W

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

May 22, 2015

Welcome to the third part of my tutorial series about multi-threaded programming in Java 8. This tutorial covers two important parts of the Concurrency API: Atomic Variables and Concurrent Maps. Both have been greatly improved with the introduction of lambda expressions and functional programming in the latest Java 8 release. All those new features are described with a bunch of easily understood code samples. Enjoy!

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

For simplicity the code samples of this tutorial make use of the two helper methods `sleep(seconds)` and `stop(executor)` as defined here.

AtomicInteger

The package `java.concurrent.atomic` contains many useful classes to perform atomic operations. An operation is atomic when you can safely perform the operation in parallel on multiple threads without using the `synchronized` keyword or locks as shown in my previous tutorial.

Internally, the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs. Those instructions usually are much faster than synchronizing via locks. So my advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.

Now let's pick one of the atomic classes for a few examples: `AtomicInteger`

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(atomicInt::incrementAndGet));

stop(executor);

System.out.println(atomicInt.get());    // => 1000
```

By using `AtomicInteger` as a replacement for `Integer` we're able to increment the number concurrently in a thread-safe manor without synchronizing the access to the variable. The method `incrementAndGet()` is an atomic operation so we can safely call this method from multiple threads.

AtomicInteger supports various kinds of atomic operations. The method `updateAndGet()` accepts a lambda expression in order to perform arbitrary arithmetic operations upon the integer:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 2000
```

The method `accumulateAndGet()` accepts another kind of lambda expression of type `IntBinaryOperator`. We use this method to sum up all values from 0 to 1000 concurrently in the next sample:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.accumulateAndGet(i, (n, m) -> n + m);
        executor.submit(task);
```

```
    });

stop(executor);

System.out.println(atomicInt.get());    // => 499500
```

Other useful atomic classes are AtomicBoolean, AtomicLong and AtomicReference.

LongAdder

The class `LongAdder` as an alternative to `AtomicLong` can be used to consecutively add values to a number.

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(adder::increment));

stop(executor);

System.out.println(adder.sumThenReset());   // => 1000
```

LongAdder provides methods `add()` and `increment()` just like the atomic number classes and is also thread-safe. But instead of summing up a single result this class maintains a set of variables internally to reduce contention over threads. The actual result can be retrieved by calling `sum()` or `sumThenReset()`.

This class is usually preferable over atomic numbers when updates from multiple threads are more common than reads. This is often the case when capturing statistical data, e.g. you want to count the number of requests served on a web server. The drawback of `LongAdder` is higher memory consumption because a set of variables is held in-memory.

LongAccumulator

LongAccumulator is a more generalized version of LongAdder. Instead of performing simple add operations the class `LongAccumulator` builds around a lambda expression of type `LongBinaryOperator` as demonstrated in this code sample:

```
LongBinaryOperator op = (x, y) -> 2 * x + y;
LongAccumulator accumulator = new LongAccumulator(op, 1L);
```

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10)
    .forEach(i -> executor.submit(() -> accumulator.accumulate(i)));

stop(executor);

System.out.println(accumulator.getThenReset());     // => 2539
```

We create a LongAccumulator with the function `2 * x + y` and an initial value of one. With every call to `accumulate(i)` both the current result and the value `i` are passed as parameters to the lambda expression.

A `LongAccumulator` just like `LongAdder` maintains a set of variables internally to reduce contention over threads.

ConcurrentMap

The interface `ConcurrentMap` extends the map interface and defines one of the most useful concurrent collection types. Java 8 introduces functional programming by adding new methods to this interface.

In the next code snippets we use the following sample map to demonstrates those new methods:

```
ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

The method `forEach()` accepts a lambda expression of type `BiConsumer` with both the key and value of the map passed as parameters. It can be used as a replacement to for-each loops to iterate over the entries of the concurrent map. The iteration is performed sequentially on the current thread.

```
map.forEach((key, value) -> System.out.printf("%s = %s\n", key, value));
```

The method `putIfAbsent()` puts a new value into the map only if no value exists for the given key. At least for the `ConcurrentHashMap` implementation of this method is thread-safe just like `put()` so you don't have to synchronize when accessing the map concurrently from different threads:

```
String value = map.putIfAbsent("c3", "p1");
System.out.println(value);    // p0
```

The method `getOrDefault()` returns the value for the given key. In case no entry exists for this key the passed default value is returned:

```
String value = map.getOrDefault("hi", "there");
System.out.println(value);    // there
```

The method `replaceAll()` accepts a lambda expression of type `BiFunction`. BiFunctions take two parameters and return a single value. In this case the function is called with the key and the value of each map entry and returns a new value to be assigned for the current key:

```
map.replaceAll((key, value) -> "r2".equals(key) ? "d3" : value);
System.out.println(map.get("r2"));    // d3
```

Instead of replacing all values of the map `compute()` let's us transform a single entry. The method accepts both the key to be computed and a bi-function to specify the transformation of the value.

```
map.compute("foo", (key, value) -> value + value);
System.out.println(map.get("foo"));   // barbar
```

In addition to `compute()` two variants exist: `computeIfAbsent()` and `computeIfPresent()`. The functional parameters of these methods only get called if the key is absent or present respectively.

Finally, the method `merge()` can be utilized to unify a new value with an existing value in the map. Merge accepts a key, the new value to be merged into the existing entry and a bi-function to specify the merging behavior of both values:

```
map.merge("foo", "boo", (oldVal, newVal) -> newVal + " was " + oldVal);
System.out.println(map.get("foo"));   // boo was foo
```

ConcurrentHashMap

All those methods above are part of the `ConcurrentMap` interface, thereby available to all implementations of that interface. In addition the most important implementation `ConcurrentHashMap` has been further enhanced with a couple of new methods to perform parallel operations upon the map.

Just like parallel streams those methods use a special `ForkJoinPool` available via `ForkJoinPool.commonPool()` in Java 8. This pool uses a preset parallelism which depends on the number of available cores. Four CPU cores are available on my machine which results in a parallelism of three:

```
System.out.println(ForkJoinPool.getCommonPoolParallelism());  // 3
```

This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

We use the same example map for demonstrating purposes but this time we work upon the concrete implementation `ConcurrentHashMap` instead of the interface `ConcurrentMap`, so we can access all public methods from this class:

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

Java 8 introduces three kinds of parallel operations: `forEach`, `search` and `reduce`. Each of those operations are available in four forms accepting functions with keys, values, entries and key-value pair arguments.

All of those methods use a common first argument called `parallelismThreshold`. This threshold indicates the minimum collection size when the operation should be executed in parallel. E.g. if you pass a threshold of 500 and the actual size of the map is 499 the operation will be performed sequentially on a single thread. In the next examples we use a threshold of one to always force parallel execution for demonstrating purposes.

ForEach

The method `forEach()` is capable of iterating over the key-value pairs of the map in parallel. The lambda expression of type `BiConsumer` is called with the key and value of the current iteration step. In order to visualize parallel execution we print the current threads name to the console. Keep in mind that in my case the underlying `ForkJoinPool` uses up to a maximum of three threads.

```
map.forEach(1, (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n",
        key, value, Thread.currentThread().getName()));

// key: r2; value: d2; thread: main
// key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1
// key: han; value: solo; thread: ForkJoinPool.commonPool-worker-2
// key: c3; value: p0; thread: main
```

Search

The method `search()` accepts a `BiFunction` returning a non-null search result for the current key-value pair or `null` if the current iteration doesn't match the desired search criteria. As soon as a non-null result is returned further processing is suppressed. Keep in mind that `ConcurrentHashMap` is unordered. The search function should not depend on the actual processing order of the map. If multiple entries of the map match the given search function the result may be non-deterministic.

```
String result = map.search(1, (key, value) -> {
    System.out.println(Thread.currentThread().getName());
    if ("foo".equals(key)) {
        return value;
    }
    return null;
});
System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// ForkJoinPool.commonPool-worker-3
// Result: bar
```

Here's another example searching solely on the values of the map:

```
String result = map.searchValues(1, value -> {
    System.out.println(Thread.currentThread().getName());
    if (value.length() > 3) {
        return value;
    }
    return null;
});

System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// main
```

```
// ForkJoinPool.commonPool-worker-1
// Result: solo
```

Reduce

The method `reduce()` already known from Java 8 Streams accepts two lambda expressions of type `BiFunction`. The first function transforms each key-value pair into a single value of any type. The second function combines all those transformed values into a single result, ignoring any possible `null` values.

```
String result = map.reduce(1,
    (key, value) -> {
        System.out.println("Transform: " + Thread.currentThread().getName());
        return key + "=" + value;
    },
    (s1, s2) -> {
        System.out.println("Reduce: " + Thread.currentThread().getName());
        return s1 + ", " + s2;
    });

System.out.println("Result: " + result);

// Transform: ForkJoinPool.commonPool-worker-2
// Transform: main
// Transform: ForkJoinPool.commonPool-worker-3
// Reduce: ForkJoinPool.commonPool-worker-3
// Transform: main
// Reduce: main
// Reduce: main
// Result: r2=d2, c3=p0, han=solo, foo=bar
```

I hope you've enjoyed reading the third part of my tutorial series about Java 8 Concurrency. The code samples from this tutorial are hosted on GitHub along with many other Java 8 code snippets. You're welcome to fork the repo and try it by your own.

If you want to support my work, please share this tutorial with your friends. You should also follow me on Twitter as I constantly tweet about Java and programming related stuff.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap