Advanced Java Concurrency Framework

Lin Zhang

Presentation executive summary

- This is beginner introduction about the modern Java concurrency API.
- Familiarity with Java language and concurrency programming is assumed.
- The structure of the presentation is as follows:
- -- An brief overview on concurrency programming
- -- The power and perils of concurrency
- -- The advanced Java concurrency framework with all the major classes and services discussed with code examples.
- -- Software engineering benefits the framework provides to developers will also be discussed

Overview – concurrency and threads

- What is concurrency?
- Concurrency is the ability to run several parts of a program or several programs in parallel. Concurrency can highly improve the throughput of a program if certain tasks can be performed asynchronously or in parallel.
- Almost every computer nowadays has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application.
- Java supports concurrency using threads
- -- A thread is a flow of execution in a process.

Concurrency and threads continued

- When we run a program, there is at least one thread of execution for its process.
- We can create threads to start additional flows of execution in order to perform additional tasks concurrently.
- The libraries or framework we use may also start additional threads behind the scene, like garbage collecting thread.

Process vs. threads

- The distinction between processes and threads is important.
- Process: A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.
- Threads: threads are so called lightweight processes which have their own call stack but an access shared data. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data, when this happens in Java will be explained in Java memory model part of this article.
- Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Concurrency on single-core/mult-icore processors

- On single-core:
- concurrent tasks are multiplexed or multitasked.
- But only one thread is executed at any given instance.

- On multi-core:
- more than one threads are executed at any given instance.
- The number depends on the number core available on the processor.

The power and perils of concurrency

- Power:
- Making apps more responsive
- Making apps faster
- Perils:
- Starvation: A slow thread being starved of a resource
- by a fast thread
- Deadlock: two or more threads are waiting on each other for some action or resource
- Race conditions: two threads compete to use the same resource or data

Solutions for these perils

- Earlier version of Java concurrency API(such as synchronization primitives)
- Disadvantages:
- It takes time and resource to create threads.
- Too many threads can lead to reduced performance, as the CPU needs to switch between these threads.
- We cannot easily control the number of threads, which may leads to out of memory errors due to too many threads.
- Overly conservative synchronization limits performance and scalability

Where does this lead us to?

- The modern Java Concurrency Framework provides a set of safe and robust services that allow Java programmers to easily create code that will be able to take advantage of concurrent programming.

- Programmers can do concurrent programming without having to worry about all those complexity introduced by older version concurrency API.

About the Java concurrency framework

- Framework was in-part developed by Doug Lea and was available
- for three years before integration into J2SE 5.0
- Added to Java in J2SE 5.0 as Java Specification Request 166
- Replaced the existing and limited Java support for concurrency
- which of ten required developers to create their own solutions to
- solve concurrency problems
- Framework also caused JVMs to be updated to properly support
- the new functionality
- **Three main packages:**
- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks

Purpose

- Meant to have the same effect on Java as java.util.Collections framework
- Provides Java with set of utilities that are:
- Standardized
- Easy to use
- Easy to understand
- High quality
- High performance
- Useful in a large set of applications with a range of expertise from beginner to expert

Primary classes and services

- The main interfaces and classes in the framework are:
- Executors
- Thread Factory
- Futures
- Queues
- Conditions
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Fork-join API(supported by Java 7)

Executor

- An executor is simply an object that executes runnable tasks
- Decouples task submission from the details of how a task will
- executed
- Does not require task to be run asynchronously
  The framework provides two sub -interfaces and three
- implementations of the Executor interface:
- ExecutorService – extends base interface to shut-down termination and
- support Futures
- ScheduledExecutorService – extends ExecutorService to include delays in
- execution
- AbstractExecutorService – default implementation of ExecutorService
- ScheduledThreadPoolExecutor – extension of ThreadPoolExecutor that
- icludes services to delay thread execution
- ThreadPoolExecutor – implementation with a set of threads to run
- submitted tasks; minimizes thread-creation overhead since this Executor
- uses its own set of threads instead of creating new threads to execute tasks

Executor example

- Create the Runnable.

```
public class MyRunnable implements Runnable {
    private final long countUntil;

    MyRunnable(long countUntil) {
        this.countUntil = countUntil;
    }

    @Override
    public void run() {
        long sum = 0;
        for (long i = 1; i < countUntil; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

- Now run the runnables with the executor framework

Executor example

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Main {
        private static final int NTHREDS = 10;

        public static void main(String[] args) {
                ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
                for (int i = 0; i < 500; i++) {
                        Runnable worker = new MyRunnable(10000000L + i);
                        executor.execute(worker);
                }
                // This will make the executor accept no new threads
                // and finish all existing threads in the queue
                executor.shutdown();
                // Wait until all threads are finish
                while (!executor.isTerminated()) {

                }
                System.out.println("Finished all threads");
        }
}
```

Thread Factory

- A ThreadFactory enables a specific type of thread to be created in a standardized way without intervention from the client

- The following examples shows how a thread factory can be used to standardize the creation of a custom thread

Thread Factory example

```
static class DefaultThreadFactory implements ThreadFactory {
        static final AtomicInteger poolNumber = new AtomicInteger(1);
        final ThreadGroup group;
        final AtomicInteger threadNumber = new AtomicInteger(1);
        final String namePrefix;

        DefaultThreadFactory() {
            SecurityManager s = System.getSecurityManager();
            group = (s != null)? s.getThreadGroup() :
                                 Thread.currentThread().getThreadGroup();
            namePrefix = "pool-" +
                          poolNumber.getAndIncrement() +
                         "-thread-";
        }

```

Thread Factory example

```
    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
                              0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
    }
}
```

Futures

- A future is an object that holds the result of an asynchronous computation
- Methods are provided to check completion status via isDone()
- The computation can be cancelled via the cancell) method if the computation has already completed
- The result is retrieved via get() which will block until the computation is complete
- Futures can be returned by Executors or used directly in code

Future example

```
public class FutureExample {
    public static void main(String[] args) throws Exception {
        System.out.println("Using fixed thread pool:");
        ExecutorService executor = Executors.newFixedThreadPool(2);
        test(executor);

        System.out.println("\nUsing cached thread pool:");
        executor = Executors.newCachedThreadPool();
        test(executor);

        System.out.println("\nUsing single thread executor:");
        executor = Executors.newSingleThreadExecutor();
        test(executor);
    }
```

Future example

```
    private static void test(ExecutorService executor)
                                throws ExecutionException, InterruptedException {
        Counter counter = new Counter();

        Future<?> f1 = executor.submit(new Worker(counter, true, 10000));
        Future<?> f2 = executor.submit(new Worker(counter, false, 10000));

        // reject new tasks, must call in order to exit VM
        executor.shutdown();

        // wait for termination, much like Thread.join()
        f1.get();
        f2.get();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Future example

```
class Counter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.getAndIncrement();
    }

    public void decrement() {
        c.getAndDecrement();
    }

    public int getCount() {
        return c.get();
    }
}
```

Future example

```
class Worker implements Runnable {
    private Counter counter;
    private boolean increment;
    private int count;

    public Worker(Counter counter, boolean increment, int count) {
        this.counter = counter;
        this.increment = increment;
        this.count = count;
    }

    public void run() {
        for (int i = 0; i < this.count; i++) {
            if (increment) {
                this.counter.increment();
            }
            else {
                this.counter.decrement();
            }
        }
    }
}
```

Runnable and Callable

- Runnable do not return result.
- In case you expect your threads to return a computed result you can use java.util.concurrent.Callable. Callables allow to return values after competition.
- Callable uses generic to define the type of object which is returned.
- If you submit a callable to an executor the framework returns a java.util.concurrent.Future. This futures can be used to check the status of a callable and to retrieve the result from the callable.
- On the executor you can use the method submit to submit a Callable and to get a future. To retrieve the result of the future use the get() method.

Callable example

```
import java.util.concurrent.Callable;

public class MyCallable implements Callable<Long> {
        @Override
        public Long call() throws Exception {
                long sum = 0;
                for (long i = 0; i <= 100; i++) {
                        sum += i;
                }
                return sum;
        }

}
```

Callable example

```
public class CallableFutures {
        private static final int NTHREDS = 10;

        public static void main(String[] args) {

                ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
                List<Future<Long>> list = new ArrayList<Future<Long>>();
                for (int i = 0; i < 20000; i++) {
                        Callable<Long> worker = new MyCallable();
                        Future<Long> submit = executor.submit(worker);
                        list.add(submit);
                }
                long sum = 0;
                System.out.println(list.size());

```

Callable example

```
// Now retrieve the result
                for (Future<Long> future : list) {
                        try {
                                sum += future.get();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println(sum);
                executor.shutdown();
        }
}
```

Queues

- Queues are a syunchroonized strucutures to hold tasks before being executed in the Java Concurrent Framework
- Standard queue commands like offer(), remove(), poll() and others are available
- Various forms in java.util.concurrency:
- AbstractQueue, ArrayBlockQueue, BlockingQueue, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue

Queue example

```
private BlockingQueue workQueue = new LinkedBlockingQueue();
private Map commandQueueMap = new ConcurrentHashMap();

public SynchronousQueue addCommand(Command command) {
    SynchronousQueue queue = new SynchronousQueue();
    commandQueueMap.put(command, queue);
    workQueue.offer(command);
    return queue;
}
```

Queue example

```
public Object call() throws Exception {
    try {
        Command command = workQueue.take();
        Result result = command.execute();
        SynchronousQueue queue = commandQueueMap.get(command);
        queue.offer(result);
        return null;
    } catch (InterruptedException e) {
        throw new WorkException(e);
    }
}
```

Queue example

- Now the consumer can safely poll with timeout on its request to have its Command executed.
- Command command;
- SynchronousQueue queue = commandRunner.addCommand(command);
- Result result = queue.poll(2, TimeUnit.MINUTES);
- if (result == null) {
- throw new CommandTooLongException(command);
- } else {
- return result;
- }

Conditions

- Provide a framework to allow a thread to suspend safely and allow another thread to enable a condition where execution can continue
- Replaces Java Object methods wait, notify and notifyAll that implemented a monitor
- Condition is bonded to a lock
- Condition can be
- Interruptable – condition causes thread to wait until signaled or interrupted before resuming
- Non interruptable – condition causes thread to wait until signaled before resuming
- Timed -- condition causes thread to wait a set amount of time (or until signaled/interrupted) before trying to resume

Condition example

- A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular Lock instance use its newCondition() method.
- As an example, suppose we have a bounded buffer which supports put and take methods. If a take is attempted on an empty buffer, then the thread will block until an item becomes available; if a put is attempted on a full buffer, then the thread will block until a space becomes available. We would like to keep waiting put threads
  and take threads in separate wait-sets so that we can use the optimization of only notifying a single thread at a time when items or spaces become available in the buffer. This can be achieved using two Condition instances.

Condition example

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

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

Condition example

```
public Object take() throws InterruptedException {
    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();
    }
  }
}
```

Synchronizers

- Semaphore – provides a way to limit access to a shared resource and can control the access to n resource
- Used with acquire and release methods in Java
- Java also supports fairness (or not) so that the order of an acquire
- request is honored by the semaphore (FIFO)
- Mutex – similar to a binary semaphore
- Implemented as Locks in Java
- Barrier – good for controlling the execution flow of a group of threads that need to synchronize at various points before continuing executing
- await is the main method in a barrier which causes the threads to wait
- until all of the threads in a barrier have called await before being released
- The constructor is called with the number of threads the barrier is
- managing

Barrier example

```
class BarrierExample
{
   static class MyThread1 implements Runnable
   {
      public MyThread1(Barrier barrier)
      {
         this.barrier = barrier;
      }
      public void run()
      {
         try
         {
            Thread.sleep(1000);
            System.out.println("MyThread1 waiting on barrier");
            barrier.block();
            System.out.println("MyThread1 has been released");
         } catch (InterruptedException ie)
         {
            System.out.println(ie);
         }
      }

```

Barrier example

```
private Barrier barrier;

  }
  static class MyThread2 implements Runnable
  {
     Barrier barrier;

     public MyThread2(Barrier barrier)
     {
        this.barrier = barrier;
     }
     public void run()
     {
        try
        {
           Thread.sleep(3000);
           System.out.println("MyThread2 releasing blocked threads\n");
           barrier.release();
           System.out.println("MyThread1 releasing blocked threads\n");
        } catch (InterruptedException ie)
        {
           System.out.println(ie);
        }
     }
  }
```

Barrier example

```
public static void main(String[] args) throws InterruptedException
   {
      /*
       *    MyThread1          MyThread2
       *       ...                ...
       *       BR.block();          ...
       *       ...                BR.release();
       */
      Barrier BR = new Barrier();
      Thread t1 = new Thread(new BarrierExample.MyThread1(BR));
      Thread t2 = new Thread(new BarrierExample.MyThread2(BR));
      t1.start();
      t2.start();
      t1.join();
      t2.join();
   }
}
```

Atomic variables

- Atomic variables ensure that access to the variable happens as a
- single instruction, preventing more than one thread from accessing the value at the same time
- java.util.concurrent.atomic implements a number of variables to
- enable atomic execution without using an outside lock while still being thread-safe
- boolean, int, arrays, etc.

  Extends the existing volatile Java behavior

  Basic set of atomic methods: get(), set()
- compareAndSet(<type> expect, <type> update) – compares the current value to expect and if equal, sets the value to update

Atomic variable example

```
public class AtomicCounter {
    private final AtomicInteger value = new AtomicInteger(0);

    public int getValue(){
        return value.get();
    }

    public int getNextValue(){
        return value.incrementAndGet();
    }

    public int getPreviousValue(){
        return value.decrementAndGet();
    }
}
```

Atomic variable example

- The incrementAndGet() and decrementAndGet() methods are two of the numeric operations provided by the AtomicLong and AtomicInteger classes. You also have getAndDecrement(), getAndIncrement(), getAndAdd(int i) and addAndGet().

Concurrentmap

- ConcurrentMap and ConcurrentNavigableMap provide
- thread-safe interfaces to Map and NavigableMap
- ConcurrentHashMap implements ConcurrentMap to
- provide a thread-safe hash map
- ConcurrentSkipListMap implements
- ConcurrentNavigableMap to provide a thread-safe skip
- list
- The following example shows a ConcurrentHashMap
- used between three threads to setup a simple
- department store with items going in an out of stock

Locks

- A lock controls access to a shared resource
- Typically access control is limited to one thread at a time
- A more flexible option over Java's built in synchronization and monitors
- With more flexibility, comes more complexity and care to
- create thread-safe code
- Different types of locks
- Reentrant
- Read/Write – allows multiple threads to read a resource, but only
- one to write the resource. A read cannot happen at the same time as a write though.
- Also known as a mutex

Java 7 Fork-join API

- Java 7 brings a specialization of ExecutorService with improved efficiency and performance – the fork-join API
- ForkJoinPool class dynamically manages threads based on the number of availabe processors and task demand.
- Fork-join employs work-stealing where threads pick up tasks created by other active tasks. This provides better performance and utilization of threads.
- This API is very useful for problems that can be broken down recursively until small enough to run sequentially.

References

- Programming Concurrency on the JVM, Mastering Synchronization, STM, and Actors, Venkat Subramaniam
- Java Concurrency / Multithreading - Tutorial
- http://www.vogella.de/articles/JavaConcurrency/article.html#concurrency_overview
- Java concurrency framework
- http://www.pascal-man.com/navigation/faq-java-browser/java-concurrent/csci5448_daugherty_jcf_pres.pdf