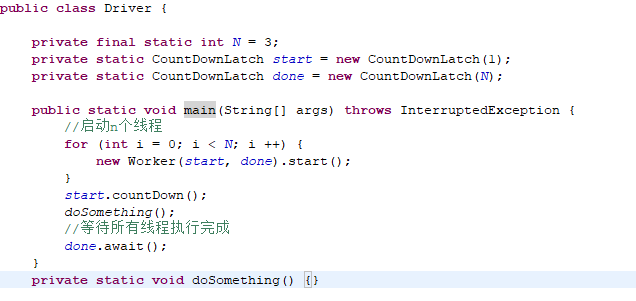
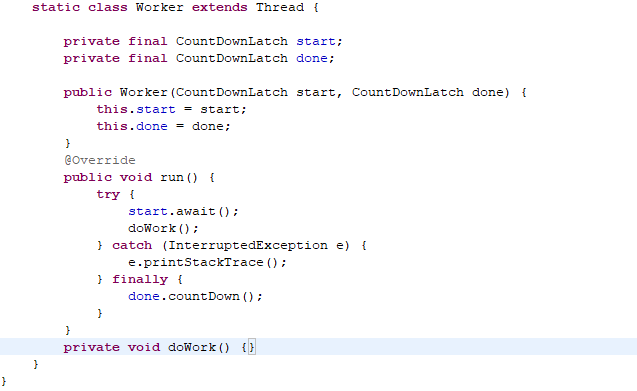
# CountDownLatch

先来看一个CountDownLatch的例子：





在主线成中开启了N个线程，所有子线程阻塞在start.await()上，直到主线程调用start.countDown上唤醒子线程。然后主线程继续执行，最后阻塞在done.await()上，等待所有子线程完成任务后被唤醒。

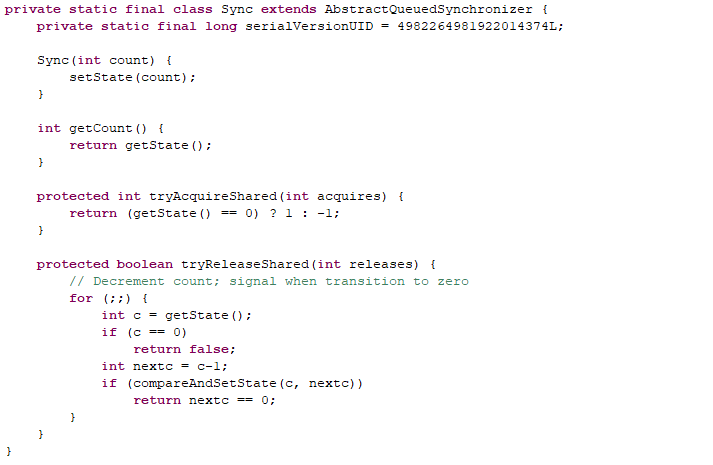
以下是官方给出的对于CountDownLatch的说明：



允许一个或多个线程阻塞直到在其他线程中的一组操作完成之后被唤醒。可以使用一个int类型的状态变量count初始化CountDownLatch，之后调用await的线程会一直阻塞直到count减少为0，通过调用countDown减少count。count减少为0后，之后调用await的线程会立即返回，也就说CountDownLatch只能使用一次，如果希望count的状态可以被重置，可以使用另一个同步工具类CyclicBarrier。

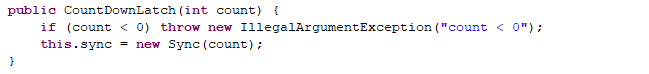
# 1源码分析

CountDownLatch使用AQS同步器实现了所有的操作，下面主要看一下CountDownLatch的内部同步Sync的实现。



可以看到Sync实现了共享模式下的同步。当同步状态state为0时，tryAcquireShared返回1，否则返回-1，也就是说只有state减少为0时，等待的线程才会被唤醒。tryReleaseShared首先判断state==0，如果state已经减少为0，立即返回false，否则减少state。

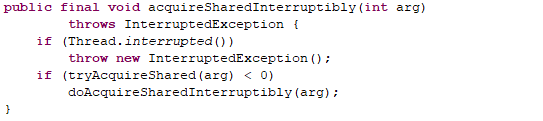
## 1.1构造器

 count的值就是同步状态state的值。

## 1.2await



await调用了AQS的acquireSharedInterruptibly，而acquireSharedInterruptibly调用了tryAcquireShared。



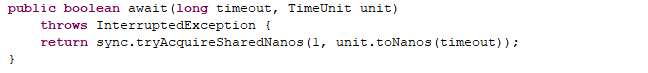
doAcquireSharedInterruptibly方法会阻塞当前线程。tryAcquireShared只有当state为0时才会返回1，否则返回-1，所以，当还没有调用countDown减少state时，调用await的线程会一直阻塞。

## 1.3countDown



releaseShared调用tryReleaseShared，每次调用countDown方法，state减少1。

## 1.4超时等待await



相比await方法，增加了超时等待特性，在timeout时间内，state一直得不到释放，该方法会返回false，不再等待state释放。

## 1.5getCount



返回当前同步状态。

# 内存一致性

当count减少为0时，线程在调用countDown之前的动作happen-before其他线程在await成功返回。也就是说线程在调用countDown之前对内存所做的修改对在await成功返回的线程可见。

# 用途

CountDownLatch可以用作开关，例如在前文给出的例子，当所有子线程都准备好时，才开始让所有子线程执行。

CountDownLatch还可以用来解决问题拆分。如果问题可以被拆分为n个互不相关的子问题，可以为每个子问题创建一个线程来执行，最后等待所有子问题解决之后返回结果。以下是代码实现：

