Synchronizers – Latches & Barriers

4010-441

Principles of Concurrent System Design

Synchronizers

- A ***synchronizer*** is any object that coordinates the control of threads based on its state.
- The basic mechanisms in java.util.concurrent are:
  - ***Latches***: gate, or switch that allows one or more threads to wait until a set of operations being performed in other threads complete.
  - ***Barriers***: allows a set of threads to wait for each other at a common barrier point.
- Latches are for waiting for events
  - **`CountDownLatch`**
- Barriers are for waiting for other threads
  - **`CyclicBarrier`**

CountDownLatch

- CountDownLatches have a count that decrements towards zero.
- Threads can wait for a latch to reach zero.
- When zero is reached, all waiting threads (and any that arrive later and try to wait) are unblocked.
- Latches are one-shot, latch remains open when count == 0
- The CountDownLatch class allows us to **coordinate** the starting and stopping of threads. Typical uses are :
  - we can make several threads **start at the same time**;
  - we can **wait for several threads** to finish
- Using CountDownLatch for starting and stopping threads in timing tests. (JCIP p96 - Listing 5.11)
- **`CountDownLatch`** JavaDoc

CountDownLatch

- **Why do we need a latch to coordinate the start of threads in the timing test example?**
- **Couldn’t we just create the threads and then start them from within a loop using thread.start()?**

Launch Threads After All Started

```
class Driver {
  private static final int N = . . . ;

  void main()  throws InterruptedException {
    CountDownLatch startSignal =
          new CountDownLatch(1);

    for (int i = 0; i < N; ++i)    // create and start threads
      new Thread(
        new Worker(startSignal)
      ).start();

    startSignal.countDown() ;

    . . .
  }
}
```

```
class Worker implements Runnable {
  private final CountDownLatch startSignal;

   Worker(CountDownLatch startSignal) {
       this.startSignal = startSignal;
   }

  public void run() {
    try {
      startSignal.await();
      doWork();
    } catch (InterruptedException ex) { } ;
  }

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

Wait for Worker Threads to Complete

```
class Driver {
  private static final int N = . . . ;
  void main()  throws InterruptedException {
    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();

    startSignal.countDown() ;

    // do something

    doneSignal.await() ;

    // cleanup

  }
}
```

```
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();
    } catch (InterruptedException ex) {
        // whatever
    } finally
        doneSignal.countDown()
    };
  }

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

CyclicBarrier

- The barrier is constructed using:
  - number of threads that will be participating in the parallel operation;
  - optionally, a method to run at the end of each stage that amalgamates the results of that iteration
- At the completion of each iteration:
  - each thread completes its portion of the work and **calls the barrier's await()** method;
  - the await() method **returns *only* when**:
    - *all* threads have called await();
    - the **amalgamation** method has run (the barrier calls this on the last thread to call await() before releasing the awaiting threads).

`CyclicBarrier` Java Doc

What types of applications could you envision using barriers?

Source: http://www.javamex.com/tutorials/threads/

CyclicBarrier

- If *any* of the threads is **interrupted or times out** while waiting for the barrier, then **the barrier is "broken"** and *all* other waiting threads receive a BrokenBarrierException.

Is this the behavior you would expect from CyclicBarrier?
Give an example of why this would/would not be useful.

Coordinating computation in a cellular automaton with CyclicBarrier. (JCIP p102 – Listing 5.15)

`CyclicBarrier` Java Doc

Source: http://www.javamex.com/tutorials/threads/

CountDownLatch or CyclicBarrier?

- The **`CountDownLatch`** class is useful for various types of "one-off" thread coordination, in particular *setting threads off* together. However, it has at least two features that can be inconvenient in certain situations:
  - a given CountDownLatch can only be used once, making it inconvenient for operations that occur in *stages*, with intermediate results from the different threads needing to be amalgamated between stages;
  - the CountDownLatch doesn't explicitly allow one thread to tell the others to "stop waiting", which is sometimes useful, for example, if an error occurs in one of the threads.
- The **`CyclicBarrier`** is generally more useful than CountDownLatch in cases where:
  - a multithreaded operation occurs in *stages* or *iterations*, and;
  - a single-threaded operation is required between stages/iterations, for example, to combine the results of the previous multithreaded portion.

Source: http://www.javamex.com/tutorials/threads/

Example: Concurrent Step Oriented Game

```
class Player extends Thread {
  private final GameState gs ;
  private final CountDownLatch latch ;
  private final CyclicBarrier cb ;
  // . . . remaining state

  Player(GameState gs, CountDownLatch latch, CyclicBarrier cb) {
    this.gs = gs ; this.latch = latch ; this.cb = cb ;
  }

  public void run() {
    latch.await() ;
    while( ! done() ) {
      computeNextAction() ;
      try {
        barrier.await();
      } catch (InterruptedException ex) { return; }
      } catch (BrokenBarrierException ex) { return; }
    }
  }
  public Change action() { . . . }
  public void computeNextAction() { . . . }
  public boolean done() { . . . }
}
```

Example: Concurrent Step Oriented Game

```
class Advance implements Runnable {
  private final Player p1 ;
  private final Player p2 ;
  private final GameState gs ;
  private final CyclicBarrier cb ;

  public void setState(Player p1, Player p2, GameState gs, CyclicBarrier cb) {
    this.p1 = p1 ; this.p2 = p2 ;
    this.gs = gs ; this.cb = cb ;
  }

  public void run() {
    gs.merge(p1.action(), p2.action()) ;
    gs.incrTime() ;
    cb.reset() ;
  }
}
```

Example: Concurrent Step Oriented Game

```
class Driver {
  private final CountDownLatch latch = new CountDownLatch(1) ;

  private final Runnable advance = new Advance() ;
  private final CyclicBarrier cb = new CyclicBarrier(2, advance);

  private final GameState gs = new GameState() ;

  private final Player p1 = new Player(gs, latch, barrier) ;
  private final Player p2 = new Player(gs, latch, barrier) ;

  public Driver() {
    advance.setState(p1, p2, gs, cb)
    p1.start() ;
    p2.start() ;
    latch.countDown() ;
  }
}

class GameState {  . . . }

class Change { . . . }
```