確実な Java ベンチマーク: 第 2 回 統計とソリューション

すぐに実行できるソフトウェア・ベンチマーク・フレームワークの紹介

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。2 回連載の第 2 回目では、ベンチマークの結果についての統計について説明し、必要なものを完備したマイクロベンチマークからアプリケーション全体を呼び出すコードに至るまで、Java™ コードのベンチマークに使用できるフレームワークを紹介します。

このシリーズの他の記事を見る

この 2 回連載の第 1 回目では、Java コードのベンチマークに伴う多くの落とし穴を説明しました。今回の第 2 回目の記事では、2 つの異なる分野を説明します。最初に取り上げるのは、ベンチマークで必ず現れてくる測定値の変動に対処するための初歩的な統計です。次に、ベンチマークを実行するためのソフトウェア・フレームワークを紹介し、このフレームワークを一連のサンプル・コードで使用しながら重要なポイントを解説します。

※訳注: 「第 1 回目」のリンク先の記事は、2008-279 として本記事と同時に 7/16 に納品済みです。

統計

実行時間を 1 回測定し、あとはその測定値を使ってさまざまなコードのパフォーマンスを比較できたとしたら簡単な話です。けれども悲しいことに、このような方法はどんなに好まれても、その効力のなさから採用されることはありません。たった 1 回の測定値を信じるには、あまりにもたくさんの変動要因があるからです。第 1 回の記事で、クロックの分解能、JVM の複雑な振る舞い、そして雑音源のような働きをする自動的なリソース再利用について説明しましたが、これらの事項は不規則に、または体系的に偏ったベンチマークの結果をもたらす要因のほんの数例でしかありません。そのうちのいくつかについては、問題を軽減させるための対策があります。問題を十分に理解していれば、デコンボリューション (deconvolution) を行うことさえ可能です (「参考文献」を参照)。けれどもこれらの対策は決して完璧なものではないので、最終的には自分でベンチマークの結果のばらつきに対処しなければなりません。そのための唯一の方法は、何度も測定を行い、統計によって信頼できる結論を出すことです。この部分を無視するなら、危険を覚悟

してください。「計算するのを拒否した者は、無意味なことを言う運命にある (He who refuses to do arithmetic is doomed to talk nonsense)」からです (「参考文献」を参照)。

補足資料

この記事の関連サイトには、完全なサンプル・コードのパッケージと、統計の問題を詳しく説明する補足が記載されています。この関連サイトへのリンクは、「参考文献」を参照してください。

ここで紹介するのは、以下の一般的なパフォーマンスに関する疑問を解決するのにかろうじて十分な統計です。

- タスク A とタスク B の処理速度はどちらが優れているか
- この結論は信頼できるものなのか、あるいは偶然的な測定値だったのか

何回か実行時間を測定した場合、最初に計算することになる統計は、測定値から実行時間の標準的な値を算出した単一の値です (この記事で使われる統計の概念については、「参考文献」に記載したリンクからウィキペディアの定義を参照してください)。このような指標として最も一般的なものは、算術平均で、一般には平均 (mean または average) として知られています。これは、以下のように測定値の合計を測定回数で割った値です。

$$mean_x = Summation_{i=1,n}(x_i) / n$$

平均以外の他の指標については、この記事の補足となる関連 Web サイト (「参考文献」を参照) の資料で説明されています。その補足資料の「Alternatives to the mean」セクションを参照してください。

複数回行った測定の平均でパフォーマンスを定量化すれば、当然、1 回の測定を使用する場合より正確になります。けれども、どちらのタスクのほうが処理速度に優れているかは、それだけでは判断しきれません。例えば、タスク A の実行時間平均は 1 ミリ秒で、タスク B の平均は 1.1 ミリ秒だとします。この場合、素直にタスク A のほうがタスク B より速いという結論を出しますか？もし、タスク A の測定値が 0.9 から 1.5 ミリ秒までのばらつきがあるのに対し、タスク B の測定値のばらつきが 1.09 から 1.1 ミリ秒だということがわかっていれば、そのような結論を出すことはまずありません。したがって、測定値のばらつきにも対処する必要があります。

測定値のばらつき (統計的ばらつき) を表す最も一般的な統計は、標準偏差です。標準偏差は以下のように計算します。

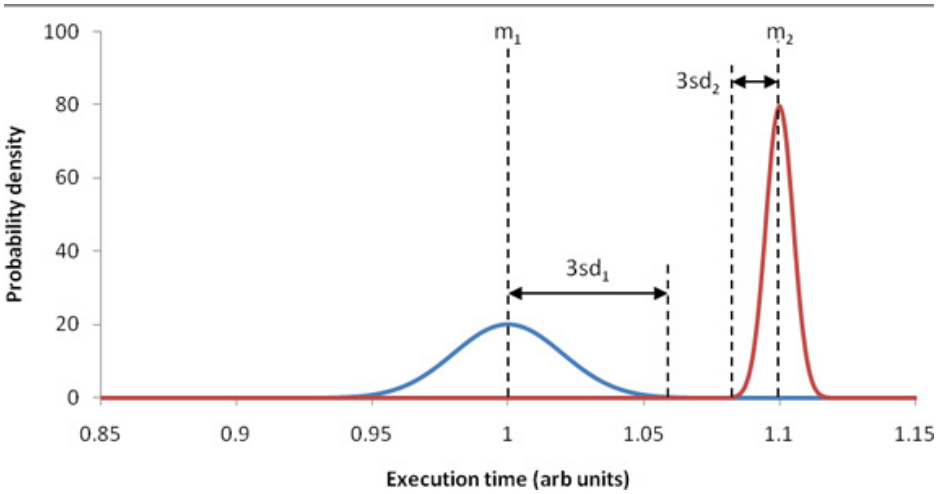
$$sd_x = sqrt\{ Sum_{i=1,n}([x_i - mean_x]^2) / n \}$$

標準偏差がどうやって測定値のばらつきを定量化するかと言えば、その説明は測定値の確率密度関数 (PDF: probability density function) についてどのようなことを知っているかによって変わってきますが、要は想定の信憑性が高ければ高いほど確実な結論が導かれるということです。補足資料の「Relating standard deviation to measurement scatter」セクションでは、この点についてさらに詳しく説明しています。この補足によると、ベンチマークというコンテキストでの妥当な経験則としては、測定値の 95% 以上が平均の 3 標準偏差の範囲内にあるべきだと結論付けています。

それでは、平均と標準偏差をどのように使って、2 つのタスクのどちらの処理速度が優れているかを判断すればよいのでしょうか。上記の経験則を想起させる簡単な例は、2 つのタスクの平均

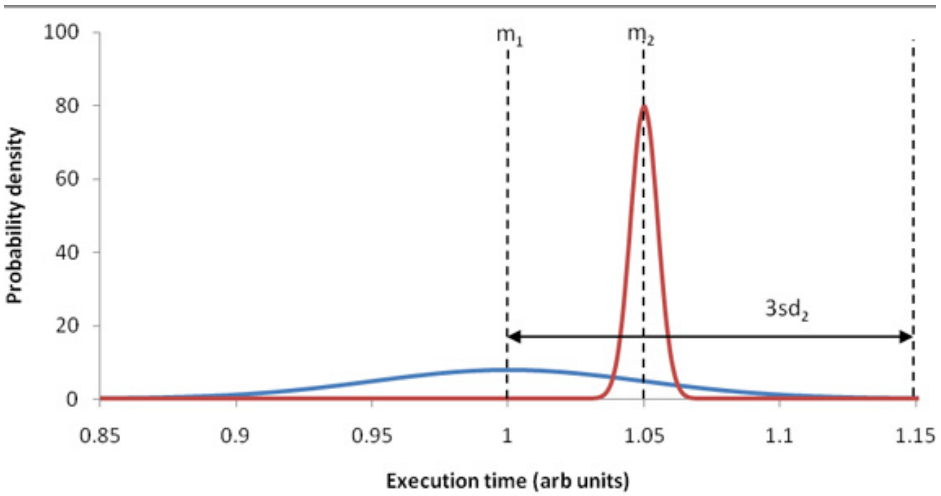
が 3 標準偏差以上 (2 つのうちの大きいほうの標準偏差を選択) 離れている場合です。その場合、平均が小さいほうのタスクがほとんど常に、処理速度に優れています (図 1 を参照)。

図 1. 3 標準偏差以上離れた平均が示唆する明らかなパフォーマンスの違い



残念ながら、2 つのタスクの重なる部分が大きくなるにつれ (2 つのタスクの平均が 1 標準偏差しか離れていない場合など)、判断は難しくなります (図 2 を参照)。

図 2. 3 標準偏差の範囲内にある平均が示唆するパフォーマンスの重複



私たちにできることは、2 つのタスクを平均によって評価することだけかもしれませんが、タスクの重なる部分がどの程度であるかに注意し、それによって結論にどの程度の不確実さが含まれているかを指摘する必要があります。

信頼性

解決しなければならないもう 1 つの疑問は、平均と標準偏差の統計自体にどれだけの信頼性があるのかという疑問です。これらの統計値は測定値から計算されるため、新たに測定を行えば、この 2 つの統計値も変わることは明らかです。とりあえず、ここでは測定プロセスは有効であるという前提とします (注: 「実際の」標準偏差を測定することは不可能かもしれません。補足資料

で、この問題について検討している「Standard deviation measurement issues」セクションを参照してください)。この前提で測定プロセスを繰り返した場合、平均と標準偏差にはどのような違いが出てくるでしょうか。新たな測定によって顕著に異なる結果が得られるのでしょうか。

上記の疑問に答えるための最も直観的な方法は、統計の信頼区間を設けることです。統計値として計算された 1 つの値 (推定値) とは異なり、信頼区間は推定値の範囲です。この範囲には、信頼レベルと呼ばれる確率 p が関連付けられます。大抵の場合、p は 95% に設定され、信頼区間を比較する間、この値は変わることがありません。信頼区間は区間の大きさが信頼性を示すため、直観的にその内容がわかります。つまり、区間が狭ければ統計値が正確であることを意味し、区間が広ければ不確実性を示すということです。例えば、平均実行時間の信頼区間がタスク A では [1, 1.1] ミリ秒で、タスク B では [0.998, 0.999] ミリ秒だとすると、B の平均は A の平均より確実であることがわかると同時に、(この信頼レベルで) B の平均のほうが A の平均より小さい値であることがわかります。詳しい説明については、補足資料の「Confidence intervals」セクションを参照してください。

歴史的には、信頼区間を簡単に計算できるのは少数の一般的な確率密度関数 (ガウス分布など) と単純な統計値 (平均など) に対してのみでした。ところが 1970年代の後半、ブートストラップという手法が開発されます。これは、信頼区間を設けるのに最適な一般的手法で、平均などといった単純な統計だけでなく、あらゆる統計に有効です。さらに、ノンパラメトリック・ブートストラップ法では、ベースとなる確率密度関数についての前提条件がありません。そのため、非物理的な結果 (信頼区間の下限に対する負の実行時間など) をもたらすことは決してなく、誤った前提の確率密度関数を選択した場合 (ガウス分布と仮定するなど) に比べ、一層絞り込まれた正確な信頼区間になります。ブートストラップ法についての詳細はこの記事では説明しませんが、以下に説明するフレームワークには、この計算を行う `Bootstrap` というクラスが組み込まれています。詳細は、該当する Javadocs とソース・コードを参照してください (「参考文献」を参照)。

以上をまとめると、以下の手順が必要となってきます。

- 何度もベンチマーク測定を実行する。
- 測定値から、平均と標準偏差を計算する。
- これらの統計値を使用して、2 つのタスクが速度の点で明らかに異なるか (つまり平均が 3標準偏差以上離れているか)、あるいは 2 つのタスクが重なっているかを判断する。
- 平均と標準偏差の信頼区間を計算し、これらの統計値の信頼性レベルを示す。
- 信頼区間を計算する最善の方法としてブートストラップ法を使用する。

フレームワークの紹介

これまで、Java コードのベンチマークを実行する際の一般原則を説明してきました。ここからは、今までに説明した問題の多くに対処する、「すぐに実行できる」ベンチマーク・フレームワークを紹介します。

プロジェクト

この記事の関連サイト (「参考文献」を参照) から、プロジェクト ZIP ファイルをダウンロードしてください。この ZIP ファイルには、ソースおよびバイナリー・ファイル、そして簡単なビルド環境が含まれています。ファイルの中身を任意のディレクトリーに解凍し、最上位レベルに配置された readMe.txt ファイルで、その詳細を確認してください。

API

このフレームワークになくてはならないクラスは、`Benchmark` です。ほとんどのユーザーが目を向ける必要があるのはこのクラスだけで、その他はすべて補助的なものです。この API は、大抵の場合は簡単に使えます。つまり、ベンチマーク対象のコードを `Benchmark` コンストラクターに指定するだけで、ベンチマークのプロセスが自動的に行われます。その後に行うステップとしては通常、結果レポートを生成すればよいだけです。

タスク・コード

当然のことながら、実行時間のベンチマークを行う対象のコードがなければなりません。対象コードには 1 つだけ制約があり、コードは `Callable` または `Runnable` 内に含まれていることが必要となります。この制約さえ満たしていれば、必要なものをすべて完備したマイクロベンチマークから、完全なアプリケーションを呼び出すコードに至るまで、Java 言語で表現可能なコードであれば、どんなコードでも対象にすることができます。

通常、タスクは `Callable` として作成したほうが便利です。`Callable.call` ではチェック `Exceptions` をスローできる一方、`Runnable.run` では `Exception` の処理を実装しなければならないからです。さらに、第 1 回の「デッド・コードの削除 (DCE)」セクションで説明したように、`Runnable` より `Callable` を使ったほうが、DCE を防止するのが多少簡単になります。一方、タスクを `Runnable` として作成した場合には、オブジェクトの作成とガーベッジ・コレクションのオーバーヘッドが最小限になるという利点があります。詳細は、補足資料の「Task code: Callable versus Runnable」セクションを参照してください。

結果レポート

ベンチマークの結果レポートを取得する最も簡単な方法は、`Benchmark.toString` メソッドを呼び出すことです。このメソッドは、最も重要な結果と警告だけを一行のサマリー・レポートに出力します。すべての結果と詳細な説明が記載された複数行の詳細レポートを取得するには、`Benchmark.toStringFull` メソッドを呼び出してください。あるいは、`Benchmark` のさまざまなアクセサーを呼び出してカスタム・レポートを生成することも可能です。

警告

`Benchmark` は一般的な問題を診断し、問題が見つかった場合には結果レポートのなかでユーザーに警告します。警告には、以下の内容が含まれます。

- **測定の合計時間が少なすぎる**: 第 1 回の「リソースの再利用」セクションで、Benchmark がガーベッジ・コレクションとオブジェクト・ファイナライゼーションのコストが十分に考慮されていないと判断した場合、どのように警告するのかを説明しています。
- **異常値および系列相関**: 実行時間の測定値は、異常値と系列相関の統計テストの対象となります。異常値は、大きな測定誤差が発生したことを示します。例えば、測定中にコンピューターで別のアクティビティーが開始された場合には、大幅な異常値が発生します。また、小さな異常値は、DCE の発生を示唆する場合があります。系列相関は、JVM が安定状態のパフォーマンス・プロファイルにまだ達していないことを示します (実行時間が不規則に小さく変動することによって示されます)。具体的には、正の系列相関は傾向 (上昇または下降の

いずれか) を示し、負の系列相関は平均回帰 (実行時間の揺れなど) を示します。いずれにしても望ましくない状態です。
- **不正確な標準偏差**: 詳細については、補足資料の「Standard deviation warnings」セクションを参照してください。

単純な例

リスト 1 のコード・フラグメントに、このセクションで説明したポイントを示します。

リスト 1. 35 番目のフィボナッチ数のベンチマーク計算

```
public static void main(String[] args) throws Exception {
    Callable<Integer> task =
        new Callable<Integer>() { public Integer call() { return fibonacci(35); } };
    System.out.println("fibonacci(35): " + new Benchmark(task));
}

protected static int fibonacci(int n) throws IllegalArgumentException {
    if (n < 0) throw new IllegalArgumentException("n = " + n + " < 0");
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

リスト 1 の `main` は、ベンチマーク対象のコードを `Callable` として定義し、このコードをそのまま `Benchmark` コンストラクターに渡しています。この `Benchmark` コンストラクターは、最初はコードのウォームアップを確実にするため、続いて実行の統計値を収集するために task を何度も実行します。コンストラクターからリターンすると、そのコードが有効なコードのコンテキスト (つまり、`println` 内) によって新しい `Benchmark` インスタンスの `toString` メソッドが呼び出され、このメソッドがベンチマークの結果についての統計の概要をレポートします。`Benchmark` の使用方法は通常、このように単純です。

私の構成では、以下の結果となります。

```
fibonacci(35): first = 106.857 ms, mean = 102.570 ms (CI deltas: -35.185 us,
+47.076 us), sd = 645.586 us (CI deltas: -155.465 us, +355.098 us)
WARNING: execution times have mild outliers, SD VALUES MAY BE INACCURATE
```

この結果は、以下のように解釈できます。

- `fibonacci(35)` の最初の呼び出しでは、実行時間が 106.857 ミリ秒でした。
- 実行時間の平均推定値は、102.570 ミリ秒です。平均の 95% の信頼区間は推定値の約 -35/+47 マイクロ秒、つまり [102.535, 102.617] ミリ秒です。これは比較的狭い区間なので、平均は信頼性があることになります。
- 実行時間の標準偏差推定値は、645.586 マイクロ秒です。標準偏差の 95% の信頼区間は推定値の約 -155/+355 マイクロ秒、つまり [490.121, 1000.684] µs です。これは比較的広い区間なので、信頼性はかなり落ちることになります。実際、最後に警告として、標準偏差が正確に測定されていなかったと示されています。
- この結果は異常値についても警告しています。この例では異常値を無視していますが、心配な場合には `Benchmark` の `toStringFull` メソッドを使ってコードを返すようにしてください。それによって異常値がすべてリストアップされるため、このリストから判断することができます。

データ構造へのアクセス時間

ベンチマークの問題、そして `Benchmark` を使用してこれらの問題に対処する方法がよくわかるのは、一般的ないくつかのデータ構造のアクセス時間を比較するベンチマークです。

データ構造のアクセス時間だけを測定するのが、正真正銘のマイクロベンチマークです。確かに、マイクロベンチマークには多くの注意事項が適用されます (例えば、アプリケーション全体の動作状態はほとんど示さないなど)。けれどもこれから説明する測定で、マイクロベンチマークの名誉を挽回します。これらの測定は、正確に行うのが難しいこと、そしてさまざまな興味深い問題 (キャッシュの影響など) が持ち上がってくることから、絶好の例となります。

リスト 2 は、配列のアクセス時間についてのベンチマーク・コードです (その他のデータ構造の場合でも、同じようなコードになります)。

リスト 2. 配列アクセスのベンチマーク・コード

```
protected static Integer[] integers;    // values are unique (integers[i] <--> i)

protected static class ArrayAccess implements Runnable {
    protected int state = 0;

    public void run() {
        for (int i = 0; i < integers.length; i++) {
            state ^= integers[i];
        }
    }

    public String toString() { return String.valueOf(state); }
}
```

このコードは `integers` 配列にアクセスする以外のことは何もしないのが理想ですが、`integers` のすべての要素にアクセスしやすいように、私はループを使いました。そのため、ベンチマークにはループに伴うオーバーヘッドがもたらされます。ここで、より便利な for (`int` i : integers) 拡張 `for` ループではなく、昔ながらの明示的 int ループを使っている点に注目してください。これは、このループのほうが配列に対しても多少処理速度に優れているためです (「参考文献」を参照)。私がここで使ったループは、取るに足らない欠点にしかなりません。まともなジャスト・イン・タイム (JIT) コンパイラーであればループのアンロールを実行するため、影響が軽減されるからです。

一方、それよりも深刻な問題は、コードが DCE を発生させないようにする必要があったことです。そこでまず、タスク・クラスには `Runnable` を選択しました。前述のとおり、このクラスはオブジェクトの作成とガーベッジ・コレクションのオーバーヘッドを最小限にするからです。これは、このような軽量のマイクロベンチマークには重要な意味を持ちます。第二に、このタスク・クラスは `Runnable` であることから、DCE を防ぐためには配列アクセスの結果を `state` フィールドに割り当てる必要がありました (さらに、`state` は `toString` をオーバーライドしたメソッドで使用する必要がありました)。第三の点として、`state` の以前の値を使用して配列アクセスでビット単位の XOR (排他的論理和) 演算を実行し、すべてのアクセスが確実に実行されるようにしました (単に `state = integers[i]` を行うだけだと、賢いコンパイラーではループ全体がスキップされたと認識する可能性があるため、代わりに `state = integers[integers.length - 1]` とします)。この 4 つの追加演算 (フィールドの読み取り、`Integer` から `int` へのオートアンボクシング、ビット単位の XOR、フィールドの書き込み) はベンチマークに影響のあるオーバーヘッドですが、避け

られないオーバーヘッドです。このように、実際には単に配列のアクセス時間を測定するだけでは済みません。他のすべてのデータ構造ベンチマークも同様ですが、比較的アクセス時間が長いデータ構造であれば、相対的な影響はごくわずかです。

図 3 と図 4 は、それぞれに異なる 2 つの `integers` のサイズを使用して、私のいつものデスクトップ構成で測定したアクセス時間の結果です。

図 3. データ構造アクセス時間 (1024 要素の場合)

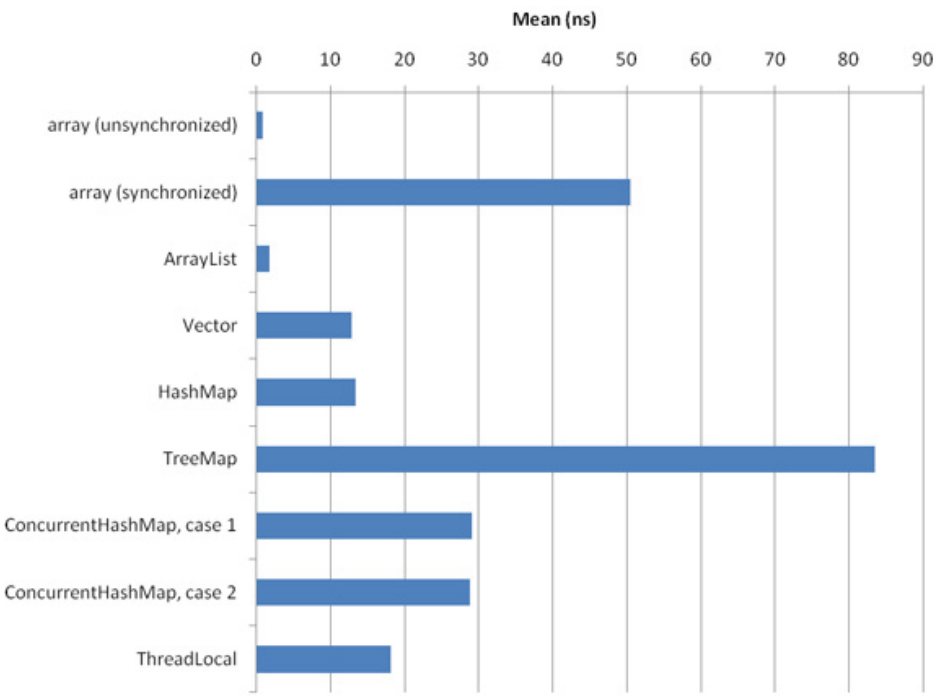
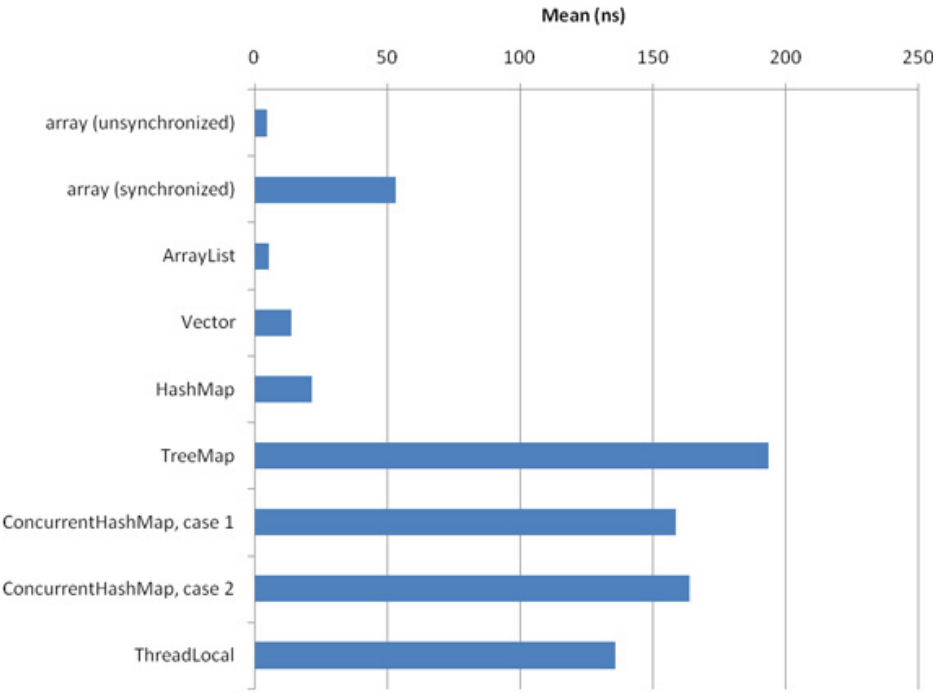


図 4. データ構造アクセス時間 (1024 × 1024 要素の場合)



上記の結果についてのコメントは、以下のとおりです。

- 平均実行時間の推定値のみが示されています (平均の信頼区間は常に狭いため (標準の幅は平均より最大で 1000 倍狭くなっています)、上記のグラフには現れません)。
- いずれのベンチマークでも、使用しているスレッドは 1 つだけです。したがって、同期化されたクラスはスレッド競合状態にありません。
- 配列要素アクセス (非同期) コードはリスト 2 に記載したとおりです。
- 配列要素アクセス (同期) コードは、ループ本体が synchronized (integers) { state ^= integers[i]; } となっている点を除けば、非同期コードとまったく同じです。 `synchronized (integers) { state ^= integers[i]; }`
- 図 3 と図 4 の `ConcurrentHashMap` の case 1 と case 2 の違いは、使用されている `ConcurrentHashMap` コンストラクターの点に関してのみです。case 1 では `concurrencyLevel = 1` を指定し、case 2 ではデフォルト (つまり、16) を使用しています。すべてのベンチマークでは 1 つのスレッドしか使用していないため、case 1 のほうが多少速くなるはずです。
- すべてのベンチマークの結果には、1 つまたは複数の警告が出されています。
  - ほとんどすべてのベンチマークで異常値がありましたが、いずれも結果を大きく左右するほどの極端な値ではなかったようです。
  - 1024 × 1024 要素の結果すべてに、系列相関がありました。これがどれだけ大きな問題かは明らかでありませんが、1024 要素の結果には、いずれも系列相関はありませんでした。
  - 標準偏差は常に測定不可能でした (マイクロベンチマークの常です)。

これらの結果は、おそらく期待していたとおりでしょう。つまり、非同期配列アクセスが最も高速なデータ構造だということです。2 番目に速いのは `ArrayList` で、未処理の配列の場合とほとんど変わりません。これはおそらく、サーバー JVM が、ダイレクト・アクセスをその基礎と

なる配列にインライン化するという素晴らしい働きをしているためで、同期配列より遥かに高速になっていますが、そうでなければ `Vector` と変わらないはずです。その次に高速なデータ構造は `HashMap`、そして `ThreadLocal` (基本的に、現行のスレッドがキーとなる特殊化されたハッシュ・テーブル) と続きます。実際には、一連のテストでは `HashMap` は `Vector` に匹敵するほど速くなっていますが、これに感心させられるのは、とりわけ高速な `hashCode` 実装 (`int` の値を返すだけです) を持つ `Integers` がキーとして使用されているということを考えるまでの話です。後は、`ConcurrentHashMap` の 2 つのケース、そして最も遅い `TreeMap` という順になっています。

ここには結果を記載していませんが、まったく同じベンチマークをまったく異なるマシンでも実行しています (このマシンは、SunOS asm03 5.10 を使用した SPARC-Enterprise-T5220 (32GB のRAM 搭載、クロック周波数 1.167GHz) です。JVM バージョンと設定は私のデスクトップと同じでしたが、今回は経過時間ではなく CPU 時間を測定する `Benchmark` を構成しました。その理由は、すべて 1 つにスレッド化されたテストには、Solaris によるサポートが極めて有効だからです)。異なるマシンでの場合の相対的な結果は上記と同じでした。

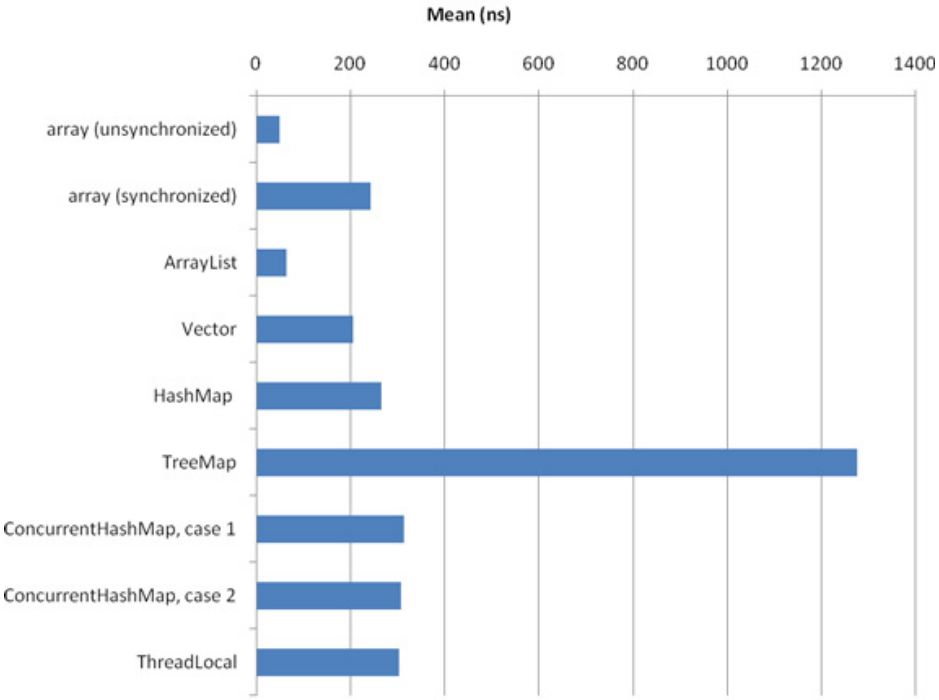
上記の結果で大きな異常が現れているのは、同期配列のアクセス時間だけです。`Vector` に相当する程度のアクセス時間になると考えていましたが、結果は一貫して 3 倍以上になっています。可能性として高いのは、ロック関連の最適化 (ロック削除やロック・バイアスなど) が開始できなかったためです (「参考文献」を参照)。この予想は、カスタマイズしたロック最適化を使用する Azul の JVM では、この異常が現れないという事実によって裏付けられているように思います。

小さな異常としては、`ConcurrentHashMap` の case 1 は、1024 × 1024 の要素が使用されている場合にだけ case 2 よりも速くなり、1024 要素が使用されているときには逆に多少速度が落ちています。この異常は、数が異なるテーブル・セグメントのメモリー配置のささいな影響によるものと考えられます。このような影響は、T5220 ボックスには現れません (要素の数にかかわらず、case 1 は常に case 2 より高速です)。

異常ではない点として、`HashMap` のパフォーマンスが `ConcurrentHashMap` と比べて速くなっていることにも注目してください。case 1 と case 2 の両方のコードは、`state ^= integers[i]` が `state ^= map.get(integers[i])` に置き換えられていること以外はリスト 2 と同様です。`integers` の要素は順次発生し (`integers[i].intValue() == i`)、それと同じ順序でキーとして提供されます。これで明らかになるのは、`HashMap` に含まれるハッシュ前処理関数の順次キャッシュの局所性は、`ConcurrentHashMap` より優れているということです (`ConcurrentHashMap` にはより優れた高ビット分散が必要なためです)。

このことによって持ち上がってくる興味深い疑問は、上記に示した結果は、`integers` が順に繰り返されていることにどれだけ依存しているのかという点です。特定のデータ構造にとって有利となるメモリー局所性が影響しているのでしょうか。この疑問に対する答えを出すため、今度は順次要素ではなく、`integers` のランダム要素を選択してベンチマークを再度実行しました (その方法として、ソフトウェアのリニア・フィードバック・シフト・レジスターを使用して、擬似ランダム値を生成しました (「参考文献」を参照)。このレジスターがデータ構造アクセスごとに追加するオーバーヘッドは、約 3 ナノ秒です)。`integers` が 1024 × 1024 要素の場合の結果を図 5 に記載します。

図 5. データ構造アクセス時間 (1024 × 1024 要素の場合、ランダム・アクセス)



ご覧のように、図 4 と比べると `HashMap` のパフォーマンスは `ConcurrentHashMap` とほぼ同様のパフォーマンスになっています (私が主張したとおりです)。`TreeMap` の成績は散々です。array と `ArrayList` データ構造は変わらず優秀ですが、パフォーマンスの違いはそれほどではなくなっています (それぞれランダム・アクセスのパフォーマンスが約 10 倍劣化している一方、`ConcurrentHashMap` のランダム・アクセスのパフォーマンスは約 2 倍しか劣化していません)。

もう 1 つのポイントです。図 3、4、5 は個別のアクセス時間を表示したもので、例えば、図 3 を見ると、`TreeMap` から単一の要素にアクセスする時間は 80 ナノ秒少々だということがわかります。しかし、タスクはいずれもリスト 2 のようなもので、それぞれのタスクは内部で何度もデータ・アクセスを行っています (つまり、`integers` の各要素をループしているということです)。複数の同一アクションからなるタスクから、個々のアクションについての統計値を抽出するにはどうすればよいのでしょうか。

リスト 2 では、このようなタスクの処理方法を示す親コードを考慮していませんでした。そこで使えるのが、リスト 3 のようなコードです。

リスト 3. 複数のアクションを持つタスクのコード

```
public static void main(String[] args) throws Exception {
    int m = Integer.parseInt(args[0]);

    integers = new Integer[m];
    for (int i = 0; i < integers.length; i++) integers[i] = new Integer(i);

    System.out.println(
        "array element access (unsynchronized): " + new Benchmark(new ArrayAccess(),m));
    // plus similar lines for the other data structures...
}
```

リスト 3 では、引数が 2 つのバージョンの `Benchmark` コンストラクターを使っています。2 番目の引数 (リスト 3 の太字で示した引数) が指定するのはタスクを構成する同一アクションの数で、この場合は `m = integers.length` となります。詳細については、補足資料の「Block statistics versus action statistics」セクションを参照してください。

最適ポートフォリオの計算

この記事ではこれまで、マイクロベンチマークについてだけ検討してきました。面白いことに、ベンチマークの真の実用性は、実際のアプリケーションのパフォーマンス測定でこそ発揮されます。

投資家である読者の多くにとって興味深いと思われる一例は、マーコヴィッツ (Markowitz) の平均分散ポートフォリオ最適化です。これは、ファイナンシャル・アドバイザーが優れたリスクおよび収益のプロファイルによってポートフォリオを作成するための標準的な手法です (「参考文献」を参照)。

このような計算を行うための Java ライブラリーを提供している会社の 1 つに、WebCab Components という会社があります (「参考文献」を参照)。リスト 4 は、この会社の Portfolio v5.0 (J2SE Edition) ライブラリーが有効フロンティアを求める上でのパフォーマンスを評価するベンチマーク・コードです。

リスト 4. ポートフォリオ最適化のベンチマーク・コード

```
protected static void benchmark_efficientFrontier(
    double[][] rets, boolean useCons, double Rf, double scale
) throws Exception {
    Benchmark.Params params = new Benchmark.Params(false);    // false to meas only first
    params.setMeasureCpuTime(true);

    for (int i = 0; i < 10; i++) {
        double[][] returnsAssetsRandomSubset = pickRandomAssets(rets, 30);
        Callable<Double. task = new EfTask(returnsAssetsRandomSubset, useCons, Rf, scale);
        System.out.println(
            "Warmup benchmark; can ignore this: " + new Benchmark(task, params) );
    }

    System.out.println();
    System.out.println("n" + "\t" + "first" + "\t" + "sharpeRatioMax");
    for (int n = 2; n <= rets.length; n++) {
        for (int j = 0; j < 20; j++) {    // do 20 runs so that can plot scatter
        double[][] returnsAssetsRandomSubset = pickRandomAssets(rets, n);
        Callable<Double> task = new EfTask(returnsAssetsRandomSubset, useCons, Rf, scale);
        Benchmark benchmark = new Benchmark(task, params);
        System.out.println(
            n + "\t" + benchmark.getFirst() + "\t" + benchmark.getCallResult());
        }
    }
}
```

この記事の話題はポートフォリオ理論ではなくベンチマークなので、リスト 4 では `EfTask` 内部クラスのコードは省略しています (注: `EfTask` は過去の資産収益に関するデータを取り、このデータから期待される収益と共分散を計算して有効フロンティア上の 50 のポイントを求め、そのうちシャープ·レシオが最大のポイントを返します (「参考文献」を参照)。この最大シャープ・レシオが、特定の資産セットにとって最適なポートフォリオの有効性の基準となり、リスク調整後の収

益が特定されます。詳細は、この記事のコード・ダウンロードに含まれる関連ソース・ファイルを参照してください)。

このコードは、資産数の関数としての実行時間とポートフォリオ品質の両方 (ポートフォリオの最適化に役立つ可能性のある情報) を同時に判断することを目的としています。この判断を行うのが、リスト 4 の `n` ループです。

このベンチマークには、いくつかの難題がありました。まず 1 つは、かなりの数の資産を検討している場合はなおさらのこと、計算に時間がかかることです。そのため、`new Benchmark(task)` といった単純なコードは使用しませんでした。このコードだと、デフォルトで 60 回の測定を実行するからです。代わりに選んだのは、カスタム `Benchmark.Params` インスタンスを作成し、このインスタンスで測定を 1 回だけ実行するように指定するという方法です (このインスタンスでは、デフォルトの経過時間ではなく、CPU時間の測定を実行するようにも指定しています。これは単に、CPU時間を測定するように指定する方法を説明することが目的です。このコンテキストでは WebCab Components ライブラリーがスレッドを作成しないため、この指定で問題はありません)。ただし、これらの単一の測定ベンチマークのいずれかが実行される前に、JVM が最初にコードを完全に最適化できるように `i` ループが何回かベンチマークを行っています。

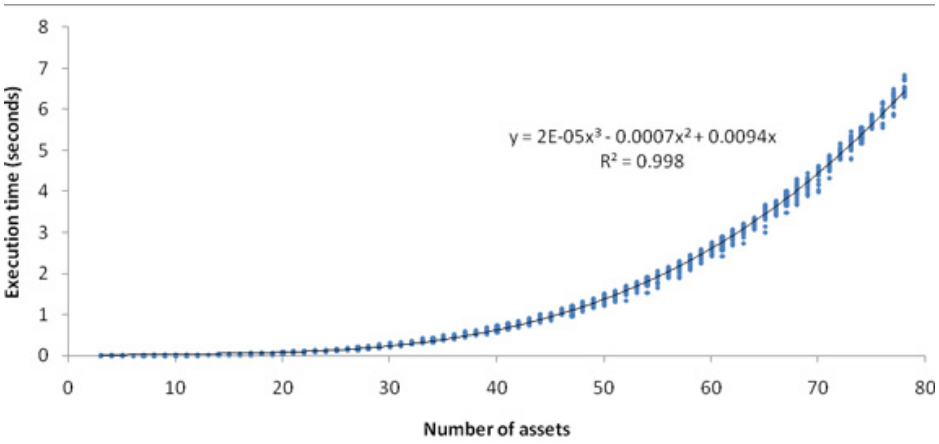
2 つ目の問題は、通常の結果レポートでは、このベンチマークの要求を満たさないことです。そのため、タブで区切った数値だけを記載するようにカスタマイズした結果レポートを生成しています。このようにすれば、以降のグラフ作成でスプレッドシートに簡単にコピー・アンド・ペーストできるからです。測定は 1 回しか行われないため、実行時間は `Benchmark` の `getFirst` アクセサー・メソッドで取得します。特定の資産セットについての最大シャープ・レシオは、`Callable` タスクの戻り値です。これは、`Benchmark` の `getCallResult` アクセサー・メソッドによって取得されます。

さらに、結果のばらつきを視覚的に表現したかったため、特定数の資産に対して、内部 `j` ループで各ベンチマークを 20 回実行するようにしています。これが、以下のグラフで示されている資産数ごとの 20 のドットです (場合によっては、ドットが重なっているため、少数のドットしかないように見えます)。

それでは、結果を見てみましょう。使用した資産は、現行のOEX (S&P 100) 指数の株です。過去の収益としては、過去 3 年間 (2005年1月1日から 2007年12月31日まで) にわたる毎週のキャピタル・ゲインを使用しました。配当は無視しています (配当を含めると、シャープ・レシオがわずかに上がるはずです)。

図 6 は、実行時間を資産数の関数として表したグラフです。

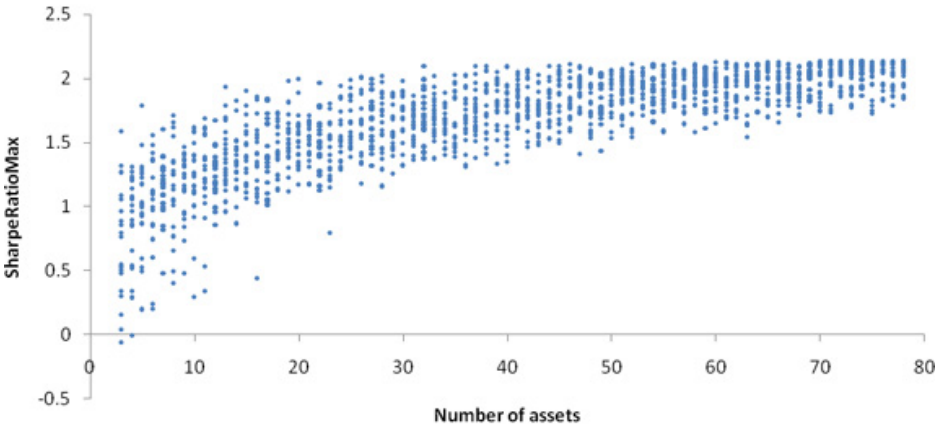
図 6. ポートフォリオ最適化の実行時間



ご覧のように、実行時間は資産数の 3 次関数として上昇しています。測定値のばらつきは実際のもので、ベンチマークのエラーではありません。ポートフォリオ最適化の実行時間は、対象とする資産のタイプによって異なるからです。具体的には、特定の共分散タイプには非常に慎重な (増分値を小さくした) 数値計算が必要となるため、実行時間に影響が出てきます。

図 7 は、ポートフォリオ品質 (最大シャープ・レシオ) を資産数の関数として表したグラフです。

図 7. ポートフォリオ品質



最大シャープ・レシオは始めのうちは上昇していますが、資産数が 15 から 20 のあたりになると横ばいになります。それ以降の資産数の増加による影響は、資産の最大数に近づくにつれ、値の変動範囲が狭まってくることくらいです。この影響も実際のもので、このようになっているのは対象の資産数が増えるにつれ、すべての資産が「ホット」資産 (最適なポートフォリオを占める) である可能性が 100% に安定するからです。この記事では説明しない他の見解については、補足資料の「Portfolio optimization」セクションを参照してください。

最後の注意点

マイクロベンチマークは実際の使用事例を反映する必要があります。例えば、私がデータ構造のアクセス時間を測定することにした理由は、JDK 収集は、標準アプリケーションではその動作の

最大約 85% が読み取り/トラバース、14% が追加/更新、1% が削除に費やされるという見込みで設計されているからです。ただし、この動作比率が変わると、相対的パフォーマンスもほぼ不規則に変わってきます。別の油断できない危険としては、クラス階層の複雑さもあります。マイクロベンチマークは大抵の場合、単純なクラス階層を使用しますが、複雑なクラス階層ではメソッド呼び出しのオーバーヘッドが顕著になる可能性があります (「参考文献」を参照)。そのため、正確なマイクロベンチマークは現実を反映していなければなりません。

ベンチマークの結果が関連性のあるものであることを確実にしてください。ヒントとして、`StringBuffer` と `StringBuilder` を比較するマイクロベンチマークは、Web サーバーのパフォーマンスについては多くを語らないはずです。それより上位レベルでのアーキテクチャーを選択することが (マイクロベンチマークには役に立ちませんが)、おそらく非常に大きな意味を持ってきます (皆さんはそれでも無意識に、ほとんどすべてのコードで以前の `Vector/Hashtable/StringBuffer` ではなく、`ArrayList/HashMap/StringBuilder` を使うと思います)。

マイクロベンチマークだけに依存することは禁物です。例えば、新しいアルゴリズムの効果を判断したい場合には、ベンチマーク環境だけではなく、実際のアプリケーション・シナリオでも測定して、実際に十分な差があるかどうかを確認してください。

当然のことながら、かなりのコンピューターと構成のサンプルをテストしなければ、全般的パフォーマンスについての結論は出せません。残念なことに、開発マシンでだけベンチマークを行い、該当するコードを実行するすべてのマシンについても、その結論を当てはめてしまうという誤りは珍しくありません。完璧を目指すなら、複数のハードウェア、さらには異なる JVM も必要です (「参考文献」を参照)。

プロファイルを作成することを無視しないでください。期待通りの振る舞いであること (例えば、時間の大半は重要だと思われるメソッドに費やされること) を確認するには、あらゆる種類のプロファイル作成ツールを使ってベンチマークを実行してください。プロファイルを作成することで、DCE によって結果が無効にされることもなくなります。

突き詰めるところ、確かな結論を導くためには、下位レベルでどのように機能するかを実際に知る以外に方法はありません。例えば、Java 言語の sin 関数の実装 (`Math.sin`) が C 言語での実装より速度に優れているかどうかを調べる場合、x86 hardware ハードウェアでは Java の sin 関数のほうが大幅に遅いという結果になるに違いありません。これは、Java が、高速ではあるけれども不正確な x86 ハードウェアのヘルパー命令を正しく回避するためです。知識がない人々は、このベンチマークによって C 言語のほうが Java 言語より大幅に速いと結論付けますが、実際に判断している内容は、専用 (ただし不正確な) ハードウェアの命令は正確なソフトウェア計算よりも処理が速いということに過ぎません。

まとめ

大抵の場合、ベンチマークでは何度も測定を実行し、統計を利用して結果を解釈する必要が出てきます。この記事で紹介したベンチマーク・フレームワークは、このような必要性をサポートするだけでなく、その他多くの問題にも対処します。このフレームワークを使うにしても (2 回の連載記事で紹介した資料を参考にしてください)、あるいは独自のベンチマーク・フレームワークを作成するにしても、Java コードの効率的な実行を確実にするための知識はこの記事を読んで身についているはずです。

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。