**1.等待多线程完成的countDownLatch**

CountDownLatch允许一个或多个线程等待其他线程完成操作。

线程中，让一个线程等待最简单的做法是使用join方法，线程A中调用B.join方法，说明让线程A等待线程B完成之后再执行。

实现原理：不停检查线程是否存活，如果join线程存活则让当前线程永远等待。Wait(0)表示永远等待下去



直到join线程终止后，线程的this.notifyAll方法被调用

CountDownLatch接受一个int类型的参数作为计数器，如果你想等待n个点完成，这里就传入n。调用countDownLatch的countDown方法时，n就会减1，如果计数器大于0，await方法等待，如果计数器等于零，await方法不等待。countDown方法可以用在任何地方，可以是n个线程，也可以1个线程里n个步骤。

|  |
| --- |
| ExecutorService service = Executors.*newCachedThreadPool*();  **final** CountDownLatch cdOrder = **new** CountDownLatch(1);  **final** CountDownLatch cdAnswer = **new** CountDownLatch(3);  **for**(**int** i = 0; i < 3; i++){  Runnable runnable = **new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **try** {  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "正在准备接受命令");  cdOrder.await();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已接收命令");  Thread.*sleep*((**long**)Math.*random*()\*1000);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "回应命令处理结果");  cdAnswer.countDown();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  };  service.execute(runnable);  }  **try**{  Thread.*sleep*((**long**)Math.*random*()\*1000);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "即将发布命令");  cdOrder.countDown();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已发送命令，正在等待结果");  cdAnswer.await();  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "已收到所有响应结果");  }**catch**(Exception e){  e.printStackTrace();  }  service.shutdown(); |

**2.同步屏障CyclicBarrier**

可循环使用的屏障，让一组线程到达一个屏障（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。

假设**CyclicBarrier=3**，如果阻塞没有到达三个，那么await方法就会一直等待，如果到达三个，那么就不会阻塞。

应用场景：

接受指令，有三个线程，需要到达指定地点集合，第一个线程先到，调用await方法，第二个线程也是一样，等到第三个线程到了，最后一个线程到达屏障，则await开始执行。

**CyclicBarrier和CountDownLatch的区别**

CountDownLatch的计数器只能用一次，而CyclicBarrier计数器可以使用reset重置，其实不用手动重置，await到达最后一个的时候，就会自动将计数器置为初始化个数，CyclicBarrier能处理更为复杂的业务，

CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得Cyclic-Barrier阻塞的线程数量。isBroken()方法用来了解阻塞的线程是否被中断

**3.控制并发线程数的semaphore**

控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源

\* semaphore实现的功能就类似厕所有5个坑，假如有十个人要上厕所

\* ，那么同事能有多少个人去上厕所呢，同时只有5个人能够占用，当5个人

\* 中的任何一个人让开后，其中在等待的另外5个人中又有一个人可以占用了

\*

\* 另外等待的5个人中可以随机获得优先机会，可以是按照先来后到的顺序获得机会

\* ，这取决于构造semaphore对象时传入的参数选项

\* final Semaphore sp = new Semaphore(3, true);true表示按照先来后来顺序

\*

\*单个信号量的semaphore对象可以实现互斥锁的功能，并且可以由一个线程

\*获得锁，另一个线程释放锁，应用于死锁回复的场合

|  |
| --- |
| **public** **class** SemaphoreTest {  **public** **static** **void** main(String[] args) {  ExecutorService service = Executors.*newCachedThreadPool*();  **final** Semaphore sp = **new** Semaphore(3);  **for**(**int** i = 0; i < 10; i++){  Runnable runnable = **new** Runnable(){  **public** **void** run(){  **try**{  sp.acquire();  }**catch**(InterruptedException e1){  e1.printStackTrace();  }  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "进入，当前已有" + (3 - sp.availablePermits()) + "并发");  **try**{  Thread.*sleep*((**long**)(Math.*random*() \* 1000));  }**catch**(InterruptedException e){  e.printStackTrace();  }  System.***out***.println("线程" + Thread.*currentThread*().getName() + "即将离开");  sp.release();  //下面代码有时候执行不准确，因为其没有和上面代码合成原子步骤  System.***out***.println("线程" + Thread.*currentThread*().getName() + "已离开，当前已有"  + (3-sp.availablePermits()) + "并发");  }  };  service.execute(runnable);  }  }  } |

首先线程使用Semaphore的acquire()方法获取一个许可证，如果semaphore小于0，就阻塞使用完之后调用release()方法归还许可证。

**4.线程间交换数据的Exchanger**

它提供一个**同步点**，在这个同步点，两个线程**可以交换彼此的数据**。这两个线程通过**exchange**方法交换数据，如果第一个线程先执行**exchange()**方法，它会**一直等待**第二个线程也**执行exchange方法**，当两个线程都**到达同步点**时，这两个线程就可以**交换数据**，将本线程生产出来的数据传递给对方

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ExecutorService service = Executors.*newCachedThreadPool*();  **final** Exchanger exchanger = **new** Exchanger();  service.execute(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **try**{  String data1 = "zxx";  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "正在把数据" + data1 + "换出去");  Thread.*sleep*((**long**) (Math.*random*()\*1000));  String data2 = (String) exchanger.exchange(data1);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "换回的数据为" + data2);  }**catch**(Exception e){  e.printStackTrace();  }  }  });    service.execute(**new** Runnable(){  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **try**{  String data1 = "lhm";  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "正在把数据" + data1 + "换出去");  Thread.*sleep*((**long**) (Math.*random*()\*1000));  String data2 = (String) exchanger.exchange(data1);  System.***out***.println("线程" + Thread.*currentThread*().getName() +  "换回的数据为" + data2);  }**catch**(Exception e){  e.printStackTrace();  }  }  });    service.shutdown();    } |