基本的信号量（Semaphore）机制

杨子凡

Nov 15, 2025

掌握这个操作系统与并发编程的基石，亲手用代码实现它。

在并发编程中，多个线程或进程同时访问共享资源时，常常会导致不可预测的结果。考虑一个经典的生产者－消费者问题场景：假设有一个共享缓冲区，生产者线程负责向缓冲区添加数据，消费者线程负责从缓冲区取出数据。如果不加任何同步控制，生产者可能在消费者尚未处理完数据时就覆盖旧数据，或者消费者可能读取到无效或重复的数据。这种数据竞争和数据不一致问题，会严重破坏程序的正确性和可靠性。那么，我们如何协调多个线程或进程，确保它们安全、有序地访问有限的共享资源呢？

信号量正是为了解决这类并发问题而诞生的。它是由荷兰计算机科学家 Dijkstra 在二十世纪六十年代提出的伟大思想。简单来说，信号量是一个计数器，用于控制对共享资源的访问线程或进程数量。本文将带领您从理论到实践，深入理解信号量的核心机制，并亲手用代码实现一个基本的信号量。

1　信号量的核心概念剖析

信号量的本质是一个非负整数计数器，它通过两个不可分割的原子操作来管理资源访问。这两个操作通常被称为 wait（或 P、down、acquire）和 signal（或 V、up、release）。wait 操作的语义是尝试获取资源：如果计数器值大于零，则将其减一并继续执行；如果等于零，则当前线程或进程被阻塞，直到计数器值变为正数。signal 操作的语义是释放资源：将计数器值加一，并唤醒一个正在等待该信号量的线程或进程。这些操作的原子性确保了在并发环境下不会出现竞态条件。

信号量有两种基本类型：二进制信号量和计数信号量。二进制信号量的计数器值只能是零或一，它常用于实现互斥锁，保护临界区，保证同一时刻只有一个线程可以访问共享资源。计数信号量的计数器值可以是任意非负整数，它用于控制对一组完全相同资源的访问，例如在数据库连接池中限制并发连接数，或者实现流量控制。

为了更直观地理解信号量，我们可以借助现实世界的类比。想象一个停车场，它代表一个计数信号量：总车位数是信号量的初始值。每进入一辆车相当于执行 wait 操作，空闲车位数减一；每离开一辆车相当于执行 signal 操作，空闲车位数加一。当车位满时（计数器为零），新车必须等待，直到有车离开。另一个例子是厕所钥匙，它代表一个二进制信号量：只有一个钥匙。一个人拿着钥匙进去相当于 wait 操作，其他人必须等待；出来时归还钥匙相当于 signal 操作，下一个人才能进去。这些类比帮助我们形象化信号量的工作原理。

2　信号量的经典应用模式

在并发编程中，信号量常用于两种基本模式：互斥和同步。互斥模式通过一个初始值为一的二进制信号量实现，确保同一时刻只有一个线程可以进入临界区。线程在进入临界区前调用 wait 操作，如果信号量值为一，则减一并进入；如果为零，则阻塞。退出临界区时调用 signal 操作，将值加一，并唤醒一个等待线程。这种模式简单

有效，但需要确保 wait 和 signal 操作成对出现，否则可能导致死锁或资源泄漏。
同步模式则更复杂，它用于协调线程间的执行顺序。一个经典例子是生产者－消费者问题：生产者线程生产数据并放入共享缓冲区，消费者线程从缓冲区取出数据。缓冲区大小有限，因此需要协调生产者和消费者的速度。这里使用三个信号量：empty_slots 表示空缓冲区数量，初始值为缓冲区大小；full_slots 表示已填充缓冲区数量，初始值为零；mutex 是一个二进制信号量，初始值为一，用于保护对缓冲区的互斥访问。生产者首先等待 empty_slots，然后获取 mutex 添加数据，最后释放 mutex 并增加 full_slots。消费者则相反，等待 full_slots，获取 mutex 取出数据，释放 mutex 并增加 empty_slots。这种模式确保了生产者和消费者之间的有序协作，避免了数据竞争。

3 动手实现：构建我们自己的信号量

现在，让我们亲手实现一个简单的信号量。我们将使用 Python 的 threading 模块，因为它提供了清晰的线程模型和同步原语。我们的目标是构建一个 MySemaphore 类，包含计数器、等待队列，并保证 wait 和 signal 操作的原子性。
设计思路如下：我们需要一个整数 value 作为计数器，一个等待队列 queue 用于存放被阻塞的线程，以及一个机制来保证操作的原子性。在 Python 中，我们可以使用 threading.Condition，它内部基于 Lock，并提供了等待和通知功能，适合实现信号量。Condition 确保了在修改共享状态时的互斥访问，并支持线程的阻塞和唤醒。
以下是 MySemaphore 类的代码实现：

```
import threading

class MySemaphore:
    def __init__(self, value=1):
        self.value = value
        self.condition = threading.Condition()

    def wait(self):
        with self.condition:
            while self.value == 0:
                self.condition.wait()
            self.value -= 1

    def signal(self):
        with self.condition:
            self.value += 1
            self.condition.notify()
```

在构造函数中，我们初始化 value 为给定值（默认为一），并创建一个 Condition 对象。wait 方法使用 with self.condition 语句获取锁，确保原子性。如果 value 大于零，则直接减一并返回；如果 value 等于零，则调用 condition.wait() 阻塞当前线程。这里使用 while 循环而非 if 语句，是为了处理伪唤醒问题：线

程可能被意外唤醒，因此需要重新检查条件。signal 方法同样在锁保护下执行，它将 value 加一，并调用 condition.notify() 唤醒一个等待线程。

关键点在于，wait 方法在阻塞线程前会释放锁，这是为了避免死锁。如果不释放锁，其他线程无法执行 signal 操作来改变条件。被唤醒的线程会重新获取锁，并再次检查 value，确保在唤醒后条件仍然满足。这种实现虽然简化，但捕捉了信号量的核心行为。

为了测试我们的实现，我们可以编写一个简单的生产者－消费者程序：

```
import threading
import time

buffer = []
buffer_size = 5
empty = MySemaphore(buffer_size)
full = MySemaphore(0)
mutex = MySemaphore(1)

def producer():
    for i in range(10):
        empty.wait()
        mutex.wait()
        buffer.append(i)
        print(f"Produced␣{i}")
        mutex.signal()
        full.signal()
        time.sleep(0.1)

def consumer():
    for i in range(10):
        full.wait()
        mutex.wait()
        item = buffer.pop(0)
        print(f"Consumed␣{item}")
        mutex.signal()
        empty.signal()
        time.sleep(0.1)

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)
t1.start()
t2.start()
```

```
t1.join()
t2.join()
```

在这个测试程序中，我们定义了一个共享缓冲区 buffer，以及三个 MySemaphore 对象：empty 初始化为缓冲区大小，full 初始化为零，mutex 初始化为一来保护缓冲区。生产者线程循环生产数据，首先等待 empty 信号量（表示有空位），然后获取 mutex 锁，添加数据到缓冲区，释放 mutex，并增加 full 信号量。消费者线程类似，等待 full 信号量（表示有数据），获取 mutex，取出数据，释放 mutex，并增加 empty 信号量。通过添加延时，我们可以观察线程间的协调行为。运行这个程序，应该能看到生产者和消费者交替执行，没有数据竞争或缓冲区溢出。

4 进阶话题与现实中的考量

尽管信号量是强大的并发工具，但它也有局限性。首先，信号量容易出错：wait 和 signal 必须成对出现，且顺序错误可能导致死锁。例如，如果一个线程在持有信号量时发生异常，可能无法释放资源。其次，信号量可能引发优先级反转问题：高优先级线程可能被低优先级线程持有的信号量所阻塞，影响系统实时性。此外，在现代并发编程中，有许多更安全、抽象的替代品，如互斥锁（Mutex）、条件变量（Condition Variable）、通道（Channel，如在 Go 语言中）或 asyncio.Semaphore（在 Python 中）。这些高级原语通常封装了信号量的复杂性，提供更直观的接口。

然而，信号量在现代编程中仍有广泛应用。例如，在限流（Rate Limiting）场景中，信号量可以控制单位时间内的请求数量；在数据库连接池中，信号量管理并发连接数；在操作系统内核中，信号量用于进程同步和资源管理。理解信号量的原理，有助于我们更好地使用这些高级工具，并在需要时实现自定义的同步机制。

信号量是并发编程领域的基石之一，它通过一个计数器及其原子操作来解决资源访问的协调问题。我们回顾了信号量的核心概念：它是一个非负整数计数器，支持 wait 和 signal 操作，有两种基本类型 —— 二进制信号量用于互斥，计数信号量用于同步。通过亲手实现一个简单的 MySemaphore 类，我们不仅理解了其内部机制，还体会了原子性、阻塞和唤醒等关键概念。尽管信号量有局限性，但它在许多场景下仍是有效的工具。掌握信号量，是迈向构建健壮、高效并发系统的重要一步。希望本文能帮助您在并发编程的旅程中更进一步。