28 | 条件变量sync.Cond （下）

2018-10-15 郝林

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。

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。