Java Concurrency Tutorial – CountDownLatch

javacodegeeks.com Byron Kiourtzoglou September 19th, 2011 view original

Some concurrency utilities in Java naturally get more attention than others just because they serve general purpose problems instead of more specific ones. Most of us encounter things like executor services and concurrent collections fairly often.
Other utilities are less common, so sometimes they may escape us, but it's good to keep them in mind. **CountDownLatch** is one of those tools.

CountDownLatch – a more general wait/notify mechanism

Java developers of all sorts should be familiar with the wait/notify approach to blocking until a condition is reached. Here is a little sample of how it works:

```
public void testWaitNotify() throws Exception {
   final Object mutex = new Object();
   Thread t = new Thread() {
      public void run() {
      // we must acquire the lock before waiting to be
notified
      synchronized(mutex) {
         System.out.println("Going to wait " +
                              "(lock held by " +
Thread.currentThread().getName() + ")");
            try {
               mutex.wait(); // this will release the lock
to be notified (optional timeout can be supplied)
            } catch (InterruptedException e) {
               e.printStackTrace();
            }

            System.out.println("Done waiting " +
                               "(lock held by " +
Thread.currentThread().getName() + ")");
         }
      }
   };

   t.start(); // start her up and let her wait()

   // not normally how we do things, but good enough for
demonstration purposes
   Thread.sleep(1000);

   // we acquire the lock released by wait(), and notify()
   synchronized (mutex) {
      System.out.println("Going to notify " +
                         "(lock held by " +
Thread.currentThread().getName() + ")");
      mutex.notify();
      System.out.println("Done notify " +
                         "(lock held by " +
Thread.currentThread().getName() + ")");
   }

}
```

Output

```
Going to wait (lock held by Thread-0)
Going to notify (lock held by main)
```

```
Done notify (lock held by main)
Done waiting (lock held by Thread-0)
```

A **CountDownLatch** can actually be used similar to a wait/notify with only one notify – that is, as long as you don't want **wait()** to stall if **notify()** is called before you have acquired the lock and invoked **wait()**. It is actually more forgiving because of this, and in some cases, it's just what you want. Here's a sample:

```
public void testWaitNotify() throws Exception {
   final CountDownLatch latch = new CountDownLatch(1); //
just one time
   Thread t = new Thread() {
      public void run() {
         // no lock to acquire!
         System.out.println("Going to count down...");
         latch.countDown();
      }
   };

   t.start(); // start her up and let her wait()
   System.out.println("Going to await...");
   latch.await();
   System.out.println("Done waiting!");
}
```

As you can see, it is simpler than wait/notify, and requires less code. It also allows us to invoke the condition that ultimately releases the block before we call **wait()**. This can mean safer code.
A Real World Example

So we know we can use it as a simpler wait/notify mechanism, but you probably saw the constructor argument that we used above. In the constructor, you specify the number of times the latch needs to be counted down before unlocking. What possible uses are there for this? Well, it can be used to make a process wait until a certain number of actions have been taken.

For example, if you have an asynchronous process that you can hook into via listeners or something similar, you can create unit tests that verify a certain number of invocations were made. This lets us long only as long as we need to in the normal case (or some limit before we bail and assume failure).

I recently ran into a case where I had to verify that JMS messages were being pulled off the queue and handled correctly. This was naturally asynchronous and outside of my control, and mocks weren't an option since it was a fully assembled application with Spring context, etc. To test this, I made minor changes to the consuming services to allow listeners to be invoked when the messages were processed. I was then able to temporarily add a listener, which used a **CountDownLatch** to keep my tests as close to synchronous as possible.

Here's an example that shows the concept:

```
public void testSomeProcessing() throws Exception {
   // should be called twice
   final CountDownLatch testLatch = new CountDownLatch(2);
   ExecutorService executor =
Executors.newFixedThreadPool(1);
   AsyncProcessor processor = new AsyncProcessor(new
Observer() {
      // this observer would be the analogue for a
listener in your async process
      public void update(Observable o, Object arg) {
         System.out.println("Counting down...");
         testLatch.countDown();
      }
   });

   //submit two tasks to be process
   // (in my real world example, these were JMS messages)
   executor.submit(processor);
   executor.submit(processor);

   System.out.println("Submitted tasks. Time to wait...");
```

```
      long time = System.currentTimeMillis();
      testLatch.await(5000, TimeUnit.MILLISECONDS); // bail
   after a reasonable time
      long totalTime = System.currentTimeMillis() - time;

      System.out.println("I awaited for " + totalTime +
                          "ms. Did latch count down? " +
   (testLatch.getCount() == 0));

      executor.shutdown();
   }

   // just a process that takes a random amount of time
   // (up to 2 seconds) and calls its listener
   public class AsyncProcessor implements Callable<Object> {
         private Observer listener;
         private AsyncProcessor(Observer listener) {
         this.listener = listener;
      }

      public Object call() throws Exception {
         // some processing here which can take all kinds of
   time...
         int sleepTime = new Random().nextInt(2000);
         System.out.println("Sleeping for " + sleepTime +
   "ms");
         Thread.sleep(sleepTime);
         listener.update(null, null); // not standard usage,
   but good for a demo
         return null;
      }
   }
```

Output

```
    Submitted tasks. Time to wait...
    Sleeping for 739ms
    Counting down...
    Sleeping for 1742ms
    Counting down...
    I awaited for 2481ms. Did latch count down? true
```

Conclusion
That’s about it for the **CountDownLatch**. It is not a complicated subject and it has limited usages, but it’s good to see examples and know it’s their in your tool chest when you hit a problem like I did. In the future, I’ll definitely keep it in mind for a simpler wait/notify, if nothing else. If you have questions or comments about this post or others in the series, just leave a message.

Reference: Java Concurrency Part 6 – CountDownLatch from our JCG partners at the Carfey Software blog.