



下载APP



16 | Semaphore: 一篇文章搞懂信号量

2020-11-16 晁岳攀

Go 并发编程实战课

[进入课程 >](#)**讲述：安晓辉**

时长 16:48 大小 15.39M



你好，我是鸟窝。

在前面的课程里，我们学习了标准库的并发原语、原子操作和 Channel，掌握了这些，你就可以解决 80% 的并发编程问题了。但是，如果你要想进一步提升你的并发编程能力，就需要学习一些第三方库。

所以，在接下来的几节课里，我会给你分享 Go 官方或者其他人提供的第三方库，这节课我们先来学习信号量，信号量（Semaphore）是用来控制多个 goroutine 同时访问多个资源的并发原语。



信号量是什么？都有什么操作？

信号量的概念是荷兰计算机科学家 Edsger Dijkstra 在 1963 年左右提出来的，广泛应用在不同的操作系统中。在系统中，会给每一个进程一个信号量，代表每个进程目前的状态。未得到控制权的进程，会在特定的地方被迫停下来，等待可以继续进行的信号到来。

最简单的信号量就是一个变量加一些并发控制的能力，这个变量是 0 到 n 之间的一个数值。当 goroutine 完成对此信号量的等待 (wait) 时，该计数值就减 1，当 goroutine 完成对此信号量的释放 (release) 时，该计数值就加 1。当计数值为 0 的时候，goroutine 调用 wait 等待该信号量是不会成功的，除非计数器又大于 0，等待的 goroutine 才有可能成功返回。

更复杂的信号量类型，就是使用抽象数据类型代替变量，用来代表复杂的资源类型。实际上，大部分的信号量都使用一个整型变量来表示一组资源，并没有实现太复杂的抽象数据类型，所以你只要知道有更复杂的信号量就行了，我们这节课主要是学习最简单的信号量。

说到这儿呢，我想借助一个生活中的例子，来帮你进一步理解信号量。

举个例子，图书馆新购买了 10 本《Go 并发编程的独家秘籍》，有 1 万个学生都想读这本书，“僧多粥少”。所以，图书馆管理员先会让这 1 万个同学进行登记，按照登记的顺序，借阅此书。如果书全部被借走，那么，其他想看此书的同学就需要等待，如果有人还书了，图书馆管理员就会通知下一位同学来借阅这本书。这里的资源是《Go 并发编程的独家秘籍》这十本书，想读此书的同学就是 goroutine，图书管理员就是信号量。

怎么样，现在是不是很好理解了？那么，接下来，我们来学习下信号量的 P/V 操作。

P/V 操作

Dijkstra 在他的论文中为信号量定义了两个操作 P 和 V。P 操作 (decrease、wait、acquire) 是减少信号量的计数值，而 V 操作 (increase、signal、release) 是增加信号量的计数值。

使用伪代码表示如下（中括号代表原子操作）：

```
1 function V(semaphore S, integer I):
```

[复制代码](#)

```
2     [S ← S + I]
3
4  function P(semaphore S, integer I):
5      repeat:
6          [if S ≥ I:
7              S ← S - I
8              break]
```

可以看到，初始化信号量 S 有一个指定数量 (n) 的资源，它就像是一个有 n 个资源的池子。P 操作相当于请求资源，如果资源可用，就立即返回；如果没有资源或者不够，那么，它可以不断尝试或者阻塞等待。V 操作会释放自己持有的资源，把资源返还给信号量。信号量的值除了初始化的操作以外，只能由 P/V 操作改变。

现在，我们来总结下信号量的实现。

初始化信号量：设定初始的资源数量。

P 操作：将信号量的计数值减去 1，如果新值已经为负，那么调用者会被阻塞并加入到等待队列中。否则，调用者会继续执行，并且获得一个资源。

V 操作：将信号量的计数值加 1，如果先前的计数值为负，就说明有等待的 P 操作的调用者。它会从等待队列中取出一个等待的调用者，唤醒它，让它继续执行。

讲到这里，我想再稍微说一个题外话，我们在 [第 2 讲](#) 提到过饥饿，就是说在高并发的极端场景下，会有些 goroutine 始终抢不到锁。为了处理饥饿的问题，你可以在等待队列中做一些“文章”。比如实现一个优先级的队列，或者先入先出的队列，等等，保持公平性，并且照顾到优先级。

在正式进入实现信号量的具体实现原理之前，我想先讲一个知识点，就是信号量和互斥锁的区别与联系，这有助于我们掌握接下来的内容。

其实，信号量可以分为计数信号量 (counting semaphore) 和二进位信号量 (binary semaphore)。刚刚所说的图书馆借书的例子就是一个计数信号量，它的计数可以是任意一个整数。在特殊的情况下，如果计数值只能是 0 或者 1，那么，这个信号量就是二进位信号量，提供了互斥的功能（要么是 0，要么是 1），所以，有时候互斥锁也会使用二进位信号量来实现。

我们一般用信号量保护一组资源，比如数据库连接池、一组客户端的连接、几个打印机资源，等等。如果信号量蜕变成二进位信号量，那么，它的 P/V 就和互斥锁的 Lock/Unlock 一样了。

有人会很细致地区分二进位信号量和互斥锁。比如说，有人提出，在 Windows 系统中，互斥锁只能由持有锁的线程释放锁，而二进位信号量则没有这个限制（[Stack Overflow](#) 上也有相关的讨论）。实际上，虽然在 Windows 系统中，它们的确有些区别，但是对 Go 语言来说，互斥锁也可以由非持有的 goroutine 来释放，所以，从行为上来说，它们并没有严格的区别。

我个人认为，没必要进行细致的区分，因为互斥锁并不是一个很严格的定义。实际在遇到互斥并发的问题时，我们一般选用互斥锁。

好了，言归正传，刚刚我们掌握了信号量的含义和具体操作方式，下面，我们就来具体了解下官方扩展库的实现。

Go 官方扩展库的实现

在运行时，Go 内部使用信号量来控制 goroutine 的阻塞和唤醒。我们在学习基本并发原语的实现时也看到了，比如互斥锁的第二个字段：

```
1 type Mutex struct {  
2     state int32  
3     sema  uint32  
4 }
```


[复制代码](#)

信号量的 P/V 操作是通过函数实现的：

```
1 func runtime_Semacquire(s *uint32)  
2 func runtime_SemacquireMutex(s *uint32, lifo bool, skipframes int)  
3 func runtime_Semrelease(s *uint32, handoff bool, skipframes int)
```

[复制代码](#)

遗憾的是，它是 Go 运行时内部使用的，并没有封装暴露成一个对外的信号量并发原语，原则上我们没有办法使用。不过没关系，Go 在它的扩展包中提供了信号量

 `semaphore`，不过这个信号量的类型名并不叫 Semaphore，而是叫 Weighted。

之所以叫做 Weighted，我想，应该是因为可以在初始化创建这个信号量的时候设置权重（初始化的资源数），其实我觉得叫 Semaphore 或许会更好。

type Weighted

- `func NewWeighted(n int64) *Weighted`
- `func (s *Weighted) Acquire(ctx context.Context, n int64) error`
- `func (s *Weighted) Release(n int64)`
- `func (s *Weighted) TryAcquire(n int64) bool`


我们来分析下这个信号量的几个实现方法。

1. **Acquire 方法**：相当于 P 操作，你可以一次获取多个资源，如果没有足够多的资源，调用者就会被阻塞。它的第一个参数是 Context，这就意味着，你可以通过 Context 增加超时或者 cancel 的机制。如果是正常获取了资源，就返回 nil；否则，就返回 `ctx.Err()`，信号量不改变。
2. **Release 方法**：相当于 V 操作，可以将 n 个资源释放，返还给信号量。
3. **TryAcquire 方法**：尝试获取 n 个资源，但是它不会阻塞，要么成功获取 n 个资源，返回 true，要么一个也不获取，返回 false。

知道了信号量的实现方法，在实际的场景中，我们应该怎么用呢？我来举个 Worker Pool 的例子，来帮助你理解。

我们创建和 CPU 核数一样多的 Worker，让它们去处理一个 4 倍数量的整数 slice。每个 Worker 一次只能处理一个整数，处理完之后，才能处理下一个。

当然，这个问题的解决方案有很多种，这一次我们使用信号量，代码如下：

 复制代码

```
1 var (  
2     maxWorkers = runtime.GOMAXPROCS(0)           // worker数量
```




```
3     sema      = semaphore.NewWeighted(int64(maxWorkers)) //信号量
4     task      = make([]int, maxWorkers*4)                // 任务数, 是worker的2
5 )
6
7 func main() {
8     ctx := context.Background()
9
10    for i := range task {
11        // 如果没有worker可用, 会阻塞在这里, 直到某个worker被释放
12        if err := sema.Acquire(ctx, 1); err != nil {
13            break
14        }
15
16        // 启动worker goroutine
17        go func(i int) {
18            defer sema.Release(1)
19            time.Sleep(100 * time.Millisecond) // 模拟一个耗时操作
20            task[i] = i + 1
21        }(i)
22    }
23
24    // 请求所有的worker, 这样能确保前面的worker都执行完
25    if err := sema.Acquire(ctx, int64(maxWorkers)); err != nil {
26        log.Printf("获取所有的worker失败: %v", err)
27    }
28
29    fmt.Println(task)
30 }
```

在这段代码中, main goroutine 相当于一个 dispatcher, 负责任务的分发。它先请求信号量, 如果获取成功, 就会启动一个 goroutine 去处理计算, 然后, 这个 goroutine 会释放这个信号量 (有意思的是, 信号量的获取是在 main goroutine, 信号量的释放是在 worker goroutine 中), 如果获取不成功, 就等到有信号量可以使用的时候, 再去获取。

需要提醒你的是, 其实, 在这个例子中, 还有一个值得我们学习的知识点, 就是最后的那一段处理 (第 25 行)。**如果在实际应用中, 你想等所有的 Worker 都执行完, 就可以获取最大计数值的信号量。**

Go 扩展库中的信号量是使用互斥锁 + List 实现的。互斥锁实现其它字段的保护, 而 List 实现了一个等待队列, 等待者的通知是通过 Channel 的通知机制实现的。

我们来看一下信号量 Weighted 的数据结构:


 复制代码

```

1 type Weighted struct {
2     size      int64      // 最大资源数
3     cur       int64      // 当前已被使用的资源
4     mu        sync.Mutex  // 互斥锁, 对字段的保护
5     waiters   list.List  // 等待队列
6 }

```

在信号量的几个实现方法里, Acquire 是代码最复杂的一个方法, 它不仅仅要监控资源是否可用, 而且还要检测 Context 的 Done 是否已关闭。我们来看下它的实现代码。

 复制代码

```

1 func (s *Weighted) Acquire(ctx context.Context, n int64) error {
2     s.mu.Lock()
3     // fast path, 如果有足够的资源, 都不考虑ctx.Done的状态, 将cur加上n就返回
4     if s.size-s.cur >= n && s.waiters.Len() == 0 {
5         s.cur += n
6         s.mu.Unlock()
7         return nil
8     }
9
10    // 如果是不可能完成的任务, 请求的资源数大于能提供的最大的资源数
11    if n > s.size {
12        s.mu.Unlock()
13        // 依赖ctx的状态返回, 否则一直等待
14        <-ctx.Done()
15        return ctx.Err()
16    }
17
18    // 否则就需要把调用者加入到等待队列中
19    // 创建了一个ready chan, 以便被通知唤醒
20    ready := make(chan struct{})
21    w := waiter{n: n, ready: ready}
22    elem := s.waiters.PushBack(w)
23    s.mu.Unlock()
24
25
26    // 等待
27    select {
28    case <-ctx.Done(): // context的Done被关闭
29        err := ctx.Err()
30        s.mu.Lock()
31        select {
32        case <-ready: // 如果被唤醒了, 忽略ctx的状态
33            err = nil
34        default: 通知waiter
35            isFront := s.waiters.Front() == elem
36            s.waiters.Remove(elem)

```

```

37         // 通知其它的waiters,检查是否有足够的资源
38         if isFront && s.size > s.cur {
39             s.notifyWaiters()
40         }
41     }
42     s.mu.Unlock()
43     return err
44 case <-ready: // 被唤醒了
45     return nil
46 }
47 }

```

其实，为了提高性能，这个方法中的 fast path 之外的代码，可以抽取成 acquireSlow 方法，以便其它 Acquire 被内联。

Release 方法将当前计数值减去释放的资源数 n，并唤醒等待队列中的调用者，看是否有足够的资源被获取。

[复制代码](#)

```

1 func (s *Weighted) Release(n int64) {
2     s.mu.Lock()
3     s.cur -= n
4     if s.cur < 0 {
5         s.mu.Unlock()
6         panic("semaphore: released more than held")
7     }
8     s.notifyWaiters()
9     s.mu.Unlock()
10 }

```

notifyWaiters 方法就是逐个检查等待的调用者，如果资源不够，或者是没有等待者了，就返回：

[复制代码](#)

```

1 func (s *Weighted) notifyWaiters() {
2     for {
3         next := s.waiters.Front()
4         if next == nil {
5             break // No more waiters blocked.
6         }
7
8
9         w := next.Value.(waiter)

```



```
10     if s.size-s.cur < w.n {
11         //避免饥饿，这里还是按照先入先出的方式处理
12         break
13     }
14
15     s.cur += w.n
16     s.waiters.Remove(next)
17     close(w.ready)
18 }
19 }
```

notifyWaiters 方法是按照先入先出的方式唤醒调用者。当释放 100 个资源的时候，如果第一个等待者需要 101 个资源，那么，队列中的所有等待者都会继续等待，即使有的等待者只需要 1 个资源。这样做的目的是避免饥饿，否则的话，资源可能总是被那些请求资源数小的调用者获取，这样一来，请求资源数巨大的调用者，就没有机会获得资源了。

好了，到这里，你就知道了官方扩展库的信号量实现方法，接下来你就可以使用信号量了。不过，在此之前呢，我想给你讲几个使用时的常见错误。这部分内容可是帮助你避坑的，我建议你好好学习。

使用信号量的常见错误

保证信号量不出错的前提是正确地使用它，否则，公平性和安全性就会受到损害，导致程序 panic。

在使用信号量时，最常见的几个错误如下：

请求了资源，但是忘记释放它；

释放了从未请求的资源；

长时间持有一个资源，即使不需要它；

不持有一个资源，却直接使用它。

不过，即使你规避了这些坑，在同时使用多种资源，不同的信号量控制不同的资源的时候，也可能会出现死锁现象，比如 [哲学家就餐问题](#)。

就 Go 扩展库实现的信号量来说，在调用 Release 方法的时候，你可以传递任意的整数。但是，如果你传递一个比请求到的数量大的错误的数值，程序就会 panic。如果传递一个

负数，会导致资源永久被持有。如果你请求的资源数比最大的资源数还大，那么，调用者可能永远被阻塞。


所以，**使用信号量遵循的原则就是请求多少资源，就释放多少资源**。你一定要注意，必须使用正确的方法传递整数，不要“耍小聪明”，而且，请求的资源数一定不能超过最大资源数。

其它信号量的实现

除了官方扩展库的实现，实际上，我们还有很多方法实现信号量，比较典型的就是使用 Channel 来实现。

根据之前的 Channel 类型的介绍以及 Go 内存模型的定义，你应该能想到，使用一个 buffer 为 n 的 Channel 很容易实现信号量，比如下面的代码，我们就是使用 `chan struct{}` 类型来实现的。

在初始化这个信号量的时候，我们设置它的初始容量，代表有多少个资源可以使用。它使用 Lock 和 Unlock 方法实现请求资源和释放资源，正好实现了 Locker 接口。

 复制代码

```
1 // Semaphore 数据结构，并且还实现了Locker接口
2 type semaphore struct {
3     sync.Locker
4     ch chan struct{}
5 }
6
7 // 创建一个新的信号量
8 func NewSemaphore(capacity int) sync.Locker {
9     if capacity <= 0 {
10         capacity = 1 // 容量为1就变成了一个互斥锁
11     }
12     return &semaphore{ch: make(chan struct{}, capacity)}
13 }
14
15 // 请求一个资源
16 func (s *semaphore) Lock() {
17     s.ch <- struct{}{}
18 }
19
20 // 释放资源
21 func (s *semaphore) Unlock() {
22     <-s.ch
```

当然，你还可以自己扩展一些方法，比如在请求资源的时候使用 Context 参数 (Acquire(ctx)) 、实现 TryLock 等功能。

看到这里，你可能会问，这个信号量的实现看起来非常简单，而且也能应对大部分的信号量的场景，为什么官方扩展库的信号量的实现不采用这种方法呢？其实，具体是什么原因，我也不知道，但是我必须要强调的是，官方的实现方式有这样一个功能：**它可以一次请求多个资源，这是通过 Channel 实现的信号量所不具备的。**

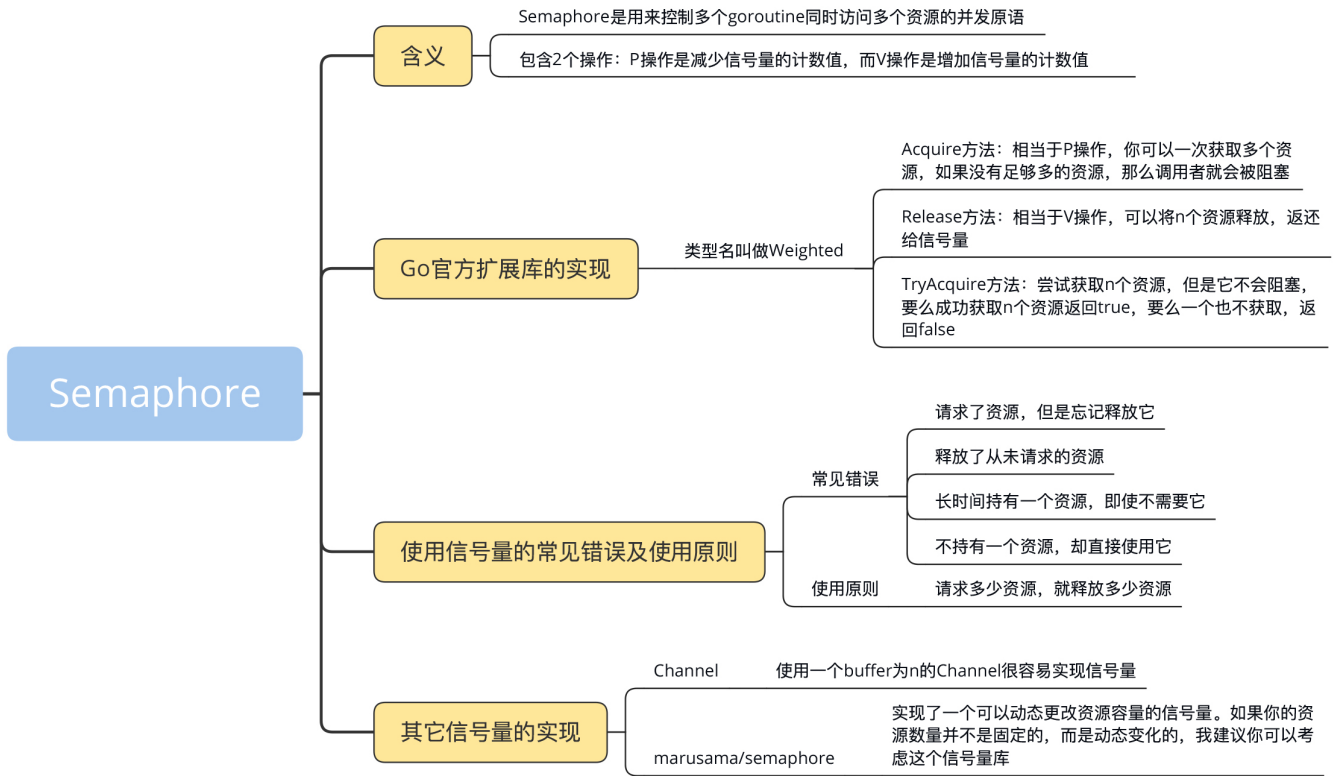
除了 Channel，[marusama/semaphore](https://github.com/marusama/semaphore) 也实现了一个可以动态更改资源容量的信号量，也是一个非常有特色的实现。如果你的资源数量并不是固定的，而是动态变化的，我建议你考虑一下这个信号量库。

总结

这是一个很奇怪的现象：标准库中实现基本并发原语（比如 Mutex）的时候，强烈依赖信号量实现等待队列和通知唤醒，但是，标准库中却没有把这个实现直接暴露出来放到标准库，而是通过第三库提供。

不管怎样，信号量这个并发原语在多资源共享的并发控制的场景中被广泛使用，有时候也会被 Channel 类型所取代，因为一个 buffered chan 也可以代表 n 个资源。

但是，官方扩展的信号量也有它的优势，就是可以一次获取多个资源。**在批量获取资源的场景中，我建议你尝试使用官方扩展的信号量。**



思考题

1. 你能用 Channel 实现信号量并发原语吗? 你能想到几种实现方式?
2. 为什么信号量的资源数设计成 int64 而不是 uint64 呢?

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, 也欢迎你今天的分享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 15 | 内存模型: Go如何保证并发读写的顺序?

下一篇 17 | SingleFlight 和 CyclicBarrier: 请求合并和循环栅栏该怎么用?

精选留言 (8)

 写留言



myrfy

2020-11-16

第一个问题:

至少两种, 写入ch算获取, 自己读取ch算获取

第二个问题应该是防止错误获取或者释放信号量时, 出现负数溢出到无穷大的问题。如果溢出到无穷大, 就会让信号量失效, 从而导致1被保护资源更大规模的破坏

展开 ∨



1



伟伟

2020-11-23

```
type Semaphore chan struct{}
```

```
func NewSemaphore(cap int) Semaphore {  
    return make(chan struct{}, cap)  
}...
```

展开 ▾

作者回复: 唯一存在的问题是可能出现死锁。

比如信号量是10, 同时有两个goroutine请求8个资源。



1



大漠胡萝卜

2020-11-21

在日常开发中, 没怎么使用信号量semaphore, 一般使用channel来解决这种问题。另外, 并发的时候使用池化技术感觉更加通用吧。

展开 ▾

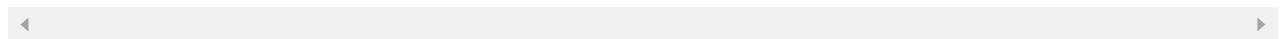


虫子樱桃

2020-11-19

老师的例子里面, 是通过 计算机的协程 runtime.GOMAXPROCS(0) 来模拟有限的资源 (比喻例子里面的书), 那么这个semaphore的场景是不是就是比较适用于请求有流量或者调用次数限制的场景呢?

作者回复: 这个更多是用ratelimiter,信号量主要并发访问n个资源的场景

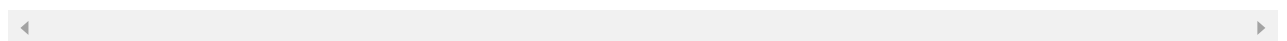


Ethan Liu

2020-11-17

老师, Acquire函数为什么还会有第二个select语句? 这部分逻辑是什么啊?

作者回复: 理解了ctx,也就理解select。当外部context通知取消请求时, 会在检查一下当前是否请求成功了



刚子

2020-11-16

不是很理解这句话：“一次请求多个资源，这是通过 Channel 实现的信号量所不具备的。”

Channel 也可以开启多个goroutine 去请求多个资源

展开

作者回复: 意思是通过一次调用，只能从chan中获取一个值，多个。goroutine需要调用多次才能得到n个值



容易
2020-11-16

老师的题还是有难度的

展开



橙子888
2020-11-16

打卡。

展开