30 限量供应，不好意思您来晚了—Semaphore详解

更新时间：2019-12-10 09:50:17

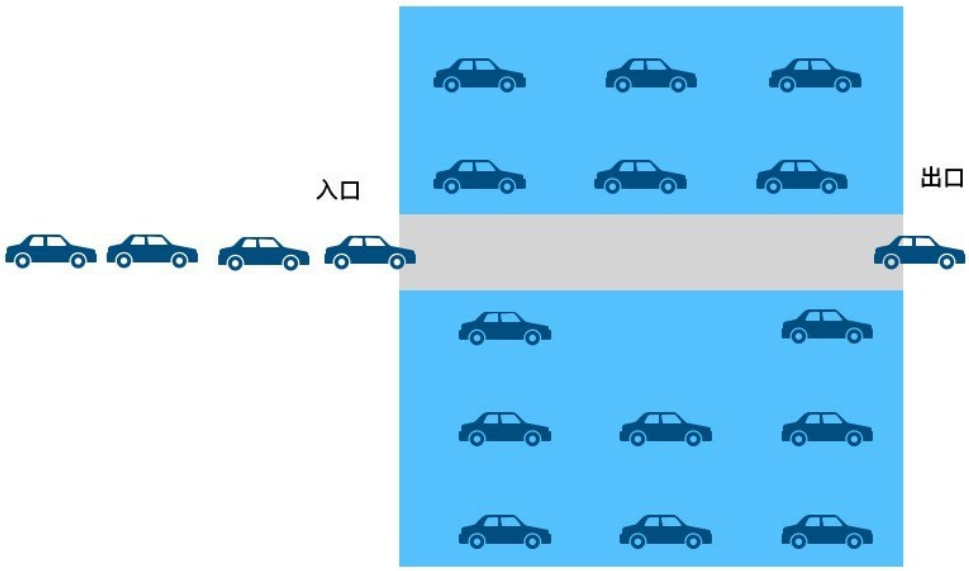
> 耐心和恒心总会得到报酬的。
>
> ——爱因斯坦

前几节我们学习了几种多线程线程同步工具，有一次性使用的倒数计数的 CountDownLatch，有循环使用的 CyclicBarrier，还有可以做数据交换的 Exchanger。今天我们再讲解一种同步工具 Semaphore。

1、Semaphore 简介

Semaphore 是信号量的意思，通过信号量可以对同一资源访问做数量的限制。我们回忆一下无论是 Synchronized 还是 ReentrantLock 都是限制每次只有一个线程并发访问资源。而信号量可以控制更多数量的线程访问资源，但是不能超过信号量的准入数。

这就像停车场，如果停车位资源不紧张，车可以随便进。但是当停车场停满了车，那么不好意思，您来晚了。你只能在入口等待。出去几辆，才能放几辆进来。这个例子中，停车场就是共享资源，停车位的数量就是信号量准入数。而每辆车就是一个线程。停车场控制系统就是今天要学习的 Semaphore。



停车场停满后，只能走一辆，才能进一辆

下面我们看看如何用代码实现以上的例子。

2、如何使用 Semaphore

下面的代码模拟 10 个车位的停车场，今天不知道附近有什么活动，突然过来了 500 辆车要停入停车场。这样必然会造成排队，前面的车出去一辆后面的车才能进来一辆。代码如下：

```
public class Client {
    public static void main(String[] args) {
        //用于生成随机停车时长
        Random random = new Random();
        //用Semaphore模拟有10个停车位的停车场管理系统
        final Semaphore parkingSystem = new Semaphore(10);

        //模拟500辆汽车来停车
        IntStream.range(0,500).forEach(i->{
            new Thread(()->{
                //取得到达停车场的时间
                Long startWaitTime = System.currentTimeMillis();
                System.out.println("第"+(i+1)+"辆汽车来到车库");

                //等待停车场系统控制抬杆。如果还有空位，立即抬杆，否则一直等到有空位才抬杆
                try {
                  //acquire方法用于获取资源，这里模拟发出抬杆放行的请求
                    parkingSystem.acquire();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //已经抬赶，计算等待时长
                Long waitingTime = (System.currentTimeMillis() - startWaitTime)/1000;
                System.out.println("第"+(i+1)+"辆汽车等待"+waitingTime+"毫秒后进入车库");
                //通过sleep模拟停车时长
                int parkingTime = random.nextInt(10)+2;
                try {
                    TimeUnit.SECONDS.sleep(parkingTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
               //release方法用于释放资源，模拟驶出停车场
                parkingSystem.release();
                System.out.println("第"+(i+1)+"辆汽车停车"+parkingTime+"毫秒离开车库");
            }).start();
        });
    }
}
```

输出比较多，我们先看开始的输出：

```
第1辆汽车来到车库
第3辆汽车来到车库
第2辆汽车来到车库
第3辆汽车等待0毫秒后进入车库
第5辆汽车来到车库
第4辆汽车来到车库
第1辆汽车等待0毫秒后进入车库
第6辆汽车来到车库
第4辆汽车等待0毫秒后进入车库
第5辆汽车等待0毫秒后进入车库
第2辆汽车等待0毫秒后进入车库
第7辆汽车来到车库
第6辆汽车等待0毫秒后进入车库
第9辆汽车来到车库
第9辆汽车等待0毫秒后进入车库
第7辆汽车等待0毫秒后进入车库
第8辆汽车来到车库
第8辆汽车等待0毫秒后进入车库
第10辆汽车来到车库
第10辆汽车等待0毫秒后进入车库
第11辆汽车来到车库
第12辆汽车来到车库
第13辆汽车来到车库
第14辆汽车来到车库
.....
```

可以看到前 10 辆车进入车库都是不需要等待的，从第 11 辆车开始已经无法进入车库了。我们继续看后买面的输出：

```
.....
第495辆汽车来到车库
第496辆汽车来到车库
第497辆汽车来到车库
第498辆汽车来到车库
第499辆汽车来到车库
第500辆汽车来到车库
.....
```

由于汽车线程启动没有间隔，也就意味着 500 辆车瞬间挤压到停车场门口，等待入场。继续看下面的输出：

```
第3辆汽车停车4毫秒离开车库
第11辆汽车等待4毫秒后进入车库
第6辆汽车停车5毫秒离开车库
第12辆汽车等待5毫秒后进入车库
第5辆汽车停车7毫秒离开车库
第2辆汽车停车7毫秒离开车库
第13辆汽车等待7毫秒后进入车库
第14辆汽车等待7毫秒后进入车库
第10辆汽车停车8毫秒离开车库
第15辆汽车等待8毫秒后进入车库
第1辆汽车停车9毫秒离开车库
第4辆汽车停车9毫秒离开车库
```

可以看到第一批进入车库的汽车，逐步离开车库。后面排队的车陆续进来。另外也可以观察到，离开汽车的停车时长和进入汽车的等待时长是一致，这也证明了只有走了一辆，才能进入一辆。

不过由于多线程输出日志，所以顺序上并不一定是一辆离开，一辆进入。但实际运行情况确实是走了一辆才放入一辆。

Semaphore 可以选择竞争策略是否公平。构造 Semaphore 时可以传入第二个参数，如下面代码所示：

```
final Semaphore parkingSystem = new Semaphore(10,true);
```

如果构造时传入第二个参数为 true，那么就是公平的，不传默认也是公平的。这一点通过以上例子的输出也有所体现。

3、Semaphore 源码分析

我们先看 Semaphore 的构造方法：

```
public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new NonfairSync(permits);
}
```

根据传入 fair 的不同，选择 sync 对象是公平还是不公平。FairSync 和 NonfairSync 都是 Semaphore 内部静态类，继承自 AQS。Semaphore 也是借助 AQS 来实现的。

我们再看 acquire 方法代码：

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

调用了 AQS 中的 acquireSharedInterruptibly 方法。继续看此方法代码：

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

核心是先调用 tryAcquireShared，尝试获取，如果获取失败则调用 doAcquireSharedInterruptibly，自旋进入等待队列，如果排到自己，那么再次尝试调用 tryAcquireShared。这个方法之前详细分析过，这里就不再展开来讲。

接下来我们看看尝试获取资源的方法 tryAcquireShared，它的实现在 Semaphore 内部静态类 Sync 中，如下：

```
protected int tryAcquireShared(int acquires) {
    for (;;) {
      //看是否有更早等待的线程，如果有，获取失败
        if (hasQueuedPredecessors())
            return -1;
      //查询剩余的信号量准入数量
        int available = getState();
      //查询剩余的信号量准入数量，看是否满足想要获取的数量
        int remaining = available - acquires;
      //剩余的数量>0则会通过CAS的方式刷新剩余信号量。并且返回剩余信号量。
        if (remaining < 0 ||
            compareAndSetState(available, remaining))
            return remaining;
    }
}
```

下面我们再来看一下 release 的源代码：

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

可以看到每次释放数量为 1。另外还有可以传入 release 资源数量的重载方法。

releaseShared 代码如下：

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

调用 tryReleaseShared 方法进行资源释放，然后调用 doReleaseShared 来发送信号通知下一个节点来获取资源。tryReleaseShared 的实现也在 Semaphore 内部静态类 Sync 中，如下：

```
protected final boolean tryReleaseShared(int releases) {
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            return true;
    }
}
```

获取当前剩余信号量计数，然后把释放的资源数量加回来。最后通过 CAS 方式刷新信号量的计数。

4、总结

信号量用来控制共享资源的访问数量。所以很适合控制有 “池” 概念的资源访问。因为池的意思就是池内有有限数量的资源可以使用。如果在池这个层面抽象为一个资源来对待，那么使用 Semaphore 来做控制就非常合适。

}

29 一手交钱，一手交货—Exchanger详解

31 凭票取餐—Future模式详解