JVM の並行性: ブロックすべきか、すべきでないか？

Java 8 での非同期イベント処理に対するブロッキング手法とノンブロッキング手法を比較する

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2014年 10月 02日

Java 8 で追加された CompletableFuture クラスには、非同期処理の完了に対処する新しい方法が用意されており、その中にはノンブロッキング手法で複数のイベントを作成して結合するやり方も含まれています。この記事では、イベントの完了に対処する際にブロッキング手法を使用する場合とノンブロッキング手法を使用する場合の違いを、読者が理解する助けとなるように説明し、さらにはノンブロッキング手法を優先するべき理由についても説明します。

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。

あらゆる並行処理アプリケーションにおいて、非同期イベント処理は極めて重要です。イベントのソースは個々の計算タスクである場合もあれば、I/O 処理、あるいは外部システムとのやりとりがソースである場合もあります。ソースが何であれ、アプリケーション・コードはイベントを追跡するとともに、それらのイベントに応じて行われるアクションを調整しなければなりません。Java アプリケーションには、非同期イベント処理に利用できる基本手法が 2 種類あります。1 つは、調整スレッドにイベントを待機させ、イベントが発生したら調整スレッドがアクションを取るという手法で、もう 1 つは、イベントの完了時に、直接イベントがアクションを実行するという手法 (通常は、アプリケーションによって提供されたコードを実行するという形をとります) です。スレッドにイベントを待機させる手法は、「ブロッキング」手法と呼ばれます。スレッドが明示的にイベントを待機することなく、イベントにアクションを実行させる手法は、「ノンブロッキング手法」と呼ばれます。

これまで使われてきた `java.util.concurrent.Future` クラスで提供している方法は、イベントの完了を予測して処理する単純な方法ですが、その手段となるのはイベント

の完了をポーリングするか、待機するかのいずれかのみでした。Java 8 で追加された `java.util.concurrent.CompletableFuture` クラスは、この処理機能を拡張し、イベントを作成または処理するさまざまなメソッドを提供しています (`CompletableFuture` の概要については、連載の以前の記事「Java 8 での並行処理の基礎」を読んでください)。特に、`CompletableFuture` はイベントの完了時にアプリケーション・コードを実行する標準的な手法となり、さまざまな方法で (future によって表現された) 複数のタスクを結合することもできます。この結合手法により、イベントを処理するためのノンブロッキング・コードを簡単に (少なくとも、これまでよりは簡単に) 作成することができます。この記事では、`CompletableFuture` を使用して、イベント処理にブロッキング手法とノンブロッキング手法のそれぞれを適用する方法を紹介します。その上で、ノンブロッキング手法には手間をかけるだけの価値がある理由をいくつか示します (著者の GitHub リポジトリーから完全なサンプル・コードを入手してください)。

ブロッキングとノンブロッキング

コンピューティングにおける「ブロッキング」と「ノンブロッキング」という用語は、コンテキストに応じて多少異なる使われ方をするケースがよくあります。例えば、共有データ構造に対するノンブロッキング・アルゴリズムでは、スレッドは、そのデータ構造へのアクセスを待機する必要がありません。ノンブロッキング I/O においては、アプリケーションのスレッドは I/O 処理を開始した後、その処理が非同期で行われている間、その処理から抜けて他の処理を行うことができます。この記事でのノンブロッキングは、アクションの実行を伴うイベントの完了に、待機スレッドが必要ないことを意味します。これらのコンテキストでの用語の使い方に共通する概念は、ブロッキング処理では何かを待機するスレッドが必要であり、ノンブロッキング処理ではその必要がないということです。

複数のイベントの作成

完了を待機するのは、単純なことです。つまり、イベントの発生時にスレッドを待機させ、スレッドが再開されると、そのイベントが完了していることがわかる、というものです。スレッドに他にも実行しなければならない処理があるとしたら、スレッドはイベントの完了を待機している間にその処理を実行することもできます。スレッドはポーリング手法を使用することで、そのスレッド自身が行っている他のアクティビティーを中断して、イベントが完了したかどうかをチェックすることすらできますが、どちらも基本原則は同じです。つまり、イベントの結果が必要になった時点でスレッドを停止するので、スレッドはイベントの完了を待機することになります。

1 つのメイン・スレッドだけがイベントの完了を待機する限り、ブロッキングは簡単かつ比較的確実に行うことができます。しかし、複数のスレッドが互いの完了を待ってブロッキング状態になる場合、次のような問題が発生する可能性があります。

- デッドロック: 複数のスレッドがそれぞれに、他のスレッドが処理を進める上で必要なリソースをコントロールしている状態。
- 枯渇: 他のスレッドによって共有リソースが使い果たされていることから、一部のスレッドが処理を進めることができない状態。
- ライブロック: 複数のスレッドが互いに調整を試みているものの、結局は処理を進めることができない状態。

ノンブロッキング手法には、ブロッキング手法よりも遥かに創意工夫の余地が残されています。ノンブロッキング・イベント処理でよく使われる手法の 1 つとしては、コールバックがあります。コールバックを使用することで、必要なあらゆるコードをイベントの発生時に実行できるこ

とから、コールバックはまさに柔軟性の典型と言えます。一方、多数のイベントをコールバックによって処理すると、コードが煩雑になるという欠点があります。また、コールバックを使用したアプリケーションでは、コードの順序が制御フローと一致しないため、デバッグ作業が極めて煩雑になる場合もあります。

Java 8 の `CompletableFutures` は、通常のコールバックを含め、ブロッキング手法とノンブロッキング手法両方のイベント処理をサポートしています。また、`CompletableFuture` には、簡潔かつ単純で理解しやすいコードによってコールバックの柔軟性を得られるように、イベントを作成して結合する方法も用意されています。このセクションでは、`CompletableFuture` で表現されたイベントを、ブロッキング手法とノンブロッキング手法のそれぞれで処理する場合のサンプル・コードについて見ていきます。

タスクと順序付け

アプリケーションでは、ある特定のオペレーションの中で、複数の処理ステップを実行しなければならないことがよくあります。例えば Web アプリケーションで、ユーザーにレスポンスを返す前に、以下の処理を実行しなければならないとします。

1. データベースでユーザーの情報を検索する
2. 検索した情報を使用して、Web サービスの呼び出し、さらに場合によっては別のデータベース・クエリーを実行する
3. 前のステップからの結果に基づいて、データベースを更新する

図 1 に、上記のようなタスクの構造を示します。

図 1. アプリケーション・タスクのフロー

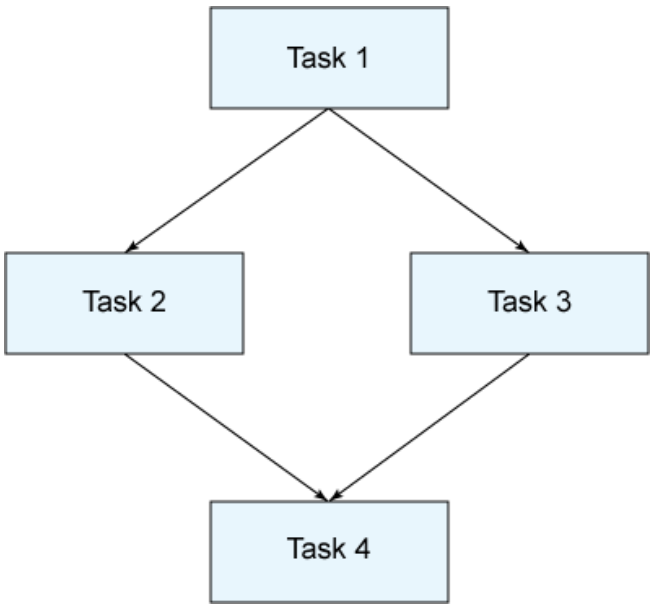

図 1 では、処理全体を 4 つの個別のタスクに分割し、タスク間を矢印で結んで順序の依存性を表しています。タスク 1 は直接実行することができます。タスク 2 とタスク 3 は両方ともタスク 1 が完了した後に実行されます。タスク 4 は、タスク 2 とタスク 3 の両方が完了した後に実行されます。記事ではこのタスク構造を使用して、非同期イベント処理の説明をします。実際のアプリ

ケーション (特に、多数の可変の構成要素があるサーバー・アプリケーション) は、これよりも遥かに複雑になるはずですが、関連する原則を説明するには、この単純な例が役立ちます。

非同期イベントのモデル化

実際のシステムでは、非同期イベントのソースは、一般に、並列でのコンピューター処理または何らかの形の I/O 処理のいずれかとなりますが、上記の例のようなシステムをモデル化するには、単純な時間遅延を使用したほうが簡単なので、この記事では時間遅延の手法を使用します。リスト 1 に、`CompletableFuture` の形でイベントを生成するために使用する、基本的な時刻指定イベントのコードを記載します。

リスト 1. 時刻指定イベント・コード

```
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.CompletableFuture;

public class TimedEventSupport {
    private static final Timer timer = new Timer();

    /**
     * Build a future to return the value after a delay.
     *
     * @param delay
     * @param value
     * @return future
     */
    public static <T> CompletableFuture<T> delayedSuccess(int delay, T value) {
        CompletableFuture<T> future = new CompletableFuture<T>();
        TimerTask task = new TimerTask() {
            public void run() {
                future.complete(value);
            }
        };
        timer.schedule(task, delay * 1000);
        return future;
    }

    /**
     * Build a future to return a throwable after a delay.
     *
     * @param delay
     * @param t
     * @return future
     */
    public static <T> CompletableFuture<T> delayedFailure(int delay, Throwable t) {
        CompletableFuture<T> future = new CompletableFuture<T>();
        TimerTask task = new TimerTask() {
            public void run() {
                future.completeExceptionally(t);
            }
        };
        timer.schedule(task, delay * 1000);
        return future;
    }
}
```

ラムダ式を使用しない理由

リスト 1 の `TimerTask` は、単一の `run()` メソッドを持つ匿名内部クラスとして実装されています。ここは、内部クラスに代えてラムダ式を使用するのに最適なところだと思うかもしれませんが、ラムダ式はインターフェースのインスタンスとして使用することしかできません。`TimerTask` は抽象クラスとして定義されています。ラムダ式の機能が将来拡張され

て、抽象クラスのサポートが追加されるか (これは可能ですが、設計の問題があるため、その見込みはあまりありません)、または TimerTask のようなケースに並列インターフェースが定義されない限り、単一メソッドの実装には引き続き Java 内部クラスを使用する必要があります。

リスト 1 のコードは、`java.util.Timer` を使用して、遅延後の `java.util.TimerTask` の実行をスケジューリングします。各 `TimerTask` は実行時に、それぞれに関連付けられた future を完了します。`delayedSuccess()` は、実行されて future を呼び出し側に返すときに `CompletableFuture<T>` が正常に完了するように、タスクをスケジューリングします。`delayedFailure()` は、実行されて future を呼び出し側に返すときに、`CompletableFuture<T>` が例外の発生によって完了するように、タスクをスケジューリングします。

リスト 2 に、リスト 1 のコードを使用して、図 1 の 4 つのタスクに対応するイベントを `CompletableFuture<Integer>` の形で作成する方法を示します (このコードは、サンプル・コードの `EventComposition` クラスから抜粋したものです)。

リスト 2. サンプル・タスクに対応するイベント

```
// task definitions
private static CompletableFuture<Integer> task1(int input) {
    return TimedEventSupport.delayedSuccess(1, input + 1);
}
private static CompletableFuture<Integer> task2(int input) {
    return TimedEventSupport.delayedSuccess(2, input + 2);
}
private static CompletableFuture<Integer> task3(int input) {
    return TimedEventSupport.delayedSuccess(3, input + 3);
}
private static CompletableFuture<Integer> task4(int input) {
    return TimedEventSupport.delayedSuccess(1, input + 4);
}
```

リスト 2 の 4 つのタスク・メソッドのそれぞれは、特定の遅延値を用いて、そのタスクが完了するときに対応します。遅延値としては、`task1` には 1 秒、`task2` には 2 秒、`task3` には 3 秒、そして `task4` には再び 1 秒が使用されます。また、それぞれのタスク・メソッドは入力値を取り、その入力値にタスク番号をプラスしたものを future の (最終的な) 結果値として使用します。これらのメソッドはすべて future の成功した形式を使用しています。失敗した形式を使用する例は、後で記載します。

これらのタスクで意図しているのは、図 1 に示した順序でタスクを実行し、各タスクに先行するタスクから返された結果値 (または `task4` の場合、2 つの先行するタスクからの結果値の合計) を渡すことです。2 つの中間タスクが同時に実行された場合、実行時間の合計は約 5 秒となります (1 秒 + max(2 秒, 3 秒) + 1 秒)。`task1` への入力値が 1 の場合、結果値は 2 です。その結果値が `task2` と `task3` に渡されると、その結果値は 4 と 5 となります。この 2 つの結果値の合計 (9) が `task4` に入力値として渡されると、最終的な結果値は 13 となります。

ブロッキング待機

これで準備はできたので、早速実行してみましょう。4 つのタスクの実行を調整する最も簡単な方法は、ブロッキング待機を使用することです。つまり、メイン・スレッドに、4 つのタスクのそれぞれが完了するのを待機させます。リスト 3 (同じくサンプル・コードの `EventComposition` クラスからの抜粋) に、この手法を示します。

リスト 3. タスクのブロッキング待機

```
private static CompletableFuture<Integer> runBlocking() {
    Integer i1 = task1(1).join();
    CompletableFuture<Integer> future2 = task2(i1);
    CompletableFuture<Integer> future3 = task3(i1);
    Integer result = task4(future2.join() + future3.join()).join();
    return CompletableFuture.completedFuture(result);
}
```

リスト 3 でブロッキング待機を行うために使用しているのは、`CompletableFuture` の `join()` メソッドです。`join()` はタスクの完了を待機して、完了に成功した場合は結果値を返し、完了に失敗した場合または完了がキャンセルされた場合には、非チェック例外をスローします。上記のコードは、まず、`task1` の結果値を待機します。結果値が返されると、`task2` と `task3` を開始し、この両方のタスクから future が返されるのを待機します。そして最後に、`task4` の結果値を待機します。`runBlocking()` が返す `CompletableFuture` は、次に紹介するノンブロッキング形式と同じですが、この例の場合、実際にはメソッドがリターンする前に、future は完了するはずです。

future を構成して結合する

リスト 4 (同じくサンプル・コードの `EventComposition` クラスからの抜粋) に、複数の future を互いに関連付けて、タスクを正しい順序と正しい依存性で一切ブロッキングすることなく、実行する方法を示します。

リスト 4. ノンブロッキングの構成と結合

```
private static CompletableFuture<Integer> runNonblocking() {
    return task1(1).thenCompose(i1 -> ((CompletableFuture<Integer>)task2(i1)
        .thenCombine(task3(i1), (i2,i3) -> i2+i3)))
        .thenCompose(i4 -> task4(i4));
}
```

リスト 4 のコードは基本的に実行計画を作成しており、異なる複数のタスクを実行する方法と、それらのタスクが互いにどのように関係するかを、この実行計画で指定しています。このコードは簡潔にまとめられていますが、`CompletableFuture` メソッドを使い慣れていなければ、もしかすると理解するのは難しいかもしれません。リスト 5 に、同じコードをよりわかりやすい形にリファクタリングしたバージョンを記載します。このコードでは、`task2` と `task3` の部分を切り出して、`runTask2and3` という新しいメソッドにしています。

リスト 5. リファクタリング後のノンブロッキングの構成と結合

```
private static CompletableFuture<Integer> runTask2and3(Integer i1) {
    CompletableFuture<Integer> task2 = task2(i1);
    CompletableFuture<Integer> task3 = task3(i1);
    BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;
    return task2.thenCombine(task3, sum);
}

private static CompletableFuture<Integer> runNonblockingAlt() {
    CompletableFuture<Integer> task1 = task1(1);
    CompletableFuture<Integer> comp123 = task1.thenCompose(EventComposition::runTask2and3);
    return comp123.thenCompose(EventComposition::task4);    }
```

リスト 5 で、`runTask2and3()` メソッドが表すのは、タスク・フローの中間部分です。ここで、`task2` と `task3` が同時に実行された後、それぞれの結果値が結合されます。このシーケンス

をコーディングするために、future では `thenCombine()` メソッドを使用しています。このメソッドは 1 番目のパラメーターとして別の future を取り、2 番目のパラメーターとして (入力値の型が future の結果値の型と同じ) バイナリー関数インスタンスを取ります。`thenCombine()` が返すのは、元の 2 つの future の結果値に関数を適用した値を表す、3 つ目の future です。この例の場合、2 つの future は `task2` と `task3` であり、関数は結果値の合計に適用されます。

`runNonblockingAlt()` メソッドは、future の `thenCompose()` メソッド呼び出しのペアを使用します。`thenCompose()` のパラメーターは、元の future の値の型を入力として取り、別の future を出力として返す、関数インスタンスです。`thenCompose()` の結果値は、関数と同じ結果値の型を持つ 3 つ目の future です。この 3 つ目の future は、元の future が完了した後に関数から返される最終的な future のプレースホルダーとしての役割を果たします。

`task1.thenCompose()` の呼び出しによって、`task1` の結果値に `runTask2and3()` 関数を適用した結果値の future が返され、`comp123` として保存されます。さらに、`comp123.thenCompose()` の呼び出しによって、最初の `thenCompose()` からの結果値に `task4()` 関数を適用した結果値の future が返されます。これが、すべてのタスクを実行した全体的な結果です。

サンプル・コードを試してみる

サンプル・コードに含まれている `main()` メソッドは、イベント・コードのそれぞれのバージョンを順に実行して、完了するまでにかかった時間 (約 5 秒) と結果値 (13) が正しいことを示します。リスト 6 に、この `main()` メソッドをコンソールから実行したときの結果を記載します。

リスト 6. `main()` メソッドの実行

```
dennis@linux-guk3:~/devworks/scala3/code/bin> java com.sosnoski.concur.article3.EventComposition
Starting runBlocking
runBlocking returned 13 in 5008 ms.
Starting runNonblocking
runNonblocking returned 13 in 5002 ms.
Starting runNonblockingAlt
runNonblockingAlt returned 13 in 5001 ms.
```

アンハッピー・パス

ここまでは、常にタスクの完了に成功する future という形でイベントを調整するコードを見てきました。実際のアプリケーションでは、常にこのハッピー・パスにとどまることを当てにすることはできません。タスクの処理中に問題が発生した場合、これらの問題は、Java の用語で言う、`Throwable` で表現されるのが通常です。

リスト 2 のタスク定義を変更して、`delayedSuccess()` メソッドの代わりに `delayedFailure()` メソッドを使用するようにするのは簡単です。以下に、このように変更した `task4` のタスク定義を記載します。

```
private static CompletableFuture<Integer> task4(int input) {
    return TimedEventSupport.delayedFailure(1, new IllegalArgumentException("This won't work!"));
}
```

`task4` だけを例外で完了するように変更してリスト 3 のコードを実行すると、`task4` に対する `join()` 呼び出しによって当然、`IllegalArgumentException` がスローされます。この問題が

runBlocking() メソッドでキャッチされない場合、呼び出しチェーンで例外が継承されます。最後までキャッチされない場合は、スレッドの実行が強制終了されることになります。幸い、タスクのいずれかが例外で完了した場合に、その例外が呼び出し側に伝えられ、返された future を通じて呼び出し側で処理されるように、コードを変更するのは簡単です。リスト 7 に、この変更を行った後のコードを記載します。

リスト 7. 例外を考慮したブロッキング待機

```
private static CompletableFuture<Integer> runBlocking() {
    try {
        Integer i1 = task1(1).join();
        CompletableFuture<Integer> future2 = task2(i1);
        CompletableFuture<Integer> future3 = task3(i1);
        Integer result = task4(future2.join() + future3.join()).join();
        return CompletableFuture.completedFuture(result);
    } catch (CompletionException e) {
        CompletableFuture<Integer> result = new CompletableFuture<Integer>();
        result.completeExceptionally(e.getCause());
        return result;
    }}
```

リスト 7 の内容は、ほとんど自明です。元のコードは try/catch でラップされていて、catch は、完了して返される future として例外を返してきます。この手法はコードを多少複雑にするものの、それでも Java 開発者であれば簡単にコードを理解できるはずです。

リスト 4 のノンブロッキング・コードには、try/catch を追加する必要さえありません。CompletableFuture の構成および結合処理が、例外を処理して自動的に渡すため、従属 future も同じく例外で完了することになります。

ブロックすべきか、すべきでないか

これまでに、CompletableFuture で表現されたイベントを、ブロッキング手法で処理するサンプル・コード、ノンブロッキング手法で処理するサンプル・コードの両方を見てきました。少なくともこの記事でモデル化した基本的なタスク・フローには、どちらの手法を適用するとしても、かなり単純なコードになりますが、タスク・フローが複雑になってくると、コードも同じく複雑になってきます。

ブロッキング手法の場合、イベントの完了を待機するだけであれば、複雑さが増すという問題はそれほど大きな問題にはなりません。ただし、それ以外のタイプの同期をスレッド間で行うとなると、スレッドの枯渇や、さらにはデッドロックという問題に突き当たります。

ノンブロッキング手法の場合、イベントの完了によってトリガーされるコードの実行をデバッグするのは、簡単なことではありません。さまざまなタイプのイベントが発生し、それらのイベント間で多数の相互作用が行われるとしたら、どのイベントがどの実行をトリガーしているのかを把握するのが難しくなります。従来のコールバックを使用しているか、CompletableFuture の構成および結合処理を使用しているかに関わらず、このような状況は基本的に、コールバック地獄の一例です。

結局、単純さの点でメリットがあるのは、一般にブロッキング手法のコードのほうですが、それにも関わらず、なぜノンブロッキング手法のほうを使用したいと思う人がいるのでしょう？このセクションでは、いくつかの重要な理由を説明します。

切り替えのコスト

スレッドがブロックされると、それまでそのスレッドを実行していたプロセッサー・コアは、別のスレッドの実行に移ります。この場合、前に実行されていたスレッドの実行状態をメモリーに保存して、新しいスレッドの状態をロードする必要が生じます。このように、あるスレッドを実行しているコアを、別のスレッドを実行するように切り替える操作は、コンテキスト・スイッチと呼ばれます。

コンテキスト・スイッチによる直接的なパフォーマンス・コストに加え、新しいスレッドは、通常、前のスレッドとは異なるデータを使用します。メモリー・アクセスの速度は、プロセッサー・クロックを大幅に下回るため、最近のシステムでは、プロセッサー・コアとメイン・メモリーの間に、キャッシュで構成される複数の層を採用しています。キャッシュのほうがメイン・メモリーより遥かに高速であるとは言え、キャッシュはメイン・メモリーよりも遥かに容量は小さくなります (一般に、高速なキャッシュであるほど、容量は小さくなります)。したがって、一度にキャッシュに入れられるのは、メモリー全体のほんのわずかな部分でしかありません。このことから、スレッドの切り替えが生じて、コアが新しいスレッドの実行を開始する時点では、まだ、新しいスレッドが必要とするメモリー・データがキャッシュ内に存在しない可能性があります。その場合、必要なデータがメイン・メモリーからロードされるまで、コアが待機しなければなりません。

コンテキスト・スイッチとメモリー・アクセスの遅延が重なると、それはそのまま顕著なパフォーマンス・コストを招きます。図 2 に、4 コア AMD システム上で Oracle の 64 ビット Linux 用の Java 8 を使用してスレッド切り替えを行ったときのオーバーヘッドを示します。このテストでは、スレッドの数を 1 から 4,096 まで 2 の累乗で増やしていき、スレッドあたりのメモリー・ブロックのサイズを 0 KB から 64 KB の範囲の 4 種類のサイズに設定して行っています。スレッドは、実行をトリガーするために `CompletableFuture` を使用して順に実行されます。スレッドは実行されるたびに、まず、スレッドごとのデータを使用して単純な計算を実行し (データをキャッシュにロードする際のオーバーヘッドを示すため)、その後、スレッド間で共有する静的変数をインクリメントします。最後に、次回の実行をトリガーするための新規 `CompletableFuture` インスタンスを作成した後、次のスレッドを起動するために、そのスレッドが待機している `CompletableFuture` を完了させます。スレッドの再実行が必要な場合、そのスレッドは新しく作成された `CompletableFuture` が完了するまで待機します。

図 2. スレッド切り替えのコスト

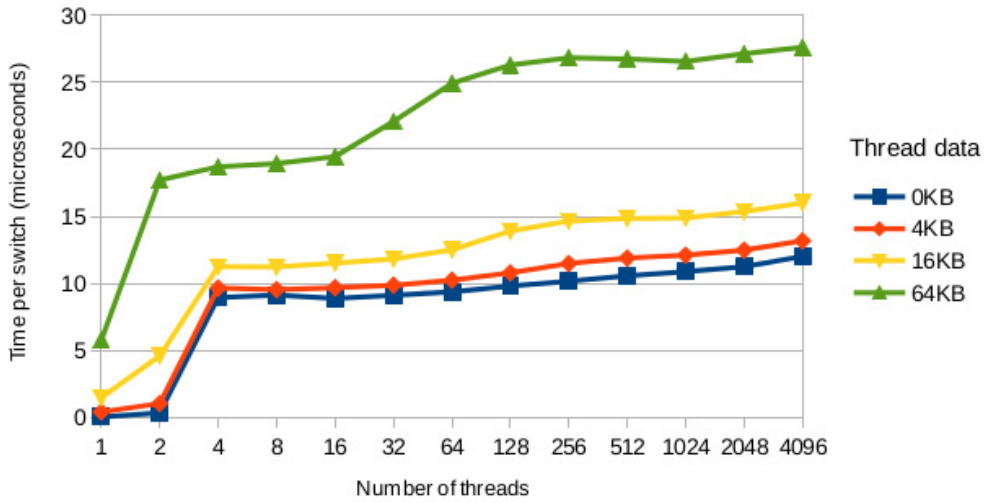

図 2 のグラフから、スレッドの数とスレッドあたりのメモリー容量の両方が与える影響がわかります。スレッドあたりのデータがかなり少量である限りは、2 つのスレッドが動作する速度は単一のスレッドが動作する速度とほとんど変わらないため、スレッドの数が 4 になるまでは、スレッドの数が大きな影響を与えます。スレッドの数が 4 を超えた後は、スレッドの数が増えたときのパフォーマンスへの影響は比較的小さくなります。スレッドあたりのメモリー容量については、その量が大きければ大きいほど、短時間でキャッシュの 2 つの層がいっぱいになってオーバーフロー状態になるため、切り替えのコストが急増します。

図 2 に示されている時間の値は、いくぶん時代遅れのメイン・システムで計測されたものです。皆さんのシステムで計測した時間はこれとは異なり、遥かに小さい値になると思いますが、グラフの曲線は、ほぼ同じような形になるはずです。

図 2 のスレッド切り替えのオーバーヘッドはマイクロ秒で示されているため、スレッド切り替えのコストが数万プロセッサー・クロックになるとしても、絶対数は大きくありません。スレッドあたりのメモリー容量が 16 KB で、スレッドの数が中程度の場合のデータ (グラフの黄色の線) に相当する 12.5 マイクロ秒の切り替え時間では、システムはスレッドの切り替えを 1 秒あたり 80,000 回実行できることになります。これは、合理的に作成されたシングル・ユーザー・アプリケーションで目にするようなスレッド切り替えの数や、サーバー・アプリケーションの多くにおいてさえ目にする可能性があるスレッド切り替えの数を遥かに超える数です。ただし、毎秒何千ものイベントを扱うハイパフォーマンス・サーバー・アプリケーションの場合、ブロッキングのオーバーヘッドがパフォーマンスに大きな影響を及ぼす要因になる可能性があります。そのようなアプリケーションでは、ノンブロッキング手法のコードを可能な限り使用して、スレッド切り替えを最小限に抑えることが重要です。

また、上記の時間の値は、最良の場合のシナリオで計測されていることを理解しておくことも重要です。スレッド切り替えプログラムの実行中には、(少なくとも私のシステムでは) すべてのコアの動作がフル・クロック・スピードで維持されるだけの十分な CPU アクティビティーが実行されます。実際のアプリケーションでは、処理の負荷はもっとバースト的なものになる傾向があります。最近のプロセッサーでは、アクティビティーが落ち着いている間、全体的な電力消費と発熱を抑えるために、コアの一部をスリープ状態にします。この電源切断プロセスに伴う唯一の

問題は、需要が高くなったときに、コアをスリープ状態から起こすのに時間が必要になることです。深いスリープ状態からフル稼働できる状態にするまでに必要な時間は、このスレッド切り替え時間の例で示されているようなマイクロ秒単位ではなく、ミリ秒単位にまでなる可能性があります。

リアクティブ・アプリケーション

多くのアプリケーションで、特定のスレッドではブロックしないもう 1 つの理由は、これらのスレッドは、タイムリーな応答を必要とするイベントを処理するために使用されるということです。その典型的な例は、UI スレッドです。UI スレッドで非同期イベントの完了を待機するためにブロックするコードを実行すると、ユーザー入力イベントの処理を遅延させることになります。入力操作、クリック操作、またはタッチ操作に対して、アプリケーションが応答するのを待つのが好きな人はいません。そのため、UI スレッドでのブロッキングは、ユーザーからのバグ・レポートに直ちに反映される傾向があります。

UI スレッドの概念の根底には、さらに一般的な原則があります。それは、ほとんどのタイプのアプリケーション、さらには GUI 以外のアプリケーションでさえも、イベントに応答する必要があるということです。このことから多くの場合、応答時間を短く維持することが重大な懸念事項となります。このようなタイプのアプリケーションでは、ブロッキング待機は許容されるものではありません。

極めて応答性に優れた、スケーラブルなアプリケーションを作成するプログラミング・スタイルは、「リアクティブ・プログラミング」という名前で表されるようになっています。リアクティブ・プログラミングの中心原則は、アプリケーションが以下の条件を満たすことができなければならないというものです。

- イベントに対する反応: アプリケーションはイベント駆動型であり、疎結合されたコンポーネント同士が、あらゆるレベルで非同期通信によってリンクされている必要があります。
- 負荷に対する反応: アプリケーションは、需要の増加に対処するために簡単にアップグレードすることができるように、スケーラブルでなければなりません。
- 障害に対する反応: アプリケーションはレジリエンシーがあり、障害の影響を局所化して迅速に修正できなければなりません。
- ユーザーに対する反応: アプリケーションは、負荷状態でも障害が発生しているときでも、ユーザーに応答する必要があります。

イベントの処理にブロッキング手法を採用するアプリケーションでは、これらの原則を満たすことはできません。スレッドは限られたリソースであるため、スレッドをブロッキング待機の状態にさせようとすると、スケーラビリティーが制限されます。さらに、ブロックされたスレッドはイベントに直ちに応答できないため、レイテンシー (アプリケーションの応答時間) も長くなります。ノンブロッキング手法のアプリケーションでは、イベントに迅速に応答することができるため、レイテンシーが短縮されるとともに、スレッド切り替えのオーバーヘッドが削減され、スループットが向上します。

リアクティブ・プログラミングは、ノンブロッキング・コードを意味するだけではありません。リアクティブ・プログラミングでは、アプリケーションのデータ・フローに注目して、受信側に大きな負荷をかけることも送信側をバックアップすることもなく、これらのデータ・フローを非

同期の相互作用として実装する必要があります。このようにデータ・フローに重点を置くことによって、従来の並行処理プログラミングに伴う複雑さのほとんどを回避できるようになります。

まとめ

この記事では、Java 8 の `CompletableFuture` クラスを使用して複数のイベントを構成して結合し、コードで簡単かつ簡潔に表現できる、ある種の実行計画にする方法を説明しました。このようなノンブロッキング・コードを構成できるということは、ワークロードに適切に対応するとともに、障害にもグレースフルに対処する、リアクティブ・アプリケーションを作成する上で不可欠となります。

次回の記事では話題を Scala に切り替えて、今回説明した方法とはかなり異なるものの、興味深い方法で、非同期でのコンピューター処理を扱う方法を見ていきます。`async` マクロを使用すると、シーケンシャルな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、内部ではそのコードを完全なノンブロッキング構造へと変換します。この手法が役立つ例をいくつか紹介し、`async` の実装方法について見ていきます。

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。