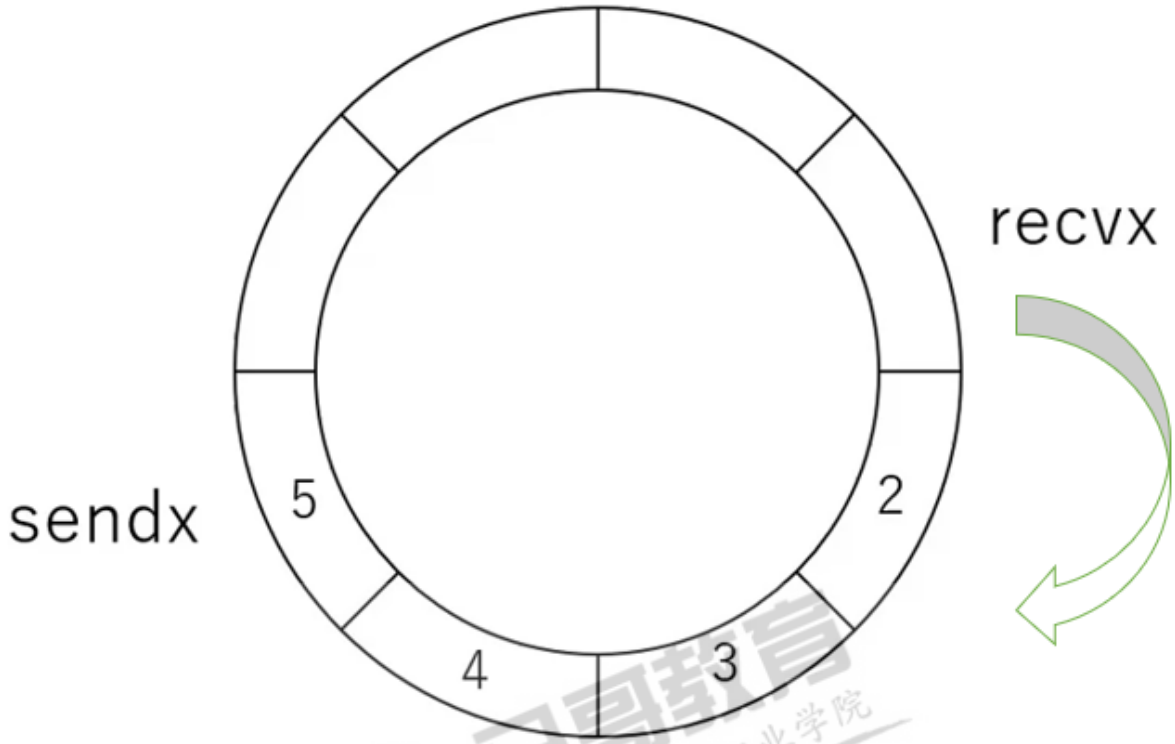
Channel通道

核心数据结构



Channel底层是一个先进先出的环形队列（固定大小环形数组实现）

- full或empty就会阻塞
- send发送
- recv接收并移除
- sendx表示最后一次插入元素的index
- recvx表示最后一次接收元素的index
- 发送、接收的操作符号都是`<-`

通道构造

runtime/chan.go/makechan

```
var c1 chan int
fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1) // c1: 0, 0, <nil>

c1 <- 111 // 阻塞，不报错。由于没有初始化容器，111塞不进去
<- c1 // 也阻塞，不报错，什么都拿不出来
```

chan零值是nil，即可以理解未被初始化通道这个容器。nil通道可以认为是一个只要操作就阻塞当前协程的容器。这种通道不要创建和使用，阻塞后无法解除，底层源码中写明了无解。

更多的时候，使用make来创建channel。

```go
// 容量为0的非缓冲通道
c2 := make(chan int, 0)
fmt.Printf("c2: %d, %d, %v\n", len(c2), cap(c2), c2)
c3 := make(chan int)
fmt.Printf("c3: %d, %d, %v\n", len(c3), cap(c3), c3)
```

非缓冲通道：容量为0的通道，也叫同步通道。这种通道发送第一个元素时，如果没有接收操作就立即阻塞，直到被接收。同样接收时，如果没有数据被发送就立即阻塞，直到有数据发送。

```go
package main

import "fmt"

func main() {
    var c1 chan int
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
    fmt.Println("准备发送数据111")
    c1 <- 111 // 往c1里面发送，阻塞在这一句，死锁，因为本例子无人接收
    fmt.Println("发送数据111结束")
}
```

缓冲通道：容量不为0的通道。通道已满，发送操作会被阻塞；通道为空，接收操作会被阻塞。

```go
package main

import "fmt"

func main() {
    c4 := make(chan int, 8) // 缓冲通道，容量为8，长度为0
    fmt.Printf("c4: %d, %d, %v\n", len(c4), cap(c4), c4)
    // 发送数据
    c4 <- 111
    c4 <- 222
    fmt.Printf("c4: %d, %d, %v\n", len(c4), cap(c4), c4) // len 2
    // 接收
    <-c4
    t := <-c4
    fmt.Printf("%T %[1]v\n", t)
}
```

单向通道

- `<- chan type` 这种定义表示只从一个channel里面拿，说明这是只读的
- `chan <- type` 这种定义表示只往一个channel里面写，说明这是只写的

```go
package main

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

```
    "time"
)

func produce(ch chan<- int) { // 生产，只写。只要该通道具有写能力就行
    for {
        ch <- rand.Intn(10)
        time.Sleep(1 * time.Second)
    }
}

func consume(ch <-chan int) { // 消费，只读。只要该通道具有读能力就行
    for {
        t := <-ch
        fmt.Println("消费，从只读通道接收", t)
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    c := make(chan int) // 创建可读/写非缓冲通道
    go produce(c)
    go consume(c)
    wg.Wait()
}
```

通道关闭

- 使用close(ch)关闭一个通道
- 只有发送方才能关闭通道，一旦通道关闭，发送者不能再往其中发送数据，否则panic
- 通道关闭作用：告诉接收者再无新数据可以发送了
- 通道关闭
    - `t, ok := <-ch` 或 `t := <-ch` 从通道中读取数据
    - 正在阻塞等待通道中的数据的接收者，由于通道被关闭，接收者将不再阻塞，获取数据失败ok为false，返回零值
    - 接收者依然可以访问关闭的通道而不阻塞
        - 如果通道内还有剩余数据，ok为true，接收数据
        - 如果通道内剩余的数据被拿完了，继续接收不阻塞，ok为false，返回零值
- 已经关闭的通道，若再次关闭则panic，因此不要重复关闭

通道遍历

1、nil通道

发送、接收、遍历都阻塞

2、缓冲的、未关闭的通道

相当于一个无限元素的通道，迭代不完，阻塞在等下一个元素来迭代上。

```go
package main

import (
    "fmt"
)

func main() {
    c1 := make(chan int, 5) // 缓冲，未关闭通道
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
    c1 <- 111
    c1 <- 222
    c1 <- 333
    fmt.Println(<-c1, "###") // 故意读走一个

    for v := range c1 {
        fmt.Println(v, "~~~") // 打印元素
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~") // 看不到这一句
}
```

3、缓冲的、关闭的通道

关闭后，通道不能在进入新的元素，那么相当于遍历有限个元素容器，遍历完就结束了。

```go
package main

import (
    "fmt"
)

func main() {
    c1 := make(chan int, 5) // 缓冲，未关闭通道
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
    c1 <- 111
    c1 <- 222
    c1 <- 333
    fmt.Println(<-c1, "###") // 故意读走一个
    close(c1)                // 关闭通道，不许再进数据
    for v := range c1 {
        fmt.Println(v, "~~~") // 打印元素
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~") // 打印了这一句
}
```

4、非缓冲、未关闭通道

相当于一个无限元素的通道，迭代不完，阻塞在等下一个元素来迭代上。

```go
package main

import (
    "fmt"
    "time"
)

```

```go
func main() {
    c1 := make(chan int) // 非缓冲，未关闭通道
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)

    go func() {
        count := 1
        for {
            time.Sleep(3 * time.Second)
            c1 <- count
            count++
        }
    }()

    for v := range c1 {
        fmt.Println(v, "~~~") // 打印元素
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~") // 看不到这一句
}
```

5、非缓冲、关闭通道

关闭后，通道不能在进入新的元素，那么相当于遍历有限个元素容器，遍历完就结束了。

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan int) // 非缓冲，未关闭通道
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)

    go func() {
        defer close(c1)
        count := 1
        for i := 0; i < 5; i++ {
            time.Sleep(3 * time.Second)
            c1 <- count
            count++
        }
    }()

    for v := range c1 {
        fmt.Println(v, "~~~") // 打印元素
    }
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~") // 打印了这一句
}
```

除nil通道外

- 未关闭通道，如同一个无限的容器，将一直迭代通道内元素，没有元素就阻塞

- 已关闭通道，将不能加入新的元素，迭代完当前通道内的元素，哪怕是0个元素，然后结束迭代

定时器

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.NewTicker(2 * time.Second)
    for {
        fmt.Println(<-t.C) // 通道每阻塞2秒就接收一次
    }
}
```

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.NewTimer(5 * time.Second)
    for {
        fmt.Println(<-t.C) // 通道阻塞2秒后只能接受一次
    }
}
```


通道死锁

channel满了，就阻塞写；channel空了，就阻塞读。容量为0的通道可以理解为有1个元素都满了。

阻塞了当前协程之后会交出CPU，去执行其他协程，希望其他协程帮助自己解除阻塞。

main函数结束了，整个进程结束了。

如果在main协程中，执行语句阻塞时，环顾四周，如果已经没有其他子协程可以执行，只剩主协程自己，解锁无望了，就自己把自己杀掉，报一个fatal error deadlock

```go
package main

import (
    "fmt"
)

```

```
func main() {
    c1 := make(chan int) // 非缓冲，未关闭通道
    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
    c1 <- 111 // 当前协程阻塞，无人能解，死锁
}

运行结果如下
go run main.go
c1: 0, 0, 0xc00001a120
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        O:/pros/main.go:10 +0xea
exit status 2
```

如果通道阻塞不在main协程中发生，而是发生在子协程中，子协程会继续阻塞着，也可能发生死锁。但是由于至少main协程是一个值得等待的希望，编译器不能帮你识别出死锁。如果真的无任何协程帮助该协程解除阻塞状态，那么事实上该子协程解锁无望，已经死锁了。

死锁的危害可能会导致进程活着，但实际上某些协程未真正工作而阻塞，应该有良好的编码习惯，来减少死锁的出现。

struct{}型通道

前面我们讲过定义结构体时struct{}部分才是类型本身。如果一个结构体类型就是struct{}，说明该结构体的实例没有数据成员，也就是实例内存占用为0。

这种类型数据构成的通道，非常节约内存，仅仅是为了传递一个信号标志。

```
package main

import (
    "fmt"
    "time"
)

func main() {
    flag := make(chan struct{}) // 比 chan bool省内存
    go func() {
        time.Sleep(3 * time.Second)
        flag <- struct{}{} // 无数据成员的结构体实例
    }()
    fmt.Printf("终于等到了信号, %T, %[1]v", <-flag)
}
```

通道多路复用

Go语言提供了select来监听多个channel。

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    count := make(chan int, 4)
    fin := make(chan bool)

    go func() {
        defer func() { fin <- true }()
        for i := 0; i < 10; i++ {
            count <- i
            time.Sleep(1 * time.Second)
        }
    }()

    for {
        select { // 监听多路通道
        case n := <-count:
            fmt.Println("count =", n)
        case <-fin:
            fmt.Println("结束")
            goto END
        // default:
        //  fmt.Println("缺省")
        }
    }
END:
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")
}
```


通道并发

Go语言采用并发同步模型叫做Communication Sequential Process通讯顺序进程，这是一种消息传递模型，在goroutine间传递消息， 而不是对数据进行加锁来实现同步访问。在goroutine之间使用channel来同步和传递数据。

- 多个协程之间通讯的管道
- 一端推入数据，一端拿走数据
- 同一时间，只有一个协程可以访问通道的数据
- 协调协程的执行顺序

如果多个线程都使用了同一个数据，就会出现竞争问题。因为线程的切换不会听从程序员的意志，时间片用完就切换了。解决办法往往需要加锁，让其他线程不能共享数据进行修改，从而保证逻辑正确。但锁的引入严重影响并行效率。

需求：

1、有一个全局数count，初始为0。编写一个函数inc，能够对count增加10万次。执行5次inc函数，请问最终count值是多少?

```go
package main

import (
    "fmt"
    "runtime"
    "time"
)

var count int64 = 0

func inc() {
    for i := 0; i < 100000; i++ {
        // count = count + 1
        count++
    }
}

func main() {
    start := time.Now()
    inc()
    inc()
    inc()
    inc()
    inc()
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~")

    fmt.Printf("执行时长：%d 微秒\n", time.Since(start).Microseconds())
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Printf("count：%d\n", count)
}


Go协程数：1
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
执行时长：508 微秒
Go协程数：1
count：500000
```

这是串行，没有并发。

2、如果并发执行inc函数，该怎么做呢，请问最终count值是多少?

```go
package main
```

```go

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

var wg sync.WaitGroup
var count int64 = 0

func inc() {
    defer wg.Done()
    for i := 0; i < 100000; i++ {
        // count = count + 1
        count++
    }
}

func main() {
    start := time.Now()
    wg.Add(5)
    for i := 0; i < 5; i++ {
        go inc()
    }
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    wg.Wait()
    fmt.Printf("执行时长：%d 微秒\n", time.Since(start).Microseconds())
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Printf("count：%d\n", count)
}


Go协程数：6
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
执行时长：1563 微秒
Go协程数：1
count：152122
```

开了5个协程并发，count结果不为50万了。为什么？count随机了。

原因在于count++不是原子操作，会被打断。所以，即使使用goroutine也会有竞争，一样会有并发安全问题。换成下句试一试

```go
atomic.AddInt64(&count, 1) // count++

Go协程数：6
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
执行时长：7166 微秒
Go协程数：1
count：500000
```

结果正确了，但是这种共享内存的方式执行时长明显增加。

也可以使用互斥锁来保证count++的原子性操作

```
var wg sync.WaitGroup
var mx sync.Mutex
var count int64 = 0

func inc() {
    defer wg.Done()
    for i := 0; i < 100000; i++ {
        // count = count + 1
        mx.Lock()
        count++
        mx.Unlock()
    }
}


Go协程数：6
~~~~~~~~~~~~~~~~~~~~~~~~~~
执行时长：16353 微秒
Go协程数：1
count：500000
```

3、是否能使用通道，来同步多个协程

```
package main

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

var wg sync.WaitGroup
var ch = make(chan int64, 1)

func inc() {
    defer wg.Done()
    for i := 0; i < 100000; i++ {
        t := <-ch
        t++
        ch <- t
    }
}

func main() {
    start := time.Now()
    ch <- 0
    wg.Add(5)
    for i := 0; i < 5; i++ {
```

```
        go inc()
    }
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Println("~~~~~~~~~~~~~~~~~~~~~~~~~~~")
    wg.Wait()
    fmt.Printf("执行时长：%d 微秒\n", time.Since(start).Microseconds())
    fmt.Printf("Go协程数：%d\n", runtime.NumGoroutine())
    fmt.Printf("count：%d\n", <-ch)
}


Go协程数：6
~~~~~~~~~~~~~~~~~~~~~~~~~~~
执行时长：131676 微秒
Go协程数：1
count：500000
```

上例是计算密集型，对同一个数据进行争抢，**不是**能发挥并行计算优势的例子，也不适合使用通道，用锁实现更有效率，更有优势。

只是为了让大家体会串行、并行执行，以及不同并行方式的思维和差异。

通道适合数据流动的场景

- 如同管道一样，一级一级处理，一个协程处理完后，发送给其他协程
- 生产者、消费者模型，M:N

协程泄露

原因

- 协程阻塞，未能如期结束，之后就会大量累积
- 协程阻塞最常见的原因都跟通道有关
- 由于每个协程都要占用内存，所以协程泄露也会导致内存泄露

因此，如果你不知道你创建的协程何时能够结束，就不要使用它。否则可能协程泄露。