通道（channel）和goroutine共同代表Go的并发编程模式和编程哲学，可以利用通道在多个goroutine之间传递数据

> Don’t communicate by sharing memory; share memory by communicating. （不要通过共享内存来通信，而应该通过通信来共享内存。）

基础使用

channel类型的值本身就是并发安全的

> channel也是Go自带的，唯一一个可以满足并发安全性的类型

- 一个通道相当于先进先出（FIFO）的队列，
- 通道中的元素按照发送顺序排列
- 先被发送的一定先被接收
- `<-` 可以发送和接收元素，箭头的方向代表元素的传输方向

基本使用

```go
package main

import "fmt"

func main() {
  // make 初始化通道，chan代表通道类型的关键字，int代表通道的类型，3为可选，表示通道的容量，即最多可以缓存的元素值，不能小于0
  // 容量为0的通道被称为非缓冲通道，大于0的称为缓冲通道，这两种通道的数据传递方式是不同的
  ch1 := make(chan int, 3)
  // 发送元素
  ch1 <- 2
  ch1 <- 1
  ch1 <- 3
  // 接收元素
  elem1 := <-ch1
  fmt.Printf("The first element received from channel ch1: %v\n",
    elem1)
}
```

基本特性

通道发送和接收操作的基本特性

- 对于同一个通道，发送操作之间是互斥的，接收操作之间也是如此
  - 元素值进入通道会被复制一份
  - 在同一时刻，Go运行时系统只会对同一通道的里任意发送操作执行某一个，直到这个元素值被完全复制，其他的发送操作才可能会被执行，接受操作也是如此
- 发送操作和接收操作中对元素值的处理是不可分割的（原子操作）
  - 要么复制完毕，要么还没开始复制，不会出现中间状态
  - 这保证了通道元素值的完整性，也保证了通道操作的唯一性
- 发送操作的代码在完全完成之前会被阻塞，接收操作之间也是如此
  - 阻塞就是为实现操作互斥和元素值的完整

注意事项

发送操作和接收操作在什么时间可能被长时间阻塞

- 对于缓冲通道
  - 如果通道已满，所有发送操作都会被阻塞，直到通道中的元素有被接收走
  - 通道会优先通知最早等待的goroutine，再次执行发送操作
  - 由于通道会按照等待顺序通知发送操作的goroutine，goroutine会顺序的进入通道内部发送等待队列，因此通知顺序是公平的，发送顺序也没有变化
  - 缓冲通道是通过异步的方式传输数据，大多数情况下，会用缓冲通道做双方的中间件，但是当发送操作在执行的时候发现空的通道中，正好有等待的接收操作，那么它会直接把元素值复制给接收方。
- 对于非缓冲通道
  - 使用同步的方式传递数据
  - 发送操作和接受操作一开始执行，对应的代码就会阻塞住
  - 只有发送和接收双方对接上，数据才会被传递
  - 并且数据是直接从发送方复制到接收方的，非缓冲通道不会做中转
- 值为nil的通道
  - 无论具体类型是怎样的，发送和接收操作永久被阻塞

> 注意：通道类型是引用类型，零值就是nil，因此只声明该类型的变量但是没有通过make对她进行初始化，该变量的值就会使nil，记得初始化通道！！

```go
package main

func main() {
    // 示例1。
    ch1 := make(chan int, 1)
    ch1 <- 1
    //ch1 <- 2 // 通道已满，因此这里会造成阻塞。

    // 示例2。
    ch2 := make(chan int, 1)
    //elem, ok := <-ch2 // 通道已空，因此这里会造成阻塞。
    //_, _ = elem, ok
    ch2 <- 1

    // 示例3。
    var ch3 chan int
    //ch3 <- 1 // 通道的值为nil，因此这里会造成永久的阻塞！
    //<-ch3 // 通道的值为nil，因此这里会造成永久的阻塞！
    _ = ch3
}
```

发送操作和接收操作在什么时候会引发 panic

- 对于已初始化，但是未关闭的通道，收发操作一定不会引发panic，但是通道一旦关闭，再进行发送操作，就会引发panic，
- 接受操作是能够感知通道的关闭，并且安全退出

```go
// 通道关闭，并且通道没有元素，ok返回false
// 如果通道关闭，但是其中还有元素值未被取出，这个元素仍然可以被取出，ok返回true
elem, ok := <-ch2
```

- 试图关闭已经关闭的通道，也会引发panic，

由于通道的上述特性，通道的关闭应该由发送方进行，简单的例子

```
```

```go
package main

import "fmt"

func main() {
    ch1 := make(chan int, 2)
    // 发送方。
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Printf("Sender: sending element %v...%d \n", i, len(ch1))
            ch1 <- i
        }
        fmt.Println("Sender: close the channel...")
        close(ch1)
    }()

    // 接收方。
    for {
        elem, ok := <-ch1
        if !ok {
            fmt.Println("Receiver: closed channel")
            break
        }
        fmt.Printf("Receiver: received an element: %v\n", elem)
    }

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

高级操作

单向通道

之前介绍的通道都是双向的，即可以发，也可以收

单向通道：只能发，或收的通道

```go
// 只发通道
var uselessChan = make(chan<- int, 1)
// 只收通道
var uselessChan = make(<-chan int, 1)
```

单向通道的作用

- 单向通道是没法传输数据的
- 它最主要的用途就是约束其他代码的行为

在函数的参数列表中使用单向通道

```go

func SendInt(ch chan<- int) {
  ch <- rand.Intn(1000)
}
```

这个函数只接受一个chan<- int类型的参数。在这个函数中的代码只能向参数ch发送元素值，而不能从它那里接收元素值。这就起到了约束函数行为的作用。

更普遍的用法是在接口类型声明中使用，这样就相当于对它的所有实现做出了约束

```
type Notifier interface {
  SendInt(ch chan<- int)
}
```

> 一个类型如果想成为一个接口类型的实现类型，那么就必须实现这个接口中定义的所有方法。

这里调用SendInt的时候只需要把一个元素的双向通道传给他就ok，Go会自动吧双向通道转为函数需要的单向通道

```
intChan1 := make(chan int, 3)
SendInt(intChan1)
```

在函数声明的结果列表中使用单向通道

```

func getIntChan() <-chan int {
  num := 5
  ch := make(chan int, num)
  for i := 0; i < num; i++ {
    ch <- i
  }
  close(ch)
  return ch
}
```

结果列表会返回 `<-chan int` 类型的通道，得到这个通道的程序就可以从通道中接受元素值，这也就是对函数调用方的一种约束，如下

```
intChan2 := getIntChan()

for elem := range intChan2 {
  fmt.Printf("The element in intChan2: %v\n", elem)
}
```

> for，range联用可以取出通道中的数据
>
> - for会不断尝试从通道中取出元素值，即使通道被关闭，也会取出所有元素值之后在结束
> - 通常，通道中没有元素值时，for语句会被阻塞在有for关键字那一行，直到有显得元素可取，如果调用函数把通道关闭了，for就会读取完元素后关闭
> - 如果通道为nil，for会永远被阻塞

select

- select语句只能与通道联用，它一般由若干个分支组成，每次只能有一个分支的代码会被运行
- select语句的分支分为两种，
  - 候选分支：候选分支总是以case开头，然后写当前分支被选中需要执行的语句
  - 默认分支：就是default case，没有分支被选中才会被执行，
- select语句是为通道专门设计的，每个case表达式只能包含操作通道的表达式

```

// 准备好几个通道。
intChannels := [3]chan int{
  make(chan int, 1),
```

```
    make(chan int, 1),
    make(chan int, 1),
}
// 随机选择一个通道，并向它发送元素值。
index := rand.Intn(3)
fmt.Printf("The index: %d\n", index)
intChannels[index] <- index
// 哪一个通道中有可取的元素值，哪个对应的分支就会被执行。
select {
case <-intChannels[0]:
    fmt.Println("The first candidate case is selected.")
case <-intChannels[1]:
    fmt.Println("The second candidate case is selected.")
case elem := <-intChannels[2]:
    fmt.Printf("The third candidate case is selected, the element is %d.\n", elem)
default:
    fmt.Println("No candidate case is selected!")
}
```

注意事项

- 有default，涉及设计通道操作的表达式是否有阻塞，select语句都不会被阻塞，如果所有的case表达式都被阻塞了，或者没有满足求值的条件，则执行default
- 无default，一旦所有case表达式没有满足求值条件，select语句会被阻塞，直到至少有一个case表达式满足条件为止
- select语句只能对每个case表达式各求值一次，如果想连续或定时操作其中的通道，通常需要for嵌套select语句实现，注意，select不会对外层的for产生作用，但是错误的用法会导致for死循环

```

intChan := make(chan int, 1)
// 一秒后关闭通道。
time.AfterFunc(time.Second, func() {
    close(intChan)
})
select {
case _, ok := <-intChan:
    if !ok {
        fmt.Println("The candidate case is closed.")
        break
    }
    fmt.Println("The candidate case is selected.")
}
```

select语句的分支选择规则

- 对于每个case表达式，至少会包含一个代表收发操作的表达式，同时也会包含其他表达式，如，包含接受表达式的短变量声明，这样就会包含了多个表达式，它包含的多个表达式总是从左往右顺序求值
- select语句包含的case表达式的求值顺序，从上往下执行，最上面的case分支最先被求值，
- 对于每个case表达式，在被求值的时候，如果相应的操作处于阻塞状态，那么对该case表达式的求值就是不成功的，这种情况被称为case表达式所在的候选分支是不满足选择条件的
- 仅当所有case表达式都被求值完成后，才会开始选择候选分支。这时候它之后挑选满足选择条件的候选分支执行，没有满足的就执行default分支，没有default分支就阻塞，，直到至少有一个候选分支满足条件才会被唤醒
- select语句发现多个候选分支满足条件，会使用伪随机的算法在选择其中一个执行。select被唤醒的时候也是采用这种方式
- 一条select语句中只能有一条default分支，default分支在没有候选分支的情况下会被执行，这与他的编写位置无关

- select语句每次执行，case表达式求值和分支选择都是独立的，注意，select执行是否是并发安全的要看分支中是否存在并发不安全的代码