

28 | 条件变量sync.Cond（下）

2018-10-15 郝林

Go语言核心36讲

[进入课程 >](#)



讲述：黄洲君

时长 08:23 大小 3.84M



你好，我是郝林，今天我继续分享条件变量 sync.Cond 的内容。我们紧接着上一篇文章的内容进行知识扩展。

问题 1：条件变量的wait方法做了什么？

在了解了条件变量的使用方式之后，你可能会有这么几个疑问。

1. 为什么先要锁定条件变量基于的互斥锁，才能调用它的wait方法？
2. 为什么要用for语句来包裹调用其wait方法的表达式，用if语句不行吗？

这些问题我在面试的时候也经常问。你需要对这个wait方法的内部机制有所了解才能回答上来。

条件变量的`wait`方法主要做了四件事。

1. 把调用它的 `goroutine` (也就是当前的 `goroutine`) 加入到当前条件变量的通知队列中。
2. 解锁当前的条件变量基于的那个互斥锁。
3. 让当前的 `goroutine` 处于等待状态, 等到通知到来时再决定是否唤醒它。此时, 这个 `goroutine` 就会阻塞在调用这个`wait`方法的那行代码上。
4. 如果通知到来并且决定唤醒这个 `goroutine`, 那么就在唤醒它之后重新锁定当前条件变量基于的互斥锁。自此之后, 当前的 `goroutine` 就会继续执行后面的代码了。

你现在知道我刚刚说的第一个疑问的答案了吗?

因为条件变量的`wait`方法在阻塞当前的 `goroutine` 之前, 会解锁它基于的互斥锁, 所以在调用该`wait`方法之前, 我们必须先锁定那个互斥锁, 否则在调用这个`wait`方法时, 就会引发一个不可恢复的 `panic`。

为什么条件变量的`wait`方法要这么做呢? 你可以想象一下, 如果`wait`方法在互斥锁已经锁定的情况下, 阻塞了当前的 `goroutine`, 那么又由谁来解锁呢? 别的 `goroutine` 吗?

先不说这违背了互斥锁的重要使用原则, 即: 成对的锁定和解锁, 就算别的 `goroutine` 可以来解锁, 那万一解锁重复了怎么办? 由此引发的 `panic` 可是无法恢复的。

如果当前的 `goroutine` 无法解锁, 别的 `goroutine` 也都不来解锁, 那么又由谁来进入临界区, 并改变共享资源的状态呢? 只要共享资源的状态不变, 即使当前的 `goroutine` 因收到通知而被唤醒, 也依然会再次执行这个`wait`方法, 并再次被阻塞。

所以说, 如果条件变量的`wait`方法不先解锁互斥锁的话, 那么就只会造成两种后果: 不是当前的程序因 `panic` 而崩溃, 就是相关的 `goroutine` 全面阻塞。

再解释第二个疑问。很显然, `if`语句只会对共享资源的状态检查一次, 而`for`语句却可以做多次检查, 直到这个状态改变为止。那为什么要做多次检查呢?

这主要是为了保险起见。如果一个 `goroutine` 因收到通知而被唤醒, 但却发现共享资源的状态, 依然不符合它的要求, 那么就应该再次调用条件变量的`wait`方法, 并继续等待下次通知的到来。

这种情况是很有可能发生的，具体如下面所示。

1. 有多个 goroutine 在等待共享资源的同一种状态。比如，它们都在等 mailbox 变量的值不为 0 的时候再把它 的值变为 0，这就相当于有多个人在等着我向信箱里放置情报。虽然等待的 goroutine 有多个，但每次成功的 goroutine 却只可能有一个。别忘了，条件变量的 wait 方法会在当前的 goroutine 醒来后先重新锁定那个互斥锁。在成功的 goroutine 最终解锁互斥锁之后，其他的 goroutine 会先后进入临界区，但它们会发现共享资源的状态依然不是它们想要的。这个时候，for 循环就很有必要了。
2. 共享资源可能有的状态不是两个，而是更多。比如，mailbox 变量的可能值不只有 0 和 1，还有 2、3、4。这种情况下，由于状态在每次改变后的结果只可能有一个，所以，在设计合理的前提下，单一的结果一定不可能满足所有 goroutine 的条件。那些未被满足的 goroutine 显然还需要继续等待和检查。
3. 有一种可能，共享资源的状态只有两个，并且每种状态都只有一个 goroutine 在关注，就像我们在主问题当中实现的那个例子那样。不过，即使是这样，使用 for 语句仍然是有必要的。原因是，在一些多 CPU 核心的计算机系统中，即使没有收到条件变量的通知，调用其 wait 方法的 goroutine 也是有可能被唤醒的。这是由计算机硬件层面决定的，即使是操作系统（比如 Linux）本身提供的条件变量也会如此。

综上所述，在包裹条件变量的 wait 方法的时候，我们总是应该使用 for 语句。

好了，到这里，关于条件变量的 wait 方法，我想你知道的应该已经足够多了。

问题 2：条件变量的 Signal 方法和 Broadcast 方法有哪些异同？

条件变量的 Signal 方法和 Broadcast 方法都是被用来发送通知的，不同的是，前者的通知只会唤醒一个因此而等待的 goroutine，而后者的通知却会唤醒所有为此等待的 goroutine。

条件变量的 wait 方法总会把当前的 goroutine 添加到通知队列的队尾，而它的 Signal 方法总会从通知队列的队首开始，查找可被唤醒的 goroutine。所以，因 Signal 方法的通知，而被唤醒的 goroutine 一般都是最早等待的那一个。

这两个方法的行为决定了它们的适用场景。如果你确定只有一个 goroutine 在等待通知，或者只需唤醒任意一个 goroutine 就可以满足要求，那么使用条件变量的 Signal 方法就好了。

否则，使用Broadcast方法总没错，只要你设置好各个 goroutine 所期望的共享资源状态就可以了。

此外，再次强调一下，与Wait方法不同，条件变量的Signal方法和Broadcast方法并不需要在互斥锁的保护下执行。恰恰相反，我们最好在解锁条件变量基于的那个互斥锁之后，再去调用它的这两个方法。这更有利于程序的运行效率。

最后，请注意，条件变量的通知具有即时性。也就是说，如果发送通知的时候没有 goroutine 为此等待，那么该通知就会被直接丢弃。在这之后才开始等待的 goroutine 只可能被后面的通知唤醒。

你可以打开 demo62.go 文件，并仔细观察它与 demo61.go 的不同。尤其是lock变量的类型，以及发送通知的方式。

总结

我们今天主要讲了条件变量，它是基于互斥锁的一种同步工具。在 Go 语言中，我们需要用 sync.NewCond函数来初始化一个sync.Cond类型的条件变量。

sync.NewCond函数需要一个sync.Locker类型的参数值。

*sync.Mutex类型的值以及*sync.RWMutex类型的值都可以满足这个要求。都可以满足这个要求。另外，后者的RLocker方法可以返回这个值中的读锁，也同样可以作为 sync.NewCond函数的参数值，如此就可以生成与读写锁中的读锁对应的条件变量了。

条件变量的wait方法需要在它基于的互斥锁保护下执行，否则就会引发不可恢复的panic。此外，我们最好使用for语句来检查共享资源的状态，并包裹对条件变量的wait方法的调用。

不要用if语句，因为它不能重复地执行“检查状态 - 等待通知 - 被唤醒”的这个流程。重复执行这个流程的原因是，一个“因为等待通知，而被阻塞”的 goroutine，可能会在共享资源的状态不满足其要求的情况下被唤醒。

条件变量的Signal方法只会唤醒一个因等待通知而被阻塞的 goroutine，而它的Broadcast方法却可以唤醒所有为此而等待的 goroutine。后者比前者的适应场景要多得

多。

这两个方法并不需要受到互斥锁的保护，我们也最好不要在解锁互斥锁之前调用它们。还有，条件变量的通知具有即时性。当通知被发送的时候，如果没有任何 goroutine 需要被唤醒，那么该通知就会立即失效。

思考题

`sync.Cond` 类型中的公开字段 `L` 是做什么用的？我们可以在使用条件变量的过程中改变这个字段的值吗？

[戳此查看 Go 语言专栏文章配套详细代码。](#)



GO语言核心36讲

3个月带你通关GO语言

郝林

《Go 并发编程实战》作者
GoHackers 技术社群发起人
前轻松筹大数据负责人

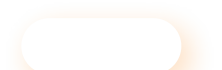


新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 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 27 | 条件变量 `sync.Cond` (上)

下一篇 29 | 原子操作 (上)





云学

2018-10-18

写留言 5

有个疑问，broadcast唤醒所有wait的goroutine，那他们被唤醒时需要去加锁(wait返回)，都能成功吗？



猫王者

2018-10-23

4

“我们最好在解锁条件变量基于的那个互斥锁之后，再去调用它的这两个方法（signal和Broadcast）。这更有利于程序的运行效率” 这个应该如何理解？

我的理解是如果先调用signal方法，然后在unlock解锁，如果在这两个操作中间该线程失去cpu，或者我人为的在signal和unlock之间调用time.Sleep();在另一个等待线程中即使...
展开



RegExp

2019-03-30

1

条件变量的Wait方法在阻塞当前的 goroutine之前，会解锁它基于的互斥锁，那是不是显示调用lock.Lock()的锁也被解锁了呢？

作者回复: 原理上是这样，但是实践中最好不要这样混用同一个锁。



甦

2018-12-13

1

源码问题问一下郝老师，cond wait方法里，多个协程走到c.L.Unlock()那一步不会出问题吗？只有一个协程可以unlock成功，其他协程重复unlock不就panic了吗？



打你

2018-11-10

1

在看了一遍，清楚了

展开



打你

1



2018-11-10

```
lock.Lock()
for mailbox == 1 {
    sendCond.Wait()
}
mailbox = 1...
```

展开 ▾



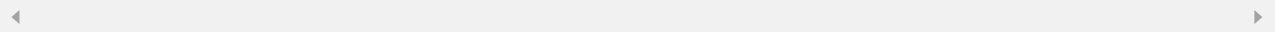
Laughing

2018-10-15



L公开变量代表cond初始化时传递进来的锁，这个锁的状态是可以改变的，但会影响cond对互斥锁的控制。

作者回复: 动这个L之前一定要三思，谨慎些，想想是不是会影响到程序。



Nixus

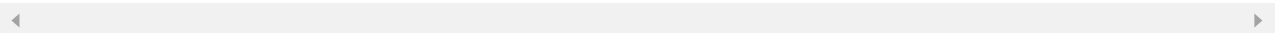
2019-04-15



请教老师一个问题:

demo62.go中, 收信只是读数据, 可是为什么使用RLock()和RUnlock()就报错, 而必须要使用Lock()和Unlock()呢?

作者回复: 你需要展示你的代码。



2019-04-09



@云学，看锁的类型吧。如果是读锁的话，都能够上锁成功。如果是互斥锁的话，只有一个能够上锁成功，其他的继续等待。

展开 ▾



Geek_7b194...

2019-04-03



关于第一个问题，有个疑问，文中说，如果不先加锁，wait方法中解锁，很容易造成wait方法对加锁的锁重复加锁从而造成goroutine阻塞，但是在wait方法调用之前，不是同样也

会造成这个问题吗，也有可能还没有走到wait的时候造成goroutine的阻塞呀，求解答
展开 ∨

作者回复: 我原话不是这么说的吧。我说的是不先加锁的话，调用基于此锁的条件变量的Wait方法时会引发不可恢复的panic。因为Wait方法会先解锁，而解锁还没锁住的锁就会引发panic。



王小勃

2019-01-24



打卡

展开 ∨



奋斗

2019-01-22



Java的notify和notifyall?

展开 ∨

作者回复: 对，两两对应。



Tokumi

2018-12-20



@云学

有个疑问，broadcast唤醒所有wait的goroutine，那他们被唤醒时需要去加锁(wait返回)，都能成功吗？

*sync.Mutex类型的值以及*sync.RWMutex类型的值都可以满足这个要求。...

展开 ∨



jacke

2018-11-17



老师，试了下、signal和broadcast唤醒放在解锁前不会死锁，为什么？解释下、留言里面有2个是同样疑问的



Leon

2018-11-14



老师，我比较疑惑的是多个协程争夺资源，mutex 和RWmutex通过lock和unlock都可以做到同步，为什么一定要用条件变量，编写的这么复杂

展开



heha37

2018-11-10



老师，mailbox使用原子操作是不是就不用for循环去检查了。



卒迹

2018-10-16



老师问个问题：

demo62.go中

```
// sendCond 代表专用于发信的条件变量。
sendCond := sync.NewCond(&lock)
// recvCond 代表专用于收信的条件变量。 ...
```

展开



卒迹

2018-10-16



老师问个问题：

```
var lock sync.RWMutex
sendCond := sync.NewCond(&lock)
recvCond := sync.NewCond(lock.RLocker())
```

我想问的是...

展开



cygnus

2018-10-15



思考题：sync.Cond类型中的公开字段L是用来保存NewCond方法传递进来的互斥锁的，这个锁是条件变量自己控制的，所以我们不能在使用过程中改变这个字段的值，否则可能会导致panic或死锁

作者回复: 没错。改动他要跟谨慎。最好不要改，以防万一。



AskerIve

2018-10-15



老师总结中句子:"sync.Mutex类型的值以及*sync.RWMutex类型的值"sync.Mutex是不是少个"*"哇?

作者回复: 我稍后去看看哈, 谢谢。

