Javaの理論と実践: 動的コンパイルとパフォーマンス測定

動的コンパイル下でのベンチマークの危険性

Brian Goetz
Principal Consultant
Quiotix

2004年 12月 21日

Javaのように動的にコンパイルされる言語では、CやC++のように静的にコンパイルされる言語よりもパフォーマンス・ベンチマークを書いたり結果を解釈したりするのがずっと困難です。Javaの理論と実践の今回の記事では、動的コンパイルによってパフォーマンス・テストが複雑になる事例の中から幾つかを、Brian Goetzが解説します。

このシリーズの他の記事を見る

今月の記事では、マイクロベンチマークの悪例を解剖します。正直なところ私達プログラマーはパフォーマンスについて強迫観念を持っており、自分たちが書き、使用し、批判するコードのパフォーマンス特性を知りたがるものです。私が時折パフォーマンスの話題に関して記事を書くと、後から「私の書いたこのプログラムでは、前回あなたが書いた記事とは異なり、動的frosternationは静的blestificationよりも高速です」といったeメールを受け取ることがあります。そうしたeメールに添付されてくる「ベンチマーク」プログラムやその実行方法の多くを見ると、JVMが実際にJavaバイトコードをどのように実行するのかについての理解が決定的に欠けていることが分かります。そこで今後のコラムで紹介する予定の記事の前に、JVMのベールの下に何があるのかを見ることにしましょう。動的コンパイルと最適化を理解することは、良いマイクロベンチマークと悪いマイクロベンチマークの違いを理解する上での鍵となります（そして良いベンチマークは極めて稀なのです）。

動的コンパイル・・・簡単な歴史

Javaアプリケーションでのコンパイル・プロセスは、CやC++など静的にコンパイルされる言語とは異なります。静的コンパイラーはソース・コードを、対象とするプラットフォームでそのまま実行されるマシン・コードに直接変換します。そして、異なるハードウェアでは異なるコンパイラーが必要になります。JavaコンパイラーはJavaソース・コードを、移植可能なJVMバイトコード（JVM用の「仮想マシン命令」）に変換します。静的コンパイラーとは異なり、javacではほとんど最適化を行いません。静的にコンパイルされる言語でコンパイラーが行うような最適化は、プログラムを実行する際にランタイムが行います。

第一世代のJVMは完全にインタープリター型でした。JVMはバイトコードをマシン・コードにコンパイルするのではなく、バイトコードを解釈して、マシン・コードを直接実行しました。この手

法では、システムはプログラムの実行よりもインタープリターの実行に多くの時間を使うので、当然ながら最善のパフォーマンスは得られません。

Just-in-timeコンパイル

インタープリター型は概念の証明程度の実装であれば問題ありませんが、初期のJVMはすぐに、「遅い」という悪評が立ったのです。次の世代のJVMは、just-in-time (JIT) コンパイラーを使って実行スピードを上げました。厳密に定義すると、JITベースの仮想マシンは実行前に全バイトコードをマシン・コードに変換することになっているのですが、実際には少し怠けた手を使います。JITは、そのコード・パスが今まさに実行されると分かった時にのみコード・パスをコンパイルするのです（そのためjust-in-timeコンパイルという名前なのです）。この手法では実行前に長々としたコンパイル・フェーズが必要無いので、プログラムはずっと速く起動するようになります。

JITの手法は有望に見えたのですが、幾つかの欠点がありました。JITコンパイルは（いくらか余分に起動コストがかかるという犠牲の下で）解釈のためのオーバーヘッドは取り除いたのですが、コード最適化のレベルは幾つかの理由から中途半端なものでした。Javaアプリケーションの起動時間が長くなりすぎないように、JITコンパイラーは速くなければなりません。これはつまり、最適化のためにあまり時間をかけられないということです。また初期のJITコンパイラーでは、後からどんなクラスがロードされるのか分からないので、インライン化の条件を控えめにしていました。

本来JITベースの仮想マシンは実行前に各バイトコードをコンパイルするのですが、JITという用語はしばしば、バイトコードをマシン・コードに動的コンパイルするもの全てに対して、つまりバイトコードを解釈するものまでを含めて使われてしまっています。

HotSpot動的コンパイル

HotSpot実行プロセスには、解釈とプロファイリング、そして動的コンパイルが組み合わされています。実行前に全バイトコードをマシン・コードに変換する代わりに、HotSpotはまずインタープリターとして実行し、「ホットな」コード（つまり最も頻繁に実行されるコード）のみをコンパイルするのです。HotSpotは実行時にプロファイル・データを集め、コンパイルを行うに値するほど最も頻繁に実行されるコード部分はどこかかを判断します。頻繁に実行されるコードのみをコンパイルするのには幾つかパフォーマンス上の利点があります。あまり頻繁に実行されないコードをコンパイルすることによる時間の浪費がなく、コンパイラーはより多くの時間をホット・コード・パスの最適化に使えるようになります。さらにコンパイラーはコンパイルを遅らせることによってプロファイル・データにアクセスでき、ある特定なメソッド・コールをインライン化すべきか否かといった最適化判断の改善に、そのデータを使えるのです。

複雑なことに、HotSpotにはクライアント・コンパイラーとサーバー・コンパイラーという、2つのコンパイラーがあります。デフォルトではクライアント・コンパイラーを使うようになっていますが、JVMを起動する時に`-server`スイッチを規定することで、サーバー・コンパイラーを選択することができます。サーバー・コンパイラーは最大演算速度を得られるように最適化されており、長期間実行するサーバー・アプリケーションを対象にしています。クライアント・コンパイラーはアプリケーションの起動時間とメモリー消費を削減するように最適化されており、複雑な最適化はサーバー・コンパイラーほど数多く使っていないため、コンパイルに要する時間も短くなっています。

HotSpotサーバー・コンパイラーは驚くほど多彩な最適化を行い、静的コンパイラーにあるような標準的最適化の多くを行うことができます。例えばコード最適化（code hoisting）、共通部分式の削除（subexpression elimination）、ループ・アンローリング（loop unrolling）、範囲チェック削除（range check elimination）、デッドコードの削除（dead-code elimination）、データ・フロー解析などです。さらに静的にコンパイルされる言語では実際的ではない最適化、例えば仮想的メソッド呼び出しの積極的なインライン化（aggressive inlining）なども行うことができます。

連続再コンパイル

HotSpotによる手法でもう一つ面白いのは、コンパイルが必ずしも「するかしないか」ではないことです。ある回数だけコード・パスを解釈した後、マシン・コードにコンパイルされますが、JVMはプロファイリングを続けます。そしてそのコード・パスが特にホットであるとか、あるいは将来のプロファイル・データがさらなる最適化の可能性を示していると判断すると、より高レベルの最適化を使って後からコードを再コンパイルする場合もあるのです。JVMは一回のアプリケーション実行で、同じバイトコードを何度も再コンパイルするかも知れません。コンパイラーが何をしているのか、ちょっと見るためには、`-XX:+PrintCompilation`フラグでJVMを呼び出してみます。このフラグでコンパイラーは（クライアントであれサーバーであれ）実行の都度、短いメッセージを出力するようになります。

On-stack replacement

HotSpotの最初のバージョンでは、一度に一つのメソッドをコンパイルしました。あるメソッドが、ある回数（最初のHotSpotでは10,000回）以上累積してループ繰り返しを実行されると、そのメソッドはホットであると判断されました。これは各メソッドにカウンターを関連付け、後方分岐(backward branch)が取られる度にカウンターを増加することで回数を判断していました。ところが、そのメソッドが終了してから再度使われるまで、そのメソッドはコンパイルされても（コンパイルされたものには）切り替わらず、コンパイルされたものはその後に続く呼び出しに対してのみ使われるのです。その結果、例えば計算を主とするプロフラムなど、あるメソッドを一度呼び出すだけで全ての計算が行われるような場合には、コンパイルされたものは使われることがありません。そうした場合には、重量級のメソッドがコンパイルされるかも知れませんが、コンパイルされたコードは結局使われないのです。

最近のバージョンのHotSpotでは、ループの最中にインタープリターからコンパイルしたコードに切り替われるように（あるいは、別々にコンパイルされたもの同士を交換できるように）on-stack replacement (OSR) と呼ばれる手法を使っています。

ではこれが、ベンチマークとどう関係するのか？

この記事はベンチマークとパフォーマンス測定に関するものだと皆さんに約束したのですが、ここまでは歴史的な話しとSunによるHotSpotホワイトペーパーの蒸し返しでした。長い寄り道をしてきた理由は、動的コンパイルのプロセスを理解しない限り、Javaクラス用のパフォーマンス・テストを正しく書いたり解釈したりすることは不可能だからです。（動的コンパイルとJVM最適化を充分に理解していても、やはり非常に難しいのです。）

Javaコード用のマイクロベンチマークは、Cコード用よりもずっと書くのが難しい

手法Aが手法Bよりも速いかどうかを判断する伝統的な方法は、小さなベンチマーク・プログラム、よくマイクロベンチマークと呼ばれるプログラムを書いてみることです。そうしたくなるのは自然なことです。科学的方法で必要なのは、個別の調査なのですから。ところが、詳細が問題

なのです。動的にコンパイルされる言語用にベンチマークを書くのは（そしてそれを解釈するのは）、静的にコンパイルされる言語の場合よりも、はるかに難しいのです。ある構成体のパフォーマンスに関して何かを知るために、その構成体を使ったプログラムを書くのは何も悪いことではありません。ところがJavaで書かれたマイクロベンチマークは多くの場合、期待したものを教えてくれないのです。

Cプログラムの場合であれば、そのパフォーマンス特性がどの程度かはプログラムを実行しなくても、コンパイルされたマシン・コードを単純に見るだけで知ることができます。コンパイラーが生成する命令は実行されるべき実際のマシン命令であり、そのタイミング特性は普通の場合、ほぼ理解できるのです。（もちろん、分岐予測ミスやキャッシュ・ミスが繰り返し起こることによって、マシン・コードを見て期待されるよりも実際のパフォーマンスがずっと悪いという病理的な例もあります。しかし大部分の場合では、マシン・コードを見ることによってCプログラムのパフォーマンスに関する多くを知ることができるのです。）

コンパイラーが、あるコード・ブロックを無関係と判断して最適化し、除去すると（実際には何も行わないベンチマークではありがちです）、生成されたマシン・コードでそれが分かります。そのコードはそこにありません。そして普通はCコードを少し実行するだけで、そのパフォーマンスについてある程度妥当な推測ができるものです。

一方HotSpot JITでは、プログラムの実行に合わせてJavaバイトコードをマシン・コードに再コンパイルし続けます。そして再コンパイルはプロファイル・データが一定量累積されたことにより、また新しいクラスのロードにより、あるいは既にロードされたクラスではトラバースされていなかったコード・パスを実行することによって、予期せぬ時にトリガーされます。連続的再コンパイルが行われる中での時間測定は極めて誤差が多く、誤った印象を与えがちであり、有用なパフォーマンス・データを得るためには、非常に長時間Javaコードを実行せざるを得ない場合が多いのです。（私はプログラムが実行し始めてから何時間も、時には何日もかかったという例を見ています。）

デッドコード削除（Dead-code elimination）

良いベンチマークを書く上で難しいのは、コンパイラーを最適化すると、デッドコード（プログラム実行の結果には何の影響もないコード）を拾ってしまいがちなのです。ところがベンチマーク・プログラムは何ら出力を生成しない場合が多いので、一部の、あるいは全てのコードが、気付かないうちに最適化によって除去されてしまう可能性があります。そうなると実際に測っている時間は、自分で想定している条件での時間よりも短いことになります。特に多くのマイクロベンチマークでは、`-client`で実行した場合よりも`-server`で実行した場合の方が、パフォーマンスが「良く」なります。これはサーバー・コンパイラーが速いためではなく（実際に速い場合が多いのですが）、デッドコード・ブロックを除去して最適化するためです。残念ながら、デッドコード最適化はベンチマークがお粗末なだけでなく（場合によると最適化で全てを除去してしまうかもしれません）、実際に何かを行うコードに対しても、あまりうまく動作しません。

怪しげな結果

リスト1は何もしないコード・ブロックを含んでいます。これは並行スレッドのパフォーマンスを測定するために作られたベンチマークから取ったものですが、実際には全く違うものを測定しています。（この例はJavaOne 2003で発表された素晴らしいプレゼンテーション、「The Black Art of Benchmarking」から借用したものです。これについては参考文献を見てください。）

リスト1. デッドコードと見なされたことによって歪曲されたベンチマーク

```
public class StupidThreadTest {
    public static void doSomeStuff() {
        double uselessSum = 0;
        for (int i=0; i<1000; i++) {
            for (int j=0;j<1000; j++) {
                uselessSum += (double) i + (double) j;
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        doSomeStuff();
        int nThreads = Integer.parseInt(args[0]);
        Thread[] threads = new Thread[nThreads];
        for (int i=0; i<nThreads; i++)
            threads[i] = new Thread(new Runnable() {
                public void run() { doSomeStuff(); }
            });
        long start = System.currentTimeMillis();
        for (int i = 0; i < threads.length; i++)
            threads[i].start();
        for (int i = 0; i < threads.length; i++)
            threads[i].join();
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end-start) + "ms");
    }
}
```

`doSomeStuff()`メソッドはスレッドに対して何かすべきことを与えるはずなので、`StupidThreadBenchmark`を実行すると、複数スレッドのスケジューリングのオーバーヘッドが何かしらあるだろうと推測できます。ところがコンパイラーは`uselessSum`が全く使われないので、`doSomeStuff`のコードは全てデッドコードであると判断し、全てを除去して最適化してしまいます。ループの中身が無くなってしまうと、そのループも無くなり、`doSomeStuff`は完全に空になって残ります。テーブル1はクライアントとサーバーでの、`StupidThreadBenchmark`のパフォーマンスを示します。どちらのJVMも大まかに言って、スレッド数の増加に対して直線的に実行時間も増えています。このためサーバーJVMはクライアントJVMよりも40倍速いのだ、と誤って理解しても不思議はありません。なぜこうなるかと言えば、サーバー・コンパイラーはより多くの最適化を行い、`doSomeStuff`全体をデッドコードとして検出するためです。多くのプログラムがサーバーJVMで高速化できるのは事実ですが、ここでは単にお粗末に書かれたベンチマークの実例を見ているに過ぎず、サーバーJVMによる高速化を見ているわけではありません。ただし注意しないと、それを誤解してしまいがちです。

表1. クライアントJVMとサーバーJVMでのStupidThreadBenchmarkのパフォーマンスの違い

スレッド数	クライアントJVM実行回数	サーバーJVM実行回数
10	43	2
100	435	10
1000	4142	80
10000	42402	1060

デッドコードを除去しすぎるのは、静的にコンパイルされる言語のベンチマークでも問題になります。ただし静的にコンパイルされる言語では、ベンチマークの大きな塊をコンパイラーが除去

してしまったことを検出するのは、ずっと簡単です。生成されたコードを見れば、プログラムの中の大きな塊が無くなっていることが分かります。動的にコンパイルされる言語では、その情報は簡単には分かりません。

ウォームアップ

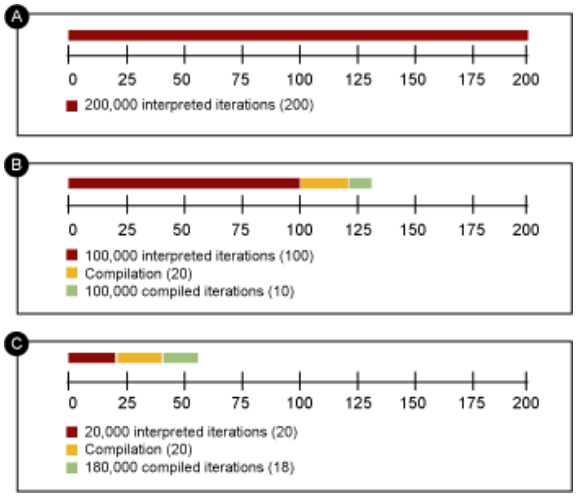
皆さんがイディオムXのパフォーマンスを特定しようとする時には普通、インタープリターを実行した場合のパフォーマンスではなく、コンパイルした場合のパフォーマンスを測ろうとするでしょう（つまり実際の現場でXがどのくらい速いかを知りたいわけです）。そのためにはJVMを「ウォームアップ」する必要があります。つまり実行時間の測定を始める前に、充分な回数だけ対象とする操作を行うのです。そうすることによって解釈したコードをコンパイラーが実行し、それをコンパイルしたコードで置き換えるだけの時間が取れるようにするのです。

on-stack replacementの無い、初期のJITや動的コンパイラーでは、あるメソッドをコンパイルした場合のパフォーマンスを測定するための簡単な公式がありました。そのメソッドの呼び出しを一定回数行い、その後でさらに一定回数だけ実行するのです。ウォームアップ呼び出しの回数がしきい値を超えれば、その時点でメソッドはコンパイルされており、実際に測定された呼び出しは全て、コンパイルしたコードの呼び出しを測定したもののはずです。こうしてコンパイルのオーバーヘッドはすべて、測定を始める前に吸収されるのです。

今日の動的コンパイラーでは、ずっと難しくなっています。コンパイラーの実行時間はより予測が難しく、JVMはいつでもインタープリターを使ったコードからコンパイルしたコードに切り替わり、実行中に同じコード・パスが何度も再コンパイルされる可能性があります。こうしたイベントに関わる時間を考慮に入れないと、測定結果が大きく歪曲されてしまうことになります。

図1は、予期せぬ動的コンパイルによって測定結果がどのように歪曲されるかを示しています。例えば200,000回繰り返すループを測定する場合で、コンパイルしたコードはインタープリターを使ったコードよりも10倍速い、としましょう。もし200,000回の繰り返しでコンパイルが始まるとすると、測定した時間はインタープリターでコードを実行した時間です（測定値A）。もし100,000回の繰り返しでコンパイルが始まるとすると、合計実行時間は200,000回インタープリターで繰り返した時間、加えてコンパイル時間（含めたくはありませんが）、さらに加えてコンパイラーでの100,000回の繰り返し時間になります（測定値B）。もし20,000回の繰り返しでコンパイルが始まるとすると、合計実行時間はインタープリターでの20,000回の繰り返し、加えてコンパイル時間、さらに加えてコンパイラーでの180,000回の繰り返し時間になります（測定値C）。コンパイラーがいつ、どのくらい長く実行するかは分からないので、測定結果がどれほど歪んだものになるか分かるでしょう。コンパイル時間と、コンパイルしたコードがインタープリターを使ったコードよりもどのくらい速いかにより、繰り返し回数を少し変えただけで、測定した「パフォーマンス」には大きな差が出てしまうのです。

図1. 動的コンパイルのタイミングによって歪曲されるパフォーマンス測定



ではどれくらいのウォームアップならば充分なのでしょう。これは分かるものではありません。せいぜいできることは、`-XX:+PrintCompilation`でベンチマークを実行して、何によってコンパイラーが始まるのかを観察してからベンチマーク・プログラムを再構成することです。そして全てのコンパイルは測定前に起こるようにし、それ以上のコンパイルがループ測定の最中に起こらないようにするのです。

ガーベジ・コレクションを忘れずに

つまり正確な測定結果を得たいのであれば、JVMのウォームアップに必要と思われる時間以上に、対象のコードを実行する必要があるということです。一方、テスト・コードが何らかのオブジェクト・アロケーションを行うとすると（ほとんど全てのコードでは行います）、ガーベジが生成されるため、やがてガーベジ・コレクターを実行する必要が出てきます。これも測定結果を大きく歪める要因です。繰り返し回数を少し変えるだけで、ガーベジ・コレクションが無い場合と一回行う場合とで差が出ることになり、「繰り返し毎の時間」の測定を歪曲することになります。

ベンチマークを`-verbose:gc`で実行すれば、ガーベジ・コレクションに使われる時間を知ることができ、それによって測定結果を調整することができます。もっと良いのは、プログラムをとにかく非常に長期間実行することです。充分多くのガーベジ・コレクションをトリガーするようにして、アロケーションとガーベジ・コレクションのコストを、より正確に平均化して考慮するのです。

ダイナミック・デオプティマイゼーション（dynamic deoptimization）

標準的な最適化の多くは「基本ブロック」内でしか実行できません。ですから適切な最適化のためには、メソッド・コールをインライン化することが重要になります。メソッド・コールをインライン化することによってメソッド・コールのオーバーヘッドが無くなるだけでなく、オプティマイザーが扱う基本ブロックがより大きくなり、デッドコード最適化にも非常に有利になります。

リスト2はインライン化で使用可能となる最適化のタイプの例です。`outer()`メソッドは引き数`null`で`inner()`を呼び、その結果として`inner()`は何もしません。ところがこのコールを`inner()`にインライン化すると、コンパイラーは`inner()`の`else`分岐は死んだコードとして見る

ので、テストを最適化して`else`分岐を除去します。この時点で`inner()` へのコール全体が最適化で取り去られます。`inner()`がインライン化されていないと、最適化は不可能です。

リスト2. インライン化によって、デッドコード最適化がいかにうまく行くか

```
public class Inline {
  public final void inner(String s) {
    if (s == null)
      return;
    else {
      // do something really complicated
    }
  }
  public void outer() {
    String s=null; inner(s);
  }
}
```

不便なことに、仮想ファンクションがインライン化の障害になります。そしてJavaでは（C++と比べて）、仮想ファンクション・コールは一般的なのです。例えばコンパイラーが次のコードで`doSomething()` へのコールを最適化しようとしているとしましょう。

```
  Foo foo = getFoo();
  foo.doSomething();
```

どちらのバージョンの`doSomething()` が実行されるか（クラス`Foo`で実装される方か`Foo`のサブクラスで実装される方か）は、このコード断片からはコンパイラーには確実に判断できません。ただし、答えが明白な場合もあります。例えば`Foo`が`final`の場合や、`doSomething()` が`Foo`の`final`メソッドとして定義されている場合などです。しかし大部分の場合、コンパイラーは適当に判断せざるを得ません。一度に一つのクラスをコンパイルする静的コンパイラーではお手上げです。ところが動的コンパイラーはグローバル情報を使って、より的確な判断ができます。例えばこのアプリケーションの`Foo`を拡張するクラスが何もロードされていないとしましょう。こうなると`doSomething()` が`Foo`での`final`メソッドの場合と同じようになります。コンパイラーは仮想メソッド・コールをダイレクト・ディパッチ（direct dispatch）に変換し（これが既に改善になっています）、そしてさらに`doSomething()` をインライン化するという選択肢もあるのです。（仮想メソッド・コールを直接メソッド・コールに変換することを単一型コール変換（monomorphic call transformation）と呼びます。）

ちょっと待ってください。クラスは動的にロードされるのです。もしコンパイラーがそんな最適化を行い、後で`Foo`を拡張するクラスがロードされたらどうなるのでしょう。もっと悪い場合として、これがファクトリー・メソッド`getFoo()`内部で行われ、次に`getFoo()` が新しい`Foo`サブクラスのインスタンスを返したとしたらどうでしょう。生成されるコードが不正にならないのでしょうか。いや、不正になってしまうでしょう。しかしJVMはこれを判断して、その時点では既に無効となった想定に基づいて生成されたコードを無効にし、インタープリターに戻るのです（または無効となったコード・パスを再コンパイルします）。

つまりコンパイラーは積極的にインライン化の判断をして高パフォーマンスを達成し、もし後で、こうした判断が有効な想定に基づくものでなくなっていたら、その判断を取り消すのです。実際この最適化は非常に効果的なので、オーバーライドされていないメソッドに`final`キーワード

を追加しても（初期の文献で提唱されたパフォーマンス向上の細工です）、実際のパフォーマンス向上にはほとんど影響を与えません。

怪しげな結果

リスト3は不適切なウォームアップや単一型コール変換、デオプティマイゼーション(deoptimization)を組み合わせたコード・パターンを含んでおり、全く無意味な、しかし誤って判断しがちな結果を生じます。

リスト3. 単一型コール変換とそれに続くデオプティマイゼーション(deoptimization)によって、テストプログラムの結果が歪曲される

```
public class StupidMathTest {
    public interface Operator {
        public double operate(double d);
    }
    public static class SimpleAdder implements Operator {
        public double operate(double d) {
            return d + 1.0;
        }
    }
    public static class DoubleAdder implements Operator {
        public double operate(double d) {
            return d + 0.5 + 0.5;
        }
    }
    public static class RoundaboutAdder implements Operator {
        public double operate(double d) {
            return d + 2.0 - 1.0;
        }
    }
    public static void runABunch(Operator op) {
        long start = System.currentTimeMillis();
        double d = 0.0;
        for (int i = 0; i < 5000000; i++)
            d = op.operate(d);
        long end = System.currentTimeMillis();
        System.out.println("Time: " + (end-start) + " ignore:" + d);
    }
    public static void main(String[] args) {
        Operator ra = new RoundaboutAdder();
        runABunch(ra); // misguided warmup attempt
        runABunch(ra);
        Operator sa = new SimpleAdder();
        Operator da = new DoubleAdder();
        runABunch(sa);
        runABunch(da);
    }
}
```

StupidMathTestはまず少しウォームアップをしようとします（成功はしません）。次にSimpleAdderとDoubleAdder、RoundaboutAdderの実行時間を測定します。この結果が表2です。直接1を加えるよりも、doubleに2を加えることで1を加えてから1を引いた方がずっと速いように見える結果となっています。そして1を加えるよりも0.5を2回加える方が、わずかに速くなっています。そんなことがあり得るでしょうか。（答えは当然、ノーです。）

表2. 無意味な、しかし誤って判断しがちなStupidMathTestの結果

メソッド	実行時間

SimpleAdder	88ms
DoubleAdder	76ms
RoundaboutAdder	14ms

何が起きたのでしょうか。つまりウォームアップ・ループの後、`RoundaboutAdder`と`runABunch()`がコンパイルされ、コンパイラーが`Operator`と`RoundaboutAdder`に対して単一型コール変換を行ったので、最初のパスは非常に速く実行したのです。ところが2回目のパス（`SimpleAdder`）では、コンパイラーはデオプティマイズ(deoptimize)を行って仮想メソッド・ディスパッチに戻らざるを得ず、その結果仮想ファンクション・コールを除去する最適化ができず、また再コンパイルの時間のために、実行が遅くなっています。3回目のパス（`DoubleAdder`）では再コンパイルが少ないので、実行が速くなっています。（実際にはコンパイラーは、`RoundaboutAdder`と`DoubleAdder`に対して定数の畳み込み(constant folding)を行い、`SimpleAdder`と全く同じコードを生成します。ですからもし実行時間に差があるとすると、それは算術コードによるものではありません。）つまり最初に実行するものが一番速くなるのです。

ではこの「ベンチマーク」から何が結論できるのでしょう。実質的には何もありません。動的にコンパイルされる言語のベンチマークは予想以上に微妙だ、ということが分かるだけです。

まとめ

この記事の例で示した結果は明らかにおかしいので、何か他のことが起きているに違いないことは明確に分かります。しかし皆さんが「何か変なことが起きているに違いない」と気が付かないところで、些細なことがパフォーマンス・テスト・プログラムの結果に大きく影響するのです。この記事で取り上げた例は誤ったマイクロベンチマークとして一般的な例ですが、他にも似た例はまだたくさんあるのです。教訓は「今測っているものは、あなたが思っているものではないかもしれない」ということです。実際、思いもかけないものを測っていることがよくあるのです。パフォーマンスの測定であっても、長期間に渡って現実的なプログラムをロードして実行するものでなければ、その結果を疑ってみる心がけが必要と言えるでしょう。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。