Javaの理論と実践: パフォーマンスの都市伝説

ガーベッジ・コレクターなどのプログラミングに棲みついているワニについて

Brian Goetz
Principal Consultant
Quiotix

2003年 4月 22日

都市伝説（urban legend）は人の心をとらえるウィルスのようなものです。多くの場合、私たちはそれが真実ではないらしいとわかっていながら、あまりにも興味深い話であるために誰かに伝えずにはいられなくなり、次々と他の「宿主」に感染していきます。ほとんどの都市伝説はある程度事実に基づいていますが、それがために根絶することが難しくなっています。残念なことに、Javaのパフォーマンス・チューニングに関する多くの指針やヒントは、この都市伝説にそっくりです。誰かが、どこかで、何らかの事実に基づく (または基づいていた)「ヒント」を誰かに伝えるのですが、語り継がれるうちに、かつて存在していた真実の部分がいつの間にか失われているのです。今月の記事では、Brian Goetz氏がパフォーマンスにまつわるこのような都市伝説をいくつか検証し、その真相を明らかにします。

このシリーズの他の記事を見る

皆さんは、雨でずぶ濡れになった愛犬を電子レンジで乾かそうとした老婦人の話 (作り話ですが) を聞いたことはありませんか？ この他にも、航空母艦の指揮官が灯台に優先通行権を譲るように主張した話 (これも作り話です)、ワニがニューヨークの下水管に住むようになったいきさつ (同じく作り話)、郵便配達車両は連邦に属しているため、パトカー、消防車、救急車などの地域の緊急車両に対して優先通行権を持つという噂(これまた作り話) なども聞いたことがあるのではないでしょうか。それでは、正直に答えてください。このような話が作り話らしいと思いながら、あるいはもっとはっきりと疑いの気持ちを抱きながらも、これまでに何回このような話をそのまま、または形を変えて他人に伝えたことがありますか？

都市伝説は、非常にもっともらしく思わせる何かを持っているために、さらに語り継がれることになり、消えることはありません。そして、残念なことに、都市伝説は、赤ちゃんワニをトイレに流したことが原因だ、などという話だけにはとどまりません。プログラマーの間でも、Javaプログラムのパフォーマンスを向上または悪化させる要因などについて、多くの間違ったアドバイスが流れており、その多くはワニの話と同じくらい科学的な正確性を欠いています。しかし、このような話は十分にもっともらしく聞こえるため、さらに語り継がれ、聞き手のほとんどはわざわざその理論に疑問を持ったり、実験的に検証しようとはしません。

© Copyright IBM Corporation 2003

今月は、Javaのパフォーマンス・チューニングに関して広く語られている、都市伝説と同様の特徴を多く持った説をいくつか取り上げて検討します。このような説の中には、ある程度事実に基づくものもありますが、どれも不合理に広まり、正統なパフォーマンス向上手段と持てはやされるに至ったものです。

都市伝説その1: 同期化によって処理は目に見えて遅くなる

本当か嘘か : 同期化されたメソッドは、同期化されていない同等のメソッドより動作が50倍遅い。この珠玉の説は、低水準のパフォーマンス・チューニングに関するDov Bulka氏の優れた書籍の中で述べられているもので、他の文献でも繰り返し引用されています。すべての都市伝説がそうであるように、この説もある程度事実に基づいています。Bulka氏に公正を期して言えば、かなり昔、つまりJDK 1.0が使用されていた頃にリスト1 のようなマイクロベンチマークを実行したとしたら、`testSync` の実行には`testUnsync` の50倍の時間がかかるという結果が出たかもしれません。

しかし、それがかつては真実だったとしても、今はそうではありません。JVMはJDK 1.0以降大幅に改良されています。同期はより効率的に実装されているうえ、何のデータ保護にもならない同期であれば、同期を使用しないようにJVM側で判断できる場合もあります。しかし、それ以上に重要なのは、リスト1 のようなマイクロベンチマークには、根本的な欠陥があるという点です。なにより、マイクロベンチマークがユーザーが思い描くとおりのものを測定していることはまずありません。動的コンパイルの下では、JVMがどのバイトコードをネイティブ・コードに変換するか、そしてそれがいつ実行されるか知ることはできないため、完全に同じ条件でパフォーマンスを比較することはできません。

また、コンパイラーやJVMが、最適化によって何を排除するかもわかりません。たとえば、実際には何も実行しないという理由で、`unsyncMethod` の呼び出しをまったく実行しないように最適化するJavaコンパイラーもあれば、同様の理由で`syncMethod` 自体や`syncMethod` の呼び出し時の同期処理をまったく実行しないように最適化するものもあります。皆さんがお使いのコンパイラーは、このどちらを、どのような状況で最適化していますか？おそらく、それはわからないと思います。これによってまず間違いなく測定は実情とかけ離れたものになってしまいます。

実際の呼び出し回数を無視して、この種のベンチマークの結果から、同期化されていないメソッドの呼び出しが同期化されたメソッドの呼び出しの何倍速いなどと結論付けるのは、非常に馬鹿げています。同期処理がコードのブロックに一定のオーバーヘッドをかけることは考えられますが、コードの実行速度を一定の比率で遅くすることはなさそうです。そうなると、ブロック内にどれだけ多くのコードが存在するかがリスト1で算出される「比率」に大きく影響することになります。そして、同期によって発生するオーバーヘッドが、空のメソッドを実行する時間に対して占める割合はほとんど取るに足らない数値と言えます。

現在のJVMで、実行時に本当に実行される同期化されたメソッドと同期化されていないメソッドを比較すれば、そのオーバーヘッドがどうやっても噂の「50倍」などという人騒がせな数字にはならないことがわかります。このシリーズ第1回の記事、「 システム負荷を軽減したスレッド化: 同期化を敵視することはありません」(参考文献を参照) もご覧下さい。同期化のオーバーヘッドの測定が大まかで非科学的であることを示す例がいくつか紹介されています。念のためにつけ加えると、競合のない同期化にも多少のオーバーヘッドはあります (競合のある同期化の場合はなお

さらです)。しかし、同期化は、多くの人が恐れているような、下水管に棲みついてパフォーマンスを食い尽くすワニではありません。

リスト1. 同期化のオーバーヘッドを測定する欠陥のあるマイクロベンチマーク

```
    public static final int N_ITERATIONS = 10000000;
    public static synchronized void syncMethod() {
    }
    public static void unsyncMethod() {
    }
    public static void testSync() {
        for (int i=0; i<N_ITERATIONS; i++)
            syncMethod();
    }
    public static void testUnsync() {
        for (int i=0; i<N_ITERATIONS; i++)
            unsyncMethod();
    }
    public static void main(String[] args) {
        long tStart, tElapsed;
        tStart = System.currentTimeMillis();
        testSync();
        tElapsed = System.currentTimeMillis() - tStart;
        System.out.println("Synchronized took " + tElapsed + " ms");
        tStart = System.currentTimeMillis();
        testUnsync();
        tElapsed = System.currentTimeMillis() - tStart;
        System.out.println("Unsynchronized took " + tElapsed + " ms");
    }
```

「同期化は処理を遅くする」という神話は非常に危険です。それは、プログラマーがパフォーマンスの悪化を恐れて同期化を回避して、プログラムのスレッド・セーフティーを危険にさらす動機となるからです。実際、多くの場合、プログラマーは同期化を回避することが非常に賢明だと考えます。開発者やコード作成者を、一見すると巧妙であっても致命的な欠陥を持つ "double-checked locking" イディオムの使用に駆り立てたのは、この神話への恐怖に他なりません。このイディオムは、共通コード・パスから同期化を排除するように見えて、実際にはコードのスレッド・セーフティーを損なう可能性のあるものです。スレッド・セーフティーの問題は、コード内で爆発を待つ時限爆弾のようなもので、爆発が起きるのは最悪のタイミング、つまり、プログラムに重い負荷がかかっているときです。正当な理由のあるパフォーマンス上の懸念であっても、スレッド・セーフティーを危険にさらす理由にはなりません。それがパフォーマンス神話を恐れてのことであれば、なおさら何の理由にもなりません。

都市伝説その2: クラスやメソッドをfinalとして宣言すると処理が速くなる

この神話については、10月のコラム (参考文献を参照) で紹介済みなので、ここで詳しい説明は繰り返しません。多くの記事では、クラスやメソッドを`final` として宣言することが推奨されています。これは、finalにすることにより、コンパイラーがそのクラスやメソッドを簡単にインライン化できるため、パフォーマンスが向上するはずだという理由によります。これは確かによくできた理論です。しかし、真実でないのは残念としか言いようがありません。

同期化の神話と比べてこの神話が面白いのは、裏付けのデータが何もなく、単にもっともらしく見えるだけであるという点です (同期化の神話には、欠陥があるとは言え、少なくともそれを裏付

けるマイクロベンチマークがあります)。誰かがきっとこうに違いないと決め付け、自信たっぷりに他人に話したところ、それが噂になり、大きく広がってしまったに違いありません。

この神話の危険なところは、同期化の神話とまったく同様に、実際には存在してもいないパフォーマンス上のメリットを求めて、開発者がオブジェクト指向の優れた設計原則を破ることになりかねない点です。クラスを`final`にするかどうかは、そのクラスが何を実行し、誰によってどのように使用されるか、および何らかの用途で継承される可能性があるかどうかに基づいて決定される設計上の判断です。クラスが不変なので`final`にするのであれば、それは適切な理由と言えます。また、継承を想定せずに設計された複雑なクラスを`final`にすることも理にかなっています。しかし、クラスを`final`にすると速くなるとどこかに書いてあったというだけでは、たとえそれが真実であってもそうする理由にはなりません。

都市伝説その3: 不変オブジェクトはパフォーマンスを悪化させる

不変オブジェクト (`String`など) を使用して変化するデータを表現し、データが変更されたときにはオブジェクトの状態を変更せずに新しいオブジェクトを生成するという方法は、一般的に使用されています。不変オブジェクトと可変オブジェクトの間には、パフォーマンスに関して複雑なトレードオフがあります。プログラム内でのオブジェクトの使用方法によっては、実際には不変オブジェクトを使用したほうがパフォーマンス的にメリットがある場合もあります (オブジェクトを防御的にコピーする必要がないため)。また、反対に、不変オブジェクトの使用がパフォーマンス上の大きなデメリットとなる場合もあります (頻繁に変更されるデータをモデル化しているために、新しいオブジェクトを多数生成することになる場合など)。そして、ここでもパフォーマンス上の違いを計測できない場合があります。

「不変オブジェクトを使用すると遅くなる」という神話は、一時オブジェクトを多数生成するとパフォーマンスに悪影響を与えるという、より一般的なパフォーマンス原則に基づいたものです。一時オブジェクトの生成は、確かにアロケーターとガーベッジ・コレクターに余分な作業を課します。しかし、現在のJVMは、一時オブジェクトの生成によるパフォーマンスへの影響を軽減するように改良されています。オブジェクトの生成がパフォーマンスに与える影響は確かに存在しますが、ほとんどのプログラムでは、以前ほど、または今も信じられているほど、その影響が大きいわけではありません。

リスト2に示すような不変と可変の各`StringHolder`クラスの違いについて考えてみましょう。一方では、オブジェクトに含まれる文字列を変更したければ、`StringHolder`の新しいインスタンスを作成することになります。これに対してもう一方では、既存の`StringHolder`の設定メソッドを呼び出して、オブジェクトに含まれる文字列を変更することになります。より具体的な例として、文字列を区切り文字で囲んでいたとしましょう。この場合、2つの方法でパフォーマンスはどのくらい違うのでしょうか。

リスト2. 不変のStringHolderクラスと可変のStringHolderクラス

```
// mutable stringHolder.setString("/" + stringHolder.getString() + "/"); // immutable
stringHolder = new StringHolder("/" + stringHolder.getString() + "/");
```

もしも皆さんが、余分な`StringHolder`オブジェクトの生成によって実質的なパフォーマンスに大きな違いが出ると考えているのなら、それは誤りです。可変オブジェクトを使用した方法でも、多数のオブジェクトが生成されます。文字列の連結を実行する場合は、`StringBuffer`オブジェク

トが生成されますが、これには`char`配列の作成が伴います。次に、最終的な文字配列を表すために`String`オブジェクトが生成されます。連結結果の文字列が、`StringBuffer`によって使用されるデフォルトのバッファー・サイズより大きい場合は、内部文字配列が再び割り当てられるため、さらにオブジェクトが1つ以上作成されることになります。つまり、可変オブジェクトを使用する方法でも、オブジェクトが少なくとも3つ生成されることになります。不変オブジェクトの場合は、生成されるオブジェクトが可変オブジェクトの場合より1つ多いだけです。パフォーマンスの違いが発生する可能性はありますが、オブジェクト生成をまったく行わない場合と多数行う場合ほどの違いとは比較にならないほど小さいものです。

また、重要ではあるものの、測定の難しいガーベッジ・コレクションとの相互作用もパフォーマンスに影響します。現代のガーベッジ・コレクターでは、新しいオブジェクトが古いオブジェクトを参照する場合のパフォーマンスは、その逆の場合よりかなり良くなっています。新しい不変のホルダー・オブジェクトの作成は、まさにこの良い方の状況であり、既存のコンテナー・オブジェクトを変更して新しく作成された文字列を参照することは、その逆です。

最初の2つの神話とまったく同じように、この神話によっても、プログラマーはパフォーマンス上のメリットのために、優れたオブジェクト指向設計の原則を犠牲にしがちです。不変オブジェクトは、可変オブジェクトに比べて単純で、作成、保守、使用でもエラーを招きにくいものです。このようなメリットを、パフォーマンスのために犠牲にすべきでしょうか。確かにそのような場合もあるかもしれません。しかしそれは、パフォーマンスに問題があることが確実であり、その原因が明らかで、ある特定のクラスの不変性を捨てることによってパフォーマンス目標が達成されることがわかっている場合だけです。はっきりとしたパフォーマンス上の問題および一定の目標がない状態では、高いパフォーマンスではなく、プログラムの正確性を重視するべきです。

教訓

ここで取り上げたパフォーマンス伝説には、共通のテーマがいくつか存在します。すべての伝説は、Javaテクノロジーのごく初期の段階で見られたパフォーマンスに関する主張に基づいたもので、その頃にはまだJVMのパフォーマンスを改善するために多大な努力は払われていませんでした。これらの伝説の中には、それが初めて示されたときには正しかったものもありますが、その後、JVMのパフォーマンスは大幅に改善されています。　同期化とオブジェクト生成のパフォーマンスへの影響はもちろん無視すべきではありませんが、それを何よりも優先して回避する必要はありません。

具体的なパフォーマンス目標を念頭に最適化を行う

これらの神話には、すべて同じ危険性が潜んでいます。それは、優れた設計原則を破ったり、さらに悪い場合はプログラムの正確性まで危険にさらして、実在することさえ怪しいパフォーマンス上のメリットを得ようとすることです。最適化は、常にリスクを伴います。そのリスクには、既に機能しているコードの破壊、コードを複雑にしたことによるバグ発生の可能性、コードの普遍性や再利用可能性の制限、制約の誘発などがあるほか、単にコードがわかりにくくなり、保守が難しくなるという点もあります。パフォーマンスにはっきりとした問題が出てくるまでは、わかりやすさ、バグのない設計、および正確性を重視するに越したことはありません。最適化は、パフォーマンスの改善が本当に必要になるまでとっておき、実際に実行するときには、意味のある変化をもたらす最適化を行ってください。

パフォーマンスに関するアドバイスの寿命は短い

どのような技術であれ、パフォーマンスがその技術にもともと備わっているわけではありません。パフォーマンスは、その技術が使用される環境にも左右されるうえ、プログラムの実行環境は常に変化しています。このような変化には、コンパイラーの高性能化、プロセッサーの高速化、ライブラリーの更新、ガーベッジ・コレクションやスケジューリング・アルゴリズムの変更などがあります。また、プロセッサー、キャッシュ、メイン・メモリー、I/Oデバイスの相対的な速度やコストも時間とともに変化します。10年前に技術Aが技術Bの10倍高速であったとしても、現在の両技術が持つパフォーマンスの相対関係を推測する際には、それは参考になりません。パフォーマンスに関する意見は短命なものです。パフォーマンスに関する助言に出くわしたときは、それを事実として受け入れる前に、それが時代遅れではないかという疑問を持ってください。

このように多くのパフォーマンスに関する助言がすぐに古くなってしまうことを考えると、噂に聞くパフォーマンスに関するヒントの有効性にもっと疑問を持ち、実際に動作しているコードに対するヒントの適用にはより保守的になったほうが良さそうです。まず最初に、変更によって本当にアプリケーションのパフォーマンスが向上するのか、そして、そもそもそのアプリケーションにパフォーマンスの改善が必要なのか自問自答してください。

もし、「このメッセージを他の人に転送したら、転送相手1人につき、ビル・ゲイツ氏が10ドル支払います」というe-mailが届いても、どうか私にそのメールを転送したりしないでください。

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。

© Copyright IBM Corporation 2003
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)