



下载APP

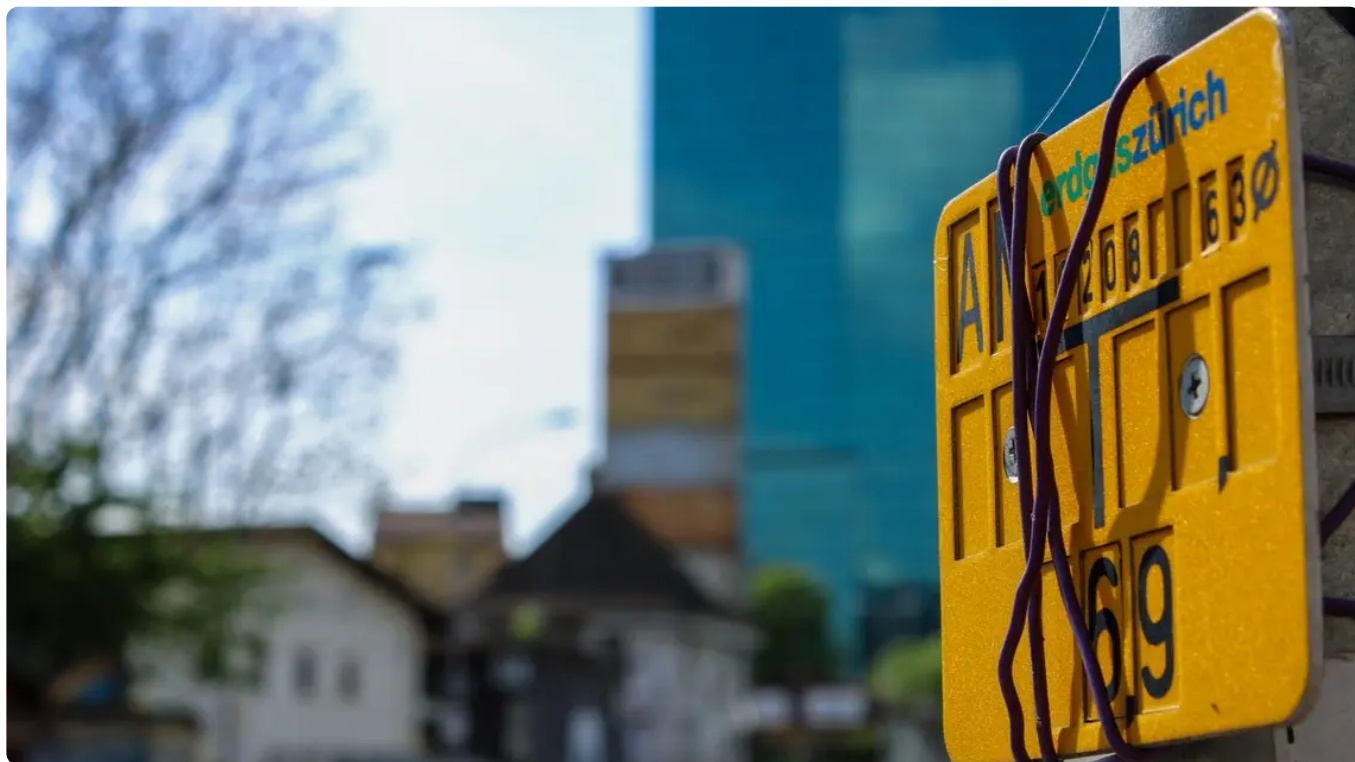


33 | 并发：小channel中蕴含大智慧

2022-01-12 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >



讲述：Tony Bai

时长 25:21 大小 23.23M



你好，我是 Tony Bai。

通过上两节课的学习，我们知道了 Go 语言实现了基于 CSP (Communicating Sequential Processes) 理论的并发方案。

Go 语言的 CSP 模型的实现包含两个主要组成部分：一个是 Goroutine，它是 Go 应
发设计的基本构建与执行单元；另一个就是 channel，它在并发模型中扮演着重要的角
色。channel 既可以用来实现 Goroutine 间的通信，还可以实现 Goroutine 间的同步
它就好比 Go 并发设计这门“武功”的秘籍口诀，可以说，学会在 Go 并发设计时灵活
用 channel，才能说真正掌握了 Go 并发设计的真谛。

领资料



所以，在这一讲中，我们就来系统学习 channel 这一并发原语的基础语法与常见使用方法。

作为一等公民的 channel

Go 对并发的原生支持可不是仅仅停留在口号上的，Go 在语法层面将并发原语 channel 作为一等公民对待。在前面的 [第 21 讲](#) 中我们已经学过“一等公民”这个概念了，如果你记不太清了可以回去复习一下。

那 channel 作为一等公民意味着什么呢？

这意味着我们可以**像使用普通变量那样使用 channel**，比如，定义 channel 类型变量、给 channel 变量赋值、将 channel 作为参数传递给函数 / 方法、将 channel 作为返回值从函数 / 方法中返回，甚至将 channel 发送到其他 channel 中。这就大大简化了 channel 原语的使用，提升了我们开发者在做并发设计和实现时的体验。

创建 channel

和切片、结构体、map 等一样，channel 也是一种复合数据类型。也就是说，我们在声明一个 channel 类型变量时，必须给出其具体的元素类型，比如下面的代码这样：

```
1 var ch chan int
```

[复制代码](#)

这句代码里，我们声明了一个元素为 int 类型的 channel 类型变量 ch。

如果 channel 类型变量在声明时没有被赋予初值，那么它的默认值为 nil。并且，和其他复合数据类型支持使用复合类型字面值作为变量初始值不同，为 channel 类型变量赋初值的唯一方法就是使用 **make** 这个 Go 预定义的函数，比如下面代码：

```
1 ch1 := make(chan int)
2 ch2 := make(chan int, 5)
```

[复制代码](#)


这里，我们声明了两个元素类型为 `int` 的 `channel` 类型变量 `ch1` 和 `ch2`，并给这两个变量赋了初值。但我们看到，两个变量的赋初值操作使用的 `make` 调用的形式有所不同。

第一行我们通过 `make(chan T)` 创建的、元素类型为 `T` 的 `channel` 类型，是**无缓冲 channel**，而第二行中通过带有 `capacity` 参数的 `make(chan T, capacity)` 创建的元素类型为 `T`、缓冲区长度为 `capacity` 的 `channel` 类型，是**带缓冲 channel**。

这两种类型的变量关于发送（`send`）与接收（`receive`）的特性是不同的，我们接下来就基于这两种类型的 `channel`，看看 `channel` 类型变量如何进行发送和接收数据元素。

发送与接收

Go 提供了 `<-` 操作符用于对 `channel` 类型变量进行发送与接收操作：

 复制代码


```
1 ch1 <- 13 // 将整型字面值13发送到无缓冲channel类型变量ch1中
2 n := <- ch1 // 从无缓冲channel类型变量ch1中接收一个整型值存储到整型变量n中
3 ch2 <- 17 // 将整型字面值17发送到带缓冲channel类型变量ch2中
4 m := <- ch2 // 从带缓冲channel类型变量ch2中接收一个整型值存储到整型变量m中
```

这里我要提醒你一句，在理解 `channel` 的发送与接收操作时，你一定要始终牢记：

channel 是用于 Goroutine 间通信的，所以绝大多数对 `channel` 的读写都被分别放在了不同的 Goroutine 中。

现在，我们先来看看无缓冲 `channel` 类型变量（如 `ch1`）的发送与接收。

由于无缓冲 `channel` 的运行时层实现不带有缓冲区，所以 Goroutine 对无缓冲 `channel` 的接收和发送操作是同步的。也就是说，对同一个无缓冲 `channel`，只有对它进行接收操作的 Goroutine 和对它进行发送操作的 Goroutine 都存在的情况下，通信才能得以进行，否则单方面的操作会让对应的 Goroutine 陷入挂起状态，比如下面示例代码：


 复制代码

```
1 func main() {
2     ch1 := make(chan int)
3     ch1 <- 13 // fatal error: all goroutines are asleep - deadlock!
4     n := <-ch1
5     println(n)
}
```

```
6 }
```

在这个示例中，我们创建了一个无缓冲的 channel 类型变量 ch1，对 ch1 的读写都放在了一个 Goroutine 中。

运行这个示例，我们就会得到 fatal error，提示我们所有 Goroutine 都处于休眠状态，程序处于死锁状态。要想解除这种错误状态，我们只需要将接收操作，或者发送操作放到另外一个 Goroutine 中就可以了，比如下面代码：

 复制代码

```
1 func main() {
2     ch1 := make(chan int)
3     go func() {
4         ch1 <- 13 // 将发送操作放入一个新goroutine中执行
5     }()
6     n := <-ch1
7     println(n)
8 }
```

由此，我们可以得出结论：**对无缓冲 channel 类型的发送与接收操作，一定要放在两个不同的 Goroutine 中进行，否则会导致 deadlock。**

接下来，我们再来看看带缓冲 channel 的发送与接收操作。

和无缓冲 channel 相反，带缓冲 channel 的运行层实现带有缓冲区，因此，对带缓冲 channel 的发送操作在缓冲区未满、接收操作在缓冲区非空的情况下是**异步**的（发送或接收不需要阻塞等待）。

也就是说，对一个带缓冲 channel 来说，在缓冲区未满的情况下，对它进行发送操作的 Goroutine 并不会阻塞挂起；在缓冲区有数据的情况下，对它进行接收操作的 Goroutine 也不会阻塞挂起。

但当缓冲区满了的情况下，对它进行发送操作的 Goroutine 就会阻塞挂起；当缓冲区为空的情况下，对它进行接收操作的 Goroutine 也会阻塞挂起。

如果光看文字还不是很理解，你可以再看看下面几个关于带缓冲 channel 的操作的例子：

[复制代码](#)

```
1 ch2 := make(chan int, 1)
2 n := <-ch2 // 由于此时ch2的缓冲区中无数据，因此对其进行接收操作将导致goroutine挂起
3
4 ch3 := make(chan int, 1)
5 ch3 <- 17 // 向ch3发送一个整型数17
6 ch3 <- 27 // 由于此时ch3中缓冲区已满，再向ch3发送数据也将导致goroutine挂起
```

也正是因为带缓冲 channel 与无缓冲 channel 在发送与接收行为上的差异，在具体使用上，它们有各自的“用武之地”，这个我们等会再细说，现在我们先继续把 channel 的基本语法讲完。

使用操作符<-，我们还可以声明**只发送 channel 类型**（send-only）和**只接收 channel 类型**（recv-only），我们接着看下面这个例子：

[复制代码](#)

```
1 ch1 := make(chan<- int, 1) // 只发送channel类型
2 ch2 := make(<-chan int, 1) // 只接收channel类型
3
4 <-ch1 // invalid operation: <-ch1 (receive from send-only type chan<- in
5 ch2 <- 13 // invalid operation: ch2 <- 13 (send to receive-only type <-chan
```

你可以从这个例子中看到，试图从一个只发送 channel 类型变量中接收数据，或者向一个只接收 channel 类型发送数据，都会导致编译错误。通常只发送 channel 类型和只接收 channel 类型，会被用作函数的参数类型或返回值，用于限制对 channel 内的操作，或者是明确可对 channel 进行的操作的类型，比如下面这个例子：

[复制代码](#)

```
1 func produce(ch chan<- int) {
2     for i := 0; i < 10; i++ {
3         ch <- i + 1
4         time.Sleep(time.Second)
5     }
6     close(ch)
7 }
8
```

```
9 func consume(ch <-chan int) {
10     for n := range ch {
11         println(n)
12     }
13 }
14
15 func main() {
16     ch := make(chan int, 5)
17     var wg sync.WaitGroup
18     wg.Add(2)
19     go func() {
20         produce(ch)
21         wg.Done()
22     }()
23
24     go func() {
25         consume(ch)
26         wg.Done()
27     }()
28
29     wg.Wait()
30 }
```

在这个例子中，我们启动了两个 Goroutine，分别代表生产者（produce）与消费者（consume）。生产者只能向 channel 中发送数据，我们使用 `chan int` 作为 produce 函数的参数类型；消费者只能从 channel 中接收数据，我们使用 `<-chan int` 作为 consume 函数的参数类型。

在消费者函数 consume 中，我们使用了 for range 循环语句来从 channel 中接收数据，for range 会阻塞在对 channel 的接收操作上，直到 channel 中有数据可接收或 channel 被关闭循环，才会继续向下执行。channel 被关闭后，for range 循环也就结束了。

关闭 channel

在上面的例子中，produce 函数在发送完数据后，调用 Go 内置的 close 函数关闭了 channel。channel 关闭后，所有等待从这个 channel 接收数据的操作都将返回。

这里我们继续看一下采用不同接收语法形式的语句，在 channel 被关闭后的返回值的情况：

```
1 n := <- ch // 当ch被关闭后，n将被赋值为ch元素类型的零值
```

[复制代码](#)


```
2 m, ok := <-ch    // 当ch被关闭后, m将被赋值为ch元素类型的零值, ok值为false
3 for v := range ch { // 当ch被关闭后, for range循环结束
4     ... ..
5 }
```

我们看到, 通过 “comma, ok” 惯用法或 for range 语句, 我们可以准确地判定 channel 是否被关闭。而单纯采用 `n := <-ch` 形式的语句, 我们就无法判定从 `ch` 返回的元素类型零值, 究竟是不是因为 channel 被关闭后才返回的。

另外, 从前面 produce 的示例程序中, 我们也可以看到, channel 是在 produce 函数中被关闭的, 这也是 channel 的一个使用惯例, 那就是**发送端负责关闭 channel**。

这里为什么要在发送端关闭 channel 呢?

这是因为发送端没有像接受端那样的、可以安全判断 channel 是否被关闭了的方法。同时, 一旦向一个已经关闭的 channel 执行发送操作, 这个操作就会引发 panic, 比如下面这个示例:

```
1 ch := make(chan int, 5)
2 close(ch)
3 ch <- 13 // panic: send on closed channel
```

[复制代码](#)

select

当涉及同时对多个 channel 进行操作时, 我们会结合 Go 为 CSP 并发模型提供的另外一个原语 **select**, 一起使用。

通过 select, 我们可以同时并在多个 channel 上进行发送 / 接收操作:

```
1 select {
2 case x := <-ch1:    // 从channel ch1接收数据
3     ... ..
4
5 case y, ok := <-ch2: // 从channel ch2接收数据, 并根据ok值判断ch2是否已经关闭
6     ... ..
7 }
```

[复制代码](#)

```
8 case ch3 <- z:           // 将z值发送到channel ch3中:
9     ... ..
10
11 default:                // 当上面case中的channel通信均无法实施时, 执行该默认分支
12 }
```

当 select 语句中没有 default 分支, 而且所有 case 中的 channel 操作都阻塞了的时候, 整个 select 语句都将被阻塞, 直到某一个 case 上的 channel 变成可发送, 或者某个 case 上的 channel 变成可接收, select 语句才可以继续进行下去。关于 select 语句的妙用, 我们在后面还会细讲, 这里我们先简单了解它的基本语法。

看到这里你应该能感受到, channel 和 select 两种原语的操作都十分简单, 它们都遵循了 Go 语言“追求简单”的设计哲学, 但它们却为 Go 并发程序带来了强大的表达能力。学习了这些基础用法后, 接下来我们再深一层, 看看 Go 并发原语 channel 的一些惯用法。同样地, 这里我们也分成无缓冲 channel 和带缓冲 channel 两种情况分析。

无缓冲 channel 的惯用法

无缓冲 channel 兼具通信和同步特性, 在并发程序中应用颇为广泛。现在来看看几个无缓冲 channel 的典型应用:

第一种用法: 用作信号传递

无缓冲 channel 用作信号传递的时候, 有两种情况, 分别是 1 对 1 通知信号和 1 对 n 通知信号。我们先来分析下 1 对 1 通知信号这种情况。

我们直接来看具体的例子:

```
1 type signal struct{}
2
3 func worker() {
4     println("worker is working...")
5     time.Sleep(1 * time.Second)
6 }
7
8 func spawn(f func()) <-chan signal {
9     c := make(chan signal)
10    go func() {
11        println("worker start to work...")
12        f()
```

[复制代码](#)



```

13         c <- signal(struct{}{})
14     }()
15     return c
16 }
17
18 func main() {
19     println("start a worker...")
20     c := spawn(worker)
21     <-c
22     fmt.Println("worker work done!")
23 }

```

在这个例子中，spawn 函数返回的 channel，被用于承载新 Goroutine 退出的“通知信号”，这个信号专门用作通知 main goroutine。main goroutine 在调用 spawn 函数后一直阻塞在对这个“通知信号”的接收动作上。

我们来运行一下这个例子：


 复制代码

```

1 start a worker...
2 worker start to work...
3 worker is working...
4 worker work done!

```

有些时候，无缓冲 channel 还被用来实现 **1 对 n 的信号通知** 机制。这样的信号通知机制，常被用于协调多个 Goroutine 一起工作，比如下面的例子：

 复制代码

```

1 func worker(i int) {
2     fmt.Printf("worker %d: is working...\n", i)
3     time.Sleep(1 * time.Second)
4     fmt.Printf("worker %d: works done\n", i)
5 }
6
7 func spawnGroup(f func(i int), num int, groupSignal <-chan signal) <-chan sign
8     c := make(chan signal)
9     var wg sync.WaitGroup
10
11     for i := 0; i < num; i++ {
12         wg.Add(1)
13         go func(i int) {
14             <-groupSignal
15             fmt.Printf("worker %d: start to work...\n", i)

```

```
16         f(i)
17         wg.Done()
18     }(i + 1)
19 }
20
21 go func() {
22     wg.Wait()
23     c <- signal(struct{}{})
24 }()
25 return c
26 }
27
28 func main() {
29     fmt.Println("start a group of workers...")
30     groupSignal := make(chan signal)
31     c := spawnGroup(worker, 5, groupSignal)
32     time.Sleep(5 * time.Second)
33     fmt.Println("the group of workers start to work...")
34     close(groupSignal)
35     <-c
36     fmt.Println("the group of workers work done!")
37 }
```

这个例子中，main goroutine 创建了一组 5 个 worker goroutine，这些 Goroutine 启动后会阻塞在名为 groupSignal 的无缓冲 channel 上。main goroutine 通过 close(groupSignal) 向所有 worker goroutine 广播“开始工作”的信号，收到 groupSignal 后，所有 worker goroutine 会“同时”开始工作，就像起跑线上的运动员听到了裁判员发出的起跑信号枪声。

这个例子的运行结果如下：

```
1 start a group of workers...
2 the group of workers start to work...
3 worker 3: start to work...
4 worker 3: is working...
5 worker 4: start to work...
6 worker 4: is working...
7 worker 1: start to work...
8 worker 1: is working...
9 worker 5: start to work...
10 worker 5: is working...
11 worker 2: start to work...
12 worker 2: is working...
13 worker 3: works done
14 worker 4: works done
```

 复制代码

```
15 worker 5: works done
16 worker 1: works done
17 worker 2: works done
18 the group of workers work done!
```

我们可以看到，关闭一个无缓冲 channel 会让所有阻塞在这个 channel 上的接收操作返回，从而实现了一种 1 对 n 的“广播”机制。

第二种用法：用于替代锁机制

无缓冲 channel 具有同步特性，这让它在某些场合可以替代锁，让我们的程序更加清晰，可读性也更好。我们可以对比下两个方案，直观地感受一下。


首先我们看一个传统的、基于“共享内存”+“互斥锁”的 Goroutine 安全的计数器的实现：

[复制代码](#)

```
1 type counter struct {
2     sync.Mutex
3     i int
4 }
5
6 var cter counter
7
8 func Increase() int {
9     cter.Lock()
10    defer cter.Unlock()
11    cter.i++
12    return cter.i
13 }
14
15 func main() {
16     var wg sync.WaitGroup
17
18     for i := 0; i < 10; i++ {
19         wg.Add(1)
20         go func(i int) {
21             v := Increase()
22             fmt.Printf("goroutine-%d: current counter value is %d\n", i, v)
23             wg.Done()
24         }(i)
25     }
26
27     wg.Wait()
28 }
```

在这个示例中，我们使用了一个带有互斥锁保护的全局变量作为计数器，所有要操作计数器的 Goroutine 共享这个全局变量，并在互斥锁的同步下对计数器进行自增操作。

接下来我们再看更符合 Go 设计惯例的实现，也就是使用无缓冲 channel 替代锁后的实现：

 复制代码

```
1  type counter struct {
2      c chan int
3      i int
4  }
5
6  func NewCounter() *counter {
7      cter := &counter{
8          c: make(chan int),
9      }
10     go func() {
11         for {
12             cter.i++
13             cter.c <- cter.i
14         }
15     }()
16     return cter
17 }
18
19 func (cter *counter) Increase() int {
20     return <-cter.c
21 }
22
23 func main() {
24     cter := NewCounter()
25     var wg sync.WaitGroup
26     for i := 0; i < 10; i++ {
27         wg.Add(1)
28         go func(i int) {
29             v := cter.Increase()
30             fmt.Printf("goroutine-%d: current counter value is %d\n", i, v)
31             wg.Done()
32         }(i)
33     }
34     wg.Wait()
35 }
```

在这个实现中，我们将计数器操作全部交给一个独立的 Goroutine 去处理，并通过无缓冲 channel 的同步阻塞特性，实现了计数器的控制。这样其他 Goroutine 通过 Increase 函数试图增加计数器值的动作，实质上就转化为了一次无缓冲 channel 的接收动作。

这种并发设计逻辑更符合 Go 语言所倡导的“**不要通过共享内存来通信，而是通过通信来共享内存**”的原则。

运行这个示例，我们可以得出与互斥锁方案相同的结果：

 复制代码

```
1 goroutine-9: current counter value is 10
2 goroutine-0: current counter value is 1
3 goroutine-6: current counter value is 7
4 goroutine-2: current counter value is 3
5 goroutine-8: current counter value is 9
6 goroutine-4: current counter value is 5
7 goroutine-5: current counter value is 6
8 goroutine-1: current counter value is 2
9 goroutine-7: current counter value is 8
10 goroutine-3: current counter value is 4
```

带缓冲 channel 的惯用法

带缓冲的 channel 与无缓冲的 channel 的最大不同之处，就在于它的**异步性**。也就是说，对一个带缓冲 channel，在缓冲区未满的情况下，对它进行发送操作的 Goroutine 不会阻塞挂起；在缓冲区有数据的情况下，对它进行接收操作的 Goroutine 也不会阻塞挂起。

这种特性让带缓冲的 channel 有着与无缓冲 channel 不同的应用场合。接下来我们一个个来分析。

第一种用法：用作消息队列

channel 经常被 Go 初学者视为在多个 Goroutine 之间通信的消息队列，这是因为，channel 的原生特性与我们认知中的消息队列十分相似，包括 Goroutine 安全、有 FIFO (first-in, first out) 保证等。


其实，和无缓冲 channel 更多用于信号 / 事件管道相比，可自行设置容量、异步收发的带缓冲 channel 更适合被用作为消息队列，并且，带缓冲 channel 在数据收发的性能上要明

显好于无缓冲 channel。

我们可以通过对 channel 读写的基本测试来印证这一点。下面是一些关于无缓冲 channel 和带缓冲 channel 收发性能测试的结果 (Go 1.17, MacBook Pro 8 核)。基准测试的代码比较多, 我就不全部贴出来了, 你可以到 [这里](#) 下载。

单接收单发送性能的基准测试

我们先来看看针对一个 channel 只有一个发送 Goroutine 和一个接收 Goroutine 的情况, 两种 channel 的收发性能比对数据:

 复制代码

```
1 // 无缓冲channel
2 // go-channel-operation-benchmark/unbuffered-chan
3
4 $go test -bench . one_to_one_test.go
5 goos: darwin
6 goarch: amd64
7 cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
8 BenchmarkUnbufferedChan1To1Send-8      6037778      199.7 ns/op
9 BenchmarkUnbufferedChan1To1Recv-8      6286850      194.5 ns/op
10 PASS
11 ok      command-line-arguments  2.833s
12
13 // 带缓冲channel
14 // go-channel-operation-benchmark/buffered-chan
15
16 $go test -bench . one_to_one_cap_10_test.go
17 goos: darwin
18 goarch: amd64
19 cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
20 BenchmarkBufferedChan1To1SendCap10-8    17089879     66.16 ns/op
21 BenchmarkBufferedChan1To1RecvCap10-8    18043450     65.57 ns/op
22 PASS
23 ok      command-line-arguments  2.460s
```

然后将 channel 的缓存由 10 改为 100, 再看看带缓冲 channel 的 1 对 1 基准测试结果:

 复制代码

```
1 $go test -bench . one_to_one_cap_100_test.go
2 goos: darwin
3 goarch: amd64
```



```
4  cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
5  BenchmarkBufferedChan1To1SendCap100-8      23089318      53.06 ns/op
6  BenchmarkBufferedChan1To1RecvCap100-8      23474095      51.33 ns/op
7  PASS
8  ok      command-line-arguments    2.542s
```

多接收多发送性能基准测试

我们再来看看，针对一个 channel 有多个发送 Goroutine 和多个接收 Goroutine 的情况，两种 channel 的收发性能比对数据（这里建立 10 个发送 Goroutine 和 10 个接收 Goroutine）：

[复制代码](#)

```
1  // 无缓冲channel
2  // go-channel-operation-benchmark/unbuffered-chan
3
4  $go test -bench . multi_to_multi_test.go
5  goos: darwin
6  goarch: amd64
7  cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
8  BenchmarkUnbufferedChanNTToNSend-8          293930        3779 ns/op
9  BenchmarkUnbufferedChanNTToNRecv-8         280904        4190 ns/op
10 PASS
11 ok      command-line-arguments    2.387s
12
13 // 带缓冲channel
14 // go-channel-operation-benchmark/buffered-chan
15
16 $go test -bench . multi_to_multi_cap_10_test.go
17 goos: darwin
18 goarch: amd64
19 cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
20 BenchmarkBufferedChanNTToNSendCap10-8       736540        1609 ns/op
21 BenchmarkBufferedChanNTToNRecvCap10-8      795416        1616 ns/op
22 PASS
23 ok      command-line-arguments    2.514s
```

这里我们也将 channel 的缓存由 10 改为 100 后，看看带缓冲 channel 的多对多基准测试结果：

[复制代码](#)

```
1  $go test -bench . multi_to_multi_cap_100_test.go
2  goos: darwin
3  goarch: amd64
4  cpu: Intel(R) Core(TM) i5-8257U CPU @ 1.40GHz
```

```
5 BenchmarkBufferedChanNTToNSendCap100-8      1236453      966.4 ns/op
6 BenchmarkBufferedChanNTToNRecvCap100-8      1279766      969.4 ns/op
7 PASS
8 ok      command-line-arguments    4.309s
```

综合前面这些结果数据，我们可以得出几个初步结论：

无论是 1 收 1 发还是多收多发，带缓冲 channel 的收发性能都要好于无缓冲 channel；

对于带缓冲 channel 而言，发送与接收的 Goroutine 数量越多，收发性能会有所下降；

对于带缓冲 channel 而言，选择适当容量会在一定程度上提升收发性能。

不过你要注意的是，Go 支持 channel 的初衷是将它作为 Goroutine 间的通信手段，它并不是专门用于消息队列场景的。如果你的项目需要专业消息队列的功能特性，比如支持优先级、支持权重、支持离线持久化等，那么 channel 就不合适了，可以使用第三方的专业的消息队列实现。

第二种用法：用作计数信号量 (counting semaphore)

Go 并发设计的一个惯用法，就是将带缓冲 channel 用作计数信号量 (counting semaphore)。带缓冲 channel 中的当前数据个数代表的是，当前同时处于活动状态（处理业务）的 Goroutine 的数量，而带缓冲 channel 的容量 (capacity)，就代表了允许同时处于活动状态的 Goroutine 的最大数量。向带缓冲 channel 的一个发送操作表示获取一个信号量，而从 channel 的一个接收操作则表示释放一个信号量。

这里我们来看一个将带缓冲 channel 用作计数信号量的例子：

```
1 var active = make(chan struct{}, 3)
2 var jobs = make(chan int, 10)
3
4 func main() {
5     go func() {
6         for i := 0; i < 8; i++ {
7             jobs <- (i + 1)
8         }
9         close(jobs)
```


[复制代码](#)

```
10     }()
11
12     var wg sync.WaitGroup
13
14     for j := range jobs {
15         wg.Add(1)
16         go func(j int) {
17             active <- struct{}{}
18             log.Printf("handle job: %d\n", j)
19             time.Sleep(2 * time.Second)
20             <-active
21             wg.Done()
22         }(j)
23     }
24     wg.Wait()
25 }
```

我们看到，这个示例创建了一组 Goroutine 来处理 job，同一时间允许最多 3 个 Goroutine 处于活动状态。

为了达成这一目标，我们看到这个示例使用了一个容量（capacity）为 3 的带缓冲 channel: **active** 作为计数信号量，这意味着允许同时处于**活动状态**的最大 Goroutine 数量为 3。

我们运行一下这个示例：

 复制代码

```
1 2022/01/02 10:08:55 handle job: 1
2 2022/01/02 10:08:55 handle job: 4
3 2022/01/02 10:08:55 handle job: 8
4 2022/01/02 10:08:57 handle job: 5
5 2022/01/02 10:08:57 handle job: 7
6 2022/01/02 10:08:57 handle job: 6
7 2022/01/02 10:08:59 handle job: 3
8 2022/01/02 10:08:59 handle job: 2
```

从示例运行结果中的时间戳中，我们可以看到，虽然我们创建了很多 Goroutine，但由于计数信号量的存在，同一时间内处理活动状态（正在处理 job）的 Goroutine 的数量最多为 3 个。

len(channel) 的应用

len 是 Go 语言的一个内置函数，它支持接收数组、切片、map、字符串和 channel 类型的参数，并返回对应类型的“长度”，也就是一个整型值。

针对 channel `ch` 的类型不同，`len(ch)` 有如下两种语义：

当 `ch` 为无缓冲 channel 时，`len(ch)` 总是返回 0；

当 `ch` 为带缓冲 channel 时，`len(ch)` 返回当前 channel `ch` 中尚未被读取的元素个数。

这样一来，针对带缓冲 channel 的 `len` 调用似乎才是有意义的。那我们是否可以使用 `len` 函数来实现带缓冲 channel 的“判满”、“判有”和“判空”逻辑呢？就像下面示例中伪代码这样：

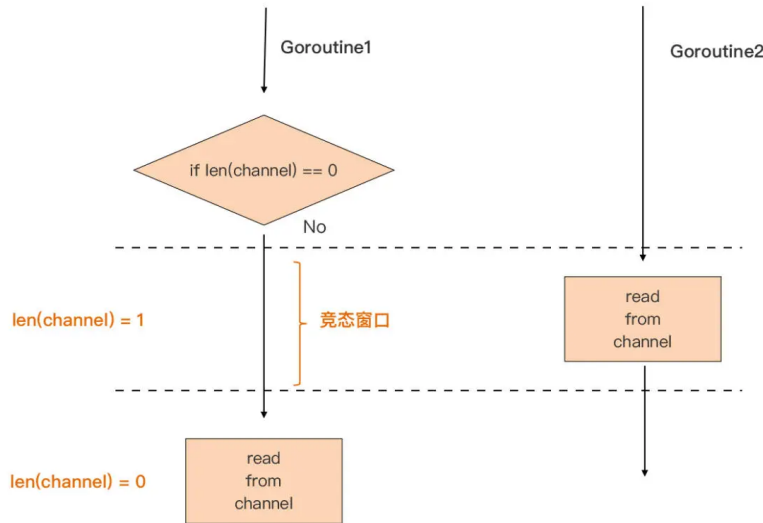
 复制代码

```
1 var ch chan T = make(chan T, capacity)
2
3 // 判空
4 if len(ch) == 0 {
5     // 此时channel ch空了?
6 }
7
8 // 判有
9 if len(ch) > 0 {
10    // 此时channel ch中有数据?
11 }
12
13 // 判满
14 if len(ch) == cap(ch) {
15    // 此时channel ch满了?
16 }
```

你可以看到，我在上面代码注释的“空了”、“有数据”和“满了”的后面都**打上了问号**。这是为什么呢？

这是因为，channel 原语用于多个 Goroutine 间的通信，一旦多个 Goroutine 共同对 channel 进行收发操作，`len(channel)` 就会在多个 Goroutine 间形成“竞态”。单纯地依靠 `len(channel)` 来判断 channel 中元素状态，是不能保证在后续对 channel 的收发时 channel 状态是不变的。

我们以判空为例看看：



极客时间

从上图可以看到，Goroutine1 使用 `len(channel)` 判空后，就会尝试从 `channel` 中接收数据。但在它真正从 `channel` 读数据之前，另外一个 Goroutine2 已经将数据读了出去，所以，Goroutine1 后面的**读取就会阻塞在 channel 上**，导致后面逻辑的失效。

因此，**为了不阻塞在 channel 上**，常见的方法是将“判空与读取”放在一个“事务”中，将“判满与写入”放在一个“事务”中，而这类“事务”我们可以通过 `select` 实现。我们来看下面示例：


复制代码

```
1 func producer(c chan<- int) {
2     var i int = 1
3     for {
4         time.Sleep(2 * time.Second)
5         ok := trySend(c, i)
6         if ok {
7             fmt.Printf("[producer]: send [%d] to channel\n", i)
8             i++
9             continue
10        }
11        fmt.Printf("[producer]: try send [%d], but channel is full\n", i)
12    }
13 }
14
15 func tryRecv(c <-chan int) (int, bool) {
```

```
16     select {
17     case i := <-c:
18         return i, true
19
20     default:
21         return 0, false
22     }
23 }
24
25 func trySend(c chan<- int, i int) bool {
26     select {
27     case c <- i:
28         return true
29     default:
30         return false
31     }
32 }
33
34 func consumer(c <-chan int) {
35     for {
36         i, ok := tryRecv(c)
37         if !ok {
38             fmt.Println("[consumer]: try to recv from channel, but the channel
39             time.Sleep(1 * time.Second)
40             continue
41         }
42         fmt.Printf("[consumer]: recv [%d] from channel\n", i)
43         if i >= 3 {
44             fmt.Println("[consumer]: exit")
45             return
46         }
47     }
48 }
49
50 func main() {
51     var wg sync.WaitGroup
52     c := make(chan int, 3)
53     wg.Add(2)
54     go func() {
55         producer(c)
56         wg.Done()
57     }()
58
59     go func() {
60         consumer(c)
61         wg.Done()
62     }()
63
64     wg.Wait()
65 }
```


我们看到，由于用到了 select 原语的 default 分支语义，当 channel 空的时候，tryRecv 不会阻塞；当 channel 满的时候，trySend 也不会阻塞。

这个示例的运行结果也证明了这一点，无论是使用 tryRecv 的 consumer 还是使用 trySend 的 producer 都不会阻塞：

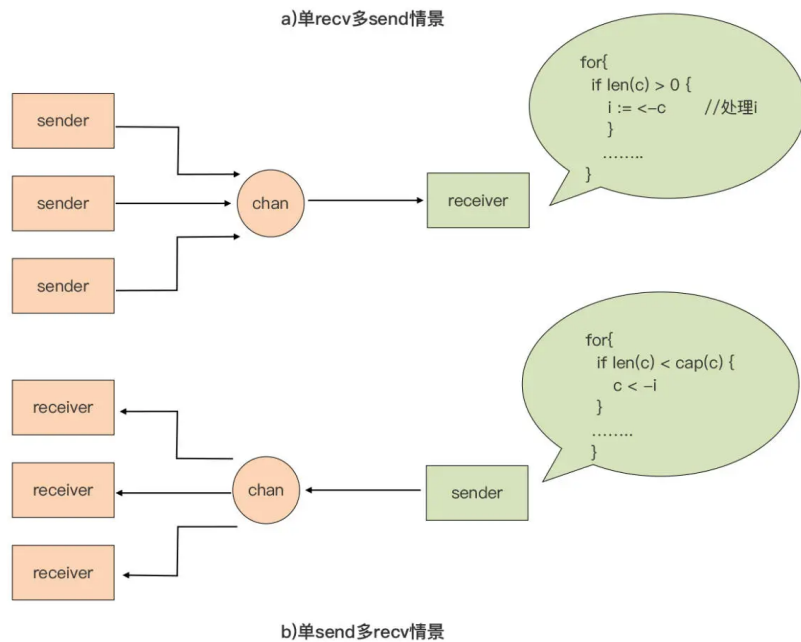
 复制代码

```
1 [consumer]: try to recv from channel, but the channel is empty
2 [consumer]: try to recv from channel, but the channel is empty
3 [producer]: send [1] to channel
4 [consumer]: recv [1] from channel
5 [consumer]: try to recv from channel, but the channel is empty
6 [consumer]: try to recv from channel, but the channel is empty
7 [producer]: send [2] to channel
8 [consumer]: recv [2] from channel
9 [consumer]: try to recv from channel, but the channel is empty
10 [consumer]: try to recv from channel, but the channel is empty
11 [producer]: send [3] to channel
12 [consumer]: recv [3] from channel
13 [consumer]: exit
14 [producer]: send [4] to channel
15 [producer]: send [5] to channel
16 [producer]: send [6] to channel
17 [producer]: try send [7], but channel is full
18 [producer]: try send [7], but channel is full
19 [producer]: try send [7], but channel is full
20 ... ..
```

这种方法适用于大多数场合，但是这种方法有一个“问题”，那就是它改变了 channel 的状态，会让 channel 接收了一个元素或发送一个元素到 channel。

有些时候我们不想这么做，我们想在不改变 channel 状态的前提下，单纯地侦测 channel 的状态，而又不会因 channel 满或空阻塞在 channel 上。但很遗憾，目前没有一种方法可以在实现这样的功能的同时，适用于所有场合。

但是在特定的场景下，我们可以用 len(channel) 来实现。比如下面这两种场景：



上图中的情景 (a) 是一个“多发送单接收”的场景，也就是有多个发送者，但**有且只有一个接收者**。在这样的场景下，我们可以在接收 goroutine 中使用 `len(channel)` 是否大于 0 来判断是否 channel 中有数据需要接收。

而情景 (b) 呢，是一个“多接收单发送”的场景，也就是有多个接收者，但**有且只有一个发送者**。在这样的场景下，我们可以在发送 Goroutine 中使用 `len(channel)` 是否小于 `cap(channel)` 来判断是否可以执行向 channel 的发送操作。

nil channel 的妙用


如果一个 channel 类型变量的值为 nil，我们称它为 **nil channel**。nil channel 有一个特性，那就是对 nil channel 的读写都会发生阻塞。比如下面示例代码：

复制代码

```
1 func main() {
2     var c chan int
3     <-c //阻塞
4 }
5
6 或者：
7
8 func main() {
9     var c chan int
10    c<-1 //阻塞
11 }
```

你会看到，无论上面的哪段代码被执行，main goroutine 都会阻塞在对 nil channel 的操作上。

不过，nil channel 的这个特性可不是一无是处，有些时候应用 nil channel 的这个特性可以得到事半功倍的效果。我们来看一个例子：

 复制代码

```
1 func main() {
2     ch1, ch2 := make(chan int), make(chan int)
3     go func() {
4         time.Sleep(time.Second * 5)
5         ch1 <- 5
6         close(ch1)
7     }()
8
9     go func() {
10        time.Sleep(time.Second * 7)
11        ch2 <- 7
12        close(ch2)
13    }()
14
15    var ok1, ok2 bool
16    for {
17        select {
18            case x := <-ch1:
19                ok1 = true
20                fmt.Println(x)
21            case x := <-ch2:
22                ok2 = true
23                fmt.Println(x)
24        }
25
26        if ok1 && ok2 {
27            break
28        }
29    }
30    fmt.Println("program end")
31 }
```

在这个示例中，我们期望程序在接收完 ch1 和 ch2 两个 channel 上的数据后就退出。但实际的运行情况却是这样的：

```
1 5
2 0
3 0
4 0
5 ... .. //循环输出0
6 7
7 program end
```

[复制代码](#)

我们原本期望上面这个在依次输出 5 和 7 两个数字后退出，但实际运行的输出结果却是在输出 5 之后，程序输出了许多的 0 值，之后才输出 7 并退出。

这是怎么回事呢？我们简单分析一下这段代码的运行过程：

前 5s，select 一直处于阻塞状态；

第 5s，ch1 返回一个 5 后被 close，select 语句的case x := <-ch1这个分支被选出执行，程序输出 5，并回到 for 循环并重新 select；

由于 ch1 被关闭，从一个已关闭的 channel 接收数据将永远不会被阻塞，于是新一轮 select 又把case x := <-ch1这个分支选出并执行。由于 ch1 处于关闭状态，从这个 channel 获取数据，我们会得到这个 channel 对应类型的零值，这里就是 0。于是程序再次输出 0；程序按这个逻辑循环执行，一直输出 0 值；

2s 后，ch2 被写入了一个数值 7。这样在某一轮 select 的过程中，分支case x := <-ch2被选中得以执行，程序输出 7 之后满足退出条件，于是程序终止。

那我们可以怎么改进一下这个程序，让它能按照我们的预期输出呢？

是时候让 nil channel 登场了！用 nil channel 改进后的示例代码是这样的：

```
1 func main() {
2     ch1, ch2 := make(chan int), make(chan int)
3     go func() {
4         time.Sleep(time.Second * 5)
5         ch1 <- 5
6         close(ch1)
7     }()
8
9     go func() {
10        time.Sleep(time.Second * 7)
```

[复制代码](#)

```
11         ch2 <- 7
12         close(ch2)
13     }()
14
15     for {
16         select {
17             case x, ok := <-ch1:
18                 if !ok {
19                     ch1 = nil
20                 } else {
21                     fmt.Println(x)
22                 }
23             case x, ok := <-ch2:
24                 if !ok {
25                     ch2 = nil
26                 } else {
27                     fmt.Println(x)
28                 }
29         }
30         if ch1 == nil && ch2 == nil {
31             break
32         }
33     }
34     fmt.Println("program end")
35 }
```

这里，改进后的示例程序的最关键的一个变化，就是在判断 ch1 或 ch2 被关闭后，显式地将 ch1 或 ch2 置为 nil。

而我们前面已经知道了，**对一个 nil channel 执行获取操作，这个操作将阻塞**。于是，这里已经被置为 nil 的 c1 或 c2 的分支，将再也不会被 select 选中执行。

改进后的示例的运行结果如下，与我们预期相符：

```
1 5
2 7
3 program end
```

[复制代码](#)

与 select 结合使用的一些惯用法

channel 和 select 的结合使用能形成强大的表达能力，我们在前面的例子中已经或多或少见识过了。这里我再强调几种 channel 与 select 结合的惯用法。

第一种用法：利用 default 分支避免阻塞

select 语句的 default 分支的语义，就是在其他非 default 分支因通信未就绪，而无法被选择的时候执行的，这就给 default 分支赋予了一种“避免阻塞”的特性。

其实在前面的“**len(channel) 的应用**”小节的例子中，我们就已经用到了“利用 default 分支”实现的trySend和tryRecv两个函数：

[复制代码](#)

```
1 func tryRecv(c <-chan int) (int, bool) {
2     select {
3     case i := <-c:
4         return i, true
5
6     default: // channel为空
7         return 0, false
8     }
9 }
10
11 func trySend(c chan<- int, i int) bool {
12     select {
13     case c <- i:
14         return true
15     default: // channel满了
16         return false
17     }
18 }
```

而且，无论是无缓冲 channel 还是带缓冲 channel，这两个函数都能适用，并且不会阻塞在空 channel 或元素个数已经达到容量的 channel 上。

在 Go 标准库中，这个惯用法也有应用，比如：

[复制代码](#)

```
1 // $GOROOT/src/time/sleep.go
2 func sendTime(c interface{}, seq uintptr) {
3     // 无阻塞的向c发送当前时间
4     select {
5     case c.(chan Time) <- Now():
6     default:
7     }
8 }
```


第二种用法：实现超时机制

带超时机制的 select，是 Go 中常见的一种 select 和 channel 的组合用法。通过超时事件，我们既可以避免长期陷入某种操作的等待中，也可以做一些异常处理工作。

比如，下面示例代码实现了一次具有 30s 超时的 select：

[复制代码](#)

```
1 func worker() {
2     select {
3     case <-c:
4         // ... do some stuff
5     case <-time.After(30 * time.Second):
6         return
7     }
8 }
```

不过，在应用带有超时机制的 select 时，我们要特别注意 **timer 使用后的释放**，尤其在大量创建 timer 的时候。

Go 语言标准库提供的 timer 实际上是由 Go 运行时自行维护的，而不是操作系统级的定时器资源，它的使用代价要比操作系统级的低许多。但即便如此，作为 time.Timer 的使用者，我们也要尽量减少在使用 Timer 时给 Go 运行时和 Go 垃圾回收带来的压力，要及时调用 timer 的 Stop 方法回收 Timer 资源。

第三种用法：实现心跳机制

结合 time 包的 Ticker，我们可以实现带有心跳机制的 select。这种机制让我们可以在监听 channel 的同时，执行一些**周期性的任务**，比如下面这段代码：

[复制代码](#)

```
1 func worker() {
2     heartbeat := time.NewTicker(30 * time.Second)
3     defer heartbeat.Stop()
4     for {
5         select {
6         case <-c:
7             // ... do some stuff
8         case <- heartbeat.C:
9             //... do heartbeat stuff
```

```
10     }  
11     }  
12 }
```

这里我们使用 `time.NewTicker`，创建了一个 `Ticker` 类型实例 `heartbeat`。这个实例包含一个 `channel` 类型的字段 `C`，这个字段会按一定时间间隔持续产生事件，就像“心跳”一样。这样 `for` 循环在 `channel c` 无数据接收时，会每隔特定时间完成一次迭代，然后回到 `for` 循环进行下一次迭代。

和 `timer` 一样，我们在使用完 `ticker` 之后，也不要忘记调用它的 `Stop` 方法，避免心跳事件在 `ticker` 的 `channel`（上面示例中的 `heartbeat.C`）中持续产生。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们系统学习了 Go CSP 并发方案中除 `Goroutine` 之外的另一个重要组成部分：`channel`。Go 为了原生支持并发，把 `channel` 视作一等公民身份，这就大幅提升了开发人员使用 `channel` 进行并发设计和实现的体验。

通过预定义函数 `make`，我们可以创建两类 `channel`：无缓冲 `channel` 与带缓冲的 `channel`。这两类 `channel` 具有不同的收发特性，可以适用于不同的应用场合：无缓冲 `channel` 兼具通信与同步特性，常用于作为信号通知或替代同步锁；而带缓冲 `channel` 的异步性，让它更适合用来实现基于内存的消息队列、计数信号量等。

此外，你也要牢记值为 `nil` 的 `channel` 的阻塞特性，有些时候它也能帮上大忙。而面对已关闭的 `channel` 你也一定要小心，尤其要避免向已关闭的 `channel` 发送数据，那会导致 `panic`。

最后，`select` 是 Go 为了支持同时操作多个 `channel`，而引入的另外一个并发原语，`select` 与 `channel` 有几种常用的固定搭配，你也要好好掌握和理解。

思考题

`channel` 作为 Go 并发设计的重要组成部分，需要你掌握的细节非常多。而且，`channel` 的应用模式也非常多，我们这一讲仅挑了几个常见的模式做了讲解。在日常开发中你还见

过哪些实用的 channel 使用模式呢？欢迎在留言区分享。

如果你觉得有收获，也欢迎你把这节课分享给更多对 Go 并发感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独订阅本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 32 | 并发：聊聊Goroutine调度器的原理

下一篇 34 | 并发：如何使用共享变量？

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取



精选留言 (9)

写留言

**罗杰**

2022-01-12

这节课比较绕，要静下心来好好学习

作者回复:



2

**Calvin**

2022-01-13

请教下，文章中说到“select 这种判空或判满的方法适用于大多数场合，但是这种方法有一个“问题”，那就是它改变了 channel 的状态，会让 channel 接收了一个元素或发送一个元素到 channel。”，怎样理解这句话？为什么“会让 channel 接收了一个元素或发送一个元素到 channel”呢？

展开

共 4 条评论

**Ransang**

2022-01-12

这里为什么要在发送端关闭 channel 呢？

这是因为发送端没有像接受端那样的、可以安全判断 channel 是否被关闭了的方法。

这里表述是不是有问题呀

**Julien**

2022-01-12

请问一下老师，“无缓冲 channel 替代锁后的实现”那个例子，NewCounter函数里的go func 是什么时候执行的啊？看到NewCounter只调用了一次。

共 1 条评论

**ibin**

2022-01-12

白老师，你好，下面这段可以模拟close(groupSignal)

```
for i := 0; i < 5; i++ {  
    groupSignal<-signal(struct{}{})  
}
```

为什么close(groupSignal) 可以给每个groupSignal都发送了{}

展开

**用0和1改变自己**

2022-01-12

赞，对channel的认知更进一步

**0mfg**

2022-01-12

白老师好，请教个问题。对于无缓冲通道的结论，“对无缓冲 channel 类型的发送与接收操作，一定要放在两个不同的 Goroutine 中进行，否则会导致 deadlock。”尝试了如下写法，发送在主goroutine，接收在新goroutine发现还是deadlock，请问具体原因是啥？谢谢

package main...

展开 ∨

作者回复: main goroutine在执行ch <-13时就阻塞住了。还没执行到创建下面那个goroutine呢。



共 3 条评论 >

**bearlu**

2022-01-12

谢谢老师。总结得非常好。虽然学了很久go，但是没看到总结得如此好的

作者回复: 👍

**王哲**

2022-01-12

厉害了，清晰易懂！

