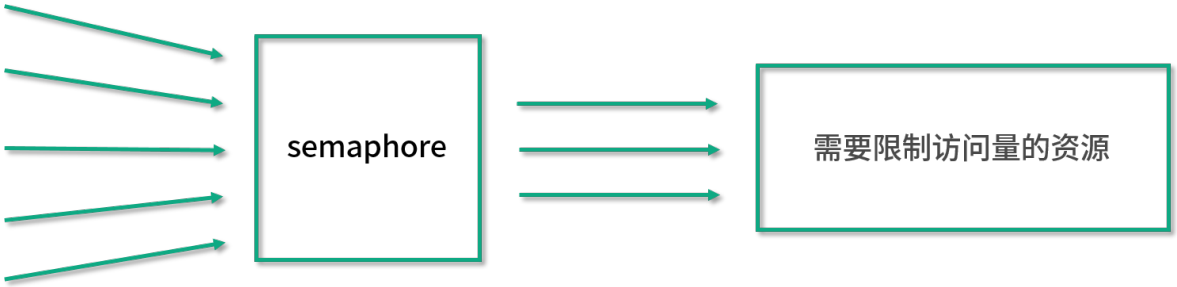
信号量（Semaphore）讲解：



信号量就是控制限制并发访问量，如上图，要访问资源，得先经过信号量，而线程去访问共享资源前，必须先拿到许可证（信号量固定了许可证的个数），如果许可证发放完了，再有线程想要获得许可证，那么这个线程就必须等待，线程执行完后释放许可证，这时才可以继续访问。

代码示例：

```
1  public class SemaphoreDemo2 {
2
3      static Semaphore semaphore = new Semaphore(3);
4
5      public static void main(String[] args) {
6          ExecutorService service = Executors.newFixedThreadPool(50);
7          for (int i = 0; i < 1000; i++) {
8              service.submit(new Task());
9          }
10         service.shutdown();
11     }
12
13     static class Task implements Runnable {
14
15         @Override
16         public void run() {
17             try {
18                 semaphore.acquire();
19             } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "拿到了许可证，花费2秒执行慢服务");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("慢服务执行完毕，" + Thread.currentThread().getName() + "释放了许可证");
            semaphore.release();
        }
    }
}
```

Semaphore相关方法：

```
public Semaphore(int permits, boolean fair);//初始化一个信号量,第一个参数是许可证的数量，另一个参数是是否公平。
acquire();//获取许可证，支持中断的：假设这个线程被中断了，那么它就会跳出 acquire() 方法，不再继续尝试获取了。
acquireUninterruptibly()//获取许可证，不支持中断
release()//释放许可证
public boolean tryAcquire();//尝试获取许可证，相当于看看现在有没有空闲的许可证，如果有就获取，如果现在获取不到也没关系，不必陷入阻塞，可以去做别的事。
public boolean tryAcquire(long timeout, TimeUnit unit);//比如传入了 3 秒钟，则意味着最多等待 3 秒钟，如果等待期间获取到了许可证，则往下继续执行；如果超时时间到，依然获取不到许可证，它就认为获取失败，且返回 false。
availablePermits();//这个方法用来查询可用许可证的数量，返回一个整型的结果。
```

Semaphore相关方法特殊用法：

比如 semaphore.acquire(2)，里面传入参数 2，这就叫一次性获取两个许可证。

为什么要这样做呢？我们列举一个使用场景。比如说第一个任务 A（Task A ）会调用很耗资源的方法一 method1()，而任务 B 调用的是方法二 method 2，但这个方法不是特别消耗资源。在这种情况下，假设我们一共有 5 个许可证，只能允许同时有 1 个线程调用方法一，或者同时最多有 5 个线程调用方法二，但是方法一和方法二不能同时被调用。

所以，我们就要求 Task A 在执行之前要一次性获取到 5 个许可证才能执行，而 Task B 只需要获取一个许可证就可以执行了。这样就避免了任务 A 和 B 同时运行，同时又很好的兼顾了效率，不至于同时只允许一个线程访问方法二，那样的话也存在浪费资源的情况，所以这就相当于我们可以根据自己的需求合理地利用信号量的许可证来分配资源。

信号量还有几个注意点：

- 获取和释放的许可证数量尽量保持一致，否则比如每次都获取 2 个但只释放 1 个甚至不释放，那么信号量中的许可证就慢慢被消耗完了，最后导致里面没有许可证了，那其他的线程就再也没办法访问

了；

- 在初始化的时候可以设置公平性，如果设置为 true 则会让它更公平，但如果设置为 false 则会让总的吞吐量更高。
- 信号量是支持跨线程、跨线程池的，而且并不是哪个线程获得的许可证，就必须由这个线程去释放。事实上，对于获取和释放许可证的线程是没有要求的，比如线程 A 获取了然后由线程 B 释放，这完全是可以的，只要逻辑合理即可。

信号量能被 FixedThreadPool 替代吗?

答：不可以，因为FixedThreadPool 是固定访问线程个数，而信号量更灵活：

假如，在调用慢服务之前需要有个判断条件，比如只想在每天的零点附近去访问这个慢服务时受到最大线程数的限制（比如 3 个线程），而在除了每天零点附近的其他大部分时间，我们是希望让更多的线程去访问的。所以在这种情况下就应该把线程池的线程数量设置为 50，甚至更多，然后在执行之前加一个 if 判断，如果符合时间限制了（比如零点附近），再用信号量去额外限制，这样做是比较合理的。

再说一个例子，比如说在大型应用程序中会有不同类型的任务，它们也是通过不同的线程池来调用慢服务的。因为调用方不只是一处，可能是 Tomcat 服务器或者网关，我们就不应该限制，或者说也无法做到限制它们的线程池的大小。但可以做的是，在执行任务之前用信号量去限制一下同时访问的数量，因为我们的信号量具有跨线程、跨线程池的特性，所以即便这些请求来自于不同的线程池，我们也可以限制它们的访问。如果用 FixedThreadPool 去限制，那就做不到跨线程池限制了，这样的话会让功能大大削弱。

基于以上的理由，如果想要限制并发访问的线程数，用信号量是更合适的。

Condition、object.wait() 和 notify() 的关系?

先看以下代码：

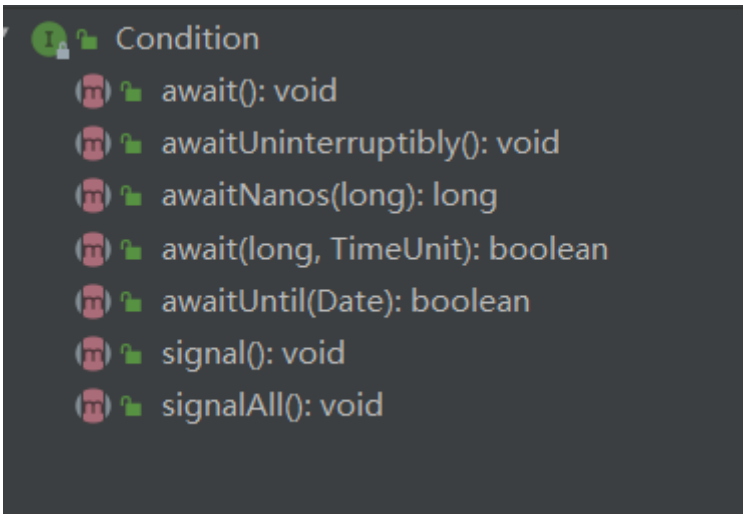
```
1  public class ConditionDemo {
2      private ReentrantLock lock = new ReentrantLock();
3      private Condition condition = lock.newCondition();
4
5      void method1() throws InterruptedException {
6          lock.lock();
7          try{
8              System.out.println(Thread.currentThread().getName()+":条件不满足，开始
   await");
9              condition.await();
10             System.out.println(Thread.currentThread().getName()+":条件满足了，开始
   执行后续的任务");
11         }finally {
12             lock.unlock();
13         }
14     }
15
16     void method2() throws InterruptedException {
17         lock.lock();
18         try{
19             System.out.println(Thread.currentThread().getName()+":需要5秒钟的准备时
   间");
20             Thread.sleep(5000);
```

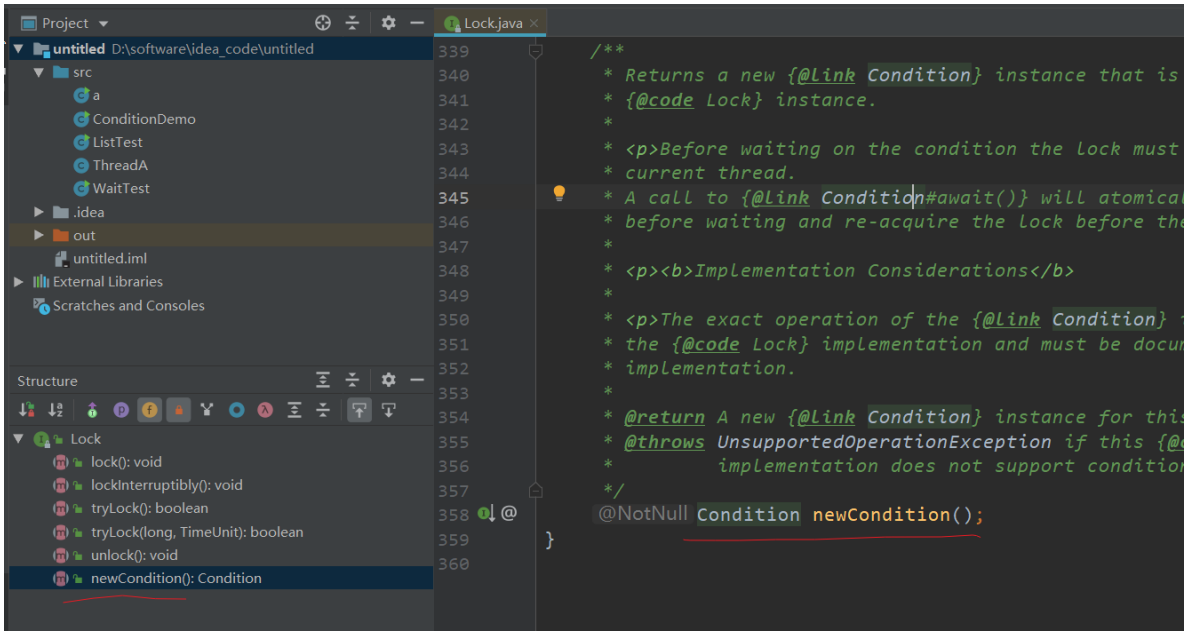
```
            System.out.println(Thread.currentThread().getName()+":准备工作完成，唤醒其他的线程");
            condition.signal();
        }finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ConditionDemo conditionDemo = new ConditionDemo();
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    conditionDemo.method2();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        conditionDemo.method1();
    }
}
```

运行结果：

```
main:条件不满足，开始 await
Thread-0:需要 5 秒钟的准备时间
Thread-0:准备工作完成，唤醒其他的线程
main:条件满足了，开始执行后续的任务
```

再看看Condition接口的方法：





有上面图和代码可知，创建Condition只有通过“lock.newCondition”创建，lock为Lock接口的实现类，可以是不同类型的锁，Condition接口方法只有signal和await方法，一个是唤醒，一个是等待。

使用Condition实现简易版的阻塞队列：

```
1  public class MyBlockingQueueForCondition {
2
3      private Queue queue;
4      private int max = 16;
5      private ReentrantLock lock = new ReentrantLock();
6      private Condition notEmpty = lock.newCondition();
7      private Condition notFull = lock.newCondition();
8
9      public MyBlockingQueueForCondition(int size) {
10         this.max = size;
11         queue = new LinkedList();
12     }
13
14     public void put(Object o) throws InterruptedException {
15         lock.lock();
16         try {
17             while (queue.size() == max) {
18                 notFull.await();
19             }
20             queue.add(o);
21             notEmpty.signalAll();
22         } finally {
23             lock.unlock();
24         }
25     }
26
27     public Object take() throws InterruptedException {
28         lock.lock();
29         try {
30             while (queue.size() == 0) {
31                 notEmpty.await();
32             }
33             Object item = queue.remove();
34             notFull.signalAll();
35             return item;
36         } finally {
37             lock.unlock();
```

```
        }
    }
}
```

使用wait/notify 实现简易版阻塞队列：

```
class MyBlockingQueueForWaitNotify {

    private int maxSize;
    private LinkedList<Object> storage;

    public MyBlockingQueueForWaitNotify (int size) {
        this.maxSize = size;
        storage = new LinkedList<>();
    }

    public synchronized void put() throws InterruptedException {
        while (storage.size() == maxSize) {
            this.wait();
        }
        storage.add(new Object());
        this.notifyAll();
    }

    public synchronized void take() throws InterruptedException {
        while (storage.size() == 0) {
            this.wait();
        }
        System.out.println(storage.remove());
        this.notifyAll();
    }
}
```

Condition 其实和Object 的 wait/notify/notifyAll用法和性质上几乎都一样。

Condition 把 Object 的 wait/notify/notifyAll 转化为了一种相应的对象，其实现的效果基本一样，但是把更复杂的用法，变成了更直观可控的对象方法，是一种升级。

await 方法会自动释放持有的 Lock 锁，和 Object 的 wait 一样，不需要自己手动释放锁。

另外，调用 await 的时候必须持有锁，否则会抛出异常，这一点和 Object 的 wait 一样。

CountDownLatch讲解：

CountDownLatch图解：

> CountDownLatch 的核心思想：其实就是等其他条件满足了才执行线程。

我们把激流勇进的例子用流程图的方式来表示：

可以看到，最开始 CountDownLatch 设置的初始值为 3，然后 T0 线程上来就调用 await 方法，它的作用是让这个线程开始等待，等待后面的 T1、T2、T3，它们每一次调用 countDown 方法，3 这个数值就会减 1，也就是从 3 减到 2，从 2 减到 1，从 1 减到 0，一旦减到 0 之后，这个 T0 就相当于达到了自己触发继续运行的条件，于是它就恢复运行了。

主要方法介绍：

（1）**构造函数：**public CountDownLatch(int count) { };

它的构造函数是传入一个参数，该参数 count 是需要倒数的数值。

（2）**await()：**调用 await() 方法的线程开始等待，直到倒数结束，也就是 count 值为 0 的时候才会继续执行。

（3）**await(long timeout, TimeUnit unit)：**await() 有一个重载的方法，里面会传入超时参数，这个方法的作　　用和 await() 类似，但是这里可以设置超时时间，如果超时就不再等待了。

（4）**countDown()：**把数值倒数 1，也就是将 count 值减 1，直到减为 0 时，之前等待的线程会被唤起。

CountDownLatch 的两个典型用法：

跑步比赛，等所有选手准备就绪，比赛才能开始：

```
1  public class RunDemo2 {
2
3      public static void main(String[] args) throws InterruptedException {
4          System.out.println("运动员有5秒的准备时间");
5          CountDownLatch countDownLatch = new CountDownLatch(1);
6          ExecutorService service = Executors.newFixedThreadPool(5);
7          for (int i = 0; i < 5; i++) {
8              final int no = i + 1;
9              Runnable runnable = new Runnable() {
10                 @Override
11                 public void run() {
12                     System.out.println(no + "号运动员准备完毕，等待裁判员的发令枪");
13                     try {
14                         countDownLatch.await();
15                         System.out.println(no + "号运动员开始跑步了");
16                     } catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                    }
                }
            };
            service.submit(runnable);
        }
        Thread.sleep(5000);
        System.out.println("5秒准备时间已过，发令枪响，比赛开始！");
        countDownLatch.countDown();
    }
}
```

跑步比赛，等所有选手跑完，比赛才算结束了：

```
public class RunDemo1 {

    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(5);
        ExecutorService service = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 5; i++) {
            final int no = i + 1;
            Runnable runnable = new Runnable() {

                @Override
                public void run() {
                    try {
                        Thread.sleep((long) (Math.random() * 10000));
                        System.out.println(no + "号运动员完成了比赛。");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        latch.countDown();
                    }
                }
            };
            service.submit(runnable);
        }
        System.out.println("等待5个运动员都跑完.....");
        latch.await();
        System.out.println("所有人都跑完了，比赛结束。");
    }
}
```

CountDownLatch 的注意点：

- 刚才讲了两种用法，其实这两种用法并不是孤立的，甚至可以把这两种用法结合起来，比如利用两个 CountDownLatch，第一个初始值为多个，第二个初始值为 1，这样就可以应对更复杂的业务场景了；
- CountDownLatch 是**不能够重用的**，比如已经完成了倒数，那可不可以在下一次继续去重新倒数呢？这是做不到的，如果你有这个需求的话，可以考虑使用 CyclicBarrier 或者创建一个新的 CountDownLatch 实例。

CyclicBarrier讲解：

CyclicBarrier其实也是和CountDownLatch 一样，等待满足条件后再执行线程。

CyclicBarrier 使用示例：

```java
public class CyclicBarrierDemo {

    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(3);
        for (int i = 0; i < 6; i++) {
            new Thread(new Task(i + 1, cyclicBarrier)).start();
        }
    }

    static class Task implements Runnable {

        private int id;
        private CyclicBarrier cyclicBarrier;

        public Task(int id, CyclicBarrier cyclicBarrier) {
            this.id = id;
            this.cyclicBarrier = cyclicBarrier;
        }

        @Override
        public void run() {
            System.out.println("同学" + id + "现在从大门出发，前往自行车驿站");
            try {
                Thread.sleep((long) (Math.random() * 10000));
                System.out.println("同学" + id + "到了自行车驿站，开始等待其他人到达");
                cyclicBarrier.await();
                System.out.println("同学" + id + "开始骑车");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```

CyclicBarrier 和 CountDownLatch 的异同：

- 相同点：

都能阻塞一个或一组线程，直到某个预设的条件达成发生，再统一出发。

- 不同点：
  - 作用对象不同：CyclicBarrier 要等固定数量的线程都到达了栅栏位置才能继续执行，而 CountDownLatch 只需等待数字倒数到 0，也就是说 **CountDownLatch 作用于事件，但 CyclicBarrier 作用于线程**；CountDownLatch 是在调用了 countDown 方法之后把数字倒数减 1，而 CyclicBarrier 是在某线程开始等待后把计数减 1。
  - 可重用性不同：CountDownLatch 在倒数到 0 并且触发门闩打开后，就不能再次使用了，除非新建一个新的实例；而 CyclicBarrier 可以重复使用，在刚才的代码中也可以看出，每 3 个同学到了之后都能出发，并不需要重新新建实例。CyclicBarrier 还可以随时调用 reset 方法进行重置，如果重置时有线程已经调用了 await 方法并开始等待，那么这些线程则会抛出 BrokenBarrierException 异常。
  - 执行动作不同：CyclicBarrier 有执行动作 barrierAction，而 CountDownLatch 没这个功能。

barrierAction介绍：

barrierAction是存在于CyclicBarrier的构造函数中

```
public CyclicBarrier(int parties, Runnable barrierAction)//parties代表需要几个线程到齐；barrierAction代表到齐后将会执行该Runnable对象
```

barrierAction其实就是满足条件后会运行的Runnable对象，至于内容可以机子定义：

```
CyclicBarrier cyclicBarrier = new CyclicBarrier(3, new Runnable() {
    @Override
    public void run() {
        System.out.println("凑齐3人了，出发！");
    }
});
```

Countlatchdown或CyclicBarrier 使用场景，举个使用例子？？？