**临界资源**：在同一时间只能被一个进程调用的资源，也称**互斥资源**。

**同步**：保证访问的时序可控性，使调用资源的的顺序合理。

**互斥**：在进程调用临界资源是，不同进程之间要竞争该资源，那么一个进程调用了该资源，另一个进程无法再调用该资源的情形就叫互斥！

**临界区**：代码中涉及到了操作临界资源的代码段叫临界区。

**原子操作**：该操作无法被打断

**同步与互斥的作用**：使访问临界资源时，具有安全性。

**信号量**

1.实质：  
具有一个**等待队列的计数器**，也属于临界资源的一种，要获取信号量资源，则对信号量进行-1操作，要释放信号量资源，则对信号量资源进行+1操作。  
**注：信号量操作是原子操作**

2.信号量实现同步

原理：进程获取临界资源之前，要先获取信号量资源；

若无信号量资源，则该进程阻塞等待，进入等待队列。

若有信号量资源，则对信号量进行P（-1）操作，再获取临界资源。

当临界资源+1时，对应的信号量资源则执行V（+1）操作，然后唤醒在等待队列中等待获取临界资源的进程。

信号量用于线程同步，互斥量用户保护资源的互斥访问。

## 信号量与互斥量的区别

* 互斥量用于线程的互斥，信号线用于线程的同步。
* 互斥量值只能为0/1，信号量值可以为非负整数。信号量可以实现多个同类资源的多线程互斥和同步。
* 互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

## 信号量Semaphore

* 信号量是在多线程环境中，线程间传递信号的一种方式。

### 简单的Semaphore实现

* public class Semaphore {
* private boolean signal = false; //使用signal可以避免信号丢失
* public synchronized void take() {
* this.signal = true;
* this.notify();
* }
* public synchronized void release() throws InterruptedException{
* while(!this.signal) //使用while避免假唤醒
* wait();
* this.signal = false;
* }
* }
* 使用场景
* Semaphore semaphore = new Semaphore();
* SendingThread sender = new SendingThread(semaphore)；
* ReceivingThread receiver = new ReceivingThread(semaphore);
* receiver.start();
* sender.start();
* public class SendingThread {
* Semaphore semaphore = null;
* public SendingThread(Semaphore semaphore){
* this.semaphore = semaphore;
* }
* public void run(){
* while(true){
* //do something, then signal
* this.semaphore.take();
* }
* }
* }
* public class RecevingThread {
* Semaphore semaphore = null;
* public ReceivingThread(Semaphore semaphore){
* this.semaphore = semaphore;
* }
* public void run(){
* while(true){
* this.semaphore.release();
* //receive signal, then do something...
* }
* }
* }

### 可计数的Semaphore

* 上面提到的Semaphore的简单实现并没有计算通过调用take方法所产生信号的数量。可以把它改造成具有计数功能的Semaphore。
* public class CountingSemaphore {
* private int signals = 0;
* public synchronized void take() {
* this.signals++;
* this.notify();
* }
* public synchronized void release() throws InterruptedException{
* while(this.signals == 0)
* wait();
* this.signals--;
* }
* }

### 有上限的Semaphore

* 可以将上面的CountingSemaphore改造成一个信号数量有上限的BoundedSemaphore
* public class BoundedSemaphore {
* private int signals = 0;
* private int bound = 0;
* public BoundedSemaphore(int upperBound){
* this.bound = upperBound;
* }
* public synchronized void take() throws InterruptedException{
* while(this.signals == bound)
* wait();
* this.signals++;
* this.notify();
* }
* public synchronized void release() throws InterruptedException{
* while(this.signals == 0)
* wait();
* this.signals--;
* this.notify();
* }
* }
* 在BoundedSemaphore中，当已经产生的信号数量达到了上限，take方法将阻塞新的信号产生请求，直到某个线程调用release方法后，被阻塞于take方法的线程才能传递自己的信号。

### Java内置的Semaphore

* java.util.concurrent包中有Semaphore的实现，可以设置参数，控制同时访问的个数。  
  下面的Demo中申明了一个只有5个许可的Semaphore，而有20个线程要访问这个资源，通过acquire()和release()获取和释放访问许可。
* final Semaphore semp = new Semaphore(5);
* ExecutorService exec = Executors.newCachedThreadPool();
* for (int index = 0; index < 20; index++) {
* final int NO = index;
* Runnable run = new Runnable() {
* public void run() {
* try {
* // 获取许可
* semp.acquire();
* System.out.println("Accessing: " + NO);
* Thread.sleep((long) (Math.random() \* 10000));
* // 访问完后，释放
* semp.release();
* System.out.println("-----------------" + semp.availablePermits());
* } catch (InterruptedException e) {
* e.printStackTrace();
* }
* }
* };
* exec.execute(run);
* }
* exec.shutdown();

## 互斥量Mutex

* 互斥量：提供对资源的独占访问，只能为0/1，如果某一个资源同时只能允许一个访问者对其访问，可以使用互斥量控制线程对其访问。
* 互斥量实现：
* public class Mutex {
* private boolean isLocked = false;
* public synchronized void lock() {
* while(this.isLocked) //使用while可以避免线程 假唤醒
* wait();
* this.isLocked= true;
* }
* }
* public synchronized void unlock() throws InterruptedException{
* this.isLocked= false;
* this.notify();
* }
* }
* 在Mutex中，我们添加了一个signal用于保存信号。
* 将互斥量当作**锁**来使用：
* Mutex mutex = new Mutex();
* mutex.lock();
* ...
* //临界区
* mutex.unlock();
* 互斥量的加锁和解锁必须由同一个线程分别对应使用。