1. 初识go语言

1.1 go语言环境搭建

1.1.1 编译工具（参考环境搭建文档）

ubuntu:

liteide

sublime3

windows:

liteide 开源 mac版本也有

sublime3

1.1.2 ide下载：

<https://sourceforge.net/projects/liteide/files>

1.1.3标准命令概述

Go语言中包含了大量用于处理Go语言代码的命令和工具。其中，go命令就是最常用的一个，它有许多子命令。这些子命令都拥有不同的功能，如下所示。

l build：用于编译给定的代码包或Go语言源码文件及其依赖包。

l clean：用于清除执行其他go命令后遗留的目录和文件。

l doc：用于执行godoc命令以打印指定代码包。

l env：用于打印Go语言环境信息。

l fix：用于执行go tool fix命令以修正给定代码包的源码文件中包含的过时语法和代码调用。

l fmt：用于执行gofmt命令以格式化给定代码包中的源码文件。

l get：用于下载和安装给定代码包及其依赖包(提前安装git或hg)。

l list：用于显示给定代码包的信息。

l run：用于编译并运行给定的命令源码文件。

l install：编译包文件并编译整个程序。

l test：用于测试给定的代码包。

l tool：用于运行Go语言的特殊工具。

l version：用于显示当前安装的Go语言的版本信息。

l vet : 报告包中可能出现的错误

l bug : 开始一个错误报告

1.1.4 学习资料

go 官网 <https://golang.org/>

go中文社区 <https://studygolang.com>

go中文在线文档 <https://studygolang.com/pkgdoc>

1.2第一个go程序

1.2.1 go语言以包作为管理单位

1.2.2每个文件必须先声明包

1.2.3 程序必须有一个main包, Go语言的main()函数不能带参数，也不能定义返回值

1.2.4 使用函数必须import导入,导入包后 必须使用

1.2.5 go程序 一个工程有且只有一个main函数为入口函数

1.2.6    // 单行注释      /\* \*/ 块注释

1.2.7函数左括号必须与函数同行

1.2.8 命令行编译go 生成可执行文件 在执行

a. go build xxx.go 编译go代码 生成一个可执行程序

./xxx

b. go run xxx.go 不生成可执行文件 直接运行

1.2.9 第一个go程序



1.3 命令行运行程序



1.4 总结

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

4.使用函数必须import导入,导入包后 必须使用

5.go程序 一个工程有且只有一个main函数为入口函数

6.// 单行注释 /\* \*/ 块注释

7.函数左括号必须与函数同行

8.命令行编译go 生成可执行文件 在执行

a. go build xxx.go 编译go代码 生成一个可执行程序

./xxx

b. go run xxx.go 不生成可执行文件 直接运行

   9.标准命令概述



2. 基础类型

数据类型：一个变量该分配多大的内存

2.1 命名

2.1.1命名规则：

一个名字必须以一个字母（Unicode字母）或下划线开头，后面可以跟任意数量的字母、数字或下划线

2.1.2 关键字：

关键字有25个(均为小写)。关键字不能用于自定义名字

break       default         func          interface        select

case        defer           go            map              struct

chan        else            goto           package          switch

const       fallthrough     if             range            type

continue     for            import        return            var

此外，还有大约30多个预定义的名字，比如int和true等，主要对应内建的常量、类型和函数。

内建常量:

    true false iota nil

内建类型: 677

        int int8 int16 int32 int64

        uint uint8 uint16 uint32 uint64 uintptr

        float32 float64 complex128 complex64

        bool byte rune string error

内建函数:

        make len cap new append copy close delete

        complex real imag

        panic recover

**byte**是uint8的别名，byte类型的值需用8个比特位表示，其表示法与uint8类型无异。因此我们就不再这里赘述了。我们下面重点说说rune类型。

**rune**是int32的别名,rune 是 Go 中的内置类型，它是 int32 的别名。在 Go 中，rune表示一个 Unicode 码点。无论一个码点会被编码为多少个字节，它都可以表示为一个 rune。  
汉字在字节中占2-4个码点。直接使用下标访问并不能得到正确的值，可以先将其转换为rune类型再使用下标访问

**range**



**string**

// 1. string默认为"" 不为nil

// 2. string类型不能赋值为nil

// 3. string类型不能与nil做比较,类型不匹配

// 4. 使用下标可以访问string类型中的某个字节，字符串下标不可以被赋值,不能⽤序号获取字节元素指针， &s[i] ⾮法,要修改字符串，可先将其转换成 []rune 或 []byte，完成后再转换为 string。⽆论哪种转换，都会重新分配内存，并复制字节数组。

2.2 变量

      变量是程序运行期间可以改变的量。变量相当于是对一块数据存储空间的命名，程序可以通过定义一个变量来申请一块数据存储空间，之后可以通过引用变量名来使用这块存储空间。

2.2.1 变量声明

    var v1 int

    var v3, v4 int  //一次定义多个变量

    var (

        v5 int

        v6 int

    )

2.2.2 变量初始化

     var v1 int = 10  // 方式1

    var v2 = 10       // 方式2，编译器自动推导出v2的类型

    v3 := 10           // 方式3，编译器自动推导出v3的类型

    //出现在 := 左侧的变量不应该是已经被声明过，:=定义时必须初始化

    var v4 int

    v4 := 2 //err

2.2.3 变量赋值

    var v1 int

    v1 = 123

    var v2, v3, v4 int

    v2, v3, v4 = 1, 2, 3    //多重赋值

    i := 10

    j := 20

    i, j = j, i    //多重赋值  交换两个变量的值

2.2.4 匿名变量

      \_（下划线）是个特殊的变量名，任何赋予它的值都会被丢弃：

\_, i, \_, j := 1, 2, 3, 4

2.2.5 获取变量类型

fmt.Println("str type:",reflect.TypeOf(str)) // reflect.TypeOf()

fmt.Printf("t2 type is %T\n",t2)                          // 自动推到类型为

2.3 常量

常量是指编译期间就已知且不可改变的值。常量可以是数值类型（包括整型、浮点型和复数类型）、布尔类型、字符串类型等。

2.3.1 字面常量（常量值）

      所谓字面常量（literal），是指程序中硬编码的常量，如

123

3.1415  // 浮点类型的常量

3.2+12i // 复数类型的常量

true  // 布尔类型的常量

"foo" // 字符串常量

2.3.2 常量定义

    const Pi float64 = 3.14

const zero = 0.0 // 浮点常量, 自动推导类型

const ( // 常量组

        size int64 = 1024

        eof        = -1 // 整型常量, 自动推导类型

    )

    const u, v float32 = 0, 3 // u = 0.0, v = 3.0，常量的多重赋值

    const a, b, c = 3, 4, "foo"

    // a = 3, b = 4, c = "foo"    //err, 常量不能修改

2.3.3 枚举（iota）

      常量声明可以使用iota常量生成器初始化，它用于生成一组以相似规则初始化的常量，但是不用每行都写一遍初始化表达式，在一个const声明语句中，在第一个声明的常量所在的行，iota将会被置为0，然后在每一个有常量声明的行加一

    const (

        x = iota // x == 0

        y = iota // y == 1

        z = iota // z == 2

        w  // 这里隐式地说w = iota，因此w == 3。其实上面y和z可同样不用"= iota"

    )

    const v = iota // 每遇到一个const关键字，iota就会重置，此时v == 0

    const (

        h, i, j = iota, iota, iota //h=0,i=0,j=0 iota在同一行值相同

    )

    const (

        a       = iota //a=0

        b       = "B"

        c       = iota             //c=2

        d, e, f = iota, iota, iota //d=3,e=3,f=3

        g       = iota             //g = 4

    )

    const (

        x1 = iota \* 10 // x1 == 0

        y1 = iota \* 10 // y1 == 10

        z1 = iota \* 10 // z1 == 20

    )

2.4 基础数据类型

2.4.1 分类

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **名称** | **长度** | **零值** | **说明** |
| bool | 布尔类型 | 1 | false | 其值不为真即为假，不可以用数字代表true或false |
| byte | 字节型 | 1 | 0 | uint8别名 |
| rune | 字符类型 | 4 | 0 | 专用于存储unicode编码，等价于uint32 |
| int, uint | 整型 | 4或8 | 0 | 32位或64位 |
| int8, uint8 | 整型 | 1 | 0 | -128 ~ 127, 0 ~ 255 |
| int16, uint16 | 整型 | 2 | 0 | -32768 ~ 32767, 0 ~ 65535 |
| int32, uint32 | 整型 | 4 | 0 | -21亿 ~21 亿, 0 ~ 42 亿 |
| int64, uint64 | 整型 | 8 | 0 |  |
| float32 | 浮点型 | 4 | 0.0 | 小数位精确到7位 |
| float64 | 浮点型 | 8 | 0.0 | 小数位精确到15位 |
| complex64 | 复数类型 | 8 |  |  |
| complex128 | 复数类型 | 16 |  |  |
| uintptr | 整型 | 4或8 |  | ⾜以存储指针的uint32或uint64整数 |
| string | 字符串 |  | "" | utf-8字符串 |

2.4.2 布尔类型

var v1 bool

v1 = true

v2 := (1 == 2) // v2也会被推导为bool类型

//布尔类型不能接受其他类型的赋值，不支持自动或强制的类型转换

var b bool

b = 1 // err, 编译错误

b = bool(1) // err, 编译错误

2.4.3 整型

     var v1 int32

v1 = 123

     v2 := 64 // v1将会被自动推导为int类型

2.4.4 浮点型

    var f1 float32

f1 = 12

f2 := 12.0 //如果不加小数点， fvalue2会被推导为整型而不是浮点型，float64

2.4.5 字符类型

支持两个字符类型，一个是byte（实际上是uint8的别名），代表utf-8字符串的单个字节的值；另一个是rune，代表单个unicode字符。

     var ch1, ch2, ch3 byte

    ch1 = 'a'  //字符赋值

    ch2 = 97   //字符的ascii码赋值

    ch3 = '\n' //转义字符

2.4.6 字符串

     var str string     // 声明一个字符串变量

    str = "abc"         // 字符串赋值

ch := str[0]        // 取字符串的第一个字符

     len(str))          //内置的函数len()来取字符串的长度

     //`(反引号)括起的字符串为Raw字符串，即字符串在代码中的形式就是打印时的形式，它没有字符转义，换行也将原样输出。

    str2 := `hello

    mike \n \r测试

    `

2.4.7 复数类型

复数实际上由两个实数（在计算机中用浮点数表示）构成，一个表示实部（real），一个表示虚部（imag）。

    var v1 complex64 // 由2个float32构成的复数类型

    v1 = 3.2 + 12i

    v2 := 3.2 + 12i        // v2是complex128类型

    v3 := complex(3.2, 12) // v3结果同v2

    fmt.Println(v1, v2, v3)

    //内置函数real(v1)获得该复数的实部

    //通过imag(v1)获得该复数的虚部

    fmt.Println(real(v1), imag(v1))

2.5 fmt包的格式化输出输入

2.5.1 格式化说明

|  |  |
| --- | --- |
| **格式** | **含义** |
| %% | 一个%字面量 |
| %b | 一个二进制整数值(基数为2)，或者是一个(高级的)用科学计数法表示的指数为2的浮点数 |
| %c | 字符型。可以把输入的数字按照ASCII码相应转换为对应的字符 |
| %d | 一个十进制数值(基数为10) |
| %e | 以科学记数法e表示的浮点数或者复数值 |
| %E | 以科学记数法E表示的浮点数或者复数值 |
| %f | 以标准记数法表示的浮点数或者复数值 |
| %g | 以%e或者%f表示的浮点数或者复数，任何一个都以最为紧凑的方式输出 |
| %G | 以%E或者%f表示的浮点数或者复数，任何一个都以最为紧凑的方式输出 |
| %o | 一个以八进制表示的数字(基数为8) |
| %p | 以十六进制(基数为16)表示的一个值的地址，前缀为0x,字母使用小写的a-f表示 |
| %q | 使用Go语法以及必须时使用转义，以双引号括起来的字符串或者字节切片[]byte，或者是以单引号括起来的数字 |
| %s | 字符串。输出字符串中的字符直至字符串中的空字符（字符串以'\0‘结尾，这个'\0'即空字符） |
| %t | 以true或者false输出的布尔值 |
| %T | 使用Go语法输出的值的类型 |
| %U | 一个用Unicode表示法表示的整型码点，默认值为4个数字字符 |
| %v | 使用默认格式输出的内置或者自定义类型的值，或者是使用其类型的String()方式输出的自定义值，如果该方法存在的话 |
| %x | 以十六进制表示的整型值(基数为十六)，数字a-f使用小写表示 |
| %X | 以十六进制表示的整型值(基数为十六)，数字A-F使用小写表示 |

2.5.2 输出

      1. fmt.Print

      2. fmt.Printf

      3. fmt.Println

      4. fmt.Fprint

      5. fmt.Fprintf

      6. fmt.Fprinln

      7. fmt.Sprint

      8. fmt.Sprintf

      9. fmt.Sprintln

    //整型

    a := 15

    fmt.Printf("a = %b\n", a) //a = 1111

    fmt.Printf("%%\n")        //只输出一个%

    //字符

    ch := 'a'

    fmt.Printf("ch = %c, %c\n", ch, 97) //a, a

    //浮点型

    f := 3.14

    fmt.Printf("f = %f, %g\n", f, f) //f = 3.140000, 3.14

    fmt.Printf("f type = %T\n", f)   //f type = float64

    //复数类型

    v := complex(3.2, 12)

    fmt.Printf("v = %f, %g\n", v, v) //v = (3.200000+12.000000i), (3.2+12i)

    fmt.Printf("v type = %T\n", v)   //v type = complex128

    //布尔类型

    fmt.Printf("%t, %t\n", true, false) //true, false

    //字符串

    str := "hello go"

    fmt.Printf("str = %s\n", str) //str = hello go

2.5.3 输入

      fmt.Scanf()

      fmt.Scan()

    var v int

    fmt.Println("请输入一个整型：")

    fmt.Scanf("%d", &v)

    //fmt.Scan(&v)

    fmt.Println("v = ", v)

2.6 类型转换

     var ch byte = 97

    //var a int = ch //err, cannot use ch (type byte) as type int in assignment

    var a int = int(ch)

2.7 类型别名

    type bigint int64 //int64类型改名为bigint

    var x bigint = 100

    type (

        myint int    //int改名为myint

        mystr string //string改名为mystr

    )

2.8 总结

*1. 零值：声明未初始化时的初始值*

2. 复数获取实部和虚部：

内置函数real(v1)获得该复数的实部

    通过imag(v1)获得该复数的虚部

3. fmt输出

   %c 字符型

   %d 一个十进制数值

   %o 八进制表示的数字

   %T 输出值的类型

4. 匿名变量

         \_（下划线）是个特殊的变量名，任何赋予它的值都会被丢弃：

\_, i, \_, j := 1, 2, 3, 4

5. int8 int16 int32 int64之间的区别

uint8 the set of all unsigned 8-bit integers (0 to 255)

uint16 the set of all unsigned 16-bit integers (0 to 65535)

uint32 the set of all unsigned 32-bit integers (0 to 4294967295)

uint64 the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8 the set of all signed 8-bit integers (-128 to 127)

int16 the set of all signed 16-bit integers (-32768 to 32767)

int32 the set of all signed 32-bit integers (-2147483648 to 2147483647)

int64 the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32 the set of all IEEE-754 32-bit floating-point numbers

float64 the set of all IEEE-754 64-bit floating-point numbers

complex64 the set of all complex numbers with float32 real and imaginary parts

complex128 the set of all complex numbers with float64 real and imaginary parts

byte alias for uint8

rune alias for int32

6. ASCII码重要的几个点

48 ----> 0

57 ----> 9

65 ----> A

90 ----> Z

97 ----> a

122----> z

3. 运算符

3.1 算数运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **术语** | **示例** | **结果** |
| + | 加 | 10 + 5 | 15 |
| - | 减 | 10 - 5 | 5 |
| \* | 乘 | 10 \* 5 | 50 |
| / | 除 | 10 / 5 | 2 |
| % | 取模(取余) | 10 % 3 | 1 |
| ++ | 后自增，没有前自增 | a=0;a++ | a=1 |
| -- | 后自减，没有前自减 | a=2;a-- | a=1 |

3.2 关系运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **术语** | **示例** | **结果** |
| == | 相等于 | 4 == 3 | false |
| != | 不等于 | 4 != 3 | true |
| < | 小于 | 4 < 3 | false |
| > | 大于 | 4 > 3 | true |
| <= | 小于等于 | 4 <= 3 | false |
| >= | 大于等于 | 4 >= 1 | true |

3.3 逻辑运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **术语** | **示例** | **结果** |
| ! | 非 | !a | 如果a为假，则!a为真；  如果a为真，则!a为假。 |
| && | 与 | a && b | 如果a和b都为真，则结果为真，否则为假。 |
| || | 或 | a || b | 如果a和b有一个为真，则结果为真，二者都为假时，结果为假。 |

3.4 位运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **术语** | **说明** | **示例** |
| & | 按位与 | 参与运算的两数各对应的二进位相与 | 60& 13 结果为12 |
| | | 按位或 | 参与运算的两数各对应的二进位相或 | 60| 13 结果为61 |
| ^ | 异或 | 参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1 | 60^ 13 结果为240 |
| << | 左移 | 左移n位就是乘以2的n次方。 左边丢弃，右边补0。 | 4<< 2 结果为16 |
| >> | 右移 | 右移n位就是除以2的n次方。 右边丢弃，左边补位。 | 4>> 2 结果为1 |

3.5 赋值运算符

|  |  |  |
| --- | --- | --- |
| **运算符** | **说明** | **示例** |
| = | 普通赋值 | c =a + b 将 a + b 表达式结果赋值给 c |
| += | 相加后再赋值 | c+= a 等价于 c = c + a |
| -= | 相减后再赋值 | c-= a 等价于 c = c - a |
| \*= | 相乘后再赋值 | c\*= a 等价于 c = c \* a |
| /= | 相除后再赋值 | c/= a 等价于 c = c / a |
| %= | 求余后再赋值 | c%= a 等价于 c = c % a |
| <<= | 左移后赋值 | c<<= 2 等价于 c = c << 2 |
| >>= | 右移后赋值 | c>>= 2 等价于 c = c >> 2 |
| &= | 按位与后赋值 | c&= 2 等价于 c = c & 2 |
| ^= | 按位异或后赋值 | c^= 2 等价于 c = c ^ 2 |
| |= | 按位或后赋值 | c|= 2 等价于 c = c | 2 |

3.6 其他运算符

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **术语** | **示例** | **说明** |
| & | 取地址运算符 | &a | 变量a的地址 |
| \* | 取值运算符 | \*a | 指针变量a所指向内存的值 |

3.7 运算符优先级

在Go语言中，一元运算符拥有最高的优先级，二元运算符的运算方向均是从左至右。

下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

|  |  |
| --- | --- |
| **优先级** | **运算符** |
| **7** | **^     !** |
| **6** | **\*      /   %   <<   >>   &     &^** |
| **5** | **+     -    |     ^** |
| **4** | **==  !=  <   <=   >=   >** |
| **3** | **<-** |
| **2** | **&&** |
| **1** | **||** |

3.8 总结

      1. ++ 和 -- 运算符，只有后，没有前

      2. << 左移 左移n位就是乘以2的n次方。 左边丢弃，右边补0

      3. >> 右移 右移n位就是除以2的n次方。 右边丢弃，左边补位。

4. 流程控制

Go语言支持最基本的三种程序运行结构：顺序结构、选择结构、循环结构。

l 顺序结构：程序按顺序执行，不发生跳转。

l 选择结构：依据是否满足条件，有选择的执行相应功能。

l 循环结构：依据条件是否满足，循环多次执行某段代码。

4.1 选择结构

4.1.1 if语句

4.1.1.1 if

            var a int = 3

    if a == 3 { //条件表达式没有括号

        fmt.Println("a==3")

    }

    //支持一个初始化表达式, 初始化字句和条件表达式直接需要用分号分隔

    if b := 3; b == 3 {

        fmt.Println("b==3")

    }

4.1.1.2 if … else

     if a := 3; a == 4 {

        fmt.Println("a==4")

    } else { //左大括号必须和条件语句或else在同一行

        fmt.Println("a!=4")

    }

4.1.1.3 if ... else if ... else

    if a := 3; a > 3 {

        fmt.Println("a>3")

    } else if a < 3 {

        fmt.Println("a<3")

    } else if a == 3 {

        fmt.Println("a==3")

    } else {

        fmt.Println("error")

    }

4.1.2 switch语句

Go里面switch默认相当于每个case最后带有break，匹配成功后不会自动向下执行其他case，而是跳出整个switch, 但是可以使用fallthrough强制执行后面的case代码：

4.1.2.1

    var score int = 90

    switch score {

    case 90:

        fmt.Println("优秀")

        //fallthrough

    case 80:

        fmt.Println("良好")

        //fallthrough

    case 50, 60, 70:

        fmt.Println("一般")

        //fallthrough

    default:

        fmt.Println("差")

    }

4.1.2.2可以使用任何类型或表达式作为条件语句：

    //1

    switch s1 := 90; s1 { //初始化语句;条件

    case 90:

        fmt.Println("优秀")

    case 80:

        fmt.Println("良好")

    default:

        fmt.Println("一般")

    }

//2

  var s2 int = 90

    switch { //这里没有写条件

    case s2 >= 90: //这里写判断语句

        fmt.Println("优秀")

    case s2 >= 80:

        fmt.Println("良好")

    default:

        fmt.Println("一般")

    }

    //3

    switch s3 := 90; { //只有初始化语句，没有条件

    case s3 >= 90: //这里写判断语句

        fmt.Println("优秀")

    case s3 >= 80:

        fmt.Println("良好")

    default:

        fmt.Println("一般")

    }

4.2 循环语句

4.2.1 for循环

 var i, sum int

    for i = 1; i <= 100; i++ {

        sum += i

        fmt.Println("sum = ", sum)

    }

4.2.2 range

关键字 range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值：

s := "abc"

    for i := range s { //支持 string/array/slice/map。

        fmt.Printf("%c\n", s[i])

    }

    for \_, c := range s { // 忽略 index

        fmt.Printf("%c\n", c)

}

    for i, c := range s {

        fmt.Printf("%d, %c\n", i, c)

    }

4.2 循环语句

4.2.1 for

    var i, sum int

    for i = 1; i <= 100; i++ {

        sum += i

    }

4.2.2 range

关键字 range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值：

    s := "abc"

    for i := range s { //支持 string/array/slice/map。

        fmt.Printf("%c\n", s[i])

    }

    for \_, c := range s { // 忽略 index

        fmt.Printf("%c\n", c)

}

     for i, c := range s {

        fmt.Printf("%d, %c\n", i, c)

    }

4.3 跳转语句

4.3.1 break和continue

break操作是跳出当前循环，continue是跳过本次循环，执行下一次循环

    for i := 0; i < 5; i++ {

        if 2 == i {

            //break    //break操作是跳出当前循环

            continue //continue是跳过本次循环

        }

        fmt.Println(i)

    }

注意：break可⽤于for、switch、select，⽽continue仅能⽤于for循环。

4.3.2 goto

func main() {

    for i := 0; i < 5; i++ {

        for {

            fmt.Println(i)

            goto LABEL //跳转到标签LABEL，从标签处，执行代码

        }

    }

    fmt.Println("this is test")

LABEL:

    fmt.Println("it is over")

}

4.4 总结

      1. if条件表达式没有括号，支持一个初始化表达式，初始化表达式与条件表达式用；隔开， else { 做大括号必须与else同行

      2. switch默认相当于每个case最后带有break，但是可以使用fallthrough强制执行后面的case代码

      3. range 会返回两个值，第一个返回值是元素的数组下标，第二个返回值是元素的值，**range支持 string/array/slice/map**

      4. break操作是跳出当前循环，continue是跳过本次循环执行下一次循环，break可⽤于for、switch、select，⽽continue仅能⽤于for循环。

      5. goto跳转到必须在当前函数内定义的标签

5. 函数

5.1 定义格式

func FuncName(/\*参数列表\*/) (o1 type1, o2 type2/\*返回类型\*/) {

    //函数体

    return v1, v2 //返回多个值

}

函数定义说明：

l func：函数由关键字 func 开始声明

l FuncName：函数名称，根据约定，函数名首字母小写即为private，大写即为public

l 参数列表：函数可以有0个或多个参数，参数格式为：变量名 类型，如果有多个参数通过逗号分隔，不支持默认参数

l 返回类型：

①　上面返回值声明了两个变量名o1和o2(命名返回参数)，这个不是必须，可以只有类型没有变量名

②　如果只有一个返回值且不声明返回值变量，那么你可以省略，包括返回值的括号

③　如果没有返回值，那么就直接省略最后的返回信息

④　如果有返回值， 那么必须在函数的内部添加return语句

5.2 自定义函数

      5.2.1无参无返函数

      5.2.2 有参无返函数

          5.2.2.1普通参数列表

**func** Test01(v1 int, v2 int)

**func** Test02(v1, v2 int)

       5.2.3 不定参数列表

          5.2.3.1 不定参数类型

                   形如 … type 格式的类型只能作为函数的参数类型存在，并且必须是最后一个参数

**func** Test(args ...int)   //函数调用可传 0到n个参数

          5.2.3.2 不定参数的传递

func Test(args ...int) {

//按原样传递, Test()的参数原封不动传递给MyFunc01

 args...

//Test()参数列表中，第2个参数及以后的参数传递给MyFunc02 参数列表下标0开始

MyFunc02(args[1:]...)

// 第0和 第1 个参数传递 不包括第2个

MyFunc02(args[:3]...)

}

5.2.3 无参有返回值

    5.2.3.1 一个返回值

必须有明确的终止语句 return，否则会引发编译错误。

func Test01() int { //方式1

    return 250

}

//官方建议：最好命名返回值，因为不命名返回值，虽然使得代码更加简洁了，但是会造成生成的文档可读性差

func Test02() (value int) { //方式2, 给返回值命名

    value = 250

    return value

}

func Test03() (value int) { //方式3, 给返回值命名

    value = 250

    return

}

5.2.3.2 返回多个值

**func** Test01() (int, string)

**func** Test02() (a int, str string)

5.2.4 有参有返回值

**func** MinAndMax(num1 int, num2 int) (min int, max int)

5.3 递归函数

      递归指函数可以直接或间接的调用自身。

**结构**：一个跳出条件和一个递归体。所谓跳出条件就是根据传入的参数判断是否需要停止递归，而递归体则是函数自身所做的一些处理。

**func** recursion\_function(num int)(value int){

**if** 100 == num{

**return** 100

}

**return** num + recursion\_function(num + 1)

}

fmt.Println(recursion\_function(1))

5.4 函数类型

        函数也是一种数据类型，我们可以通过type来定义它，它的类型就是所有拥有相同的参数，相同的返回值的一种类型。

type FuncType func(int, int) int //声明一个函数类型, func后面没有函数名

5.5 匿名与闭包

      闭包就是一个函数“捕获”了和它在同一作用域的其它常量和变量，这就意味着当闭包被调用的时候，不管在程序什么地方调用，闭包能够使用这些常量或者变量。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，所以只有闭包还在使用它，这些变量就还会存在。

5.6延迟调用defer

5.6.1 defer作用

   关键字 defer ⽤于延迟一个函数或者方法（或者当前所创建的匿名函数）的执行。注意，defer语句只能出现在函数或方法的内部。

defer语句经常被用于处理成对的操作，如打开、关闭、连接、断开连接、加锁、释放锁。通过defer机制，不论函数逻辑多复杂，都能保证在任何执行路径下，资源被释放。释放资源的defer应该直接跟在请求资源的语句后。defer调用会先把数据加载到内存中，最后执行

5.6.2 多个defer执行顺序

func defer\_test\_02(){

    fmt.Println("test\_02")

}

func defer\_test(){

    defer fmt.Println("aaa")

    defer\_test\_02()

    defer fmt.Println("bbb")

    defer fmt.Println("ccc")

}

输出：

test\_02

ccc

bbb

aaa

5.6.3 defer和匿名函数结合使用

*// 闭包和defer结合使用*

**func** bibao\_defer\_test(){

   v1,v2 := 10,100

**defer func**(a int){

       fmt.Println(**"defer: "**,a,v2)

   }(v1)

   v1++

   v2++

   fmt.Println(**"out"**,v1,v2)

}

输出：

out 11 101

defer: 10 101

5.7 获取命令行参数

      import os                          添加os包

      args := os.Args                获取命令行参数列表

      fmt.Println(“ args[1] ”,args[1])

5.8 作用域

在函数体内声明的变量、参数和返回值变量就是局部变量，它们的作用域只在函数体内：

作用域：变量、函数等的作用范围

5.8.1 局部变量

      定义在{}中的是局部变量，只能在{}中有效，执行定义变量才会分配内存，离开作用域自动释放

5.8.2 全局变量

      定义在函数外部的变量称为全局变量，全局变量在任何地方都可以调用

5.8.3 不同作用域同名

      不同作用域函数、变量可以同名

      不同作用域调用原则：就近原则

5.9 总结

* 1. 函数，避免代码冗余，用函数封装，提高复用，方便维护
     1. 自定义函数
        1. 无参无返，
        2. 有返回值（一个或多个返回值）
        3. 无参有返回值
        4. 有参数无返（不定参数使用，传递），不定参放在参数列表的最后
        5. 有参数有返回值
     2. 递归函数
        1. 函数的调用流程
        2. 函数调用本身
        3. 递归终止条件
     3. 函数类型(函数指针、回调函数、闭包)
        1. type func\_type func(v1 int, v2 string){}
     4. 匿名函数和闭包

1.func(a int, b string){

xxxxx

xxxxxxx

}(10,"mark")

2.必须放在函数内部

3.可以使用上级函数作用域中的变量

* + 1. defer
       1. 延迟调用，多个defer调用执行顺序与调用顺序相反
       2. 匿名函数与闭包结合使用，注意 值改变后延迟调用使用的是值改变之前的状态

3. defer调用会先把数据加载到内存中，最后执行

* + 1. 命令行参数
       1. 导入 os包 args := os.Args
       2. 参数个数len(args) 获取每个参数 args[0] 使用下表访问
    2. 作用域
       1. 局部变量，定义在{}中的变量 注意局部变量的生命周期为{}
       2. 全局变量，定义在函数外，生命周期为整个包内
       3. 全局变量与局部变量同名，就近原则

6. 工程管理

在实际的开发工作中，直接调用编译器进行编译和链接的场景是少而又少，因为在工程中不会简单到只有一个源代码文件，且源文件之间会有相互的依赖关系。如果这样一个文件一个文件逐步编译，那不亚于一场灾难。 Go语言的设计者作为行业老将，自然不会忽略这一点。早期Go语言使用makefile作为临时方案，到了Go 1发布时引入了强大无比的Go命令行工具。

Go命令行工具的革命性之处在于彻底消除了工程文件的概念，完全用目录结构和包名来推导工程结构和构建顺序。针对只有一个源文件的情况讨论工程管理看起来会比较多余，因为这可以直接用go run和go build搞定。

6.1 工作区

6.1.1 工作区介绍

Go代码必须放在工作区中。工作区其实就是一个对应于特定工程的目录，它应包含3个子目录：src目录、pkg目录和bin目录。

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

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

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

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

需要特别注意的是，只有当环境变量GOPATH中只包含一个工作区的目录路径时，go install命令才会把命令源码安装到当前工作区的bin目录下。若环境变量GOPATH中包含多个工作区的目录路径，像这样执行go install命令就会失效，此时必须设置环境变量GOBIN。

6.1.2 GOPATH设置

      为了能够构建这个工程，需要先把所需工程的根目录加入到环境变量GOPATH中。否则，即使处于同一工作目录(工作区)，代码之间也无法通过绝对代码包路径完成调用。

在实际开发环境中，工作目录往往有多个。这些工作目录的目录路径都需要添加至GOPATH。当有多个目录时，请注意分隔符，多个目录的时候Windows是分号，Linux系统是冒号，当有多个GOPATH时，默认会将go get的内容放在第一个目录下。

GOPATH下的src目录就是接下来开发程序的主要目录，所有的源码都是放在这个目录下面，那么一般我们的做法就是一个目录一个项目，

例如: $GOPATH/src/mymath 表示mymath这个应用包或者可执行应用，这个根据package是main还是其他来决定，main的话就是可执行应用，其他的话就是应用包，这个会在后续详细介绍package。

6.2 包

6.2.1 自定义包

对于一个较大的应用程序，我们应该将它的功能性分隔成逻辑的单元，分别在不同的包里实现。我们创建的的自定义包最好放在GOPATH的src目录下（或者GOPATH src的某个子目录）。

在Go语言中，代码包中的源码文件名可以是任意的。但是，这些任意名称的源码文件都必须以包声明语句作为文件中的第一行，每个包都对应一个独立的名字空间：

package calc

包中成员以名称⾸字母⼤⼩写决定访问权限：

l public: ⾸字母⼤写，可被包外访问

l private: ⾸字母⼩写，仅包内成员可以访问

**注意：**同一个目录下不能定义不同的package。

分文件编程

1.    多个源文件，必须放在src目录下

2.    设置GOPATH环境变量

3.    同一个目录包名必须一样

4.    go env 查看GOPATH环境变量

5.    同一个目录调用别的包函数，直接调用即可，不需import 包名

6.    不同目录包名不一样

7.    调用不同目录包函数格式： 包名.函数名（）

8.    调用其他包函数，函数名首字母为小写为private不可调用，函数名首字母为小写为public可调用

Mac goland 如何分文件编写

<https://studygolang.com/articles/17688?fr=sidebar>

6.2.2 main包

所有用 Go 语言编译的可执行程序都必须有一个名叫 main 的包。一个可执行程序有且仅有一个 main函数。

6.2.3 main函数和init函数

Go里面有两个保留的函数：init函数（能够应用于所有的package）和main函数（只能应用于package main），这两个函数在定义时不能有任何的参数和返回值，虽然一个package里面可以写任意多个init函数，但这无论是对于可读性还是以后的可维护性来说，我们都强烈建议用户在一个package中每个文件只写一个init函数。

Go程序会自动调用init()和main()，所以你不需要在任何地方调用这两个函数。每个package中的init函数都是可选的，但package main就必须包含一个main函数。

init函数都会在程序执行开始的时候被调用。所有被编译器发现的 init 函数都会安排在 main 函数之前执行。 init 函数用在设置包、初始化变量或者其他要在程序运行前优先完成的引导工作。

程序的初始化和执行都起始于main包。如果main包还导入了其它的包，那么就会在编译时将它们依次导入。

有时一个包会被多个包同时导入，那么它只会被导入一次（例如很多包可能都会用到fmt包，但它只会被导入一次，因为没有必要导入多次）。

当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行init函数（如果有的话），依次类推。等所有被导入的包都加载完毕了，就会开始对main包中的包级常量和变量进行初始化，然后执行main包中的init函数（如果存在的话），最后执行main函数。下图详细地解释了整个执行过程：

init函数

/src/main.go 文件



与main.go文件同级 src/test.go文件



src/calc/calc.go 文件



运行结果



6.2.4 导入包

导入包需要使用关键字import，它会告诉编译器你想引用该位置的包内的代码。包的路径可以是相对路径，也可以是绝对路径。

//方法1

import "calc"

import "fmt"

//方法2

import (

    "calc"

    "fmt"

)

标准库中的包会在安装 Go 的位置找到。 Go 开发者创建的包会在 GOPATH 环境变量指定的目录里查找。GOPATH 指定的这些目录就是开发者的个人工作空间。

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

的时候就会出错。

注意：如果导入包之后，未调用其中的函数或者类型将会报出编译错误。

**点操作**

import (

    //这个点操作的含义是这个包导入之后在你调用这个包的函数时，可以省略前缀的包名

    . "fmt"

)

     Println("hello go")

**别名操作**

import (

    io "fmt" //fmt改为为io

)

    io.Println("hello go") //通过io别名调用

\_操作 忽略此包

有时，用户可能需要导入一个包，但是不需要引用这个包的标识符。在这种情况，可以使用空白标识符\_来重命名这个导入：

import (

    \_ "fmt"

)

\_操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的init函数。

6.3 GOPATH设置

6.3.1 windows

   计算机右键->属性->高级系统设置->环境变量->添加环境变量GOPATH value为src上一级目录

6.3.2 linux

   export GOPATH=/xxx/xx/xxx/xxx/xxx

   go env查看GOPATH是否已更改

6.4 go命令行

  假如你已安装了Go语言，你可以在命令行执行go命令查看相关的Go语言命令。如图：

### go build

go build 命令主要是用于测试编译。在包的编译过程中，若有必要，会同时编译与之相关联的包。

如果是普通包，当你执行go build命令后，不会产生任何文件。

如果是main包，当只执行go build命令后，会在当前目录下生成一个可执行文件。如果需要在$GOPATH/bin木下生成相应的exe文件，需要执行go install 或者使用 go build -o 路径/a.exe。

如果某个文件夹下有多个文件，而你只想编译其中某一个文件，可以在 go build 之后加上文件名，例如 go build a.go；go build 命令默认会编译当前目录下的所有go文件。

你也可以指定编译输出的文件名。比如，我们可以指定go build -o myapp.exe，默认情况是你的package名(非main包)，或者是第一个源文件的文件名(main包)。

go build 会忽略目录下以”\_”或者”.”开头的go文件。

如果你的源代码针对不同的操作系统需要不同的处理，那么你可以根据不同的操作系统后缀来命名文件。例如有一个读取数组的程序，它对于不同的操作系统可能有如下几个源文件：

array\_linux.go

array\_darwin.go

array\_windows.go

array\_freebsd.go

go build的时候会选择性地编译以系统名结尾的文件（Linux、Darwin、Windows、Freebsd）。例如Linux系统下面编译只会选择array\_linux.go文件，其它系统命名后缀文件全部忽略。

### go clean

go clean 命令是用来移除当前源码包里面编译生成的文件，这些文件包括

\_obj/ 旧的object目录，由Makefiles遗留

\_test/ 旧的test目录，由Makefiles遗留

\_testmain.go 旧的gotest文件，由Makefiles遗留

test.out 旧的test记录，由Makefiles遗留

build.out 旧的test记录，由Makefiles遗留

\*.[568ao] object文件，由Makefiles遗留

DIR(.exe) 由 go build 产生

DIR.test(.exe) 由 go test -c 产生

MAINFILE(.exe) 由 go build MAINFILE.go产生

### go fmt

go fmt 命令主要是用来帮你格式化所写好的代码文件。

比如我们写了一个格式很糟糕的 test.go 文件，我们只需要使用 fmt go test.go 命令，就可以让go帮我们格式化我们的代码文件。但是我们一般很少使用这个命令，因为我们的开发工具一般都带有保存时自动格式化功能，这个功能底层其实就是调用了 go fmt 命令而已。

使用go fmt命令，更多时候是用gofmt，而且需要参数-w，否则格式化结果不会写入文件。gofmt -w src，可以格式化整个项目。

### go get

go get 命令主要是用来动态获取远程代码包的，目前支持的有BitBucket、GitHub、Google Code和Launchpad。这个命令在内部实际上分成了两步操作：第一步是下载源码包，第二步是执行go install。下载源码包的go工具会自动根据不同的域名调用不同的源码工具，对应关系如下：

BitBucket (Mercurial Git)

GitHub (Git)

Google Code Project Hosting (Git, Mercurial, Subversion)

Launchpad (Bazaar)

所以为了go get 能正常工作，你必须确保安装了合适的源码管理工具，并同时把这些命令加入你的PATH中。其实go get支持自定义域名的功能，具体参见go help remote。

go get 命令本质上可以理解为：首先通过源码工具clone代码到src目录，然后执行go install。

### go install

go install 命令在内部实际上分成了两步操作：第一步是生成结果文件(可执行文件或者.a包)，第二步会把编译好的结果移到 $GOPATH/pkg 或者 $GOPATH/bin。

.exe文件： 一般是 go install 带main函数的go文件产生的，有函数入口，所有可以直接运行。

.a应用包： 一般是 go install 不包含main函数的go文件产生的，没有函数入口，只能被调用。

### go test

go test 命令，会自动读取源码目录下面名为\*\_test.go的文件，生成并运行测试用的可执行文件。输出的信息类似

ok archive/tar 0.011s

FAIL archive/zip 0.022s

ok compress/gzip 0.033s

默认的情况下，不需要任何的参数，它会自动把你源码包下面所有test文件测试完毕，当然你也可以带上参数，详情请参考go help testflag

### go doc

go doc 命令其实就是一个很强大的文档工具。

如何查看相应package的文档呢？ 例如builtin包，那么执行go doc builtin；如果是http包，那么执行go doc net/http；查看某一个包里面的函数，那么执行godoc fmt Printf；也可以查看相应的代码，执行godoc -src fmt Printf；

通过命令在命令行执行 godoc -http=:端口号 比如godoc -http=:8080。然后在浏览器中打开127.0.0.1:8080，你将会看到一个golang.org的本地copy版本，通过它你可以查询pkg文档等其它内容。如果你设置了GOPATH，在pkg分类下，不但会列出标准包的文档，还会列出你本地GOPATH中所有项目的相关文档，这对于经常被限制访问的用户来说是一个不错的选择。

### 其他命令

Go语言还提供了其它有用的工具，例如下面的这些工具

go fix 用来修复以前老版本的代码到新版本，例如go1之前老版本的代码转化到go1

go version 查看go当前的版本

go env 查看当前go的环境变量

go list 列出当前全部安装的package

go run 编译并运行Go程序

6.5 总结

1. 工作区：特定的一个目录，此目录包含src,pkg,bin目录
2. GOPATH设置，GOPATH为工作区目录（src上一级目录）
3. 包，public: ⾸字母⼤写，可被包外访问private: ⾸字母⼩写，仅包内成员可以访问，同一个目录下不能定义不同的package。
4. init函数，导入包先执行包中init函数再执行main函数中语句，如果同级别不用导入包，会在调用同级别包函数时先执行init函数，或者使用 import \_ "fmt"
5. 命令行

7.复合类型（容器）

7.1 分类

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类型** | **名称** | **长度** | **默认值** | **说明** |
| pointer | 指针 |  | nil |  |
| array | 数组 |  | 0 |  |
| slice | 切片 |  | nil | 引⽤类型 |
| map | 字典 |  | nil | 引⽤类型 |
| struct | 结构体 |  |  |  |

7.2 指针

      指针是一个代表着某个**内存地址的值**。这个内存地址往往是在内存中存储的另一个变量的值的起始位置。Go语言对指针的支持介于Java语言和C/C++语言之间，它既没有像Java语言那样**取消了代码对指针的直接操作的能力**，也避免了C/C++语言中由于对**指针的滥用而造成的安全和可靠性问题**。

7.2.1 基本操作

Go语言虽然保留了指针，但与其它编程语言不同的是：

 默认值 nil，没有 NULL 常量

 操作符 "&" 取变量地址， "\*" 通过指针访问目标对象

不支持指针运算，不支持 "->" 运算符，直接⽤ "." 访问目标成员

不要操作非法内存，指针要指向合法内存

7.2.2 new函数

new(T)函数返回 T类型的指针，go语言只管申请内存，不用释放,有gc垃圾回收

7.2.3 指针做函数参数

指针做函数参数为引用传递，

指针做函数参数，修改的是内存地址中变量的值

func change\_value(a, b \*int){

\*a ,\*b = \*b, \*a

}

7.2.4 数组指针和指针数组

指针数组 var arr \* [5]int 定义一个指针变量，该指针指向一个int类型的数组

数组指针 var arr [5] \*int 定义一个数组,该数组中每个元素是int \* 类型的指针

7.3 数组

7.3.1 数组概述

数组是指一系列同一类型数据的集合。数组中包含的每个数据被称为数组元素（element），一个数组包含的元素个数被称为数组的长度。

数组长度必须是常量(数组的长度是固定的，不能修改长度)，[2]int 和 [3]int 是不同类型

    var n int = 10

    var a [n]int  //err, non-constant array bound n

    var b [10]int //ok

7.3.2 操作数组

 数组的每个元素可以通过索引下标来访问，从0开始

内置函数 len(长度) 和 cap(容量) 都返回数组⻓度 (元素数量)：

    a := [10]int{}

    fmt.Println(len(a), cap(a))//10 10

初始化

  a := [3]int{1, 2}           // 未初始化元素值为 0

    b := [...]int{1, 2, 3}      // 通过初始化值确定数组长度

    c := [5]int{2: 100, 4: 200} // 通过索引号初始化元素，未初始化元素值为 0

    fmt.Println(a, b, c)        //[1 2 0] [1 2 3] [0 0 100 0 200]

多维数组

 d := [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}

 e := [...][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}} //第二维不能写"..."

 f := [4][2]int{1: {20, 21}, 3: {40, 41}}

 g := [4][2]int{1: {0: 20}, 3: {1: 41}}

有多少个[]就是多少维数组，有多少[]就用多少循环

数组的比较

只支持 == 或者 != 比较，数组的类型必须相同

### go随机数

import "math/rand"

// 设置随机数种子，如果种子参数一样，每一次运行得到的随机数都一样

rand.Seed(666)

// 用系统当前时间做随机数种子，

rand.Seed(time.Now().UnixNano())

for i := 0;i < 10; i++{

// 随机数为 100 以内的值， 用 rand.Intn(100)

fmt.Println("rand = ",rand.Intn(100))

}

冒泡排序

for i := 0; i < n - 1; i++{

for j := 0; j < n - 1 - i; j++{

if arr1[j] < arr1[j + 1]{

arr1[j],arr1[j+1] = arr1[j+1],arr1[j]

}

}

}

### 7.3.3 数组做函数参数

以值的方式传递：

func modify(array [5]int)

根据内存和性能来看，在函数间传递数组是一个开销很大的操作。在函数之间传递变量时，总是以值的方式传递的，在函数内部修改数组的值，不影响外部

以地址的方式传递：

func modify(array \*[5]int)

以地址传递，指针指向数组的首地址，用指针修改数组的值是修改了内存地址中的值，函数内修改影响外部（外部访问数组是修改以后的值）

数组的缺点：

1. 数组在初始化时，大小是固定的，不能动态增加容量
2. 值传递需要完整的复制初始数组并将这份拷贝放到栈中，这将耗费大量运行时间，因而值传递方式的效率比较低。
3. 编译程序需要专门产生一部分用来复制初始数组的代码，这将使程序变大。

如何改进：

1. 利用指针，即引用传递
2. 使用切片，因为切片是引用类型，默认会使用引用传递。

7.4 slice 切片

7.4.1 概述

数组和切片的区别

数组[ ]里面的长度是固定的一个常量，数组不能修改长度，lenth和capacity都是固定的

切片[ ]里面为空或者... 切片的lenth和capacity可以不固定

make定义切片可以指定cap

切片并不是数组或数组指针，通过内部指针和引用数组片段来实现变长方案。

切片有3个字段，这 3 个字段分别是

指向底层数组的指针

切片访问的元素的个数(即长度)

切片允许增长 到的元素个数(即容量)

7.4.2 切片的创建和初始化

传统初始化

    var s1 []int //声明切片和声明array一样，只是少了长度，此为空(nil)切片

    s2 := [...]int{}

make函数，make( type, len, cap)

slice2 := make([]int,5,10)

slice3 := make([]int,10) //未指定容量，则容量与长度相等

7.4.3 切片的操作

1. **切片的截取，切片和数组的底层关系**

切片的截取创建新的切片，**底层只有一份切片**

s1 := []int {0,1,2,3,4,5,6,7}

s2 := s1[2:4] //切片的截取

fmt.Printf("s1: %v\ns2:%v\n",s1,s2)

s2[1] = 100

fmt.Printf("s1: %v\ns2:%v\n",s1,s2)

output:

s1: [0 1 2 3 4 5 6 7]

s2:[2 3]

s1: [0 1 2 100 4 5 6 7]

s2:[2 100]

1. **切片的追加**

**append**

append函数向 slice 尾部添加数据，返回新的 slice 对象

    var s1 []int //创建nil切换

    //s1 := make([]int, 0)

    s1 = append(s1, 1)       //追加1个元素

    s1 = append(s1, 2, 3)    //追加2个元素

append函数会智能地底层数组的容量增长，一旦超过原底层数组容量，通常以2倍容量重新分配底层数组，并复制原来的数据：

1. 切片的插入
2. 切片的删除
3. 切面的拷贝

copy

copy(dst, src)

slice的增删改查

https://www.cnblogs.com/vipzhou/p/6362832.html

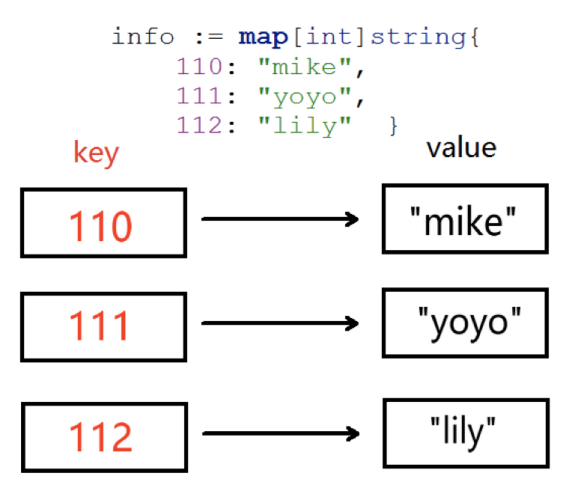
7.4.4 切片做函数参数

切片是引用传递

func slice\_init(s1 []int)

7.5 map 映射、字典

Go语言中的map(映射、字典)是一种内置的数据结构，它是一个无序的key—value对的集合，比如以身份证号作为唯一键来标识一个人的信息。



map格式为：

map[keyType]valueType

map 只有lenth，没有capacity

map是一个**无序**的key—value对的集合

在一个map里所有的键都是**唯一**的,而且必须是支持==和!=操作符的类型,切片、函数以及包含切片的结构类型这些类型由于具有引用语义，不能作为映射的key

7.5.1 创建、初始化

var m1 map[int]string

fmt.Println(m1," lenth = ",len(m1))

m2 := make(map[int]string,5)

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

m3 := map[int]string {1:"aaa",2:"bbb",3:"ccc"}

fmt.Println("m3 = ",m3)

注意

map中的key可以是任何值，可以是内置类型也可以是结构类型，只要这个值可以使用==运算符做比较，切片、函数以及包含切薄片的结构类型由于具有引用语意，不能作为map的key。

在初始化中键必须是唯一的

赋值在遇到key值存在时，视为修改，key不存在时视为添加新的key-value

7.5.2常用操作

**赋值**

  m1 := map[int]string{1: "mike", 2: "yoyo"}

    m1[1] = "xxx"   //修改

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

**遍历**

  m1 := map[int]string{1: "mike", 2: "yoyo"}

    //迭代遍历1，第一个返回值是key，第二个返回值是value

    for k, v := range m1 {

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

    }

//查找相应的元素是否存在 ok是bool类型

vlue,ok := m5[2]

fmt.Println("value:",vlue," ok:",ok)

**删除**

go内置了delete函数

delete(m1, 2) //删除key值为3的map

**清空**

没有为 map 提供任何清空所有元素的函数、方法。清空 map 的唯一办法就是重新 make 一个新的 map。不用担心垃圾回收的效率，Go 语言中的并行垃圾回收效率比写一个清空函数高效多了。

**判断key是否存在**

if \_, ok := m6[3];ok{  
 fmt.Println("is contains")  
}else{  
 fmt.Println("doesn't contains")  
}

**sync.Map（在并发环境中使用的map）**

map 在并发情况下，只读是线程安全的，同时读写线程不安全。

<http://c.biancheng.net/view/34.html>

7.5.3 map做函数参数

引用传递

func map\_delete(m map[int] string){

delete(m,2)

}

func map\_test04(){

m1 := map[int] string {1:"aaa",2:"bbb",3:"ccc"}

map\_delete(m1)

fmt.Println(m1)

}

7.6 结构体

有时我们需要将不同类型的数据组合成一个有机的整体，通过用自定义的方式形成新的类型

结构体成员是由一系列的成员变量构成，这些成员变量也被称为“字段”。字段有以下特性：

1. 字段拥有自己的类型和值。

2. 字段名必须唯一。

3. 字段的类型也可以是结构体，甚至是字段所在结构体的类型。

结构体的定义只是一种内存布局的描述，只有当结构体实例化时，才会真正地分配内存。因此必须在定义结构体并实例化后才能使用结构体的字段。

实例化就是根据结构体定义的格式创建一份与格式一致的内存区域，结构体实例与实例间的内存是完全独立的。

7.6.1 结构体初始化以及使用

普通变量

// 1. 顺序初始化

var s1 student = student{1,"xiaohong",20}

//2、指定初始化某个成员，没有初始化的成员为零值

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

 var s2 Student

    s2.id = 2

    s2.name = "yoyo"

    s2.sex = 'f'

    s2.age = 16

指针变量

s3 := &student{id:3,name:"xxx"}

//===============结构体变量为指针变量

//先分配空间，再赋值

    s3 := new(Student)

    s3.id = 3

    s3.name = "xxx"

fmt.Println("name: ",(\*s3).name)

**对结构体进行&取地址操作时，视为对该类型进行一次 new 的实例化操作。**

type Command struct {

Name string // 指令名称

Var \*int // 指令绑定的变量

Comment string // 指令的注释

}

var version int = 1

cmd := &Command{}

cmd.Name = "version"

cmd.Var = &version

cmd.Comment = "show version"

初始化匿名结构体

msg := &struct { // 定义部分

id int

data string

}{ // 值初始化部分

1024,

"hello",

}

7.6.2 结构体比较

结构体比较只能使用 == 或 != 不可使用大于小于

s1 := student{2,"xiaohong",20}

s2 := student{2,"xiaohong",22}

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

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

7.6.3 结构体作为函数参数

值传递

函数内部修改，在函数外部不改变

func modify\_struct1(s student){

s.age = 100

fmt.Println("in s = ",s)

}

指针传递

函数内部修改，函数外同样改变，因为修改了内存地址重的值

func modify\_struct2(s \*student){

(\*s).age = 200

fmt.Println("in pointer = ",\*s)

}

7.6.4 可见性

如果想使用别的包的函数、结构体类型、结构体变量

函数名，类型名，结构体成员变量名，首字母必须大写，外部才可见，如果首字母是小写，只能在同一个包里使用

7.6.5 构造函数

其他编程语言构造函数的一些常见功能及特性如下：

@ 每个类可以添加构造函数，多个构造函数使用函数重载实现。

@ 构造函数一般与类名同名，且没有返回值。

@ 构造函数有一个静态构造函数一般用这个特性来调用父类的构造函数。

@ 对于 [C++](http://c.biancheng.net/cplus/" \t "/Users/ouchouchou/Documents\\x/_blank) 来说，还有默认构造函数、拷贝构造函数等。

<http://c.biancheng.net/view/68.html>

7.6.6结构体数组

persion\_group := [5]persion{{"aaa", 10, 'x'}, {"bbb", 20, 'm'}, {"ccc", 30, 'x'}}  
fmt.Println("persion\_group[1] = ", persion\_group[1])

7.6.7结构体切片

persion\_slice := make([]persion, 2)  
persion\_slice = append(persion\_slice, persion{"fff", 11, 'm'})

7.6.8 结构体作map的value

## 7.7 container/list 双向链表

双向链表是一种非连续存储的容器，由多个节点组成，节点通过一些变量记录彼此之间的关系。列表有多种实现方法，如单链表、双链表等。

### 7.7.1 初始化列表

list 的初始化有两种方法：New 和声明。两种方法的初始化效果都是一致的。  
1) 通过 container/list 包的 New 方法初始化 list

变量名 := list.New()

2) 通过声明初始化list

var 变量名 list.List

列表与切片和 map 不同的是，列表并没有具体元素类型的限制。因此，列表的元素可以是任意类型。这既带来遍历，也会引来一些问题。给一个列表放入了非期望类型的值，在取出值后，将 interface{} 转换为期望类型时将会发生宕机。

**初始化或清除列表l。**

func (l \*[List](" \l "List)) Init() \*[List](" \l "List)

### 7.7.2 插入元素

双链表支持从队列前方或后方插入元素，分别对应的方法是 PushFront 和 PushBack。

|  |  |
| --- | --- |
| **方  法** | **功  能** |
| InsertAfter(v interface {}, mark \* Element) \* Element | 在 mark 点之后插入元素，mark 点由其他插入函数提供 |
| InsertBefore(v interface {}, mark \* Element) \*Element | 在 mark 点之前插入元素，mark 点由其他插入函数提供 |
| PushBackList(other \*List) | 添加 other 列表元素到尾部 |
| PushFrontList(other \*List) | 添加 other 列表元素到头部 |

### 7.7.3 删除元素

[func (l \*List) Remove(e \*Element) interface{}](" \l "List.Remove)

### **7.7.4 遍历**

for i := l.Front(); i != nil; i = i.Next() {

fmt.Println(i.Value)

}

### **7.7.5总揽**

[type Element](" \l "Element)

[func (e \*Element) Next() \*Element](" \l "Element.Next)

[func (e \*Element) Prev() \*Element](" \l "Element.Prev)

[type List](" \l "List)

[func New() \*List](" \l "New)

[func (l \*List) Back() \*Element](" \l "List.Back)

[func (l \*List) Front() \*Element](" \l "List.Front)

[func (l \*List) Init() \*List](" \l "List.Init)

[func (l \*List) InsertAfter(v interface{}, mark \*Element) \*Element](" \l "List.InsertAfter)

[func (l \*List) InsertBefore(v interface{}, mark \*Element) \*Element](" \l "List.InsertBefore)

[func (l \*List) Len() int](" \l "List.Len)

[func (l \*List) MoveAfter(e, mark \*Element)](" \l "List.MoveAfter)

[func (l \*List) MoveBefore(e, mark \*Element)](" \l "List.MoveBefore)

[func (l \*List) MoveToBack(e \*Element)](" \l "List.MoveToBack)

[func (l \*List) MoveToFront(e \*Element)](" \l "List.MoveToFront)

[func (l \*List) PushBack(v interface{}) \*Element](" \l "List.PushBack)

[func (l \*List) PushBackList(other \*List)](" \l "List.PushBackList)

[func (l \*List) PushFront(v interface{}) \*Element](" \l "List.PushFront)

[func (l \*List) PushFrontList(other \*List)](" \l "List.PushFrontList)

[func (l \*List) Remove(e \*Element) interface{}](" \l "List.Remove)

## 7.8 string

**string**

// 1. string默认为"" 不为nil

// 2. string类型不能赋值为nil

// 3. string类型不能与nil做比较,类型不匹配

// 4. 使用下标可以访问string类型中的某个字节，字符串下标不可以被赋值,不能⽤序号获取字节元素指针， &s[i] ⾮法,要修改字符串，可先将其转换成 []rune 或 []byte，完成后再转换为 string。⽆论哪种转换，都会重新分配内存，并复制字节数组。

7.9总结 最后三点有待补充

1. 指针
   1. 一般用在函数传递参数，指针做函数参数，函数内部改变的是没存地址的值
   2. 可以用new来给指针申请空间，不必释放空间
2. 数组
   1. 值类型
   2. 同一个类型的集合
   3. 数组长度必须是常量
   4. 数组可以部分初始化，不初始化默认为0
   5. len()函数可以查看数组长度
   6. 数组做函数参数传递是值拷贝
3. slice
   1. 引用类型
   2. slice维护3个字段，指向底层数组的指针，长度，容量
   3. newslice := slice[i:j:k] 或[2:3:4] 长度 j-i或 3-2=1 容量 k-i 或 4-2=2,原始切片的容量会限制newslice的元素长度
   4. cap(), len()，append(),copy()等函数操作，用make函数创建时，cap必须大于len，否则报错, append是深拷贝，动态增加数组长度和容量，重新分配空间，再把数据拷贝到新的内存中
   5. slice与底层数组的关系，切片赋值是浅拷贝，
   6. slice是引用传递。
   7. nil切片是创建未初始化的slice
4. map
   1. 引用类型
   2. 无序的key—value对的集合，map中所有key都是唯一的,如果key值已存在，重新赋值对应的value则会被修改
   3. map[key type] value type,
   4. Make指定map中的lenth，如果实际大于指定的，map会自动扩容， 切片、函数、map以及包含切片的结构类型这些类型由于具有引用语义不可以做map key,
   5. **for** k, v := **range** m1 遍历
   6. value, ok := m1[1] 判断某个值是否存在
   7. delete()函数删除
   8. map做函数参数引用传递
   9. Map[key]value value不能为slice
   10. 修改map的value时， 需要修改key对应value的对象，不可以使用m[1].name = “tom”
5. struct
   1. 值类型
   2. 将属性结合起来成为一个整体
   3. 顺序初始化 var s1 Student = Student{1, "mike", 'm', 18, "sz"}
   4. 指定初始化 s4 := Student{id: 2, name: "lily"}
   5. 结构体指针变量
   6. 结构体比较只能 == 或 != 运算符进行比较，但不支持 > 或 <
   7. 结构体做函数参数默认值传递，可以引用传递
   8. 可见性函数名，类型名，结构体成员变量名，首字母必须大写，外部才可见
   9. 对结构体&相当于调用new()在堆上分配内存
6. container/list
7. Make append copy len cap new delete在不同数据类型中的使用

7.1 make 只能创建slice、map和channel

7.2 append slice map

7.3 delete map

7.4 copy

1. 各种数据类型做函数参数

引用传递： pointer slice map   
值传递： array string struct

8.面向对象编程

8.1 概述

Go语言并没有沿袭传统面向对象编程中的诸多概念，比如继承(不支持继承，尽管匿名字段的内存布局和行为类似继承，但它并不是继承)、虚函数、构造函数和析构函数、隐藏的this指针等。

尽管Go语言中没有封装、继承、多态这些概念，但同样通过别的方式实现这些特性：

l 封装：通过方法实现

l 继承：通过匿名字段实现

l 多态：通过接口实现

8.2 匿名组合

type Persion struct{

name string

sex byte

age int

}

type Student struct {

Persion // 匿名字段 只有类型，没有名字，继承了Persion的成员

id int

addr string

}

8.2.1 顺序初始化

var s1 Student = Student{Persion{"xiaoming",'m',20},0001,"bj"}

fmt.Println(s1)

s2 := Student{Persion{"xiaohong",'m',24},12,"hb"}

fmt.Printf("s2 = %+v \n",s2)

8.2.2 成员操作

var s3 Student

s3.name = "mark"

s3.sex = 'm'

s3.age = 30

fmt.Printf("s3 = %+v \n",s3)

8.2.3 同名字段

type Persion struct{

name string //匿名区存在name 属性

sex byte

age int

}

type Student struct {

Persion // 匿名字段 只有类型，没有名字，继承了Persion的成员

id int

addr string

name string //结构体中也存在 name 属性

}

func niming\_test01(){

var s3 Student

s3.name = "mark" //操作的是student中的name 还是 Persion 中的name

s3.sex = 'm'

s3.age = 30

}

同名字段

默认就近原则，如果能在本作用域内找到此成员属性，那就操作本作用域内，未找到则找继承的字段，使用Student中的name同名属性

如果想操作匿名字段中的成员属性，则使用点的方式来进行初始化或赋值

s3.Persion.name = "yoyo"

8.2.4 其他匿名字段

**非结构体匿名字段**

type mystr string //自定义类型

type Student struct {

Persion // 结构体匿名字段 只有类型，没有名字，继承了Persion的成员

id int

mystr // 非结构体匿名字段

}

func main() {

    s1 := Student{Person{"mike", 'm', 18}, 1, "bj"}

    //{Person:{name:mike sex:109 age:18} int:1 mystr:bj}

}

结构体指针

type Person struct { //人

    name string

    sex  byte

    age  int

}

type Student struct { //学生

    \*Person // 匿名字段，结构体指针类型

    id      int

    addr    string

}

func test(){

//初始化 添加去地址符号

    s1 := Student{&Person{"mike", 'm', 18}, 1, "bj"}

  //声明变量

    var s2 Student

    s2.Person = new(Person) //分配空间

    s2.name = "yoyo"

    s2.sex = 'f'

    s2.age = 20

}

继承中匿名指针字段的初始化需要给基类分配内存，在栈上 或者用new在堆上分配内存

8.3 方法

8.3.1 概述

带有接收者的函数叫做方法

语法

func (r ReceiverType) funcName(parameters) (results)

l 参数 r 官方建议使用接收器类型名的第一个小写字母。如⽅法中未曾使⽤，可省略参数名。

l 参数 r 类型可以是 T 或 \*T，基类型 T 不能是接⼝或指针？？。指针类型的接收器由一个指针类型组成，更接近于面向对象中的 this 或者 self。非指针接收器会在代码运行时将接收器的值复制一份。在非指针接收器的方法中可以获取接收器的成员值，但修改后无效。指针接收器由于指针的特性，调用方法时，修改接收器指针的任意成员变量，在方法结束后，修改都是有效的。

l 不支持重载方法，也就是说，不能定义名字相同但是不同参数的方法。

ReceiverType本身不可以是指针

func (r \*ReceiverType) funcName(parameters) (results) //这样可以

方法和函数的区别

func test(){ // 普通函数

func (){ // 匿名函数

}()

}

// 带有接收者函数的叫做方法

func (temp Students) Printf\_info(){

fmt.Println(temp)

}

// 通过一个函数给成员属性初始化

func (p \*Students)Set\_info(n string, a int, s byte){

p.name = n

p.age = a

p.sex = s

}

var s3 Students

(&s3).Set\_info("xiaoxiao",30,'f')

s3.Printf\_info()

8.3.2 为类型添加方法

8.3.2.1 基础类型做为接收者

type myInt int

func (a myInt) Add(b myInt)(ret myInt){

return a + b

}

func Struct\_test11(){

var v1 myInt = 10

var v2 myInt = 20

fmt.Println("v1 + v2 = ",v1.Add(v2))

}

8.3.2.2 结构体做为接受者

type Students struct{

name string

age int

sex byte

}

func (temp Students) Printf\_info(){

fmt.Println(temp)

}

func (p \*Students)Set\_info(n string, a int, s byte){

p.name = n

p.age = a

p.sex = s

}

func Struct\_test01(){

s1 := Students{"xiaohong",22,'m'}

s1.Printf\_info()

var s2 Students

s2.Set\_info("feifei",24,'m')

s2.Printf\_info()

var s3 Students

(&s3).Set\_info("xiaoxiao",30,'f')

s3.Printf\_info()

}

8.3.3 值语义和引用语义

通俗说就是值传递和引用传递

type Students struct{

name string

age int

sex byte

}

//值作为接收者，值语义

func (p Students)Set\_info\_value(n string, a int, s byte){

p.name = n

p.age = a

p.sex = s

}

//指针作为接收者，引用语义

func (p \*Students)Set\_info\_pointer(n string, a int, s byte){

p.name = n

p.age = a

p.sex = s

}

func Struct\_test12(){

var s1 Students

var s2 Students

//引用语义

s1.Set\_info\_pointer("xiaohong",29,'f')

//值语义

s2.Set\_info\_value("xiaoming",23,'m')

s1.Printf\_info()

s2.Printf\_info()

}

output:

{xiaohong 29 102}

{ 0 0}

8.3.4 方法集

类型的方法集是指可以被该类型的值调用的所有方法的集合。

用实例实例 value 和 pointer 调用方法（含匿名字段）**不受⽅法集约束**，编译器编总是查找全部方法，并自动转换 receiver 实参。

8.3.4.1 类型 \*T 方法集

一个指向自定义类型的值的指针，它的方法集由该类型定义的所有方法组成，**无论这些方法接受的是一个值还是一个指针。**

如果在指针上调用一个接受值的方法，Go语言会聪明地将该指针解引用，并将指针所指的底层值作为方法的接收者。

type Persions struct{

name string

age int

sex byte

}

//值作为接收者，值语义

func (p Persions)Printf\_info\_value(){

fmt.Println("Printf\_info\_value")

}

//指针作为接收者，引用语义

func (p \*Persions)Printf\_info\_pointer(){

fmt.Println("Printf\_info\_pointer")

}

func fangfaji\_test01(){

var s3 \*Persions = &Persions{"ff",24,'f'}

s3.Printf\_info\_value()

s3.Printf\_info\_pointer()

}

8.3.4.2 类型 T 方法集

一个自定义类型值的方法集则由为该类型定义的接收者类型为值类型的方法组成，但是不包含那些接收者类型为指针的方法。

但这种限制通常并不像这里所说的那样，因为如果我们只有一个值，仍然可以调用一个接收者为指针类型的方法，这可以借助于Go语言传值的地址能力实现。

type Persions struct{

name string

age int

sex byte

}

//值作为接收者，值语义

func (p Persions)Printf\_info\_value(){

fmt.Println("Printf\_info\_value")

}

//指针作为接收者，引用语义

func (p \*Persions)Printf\_info\_pointer(){

fmt.Println("Printf\_info\_pointer")

}

func fangfaji\_test02(){

var s2 Persions = Persions{"xiaohong",22,'m'}

s2.Printf\_info\_value()

s2.Printf\_info\_pointer()

}

8.3.5 匿名字段

8.3.5.1 方法的继承

**如果匿名字段实现了一个方法，那么包含这个匿名字段的struct也能调用该方法。**

type Persions struct{

name string

age int

sex byte

}

func (p Persions)Printf\_info\_value(){

fmt.Println(p)

}

type Student\_02 struct{

Persions

id int

}

func fangfaji\_test01(){

var p1 Persions = Persions{"xiaofei",24,'f'}

p1.Printf\_info\_value()

var s1 Student\_02 = Student\_02{Persions{"feifei",22,'m'},5}

s1.Printf\_info\_value()

}

output

{xiaofei 24 102}

{feifei 22 109} //只能按照Persion类型输出

8.3.5.2 方法重写

type Persions struct{

name string

age int

sex byte

}

func (p Persions)Printf\_info(){

fmt.Println(p)

}

type Student\_02 struct{

Persions

id int

}

// 方法重写

// Student\_02继承Persion 并重写了Printf\_info()方法

func (s Student\_02)Printf\_info(){

fmt.Println(s)

}

func fangfaji\_test01(){

var p1 Persions = Persions{"xiaofei",24,'f'}

p1.Printf\_info\_value()

var s1 Student\_02 = Student\_02{Persions{"feifei",22,'m'},5}

s1.Printf\_info()

}

8.3.6 表达式

类似于我们可以对函数进行赋值和传递一样，方法也可以进行赋值和传递。

根据调用者不同，方法分为两种表现形式：方法值和方法表达式。两者都可像普通函数那样赋值和传参，区别在于方法值绑定实例，⽽方法表达式则须显式传参。

方法值

var p1 Persion\_01

// 方法值，保存了方法的入口地址，隐藏了接收者，

pFunc01 := p1.Set\_info\_value

// 调用时无需再传递接收者

// 等价于 p1.Set\_info\_value

pFunc01("xiaohong",22,'m')

方法表达式

var p2 Persion\_01

// 通过类型名点出方法 pFunc02保存方法入口地址

pFunc02 := (\*Persion\_01).Set\_info\_value

// 调用时需要 显示传递参数

pFunc02(&p2,"fff",33,'m') // 第一个参数为类型变量，第二个以后为函数参数列表

8.4 接口

8.4.1 概述

在Go语言中，接口(interface)是一个自定义类型，接口类型具体**描述了一系列方法的集合**

接口类型是一种抽象的类型，它不会暴露出它所代表的对象的内部值的结构和这个对象支持的基础操作的集合，它们只会展示出它们自己的方法。因此**接口类型不能将其实例化。**

接口底层实现：

runtime.h

struct Iface

{

Itab\* tab;

void\* data;

};

struct Itab

{

InterfaceType\* inter;

Type\* type;

void (\*fun[])(void);

};

type User struct {

id int

name string

}

接⼝对象由接⼝表 (interface table) 指针和数据指针组成。

接⼝表存储元数据信息，包括接⼝类型、动态类型，以及实现接⼝的⽅法指针。⽆论是反

射还是通过接⼝调⽤⽅法，都会⽤到这些信息。

数据指针持有的是目标对象的只读复制品，复制完整对象或指针。

只有 tab 和 data 都为 nil 时，接⼝才等于 nil。

8.4.2 接口的使用

8.4.2.1接口定义

type Humaner interface {

    SayHi()

}

l 接⼝命名习惯以 er 结尾

l 接口只有方法声明，没有实现，没有数据字段

l 接口**可以匿名嵌入其它接口，或嵌入到结构中**

8.4.2.2 接口实现

// 定义接口

type Humaner interface{

SayHi()

}

type Stu struct {

name string

age int

}

type Tha struct {

name string

group string

}

type MyStr string

// 实现接口

func (s \* Stu)SayHi(){

fmt.Printf("name:%s age:%d say hi...\n",s.name,s.age)

}

// 实现接口

func (t \* Tha)SayHi(){

fmt.Printf("name:%s group:%s say hi...\n",t.name,t.group)

}

// 实现接口

func (m \* MyStr)SayHi(){

fmt.Printf("%s say hi... \n",\*m)

}

func interface\_test01(){

s := &Stu{"xiaohong",22}

t := &Tha{"fengge","bj"}

var tmp MyStr = "golang"

s.SayHi()

t.SayHi()

tmp.SayHi()

}

func interface\_test02(){

s := &Stu{"xiaohong",22}

t := &Tha{"fengge","bj"}

var tmp MyStr = "golang"

slice\_01 := make([]Humaner,3)

slice\_01[0] = s

slice\_01[1] = t

slice\_01[2] = &tmp

for \_, x := range slice\_01{

WhoSayHi(x)

}

}

func interface\_var\_test03(){

// 定义接口类型的变量

var i Humaner

s := &Stu{"xiaohong",20}

i = s

i.SayHi()

t := &Tha{"fengge","bj"}

i = t

i.SayHi()

}

8.4.3 多态

定义了一个普通函数，函数的参数为接口类型

// 定义一个接口类型以及多个对象类型

// 对象实现接口类型 sayhi方法

// 实现多态函数，参数为接口类型，内部调用该类型实现的sayHi方法，实现调用同一个方法传入不同的对象展现多种形态

func WhoSayHi(i Humaner){

i.SayHi()

}

func interface\_test02(){

s := &Stu{"xiaohong",22}

t := &Tha{"fengge","bj"}

var tmp MyStr = "golang"

WhoSayHi(s)

WhoSayHi(t)

WhoSayHi(&tmp)

}

output

name:xiaohong  age:22 say hi...

name:fengge group:bj say hi...

golang say hi...

8.4.4 接口组合

8.4.3.1 接口嵌用（接口的继承）

type Humaner interface{ //子集接口

SayHi()

}

type Persion\_03 interface{ //超集接口

Humaner // 继承了匿名字段

Sing(lrc string)

}

type Student\_03 struct{

name string

id int

}

func (temp \* Student\_03) SayHi(){

fmt.Printf("%s SayHi...\n",temp.name)

}

func (temp \* Student\_03) Sing(lrc string){

fmt.Printf("%s 在唱着 %s\n",temp.name,lrc)

}

func qianru\_test01(){

s := &Student\_03{"mark",6}

var i Persion\_03

i = s

i.SayHi() // 继承过来的

i.Sing("xiaopingguo")

}

8.4.3.1 接口转换

超集变量可以转换为子集变量，但子集变量不可以转换超集变量

type Humaner interface{

SayHi()

}

type Persion\_03 interface{

Humaner // 继承了匿名字段

Sing(lrc string)

}

type Student\_03 struct{

name string

id int

}

func (temp \* Student\_03) SayHi(){

fmt.Printf("%s SayHi...\n",temp.name)

}

func (temp \* Student\_03) Sing(lrc string){

fmt.Printf("%s 在唱着 %s\n",temp.name,lrc)

}

func qianru\_test01(){

s := &Student\_03{"mark",6}

var ipro Persion\_03

var i Humaner

ipro = s

i = ipro

i.SayHi() // 类型转换后 只能调用SayHi 原来的Sing不可调用

}

8.4.3.3 空接口

空接口(interface{})不包含任何的方法，所有的类型都实现了空接口，因此空接口可以存储任意类型的数值。它有点类似于C语言的void \*类 型。

    var v1 interface{} = 1     // 将int类型赋值给interface{}

    var v2 interface{} = "abc" // 将string类型赋值给interface{}

    var v3 interface{} = &v2   // 将\*interface{}类型赋值给interface{}

    var v4 interface{} = struct{ X int }{1}

    var v5 interface{} = &struct{ X int }{1}

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

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

    func Println(args ...interface{})

8.4.5 类型查询

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

l comma-ok断言

l switch测试

8.4.5.1 comma-ok 断言

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

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

type Element interface {}

type Student\_11 struct{

name string

age int

}

func test01(){

list := make([]Element,3)

list[0] = 1

list[1] = "hello"

list[2] = Student\_11{"mark",22}

for index,element := range list{

if value,ok := element.(int); ok{

fmt.Printf("list[%d] is an int, value = %d\n",index,value)

}

if value,ok := element.(string); ok{

fmt.Printf("list[%d] is an string, value = %s\n",index,value)

}

if value,ok := element.(Student\_11); ok{

fmt.Printf("list[%d] is an string, value = %s %d\n",index,value.name,value.age)

}

}

}

8.4.5.2 switch

func test02(){

list := make([]Element,3)

list[0] = 1

list[1] = "hello"

list[2] = Student\_11{"mark",22}

for index,element := range list{

switch value := element.(type) {

case int:

fmt.Println("list[0]","is int,value = ",value)

case string:

fmt.Println("list[1]","is string,value = ",value)

case Student\_11:

fmt.Println("list[2]","is student name = ",value.name," age = ",value.age)

default:

fmt.Printf("list[%d] is of a different type",index)

}

}

}

8.5 总结

1. 面向对象的特性

go语言中没有面向对象的概念，但是可以实现面向对象的特性

* 1. 封装：通过方法实现，将成员属性和成员函数封装在一起
  2. 继承：通过匿名字段实现，来实现代码复用
  3. 多态： 通过接口实现，interface只关心行为，而不关心类型

1. 匿名组合
   1. 匿名字段（嵌入字段）， 只有类型没有变量

type Persion struct{

name string

sex byte

}

type Student struct {

Persion // 匿名字段 只有类型，没有名字，继承了Persion的所有字段

id int

}

* + 1. 匿名字段可以是整型，string，结构体 type mystr string
    2. 顺序初始化,指针变量初始化需要对Student类型取地址
    3. 成员操作初始化 s.name == s.Persion.name
    4. 同名字段，继承中的同名变量遵循就近原则，也可以显示调用
    5. 指针类型可以用new来分配空间

1. 方法

func (receiver receiverType) xxx(){}

func (a MyInt) Add(b MyInt) {}

var a , b MyInt = 10,20

a.Add(b) 调用

* + 1. 比一般函数多了接收者，接收者类型可以是基本类型、自定义类型、结构体等
    2. 值语义和引用语义，接收者是值类型还是指针类型，值语义方法调用无法改变内存地址中等值，引用语义可以
    3. GO中没有方法重载(所谓重载，指的是方法名称一致，参数类型，个数不一致)。
    4. 方法继承
    5. 方法重写，子类方法与父类方法重名，就近原则先调用子类方法，可显示调用父类方法
    6. 接收者
       1. 接收者本身不可以为指针类型
          1. Type Myint \*int // 错误
          2. Type Myint int //正确
       2. 接收者为普通变量，非指针，值传递。接收者为指针变量，引用传递
  1. 方法集
     1. 类型的方法集是指可以被该类型的值调用的所有方法的集合
     2. 不管是T类型的方法集还是\*T类型的方法集都可以互相调用内部会隐式转换
  2. 匿名字段方法的继承
     1. **包含这个匿名字段的struct变量也能调用匿名struct中的方法。**
  3. 匿名字段方法的重写
     1. Student\_02继承了Persion 并重写了Printf\_infor\_value()方法
     2. 方法重写二义性，就近原则，如果想调用匿名字段中的方法可以显示调用
  4. 方法值
     1. 将某个**类型值（对象）**的某个方法赋值给函数变量，所以调用时直接调用函数变量即可，隐藏了接受者
  5. 方法表达式
     1. 将某个**类型**的某个方法赋值给函数变量，没有默认是哪个接受者变量的方法，调用该方法需要显示的传递接收者

1. 接口 interface
   1. 重点
      1. 接口类型类似于结构体类型，结构体类型中是变量，而接口类型中是函数
      2. 接口底层是以结构体形式封装，内部包含了接口表指针、数据指针，接口表中存储的是元数据，接口类型、接口动态类型、实现接口的方法指针
      3. 接口只有方法声明，没有方法实现，没有数据字段
      4. 接口可以匿名嵌入其他接口，或嵌入结构体中
   2. 接口的调用方式 三种
      1. 每种类型都可以对这个接口进行实现，也可以给接口变量赋值为该类型，调用可以通过接口变量点的方式
      2. 在每种类型实现接口的基础上，可以用该类型点的方式调用
      3. 将该接口的参数定义为interface，可以直接进行传参调用（实现多态）
   3. 接口嵌入
      1. 接口嵌入，接口中方法多的称为超集合接口，接口中方法少的称为子集接口
      2. 如果一个interface01 作为 interface02的一个嵌入字段，那么interface02就包含了interface01中的方法，当某个类型实现该方法的前提下，可以调用两个interface中的任意接口
   4. 接口转换
      1. 超集变量可以转换为子集变量，但子集变量不可以转换超集变量
   5. 空接口
      1. var v1 interface{} = 1 //可以是int string struct 等
      2. 空接口不包含任何的方法，所以，所有的类型都实现了空接口，空接口可以储存任意类型的数值，类似于c语言中的 void \*类型
      3. 当函数可以接受对象实例时，会将其参数生命为 interface{}，最经典的是fmt.Printf()函数

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

func Println(args ...interface{})

* 1. 类型查询

当函数使用空接口来作为形参时，如何判断传入参数的类型以便后续操作

a. comma-ok 断言

if value,ok := element.(int); ok{

b. switch

switch value := element.(type) {

1. 异常处理

9.1 error接口

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

a. error接口定义

type error interface {

    Error() string

}

b. Go语言的标准库代码包errors为用户提供如下方法

package errors

type errorString struct {

    text string

}

func New(text string) error {

    return &errorString{text}

}

func (e \*errorString) Error() string {

    return e.text

}

c. 可以使用errors.New() 和 fmt.Errorf()

err02 := errors.New("This is err02")

err01 := fmt.Errorf("%s\n","This is normal err")

通常在返回值中返回错误

func Divide(a, b int) (result int, err error) {

    if b == 0 {

        result = 0.0

        err = errors.New("runtime error: divide by zero")

        return

    }

    result = a / b

    err = nil

    return

}

9.2 panic

显示调用 panic函数，当遇到不可修复的错误时应该使用panic函数，我们不应通过调用panic函数来报告普通的错误，而应该只把它作为报告致命错误的一种方式

一般而言，当panic异常发生时，程序会中断运行，并立即执行在该goroutine（可以先理解成线程，在中被延迟的函数（defer 机制）。随后，程序崩溃并输出日志信息。日志信息包括panic value和函数调用的堆栈跟踪信息。

不是所有的panic异常都来自运行时，直接调用内置的panic函数也会引发panic异常；panic函数接受任何值作为参数。

func panic(v interface{})

panic函数调用测试

func panic\_test01(){

fmt.Println("This is panic\_test01()")

}

func panic\_test02(){

//显示调用panic函数 导致程序中断

panic("func panic\_test02(): panic")

}

func panic\_test03(){

fmt.Println("This is panic\_test03()")

}

func panic\_test(){

panic\_test01()

panic\_test02()

panic\_test03()

}

func main(){

panic\_test()

}

output:

s panic\_test01()

panic: func panic\_test02(): panic

goroutine 1 [running]:

main.panic\_test02()

/Users/ouchouchou/learn\_something/golang/wdd/pointer\_test/src/12\_error.go:41 +0x39

main.panic\_test()

/Users/ouchouchou/learn\_something/golang/wdd/pointer\_test/src/12\_error.go:52 +0x25

main.main()

/Users/ouchouchou/learn\_something/golang/wdd/pointer\_test/src/12\_error.go:65 +0x20

exit status 2

9.3 recover

运行时panic异常一旦被引发就会导致程序崩溃。Go语言为我们提供了专用于“拦截”运行时panic的内建函数——recover。它可以是当前的程序从运行时panic的状态中恢复并重新获得流程控制权。

        func recover() interface{}

注意：recover必须在defer调用的函数中有效。

如果调用了内置函数recover，并且定义该defer语句的函数发生了panic异常，recover会使程序从panic中恢复，并返回panic value。导致panic异常的函数不会继续运行，但能正常返回。在未发生panic时调用recover，recover会返回nil。

func recover\_test02(x int){

// 必须在defer关键字后

defer func(){

// 判断是否运行崩溃引发panic异常

if err := recover();err != nil {

fmt.Println(err)

}

}()

var a[10] int

a[x] = 666

}

9.4 总结

1. error

可以使用errors.New() 和 fmt.Errorf()

err02 := errors.New("This is err02")

err01 := fmt.Errorf("%s\n","This is normal err")

1. panic

把它作为报告致命错误的一种方式，当panic异常发生时，程序会中断运行，并立即执行在该goroutine

panic("func panic\_test02(): panic")

1. recover

3.1 必须在defer关键字后

3.2 调用recover必须要在出错之前

defer func(){

// 判断是否运行崩溃引发panic异常

if err := recover();err != nil {

fmt.Println(err)

}

}()

10 文件处理

10.1 字符串处理

可以通过Go标准库中的strings和strconv两个包中的函数进行相应的操作。

10.1.1 字符串操作

下面这些函数来自于strings包，这里介绍一些我平常经常用到的函数，更详细的请参考官方的文档。

1. **Contains**

func Contains(s, substr string) bool

功能：字符串s中是否包含substr，返回bool值

1. **Join**

func Join(a []string, sep string) string

功能：字符串链接，把slice a通过sep链接起来

1. **Index**

func Index(s, sep string) int

功能：在字符串s中查找sep所在的位置，返回位置值，找不到返回-1

1. **Repeat**

func Repeat(s string, count int) string

功能：将s复制count份，返回新的字符串

1. **Replace**

func Replace(s, old, new string, n int) string

功能：在s字符串中，把old字符串替换为new字符串，n表示替换的次数，小于0表示全部替换

1. **Split**

func Split(s, sep string) []string

功能：把s字符串按照sep分割，返回slict

1. **Trim**

func Trim(s string, cutset string) string

功能：在s字符串的头部和尾部去除cutset指定的字符串

1. **Fields**

func Fields(s string) []string

功能：去除s字符串的空格符，并且按照空格分割返回slice

10.1.2 字符串

字符串转化的函数在strconv中，如下也只是列出一些常用的。

1. **Append**

Append 系列函数各种类型转换为字符串后，添加到现有的字节数组中。

[func AppendBool(dst []byte, b bool) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendBool)

[func AppendInt(dst []byte, i int64, base int) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendInt)

[func AppendUint(dst []byte, i uint64, base int) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendUint)

[func AppendFloat(dst []byte, f float64, fmt byte, prec int, bitSize int) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendFloat)

[func AppendQuote(dst []byte, s string) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendQuote)

[func AppendQuoteToASCII(dst []byte, s string) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendQuoteToASCII)

[func AppendQuoteRune(dst []byte, r rune) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendQuoteRune)

[func AppendQuoteRuneToASCII(dst []byte, r rune) []byte](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "AppendQuoteRuneToASCII)

1. **Format**

Format 系列函数把其他类型的转换为字符串。

[func FormatBool(b bool) string](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "FormatBool)

[func FormatInt(i int64, base int) string](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "FormatInt)

[func FormatUint(i uint64, base int) string](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "FormatUint)

[func FormatFloat(f float64, fmt byte, prec, bitSize int) string](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "FormatFloat)

1. **Parse**

Parse 系列函数把字符串转换为其他类型。

[func ParseBool(str string) (value bool, err error)](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "ParseBool)

[func ParseInt(s string, base int, bitSize int) (i int64, err error)](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "ParseInt)

[func ParseUint(s string, base int, bitSize int) (n uint64, err error)](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "ParseUint)

[func ParseFloat(s string, bitSize int) (f float64, err error)](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "ParseFloat)

[func Atoi(s string) (i int, err error)](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "Atoi)

[func Itoa(i int) string](https://studygolang.com/static/pkgdoc/pkg/strconv.htm" \l "Itoa)

10.2 正则表达式

Go实现的是RE2标准，除了\C，详细的语法描述参考：<http://code.google.com/p/re2/wiki/Syntax>

导入包： import "regexp"

如果可以用strings?实现，就不要用正则

` ` 反引号为原生的字符串

10.3 JSON处理

JSON （JavaScript Object Notation）是一种比XML更轻量级的数据交换格式

开发者可以用 JSON 传输简单的字符串、数字、布尔值，也可以传输一个数组，或者一个更复杂的复合结构。在 Web 开发领域中， JSON被广泛应用于 Web 服务端程序和客户端之间的数据通信。

Go语言内建对JSON的支持。使用Go语言内置的encoding/json 标准库，开发者可以轻松使用Go程序生成和解析JSON格式的数据。

JSON官方网站：<http://www.json.org/>

在线格式化：<http://www.json.cn/>

1. JSON 编码

**1.1 通过结构体生成JSON**

使用json.Marshal()函数可以对一组数据进行JSON格式的编码。 json.Marshal()函数的声明如下：

func [Marshal](https://github.com/golang/go/blob/master/src/encoding/json/encode.go?name=release" \l "131)

func Marshal(v interface{}) ([]byte, error)

Marshal函数返回v的json编码。

Marshal函数会递归的处理值。如果一个值实现了Marshaler接口切非nil指针，会调用其MarshalJSON方法来生成json编码。nil指针异常并不是严格必需的，但会模拟与UnmarshalJSON的行为类似的必需的异常。

func [MarshalIndent](https://github.com/golang/go/blob/master/src/encoding/json/encode.go?name=release" \l "141)

func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)

MarshalIndent类似Marshal但会使用缩进将输出格式化。

**二次编译**

type IT struct{

Company string `json:"-"`

Subjects []string `json:"SUBJECTS"`

IsOk bool `json:"isok,omitempty`

Price float64 `json:",omitempty"`

}

// 字段被本包忽略

`json:"-"`

// 字段在json里的键为"myName"

`json:"myName"`

// 字段在json里的键为"myName"且如果字段为空值将在对象中省略掉

`json:"myName,omitempty"`

// 字段在json里的键为"Field"（默认值），但如果字段为空值会跳过；注意前导的逗号

`json:",omitempty"`

// 转化为string

`json:",string"`

1.1 **通过struct生成json**

type IT struct{

Company string `json:"-"`

Subjects []string`json:"SUBJECTS"`

IsOk bool `json:"isok,omitempty"`

Price float64 `json:",omitempty"`

}

// 通过struct生成json

func json\_test01(){

s := IT{"",[]string{"c++","python","golang","test"},true,66.66}

// json.Marshal()

buf,err := json.Marshal(s)

if err != nil{

fmt.Println("marshal err:",err)

return

}

fmt.Println(string(buf))

fmt.Println()

// json.MarshalIndent()

s2 := IT{"itcast",[]string{"c++","python","golang","test"},true,66.66}

buf2,err2 := json.MarshalIndent(s2,""," ")

if err2 != nil{

fmt.Println("MarshalIndet err:",err2)

return

}

fmt.Println(string(buf2))

}

**1.2 通过map生成json**

func json\_test02(){

M := make(map[string]interface{},4)

M["company"] = "itcast"

M["subjecte"] = []string{"go","c++","python","test"}

M["isok"] = true

M["price"] = 666.66

ret,err := json.Marshal(M)

if err != nil{

fmt.Println("map Marshal err")

return

}

fmt.Println(string(ret))

ret2,err2 := json.MarshalIndent(M,""," ")

if err2 != nil{

fmt.Println("map MarshalIndent err")

return

}

fmt.Println(string(ret2))

}

1. JSON 解码

可以使用json.Unmarshal()函数将JSON格式的文本解码为Go里面预期的数据结构。

json.Unmarshal()函数的原型如下：

func Unmarshal(data []byte, v interface{}) error

2.1 解码 struct json

type receive struct{

Company string `json:"company"`

Subjects []string `json:"subjects"`

IsOk bool `json:"isok"`

Price float64 `json:"price"`

}

// 解码 struct json

func struct\_json\_test01(){

json\_buf := `

{

"company": "itcast",

"isok": true,

"price": 666.66,

"subjects": [

"go",

"c++",

"python",

"test"

]

}

var IT receive

err := json.Unmarshal([]byte(json\_buf),&IT)

if err != nil{

fmt.Println("unmarshal err")

return

}

fmt.Printf("%+v\n",IT)

}

2.2 解码interface json

// 解码 map json

func map\_json\_test01(){

json\_buf := `

{

"company": "itcast",

"isok": true,

"price": 666.66,

"subjects": [

"go",

"c++",

"python",

"test"

]

}

`

m := make(map[string]interface{},4)

err := json.Unmarshal([]byte(json\_buf),&m)

if err != nil {

fmt.Println("map json Unmarshal err",err)

return

}

var str1 string

//str1 := m["company"] //err m["company"] 是[]interface{}类型

//str1 := string(m["company"]) //err 强转类型无效

//fmt.Printf("%s\n",str1)

// 使用断言来获取string类型元素

for \_,element := range m{

if value,ok := element.(string);ok{

fmt.Printf("is string =======> %s\n",value)

}

if value,ok := element.(bool) ; ok{

fmt.Printf("is bool =======> %v\n",value)

}

if value,ok := element.(float64) ; ok{

fmt.Printf("is float64 =======> %v\n",value)

}

if value,ok := element.([]interface{}); ok{

fmt.Printf("is []interface{} =======> %v\n",value)

}

}

}

1. encoding/Json

[type InvalidUTF8Error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "InvalidUTF8Error)

* [func (e \*InvalidUTF8Error) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "InvalidUTF8Error.Error)

[type InvalidUnmarshalError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "InvalidUnmarshalError)

* [func (e \*InvalidUnmarshalError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "InvalidUnmarshalError.Error)

[type SyntaxError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "SyntaxError)

* [func (e \*SyntaxError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "SyntaxError.Error)

[type UnmarshalFieldError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnmarshalFieldError)

* [func (e \*UnmarshalFieldError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnmarshalFieldError.Error)

[type UnmarshalTypeError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnmarshalTypeError)

* [func (e \*UnmarshalTypeError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnmarshalTypeError.Error)

[type UnsupportedTypeError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnsupportedTypeError)

* [func (e \*UnsupportedTypeError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnsupportedTypeError.Error)

[type UnsupportedValueError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnsupportedValueError)

* [func (e \*UnsupportedValueError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "UnsupportedValueError.Error)

[type MarshalerError](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "MarshalerError)

* [func (e \*MarshalerError) Error() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "MarshalerError.Error)

[type Number](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Number)

* [func (n Number) Int64() (int64, error)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Number.Int64)
* [func (n Number) Float64() (float64, error)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Number.Float64)
* [func (n Number) String() string](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Number.String)

[type RawMessage](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "RawMessage)

* [func (m \*RawMessage) MarshalJSON() ([]byte, error)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "RawMessage.MarshalJSON)
* [func (m \*RawMessage) UnmarshalJSON(data []byte) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "RawMessage.UnmarshalJSON)

[type Marshaler](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Marshaler)

[type Unmarshaler](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Unmarshaler)

[func Compact(dst \*bytes.Buffer, src []byte) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Compact)

[func HTMLEscape(dst \*bytes.Buffer, src []byte)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "HTMLEscape)

[func Indent(dst \*bytes.Buffer, src []byte, prefix, indent string) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Indent)

[func Marshal(v interface{}) ([]byte, error)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Marshal)

[func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "MarshalIndent)

[func Unmarshal(data []byte, v interface{}) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Unmarshal)

[type Decoder](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Decoder)

* [func NewDecoder(r io.Reader) \*Decoder](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "NewDecoder)
* [func (dec \*Decoder) Buffered() io.Reader](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Decoder.Buffered)
* [func (dec \*Decoder) UseNumber()](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Decoder.UseNumber)
* [func (dec \*Decoder) Decode(v interface{}) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Decoder.Decode)

[type Encoder](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Encoder)

* [func NewEncoder(w io.Writer) \*Encoder](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "NewEncoder)
* [func (enc \*Encoder) Encode(v interface{}) error](https://studygolang.com/static/pkgdoc/pkg/encoding_json.htm" \l "Encoder.Encode)

10.4 文件操作

1. 分类
   1. 文本文件
   2. 二进制文件
2. 创建文件

func Create(name string) (file \*File, err Error)

根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666的文件，返回的文件对象是可读写的。

func NewFile(fd uintptr, name string) \*File

根据文件描述符创建相应的文件，返回一个文件对象

1. 打开文件

func Open(name string) (file \*File, err Error)

该方法打开一个名称为name的文件，但是是只读方式，内部实现其实调用了OpenFile。

func OpenFile(name string, flag int, perm uint32) (file \*File, err Error)

打开名称为name的文件，flag是打开的方式，只读、读写等，perm是权限

1. 写文件

func (file \*File) Write(b []byte) (n int, err Error)

写入byte类型的信息到文件

func (file \*File) WriteAt(b []byte, off int64) (n int, err Error)

在指定位置开始写入byte类型的信息

func (file \*File) WriteString(s string) (ret int, err Error)

写入string信息到文件

1. 读文件

func (file \*File) Read(b []byte) (n int, err Error)

读取数据到b中

func (file \*File) ReadAt(b []byte, off int64) (n int, err Error)

从off开始读取数据到b中

行读

import "bufio"

// 借助 bufio实现行读取

func bufio\_read(path string){

// 1. open file

fp,open\_err := os.Open(path)

if open\_err != nil{

fmt.Println("open err: ",open\_err)

return

}

// 2. read

r := bufio.NewReader(fp)

for {

buf,read\_err := r.ReadBytes('\n')

if read\_err != nil {

if read\_err == io.EOF { // 读到文件结尾

break

}

fmt.Println("read err: ",read\_err)

}

fmt.Printf("buf = #%s#\n",string(buf))

}

// 3. close

close\_err := fp.Close()

if close\_err != nil{

fmt.Println("close err: ",close\_err)

return

}

}

1. 删除文件

func Remove(name string) Error

调用该函数就可以删除文件名为name的文件

1. 文件末尾

len,read\_err := fd.Read(sli)

if read\_err != nil && read\_err == io.EOF{

1. bufio 行读

r := bufio.NewReader(fp)

for {

buf,read\_err := r.ReadBytes('\n')

if read\_err != nil {

if read\_err == io.EOF { // 读到文件结尾

break

}

fmt.Println("read err: ",read\_err)

}

fmt.Printf("buf = #%s#\n",string(buf))

}

9. 总结

1. error接口处理
   1. 可以使用errors.New() 和 fmt.Errorf()
   2. 通常在返回值中用到error
2. panic
   1. 致命的错误调用
   2. 主动调用和内置调用（除数为0、数组越界等）
   3. panic调用会引起程序崩溃，可以通过recover恢复
3. recover
   1. recover会使程序从panic中恢复，并返回panic value。导致panic异常的函数不会继续运行，但能正常返回。在未发生panic时调用recover，recover会返回nil。
   2. 注意：recover只有在defer调用的函数中有效。
4. 字符串处理函数
   1. contains

功能：字符串s中是否包含substr，返回bool值

* 1. join

功能：字符串链接，把slice a通过sep链接起来

* 1. index

功能：在字符串s中查找sep所在的位置，返回位置值，找不到返回-1

* 1. repeat

功能：重复s字符串count次，最后返回重复的字符串

* 1. replace

功能：在s字符串中，把old字符串替换为new字符串，n表示替换的次数，小于0表示全部替换

* 1. split

功能：把s字符串按照sep分割，返回slice

* 1. trim

功能：在s字符串的头部和尾部去除cutset指定的字符串

* 1. fields

功能：去除s字符串的空格符，并且按照空格分割返回slice

1. 字符串转换
   1. Append
      1. Append 系列函数各种类型转换为字符串后，添加到现有的字节数组中。
   2. Format
      1. Format 系列函数把其他类型的转换为字符串。
   3. Parse
      1. Parse 系列函数把字符串转换为其他类型。
2. 正则表达式
3. json处理
   1. 编码json

func Marshal(v interface{}) ([]byte, error)

func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)

* + 1. 通过struct生成json，

struct成员变量名必须大写，如果想要小写只能通过二次编码

二次编码：

如果想输出小写的成员需要在成员尾部添加以下内容

Company string `json:"company"`(成员变量comp可以随意写)

Company string `json:"-"` // 此字段不会输出到屏幕

IsOk bool`json:"，string"` // 转化为string类型再输出到屏幕

* + 1. 通过map生成json
  1. 解码json
     1. 解码 struct json

结构体成员变量名不可以加下划线

如果想解析其中某一个字段，解码结构体可以只定义其中一个字段

* + 1. 解码 map json

解码后，如果想取出map[string]interface{}数据，不能使用强制类型转换，需要**使用类型断言反推为基础数据类型**(string, int)等再给其他变量赋值，map中的 []string类型数据，反推类型为**[]interface{},**

然后在range迭代取出每个value再反推为string类型

1. 文件操作
   1. Syscall.Umask创建文件之前修改文件权限掩码
   2. API

* [func Create(name string) (file \*File, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "Create)
* [func Open(name string) (file \*File, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "Open)
* [func OpenFile(name string, flag int, perm FileMode) (file \*File, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "OpenFile)
* [func NewFile(fd uintptr, name string) \*File](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "NewFile)
* [func Pipe() (r \*File, w \*File, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "Pipe)
* [func (f \*File) Name() string](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Name)
* [func (f \*File) Stat() (fi FileInfo, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Stat)
* [func (f \*File) Fd() uintptr](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Fd)
* [func (f \*File) Chdir() error](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Chdir)
* [func (f \*File) Chmod(mode FileMode) error](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Chmod)
* [func (f \*File) Chown(uid, gid int) error](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Chown)
* [func (f \*File) Readdir(n int) (fi []FileInfo, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Readdir)
* [func (f \*File) Readdirnames(n int) (names []string, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Readdirnames)
* [func (f \*File) Truncate(size int64) error](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Truncate)
* [func (f \*File) Read(b []byte) (n int, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Read)
* [func (f \*File) ReadAt(b []byte, off int64) (n int, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.ReadAt)
* [func (f \*File) Write(b []byte) (n int, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Write)
* [func (f \*File) WriteString(s string) (ret int, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.WriteString)
* [func (f \*File) WriteAt(b []byte, off int64) (n int, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.WriteAt)
* [func (f \*File) Seek(offset int64, whence int) (ret int64, err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Seek)
* [func (f \*File) Sync() (err error)](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Sync)
* [func (f \*File) Close() error](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File.Close)
  1. 拷贝文件案例

package main

import (

"fmt"

"os"

"errors"

"io"

)

func copy\_file(src\_file string, dest\_file string){

// 1. open file

src,src\_err := os.Open(src\_file)

if src\_err != nil {

fmt.Println(src\_err)

return

}

dest, dest\_err := os.Create(dest\_file)

if dest\_err != nil {

fmt.Println(dest\_err)

return

}

// 2. read

buf := make([]byte,1024)

for {

read\_len,read\_err := src.Read(buf)

if read\_err != nil{

if read\_len == 0 && read\_err == io.EOF{ // 文件末尾

fmt.Println("readding completed")

break

}

fmt.Println(read\_err)

return

}

// 3. write

\_,write\_err := dest.Write(buf[:read\_len])

if write\_err != nil {

fmt.Println(write\_err)

return

}

}

// 4. close

src.Close()

dest.Close()

}

func main() {

// 1. 检查命令行参数个数

args := os.Args

if args == nil || len(args) != 3{

err := errors.New("Command line parameters != 3")

fmt.Println(err)

return

}

src\_file := args[1]

dest\_file := args[2]

// 2. 源文件名与目标文件名不可以相同

if src\_file == dest\_file{

fmt.Println("The source file is the same as the target file")

}

// 3. copy file

copy\_file(src\_file,dest\_file)

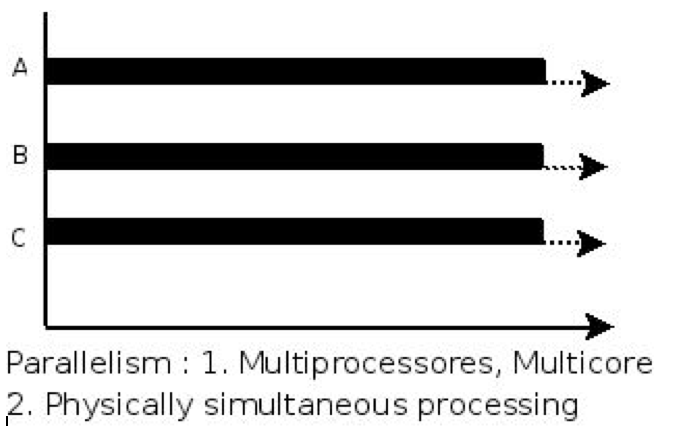
}

1. 并发编程

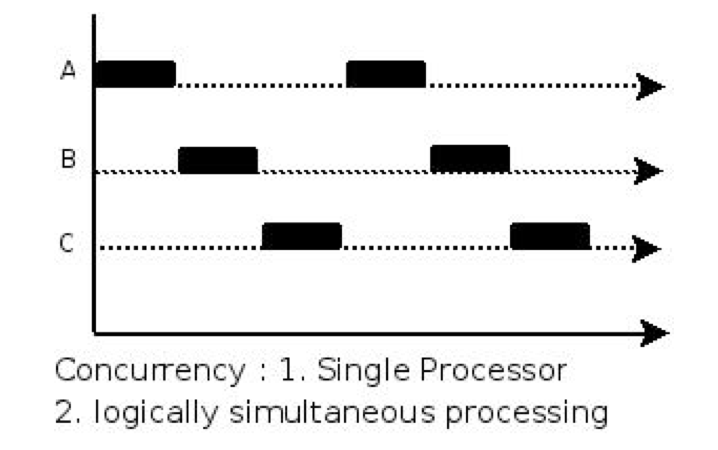
11.1 概述

11.1.1 并行和并发

**并行(parallel)：**指在同一时刻，有多条指令在多个处理器上同时执行。

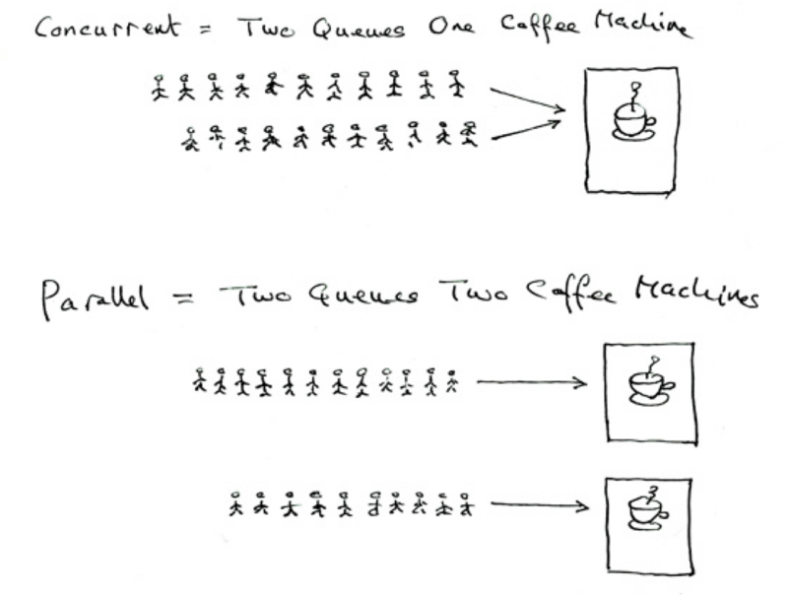


**并发(concurrency)：**指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



l 并行是两个队列**同时**使用两台咖啡机

l 并发是两个队列**交替**使用一台咖啡机



11.1.2 Go语言并发优势

有人把Go比作21世纪的C语言，第一是因为Go语言设计简单，第二，21世纪最重要的就是并行程序设计，而Go从语言层面就支持了并行。同时，并发程序的内存管理有时候是非常复杂的，而Go语言提供了自动垃圾回收机制（gc）。

Go语言为并发编程而内置的上层API基于CSP(communicating sequential processes, 顺序通信进程)模型。这就意味着显式锁都是可以避免的，因为Go语言**通过相册安全的通道发送和接受数据以实现同步**，这大大地简化了并发程序的编写。

一般情况下，一个普通的桌面计算机跑十几二十个线程就有点负载过大了，但是同样这台机器却可以轻松地让成百上千甚至过万个goroutine进行资源竞争。

11.2 goroutine

11.2.1 goroutine是什么

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

11.2.2 创建goroutine

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

在并发编程里，我们通常想讲一个过程切分成几块，然后让每个goroutine各自负责一块工作。当一个程序启动时，其主函数即在一个单独的goroutine中运行，我们叫它main goroutine。新的goroutine会用go语句来创建。

import (

"fmt"

"time"

)

func new\_task(){

i := 0

for{

i++

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

time.Sleep(1\*time.Second)

}

}

func main(){

go new\_task()

i := 0

for{

i++

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

time.Sleep(1\*time.Second)

}

// 主协程退出，子协程也会退出

//time.Sleep(3\*time.Second)

//fmt.Printf("main goroutine exit\n")

}

主goroutine退出后，其它的工作goroutine也会自动退出

11.2.4 runtime包

11.2.4.1 Gosched

runtime.Gosched() 用于**让出CPU时间片**，让出当前goroutine的执行权限，调度器安排其他等待的任务运行，**并在下次某个时候从该位置恢复执行**。

import (

"fmt"

"runtime"

)

func new\_task\_02(){

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

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

}

}

func main(){

go new\_task\_02()

for i := 0; i < 2; i++{

runtime.Gosched()

fmt.Printf("main goroutine exit\n")

}

}

11.2.4.2 Goexit

Goexit终止调用它的go程。其它go程不会受影响。Goexit会在终止该go程前执行所有defer的函数。

在程序的main go程调用本函数，会终结该go程，而不会让main返回。因为main函数没有返回，程序会继续执行其它的go程。如果所有其它go程都退出了，程序就会崩溃。

func new\_task\_03(){

defer fmt.Printf("cccccccccccc\n")

fmt.Printf("dddddddddd\n")

runtime.Goexit()

//return // 终止此函数

fmt.Printf("eeeeeeeeee\n")

}

// runtime.goexit

func test02(){

go func(){

fmt.Printf("aaaaaaaaaa\n")

new\_task\_03()

fmt.Printf("bbbbbbbbbb\n")

}()

for{}

}

11.2.4.3 GOMAXPROCS

func GOMAXPROCS(n int) int

// runtime.GOMAXPROCS

func test03(){

n := runtime.GOMAXPROCS(1)

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

// 时间片轮转

for{

go fmt.Printf("%d",1)

fmt.Printf("%d",0)

}

}

GOMAXPROCS设置可同时执行的最大CPU数，并返回先前的设置。 若 n < 1，它就不会更改当前设置。本地机器的逻辑CPU数可通过 NumCPU 查询。本函数在调度程序优化后会去掉。

11.2.5 多任务资源竞争问题

package main

import (

"fmt"

"time"

)

func pointer(str string){

for \_,data := range str{

time.Sleep(time.Second)

fmt.Printf("%c",data)

}

}

func persion1(){

pointer("hello")

}

func persion2(){

pointer("world")

}

func main(){

go persion1()

go persion2()

// 主协程不退出

for{

}

}

如何解决：添加同步机制

11.3 channel

channel 是 Go 语言在语言级别提供的 goroutine 间的通信方式。我们可以使用 channel 在两 个或多个 goroutine 之间传递消息。channel 是进程内的通信方式，因此通过 channel 传递对象的 过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。如果需要跨进程通信， 我们建议用分布式系统的方法来解决，比如使用 Socket 或者 HTTP 等通信协议。Go 语言对于网 络方面也有非常完善的支持。

channel 是类型相关的。也就是说，一个 channel 只能传递一种类型的值，这个类型需要在 声明 channel 时指定。如果对 Unix 管道有所了解的话，就不难理解 channel，可以将其认为是一 种类型安全的管道。

11.3.1 channel类型

和map类似，channel也一个对应make创建的底层数据结构的引用。

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

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

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

    make(chan Type, capacity)

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

channel通过操作符<-来接收和发送数据，发送和接收数据语法：

    channel <- value      // 写入value到channel

    <-channel             // 读取并将其丢弃

    x := <-channel        //从channel中读取数据，并赋值给x

    x, isClose := <-c

if !isClose {

fmt.Println("channel closed!")

break

}

   //功能同上，同时检查通道是否已关闭或者是否为空

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

通过channel实现同步

package main

import (

"fmt"

"time"

)

var cha = make(chan int)

func printer\_01(str string){

for \_,data := range str{

time.Sleep(time.Second)

fmt.Printf("%c",data)

}

}

func persion\_01(){

printer\_01("hello")

fmt.Printf("\n")

cha <- 666 //将内容输入到channel 解除阻塞

}

func persion\_02(){

<- cha //没有数据输出时会阻塞

printer\_01("world")

}

func test\_01(){

go persion\_01()

go persion\_02()

for{} //不让主协程退出

}

func main(){

test\_01()

}

channel同步和数据交互

func test\_02(){

cha2 := make(chan string)

go func(){

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

fmt.Printf("子协程循环 %d\n",i)

}

cha2 <- "我是子协程，工作结束"

}()

str := <- cha2 //channel中没有数据时阻塞等待

fmt.Println(str)

}

func main(){

test\_02()

}

11.3.2 无缓冲的channel

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

这种类型的通道要求发送goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果任意一个goroutine没有准备好，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。

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

**无缓冲的channel创建：**

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

// 1. 无缓冲

func unbuffered\_channel(){

cha\_03 := make(chan int)

go func(){

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

cha\_03 <- i // 当另一端没有人读时，阻塞等待

fmt.Println("子协程： i = ",i)

}

}()

time.Sleep(3 \* time.Second)

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

num := <- cha\_03 // 当channel中没有数据时 阻塞等待

fmt.Println(num)

}

}

11.3.3 有缓冲的channel

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

这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

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

有缓冲的channel创建格式：

    make(chan Type, capacity)

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

// 2. 有缓冲\_

func buffered\_channel(){

cha\_04 := make(chan int, 10) //set channel capacity

go func(){

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

time.Sleep(1 \* time.Second)

fmt.Println("i = ",i)

cha\_04 <- i // 当另一端没有读取时，容量未满时写入channel不阻塞，异步

}

}()

time.Sleep(4 \* time.Second)

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

num := <- cha\_04 // 当channel中没有数据时，阻塞等待

fmt.Println(num)

}

}

11.3.4 range和close

1. close

// 3. close channel

func close\_channel(){

cha\_05 := make(chan int,10)

go func(){

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

time.Sleep(time.Second)

cha\_05 <- i

}

close(cha\_05) // 关闭channel

}()

for{

if num, ok := <- cha\_05;ok == false{ // 判断channel是否关闭

break

}else{

fmt.Println("num = ",num)

}

}

}

1. range

// 4. range channel关闭

func range\_channel(){

cha\_06 := make(chan int,10)

go func(){

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

time.Sleep(time.Second)

cha\_06 <- i

}

close(cha\_06)

}()

for num := range cha\_06{

fmt.Println("num = ",num)

}

}

**注意：**

1. **channel不像文件需要经常关闭，只有确定没有数据写入或结束range循环之类的才会显示调用close() 关闭channel**
2. **关闭channel后无法向channel写入数据，(引发panic错误后导致接收立即返回0值)**
3. **关闭channel后可以从channel中读取数据**
4. **对于nil channel 无论收发数据都会阻塞**

11.3.5 单方向的channel

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

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

单向channel变量的声明非常简单，如下：

var ch1 chan int       // ch1是一个正常的channel，不是单向的

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

var ch3 <-chan int     // ch3是单向channel，只用于读取int数据

l chan<- 表示数据进入管道，要把数据写进管道，对于调用者就是输出。

l <-chan 表示数据从管道出来，对于调用者就是得到管道的数据，当然就是输入。

// 5. 单项channel

func single\_channel(){

cha\_07 := make(chan int)

var read\_cha <- chan int = cha\_07

var write\_cha chan <- int = cha\_07

r := <- read\_cha

fmt.Println(r)

write\_cha <- 666

read\_cha <- 777 // err 只读channel 不可写入数据

w := <- write\_cha // err 只写channel 不可读数据

fmt.Println(w)

}

**output**

./23\_channel\_02.go:98:11: invalid operation: read\_cha <- 777 (send to receive-only type <-chan int)

./23\_channel\_02.go:99:7: invalid operation: <-write\_cha (receive from send-only type chan<- int)

注意：

双向channel可转换为单项 单项channel不可转换为双向

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

var ch3 <-chan int // ch3是单向channel，只用于读取int数据

// 6. 单项channel做函数参数

// producer 参数只能写入 不能读取

func producer(out chan <- int){

for i := 0; i < 8; i++{

time.Sleep(time.Second)

out <- i

}

close(out) // 关闭 channel

}

// consumer 参数只能读取 不能写入

func consumer(in <- chan int){

// channel range 的使用 一端关闭 结束

for num := range in{

fmt.Printf("%d ",num)

}

}

func single\_function\_channel(){

cha\_08 := make(chan int)

// 新开一个协程生产 生产者写入数据

go producer(cha\_08)

// 消费者读取数据 输出

consumer(cha\_08)

}

11.3.6 定时器

11.3.6.1 Timer

Timer是一个定时器，**代表未来的一个单一事件**，你可以告诉timer你要等待多长时间，它提供一个channel，在将来的那个时间那个channel提供了一个时间值。

// 时间到 只会响应一次 往time.c channel中写一次

timer := time.NewTimer(1 \* time.Second)

func timer\_test\_01(){

timer := time.NewTimer(1 \* time.Second)

fmt.Println("当前时间：",time.Now())

t := <- timer.C

fmt.Println("t = ",t)

}

// 验证timer.NewTimer 只写一次

func timer\_test\_02(){

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

for{

t := <- timer.C

fmt.Println("时间到：", t)

}

}

output:

fatal error: all goroutines are asleep - deadlock!

通过timer实现延时功能

// timer 延时

func timer\_test\_03(){

// 1. 使用time.NewTimer

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

<- timer.C

fmt.Println("延时 2秒 输出")

// 2. sleep

time.Sleep(3 \* time.Second)

fmt.Println("延时 3秒 输出")

// 3. time.After() 返回channel

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

fmt.Println("延时 5秒 输出")

}

停止和重设定时器

// timer stop

func timer\_stop(){

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

go func (){

// 时间未到时 阻塞等待

<- timer.C

fmt.Println("定时器时间到，子协程输出")

}()

timer.Stop() // 停止定时器

fmt.Println("定时器已停止")

for{} // 死循环

}

func timer\_reset(){

timer := time.NewTimer(10 \* time.Second)

// 新开协程 重置定时器

go func(){

<- timer.C

fmt.Println("time out")

}()

timer.Reset(2 \* time.Second)

fmt.Println("timer reset 2 \* Second")

for{}

}

11.3.6.2 Ticker

Ticker是一个定时触发的计时器，它会以一个间隔(interval)往channel发送一个事件(当前时间)，而channel的接收者可以以固定的时间间隔从channel中读取事件。

func ticker\_test(){

timer := time.NewTicker(1 \* time.Second)

i := 0

for{

<- timer.C

i++

fmt.Println("i = ",i)

}

}

11.4 select

11.4.1 select作用

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

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

与switch语句可以选择任何可使用相等比较的条件相比， select有**比较多的限制**，其中最大的一条限制就是**每个case语句里必须是一个IO操作**，大致的结构如下：

   select {

    case <-chan1:

      // 如果chan1成功读到数据，则进行该case处理语句

case chan2 <- 1:

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

    default:

        // 如果上面都没有成功，则进入default处理流程

    }

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

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

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

l 如果给出了default语句，那么就会执行default语句，同时程序的执行会从select语句后的语句中恢复。

l 如果没有default语句，那么select语句将被阻塞，直到至少有一个通信可以进行下去。

**select实现fibonacci数列**

import(

"fmt"

"runtime"

)

func fibonacci(ch chan <- int, quit <- chan bool){

x, y := 1, 1

for{

select {

case ch <- x:

x, y = y, x+y

case flag := <- quit:

fmt.Println("flag = ",flag)

//return

runtime.Goexit()

}

}

}

func select\_test\_01(){

ch := make(chan int) // 通信数字

quit := make(chan bool) // 是否退出

// 开启新协程来运行

go fibonacci(ch,quit)

func(){

for i := 0;i < 8; i++{

num := <- ch

fmt.Println(num)

}

quit <- true

}()

}

11.4.2 select超时机制

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

**select实现超时机制**

// select 实现超时断开

func select\_test\_02(){

ch := make(chan int)

quit := make(chan bool)

go func(){

for{

select{

case num := <- ch:

fmt.Println("num = ",num)

case <-time.After(3 \* time.Second):

fmt.Println("time out")

quit <- true

}

}

}()

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

time.Sleep(time.Second)

ch <- i

}

<- quit // 阻塞

fmt.Println("")

}

11.5 总结

**1. 概述**

**并行(parallel)：**指在同一时刻，有多条指令在多个处理器上同时执行。

**并发(concurrency)：**指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。

**go语言的并发优势**

* 1. Go从语言层面就支持了并行。同时，并发程序的内存管理有时候是非常复杂的，而Go语言提供了自动垃圾回收机制。
  2. Go语言为并发编程而内置的上层API基于CSP模型

1. **goroutine**
   1. goroutine 是 Go 语言中的轻量级线程实现
   2. 函数调用前加上 go 关键字，这次调用就会在一个新的 goroutine 中并发执行。当被调用的函数返回时，这个goroutine也自动结束了，需要注意，如果这个函数有返回值，那么这个返回值会被丢弃
2. **runtime包**
   1. Gosched
      1. runtime.Gosched() 用于让出CPU时间片，让出当前goroutine的执行权限，调度器安排其他等待的任务运行，并在下次某个时候从该位置恢复执行。
   2. Goexit
      1. runtime.Goexit() 将立即终止当前 goroutine 执⾏，调度器确保所有已注册 defer延迟调用被执行。
   3. GOMAXPROCS
      1. runtime.GOMAXPROCS() 用来设置可以并行计算的CPU核数的最大值，并返回之前的值。
3. **channel**

**两个作用 1.用于同步 2. 用于数据交互**

* 1. 无缓冲的通道（unbuffered channel）

    // 任意一端在同一时刻未传递或未接收 都会导致阻塞

c := make(chan type, 0) //无缓冲的通道

* 1. 有缓冲的通道（buffered channel）

// 容量满时才会阻塞

c := make(chan type, capacity) //无缓冲的通道

* 1. range和close

注意点：

l channel不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的，才去关闭channel；

l 关闭channel后，无法向channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；

l 关闭channel后，可以继续向channel接收数据；

l 对于nil channel，无论收发都会被阻塞。

// range

for num := range cha\_06{ // range监测到关闭后会直接跳出循环

fmt.Println("num = ",num)

}

// close

close(ch) //关闭 channel

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

       fmt.Println(data)

   } else {

       break

   }

* 1. 单方向channel

var ch1 chan int       // ch1是一个正常的channel，不是单向的

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

var ch3 <-chan int     // ch3是单向channel，只用于读取int数据

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

* 1. 定时器
     1. Timer
        1. **代表未来的一个单一事件**
     2. Ticker
        1. 定时触发的计时器，它会以一个间隔(interval)往channel发送一个事件(**循环**)
  2. select
     1. 通过select可以监听channel上的数据流动
     2. 如果给出了default语句，那么就会执行default语句，同时程序的执行会从select语句后的语句中恢复。
     3. 如果没有default语句，那么select语句将被阻塞，直到至少有一个通信可以进行下去。如果其中的任意一语句可以继续执行(即没有被阻塞)，那么就从那些可以执行的语句中任意选择一条来使用。
     4. select超时
        1. 有时候会出现goroutine阻塞的情况，那么我们如何避免整个程序进入阻塞的情况呢？我们可以利用select来设置超时

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

1. 网络编程

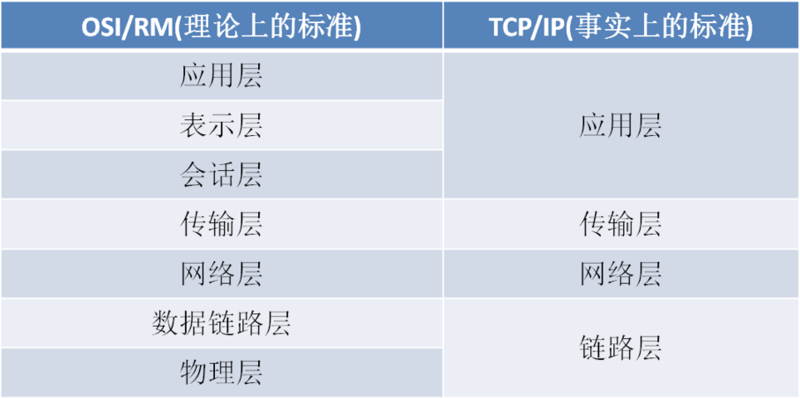
12.1 网络概述

12.1.1 网络协议

12.1.2 分层模型

12.1.2.1 网络分层架构

为了减少协议设计的复杂性，大多数网络模型均采用分层的方式来组织。每一层都有自己的功能，就像建筑物一样，每一层都靠下一层支持。每一层利用下一层提供的服务来为上一层提供服务，本层服务的实现细节对上层屏蔽。



1)      **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

2)      **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的115200、8、N、1

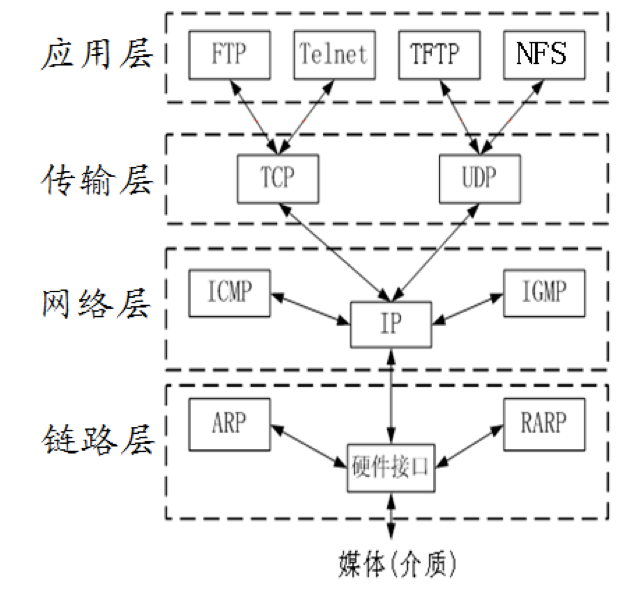
3)      **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

4)      **传输层**：定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

5)      **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。

6)      **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。

7)      **应用层**：是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。



12.1.2.3 每层协议的功能



1) 链路层

以太网规定，连入网络的所有设备，都必须具有“网卡”接口。数据包必须是从一块网卡，传送到另一块网卡。通过网卡能够使不同的计算机之间连接，从而完成数据通信等功能。网卡的地址——MAC 地址，就是数据包的物理发送地址和物理接收地址。

2) 网络层

网络层的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做“网络地址”，这是我们平时所说的IP地址。这个IP地址好比我们的手机号码，通过手机号码可以得到用户所在的归属地。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。网络层协议包含的主要信息是源IP和目的IP。

于是，“网络层”出现以后，每台计算机有了两种地址，一种是 MAC 地址，另一种是网络地址。**两种地址之间没有任何联系**，MAC 地址是绑定在网卡上的，网络地址则是管理员分配的，它们只是随机组合在一起。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理 MAC 地址。

3) 传输层

当我们一边聊QQ，一边聊微信，当一个数据包从互联网上发来的时候，我们怎么知道，它是来自QQ的内容，还是来自微信的内容？

也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做“端口”（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

端口特点：

l 对于同一个端口，在不同系统中对应着不同的进程

l 对于同一个系统，一个端口只能被一个进程拥有

4) 应用层

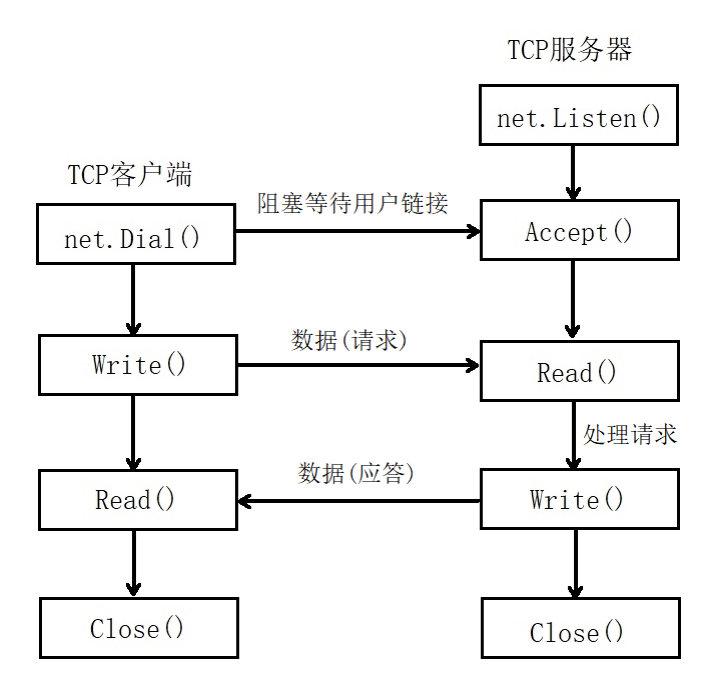
应用程序收到“传输层”的数据，接下来就要进行解读。由于互联网是开放架构，数据来源五花八门，必须事先规定好格式，否则根本无法解读。“应用层”的作用，就是规定应用程序的数据格式。

12.2 socket编程

Socket起源于Unix，而Unix基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。Socket就是该模式的一个实现，网络的Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的Socket描述符，随后的连接建立、数据传输等操作都是通过该Socket实现的。

常用的Socket类型有两种：流式Socket（SOCK\_STREAM）和数据报式Socket（SOCK\_DGRAM）。流式是一种面向连接的Socket，针对于面向连接的TCP服务应用；数据报式Socket是一种无连接的Socket，对应于无连接的UDP服务应用。

12.2.1 TCP c/s架构



**server代码**

func tcp\_server\_test01(){

listener,listen\_err := net.Listen("tcp","127.0.0.1:8000")

if listen\_err != nil{

fmt.Println("listen err: ",listen\_err)

return

}

// buf

buf := make([]byte,1024)

conn, accept\_err := listener.Accept()

if accept\_err != nil{

fmt.Println("accept err: ",listen\_err)

return

}

for{

n, read\_err := conn.Read(buf)

if read\_err != nil {

if read\_err == io.EOF{

fmt.Println("eof...")

break

}

fmt.Println("read err: ", read\_err)

break

}

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

\_,write\_err := conn.Write(buf)

if write\_err != nil{

fmt.Println("write err: ", write\_err)

break

}

}

defer conn.Close()

defer listener.Close()

}

**client代码**

func tcp\_client\_test01(){

conn, dial\_err := net.Dial("tcp","127.0.0.1:8000")

if dial\_err != nil {

fmt.Println("dial err: ", dial\_err)

return

}

buf\_w := make([]byte,1024)

buf\_r := make([]byte,1024)

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

time.Sleep(time.Second)

buf\_w = []byte (fmt.Sprintf("%s %d","hello",i))

\_,write\_err := conn.Write(buf\_w)

if write\_err != nil{

fmt.Println("write err: ",write\_err)

return

}

n, read\_err := conn.Read(buf\_r)

if read\_err != nil{

fmt.Println("read err: ",read\_err)

return

}

fmt.Println(string(buf\_r[:n]))

}

defer conn.Close()

}

12.2.2 简单的并发服务器

**server端**

import (

"fmt"

"net"

"strings"

"io"

"runtime"

)

func handle\_conncet(conn net.Conn){

fmt.Println(conn.RemoteAddr()," is conncet")

buf := make([]byte,1024)

for{

buf = 0

rn,read\_err := conn.Read(buf)

if read\_err != nil {

if read\_err == io.EOF{

fmt.Println(conn.RemoteAddr()," client is closed")

runtime.Goexit()

}

fmt.Println("read\_err: ",read\_err)

return

}

fmt.Println("client: ",string(buf[:rn]))

\_,write\_err := conn.Write([]byte(strings.ToUpper(string(buf[:rn]))))

if write\_err != nil {

fmt.Println("write\_err: ",write\_err)

return

}

}

}

func concurrency\_server() {

// 1. 监听

listenner, listen\_err := net.Listen("tcp", "127.0.0.1:8000")

if listen\_err != nil {

fmt.Println("listen\_err: ", listen\_err)

return

}

defer listenner.Close()

for {

// 2. accept

conn, conn\_err := listenner.Accept()

if conn\_err != nil {

fmt.Println("conn\_err: ", conn\_err)

return

}

go handle\_conncet(conn)

}

}

func main(){

concurrency\_server()

}

client端

import (

"fmt"

"net"

"os"

)

func client\_01(){

conn,conn\_err := net.Dial("tcp","127.0.0.1:8000")

if conn\_err != nil {

fmt.Println("conn\_err: ",conn\_err)

return

}

defer conn.Close()

buf := make([]byte,1024)

for{

rn,stdin\_read\_err := os.Stdin.Read(buf)

if stdin\_read\_err != nil {

fmt.Println("client read\_err: ",stdin\_read\_err)

return

}

if string(buf[:rn - 1]) == "exit" {

fmt.Println("bye...")

return

}

\_,write\_err := conn.Write(buf[:rn])

if write\_err != nil {

fmt.Println("client write\_err: ",write\_err)

return

}

\_,conn\_read\_err := conn.Read(buf)

if conn\_read\_err != nil {

fmt.Println("client conn\_read\_err: ",conn\_read\_err)

return

}

fmt.Println("server: ",string(buf))

}

}

func main(){

client\_01()

}

12.2.4 os.stat

func Stat(name string) (fi FileInfo, err error)

Stat返回一个描述name指定的文件对象的FileInfo。如果指定的文件对象是一个符号链接，返回的FileInfo描述该符号链接指向的文件的信息，本函数会尝试跳转该链接。如果出错，返回的错误值为\*PathError类型。

type [FileInfo](https://github.com/golang/go/blob/master/src/os/types.go?name=release" \l "16)

type FileInfo interface {

Name() string // 文件的名字（不含扩展名）

Size() int64 // 普通文件返回值表示其大小；其他文件的返回值含义各系统不同

Mode() FileMode // 文件的模式位

ModTime() time.Time // 文件的修改时间

IsDir() bool // 等价于Mode().IsDir()

Sys() interface{} // 底层数据来源（可以返回nil）

}

FileInfo用来描述一个文件对象。

12.2.5 file\_transfer

12.3 HTTP编程

12.3.1 概述

**12.3.1.1 web工作方式**

Web服务器的工作原理可以简单地归纳为：

l 客户机通过TCP/IP协议建立到服务器的TCP连接

l 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档

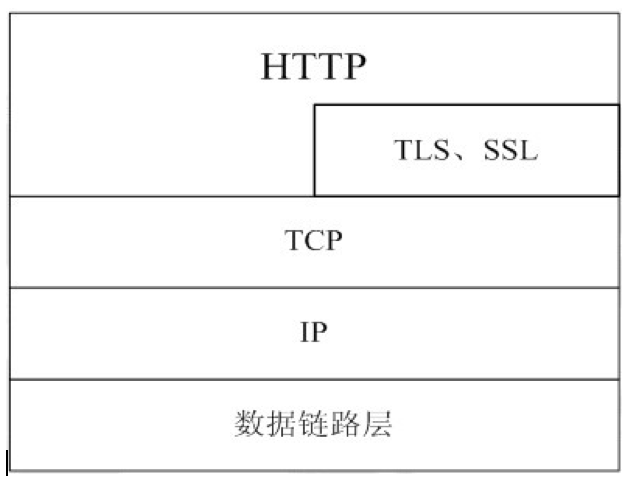
l 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端

l 客户机与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

**12.3.1.2 HTTP协议**

超文本传输协议(HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议，它详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

HTTP协议通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了我们常说的HTTPS。如下图所示：



**12.3.1.3 地址（URL）**

URL全称为Unique Resource Location，用来表示网络资源，可以理解为网络文件路径。

URL的格式如下：

http://host[":"port][abs\_path]

http://192.168.31.1/html/index

URL的长度有限制，不同的服务器的限制值不太相同，但是不能无限长。

12.3.2 HTTP报文浅析

**12.3.2.1 请求报文格式**

1. **HTTP server测试代码**

package main

import (

"net"

"fmt"

"log"

)

func server(){

listenner, listen\_err := net.Listen("tcp","127.0.0.1:8888")

if listen\_err != nil {

log.Fatal(listen\_err)

return

}

buf := make([]byte,1024)

for{

conn,conn\_err := listenner.Accept()

if conn\_err != nil {

log.Println(conn\_err)

return

}

ip\_addr := conn.RemoteAddr().String()

n,read\_err := conn.Read(buf)

if read\_err != nil {

log.Println(read\_err)

return

}

result := buf[:n]

fmt.Printf("接收到[%s] ------> \n%s\n",ip\_addr,string(result))

}

}

func main(){

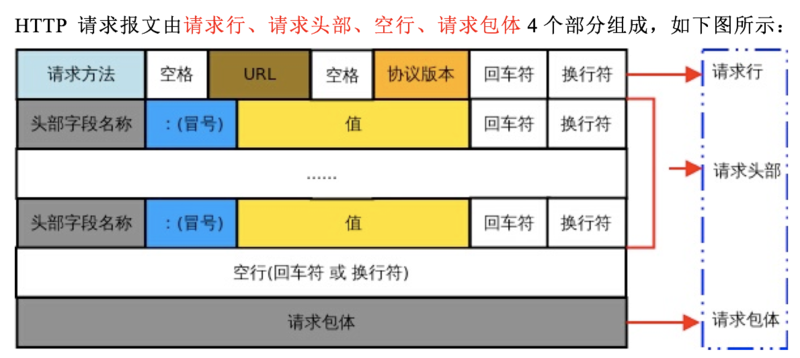
server()

}

浏览器输入 127.0.0.1:8888

server端显示：

1. **请求报文格式：**



1)      请求行

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

GET：

l 当客户端要从服务器中读取某个资源时，使用GET 方法。GET 方法要求服务器将URL 定位的资源放在响应报文的数据部分，回送给客户端，即向服务器请求某个资源。

l 使用GET方法时，请求参数和对应的值附加在 URL 后面，利用一个问号(“?”)代表URL 的结尾与请求参数的开始，传递参数长度受限制，因此GET方法不适合用于上传数据。

l 通过GET方法来获取网页时，参数会显示在浏览器地址栏上，因此保密性很差。

POST：

l 当客户端给服务器提供信息较多时可以使用POST 方法，POST 方法向服务器提交数据，比如完成表单数据的提交，将数据提交给服务器处理。

l GET 一般用于获取/查询资源信息，POST 会附带用户数据，一般用于更新资源信息。POST 方法将请求参数封装在HTTP 请求数据中，而且长度没有限制，因为POST携带的数据，在HTTP的请求正文中，以名称/值的形式出现，可以传输大量数据。

2)      请求头部

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。

请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

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

1)      空行

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

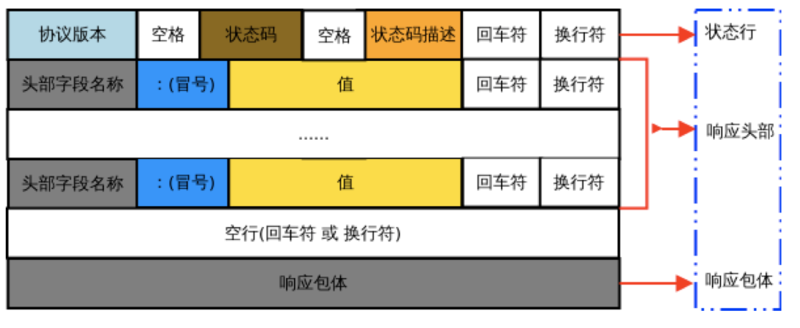
2)      请求包体

请求包体不在GET方法中使用，而是POST方法中使用。

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

2) 响应报文格式说明

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



1)      状态行

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

状态码：

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

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

常见的状态码举例：

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

2)      响应头部

响应头可能包括：

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

3)      空行

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

4)      响应包体

服务器返回给客户端的文本信息。

12.3.3 HTTP编程

Go语言标准库内建提供了net/http包，涵盖了HTTP客户端和服务端的具体实现。使用

net/http包，我们可以很方便地编写HTTP客户端或服务端的程序。