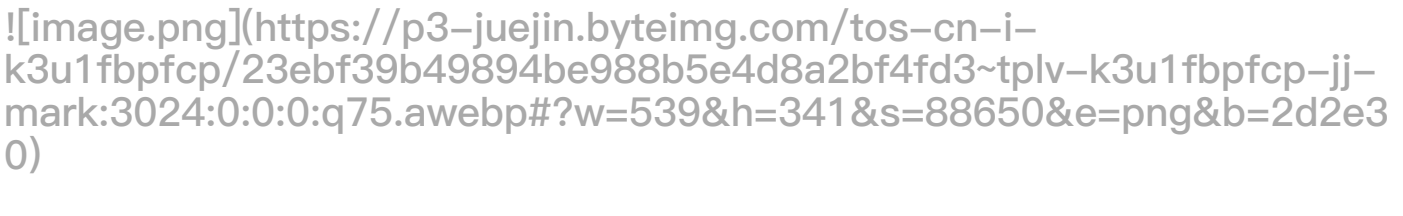
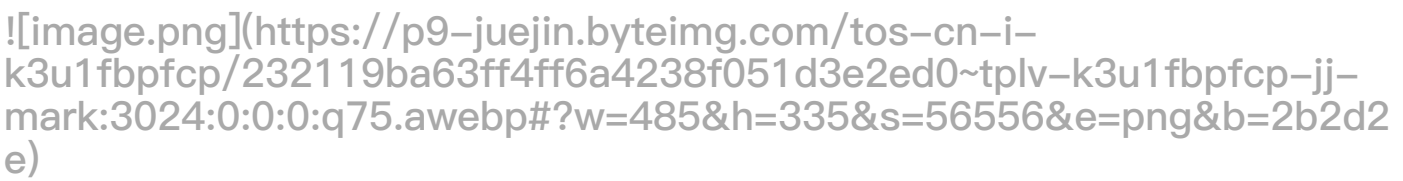
```
se();
        }
    }
    private void log(String msg){
        if (name == null) {
            name = Thread.currentThread().getName();
        }
        System.out.println(name + " " + msg);
    }
}
```



这段代码是比较典型的 Semaphore 示例，其逻辑是，线程试图获得工作允许，得到许可则进行任务，然后释放许可，这时等待许可的其他线程，就可获得许可进入工作状态，直到全部处理结束。编译运行，我们就能看到 Semaphore 的允许机制对工作线程的限制。但是，从具体节奏来看，其实并不符合我们前面场景的需求，因为本例中 Semaphore 的用法实际是保证，一直有 5 个人可以试图乘车，如果有 1 个人出发了，立即就有排队的人获得许可，而这并不完全符合我们前面的要求。

```
import java.util.concurrent.Semaphore;
public class AbnormalSemaphoreSample {
    public static void main(String[] args) throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new MyWorker(semaphore));
            t.start();
        }
        System.out.println("Action...GO!");
        semaphore.release(5);
        System.out.println("Wait for permits off");
        while (semaphore.availablePermits()!=0) {
            Thread.sleep(100L);
        }
        System.out.println("Action...GO again!");
```

```
        semaphore.release(5);
    }
}
class MyWorker implements Runnable {
    private Semaphore semaphore;
    public MyWorker(Semaphore semaphore) {
        this.semaphore = semaphore;
    }
    @Override
    public void run() {
        try {
            semaphore.acquire();
            System.out.println("Executed!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```



注意，上面的代码，更侧重的是演示 Semaphore 的功能以及局限性，其实有很多线程编程中的反实践，比如使用了 sleep 来协调任务执行，而且使用轮询调用 availalePermits 来检测信号量获取情况，这都是很低效并且脆弱的，通常只是用在测试或者诊断场景。
总的来说，我们可以看出 Semaphore 就是个计数器，其基本逻辑基于 acquire/release，并没有太复杂的同步逻辑。
如果 Semaphore 的数值被初始化为 1，那么一个线程就可以通过 acquire 进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比如互斥

```
public void run() {
            try {
                for (int i=0; i<3 ; i++) {
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierSample {

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(5, () ->
System.out.println("Action...GO again!"));
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new CyclicWorker(barrier));
            t.start();
        }
    }

    static class CyclicWorker implements Runnable {
        private CyclicBarrier barrier;
        public CyclicWorker(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for (int i=0; i<3 ; i++) {
                    System.out.println("Executed!");
                    barrier.await();
                }
            } catch (BrokenBarrierException | InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

为了让输出更能表达运行时序，我使用了 CyclicBarrier 特有的 barrierAction，当屏障被触发时，Java 会自动调度该动作。因为 CyclicBarrier 会自动进行重置，所以这个逻辑其实可以非常自然的支持更多排队人数。其编译输出如下：


```

mark:3024:0:0:0:q75.awebp#?w=593&h=489&s=132787&e=png&b=2e3031)

并发包里提供的线程安全 Map、List 和 Set

如果我们的应用侧重于 Map 放入或者获取的速度，而不在乎顺序，大多推荐使用 ConcurrentHashMap，反之则使用 ConcurrentSkipListMap；如果我们需要对大量数据进行非常频繁地修改，ConcurrentSkipListMap 也可能表现出优势

原文链接: https://juejin.cn/post/7379786670252802059