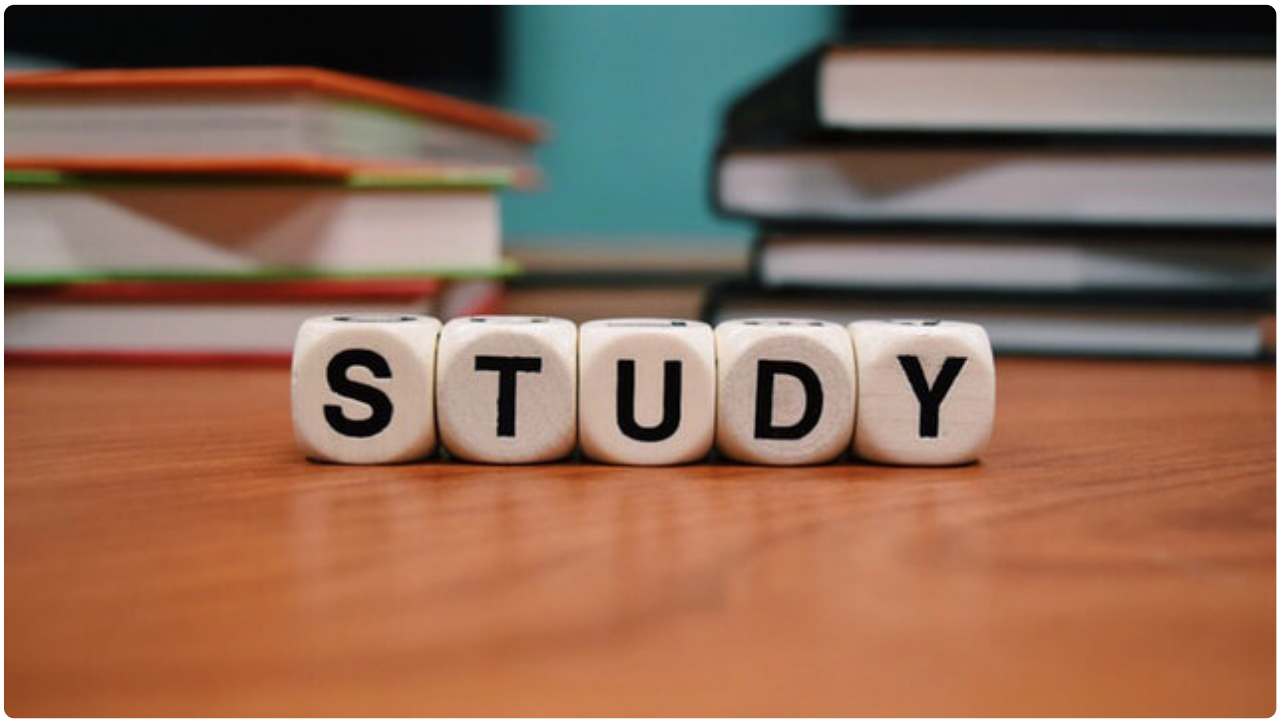
# 27 倒数计时开始，三、二、一—CountDownLatch详解

更新时间：2019-11-27 11:05:50

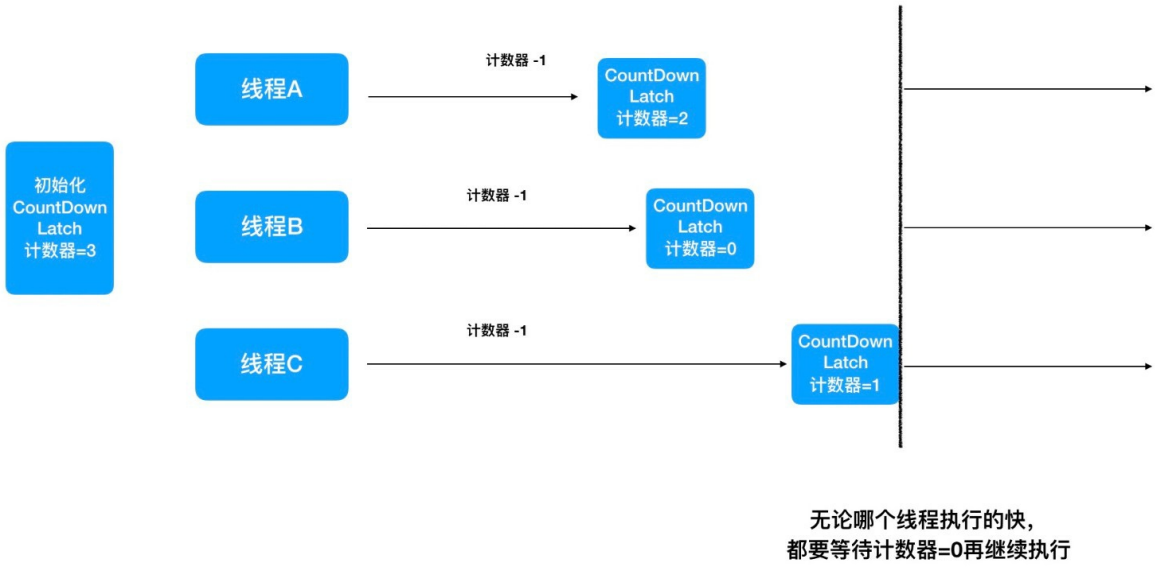



> 时间像海绵里的水，只要你愿意挤，总还是有的。
>
> ——鲁迅

本节开始我们学习一些新的东西，不再局限于线程、锁、并发容器这些内容。JCU 包中提供了一些工具，用于线程间的协调。这些工具并不是每个并发编程的场景都需要使用。大部分场景通过 wait/nofity 或者 join 等操作就可以解决。但是在一些特定的场景下，我们则需要借助这些工具来解决问题。本节我们先来学习 CountDownLatch。

## 1、理解 CountDownLatch

从字面理解 CountDownLatch，意思是倒数门闩。它的作用是多个线程做汇聚。主线程开启了 A、B、C 三个线程做不同的事情，但是主线程需要等待 A、B、C 三个线程全部完成后才能继续后面的步骤。此时就需要 CountDownLatch 出马了。CountDownLatch 会阻塞主线程，直到计数走到 0，门闩才会打开，主线程继续执行。而计数递减是每个线程自己操作 CountDownLatch 对象实现的。如下图：




这种场景在我们的生活中十分常见。比如篮球比赛中，作为控球后卫，如果没有快攻机会，那就需要等到中锋、大前锋、小前锋、得分后卫都跑到位了，我才能决定怎么组织进攻。又比如我们报团去旅游，必须所有人都到机场了，才能一起出发。

对于我们的程序来说这种场景也挺多的，比如你的订单信息可能需要从多个微服务取得数据，汇总后加工才返回给前台。此时从多个微服务取得数据可以是多个子线程来完成。

对于以上场景，都是 CountDownLatch 的用武之地。

## 2、如何使用 CountDownLatch

我们来模拟打篮球的例子，主线程假如是控球后卫，我们看一下如果不用 CountDownLatch 会有什么问题：

```
public static void main(String[] args) throws InterruptedException {

    System.out.println("控球后卫到位！等待所有位置球员到位！");

    new Thread(()->{
        System.out.println("得分后卫到位！");
    }).start();

    new Thread(()->{
        System.out.println("中锋到位！");
    }).start();

    new Thread(()->{
        System.out.println("大前锋到位！");
    }).start();

    new Thread(()->{
        System.out.println("小前锋到位！");
    }).start();

    System.out.println("全部到位，开始进攻！");
}
```

输出为：

```
控球后卫到位！等待所有位置球员到位！
得分后卫到位！
中锋到位！
大前锋到位！
全部到位，开始进攻！
小前锋到位！
```

可以看到小前锋还没有到位，就开始进攻了。这显然和需求不符。出现这种结果也很好理解，因为代码中控球后卫并没有等每个球员的线程到位，就开始进攻了。

正确的姿势应该如下：

```
public class Client {
    private static  final CountDownLatch countDownLatch = new CountDownLatch(5);
    public static void main(String[] args) throws InterruptedException {

        System.out.println("控球后卫到位！等待所有位置球员到位！");
        countDownLatch.countDown();

        new Thread(()->{
            System.out.println("得分后卫到位！");
            countDownLatch.countDown();
        }).start();

        new Thread(()->{
            System.out.println("中锋到位！");
            countDownLatch.countDown();
        }).start();

        new Thread(()->{
            System.out.println("大前锋到位！");
            countDownLatch.countDown();
        }).start();

        new Thread(()->{
            System.out.println("小前锋到位！");
            countDownLatch.countDown();
        }).start();

        countDownLatch.await();

        System.out.print("全部到位，开始进攻！");
    }
}
```

首先声明声明了一个 countDownLatch 对象，由于有5名球员，所以传入 count=5。每个球员的线程在球员到位后，都会执行 countDownLatch.countDown()，这个方法可以理解为我们把初始值的计数数量5做递减。当减到零时才会执行 countDownLatch.await(); 后面的代码。countDownLatch.await() 就是我们的门闩，这行代码做的是锁门操作，而每次 countDown()，调用5次后，门闩打开，后面的代码才被执行。

这段代码输出如下：

```
控球后卫到位！等待所有位置球员到位！
得分后卫到位！
中锋到位！
大前锋到位！
小前锋到位！
全部到位，开始进攻！
```

可以看出完全符合我们的预期，如果你还对此表示怀疑，那么你可以在某个线程中让其 sleep 上几秒，再看看是否还是全部到位才开始进攻。

# 3、CountDownLatch 的原理解析

CountDownLatch 内部其实还是借助 AQS 实现的。它内部实现了 AbstractQueuedSynchronizer。使用 AQS 的 state 变量来存储计数器的值，初始化 CountDownLatch，实际在初始化 state 值。

### 3.1 构造函数

我们看其构造函数：

```
public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    this.sync = new Sync(count);
}
```

```
Sync(int count) {
    setState(count);
}
```

```
protected final void setState(int newState) {
    state = newState;
}
```

三个方法串起来看，发现最后就是把传入的 count 设置给了 state。

**3.2 await 方法**

await 方法会阻塞当前线程，代码如下：

```
public void await() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
```

调用了 AQS 的方法：

```
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
```

尝试获取共享锁 tryAcquireShared，如果不能获取进入等待队列。

tryAcquireShared 方法由 CountDownLatch 的内部类 Sync 实现，如下：

```
protected int tryAcquireShared(int acquires) {
    return (getState() == 0) ? 1 : -1;
}
```

可以看到如果 state 为0就直接返回了，但如果不为零，才进入等待队列。调用 tryAcquireShared 仅仅检查 state 值，而不会对其减 1，可以看到传入的参数 acquires根本没有用。

我们再看看 countDown 方法。

**3.3 countDown 方法**

这个方法会对 state 递减。当计数器减为 0 时，所有阻塞的线程都被唤醒。代码如下：

```
public void countDown() {
    sync.releaseShared(1);
}
```

可见其也是通过对自己的 AQS 子类调用 releaseShared 方法：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

而在这个方法里，tryReleaseShared 是由子类实现的，也就是 countDown 中的 Sync 类，实现代码如下：

```
protected boolean tryReleaseShared(int releases) {
    // Decrement count; signal when transition to zero
    for (;;) {
        int c = getState();
        if (c == 0)
            return false;
        int nextc = c-1;
        if (compareAndSetState(c, nextc))
            return nextc == 0;
    }
}
```

以上代码在自旋中，通过 CAS 的方式对 state 值-1，如果 c-1 后等于 0，说明计数到 0。那么 releaseShared 中会调用 doReleaseShared()，让 AQS 释放资源出来。

以上对做 CountDownLatch 源代码做了简单的分析，可以看出主要是使用 AQS 来实现。通过阻塞队列阻塞线程。然后通过 state 值的初始化和递减，实现 state 为 0 时，激活阻塞的线程。

## 4、总结

CountDownLatch 有其一定的应用场景，对于多线程协调和串起流程有很大的帮助。我们在多线程开发中，可以留意是否有类似的场景，能够通过 CountDownLatch 来解决。CountDownLatch 自身也有一定的局限性，它只能被使用一次，而不能被恢复再次使用。下一节我们讲学习 CyclicBarrier，它可以重置以重复使用。

}

26不让我进门，我就在门口一直等！—BlockingQueue和ArrayBlockingQueue

28 人齐了，一起行动—CyclicBarrier详解