確実な Java ベンチマーク: 第 1 回 問題

Java コードのベンチマークに潜む落し穴を理解する

Brent Boyer
Programmer

2008年 6月 24日

プログラムのパフォーマンスは常に関心のある問題です。それは、このハイパフォーマンス・ハードウェアの時代にあっても変わりません。この 2 回連載の第 1 回目では、まず Java™ コードのベンチマークに伴うさまざまな落し穴を説明します。第 2 回目の記事ではベンチマークの結果についての統計を取り上げ、Java のベンチマークを実行するためのフレームワークを紹介します。ほとんどすべての新しい言語は仮想マシンをベースとしているため、この記事で解説する一般原則は、プログラミング・コミュニティー全体にわたって重要な意味を持ちます。

このシリーズの他の記事を見る

マルチギガヘルツ/マルチコア・プロセッサー、そしてマルチギガバイトの RAM の時代になっても、プログラムのパフォーマンスは相変わらず消えることのない懸念事項です。新たなことに取り組むアプリケーション (あるいは、ますます大きくなっていくプログラマーの怠惰) はハードウェアの能力のあらゆる進化に釣り合ってきましたが、コードのベンチマークの実行 (そして結果から正しい結論を導き出すこと) は常に問題をはらんでいます。その上、Java 言語のベンチマークとなると、最近の高度な仮想マシンではなおさらのこと、他の言語に比べて特に厄介です。

この 2 回の連載で取り上げるのは、プログラム実行時間のみです。その他の重要な実行特性 (メモリー使用量など) については検討しません。このようにパフォーマンスの定義を限定した場合でも、コードのベンチマークを正確に行う際の落し穴はたくさんあります。その落とし穴の数の多さと複雑さのため、ほとんどの「自己流」ベンチマークの試みは正確さを欠くことになり、誤解を招くことになる場合も多々あります。第 1 回の記事ではこのような問題に的を絞り、独自のベンチマーク・フレームワークを作成する際に抑えておかなければならない問題領域を明らかにします。

パフォーマンスに仕掛けられた謎

まずは、ベンチマークの問題を明らかにするパフォーマンスに仕掛けられた謎を説明するところから始めましょう。リスト 1 のコードを見てください (この記事のサンプル・コードの完全版へのリンクは、「参考文献」に記載しています)。

リスト 1. パフォーマンスに仕掛けられた謎

```
protected static int global;
```

```
public static void main(String[] args) {
    long t1 = System.nanoTime();

    int value = 0;
    for (int i = 0; i < 100 * 1000 * 1000; i++) {
        value = calculate(value);
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-6) + " milliseconds");
}

protected static int calculate(int arg) {
    //L1: assert (arg >= 0) : "should be positive";
    //L2: if (arg < 0) throw new IllegalArgumentException("arg = " + arg + " < 0");

    global = arg * 6;
    global += 3;
    global /= 2;
    return arg + 2;
}
```

以下のうち、実行速度に最も優れているバージョンはどれだと思いますか？

A. 現状のままのコードを使う (`calculate` 内で `arg` テストを行わない)
B. `L1` 行のコメントのみを外す。ただし、アサーションを無効にして実行する (`-disableassertions` JVM オプションを使用。これはデフォルトの JVM オプションです)
C. `L1` 行のコメントのみを外す。ただし、アサーションを有効にして実行する (`-enableassertions` JVM オプションを使用)
D. `L2` 行のコメントのみを外す。

少なくとも A (テストなし) が最も速いのは間違いなく、さらに B はアサーションが無効なので、`L1` 行は優れた動的最適化コンパイラーなら削除してしまうデッド・コードになるため、B は A とほぼ同じぐらいの速さになるはず、といった推測をしたのではないでしょうか？そうであれば、残念ながら正解ではありません。リスト 1 のコードは、Cliff Click の 2002 JavaOne でのプレゼンテーション (「参考文献」を参照) を元に編集しました。彼のスライドでは、以下の実行時間が報告されています。

A. 5 秒
B. 0.2 秒
C. (このケースについてのレポートはありません)
D. 5 秒

衝撃的な結果は当然、B です。一体何故、B の実行時間が A の 25 倍も速いのでしょうか。

6 年後の今、リスト 1 のコードを以下の最近の構成で実行してみます (特に記載していない限り、この記事のすべてのベンチマークの結果は、この構成を使用したものです)。

- ハードウェア: 2.2 GHz の Intel Core 2 Duo E4500、2 GB の RAM
- オペレーティング・システム: Windows® XP SP2、2008年3月13日現在のすべての更新を適用
- JVM: 1.6.0_05、すべてのテストに `-server` を使用

結果は以下のとおりです。

A. 38.601 ミリ秒
B. 56.382 ミリ秒
C. 38.502 ミリ秒
D. 39.318 ミリ秒

今度は B が A、C、D より明らかに遅くなっています。しかし、結果はまだ納得できるものではありません。B は A と同じ結果になるはずですが、驚いたことに、実際には B は C よりも遅くなっています。ちなみに、このベンチマークは構成ごとに 4 回測定を行い、完全に再現可能な結果 (1 ミリ秒以内) を取得しました。

Click のスライドでは、なぜ彼があのような奇妙な結果を得ることになったのかを説明しています (結局は、JVM の複雑な振る舞いが原因であることが明らかにされ、そこにはバグも関係していました)。Click は HotSpot JVM の設計者なので、合理的な説明に辿り着いたのは当然と言えるでしょう。それでは普通のプログラマーが正確なベンチマークを実現する望みはないのでしょうか。

望みはあります。この連載の第 2 回では、ダウンロードして自信を持って使える Java ベンチマーク・フレームワークを紹介します。このフレームワークは、ベンチマークにまつわる多くの罠に対処しており、使用するのも簡単で、ほとんどのベンチマークの要求に対しては、ターゲット・コードを一定のタイプのタスク・オブジェクト (`Callable` または `Runnable` のいずれか) にパッケージ化して、`Benchmark` クラスを 1 回呼び出せばよいだけです。あとはすべて (パフォーマンス測定、統計計算、結果レポート)、自動的に行われます。

このフレームワークの簡単なアプリケーションとして、リスト 1 の `main` をリスト 2 のコードに置き換えて、コードのベンチマークを再度実行してみましょう。

リスト 2. `Benchmark` クラスを使用して解決されたパフォーマンスの謎

```
public static void main(String[] args) throws Exception {
    Runnable task = new Runnable() { public void run() {
        int value = 0;
        for (int i = 0; i < 100 * 1000 * 1000; i++) {
            value = calculate(value);
        }
    } };
    System.out.println("Cliff Click microbenchmark: " + new Benchmark(task));
}
```

このコードを先ほどの構成で実行すると、以下の結果になります。

A. 平均 = 20.241 ミリ秒 ...
B. 平均 = 20.246 ミリ秒 ...
C. 平均 = 26.928 ミリ秒 ...
D. 平均 = 26.863 ミリ秒 ...

やっと、正常な結果になりました。A と B の実行時間が基本的に同じという結果です。また、C と D (どちらも同じ引数チェックを実行) もほぼ同じ (ただし A と B の場合と比べると多少長い) 実行時間になりました。

このように `Benchmark` を使うことで期待どおりの結果になった理由は、おそらく `Benchmark` が内部で `task` を何度も実行し、安定した状態の実行プロファイルになるまで「ウォームアップ」結果

を破棄し、安定状態になった上で、一連の正確な測定値を取得したからでしょう。それとは対照的に、リスト 1 のコードはすぐに測定を開始しているため、実行結果は実際のコードにはほとんど関係なく、それよりも JVM の振る舞いに関係していることを意味します。上記の結果では省略していますが (... で示しています)、`Benchmark` は何らかの統計計算を実行し、それによって結果の信頼性を知らせます。

ただし、このフレームワークにすぐに飛びつくことは禁物です。この記事全体をある程度まで理解し、特に「動的最適化」による難解な問題について理解し、また第 2 回で説明する解釈の問題についてもある程度理解してから使用してください。数値を闇雲に信じるのではなく、その数値がどのようにして得られたかを理解することが肝心です。

実行時間の測定

基本的に、コードの実行時間の測定は以下のような簡単な方法で行います。

1. 開始時間を記録する。
2. コードを実行する。
3. 終了時間を記録する。
4. 時間の差を計算する。

ほとんどの Java プログラマーは、直観的にリスト 3 のようなコードを作成するでしょう。

リスト 3. 典型的な Java ベンチマーク・コード

```
long t1 = System.currentTimeMillis();
task.run();    // task is a Runnable which encapsulates the unit of work
long t2 = System.currentTimeMillis();
System.out.println("My task took " + (t2 - t1) + " milliseconds to execute.");
```

リスト 3 の手法は通常、長時間実行するタスクには上手くいきます。例えば、`task` の実行に要する時間が 1 分間であれば、以下に説明するような分解能の問題はないはずです。しかし、`task` の実行時間が短くなるにつれ、このコードは正確でなくなっていきます。ベンチマーク・フレームワークはどんな `task` も自動的に処理するため、リスト 3 は当然、検証を行う必要があります。

1 つの問題は、分解能です。`System.currentTimeMillis` はその名前が示すとおり、ミリ秒というわずかな分解能で結果を返します (「参考文献」を参照)。結果に ±1 ミリ秒の誤差が無作為に含まれるという前提で、実行時間の測定誤差を 1 パーセント以下に抑えなければならないとしたら、`System.currentTimeMillis` は 200 ミリ秒以下で実行するタスクには使えません (差分の測定に伴う 2 つの誤差が合計 2 ミリ秒になるからです)。

実際のところ、`System.currentTimeMillis` の分解能は 10 倍から 100 倍劣る可能性があります。以下は、Javadoc による説明です。

> 戻り値の時間単位はミリ秒ですが、値の細分度はオペレーティング・システムに依存するため、ミリ秒より大きい場合があることに注意してください。例えば、多くのオペレーティング・システムでは、時間の測定単位が何十ミリ秒にもなっています。

表 1 に、さまざまな人によってレポートされた数値を記載します。

表 1. 見出しタグを使用した表

分解能	プラットフォーム	出典 (「参考文献」を参照)
55 ミリ秒	Windows 95/98	Java Glossary
10 ミリ秒	Windows NT、2000、XP (シングル・プロセッサー)	Java Glossary
15.625 ミリ秒	Windows XP (マルチ・プロセッサー)	Java Glossary
最大 15 ミリ秒	Windows (おそらく XP)	Simon Brown
10 ミリ秒	Linux 2.4 カーネル	Markus Kobler
1 ミリ秒	Linux 2.6 カーネル	Markus Kobler

したがって、リスト 3 のコードはタスクの実行時間が約 10 秒を切ると、たちまち機能しなくなってしまいます。

`System.currentTimeMillis` のもう 1 つの問題は、長時間実行するタスクにさえも影響してきます。それは、System.currentTimeMillis が「ウォール・クロック」時間を反映することになっているための問題です。つまり、標準時間からサマータイムへの変更や、NTP (Network Time Protocol) の同期などのイベントが原因となって、System.currentTimeMillis の値は急激に増減することがあります。これらの調整は、ごくまれに誤ったベンチマークの結果を生み出す原因になり得ます。

JDK 1.5 ではが遥かに優れた分解能の API、`System.nanoTime` を導入しました (「参考文献」を参照)。この API は名目上、任意のオフセット以降のナノ秒数を返します。その主な特徴は以下のとおりです。

- 差分時間の測定にのみ役立ちます。
- 正確度と精度 (「参考文献」を参照) が `System.currentTimeMillis` に劣ることは決してありません (ただし、同じ程度という可能性はあります)。
- 最新のハードウェアとオペレーティング・システムでは、マイクロ秒以内の正確度と精度をもたらすことができます。.

結論: ベンチマークを行う際には常に `System.nanoTime` を使うべきです。一般に、System.nanoTime の方が分解能に優れているためです。ただし、ベンチマーク・コードはこの API の分解能が `System.currentTimeMillis` と変わらないという場合にも対処しなければなりません。

JDK 1.5 では、`ThreadMXBean` インターフェースも導入しています (「参考文献」を参照)。このインターフェースにはいくつかの機能がありますが、そのうちベンチマークを行う際に特に関連するのは `getCurrentThreadCpuTime` メソッドです。このメソッドは、(ウォール・クロックの) 経過時間で計測するのではなく、現行のスレッドによって費やされた実際の CPU 時間で計測するため、正確な測定の可能性に期待が持てます。

しかし残念ながら、`getCurrentThreadCpuTime` にはいくつかの問題があります。

- プラットフォームでサポートされない可能性があります。

- サポートされるプラットフォームの間で、セマンティクスが異なる可能性があります (例えば、I/O にかかった CPU 時間は I/O を使用するスレッドの実行時間とされるのが当然ですが、この時間が OS スレッドの実行時間とされる可能性があります)。
- `ThreadMXBean` Javadocs に記載されている不吉な警告によると、「スレッド CPU 測定を有効にすると、一部の Java 仮想マシンの実装では高いコストがかかる可能性がある」ということです (これは、OS 固有の問題です。一部の OS では、スレッド CPU 使用率の測定に必要なマイクロ秒単位での処理が常にオンに設定されているため、`getCurrentThreadCpuTime` によるパフォーマンスへの影響はありません。しかしそれ以外の、マイクロ秒単位での処理がデフォルトでオフに設定されている OS では、これをオンにすると、該当するプロセス内、場合によってはすべてのプロセス内のあらゆるスレッドでのパフォーマンスが劣化することになります)。
- 分解能が定かではありません (ナノ秒の公称分解能で結果を返すため、当然、正確度と精度に関する制約は `System.nanoTime` と同じだと考えがちですが、それについて記載している資料は見たことがありません。あるレポートでは、分解能は大幅に劣ると述べています (「参考文献」を参照)。私の経験では、`getCurrentThreadCpuTime` を使用すると、`nanoTime` よりも平均実行時間が短くなる傾向にあります。例えば、私のデスクトップ構成では、実行時間が約 0.5 から1 パーセント短縮されます。しかし残念ながら測定値のばらつきは激しくなり、標準偏差はすぐに 3 倍にも跳ね上がってしまいます。一方、N2 Solaris 10 マシンでは、実行時間の短縮は 5 パーセントから 10 パーセントで、測定値のばらつきは大幅に減ることはあっても、増えることは決してありませんでした)。
- 最悪なのは、現行のスレッドが使用する CPU 時間が不適切な結果になってしまうという点です。呼び出し側スレッド (CPU 時間の測定対象となる現行スレッド) を持つタスクを考えてみてください。このタスクはただ単に、スレッド・プールを確立し、一連のサブタスクをそのスレッド・プールに送ってから、スレッド・プールの中身の処理が完了するまでアイドル状態になります。呼び出し側スレッドが使用する CPU 時間はごくわずかですが、タスクを完了するまでの全体としての経過時間は長くなります。そのため、まったく誤った実行時間がレポートされる恐れがあります。

上記の問題により、汎用ベンチマーク・フレームワークがデフォルトで `getCurrentThreadCpuTime` を使用するのはあまりにも危険です。そのため、第 2 回で紹介する `Benchmark` クラスでは、特殊な構成でないとこのメソッドが有効にならないようになっています。

以上の時間測定 API すべてに共通して言える注意点は、これらの API には実行オーバーヘッドがあるため、どれくらいの頻度で API を呼び出せば測定時のオーバーヘッドがそれほど結果に影響しないかは、そのオーバーヘッドによって決まるということです。この影響は、プラットフォームに非常に大きく依存します。最近の Windows バージョンを例に挙げると、`System.nanoTime` にはマイクロ秒で実行する OS 呼び出しが伴うため、測定への影響を 1 パーセント未満に抑えるには、呼び出し間隔を 100 マイクロ秒に 1 回以下にしなければなりません (それとは対照的に、`System.currentTimeMillis` は単にグローバル変数を読み出すだけなので、実行時間は極めて短くなります (数ナノ秒)。測定の影響という点に関する限り、呼び出し間隔をさらに縮めることはできますが、グローバル変数はそれほど頻繁に更新されないため (表 1 によると、10 ミリ秒から 15 ミリ秒間隔)、間隔を縮めても意味はありません)。一方、ほとんどの Solaris (ならびに一部の Linux®) マシンでは、`System.nanoTime` の実行速度は System.currentTimeMillis よりも速いのが通常です。

コードのウォームアップ

「パフォーマンスに仕掛けられた謎」のなかで、`Benchmark` の正常な結果は、初期のパフォーマンスではなく、`task` の安定状態の実行プロファイルを測定することによってもたらされると述べました。しかし、Java 実装には大抵、複雑なパフォーマンスのライフサイクルがあります。概して、初期パフォーマンスは比較的遅く、その後安定状態に至るまでは暫くの間、パフォーマンスは大幅に向上します (通常は段階的に急激に向上します)。この安定状態でのパフォーマンスを測定する必要があるのであれば、安定状態に至るまでのすべての要因を理解しなければなりません。

クラスのロード

JVM は通常、最初にクラスを使用するときにだけクラスをロードします。そのため、`task` の最初の実行時間には、JVM が使用する (まだロードされていない) すべてのクラスをロードする時間が含まれます。クラスのロードには通常、ディスク I/O、解析、検証が伴うため、`task` の最初の実行時間はかなり長くなります。このクラスのロードによる影響をなくす一般的な方法は、`task` を何度も実行することです (絶対的にとは言わずに一般的に解決する方法と言っているわけは、`task` には複雑な分岐動作があるかもしれず、`task` を実行したときにすべてのクラスが使われるというわけではないからです。ここで望みとなるのは、`task` を十分な回数実行すれば、これらの分岐が完全に調べられて、関連するあらゆるクラスがまもなくロードされるだろうということです)。

一方、カスタム・クラス・ローダーを使う場合には、JVM が不要になったクラスのアンロードを決めてしまうという別の問題もあります。これによってパフォーマンスに重大な影響が及ぼされることはないはずですが、ベンチマークが行われているときにクラスがアンロードされるのは理想的ではありません。

ベンチマークが行われているときにクラスのロード/アンロードが行われているかどうかをチェックするには、ベンチマークの前後に `ClassLoadingMXBean` の `getTotalLoadedClassCount` メソッドと `getUnloadedClassCount` メソッドを呼び出します (「参考文献」を参照)。ベンチマーク前後の結果が変わっていたら、安定状態にはなっていなかったと判断できます。

混合モード

最近の JVM では、ジャスト・イン・タイム (JIT) コンパイル (「参考文献」を参照) を行う前に (純粋に解釈すると) プロファイル作成のための情報を収集するため、コードを暫く実行させたままにするのが通常となっています。これがベンチマークにとって何を意味するのかと言えば、安定状態の実行プロファイルになるまでに、task を何度も実行しなければならない場合があるということです。例えば、Sun のクライアント/サーバー HotSpot JVM の現行のデフォルトの動作では、コード・ブロックに対して 1,500 回 (クライアント) または 10,000 回 (サーバー) の呼び出しを行ってから、収容メソッドの JIT コンパイルが行われます。

ここでは、一般的な表現としてコード・ブロックと言っているので、コード・ブロックがメソッド全体だけでなく、メソッド内のブロックを意味する場合もあることに注意してください。例えば多くの JVM は、ループされているコード・ブロックを含むメソッドが 1 度しか呼び出されないとしても、そのコード・ブロックが「ホット」コードで構成されていることを認識するだけの機能を備えています。この点については、この後の「OSR」のセクションで詳しく説明します。

以上のことから、安定状態のパフォーマンスのベンチマークを行うには、以下のような作業が必要になります。

1. `task` を 1 回実行して、すべてのクラスをロードする。
2. 安定した状態の実行プロファイルになるまで、`task` を繰り返し実行する。
3. `task` をさらに何回か実行して、実行時間を概算する。
4. ステップ 3 を使用して `n` (累積実行時間が十分長くなるまでの `task` の実行回数) を計算する。
5. `n` 回分の `task` 呼び出しの合計実行時間 `t` を測定する。
6. 実行時間を `t/n`で評価する。

`task` の `n` 回分の実行時間 (ここで、`n` >= 1) を測定する目的は、累積実行時間を長くして、前述したすべての時間測定値の誤差がわずかになるようにするためです。

ステップ 2 には厄介な問題があります。それは、JVM が task の最適化を完了した時点をどうやって判断するかという問題です。

実行時間が収束するまで測定するのが賢い方法のように思えますが、実際には上手くいきません。例えば JVM がまだプロファイルを作成している最中だとしたら、ステップ 5 を開始した時点で、安定状態になる前のプロファイルが JIT コンパイルに適用されてしまいます。そうなると、特に後になってから問題が出てきます。

さらに、収束を定量化する方法も問題になります。

連続コンパイルについて

現在、Sun の HotSpot JVM では、1 回のプロファイル作成フェーズの後に、可能なコンパイルが行われるだけです。脱最適化は別としても、ホットスポット技術ではプロファイル作成コードのオーバーヘッドがあまりにも大きいため、連続コンパイルは現時点では行われません (「参考文献」を参照)。

このプロファイル作成のオーバーヘッドという問題を解決する策はあります。例えば、JVM では 2 つのバージョンのメソッドを保持することができます。一方はプロファイル作成コードが含まれない高速バージョン、そしてもう一方はプロファイル作成を行う遅いバージョンです (「参考文献」を参照)。JVM が通常は高速バージョンを使用し、時折遅いバージョンに交換すれば、パフォーマンスに大きな影響を与えずにプロファイル情報を維持することができます。あるいは、アイドル状態のコアが使用可能な場合には、遅いバージョンを同時に実行するという手もあります。このような手法によって、いずれは連続コンパイルが標準になると考えられます。

もう 1 つの方法 (`Benchmark` クラスが使用する方法) は、ただ単に、事前定義した期間 (かなり長い期間)、task を連続して実行することです。10 秒間のウォームアップ・フェーズがあれば十分なはずです (Click のプレゼンテーションの 33 ページを参照)。この方法は実行時間が収束するまで測定する方法に比べ、信頼性に優れているとは言えませんが、実装はしやすくなります。また、パラメーター化するのも簡単です。ユーザーは直観的にこの概念を理解し、ウォームアップ期間が長ければ長いほど信頼性の高い結果がもたらされることに気付くはずです (ただし、ベンチマークの時間が長くなるという代償はあります)。

JIT コンパイルがいつ発生するかを判断することができれば、パフォーマンスが安定状態に至っているかどうかについて確信を持てるようになります。具体的には、安定状態のパフォーマンスを

達成したと思ってベンチマークを開始した後、ベンチマークのなかでコンパイルが実行されたことに気が付けば、ベンチマークを中止して再試行することが可能になります。

私の知る限り、JIT コンパイルの有無を検出する完璧な方法はありません。最善の方法は、ベンチマークの前後に `CompilationMXBean.getTotalCompilationTime` を呼び出すことです。しかし残念ながら、`CompilationMXBean` の実装は失敗に終わったため、この方法には問題があります。また、別の方法として `-XX:+PrintCompilation` JVM オプションを使う場合には、`stdout` を解析 (または手動で監視) しなければならないことに注意してください (「参考文献」を参照)。

動的最適化

ウォームアップの問題に加え、JVM が行う動的コンパイルにもベンチマークに影響するいくつかの問題があります。これらの問題は捕らえにくいだけでなく、問題に対処する責任は唯一、ベンチマーク・プログラマーにあります。ベンチマーク・フレームワークで対処できることはほとんどありません (この記事の「キャッシング」および「準備」セクションでも、ベンチマーク・プログラマーが責任を担う問題を取り上げていますが、そのほとんどは常識的な問題です)。

脱最適化

問題の 1 つは、脱最適化 (deoptimization) です (「参考文献」を参照)。JVM はコンパイルしたメソッドの使用を停止し、メソッドの解釈に再び時間を費やしてからメソッドを再コンパイルすることがあります。この状態は、動的最適化を行うコンパイラーが前提としていた解釈が現状と合わなくなると発生します。その一例は、単一型コール変換 (monomorphic call transformation) を無効にするクラス・ロードです。別の例としては、例外トラップもあります。つまり、コード・ブロックが最初にコンパイルされるときには最も実行される可能性の高いコード・パスがコンパイルされ、例外的な分岐 (例外パスなど) は解釈されないままになります。しかし、この例外トラップが通例は実行されることが判明するとホットスポット・パスとなり、再コンパイルが行われるというわけです。

つまり、前のセクションでのアドバイスに従って安定状態のパフォーマンスに到達できたように思えたとしても、パフォーマンスが突然変わる可能性があることを認識しておかなければなりません。これも、ベンチマークに含まれる JIT コンパイルを見つけることが不可欠なもう 1 つの理由です。

OSR

別の問題は、特定のコード構造を最適化できるようにする高度な JVM の機能、OSR (On-Stack Replacement) です。リスト 4 のコードを見てください。

リスト 4. OSR が適用されるコードの例

```
private static final int[] array = new int[10 * 1000];
static {
    for (int i = 0; i < array.length; i++) {
        array[i] = i;
    }
}

public static void main(String[] args) {
    long t1 = System.nanoTime();
```

```
    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // outer loop
        for (int j = 0; j < array.length; j++) {    // inner loop 1
            result += array[j];
        }
        for (int j = 0; j < array.length; j++) {    // inner loop 2
            result ^= array[j];
        }
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}
```

JVM が単にメソッド呼び出しのカウントを保持するだけなのであれば、一度しか呼び出されない `main` のコンパイル済みバージョンは決して使用されることはありません。この問題を解決するため、JVM はメソッド内部のコード・ブロックの実行回数をカウントします。リスト 4 のコードで言うと、JVM はそれぞれのループが実行された回数を追跡できます (ループを閉じる中括弧は「後方分岐」となります)。デフォルトでは、どのループも約 10,000 回繰り返された後にメソッド全体のコンパイルを実行するように設定されます。main が再度呼び出されることはないため、単純な JVM では、この `main` のコンパイル済みコードは決して使用されません。一方、OSR を使用する JVM は、メソッド呼び出しの最中に現行のコードを新しくコンパイルされたコードに置換するだけの賢さが備わります。

一見したところ、OSR は素晴らしい機能です。まるで、JVM があらゆるコード構造を処理しながらも、引き続き最適なパフォーマンスをもたらすかのように思えます。しかし OSR には残念なことに、あまり知られていない欠点があります。それは、OSR を使用すると、コードの品質が準最適になる可能性があるということです。例えば OSR は、ループ内不変式の移動、配列範囲チェックの削除、ループのアンロールなどを実行できない場合があります (「参考文献」を参照)。つまり OSR が使用されている場合、最適なパフォーマンスのベンチマークが行われていない可能性があるのです。

最適なパフォーマンスを前提としなければならない場合、OSR に対処する唯一の方法は、OSR が発生しそうな場所を突き止め、可能であれば OSR が発生しないようにコードを作成し直すことです。それには通常、別個のメソッドにキーとなる内部ループを配置する必要があります。リスト 5 は、リスト 4 のコードをコーディングし直した場合の一例です。

リスト 5. OSR が適用されないようにコーディングし直したコード

```
public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // sole loop
        result = add(result);
        result = xor(result);
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}

private static int add(int result) {    // method extraction of inner loop 1
    for (int j = 0; j < array.length; j++) {
```

```
        result += array[j];
    }
    return result;
}

private static int xor(int result) {    // method extraction of inner loop 2
    for (int j = 0; j < array.length; j++) {
        result ^= array[j];
    }
    return result;
}
```

リスト 5 は、`add` メソッドと `xor` メソッドをそれぞれ 1,000,000 回呼び出すため、完全に JIT コンパイルが行われて最適な形になります。この特定のコードの場合、私の構成で最初の 3 回で測定した実行時間は、10.81 秒、10.79 秒、10.80 秒でした。リスト 4 のコード (すべてのループが `main` 内に含まれるため、OSR がトリガーされます) はこれとは対照的に、実行時間が 2 倍になります (最初の 3 回の実行では、21.61 秒、21.61 秒、21.6 秒でした)。

OSR について最後に言っておくことは、これはプログラマーが怠けて `main` などの単一のメソッドにあらゆるものを詰め込んだ場合に起こる、ベンチマークに限ったパフォーマンス問題だということです。実際のアプリケーションでは、プログラマーは当然 (そうであることを祈ります)、多数の細分化したメソッドを作成します。さらに、パフォーマンスが重要となるコードは、長時間実行して重要なメソッドを何度も呼び出すのが通常です。したがって通常は、実際のコードが OSR のパフォーマンス問題に影響されやすいということはありません。アプリケーションでは OSR にあまり神経質にならないようにし、OSR のために簡潔なコードを台無しにしないようにしてください (OSR が問題であることが確実な場合を除き)。注目する点として、`Benchmark` はデフォルトで task を数回実行して統計を取りますが、この数回の実行には、OSR のパフォーマンス問題をなくす副次作用があります。

デッド・コードの削除

もう 1 つの捕えにくい問題は、デッド・コードの削除 (DCE) です (「参考文献」を参照)。コンパイラーは状況次第では、一部のコードが出力にまったく影響しないと判断し、そのコードを削除することがあります。リスト 6 に、標準的なコード例を記載します。このコードは、静的に (つまり、コンパイル時に `javac` を使用して) 実行することもできます。

リスト 6. DCE が適用されるコードの例

```
private static final boolean debug = false;

private void someMethod() {
    if (debug) {
        // do something...
    }
}
```

`javac` は、リスト 6 の `if (debug)` ブロック内にあるコードは実行されることがないと認識し、このコードを削除します。動的コンパイラーは、メソッドのインライン化が行われた場合をはじめ、さまざまな方法でコードがデッド・コードであることを判断します。ベンチマークの実行中に DCE が行われると問題になることは、実行されるコードが、結局はコード全体のわずかなサブセットでしかなくなってしまうことです。さらに、一部の計算が行われなくなる可能性もあるため、実行時間が不当に短くなってしまう可能性があります。

コンパイラーがデッド・コードを判断するために使用する可能性のあるすべての基準について、私は十分な説明をまだ見つけていません (「参考文献」を参照)。決して実行されることのない (到達不能) コードは明らかにデッド・コードですが、JVM は多くの場合、それよりも積極的な DCE ポリシーに従います。

例えば、リスト 4 のコードをもう一度見てください。`main` は `result` を計算するだけでなく、`result` をその出力にも使用しています。ほんのわずかな変更を加え、`println` から `result` を取り除いたとしたらどうなるでしょうか。この場合、積極的なコンパイラーは `result` を計算する必要はまったくないと結論付けるはずです。

これは理論上の問題だけではありません。今度はリスト 7 のコードを見てください。

リスト 7. 出力での `result` の使用による DCE の停止

```
public static void main(String[] args) {
    long t1 = System.nanoTime();

    int result = 0;
    for (int i = 0; i < 1000 * 1000; i++) {    // sole loop
        result += sum();
    }

    long t2 = System.nanoTime();
    System.out.println("Execution time: " + ((t2 - t1) * 1e-9) +
        " seconds to compute result = " + result);
}

private static int sum() {
    int sum = 0;
    for (int j = 0; j < 10 * 1000; j++) {
        sum += j;
    }
    return sum;
}
```

リスト 7 のコードは、私の構成では常に 4.91 秒で実行します。ここで、`println` 文を変更して `result` への参照を取り除いたとします。つまり、この文を `System.out.println("Execution time: " + ((t2 - t1) * 1e-9) + " seconds to compute result");` に変更したとすると、コードは常に 0.08 秒で実行するようになります。明らかに、DCE によって処理全体が削除されています (DCE の別の例については、「参考文献」を参照)。

DCE がベンチマーク対象の処理を削除しないことを保証する唯一の方法は、その処理が何らかの結果を生成するようにし、生成された結果を何らかの方法で使用することです (例えば、リスト 7 の `println` のように出力で使用するなど)。この方法は、`Benchmark` クラスがサポートします。task が `Callable` の場合には、必ずベンチマーク対象の処理を使って `call()` メソッドが返す結果を計算してください。task が `Runnable` の場合には、必ずベンチマーク対象の処理を使って `toString` メソッド (`Object` クラスの `toString` メソッドをオーバーライドしたものでなければなりません) が使用する内部状態を計算してください。以上のルールに従えば、`Benchmark` は完全に DCE を防げるはずです。

OSR と同じく、DCE は通常、実際のアプリケーションでは問題になりません (特定の時間内で実行するコードを頼りにしているのでない限り)。一方、OSR と異なる点は、DCE は不完全に作成され

たベンチマークでは大きな問題になることです。OSR は正確さに欠ける結果をもたらす可能性があるだけですが、DCE は、まったく誤った結果をもたらす恐れがあります。

リソースの再利用

通常、JVM が自動的に行うリソースの再利用には、ガーベッジ・コレクション (GC: Garbage Collection) とオブジェクト・ファイナライゼーション (OF: Object Finalization) という 2 つのタイプがあります。プログラマーの見方からすれば、GC/OF にはほとんど確定性がありません。つまり、結局のところ GC/OF はプログラマーの制御範囲外にあり、JVM が必要と判断すればいつでも行われます。

ベンチマークでは、タスク自体によって発生した GC/OF 時間を考慮しなければなりません。例えば、初期の実行が短時間だという理由でタスクの処理速度が速いと主張するのは誤っています。タスクが最終的には大々的な GC 時間を発生させる可能性があるからです (ただし、一部のタスクはオブジェクトを作成する必要がなく、作成済みのオブジェクトにアクセスすればよいことに注意してください。例えば、配列要素にアクセスするのに必要な時間を判断することを目的としたベンチマークの場合、タスクで配列を作成するのではなく、配列を別途作成し、その配列をタスクが参照できるようにします)。

一方、対象タスクで発生した GC/OF は、同じ JVM セッションでの他のコードによって発生した GC/OF から切り分ける必要もあります。そのためにプログラマーができることと言えば、JVM をクリーンアップしてからベンチマークを実行すること、さらにそのタスク自体による GC/OF が完全に完了してから測定が終わるようにすることぐらいです。

JVM のクリーンアップには、`System` クラスが公開する `gc` および `runFinalization` メソッドを使用することができます。これらのメソッドの Javadoc には、「メソッド呼び出しから制御が戻る時点で、Java 仮想マシンは [GC/OF を実行] するための最大限の努力を済ませています」としか記載されていないことに注意してください。

第 2 回で紹介する `Benchmark` クラスでは、以下の方法で GC/OF の対処を試みています。

1. 測定を行う前に、`cleanJvm,` というメソッドを呼び出します。このメソッドは、メモリー使用量が安定し、ファイナライズ対象のオブジェクトがなくなるまで必要なだけ `System.gc` および `System.runFinalization` を呼び出します。
2. デフォルトでは、タスクの実行時間を 60 回測定します。毎回の測定は最低でも 1 秒は続くので (測定ごとに、必要に応じてタスクを複数回呼び出すため)、実行時間は全体として 1 分以上になります。この合計時間には、60 回の測定にわたっての GC/OF ライフサイクルが含まれるため、完全な振る舞いを正確にサンプリングするには十分なはずです。
3. すべての測定が終了した後、最後に 1 回、`cleanJvm,` を呼び出します。ただしこの場合に測定するのは、呼び出しにかかる時間です。最終クリーンアップ時間がタスクの合計実行時間の 1 パーセント以上を占める場合、ベンチマーク・レポートによって、GC/OF コストが正確に測定値に計上されていない可能性が警告されます。
4. GC/OF は毎回の測定で雑音源のような働きをするため、統計を使用して信頼できる結論を引き出します。

注意する点として、私が最初に `Benchmark` を作成したときには、GC/OF コストを毎回の測定の中に含められるようにしようとして、リスト 8 のようなコードを使っていました。

リスト 8. GC/OF を考慮した誤った方法

```
protected long measure(long n) {
    cleanJvm();    // call here to cleanup before measurement starts

    long t1 = System.nanoTime();
    for (long i = 0; i < n; i++) {
        task.run();
    }
    cleanJvm();    // call here to ensure that task's GC/OF is fully included
    long t2 = System.nanoTime();
    return t2 - t1;
    }
```

問題は、`System.gc` と `System.runFinalization` を測定ループ内で呼び出すと、GC/OF コストを誤って解釈される恐れがあるという点です。具体的に言うと、`System.gc` は STW (Stop-The-World) コレクター (「参考文献」を参照) を使って、生成されたすべてのオブジェクトの完全なガーベッジ・コレクションを行います (これはデフォルトの動作ですが、`-XX:+ExplicitGCInvokesConcurrent` や `-XX:+DisableExplicitGC.` などの JVM オプションもあることに注意してください)。その一方で、アプリケーションで通常使用するガーベッジ・コレクターはそれとはかなり異なり、例えば同時に動作するように構成されるなど、ほとんどコストをかけずに部分的なコレクションを行います (特に生成されてから間もない場合)。同様に、ファイナライザーも通常はバックグラウンド・タスクとして処理されるため、これらのコストは通常システムのアイドル時間に分散されます。

キャッシング

ハードウェア/オペレーティング・システムのキャッシュは、時としてベンチマークを複雑にします。その単純な一例はファイルシステムのキャッシングで、これはハードウェアまたは OS で行われます。ファイルからバイトを読み取るためにかかる時間をベンチマークで評価している一方、ベンチマーク・コードが同じファイルを何度も読み取るとしたら (または、同じベンチマークを何回も実行する場合)、I/O 時間は最初の読み取りの後、劇的に短くなります。不規則なファイル読み取りのベンチマークが必要な場合には、キャッシングを避けるために、異なるファイルが読み取られるようにする必要が出てくるはずです。

メイン・メモリーの CPU キャッシングは非常に重要なため、特に注意が必要です (「参考文献」を参照)。この約 20 年間で CPU の処理速度は指数関数的に向上しましたが、メイン・メモリーの処理速度はなだらかな直線での向上にとどまっています。この速度の不釣合いを改善するため、最近の CPU では広範にキャッシングを使っています (最近の CPU で使われているトランジスターの大半がキャッシング専用になっているほどです)。CPU キャッシュとの相性がよいプログラムは、そうでないプログラムよりも大幅にパフォーマンスが向上するはずです (実際のワークロードの大部分を達成しても、CPU の理論上のスループットはわずかです)。

プログラムと CPU キャッシュとの相性を左右する要因は数多くあります。例えば、最近の JVM はメモリー・アクセスの最適化に四苦八苦して、ヒープ領域を再配置したり、ヒープにあった値を CPU レジスターに持ってきたり、スタックを割り当てたり、あるいはオブジェクト展開を実行したりしています (「参考文献」を参照)。しかし重要な要因は、単にデータ・セットのサイズです。例えば、`n` がタスクのデータ・セットのサイズだとすると (タスクが長さ `n` の配列を使用するとします)、たった 1 つの `n` の値を使用したベンチマークから引き出された結論は誤解を招く可

能性が非常に高いため、さまざまな n の値を使って一連のベンチマークを行わなければなりません。その絶好の例が、J. P. Lewis と Ulrich Neumann による記事 (「参考文献」を参照) に載っています。彼らは C と Java FFT のパフォーマンス相対グラフを n (n はこの場合、配列の大きさ) の関数として作り直し、n に選択した値によって Java パフォーマンスの速度が C に比べて、最高で 2 倍に向上し、最低で 1/2 に劣化することを明らかにしています。

準備

ベンチマークの落とし穴は、開発するベンチマーク・フレームワークのなかだけで片付けられるものではありません。ベンチマーク・プログラムを実行する前に、そのプログラムを実行するシステムのいくつかの領域にも対処する必要があります。

省電力機能

ノート PC ではなおさらのことですが、下位レベルのハードウェアでの問題は、省電力機能 (APM (Advanced Power Management) や ACPI (Advanced Configuration and Power Interface) など) によってベンチマークの最中に電源状態の遷移が起こらないようにすることです。ベンチマーク自体の CPU アクティビティーによって、コンピューターがハイバーネーション・モードに入るなどの急激な電源状態の変化が起こることはないでしょう。また、起こったとしてもすぐに検出されます。しかし、これ以外の電源状態の変化は油断できません。例えば、ベンチマークが最初に CPU バウンドになっている間、OS がハード・ディスクの電源オフを決定したとします。その後、タスクが実行の終了時にハード・ディスクを使用しなければならなくなったとしたらどうなるでしょうか。この場合、ベンチマークは完了するでしょうが、I/O の時間が長引くはずです。別の例としては、Intel SpeedStep や同様の技術を使用して CPU を動的に節電するシステムがあります。ベンチマークを行う前には、これらの影響が出ないように OS を構成してください。

他のプログラム

タスクのベンチマークを行っている間は当然、他のプログラムを実行することは禁物です (負荷をかけたマシンでのタスクの振る舞いを確認することが目的の場合は別として)。必須ではないバックグラウンド・プロセスをすべて終了するとともに、スケジュールされたプロセス (スクリーン・セーバーやウィルス・スキャナーなど) がベンチマークの最中に開始しないようにしてください。

Windows には、ベンチマークを実行する前に、保留中のすべてのアイドル・プロセスを実行するための `ProcessIdleTask` API が用意されています。 この API にアクセスするには、コマンドラインから以下を実行します。

```
Rundll32.exe advapi32.dll,ProcessIdleTasks
```

この API をしばらく呼び出していない場合は、実行に数分かかる場合があるので注意してください (それ以降は、通常数秒間で実行が終了します)。

JVM オプション

ベンチマークに影響する JVM オプションはいくつもあります。そのうちのいくつかを以下に記載します。

- JVM のタイプ (サーバー (`-server`) またはクライアント (`-client`))。

- 使用可能なメモリーが十分にあることを確認します (`-Xmx`)。
- 使用されるガーベッジ・コレクターのタイプ (高度な JVM では多くの調整オプションを提供していますが、注意して使用してください)。
- クラス・ガーベッジ・コレクションが許可されるかどうか (`-Xnoclassgc`)。デフォルトでは、クラス GC が発生しますが、`-Xnoclassgc` を使用するのが適切かどうかは議論が続いています。
- エスケープ分析が実行されているかどうか (`-XX:+DoEscapeAnalysis`)。
- 大量のページ・ヒープがサポートされるかどうか (`-XX:+UseLargePages`)。
- スレッド・スタック・サイズが変更されたかどうか (`-Xss128k` など)。
- JIT コンパイルが常に使用されるのか (`-Xcomp`)、使用されないのか (`-Xint`)、あるいはホットスポットでのみ行われるのか (`-Xmixed`。これがデフォルトで、最高レベルのパフォーマンス・オプションです)。
- JIT コンパイルが発生するまでに蓄積されるプロファイルの量 (`-XX:CompileThreshold`)、バックグラウンド JIT コンパイル (`-Xbatch`)、段階的 JIT コンパイル (`-XX:+TieredCompilation`。
- バイアス・ロックが実行されているかどうか (`-XX:+UseBiasedLocking`)。JDK 1.6 以降では自動的に行われることに注意してください。
- 最新の試験的パフォーマンス調整が有効になっているかどうか (`-XX:+AggressiveOpts`)。
- アサーションが有効になっているか、無効になっているか (`-enableassertions` および `-enablesystemassertions`。
- 厳密なネイティブ呼び出しチェックが有効になっているか、無効になっているか (`-Xcheck:jni`)。
- NUMA マルチ CPU システムのメモリー・ロケーション最適化が有効になっているか (`-XX:+UseNUMA`)。

第 1 回のまとめ

ベンチマークを行うのは極めて困難です。明白なものも微妙なものも含め、さまざまな要因が結果に影響します。正確な結果を出すためには、この記事で説明した問題に対する完全な対処能力がなければなりません。そこで考えられるのが、これらの問題の一部をベンチマーク・フレームワークによって解決するという方法です。第 2 回の記事では、そのような堅牢な Java ベンチマーク・フレームワークについて学んでください。

著者について

Brent Boyer

Brent Boyer は、9 年以上の経験を積むプロのソフトウェア開発者です。彼はニューヨーク州ニューヨークのソフトウェア開発会社、Elliptic Group, Inc. の代表を務めています。