JVM の並行性: Java と Scala での並行処理の基礎

Java 言語での並行処理と、Scala が提供する追加オプションを理解する

Dennis Sosnoski
Principal Consultant
Sosnoski Software Solutions Inc.

2014年 5月 29日

Java プラットフォームでは、あらゆる JVM ベースの言語での並行プログラミングを強力にサポートしています。Scala は Java 言語での並行性サポートを拡張し、さらに多くの方法によりプロセッサー間で処理を共有して結果を調整できるようにしています。この JVM の並行性に関する新しい連載の第 1 回では、Java 7 における最先端の並行プログラミングを取り上げ、Scala による機能強化をいくつか紹介します。この記事は、Java 8 における並行性の機能を理解するための準備にも役立ちます。

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。

プロセッサーの速度は数十年にわたって急速に進化し続けてきましたが、その進化も世紀の変わり目あたりで終わりを遂げました。それ以降、プロセッサー・メーカーはチップのパフォーマンスを高める手段として、クロック速度を上げるよりもコアの数を増やす方法を採るようになっています。今や、マルチコア・システムは、携帯電話からエンタープライズ・サーバーに至るあらゆる機器で標準的に採用されるようになっています。この傾向は今後も続き、さらに拍車がかかっていくことでしょう。開発者はますます、パフォーマンス要件を満たすために、アプリケーション・コードで複数のコアに対処しなければならなくなってきています。

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について見ていきます。そのなかでは、Scala や他の JVM ベースの言語で既に掘り下げてある概念を Java がどのように採り入れているかについても説明します。第 1 回となるこの記事では、JVM における並行プログラミングのより広範な全体像を理解するための背景知識として、Java 7 と Scala での最先端の手法

をいくつか紹介します。具体的には、Java の `ExecutorService` クラスと `ForkJoinPool` クラスを使って並行プログラミングを単純化する方法を学びます。また、プレーン Java に用意されている並行プログラミングのオプションを拡張した、Scala の基本機能もいくつか紹介します。その過程で、異なる手法によって並行プログラミングのパフォーマンスがどのように影響されるかを理解できるはずです。今後の記事では、Java 8 で改善された並行性を取り上げるとともに、Java および Scala でスケーラブルなプログラミングを行うための (Akka ツールキットをはじめとする) 拡張機能についても説明します。

Java の並行性サポート

Java プラットフォームの初期の頃から、並行性のサポートは Java の特徴の 1 つであり、スレッドと同期化の簡潔な実装が、他の競合する言語よりも Java を優位に立たせていました。Java をベースとする Scala は JVM 上で動作し、Java ランタイムのすべての機能 (すべての並行性サポートを含みます) に直接アクセスします。そこで、Scala の機能を探る前に、Java 言語が現在提供している機能を簡単に説明するところから始めます。

Java の基本的なスレッド処理

Java プログラミングでは、スレッドを作成して使用するのは簡単です。スレッドは `java.lang.Thread` クラスで表され、スレッドが実行するコードは `java.lang.Runnable` インスタンスの形をしています。アプリケーションに多数のスレッドが必要であれば、数千個でもスレッドを作成することができます。複数のコアを使用できる場合、JVM はそれらのコアを使用して複数のスレッドを同時に実行します。スレッドの数がコアの数を超えている場合は、スレッド間でコアが共有されます。

Java 5: 並行処理の転換点

Java には当初から、スレッドと同期化のサポートが組み込まれていました。しかし、スレッド間でのデータ共有に関する初期仕様はまだ万全ではなかったため、Java 5 の Java 言語更新 (JSR-133) で大々的な変更が行われました。Java 5 の Java 言語仕様では、`synchronized` と `volatile` を指定したときの動作に修正を加えて正式なものにしています。この仕様では、不変オブジェクトでマルチスレッド処理を扱う方法も詳細に規定しています (基本的に、コンストラクターが実行されているときに、参照を変更することが許可されていなければ、不変オブジェクトは常にスレッド・セーフです)。それ以前のスレッド間でのやりとりには、`synchronized` を指定してブロック化する処理が要求されるのが一般的でしたが、Java 5 での変更により、`volatile` を指定することで、スレッド間でのブロック化を行わない調整が可能になりました。その結果、ブロック化を行わない処理をサポートする新しい並行コレクション・クラスが Java 5 で追加されました。このことは、それまでの必ずブロック化を使用するスレッド・セーフ手法に比べると大幅な改善です。

スレッドの動作を調整するとなると、事態は複雑になってきます。複雑な事態の 1 つは、Java コンパイラーと JVM は、プログラムの観点で整合性が失われない限り、コードに含まれる処理の実行順序を自由に変更できることから生じます。例えば、異なる変数を使用する 2 つの加算処理があるとすると、両方の処理が完了するまで、これらの処理の結果を使用しないようなプログラムになっていれば、コンパイラーや JVM は、コードに指定されているのとは異なる順序で処理を実行することができます。このように処理の順序を変更できる柔軟性があると、Java のパフォーマンスを向上させる上では有効ですが、整合性が保障されるのは単一スレッド内でのみとなります。また、ハードウェアもスレッドに関する問題を作り出す可能性があります。最近のシステムでは複数のレベルのキャッシュ・メモリーを使用しますが、一般に、キャッシュはシステム内の

すべてのコアに同一に映るわけではありません。あるコアがメモリー内の値を変更しても、他のコアにはその変更が即時に可視にならない場合があります。

このような問題があるため、あるスレッドが別のスレッドによって変更されたデータを扱っている間は、この 2 つのスレッド間の相互作用を明示的に制御しなければなりません。この制御を可能にするために、Java では特殊な処理を使用して、別のスレッドが認識するデータのビューに順序付けを設定します。基本的な処理は、スレッドが `synchronized` キーワードを使用してオブジェクトにアクセスするというものです。スレッドは、オブジェクト上で同期をとるときに、そのオブジェクトに固有のロックへの排他アクセスを取得します。そのロックをすでに別のスレッドが保持している場合、ロックを獲得する必要のあるスレッドは、ロックが解放されるまで待機しなければなりません (ブロックされることになります)。スレッドがコードの `synchronized` ブロック内で実行を再開する時点で、Java はそのスレッドが、同じロックを保持していた他のスレッドによって書き込まれたすべてのデータを「認識」することを保証します。ただし、認識されるデータは、他のスレッドがそれぞれの `synchronized` ブロックを離れてロックを解放した時点までに書き込んだデータに限られます。この保証は、コンパイラーや JVM が行う処理の順序変更にも、ハードウェア・メモリー・キャッシュにも適用されます。従って、`synchronized` ブロック内部はコード内で安定性が確保された場所であり、複数のスレッドが順に実行し、相互作用し、情報を安全に共有できる場所です。

変数に `volatile` キーワードを指定すると、やや弱い形でのスレッド・セーフな相互作用が実現されます。 `synchronized` キーワードが保証するのは、スレッドがロックを取得した時点で他のスレッドのストアを認識すること、そしてこのスレッドのストアを、次にロックを取得した他のスレッドが認識することです。volatile キーワードは、この保証を 2 つの部分に分割します。スレッドが `volatile` 変数に書き込む場合、最初に、その時点までに書き込まれたすべての値がフラッシュされます。スレッドが変数を読み取る場合は、スレッドはその変数に書き込まれた値だけでなく、その書き込み処理を実行したスレッドが書き込んだ他のすべての値も認識します。従って、`volatile` 変数の読み取りは、`synchronized` ブロックに入る場合と同様のメモリー保証となり、`volatile` 変数の書き込みは、`synchronized` ブロックから出る場合と同様のメモリー保証となります。ただし、1 つの大きな違いとして、`volatile` 変数の読み取りにしても、書き込みにしても、ブロックされることは決してありません。

Java の並行性の抽象化

同期化は有用であり、Java で開発されている多くのマルチスレッド・アプリケーションは、基本的な `synchronized` ブロックだけを使用しています。その一方、複数のスレッドを調整するという部分が厄介な作業になる可能性があります。特に、多数のスレッドと多数のロックを扱う場合は厄介な作業になりがちです。スレッド・セーフな方法でのみスレッドが相互作用することを確実にするとともに、潜在的なデッドロック (複数のスレッドが、互いにロックが解放されるのを待って、実行を続行できないこと) が回避されることを確実にするのは困難です。スレッドとロックを直接扱うことなく並行性をサポートする抽象化は、開発者が一般的な使用ケースに対処する望ましい方法となります。

`java.util.concurrent` 階層には、同時アクセス、アトミックな処理のラッパー・クラス、同期化プリミティブをサポートするさまざまなコレクションが含まれています。これらのクラスの多くは、ノンブロッキング・アクセスをサポートするように設計されているため、デッドロックの問題が回避され、より効率的なスレッド化が可能になります。これらのクラスを使用すると、ス

レッド間の相互作用の定義および調整が容易になりますが、基本的なスレッド化モデルの複雑さが完全に排除されるわけではありません。

`java.util.concurrent` パッケージには、並行性を扱うためのより疎結合の手法をサポートする抽象化のペアとして、`Future<T>` インターフェースと `Executor` および `ExecutorService` インターフェースが含まれています。これらの関連するインターフェースは、Java の並行性サポートに対する多くの Scala および Akka の拡張機能の基礎となっているため、この 3 つのインターフェースとそれぞれの実装については詳しく調べる価値があります。

`Future<T>` は `T` 型の値を格納するホルダーですが、`Future` が作成された後でないと、通常は値が使用可能にならないように工夫されています。値は、同時に実行される可能性もある非同期処理の結果です。`Future` を受け取るスレッドは、以下の機能を持つメソッドを呼び出すことができます。

- 値が使用可能であるかどうかを確認する
- 値が使用可能になるまで待機する
- 値が使用可能になった時点で取得する
- 値が不要になった場合、処理をキャンセルする

`Future` の実装のそれぞれは、非同期処理に対処するための異なる方法をサポートするように構成されています。

`Executor` は、タスクを実行する「もの」をラップする抽象化です。ここで言う「もの」とは、結局のところ、スレッドのことですが、スレッドがタスクを実行する方法の詳細は、このインターフェースによって隠されます。`Executor` は、単独ではその有用性が限られますが、`ExecutorService` サブインターフェースを併せて使用することで、タスクの終了を管理したり、タスクの結果として `Future` を生成したりするなどの拡張メソッドを提供することができます。`Executor` の標準的な実装では、いずれも `ExecutorService` を実装するため、実際にはルート・インターフェースを無視することができます。

スレッドは比較的重たいリソースなので、スレッドを割り当てて破棄するよりも、再利用する方が賢明です。`ExecutorService` によって、スレッド間での作業の共有が単純になると同時に、スレッドを自動的に再利用できるようになるため、プログラミングが容易になり、パフォーマンスも向上する結果となります。`ExecutorService` の `ThreadPoolExecutor` 実装は、タスクを実行するスレッドのプールを管理します。

Java の並行性の適用

並行性を備えた実際のアプリケーションには、メインの処理ロジックとは独立した、外部 (ユーザー、ストレージ、他のシステム) との相互作用が必要になるタスクを伴うことがよくあります。そのようなアプリケーションを単純な例に簡略化するのは難しいことから、並行性のデモとしては、数値計算やソートといった、単純なコンピューター処理が集約されたタスクがよく使用されます。この記事でも、同様の例を使用します。

ここで取り上げるタスクは、不明な入力に最も近い既知の単語を見つけるというものです。ここで言う「最も近い」とは、「レーベンシュタイン距離」の観点で定義されており、不明な入力を既知の単語に変換するために追加、削除、置換しなければならない文字数が最も少ないことを意

味します。使用するコードは、Wikipedia の記事「Levenshtein distance」に記載されているサンプル・コードに基づいています。このコードは、既知の単語ごとにレーベンシュタイン距離を計算して、最もよく一致したものを返します (または、複数の既知の単語が同じ距離である場合には、不定の結果を返します)。

リスト 1 に示す Java コードで、レーベンシュタイン距離を計算します。この計算では、比較対象の 2 つのテキストの長さにそれぞれ 1 を加えたサイズと一致する行と列からなる行列を生成します。効率化を図るため、この実装ではターゲット・テキストのサイズに合わせた配列のペアを用いて、行列内の連続する行を表し、繰り返し処理のパスごとにこれらの配列を交換します。このようにするのは、次の行を計算するには、直前の行の値だけが必要なためです。

リスト 1. Java でのレーベンシュタイン距離の計算

```
/**
 * Calculate edit distance from targetText to known word.
 *
 * @param word known word
 * @param v0 int array of length targetText.length() + 1
 * @param v1 int array of length targetText.length() + 1
 * @return distance
 */
private int editDistance(String word, int[] v0, int[] v1) {

    // initialize v0 (prior row of distances) as edit distance for empty 'word'
    for (int i = 0; i < v0.length; i++) {
        v0[i] = i;
    }

    // calculate updated v0 (current row distances) from the previous row v0
    for (int i = 0; i < word.length(); i++) {

        // first element of v1 = delete (i+1) chars from target to match empty 'word'
        v1[0] = i + 1;

        // use formula to fill in the rest of the row
        for (int j = 0; j < targetText.length(); j++) {
            int cost = (word.charAt(i) == targetText.charAt(j)) ? 0 : 1;
            v1[j + 1] = minimum(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost);
        }

        // swap v1 (current row) and v0 (previous row) for next iteration
        int[] hold = v0;
        v0 = v1;
        v1 = hold;
    }

    // return final value representing best edit distance
    return v0[targetText.length()];
}
```

マルチコア・システムでコードを実行する場合、不明な入力に対して比較する既知の単語の数が多ければ、並行処理を使用することで処理速度を上げることができます。そのために、既知の単語のセットを複数のチャンクに分割して、各チャンクを個別のタスクとして処理します。各チャンクの単語数を変えることで、簡単にタスクの分割の細分度を変更して、全体的なパフォーマンスに対する細分度の影響を確認することができます。リスト 2 は、サンプル・コードの `ThreadPoolDistance` クラスから抜粋した、チャンクに分割した計算に対応する Java コードです。リスト 2 では、使用可能なプロセッサー数をスレッド・カウントに設定した標準の `ExecutorService` を使用しています。

リスト 2. 複数のスレッドを使用して、チャンクに分割した Java での距離の計算

```
private final ExecutorService threadPool;
private final String[] knownWords;
private final int blockSize;

public ThreadPoolDistance(String[] words, int block) {
    threadPool = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
    knownWords = words;
    blockSize = block;
}

public DistancePair bestMatch(String target) {

    // build a list of tasks for matching to ranges of known words
    List<DistanceTask> tasks = new ArrayList<DistanceTask>();

    int size = 0;
    for (int base = 0; base < knownWords.length; base += size) {
        size = Math.min(blockSize, knownWords.length - base);
        tasks.add(new DistanceTask(target, base, size));
    }
    DistancePair best;
    try {

        // pass the list of tasks to the executor, getting back list of futures
        List<Future<DistancePair>> results = threadPool.invokeAll(tasks);

        // find the best result, waiting for each future to complete
        best = DistancePair.WORST_CASE;
        for (Future<DistancePair> future: results) {
            DistancePair result = future.get();
            best = DistancePair.best(best, result);
        }

    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } catch (ExecutionException e) {
        throw new RuntimeException(e);
    }
    return best;
}

/**
 * Shortest distance task implementation using Callable.
 */
public class DistanceTask implements Callable<DistancePair>
{
    private final String targetText;
    private final int startOffset;
    private final int compareCount;

    public DistanceTask(String target, int offset, int count) {
        targetText = target;
        startOffset = offset;
        compareCount = count;
    }

    private int editDistance(String word, int[] v0, int[] v1) {
        ...
    }

    /* (non-Javadoc)
     * @see java.util.concurrent.Callable#call()
     */
    @Override
    public DistancePair call() throws Exception {
```

```
        // directly compare distances for comparison words in range
        int[] v0 = new int[targetText.length() + 1];
        int[] v1 = new int[targetText.length() + 1];
        int bestIndex = -1;
        int bestDistance = Integer.MAX_VALUE;
        boolean single = false;
        for (int i = 0; i < compareCount; i++) {
            int distance = editDistance(knownWords[i + startOffset], v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestIndex = i + startOffset;
                single = true;
            } else if (bestDistance == distance) {
                single = false;
            }
        }
        return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :
             new DistancePair(bestDistance);
    }
}
```

リスト 2 の `bestMatch()` メソッドは、`DistanceTask` インスタンスのリストを作成して、そのリストを `ExecutorService` に渡します。この形の `ExecutorService` の呼び出しは、実行するタスクを表す `Collection<? extends Callable<T>>` 型のパラメーターを取ります。この呼び出しから返されるのは、実行結果を表す `Future<T>` のリストです。`ExecutorService` はこれらの結果に、各タスクに対して `call()` メソッドを呼び出して返された値を非同期で取り込みます。この例の場合、`T` 型は `DistancePair` です。ここに、距離の単純な値オブジェクトおよび一致する単語、または一致する固有の単語がなかった場合は、距離だけが格納されます。

`bestMatch()` メソッド内の元の実行スレッドは、Future ごとにそれまで距離を計算した中で最もよく一致している結果を生成し、すべての `Future` が完了するのを待って、最もよく一致する結果を返します。複数のスレッドで `DistanceTask` を実行する場合、元のスレッドが待機するのは、結果の一部だけです。残りの結果は、元のスレッドが待機する結果と並行して生成されます。

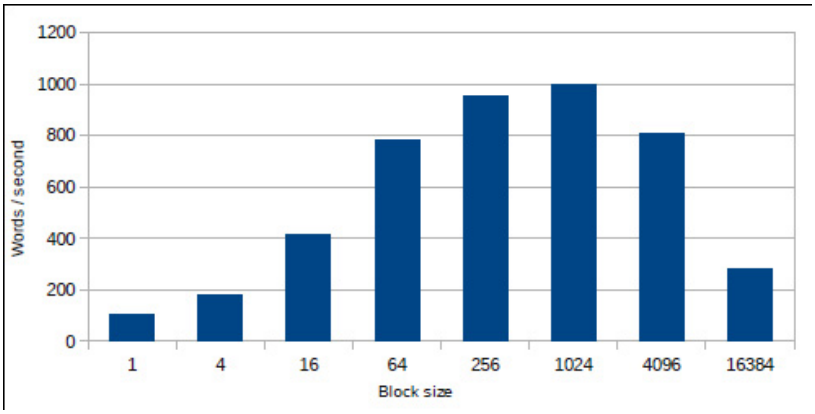
並行処理のパフォーマンス

システムで使用可能なプロセッサーをフル活用するには、少なくともプロセッサーと同じ数のスレッドを設定して `ExecutorService` を構成する必要があります。また、`ExecutorService` には、プロセッサーの数と同じか、それ以上のタスクを渡して実行させる必要もあります。実際には、パフォーマンスを最大限にするために、タスクの数は、プロセッサーの数よりも大幅に多くすることをお勧めします。その場合、プロセッサーは次から次へとタスクを処理してビジー状態を維持し、すべてのタスクが完了するとアイドル状態になります。ただし、タスクと Future を作成し、タスク間でスレッドを切り替え、最後にタスクの結果を返すという流れにはオーバーヘッドが伴うため、タスクのサイズを十分大きくしておくことで、それに比してこのオーバーヘッドが小さくなるようにする必要があります。

図 1 に、4 コア AMD システム上で 64 ビット版 Linux 用の Oracle Java 7 を使用してテスト・コードを実行したときに、タスクの数によって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。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コード・パス間で一時停止して 10 回コード・パスを実行した後の最短時間がグラフには使用されています。図 1 からわかるように、1

秒あたりの入力単語数のパフォーマンスは、適度なブロック・サイズの範囲 (基本的に、256 から 1,024 未満) では安定していて、タスクが非常に小さいか非常に大きい極端な場合にだけパフォーマンスが低下しています。ブロック・サイズが 16,384 になっているケースでは、タスクが 1 つしか作成されないため、シングル・スレッドのパフォーマンスを表しています。

図 1. `ThreadPoolDistance` のパフォーマンス



Fork-Join

Java 7 で、`ExecutorService` のもう 1 つの実装として `ForkJoinPool` クラスが導入されました。`ForkJoinPool` は、`RecursiveAction` クラス (タスクが結果を生成しない場合) または `RecursiveTask<T>` クラス (タスクの結果が `T` 型である場合) を使用して、サブタスクに繰り返し分割可能なタスクを効率的に処理するように設計されています。サブタスクからの結果を統合するには、`RecursiveTask<T>` が便利な方法となります (リスト 3 を参照)。

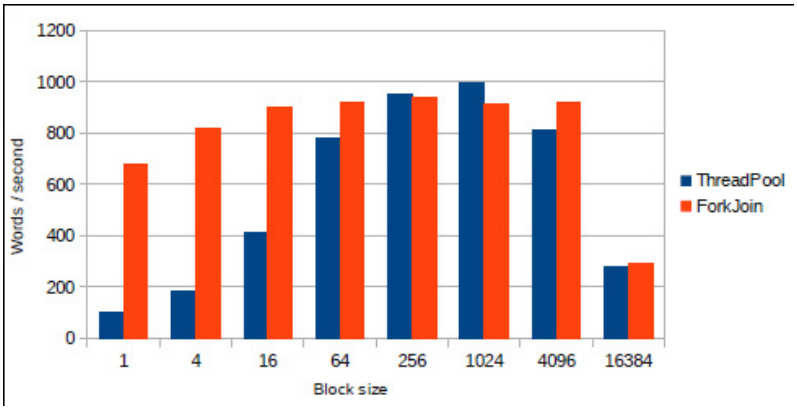
リスト 3. `RecursiveTask<DistancePair>` の例

```
private ForkJoinPool threadPool = new ForkJoinPool();

private final String[] knownWords;

private final int blockSize;

public ForkJoinDistance(String[] words, int block) {
    knownWords = words;
    blockSize = block;
}

public DistancePair bestMatch(String target) {
    return threadPool.invoke(new DistanceTask(target, 0, knownWords.length, knownWords));
}

/**
 * Shortest distance task implementation using RecursiveTask.
 */
public class DistanceTask extends RecursiveTask<DistancePair>
{
    private final String compareText;
    private final int startOffset;
    private final int compareCount;
    private final String[] matchWords;

    public DistanceTask(String from, int offset, int count, String[] words) {
        compareText = from;
        startOffset = offset;
        compareCount = count;
        matchWords = words;
```

```
    }

    private int editDistance(int index, int[] v0, int[] v1) {
        ...
    }

    /* (non-Javadoc)
     * @see java.util.concurrent.RecursiveTask#compute()
     */
    @Override
    protected DistancePair compute() {
        if (compareCount > blockSize) {

            // split range in half and find best result from bests in each half of range
            int half = compareCount / 2;
            DistanceTask t1 = new DistanceTask(compareText, startOffset, half, matchWords);
            t1.fork();
            DistanceTask t2 = new DistanceTask(compareText, startOffset + half,
                compareCount - half, matchWords);
            DistancePair p2 = t2.compute();
            return DistancePair.best(p2, t1.join());
        }

        // directly compare distances for comparison words in range
        int[] v0 = new int[compareText.length() + 1];
        int[] v1 = new int[compareText.length() + 1];
        int bestIndex = -1;
        int bestDistance = Integer.MAX_VALUE;
        boolean single = false;
        for (int i = 0; i < compareCount; i++) {
            int distance = editDistance(i + startOffset, v0, v1);
            if (bestDistance > distance) {
                bestDistance = distance;
                bestIndex = i + startOffset;
                single = true;
            } else if (bestDistance == distance) {
                single = false;
            }
        }
        return single ? new DistancePair(bestDistance, knownWords[bestIndex]) :
            new DistancePair(bestDistance);
    }
}
```

図 2 に、リスト 3 の `ForkJoin` コードとリスト 2 の `ThreadPool` コードのパフォーマンスを比較した結果を示します。`ForkJoin` コードの方が、ブロック・サイズの範囲全体で安定しています。パフォーマンスが大幅に低下するのは、ブロックが 1 つの場合 (つまり、シングル・スレッドで実行した場合) だけです。標準的な `ThreadPool` コードのパフォーマンスは、ブロック・サイズが 256 と 1,024 の場合に限り、ForkJoin コードより上回っています。

図 2. `ThreadPoolDistance` と `ForkJoinDistance` のパフォーマンス比較



これらの結果から、アプリケーションのタスク・サイズを調整することで最大限のパフォーマンスを引き出せるとしたら、`ForkJoin` よりも標準的な `ThreadPool` での方が、調整による効果が少し大きいことがわかります。ただし、`ThreadPool` の「スイート・スポット」は、タスクと使用可能なプロセッサー数に依存するだけでなく、システムの他の側面にも依存する可能性があることを理解しておいてください。一般に、`ForkJoin` では最小限の調整で卓越したパフォーマンスを引き出せるため、できる限り ForkJoin を使用するべきです。

Scala の並行性の基礎

Scala は、さまざまな点で Java プログラミング言語と Java ランタイムを拡張しており、例えば、並行性を扱うためのもっと多くのさらに容易な方法を追加したりしています。第一に、Scala における `Future<T>` は、Java のときより遥かに柔軟性があります。Future をコードのブロックから直接作成できるだけでなく、完了処理をするためのコールバックを Future に関連付けることもできます。リスト 4 に、Scala の Future を使用する例をいくつか記載します。このコードでは最初に、要求に応じて `Future<Int>` を提供するための `futureInt()` メソッドを定義し、その後 3 つの異なる方法で Future を使用しています。

リスト 4. Scala の `Future<T>` サンプル・コード

```
import ExecutionContext.Implicits.global

val lastInteger = new AtomicInteger
def futureInt() = future {
  Thread sleep 2000
  lastInteger incrementAndGet
}

// use callbacks for completion of futures
val a1 = futureInt
val a2 = futureInt
a1.onSuccess {
    case i1 => {
      a2.onSuccess {
        case i2 => println("Sum of values is " + (i1 + i2))
      }
    }
}
Thread sleep 3000

// use for construct to extract values when futures complete
val b1 = futureInt
val b2 = futureInt
```

```
for (i1 <- b1; i2 <- b2) yield println("Sum of values is " + (i1 + i2))
Thread sleep 3000

// wait directly for completion of futures
val c1 = futureInt
val c2 = futureInt
println("Sum of values is " + (Await.result(c1, Duration.Inf) +
  Await.result(c2, Duration.Inf)))
```

リスト 4 の最初の例では、コールバック・クロージャーを Future のペアに関連付け、両方の Future が完了した時点で 2 つの結果の値の合計がコンソールに出力されるようにしています。コールバックは Future の作成順に直接 Future 上でネストされていますが、順序を変えても同じように機能します。コールバックを関連付けるときに、すでに Future が完了している場合、コールバックはそれでも実行されますが、すぐに実行される保証はありません。元の実行スレッドが `Thread sleep 3000` の行で一時停止しているのは、Future が完了してから次の例に移れるようにするためです。

2 番目の例は、Scala の `for` 内包を使用して、非同期で Future から値を抽出し、それらの値を式で直接使用する方法を示しています。`for` 内包は、各種処理 (`map`、`filter`、`flatMap`、および `foreach`) の複雑な組み合わせを簡潔に表現するために使用できる、Scala の構成体で、通常はさまざまな形のコレクションで使用されますが、Scala の Future は、コレクションの値にアクセスするために使用されるのと同じ単項のメソッドを実装します。従って、Future を、最大 1 つの値が含まれる (ある時点になるまでは、その値さえ含まれない) 特殊な類のコレクションとして使用することができます。この例での `for` 文は、2 つの Future の結果を取って、その結果の値を式で使用するように指定しています。この手法が裏で生成するコードは、最初の例とほとんど同じですが、線形コードの形で作成すると、理解しやすい単純な式が生成されます。最初の例と同じく、元の実行スレッドは一時停止するので、Future が完了してから次の例に移ることができます。

3 番目の例では、ブロッキング待機を用いて Future の結果を取得します。これは、Java の Future が機能する方法と同等ですが、Scala の場合、最大待機時間を引数に取る特殊な `Await.result()` メソッド呼び出しにより、明示的にブロッキング待機が行われます。

リスト 4 のコードは、Future を `ExecutorService` や同等のインターフェースに明らかな形で渡していないため、Scala を扱ったことのない読者は、Future の背後でコードがどのように実行されるのか疑問に思うかもしれませんが、その答えはリスト 4 の先頭行 `import ExecutionContext.Implicits.global` にあります。Scala の API は、コードのブロック全体で何度も頻繁に使用されるパラメーターに暗黙値を使用することがよくあります。`future { }` 構成体では、暗黙パラメーターとして `ExecutionContext` が使用可能でなければなりません。この `ExecutionContext` は、Java `ExecutorService` の Scala ラッパーであるため、同じように 1 つ以上の管理対象スレッドを使ってタスクを実行するために使用されます。

以上の Future を使用した基本的な処理の他に、Scala には任意のコレクションを、並列プログラミングを使用するコレクションに変換する方法があります。コレクションを並列処理の形に変換すると、(コレクション上で実行される) Scala コレクションの標準的な任意の処理 (`map`、`filter`、`fold` など) は、可能な場合には自動的に並列に実行されるようになります (その一例は、この記事に記載するリスト 7 のコードに、Scala を使用して最もよく一致する単語を見つける部分として含まれています)。

エラー処理

Java の Future と Scala の Future は、どちらもエラー処理の問題に対処する必要があります。Java の場合 (Java 7 の時点)、Future は結果を返さずに `ExecutionException` をスローすることができます。アプリケーションでは、特定のタイプの失敗に対して固有の `ExecutionException` サブクラスを定義することも、複数の例外をつなげて詳細を渡すこともできますが、柔軟性には限りがあります。

Scala の Future では、Javaの Future よりも柔軟にエラーを処理することができます。Scala の Future を完了するには 2 つの方法があります。具体的には、成功した場合には結果の値を返し (値が要求される場合)、失敗した場合には、関連付けられた `Throwable` を発生させることですが、それ以外にも、さまざまな方法で Future の完了を処理することができます。リスト 4 では、`onSuccess` メソッドを使用して、Future の正常な完了を処理するためのコールバックを関連付けていますが、その他に `onComplete` を使用して任意の形の完了を処理することも (結果または Throwable を `Try` にラップして両方のケースに対応します)、`onFailure` を使用して具体的にエラーの結果を処理することもできます。このような Scala Future の柔軟性は、Future を使用して実行できるすべての処理に拡張されるので、エラー処理を直接コードに統合することができます。

Scala の `Future<T>` には、密接に関連した `Promise<T>` クラスもあります。Future は、ある特定の時点で入手できる可能性がある結果のホルダーです (あるいは、結果が入手できない可能性もあります。Future には、いつか完了するという保証は本来備わっていません)。Future が完了すると、その結果は固定されて変更することができません。これと同じことを請け負った場合の別の側面が Promise であり、Promise は結果の値または Throwable といういずれかの形の結果に 1 回だけ割り当てられるホルダーです。Promise からは Future を取得することができ、結果が Promise 上で設定されると、Future にも同じく結果が設定されます。

Scala の並行性の適用

ここまでで基本的な Scala の並行性の概念のいくつかを理解したので、今度はレーベンシュタイン距離の問題のコードを見ていきます。リスト 5 に、Scala で実装した、多少イディオムのようなレーベンシュタイン距離を計算するコードを示します。このコードは基本的に、リスト 1 の Java コードに対応していますが、関数型のスタイルになっています。

リスト 5. Scala でのレーベンシュタイン距離の計算

```
val limit = targetText.length
/** Calculate edit distance from targetText to known word.
  *
  * @param word known word
  * @param v0 int array of length targetText.length + 1
  * @param v1 int array of length targetText.length + 1
  * @return distance
  */
def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {

  val length = word.length

  @tailrec
  def distanceByRow(rnum: Int, r0: Array[Int], r1: Array[Int]): Int = {
    if (rnum >= length) r0(limit)
    else {

      // first element of r1 = delete (i+1) chars from target to match empty 'word'
```

```
      r1(0) = rnum + 1

      // use formula to fill in the rest of the row
      for (j <- 0 until limit) {
        val cost = if (word(rnum) == targetText(j)) 0 else 1
        r1(j + 1) = min(r1(j) + 1, r0(j + 1) + 1, r0(j) + cost);
      }

      // recurse with arrays swapped for next row
      distanceByRow(rnum + 1, r1, r0)
    }
  }

  // initialize v0 (prior row of distances) as edit distance for empty 'word'
  for (i <- 0 to limit) v0(i) = i

  // recursively process rows matching characters in word being compared to find best
  distanceByRow(0, v0, v1)
}
```

リスト 5 のコードでは、各行の値の計算に、末尾再帰の `distanceByRow()` メソッドを使用しています。このメソッドはまず、計算が完了した行の数を確認し、その数がチェック対象の単語に含まれる文字数と一致する場合、結果の距離を返します。一致しない場合は新しい行の値を計算し、次の行を計算するために自身を再帰的に呼び出して終了します (このプロセスでは 2 つの行配列を交換するため、新しい現在行の値が正しく渡されます)。Scala は、末尾再帰のメソッドを Java の `while` ループに相当するものに変換するので、Java コードとの類似性が保たれます。

ただし、上記のコードと Java コードには、1 つの大きな違いがあります。リスト 5 のコードでは、`for` 内包がクロージャーを使用していますが、クロージャーは現在の JVM で効率的に処理されるとは限らず (詳細については、「Why is using for/foreach on a Range slow?」を参照してください)、計算の最も内側のループにかなりのオーバーヘッドを追加します。従って、リスト 5 のように作成されたコードは、Java のときほど高速には実行されません。そこで、リスト 6 ではコードを書き換えて、末尾再帰のメソッドを追加して `for` 内包を置き換えています。このコードはかなり冗長になっていますが、パフォーマンスは Java のコードに匹敵します。

リスト 6. パフォーマンスのために作成し直された計算コード

```
val limit = targetText.length

/** Calculate edit distance from targetText to known word.
  *
  * @param word known word
  * @param v0 int array of length targetText.length + 1
  * @param v1 int array of length targetText.length + 1
  * @return distance
  */
def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {

  val length = word.length

  @tailrec
  def distanceByRow(row: Int, r0: Array[Int], r1: Array[Int]): Int = {
    if (row >= length) r0(limit)
    else {

      // first element of v1 = delete (i+1) chars from target to match empty 'word'
      r1(0) = row + 1

      // use formula recursively to fill in the rest of the row
      @tailrec
```

```
      def distanceByColumn(col: Int): Unit = {
        if (col < limit) {
          val cost = if (word(row) == targetText(col)) 0 else 1
          r1(col + 1) = min(r1(col) + 1, r0(col + 1) + 1, r0(col) + cost)
          distanceByColumn(col + 1)
        }
      }
      distanceByColumn(0)

      // recurse with arrays swapped for next row
      distanceByRow(row + 1, r1, r0)
    }
  }

  // initialize v0 (prior row of distances) as edit distance for empty 'word'
  @tailrec
  def initArray(index: Int): Unit = {
    if (index <= limit) {
      v0(index) = index
      initArray(index + 1)
    }
  }
  initArray(0)

  // recursively process rows matching characters in word being compared to find best
  distanceByRow(0, v0, v1)
}
```

リスト 7 に記載する Scala コードは、リスト 2 の Java コードで行っているのと同じようなブロッキング方式の距離計算を実行します。`bestMatch()` メソッドは、`Matcher` クラス・インスタンスが処理する特定の単語ブロック内のターゲット・テキストに最もよく一致するものを見つけるために、末尾再帰の `best()` メソッドを使って単語を走査します。`*Distance` クラスは、単語の各ブロックに対して `Matcher` インスタンスを作成し、`Matcher` の実行と `Matcher` の結果の結合を調整します。

リスト 7. Scala での複数のスレッドを使用したブロックごとの距離計算

```
class Matcher(words: Array[String]) {

  def bestMatch(targetText: String) = {

    val limit = targetText.length
    val v0 = new Array[Int](limit + 1)
    val v1 = new Array[Int](limit + 1)

    def editDistance(word: String, v0: Array[Int], v1: Array[Int]) = {
      ...
    }

    @tailrec
    /** Scan all known words in range to find best match.
      *
      * @param index next word index
      * @param bestDist minimum distance found so far
      * @param bestMatch unique word at minimum distance, or None if not unique
      * @return best match
      */
    def best(index: Int, bestDist: Int, bestMatch: Option[String]): DistancePair =
      if (index < words.length) {
        val newDist = editDistance(words(index), v0, v1)
        val next = index + 1
        if (newDist < bestDist) best(next, newDist, Some(words(index)))
        else if (newDist == bestDist) best(next, bestDist, None)
```

```
        else best(next, bestDist, bestMatch)
      } else DistancePair(bestDist, bestMatch)

    best(0, Int.MaxValue, None)
  }
}

class ParallelCollectionDistance(words: Array[String], size: Int) extends TimingTestBase {

  val matchers = words.grouped(size).map(l => new Matcher(l)).toList

  def shutdown = {}

  def blockSize = size

  /** Find best result across all matchers, using parallel collection. */
  def bestMatch(target: String) = {
    matchers.par.map(m => m.bestMatch(target)).
      foldLeft(DistancePair.worstMatch)((a, m) => DistancePair.best(a, m))
  }
}

class DirectBlockingDistance(words: Array[String], size: Int) extends TimingTestBase {

  val matchers = words.grouped(size).map(l => new Matcher(l)).toList

  def shutdown = {}

  def blockSize = size

  /** Find best result across all matchers, using direct blocking waits. */
  def bestMatch(target: String) = {
    import ExecutionContext.Implicits.global
    val futures = matchers.map(m => future { m.bestMatch(target) })
    futures.foldLeft(DistancePair.worstMatch)((a, v) =>
      DistancePair.best(a, Await.result(v, Duration.Inf)))
  }
}
```

リスト 7 の 2 つの `*Distance` クラスは、それぞれに別の方法で、Matcher の実行と `Matcher` の結果の結合を調整します。`ParallelCollectionDistance` では、前述の Scala の並列コレクション機能を使用して並列計算の詳細を隠すため、結果を結合するのに必要なのは、単純な `foldLeft` だけとなります。

`DirectBlockingDistance` はそれよりも少し明示的で、Future のリストを作成してから、そのリストに対して `foldLeft` を実行し、ネストされたブロッキング待機によって個々の結果に対処します。

パフォーマンスの再比較

リスト 7 の両方の `*Distance` 実装は、`Matcher` の結果を扱うには妥当な手法です (これらの実装は、唯一の妥当な手法と言うには程遠く、私が実験で試した他の実装を (記事には記載していませんが) サンプル・コードには含めてあります)。この例の場合、パフォーマンスが一番の関心事であるため、この 2 つの実装のパフォーマンスを Java の `ForkJoin` コードのパフォーマンスと比較した結果を図 3 に示します。

図 3. Scala での代替手法と `ForkJoinDistance` のパフォーマンス比較

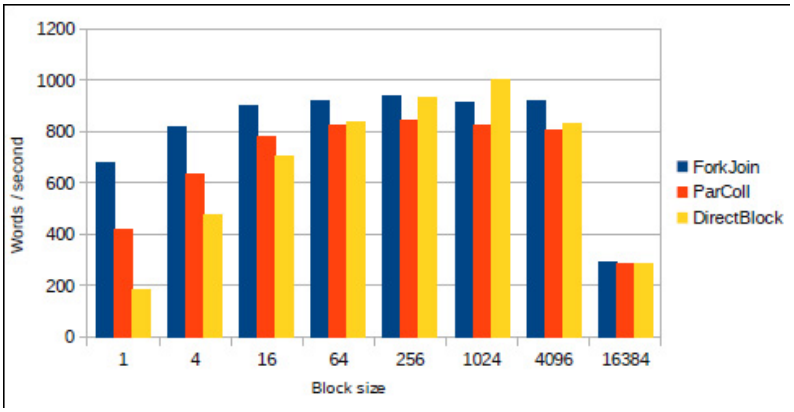


図 3 に示されているように、ブロック・サイズ 1,024 では `DirectBlockingDistance` のパフォーマンスが優れていますが、全体的には Java の `ForkJoin` コードの方が、Scala で実装したいずれのコードよりも優れたパフォーマンスを示しています。また、ほとんどのブロック・サイズで、Scala の両実装は、リスト 1 の `ThreadPool` コードよりも優れたパフォーマンスを示しています。

これらのパフォーマンスの結果は、あくまでも説明のために示したものであり、確定的な結果というわけではありません。皆さんのシステムでタイミング・テストを実行すると、相対的なパフォーマンスに違いが出る可能性があります。動作しているコアの数が異なっていれば、その違いは顕著になるはずです。この距離の計算タスクで最大限のパフォーマンスを実現したければ、最適化を実装することになります。最適化の方法としては、既知の単語をその長さでソートして、入力される単語と同じ長さの単語との比較から開始する方法や (編集距離は、必ず単語の長さの差以上になるため)、距離の計算結果がこれまでの最短距離の値より長くなった時点で、計算から早期に抜ける方法などが考えられます。しかしこの実験では、並行処理によってパフォーマンスがどれだけ改善するか、さらには作業の共有方法の違いによる影響について明らかにするために、比較的単純なアルゴリズムとして公平なジョブを行っています。

パフォーマンスは別にして、リスト 7 に示されている 2 つのバージョンの Scala 制御コードをリスト 2 およびリスト 3 の Java コードと比べると、興味深いことがわかります。それは、Scala のコードは Java コードに比べて非常に短くて (Scala を理解している人にとって) 簡潔になっていることです。この記事の完全なサンプル・コードを見るとわかるように、Scala と Java は共存させることが可能です。Scala コードによって Scala と Java 両方のコードのタイミング・テストを実行したり、Java コードによって Scala コードの一部を直接扱ったりすることができます。このように簡単に共存させられるおかげで、既存の Java コード・ベースに Scala を導入する場合でも、大掛かりな変更をする必要はありません。Java コードの大まかな制御には、最初は Scala を使用するのが妥当なケースはよくあります。その場合、強力な表現力を持つ Scala の機能をフルに利用しても、クロージャーや変換がパフォーマンスに大きな影響を及ぼすことはありません。

リスト 7 に示されている Scala コード `ParallelCollectionDistance` の単純さは、特に魅力的です。この手法を利用すれば、並行処理をコードから完全に抽象化できるため、シングル・スレッドのアプリケーションのようなコードを作成しながらも、複数のプロセッサーのメリットを得ることができます。この手法の単純さには惹かれる一方、Scala 開発に飛び込むのは気が進まなかっ

たり不可能だったりする場合には、幸い、Java 8 が通常の Java プログラミングに同様の機能を提供しています。

今後の連載

今回の記事では、Java と Scala の両方における並行処理の基礎を説明しました。連載の次回の記事では、Java 8 がどのように Java の並行性サポートを (そして、長期的にはおそらく Scala の並行性サポートも) 改善しているかを見ていきます。Java 8 で行われている変更の多くは見覚えがあるものなので (Scala の並行性機能で使用されているのと同じ概念の多くが Java 8 に導入されています)、通常の Java コードで Scala のいくつかの手法をすぐに使えるようになるはずです。その方法については、次回の記事で説明します。

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。