CountDownLatch CyclicBarrier 解析

CountDownLatch 是一个同步工具，它可以让一个或多个线程等待一系列线程执行完成。它通过一个给定的数值 count 进行初始化，直到调用 countDown 方法使得 count 变为 0 。此时，所有等待的线程会马上被释放去执行之后的操作。

适用场景：

1、开关控制场景，此时通过把 CountDownLatch 的 count 用 1去初始化。

2、需要等待 N 个线程执行完后才能继续执行的场景。此时可利用 N 去初始化 CountDownLatch 。

3、分布计算，可以把一个问题分解为 N 部分，每个部分用一个线程去执行。然后用 N 去初始化 CountDownLatch 。等所有线程执行完后再去调用协作线程进行相关操作。

特性：

count 的值为一次性值，不能够重置。如果需要多次重复使用，可以使用 CyclicBarrier。

例子：

```
class Driver{
    void main() throws IntegerruptedException e{
      CountDownLatch startSignal = new CountDownLatch(1);
      CountDownLatch doneSignal = new CountDownLatch(N);
      for(int i = 0;i<N;++i)    // create and start threads
         new Thread(new Worker(startSignal,doneSignal)).start();
      doSomethingElse();    // don't let run yet
      startSignal.countDown();    // let all threads proceed
      doSomethingElse();
      doneSignal.await();    //wait for all to finish
    }
}
class Worker implements Runnable{
   private final CountDownLatch startSignal;
   private final CountDownLatch doneSignal;
   Worker(CountDownLatch startSignal,CountDownLatch doneSignal){
      this.startSignal = startSignal;
      this.doneSignal = doneSignal;
   }
   public void run(){
      try{
         startSignal.await();
         doWork();
         doneSignal.countDown();
      }catch(InterruptedException e){
            e.printStackTrace();
      }
   }
   void doWork(){...}
}
class Driver2{
   void main() throws IntegerruptedException {
      CountDownLatch doneSignal = new CountDownLatch(N);
      ExecutorService e = ...
      for(int i = 0 ; i < N ; ++i) //create and start threads
         e.execute(new WorkerRunnable(doneSignal,i));
      doneSignal.await();    // wait for all to finish
   }
}
class WorkerRunnable implements Runnable{
   private final CountDownLatch doneSignal;
   private final int i;
   WorkerRunnable(CountDownLatch doneSignal,int i){
      this.doneSignal = doneSignal;
      this.i = i;
   }
   public void run(){
      try{
         doWork(i);
         doneSignal.countDown();
      }catch(IntegerruptedException e){
         e.printStackTrace(0;)
      }
```

```
    }
    void doWork(){...}
}
```

CyclicBarrier 是一个同步工具，它可以让一系列线程相互等待直到某一个屏障点，然后再开始执行。

适用场景：

1、线程必须相互等待才能执行的场景。

2、需要多线同步一系列线程，要求可复用的场景。

特性：

1、count 的值可复用，每次屏障被打破后，count 会被重置，达到复用的目的。可用于复杂的业务场景

2、提供一个可选的 Runnable 参数，当所有等待的线程达到 count（屏障值）时就会执行，即在最后一个等待线程到达之后，所有等待线程得到释放之前执行。 这样一个 barrier action （屏幕响应）可以用来更新所有等待线程的共享状态值（共享信息）。

例子：

```
class Solver{
    final int N;
    final float[][] data;
    final CyclicBarrier barrier;
    class Worker implements Runnable{
        int myRow;
        Worker(int row){
            myRow = row;
        }
        public void run(){
            while(!done()){
                processRow(myRow);
                try{
                    barrier.await();
                }catch(InterruptedException e){
                    return ;
                }catch(BrokenBarrierException ex){
                    return ;
                }
            }
        }
    }
    public Solver(float[][] matrix){
        data = matrix;
        N = matrix.length;
        barrier = new CyclicBarrier(N,new Runnable(){
            public void run(){
                mergeRows(....);
            }
        });
        for(int i = 0;i < N;++i){
            new Thread(new Worker(i)).start;
        }
        waitUntilDone();
    }
}
```

CyclicBarrier 使用一种 全做或全不做的中断模型来应对同步线程失败的情况：如果一个线程由于 中断、失败、超时而过早的离开了 barrier 那么其他所有在 barrier 上等待的线程也会通过 BrokenBarrierException 或者 InterruptedException（如果它们也全部同时被打断）的方式非正常的离开。

例子：

```
public class HelloCyclicBarrierDemo {
    private static CyclicBarrier barrier = new CyclicBarrier(4, new Runnable() {
        @Override
        public void run() {
            System.out.println("barrier end...");
            System.out.println("end time:"+System.currentTimeMillis());
            System.out.println(barrier.getNumberWaiting());
        }
    });
    public static void main(String[] args) {
            System.out.println("start time:"+System.currentTimeMillis());
            System.out.println(barrier.getNumberWaiting());
            for (int i = 0; i < 3; i++) {
                Thread thread = new Thread(new Worker(barrier),"thread"+i);
                thread.start();
            }
```

```
        Thread thread4 = new Thread(new Worker2(barrier),"thread4");
        thread4.start();
        thread4.interrupt();
    }
    static  class Worker implements Runnable{
        private final CyclicBarrier barrier;
        public Worker(CyclicBarrier barrier){
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                System.out.println("work start....");
                Thread.sleep(4000);    // simulate doing work.
                System.out.println("work end....");
                System.out.println(Thread.currentThread().getName()+" barrier size:"+barrier.getNumberWaiting());
            }catch (InterruptedException  e) {
                System.out.println(Thread.currentThread().getName()+" is Interrupted.");
                System.out.println(Thread.currentThread().getName()+" is Interrupted:"+Thread.currentThread().isInterrupted());
                Thread.currentThread().interrupt();//reset interrupt state because of await will use it.
                System.out.println(Thread.currentThread().getName()+" is Interrupted:"+Thread.currentThread().isInterrupted());
            }
            try {
                barrier.await();
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName()+" is Interrupted...");
            } catch (BrokenBarrierException e) {
                System.out.println(Thread.currentThread().getName()+" is Broken.");
            }
            System.out.println(Thread.currentThread().getName()+" continue...");
        }
    }
    static  class Worker2 implements Runnable{
        private final CyclicBarrier barrier;
        public Worker2(CyclicBarrier barrier){
            this.barrier = barrier;
        }
        @Override
        public void run() {
            try {
                for(int i = 0 ; i<1000000;++i){
                    //an empty loop to simulate doing something.
                }
                barrier.await();
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName()+" is Interrupted...");
                return;// return after interrupted.
            } catch (BrokenBarrierException e) {
                System.out.println(Thread.currentThread().getName()+" is Broken.");
            }
        }

    }
}
```

PS: JAVA中的中断机制是一种协作机制，只有被中断的线程才能对中断进行相应的操作和处理。所有如果一个例子中有多个地方涉及中断，那么当其中一个捕获了中断后，要记得重设中断状态，如上例中所示，如果不重设会导致 barrier 无法感知中断从而所有 等待的线程无法通过 BrokenBarrier 释放。