如果说 goroutine 是 Go语言程序的并发体的话，那么 channels 就是它们之间的通信机制。

Go语言中的通道（channel）是一种特殊的类型。在任何时候，同时只能有一个 goroutine 访问通道进行发送和获取数据。goroutine 间通过通道就可以通信。  
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。

**声明通道类型**

通道本身需要一个类型进行修饰，就像切片类型需要标识元素类型。通道的元素类型就是在其内部传输的数据类型，声明如下：

var 通道变量 chan 通道类型

* 通道类型：通道内的数据类型。
* 通道变量：保存通道的变量。  
  chan 类型的空值是 nil，声明后需要配合 make 后才能使用。

**创建通道**

通道是引用类型，需要使用 make 进行创建，格式如下：

通道实例 := make(chan 数据类型)

* 数据类型：通道内传输的元素类型。
* 通道实例：通过make创建的通道句柄。

**Go语言中无缓冲的通道（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。**

**Go语言中有缓冲的通道（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。**

var chans=make(chan int)  
go func() {//不写携程会报错 因为不是同时接收和发送 在没缓冲通道时 如果：var chans=make(chan int,缓冲大小)这种情况就不需要写携程不需要同时接收和发送，只要存在接收和发送就行  
 chans<-1  
}()  
data,ok:=<-chans  
if ok{  
 fmt.Println(data)  
}

通道实例 := make(chan 通道类型, 缓冲大小)

* 通道类型：和无缓冲通道用法一致，影响通道发送和接收的数据类型。
* 缓冲大小：决定通道最多可以保存的元素数量。
* 通道实例：被创建出的通道实例。

**使用非阻塞方式从通道接收数据时，语句不会发生阻塞，格式如下：**

data, ok := <-ch

* data：表示接收到的数据。未接收到数据时，data 为通道类型的零值。
* ok：表示是否接收到数据。  
  非阻塞的通道接收方法可能造成高的 CPU 占用，因此使用非常少。如果需要实现接收超时检测，可以配合 select 和计时器 channel 进行，可以参见后面的内容。

#### 循环接收

通道的数据接收可以借用 for range 语句进行多个元素的接收操作，格式如下：

for data := range ch {

}

通道 ch 是可以进行遍历的，遍历的结果就是接收到的数据。数据类型就是通道的数据类型。通过 for 遍历获得的变量只有一个，即上面例子中的 data。

#### 接收任意数据，忽略接收的数据

阻塞接收数据后，忽略从通道返回的数据，格式如下：

<-ch

执行该语句时将会发生阻塞，直到接收到数据，但接收到的数据会被忽略。这个方式实际上只是通过通道在 goroutine 间阻塞收发实现并发同步

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。

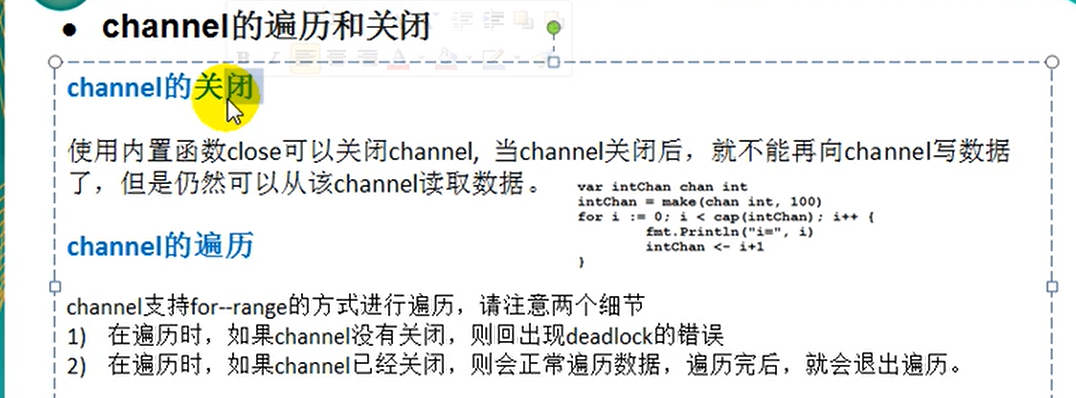
如果其中的任意一语句可以继续执行(即没有被阻塞)，那么就从那些可以执行的语句中任意选择一条来使用。

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 {} *// 阻塞*



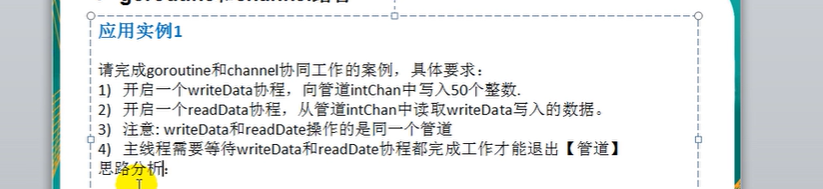
close函数是一个内建函数， 用来关闭channel，这个channel要么是双向的， 要么是只写的（chan<- Type）。  
这个方法应该只由发送者调用， 而不是接收者。  
当最后一个发送的值都被接收者从关闭的channel(下简称为c)中接收时,  
接下来所有接收的值都会非阻塞直接成功，返回channel元素的零值。  
如下的代码：  
如果c已经关闭（c中所有值都被接收）， x, ok := <- c， 读取ok将会得到false。

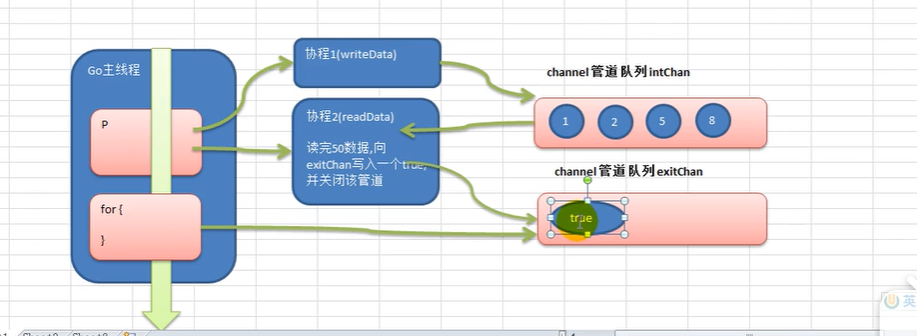
chan在go语言中相当于一个文件操作符，使用完成之后需要使用Close()函数关闭。

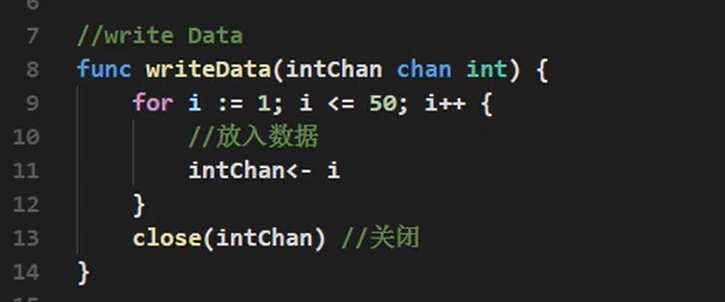
知道什么时候接收完毕使用close() 不知道什么时候接收完毕使用如下示例：

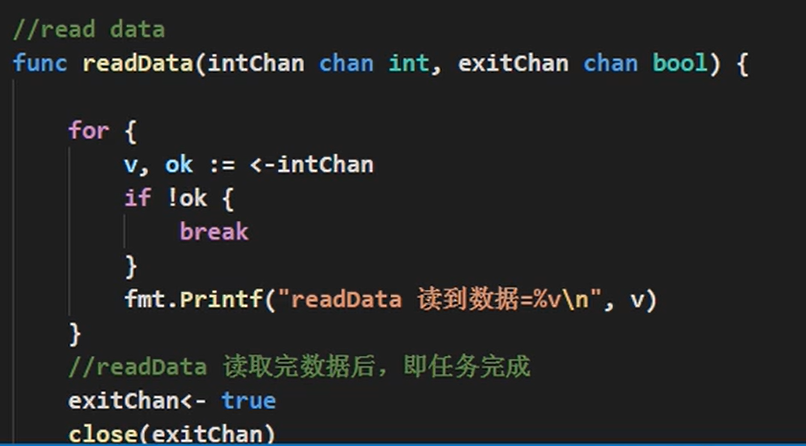
for {  
 for n:=1;n<=4;n++{  
 <-exitchan //这一步 exitchan有数据才会执行 没数据会停在这里等待  
 }  
 fmt.Printf("完毕！\n")  
 break  
}

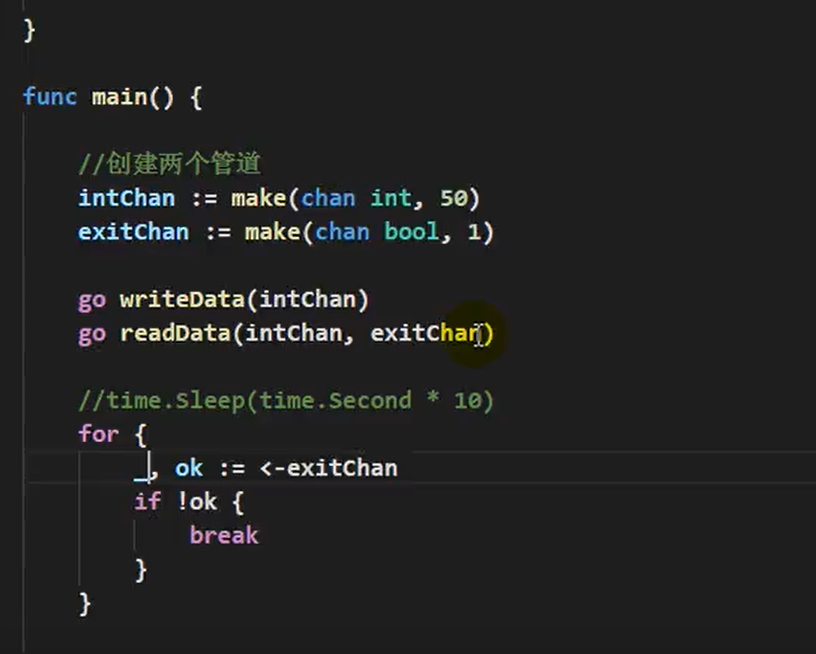












用exitchan做判断 管道不需要返回就能直接从携程中取 很好用来做判断；

func main() {  
 inchan:=make(chan interface{},10)  
 exitchan:=make(chan interface{},10)  
 go func(){  
 for i:=1;i<=50;i++{  
 inchan<-i  
 }  
 close(inchan) //写入完成需要close()  
 }()  
  
 go func(){  
 for{  
 data,ok:=<-inchan  
 if !ok{  
 exitchan<-true//管道操作用来做判断，因为不需要返回值 可以在携程中直接判断  
 close(exitchan) //写入完成需要close()  
 break  
 }else{  
 fmt.Println(data)  
 }  
 }  
 }()  
  
 for{  
 exitchandata,ok:=<-exitchan//用来判断是否接收全部完成  
 if !ok{  
 fmt.Println(exitchandata)  
 fmt.Println("结束！")  
 break  
 }  
 }  
}