Go - channel 通道

```go
var c chan int
var wg sync.WaitGroup

func main() {
    // noBufChan()
    // bufChan()
    channelDemo()
}

func noBufChan() {
    c = make(chan int) //chan必须使用make来实例化分配一块内存才可以使用，无缓冲区，必须先定义一个接受方，才可以发送
    //c <- 10               //hang住了 程序无法执行下去了  值发送不进去，没有接受的缓存区；所以必须先定义一个goroutine来接受，
    wg.Add(1)
    go func() {
        defer wg.Done()
        x := <-c
        fmt.Println("X:", x)
    }()
    c <- 10
    fmt.Println("C:", c)
    wg.Wait()
    close(c)

}

func bufChan() {
    //有缓冲区 ，可以往里面先存值,通道可以先直接返送，而不需接受方
    c = make(chan int, 1) //定义1个就是说只能往通道中发送一个值，发送多个就会报错，原理同无buf
    c <- 10
    fmt.Println(c)
    close(c)
}
```

背景：

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存在不同的`goroutine`中容易发生竞态问题。为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

Go语言的并发模型是`CSP（Communicating Sequential Processes）`，提倡**通过通信共享内存**而不是**通过共享内存而实现通信。**

如果说`goroutine`是Go程序并发的执行体，`channel`就是它们之间的连接。`channel`是可以让一个`goroutine`发送特定值到另一个`goroutine`的通信机制。

描述：

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。

1. `channel`是一种类型，一种引用类型。声明通道类型的格式如下：

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

2. 通道是引用类型，通道类型的空值是`nil`。

```
var ch chan int
fmt.Println(ch) // <nil>
```

声明的通道后需要使用`make`函数初始化之后才能使用。

创建channel的格式如下：

```
make(chan 元素类型, [缓冲大小])
ch4 := make(chan int)
```

3. 通道有发送（send）、接收(receive）和关闭（close）三种操作。

   发送和接收都使用`<-`符号。

   现在我们先使用以下语句定义一个通道：

```
ch <- 10 // 把10发送到ch中   send
x := <- ch // 从ch中接收值并赋值给变量x
<-ch       // 从ch中接收值，忽略结果
close(ch)//调用内置的close函数来关闭通道。
```

4. 关于关闭通道需要注意的事情是，只有在通知接收方goroutine所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但关闭通道不是必须的。
5. 关闭后的通道有以下特点：
   - 对一个关闭的通道再发送值就会导致panic。
   - 对一个关闭的通道进行接收会一直获取值直到通道为空。
   - 对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
   - 关闭一个已经关闭的通道会导致panic。