信号量（Semaphore）机制

杨岢瑞

Oct 02, 2025

在并发编程的世界中，多个线程或进程同时访问共享资源时，常常会引发数据不一致或资源冲突的问题。想象一个电影院售票场景：如果多个顾客同时尝试购买同一场次的最后一张票，而没有协调机制，可能会导致超售或数据错误。这种问题被称为竞态条件，其核心在于对有限资源的无序竞争。为了解决这类问题，计算机科学家 Edsger Dijkstra 提出了信号量机制，它作为一种经典的同步工具，能够有效管理资源分配并确保线程安全。本文将带领读者从理论概念入手，逐步深入，最终亲手实现一个基本的信号量，并将其应用于实际场景中，从而夯实并发编程的基础。

1　信号量是什么？—— 理论与概念

信号量的核心思想基于一个非负整数计数器，该计数器表示可用资源的数量。它通过两种原子操作来管理资源访问：P 操作（也称为 wait 或 acquire）和 V 操作（也称为 signal 或 release）。P 操作用于尝试获取资源，如果计数器值大于零，则将其减一并允许线程继续执行；如果计数器值为零，则线程被阻塞，直到其他线程执行 V 操作释放资源。V 操作则负责释放资源，将计数器值加一，并唤醒等待队列中的某个线程。这两种操作必须保证原子性，即执行过程中不可被中断，否则会导致同步失效。

信号量主要分为两种类型：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为零或一，常用于实现互斥锁，确保同一时刻只有一个线程可以进入临界区。例如，在保护共享变量时，二进制信号量可以充当门卫角色。计数信号量的计数器值可以是任意非负整数，适用于控制多实例资源的访问，比如数据库连接池中有十个连接，则计数信号量初始值设为十，允许最多十个线程同时使用。信号量的行为规范还包括使用等待队列来管理被阻塞的线程，确保公平性和效率。原子性是信号量正确工作的基石，它依赖于底层硬件或操作系统的支持，防止在多线程环境下出现数据竞争。

2　从零开始实现一个信号量

为了深入理解信号量的内部机制，我们将使用 Java 语言实现一个简单的信号量类。首先，设计类结构，包括成员变量和方法。成员变量包括一个整型值 `value` 表示当前计数，以及一个等待队列 `waitQueue` 用于存储阻塞线程。方法包括构造函数 `Semaphore(int initialValue)`、`wait()` 方法实现 P 操作，以及 `signal()` 方法实现 V 操作。在实现中，我们将使用 `synchronized` 关键字来确保操作的原子性。

以下是 `wait()` 方法的实现代码：

```
public synchronized void wait() {
    value--;
    if (value < 0) {
```

```
        try {
            this.wait();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

在这段代码中，synchronized 关键字确保了对 value 的修改和判断操作是原子的，防止多线程同时执行导致的竞态条件。当线程调用 wait() 方法时，首先将 value 减一，然后检查是否小于零。如果 value 小于零，表示资源不足，当前线程通过 this.wait() 进入等待状态，直到被其他线程唤醒。异常处理部分负责处理线程中断情况，确保程序的健壮性。这里使用 notify() 而非 notifyAll()，是因为通常只需要唤醒一个等待线程，避免不必要的性能开销。

接下来是 signal() 方法的实现代码：

```
public synchronized void signal() {
    value++;
    if (value <= 0) {
        this.notify();
    }
}
```

在 signal() 方法中，同样使用 synchronized 保证原子性。首先将 value 加一，然后检查是否小于等于零。如果条件成立，说明有线程在等待队列中，此时调用 notify() 唤醒其中一个线程。这种实现方式简单高效，但需要注意的是，在复杂场景下，可能需要更精细的队列管理来避免饥饿问题。这种基于单个锁的实现虽然易于理解，但在高并发环境下可能存在性能瓶颈；实际操作系统中的信号量通常依赖硬件原子指令和调度器优化。

3 实战应用 —— 用我们实现的信号量解决经典问题

信号量在实际应用中非常广泛，我们以两个经典场景为例：实现互斥锁和解决生产者-消费者问题。首先，在实现互斥锁时，可以使用一个初始值为一的二进制信号量。线程在进入临界区前调用 wait() 方法获取锁，退出时调用 signal() 方法释放锁。例如，创建一个信号量实例 Semaphore mutex = new Semaphore(1);，线程代码中在临界区前后分别执行 mutex.wait() 和 mutex.signal()，确保同一时间只有一个线程访问共享资源。

另一个经典应用是生产者-消费者问题，其中多个生产者和消费者共享一个有限大小的缓冲区。我们需要三个信号量：emptySlots 表示空槽位数量，初始值为缓冲区大小 N；fullSlots 表示已填充槽位数量，初始值为零；mutex 作为二进制信号量，保护缓冲区的互斥访问。生产者线程在生产物品后，先等待 emptySlots 信号量确保有空位，然后获取 mutex 锁，将物品放入缓冲区，最后释放 mutex 并通知 fullSlots。消费者线程则相反，先等待 fullSlots 信号量确保有物品，然后获取 mutex 锁，取出物品后释放 mutex 并通知 emptySlots。

以下是生产者线程的代码示例：

```
void producer() {
    while (true) {
```

```
        Item item = produceItem();
        emptySlots.wait();
        mutex.wait();
        buffer.add(item);
        mutex.signal();
        fullSlots.signal();
    }
}
```

在这段代码中，emptySlots.wait() 确保缓冲区有空位时才生产，防止溢出；mutex.wait() 和 mutex.signal() 保护对缓冲区的互斥访问，避免数据竞争；fullSlots.signal() 通知消费者有新物品可用。消费者线程的代码类似，通过等待 fullSlots 和 emptySlots 信号量协调生产与消费的节奏。这种设计确保了缓冲区在满时不会生产，空时不会消费，同时保证了线程安全。

4 信号量的陷阱与现代替代方案

尽管信号量是强大的同步工具，但在使用过程中容易陷入一些陷阱。死锁是常见问题，例如当多个线程以不同顺序获取信号量时，可能导致互相等待而无法继续执行。优先级反转则发生在高优先级线程等待低优先级线程持有的信号量时，造成系统响应延迟。此外，编程错误可能导致遗忘唤醒，即本该唤醒的线程未被正确处理，进而引发线程饥饿。

在现代并发编程中，信号量有更成熟的实现和替代方案。例如，Java 标准库提供了 java.util.concurrent.Semaphore 类，它支持公平性和非公平性策略，并集成了更高级的功能。与互斥锁相比，信号量更通用，不仅用于互斥，还能用于同步；而互斥锁严格限制于临界区保护。条件变量则常与互斥锁配合使用，提供更灵活的等待和通知机制，但信号量通过计数器自带状态管理，更适用于资源计数场景。开发者应根据具体需求选择合适工具，并注意测试和调试以避免潜在问题。

信号量作为并发编程的基石，通过计数器模型和原子操作，有效解决了资源同步与互斥问题。从理论到实践，我们不仅理解了其核心概念，还亲手实现了简单的信号量类，并应用于互斥锁和生产者-消费者等经典场景。这种深入的学习方式有助于巩固操作系统和并发编程原理的知识。在实际项目中，建议使用标准库提供的信号量实现，同时注意规避死锁和优先级反转等陷阱。通过不断实践和优化，读者可以更好地掌握并发编程技能，构建高效可靠的系统。