通结构体变量
    var s Student
    // 再定义一个指针变量，保存s的地址
    var p1 *Student
    p1 = &s

    // 通过指针操作成员，p1.id和(*p1).id完全等价，只能使用.运算符
    p1.id = 1
    (*p1).name = "mike"
    p1.age = 19
    p1.addr = "bj"
```

```
fmt.Println("p1 = ", p1)

// 2、通过new申请一个结构体
p2 := new(Student)
p2.id = 1
(*p2).name = "mike"
p2.age = 19
p2.addr = "bj"
fmt.Println("p2 = ", p2)
}
```

8.6.4 结构体比较和赋值

```
package main

import "fmt"

// 定义一个结构体类型
type Student struct {
    id int
    name string
    sex byte // 字符类型
    age int
    addr string
}

func main() {
    s1 := Student{1, "mike", 'm', 18, "bj"}
    s2 := Student{1, "mike", 'm', 18, "bj"}
    s3 := Student{2, "mike", 'm', 18, "bj"}
    fmt.Println("s1 == s2 ", s1 == s2)
    fmt.Println("s1 == s2 ", s1 == s3)

    // 同类型的2个结构体变量可以相互转换
    var temp Student
    temp = s3
    fmt.Println("temp = ", temp)
}
```

8.6.5 结构体做函数参数：值传递

```
package main

import "fmt"

// 定义一个结构体类型
type Student struct {
    id int
    name string
    sex byte // 字符类型
    age int
    addr string
}

func test01(s Student) {
    s.id = 16
    fmt.Println("test01: ", s)
}

func main() {
    s := Student{1, "mike", 'm', 19, "bj"}

    test01(s) // 值传递，形参无法改实参
    fmt.Println("main: ", s)
}
```

8.6.6 结构体做函数参数：地址传递

```
package main

import "fmt"

// 定义一个结构体类型
type Student struct {
    id int
    name string
    sex byte // 字符类型
    age int
    addr string
}

func test01(p *Student) {
```

```
p.id = 16
fmt.Println("test01: ", p)
}

func main() {
    s := Student{1, "mike", 'm', 19, "bj"}

    test01(&s) // 地址传递(引用传递), 形参可以改实参
    fmt.Println("main: ", s)
}
```

8.6.7 结构体内数据默认值和使用陷阱

```
type Student struct {
    ptr *int // 默认为nil, 使用时需要new
    slice []int // 默认为nil
    map1 map[string]string // 默认为nil
}

// 使用slice需要先make分配空间
var s Student
s.slice = make([]int, 10)
s.slice[0] = 100

// 使用map, 一定要先make
s.map1 = make(map[string]string)
s.map1["key1"] = "tom"
```

8.6.8 结构体创建的四种方式

```
// 第一种 直接声明
var person Person

// 第二种 {}
// 推荐使用这种
var person Person = Person{"mary", 18}
// person.Name = tom
// person.Age = 18

// 第三种
var person *Person = new(Person)
// 因为person是个指针, 因此标准的给字段赋值方式
```

```
(*person).Name = "smith" // 也可以这样写 person.Name = "smith", 原因: go的设计者为了程序员  
使用方便, 底层会对person.Name = "smith"进程处理, 会给person加上取值运算(*person).Name =  
"smith"  
(*person).Age = 30  
  
// 第四种  
var person *Person = &Person{} // 有初始值 = &Person{"mary", 60}  
// 因为person是个指针, 因此标准的给字段赋值方式  
(*person).Name = "smith"
```

8.6.9 结构体使用细节

案例一-结构体内存地址

```
package main  
  
import "fmt"  
  
type Point struct {  
    x int  
    y int  
}  
  
type Rect struct {  
    leftUp, rightDown Point  
}  
  
func main() {  
    r1 := Rect{Point{1, 2}, Point{3, 4}}  
  
    // r1有四个int, 在内存中是连续分布  
    // 打印地址  
    fmt.Printf("r1.leftUp.x 地址=%p r1.leftUp.y 地址=%p "+  
              "r1.rightDown.x 地址=%p r1.rightDown.y 地址=%p", &r1.leftUp.x, &r1.leftUp.y,  
              &r1.rightDown.x, &r1.rightDown.y)  
}
```

案例二-结构体内存地址

```
package main  
  
import "fmt"
```

```

type Point struct {
    x int
    y int
}

type Rect struct {
    leftUp, rightDown *Point
}

func main() {
    // r1有两个 *Point类型，这个两个*Point类型的本身地址也是连续的，但是他们指向的地址不一定连续
    r1 := Rect{&Point{1, 2}, &Point{3, 4}}

    // 打印地址
    fmt.Printf("r1.leftUp 本身地址=%p r1.rightDown 本身地址=%p\n", &r1.leftUp,
    &r1.rightDown)
    fmt.Printf("r1.leftUp 地址=%p r1.rightDown 地址=%p", r1.leftUp, r1.rightDown)
}

```

案例三-结构体转换

```

package main

import "fmt"

type A struct {
    Num int
}

type B struct {
    Num int
}

func main() {
    var a A
    var b B
    a = A(b) // 可以转换 要求结构体字段(名字、个数和类型)完全一样
    fmt.Println(a, b)
}

```

案例四

案例五

struct每个字段上，都可以写上一个tag，该tag可以通过反射机制获取，常用的使用场景是序列化和反序列化

```
package main

import (
    "encoding/json"
    "fmt"
)

type Monster struct {
    Name  string `json:"name"`
    Age   int    `json:"age"`
    Skill string `json:"skill"`
}

func main() {
    monster := Monster{"牛魔王", 500, "牛气冲天"}

    jsonStr, _ := json.Marshal(monster)
    fmt.Println("jsonStr = ", string(jsonStr))
}
```

9. 并发

9.1 goroutine

goroutine说到底其实是协程，它比线程更小，十几个goroutine可能体现在底层就是五六个线程

1.主线程是一个物理线程，直接作用在cpu上。是重量级的，非常耗费cpu资源。

2.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。

3.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。

9.1.1 MPG模式基本介绍

M:操作系统的主线程（是物理线程）

P:协程执行需要的上下文

G:协程

9.2 主goroutine退出了，其它子协程也要跟着退出

```
package main

import (
    "fmt"
    "time"
)

// 主协程退出了，其它子协程也要跟着退出
func main() {

    go func() {
        i := 0
        for {
            i++
            fmt.Println("子协程 i = ", i)
            time.Sleep(time.Second)
        }
    }()
}

i := 0
for {
    i++
    fmt.Println("main i = ", i)
    time.Sleep(time.Second)

    if i == 2 {
        break
    }
}
}
```

9.3 runtime包

9.3.1 Gosched的使用

```
package main

import (
    "fmt"
    "runtime"
)

// 主协程退出了，其它子协程也要跟着退出
func main() {

    go func() {
        for i := 0; i < 5; i++ {
            fmt.Println("go")
        }
    }()
}

// 让出时间片，先让别的协程执行完，它执行完，再回来执行主协程？
runtime.Gosched()
for i := 0; i < 2; i++ {
    fmt.Println("hello")
}
}
```

9.3.2 Goexit终止协程

9.3.3 GOMAXPROCS的使用

```
n := runtime.GOMAXPROCS(5) // 指定以多少核数运行
```

9.4 channel

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

9.4.1 通过channel实现同步

```
// 全局变量，创建一个channel
var ch = make(chan int)

// person1执行完后，才能到person2执行
func person1() {
    Printer("hello")
    ch <- 666 // 给管道写数据，发送
}

func person2() {
    <-ch
    // 从管道取数据，接收，如果通道没有数据他就会阻塞
    Printer("world")
}
```

9.4.2 通过channel实现同步和数据交互

```
package main

import (
    "fmt"
    "time"
)

// 主协程退出了，其它子协程也要跟着退出
func main() {

    ch := make(chan string)
    defer fmt.Println("主协程也结束")

    go func() {
        defer fmt.Println("子协程调用完毕")

        for i := 0; i < 2; i++ {
            fmt.Println("子协程 i = ", i)
            time.Sleep(time.Second)
        }

        ch <- "我是子协程，工作完毕"
    }()
}
```

```
    str := <-ch // 没有数据前, 阻塞
    fmt.Println("str = ", str)
}
```

9.4.3 关闭channel

```
func main() {
    ch := make(chan int) // 创建一个无缓冲的channel

    // 新建一个goroutine
    go func() {
        for i:= 0; i < 5; i++ {
            ch <- i
        }
        // 不需要再写数据时, 关闭channel
        close(ch) // 关闭后无法再发送数据
    }()
}

for {
    // 如果ok为true, 说明管道没有关闭
    if num, ok := <-ch; ok == true {
        fmt.Println("num = ", num)
    } else {
        break
    }
}
// 另一种写法
for num := range ch {
    fmt.Println("num = ", num)
}
```

9.4.4 单向channel的特点

```
package main

func main() {
    // 创造一个channel, 双向的
    ch := make(chan int)

    // 双向channel能隐式转换为单向channel
    var writeCh chan<- int = ch // 只能写, 不能读
    var readCh <-chan int = ch // 只能读, 不能写
}
```

```
writeCh <- 666 // 写
//<- writeCh // err

<- readCh // 读
//readCh <- 666 // err

// 单向无法转换为双向
//var ch2 chan int = writeCh
}
```

9.4.5 单向channel的应用

```
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
    }
    close(out)
}

// 此channel只能读，不能写
func consumer(in <-chan int) {
    for num := range in {
        fmt.Println("num = ", num)
    }
}

func main() {
    // 创造一个channel，双向的
    ch := make(chan int)

    // 生产者，生产数字，写入channel
    // 新开一个协程
    go producer(ch) // channel传参，引用传递

    // 消费者，从channel读取内容，打印
    consumer(ch)
}
```

9.5 定时器

9.5.1 Timer的使用

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 创建一个定时器，设置时间为2s，2s后，往time通道写内容（当前时间）
    timer := time.NewTimer(2 * time.Second) // 时间到了，只会响应一次
    fmt.Println("当前时间：", time.Now())

    // 2s后，往timer.C写数据，有数据后，就可以读取
    t := <-timer.C
    fmt.Println("t = ", t)
}
```

9.5.2 通过Timer实现延时功能

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 第1种写法
    // 延时2s后打印一句话
    timer := time.NewTimer(2 * time.Second)
    <-timer.C
    fmt.Println("时间到")

    // 第2种写法
    time.Sleep(2 * time.Second)
    fmt.Println("时间到")

    // 第3种写法
    <-time.After(2 * time.Second)
```

```
    fmt.Println("时间到")
}
```

9.5.3 定时器停止和重置

```
package main

import (
    "fmt"
    "time"
)

func main() {
    timer := time.NewTimer(3 * time.Second)
    ok := timer.Reset(1 * time.Second) // 重新设置为1s
    fmt.Println("ok = ", ok)

    go func() {
        <-timer.C
        fmt.Println("子协程可以打印了，因为定时器的时间到")
    }()
}

timer.Stop()
}
```

9.5.4 Ticker的使用

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ticker := time.NewTicker(1 * time.Second)

    i := 0
    for {
        <-ticker.C

        i++
        fmt.Println("i = ", i)
    }
}
```

```
        if i == 5 {
            ticker.Stop()
            break
        }
    }
}
```

9.6 select

9.6.1 通过select实现fibonacci数列

```
package main

import "fmt"

// ch只写, quit只读
func fibonacci(ch chan<- int, quit <-chan bool) {
    x, y := 1, 1
    for {
        // 监听channel数据的流动
        select {
        case ch <- x:
            x, y = y, x+y
        case flag := <-quit:
            fmt.Println("flag = ", flag)
            return
        }
    }
}

func main() {
    ch := make(chan int)      // 数字通信
    quit := make(chan bool) // 程序是否结束

    // 消费者, 从channel读取内容
    // 新建协程
    go func() {
        for i := 0; i < 8; i++ {
            num := <-ch
            fmt.Println("num = ", num)
        }
        // 可以停止
    }
}
```

```
    quit <- true
}

// 生产者, 产生数字, 写入channel
fibonacci(ch, quit)
}
```

9.6.2 select实现的超时机制

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ch := make(chan int)
    quit := make(chan bool)

    // 新开一个协程
    go func() {
        for {
            select {
            case num := <-ch:
                fmt.Println("num = ", num)
            case <-time.After(3 * time.Second):
                fmt.Println("超时")
                quit <- true
            }
        }
    }()

    for i := 0; i < 5; i++ {
        ch <- i
        time.Sleep(time.Second)
    }

    quit <- true
    fmt.Println("程序退出")
}
```

10. 异常处理

10.1 error接口的使用

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    err := fmt.Errorf("%s", "this is normal err")
    fmt.Println("err = ", err)

    err1 := errors.New("this is normal err")
    fmt.Println("err1 = ", err1)
}
```

10.2 error接口的应用

```
package main

import (
    "errors"
    "fmt"
)

func MyDiv(a, b int) (result int, err error) {
    err = nil
    if b == 0 {
        err = errors.New("分母不能为空")
    } else {
        result = a / b
    }
    return
}

func main() {
    result, err := MyDiv(10, 0)
    if err != nil {
        fmt.Println("err = ", err)
    }
}
```

```
    } else {
        fmt.Println("result = ", result)
    }
}
```

10.3 panic函数

我们不应该通过调用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panic发生时，程序会中断运行。

10.3.1 显示调用panic函数

```
package main

import "fmt"

func testa() {
    fmt.Println("aaaaaa")
}

func testb() {
    //fmt.Println("bbbbbb")
    panic("this is a panic test")
}

func testc() {
    fmt.Println("cccccc")
}

func main() {
    testa()
    testb()
    testc()
}
```

10.4 recover

```
package main

import "fmt"

func testa() {
    fmt.Println("aaaaaa")
```

```
}

func testb(x int) {
    // 设置recover
    defer func() {
        // recover可以打印panic的错误信息
        // fmt.Println(recover())
        if err := recover(); err != nil { // 产生了panic异常
            fmt.Println(err)
        }
    }()
}

var a [10]int
a[x] = 111 // x大于等于10的时候，数组越界
}

func testc() {
    fmt.Println("cccc")
}

func main() {
    testa()
    testb(20)
    testc()
}
```

11. 文本文件处理

11.1 字符串处理

11.1.1 字符串操作

11.1.1.1 Contains

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // "hellogo"中是否包含"hello", 包含返回true, 不包含返回false
    fmt.Println(strings.Contains("hellogo", "go")) // 输出: true
}
```

11.1.1.2 Joins

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Joins组合
    s := []string{"abc", "hello", "jack", "go"}
    buf := strings.Join(s, "@")
    fmt.Println("buf = ", buf) // 输出: buf = abc@hello@jack@g
}
```

11.1.1.3 Index

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Index
    fmt.Println(strings.Index("abcdhello", "hello")) // 输出: 4
    fmt.Println(strings.Index("abcdhello", "go")) // 输出: -1
}
```

11.1.1.4 Repeat

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Repeat
    buf := strings.Repeat("go", 3)
    fmt.Println("buf = ", buf)
}
```

11.1.1.5 Index

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Split 以指定的分隔符拆分
    buf := "hello\abc\gol\jack"
    s := strings.Split(buf, "\\")

    fmt.Println("s = ", s)
}
```

11.1.1.6 Trim

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Trim去掉两头的字符
    buf := strings.Trim("  are u ok?  ", " ") // 去掉两头空格
    fmt.Printf("buf = #%s#\n", buf) // 输出: buf = #are u ok?#
}
```

11.1.1.7 Fields

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := strings.Fields("  are u ok?      ")
    for i, data := range s {
        fmt.Println(i, ":", data)
    }
}
```

11.1.2 字符串转换

11.1.2.1 Append

```
// 转换成字符串后追加到字节数组
slice := make([]byte, 0, 1024)
slice = strconv.AppendBool(slice, true)
// 第二个数为要追加的数, 第3个为指定10进制方式追加
slice = strconv.AppendInt(slice, 1234, 10)
slice = strconv.AppendQuote(slice, "abcgohello") // 双引号也带着
fmt.Println("slice = ", string(slice))
```

11.1.2.2 Format

```
// 其它类型转换为字符串
var str string
str = strconv.FormatBool(false)
// 'f' 指打印格式, 以小数方式, -1指小数点位数 (紧缩模式) , 64指以float64处理
str = strconv.FormatFloat(3.14, 'f', -1, 64)
fmt.Println("str = ", str)
```

11.1.2.3 Parse

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    // 整型转字符串 常用
    str := strconv.Itoa(666)
    fmt.Println("str = ", str)

    // 字符串转换为整型
    a, _ := strconv.Atoi("567")
    fmt.Println("a = ", a)

    // 字符串转其它类型
    flag, err := strconv.ParseBool("true")
    if err == nil {
        fmt.Println("flag = ", flag)
    } else {
        fmt.Println("err = ", err)
    }
}
```

12. 网络编程

12.1 TCP服务器代码的编写

```
package main

import (
    "fmt"
    "net"
)

func main() {
    // 监听
    listener, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        fmt.Println("err = ", err)
        return
    }

    defer listener.Close()

    // 阻塞等待用户链接
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("err = ", err)
            continue
        }

        // 接收用户的请求
        buf := make([]byte, 1024) // 1024大小的缓冲区
        n, err1 := conn.Read(buf) // n表示长度
        if err1 != nil {
            fmt.Println("err1 = ", err1)
            continue
        }

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

12.2 TCP客户端

```
package main

import (
    "fmt"
    "net"
)

func main() {
    // 主动连接服务器
    conn, err := net.Dial("tcp", "127.0.0.1:8000")
    if err != nil {
        fmt.Println("err = ", err)
        return
    }

    defer conn.Close()

    // 发送数据
    conn.Write([]byte("are u ok?"))
}
```

12.3 简单版并发服务器

```
package main

import (
    "fmt"
    "net"
    "strings"
)

// 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, " connect success")

    buf := make([]byte, 2048) // 2048大小的缓冲区
```

```
for {
    // 读取用户数据
    n, err := conn.Read(buf) // n表示长度
    if err != nil {
        fmt.Println("err = ", err)
        return
    }
    fmt.Printf("[%s]: = %s\n", addr, string(buf[:n]))

    if "exit" == string(buf[:n-1]) { // 发送时, windows会多了2个字符, "\r\n", mac会多1个
        // 字符
        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]))))
}
}

func main() {
    // 监听
    listener, err := net.Listen("tcp", "127.0.0.1:8000")
    if err != nil {
        fmt.Println("err = ", err)
        return
    }

    defer listener.Close()

    // 阻塞等待用户链接
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("err = ", err)
            continue
        }

        // 处理用户请求, 新建一个协程
        go HandleConn(conn)
    }
}
```

12.4 客户端可输入也可接收服务器回复

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    // 主动连接服务器
    conn, err := net.Dial("tcp", "127.0.0.1:8000")
    if err != nil {
        fmt.Println("err = ", err)
        return
    }

    defer conn.Close()

    go func() {
        // 从键盘输入内容，给服务器发送内容
        str := make([]byte, 1024)
        for {
            n, err := os.Stdin.Read(str) // 从键盘读取内容
            if err != nil {
                fmt.Println("os.Stdin. err = ", err)
                return
            }
            // 把输入的内容发送给服务器
            conn.Write(str[:n])
        }
    }()
}

// 接收服务器回复的数据，新任务
// 切片缓冲
buf := make([]byte, 1024)
for {
    n, err := conn.Read(buf) // 接收服务器的请求
    if err != nil {
        fmt.Println("conn.Read err = ", err)
        return
    }
```

```
    fmt.Println(string(buf[:n])) // 打印接收到的请求
}
}
```

12.5 并发聊天服务器

```
package main

import (
    "fmt"
    "net"
    "strings"
    "time"
)

type Client struct {
    C     chan string // 用户发送数据的管道
    Name string      // 用户名
    Addr string      // 网络地址
}

// 保存在线用户
var onlineMap map[string]Client

var message = make(chan string)

// 新开一个协程，转发消息，只要有消息来了，遍历map，给map每个成员都发送此消息
func Manager() {
    // 给map分配空间？
    onlineMap = make(map[string]Client)

    for {
        msg := <-message

        // 遍历map，给map每个成员都发送此消息
        for _, cli := range onlineMap {
            cli.C <- msg
        }
    }
}

func WriteMsgToClient(cli Client, conn net.Conn) {
    for msg := range cli.C { // 给当前客户端发送消息

```

```
conn.Write([]byte(msg + "\n"))
}

}

func MakeMsg(cli Client, msg string) (buf string) {
    buf = "[" + cli.Addr + "]" + cli.Name + ": " + msg
    return
}

func HandleConn(conn net.Conn) { // 处理用户连接
    defer conn.Close()

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

    // 创建一个结构体
    cli := Client{make(chan string), cliAddr, cliAddr}
    // 把结构体添加到map
    onlineMap[cliAddr] = cli

    // 新开一个协程，专门给当前客户端发送信息
    go WriteMsgToClient(cli, conn)
    // 广播某个在线
    message <- MakeMsg(cli, "login")
    // 提示，我是谁
    message <- MakeMsg(cli, "I am here")

    isQuit := make(chan bool) // 对方是否主动退出
    hasData := make(chan bool) // 对方是否有数据发送

    // 新建一个协程，接收用户发送过来的数据
    go func() {
        buf := make([]byte, 2048)
        for {
            n, err := conn.Read(buf)
            if n == 0 { // 对方断开，或者 出问题
                isQuit <- true
                fmt.Println("conn.Read err = ", err)
                return
            }

            msg := string(buf[:n])
            if len(msg) == 3 && msg == "who" { // 查询在线玩家
                // 遍历map 给当前用户发送所有成员
                conn.Write([]byte("user list:\n"))
            }
        }
    }
}
```

```
        for _, tmp := range onlineMap {
            msg = tmp.Addr + ":" + tmp.Name + "\n"
            conn.Write([]byte(msg))
        }
    } else if len(msg) >= 8 && msg[:6] == "rename" { // 修改当前用户的用户名
        name := strings.Split(msg, "|")[1]
        cli.Name = name
        onlineMap[cliAddr] = cli
        conn.Write([]byte("rename ok:\n"))
    } else {
        // 转发此内容
        message <- MakeMsg(cli, msg)
    }

    hasData <- true
}
}()

for {
    // 通过select检测channel的流动
    select {
        case <-isQuit:
            delete(onlineMap, cliAddr)           // 当前用户从map移除
            message <- MakeMsg(cli, "login out") // 广播谁下线了
            return
        case <-hasData:

            case <-time.After(60 * time.Second): // 60s后
                delete(onlineMap, cliAddr)           // 当前用户从map移除
                message <- MakeMsg(cli, "time out leave out") // 广播谁下线了
                return
            }
    }
}

func main() {
    // 监听
    listener, err := net.Listen("tcp", ":8000")
    if err != nil {
        fmt.Println("net.Listen err = ", err)
        return
    }

    defer listener.Close()
```

```
// 新开一个协程，转发消息，只要有消息来了，遍历map，给map每个成员都发送此消息
go Manager()

// 主协程，循环阻塞等待用户连接
for {
    conn, err := listener.Accept()
    if err != nil {
        fmt.Println("listener.Accept err = ", err)
        continue
    }

    // 处理用户请求，新建一个协程
    go HandleConn(conn)
}
}
```

13. 常用配置设置命令

13.1 下载包

```
go get -u github.com/gogo/protobuf/protoc-gen-gogo
```

参数介绍：

- d 只下载不安装
- f 只有在你包含了 -u 参数的时候才有效，不让 -u 去验证 import 中的每一个都已经获取了，这对于本地 fork 的包特别有用
- fix 在获取源码之后先运行 fix，然后再去做其他的事情
- t 同时也下载需要为运行测试所需要的包
- u 强制使用网络去更新包和它的依赖包
- v 显示执行的命令

13.2 设置环境参数

```
// 设置代理  
go env -w GOPROXY=https://goproxy.cn  
// 设置GOPATH路径  
go env -w GOPATH=E:\go_game_server
```

13.3 生成proto文件

```
// 普通protoc生成  
protoc --go_out=../. login.proto  
// 使用的是protoc-gen-go  
protoc --gogo_out=../. login.proto
```

14. 面向对象编程

14.1 封装

14.1.1 封装入门案例

person.go

```
package model  
  
import "fmt"  
  
type person struct {  
    Name string  
    age  int  
    sal   float64  
}  
  
// 写一个工厂模式的函数，相当于构造函数  
func NewPerson(name string) *person {  
    return &person{  
        Name: name,  
    }  
}  
  
// 为了访问age和sal，编写一堆SetXxx的方法和GetXxx的方法  
func (p *person) SetAge(age int) {
```

```
if age > 0 && age < 150 {
    p.age = age
} else {
    fmt.Println("年龄范围不正确...")
}
}

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

func (p *person) SetSal(sal float64) {
    if sal > 3000 && sal <= 30000 {
        p.sal = sal
    } else {
        fmt.Println("薪水范围不正确...")
    }
}

func (p *person) GetSal() float64 {
    return p.sal
}
```

test.go

```
package main

import (
    "fmt"
    "model"
)

func main() {
    p := model.NewPerson("smith")
    p.SetAge(19)
    p.SetSal(30000)
    fmt.Println(p)
    fmt.Println(p.Name, "age =", p.GetAge(), "sal =", p.GetSal())
}
```

14.2 继承

通过匿名结构体实现继承特性

14.2.1 继承快速入门案例

```
package main

import "fmt"

type Student struct {
    Name  string
    Age   int
    Score int
}

// 小学生
type Pupil struct {
    Student // 嵌入了Student匿名结构体
}

// 大学生
type Graduate struct {
    Student // 嵌入了Student匿名结构体
}

func (p *Pupil) testing() {
    fmt.Println("小学生正在考试中...")
}

func (p *Graduate) testing() {
    fmt.Println("大学生正在考试中...")
}

func main() {
    // 当我们对结构体嵌入了匿名结构体 使用方法会发生变化
    pupil := &Pupil{}
    pupil.Student.Name = "tom"
    pupil.Student.Age = 8
    pupil.testing()
}
```

14.2.2 继承的深入讨论

1. 结构体可以使用嵌套匿名结构体所有的字段和方法，即：首字母大写或小写的字段、方法，都可以使用

2. 匿名结构体字段访问可以简化

The diagram illustrates the simplification of code using anonymous structures. On the left, a code snippet shows a variable `b` of type `B` containing fields `A.name`, `A.Age`, `A.say()`, and `A.Hello()`. A large red arrow points to the right, where the same code is shown but with the fields directly accessed via `b.name`, `b.Age`, `b.say()`, and `b.Hello()`.

```
func main() {
    var b B
    b.A.name = "tom"
    b.A.Age = 78
    b.A.say()
    b.A.Hello()
}
```

```
func main() {
    var b B
    b.name = "tom"
    b.Age = 78
    b.say()
    b.Hello()
}
```

3. 当结构体和匿名结构体有相同的字段或者方法时，编译期采用“就近访问原则”访问，如希望访问匿名结构体的字段和方法，可以通过匿名结构体名来区分

This diagram shows code with annotations explaining scope resolution rules. It highlights specific lines with red boxes:

- `b.A.name = "jack"` is annotated with a red box.
- `b.A.Hello()` is annotated with a red box.

The code itself includes comments explaining the rules:

```
func main() {
    var b B
    b.name = "tom" // 这时就近原则，会访问B结构体的name字段
    // b.A.name 就明确指定访问A匿名结构体的字段name
    b.A.name = "jack"
    b.Age = 78
    b.say() // 这时就近原则，会访问B结构体的say函数
    b.Hello()
    // b.A.Hello() 就明确指定访问A匿名结构体的方法Hello()
    b.A.Hello()
}
```

4. 如果一个struct嵌套了一个有名称的结构体，这种模式就是“组合”，如果是组合关系，那么在访问组合的结构体的字段或方法时，必须带上结构体的名字

This diagram shows a composite struct definition where struct `C` contains a field of type `A`. To access the fields of `A` within `C`, the prefix `a.` must be used.

```
type A struct {
    Name string
    Age int
}
type C struct {
    a A
}
```

5.

```
// 嵌套匿名结构体后，也可以在创建结构体变量(实例)时，直接指定各个匿名结构体字段的值
tv := TV{ Goods{"电视机001", 5000.99}, Brand{"海尔", "山东"}, }
I
fmt.Println("tv", tv)
```

14.3.3 多重继承

```
type Goods struct {
    Name string
    Price float64
}

type Brand struct {
    Name string
    Address string
}

type TV struct {
    Goods
    Brand
}
```

14.3 接口

14.3.1 定义一个接口

```
type Usb interface {
    // 声明2个没有实现的方法
    Start()
    Stop()
}

type Phone struct {

}

// 让Phone实现Usb接口的方法
func (p Phone) Start() {
    fmt.Println("手机开始工作...")
}

// 让Phone实现Usb接口的方法
```

```
func (p Phone) Stop() {
    fmt.Println("手机开始工作...")
}

type Camera struct {

}

func (c Camera) Start() {
    fmt.Println("相机开始工作...")
}

func (c Camera) Stop() {
    fmt.Println("相机停止工作...")
}

type Computer struct {

}

func (c Computer) Working(usb Usb) {
    usb.Start()
    usb.Stop()
}

func main() {
    computer := Computer{}
    phone := Phone{}
    camera := Camera{}

    computer.Working(phone)
    computer.Working(camera)
}
```

14.3.2 接口的特点和语法说明

● 接口(interface)

基本介绍

interface类型可以定义一组方法，但是这些不需要实现。并且interface不能包含任何变量。到某个自定义类型(比如结构体Phone)要使用的时候，在根据具体情况把这些方法写出来(实现)。

基本语法

```
type 接口名 interface{  
    I  
        method1(参数列表) 返回值列表  
        method2(参数列表) 返回值列表  
        ...  
}
```

```
func (t 自定义类型) method1(参数列表) 返回值列表 {  
    //方法实现  
}  
func (t 自定义类型) method2(参数列表) 返回值列表 {  
    //方法实现  
}  
//....
```

小结说明：

- 1) 接口里的所有方法都没有方法体，即接口的方法都是没有实现的方法。接口体现了程序设计的**多态和高内聚低偶合**的思想。
- 2) Golang 中的接口，不需要显式的实现。只要一个变量，含有接口类型中的所有方法，那么这个变量就实现这个接口。因此，**Golang中没有implement这样的关键字**

14.3.3 接口注意事项

注意事项和细节

- 1) 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)
- 2) 接口中所有的方法都没有方法体，即都是没有实现的方法。
- 3) 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。
- 4) 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。
- 5) 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。

6.一个接口（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。

7.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值给空接口）。

8.如果实现了两个接口，两个接口里有相同的方法，编译会报错，而在java里即是A接口，也是B接口，可以正常使用。但是后续的go版本好像又支持重复的方法了..

14.3.4 接口编程的经典案例

结构体排序

```
package main

import (
    "fmt"
    "math/rand"
    "sort"
)

// 1. 声明Hero结构体
type Hero struct {
    Name string
    Age  int
}

// 2. 声明一个Hero结构体切片类型
type HeroSlice []Hero

// 3. 实现Interface接口
func (h HeroSlice) Len() int {
    return len(h)
}

// Less方法就是决定你使用什么标准进行排序
// 1. 按Hero的年龄从小到大排序!
func (h HeroSlice) Less(i, j int) bool {
    return h[i].Age > h[j].Age
}

func (h HeroSlice) Swap(i, j int) {
    //temp := h[i]
    //h[i] = h[j]
    //h[j] = temp
    h[i], h[j] = h[j], h[i]
}

func main() {
    //var intSlice = []int{0, -1, 10, 7, 90}
    //sort.Ints(intSlice)
    //fmt.Println(intSlice)
```

```
var heros HeroSlice
for i := 0; i < 10; i++ {
    hero := Hero{
        Name: fmt.Sprintf("英雄~%d", rand.Intn(100)),
        Age:  rand.Intn(100),
    }
    heros = append(heros, hero)
}

// 排序前
for _, v := range heros {
    fmt.Println(v)
}

fmt.Println("排序后-----")
sort.Sort(heros)
for _, v := range heros {
    fmt.Println(v)
}
```

14.4 实现接口和继承比较

接口是对继承的补充

实现接口 vs 继承

➤ 大家听到现在，可能会对实现接口和继承比较迷茫了，这个问题，那么他们究竟有什么区别呢？

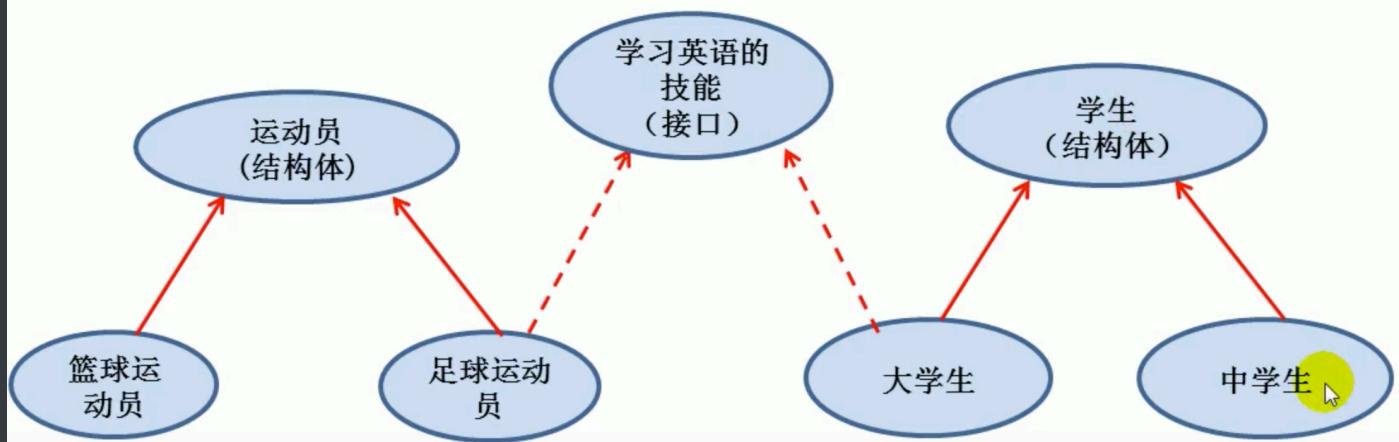


小猴子继承了老猴子的属性，可是又向学习像小鸟一样飞翔，像鱼一样游泳，这时候就需要实现接口。

- 接口(interface)

实现接口 vs 继承类

- 实现接口可以看作是对 继承的一种补充



- 接口(interface)

实现接口 vs 继承

- 接口和继承解决的解决问题不同

继承的价值主要在于：解决代码的复用性和可维护性。

接口的价值主要在于：设计，设计好各种规范(方法)，让其它自定义类型去实现这些方法。

- 接口比继承更加灵活

接口比继承更加灵活，继承是满足 `is - a` 的关系，而接口只需满足 `like - a` 的关系。

- 接口在一定程度上实现代码解耦

14.5 多态

● 面向对象编程-多态

基本介绍

变量(实例)具有多种形态。面向对象的第三大特征，在Go语言，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。

快速入门

在前面的Usb接口案例，Usb usb，既可以接收手机变量，又可以接收相机变量，就体现了Usb 接口 多态特性。

接口体现多态特征

1) 多态参数

在前面的Usb接口案例，Usb usb，既可以接收手机变量，又可以接收相机变量，就体现了Usb 接口 多态

2) 多态数组

演示一个案例：给Usb数组中，存放Phone结构体 和 Camera结构体变量，Phone还有一个特有的方法call()，请遍历Usb数组，如果是Phone变量，除了调用Usb 接口声明的方法外，还需要调用Phone 特有方法 call.

14.6 类型断言

```
package main

import "fmt"

type Point struct {
    x int
    y int
}

func main() {
    var a interface{}
    var point Point = Point{1, 2}
    a = point
    fmt.Println("a = ", a)
    var b Point
    //b = a // 不可以
```

```
b = a.(Point)
fmt.Println(b)
}
```

带检查的断言

```
package main

import "fmt"

type Point struct {
    x int
    y int
}

func main() {
    var a interface{}
    var point Point = Point{1, 2}
    a = point
    fmt.Println("a = ", a)
    //var b Point
    //b = a // 不可以
    if b, ok := a.(Point); ok {
        fmt.Println("转换成功 b = ", b)
    } else {
        fmt.Println("转换失败")
    }
}
```

类型断言的最佳实践

```
package main

import "fmt"

type Student struct {}

// 编写一个函数，可以判断输入的参数是什么类型
func TypeJudge(items ...interface{}) {
    for index, v := range items {
        switch v.(type) {
        case bool:
```

```

    fmt.Sprintf("第%v个参数是bool类型, 值是%v\n", index, v)
    case float32:
        fmt.Sprintf("第%v个参数是float32类型, 值是%v\n", index, v)
    case float64:
        fmt.Sprintf("第%v个参数是float64类型, 值是%v\n", index, v)
    case int, int32, int64:
        fmt.Sprintf("第%v个参数是int类型, 值是%v\n", index, v)
    case string:
        fmt.Sprintf("第%v个参数是string类型, 值是%v\n", index, v)
    case Student:
        fmt.Sprintf("第%v个参数是Student类型, 值是%v\n", index, v)
    case *Student:
        fmt.Sprintf("第%v个参数是*Student类型, 值是%v\n", index, v)
    default:
        fmt.Sprintf("第%v个参数类型不确定, 值是%v\n", index, v)
    }
}
}

func main() {
    var n1 float32 = 1.1
    var n2 float64 = 2.3
    var n3 int32 = 30
    var name string = "tom"
    address := "深圳"
    n4 := 300
    TypeJudge(n1, n2, n3, name, address, n4, Student{}, &Student{})
}

```

15. 反射

15.1 反射重要的函数和概念

(1) reflect.TypeOf(变量名), 获取变量的类型, 返回reflect.Type类型。

(2) reflect.ValueOf(变量名), 获取变量的值, 返回reflect.Value类型, reflect.Value是一个结构体类型。通过reflect.Value, 可以获取到关于变量的很多信息。

(3) 变量、interface{}和reflect.Value是可以相互转换的, 这点在实际开发中, 会经常使用到。

```
// 专门用于做反射
func test(b interface{}) {
    // 1.如何将interface{}转成reflect.Value
    rVal := reflect.ValueOf(b)
    // 2.如何将reflect.Value转成interface{}
    iVal := rVal.Interface()
    // 3.如何将interface{}转成原来的变量类型, 使用类型断言
    v := iVal.(Student)
}
```

15.2 快速入门案例

1.对(基本数据类型、interface{}、reflect.Value)进行反射的基本操作

```
package main

import (
    "fmt"
    "reflect"
)

func reflectTest01(b interface{}) {
    // 通过反射获取到传入的变量的type, kind, 值
    // 1.先获取到reflect.Type
    rTyp := reflect.TypeOf(b)
    fmt.Println("rTyp = ", rTyp)

    // 2.获取到reflect.Value
    rVal := reflect.ValueOf(b)
    fmt.Println("rVal = ", rVal.Int() + 2) // 真正的类型是reflect.Value

    // 3.将rVal转成interface{}
    iV := rVal.Interface()
    // 将interface通过断言转成需要的类型
    num2 := iV.(int)
    fmt.Println("num2 = ", num2)
}

func main() {
    var num int = 100
    reflectTest01(num)
}
```

2.对结构体进行反射的基本操作

```
package main

import (
    "fmt"
    "reflect"
)

func testReflect02(b interface{}) {
    // 通过反射获取到传入的变量的type, kind, 值
    // 1.先获取到reflect.Type
    rTyp := reflect.TypeOf(b)
    fmt.Println("tTyp = ", rTyp)

    // 2.获取到reflect.Value
    rVal := reflect.ValueOf(b)
    iV := rVal.Interface()
    fmt.Printf("iv=%v iv type=%T\n", iV, iV)

    // 3.获取变量对应的Kind
    fmt.Printf("kind = %v, kind = %v\n", rTyp.Kind(), rVal.Kind())

    // 将interface{} 通过断言转成需要的类型
    // 这里，我们就简单使用了一带检测的类型断言。
    // 也可以使用 switch 的断言形式来做的更加的灵活
    stu, ok := iV.(student)
    if ok {
        fmt.Printf("stu.name=%v\n", stu.Name)
    }
}

type student struct {
    Name string
    Age  int
}

func main() {
    stu := student{Name: "tom", Age: 18}
    testReflect02(stu)
}
```

15.3 反射注意事项和细节说明

1.reflect.Value.Kind, 获取变量的类别, 返回的是一个常量 (看手册)

2.Type是类型, Kind是类别, Type和Kind可能是相同的, 也可能是不同的。

比如: var num int = 10 num的Type是int, Kind也是int

比如: var stu Studnet stu的Type是pkg1.Student, Kind是struct

3.通过反射可以让变量在interface{}和Reflect.Value之间相互转换。

4.使用反射的方式来获取变量的值 (**并返回对应的类型**), 要求数据类型匹配, 比如x是int, 那么就应用使用reflect.Value(x.Int()), 而不能使用其它的, 否则报panic。