Advanced Java Concurrency Framework

By Nisarg Shah
Rutvi Joshi

Concurrency in Java

- Developers had to create their own solution to solve concurrency problems
- Such codes may be prone to many bugs
- The existing primitives in Java do not provide much granularity for concurrency
- These primitives may not always scale for real situations
- The Java Concurrency Framework abstracts some of this complexity away

The Framework

- Replaced existing and limited Java support for concurrency
- Even updated JVMs to include support for new functionality
- Provided with utility classes commonly used for concurrent programming in Java with these packages:
  - java.util.concurrent
  - java.util.locks
  - java.util.atomic
- Easy to use, understand, provides a standard, higher performance

What's New ?

- Synchronizers
- Callable <T> and Future <T>
- Executor Service and Thread Pool
- Atomic <T>
- Locks and Latches
- Fork-Join
- Queues
- **Software Transactional Memory**
- **Barriers**
- **Exchangers**
- **Concurrent Collections**

Software Transactional Memory

- Provides an alternative to lock-based and actor-based concurrency mainly for applications with shared memory
- Similar to database transactions with ACID characteristics (though minus Durability)
- Works best with programming languages that differentiate mutable and immutable variables and require transactions to change mutable variables like Clojure and Haskell
- There are multiple STM implementations for various languages including Java (apart from Clojure and Akka)

STM Implementations

- Deuce STM
  - Open source Java support for STM
  - Generates highly concurrent code
  - Extensible Framework
- Multiverse
  - Language Independent, all languages that run on the JVM
  - 2011 release of Multiverse 0.7
- AtomJava
  - Polygot extension that performs source-to-source translation
  - Producing Java code which runs normally
- JVSTM
  - Java Versioned Software Transactional Memory
  - Two Core Concepts: Versioned Boxes and Transactions

JVSTM Example

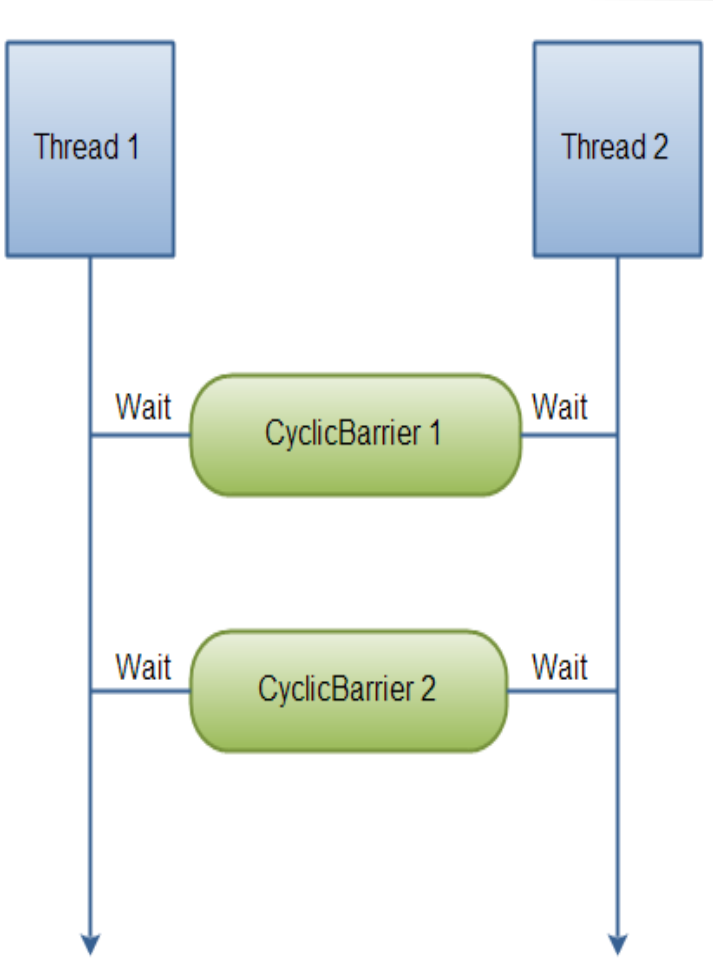
- JVSTM basically creates versioned copies of the shared data and each such copy is related to a Transaction(TX)
- Consider a counter which is incremented and displayed by a bunch of threads each defined as a TX
- Depending on which concurrent TX is working on counter, That corresponding version is updated or displayed and then committed to save that version
- Each TX updates its version, so an older TX updates its version of the counter and if it is not the current version then the changes do not percolate

```
public class CFOCounter {
  private VBox<Long> count = new VBox<Long>(0L);

  private PerTxBox<Long> toAdd =
    new PerTxBox<Long>(0L) {
      public void commit(Long value) {
        count.put(count.get() + value);
      }
    };

  public long getCount() {
    return count.get() + toAdd.get();
  }

  public @Atomic void inc() {
    toAdd.put(toAdd.get() + 1);
  }
}
```

```
new Thread() {
    public void run() {
    while (true) {
        Transaction.begin();
        counter.inc();
        Transaction.commit();
        try {
        Thread.sleep(100);
        } catch (Exception e) {
        // ok
        }
    }
    }
}.start();

new Thread() {
    public void run() {
    while (true) {
        Transaction.begin();
        System.out.println("Value = " + counter.getCount());
        Transaction.commit();
        try {
        Thread.sleep(100);
        } catch (Exception e) {
        // ok
        }
    }
    }
}.start();
}
```

Barriers (I)

- Similar to Latches, difference is the barrier waits on all the threads to reach a common point.
- In Java the barriers are referred to as CyclicBarrier because they can be reused.
- The threads wait on the Barrier by calling await() on it.
- There is BrokenBarrierException if the await() for a particular thread times out waiting on the barrier.



Barriers (II)

- Partitions should be small so that threads do not have to wait for a long time.

- Useful when all threads need to reach a point simultaneously for further operation.

- Better then Latches as Barriers can be reused.

- Syntax

-CyclicBarrier(int no_of _threads)

-CyclicBarrier(int no_of_threads, Runnable BarrierAction)

BarrierAction – task done after barrier is tripped

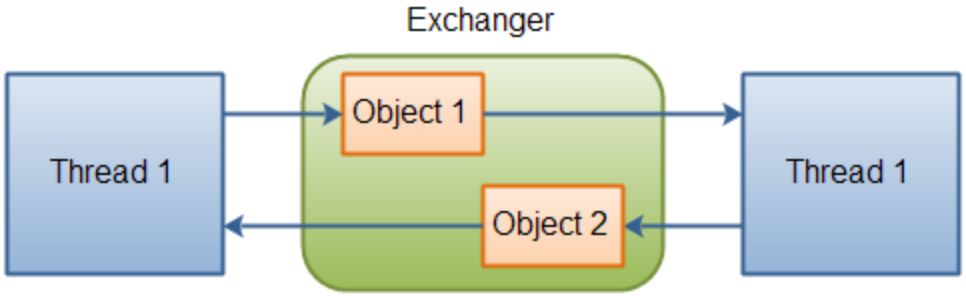
Example

```
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 ex) {
          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();
  }
}
```

Explanation

- In the above code be pass a matrix to a solver class to achieve some processing on the rows of the matrix.
- We create a number of threads equal to the number of rows (N).
- In our case it is a method where we merge all processed rows.
- We generate N worker threads which process the respective rows and await on the barrier.
- In mergerows() depending on the row number we can combine the results of independent rows.
- Thus we achieve concurrency without using locks and latches in a much neater way.

Exchangers



- Two way Barrier.
- Thread safe way of exchanging objects between threads.
- Useful when two threads perform asymmetric tasks.
- Syntax :

- Exchange(TypeObject, long timeout, timeunits):

Exchange object when both threads at exchange point. Time out after timeunits if other thread does not arrive at exchange point.

Example

```
class FillAndEmpty {
  Exchanger<DataBuffer> exchanger = new Exchanger();
  DataBuffer initialEmptyBuffer = ... a made-up type
  DataBuffer initialFullBuffer = ...

  class FillingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialEmptyBuffer;
      try {
        while (currentBuffer != null) {
          addToBuffer(currentBuffer);
          if (currentBuffer.full())
            currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ... }
    }
  }

  class EmptyingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialFullBuffer;
      try {
        while (currentBuffer != null) {
          takeFromBuffer(currentBuffer);
          if (currentBuffer.empty())
            currentBuffer = exchanger.exchange(currentBuffer);
        }
      } catch (InterruptedException ex) { ... handle ...}
    }
  }

  void start() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
  }
}
```

Explanation

- In the above code there are two threads. One threads fills up a data buffer and the other thread reads the buffer.
- Here the FillingLoop fills up the buffer until the buffer is completely full and then exchanges the filled buffer for an empty buffer obtained from the Emptyingloop task.
- Here both threads need to wait at the exchange step for both the buffer conditions to satisfy.
- This approach involves least number of excahnge operations but can lead to delay in case of unpredictable input data rate.
- Another approach that can be used is exchanging partially filled data buffers thus reducing delay but increasing the exchanges leading to more overhead.

Concurrent Collections

- Includes data structures used for thread safe programs.
- In JDK 1.2 there were synchronization collections such as Vector and HashTables.
- General operations involve iterations(continuous fetching from data structure), Navigation , put-if-absent operation.
- In addition to these to collections we require additional locks to ensure expected behavior.
- This makes the performance of these collections very similar to sequential version.

Example

```
public static Object getLast(Vector list) {
    int lastIndex = list.size() - 1;
    return list.get(lastIndex);
}

public static void deleteLast(Vector list) {
    int lastIndex = list.size() - 1;
    list.remove(lastIndex);
}
```

Vectors used without Synchronization.
In this case if one thread deletes the last entry and the another thread tries to read that value it throws an ArrauOutOfBoundsException.

```
public static Object getLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        return list.get(lastIndex);
    }
}

public static void deleteLast(Vector list) {
    synchronized (list) {
        int lastIndex = list.size() - 1;
        list.remove(lastIndex);
    }
}
```

The Synchronized keyword is used to ensure that only one thread does the change to the list.
This reduces the performance of the code in terms of execution time.

Concurrent Collections (II)

- In JDK 5.0 the old synchronization collections are replaced by new Concurrent collections.
- They include ConcurrentHashMaps and ConcurrentOnWriteArrayList which are replacements for the old Synchronized list and HastTables.
- There are two new types which are Queue and Blocking Queues.
- Java 6 also includes ConcurrentSkipListMap and ConcurrentSkiplistSet which are replacements for synchronized SortedMap and SortedSet.

ConcurrentHashMap

- Improved Concurrency and Scalability.
- Finer-grained locking mechanism called as striping.
- Striping involves splitting up of locks on the HashMap rather than having just one lock.
- This allows multiple threads to read and write to the data structure thus improving throughput.
- There are few trade-offs in terms of the methods operating on the entire structure such as isEmpty and size as they return approximate value.

ConcurrentHashMap

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
    // Insert into map only if no value is mapped from K
    V putIfAbsent(K key, V value);

    // Remove only if K is mapped to V
    boolean remove(K key, V value);

    // Replace value only if K is mapped to oldValue
    boolean replace(K key, V oldValue, V newValue);

    // Replace value only if K is mapped to some value
    V replace(K key, V newValue);
}
```

- We can very easily achieve complex functions such as PutIfAbsent, remove replace.
- These functions are thread safe and very efficient.

CopyOnWriteArrayList

- Immutable structure
  - Creates a new copy of the List on any change
- Unlike ConcurrentHashMap only one write operation can be performed at a time.
- The Write operation does not block the read threads.
- The read operation will never return an intermediate value.
- Huge advantage for a case where reads are more than writes.
- AtomicArray give better results when the there are no write operations involved.

Conclusion

- This was a brief overview of some of the other utility classes of the Java Framework
- There is lots more still left to be explored
- But it reinforces that, compared to the manually trying to synchronize all threads and the contentions surrounding it..
- Java Framework provides a lot of flexibility to the concurrent developer
- Thank You

References

- Books:
  - The Pragmatic Programmers: Programming Concurrency on the JVM
  - Java Concurrency in Practice

- Links:
  - http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html
  - http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CyclicBarrier.html
  - http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Exchanger.html