1. Semaphore 是什么?

Semaphore 字面意思是信号量的意思，它的作用是控制访问特定资源的线程数目。

2. 怎么使用 Semaphore?

2.1 构造方法

```
public Semaphore(int permits)
public Semaphore(int permits, boolean fair)
```

- permits 表示许可线程的数量
- fair 表示公平性，如果这个设为 true 的话，下次执行的线程会是等待最久的线程

2.2 重要方法

```
public void acquire() throws InterruptedException
public void release()
tryAcquire（long timeout, TimeUnit unit）
```

- acquire() 表示阻塞并获取许可
- release() 表示释放许可

2.3 基本使用

2.3.1 需求场景

资源访问，服务限流。

2.3.2 代码实现

```
public class SemaphoreSample {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(2);
        for (int i=0;i<5;i++){
            new Thread(new Task(semaphore,"yangguo+"+i)).start();
        }
    }

    static class Task extends Thread{
        Semaphore semaphore;
```

```
        public Task(Semaphore semaphore,String tname){
            this.semaphore = semaphore;
            this.setName(tname);
        }

        public void run() {
            try {
                semaphore.acquire();
System.out.println(Thread.currentThread().getName()+":aquire() at
time:"+System.currentTimeMillis());
                Thread.sleep(1000);
                semaphore.release();
System.out.println(Thread.currentThread().getName()+":aquire() at
time:"+System.currentTimeMillis());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
}
```

打印结果:

```
Thread-3:aquire() at time:1563096128901
Thread-1:aquire() at time:1563096128901
Thread-1:aquire() at time:1563096129903
Thread-7:aquire() at time:1563096129903
Thread-5:aquire() at time:1563096129903
Thread-3:aquire() at time:1563096129903
Thread-7:aquire() at time:1563096130903
Thread-5:aquire() at time:1563096130903
Thread-9:aquire() at time:1563096130903
Thread-9:aquire() at time:1563096131903
```

从打印结果可以看出，一次只有两个线程执行 acquire()，只有线程进行 release() 方法后才会有别的线程执行 acquire()。

CountDownLatch使用及应用场景例子

CountDownLatch是什么?

CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行。

CountDownLatch如何工作?

CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

API

```
CountDownLatch.countDown()
CountDownLatch.await();
```

CountDownLatch应用场景例子

比如陪媳妇去看病。

医院里边排队的人很多，如果一个人的话，要先看大夫，看完大夫再去排队交钱取药。

现在我们是双核，可以同时做这两个事（多线程）。

假设看大夫花3秒钟，排队交费取药花5秒钟。我们同时搞的话，5秒钟我们就能完成，然后一起回家（回到主线程）。

代码如下:

```
/**
 * 看大夫任务
 */
public class SeeDoctorTask implements Runnable {
    private CountDownLatch countDownLatch;

    public SeeDoctorTask(CountDownLatch countDownLatch){
        this.countDownLatch = countDownLatch;
    }

    public void run() {
```

```
            try {
                System.out.println("开始看医生");
                Thread.sleep(3000);
                System.out.println("看医生结束，准备离开病房");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                if (countDownLatch != null)
                    countDownLatch.countDown();
            }
        }

    }

    /**
     * 排队的任务
     */
    public class QueueTask implements Runnable {

        private CountDownLatch countDownLatch;

        public QueueTask(CountDownLatch countDownLatch){
            this.countDownLatch = countDownLatch;
        }
        public void run() {
            try {
                System.out.println("开始在医院药房排队买药....");
                Thread.sleep(5000);
                System.out.println("排队成功，可以开始缴费买药");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                if (countDownLatch != null)
                    countDownLatch.countDown();
            }
        }
```

```
}

/**
 * 配媳妇去看病，轮到媳妇看大夫时
 * 我就开始去排队准备交钱了。
 */
public class CountDownLaunchSample {

    public static void main(String[] args) throws
InterruptedException {
        long now = System.currentTimeMillis();
        CountDownLatch countDownLatch = new CountDownLatch(2);

        new Thread(new SeeDoctorTask(countDownLatch)).start();
        new Thread(new QueueTask(countDownLatch)).start();
        //等待线程池中的2个任务执行完毕，否则一直
        countDownLatch.await();
        System.out.println("over, 回家 cost:"+
(System.currentTimeMillis()-now));
    }
}
```

CyclicBarrier

栅栏屏障，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。

CyclicBarrier默认的构造方法是CyclicBarrier（int parties），其参数表示屏障拦截的线程数量，每个线程调用await方法告CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

API

```
cyclicBarrier.await();
```

应用场景

可以用于多线程计算数据，最后合并计算结果的场景。例如，用一个Excel保存了用户所有银行流水，每个Sheet保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日

均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

示例代码：

```
public class CyclicBarrierTest implements Runnable {
    private CyclicBarrier cyclicBarrier;
    private int index ;

    public CyclicBarrierTest(CyclicBarrier cyclicBarrier, int
index) {
        this.cyclicBarrier = cyclicBarrier;
        this.index = index;
    }

    public void run() {
        try {
            System.out.println("index: " + index);
            index--;
            cyclicBarrier.await();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(11, new
Runnable()                                                    {
            public void run() {
                System.out.println("所有特工到达屏障，准备开始执行秘密任
务");
            }
        });
        for (int i = 0; i < 10; i++) {
            new Thread(new CyclicBarrierTest(cyclicBarrier,
i)).start();
```

```
            }
            cyclicBarrier.await();
            System.out.println("全部到达屏障....");
        }
}
```

Executors

主要用来创建线程池，代理了线程池的创建，使得你的创建入口参数变得简单

重要方法

- newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
- newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
- newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
- newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

Exchanger

当一个线程运行到exchange()方法时会阻塞，另一个线程运行到exchange()时，二者交换数据，然后执行后面的程序。

应用场景

极少，大家了解即可

```
public class ExchangerTest {

    public static void main(String []args) {
        final Exchanger<Integer> exchanger = new
Exchanger<Integer>();
        for(int i = 0 ; i < 10 ; i++) {
            final Integer num = i;
            new Thread() {
                public void run() {
```

```
                    System.out.println("我是线程：Thread_" +
this.getName() + "我的数据是：" + num);
                    try {
                        Integer exchangeNum =
exchanger.exchange(num);
                        Thread.sleep(1000);
                        System.out.println("我是线程：Thread_" +
this.getName() + "我原先的数据为：" + num + " , 交换后的数据为：" +
exchangeNum);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }.start();
        }
    }
}
```

源码原理请见课堂分析：