



下载APP



07 | Cond: 条件变量的实现机制及避坑指南

2020-10-26 鬼岳攀

Go 并发编程实战课

[进入课程 >](#)



讲述：安晓辉

时长 17:08 大小 15.70M



你好，我是鸟窝。

在写 Go 程序之前，我曾经写了 10 多年的 Java 程序，也面试过不少 Java 程序员。在 Java 面试中，经常被问到的一个知识点就是等待 / 通知 (wait/notify) 机制。面试官经常会这样考察候选人：请实现一个限定容量的队列 (queue)，当队列满或者空的时候，利用等待 / 通知机制实现阻塞或者唤醒。

在 Go 中，也可以实现一个类似的限定容量的队列，而且实现起来也比较简单，只要用条件变量 (Cond) 并发原语就可以。Cond 并发原语相对来说不是那么常用，但是在特定场景使用会事半功倍，比如你需要在唤醒一个或者所有的等待者做一些检查操作的时候。

那么今天这一讲，我们就学习下 Cond 这个并发原语。

Go 标准库的 Cond

Go 标准库提供 Cond 原语的目的是，为等待 / 通知场景下的并发问题提供支持。Cond 通常应用于等待某个条件的一组 goroutine，等条件变为 true 的时候，其中一个 goroutine 或者所有的 goroutine 都会被唤醒执行。

顾名思义，Cond 是和某个条件相关，这个条件需要一组 goroutine 协作共同完成，在条件还没有满足的时候，所有等待这个条件的 goroutine 都会被阻塞住，只有这一组 goroutine 通过协作达到了这个条件，等待的 goroutine 才可能继续进行下去。

那这里等待的条件是什么呢？等待的条件，可以是某个变量达到了某个阈值或者某个时间点，也可以是一组变量分别都达到了某个阈值，还可以是某个对象的状态满足了特定的条件。总结来讲，等待的条件是一种可以用来计算结果是 true 还是 false 的条件。

从开发实践上，我们真正使用 Cond 的场景比较少，因为一旦遇到需要使用 Cond 的场景，我们更多地会使用 Channel 的方式（我会在第 12 和第 13 讲展开 Channel 的用法）去实现，因为那才是更地道的 Go 语言的写法，甚至 Go 的开发者有个“把 Cond 从标准库移除”的提议（[issue 21165](#)）。而有的开发者认为，Cond 是唯一难以掌握的 Go 并发原语。至于其中原因，我先卖个关子，到这一讲的后半部分我再和你解释。

今天，这一讲我们就带你仔细地学一学 Cond 这个并发原语吧。

Cond 的基本用法

标准库中的 Cond 并发原语初始化的时候，需要关联一个 Locker 接口的实例，一般我们使用 Mutex 或者 RWMutex。

我们看一下 Cond 的实现：

```

1 type Cond
2 func NewCond(l Locker) *Cond
3 func (c *Cond) Broadcast()
4 func (c *Cond) Signal()
5 func (c *Cond) Wait()
```

 复制代码

首先，Cond 关联的 Locker 实例可以通过 c.L 访问，它内部维护着一个先入先出的等待队列。

然后，我们分别看下它的三个方法 Broadcast、Signal 和 Wait 方法。

Signal 方法，允许调用者 Caller 唤醒一个等待此 Cond 的 goroutine。如果此时没有等待的 goroutine，显然无需通知 waiter；如果 Cond 等待队列中有一个或者多个等待的 goroutine，则需要从等待队列中移除第一个 goroutine 并把它唤醒。在其他编程语言中，比如 Java 语言中，Signal 方法也被叫做 notify 方法。

调用 Signal 方法时，不强求你一定要持有 c.L 的锁。

Broadcast 方法，允许调用者 Caller 唤醒所有等待此 Cond 的 goroutine。如果此时没有等待的 goroutine，显然无需通知 waiter；如果 Cond 等待队列中有一个或者多个等待的 goroutine，则清空所有等待的 goroutine，并全部唤醒。在其他编程语言中，比如 Java 语言中，Broadcast 方法也被叫做 notifyAll 方法。

同样地，调用 Broadcast 方法时，也不强求你一定持有 c.L 的锁。

Wait 方法，会把调用者 Caller 放入 Cond 的等待队列中并阻塞，直到被 Signal 或者 Broadcast 的方法从等待队列中移除并唤醒。

调用 Wait 方法时必须要持有 c.L 的锁。

Go 实现的 sync.Cond 的方法名是 Wait、Signal 和 Broadcast，这是计算机科学中条件变量的通用方法名。比如，C 语言中对应的方法名是 pthread_cond_wait、pthread_cond_signal 和 pthread_cond_broadcast。

知道了 Cond 提供的三个方法后，我们再通过一个百米赛跑开始时的例子，来学习下 **Cond 的使用方法**。10 个运动员进入赛场之后需要先做拉伸活动活动筋骨，向观众和粉丝招手致敬，在自己的赛道上做好准备；等所有的运动员都准备好之后，裁判员才会打响发令枪。

每个运动员做好准备之后，将 ready 加一，表明自己做好准备了，同时调用 Broadcast 方法通知裁判员。因为裁判员只有一个，所以这里可以直接替换成 Signal 方法调用。调用 Broadcast 方法的时候，我们并没有请求 c.L 锁，只是在更改等待变量的时候才使用到了锁。

裁判员会等待运动员都准备好（第 22 行）。虽然每个运动员准备好之后都唤醒了裁判员，但是裁判员被唤醒之后需要检查等待条件是否满足（**运动员都准备好了**）。可以看到，裁判员被唤醒之后一定要检查等待条件，如果条件不满足还是要继续等待。

复制代码

```

1 func main() {
2     c := sync.NewCond(&sync.Mutex{})
3     var ready int
4
5     for i := 0; i < 10; i++ {
6         go func(i int) {
7             time.Sleep(time.Duration(rand.Int63n(10)) * time.Second)
8
9             // 加锁更改等待条件
10            c.L.Lock()
11            ready++
12            c.L.Unlock()
13
14            log.Printf("运动员%d 已准备就绪\n", i)
15            // 广播唤醒所有的等待者
16            c.Broadcast()
17        }(i)
18    }
19
20    c.L.Lock()
21    for ready != 10 {
22        c.Wait()
23        log.Println("裁判员被唤醒一次")
24    }
25    c.L.Unlock()
26
27    //所有的运动员是否就绪
28    log.Println("所有运动员都准备就绪。比赛开始, 3, 2, 1, ....")
29 }
```

你看，Cond 的使用其实没那么简单。它的复杂在于：一，这段代码有时候需要加锁，有时候可以不加；二，Wait 唤醒后需要检查条件；三，条件变量的更改，其实是需要原子操

作或者互斥锁保护的。所以，有的开发者会认为，Cond 是唯一难以掌握的 Go 并发原语。

我们继续看看 Cond 的实现原理。

Cond 的实现原理

其实，Cond 的实现非常简单，或者说复杂的逻辑已经被 Locker 或者 runtime 的等待队列实现了。我们直接看看 Cond 的源码吧。

 复制代码

```
1 type Cond struct {
2     noCopy noCopy
3
4     // 当观察或者修改等待条件的时候需要加锁
5     L Locker
6
7     // 等待队列
8     notify notifyList
9     checker copyChecker
10 }
11
12 func NewCond(l Locker) *Cond {
13     return &Cond{L: l}
14 }
15
16 func (c *Cond) Wait() {
17     c.checker.check()
18     // 增加到等待队列中
19     t := runtime_notifyListAdd(&c.notify)
20     c.L.Unlock()
21     // 阻塞休眠直到被唤醒
22     runtime_notifyListWait(&c.notify, t)
23     c.L.Lock()
24 }
25
26 func (c *Cond) Signal() {
27     c.checker.check()
28     runtime_notifyListNotifyOne(&c.notify)
29 }
30
31 func (c *Cond) Broadcast() {
32     c.checker.check()
33     runtime_notifyListNotifyAll(&c.notify)
34 }
```

这部分源码确实很简单，我来带你学习下其中比较关键的逻辑。

`runtime_notifyListXXX` 是运行时实现的方法，实现了一个等待 / 通知的队列。如果你想深入学习这部分，可以再去看看 `runtime/sema.go` 代码中。

`copyChecker` 是一个辅助结构，可以在运行时检查 `Cond` 是否被复制使用。

`Signal` 和 `Broadcast` 只涉及到 `notifyList` 数据结构，不涉及到锁。

`Wait` 把调用者加入到等待队列时会释放锁，在被唤醒之后还会请求锁。在阻塞休眠期间，调用者是不持有锁的，这样能让其他 `goroutine` 有机会检查或者更新等待变量。

我们继续看看使用 `Cond` 常见的两个错误，一个是调用 `Wait` 的时候没有加锁，另一个是没有检查条件是否满足程序就继续执行了。

使用 `Cond` 的 2 个常见错误

我们先看 `Cond` 最常见的使用错误，也就是调用 `Wait` 的时候没有加锁。

以前面百米赛跑的程序为例，在调用 `cond.Wait` 时，把前后的 `Lock/Unlock` 注释掉，如下面的代码中的第 20 行和第 25 行：

 复制代码

```

1 func main() {
2     c := sync.NewCond(&sync.Mutex{})
3     var ready int
4
5     for i := 0; i < 10; i++ {
6         go func(i int) {
7             time.Sleep(time.Duration(rand.Int63n(10)) * time.Second)
8
9             // 加锁更改等待条件
10            c.L.Lock()
11            ready++
12            c.L.Unlock()
13
14            log.Printf("运动员%d 已准备就绪\n", i)
15            // 广播唤醒所有的等待者
16            c.Broadcast()
17        }(i)
18    }

```

```

19     // c.L.Lock()
20     for ready != 10 {
21         c.Wait()
22         log.Println("裁判员被唤醒一次")
23     }
24     // c.L.Unlock()
25
26     //所有的运动员是否就绪
27     log.Println("所有运动员都准备就绪。比赛开始, 3, 2, 1, ....")
28 }
29 }
```

再运行程序，就会报释放未加锁的 panic：

```

smallnest cond2 master ➜ go run main.go
fatal error: sync: unlock of unlocked mutex

goroutine 1 [running]:
runtime.throw(0x4d39c6, 0x1e)
    /usr/local/go/src/runtime/panic.go:1116 +0x72 fp=0xc00006ee80 sp=0xc00006ee50 pc=0x42f4c2
sync.throw(0x4d39c6, 0x1e)
    /usr/local/go/src/runtime/panic.go:1102 +0x35 fp=0xc00006eea0 sp=0xc00006ee80 pc=0x42f445
sync.(*Mutex).unlockSlow(0xc000012068, 0xffffffff)
    /usr/local/go/src/sync/mutex.go:196 +0xd6 fp=0xc00006eec8 sp=0xc00006eea0 pc=0x46c0e6
sync.(*Mutex).Unlock(0xc000012068)
    /usr/local/go/src/sync/mutex.go:190 +0x48 fp=0xc00006eee8 sp=0xc00006eec8 pc=0x46bff8
sync.(*Cond).Wait(0xc00007e040)
    /usr/local/go/src/sync/cond.go:55 +0x87 fp=0xc00006ef20 sp=0xc00006eee8 pc=0x46a637
main.main()
    /mnt/d/gopath/src/github.com/smallnest/dive-to-gosync-workshop/1.basic/cond2/main.go:32 +0xe8 fp=0xc00006ef20 pc=0x49e878
runtime.main()
    /usr/local/go/src/runtime/proc.go:203 +0x212 fp=0xc00006efe0 sp=0xc00006ef88 pc=0x431b42
runtime.goexit()
    /usr/local/go/src/runtime/asm_amd64.s:1373 +0x1 fp=0xc00006efe8 sp=0xc00006efe0 pc=0x45c571
```

出现这个问题的原因在于，cond.Wait 方法的实现是，把当前调用者加入到 notify 队列之中后会释放锁（如果不释放锁，其他 Wait 的调用者就没有机会加入到 notify 队列中了），然后一直等待；等调用者被唤醒之后，又会去争抢这把锁。如果调用 Wait 之前不加锁的话，就有可能 Unlock 一个未加锁的 Locker。所以切记，**调用 cond.Wait 方法之前一定要加锁。**

使用 Cond 的另一个常见错误是，只调用了一次 Wait，没有检查等待条件是否满足，结果条件没满足，程序就继续执行了。出现这个问题的原因在于，误以为 Cond 的使用，就像 WaitGroup 那样调用一下 Wait 方法等待那么简单。比如下面的代码中，把第 21 行和第 24 行注释掉：

 复制代码

```

1 func main() {
2     c := sync.NewCond(&sync.Mutex{})
```

```

3  var ready int
4
5  for i := 0; i < 10; i++ {
6      go func(i int) {
7          time.Sleep(time.Duration(rand.Int63n(10)) * time.Second)
8
9          // 加锁更改等待条件
10         c.L.Lock()
11         ready++
12         c.L.Unlock()
13
14         log.Printf("运动员%d 已准备就绪\n", i)
15         // 广播唤醒所有的等待者
16         c.Broadcast()
17     }(i)
18 }
19
20 c.L.Lock()
21 // for ready != 10 {
22 c.Wait()
23 log.Println("裁判员被唤醒一次")
24 // }
25 c.L.Unlock()
26
27 //所有的运动员是否就绪
28 log.Println("所有运动员都准备就绪。比赛开始, 3, 2, 1, ....")
29 }
```

运行这个程序，你会发现，可能只有几个运动员准备好之后程序就运行完了，而不是我们期望的所有运动员都准备好才进行下一步。原因在于，每一个运动员准备好之后都会唤醒所有的等待者，也就是这里的裁判员，比如第一个运动员准备好后就唤醒了裁判员，结果这个裁判员傻傻地没做任何检查，以为所有的运动员都准备好了，就继续执行了。

所以，我们一定要**记住**，waiter goroutine 被唤醒**不等于**等待条件被满足，只是有 goroutine 把它唤醒了而已，等待条件有可能已经满足了，也有可能不满足，我们需要进一步检查。你也可以理解为，等待者被唤醒，只是得到了一次检查的机会而已。

到这里，我们小结下。如果你想在使用 Cond 的时候避免犯错，只要时刻记住调用 cond.Wait 方法之前一定要加锁，以及 waiter goroutine 被唤醒不等于等待条件被满足这两个知识点。

知名项目中 Cond 的使用

Cond 在实际项目中被使用的机会比较少，原因总结起来有两个。

第一，同样的场景我们会使用其他的并发原语来替代。Go 特有的 Channel 类型，有一个应用很广泛的模式就是通知机制，这个模式使用起来也特别简单。所以很多情况下，我们会使用 Channel 而不是 Cond 实现 wait/notify 机制。

第二，对于简单的 wait/notify 场景，比如等待一组 goroutine 完成之后继续执行余下的代码，我们会使用 WaitGroup 来实现。因为 WaitGroup 的使用方法更简单，而且不容易出错。比如，上面百米赛跑的问题，就可以很方便地使用 WaitGroup 来实现。

所以，我在这一讲开头提到，Cond 的使用场景很少。先前的标准库内部有几个地方使用了 Cond，比如 io/pipe.go 等，后来都被其他的并发原语（比如 Channel）替换了，sync.Cond 的路越走越窄。但是，还是有一批忠实的“粉丝”坚持在使用 Cond，原因在于 Cond 有三点特性是 Channel 无法替代的：

Cond 和一个 Locker 关联，可以利用这个 Locker 对相关的依赖条件更改提供保护。

Cond 可以同时支持 Signal 和 Broadcast 方法，而 Channel 只能同时支持其中一种。

Cond 的 Broadcast 方法可以被重复调用。等待条件再次变成不满足的状态后，我们又可以调用 Broadcast 再次唤醒等待的 goroutine。这也是 Channel 不能支持的，Channel 被 close 掉了之后不支持再 open。

开源项目中使用 sync.Cond 的代码少之又少，包括标准库原先一些使用 Cond 的代码也改成使用 Channel 实现了，所以别说找 Cond 相关的使用 Bug 了，想找到的一个使用的例子都不容易，我找了 Kubernetes 中的一个例子，我们一起看看它是如何使用 Cond 的。

Kubernetes 项目中定义了优先级队列  PriorityQueue 这样一个数据结构，用来实现 Pod 的调用。它内部有三个 Pod 的队列，即 activeQ、podBackoffQ 和 unschedulableQ，其中 activeQ 就是用来调度的活跃队列（heap）。

Pop 方法调用的时候，如果这个队列为空，并且这个队列没有 Close 的话，会调用 Cond 的 Wait 方法等待。

你可以看到，调用 Wait 方法的时候，调用者是持有锁的，并且被唤醒的时候检查等待条件（队列是否为空）。

复制代码

```

1 // 从队列中取出一个元素
2 func (p *PriorityQueue) Pop() (*framework.QueuedPodInfo, error) {
3     p.lock.Lock()
4     defer p.lock.Unlock()
5     for p.activeQ.Len() == 0 { // 如果队列为空
6         if p.closed {
7             return nil, fmt.Errorf(queueClosed)
8         }
9         p.cond.Wait() // 等待, 直到被唤醒
10    }
11    .....
12    return pInfo, err
13 }
14

```

当 activeQ 增加新的元素时，会调用条件变量的 Broadcast 方法，通知被 Pop 阻塞的调用者。

复制代码

```

1 // 增加元素到队列中
2 func (p *PriorityQueue) Add(pod *v1.Pod) error {
3     p.lock.Lock()
4     defer p.lock.Unlock()
5     pInfo := p.newQueuedPodInfo(pod)
6     if err := p.activeQ.Add(pInfo); err != nil { //增加元素到队列中
7         klog.Errorf("Error adding pod %v to the scheduling queue: %v", nsNameFor
8         return err
9     }
10    .....
11    p.cond.Broadcast() //通知其它等待的goroutine, 队列中有元素了
12
13    return nil
14 }

```

这个优先级队列被关闭的时候，也会调用 Broadcast 方法，避免被 Pop 阻塞的调用者永远 hang 住。

复制代码

```

1 func (p *PriorityQueue) Close() {
2     p.lock.Lock()
3     defer p.lock.Unlock()
4     close(p.stop)
5     p.closed = true

```

```
6     p.cond.Broadcast() //关闭时通知等待的goroutine, 避免它们永远等待
7 }
```

你可以思考一下，这里为什么使用 Cond 这个并发原语，能不能换成 Channel 实现呢？

总结

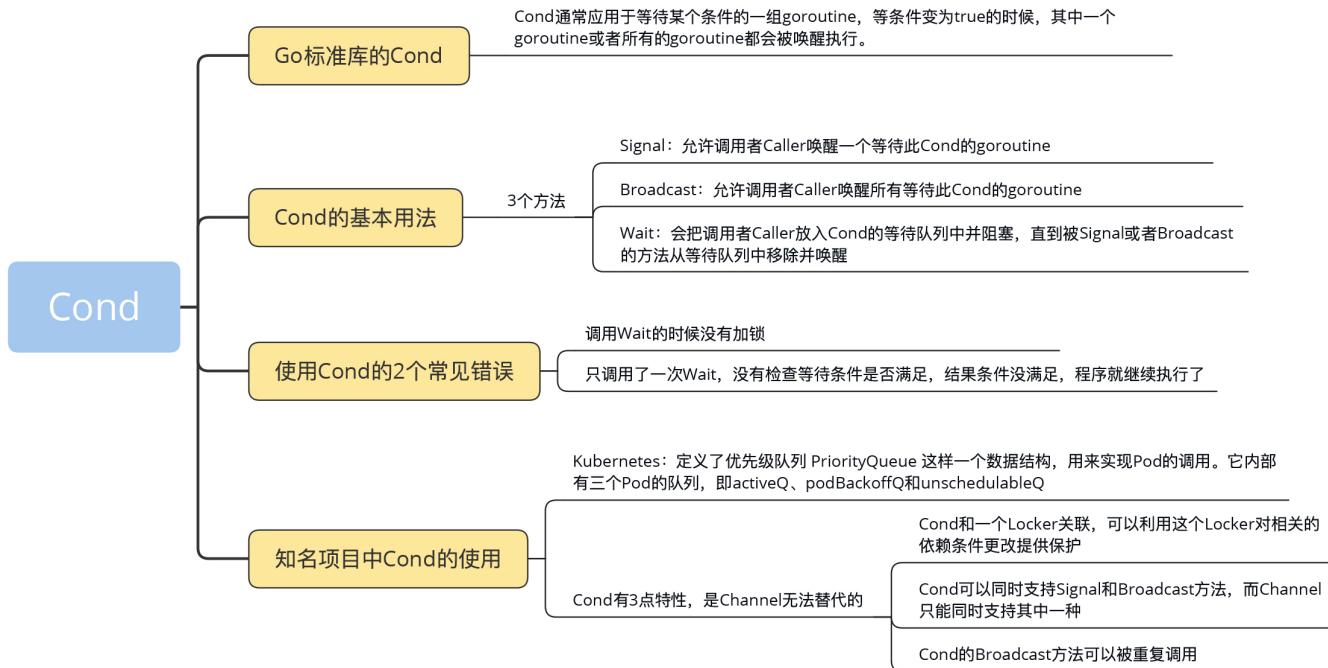
好了，我们来做个总结。

Cond 是为等待 / 通知场景下的并发问题提供支持的。它提供了条件变量的三个基本方法 Signal、Broadcast 和 Wait，为并发的 goroutine 提供等待 / 通知机制。

在实践中，处理等待 / 通知的场景时，我们常常会使用 Channel 替换 Cond，因为 Channel 类型使用起来更简洁，而且不容易出错。但是对于需要重复调用 Broadcast 的场景，比如上面 Kubernetes 的例子，每次往队列中成功增加了元素后就需要调用 Broadcast 通知所有的等待者，使用 Cond 就再合适不过了。

使用 Cond 之所以容易出错，就是 Wait 调用需要加锁，以及被唤醒后一定要检查条件是否真的已经满足。你需要牢记这两点。

虽然我们讲到的百米赛跑的例子，也可以通过 WaitGroup 来实现，但是本质上 WaitGroup 和 Cond 是有区别的：WaitGroup 是主 goroutine 等待确定数量的子 goroutine 完成任务；而 Cond 是等待某个条件满足，这个条件的修改可以被任意多的 goroutine 更新，而且 Cond 的 Wait 不关心也不知道其他 goroutine 的数量，只关心等待条件。而且 Cond 还有单个通知的机制，也就是 Signal 方法。



思考题

1. 一个 Cond 的 waiter 被唤醒的时候, 为什么需要再检查等待条件, 而不是唤醒后进行下一步?
2. 你能否利用 Cond 实现一个容量有限的 queue?

欢迎在留言区写下你的思考和答案, 我们一起交流讨论。如果你觉得有所收获, 也欢迎你把今天的内容分享给你的朋友或同事。

提建议

Go并发编程实战课

鸟窝带你攻克并发编程难题

晁岳攀 (鸟窝)

前微博技术专家

知名微服务框架 rpcx 的作者



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金奖励**。

© 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 06 | WaitGroup：协同等待，任务编排利器

下一篇 08 | Once：一个简约而不简单的并发原语

精选留言 (13)

写留言



Alexdown

2020-10-27

思考题：

1. 唤醒的方式有broadcast，第N个waiter被唤醒后需要检查等待条件，因为不知道前N-1个被唤醒的waiter所作的修改是否使等待条件再次成立。
2. 以下是我实现的一个，有限容量Queue，欢迎讨论！

<https://play.studygolang.com/p/11K2iPVYErn...>

展开 ▼



2



Junes

2020-10-26

1. 是否需要等待，是看业务实现的需求吧：
每个caller会唤醒一个或者所有的waiter
caller和waiter的数量对应是不确定的，如N:M

waiter唤醒后的处理逻辑是自己决定的，比如示例中的ready和队列长度

...

展开 ▾

💬 1

👍 2



那时刻

2020-10-26

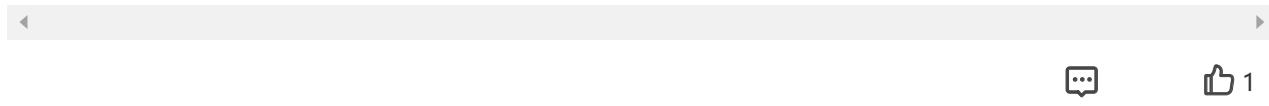
老师，上节课你提到noCopy，是一个辅助的、用来帮助 vet 检查用的类型，而Cond还有一个copyChecker 是一个辅助结构，可以在运行时检查 Cond 是否被复制使用。

nocpoy是静态检查，copyChecker是运行时检查，不是理解是否正确？

...

展开 ▾

作者回复: 是的



myrfy

2020-10-28

还是没有想明白k8s为什么不能用channel通知

close可以实现broadcast的功效，在pop的时候，也是只有一个goroutine可以拿到数据，感觉除了关闭队列之外，不存在需要broadcast的情况。也就是感觉不需要多次broadcast，这样channel应该是满足要求的.....请老师明示

展开 ▾

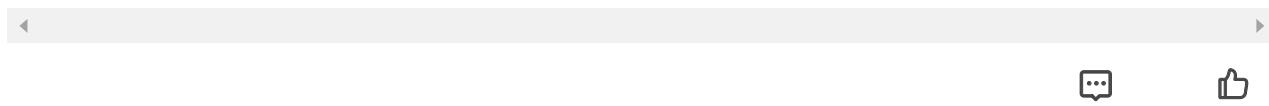


S.S Mr Lin

2020-10-28

每次调用wait前都要加锁，为啥加锁语句放在了fo的外面？第二次wait是不是就没有加锁了？

作者回复: 放在外面的原因是可以利用锁保护共享数据的读写。wait总是需要锁



moooofly

2020-10-27

请教一个问题，nocpoy 是用于 vet 静态检查，copyChecker 是为了运行时检查，都是为

了检查 copy 问题，为啥 Cond 要在两处检查，而 Mutex 只需要一处？

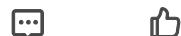
作者回复: 你第一句给出了答案。分别应用不同阶段



 **会飞的大象**
2020-10-27

打卡

展开 ▾



 **约书亚**
2020-10-27

第一题的答案应该应该还包括spurious wakeup的因素

展开 ▾



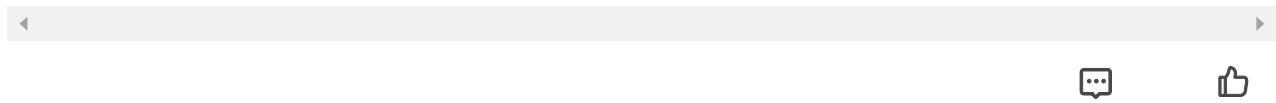
 **syuan**
2020-10-26

Wait 方法，会把调用者 Caller 放入 Cond 的等待队列中并阻塞，直到被 Signal 或者 Broadcast 的方法从等待队列中移除并唤醒。

百米跑代码第22行: c.Wait(),把调用者加入队列阻塞，不理解? for循环一直检查，是把c一直加入阻塞队列吗？还是waiter方法自己生成t对象加入阻塞队列？如果是，同一个c对应t(t := runtime_notifyListAdd(&c.notify)对象唯一吗？

展开 ▾

作者回复: 因为会被唤醒，唤醒之后如果条件还不满足，又会被加入到等待队列中



 **儿戏**
2020-10-26

老师，请教个课程外的问题，使用httputil.ReverseProxy 做方向代理，压测的时候大量报出 http: proxy error: context canceled 这个错误。linux 打开文件数调整了，time_wait 也都优化过了，一直没有找到问题，您的博客也看了还是不能定位，求赐教

展开 ▾

作者回复: 看cpu带宽，看cancel是不是超时导致的，设大超时时间，看pprof

**RRR**

2020-10-26

感觉 Cond 是不是和 Java 中的 notify / wait 机制对应而存在的呢？Golang 和 Java 的异步模式最大的区别是不是就在这里呢？

展开 ▼

作者回复：两者的同步原语都是独立实现的，没有可比性。从功能上来说，cond 和 java notify/wait 机制是类似的，其它语言也有对应的同步原语

**牛晓东**

2020-10-26

咱们本节课有很多图片，链接.....这句话好想写成一个公开的函数，需要的时候调一下 😊
😊

老师真的很细心，每次都提醒以达到最佳效果 ✌️ ✌️ ✌️

展开 ▼

**橙子888**

2020-10-26

打卡。

展开 ▼