JVM concurrency: Java 8 concurrency basics

See how Java 8 features make concurrent programming easier

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

08 April 2014

Java™ 8 incorporates new language features and added classes that give you easier ways to construct programs, including concurrent programs. Learn about new, powerful parallel-processing support in the language made possible by Java 8 extensions, including `CompletableFuture` and streams. You'll recognize similarities between these new features and some of the Scala capabilities that you explored in the first article of this series.

View more content in this series

Several of the main enhancements in the long-awaited Java 8 release relate to concurrency, including added classes in the `java.util.concurrent` hierarchy and the powerful new parallel *streams* feature. Streams are designed to be used with *lambda expressions*, a Java 8 addition that also makes many other aspects of day-to-day programming easier. (See the companion article on Java 8 language extensions for an introduction to lambda expressions and related `interface` changes.)

In this article, I first show you how the new `CompletableFuture` class makes asynchronous operations easier to coordinate. Next, I show how to use parallel streams — the big win for concurrency in Java 8 — to execute operations on sets of values in parallel. I finish with a look at how the new Java 8 features perform, including a comparison with some of the code from the first article in this series. (See Resources for a link to the full sample code for this article.)

Back to the `Futures`

The first article in this series gave you a quick introduction to both Java and Scala `Future`s. The (pre-Java 8) Java version is weak, supporting only two types of use: You can check whether the future has completed, or you can wait for the future to complete. The Scala version is much more flexible: You can execute callbacks when the future completes, and abnormal completions are handled in the form of `Throwable`s.

About this series

Now that multi-core systems are ubiquitous, concurrent programming must be applied more widely than ever before. But concurrency can be difficult to implement correctly, and you need new tools to help you use it. Many of the JVM-based languages are developing tools of this type, and Scala has been particularly active in this area. This series gives you a look at some of the newer approaches to concurrent programming for the Java and Scala languages.

Java 8 adds the `CompletableFuture<T>` class, which implements the new `CompletionStage<T>` interface and extends `Future<T>`. (All concurrency classes and interfaces discussed in this section are in the `java.util.concurrent` package.) `CompletionStage` represents a stage or step in a possibly asynchronous computation. The interface defines many different ways to chain `CompletionStage` instances with other instances or with code, such as methods to be called on completion (a total of 59 methods, compared to 5 methods in the `Future` interface).

Listing 1 shows the `ChunkDistanceChecker` class, based on the edit-distance comparison code in the first article.

Listing 1. `ChunkDistanceChecker`

```
public class ChunkDistanceChecker {
    private final String[] knownWords;

    public ChunkDistanceChecker(String[] knowns) {
        knownWords = knowns;
    }

    /**
    * Build list of checkers spanning word list.
    *
    * @param words
    * @param block
    * @return checkers
    */
    public static List<ChunkDistanceChecker> buildCheckers(String[] words, int block) {
        List<ChunkDistanceChecker> checkers = new ArrayList<>();
        for (int base = 0; base < words.length; base += block) {
            int length = Math.min(block, words.length - base);
            checkers.add(new ChunkDistanceChecker(Arrays.copyOfRange(words, base, base + length)));
        }
        return checkers;
        }
    ...
    /**
     * Find best distance from target to any known word.
     *
     * @param target
     * @return best
     */
    public DistancePair bestDistance(String target) {
        int[] v0 = new int[target.length() + 1];
        int[] v1 = new int[target.length() + 1];
        int bestIndex = -1;
        int bestDistance = Integer.MAX_VALUE;
        boolean single = false;
        for (int i = 0; i < knownWords.length; i++) {
            int distance = editDistance(target, knownWords[i], v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestIndex = i;
                single = true;
            } else if (bestDistance == distance) {
                single = false;
            }
        }
        return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :
            new DistancePair(bestDistance);
    }
}
```

Each instance of the `ChunkDistanceChecker` class handles checking a target word against an array of known words to find the best match. The static `buildCheckers()` method creates a `List<ChunkDistanceChecker>` from the entire array of known words and a desired block size. This `ChunkDistanceChecker` class is the basis for several concurrent implementations of best-match searches in this article, starting with the `CompletableFutureDistance0` class in Listing 2.

Listing 2. Edit-distance calculation using `CompletableFuture`

```
public class CompletableFutureDistance0 extends TimingTestBase {
    private final List<ChunkDistanceChecker> chunkCheckers;

    private final int blockSize;

    public CompletableFutureDistance0(String[] words, int block) {
        blockSize = block;
        chunkCheckers = ChunkDistanceChecker.buildCheckers(words, block);
    }
    ...
    public DistancePair bestMatch(String target) {
        List<CompletableFuture<DistancePair>> futures = new ArrayList<>();
        for (ChunkDistanceChecker checker: chunkCheckers) {
            CompletableFuture<DistancePair> future =
                CompletableFuture.supplyAsync(() -> checker.bestDistance(target));
            futures.add(future);
        }
        DistancePair best = DistancePair.worstMatch();
        for (CompletableFuture<DistancePair> future: futures) {
            best = DistancePair.best(best, future.join());
        }
        return best;
    }
}
```

Learn about lambdas

The lambda expression that I pass to the `supplyAsync()` method is a *capturing* lambda, because it references the `target` parameter value. Get an introduction to lambda expressions, and learn about the distinction between capturing and noncapturing lambdas, in "Java 8 language changes."

The Listing 2 `CompletableFutureDistance0` class shows one way of using `CompletableFuture`s for concurrent computations. The `supplyAsync()` method takes a `Supplier<T>` instance (a functional interface with the method returning a value of the type `T`) and returns a `CompletableFuture<T>` while queuing the `Supplier` to be run asynchronously. I pass a lambda expression to the `supplyAsync()` method in the first `for` loop to build a list of futures matching the `ChunkDistanceChecker` array. The second `for` loop waits for each future to complete (though most complete before this loop gets to them, because they execute asynchronously) and accumulates the best match from all results.

Building on `CompletableFutures`

In the first article in this series, you saw that with Scala `Future`s, you can attach completion handlers and combine futures in different ways. `CompletableFuture` provides similar flexibility for Java 8. In this section, you'll learn some of the ways to use these features, in the context of the edit-distance-checking code.

Listing 3 shows another version of the `bestMatch()` method from Listing 2. This one uses a completion handler with `CompletableFuture`, together with a couple of older concurrency classes.

Listing 3. `CompletableFuture` with completion handler

```
public DistancePair bestMatch(String target) {
    AtomicReference<DistancePair> best = new AtomicReference<>(DistancePair.worstMatch());
    CountDownLatch latch = new CountDownLatch(chunkCheckers.size());
    for (ChunkDistanceChecker checker: chunkCheckers) {
        CompletableFuture.supplyAsync(() -> checker.bestDistance(target))
            .thenAccept(result -> {
                best.accumulateAndGet(result, DistancePair::best);
                latch.countDown();
            });
    }
    try {
        latch.await();
    } catch (InterruptedException e) {
        throw new RuntimeException("Interrupted during calculations", e);
    }
    return best.get();
}
```

In Listing 3, the `CountDownLatch` is initialized to the number of futures created in the code. As I create each future, I attach a handler (in the form of a lambda instance of the `java.util.function.Consumer<T>` functional interface) using the `CompletableFuture.thenAccept()` method. The handler, which executes when the future completes normally, uses the `AtomicReference.accumulateAndGet()` method (added in Java 8) to update the best value found and then decrements the latch. Meanwhile, the main thread of execution enters the `try-catch` block and waits for the latch to release. After all the futures have completed, the main thread continues, returning the final best value found.

Listing 4 shows yet another variation of the `bestMatch()` method from Listing 2.

Listing 4. Combining `CompletableFutures`

```
public DistancePair bestMatch(String target) {
    CompletableFuture<DistancePair> last =
        CompletableFuture.supplyAsync(bestDistanceLambda(0, target));
    for (int i = 1; i < chunkCheckers.size(); i++) {
        last = CompletableFuture.supplyAsync(bestDistanceLambda(i, target))
            .thenCombine(last, DistancePair::best);
    }
    return last.join();
}

private Supplier<DistancePair> bestDistanceLambda(int i, String target) {
    return () -> chunkCheckers.get(i).bestDistance(target);
}
```

This code uses the `CompletableFuture.thenCombine ()` method to merge two futures by applying a `java.util.function.BiFunction` (in this case, the `DistancePair.best()` method) to the two results, returning a future for the result of the function.

Listing 4 is the most concise and perhaps cleanest version of the code, but it has the drawback that it creates an extra layer of `CompletableFuture`s to represent the combination of each chunk operation with the prior operations. As of the initial Java 8 release, this has the potential to cause a `StackOverflowException` which is lost within the code, resulting in the final future never completing. The bug is being addressed and should be fixed in a near-term future release.

`CompletableFuture` defines many variations on the methods used in these examples. When you use `CompletableFuture` for your applications, check the full list of completion methods and combining methods to find the one that best matches your needs.

`CompletableFuture` is best used when you're doing different types of operations and must coordinate the results. When you're running the same calculation on many different data values, parallel streams give you a simpler approach and likely better performance. The edit-distance-checking example is a better match with the parallel streams approach.

Streams

Streams, a major new feature of Java 8, work in conjunction with lambda expressions. Streams are essentially push iterators over a sequence of values. Streams can be chained with adapters to perform operations such as filtering and mapping, much like Scala sequences. Streams also have both sequential and parallel variations, again much like Scala sequences (though Scala has a separate class hierarchy for parallel sequences, whereas Java 8 uses an internal flag to indicate serial or parallel). Variations of streams exist for primitive `int`, `long`, and `double` types, along with typed object streams.

The new streams API is too complex to cover fully in this article, so I'll focus on the concurrency aspects. See the Resources section for more-detailed coverage of streams.

Listing 5 shows another variation of the edit-distance best-match code. This version uses the `ChunkDistanceChecker` from Listing 1 to do the distance calculations, and `CompletableFuture`s as in the Listing 2 example, but this time I use streams to get the best-match result.

Listing 5. `CompletableFuture` using streams

```
public class CompletableFutureStreamDistance extends TimingTestBase {
    private final List<ChunkDistanceChecker> chunkCheckers;

    ...
    public DistancePair bestMatch(String target) {
        return chunkCheckers.stream()
            .map(checker -> CompletableFuture.supplyAsync(() -> checker.bestDistance(target)))
            .collect(Collectors.toList())
            .stream()
            .map(future -> future.join())
            .reduce(DistancePair.worstMatch(), (a, b) -> DistancePair.best(a, b));
    }
}
```

The multiple-line statement at the bottom of Listing 5 does all the work using the fluent streams API:

1. `chunkCheckers.stream()` creates a stream from the `List<ChunkDistanceChecker>`.
2. `.map(checker -> ...` applies a mapping to the values in the stream, in this case using the same technique as the Listing 2 example to construct a `CompletableFuture` for the result of an asynchronous execution of the `ChunkDistanceChecker.bestDistance()` method.
3. `.collect(Collectors.toList())` collects the values into a list, which `.stream()` turns back into a stream.
4. `.map(future -> future.join())` waits for the result of each future to be available, and `.reduce(...` finds the best value by repeatedly applying the `DistancePair.best()` method to the prior best result and the latest result.

Admittedly, that's kind of a mess. Before you stop reading, let me assure you that the next variation is cleaner and simpler. The point of Listing 5 is to show how you can use streams as an alternative to normal loops.

The Listing 5 code would be simpler without the multiple conversions, from stream to list and back to stream. In this case, the conversion is needed, because otherwise the code would just wait for the `CompletableFuture.join()` method immediately after the future is created.

Parallel streams

Fortunately, there's an easier way to implement parallel operations on streams than the cumbersome approach in Listing 5. Sequential streams can be made into parallel streams, and parallel streams automatically share work across multiple threads while enabling the results to be collected at a later stage. Listing 6 shows how you can use this approach to find the best match from a `List<ChunkDistanceChecker>`.

Listing 6. Best match using parallel stream of chunks

```
public class ChunkedParallelDistance extends TimingTestBase {
    private final List<ChunkDistanceChecker> chunkCheckers;
    ...
    public DistancePair bestMatch(String target) {
        return chunkCheckers.parallelStream()
            .map(checker -> checker.bestDistance(target))
            .reduce(DistancePair.worstMatch(), (a, b) -> DistancePair.best(a, b));
    }
}
```

Here again, the multiple-line statement at the end does all the work. As in Listing 5, the statement starts by creating a stream from the list, but this version uses the `parallelStream()` method to get a stream that's set up for parallel processing. (You can also convert a regular stream to parallel processing by calling the `parallel()` method on the stream.) The next part, `.map(checker -> checker.bestDistance(target))`, finds the best match within a chunk of known words. The last part, `.reduce(...`, accumulates the best result across all chunks, again as in Listing 5.

Parallel streams execute certain steps, such as `map` and `filter` operations, in parallel. So behind the scenes, the Listing 6 code spreads the map step across multiple threads before consolidating the results in the reduce step (not necessarily in any particular order, because the results are coming from operations being performed in parallel).

The ability to partition the work to be done in a stream relies on the new `java.util.Spliterator<T>` interface used in streams. As you might guess from the name, a `Spliterator` is similar to an `Iterator`. With a `Spliterator`, as with an `Iterator`, you can work with a collection of elements one at a time — though rather than getting the elements from the `Spliterator`, you apply an action to the elements using the `tryAdvance()` or `forEachRemaining()` method. But a `Spliterator` can also supply an estimate of how many elements it holds, and it can potentially be split in two like a cell undergoing mitosis. These added abilities make it easy for the stream parallel-processing code to spread the work to be done across the available threads.

If that Listing 6 code looks somehow familiar to you, that's because it's much like the Scala parallel collections example from the first article of the series:

```
def bestMatch(target: String) =
  matchers.par.map(m => m.bestMatch(target)).
    foldLeft(DistancePair.worstMatch)((a, m) => DistancePair.best(a, m))
```

You can see some differences in both syntax and operation, but in essence the Java 8 parallel stream code is doing the same thing, in the same way, as the Scala parallel collections code.

Streams all the way

So far, all the examples have preserved the chunked structure of the comparison task held over from the first article of the series, which was needed for efficiently handling parallel tasks in older versions of Java. Java 8 parallel streams are designed to handle division of work on their own, so you can pass a set of values to be processed as a stream, and the built-in concurrency handling breaks the set down to spread the work across the available processors.

A couple of problems occur when you try to apply this approach to the edit-distance task. If you chain processing steps together into a *pipeline* (the official term for a sequence of stream operations), you can pass only one result from each step on to the next stage of the pipeline. If you want multiple results (such as the best-distance value and corresponding known word used in the edit-distance task), you must pass them as an object. But creating a object for the result of each individual comparison would hurt the performance of a direct stream approach compared to the chunked approaches. Even worse, the edit-distance calculation reuses a pair of allocated arrays. The arrays can't be shared across parallel calculations, so they would need to be allocated anew for each calculation.

Fortunately, the streams API gives you a way to handle this case efficiently, though it requires a bit more work. Listing 7 demonstrates how you can use a stream to handle the full set of calculations without creating intermediate objects or excess copies of the working arrays.

Listing 7. Stream handling of individual edit-distance comparisons

```
public class NonchunkedParallelDistance extends TimingTestBase
{
    private final String[] knownWords;
    ...
    private static int editDistance(String target, String known, int[] v0, int[] v1) {
    ...
    }

    public DistancePair bestMatch(String target) {
        int size = target.length() + 1;
        Supplier<WordChecker> supplier = () -> new WordChecker(size);
        ObjIntConsumer<WordChecker> accumulator = (t, value) -> t.checkWord(target, knownWords[value]);
        BiConsumer<WordChecker, WordChecker> combiner = (t, u) -> t.merge(u);
        return IntStream.range(0, knownWords.length).parallel()
            .collect(supplier, accumulator, combiner).result();
    }

    private static class WordChecker {
        protected final int[] v0;
        protected final int[] v1;
        protected int bestDistance = Integer.MAX_VALUE;
        protected String bestKnown = null;

        public WordChecker(int length) {
            v0 = new int[length];
            v1 = new int[length];
        }

        protected void checkWord(String target, String known) {
            int distance = editDistance(target, known, v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestKnown = known;
            } else if (bestDistance == distance) {
                bestKnown = null;
            }
        }

        protected void merge(WordChecker other) {
            if (bestDistance > other.bestDistance) {
                bestDistance = other.bestDistance;
                bestKnown = other.bestKnown;
            } else if (bestDistance == other.bestDistance) {
                bestKnown = null;
            }
        }

        protected DistancePair result() {
            return (bestKnown == null) ? new DistancePair(bestDistance) : new
                DistancePair(bestDistance, bestKnown);
        }
    }
}
```

Listing 7 uses a mutable result container class (here the `WordChecker` class) to combine results. The `bestMatch()` method implements the comparison, using three moving parts in the form of lambdas:

- The `Supplier<WordChecker> supplier` lambda supplies instances of the result container.
- The `ObjIntConsumer<WordChecker> accumulator` lambda accumulates a new value into a result container.
- The `BiConsumer<WordChecker, WordChecker> combiner` lambda merges two result containers for a combined value.

After these three lambdas are defined, the final statement of `bestMatch()` creates a parallel stream of `int` values for indexes into the array of known words and feeds the stream to the

`IntStream.collect()` method. The `collect()` method uses the three lambdas to do all the actual work.

Java 8 concurrency performance

Figure 1 shows how measured performance varies with different block sizes when the test code is run on my four-core AMD system using Oracle's Java 8 for 64-bit Linux®. As with the timings in the first article of the series, each input word is compared in turn with 12,564 known words, and each task finds the best match within a range of the known words. The full set of 933 misspelled input words is run repeatedly, with pauses between passes for the JVM to settle. The best time after 10 passes is used in the Figure 1 graph. The final block size of 16,384 is greater than the number of known words, so this case shows single-threaded performance. The implementations included in the timing test are the four main variations in this article along with the best overall variation from the first article:

- CompFuture: `CompletableFutureDistance0` from Listing 2
- CompFutStr: `CompletableFutureStreamDistance` from Listing 5
- ChunkPar: `ChunkedParallelDistance` from Listing 6
- ForkJoin: `ForkJoinDistance` from Listing 3 of the first article
- NchunkPar: `NonchunkedParallelDistance` from Listing 7

Figure 1. Java 8 performance

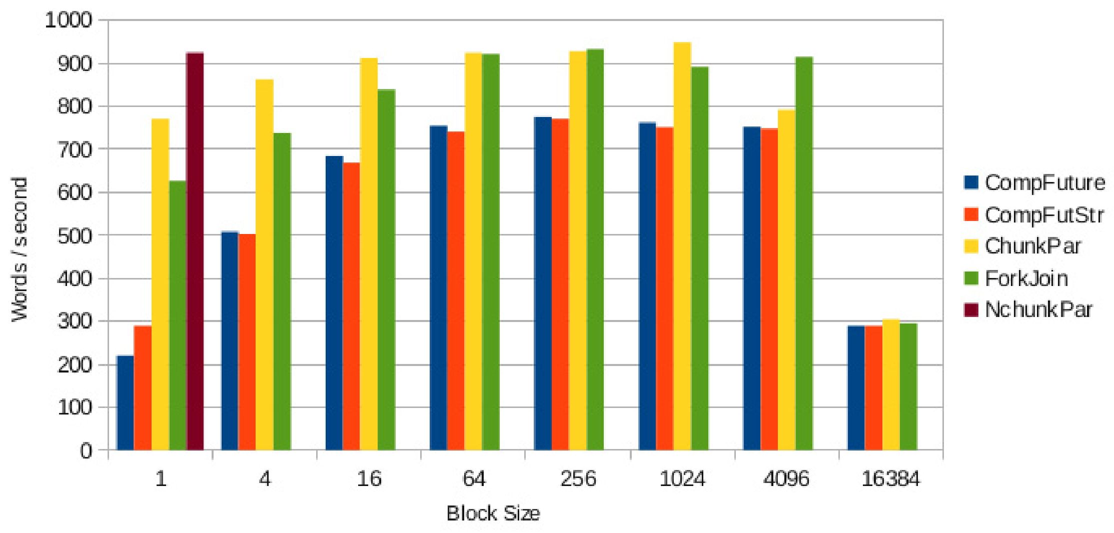

Figure 1 shows impressive results for the new Java 8 parallel streams approach, especially the fully streamified Listing 7 `NchunkPar`. The optimizations used to eliminate object creation show in the timing result (only one value in the chart, because this approach does not use a block size), which matches the best performance of any of the other alternatives. The `CompletableFuture` approaches are a little weak on performance, but that's not unexpected because this example doesn't play to the class's strengths. The Listing 5 `ChunkPar` time is approximately even with the `ForkJoin` code from the first article, though with less block-size sensitivity. All the variations that test chunks of words at a time show poorer performance with small chunk sizes, as you'd expect to see because object-creation overhead is higher relative to the actual computational work.

Like the timing results from the first article, these results are only a general sort of guide to the performance you might see for your own applications. The most important lesson here is that the new Java 8 parallel streams deliver superior performance when used appropriately. Combine good performance with the development benefits of a functional coding style for streams, and you have a winning combination for any time you want to do computations on collections of values.

Java 8 concurrency summary

Java 8 adds some important new features to the developer's toolkit. On the concurrency front, the parallel streams implementation is fast and easy to use, especially when combined with lambda expressions for a functional-ish programming style that clearly and concisely expresses your intent. The new `CompletableFuture` class also helps ease concurrent programming when you're working with individual activities, which the stream model doesn't easily apply to.

The next *JVM concurrency* article will swing over to the Scala side and look into a different and interesting way to handle asynchronous computations. With the `async` macro, you can write code that looks like you're doing sequential blocking operations, but under the covers, Scala translates the code into a completely non-blocking structure. I'll give some examples of how this feature is useful and also take a look at how it's implemented. Who knows — perhaps some of this new work from Scala will make it into Java 9.

Resources

Learn

- Sample code for this article: Get this article's full sample code from the author's repository on GitHub.
- Scalable Scala: Series author Dennis Sosnoski shares insights and behind-the-scenes information on the content in this series and Scala development in general.
- "Java 8 language changes" (Dennis Sosnoski, developerWorks, April 2014): Find out about the Java 8 language changes for lambda expressions and interfaces.
- "Java 8: Definitive guide to CompletableFuture" (Tomasz Nurkiewicz): Get a guided tour of the full `CompletableFuture` API, including a comparison with the equivalents in other JVM languages.
- Aggregate Operations: Learn about using Java 8 streams in this section of the Java Tutorial, including how streams can execute in parallel.
- Streams in Top Gear: View this JavaOne 2013 presentation for an in-depth examination of the architecture of the streams API and its implementation, including insights into how to get the best performance and scalability from the library.
- Concurrency JSR-166 Interest Site: This site, maintained by Doug Lea, is the home of the continuing work on concurrency support in Java that started with JSR-166.
- *Language designer's notebook*: In this developerWorks series, Java Language Architect Brian Goetz explores some of the language design issues that have presented challenges for the evolution of the Java language in Java SE 7, Java SE 8, and beyond.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- IBM SDK, Java Technology Edition Version 8: Participate in the IBM SDK for Java 8.0 Beta Program.
- IBM Java developer kits: Discover the IBM Java SDK and runtime environment for your platform.
- Evaluate IBM products.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Dennis Sosnoski

Dennis Sosnoski is a Java and Scala developer with extensive experience developing scalable systems. He's well-known in the XML and web services areas, where his background includes the development of JiBX XML data binding and work on several open source web services frameworks (most recently, Apache CXF). Dennis is a frequent presenter at Java user groups and conferences and has written many articles for developerWorks, including the popular *Java web services* series. Learn more about his web services training and consulting work at his Sosnoski Software Associates Ltd site, and follow his ongoing explorations of concurrent programming on the JVM at his Scalable Scala site.