

12 | 进程内协同：同步、互斥与通讯

2019-05-24 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 13:36 大小 18.69M



你好，我是七牛云许式伟。

上一节开始我们进入了多任务的世界，我们详细介绍了三类执行体：进程、线程和协程，并且介绍了每一种执行体的特点。

既然启动了多个执行体，它们就需要相互协同，今天我们先讨论进程内的执行体协同。

考虑到进程内的执行体有两类：用户态的协程（以 Go 语言的 goroutine 为代表）、操作系统的线程，我们对这两类执行体的协同机制做个概要。如下：

类别	goroutine	线程	语义
原子操作	atomic.AddInt32/AddInt64		val += delta return val
	atomic.SwapInt32/SwapInt64		oldval, val = val, newval return oldval
	atomic.CompareAndSwapInt32 atomic.CompareAndSwapInt64		if val == oldval { val = newval }
互斥体	sync.Mutex	pthread_mutex_t	锁：用于多个执行体互斥访问，避免多个执行体同时操作一组数据产生竞争。
	sync.RWMutex	pthread_rwlock_t	读写锁：锁在读多写少情况下的优化。把操作分为读操作和写操作，调用不同的互斥操作。
等待组	sync.WaitGroup		同步：等待一组在做不同任务的执行体的任务全部结束。
条件变量	sync.Cond	pthread_cond_t	同步：在锁保护下，如果资源不足下等待，否则对该资源执行某种操作。
管道	io.Pipe	pipe	执行体之间收发无类型的二进制消息。
消息传递	channel	缺失	执行体之间收发类型安全的消息。

让我们逐一详细分析一下它们。

原子操作

首先让我们看一下原子操作。需要注意的是，原子操作是 CPU 提供的能力，与操作系统无关。这里列上只是为了让你能够看到进程内通讯的全貌。

顾名思义，原子操作的每一个操作都是原子的，不会中途被人打断，这个原子性是 CPU 保证的，与执行体的种类无关，无论 goroutine 还是操作系统线程都适用。

从语义上来说，原子操作可以用互斥体来实现，只不过原子操作要快得多。

例如：

```
1 var val int32
2 ...
3 newval = atomic.AddInt32(&val, delta)
```


等价于：

 复制代码

```
1 var val int32
2 var mutex sync.Mutex
3 ...
4 mutex.Lock()
5 val += delta
6 newval = val
7 mutex.Unlock()
```

执行体的互斥

互斥体也叫锁。锁用于多个执行体之间的互斥访问，避免多个执行体同时操作一组数据产生竞争。其使用界面上大概是这样的：

 复制代码

```
1 func (m *Mutex) Lock()
2 func (m *Mutex) Unlock()
```

锁的使用范式比较简单：在操作需要互斥的数据前，先调用 Lock，操作完成后就调用 Unlock。但总是存在一些不求甚解的人，对锁存在各种误解。

有的人会说锁很慢。甚至我曾看到有 Go 程序员用 channel 来模拟锁，原因就是锁太慢了，尽量不要用锁。产生“锁慢，channel 快”这种错觉的一个原因，可能是人们经常看到这样的忠告：

不要通过共享内存（锁）来通信，要通过通信（channel）来共享内存。


不明就里的人们看到这话后，可能就有了这样的印象：锁是坏的，锁是性能杀手，channel 是好的，是 Go 发明的先进武器，应该尽可能用 channel，而不要用锁。

快慢是相对而言的。锁的确会导致代码串行执行，所以在某段代码并发度非常高的情况下，串行执行的确会导致性能的显著降低。但平心而论，相比其他的进程内通讯的原语来说，锁并不慢。从进程内通讯来说，比锁快的东西，只有原子操作。

例如 channel，作为进程内执行体间传递数据的设施来说，它本身是共享变量，所以 channel 的每个操作必然是有锁的。事实上，channel 的每个操作都比较耗时。关于这一点，在下文解释 channel 背后的工作机理后，你就会清楚知道。

那么锁的问题在哪里？锁的最大问题在于不容易控制。锁 Lock 了但是忘记 Unlock 后是灾难性的，因为相当于服务器挂了，所有和该锁相关的代码都不能被执行。


比如：

 复制代码

```
1 mutex.Lock()  
2 doSth()  
3 mutex.Unlock()
```

在考虑异常的情况下，这段代码是不安全的，如果 doSth 抛出了异常，那么服务器就会出现问题。

为此 Go 语言还专门发明了一个 defer 语法来保证配对：

 复制代码

```
1 mutex.Lock()  
2 defer mutex.Unlock()  
3 doSth()
```

这样可以保证即使 doSth 发生异常，mutex.Unlock 仍然会被正确地执行。这类在异常情况下也能够正常工作的代码，我们称之为“对异常安全的代码”。如果语言不支持 defer，而是支持 try ... catch，那么代码可能是这样的：

```
1 mutex.Lock()
2 try {
3     doSth()
4 } catch (e Exception) {
5     mutex.Unlock()
6     throw e
7 }
8 mutex.Unlock()
```

锁不容易控制的另一个表现是锁粒度的问题。例如上面 `doSth` 函数里面如果调用了网络 IO 请求，而网络 IO 请求在少数特殊情况下可能会出现慢请求，要好几秒才返回。那么这几秒对服务器来说就好像挂了，无法处理请求。

对服务器来说这是极为致命的。对后端程序员来说，有一句箴言要牢记：

不要在锁里面执行费时操作。

这里“锁里面”是指在 `mutex.Lock` 和 `mutex.Unlock` 之间的代码。

在锁的最佳编程实践中，如果明确一组数据的并发访问符合“绝大部分情况下是读操作，少量情况有写操作”，这种“读多写少”特征，那么应该用读写锁。

所谓读写锁，是把锁里面的操作分为读操作和写操作两种，对应调用不同的互斥操作。

如果是读操作，代码如下：

```
1 mutex.RLock()
2 defer mutex.RUnlock()
3 doReadOnlyThings
```

如果是锁里面是写操作，代码就和普通锁一样，如下：

```
1 mutex.Lock()
2 defer mutex.Unlock()
3 doWriteThings
```

为什么在“读多写少”的情况下，这样的使用范式能够优化性能？

因为从需求上来说，如果当前我们正在执行某个读操作，那么再来一个新的读操作，是不应该挡在外面的，大家都不修改数据，可以安全地并发执行。但如果来的是写操作，就应该挡在外面，等待读操作执行完。整体来说，读写锁的特性就是：

读操作不阻止读操作，阻止写操作；
写操作阻止一切，不管读操作还是写操作。

执行体的同步

聊完了执行体的互斥，我们再来看下执行体之间的同步。

同步的一个最常见的场景是：把一个大任务分解为 n 个小任务，分配给 n 个执行体并行去做，等待它们一起做完。这种同步机制我们叫“等待组”。

其使用界面上大概是这样的：

 复制代码

```
1 func (wg *WaitGroup) Add(n int)
2 func (wg *WaitGroup) Done()
3 func (wg *WaitGroup) Wait()
```

用法上大概是这样的：

 复制代码

```
1 var wg WaitGroup
2 ...
3 wg.Add(n)
4 for 循环 n 次 {
5     go func() {
6         defer wg.Done()
```


```
7         doTaski // 执行第 i 个任务
8     }()
9 }
10 wg.Wait()
```

简而言之，在每个任务开始的时候调用 `wg.Add(1)`，结束的时候调用 `wg.Done()`，然后在主执行体调用 `wg.Wait()` 等待这些任务结束。

需要注意的是，`wg.Add(1)` 是要在任务的 goroutine 还没有开始就先调用，否则可能出现某个任务还没有开始执行就被认为结束了。

条件变量（Condition Variable）是一个更通用的同步原语，设计精巧又极为强大。强大到什么程度？像 channel 这样的通讯机制都可以用它来实现。


条件变量的使用界面上大概是这样的：

 复制代码

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

那么，怎么用条件变量？

我们先看下初始化。条件变量初始化的时候需要传入一个互斥体，它可以是普通锁（Mutex），也可以是读写锁（RWMutex）。如下：

 复制代码

```
1 var mutex sync.Mutex // 也可以是 sync.RWMutex
2 var cond = sync.NewCond(&mutex)
3 ...
```

为什么创建条件变量需要传入锁？因为 `cond.Wait()` 的需要。Wait 内部实现逻辑是：


```
1 把自己加入到挂起队列
2 mutex.Unlock()
3 等待被唤醒 // 挂起的执行体会被后续的 cond.Broadcast 或 cond.Signal() 唤醒
4 mutex.Lock()
```

初始化了条件变量后，我们再来看看它的使用方式。条件变量的用法有一个标准化的模板，看起来大概是这样的：

```
1 mutex.Lock()
2 defer mutex.Unlock()
3 for conditionNotMetToDo {
4     cond.Wait()
5 }
6 doSomething
7 if conditionNeedNotify {
8     cond.Broadcast()
9     // 有时可以优化为 cond.Signal()
10 }
```

看起来有些复杂，让我们来解释一下。加锁后，先用一个 for 循环判断当前是否能够做我们想做的事情，如果做不了就调用 cond.Wait() 进行等待。

这里很重要的一个细节是注意用的是 for 循环，而不是 if 语句。这是因为 cond.Wait() 得到了执行权后不代表我们想做的事情就一定能够干了，所以要再重新判断一次条件是否满足。

确定能够做事情了，于是 doSomething。在做的过程中间，如果我们判断可能挂起队列中的部分执行体满足了重新执行的条件，就用 cond.Broadcast 或 cond.Signal 唤醒它们。

cond.Broadcast 比较粗暴，它唤醒了所有在这个条件变量挂起的执行体，而 cond.Signal 则只唤醒其中的一个。

什么情况下应该用 cond.Broadcast，什么情况下应该用 cond.Signal？最偷懒的方式当然是不管三七二十一，用 cond.Broadcast 一定没问题。但是本着经济的角度，我们还是要交


代清楚 cond.Signal 的适用范围：

挂起在这个条件变量上的执行体，它们等待的条件是一致的；

本次 doSomething 操作完成后，所释放的资源只够一个执行体来做事情。

Cond 原语虽然叫条件变量，但是实际上它既没有明白说变量具体是什么样的，也没有说条件具体是什么样的。变量是指 “一组要在多个执行体之间协同的数据”。条件是指做任务前 Wait 的 “前置条件”，和做任务时需要唤醒其它人的 “唤醒条件”。

这样的介绍相当的抽象。我们拿 Go 语言的 channel 开刀，自己实现一个。代码如下：

 复制代码

```
1 type Channel struct {
2     mutex sync.Mutex
3     cond *sync.Cond
4     queue *Queue
5     n int
6 }
7
8 func NewChannel(n int) *Channel {
9     if n < 1 {
10         panic("todo: support unbuffered channel")
11     }
12     c := new(Channel)
13     c.cond = sync.NewCond(&c.mutex)
14     c.queue = NewQueue()
15     // 这里 NewQueue 得到一个普通的队列
16     // 代码从略
17     c.n = n
18     return c
19 }
20
21 func (c *Channel) Push(v interface{}) {
22     c.mutex.Lock()
23     defer c.mutex.Unlock()
24     for c.queue.Len() == c.n { // 等待队列不满
25         c.cond.Wait()
26     }
27     if c.queue.Len() == 0 { // 原来队列是空的，可能有人等待数据，通知它们
28         c.cond.Broadcast()
29     }
30     c.queue.Push(v)
31 }
32
33 func (c *Channel) Pop() (v interface{}) {
```

```

34     c.mutex.Lock()
35     defer c.mutex.Unlock()
36     for c.queue.Len() == 0 { // 等待队列不空
37         c.cond.Wait()
38     }
39     if c.queue.Len() == c.n { // 原来队列是满的，可能有人等着写数据，通知它们
40         c.cond.Broadcast()
41     }
42     return c.queue.Pop()
43 }
44
45 func (c *Channel) TryPop() (v interface{}, ok bool) {
46     c.mutex.Lock()
47     defer c.mutex.Unlock()
48     if c.queue.Len() == 0 { // 如果队列为空，直接返回
49         return
50     }
51     if c.queue.Len() == c.n { // 原来队列是满的，可能有人等着写数据，通知它们
52         c.cond.Broadcast()
53     }
54     return c.queue.Pop(), true
55 }
56
57 func (c *Channel) TryPush(v interface{}) (ok bool) {
58     c.mutex.Lock()
59     defer c.mutex.Unlock()
60     if c.queue.Len() == c.n { // 如果队列满，直接返回
61         return
62     }
63     if c.queue.Len() == 0 { // 原来队列是空的，可能有人等待数据，通知它们
64         c.cond.Broadcast()
65     }
66     c.queue.Push(v)
67     return true
68 }

```

对着这个 Channel 的实现，你是否对条件变量有感觉很多？顺便提醒一点，这个 Channel 的实现不支持无缓冲 channel，也就是不支持 `NewChannel(0)` 的情况。如果你感兴趣，可以改改这个问题。

执行体的通讯

聊完同步与互斥，我们接着聊执行体的通讯：怎么在执行体间收发消息。

管道是大家都很熟知的执行体间的通讯机制。规格如下：

```
1 func Pipe() (pr *PipeReader, pw PipeWriter)
```

用法上，先调用`pr, pw := io.Pipe()`得到管道的写入端和读出端，分别传给两个并行执行的 goroutine（其他语言也类似），然后一个 goroutine 读，一个 goroutine 写就好了。

管道用处很多。一个比较常见的用法是做读写转换，例如，假设我手头有一个算法：

```
1 func Foo(w io.Writer) error
```

这个算法生成的数据流，需要作为另一个函数的输入，但是这个函数的输入是 `io.Reader`，原型如下：


```
1 func Bar(r io.Reader)
```

那么怎么把它们串起来呢？用管道我们很容易实现这样的变换：

```
1 func FooReader() io.ReadCloser {  
2     pr, pw := io.Pipe()  
3     go func() {  
4         err := Foo(pw)  
5         pw.CloseWithError(err)  
6     }()  
7     return pr  
8 }
```

这个 `FooReader` 函数几句话就把 `Foo` 变成了一个符合 `io.Reader` 接口的对象，它就可以很方便的和 `Bar` 函数结合了。

其实 Go 语言中引入的 `channel` 也是管道，只不过它是类型安全的管道。具体用法如下：

 复制代码

```
1 c := make(chan Type, n) // 创建一个能够传递 Type 类型数据的管道，缓冲大小为 n
2 ...
3 go func() {
4     val := <-c // 从管道读入
5 }()
6 ...
7 go func() {
8     c <- val // 向管道写入
9 }()
```

我们后面在“服务端开发”一章，我们还会比较详细讨论 `channel`，今天先了解一个大体的语义。

结语

总结一下，我们今天主要聊了执行体间的协同机制：原子操作、同步、互斥与通讯。我们重点聊了锁和同步原语“条件变量”。

锁在一些人心中是有误解的，但实际上锁在服务端编程中的比重并不低，我们可能经常需要和它打交道，建议多花精力理解它们。

条件变量是最复杂的同步原语，功能强大。虽然平常我们直接使用条件变量的机会不是太多，大部分常见的场景往往有更高阶的原语（例如 `channel`）可以取代。但是它的设计精巧而高效，值得细细体会。

你会发现，操作系统课本上的信号量这样的同步原语，我们这里没有交代，这是因为它被更强大而且性能更好的同步原语“条件变量”所取代了。

上面我们为了介绍条件变量的用法，我们实作了一个 `channel`，你也可以考虑用信号量这样的东西来实现一遍，然后分析一下为什么我们说基于“条件变量”的版本是更优的。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。在下期，我们将讨论进程与进程之间的协同：进程间的同步互斥、资源共享与通讯。

如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。



许式伟的架构课

从源头出发，带你重新理解架构设计

许式伟
七牛云 CEO



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 11 | 多任务：进程、线程与协程

下一篇 13 | 进程间的同步互斥、资源共享与通讯

精选留言 (39)

写留言



笨拙的自由

2019-05-24

10

希望老师类比java说一下,特别是条件原语那块看得有点懵

展开

作者回复: 如果用 Java，代码看起来是这样的：

```

class Channel {
    private final Lock lock = new ReentrantLock();
    private Condition cond = lock.newCondition();
    private final Queue queue = new Queue();
    private int n;

    public Channel(int cap) {
        n = cap;
    }

    public void push(Object v) {
        lock.lock();
        try {
            while (queue.size() == n) {
                cond.await();
            }
            if (queue.size() == 0) {
                cond.signalAll();
            }
            queue.push(v);
        } finally {
            lock.unlock();
        }
    }

    public Object pop() {
        lock.lock();
        try {
            while (queue.size() == 0) {
                cond.await();
            }
            if (queue.size() == n) {
                cond.signalAll();
            }
            return queue.pop();
        } finally {
            lock.unlock();
        }
    }
}

```



立耳

2019-05-27

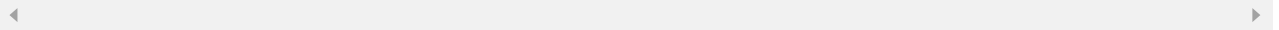
👍 7

许老师，下面的这段代码是不是存在问题，以Push为例，应该先执行 `c.queue.Push` 再进行广播，否则可能通知到其他协程进行Pop，系统调度可能先进行了另外一个协程的 `c.queue.Pop()`，这个时候还没有入队列。

```
func (c *Channel) Push(v interface{}) {...
```

展开 ▾

作者回复: 代码没有问题的。先 Push 还是先通知都可以，次序可以交换的。因为反正锁还没有释放，这里只是标记一下哪些执行体可以调度，并没有真正发生控制权的转移。而且就算转移了也没问题的，你可以留意下本文贴的条件变量的 Wait 函数实现，它获得控制权后下一句就是 `mutex.Lock` 去申请锁，而我们这里是 Push 后才调用 `mutex.Unlock` 释放锁的，所以 Broadcast 和 Push 的次序可以随意交换。



Geek_gooy

2019-05-26

👍 5

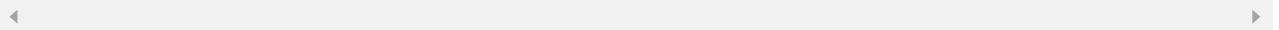
老师

我明白了，比如A线程notify或者signal，被唤醒的线程并不会马上执行，而是需要等待A线程退出同步块或者unlock才会执行。

如果是notifyAll，也同样如此，但是等到唤醒并获得执行权的线程执行结束后，CPU会...

展开 ▾

作者回复: 是这样，你可以看我文章中Wait的代码，在唤醒后第一件事情是lock，也就是请求锁，所以只有A线程unlock后，其他被唤醒的线程中的一个会得到锁往下走。



小美

2019-05-24

👍 4

协程和线程还是没区别清楚

展开 ▾

作者回复: 一个操作系统调度，一个用户态自己来调度





白小狮

2019-05-25

👍 3

go中的panic会导致整个主进程都挂掉，goroutine里面的panic后recover不住，是逻辑上就是应该整个主进程都退出吗

作者回复: 设计上，goroutine 应该自己去 recover 错误，而不是主进程来 recover。



Cordova

2019-05-28

👍 2

看完代码后发现go做的通信发现并没有什么优势，其他语言做通信也这么干、刷了下评论区提到libevent、才恍然发现标题这节讲的是同步！可能是我太期待许老师讲异步了😁~ 目前python我用异步首选会把异步过程交给libuv。听了老师上节讲到python的协程只是一种编程范式，想到内置的asyncio虽然是做了异步但还是有很较大的性能提升空间这个逻辑也就通了！希望许老师在讲异步的时候能多提一提跨平台异步库他们是怎么实现

展开 ∨

作者回复: 1、go和java的代码只是形似，实质不同，因为go里面的channel是协程的通讯设施，java版本的是线程的通讯设施，大相径庭；

2、我们本节提的同步，和同步io的同步，两个是完全不同含义的同步；

3、我们课程不太会讲资料已经相对多的某个细节，除非这个细节非常关键影响到全局的理解。



Geek_gooy

2019-05-25

👍 2

老师

1、像这种有进有出的是不是应该创建两个condition。大小为满时，避免进的线程，唤醒的可能还是进的线程。大小为零时，出的唤醒的还是出的线程。

2、cond.signal()方法把lock锁释放了吗，如果释放了，后面再unlock是不是没做任何操作。...

展开 ∨

作者回复: 1、你是对的，用两个cond性能会更好。但是用一个也是可以正常工作的。

2、cond.signal 不是把锁释放了，是让等待在这个cond上的执行体改变状态（从挂起到可被调

度)，从而允许调度程序给它执行权。

3、对的

CC

2019-05-25

👍 2

老师，我看你的chanel代码实现，比如pop方法。是先broadcast，后pop。我的理解刚好相反。应该是先pop后通知。请教下问什么

作者回复: 次序可以交换的。因为反正锁还没有释放，这里只是标记一下哪些执行体可以调度

輪迴

2019-05-24

👍 2

同时在看《深入浅出计算机组成原理》和《GO 语言从入门到实战》，发现三个课程之间的关联性还是蛮多的，相辅相成，更加帮助知识点的深入理解

作者回复: 👍

杨雪峰

2019-05-24

👍 2

感觉和 Java 的 wait notify notifyall 很像

展开 ▾

觉

2019-05-24

👍 2

感恩大佬分享 随喜大佬

展开 ▾

Ender

2019-06-06

👍 1

还是没太明白条件变量在channel代码里面的意义，所有操作都是先获取锁，在一个操作没

完成的情况下其他都不会进到cond.Wait()呀。按理只需要锁就能做到了channel的实现

了。
展开

作者回复: 向 channel.push 一个对象时, 要考虑 channel 满了, 这时会等待, 这就是 cond.Wait 的逻辑



thewangzli

2019-05-29

1

老师你好, 文章中说Signal比Broadcast好些, 但是王宝令老师的专栏《Java并发编程实战》第15章提到Dubbo唤醒等待线程从signal优化为signalAll。是因为java中的signal/signalAll和Go的Signal/Broadcast有差异吗?

展开

作者回复: 你说的是对的。我其实没想到什么情况下用signal, 大部分情况下都是用 broadcast, 包括本文中的例子。因为用 signal 意味着每次资源的使用都要通知, 其实退化为信号量的 PV 操作了, 这一定性能是变差的。



Bachue Zh...

2019-05-28

1

管道合理的使用场景是什么? 难道不是父子间进程通讯或是子进程之间的通讯吗? 进程间通讯可不是 Goroutine 能轻易取代的。

作者回复: Go语言里面有两个管道实现, 一个io.Pipe, 是用于goroutine之间的; 一个是 os.Pipe, 是用于进程之间的。



Bachue Zh...

2019-05-28

1

看不懂为什么要把管道的使用场景放在 Goroutine 之间通讯, 这并不是管道合理的使用场景啊, 至于和 Channel 做对比就更没有意义了。

作者回复: 管道合理的使用场景是什么?



82

2019-05-28

👍 1

在高并发下获取锁操作，谁来保证单次操作的原子性，操作系统还是cpu或者其他呢？

作者回复: 当然是锁。如果这都保证不了，那它就不是锁了



82

2019-05-28

👍 1

获取锁本身是什么样的操作，怎么保证在这个点上不出现异常呢？

展开 ▾

作者回复: 获取锁失败只有一种可能就是mutex对象非法（比如为nil），那就抛出异常，本身也是属于异常安全的代码。



keshawn

2019-05-27

👍 1

Channel只是一个并发阻塞队列？

展开 ▾

作者回复: 本质上是的，不过它有一些高级用法是常规队列没有的，比如用select去多个channel同时进行读写。



Dean

2019-05-26

👍 1

请问java如何实现管道，是类似SynchronousQueue的方式吗

展开 ▾

作者回复: 如果只是简单实现，我文章中给出的是一种参考。不过Go语言的channel功能其实不止于此，要完整实现是比较难的，尤其是Go channel还支持多channel一起读或取。



f抵达

2019-05-26

👍 1

如果是用户太自己对寄存器进行操作?
对物理器件的操作不都是要经过系统调用么?
难道协程x是用户态的操作系统?

展开 ▾

作者回复: 寄存器不是输入输出设备, 操作寄存器不需要经过操作系统, 编译器整天和寄存器打交道的。