Java Concurrency Utilities

Based on JavaOne talk given by
David Holmes & Brian Goetz

Overview

• Rationale and goals for JSR 166
  – Java community process – concurrency utilities
• Executors – thread pools and scheduling
• Futures
• Concurrent Collections
• Locks, conditions and synchronizers
• Atomic variables
Why Concurrency Utilities

- Java’s built-in concurrency primitives -- wait(), notify(), and synchronized – are:
  - Hard to use correctly
  - Easy to use incorrectly
  - Too low level for many applications
  - Can lead to poor performance if used incorrectly
  - Leave out lots of useful concurrency constructs

Goals

- Provide efficient, correct & reusable concurrency building blocks
- Enhance scalability, performance, readability, maintainability, and thread-safety of concurrent Java applications
Background

- Utilities are in the java.util.concurrent package
  - based on Doug Lea’s EDU.oswego.cs.dl.util.concurrent package
- APIs take advantage of native JVM constructs & Java Memory Model guarantees specified in JSR 133

Building Blocks

- Executors, Thread Pools, and Futures
- Concurrent collections:
  - BlockingQueue, ConcurrentHashMap, CopyOnWriteArray
- Locks and Conditions
- Synchronizers: Semaphores, Barriers, etc.
- Atomic Variables
  - Low-level compare-and-set operation
Executor

- Standardizes asynchronous invocation
- Separates job submission from execution policy
  - `anExecutor.execute(aRunnable)`
  - *not new Thread(aRunnable).start()*
- Two code styles supported:
  - Actions: `Runnables`
  - Functions: `Callables`
  - Also has lifecycle mgmt: cancellation, shutdown, etc.
- Executor usually created via `Executors` factory class
  - Configures `ThreadPoolExecutor`
  - Customizes shutdown methods, before/after hooks, saturation policies, queuing

Executor & ExecutorService

- ExecutorService adds lifecycle management to Executor

```java
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    // other convenience methods for submitting tasks
}
```
Creating Executors

- Executors factory methods

```java
public class Executors {
    static ExecutorService newSingleThreadedExecutor();
    static ExecutorService newFixedThreadPool(int n);
    static ExecutorService newCachedThreadPool(int n);
    static ScheduledExecutorService newScheduledThreadPool(int n);
    // additional versions & utility methods
}
```

(Not) Executor Example

- Thread per message Web Server (no limit on thread creation)

```java
class WebServer {
    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {handleRequest(connection);}
            };
            new Thread(r).start();
        }
    }
}
```
Executor Example

- Thread pool web server - better resource management
  
  ```java
  class WebServer {
      Executor pool = Executors.newFixedThreadPool(7);
      public static void main(String[] args) {
          ServerSocket socket = new ServerSocket(80);
          while (true) {
              final Socket connection = socket.accept();
              Runnable r = new Runnable() {
                  public void run() {handleRequest(connection);}
              };
              pool.execute(r);
          }
      }
  }
  ```

Future and Callable

- Callable is functional analog of Runnable
  ```java
  interface Callable<V> {
      V call() throws Exception;
  }
  ```
- Future represents asynchronous tasks
- Future holds result of async call, norm to a Callable
  ```java
  interface Future<V> {
      V get() throws InterruptedException, ExecutionException;
      V get(long timeout, TimeUnit unit);
      boolean cancel(boolean mayInterrupt);
      boolean isCancelled();
      boolean isDone();
  }
  ```
Future Example

- See: FutureTaskStringReverser.java

Another Future Example

- Implementing a cache with Future

```java
public class Cache<K, V> {
    Map<K, Future<V>> map = new ConcurrentHashMap();
    Executor executor = Executors.newFixedThreadPool(8);
    public V get (final K key) {
        Future<V> f = map.get(key);  // null if key not found
        if (f == null) {
            Callable<V> c = new Callable<V>() {
                public V call() {// compute value associated with key}
                
                public V call() {// compute value associated with key}
                
                return f.get();
            }
            Future old = map.putIfAbsent(key, f); // return null if key not found. put(key,f)
            if (old == null)  // otherwise return get(key)
                executor.execute(f);
            else
                f = old;
        }
        return f.get();
    }
}
```
ScheduledExecutorService

- For deferred and recurring tasks, can schedule
  - Callable or Runnable to run once with a fixed delay after submission
  - Schedule a Runnable to run periodically at a fixed rate
  - Schedule a Runnable to run periodically with a fixed delay between executions
- Submission returns a ScheduledFutureTask handle which can be used to cancel the task
- Like Timer, but supports pooling and is more robust

Concurrent Collections

- Pre-1.5 Java class libraries had few concurrent (vs Synchronized) classes
  - Synchronized collections: Hashtable, Vector, and Collections.synchronizedXXX
    - Often required locking during iteration
    - Locking becomes is a source of contention
- Java 1.5 concurrent collections:
  - Allow multiple operations to overlap
    - Some differences in semantics
Queue interface added to java.util

```java
interface Queue<E> extends Collection<E> {
    boolean offer(E x); // try to insert.
    E poll(); // return null if empty
    E remove() throws NoSuchElementException;
    E peek(); // return null if empty
    E element() throws NoSuchElementException;
}
```

• Thread-safe and non-thread safe implementations
  – Non-thread-safe - LinkedList
  – Non-thread-safe - PriorityQueue
  – Thread-safe non-blocking - ConcurrentLinkedQueue

Blocking Queues

• Extends Queue to provide blocking operations
  – Retrieval: wait for queue to become nonempty
  – Insertion: wait for capacity to be available
• Common in producer-consumer designs
• Can support multiple producers and consumers
• Can be bounded or unbounded
• Implementations provided:
  – LinkedBlockingQueue (FIFO, may be bounded)
  – PriorityBlockingQueue (priority, unbounded)
  – ArrayBlockingQueue (FIFO, bounded)
  – SynchronousQueue (rendezvous channel)
Producer-Consumer Examples

- See:
  - ProducerConsumerPrimitive.java (wait/notify)
  - ProducerConsumerConcUtil.java (BlockingQueue)

Concurrent Collections

- **ConcurrentHashMap** - Concurrent (scalable) alt. to Hashtable or Collections.synchronizedMap
  - Multiple reads can overlap each other
  - Reads can overlap writes
  - Retrieval operations reflect the results of the most recently completed update operations holding at onset of operation
  - Up to 16 writes can overlap
  - Iterators do not throw ConcurrentModificationException

- **CopyOnWriteArrayList**
  - Optimized for case where iteration is much more frequent than insertion or removal. E.g., event listeners
Performance Comparison

- **ConcurrentHashMap** vs. **Collections.synchronizedMap**
- See **HashMapPerfTest.java**
- **Note**: incrementCount() is not safe

Locks and Lock Support

- High-level locking interface. Adds non-blocking lock acquisition

```java
interface Lock {
    void lock();
    void lockInterruptibly() throws IE;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws IE;
    void unlock();
    Condition newCondition() throws UnsupportedOperationExce$$
ReentrantLock

- Flexible, high-performance lock implementation
- **ReentrantLock** implements a reentrant mutual exclusion lock with the same semantics as built-in monitor locks, but with extra features
  - Can interrupt a thread waiting to acquire a lock
  - Can specify a timeout while waiting for a lock
  - Can poll for lock availability
  - Multiple wait-sets per lock via the **Condition** interface
- Outperforms built-in monitor locks in most cases, but slightly less convenient to use (requires finally block to release lock)

Lock Example

- Locks not automatically released. Must release lock in **finally** block

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // perform operations protected by lock
} catch (Exception ex) {
    // restore invariants
} finally {
    lock.unlock();
}
```
Read/write Locks

- **ReadWriteLock** defines a pair of locks; one for readers; one for writers

  ```java
  interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
  }
  ```

- **ReentrantReadWriteLock** class
  - Provides reentrant read and write locks
  - Allows writer to acquire read lock
  - Allows writer to downgrade to read lock
  - Supports “fair” and “writer preference” acquisition

Read/Write Lock Example

```java
class RWDictionaryRWL {
  private final Map<String, Data> m = new TreeMap<String, Data>();
  private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
  private final Lock r = rwl.readLock();
  private final Lock w = rwl.writeLock();
  public Data get(String key) {
    r.lock();
    try { return m.get(key); }
    finally { r.unlock(); }
  }
  public Data put(String key, Data value)
    w.lock();
    try { return m.put(key, value); }
    finally { w.unlock(); }
}
```
Read/Write Lock Example

• See
  – RWDictionary.java & RWDictionaryRWL.java

Condition

• Condition lets you wait for a condition to hold

interface Condition {
    void await() throws IE;
    boolean await( long time, TimeUnit unit ) throws IE;
    long awaitNanos( long nanosTimeout) throws IE;
    void awaitUninterruptibly()
    boolean awaitUntil( Date deadline) throws IE;
    void signal();
    void signalAll();
}
• Many improvements over wait()/notify()
  – Multiple conditions per lock
  – Absolute and relative time-outs
  – Timed waits tell you why you returned
  – Convenient uninterruptible wait

---

**Condition Example**

class BoundedBufferCond {
  Lock lock = new ReentrantLock();
  Condition notFull = lock.newCondition();
  Condition notEmpty = lock.newCondition();
  Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws IE {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally { lock.unlock(); }
  }
}
public Object take() throws IE {
    lock.lock();
    try {
        while (count == 0) notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return x;
    } finally { lock.unlock(); }
}

Condition Example (cont.)

- Previous example in BoundedBufferCond.java
- See also: BoundedBufferPrim.java
Synchronizers

- Utilities for coordinating access and control
- **CountDownLatch** – allows one or more threads to wait for a set of threads to complete an action
- **CyclicBarrier** – allows a set of threads to wait until they all reach a specified barrier point
- **Semaphore** – Dijkstra counting semaphore, managing some number of permits
- **Exchanger** – allows two threads to rendezvous and exchange data, such as exchanging an empty buffer for a full one

CountDownLatch

- Latching variables are conditions that once set never change.
- Often used to start several threads, but have them wait for a signal before continuing
- See: CountDownLatchTest.java
CyclicBarrier

- Allows threads to wait at a common barrier point
- Useful when a fixed-sized party of threads must occasionally wait for each other
- Cyclic Barriers can be re-used after threads released
- Can run a Runnable command once per barrier point
  - After the last thread arrives, but before any are released
  - Useful for updating shared-state before threads continue
- See: CyclicBarrierEx1.java & CyclicBarrierEx2.java

Semaphore

- Semaphore maintain a logical set of permits
- acquire() blocks until a permit is free, then takes it
- release() adds a permit, releasing a blocking acquirer
- Often used to restrict the number of threads that can access some resource
  - But can be used to implement many sync disciplines
- See: SemaphoreTunnel.java & SemaphoreBuffer.java
Exchanger

- Synch. point where two threads exchange objects
- A bidirectional SynchronizedQueue
- Each thread presents some object on entry to the exchange() method, and receives the object presented by the other thread on return
- See ExchangerTest.java

Atomic Variables

- Holder classes for scalars, references and fields
- Supports atomic operations
  - Compare-and-set (CAS)
  - Get and set and arithmetic (where applicable)
- Ten main classes: \{ int, long, ref \} X \{ value, field, array \}
  - E.g. AtomicInteger useful for counters, sequences, statistics
- Essential for writing efficient code on MPs
  - Nonblocking data structures & optimistic algorithms
  - Reduce overhead/contention updating “hot” fields
- JVM uses best construct available on platform
  - CAS, load-linked/store-conditional, locks
Atomic Variables

• See: CounterTest.java