Go并发编程——channel

目录：

一、多个goroutine间通信的通道channel

（一）、通道的概述

1、使用通道的意义

- 1、单纯地将函数并发执行没有意义。函数与函数间需要交换数据才能体现出并发执行的意义。
- 2、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goroutine中容易发生竞态问题，必须使用互斥对内存进行加锁，所以造成性能问题。
- 3、Go语言中提倡使用通道channel的方式代替共享内存。也就是说，Go语言中主张，应该通过数据传递来实现共享内存，而不是通过共享内存来实现消息传递。
- 4、排队的目的是避免拥堵、插队造成的资源使用和交换过程低效问题。多个goroutine为了争抢数据，势必造成低效，使用队列的方式是最高效的，channel就是一种队列结构。

2、什么是通道?

- 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。

- 通道可以被认为是Goroutine通信的管道。类似于管道中的水从一端到另一端的流动，数据可以从一端发送到另一端，通过通道接收。
- 每个通道都有与其相关的类型。该类型是通道允许传输的数据类型。(通道的零值为nil。nil通道没有任何用处，因此通道必须使用类似于map和slice的方法来定义。)

（二）、声明通道类型

1、语法

var 通道变量 chan 通道类型

2、chan类型的空值是nil，声明后需要配合make才能使用。

（三）、创建通道

1、通道是引用类型，需要使用make进行创建

语法：通道示例 := make(chan 数据类型)

2、例如：

ch1 := make(chan int) //创建一个整数类型通道

ch2 := make(chan interface{}) //创建一个空接口类型的通道，可以存放任意数据

type Equip struct {/* 属性 */}

ch3 := make(chan *Equip) //创建一个Equip指针类型的通道，可以存放Equip指针

（四）、通道发送数据

1、使用通道发送数据的格式

通道发送使用特殊的操作符"<-"，将数据通过通道发送的语法为：

通道变量 <- 值

- 通道发送的值可以是变量、常量、表达式或函数返回值等。值的类型必须与ch通道的元素类型一致。
- 把数据往通道中发送，如果接收方一直没有接收，那么发送操作将持续阻塞。此时所有的goroutine，包括main的goroutine都处于等待状态。
    - 运行会提示报错：fatal error: all goroutines are asleep - deadlock!

2、死锁deadlock

- 1、使用通道时要考虑的一个重要因素是死锁。
- 如果Goroutine在一个通道上发送数据，那么预计其他的Goroutine应该接收数据。如果这种情况不发生，那么程序将在运行时出现死锁。
- 类似地，如果Goroutine正在等待从通道接收数据，那么另一些Goroutine将会在该通道上写入数据，否则程序将会死锁。

3、示例代码：

```
func main() {
  //创建一个空指针型通道
  ch := make(chan interface{})
  //将0通过通道发送
  ch <- 0
  //发送字符串
  ch <- "StevenWang"
}
```

（五）、阻塞

1、一个通道发送和接收数据，默认是阻塞的。

- 当一个数据被发送到通道时，在发送语句中被阻塞，直到另一个Goroutine从该通道读取数据。类似地，当从通道读取数据时，读取被阻塞，直到一个Goroutine将数据写入该通道。
- 这些通道的特性是帮助Goroutines有效地进行通信，而无需像使用其他编程语言中非常常见的显式锁或条件变量。

2、示例代码：

```
func main() {
  var ch1 chan int
  fmt.Println(ch1)      //<nil>,
  fmt.Printf("%T\n", ch1) //chan int

  ch1 = make(chan int)
  fmt.Println(ch1) //0xc4200200c0
  ch2 := make(chan bool)

  go func() {
```

```
    fmt.Println("子goroutine。。。")
    data, ok := <-ch1 //阻塞式，从通道中读取数据
    time.Sleep(1 * time.Second)
    fmt.Println("子goroutine从通道中读取到main传来的数据是：", ok, data)
    ch2 <- true //向通道中写入数据，表示结束
  }()

  ch1 <- 100 //阻塞式，main goroutine向通道中写入数据
  <-ch2  //目的是防止main goroutine先执行完毕后退出。因为如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。
  fmt.Println("main..over")
}
```

运行结果：

```
<nil>
chan int
0xc420076060
子goroutine。。。
子goroutine从通道中读取到main传来的数据是： true 100
main..over
```

（六）、通道接收数据

1、使用通道接收数据的格式

通道接收同样使用特殊的操作符"<-"。

- 通道变量 <- 值
- 通道收发操作在不同的两个goroutine间进行。
- 接收操作将持续阻塞，直到发送方发送数据。
- 每次接收一个元素。

（七）、通道接收数据的四种写法

1、阻塞接收数据

- data := <-ch
- 执行该语句时将会阻塞，直到接收到数据并赋值给data变量。

2、阻塞接收数据的完整写法

- 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。
- 通过ok值可以判断当前通道是否被关闭。

3、接收任意数据，忽略接收的数据

- <-ch
- 执行该语句时将会阻塞。
- 其目的不在于接收通道中数据，而是为了阻塞goroutine。

4、循环接收数据

1）、循环接收数据

- 循环接收数据，需要配合使用关闭通道
- 借助普通for循环和for ... range语句循环接收多个元素
- 遍历通道，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。
- 普通for循环接收通道数据，需要有break循环的条件；for range会自动判断出通道已关闭，而无需通过判断来终止循环。

2）、循环接收数据的三种方式

```
func main() {
  ch1 := make(chan string)
  go sendData(ch1)

  //1、循环接收数据方式1
  for {
    data := <-ch1
    //如果通道关闭，通道中传输的数据则为各数据类型的默认值。chan int 默认值为0，
chan string默认值为"" 等。
    if data == "" {
      break
    }
    fmt.Println("从通道中读取数据方式1：", data)
  }

  //2、循环接收数据方式2
  for {
```

```
        data, ok := <-ch1
        //通过多个返回值的形式来判断通道是否关闭，如果通道关闭，则ok值为false。
        if !ok {
            break
        }
        fmt.Println("从通道中读取数据方式2：", data)
    }

    //3、循环接收数据方式3
    //for range循环会自动判断通道是否关闭，自动break循环。
    for value := range ch1 {
        fmt.Println("从通道中读取数据方式3：", value)
    }
}

func sendData(ch1 chan string) {
    for i := 1; i <= 10; i++ {
        ch1 <- fmt.Sprintf("发送数据%d\n", i)
    }
    fmt.Println("发送数据完毕。。")
    //显式调用close()实现关闭通道
    close(ch1)
}
```

(八)、关闭通道

1、发送方如果数据写入完毕，需要关闭通道，用于通知接受方数据传递完毕。一般都是发送方关闭通道。

2、如何判断一个channel是否已经关闭？可以在读取的时候使用多重返回值的方式。如果返回值是false，则表示通道已经被关闭。

3、如果往关闭的通道中写入数据，会报错：panic: send on closed channel。但是可以从关闭后的通道中取数据，不过返回数据默认值和false。

4、示例代码

```
func main() {
    //通道关闭后是否可以写入和读取呢?
    ch1 := make(chan int)
    go func() {
```

```
    ch1 <- 100
    ch1 <- 200
    close(ch1)
    //ch1 <- 10  //关闭的通道，无法写入数据
  }()

  data, ok := <-ch1
  fmt.Println("main读取数据：", data, ok)
  data, ok = <-ch1
  fmt.Println("main读取数据：", data, ok)
  data, ok = <-ch1
  fmt.Println("main读取数据：", data, ok)
  data, ok = <-ch1
  fmt.Println("main读取数据：", data, ok)
  data, ok = <-ch1
  fmt.Println("main读取数据：", data, ok)
}
```

返回结果：

```
main读取数据：  100 true
panic: send on closed channel
main读取数据：  200 true
main读取数据：  0 false
main读取数据：  0 false

main读取数据：  0 false
goroutine 5 [running]:
main.main.func1(0xc42006e060)

/Users/steven/Documents/go_project/src/ch10_2/demo01_test1.go:12
+0x79
created by main.main
        /Users/steven/Documents/go_project/src/ch10_2/demo01_test1.go:8
+0x70
```

二、缓冲通道和定向通道

（一）、缓冲通道

1、非缓冲通道：默认创建的通道都是非缓冲通道，读写都是即时阻塞；

2、缓冲通道：自带一块缓冲区，可以暂时存储数据，如果缓冲区满了， 那么才会阻塞；

3、示例代码

```
func main() {
  //1.非缓冲通道
  ch1 := make(chan int)
  fmt.Println("非缓冲通道： ", len(ch1), cap(ch1)) //0, 0
  go func() {
    data := <-ch1 //阻塞
    fmt.Println("获取数据： ", data)
  }()

  ch1 <- 100 //阻塞
  //time.Sleep(1)
  fmt.Println("写入数据ok\n----------------------")

  //2.缓冲通道,缓冲区满了才会阻塞
  //ch2 := make(chan int, 5)
  //fmt.Println("缓冲通道： ", len(ch2), cap(ch2)) //0,5
  //go func() {
  // for data := range ch2 {
  //   //time.Sleep(1)
  //   fmt.Println("获取数据： ", data)
  // }
  //}()
  //
  //ch2 <- 1
  //fmt.Println(len(ch2), cap(ch2))
  //ch2 <- 2
  //fmt.Println(len(ch2), cap(ch2))
  //ch2 <- 3
  //fmt.Println(len(ch2), cap(ch2)) //3,5
  //ch2 <- 4
  //fmt.Println(len(ch2), cap(ch2))
```

```
    //ch2 <- 5
    //fmt.Println(len(ch2), cap(ch2)) //5, 5
    //ch2 <- 6                  //阻塞
    //fmt.Println(len(ch2), cap(ch2))
    //close(ch2)
    //fmt.Println("main。。。over。。")

    //3.缓冲通道
    ch3 := make(chan string, 5)
    fmt.Printf("%T\n", ch3)
    go sendData(ch3)

    for data := range ch3 {
        //time.Sleep(1 * time.Second)
        fmt.Println("\t读取数据：", data)
    }
    fmt.Println("读取完毕。。")
}

func sendData(ch3 chan string) {
    for i := 1; i <= 10; i++ {
        ch3 <- fmt.Sprintf("data%d", i) //1,2,3,4,5,6,7
        fmt.Println("写入数据：", i)       //1,2,3,4,5
        fmt.Println(len(ch3), cap(ch3))
    }
    close(ch3)
}
```

4、缓冲通道模拟生产者和消费者

```
func main() {
    ch1 := make(chan int , 5)
    ch2 := make(chan bool)//判断结束
    rand.Seed(time.Now().UnixNano())
    //写入数据：生产者
    go func() {
        for i := 1; i <= 20; i++ {
            ch1 <- i
            fmt.Println("写入数据：", i)
```

```
        time.Sleep(time.Duration(rand.Intn(1000))*time.Millisecond)
    }
    close(ch1)
  }()

  //读取数据：消费者
  go func() {
    for data := range ch1 {
      fmt.Println("\t1号消费者：", data)
      time.Sleep(time.Duration(rand.Intn(1000))*time.Millisecond)
    }
    ch2 <- true
  }()
  go func() {
    for data := range ch1 {//1
      fmt.Println("\t2号消费者：", data)
      time.Sleep(time.Duration(rand.Intn(1000))*time.Millisecond)
    }
    ch2 <- true
  }()
  <- ch2
  fmt.Println("main...over...")
}
```

（二）、定向通道

1、通道默认都是双向通道。即可写入数据，又可读取数据。

2、定向通道：也叫单向通道，只读，或只写。

- 只读：make(<- chan Type)，只能读取数据，不能写入数据
  <- chan
- 只写：make(chan <- Type)，只能写入数据，不能读取数据
  chan <- data

3、创建通道时，采用单向通道，没有意义的。都是创建双向通道。

- 将通道作为参数传递的时候使用单向通道。
  定义函数，只有写入数据功能，或定义函数，只有读取数据功能。

4、定向通道的意义：在语法级别，保证通道的操作安全

5、示例代码

```
func main() {
  //1.双向通道
  ch1 := make(chan string)
  go fun1(ch1)

  data := <-ch1
  fmt.Println("main，接受到数据：", data)
  ch1 <- "Go语言好学么？"
  ch1 <- "区块链好学么?"

  go fun2(ch1)
  go fun3(ch1)

  time.Sleep(1 * time.Second)
  fmt.Println("main over!")
}

func fun1(ch1 chan string) {
  ch1 <- "我是Steven老师"
  data := <-ch1
  data2 := <-ch1
  fmt.Println("回应：", data, data2)
}

//功能：只有写入数据
func fun2(ch1 chan<- string) {
  //只能写入
  ch1 <- "How are you?"
  //<- ch1 //invalid operation: <-ch1 (receive from send-only type chan<- string)
}

//功能：只有读取数据
func fun3(ch1 <-chan string) {
  data := <-ch1
  fmt.Println("只读：", data)
  //ch1 <- "hello" //invalid operation: ch1 <- "hello" (send to receive-only type <-chan
```

```
string）
}
```

三、time包中的定向通道

（一）、Timer结构体

1、计时器类型表示单个事件。当计时器过期时，当前时间将被发送到C上(C是一个只读通道<-chan time.Time，该通道中放入的是Time结构体)，除非计时器是AfterFunc创建的。计时器必须使用NewTimer或AfterFunc创建。

2、Timer结构体的源码

```
// The Timer type represents a single event.
// When the Timer expires, the current time will be sent on C,
// unless the Timer was created by AfterFunc.
// A Timer must be created with NewTimer or AfterFunc.
type Timer struct {
  C <-chan Time
  r runtimeTimer
}
```

（二）、NewTimer函数

1、NewTimer创建一个新的计时器，它会在至少持续时间d之后将当前时间发送到其通道上。

2、NewTimer()函数的源码

```
// NewTimer creates a new Timer that will send
// the current time on its channel after at least duration d.
func NewTimer(d Duration) *Timer
```

3、示例代码

```
//.创建计时器
timer1 := time.NewTimer(5 * time.Second)
fmt.Printf("%T\n", timer1) //*time.Timer
fmt.Println(time.Now())
data := <-timer1.C //<-chan time.Time
fmt.Printf("%T\n",timer1.C) //<-chan time.Time
```

```
fmt.Printf("%T\n",data) //time.Time
fmt.Println(data)
```

（三）、After()函数

1、After()函数相当于NewTimer(d). C。

2、After()函数的源码

```
// After waits for the duration to elapse and then sends the current time
// on the returned channel.
// It is equivalent to NewTimer(d).C.
// The underlying Timer is not recovered by the garbage collector
// until the timer fires. If efficiency is a concern, use NewTimer
// instead and call Timer.Stop if the timer is no longer needed.
func After(d Duration) <-chan Time {
  return NewTimer(d).C
}
```

3、示例代码

```
//使用After(),返回值<- chan Time,同Timer.C
ch1 := time.After(5 * time.Second)
fmt.Println(time.Now())
data := <-ch1
fmt.Printf("%T\n",data) //time.Time
fmt.Println(data)
```

四、select分支语句

- select 语句类似于 switch 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。
    - 每个case都必须是一个通道
    - 所有channel表达式都会被求值
    - 所有被发送的表达式都会被求值
    - 如果任意某个通道可以进行，它就执行；其他被忽略。

（一）、执行流程

1. 如果有多个case都可以运行，select会随机公平地选出一个执行，其他不

会执行；

2. 如果有default子句，则执行该语句；

3. 如果没有default字句，select将阻塞，直到某个通道可以运行；

4. Go不会重新对channel或值进行求值。

（二）、示例代码

1、示例代码1

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)

    go func() {
        time.Sleep(1 * time.Second)
        ch1 <- 100
    }()

    go func() {
        time.Sleep(1 * time.Second)
        ch2 <- 200
    }()
    select {
    case data := <-ch1:
        fmt.Println("ch1中读取数据了:", data)
    case data := <-ch2:
        fmt.Println("ch2中读取数据了： ", data)
    default:
        fmt.Println("执行了default。。。")
    }
}
```

运行结果：

执行了default。。。

2、示例代码2

```
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
```

```
    go func() {
        time.Sleep(1 * time.Second)
        data := <-ch1
        fmt.Println("ch1：", data)
    }()

    go func() {
        time.Sleep(2 * time.Second)
        data := <-ch2
        fmt.Println("ch2：", data)
    }()

    select {
    case ch1 <- 100: //阻塞
        close(ch1)
        fmt.Println("ch1中写入数据。。")

    case ch2 <- 200: //阻塞
        close(ch2)
        fmt.Println("ch2中写入数据。。")

    case <-time.After(2 * time.Second): //阻塞
        fmt.Println("执行延时通道")

        //default:
        // fmt.Println("default..")
    }

    time.Sleep(4 * time.Second)
    fmt.Printf("main  over ")
}
```

运行结果：

ch1： 100

ch1中写入数据。。

main over

五、sync包中的WaitGroup

sync包提供了基本的同步单元，如互斥锁。除了Once和WaitGroup类型，大部分都是适用于低水平程序线程，高水平的同步使用channel通信更好一些。本包的类型的值不应被拷贝。

之前的案例中，我们都使用time.Sleep()函数，通过睡眠将主线程阻塞至所有线程结束。而更好的做法是使用WaitGroup来实现。

（一）、WaitGroup同步等待组

1、同步sync与异步async

- 同步：sync是串行执行
- 异步：async是同时执行

2、WaitGroup：同步等待组

```
type WaitGroup struct {
        noCopy noCopy
        state1 [12]byte
        sema   uint32
}
```

WaitGroup用于等待一组线程的结束。父线程调用Add方法来设定应等待的线程的数量。每个被等待的线程在结束时应调用Done方法。同时，主线程里可以调用Wait方法阻塞至所有线程结束。

3、WaitGroup中的方法

- func (wg *WaitGroup) Add(delta int)

Add方法向内部计数加上delta，delta可以是负数；如果内部计数器变为0，Wait方法阻塞等待的所有线程都会释放，如果计数器小于0，方法panic。注意Add加上正数的调用应在Wait之前，否则Wait可能只会等待很少的线程。一般来说本方法应在创建新的线程或者其他应等待的事件之前调用。

- func (wg *WaitGroup) Done()

Done方法减少WaitGroup计数器的值，应在线程的最后执行。

- func (wg *WaitGroup) Wait()

Wait方法阻塞直到WaitGroup计数器减为0。

4、示例代码

```
func main() {
  var wg sync.WaitGroup
  fmt.Printf("%T\n", wg) //sync.WaitGroup
  fmt.Println(wg)       //{{} [0 0 0 0 0 0 0 0 0 0 0 0]0}
  wg.Add(3)
  rand.Seed(time.Now().UnixNano())

  go printNum1(&wg)
  go printNum2(&wg)
  go printNum3(&wg)

  wg.Wait() //main goroutine进入阻塞状态，当计数器为0后解除阻塞
  fmt.Println("main解除阻塞，main over...")
}

func printNum1(wg *sync.WaitGroup) {
  for i := 1; i <= 10; i++ {
    fmt.Println("子goroutine1,i：", i)
    time.Sleep(time.Duration(rand.Intn(1000)))
  }
  wg.Done() //计数器减1
}

func printNum2(wg *sync.WaitGroup) {
  for i := 1; i <= 10; i++ {
    fmt.Println("\t 子goroutine1,i：", i)
    time.Sleep(time.Duration(rand.Intn(1000)))
  }
  wg.Done() //计数器减1
}

func printNum3(wg *sync.WaitGroup) {
  for i := 1; i <= 10; i++ {
    fmt.Println("\t\t 子goroutine1,i：", i)
    time.Sleep(time.Duration(rand.Intn(1000)))
```

```
    }
    wg.Done() //计数器减1
}
```

六、sync包中的Mutex

（一）、互斥锁Mutex

1、原型

```
type Mutex struct {
    state int32
    sema  uint32
}
```

Mutex是一个互斥锁，可以创建为其他结构体的字段；零值为解锁状态。Mutex类型的锁和线程无关，可以由不同的线程加锁和解锁。

2、Mutex中的方法

- func (m *Mutex) Lock()

Lock方法锁住m，如果m已经加锁，则阻塞直到m解锁。

- func (m *Mutex) Unlock()

Unlock方法解锁m，如果m未加锁会导致运行时错误。锁和线程无关，可以由不同的线程加锁和解锁。

3、互斥锁实现售票示例代码

```
var tickts = 100 //全局变量，仅一份
var wg sync.WaitGroup
var mutex sync.Mutex //互斥锁
func main() {
    /*
    练习题：模拟火车站卖票
    火车票100张，4个售票口出售(4个goroutine)。
     */
    var wg sync.WaitGroup
    wg.Add(4)
    go saleTickets("售票口1", &wg) //g1
    go saleTickets("售票口2", &wg) //g2
    go saleTickets("售票口3", &wg) //g3
```

```
    go saleTickets("售票口4", &wg) //g4

    wg.Wait()
    fmt.Println("所有车票已售空。程序结束！")
}

func saleTickets(name string, wg *sync.WaitGroup) {
    for {
        mutex.Lock()
        if tickts > 0 {
            time.Sleep(1 * time.Second)
            fmt.Println(name, "：", tickts) //1
            tickts--
        } else {
            fmt.Println(name, ",结束卖票。。")
            mutex.Unlock()
            break
        }
        mutex.Unlock() //解锁
    }
    wg.Done()
}
```

（二）、读写互斥锁RWMutex

1、原型

```
type RWMutex struct {
        w           Mutex  // held if there are pending writers
        writerSem   uint32 // semaphore for writers to wait for completing readers
        readerSem   uint32 // semaphore for readers to wait for completing writers
        readerCount int32  // number of pending readers
        readerWait  int32  // number of departing readers
}
```

RWMutex是读写互斥锁。该锁可以被同时多个读取者持有或唯一个写入者持有。RWMutex可以创建为其他结构体的字段；零值为解锁状态。RWMutex类型的锁也和线程无关，可以由不同的线程加读取锁/写入和解读取锁/写入锁。

锁定的规则：

读写锁的使用中：写操作都是互斥的、读和写是互斥的、读和读不互

斥。

理解为：

可以多个goroutine同时读取数据，但是写只允许一个goroutine写数据。

2、Mutex中的方法

- `func (rw *RWMutex) Lock()`

Lock方法将rw锁定为写入状态，禁止其他线程读取或者写入。

- `func (rw *RWMutex) Unlock()`

Unlock方法解除rw的写入锁状态，如果m未加写入锁会导致运行时错误。

- `func (rw *RWMutex) RLock()`

RLock方法将rw锁定为读取状态，禁止其他线程写入，但不禁止读取。

- `func (rw *RWMutex) RUnlock()`

Runlock方法解除rw的读取锁状态，如果m未加读取锁会导致运行时错误。

- `func (rw *RWMutex) RLocker() Locker`

Rlocker方法返回一个互斥锁，通过调用rw.Rlock和rw.Runlock实现了Locker接口。

3、示例代码

```
func main() {
  var rwm sync.RWMutex
  for i := 1; i <= 3; i++ {
    go func(i int) {
      fmt.Printf("goroutine %d，尝试读锁定。。\n", i)
      rwm.RLock()
      fmt.Printf("goroutine %d，已经读锁定了。。\n", i)
      time.Sleep(5 * time.Second)
      fmt.Printf("goroutine %d,读解锁。。\n", i)
      rwm.RUnlock()
    }(i)
  }

  time.Sleep(1*time.Second)
  fmt.Println("main..尝试写锁定。。")
```

```
    rwm.Lock()
    fmt.Println("main。。已经写锁定了。。")
    rwm.Unlock()
    fmt.Printf("main。。写解锁。。。")
}
```

七、sync包中的条件变量Cond

（一）、条件变量Cond

1、原型

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

Cond实现了一个条件变量，一个线程集合地，供线程等待或者宣布某事件的发生。

每个Cond实例都有一个相关的锁（一般是*Mutex或*RWMutex类型的值），它必须在改变条件时或者调用Wait方法时保持锁定。Cond可以创建为其他结构体的字段，Cond在开始使用后不能被拷贝。条件变量：sync.Cond,多个goroutine等待或接受通知的集合地

2、Cond中的方法

- `func NewCond(l Locker) *Cond`

使用锁l创建一个*Cond。Cond条件变量，总是要和锁结合使用。

- `func (c *Cond) Broadcast()`

Broadcast唤醒所有等待c的线程。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

- `func (c *Cond) Signal()`

Signal唤醒等待c的一个线程（如果存在）。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。发送通知给一个人。

- `func (c *Cond) Wait()`

Wait自行解锁c.L并阻塞当前线程，在之后线程恢复执行时，Wait方法会在返回前锁定c.L。和其他系统不同，Wait除非被Broadcast或者Signal唤醒，不会主动返回。广播给所有人。

因为线程中Wait方法是第一个恢复执行的，而此时c.L未加锁。调用者不应假设Wait恢复时条件已满足，相反，调用者应在循环中等待：

3、示例代码

```
func main() {
    var mutex sync.Mutex
    cond := sync.Cond{L:&mutex}
    condition := false

    go func() {
        time.Sleep(1*time.Second)
        cond.L.Lock()
        fmt.Println("子goroutine已经锁定。。。")
        fmt.Println("子goroutine更改条件数值，并发送通知。。")
        condition = true//更改数值
        cond.Signal() //发送通知：一个goroutine
        fmt.Println("子gorutine。。。继续。。。")
        time.Sleep(5*time.Second)
        fmt.Println("子groutine解锁。。")
        cond.L.Unlock()
    }()

    cond.L.Lock()
    fmt.Println("main..已经锁定。。。")
    if !condition{
        fmt.Println("main..。即将等待。。。")
        //wait()
        // 1.wait尝试解锁,
        // 2.等待--->当前的groutine进入了阻塞状态，等待被唤醒：signal(),broadcast()
        // 3.一旦被唤醒后，又会锁定
        cond.Wait()
        fmt.Println("main.被唤醒。。")
    }
    fmt.Println("main。。。继续")
    fmt.Println("main..解锁。。。")
```

```
    cond.L.Unlock()
}
```