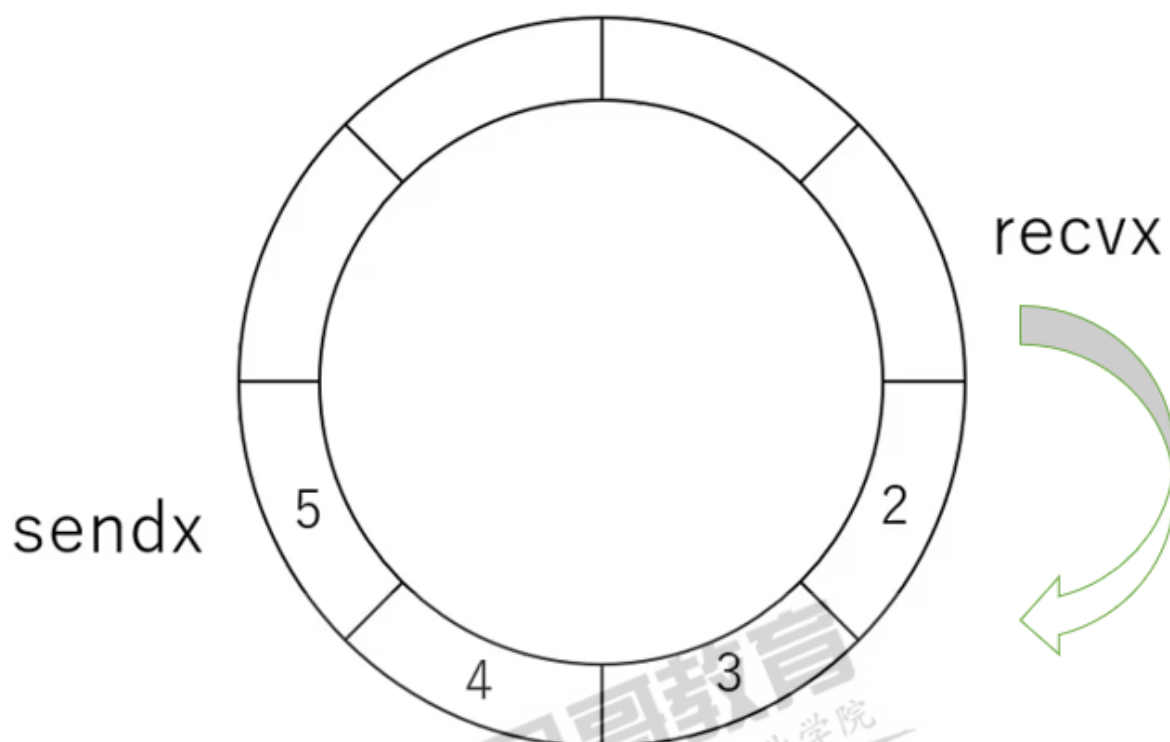


Channel通道

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

源码runtime/chan.go/makechan

```
1 var c1 chan int
2 fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1) // c1: 0, 0, <nil>
3
4 c1 <- 111 // 阻塞，不报错。由于没有初始化容器，111塞不进去
5 <- 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法解除。

更多的时候，使用make来创建channel。

```

1 // 容量为0的非缓冲通道
2 c2 := make(chan int, 0)
3 fmt.Printf("c2: %d, %d, %v\n", len(c2), cap(c2), c2)
4 c3 := make(chan int)
5 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。

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     var c1 chan int
7     fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
8     fmt.Println("准备发送数据111")
9     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
10    fmt.Println("发送数据111结束")
11 }

```

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。

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     c4 := make(chan int, 8) // 缓冲通道，容量为8，长度为0
7     fmt.Printf("c4: %d, %d, %v\n", len(c4), cap(c4), c4)
8     // 发送数据
9     c4 <- 111
10    c4 <- 222
11    fmt.Printf("c4: %d, %d, %v\n", len(c4), cap(c4), c4) // len 2
12    // 接收
13    <-c4
14    t := <-c4
15    fmt.Printf("%T %[1]v\n", t)
16 }

```

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

```

1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sync"

```

```

7     "time"
8 )
9
10 func produce(ch chan<- int) { // 生产，只写。只要该通道具有写能力就行
11     for {
12         ch <- rand.Intn(10)
13         time.Sleep(1 * time.Second)
14     }
15 }
16
17 func consume(ch <-chan int) { // 消费，只读。只要该通道具有读能力就行
18     for {
19         t := <-ch
20         fmt.Println("消费，从只读通道接收", t)
21     }
22 }
23
24 func main() {
25     var wg sync.WaitGroup
26     wg.Add(1)
27     c := make(chan int) // 创建可读/写非缓冲通道
28     go produce(c)
29     go consume(c)
30     wg.Wait()
31 }

```

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到达了
- 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到达。

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     c1 := make(chan int, 5) // 缓冲, 未关闭通道
9     fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
10    c1 <- 111
11    c1 <- 222
12    c1 <- 333
13    fmt.Println(<-c1, "###") // 故意读走一个
14
15    for v := range c1 {
16        fmt.Println(v, "~~~") // 打印元素
17    }
18    fmt.Println("~~~~~") // 看不到这一句
19 }

```

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。

```

1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     c1 := make(chan int, 5) // 缓冲, 未关闭通道
9     fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
10    c1 <- 111
11    c1 <- 222
12    c1 <- 333
13    fmt.Println(<-c1, "###") // 故意读走一个
14    close(c1)                // 关闭通道, 不许再进数据
15    for v := range c1 {
16        fmt.Println(v, "~~~") // 打印元素
17    }
18    fmt.Println("~~~~~") // 打印了这一句
19 }

```

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到达。

```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7

```

```

8 func main() {
9     c1 := make(chan int) // 非缓冲, 未关闭通道
10    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
11
12    go func() {
13        count := 1
14        for {
15            time.Sleep(3 * time.Second)
16            c1 <- count
17            count++
18        }
19    }()
20
21    for v := range c1 {
22        fmt.Println(v, "~~~") // 打印元素
23    }
24    fmt.Println("~~~~~") // 看不到这一句
25 }

```

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。

```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     c1 := make(chan int) // 非缓冲, 未关闭通道
10    fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
11
12    go func() {
13        defer close(c1)
14        count := 1
15        for i := 0; i < 5; i++ {
16            time.Sleep(3 * time.Second)
17            c1 <- count
18            count++
19        }
20    }()
21
22    for v := range c1 {
23        fmt.Println(v, "~~~") // 打印元素
24    }
25    fmt.Println("~~~~~") // 打印了这一句
26 }

```

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

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     t := time.NewTicker(2 * time.Second)
10    for {
11        fmt.Println(<-t.C) // 通道每阻塞2秒就接收一次
12    }
13 }
```

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     t := time.NewTimer(5 * time.Second)
10    for {
11        fmt.Println(<-t.C) // 通道阻塞2秒后只能接受一次
12    }
13 }
```

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

```
1 package main
2
3 import (
4     "fmt"
5 )
6
```

```

7 func main() {
8     c1 := make(chan int) // 非缓冲，未关闭通道
9     fmt.Printf("c1: %d, %d, %v\n", len(c1), cap(c1), c1)
10    c1 <- 111 // 当前协程阻塞，无人能解，死锁
11 }
12
13 运行结果如下
14 $ go run main.go
15 c1: 0, 0, 0xc00001a120
16 fatal error: all goroutines are asleep - deadlock!
17
18 goroutine 1 [chan send]:
19 main.main()
20      o:/pros/main.go:10 +0xea
21 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。

```

1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     flag := make(chan struct{}) // 比 chan bool省内存
10    go func() {
11        time.Sleep(3 * time.Second)
12        flag <- struct{}{} // 无数据成员的结构体实例
13    }()
14    fmt.Printf("终于等到了信号，%T, %[1]v", <-flag)
15 }

```

通道多路复用

Go语言提供了select来监听多个channel。

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     count := make(chan int, 4)
10    fin := make(chan bool)
11
12    go func() {
13        defer func() { fin <- true }()
14        for i := 0; i < 10; i++ {
15            count <- i
16            time.Sleep(1 * time.Second)
17        }
18    }()
19
20    for {
21        select { // 监听多路通道
22            case n := <-count:
23                fmt.Println("count =", n)
24            case <-fin:
25                fmt.Println("结束")
26                goto END
27            // default:
28            //     fmt.Println("缺省，以上都不匹配就进入缺省")
29        }
30    }
31    END:
32    fmt.Println("~~~~~")
33 }
```

```
1 package main
2
3 import (
4     "fmt"
5     "time"
6 )
7
8 func main() {
9     count := make(chan int, 4)
10    fin := make(chan bool)
11    global := 1000
12    t1 := time.NewTicker(time.Second)
13    t2 := time.NewTicker(5 * time.Second)
14
15    go func() {
16        defer func() { fin <- true }()
17    }
```



```

17         for i := 0; i < 4; i++ {
18             count <- i
19             // time.Sleep(1 * time.Second)
20         }
21     }()
22
23     time.Sleep(1 * time.Second)
24     fmt.Println(len(count), "~~~@@@")
25     for {
26         select { // 监听多路通道
27             case <-t1.C:
28                 fmt.Println("每隔一秒看看长度", len(count))
29             case <-t2.C:
30                 fmt.Println("每隔5秒取一次", <-count)
31             case count <- global: // 发送数据成功进入通道执行该case
32                 global++
33                 fmt.Println("~~~~~")
34         }
35     }
36 }

```

```

1 // 永远阻塞
2 select {}

```

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？

```

1 package main
2
3 import (
4     "fmt"
5     "runtime"

```

```

6      "time"
7  )
8
9  var count int64 = 0
10
11 func inc() {
12     for i := 0; i < 100000; i++ {
13         // count = count + 1
14         count++
15     }
16 }
17
18 func main() {
19     start := time.Now()
20     inc()
21     inc()
22     inc()
23     inc()
24     inc()
25     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
26     fmt.Println("~~~~~")
27
28     fmt.Printf("执行时长: %d 微秒\n", time.Since(start).Microseconds())
29     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
30     fmt.Printf("count: %d\n", count)
31 }
32
33
34 Go协程数: 1
35 ~~~~~
36 执行时长: 508 微秒
37 Go协程数: 1
38 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？

```

1  package main
2
3  import (
4      "fmt"
5      "runtime"
6      "sync"
7      "time"
8  )
9
10 var wg sync.WaitGroup
11 var count int64 = 0
12
13 func inc() {
14     defer wg.Done()
15     for i := 0; i < 100000; i++ {
16         // count = count + 1
17         count++

```

```

18     }
19 }
20
21 func main() {
22     start := time.Now()
23     wg.Add(5)
24     for i := 0; i < 5; i++ {
25         go inc()
26     }
27     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
28     fmt.Println("~~~~~")
29     wg.Wait()
30     fmt.Printf("执行时长: %d 微秒\n", time.Since(start).Microseconds())
31     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
32     fmt.Printf("count: %d\n", count)
33 }
34
35
36 Go协程数: 6
37 ~~~~~
38 执行时长: 1563 微秒
39 Go协程数: 1
40 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

```

1 atomic.AddInt64(&count, 1) // count++
2
3 Go协程数: 6
4 ~~~~~
5 执行时长: 7166 微秒
6 Go协程数: 1
7 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

```

1 var wg sync.WaitGroup
2 var mx sync.Mutex
3 var count int64 = 0
4
5 func inc() {
6     defer wg.Done()
7     for i := 0; i < 100000; i++ {
8         // count = count + 1
9         mx.Lock()
10        count++
11        mx.Unlock()
12    }

```

```

13 }
14
15
16 Go协程数: 6
17 ~~~~~
18 执行时长: 16353 微秒
19 Go协程数: 1
20 count: 500000

```

3、是否能使用通道，来同步多个协程

```

1 package main
2
3 import (
4     "fmt"
5     "runtime"
6     "sync"
7     "time"
8 )
9
10 var wg sync.WaitGroup
11 var ch = make(chan int64, 1)
12
13 func inc() {
14     defer wg.Done()
15     for i := 0; i < 100000; i++ {
16         t := <-ch
17         t++
18         ch <- t
19     }
20 }
21
22 func main() {
23     start := time.Now()
24     ch <- 0
25     wg.Add(5)
26     for i := 0; i < 5; i++ {
27         go inc()
28     }
29     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
30     fmt.Println("~~~~~")
31     wg.Wait()
32     fmt.Printf("执行时长: %d 微秒\n", time.Since(start).Microseconds())
33     fmt.Printf("Go协程数: %d\n", runtime.NumGoroutine())
34     fmt.Printf("count: %d\n", <-ch)
35 }
36
37
38 Go协程数: 6
39 ~~~~~
40 执行时长: 131676 微秒
41 Go协程数: 1
42 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。

上例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。

