JVM の並行性: Java 8 での並行処理の基礎

Java 8 の機能が並行プログラミングを容易にする方法を探る

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2014年 7月 03日

Java 8 には、並行プログラムをはじめとする各種プログラムの作成を容易にする、新しい言語機能と追加クラスが組み込まれています。この記事で、`CompletableFuture` や Stream などの Java 8 の機能拡張によって実現された、この言語の新しい強力な並列処理サポートについて学んでください。これらの新しい機能と、この連載の最初の記事で探った Scala の機能のいくつかの間には、類似点があることに気付くはずです。

このシリーズの他の記事を見る

この連載について

マルチコア・システムが至るところで使われるようになった今、これまで以上に幅広く並行プログラミングを適用しなければならなくなっています。しかし、並行処理を適切に実装するのは難しい場合があり、並行処理を利用するための新しいツールも必要になってきます。このようなツールは、JVM ベースの多くの言語で開発されていますが、なかでも Scala は、並行処理の分野で特に積極的です。この連載では、Java 言語と Scala 言語での新しい並行プログラミング手法をいくつか取り上げて検討します。

長いこと待ち望んでいた Java 8 リリースの主な機能強化には、`java.util.concurrent` 階層に追加されたクラス、強力な新しい並列 Stream 機能など、並行性に関連するものがあります。Stream はラムダ式とともに使用するように意図されており、このラムダ式にしても、日々のプログラミング作業の他の多くの側面を容易にする Java 8 の追加機能です (ラムダ式および関連する interface の変更についての概要は、Java 8 言語で拡張された内容に関する関連記事を参照してください)。

この記事ではまず、新しいクラス CompletableFuture が非同期処理の調整を容易にする仕組みを紹介します。次に、並列 Stream (Java 8 での並行処理にとって大成功の機能) を使用して値のセットに対する複数の処理を並列に実行する方法を説明します。最後に、連載の最初の記事で使用したコードとの比較も盛り込んで、Java 8 の新機能のパフォーマンスを調べます (この記事の完全なサンプル・コードへのリンクについては、「参考文献」を参照してください)。

`Future` の復習

この連載の最初の記事では、Java と Scala 両方の `Future` について簡単に概説しました。(Java 8 より前の) Java における Future のサポートは十分ではなく、サポートしている使用法には 2 つのタ

イプしかありません。それは、Future が完了したかどうかをチェックするか、Future が完了するまで待機することしかできません。一方、Scala における Future のサポートは遥かに柔軟性があり、Future が完了した時点でコールバックを実行することや、異常終了も `Throwable` という形で処理されます。

Java 8 が追加する `CompletableFuture<T>` クラスは、新しい `CompletionStage<T>` インターフェースを実装して `Future<T>` を継承します (このセクションで取り上げる並行処理クラスおよびインターフェースはすべて `java.util.concurrent` パッケージに含まれているものです)。`CompletionStage` は、場合によっては非同期で行われる計算の 1 つの段階またはステップを表します。このインターフェースは、`CompletionStage` インスタンスを他のインスタンスやコード (完了時に呼び出されるメソッドなど) とチェーニングするためのさまざまな手段を定義しています (`Future` インターフェースでのメソッド数は 5 であるのに対し、合計で 59 のメソッドを定義しています)。

リスト 1 に、連載の最初の記事に記載した編集距離の比較コードをベースにした、`ChunkDistanceChecker` クラスを記載します。

リスト 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);
    }
}
```

ChunkDistanceChecker クラスの各インスタンスが、ターゲットの単語を既知の単語からなる配列と照らし合わせて最もよく一致するものを見つけます。静的 buildCheckers() メソッドは、既知の単語からなる配列全体と必要なブロック・サイズから、List<ChunkDistanceChecker> を作成します。この ChunkDistanceChecker クラスは、この記事で最もよく一致する単語の検索を並行処理する各種実装のベースとなります。その最初の実装は、リスト 2 の CompletableFutureDistance0 クラスです。

リスト 2. CompletableFuture を使用した編集距離の計算

```
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;
    }
}
```

ラムダ式について学ぶ

supplyAsync() メソッドに渡すラムダ式は、target パラメーター値を参照しているため、キャプチャー型ラムダ式です。「Java 8 言語での変更内容」で、ラムダ式の概要を読んで、キャプチャー型ラムダ式と非キャプチャー型ラムダ式の違いについて学んでください。

リスト 2 の CompletableFutureDistance0 クラスは、CompletableFuture を並行計算に使用する 1 つの方法を示しています。supplyAsync() メソッドは Supplier<T> インタンス (T 型の値を返すメソッドを持つ関数型インターフェース) を引数に取り、非同期で実行される Supplier のキューイング中に CompletableFuture<T> を返します。上記のコードでは、最初の for ループで supplyAsync() にラムダ式を渡し、ChunkDistanceChecker 配列と一致する Future のリストを作成します。2 番目の for ループは、各 Future が完了するまで待ち (ただし、Future は非同期で実行されるため、このループが到達する前にほとんど完了します)、すべての結果から最もよく一致するものを集めます。

CompletableFuture をベースとした並行処理

連載の最初の記事で説明したように、Scala の Future では、完了ハンドラーをアタッチして、さまざまな方法で Future を結合することができます。CompletableFuture は、Java 8 に同様の柔軟性を提供します。このセクションでは、これらの機能を使用する方法をいくつか取り上げて、編集距離をチェックするコードのコンテキストで説明します。

リスト 3 に示しているのは、リスト 2 の bestMatch() メソッドの別のバージョンです。このバージョンでは、以前のいくつかの並行処理クラスとともに、CompletableFuture による完了ハンドラーを使用しています。

リスト 3. 完了ハンドラーを使用した CompletableFuture

```
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();
}
```

リスト 3 の CountDownLatch は、このコードで作成される Future の数に初期化されます。それぞれの Future を作成するごとに、CompletableFuture.thenAccept() メソッドを使用してハンドラーを (java.util.function.Consumer<T> 関数型インターフェースのラムダ式インスタンスという形で) アタッチします。このハンドラーは、Future が正常に完了すると実行され、検出した最もよく一致する値を AtomicReference.accumulateAndGet() メソッド (Java 8 で追加されたメソッド) によって更新してから、ラッチをデクリメントします。その間、メインの実行スレッドが try-catch ブロックに入り、ラッチがリリースするまで待機します。すべての Future が完了すると、メインのスレッドが処理を続行し、最終的に見つかった最もよく一致する値を返します。

リスト 4 に、リスト 2 の bestMatch() メソッドのさらに別のバリエーションを記載します。

リスト 4. CompletableFuture の結合

```
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);
}
```

上記のコードは、`CompletableFuture.thenCombine ()` メソッドを使って 2 つの Future をマージするために、`java.util.function.BiFunction` (この例の場合、`DistancePair.best()` メソッド) を 2 つの結果に適用し、この関数の結果として Future を返します。

リスト 4 のコードは、最も簡潔でおそらく最も無駄がないコードですが、1 つの欠点があります。それは、各チャンクの処理とその前の処理との組み合わせを表すために、`CompletableFuture` の追加層を作成しているという点です。Java 8 の初期リリースの時点では、この追加層が、`StackOverflowException` を発生させてコード内に紛れ込ませてしまう可能性があります。その場合、決して最後の Future が完了することはありません。現在、このバグに対処するための取り組みが行われているので、近い将来のリリースでは修正されるはずです。

`CompletableFuture` は、この記事の例で使用されているメソッドのさまざまなバリエーションを定義しています。アプリケーションに `CompletableFuture` を使用するときには、完了メソッドと結合メソッドのすべてを網羅したリストを調べて、ニーズに最も適したメソッドを見つけてください。

`CompletableFuture` を使用するのが最適なのは、異なるタイプの処理を実行して、それらの結果を調整しなければならない場合です。同じ計算を多種多様なデータ値に対して実行する場合は、並列 Stream のほうが単純な手法となり、おそらくパフォーマンスにも優れています。従って、編集距離をチェックする例には、並列 Stream 手法のほうが有効です。

Stream

Java 8 の主要な新機能である Stream は、ラムダ式と連動して機能します。Stream とは基本的に、一連の値に対するプッシュ・イテレーターです。Stream をアダプターとチェーニングすることで、Scala のシーケンスと同じようなフィルタリングやマッピングなどの処理を実行することができます。Stream には、直列と並列の両バリエーションがあります。これも同じく、Scala のシーケンスと同様です (ただし、Scala には並列シーケンスに対して個別のクラス階層があるのに対し、Java 8 は内部フラグを使用して直列 Stream であるか、並列 Stream であるかを示します)。Stream には `int`、`long`、`double` のプリミティブ型に応じたバリエーションがある他、型付けされたオブジェクト Stream もあります。

新しく追加された Stream API は、この記事で完全に説明するにはあまりにも複雑なので、並行処理の側面に焦点を絞ります。Stream の詳細な説明については、「参考文献」セクションを参照してください。

リスト 5 に、編集距離を使用して最もよく一致するものを求めるコードの別のバリエーションを記載します。このコードでは、リスト 1 の `ChunkDistanceChecker` を使用して距離を計算し、リスト 2 の例と同じように `CompletableFuture` を使用します。ただし今回は、Stream を使用して最もよく一致する結果を取得します。

リスト 5. Stream を使用した `CompletableFuture`

```
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));
    }
}
```

リスト 5 の最後にある複数行のステートメントは、以下のように、流れるような Stream API を使用してすべての作業を処理します。

1. `chunkCheckers.stream()` は、`List<ChunkDistanceChecker>` から Stream を作成します。
2. `.map(checker -> ...` は、作成された Stream に含まれる値にマッピングを適用します。この例の場合、リスト 2 の例と同じ手法を使用して、`ChunkDistanceChecker.bestDistance()` メソッドを非同期で実行した結果を返すための `CompletableFuture` を作成します。
3. `.collect(Collectors.toList())` は値をリストに収集し、`.stream()` がその値のリストを Stream に返します。
4. `.map(future -> future.join())` は、各 Future の結果が使用可能になるまで待機します。`.reduce(...` は、`DistancePair.best()` メソッドをこれまでの最もよく一致する結果と最新の結果に繰り返し適用することで、最もよく一致する値を見つけます。

正直なところ、この方法は少し面倒ですが、次のバリエーションはこれよりも簡潔で単純になることを保証しますので、まださじを投げないでください。リスト 5 の狙いは、通常のループの代わりとして Stream をどのように使用できるかを示すことにあるからです。

リスト 5 のコードは、Stream からリストへの変換と、リストから Stream への変換を使用しなければもっと単純になりますが、このコードではこれらの変換が必要です。それは、変換を使用しないと、Future が作成された直後にコードが `CompletableFuture.join()` メソッドを待機したままになってしまうためです。

並列 Stream

幸いなことに、リスト 5 の面倒な手法よりも簡単に、Stream に対する並列処理を実装する方法があります。シーケンシャル Stream は並列 Stream にすることができるので、並列 Stream が複数のスレッドの間で自動的に作業を共有し、後の段階で結果が収集されるようにすることができます。リスト 6 に、この手法を使用して、`List<ChunkDistanceChecker>` から最もよく一致するものを見つける方法を示します。

リスト 6. チャンクの並列 Stream を使用して最もよく一致するものを見つける

```
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));
    }
}
```

上記のコードでも、最後の複数行のステートメントがすべての処理を行います。リスト 5 の場合と同じく、このステートメントは最初にリストから Stream を作成します。ただし、このバージョンでは `parallelStream()` メソッドを使用して、並列処理のためにセットアップされた Stream を取得します (Stream に対して `parallel()` メソッドを呼び出すことで、通常の Stream を並列処理に変換することもできます)。次の `.map(checker -> checker.bestDistance(target))` の部分は、既知の単語のチャンク内で最もよく一致するものを探します。そして最後の `.reduce(...` の部分が、リスト 5 と同じく、すべてのチャンクから最もよく一致するものを収集します。

並列 Stream は、`map` 処理や `filter` 処理などの特定のステップを並列に実行します。そのため、リスト 6 のコードは裏で、map ステップを複数のスレッドに分散させた後、それらのスレッドの結果を reduce ステップで集約します (結果の集約は、必ずしも特定の順序で行われる必要はありません。これらの結果は、並行して実行されている複数の処理から生成されるためです)。

Stream 内で実行すべき作業を分割する機能は、Stream 内で使用される新しい `java.util.Spliterator<T>` インターフェースを利用します。名前から想像できるように、`Spliterator` は `Iterator` とよく似ています。`Spliterator` を使用する場合、Iterator を使用する場合と同様に、要素のコレクションを 1 つずつ扱うことができます (ただし、Spliterator から要素を取得するのではなく、`tryAdvance()` メソッドまたは `forEachRemaining()` メソッドを使用して要素にアクションを適用します)。ただし、`Spliterator` は、保持する要素の推定数を提供することが可能です。さらに、場合によっては有糸分裂中の細胞のように 2 つに分裂することもできます。追加されているこれらの機能により、Stream の並列処理コードでは実行すべき作業を、使用可能なスレッド間に簡単に分散させることができます。

リスト 6 のコードにどことなく馴染みがあるとしたら、それはこのコードが、連載の最初の記事で使用した Scala の並列コレクションの例 (以下を参照) とそっくりだからです。

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

構文にも処理にも異なる点はいくつかありますが、Java 8 の並列 Stream コードは基本的に、Scala の並列コレクション・コードと同じことを同じ方法で行っています。

Stream をフルに使用する

ここまでのすべての例では、連載の最初の記事から引き継いだ、比較タスクをチャンク化した構造を維持してきました。この構造は、古いバージョンの Java で効率的に並列タスクを処理するために必要なものでした。Java 8 の並列 Stream は、それ自体が作業の分割に対処するように設計さ

れています。従って、Stream として処理する値のセットを渡すと、組み込み並行処理でそのセットを分割して、使用可能なプロセッサー間に作業を分散させることができます。

この手法を編集距離タスクに適用する場合、いくつかの問題が発生します。処理ステップをチェーニングして 1 つのパイプライン (一連の Stream 処理の正式な用語) にすると、各ステップから 1 つの結果だけをパイプラインの次のステージに渡すことができます。複数の結果 (例えば、編集距離タスクで使用している最短距離の値と、対応する既知の単語など) が必要な場合には、それらの結果をオブジェクトとして渡さなければなりません。しかし、個々の比較ごとの結果としてオブジェクトを作成すると、チャンク化の手法に比べ、直接 Stream を使用する手法のほうが、パフォーマンスが劣ることになるはずです。さらに悪いことに、編集距離の計算は、割り当てられた配列のペアを再利用します。配列を並列計算で共有することはできないため、計算ごとに改めて配列を割り当てなければなりません。

幸い、Stream API には、このような事態に効率的に対処する方法がありますが、その方法を実行するにはもう少し作業が必要です。リスト 7 に、作業配列の余分なコピーや中間オブジェクトを作成することなく、すべての計算に対処するために、どのように Stream を使用できるかを示します。

リスト 7. 個々の編集距離の比較を処理する Stream

```
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);
        }
    }
}
```

リスト 7 は、可変の結果コンテナー・クラス (ここでは `WordChecker` クラス) を使用して結果を結合します。`bestMatch()` メソッドは比較を実装するために、以下のように、3 つの可変の構成要素をラムダ式という形で使用します。

- `Supplier<WordChecker> supplier` ラムダ式: 結果コンテナーのインスタンスを提供します。
- `ObjIntConsumer<WordChecker> accumulator` ラムダ式: 新しい値を結果コンテナーに収集します。
- `BiConsumer<WordChecker, WordChecker> combiner` ラムダ式: 2 つの結果コンテナーを 1 つの結合された値としてマージします。

これらのラムダ式が定義された後、`bestMatch()` の最後のステートメントによって、インデックスに対して `int` 値の並列 Stream が作成されて既知の単語の配列にされ、その Stream が `IntStream.collect()` メソッドにフィードされます。すると、`collect()` メソッドが 3 つのラムダ式を使用して、実際のすべての作業を行います。

Java 8 の並行処理のパフォーマンス

図 1 に、4 コア AMD システム上で Oracle の Java 8 (64 ビット Linux 版) を使用してテスト・コードを実行したときに、ブロック・サイズを変えると測定パフォーマンスがどのように変化するかを示します。連載の最初の記事でタイミング・テスト (パフォーマンス評価のために実行時間を測定するテスト) を行ったときと同じく、このテストでは、入力された単語を順に 12,564 個の既知の単語と比較し、タスクごとに、その既知の単語の範囲内で最もよく一致するものを見つけます。スペルに誤りがある 933 個の単語セットの入力を繰り返し実行し、JVM を安定させるためにパスとパスの間では一時停止します。図 1 のグラフでは、10 回のパスの後での最短時間が使用されています。16,384 という最終ブロック・サイズは、既知の単語の数より大きい値であるため、このケースではシングル・スレッドのパフォーマンスを表しています。タイミング・テストに含まれる実装は、この記事に記載した 4 つの主要なバリエーションと、最初の記事で全体としての結果が最も良かったバリエーションを使用しました。具体的には、以下の実装です。

- CompFuture: `CompletableFutureDistance0` (リスト 2)
- CompFutStr: `CompletableFutureStreamDistance` (リスト 5)
- ChunkPar: `ChunkedParallelDistance` (リスト 6)
- ForkJoin: `ForkJoinDistance` (最初の記事のリスト 3)
- NchunkPar: `NonchunkedParallelDistance` (リスト 7)

図 1. Java 8 のパフォーマンス

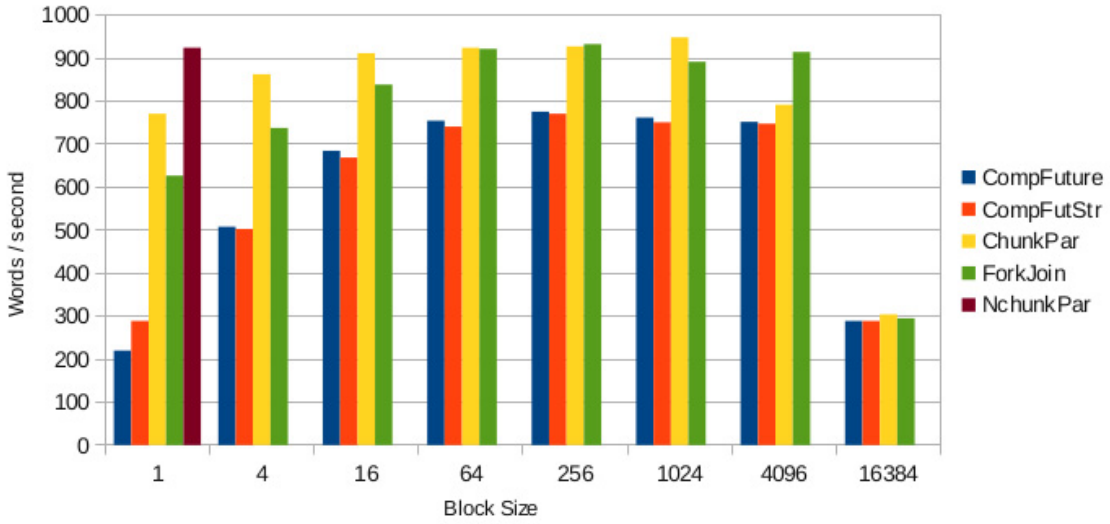

図 1 には、新しい Java 8 の並列 Stream 手法による見事な結果が示されています。特に、フルに Stream を使用するように作られたリスト 7 の `NchunkPar` のパフォーマンスは際立って優れています。この、オブジェクトの作成をなくすために用いられた最適化の効果は、タイミング・テストに示されており (この手法ではブロック・サイズを使用しないため、このグラフ内にはただ 1 つの値しかありません)、他のどの手法の最高パフォーマンスにも匹敵します。`CompletableFuture` の手法のパフォーマンスはやや劣りますが、この例ではこのクラスの長所が発揮されていないため、これは意外な結果ではありません。リスト 6 の `ChunkPar` のタイム (つまりパフォーマンス) は、最初の記事の `ForkJoin` コードとほぼ同じです。単語のチャンクを一度にテストするすべてのバリエーションは、チャンクのサイズが小さい場合、パフォーマンスの低下を示しています。これは、実際の計算作業の割にはオブジェクト作成のオーバーヘッドが大きいことから、当然予測できることです。

最初の記事のタイミング・テストの結果と同じように、これらの結果は、独自に作成するアプリケーションに期待できるパフォーマンスのおおまかなガイドに過ぎません。このテストでの最も重要な教訓は、新しい Java 8 の並列 Stream を適切に使用すれば、卓越したパフォーマンスを実現できることです。優れたパフォーマンスを、関数型のコーディング・スタイルで Stream を開発するメリットと組み合わせれば、値のコレクションに対する計算が必要なときには常に最強の組み合わせになります。

Java 8 の並行性のまとめ

Java 8 は、いくつかの重要な新機能を開発者のツールキットに加えます。並行処理の面では、並列 Stream の実装は、簡単かつすぐに使用することができます。目的を明確かつ簡潔に表現する、関数型の性質を持つプログラミング・スタイルのラムダ式と組み合わせた場合は、なおのことです。また、Stream モデルを簡単に適用できない個々のアクティビティーを扱う場合には、新しい `CompletableFuture` クラスも容易な並行プログラミングを支援します。

「JVM の並行性」の次回の記事では、話題を Scala に切り替えて、非同期計算を扱う別の興味深い方法について見ていきます。`async` マクロを使用すると、シーケンシャル・ブロッキング処理を

実行するようなコードを作成することができますが、内部では Scala がそのコードを完全なノンブロッキング構造へと変容させています。そこで、この機能がいかに役立つかを示す例をいくつか紹介するとともに、その実装方法についても見ていきます。場合によっては、Scala のこの新しい取り組みが部分的に Java 9 に取り込まれる可能性もあります。

著者について

Dennis Sosnoski

Dennis Sosnoski は、スケーラブルなシステムの開発経験が豊富にある、Java および Scala の開発者です。XML と Web サービスの分野で有名な彼のバックグラウンドとしては、JiBX XML データ・バインディングの開発や、いくつかのオープンソース Web サービス・フレームワーク (一番最近のものでは Apache CXF) に関する取り組みなどがあります。Dennis は Java ユーザー・グループや Java カンファレンスで頻繁にプレゼンターを務めており、人気のある連載「Java Web サービス」をはじめとし、developerWorks の数多くの記事を執筆しています。彼が行っている Web サービスのトレーニングと、コンサルティング作業について Sosnoski Software Associates Ltd サイトで詳しい情報を得てください。また、彼が現在行っている JVM に関する並行プログラミングの探求を Scalable Scala サイトでチェックして読んでください。