# Go基础巩固加强

## 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。

### Go语言中的指针

Go语言保留了指针，但与C语言指针有所不同。主要体现在：

l 默认值 nil

l 操作符 "&" 取变量地址， "*" 通过指针访问目标对象

l 不支持指针运算，不支持 "->" 运算符，直接用 "." 访问目标成员

```
func main()  {
   var a int = 100                // 声明 int 变量 a
   fmt.Printf("&a = %p\n", &a)    // "&" 取 a 地址

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

### 函数new

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

new创建的内存空间位于heap上，空间的默认值为数据类型默认值。如：new(int) 则 *p为0，new(bool) 则 *p为false

```
func main()  {
   var p1 *int
   p1 = new(int)                     //p1为*int 类型，指向匿名的int变量
   fmt.Println("*p1 = ", *p1)       //*p1 =  0

   p2 := new(bool)              //p2为*bool 类型，指向匿名的bool变量
   fmt.Println("*p2 = ", *p2)       //*p2 = false
   *p2 = true
   fmt.Println("*p2 = ", *p2)       //*p1 = true
}
```

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

## 指针做函数参数

```
func swap01(a, b int) {
    a, b = b, a
    fmt.Printf("swap01 a = %d, b = %d\n", a, b)
}

func swap02(x, y *int) {
    *x, *y = *y, *x
}

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

    //swap01(a, b)  //值传递（传值）
    swap02(&a, &b)  //地址传递（传引用）
    fmt.Printf("a = %d, b = %d\n", a, b)
}
```

# slice

## 切片简述

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

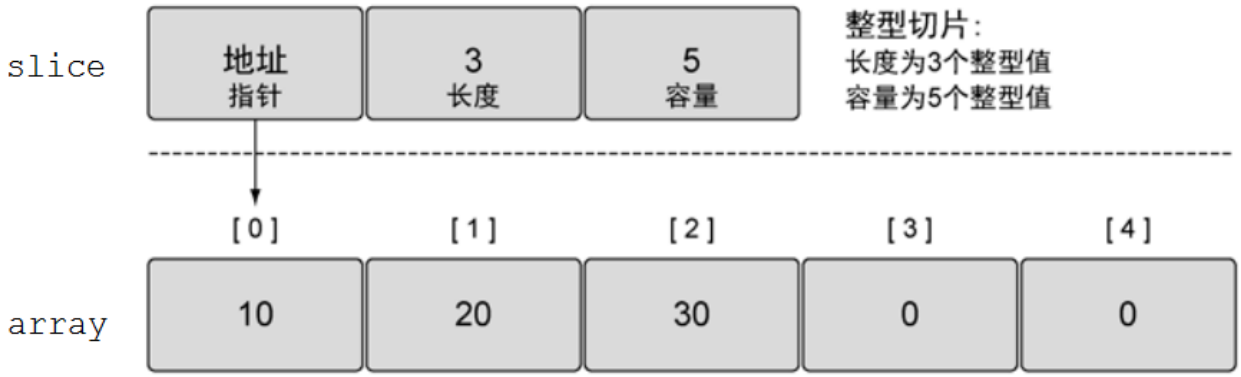
Slice（切片）代表变长的序列，序列中每个元素都有相同的类型。一个slice类型一般写作[]T，其中T代表slice中元素的类型；slice的语法和数组很像，只是没有固定长度而已。

数组和slice之间有着紧密的联系。一个slice是一个轻量级的数据结构，提供了访问数组子序列（或者全部）元素的功能，而且slice的底层确实引用一个数组对象。一个slice由三个部分构成：指针、长度和容量。指针指向第一个slice元素对应的底层数组元素的地址，要注意的是slice的第一个元素并不一定就是数组的第一个元素。

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

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

```
array := [...]int{10, 20, 30, 0, 0}
slice := array[0:3:5]
```




## 创建切片

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

1.自动推导类型创建slice

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

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

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

3.**make时，没有指定容量，那么 长度==容量**

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

```
func main()  {
   s1 := [] int {1, 2, 3, 4}     // 创建 有4个元素的切片
   fmt.Println("s1=", s1)

   s2 := make([]int, 5, 10)      // 借助make创建 slice，格式：make(切片类型，长度，容量)
   s2[4] = 7
   //s2[5] = 9                   // 报错：panic: runtime error: index out of range
   fmt.Println("s2=", s2)
   fmt.Printf("len(s2)=%d, cap(s2)=%d\n", len(s2), cap(s2))

   s3 := make([]int, 5)         // make时，没指定容量，那么 长度 == 容量
   s3[2] = 3
   fmt.Println("s3=", s3)
   fmt.Printf("len(s2)=%d, cap(s2)=%d\n", len(s3), cap(s3))
}
```

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

# 切片操作

## 切片截取

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

截取可表示为s[low：high：max]。low：表示下标的起点。 high：表示下标的终点（左闭右开，不包括此下标）。 长度 len = high – low。容量 cap = max – low。长度对应slice中元素的数目；长度不能超过容量，容量一般是从slice的开始位置到底层数据的结尾位置。内置的len()和cap() 函数分别返回slice的长度和容量。

示例说明：

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

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

## 切片和底层数组关系

```
func main()  {
   arr := [] int {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
   s1 := arr[2:5]               // 从arr[2]开始，取 5-2 个元素，组成切片s1。
   fmt.Println("s1=", s1)       // s1= [2 3 4]

   s1[1] = 666                      // 这样将arr数组中 3 --> 666。
   fmt.Println("arr=", arr)     // arr= [0 1 2 666 4 5 6 7 8 9]

   s2 := s1[2:7]             // 从s1[2]开始， 取 7-2 个元素，组成 s2。
```

```
    fmt.Println("s2=", s2)       // 实际上还是取的 数组arr。    s2= [4 5 6 7 8]

    s2[2] = 777                      // 这会将arr中的 6 --> 777
    fmt.Println("arr=", arr)     // arr= [0 1 2 666 4 5 777 7 8 9]
}
```

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

## 切片做函数参数

切片作为函数参数时，**传引用。**

```
func testFunc(s []int) {    // 切片做函数参数
    s[0] = -1            // 直接修改 main中的 slice
}

func main() {
    slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    fmt.Println(slice)

    testFunc(slice)          // 传引用
    fmt.Println(slice)
}
```

## 常用操作函数

append函数

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

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

    s1 = append(s1, 1)           //追加1个元素
    s1 = append(s1, 2, 3)        //追加2个元素
    s1 = append(s1, 4, 5, 6)     //追加3个元素
    fmt.Println(s1)              //[1 2 3 4 5 6]

    s2 := make([]int, 5)
    s2 = append(s2, 6)
    fmt.Println(s2)          //[0 0 0 0 0 6]

    s3 := []int{1, 2, 3}
    s3 = append(s3, 4, 5)
    fmt.Println(s3)      //[1 2 3 4 5]
```

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

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

输出结果如下：

```
cap: 1 -> 2
cap: 2 -> 4
cap: 4 -> 8
cap: 8 -> 16
cap: 16 -> 32
cap: 32 -> 64
cap: 64 -> 128
```

### copy函数

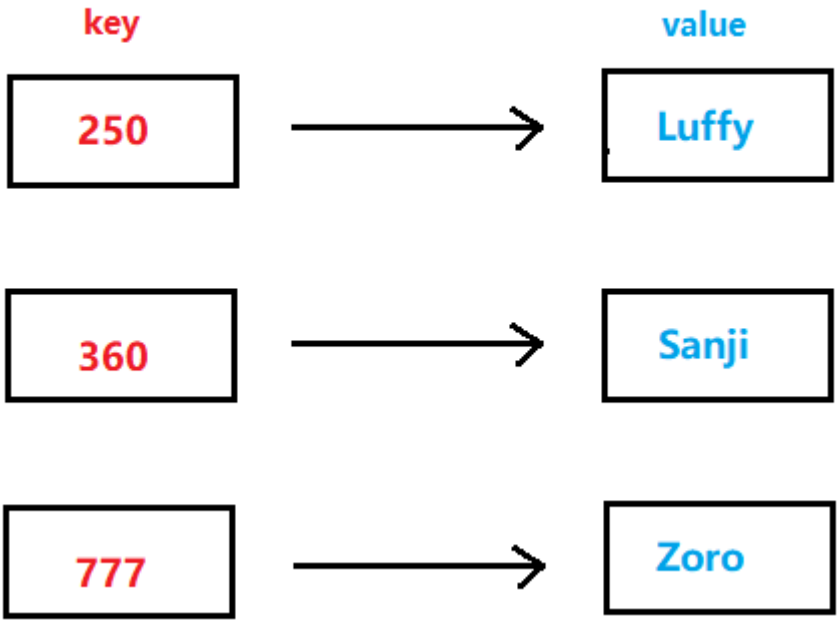
函数 copy 在两个 slice 间复制数据，复制长度以 len 小的为准，两个 slice 指向同一底层数组。直接对应位置覆盖。

# map

## map简述

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

```
pirate := map[int]string {250:"Luffy", 360:"Sanji", 777:"Zoro"}
```




map格式为：

```
map[keyType]valueType
```

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

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

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

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

# 创建、初始化map

## 创建map

```
    var m1 map[int]string   //只是声明一个map，没有初始化，为空(nil)map
    fmt.Println(m1 == nil)  //true
    //m1[1] = "Luffy"       //nil的map不能使用err, panic: assignment to entry in nil map

    m2 := map[int]string{}  //m2, m3的创建方法是等价的
    m3 := make(map[int]string)
    fmt.Println(m2, m3)          //map[] map[]

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

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

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

### 初始化map

也可以直接指定初值，要保证key不重复。

```
//1、定义同时初始化
var m1 map[int]string = map[int]string{1: "Luffy", 2: "Sanji"}
fmt.Println(m1) //map[1:Luffy 2:Sanji]

//2、自动推导类型 :=
m2 := map[int]string{1: "Luffy", 2: "Sanji"}
fmt.Println(m2)
```

## 常用操作

### 赋值

```
m1 := map[int]string{1: "Luffy", 2: "Sanji"}
m1[1] = "Nami"   //修改
m1[3] = "Zoro"  //追加， go底层会自动为map分配空间
fmt.Println(m1) //map[1:Nami 2:Sanji 3:Zoro]

m2 := make(map[int]string, 10)  //创建map
m2[0] = "aaa"
m2[1] = "bbb"
fmt.Println(m2)                 //map[0:aaa 1:bbb]
fmt.Println(m2[0], m2[1])       //aaa bbb
```

### 遍历

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

```
m1 := map[int]string{1: "Luffy", 2: "Sanji"}
//遍历1，第一个返回值是key，第二个返回值是value
for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
    //1 ----> Luffy
    //2 ----> yoyo
}

//遍历2，第一个返回值是key，第二个返回值是value（可省略）
for k := range m1 {
    fmt.Printf("%d ----> %s\n", k, m1[k])
```

```
    //1 ----> Luffy
    //2 ----> Sanji
}
```

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

如果key存在，第一个返回值返回value的值。第二个返回值为true。

```
value, ok := m1[1]
fmt.Println("value = ", value, ", ok = ", ok) //value =  mike , ok =  true
```

### 删除

使用delete()函数，指定key值可以方便的删除map中的k-v映射。

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

for k, v := range m1 {  //遍历，第一个返回值是key，第二个返回值是value
    fmt.Printf("%d ----> %s\n", k, v)
}
//1 ----> Sanji
//2 ----> Sanji
//3 ----> Zoro
delete(m1, 2)       //删除key值为2的map

for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
}
//1 ----> Luffy
//3 ----> Zoro
```

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

如：

```
delete(m1, 5)       //删除key值为5的map

for k, v := range m1 {
    fmt.Printf("%d ----> %s\n", k, v)
}
//1 ----> Luffy
//3 ----> Zoro
```

map输出结果依然是原来的样子，且不会有任何错误提示。

## map做函数参数

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

```
func DeleteMap(m map[int]string, key int) {
    delete(m, key) //删除key值为2的map
    for k, v := range m {
        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)
}
    //len(m)=2, 1 ----> Luffy
    //len(m)=2, 3 ----> Zoro
}

func main() {
    m := map[int]string{1: "Luffy", 2: "Sanji", 3: "Zoro"}
    DeleteMap(m, 2)     //删除key值为2的map

    for k, v := range m {
        fmt.Printf("len(m)=%d, %d ----> %s\n", len(m), k, v)
}
    //len(m)=2, 1 ----> Luffy
    //len(m)=2, 3 ----> Zoro
}
```

## map做函数返回值

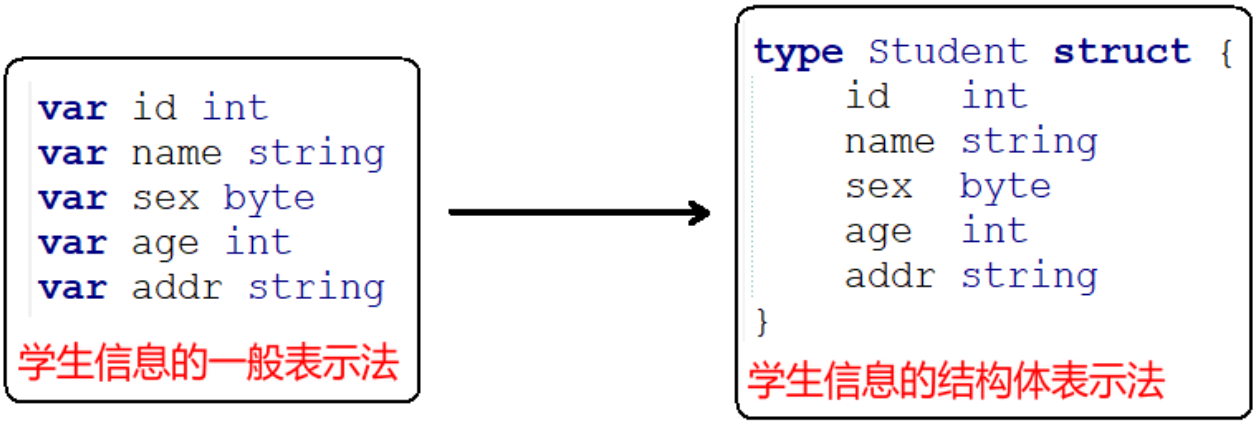
返回的依然是**引用**：

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

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


输出:
m1 =  map[1:Luffy 2:Nami 67:Zoro]
m2 =  map[2:Nami 67:Zoro 1:Luffy]
```

# 结构体

## 结构体类型

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




结构体是一种聚合的数据类型，它是由一系列具有相同类型或不同类型的数据构成的数据集合。每个数据称为结构体的成员。

## 结构体初始化

### 普通变量

```
type Student struct {
    id   int
    name string
    sex  byte
    age  int
    addr string
}

func main() {
    //1、顺序初始化，必须每个成员都初始化
    var s1 Student = Student{1, "Luffy", 'm', 18, "EastSea"}
    s2 := Student{2, "Sanji", 'f', 20, "EastSea"}
    //s3 := Student{2, "Nami", 'm', 20} //err, too few values in struct initializer

    //2、指定初始化某个成员，没有初始化的成员为零值
    s4 := Student{id: 2, name: "Zoro"}
}
```

### 指针变量

```
type Student struct {
    id   int
    name string
    sex  byte
    age  int
    addr string
}

func main() {
    var s5 *Student = &Student{3, "Nami", 'm', 16, "EastSea"}
    s6 := &Student{4, "ro", 'm', 3, "NorthSea"}
}
```

# 使用结构体成员

## 普通变量

```
    //===============结构体变量为普通变量
    //1、打印成员
    var s1 Student = Student{1, "Luffy", 'm', 18, "EastSea"}
    //结果：id = 1, name = Luffy, sex = m, age = 18, addr = EastSea
    fmt.Printf("id = %d, name = %s, sex = %c, age = %d, addr = %s\n", s1.id, s1.name, s1.sex,
s1.age, s1.addr)

    //2、成员变量赋值
    var s2 Student
    s2.id = 2
    s2.name = "Sanji"
    s2.sex = 'f'
    s2.age = 16
    s2.addr = "EastSea"
    fmt.Println(s2) //{2 yoyo 102 16 EastSea}
```

## 指针变量

```
    //===============结构体变量为指针变量
    //3、先分配空间，再赋值
    s3 := new(Student)
    s3.id = 3
    s3.name = "Nami"
    fmt.Println(s3) //&{3 Nami 0 0 }

    //4、普通变量和指针变量类型打印
    var s4 Student = Student{4, "Sanji", 'm', 18, "EastSea"}
    fmt.Printf("s4 = %v, &s4 = %v\n", s4, &s4) //s4 = {4 Sanji 109 18 sz}, &s4 = &{4 Sanji 109
18 EastSea}

    var p *Student = &s4

    //p.成员 和(*p).成员 操作是等价的
```

```
    p.id = 5
    (*p).name = "ro"
    fmt.Println(p, *p, s4) //&{5 ro 109 18 EastSea} {5 ro 109 18 EastSea} {5 ro 109 18 EastSea}
```

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

# 结构体比较

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

```
func main() {
    s1 := Student{1, "Luffy", 'm', 18, "EastSea"}
    s2 := Student{1, "Luffy", 'm', 18, "EastSea"}

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

# 作函数参数

## 传值

```
func printValue(stu Student) {
    stu.id = 250
    //printValue stu =  {250 Luffy 109 18 s EastSea}
    fmt.Println("printValue stu = ", stu)
}

func main() {
     var s Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    printValue(s)        //值传递，形参修改不会影响到实参值
    fmt.Println("main s = ", s) //main s =  {1 Luffy 109 18 EastSea}
}
```

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

## 传引用

```
func printPointer(p *Student) {
    p.id = 250
    //printPointer p =  &{250 Luffy 109 18 EastSea}
    fmt.Println("printPointer p = ", p)
}

func main() {
    var s Student = Student{1, "Luffy", 'm', 18, "EastSea"}

    printPointer(&s)      //传引用(地址)，形参修改会影响到实参值
    fmt.Println("main s = ", s) //main s = {250 Luffy 109 18 EastSea}
}
```

# 文件操作

## 字符串处理函数

字符串在开发中使用频率较高，我们经常需要对字符串进行拆分、判断等操作，可以借助Go标准库中的strings包快速达到处理字符串的目录。除Contains、Join、Trim、Replace等我们学过的字符串处理函数之外，以下函数也常常会被用到。

### 字符串分割

```
func Split(s, sep string) []string
功能：把s字符串按照sep分割，返回slice
```

参1：s，表示待拆分的字符串

参2：sep，表示分割符，该参数为string 类型

返回值：切片，存储拆分好的子串

示例代码：

```
    fmt.Printf("%q\n", strings.Split("a,b,c", ","))
    fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
    fmt.Printf("%q\n", strings.Split(" xyz ", ""))
    fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
    //运行结果:
    //["a" "b" "c"]
    //["" "man " "plan " "canal panama"]
    //[" " "x" "y" "z" " "]
    //[""]
```

### 按空格拆分字符串

```
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
```

参1：s，表示待拆分的字符串

返回值：切片，存储拆分好的子串

示例代码：

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz   "))
//运行结果:Fields are: ["foo" "bar" "baz"]
```

### 判断字符串后缀

```
func HasSuffix(s, suffix string) bool
功能：判断s字符串是否有后缀子串suffix
```

参1：s，表示待判定字符串

参2：suffix，表示前缀子串

返回值：true or false

示例代码：

```
fmt.Printf("%v\n", strings.HasSuffix("World Cup.png", ".png"))
//运行结果:true
```

### 判断字符串前缀

```
func HasPrefix(s, prefix string) bool
功能：判断s字符串是否有前缀子串suffix
```

参1：s，表示待判定字符串

参2：prefix，表示前缀子串

返回值：true or false

示例代码：

```
fmt.Printf("%v\n", strings.HasPrefix("World Cup.png", "world"))
//运行结果:false
```

## 文件操作常用API

## 建立与打开文件

新建文件可以通过如下两个方法：

```
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。
```

通过如下两个方法来打开文件：

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

Open()是以只读权限打开文件名为name的文件，得到的文件指针file，只能用来对文件进行“读”操作。如果我们有“写”文件的需求，就需要借助Openfile函数来打开了。

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

OpenFile()可以选择打开name文件的读写权限。这个函数有三个默认参数：

参1：name，表示打开文件的路径。可使用相对路径 或 绝对路径

参2：flg，表示读写模式，常见的模式有：

**O_RDONLY**(只读模式), **O_WRONLY**(只写模式), **O_RDWR**(可读可写模式), **O_APPEND**(追加模式)。

参3：perm，表权限取值范围（0-7），表示如下：

0：没有任何权限

1：执行权限(如果是可执行文件，是可以运行的)

2：写权限

3: 写权限与执行权限

4：读权限

5: 读权限与执行权限

6: 读权限与写权限

7: 读权限，写权限，执行权限

关闭文件函数：

```
func (f *File) Close() error
```

## 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
```

## 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
```

## 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
```

## 大文件拷贝

示例代码：

```
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    args := os.Args //获取命令行参数，  并判断输入是否合法

    if args == nil || len(args) != 3 {
        fmt.Println("useage : xxx srcFile dstFile")
        return
    }

    srcPath := args[1] //获取参数1
    dstPath := args[2] //获取参数2
    fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

    if srcPath == dstPath {
        fmt.Println("error：源文件名 与 目的文件名雷同")
```

```
        return
    }

    srcFile, err1 := os.Open(srcPath) // 打开源文件
    if err1 != nil {
        fmt.Println(err1)
        return
    }

    dstFile, err2 := os.Create(dstPath) //创建目标文件
    if err2 != nil {
        fmt.Println(err2)
        return
    }

    buf := make([]byte, 1024) //切片缓冲区
    for {
        //从源文件读取内容，n为读取文件内容的长度
        n, err := srcFile.Read(buf)
        if err != nil && err != io.EOF {
            fmt.Println(err)
            break
        }

        if n == 0 {
            fmt.Println("文件处理完毕")
            break
        }

        //切片截取
        tmp := buf[:n]
        //把读取的内容写入到目的文件
        dstFile.Write(tmp)
    }

    //关闭文件
    srcFile.Close()
    dstFile.Close()
}
```

# 目录操作常用API

我们读写的文件一般存放于目录中。因此，有时需要指定到某一个目录下，根据目录存储的状况再进行文件的特定操作。接下来我们看看目录的基本操作方法。

## 打开目录

打开目录我们也使用 OpenFile 函数，但要指定不同的参数来通知系统，要打开的是一个目录文件。

```
func OpenFile(name string, flag int, perm FileMode) (*File, error)
```

参数1：name，表示要打开的目录名称。使用绝对路径较多

参数2：flg，表示打开文件的读写模式。可选择：

```
O_RDONLY只读模式、O_WRONLY只写模式、O_RDWR读写模式
```

参数3：perm，表示打开权限。但对于目录来说略有不同。通常传os.ModeDir。

返回值：由于是操作目录，所以file是指向目录的文件指针。error中保存错误信息。

## 读目录内容

这与读文件有所不同。目录中存放的是文件名和子目录名。所以使用Readdir函数来完成。

```
func (f *File) Readdir(n int) ([]FileInfo, error)
```

参数：n,表读取目录的成员个数。**通常传 -1**,表读取目录所有文件对象。

返回值：FileInfo类型的切片。其内部保存了文件名。error中保存错误信息。

```
type FileInfo interface {
   Name() string       // base name of the file
   Size() int64        // length in bytes for regular files; system-dependent for others
   Mode() FileMode     // file mode bits
   ModTime() time.Time // modification time
   IsDir() bool        // abbreviation for Mode().IsDir()
   Sys() interface{}   // underlying data source (can return nil)
}
```

得到 FileInfo类型切片后，我们可以range遍历切片元素，使用.Name()获取文件名。使用.Size()获取文件大小，使用.IsDir()判断文件是目录还是非目录文件。

如：我们可以提示用户提供一个目录位置，打开该目录，查看目录下的所有成员，并判别他们是文件还是目录。

示例代码：

```
func main()  {
   fmt.Println("请输入要找寻的目录：")
   var path string
   fmt.Scan(&path)                    // 获取用户指定的目录名

   dir, _ := os.OpenFile(path, os.O_RDONLY, os.ModeDir)   // 只读打开该目录

   names, _ := dir.Readdir(-1)        // 读取当前目录下所有的文件名和目录名，存入names切片

   for _, name := range names {       // 遍历切片，获取文件/目录名
      if !name.IsDir() {
         fmt.Println(name.Name(), "是一个文件")
      } else {
         fmt.Println(name.Name(), "是一个目录")
      }
```

```
    }
}
```

### 其他目录操作API

其实，目录也可以看成“文件”。我们通常读写的文件内容是可见的ASCII码。目录文件的内容就是文件名和目录名，称之为目录项。我们读写目录文件，实质上就是在读写目录项。

目录操作还有其他的一系列API，这里简单罗列几个较为常用的，大家可自行酌情学习。

将当前工作目录修改为dir指定的目录：

```
func Chdir(dir string) error
```

返回当前工作目录的绝对路径：

```
func Getwd() (dir string, err error)
```

使用指定的权限和名称创建一个目录：

```
func Mkdir(name string, perm FileMode) error
```

获取更多文件、目录操作API可查看Go标库文档： https://studygolang.com/pkgdoc

# Go并发编程

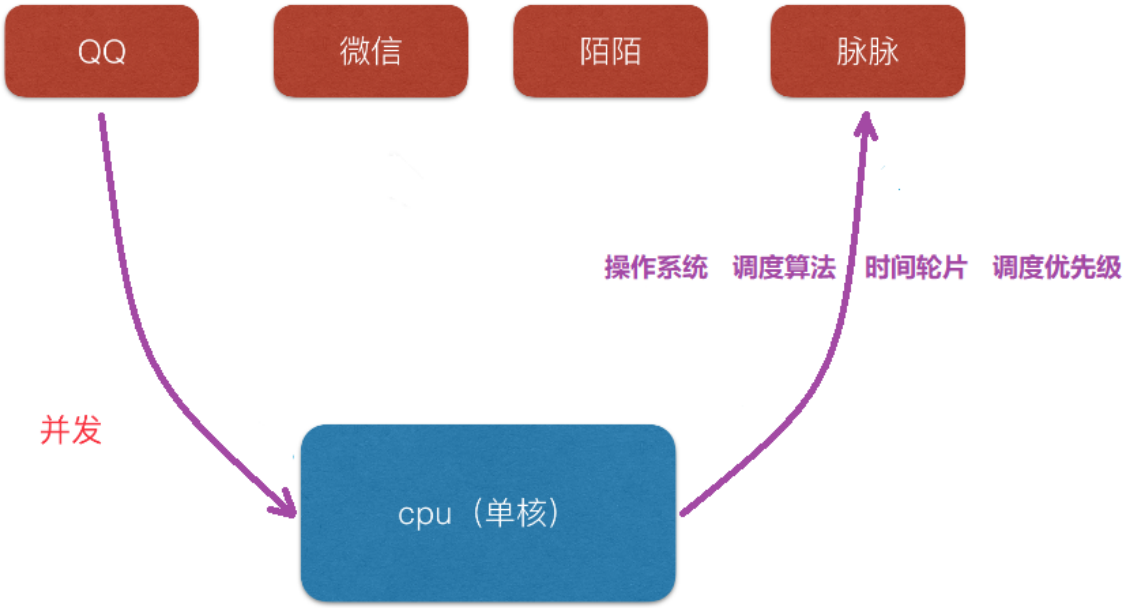
## 概述

简而言之，所谓并发编程是指在一台处理器上“同时”处理多个任务。

随着硬件的发展，并发程序变得越来越重要。Web服务器会一次处理成千上万的请求。平板电脑和手机app在渲染用户画面同时还会后台执行各种计算任务和网络请求。即使是传统的批处理问题--读取数据，计算，写输出--现在也会用并发来隐藏掉I/O的操作延迟以充分利用现代计算机设备的多个核心。计算机的性能每年都在以非线性的速度增长。

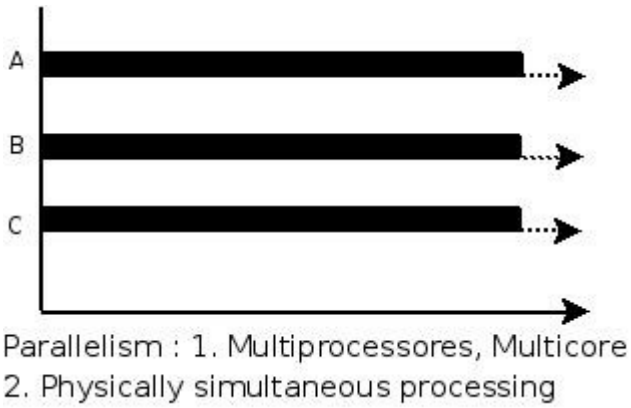
宏观的并发是指在一段时间内，有多个程序在同时运行。

并发在微观上，是指在同一时刻只能有一条指令执行，但多个程序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，使多个程序快速交替的执行。

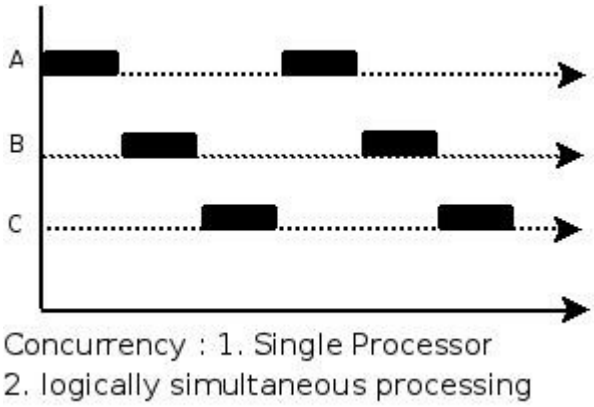



## 并行和并发

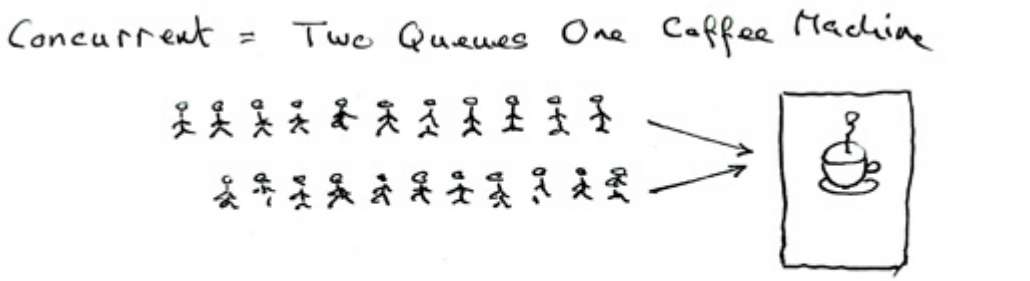
**并行(parallel)**：指在同一时刻，有多条指令在多个处理器上同时执行。

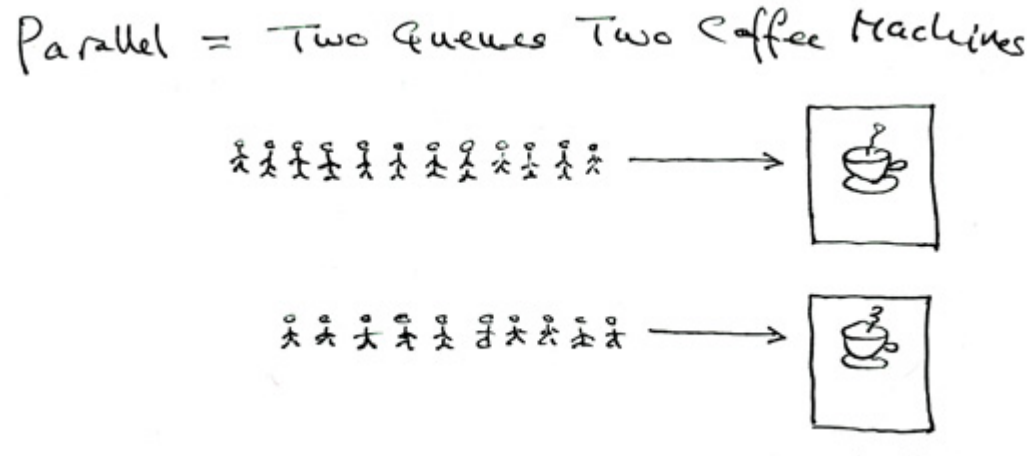



**并发(concurrency)**：**指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，通过cpu**时间片轮转使多个进程快速交替的执行。




以咖啡机的例子来解释并行和并发的区别。




l **并行**是两个队列**同时**使用**两台**咖啡机 （真正的多任务）

l **并发**是两个队列**交替**使用**一台**咖啡机 （ 假 的多任务）

# 常见并发编程技术

## 进程并发

### 程序和进程

程序，是指编译好的二进制文件，在磁盘上，不占用系统资源(cpu、内存、打开的文件、设备、锁....)

进程，是一个抽象的概念，与操作系统原理联系紧密。进程是活跃的程序，占用系统资源。在内存中执行。(程序运行起来，产生一个进程)

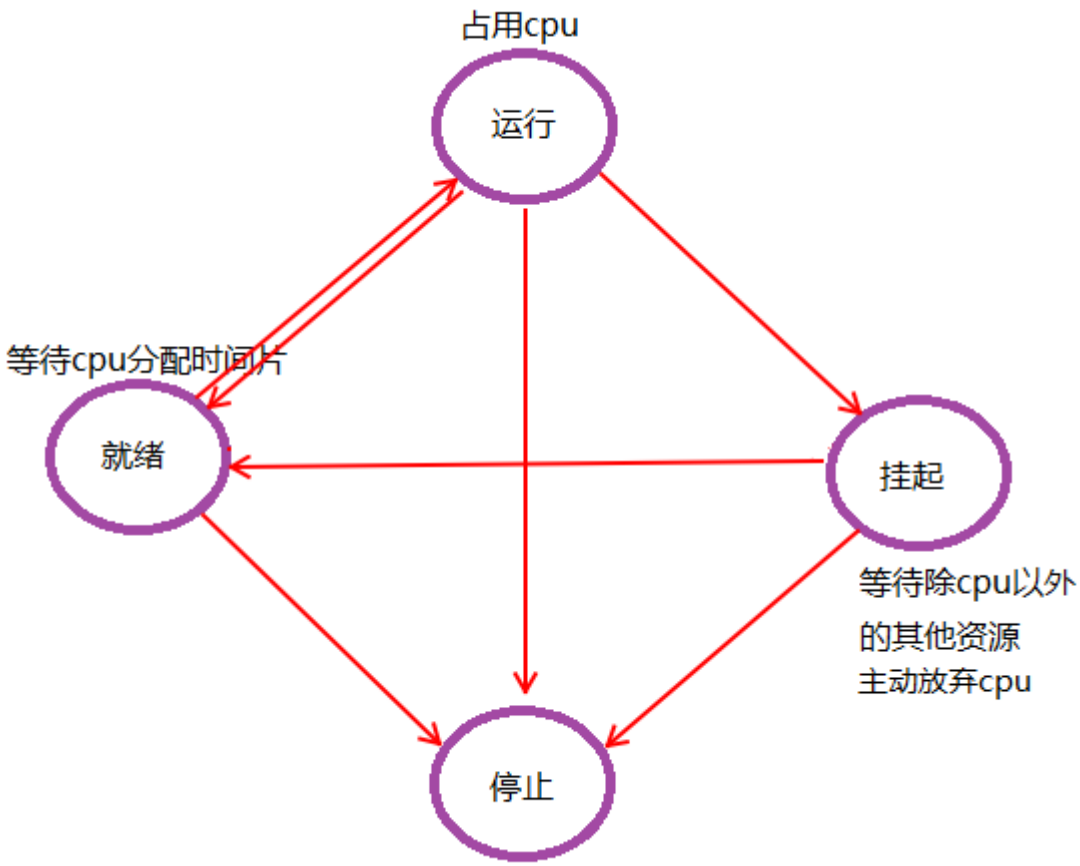
程序→ 剧本(纸) 进程→ 戏(舞台、演员、灯光、道具...)

同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)

如：同时开两个终端。各自都有一个bash但彼此ID不同。

## 进程状态

进程基本的状态有5种。分别为初始态，就绪态，运行态，挂起态与终止态。其中初始态为进程准备阶段，常与就绪态结合来看。




## 进程并发

在使用进程 实现并发时会出现什么问题呢?

1：系统开销比较大，占用资源比较多，开启进程数量比较少。

2：在unix/linux系统下，还会产生“孤儿进程”和“僵尸进程”。

通过前面查看操作系统的进程信息，我们知道在操作系统中，可以产生很多的进程。在unix/linux系统中，正常情况下，子进程是通过父进程fork创建的，子进程再创建新的进程。

并且父进程永远无法预测子进程到底什么时候结束。 当一个 进程完成它的工作终止之后，它的父进程需要调用系统调用取得子进程的终止状态。

**孤儿进程**

孤儿进程: 父进程先于子进程结束，则子进程成为孤儿进程，子进程的父进程成为init进程，称为init进程领养孤儿进程。

**僵尸进程**

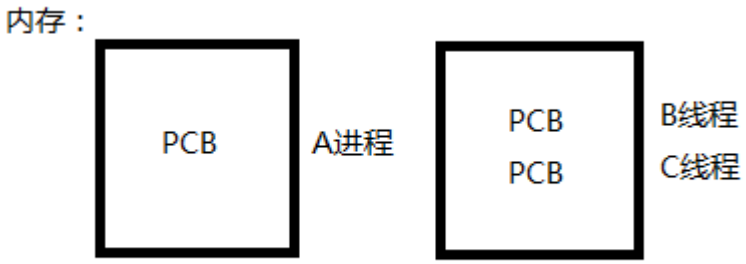
僵尸进程: 进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

Windows下的进程和Linux下的进程是不一样的，它比较懒惰，从来不执行任何东西，只是为线程提供执行环境。然后由线程负责执行包含在进程的地址空间中的代码。当创建一个进程的时候，操作系统会自动创建这个进程的第一个线程，成为主线程。

# 线程并发

## 什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程 (Linux下)




进程：独立地址空间，拥有PCB

线程：有独立的PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。独居(进程)；合租(线程)。

线程：最小的执行单位

进程：最小分配资源单位，可看成是只有一个线程的进程。

Windows系统下，可以直接忽略进程的概念，只谈线程。因为线程是最小的执行单位，是被系统独立调度和分派的基本单位。而进程只是给线程提供执行环境。

## 线程同步

同步即协同步调，按预定的先后次序运行。

**线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。**

举例1： 银行存款 5000。柜台，折：取3000；提款机，卡：取 3000。剩余：2000

举例2： 内存中100字节，线程T1欲填入全1，线程T2欲填入全0。但如果T1执行了50个字节失去cpu，T2执行，会将T1写过的内容覆盖。当T1再次获得cpu继续 从失去cpu的位置向后写入1，当执行结束，内存中的100字节，既不是全1，也不是全0。

产生的现象叫做“与时间有关的错误”(time related)。为了避免这种数据混乱，线程需要同步。

“同步”的目的，是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等等都需要同步机制。

因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步。**

锁的应用

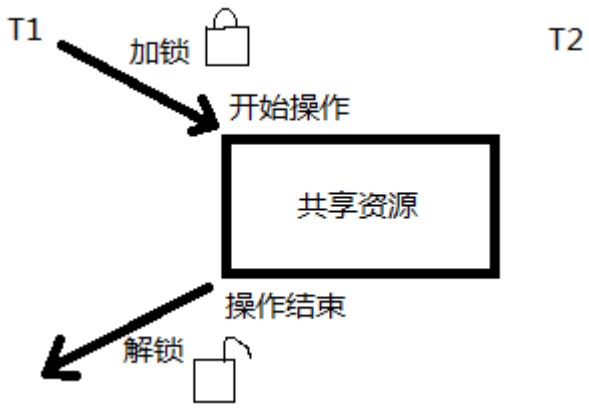
**互斥量mutex**

Linux中提供一把互斥锁mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。




但，应注意：同一时刻，只能有一个线程持有该锁。

当A线程对某个全局变量加锁访问，B在访问前尝试加锁，拿不到锁，B阻塞。C线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

**读写锁**

与互斥量类似，但读写锁允许更高的并行性。其特性为：**写独占，读共享。**

l **读写锁状态：**

特别强调：读写锁**只有一把**，但其具备两种状态：

1. 读模式下加锁状态 (读锁)
2. 写模式下加锁状态 (写锁)

l **读写锁特性：**

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

读写锁非常适合于对数据结构读的次数远大于写的情况。

## 协程并发

协程：coroutine。也叫轻量级线程。

与传统的系统级线程和进程相比，协程最大的优势在于“轻量级”。可以轻松创建上万个而不会导致系统资源衰竭。而线程和进程通常很难超过1万个。这也是协程别称“轻量级线程”的原因。

一个线程中可以有任意多个协程，但某一时刻只能有一个协程在运行，**多个协程分享该线程分配到的计算机资源。**

多数语言在语法层面并不直接支持协程，而是通过库的方式支持，但用库的方式支持的功能也并不完整，比如仅仅提供协程的创建、销毁与切换等能力。如果在这样的轻量级线程中调用一个同步 IO 操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行轻量级线程，从而无法真正达到轻量级线程本身期望达到的目标。

在协程中，调用一个任务就像调用一个函数一样，消耗的系统资源最少！但能达到进程、线程并发相同的效果。

在一次并发任务中，进程、线程、协程均可以实现。从系统资源消耗的角度出发来看，进程相当多，线程次之，协程最少。

## Go并发

Go 在语言级别支持协程，叫goroutine。Go 语言标准库提供的所有系统调用操作（包括所有同步IO操作），都会出让CPU给其他goroutine。这让轻量级线程的切换管理不依赖于系统的线程和进程，也不需要依赖于CPU的核心数量。

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。

Go语言为并发编程而内置的上层API基于顺序通信进程模型CSP(communicatingsequential processes)。这就意味着显式锁都是可以避免的，因为Go通过相对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。

Go语言中的并发程序主要使用两种手段来实现。goroutine和channel。

# Goroutine

## 什么是Goroutine

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

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

## Goroutine的创建

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

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

示例代码：

```
package main

import (
    "fmt"
    "time"
)

func newTask() {
    i := 0
    for {
        i++
        fmt.Printf("new goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时1s
    }
}

func main() {
    //创建一个 goroutine，启动另外一个任务
    go newTask()
    i := 0
    //main goroutine 循环打印
    for {
        i++
        fmt.Printf("main goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时1s
    }
}
```

程序运行结果：

```
main goroutine: i = 1
new goroutine: i = 1
main goroutine: i = 2
new goroutine: i = 2
new goroutine: i = 3
main goroutine: i = 3
main goroutine: i = 4
new goroutine: i = 4
```

## Goroutine特性

主goroutine退出后，其它的工作goroutine也会自动退出：

```
package main

import (
"fmt"
"time"
)

func newTask() {
    i := 0
    for {
        i++
        fmt.Printf("new goroutine: i = %d\n", i)
        time.Sleep(1 * time.Second) //延时1s
    }
}

func main() {
    //创建一个 goroutine，启动另外一个任务
    go newTask()

    fmt.Println("main goroutine exit")
}
```

程序运行结果：

```
main goroutine exit
成功: 进程退出代码 0.
```

# runtime包

## 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，并在下次再获得cpu时间轮片的时候，从该出让cpu的位置恢复执行。

有点像跑接力赛，A跑了一会碰到代码runtime.Gosched()就把接力棒交给B了，A歇着了，B继续跑。

示例代码：

```
package main

import (
"fmt"
"runtime"
)

func main() {
    //创建一个goroutine
    go func(s string) {
        for i := 0; i < 2; i++ {
            fmt.Println(s)
```

```
        }
    }("world")

    for i := 0; i < 2; i++ {
        runtime.Gosched()  //import "runtime" 包
        /*
            屏蔽runtime.Gosched()运行结果如下:
                hello
                hello

            没有runtime.Gosched()运行结果如下:
                world
                world
                hello
                hello
        */
        fmt.Println("hello")
    }
}
```

以上程序的执行过程如下:

主协程进入main()函数，进行代码的执行。当执行到go func()匿名函数时，创建一个新的协程，开始执行匿名函数中的代码，主协程继续向下执行，执行到runtime.Gosched( )时会暂停向下执行，直到其它协程执行完后，再回到该位置，主协程继续向下执行。

## Goexit

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

示例代码:

```
package main

import (
"fmt"
"runtime"
)

func main() {
    go func() {
        defer fmt.Println("A.defer")

        func() {
            defer fmt.Println("B.defer")
            runtime.Goexit() // 终止当前 goroutine, import "runtime"
            fmt.Println("B") // 不会执行
        }()

        fmt.Println("A") // 不会执行
    }()     //不要忘记()
```

```
    //死循环，目的不让主goroutine结束
    for {
    }
}
```

程序运行结果：


### GOMAXPROCS

调用 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。

示例代码：

```
package main

import (
    "fmt"
)

func main() {
//n := runtime.GOMAXPROCS(1)    // 第一次 测试
//打印结果：11111111111111111111000000000000000000000011111...

n := runtime.GOMAXPROCS(2)         // 第二次 测试
//打印结果：01010101010101010101100110010101101001010100110...
    fmt.Printf("n = %d\n", n)

    for {
        go fmt.Print(0)
        fmt.Print(1)
    }
}
```

在第一次执行runtime.GOMAXPROCS(1)时，最多同时只能有一个goroutine被执行。所以会打印很多1。过了一段时间后，GO调度器会将其置为休眠，并唤醒另一个goroutine，这时候就开始打印很多0了，在打印的时候，goroutine是被调度到操作系统线程上的。

在第二次执行runtime.GOMAXPROCS(2)时， 我们使用了两个CPU，所以两个goroutine可以一起被执行，以同样的频率交替打印0和1。

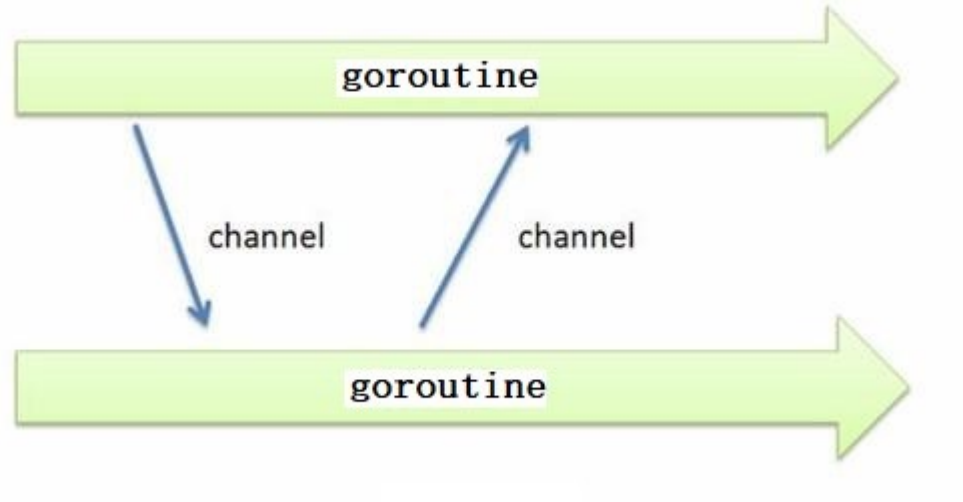
# channel

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

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

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

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




## 定义channel变量

和map类似，channel也一个对应make创建的底层数据结构的**引用**。

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

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

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

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

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

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

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

```
    channel <- value      //发送value到channel
    <-channel             //接收并将其丢弃
    x := <-channel        //从channel中接收数据，并赋值给x
    x, ok := <-channel    //功能同上，同时检查通道是否已关闭或者是否为空
```

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

示例代码：

```
package main

import (
    "fmt"
)

func main() {
    c := make(chan int)

    go func() {
        defer fmt.Println("子协程结束")

        fmt.Println("子协程正在运行……")

        c <- 666 //666发送到c
    }()

    num := <-c //从c中接收数据，并赋值给num

    fmt.Println("num = ", num)
    fmt.Println("main协程结束")
}
```

程序运行结果：

```
子协程正在运行……
子协程结束
num =  666
main协程结束
成功：进程退出代码 0.
```

## 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。

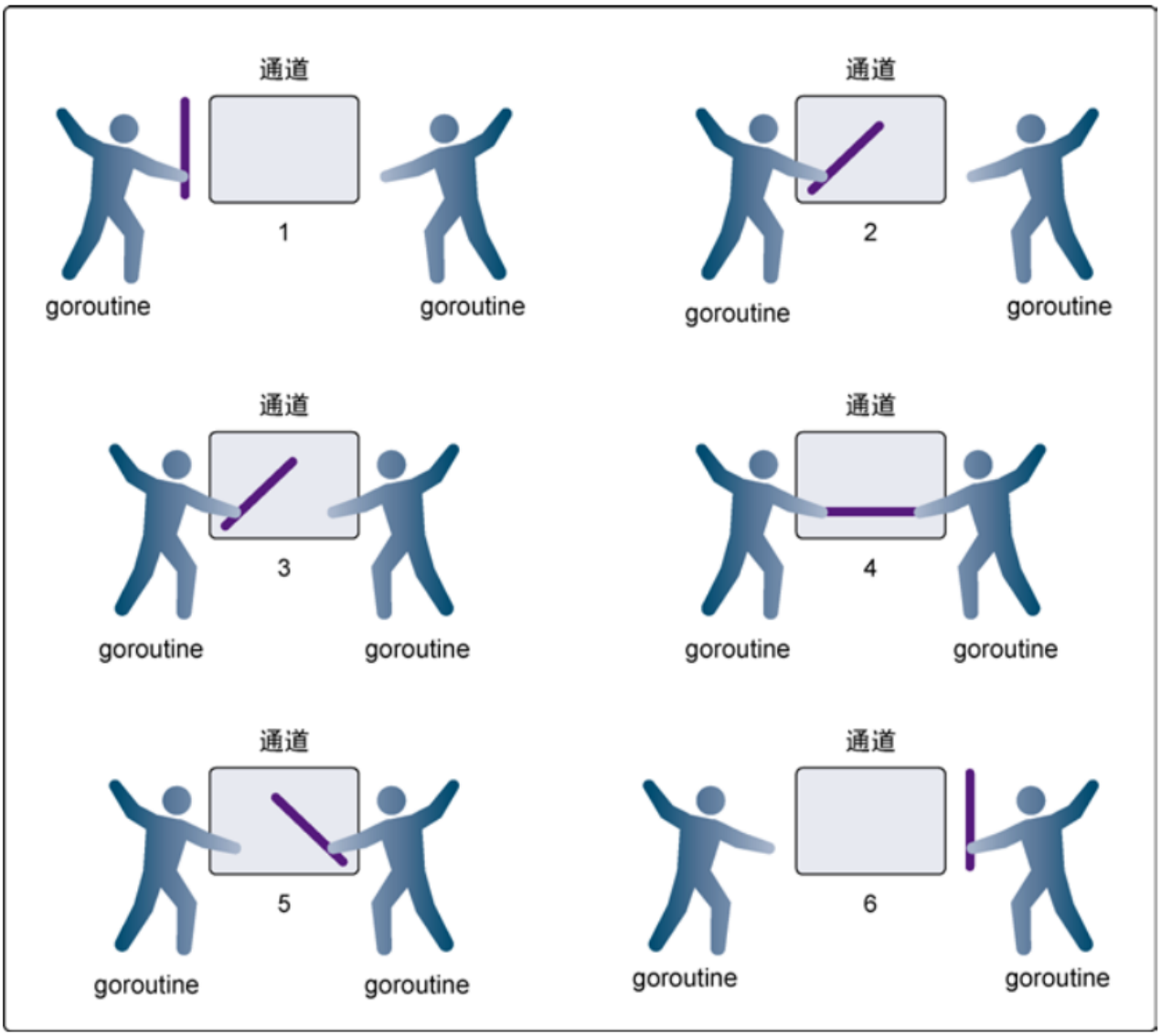
这种类型的通道要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。否则，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。

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

**阻塞：**由于某种原因数据没有到达，当前协程（线程）持续处于等待状态，直到条件满足，才接触阻塞。

**同步：**在两个或多个协程（线程）间，保持数据内容一致性的机制。

下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值：




使用无缓冲的通道在 goroutine 之间同步

l 在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。

l 在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。

l 在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。

l 在第 4 步和第 5 步，进行交换，并最终，在第6 步，两个 goroutine都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做别的事情了。

无缓冲的channel创建格式：

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

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

示例代码：

```
package main

import (
    "fmt"
    "time"
)

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

    //内置函数 len 返回未被读取的缓冲元素数量，cap 返回缓冲区大小
    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

    go func() {
        defer fmt.Println("子协程结束")

        for i := 0; i < 3; i++ {
            c <- i
            fmt.Printf("子协程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))
        }
    }()

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

    for i := 0; i < 3; i++ {
        num := <-c //从c中接收数据，并赋值给num
        fmt.Println("num = ", num)
    }

    fmt.Println("main协程结束")
}
```

程序运行结果：

```
len(c)=0, cap(c)=0
子协程正在运行[0]: len(c)=0, cap(c)=0
num =  0
num =  1
子协程正在运行[1]: len(c)=0, cap(c)=0
子协程正在运行[2]: len(c)=0, cap(c)=0
子协程结束
num =  2
main协程结束
```

## 有缓冲的channel

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

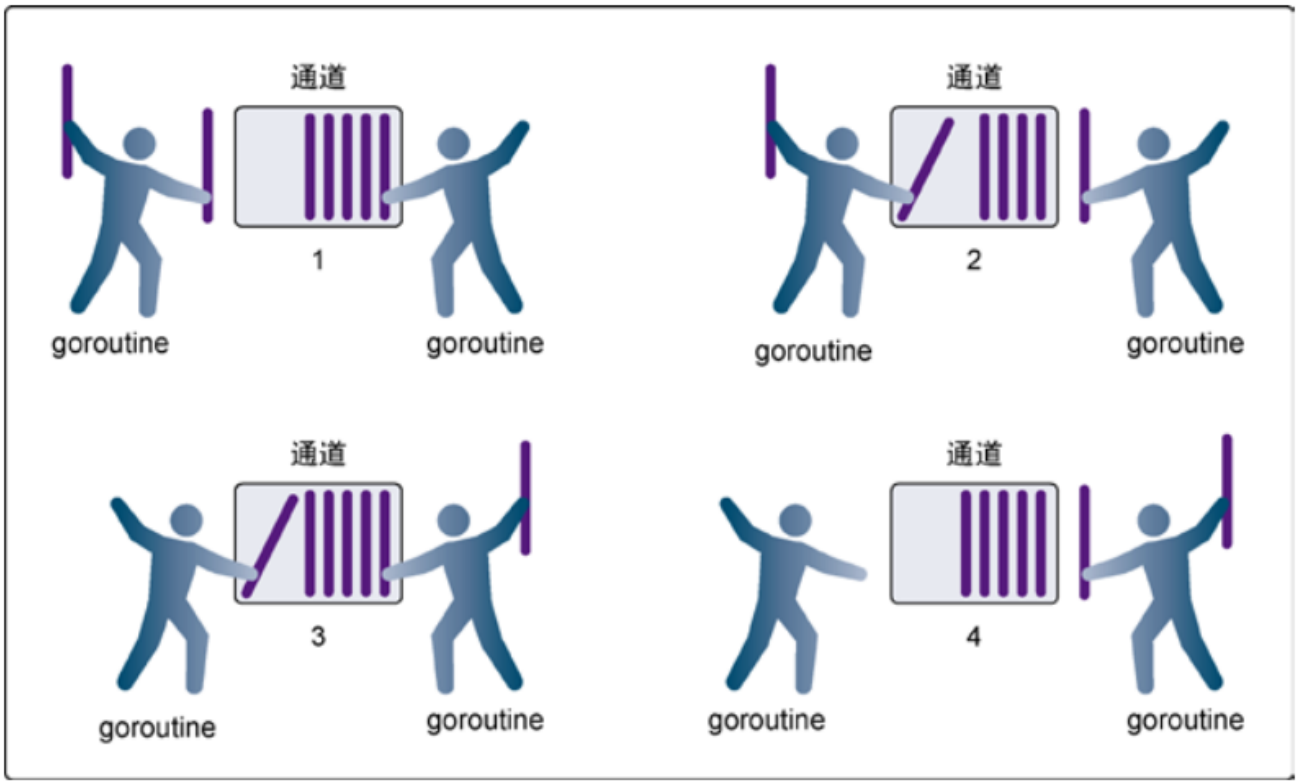
这种类型的通道并不强制要求goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也不同。

只有通道中没有要接收的值时，接收动作才会阻塞。

只有通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

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

示例图如下：




使用有缓冲的通道在 goroutine 之间同步数据

l 在第 1 步，右侧的 goroutine 正在从通道接收一个值。

l 在第 2 步，右侧的这个 goroutine独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。

l 在第 3 步，左侧的goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。

l 最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓冲的channel创建格式：

```
make(chan Type, capacity)
```

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

示例代码：

```
func main() {
    c := make(chan int, 3) //带缓冲的通道

    //内置函数 len 返回未被读取的缓冲元素数量， cap 返回缓冲区大小
    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

    go func() {
        defer fmt.Println("子协程结束")

        for i := 0; i < 3; i++ {
            c <- i
            fmt.Printf("子协程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))
        }
    }()

    time.Sleep(2 * time.Second) //延时2s
    for i := 0; i < 3; i++ {
        num := <-c //从c中接收数据，并赋值给num
        fmt.Println("num = ", num)
    }
    fmt.Println("main协程结束")
}
```

程序运行结果：

```
len(c)=0, cap(c)=3
子协程正在运行[0]: len(c)=0, cap(c)=3
子协程正在运行[1]: len(c)=1, cap(c)=3
子协程正在运行[2]: len(c)=2, cap(c)=3
子协程结束
num =  0
num =  1
num =  2
main协程结束
```

## 关闭channel

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

示例代码：

```
package main

import (
    "fmt"
)
```

```
func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            c <- i
        }
        //把 close(c) 注释掉，程序会一直阻塞在 if data, ok := <-c; ok 那一行
        close(c)
    }()

    for {
        //ok为true说明channel没有关闭，为false说明管道已经关闭
        if data, ok := <-c; ok {
            fmt.Println(data)
        } else {
            break
        }
    }

    fmt.Println("Finished")
}
```

程序运行结果：

```
0
1
2
3
4
Finished
```

**注意：**

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，可以继续从channel接收数据；

l 对于nil channel，无论收发都会被阻塞。

可以使用**range** 来迭代不断操作channel：

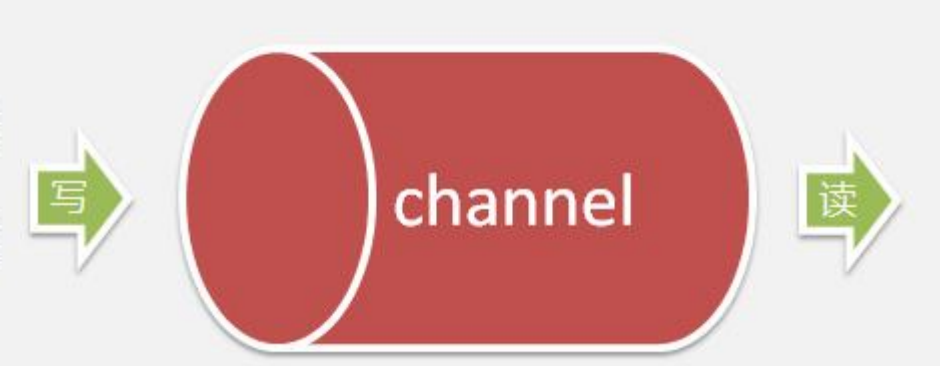
```
package main

import (
    "fmt"
)
```

```
func main() {
    c := make(chan int)

    go func() {
        for i := 0; i < 5; i++ {
            c <- i
        }
        //把 close(c) 注释掉，程序会一直阻塞在 for data := range c 那一行
        close(c)
    }()

    for data := range c {
        fmt.Println(data)
    }
    fmt.Println("Finished")
}
```

## 单向channel及应用

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

但是，我们经常见一个通道作为参数进行传递而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，是双向的
var ch2 chan<- float64 // ch2是单向channel，只用于写float64数据
var ch3 <-chan int     // ch3是单向channel，只用于读int数据
```

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。

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

```
    c := make(chan int, 3)
    var send chan<- int = c // send-only
    var recv <-chan int = c // receive-only
    send <- 1
    //<-send //invalid operation: <-send (receive from send-only type chan<- int)
    <-recv
    //recv <- 2 //invalid operation: recv <- 2 (send to receive-only type <-chan int)

    //不能将单向 channel 转换为普通 channel
    d1 := (chan int)(send) //cannot convert send (type chan<- int) to type chan int
    d2 := (chan int)(recv) //cannot convert recv (type <-chan int) to type chan int
```

示例代码：

```
// chan<- //只写
func counter(out chan<- int) {
    defer close(out)
    for i := 0; i < 5; i++ {
        out <- i //如果对方不读 会阻塞
    }
}

// <-chan //只读
func printer(in <-chan int) {
    for num := range in {
        fmt.Println(num)
    }
}

func main() {
    c := make(chan int) //   chan   //读写

    go counter(c) //生产者
    printer(c)    //消费者

    fmt.Println("done")
}
```

## 生产者消费者模型

单向channel最典型的应用是“生产者消费者模型”

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

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


举一个寄信的例子来辅助理解一下，假设你要寄一封平信，大致过程如下：

1．把信写好——相当于生产者制造数据

2．把信放入邮筒——相当于生产者把数据放入缓冲区

3．邮递员把信从邮筒取出——相当于消费者把数据取出缓冲区

4．邮递员把信拿去邮局做相应的处理——相当于消费者处理数据

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

缓冲区的好处大概如下：

**1：解耦**

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

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

**2：处理并发**

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

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

其实最当初这个生产者消费者模式，主要就是用来处理并发问题的。

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

**3：缓存**

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

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

**示例代码：**

```
package main

import "fmt"
```

```
// 此通道只能写，不能读。
func producer(out chan<- int)  {
   for i:= 0; i < 10; i++ {
      out <- i*i                        // 将 i*i 结果写入到只写channel
   }
   close(out)
}

// 此通道只能读，不能写
func consumer(in <-chan int)  {
   for num := range in {                // 从只读channel中获取数据
      fmt.Println("num =", num)
   }
}

func main()  {
   ch := make(chan int)         // 创建一个双向channel

   // 新建一个groutine， 模拟生产者，产生数据，写入 channel
   go producer(ch)              // channel传参， 传递的是引用。

   // 主协程，模拟消费者，从channel读数据，打印到屏幕
   consumer(ch)                 // 与 producer 传递的是同一个 channel
}
```

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

注意：channel作为参数传递，是**引用传递。**

模拟订单

在实际的开发中，生产者消费者模式应用也非常的广泛，例如：在电商网站中，订单处理，就是非常典型的生产者消费者模式。

当很多用户单击下订单按钮后，订单生产的数据全部放到缓冲区（队列）中，然后消费者将队列中的数据取出来发送者仓库管理等系统。

通过生产者消费者模式，将订单系统与仓库管理系统隔离开，且用户可以随时下单（生产数据）。如果订单系统直接调用仓库系统，那么用户单击下订单按钮后，要等到仓库系统的结果返回。这样速度会很慢。

下面模拟一个下订单处理的过程。

```
package main

import "fmt"

type OrderInfo struct {     // 创建结构体类型OrderInfo，只有一个id 成员
   id int
}
```

```
func producer2(out chan <- OrderInfo)  {        // 生成订单—生产者

   for i:=0; i<10; i++ {                // 循环生成10份订单
      order := OrderInfo{id: i+1}
      out <- order                              // 写入channel
   }
   close(out)                           // 写完，关闭channel
}

func consumer2(in <- chan OrderInfo)  {       // 处理订单—消费者

   for order := range in {                      // 从channel 取出订单
      fmt.Println("订单id为: ", order.id)    // 模拟处理订单
   }
}

func main()  {
   ch := make(chan OrderInfo)  // 定义一个双向 channel， 指定数据类型为OrderInfo
   go producer2(ch)            // 建新协程，传只写channel
   consumer2(ch)               // 主协程，传只读channel
}
```

OrderInfo为订单信息，这里为了简单只定义了一个订单编号属性，然后生产者模拟10个订单，消费者对产生的订单进行处理。

## 定时器

time.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。

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

它提供一个channel，在定时时间到达之前，没有数据写入timer.C会一直阻塞。直到定时时间到，向channel写入值，阻塞解除，可以从中读取数据。

示例代码:

```
package main

import (
        "fmt"
        "time"
)

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

```
    timer1 := time.NewTimer(time.Second * 2)
    t1 := time.Now() //当前时间
    fmt.Printf("t1: %v\n", t1)

    t2 := <-timer1.C
    fmt.Printf("t2: %v\n", t2)

    //如果只是想单纯的等待的话，可以使用 time.Sleep 来实现
    timer2 := time.NewTimer(time.Second * 2)
    <-timer2.C
    fmt.Println("2s后")

    time.Sleep(time.Second * 2)
    fmt.Println("再一次2s后")

    <-time.After(time.Second * 2)
    fmt.Println("再再一次2s后")

    timer3 := time.NewTimer(time.Second)
    go func() {
        <-timer3.C
        fmt.Println("Timer 3 expired")
    }()

    stop := timer3.Stop() //停止定时器
    if stop {
        fmt.Println("Timer 3 stopped")
    }

    fmt.Println("before")
    timer4 := time.NewTimer(time.Second * 5)    //原来设置3s
    timer4.Reset(time.Second * 1)               //重新设置时间
    <-timer4.C
    fmt.Println("after")
}
```

## 定时器的常用操作：

**1. 实现延迟功能**

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

fmt.Println("时间到")

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

fmt.Println("时间到")

3) 延时2s后打印一句话

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

<- timer.C

fmt.Println("时间到")

### 2. 定时器停止

```
    timer := time.NewTimer(3 * time.Second)
    go func() {
        <-timer.C
        fmt.Println("子协程可以打印了，因为定时器的时间到")
    }()
    timer.Stop() //停止定时器

    for {
    }
```

### 3. 定时器重置

```
    timer := time.NewTimer(3 * time.Second)
    ok := timer.Reset(1 * time.Second) //重新设置为1s
    fmt.Println("ok = ", ok)
    <-timer.C
    fmt.Println("时间到")
```

## time.Ticker

Ticker是一个周期触发定时的计时器，它会按照一个时间间隔往channel发送系统当前时间，而channel的接收者可以以固定的时间间隔从channel中读取事件。

```
type Ticker struct {
   C <-chan Time        // The channel on which the ticks are delivered.
   r runtimeTimer
}
```

示例代码：

```
package main

import (
        "fmt"
        "time"
)

func main() {
    //创建定时器，每隔1秒后，定时器就会给channel发送一个事件(当前时间)
    ticker := time.NewTicker(time.Second * 1)

    i := 0
    go func() {
        for { //循环
            <-ticker.C
            i++
```

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

            if i == 5 {
                ticker.Stop() //停止定时器
            }
        }
    }() //别忘了()

    //死循环，特地不让main goroutine结束
    for {
    }
}
```

# select

## select作用

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

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

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

```
select {
    case <-chan1:
        // 如果chan1成功读到数据，则进行该case处理语句
    case chan2 <- 1:
        // 如果成功向chan2写入数据，则进行该case处理语句
    default:
        // 如果上面都没有成功，则进入default处理流程
    }
```

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

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

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

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。

示例代码：

```
package main

import (
    "fmt"
)
```

```
func fibonacci(c, quit chan int) {
    x, y := 1, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)

    go func() {
        for i := 0; i < 6; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()

    fibonacci(c, quit)
}
```

运行结果如下：

```
1
1
2
3
5
8
quit
```

## 超时

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

```
func main() {
    c := make(chan int)
    o := make(chan bool)
    go func() {
        for {

            select {
```

```
            case v := <-c:
                fmt.Println(v)
            case <-time.After(5 * time.Second):
                fmt.Println("timeout")
                o <- true
                break
            }
        }
    }()
    //c <- 666 // 注释掉，引发 timeout
    <-o
}
```

## (扩展进阶) 锁和条件变量

前面我们为了解决协程同步的问题我们使用了channel，但是GO也提供了传统的同步工具。

它们都在GO的标准库代码包sync和sync/atomic中。

下面我们看一下锁的应用。

什么是锁呢？就是某个协程（线程）在访问某个资源时先锁住，防止其它协程的访问，等访问完毕解锁后其他协程再来加锁进行访问。这和我们生活中加锁使用公共资源相似，例如：公共卫生间。

## 死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，

示例代码：

```
package main

import "fmt"

func main() {
    ch := make(chan int)
    ch <- 1         // I'm blocked because there is no channel read yet.
    fmt.Println("send")
    go func() {
        <-ch        // I will never be called for the main routine is blocked!
        fmt.Println("received")
    }()
    fmt.Println("over")
}
```

## 互斥锁

每个资源都对应于一个可称为 "互斥锁" 的标记，这个标记用来保证在任意时刻，只能有一个协程（线程）访问该资源。其它的协程只能等待。

互斥锁是传统并发编程对共享资源进行访问控制的主要手段，它由标准库sync中的Mutex结构体类型表示。sync.Mutex类型只有两个公开的指针方法，Lock和Unlock。Lock锁定当前的共享资源，Unlock进行解锁。

在使用互斥锁时，一定要注意：对资源操作完成后，一定要解锁，否则会出现流程执行异常，死锁等问题。通常借助defer。锁定后，立即使用defer语句保证互斥锁及时解锁。如下所示：

```
var mutex sync.Mutex            // 定义互斥锁变量 mutex

func write(){
   mutex.Lock( )
   defer mutex.Unlock( )
}
```

我们可以使用互斥锁来解决前面提到的多任务编程的问题，如下所示：

```
package main

import (
   "fmt"
   "time"
   "sync"
)

var mutex sync.Mutex

func printer(str string)  {
   mutex.Lock()                 // 添加互斥锁
   defer mutex.Unlock()                 // 使用结束时解锁

   for _, data := range str {           // 迭代器
      fmt.Printf("%c", data)
      time.Sleep(time.Second)       // 放大协程竞争效果
   }
   fmt.Println()
}

func person1(s1 string)  {
   printer(s1)
}

func person2()  {
   printer("world")             // 调函数时传参
}

func main()  {
   go person1("hello")                  // main 中传参
   go person2()
   for {
      ;
   }
```

```
}
```

程序执行结果与多任务资源竞争时一致。最终由于添加了互斥锁，可以按序先输出hello再输出 world。但这里需要我们自行创建互斥锁，并在适当的位置对锁进行释放。

## 读写锁

互斥锁的本质是当一个goroutine访问的时候，其他goroutine都不能访问。这样在资源同步，避免竞争的同时也降低了程序的并发性能。程序由原来的并行执行变成了串行执行。

其实，当我们对一个不会变化的数据只做“读”操作的话，是不存在资源竞争的问题的。因为数据是不变的，不管怎么读取，多少goroutine同时读取，都是可以的。

所以问题不是出在“读”上，主要是修改，也就是“写”。修改的数据要同步，这样其他goroutine才可以感知到。所以真正的互斥应该是读取和修改、修改和修改之间，读和读是没有互斥操作的必要的。

因此，衍生出另外一种锁，叫做**读写锁**。

读写锁可以让多个读操作并发，同时读取，但是对于写操作是完全互斥的。也就是说，当一个goroutine进行写操作的时候，其他goroutine既不能进行读操作，也不能进行写操作。

GO中的读写锁由结构体类型sync.RWMutex表示。此类型的方法集合中包含两对方法：

一组是对写操作的锁定和解锁，简称“写锁定”和“写解锁”：

func (*RWMutex)Lock()

func (*RWMutex)Unlock()

另一组表示对读操作的锁定和解锁，简称为“读锁定”与“读解锁”：

func (*RWMutex)RLock()

func (*RWMutex)RUlock()

读写锁基本示例：

```
package main

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

var count int                    // 全局变量count
var rwlock sync.RWMutex          // 全局读写锁 rwlock

func read(n int)  {
   rwlock.RLock()
   fmt.Printf("读 goroutine %d 正在读取数据...\n", n)
   num := count
   fmt.Printf("读 goroutine %d 读取数据结束，读到 %d\n", n, num)
   defer rwlock.RUnlock()
}
```

```
func write(n int)  {
   rwlock.Lock()
   fmt.Printf("写 goroutine %d 正在写数据...\n", n)
   num := rand.Intn(1000)
   count = num
   fmt.Printf("写 goroutine %d 写数据结束，写入新值 %d\n", n, num)
   defer rwlock.Unlock()
}

func main()  {
   for i:=0; i<5; i++ {
      go read(i+1)
   }
   for i:=0; i<5; i++ {
      go write(i+1)
   }
   for {
      ;
   }
}
```

程序的执行结果：

```
读 goroutine 2 正在读取数据...
读 goroutine 2 读取数据结束，读到 0
读 goroutine 3 正在读取数据...
读 goroutine 3 读取数据结束，读到 0
读 goroutine 1 正在读取数据...
读 goroutine 1 读取数据结束，读到 0
写 goroutine 1 正在写数据...
写 goroutine 1 写数据结束，写入新值 81
读 goroutine 4 正在读取数据...
读 goroutine 4 读取数据结束，读到 81
读 goroutine 5 正在读取数据...
读 goroutine 5 读取数据结束，读到 81
写 goroutine 3 正在写数据...
写 goroutine 3 写数据结束，写入新值 887
写 goroutine 2 正在写数据...
写 goroutine 2 写数据结束，写入新值 847
写 goroutine 4 正在写数据...
写 goroutine 4 写数据结束，写入新值 59
写 goroutine 5 正在写数据...
写 goroutine 5 写数据结束，写入新值 81
```

我们在read里使用读锁，也就是RLock和RUnlock，写锁的方法名和我们平时使用的一样，是Lock和Unlock。这样，我们就使用了读写锁，可以并发地读，但是同时只能有一个写，并且写的时候不能进行读操作。

我们从结果可以看出，读取操作可以并行，例如2,3,1正在读取，但是同时只能有一个写，例如1正在写，只能等待1写完，这个过程中不允许进行其它的操作。

处于读锁定状态，那么针对它的写锁定操作将永远不会成功，且相应的Goroutine也会被一直阻塞。因为它们是互斥的。

总结：读写锁控制下的多个写操作之间都是互斥的，并且写操作与读操作之间也都是互斥的。但是，多个读操作之间不存在互斥关系。

从互斥锁和读写锁的源码可以看出，它们是同源的。读写锁的内部用互斥锁来实现写锁定操作之间的互斥。可以把读写锁看作是互斥锁的一种扩展。

## 条件变量

在讲解条件变量之前，先回顾一下前面我们所涉及的“生产者消费者模型”：

```
package main

import "fmt"

//只写，不读。
func producer(out chan<- int)  {
   for i:= 0; i < 10; i++ {
      out <- i*i
   }
   close(out)
}
//只读，不写
func consumer(in <-chan int)  {
   for num := range in {
      fmt.Println("num = ", num)
   }
}
func main()  {
   ch := make(chan int)      // 创建一个双向channel
   go producer(ch)           // 生产者，产生数据，写入 channel
   consumer(ch)              // 消费者，从channel读数据，打印到屏幕
}
```

这个案例中，虽然实现了生产者消费者的功能，但有一个问题。如果有多个消费者来消费数据，并且并不是简单的从channel中取出来进行打印，而是还要进行一些复杂的运算。在consumer( )方法中的实现是否有问题呢？如下所示：

```
package main

import "fmt"
import "sync"
import "time"

var sum int

func producer(out chan<- int) {
```

```
for i := 0; i < =100; i++ {
    out <-i
}
close(out);
}

// 此chanel 只能读，不能写
func consumer(in <-chan int) {
for num := range in {
    sum +=num
}
fmt.println(“sum = ”, sum)
}

func main() {

   ch:= make(chan int)  // 创建一个双向通道
   go producer(ch)       // 协程1，生产者，生产数字，写入channel
   go consumer(ch)       // 协程2，消费者1
   consumer(ch)      // 主协程，消费者。从channel读取内容打印
   for {
       ;
     }
}
```

在上面的代码中，加了一个消费者，同时在consumer方法中，将数据取出来后，又进行了一组运算。这时可能会出现一个协程从管道中取出数据，参与加法运算，但是还没有算完另外一个协程又从管道中取出一个数据赋值给了num变量。所以这样累加计算，很有可能出现问题。当然，按照前面的知识，解决这个问题的方法很简单，就是通过加锁的方式来解决。增加生产者也是一样的道理。

另外一个问题，如果消费者比生产者多，仓库中就会出现没有数据的情况。我们需要不断的通过循环来判断仓库队列中是否有数据，这样会造成cpu的浪费。反之，如果生产者比较多，仓库很容易满，满了就不能继续添加数据，也需要循环判断仓库满这一事件，同样也会造成CPU的浪费。

我们希望当仓库满时，生产者停止生产，等待消费者消费；同理，如果仓库空了，我们希望消费者停下来等待生产者生产。为了达到这个目的，这里引入条件变量。（需要注意：如果仓库队列用channel，是不存在以上情况的，因为channel被填满后就阻塞了，或者channel中没有数据也会阻塞）。

**条件变量：**条件变量的作用并不保证在同一时刻仅有一个协程（线程）访问某个共享的数据资源，而是在对应的共享数据的状态发生变化时，通知阻塞在某个条件上的协程（线程）。条件变量不是锁，在并发中不能达到同步的目的，因此**条件变量总是与锁一块使用。**

例如，我们上面说的，如果仓库队列满了，我们可以使用条件变量让生产者对应的goroutine暂停（阻塞），但是当消费者消费了某个产品后，仓库就不再满了，应该唤醒（发送通知给）阻塞的生产者goroutine继续生产产品。

GO标准库中的sys.Cond类型代表了条件变量。条件变量要与锁（互斥锁，或者读写锁）一起使用。成员变量L代表与条件变量搭配使用的锁。

```
type Cond struct {
   noCopy noCopy
   // L is held while observing or changing the condition
   L Locker
   notify  notifyList
   checker copyChecker
}
```

对应的有3个常用方法，Wait，Signal，Broadcast。

1) **func** (c *Cond) Wait()

该函数的作用可归纳为如下三点：

a) 阻塞等待条件变量满足

b) 释放已掌握的互斥锁相当于cond.L.Unlock()。 注意：**两步为一个原子操作。**

c) 当被唤醒，Wait()函数返回时，解除阻塞并重新获取互斥锁。相当于cond.L.Lock()

2) **func** (c *Cond) Signal()

单发通知，给一个正等待（阻塞）在该条件变量上的goroutine（线程）发送通知。

3) **func** (c *Cond) Broadcast()

广播通知，给正在等待（阻塞）在该条件变量上的所有goroutine（线程）发送通知。

下面我们用条件变量来编写一个“生产者消费者模型”

示例代码：

```
package main
import "fmt"
import "sync"
import "math/rand"
import "time"

var cond sync.Cond             // 创建全局条件变量

// 生产者
func producer(out chan<- int, idx int) {
   for {
      cond.L.Lock()              // 条件变量对应互斥锁加锁
      for len(out) == 3 {            // 产品区满 等待消费者消费
         cond.Wait()                 // 挂起当前协程， 等待条件变量满足，被消费者唤醒
      }
      num := rand.Intn(1000)     // 产生一个随机数
      out <- num                 // 写入到 channel 中 （生产）
      fmt.Printf("%dth 生产者，产生数据 %3d，公共区剩余%d个数据\n", idx, num, len(out))
      cond.L.Unlock()                // 生产结束，解锁互斥锁
      cond.Signal()              // 唤醒 阻塞的 消费者
      time.Sleep(time.Second)        // 生产完休息一会，给其他协程执行机会
   }
}
```

```
//消费者
func consumer(in <-chan int, idx int) {
   for {
      cond.L.Lock()             // 条件变量对应互斥锁加锁（与生产者是同一个）
      for len(in) == 0 {        // 产品区为空 等待生产者生产
         cond.Wait()                // 挂起当前协程， 等待条件变量满足，被生产者唤醒
      }
      num := <-in                   // 将 channel 中的数据读走 （消费）
      fmt.Printf("---- %dth 消费者, 消费数据 %3d,公共区剩余%d个数据\n", idx, num, len(in))
      cond.L.Unlock()               // 消费结束，解锁互斥锁
      cond.Signal()             // 唤醒 阻塞的 生产者
      time.Sleep(time.Millisecond * 500)        //消费完 休息一会，给其他协程执行机会
   }
}
func main() {
   rand.Seed(time.Now().UnixNano())  // 设置随机数种子
   quit := make(chan bool)           // 创建用于结束通信的 channel

   product := make(chan int, 3)      // 产品区（公共区）使用channel 模拟
   cond.L = new(sync.Mutex)          // 创建互斥锁和条件变量

   for i := 0; i < 5; i++ {          // 5个消费者
      go producer(product, i+1)
   }
   for i := 0; i < 3; i++ {          // 3个生产者
      go consumer(product, i+1)
   }
   <-quit                           // 主协程阻塞 不结束
}
```

1) main函数中定义quit，其作用是让主协程阻塞。

2) 定义product作为队列，生产者产生数据保存至队列中，最多存储3个数据，消费者从中取出数据模拟消费

3) 条件变量要与锁一起使用，这里定义全局条件变量cond，它有一个属性：L Locker。是一个互斥锁。

4) 开启5个消费者协程，开启3个生产者协程。

5) producer生产者，在该方法中开启互斥锁，保证数据完整性。并且判断队列是否满，如果已满，调用wait()让该goroutine阻塞。当消费者取出数后执行cond.Signal()，会唤醒该goroutine，继续生产数据。

6) consumer消费者，同样开启互斥锁，保证数据完整性。判断队列是否为空，如果为空，调用wait()使得当前goroutine阻塞。当生产者产生数据并添加到队列，执行cond.Signal() 唤醒该goroutine。

# Go网络编程

## 网络概述

### 网络协议

从应用的角度出发，协议可理解为“**规则”**，是数据传输和数据的解释的规则。假设，A、B双方欲传输文件。规定：

l 第一次，传输文件名，接收方接收到文件名，应答OK给传输方；

l 第二次，发送文件的尺寸，接收方接收到该数据再次应答一个OK；

l 第三次，传输文件内容。同样，接收方接收数据完成后应答OK表示文件内容接收成功。

由此，无论A、B之间传递何种文件，都是通过三次数据传输来完成。A、B之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B之间达成的这个相互遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称之为原始协议。

当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个标准协议。最早的ftp协议就是由此衍生而来。

# 典型协议

传输层常见协议有TCP/UDP协议。

应用层常见的协议有HTTP协议，FTP协议。

网络层常见协议有IP协议、ICMP协议、IGMP协议。

网络接口层常见协议有ARP协议、RARP协议。

TCP传输控制协议（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

UDP用户数据报协议（User Datagram Protocol）是OSI参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

HTTP超文本传输协议（Hyper Text Transfer Protocol）是互联网上应用最为广泛的一种网络协议。

FTP文件传输协议（File Transfer Protocol）

IP协议是因特网互联协议（Internet Protocol）

ICMP协议是Internet控制报文协议（Internet Control Message Protocol）它是TCP/IP协议族的一个子协议，用于在IP主机、路由器之间传递控制消息。

IGMP协议是 Internet 组管理协议（Internet Group ManagementProtocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

ARP协议是正向地址解析协议（Address Resolution Protocol），通过已知的IP，寻找对应主机的MAC地址。

RARP是反向地址转换协议，通过MAC地址确定IP地址。

# 分层模型

## 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。

| OSI/RM(理论上的标准) | TCP/IP(事实上的标准) |
| --- | --- |
| 应用层 | 应用层 |
| 表示层 | |
| 会话层 | |
| 传输层 | 传输层 |
| 网络层 | 网络层 |
| 数据链路层 | 链路层 |
| 物理层 | |

越下面的层，越靠近硬件；越上面的层，越靠近用户。至于每一层叫什么名字，对应编程而言不重要，但面试的时候，面试官可能会问每一层的名字。

业内普遍的分层方式有两种。OSI七层模型 和TCP/IP四层模型。可以通过背诵两个口诀来快速记忆：

OSI七层模型：物、数、网、传、会、表、应

TCP/IP四层模型：链、网、传、应

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）。

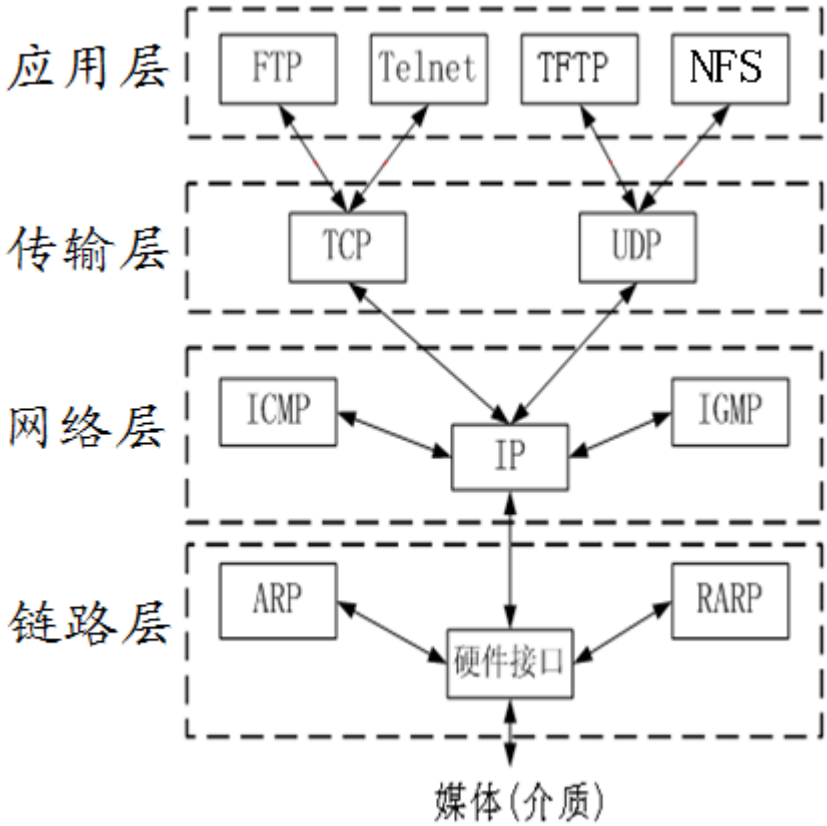
6) **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。

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。

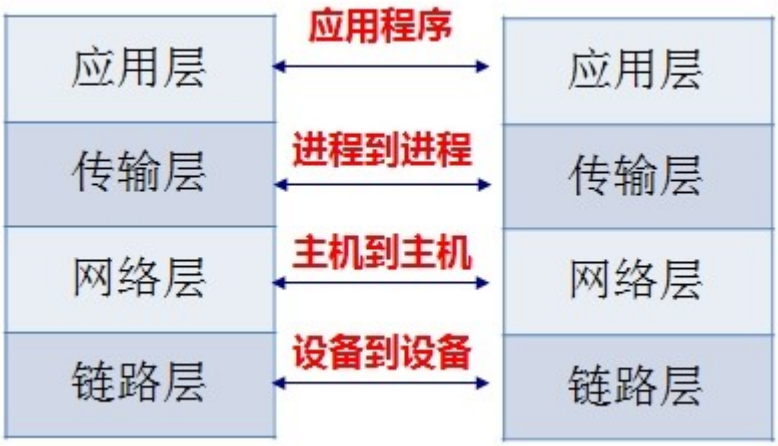
## 层与协议

每一层都是为了完成一种功能，为了实现这些功能，就需要大家都遵守共同的规则。大家都遵守这规则，就叫做“协议”（protocol）。

网络的每一层，都定义了很多协议。这些协议的总称，叫“TCP/IP协议”。TCP/IP协议是一个大家族，不仅仅只有TCP和IP协议，它还包括其它的协议，如下图：




## 协议功能




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。

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。

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

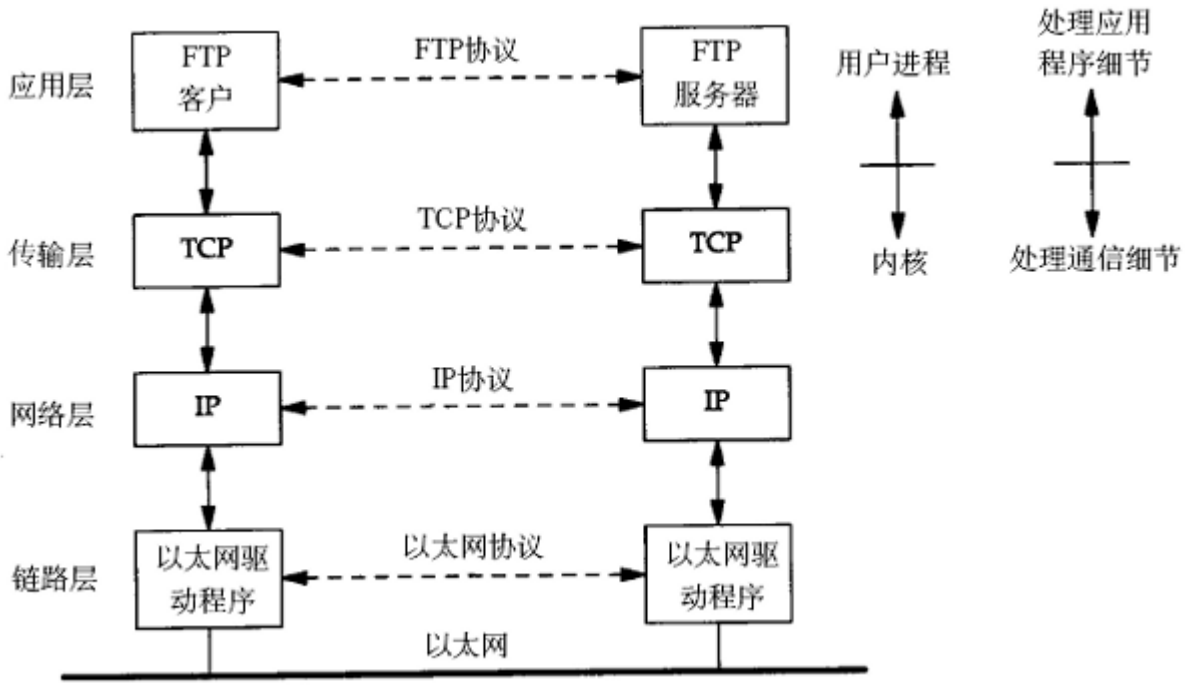
l 对于同一个系统，一个端口只能被一个进程拥有

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。

## 通信过程

两台计算机通过TCP/IP协议通讯的过程如下所示：



# Socket编程

## 什么是Socket

Socket，英文含义是【插座、插孔】，一般称之为套接字，用于描述IP地址和端口。可以实现不同程序间的数据通信。

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。

套接字的内核实现较为复杂，不宜在学习初期深入学习，了解到如下结构足矣。


在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程。“IP地址+端口号”就对应一个socket。欲建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。因此可以用Socket来描述网络连接的一对一关系。

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。

## 网络应用程序设计模式

### C/S模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

### B/S模式

浏览器(Browser)/服务器(Server)模式。只需在一端部署服务器，而另外一端使用每台PC都默认配置的浏览器即可完成数据的传输。

优缺点

对于C/S模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务器程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯所采用的通信协议，即为ftp协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选C/S模式进行开发。如，知名的网络游戏魔兽世界。3D画面，数据量庞大，使用C/S模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

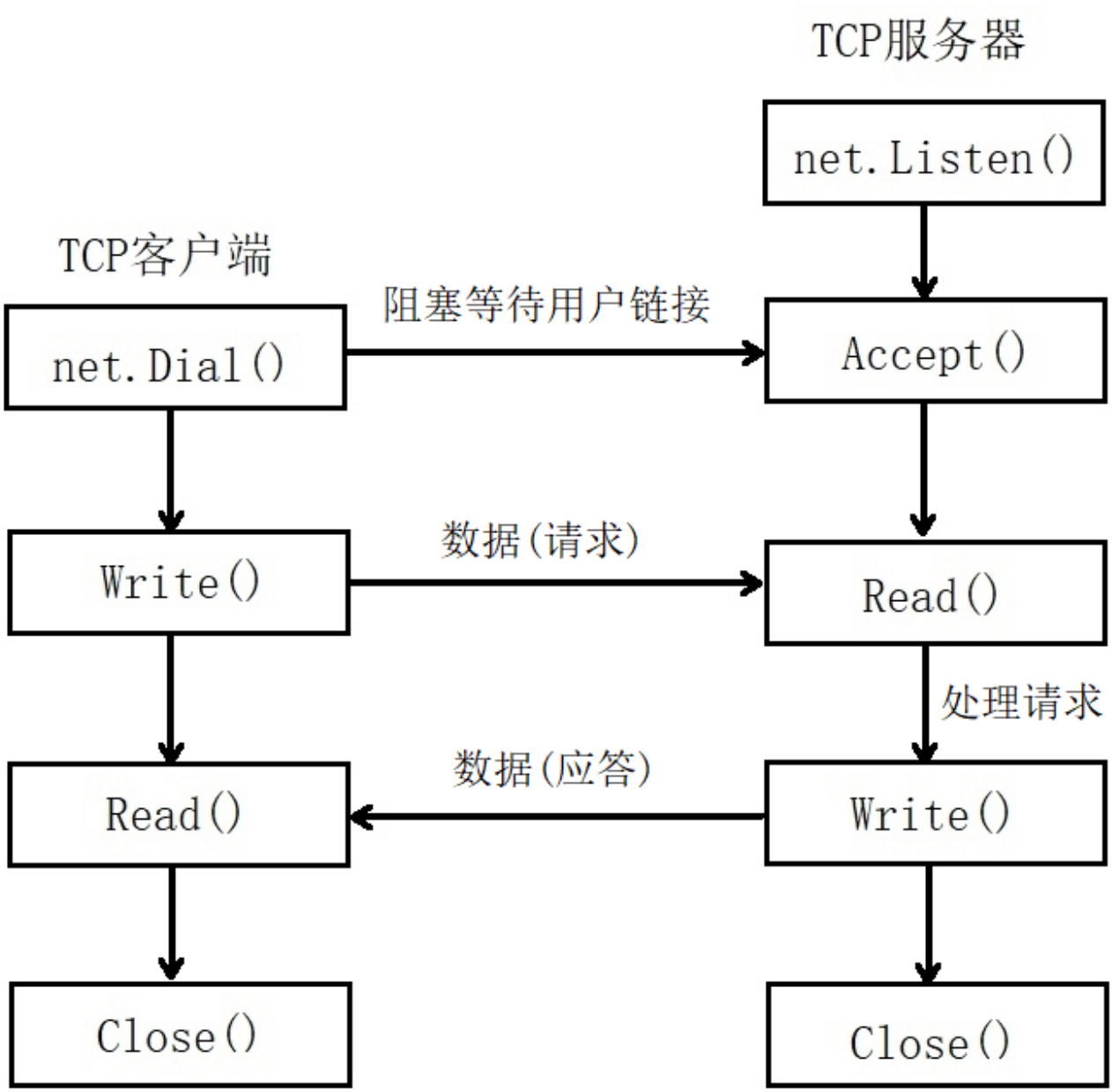
C/S模式的缺点也较突出。由于客户端和服务器都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用C/S模式应用程序的重要原因。

B/S模式相比C/S模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其工作**开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准http协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

## TCP的C/S架构




## 简单的C/S模型通信

### Server端

Listen函数:

```
func Listen(network, address string) (Listener, error)
        network: 选用的协议: TCP、UDP,   如: “tcp”或 “udp”
        address: IP地址+端口号,   如: “127.0.0.1:8000”或 “:8000”
```

Listener接口:

```
type Listener interface {
                        Accept() (Conn, error)
                        Close() error
                        Addr() Addr
}
```

Conn 接口：

```
type Conn interface {
     Read(b []byte) (n int, err error)
     Write(b []byte) (n int, err error)
     Close() error
     LocalAddr() Addr
     RemoteAddr() Addr
     SetDeadline(t time.Time) error
     SetReadDeadline(t time.Time) error
     SetWriteDeadline(t time.Time) error
}
```

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

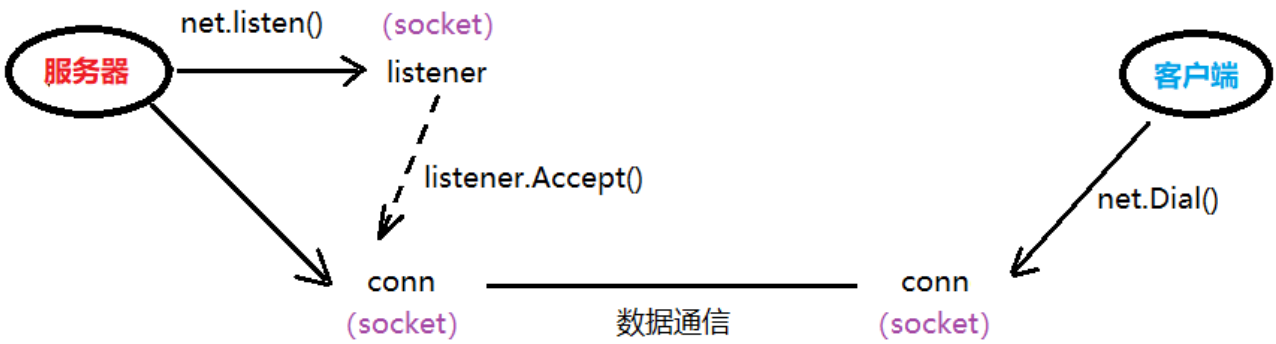
示例代码：

TCP服务器.go

```
package main

import (
   "net"
   "fmt"
)

func main()  {
   // 创建监听
   listener, err:= net.Listen("tcp", ":8000")
   if err != nil {
      fmt.Println("listen err:", err)
      return
   }
   defer listener.Close()          // 主协程结束时，关闭listener

   fmt.Println("服务器等待客户端建立连接...")
   // 等待客户端连接请求
   conn, err := listener.Accept()
   if err != nil {
      fmt.Println("accept err:", err)
      return
   }
   defer conn.Close()             // 使用结束，断开与客户端链接

   fmt.Println("客户端与服务器连接建立成功...")
```

```
    // 接收客户端数据
    buf := make([]byte, 1024)        // 创建1024大小的缓冲区，用于read
    n, err := conn.Read(buf)
    if err != nil {
       fmt.Println("read err:", err)
       return
    }
    fmt.Println("服务器读到:", string(buf[:n])) // 读多少，打印多少。
}
```

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




## Client 端

Dial函数：

```
func Dial(network, address string) (Conn, error)
                network：选用的协议：TCP、UDP，如：“tcp”或 “udp”
                address：服务器IP地址+端口号，如：“121.36.108.11:8000”或 “www.itcast.cn:8000”
```

Conn 接口：

```
type Conn interface {
        Read(b []byte) (n int, err error)
        Write(b []byte) (n int, err error)
        Close() error
        LocalAddr() Addr
        RemoteAddr() Addr
        SetDeadline(t time.Time) error
        SetReadDeadline(t time.Time) error
        SetWriteDeadline(t time.Time) error
}
```

```
package main

import (
   "net"
   "fmt"
)

func main() {
   // 主动发起连接请求
   conn, err := net.Dial("tcp", "127.0.0.1:8000")
   if err != nil {
      fmt.Println("Dial err:", err)
      return
   }
   defer conn.Close()         // 结束时，关闭连接

   // 发送数据
   _, err = conn.Write([]byte("Are u ready?"))
   if err != nil {
      fmt.Println("Write err:", err)
      return
   }
}
```

# 并发的C/S模型通信

## 并发Server

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

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

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

```
    //监听
    listener, err := net.Listen("tcp", "127.0.0.1:8001")
    if err != nil {
            fmt.Println("err = ", err)
            return
    }
    defer listener.Close()
    //接收多个用户
    for {
        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("err = ", err)

            return
```

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

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

Conn 接口：

```
type Conn interface {
        Read(b []byte) (n int, err error)
        Write(b []byte) (n int, err error)
        Close() error
        LocalAddr() Addr
        RemoteAddr() Addr
        SetDeadline(t time.Time) error
        SetReadDeadline(t time.Time) error
        SetWriteDeadline(t time.Time) error
}
```

```
    //获取客户端的网络地址信息
    addr := conn.RemoteAddr().String()
    fmt.Println(addr, " conncet sucessful")
```

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

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

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

for {
        //读取用户数据
        n, err := conn.Read(buf)
        if err != nil {
            fmt.Println("err = ", err)
            return
        }
        fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))
        if  "exit" == string(buf[:n-2]) {        //自己写的客户端测试，发送时，多了2个字符，"\r\n"
            fmt.Println(addr, " exit")
            return
        }
        //服务器处理数据：把客户端数据转大写，再写回给client
        conn.Write([]byte(strings.ToUpper(string(buf[:n]))))
 }
```

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

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

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

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

综上，HandleConn方法完整定义如下：

```
//处理用户请求
func HandleConn(conn net.Conn) {
          //函数调用完毕，自动关闭conn
          defer conn.Close()

          //获取客户端的网络地址信息
          addr := conn.RemoteAddr().String()
          fmt.Println(addr, " conncet sucessful")

          buf := make([]byte, 2048)

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

                //if "exit" == string(buf[:n-1]) {// nc测试，发送时，只有 \n
                if  "exit" == string(buf[:n-2]) {// 自己写的客户端测试，发送时，多了2个字符，"\r\n"
                    fmt.Println(addr, " exit")
                    return
                }

                //把数据转换为大写，再给用户发送
                conn.Write([]byte(strings.ToUpper(string(buf[:n]))))
          }
}
```

## 并发Client

客户端不仅需要持续的向服务端发送数据，同时也要接收从服务端返回的数据。因此可将发送和接收放到不同的协程中。

主协程循环接收服务器回发的数据（该数据应已转换为大写），并打印至屏幕；子协程循环从键盘读取用户输入数据，写给服务器。读取键盘输入可使用os.Stdin.Read(str)。定义切片str，将读到的数据保存至str中。

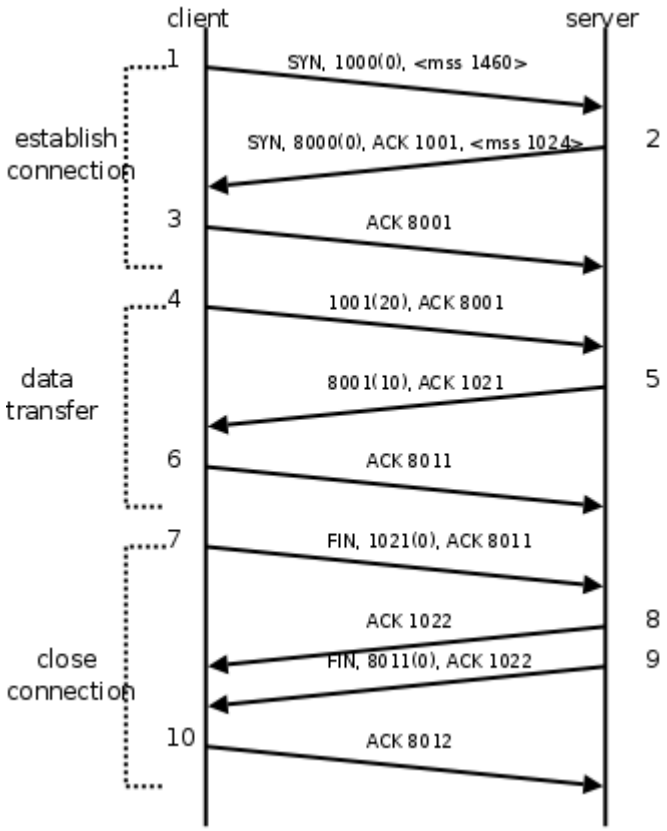
这样，客户端也实现了多任务。

客户端代码实现：

```
package main

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

func main() {
   // 主动发起连接请求
   conn, err := net.Dial("tcp", "127.0.0.1:8001")
   if err != nil {
      fmt.Println("Dial err:", err)
      return
   }
   defer conn.Close()                    // 客户端终止时，关闭与服务器通信的 socket

   // 启动子协程，接收用户键盘输入
   go func() {
      str := make([]byte, 1024)          // 创建用于存储用户键盘输入数据的切片缓冲区。
      for {                              // 反复读取
         n, err :=os.Stdin.Read(str)     // 获取用户键盘输入
         if err != nil {
            fmt.Println("os.Stdin.Read err:", err)
            return
         }
         // 将从键盘读到的数据，发送给服务器
         _, err = conn.Write(str[:n])       // 读多少，写多少
         if err != nil {
            fmt.Println("conn.Write err:", err)
            return
         }
      }
   }()

   // 主协程，接收服务器回发数据，打印至屏幕
   buf := make([]byte, 1024)                // 定义用于存储服务器回发数据的切片缓冲区
   for {
      n, err := conn.Read(buf)           // 从通信 socket 中读数据，存入切片缓冲区
      if err != nil {
         fmt.Println("conn.Read err:", err)
         return
      }
      fmt.Printf("服务器回发: %s\n", string(buf[:n]))
   }
}
```

## TCP通信

下图是一次TCP通讯的时序图。TCP连接建立断开。包含大家熟知的**三次握手**和四次握手。




在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序。注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。

## 三次握手

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。好比两个人在打电话：

Client:“喂，你听得到吗？”

Server:“我听得到，你听得到我吗？”

Client:“我能听到你，今天balabala...”

建立连接（三次握手）的过程：

1. 客户端发送一个带SYN标志的TCP报文到服务器。这是上图中三次握手过程中的段1。客户端发出SYN位表示连接请求。序号是1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况。

另外，规定SYN位和FIN位也要占一个序号，这次虽然没发数据，但是由于发了SYN位，因此下次再发送应该用序号1001。

mss表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大长度，就必须在IP层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

2. 服务器端回应客户端，是三次握手中的第2个报文段，同时带ACK标志和SYN标志。表示对刚才客户端SYN的回应；同时又发送SYN给客户端，询问客户端是否准备好进行数据通讯。

服务器发出段2，也带有SYN位，同时置ACK位表示确认，确认序号是1001，表示“我接收到序号1000及其以前所有的段，请你下次发送序号为1001的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为1024。

3. 客户必须再次回应服务器端一个ACK报文，这是报文段3。

客户端发出段3，对服务器的连接请求进行应答，确认序号是8001。在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出。

因此一共有三个段用于建立连接，称为“三方握手”。在建立连接的同时，双方协商了一些信息，例如，双方发送序号的初始值、最大段尺寸等。

数据传输的过程：

1. 客户端发出段4，包含从序号1001开始的20个字节数据。
2. 服务器发出段5，确认序号为1021，对序号为1001-1020的数据表示确认收到，同时请求发送序号1021开始的数据，服务器在应答的同时也向客户端发送从序号8001开始的10个字节数据。
3. 客户端发出段6，对服务器发来的序号为8001-8010的数据表示确认收到，请求发送序号8011开始的数据。

在数据传输过程中，ACK和确认序号是非常重要的，应用程序交给TCP协议发送的数据会暂存在TCP层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的ACK段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的ACK段，经过等待超时后TCP协议自动将发送缓冲区中的数据包重发。

四次挥手：

所谓四次挥手（Four-Way-Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务器任一方执行close来触发。好比两个人打完电话要挂断：

Client:“我要说的事情都说完了，我没事了。挂啦？”

Server:“等下，我还有一个事儿。Balabala...”

Server:“好了，我没事儿了。挂了啊。”

Client:“ok！拜拜”

关闭连接（四次握手）的过程：

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

1. 客户端发出段7，FIN位表示关闭连接的请求。
2. 服务器发出段8，应答客户端的关闭连接请求。
3. 服务器发出段9，其中也包含FIN位，向客户端发送关闭连接请求。
4. 客户端发出段10，应答服务器的关闭连接请求。

建立连接的过程是三次握手，而关闭连接通常需要4个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

## UDP通信

在之前的案例中，我们一直使用的是TCP协议来编写Socket的客户端与服务端。其实也可以使用UDP协议来编写Socket的客户端与服务端。

## UDP服务器

由于UDP是“无连接”的，所以，服务器端不需要额外创建监听套接字，只需要指定好IP和port，然后监听该地址，等待客户端与之建立连接，即可通信。

创建监听地址：

```
func ResolveUDPAddr(network, address string) (*UDPAddr, error)
```

创建监听连接：

```
func ListenUDP(network string, laddr *UDPAddr) (*UDPConn, error)
```

接收udp数据：

```
func (c *UDPConn) ReadFromUDP(b []byte) (int, *UDPAddr, error)
```

写出数据到udp：

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

服务端完整代码实现如下：

```
package main

import (
   "fmt"
   "net"
)

func main() {
   //创建监听的地址，并且指定udp协议
   udp_addr, err := net.ResolveUDPAddr("udp", "127.0.0.1:8002")
   if err != nil {
      fmt.Println("ResolveUDPAddr err:", err)
      return
   }
   conn, err := net.ListenUDP("udp", udp_addr)    //创建监听链接
   if err != nil {
      fmt.Println("ListenUDP err:", err)
      return
   }
   defer conn.Close()

   buf := make([]byte, 1024)
   n, raddr, err := conn.ReadFromUDP(buf)        //接收客户端发送过来的数据，填充到切片buf中。
   if err != nil {
      return
```

```
    }
    fmt.Println("客户端发送: ", string(buf[:n]))

    _, err = conn.WriteToUDP([]byte("nice to see u in udp"), raddr) //向客户端发送数据
    if err != nil {
        fmt.Println("WriteToUDP err:", err)
        return
    }
}
```

## UDP客户端

udp客户端的编写与TCP客户端的编写，基本上是一样的，只是将协议换成udp.代码如下：

```
package main

import (
    "net"
    "fmt"
)

func main() {
    conn, err := net.Dial("udp", "127.0.0.1:8002")
    if err != nil {
        fmt.Println("net.Dial err:", err)
        return
    }
    defer conn.Close()

    conn.Write([]byte("Hello! I'm client in UDP!"))

    buf := make([]byte, 1024)
    n, err1 := conn.Read(buf)
    if err1 != nil {
        return
    }
    fmt.Println("服务器发来: ", string(buf[:n]))
}
```

## 并发

其实对于UDP而言，服务器不需要并发，只要循环处理客户端数据即可。客户端也等同于TCP通信并发的客户端。

**服务器**

```
package main

import (
    "net"
    "fmt"
)
```

```
func main() {
    // 创建 服务器 UDP 地址结构。指定 IP + port
    laddr, err := net.ResolveUDPAddr("udp", "127.0.0.1:8003")
    if err != nil {
        fmt.Println("ResolveUDPAddr err:", err)
        return
    }
    // 监听 客户端连接
    conn, err := net.ListenUDP("udp", laddr)
    if err != nil {
        fmt.Println("net.ListenUDP err:", err)
        return
    }
    defer conn.Close()

    for {
        buf := make([]byte, 1024)
        n, raddr, err := conn.ReadFromUDP(buf)
        if err != nil {
            fmt.Println("conn.ReadFromUDP err:", err)
            return
        }
        fmt.Printf("接收到客户端[%s]: %s", raddr, string(buf[:n]))

        conn.WriteToUDP([]byte("I-AM-SERVER"), raddr) // 简单回写数据给客户端
    }
}
```

**客户端**

```
package main

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

func main() {
    conn, err := net.Dial("udp", "127.0.0.1:8003")
    if err != nil {
        fmt.Println("net.Dial err:", err)
        return
    }
    defer conn.Close()
    go func() {
        str := make([]byte, 1024)
        for {
            n, err := os.Stdin.Read(str) //从键盘读取内容， 放在str
            if err != nil {
                fmt.Println("os.Stdin. err1 = ", err)
```

```
            return
         }
         conn.Write(str[:n])       // 给服务器发送
      }
   }()
   buf := make([]byte, 1024)
   for {
      n, err := conn.Read(buf)
      if err != nil {
         fmt.Println("conn.Read err:", err)
         return
      }
      fmt.Println("服务器写来: ", string(buf[:n]))
   }
}
```

## UDP与TCP的差异

| TCP | UDP |
| --- | --- |
| 面向连接 | 面向无连接 |
| 要求系统资源较多 | 要求系统资源较少 |
| TCP程序结构较复杂 | UDP程序结构较简单 |
| 使用流式 | 使用数据包式 |
| 保证数据准确性 | 不保证数据准确性 |
| 保证数据顺序 | 不保证数据顺序 |
| 通讯速度较慢 | 通讯速度较快 |

## 文件传输

流程简析

借助TCP完成文件的传输，基本思路如下：

1：发送方（客户端）向服务端发送文件名，服务端保存该文件名。

2：接收方（服务端）向客户端返回一个消息ok，确认文件名保存成功。

3：发送方（客户端）收到消息后，开始向服务端发送文件数据。

4：接收方（服务端）读取文件内容，写入到之前保存好的文件中。


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

```
func Stat(name string) (FileInfo, error)
type FileInfo interface {
   Name() string
   Size() int64
   Mode() FileMode
   ModTime() time.Time
   IsDir() bool
   Sys() interface{}
}
```

获取文件属性示例：

```
package main

import (
   "os"
   "fmt"
)

func main()  {
   list := os.Args                          // 获取命令行参数，存入list中
   if len(list) != 2 {                     // 确保用户输入了一个命令行参数
      fmt.Println("格式为：xxx.go 文件名")
      return
   }
   fileName := list[1]                     // 从命令行保存文件名(含路径)
```

```
    fileInfo, err := os.Stat(fileName)    //根据文件名获取文件属性信息 fileInfo
    if err != nil {
        fmt.Println("os.Stat err:", err)
        return
    }
    fmt.Println("文件name为: ", fileInfo.Name())   // 得到文件名(不含路径)
    fmt.Println("文件size为: ", fileInfo.Size())   // 得到文件大小。单位字节
}
```

## 客户端实现

实现流程大致如下:

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

代码实现:

```
package main

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

func SendFile(path string, conn net.Conn)  {
    // 以只读方式打开文件
    f, err := os.Open(path)
    if err != nil {
        fmt.Println("os.Open err:", err)
        return
    }
    defer f.Close()                    // 发送结束关闭文件。

    // 循环读取文件，原封不动的写给服务器
    buf := make([]byte, 4096)
    for {
        n, err := f.Read(buf)         // 读取文件内容到切片缓冲中
        if err != nil {
            if err == io.EOF {
                fmt.Println("文件发送完毕")
            } else {
                fmt.Println("f.Read err:", err)
```

```
            }
            return
        }
        conn.Write(buf[:n])  // 原封不动写给服务器
    }
}

func main()  {
    // 提示输入文件名
    fmt.Println("请输入需要传输的文件：")
    var path string
    fmt.Scan(&path)

    // 获取文件名   fileInfo.Name()
    fileInfo, err := os.Stat(path)
    if err != nil {
        fmt.Println("os.Stat err:", err)
        return
    }

    // 主动连接服务器
    conn, err := net.Dial("tcp", "127.0.0.1:8005")
    if err != nil {
        fmt.Println("net.Dial err:", err)
        return
    }
    defer conn.Close()

    // 给接收端，先发送文件名
    _, err = conn.Write([]byte(fileInfo.Name()))
    if err != nil {
        fmt.Println("conn.Write err:", err)
        return
    }

    // 读取接收端回发确认数据 —— ok
    buf := make([]byte, 1024)
    n, err := conn.Read(buf)
    if err != nil {
        fmt.Println("conn.Read err:", err)
        return
    }

    // 判断如果是ok，则发送文件内容
    if "ok" == string(buf[:n]) {
        SendFile(path, conn)   // 封装函数读文件，发送给服务器，需要path、conn
    }
}
```

## 服务端实现

实现流程大致如下：

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

代码实现：

```
package main

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

func RecvFile(fileName string, conn net.Conn)  {
   // 创建新文件
   f, err := os.Create(fileName)
   if err != nil {
      fmt.Println("Create err:", err)
      return
   }
   defer f.Close()

   // 接收客户端发送文件内容，原封不动写入文件
   buf := make([]byte, 4096)
   for {
      n, err := conn.Read(buf)
      if err != nil {
         if err == io.EOF {
            fmt.Println("文件接收完毕")
         } else {
            fmt.Println("Read err:", err)
         }
         return
      }
      f.Write(buf[:n])   // 写入文件，读多少写多少
   }
}

func main()  {
   // 创建监听
   listener, err := net.Listen("tcp", "127.0.0.1:8005")
   if err != nil {
      fmt.Println("Listen err:", err)
      return
   }
   defer listener.Close()
```

```
    // 阻塞等待客户端连接
    conn, err := listener.Accept()
    if err != nil {
        fmt.Println("Accept err:", err)
        return
    }
    defer conn.Close()

    // 读取客户端发送的文件名
    buf := make([]byte, 1024)
    n, err := conn.Read(buf)
    if err != nil {
        fmt.Println("Read err:", err)
        return
    }
    fileName := string(buf[:n])       // 保存文件名

    // 回复 0k 给发送端
    conn.Write([]byte("ok"))

    // 接收文件内容
    RecvFile(fileName, conn)      // 封装函数接收文件内容， 传fileName 和 conn
}
```

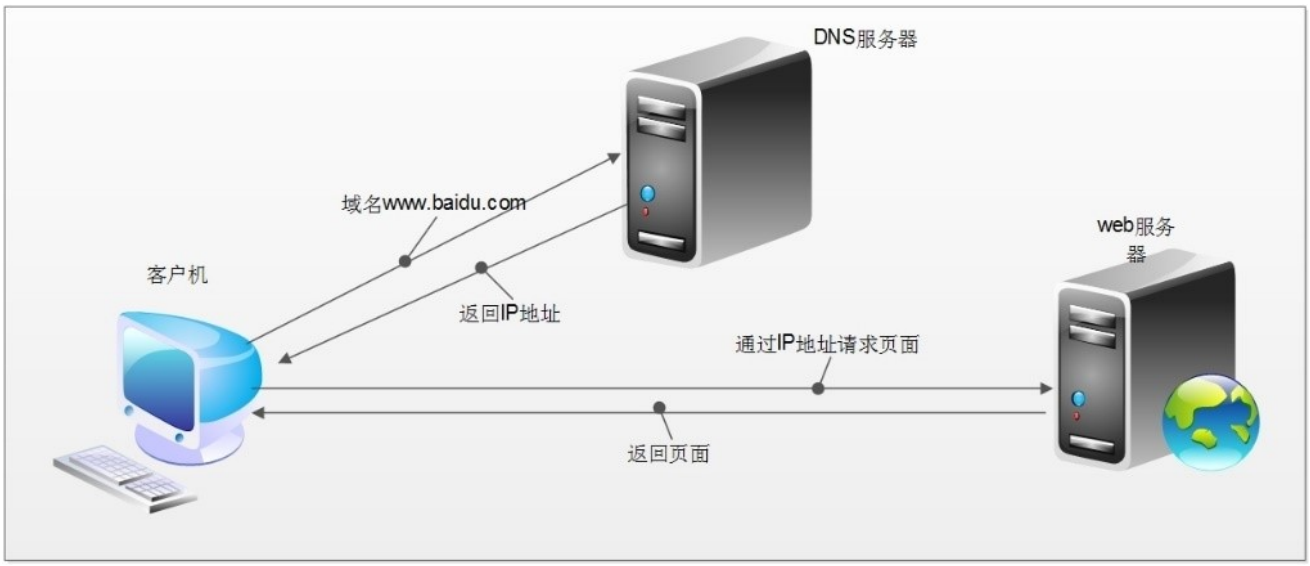
# HTTP编程

## 概述

## Web工作方式

我们平时浏览网页的时候,会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后，到底隐藏了些什么呢?

对于普通的上网过程，系统其实是这样做的：浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，然后通过IP地址找到IP对应的服务器后，要求建立TCP连接，等浏览器发送完HTTP Request（请求）包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，返回HTTP Response（响应）包；客户端收到来自服务器的响应后开始渲染这个Response包里的主体（body），等收到全部的内容随后断开与该服务器之间的TCP连接。




DNS域名服务器（Domain Name Server）是进行域名(domain name)和与之相对应的IP地址转换的服务器。DNS中保存了一张域名解析表，解析消息的域名。

一个**Web**服务器也被称为**HTTP**服务器，它通过HTTP (HyperTextTransfer Protocol 超文本传输协议)协议与客户端通信。这个客户端通常指的是Web浏览器(其实手机端客户端内部也是浏览器实现的)。

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

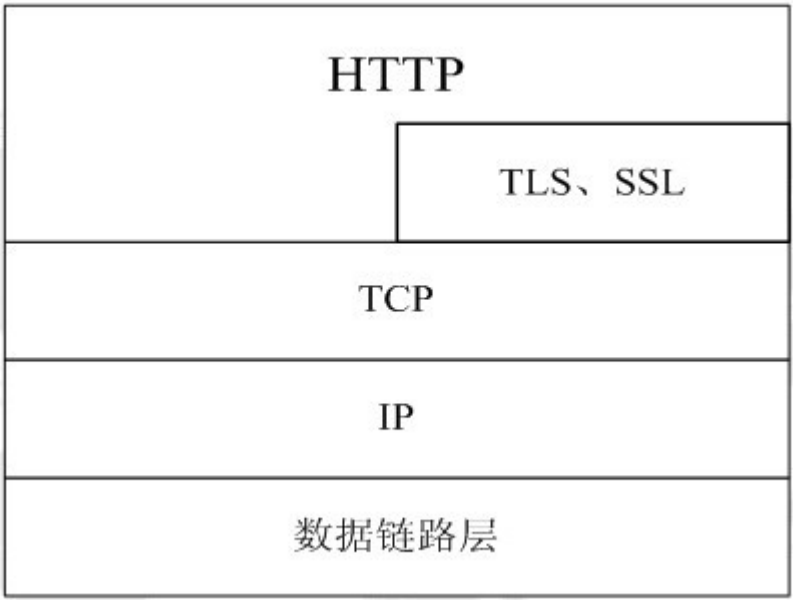
l 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端

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

## 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：




### 地址（URL）

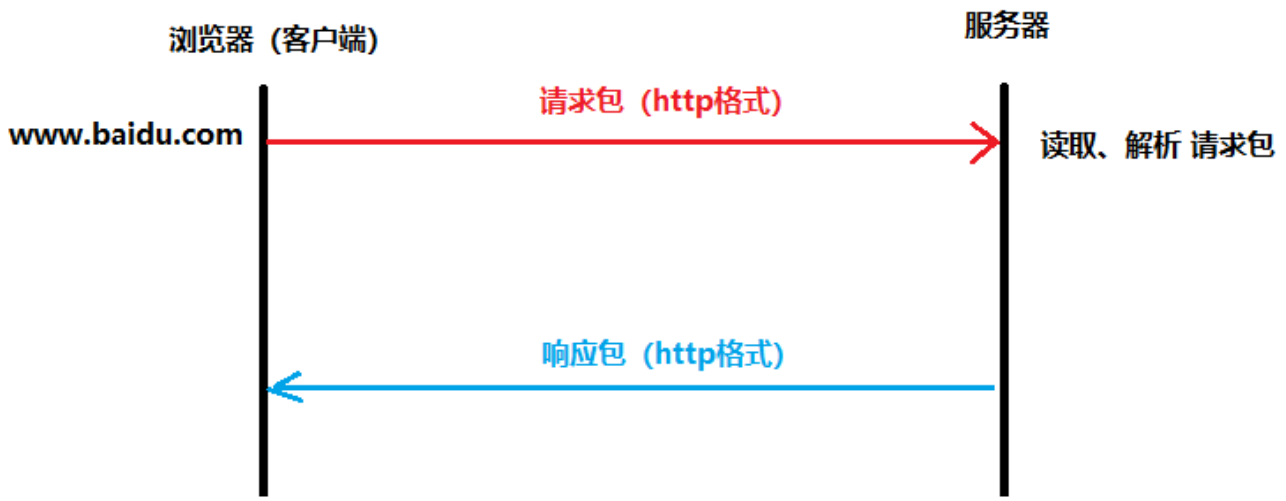
URL全称为Unique Resource Location，用来表示网络资源，可以理解为网络文件路径。

基本URL的结构包含模式（协议）、服务器名称（IP地址）、路径和文件名。常见的协议/模式如http、https、ftp等。服务器的名称或IP地址后面有时还跟一个冒号和一个端口号。再后面是到达这个文件的路径和文件本身的名称。如：

```
http://localhost[":"port][abs_path]
http://192.168.31.1/html/index
https://pan.baidu.com/
```

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。

## HTTP报文解析




### 请求报文格式

获取请求报文

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

服务器测试代码：

```
package main

import (
   "net"
   "fmt"
)

func main() {
   //创建、监听socket
   listenner, err := net.Listen("tcp", "127.0.0.1:8000")
   if err != nil {
      fmt.Println("Listen err:", err)
      return
   }
   defer listenner.Close()

   //阻塞等待客户端连接
   conn, err := listenner.Accept()
   if err != nil {
      fmt.Println("Accept err:", err)
      return
   }
   defer conn.Close()

   fmt.Println(conn.RemoteAddr().String(), "连接成功")       //连接客户端的网络地址

   buf := make([]byte, 4096)  //切片缓冲区，接收客户端发送数据
   n, err := conn.Read(buf)   //n 接收数据的长度
   if err != nil {
      fmt.Println("Read err:", err)
      return
   }
   result := buf[:n]        //切片截取

   fmt.Printf("#\n%s#", string(result))
}
```

在浏览器中输入url地址： 127.0.0.1:8000

服务器端运行打印结果如下：

```
127.0.0.1:12389 连接成功
#
GET / HTTP/1.1   请求行
Host: 127.0.0.1:8000
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1   请求头
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.99 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
                                   空行
#
```

请求报文格式说明

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


**1) 请求行**

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

GET:

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 |

**3)　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。

**4)　请求包体**

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

响应报文格式

要想获取响应报文，必须先发送请求报文给web服务器。服务器收到并解析浏览器（客户端）发送的请求报文后，借助http协议，回复相对应的响应报文。

下面我们借助net/http包，创建一个最简单的服务器，给浏览器回发送响应包。首先注册处理函数http.HandleFunc()，设置回调函数handler。而后绑定服务器的监听地址http.ListenAndserve()。

这个服务器启动后，当有浏览器发送请求，回调函数被调用，会向浏览器回复“hello world”作为网页内容。当然，是按照http协议的格式进行回复。

创建简单的响应服务器

## 服务器示例代码

```
package main

import "net/http"


// 浏览器访问时，该函数被回调
```

```
func handler(w http.ResponseWriter, r *http.Request) {
   w.Write([]byte("hello http"))
}

func main()  {
   // 注册处理函数
   http.HandleFunc("/hello", handler)

   // 绑定服务器监听地址
   http.ListenAndServe("127.0.0.1:8000", nil)
}
```

测试：启动服务器。打开浏览器，在URL中写入127.0.0.1:8000/hello，向服务器发送请求。会在浏览器中看到服务器回发的“hello http”。

回调函数handler会在浏览器访问服务器时被调用。服务器使用w.Write向浏览器写了“hello http”。

但服务器是怎样借助http协议将“hello http”字符串写回来的呢，服务器响应报文的具体格式是什么样的呢?

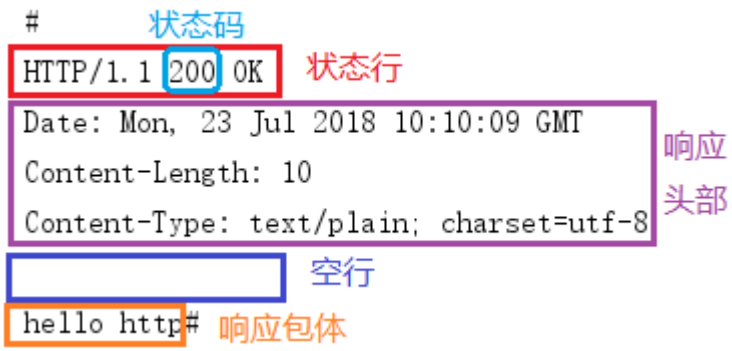
接下来我们编写一个客户端，模拟浏览器给服务器发送“请求报文”的行为，然后将服务器回发的响应报文打印出来就可以看到了。

## 客户端测试示例代码

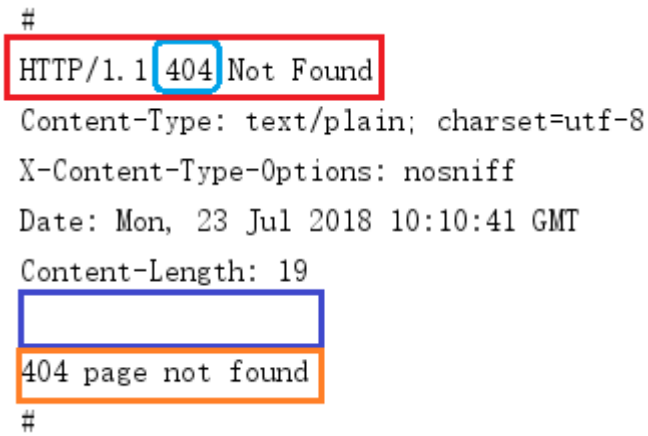
```
package main

import (
   "net"
   "fmt"
)

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

   // 模拟浏览器，组织一个最简单的请求报文。包含请求行，请求头，空行即可。
   requestHttpHeader := "GET /hello HTTP/1.1\r\nHost:127.0.0.1:8000\r\n\r\n"

   // 给服务器发送请求报文
   conn.Write([]byte(requestHttpHeader))

   buf := make([]byte, 4096)
   // 读取服务器回复 响应报文
   n, err := conn.Read(buf)
   if err != nil {
      fmt.Println("Read err:", err)
      return
   }
```

```
    // 打印观察
    fmt.Printf("#\n%s#", string(buf[:n]))
}
```

启动程序，测试http的成功响应报文：

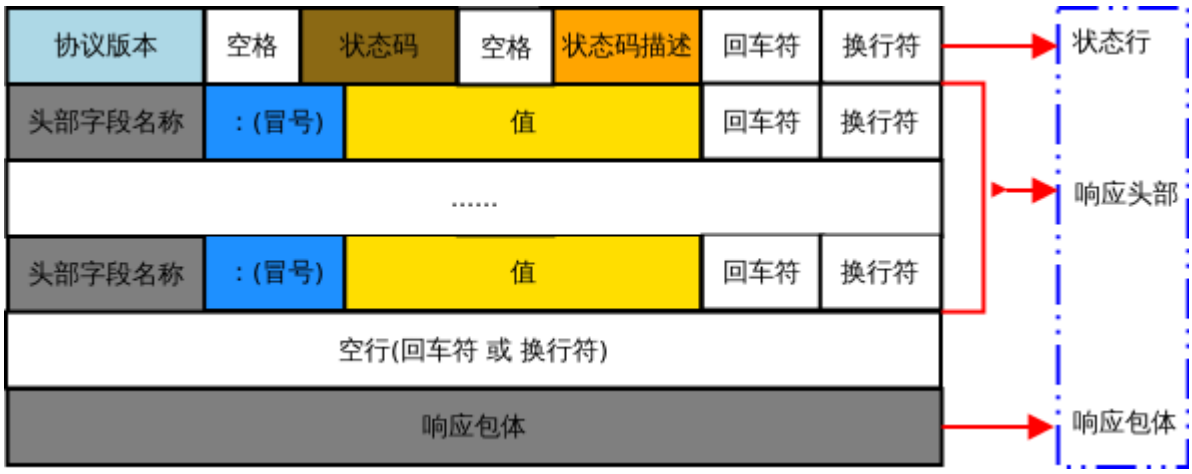



启动程序，测试http的失败响应报文：




响应报文格式说明

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




1) 状态行

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

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

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

常见的状态码举例：

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

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。

# Go语言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。

## HTTP服务端

示例代码：

```
package main

import (
    "fmt"
    "net/http"
)

//服务端编写的业务逻辑处理程序 —— 回调函数
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("method = ", r.Method)  //请求方法
    fmt.Println("URL = ", r.URL)                // 浏览器发送请求文件路径
    fmt.Println("header = ", r.Header)          // 请求头
    fmt.Println("body = ", r.Body)              // 请求包体
    fmt.Println(r.RemoteAddr, "连接成功")        //客户端网络地址

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

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

    //该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理传入的连接请求。
    //该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为nil
    //当参2为nil时，服务端调用 http.DefaultServeMux 进行处理
    http.ListenAndServe("127.0.0.1:8000", nil)
}
```

浏览器输入url地址：127.0.0.1:8000/hello

回调函数myHandler的函数原型固定。func myHandler(w http.ResponseWriter, r *http.Request) 有两个参数：

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

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

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

```
type Request struct {
    Method string        // 浏览器请求方法 GET、POST…
    URL *url.URL         // 浏览器请求的访问路径
    ……
    Header Header        // 请求头部
    Body io.ReadCloser   // 请求包体
    RemoteAddr string    // 浏览器地址
    ……
    ctx context.Context
}
```

查看一下结构体成员：

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

查看到如下内容：

```
Method =  GET
URL =  /hello
Header =  map[Upgrade-Insecure-Requests:[1] User-Agent:[Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
 Chrome/63.0.3239.132 Safari/537.36] Accept:[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8]
 Accept-Encoding:[gzip, deflate, br] Accept-Language:[zh-CN,zh;q=0.9] Connection:[keep-alive]]
Body =  {}
127.0.0.1:6381 连接成功
```

## HTTP客户端

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

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

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

结束的时候，需要调用Body中的Close()方法关闭io。

示例代码：

```
package main

import (
   "net/http"
   "fmt"
)

func main()  {
   // 使用Get方法获取服务器响应包数据
   //resp, err := http.Get("http://www.baidu.com")
   resp, err := http.Get("http://127.0.0.1:8000/hello")
   if err != nil {
      fmt.Println("Get err:", err)
      return
   }
   defer resp.Body.Close()

   // 获取服务器端读到的数据
   fmt.Println("Status = ", resp.Status)           // 状态
   fmt.Println("StatusCode = ", resp.StatusCode)   // 状态码
   fmt.Println("Header = ", resp.Header)           // 响应头部
   fmt.Println("Body = ", resp.Body)               // 响应包体

   buf := make([]byte, 4096)         // 定义切片缓冲区，存读到的内容
   var result string
   // 获取服务器发送的数据包内容
   for {
      n, err := resp.Body.Read(buf)  // 读body中的内容。
      if n == 0 {
         fmt.Println("Body.Read err:", err)
         break
      }
      result += string(buf[:n])     // 累加读到的数据内容
   }
   // 打印从body中读到的所有内容
   fmt.Println("result = ", result)
}
```