JUC 并发工具类 CyclicBarrier

1. 简介
2. 实现分析 jdk1.8

   2.1 await 2.2 await 2.3 dowait 2.4 Generation 2.5 breakBarrier 2.6 nextGeneration 2.7 reset 2.8 getNumberWaiting 2.9 isBroken
3. 应用场景
4. 应用示例

1. 简介

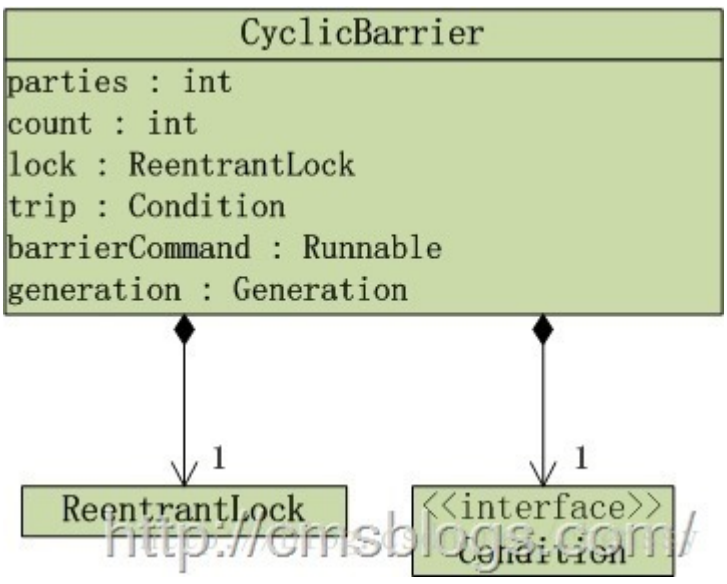
CyclicBarrier ，一个同步辅助类，在 AP I中是这么介绍的：

它允许一组线程互相等待，直到到达某个**公共屏障点** (Common Barrier Point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 Barrier 在释放等待线程后可以重用，所以称它为**循环(Cyclic) 的 屏障(Barrier)** 。

通俗点讲就是：让一组线程到达一个屏障时被阻塞，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干活。

2. 实现分析

`java.util.concurrent.CyclicBarrier` 的结构如下：



通过上图，我们可以看到 CyclicBarrier 的内部是使用重入锁 ReentrantLock 和 Condition 。

它有两个构造函数：

- `CyclicBarrier(int parties)`：创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。
- `CyclicBarrier(int parties, Runnable barrierAction)`：创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。
- 代码如下：

```
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}

public CyclicBarrier(int parties) {
    this(parties, null);
}
```

- `parties` 变量，表示拦截线程的**总**数量。
- `count` 变量，表示拦截线程的**剩余需要**数量。
- `barrierAction` 变量，为 CyclicBarrier 接收的 Runnable 命令，用于在线程到达屏障时，优先执行 `barrierAction` ，用于处理更加复杂的业务场景。
- `generation` 变量，表示 CyclicBarrier 的更新换代。

2.1 await

> 或者说，每个线程调用 `#await()` 方法，告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。当所有线程都到达了屏障，结束阻塞，所有线程可继续执行后续逻辑。

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);//不超时等待
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

- 内部调用 `#dowait(boolean timed, long nanos)` 方法，执行阻塞等待(`timed=true`)。详细解析，见 「2.3 dowait」 。
- 理论来说，不会出现 TimeoutException 异常，所以在发生时，直接抛出 Error 错误。
- > the arrival index of the current thread, where index {@code getParties() - 1} indicates the first to arrive and zero indicates the last to arrive

2.2 await

`#await(long timeout, TimeUnit unit)` 方法，在 `#await()` 的基础上，增加了等待超时的特性。代码如下：

```
public int await(long timeout, TimeUnit unit)
    throws InterruptedException,
           BrokenBarrierException,
           TimeoutException {
    return dowait(true, unit.toNanos(timeout));
}
```

- 内部调用 `#dowait(boolean timed, long nanos)` 方法，执行阻塞等待(`timed=true`)。详细解析，见「2.3 dowait」。

2.3 dowait

`#dowait(boolean timed, long nanos)` 方法，代码如下：

```
private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
        TimeoutException {
    //获取锁
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        //分代
        final Generation g = generation;

        //当前generation“已损坏”，抛出BrokenBarrierException异常
        //抛出该异常一般都是某个线程在等待某个处于“断开”状态的CyclicBarrie
        if (g.broken)
            //当某个线程试图等待处于断开状态的 barrier 时，或者 barrier 进入断开状态而线程处于等待状态
时，抛出该异常
            throw new BrokenBarrierException();

        //如果线程中断，终止CyclicBarrier
        if (Thread.interrupted()) {
            breakBarrier();
            throw new InterruptedException();
        }

        //进来一个线程 count - 1
        int index = --count;
        //count == 0 表示所有线程均已到位，触发Runnable任务
        if (index == 0) {  // tripped
            boolean ranAction = false;
            try {
                final Runnable command = barrierCommand;
                //触发任务
                if (command != null)
                    command.run();
                ranAction = true;
                //唤醒所有等待线程，并更新generation
                nextGeneration();
                return 0;
            } finally {
```

```
                if (!ranAction) // 未执行，说明 barrierCommand 执行报错，或者线程打断等等情况。
                    breakBarrier();
            }
        }

        for (;;) {
            try {
                //如果不是超时等待，则调用Condition.await()方法等待
                if (!timed)
                    trip.await();
                else if (nanos > 0L)
                    //超时等待，调用Condition.awaitNanos()方法等待
                    nanos = trip.awaitNanos(nanos);
            } catch (InterruptedException ie) {
                if (g == generation && ! g.broken) {
                    breakBarrier();
                    throw ie;
                } else {
                    // We're about to finish waiting even if we had not
                    // been interrupted, so this interrupt is deemed to
                    // "belong" to subsequent execution.
                    Thread.currentThread().interrupt();
                }
            }

            if (g.broken)
                throw new BrokenBarrierException();

            //generation已经更新，返回index
            if (g != generation)
                return index;

            //“超时等待”，并且时间已到,终止CyclicBarrier，并抛出异常
            if (timed && nanos <= 0L) {
                breakBarrier();
                throw new TimeoutException();
            }
        }
    } finally {
        //释放锁
        lock.unlock();
    }
}
```

如果该线程不是到达的最后一个线程，则他会一直处于等待状态，除非发生以下情况：

1. 最后一个线程到达，即 `index == 0` 。
2. 超出了指定时间（超时等待）。
3. 其他的某个线程中断当前线程。
4. 其他的某个线程中断另一个等待的线程。
5. 其他的某个线程在等待 barrier 超时。

6. 其他的某个线程在此 barrier 调用 `#reset()` 方法。`#reset()` 方法，用于将屏障重置为初始状态。

在 `#dowait(boolean timed, long nanos)` 方法的源代码中，我们总是可以看到抛出 **BrokenBarrierException** 异常，那么什么时候抛出异常呢？例如：

- 如果一个线程处于等待状态时，如果其他线程调用 `#reset()` 方法。详细解析，见 「2.7 reset」 。
- 调用的 barrier 原本就是被损坏的，则抛出 BrokenBarrierException 异常。
- 任何线程在等待时被中断了，则其他所有线程都将抛出 BrokenBarrierException 异常，并将 barrier 置于**损坏**状态。详细解析，见 「2.6 breakBarrier」 。

2.4 Generation

Generation 是 CyclicBarrier 内部静态类，描述了 CyclicBarrier 的更新换代。**在CyclicBarrier中，同一批线程属于同一代。**当有 `parties` 个线程全部到达 barrier 时，`generation` 就会被更新换代。其中 `broken` 属性，标识该当前 CyclicBarrier 是否已经处于中断状态。代码如下：

```
private static class Generation {
    boolean broken = false;
}
```

默认 barrier 是没有损坏的。

2.5 breakBarrier

当 barrier 损坏了，或者有一个线程中断了，则通过 `#breakBarrier()` 方法，来终止所有的线程。代码如下：

```
private void breakBarrier() {
    generation.broken = true;
    count = parties;
    trip.signalAll();
}
```

- 在 `breakBarrier()` 方法中，中除了将 `broken` 设置为 true ，还会调用 `#signalAll()` 方法，将在 CyclicBarrier 处于**等待状态的线程**全部唤醒。

2.6 nextGeneration

当所有线程都已经到达 barrier 处（`index == 0`），则会通过 `nextGeneration()` 方法，进行更新换代操作。在这个步骤中，做了三件事：

1. 唤醒所有线程。
2. 重置 `count` 。
3. 重置 `generation` 。

代码如下：

```
private void nextGeneration() {
    trip.signalAll();
    count = parties;
    generation = new Generation();
}
```

2.7 reset

`#reset()` 方法，重置 barrier 到初始化状态。代码如下：

```
public void reset() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        breakBarrier();   // break the current generation
        nextGeneration(); // start a new generation
    } finally {
        lock.unlock();
    }
}
```

- 通过组合 `#breakBarrier()` 和 `#nextGeneration()` 方法来实现。

2.8 getNumberWaiting

`#getNumberWaiting()` 方法，获得等待的线程数。代码如下：

```
public int getNumberWaiting() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return parties - count;
    } finally {
        lock.unlock();
    }
}
```

2.9 isBroken

`#isBroken()` 方法，判断 CyclicBarrier 是否处于中断。代码如下：

```
public boolean isBroken() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return generation.broken;
    } finally {
        lock.unlock();
    }
}
```

3. 应用场景

CyclicBarrier 适用于多线程结果合并的操作，用于多线程计算数据，最后合并计算结果的应用场景。比如，我们需要统计多个 Excel 中的数据，然后等到一个总结果。我们可以通过多线程处理每一个 Excel ，执行完成后得到相应的结果，最后通过 `barrierAction` 来计算这些线程的计算结果，得到所有Excel的总和。

4. 应用示例

比如我们开会只有等所有的人到齐了才会开会，如下：

```
public class CyclicBarrierTest {

    private static CyclicBarrier cyclicBarrier;

    static class CyclicBarrierThread extends Thread{
        public void run() {
            System.out.println(Thread.currentThread().getName() + "到了");
            //等待
            try {
                cyclicBarrier.await();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args){
        cyclicBarrier = new CyclicBarrier(5, new Runnable() {
            @Override
            public void run() {
                System.out.println("人到齐了，开会吧....");
            }
        });

        for(int i = 0 ; i < 5 ; i++){
            new CyclicBarrierThread().start();
```

```
        }
    }

}
```