257 CountDownLatch vs Phaser

Posted: 2018-04-02 | Category: Concurrency | Java Version: 7+ | Dr. Heinz M. Kabutz

Abstract: Java 7 gave us a brilliant new class called Phaser, which we can use to coordinate actions between threads. It replaces both CountDownLatch and CyclicBarrier, which are easier to understand, but harder to use.

Welcome to the 257th edition of **The Java(tm) Specialists' Newsletter**. Two newsletters in three days? I woke up this morning with a dream that someone had sent me an email, complaining that my newsletters had become like JAX magazine articles. A weird dream indeed, since I do write for their Java Magazine and like it a lot. Perhaps it was my subconscious reminding me that no one had commented on the brilliance of my previous newsletter, but then, I did receive hundreds of OutOfOfficeException emails :-)

javaspecialists.teachable.com: Please visit our new self-study course catalog to see how you can upskill your Java knowledge.

CountDownLatch vs CyclicBarrier vs Phaser

The only talk that I have heard someone speak about Phaser was my own. I have taught about it for many years in my Extreme Java - Concurrency Performance Course, and have yet to find a student that tells me: "Oh yes, that's an awesome class, we use it all the time!" They have usually heard of CountDownLatch and perhaps CyclicBarrier, but Phaser remains elusive.

How can this be, when Phaser has been around since Java 7 and makes synchronizing between threads so much easier than other similar constructs?

I like to say that CountDownLatch is easy to understand, but hard to use. Phaser, on the other hand, is hard to understand, but easy to use. Last week, I taught my concurrency course to a group of smart programmers in Athens. One of the many excellent questions was: "How can we coordinate a bunch of tasks that all take slightly different times?" My initial response was to use `CompletionStage`, but then the more we looked at the problem we were trying to solve, the better Phaser seemed to fit. In class, I first coded Phaser. Someone then asked whether the same would be possible with CountDownLatch. So we coded that too. In this newsletter I will do it the other way round. We will start with the CountDownLatch and then refactor it to use Phaser instead.

We will execute 5 batches of tasks. Each batch will have 3 tasks taking between 500 milliseconds and 3 seconds to complete. Tasks within a batch should all start at the same time. To make the code easier to read, we define a common superclass `LockStepExample`:

```java
import java.util.concurrent.*;

public abstract class LockStepExample {
  protected final static int TASKS_PER_BATCH = 3;
  protected final static int BATCHES = 5;

  protected final void doTask(int batch) {
    System.out.printf("Task start %d%n", batch);
    int ms = ThreadLocalRandom.current().nextInt(500, 3000);
    try {
      Thread.sleep(ms);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    System.out.printf("Task in batch %d took %dms%n", batch, ms);
  }
}
```

Next we extends that with our `LockStepCountDownLatch`. Since the `CountDownLatch` cannot be reset, we need to make a new latch for each batch of tasks. The latch also has rather old-fashioned interrupt handling. There is no way to silently save the interrupt until we are done, as we can do with `Semaphore.acquireUninterruptibly()` or `Lock.lock()`. The code in `task()` is thus rather involved. We first call `latch.countDown()` to signal that our thread as arrived at the starting gate. We then call `latch.await()`, but need to manage the `InterruptedException` ourselves. We do this by clearing the interrupted status with `Thread.interrupted()`. We then keep on calling `latch.await()` until we are able to exit "normally". If during the `latch.await()` we get interrupted again, we remember the state, but keep on waiting. Finally, once we exit the `while(true)` with the `break`, we self-interrupt if we were interrupted at some point in our waiting code. `interrupt()` does not stop the thread, it merely changes the state to be interrupted. We then call the `doTask()` method, passing in the batch number.

```java
import java.util.concurrent.*;

import static java.util.concurrent.Executors.newFixedThreadPool;

public class LockStepCountDownLatch extends LockStepExample {
  public static void main(String... args) {
    LockStepCountDownLatch lse = new LockStepCountDownLatch();
    ExecutorService pool = newFixedThreadPool(TASKS_PER_BATCH);
    for (int batch = 0; batch < BATCHES; batch++) {
      // We need a new CountDownLatch per batch, since they
      // cannot be reset to their initial value
      CountDownLatch latch = new CountDownLatch(TASKS_PER_BATCH);
      for (int i = 0; i < TASKS_PER_BATCH; i++) {
        int batchNumber = batch + 1;
        pool.submit(() -> lse.task(latch, batchNumber));
      }
    }
    pool.shutdown();
  }

  public void task(CountDownLatch latch, int batch) {
    latch.countDown();
    boolean interrupted = Thread.interrupted();
    while (true) {
      try {
        latch.await();
        break;
      } catch (InterruptedException e) {
        interrupted = true;
      }
    }
    if (interrupted) Thread.currentThread().interrupt();
    doTask(batch);
  }
}
```

Output would look like this:

```
Task start 1
Task start 1
Task start 1
Task in batch 1 took 747ms
Task in batch 1 took 1087ms
Task in batch 1 took 2780ms
Task start 2
Task start 2
Task start 2
Task in batch 2 took 584ms
Task in batch 2 took 634ms
Task in batch 2 took 2194ms
Task start 3
Task start 3
Task start 3
Task in batch 3 took 603ms
Task in batch 3 took 1868ms
Task in batch 3 took 2874ms
Task start 4
Task start 4
Task start 4
Task in batch 4 took 1035ms
Task in batch 4 took 1724ms
Task in batch 4 took 2527ms
Task start 5
Task start 5
Task start 5
Task in batch 5 took 1579ms
Task in batch 5 took 1602ms
Task in batch 5 took 2752ms
```

Our first challenge was having to create a new `CountDownLatch` for every batch. We could avoid this by using a `CyclicBarrier`. This allows us to reuse the barrier, but the interrupt handling is still from last millenium:

```java
import java.util.concurrent.*;

import static java.util.concurrent.Executors.*;

public class LockStepCyclicBarrier extends LockStepExample {
  public static void main(String... args) {
    LockStepCyclicBarrier lse = new LockStepCyclicBarrier();
    ExecutorService pool = newFixedThreadPool(TASKS_PER_BATCH);
    CyclicBarrier barrier = new CyclicBarrier(TASKS_PER_BATCH);
    for (int batch = 0; batch < BATCHES; batch++) {
      for (int i = 0; i < TASKS_PER_BATCH; i++) {
        int batchNumber = batch + 1;
        pool.submit(() -> lse.task(barrier, batchNumber));
      }
    }
    pool.shutdown();
  }

  public void task(CyclicBarrier barrier, int batch) {
    boolean interrupted = Thread.interrupted();
    while (true) {
      try {
        barrier.await();
        break;
      } catch (InterruptedException e) {
        interrupted = true;
      } catch (BrokenBarrierException e) {
        throw new AssertionError(e);
      }
    }
    if (interrupted) Thread.currentThread().interrupt();
    doTask(batch);
  }
}
```

Lastly, we show the `LockStepPhaser`. We can reuse the phaser for the batches, like with the `CyclicBarrier`. The Phaser also knows which *phase* it is in, thus we do not need to pass along the batch number. And the `task()` method? All the complicated interrupt handling code gets reduced to a one-liner `phaser.arriveAndAwaitAdvance()`. Simply brilliant!

```
import java.util.concurrent.*;

import static java.util.concurrent.Executors.newFixedThreadPool;

public class LockStepPhaser extends LockStepExample {
  public static void main(String... args) {
    LockStepPhaser lse = new LockStepPhaser();
    ExecutorService pool = newFixedThreadPool(TASKS_PER_BATCH);
    Phaser phaser = new Phaser(TASKS_PER_BATCH);
    for (int batch = 0; batch < BATCHES; batch++) {
      for (int i = 0; i < TASKS_PER_BATCH; i++) {
        pool.submit(() -> lse.task(phaser));
      }
    }
    pool.shutdown();
  }

  public void task(Phaser phaser) {
    phaser.arriveAndAwaitAdvance();
    doTask(phaser.getPhase());
  }
}
```

Some more reasons why Phaser is the preferred solution over CountDownLatch and CyclicBarrier: It is implemented with a ManagedBlocker. This means that if our Phaser blocks a thread in the common fork-join pool, another will be created to keep the parallelism at the desired level. Also, we can set up Phaser in a tree to reduce contention. This is a bit complicated, I admit. But it can be done. We cannot do this with the other synchronizers like CountDownLatch and CyclicBarrier.

Kind regards from Crete

Heinz

Comments

We are always happy to receive comments from our readers. Feel free to send me a comment via email or discuss the newsletter in our JavaSpecialists Slack Channel (Get an invite here)

Related Articles

149) The Law of the Overstocked Haberdashery 2007-08-20

Learn how to write correct concurrent code by understanding the Secrets of Concurrency. This is the third part of a series of laws that help explain how we should be writing concurrent code in Java. In this section, we look at why we should avoid creating unnecessary threads, even if they are n... Full Article

192) Implicit Escape of "this" 2011-05-31