Part1

Javaプログラマーに贈る

最適化の手引き

首藤一幸　早稲田大学 村岡研究室　http://www.shudo.net/

プログラムの性能はコードの書き方ひとつで大きく変わるものだ。同じ目的を達成するプログラムであっても、書き方によって性能が数ケタ異なる場合すらある。そこで本パートでは、性能の良いプログラムの開発や性能改善、つまり最適化について、プログラマーが知っておくべき事柄を述べてみたい。また、最適化という視点から見たJavaの特徴や、Javaに固有の最適化手法にも言及する。さらに後半では、特別リポートのかたちで、最新のJVMやJITコンパイラの性能評価について報告する。

最適化の指標

「最適化」とは何だろう。あらためて問われると、返答に窮する方も多いのではなかろうか。「プログラムを速くすること」と答える方が多いかもしれない。そこでまずは、そもそも最適化とは何なのかを確認しておきたい。

「最適化する」に対応する英単語は「optimize」である。辞書によれば、この語の意味は次のとおりである。

> 最高に活用する、できるだけ能率的に利用する（研究社新英和中辞典より）

つまり、プログラムの最適化とは「プログラムの実行環境をできるだけ能率的に、最高に活用できるようにプログラムを書くこと、改善すること」を指すのである。ここで言う環境とは、プロセッサ（CPU）はもちろん、プログラムによってはハードディスクやネットワーク、その向こうにある別のマシンまで含むかもしれない。さらには、コードの書き方だけではなく、JVM起動時のパラメータやOS側の設定といった“プログラムの外側”での最適化が有効な局面も多い。

「最高に活用する」とは何とも大ざっぱであいまいな表現だが、「最適化」が指す範囲はそれくらい広い。それゆえに、広範な知識や経験に基づいた、環境や状況、目的に応じた対応が必要となる。

また、「最高に活用」「能率的に利用」とは、単にプログラムの速さだけを指すわけではない。例えば、手もとのプログラムを2倍速くする方法を考えついたとしよう。しかし、その方法が従来の3倍のメモリを消費するとしたら、その方法を採用してよいかどうかは、実行速度と省メモリのそれぞれがどれくらい重要なのかによって決まるだろう。

最適化の指標のうち、代表的なものを以下に挙げてみる。

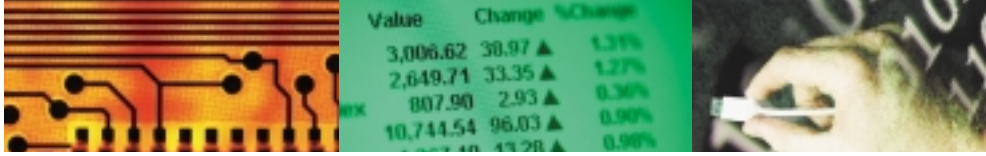
●速さ：処理時間の短さである。最適化の対象として見た場合、「速さ」は処理が発生してから完了するまでの時間である「遅延」と、単位時間当たりの処理量である「スループット」に分けて考えられる。この2つを同時に最適化できない場合も多い。

●メモリ消費量：いかにメモリを消費しないかということである。このことは、一般に利用可能なメモリが少ない組み込み用途で特に重要となる。また、メモリ消費量は、同時に動作できるプログラムの数や扱える問題の大きさにも影響する。Javaでは、ガーベジ・コレクション（GC）の頻度やGCに費やされる時間にも影響を与えるので、上記「速さ」とも無関係ではない。

●スケーラビリティ：問題や環境のさまざまな大きさ（量、数）への適合性を指す。スケーラビリティには、例えばネットワーク経由の要求を処理するプログラムが1秒当たりに処理できる要求の数や、マルチプロセッサ・マシンにおいてプロセッサ数が増えたときにそれをどれだけ有効に活用できるか、などが該当する。スケーラビリティは、「速さ」をはじめとして、さまざまな指標と深く関係している。

上記以外にも、最適化の指標にはさまざまなものが考えられる。ディスクへの入出力やネットワーク経由での通信が多いプログラムでは、入出力や通信の量と回数をいかに減らすかが重要であろうし、アプレットのようにネットワーク経由でプログラム自身が送受される場合には、プログラム自体のサイズを小さくすることも重要となろう。

ここに挙げた指標は、それぞれが独立しているわけではなく、お互いに深く関係し合っている。先ほど挙げた例のように、2倍速くするためのメモリ消費量が従来の3倍になってしまうこともあるし、それとは逆にメモリ消費量を減らすことで処理が速くなることも珍しくない。このように複数の指標が同時に向上するのであればよいが、指標間の関係はトレードオフであることが多い



(図1)。

ここで、指標のトレードオフにまつわる筆者の経験について述べてみたい。それは、筆者がある暗号方式をJava言語で実装していたときのことである。

筆者はあるとき、開発中のプログラムの処理性能を2〜3割向上させる方法があることに気がついた。ところが、その方法を使うためには、暗号化や復号に先立って、あらかじめ128KBの表を作成しておく必要があった。同時に、表のサイズを256KBとすることで、最高の性能となることもわかった。プログラムが表以外に消費するメモリは数KBから10数KBだけだった。

さて、その方法は採用できるだろうか。結論から言うと、当時の筆者らは採用した。なぜなら、そのプログラムは、64MBのメモリ環境で速さを評価するという目的でのみ使われることがわかっていたからである。もし、そのプログラムが、メモリ量の制限がずっと厳しい環境で使用される予定だったら、採用できなかっただろう。

トレードオフが発生するのは、上記のような複数の指標(性能とメモリ消費)の間だけではない。最適化の多くはプログラムの読みやすさ、つまり可読性を下げることにつながる。可読性が下がれば、プログラムのメンテナンスのしやすさ、つまり保守性も下がることになる。

可読性を下げる例を1つお目にかけよう。行列の乗算という数値計算を考える。サイズがn×nの正方行列AとBを乗じた結果をCに格納するJavaコードはリスト1-①のように書ける。

行列積が何であるかを知っていれば、リスト1-①のコードが何をしているかはすぐに理解できるだろう。これに対し、よく知られている最適化手法のループ展開をリスト1-②のように施す。これは、最も内側のループについての4段のループ展開である。

リスト1-②は、ループの終了条件が満たされているかどうかの判定の回数を減らすことで高速化を図ろうという最適化である。ところが、ループ展開を知らない人がこれを見ても、何をするコードなのか、すぐには理解できないだろう。つまり、コードの可読性が低下したのである。

性能と可読性のどちらを優先させるべきかについては、もちろんケース・バイ・ケースだ。ここで知っておくべきことは、可読性の低下を招くような最適化の多くは、プログラマーの手で行わなくても、JITコンパイラなどの処理系や実行系が行ってくれるということである。そのため、可読性を下げてまで最適化を手作業で行うよりも実行系に任せたほうが、大抵の場合は得策である。また、たとえ現時点では実行系による最適化が行われないとしても、ループ展開のように原理的には自動で行える最適化であれば、将来登場する実行系ではそれを行ってくれるかもしれない。

最適化の際に考慮せねばならないトレードオフは、ほかにもまだある。プログラムの性質に関するトレードオフ以前に、その最適化を施すか否かの時点で、最適化にかかる手間とそれによって得られる利益の間にトレードオフは発生する。

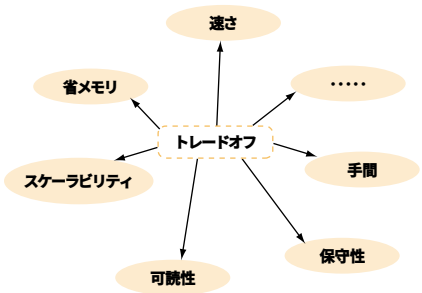
例えば、次の場合を考えてみよう。1回の実行に1カ月を要するプログラムに対し、ある最適化を施すとそれが1週間に短縮できるとする。しかし、その最適化を施す作業にかかる時間が1カ月だとしたら、どうすべきであろうか。

そのプログラムが2回、そして3回と使われるのであれば、最適化の意味は大きい。しかし、もし1回しか実行されないものならば最適化を施す意味はない。最適化を施すことが目的である場合を除き、手間に見合った利益があるかどうかについて考えるべきである。

最適化に求められる知識

最適化の結果は、それを施したプログラマーの質を如実に反映したものになる。プログラマーに対しては、コンピュータやアルゴリズム、最適化への取り組みについての相応の見識が求められるのだ。そこで、

図1:最適化に伴うトレードオフ



リスト1:コードの可読性を下げる最適化の例

①行列積(ループ展開を行う前)

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      double sum = 0;
      for (k = 0; k < n; k++) {
         sum += A[i][k] * B[k][j];
      }
      C[i][j] = sum;
   }
}
```

②行列積(ループ展開を行った後)

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      double sum = 0;
      for(k = 0; k+ 3 < n; k += 4){
         sum += A[i][k] * B[k][j];
         sum += A[i][k + 1] * B[k + 1][j];
         sum += A[i][k + 2] * B[k + 2][j];
         sum += A[i][k + 3] * B[k + 3][j];
      }
      for (; k < n; k++) {
         sum += A[i][k] * B[k][j];
      }
      C[i][j] = sum;
   }
}
```

ここでは最適化にはどんな知識や理解が求められるのかについて述べてみたい。まずは、いくつかの例を通じて、プログラマーに何が要求されるのかを見ていこう。

例1

配列に格納されている値の中から最大値と最小値を探すという処理を考える。そのコードは、素直に書くと60ページのリスト2-①のようになるだろう。ある要素を最大値候補(max)と比較し、それよりも大きければ新しい最大値候補とする。次に、その要素を最小値候補(min)と比較し、それよりも小さければ新しい最小値候補とするのである。この方法は誰でも思いつく。

このアルゴリズムでは、大小比較の回数は最悪の場合、配列長(SIZE)の倍である。elseによって若干の比較をさぼれるが、ほとんど期待はできない。ところが、リスト2-②のようにすることで、大小比較の回数を約4分の3に削減することができる。比較

回数が減るぶん、処理にかかる時間は短くなるだろう。

その仕掛けは次のとおりである。配列を1要素ずつ見ていくのではなく、2要素ずつ見ていく。2要素を取り出して比較する。2要素のうち、大きいほうを最大値候補と比較し、小さいほうを最小値候補と比較する。リスト2-①のアルゴリズムでは、配列の1要素当たりおよそ2回の大小比較をしていたが、リスト2-②のアルゴリズムでは2要素当たり3回の比較で済む。

例2

int型の1次元配列a[]の全要素の値を2分の1にする。値を2分の1にする方法としては「2で割る」と「1/2を乗じる」のほかに、1ビットだけ算術右シフトするという方法もある。それぞれの方法は、表1の中央の列のように書くことができる。

リスト2:最大値と最小値の検索

①素直に書いたもの

```
int a[] = new int[SIZE];
int max, min;
int i;
   ……
max = min = a[0];
for (i = 1; i < SIZE; i++) {
   if (a[i] > max)
      max = a[i];
   else if (a[i] < min)
      min = a[i];
}
```

②比較の回数を減らしたもの

```
int a0 = a[0], a1 = a[1];
if (a0 > a1) {
   max = a0; min = a1;
}
else {
   max = a1; min = a0;
}
for (i = 2 - (SIZE & 1); i < SIZE; i += 2) {
   a0 = a[i];
   a1 = a[i + 1];
   if (a0 > a1) {
      if (a0 > max)
         max = a0;
      if (a1 < min)
         min = a1;
   }
   else {
      if (a0 < min)
         min = a0;
      if (a1 > max)
         max = a1;
   }
}
```

表1:int型の1次元配列a[ ]の全要素の値を2分の1にする方法と、その実行時間

方法	コード	実行時間(ミリ秒)
除算(1)	a[i] /= 2;	600
除算(2)	a[i] /= 2.0;	1,020
乗算	a[i] *= 0.5;	1,030
シフト	a[i] >>>= 1;	540

※配列長は1,000万、実行環境はPentium II/333MHz、Linux、IBM JDK 1.3.0

配列長を1,000万として、それぞれの方法を筆者の手もとの環境(Pentium II/333MHz、Linux、IBM製のJDK 1.3.0)で実行した際に要した時間は表1の右側の列のようになった。

この表からわかるとおり、筆者の環境では算術シフトを使った方法が最も速い。こうした実行時間の差がどこからくるのかを推定、調査することは最適化に必ず役に立つ。大抵の場合、シフトは乗除算よりも軽い処理であり、除算のように実行時間がオペランド(計算対象の値)に依存するようなこともない。

では、乗算は大抵除算よりも軽いからといって、除算よりは乗算を使えばよいのかと言えば、それだけでは済まない。なぜならば、乗算より除算(1)のほうが速いという結果が出ているからである。除算(1)ではint型の整数で割っているので、整数同士の除算が行われる。それに対し、除算(2)と乗算では右辺がdouble型のため、int型である左辺のa[i]は演算の前にdouble型に変換される。Java言語の仕様が「変換せよ」と定めているのである。また、演算の後で結果をa[i]に格納する際には、結果をdouble型からint型に変換しなければならない。除算(2)と乗算の方法には、この変換のオーバーヘッドがある。

しかし実を言うと、JVMとJITコンパイラがブラックボックスであるために、プロセッサが本当に上記のような処理を行っているかどうかはわからない。例えば、JITコンパイラが浮動小数点演算を整数演算に置き換えているかもしれないのである。だが、除算(1)と(2)の実行時間に大きな差が見られたこと、さらにそのような置き換えは慎重な検査を要するわりに有効な局面が少ないことから、上述の言語仕様どおりの変換処理が行われているものと思われる。

例3

大きなサイズの配列の内容をファイルに保存する処理について考える。リスト3のコードは、何も考えずに書き下した場合を想定したものである。

配列長(SIZE)を100万として、このプログラムを筆者の環境で実行すると27秒かかる。しかし、「new FileOutputStream(…)」の次の行に以下の1行を加えると、とたんに実行時間は1.01秒となった。およそ27分の1になった計算である。

```
out = new BufferedOutputStream(out);
```

元のプログラム(リスト3)では、メソッドwriteIntの呼び出しごとにOSに対して書き込み要求が起きていた。それが、クラスBufferedOutputStreamのインスタンスとしてバッファを用意したところ、ある程度の書き込みデータがたまった時点で初めてOSに書き込みが要求されるように変わったのである。OSへの要求という、時間のかかる重い処理の回数が減ったことで、実行時間が短縮されたわけだ。

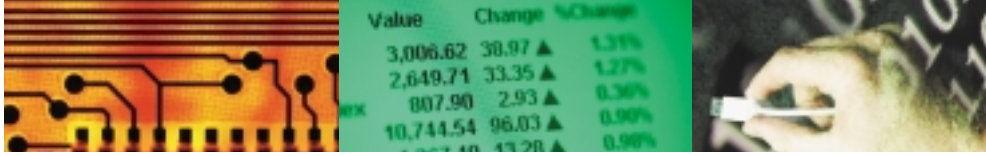
これら3つの例は、最適化の際にプログラマーに求められるものを示唆している。

- ●例1:アルゴリズム
- ●例2:コンピュータ・アーキテクチャ(プロセッサ)、Java言語、処理系(JITコンパイラ)
- ●例3:システム・ソフトウェア(OS)、クラス・ライブラリ

Javaに固有の事柄が多いものの、これらのほとんどはあらゆるコンピュータ・プログラムの最適化に通用する。

アルゴリズムとデータ構造、特に計算量に関する理解は、プログラミング言語の入門書には載っていないにもかかわらず、どんなプログラマーにも必須である。

あるアルゴリズムでデータ量nが3倍、4倍になったときにその実行時間が3^2=9倍、4^2=16倍と増える場合、そのアルゴリズムのオーダはn^2であると言い、$O(n^2)$と書く。このオーダの違いは、実行時間に決定的な差をもたらす。例えば、同じ結果が得られるアルゴリズムAとBがあり、それぞれ$O(n)$、$O(n^2)$だったとする。このとき、データ量が1,000倍になると、Aの実行時間は1,000倍



で済むが、Bの実行時間は1000^2=100万倍となってしまう。

データを指定された順に並び替えるソート操作は、オーダの異なるいくつもの方法があることで知られている。単純なバブル・ソート、挿入ソート、選択ソートは$O(n^2)$であるが、マージ・ソートやクイック・ソートなどは$O(n\log n)$というように、より効率が高い。nを増やした場合の伸びを比べると、$n\log n$のほうがn^2より小さいのである。

オーダの低いアルゴリズムのほうがベターではあるが、もちろん状況に応じた選択をすべきである。例えば、$O(n^2)$のアルゴリズムを使用すれば100個のデータを1秒で済ませられる処理があるとしよう。これに$O(n)$のアルゴリズムを適用すると10秒かかるとする。このとき、データ数が10倍以上になれば実行時間の逆転が起こるわけだが、もしデータ数がこれ以上増えないことがわかりきっているなら、何も$O(n)$のアルゴリズムを採用する必要はない。

例2のコンピュータ・アーキテクチャ（つまりコンピュータの仕組み）に関する理解は、Javaにかぎらず、プログラムがコンピュータ上で実行されるものである以上、極めて重要なことである。高速化の観点からは特に、メモリ階層とプロセッサ（CPU）に対する理解がポイントとなる。

近年のコンピュータでは、プロセッサ・データ処理速度とメイン・メモリのデータ供給速度の差が開く一方なので、プロセッサのレジスタとメモリの間にキャッシュ・メモリを設けることが一般的だ。記憶装置はレジスタから1次キャッシュ、…、メモリ、ディスクと階層を成している。プロセッサに近いほどデータ供給速度が速いのだが、逆に容量は限られてくる。メモリ上のデータすべてをキャッシュに載せておくことはできないので、最近アクセスされた近辺のデータを載せておくという手法がよく用いられる。これは、最近使われた近辺のデータはまたすぐに使われる可能性が高いという経験則に基づいた方法である。

このような理由から、メモリ上で連続しているデータの順次アクセスは速いが、メモリ上に分散しているデータにばらばらとアクセスすると時間がかかるようになっている。例えば、筆者のマシン環境において、長さ400万のint型の配列をある値で初期化する場合、配列の先頭から順に初期化していくと170ミリ秒で済むのだが、とびとびに初期化すると倍以上の360ミリ秒かかってしまう。このように、ディスクよりはメモリ、メモリよりはキャッシュに載っているデータを使えるように考えて、なるべく局所的にメモリにアクセスすることが有効である。

プロセッサについては、前述の例2のところで解説したように、あるJavaプログラムに対してどんな命令が実行されるのかを推測できるとすれば、それが最適化の指針となる。さらに、もう1つ知っておくとよいのは、最近のプロセッサでは条件分岐のコストが高いということだ。より正確に言うと、プロセッサによる分岐先予測が外れた際のペナルティが大きいのである。例えば、Pentium II/IIIの基となっているPentium Proでは、予測が当たると1クロックでジャンプできるのだが、もし外れると9〜26クロックものペナルティが発生する。1万回繰り返されるループならば9,999回は予測が当たるのでまず問題にはならないが、switch文で周期性もなく毎回さまざまに分岐する場合などは、分岐のたびに大きなペナルティが発生することになるだろう。

例3で示したように、OSといったシステム・ソフトウェアに関する理解も、最適化に必ず役に立つものだ。また、Javaプログラムがプロセッサによって実行されるまでには、Javaコンパイラ、JITコンパイラ、Ahead-of-Time（AOT）コンパイラなどさまざまなコンパイラが関係するので、コンパイラに関する理解も最適化の大きな助けとなる。Javaコンパイラはさまざまな制約から大幅な最適化は行わないのだが、JITコンパイラやAOTコンパイラは、これまで他の言語で培われてきた各種の最適化を行う。したがって、それらの最適化手法を知ることで、プロセッサが何を実行するのかを推測できたり、コンパイラが最適化しやすいコードを記述するための指針がわかったりするようになる。一例を挙げると、メソッドをpublicよりはprivateやfinalとすることで、JITコンパイラがそのメソッドの呼び出しを高速化できる可能性が高まる、ということがわかってくるのである。

リスト3：配列の内容をファイルに保存（素直に書いたもの）

```
int[] a = new int[SIZE];
   ……
OutputStream out = new FileOutputStream("ファイル名");
DataOutputStream dout = new DataOutputStream(out);
for (int i = 0; i < SIZE; i++) {
   dout.writeInt(a[i]);
}
```

ここまで述べてきたことは、Javaだけでなく、プログラミング一般に通じる事柄である。もちろんほかに、Javaに固有の有用な知識もたくさんある。バイト・コード、実行時（JIT）コンパイラとインタープリタ、GC、例外、マルチスレッド、同期、さらには標準クラス・ライブラリの理解も役立つに違いない。それらを意識した、Javaに特化した最適化手法については後ほど述べることにする。

最適化へのアプローチ

プログラムを速くしたい。メモリの消費量を少なくしたい。さて、そのためには何をすべきだろうか。やみくもにプログラムを書き換えてもしかたがない。まず、やるべきことは計測である。処理に要する時間や消費メモリ量を把握しないことには、最適化の効果を知ることすらできないからだ。

C言語の関数gettimeofdayに相当するメソッドとして、Java言語にはクラスjava.lang.SystemのメソッドccurrentTimeMillisがあり、ミリ秒単位で現在時刻が得られる。このメソッドを62ページのリスト4のように使用すれば、処理にかかった時間を変数elapsedTimeで得ることができる。なお、時間の計測にはさまざまな方法があり、例えばUNIX系のシステムであればtimeコマンドを使うという手もあるだろう。

プログラムの実行時間については、よく「実行時間の90%がプログラムの10%の部分に費やされる」と言われる（62ページ

の図2)。この割合はプログラムの性質によってさまざまだが、どんなプログラムでもその部分ごとに、実行される頻度や時間が異なるのは確かである。この頻繁に実行される10%の部分を指してホットスポット(hot spot)と言う。HotSpot VMという名前の由来は、ここにある。

プログラムを高速化するにあたり、あまり実行されない部分に力を注いでも効果は薄い。実行時間の90%を占める部分をほっといて、いくら高速化に努めたところで、たとえ残り10%の実行時間をゼロにできたとしても、結局は1割くらいしか速くならない。だが、実行時間の90%を占める部分を2倍速くできたなら、総実行時間は45%短縮され、およそ倍は速くなるだろう。やみくもに取り組んでも速くならない理由はここにある。高速化に取り組む際は、まずは実行時間の長い部分を見つけることが肝要である。

「どの部分が頻繁に実行されているのか」といったプログラムの挙動を調べることをプロファイリングと言う。C言語の場合、大抵はプロファイリングに対応した方法でコンパイルしておく必要があるが、JavaでJVMを使う場合は特別な準備は要らない。いつものようにJavaコンパイラでクラス・ファイルを作っておけばよい。

一般に、実行頻度はメソッドごとの実行時間として得られる。開発ツールの中には所要時間の偏りをわかりやすく示してくれるものもあるが、JDKだけでもプロファイリングは可能だ。以下のようにすることで、log.txtファイルにプロファイリングの結果が残る。なお、ファイル名を指定しなかった場合はjava.hprof.txtファイルに出力される。

```
JDK 1.2(Java 2 SDK 1.2)以降の場合
>java -Xrunhprof:cpu=times,file=log.txt クラス名

JDK 1.1の場合
>java -prof:log.txt クラス名
```

また、appletviewerコマンドでも以下のようにすることでプロファイリングができる。

```
>appletviewer -J-Xrunhprof:…
```

プロファイリング結果を記録したファイルの内容はリスト5のようになる。

この例では、クラスLinpackのメソッドdaxpyが4,950回呼ばれて、総実行時間のおよそ3割を占めていることがわかる。仮にメソッドdaxpyの処理時間を短縮できるとすれば、全体の高速化に大きく寄与するだろう。

また、クラスStringのメソッドcharAtとクラスLinpackのメソッドabsの呼び出し回数が多いこともわかる。これらを呼び出すためのオーバーヘッドを減らせれば、高速化が期待できるというわけだ。もし性能が厳しく問われるのであれば、JITコンパイラによるインライン展開という最適化を促進するために、これらのメソッドをfinalやprivateにするという手がある(実はメソッドabsもクラスStringもすでにfinalである)。性能に対する要求がより厳しいのなら、呼び出し側のソース・コード中にabsやcharAtの処理内容を書いてしまうことも検討に値する。とはいえ、大抵の場合、このようなインライン展開はJIT、AOTコンパイラに任せるのが賢明である。

JDK 1.2(以下、本稿ではJava 2 SDK 1.2のことをJDK 1.2と表記する)以降では、メソッドごとの実行時間だけではなく、ヒープ上にどれだけのオブジェクトが生成されたのかということや、モニタ(ロック)に関する挙動を調べることもできる(詳細は以下のコマンドで表示されるヘルプ・メッセージを参照されたい)。

```
>java -Xrunhprof:help
```

最適化に関するJavaの特徴

コンピュータで実行されるものである以上、Javaプログラムの最適化に必要な多くの知識や理解は他の言語と共通である。しかし、Javaは、これまでJavaの適用範囲において使われてきた他の言語と異なる性質も持っている。ここでは、主にPCより大きい環境を想定して、最適化に関するJavaの特徴を述べてみたい。

最適化について、CやC++、FORTRAN、COBOLなどと比較した場合、Javaは次の特徴を持っている。

●コードの書き方と性能の関係が安定していない。どう書けば速くなるのかを予測しにくい。

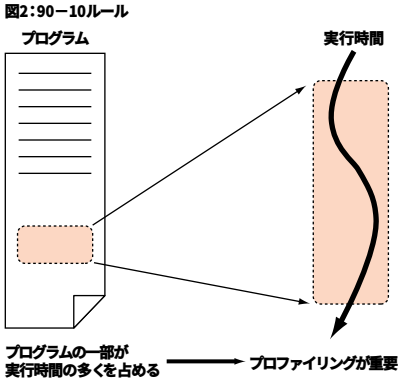
●さまざまな実行方法があり、同じコードでも方法ごとにその性能が変化する。つまり、実行方法やJVM、JITの種類に応じて最適化の指針が異なる。

Javaでは、プログラムの書き方と性能の対応関係が、Cなどに比べてさらに安定していない(図3)。困ったことに、どのように書けば速くなるのか、または遅くなるのかを

リスト4:メソッドcurrentTimeMillisによる処理時間の計測

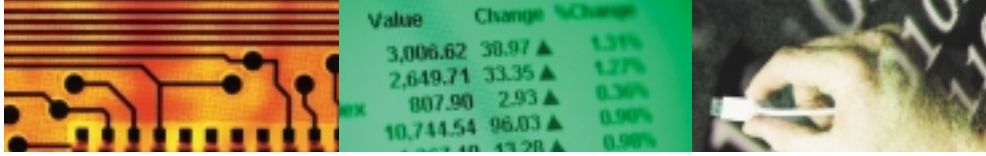
```
long t0, t1, elapsedTime;
t0 = System.currentTimeMillis();

時間を計測したい処理

t1 = System.currentTimeMillis();
elapsedTime = t1 - t0;
```

図2:90−10ルール



リスト5:プロファイリング結果

```
CPU TIME (ms) BEGIN (total = 1020) Thu Jun 1 10:08:00 2000
rank      self    accum   count  trace  method
   1   31.37%   31.37%    4950     28  Linpack.daxpy
   2    3.92%   35.29%       2     10  Linpack.matgen
   3    3.92%   39.22%    2872     45  sun/io/ByteToCharSingleByte.getUnicode
   4    2.94%   42.16%    5808     32  java/lang/String.charAt
  ……
  10    1.96%   54.90%    5049     22  Linpack.abs
  ……
```



予測しにくいのである。その理由としては、踏まえるべき知識がCよりも多いために、現時点では熟練者が少なく、ノウハウが浸透していないせいもあるかもしれない。

しかし、理由はそれだけではない。JITコンパイラの有無やAOTコンパイラの使用などさまざまな実行方法があり、同じプログラムでも実行方法によって性能が変わる。さらに、JVMやJITコンパイラの種類によっても性能が変化する。そして、「ある実行方法では速かった書き方が、別の実行方法では遅い」ということが当たり前のように起こってしまうのである。

その具体的な例を挙げよう。配列のコピー処理を考える。byte型の配列src[]の中身をdst[]にコピーするコードは、素直に書くとリスト6-①のようになるだろう。このコードでは、終了条件を添え字の値で判断している。

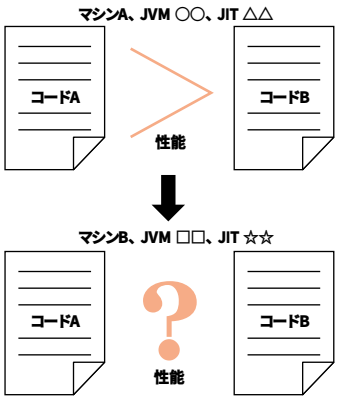
また、リスト6-②や③のような方法もある（通常、このように記述することはないだろう）。これらの方法では、範囲外の不正な添え字で配列にアクセスした場合に発生する例外を利用している。

配列のサイズを16MBとして、筆者の環境（Pentium II/333MHz、Linux）で上記3種類のコードの実行時間を計測してみた。以下、その結果を基に、JITコンパイラの有無やJITの種類が性能にどう影響を与えているのかを見ていこう。

まずはIBMのJDK 1.3.0（以下、IBM JDK 1.3.0）での結果である（表2-①）。ここで注目していただきたいのは、添え字を使った方法と例外を使った方法（1）（2）の速さが、JITコンパイラの有無によって入れ替わっていることだ。インタープリタでは例外を使ったほうが速いのだが、JITコンパイラでは添え字を使ったほうが速い。

次に、サン・マイクロシステムズのJDK 1.3.0（以下、Sun JDK 1.3.0）での結果を見てみよう（表2-②）。IBM JDK 1.3.0のときとは異なり、JITコンパイラの有無で性能が逆転することはなかった。しかし、IBM JDKのJITとSun JDKのJITを比較すると、

図3：コードの書き方と性能の関係



添え字を使った方法と例外を使った方法（2）の結果が逆転している。

以上、JITコンパイラの有無やJVM、JITコンパイラの種類によって性能の優劣が入れ替わる例を見てきた。さて、この性能差、また逆転はどうして起きたのだろうか。それを詳細に調べるには、与えられたプログラムに対してプロセッサが何を実行したのかを知る必要がある。つまり、「JVMが何をしたのか」「JITコンパイラがどんなコードを生成したのか」である。

だが、残念ながら一般にはJITコンパイル結果を見ることはできない（64ページの図4）。javapコマンドなどでクラス・ファイルを逆アセンブルして見ることができるのはバイト・コード命令までであり、プロセッサが実際に何を実行したのかはわからないのである。この点が、コンパイル結果を逆アセンブルしてプロセッサの命令列を直接見れるCやC++、FORTRANとは異なるところである。

もっとも、JITコンパイラのソース・コードが手に入ればどんなコードを生成するのかを知ることもできようし、JITコンパイラの内部について記した記事や論文もあるにはあるので、手がかりは皆無ではない。また、AOTコンパイラを使う場合は実行前にそ

リスト6：配列のコピー

```
①添え字を使った方法
for (int i = 0; i < SIZE; i++) {
   dst[i] = src[i];
}

②例外を使った方法(1)
int i = 0;
try {
   while (true) {
      dst[i] = src[i];
      i++;
   }
}
catch (ArrayIndexOutOfBoundsException e) {}

③例外を使った方法(2)
i = 0;
while (true) {
   try {
      dst[i] = src[i];
      i++;
   }
   catch (ArrayIndexOutOfBoundsException e) {
      break;
   }
}
```

のプロセッサ用のコードが得られるので、ライセンス条項に抵触しないのなら逆アセンブルによる調査が可能である。

挙動の調べやすさという点では、インタープリタやそれを含むJVMについても状況は芳しくない。現在手に入るJVMのほとんどは、ソース・コードを入手できない。「Japhar」や「Kaffe」といったソース・コードから公開されているJVMもあるし、サンプル実装であるClassic VMやHotSpot VMの旧バージョン1.0.1については、サンもソース・コードを公開してはいる。しかし、JVMの性能でトップランナーであるIBMやサンなどは内部で大幅にJVMを改良しており、そのソース・コードは公開されていない。残念ながら、ブラックボックスが多いのである。

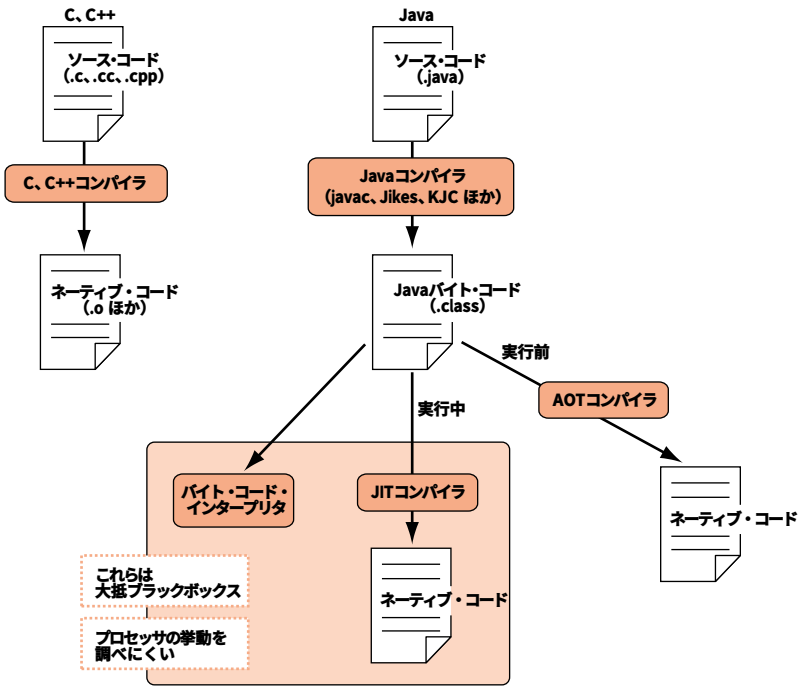
また、適応的コンパイルや同期コストの削減手法など、現在盛んに研究されているテクニックがJVMやJITに取り入れられてきているために、プログラムの書き方と性能の対応関係は今なお変化し続けているという事情もある。

このように、Javaでは、頑張ってカリカリ調整して速くしたとしても、その最適化が他の実行方法や将来の環境でどのくらい有効なのかはかなり不透明である。速くし

表2：3つの方法による配列（サイズは16MB）のコピー

	①IBM JDK 1.3.0での実行時間（ミリ秒）			②Sun JDK 1.3.0での実行時間（ミリ秒）		
	添え字	例外（1）	例外（2）	添え字	例外（1）	例外（2）
JITコンパイラ	410	470	520	780	780	740
インタープリタ	4,540	2,570	2,570	4,810	3,550	3,550

図4：Javaプログラムの実行形態



たつもりが他の環境では遅かったり、その最適化が3カ月後にはJITコンパイラ側で行われたりするかもしれない。

Javaでは、可読性などを犠牲にしてまでの極度の最適化は、プログラマーの手で行ってもしかたのないことが多いのである。もちろん、特定の実行環境やソフトウェア上で最大限の性能を発揮するコードを書くことが必要な局面はあるだろう。その場合は、その環境をよく知り、実際の環境でまめに効果を測定しながら最適化を進めていくことが大切である。

Javaの特徴に応じた最適化

以上、最適化に関するJavaの性質を述べてきたが、ここからは、より詳細にJavaの特徴とそれに応じた最適化の指針を見ていく。想定する環境はやはりPCより大きいコンピュータとし、主にJVM上のJITコンパイラを用いた場合の実行を考えてみる。もちろん、JITコンパイラの有無に関係しない指針も多いし、AOTコンパイラにはほぼすべての指針が通用する。

インスタンスの生成

インスタンスの生成は重い処理である。Javaでは配列もインスタンスなので、配列の生成も同様だ。「重い」や「軽い」というのは相対的にしか言えないことなので、具体的な例を挙げてみると、筆者の環境では100万回の整数除算が90ミリ秒で済むのに対し、同じ回数のインスタンス生成（クラスjava.lang.Objectのインスタンス生成）には890ミリ秒かかる。

インスタンスの生成は、それ自体が重いだけではなくGCにも影響を与える。インスタンスを多量に生成すると、当然GCが起こりやすくなる。例えば、GUI中心のプログラムでアイドル時間が十分にあるとしよう。プログラムが自身の処理をしていない間にGCが起こっても、その使用感には影響がないかもしれない。しかしアイドル時間がない種類のプログラムの場合は、GCがプログラム本体の処理時間を奪うことになる。このように、インスタンスの生成量は消費メモリ量だけではなく、プログラムの性能にも影響を与えうる。

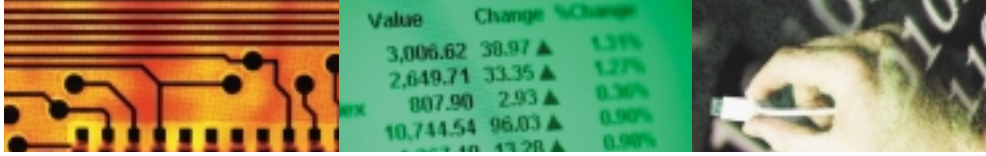
また、悪いことに、インスタンスの生成はJITコンパイラで高速化されにくい。インスタンス生成はJVM自体が初めから有している機能であるため、大抵のJVMではJITコンパイルの対象とならないからである。JITコンパイラが生成自体を省いてくれたり、インスタンスが本当に必要になるまで生成を遅らせてくれる場合もあるにはあるが、それらが行われるケースは非常に限られており、あまり期待はできない。

よって、プログラマー自身の手によってインスタンス生成の回数を減らすことは有効な最適化である。そのアプローチはさまざまだが、例えば（あまりお勧めはできないものの）1プロセッサで優れた性能を発揮すればよいプログラムの場合には、リスト7に示す方法がある。メソッド呼び出しのたびにインスタンス（buffer[]）を生成しているコード（リスト7-①）を、synchronizedを使って書き換えるのである（リスト7-②）。

配列（buffer[]）をインスタンス変数からクラス変数に変更することで、生成の回数をプログラムの実行につき1回にまで削減することができる。ただし、生成した配列をすべてのスレッドで共有して使い回すことになるので、メソッドをsynchronized宣言している。このsynchronizedによる同期のコストは増えるが、少なくとも現在はインスタンス生成のほうが重いことが一般的なので、大抵は速くなる。

生成したインスタンスを使い回すことで生成を回避する手法としては、オブジェクト・プールというものがよく知られている。この手法はクラスjava.lang.Thread、つまりスレッドに対して非常によく用いられ、特にスレッド・プールと呼ばれている。スレッドに用いられることが多い理由は、スレッドの生成コストが他の多くのクラスよりもさらに大きいからである。

いわゆるサーバ側プログラムでは、ネットワーク経由の要求に対してスレッドを割り当てていくことが多いので、スレッド・プールは必須のテクニックとなっている。どんな処理でも受け付けられる、汎用的で高効率のスレッド・プールを書くことは、それ自体が興味深い演習問題である。ぜひ一度は



挑戦していただきたい。

性能面で鍵となるのは、「いつ、どの時点で下位のスレッドが作られるのか」である。クラスThreadのインスタンスは、自身に対応し、より下位に位置する、Cライブラリまたはオペレーティングシステムのスレッドを持つ。そうした下位のスレッドを生成する処理がまた重いので、性能を求めるのであれば下位のスレッドが生成された状態までもっていってからプールにためておくべきである。

注意してほしいのは、クラスThreadのインスタンスを生成しただけでは、下位のスレッドが生成されたとはかぎらないということだ。下位のスレッドはメソッドstartが呼ばれた時点で生成されれば間に合うので、それまで下位のスレッドは生成されていないかもしれない。そのため、プールにためておく前に、スレッドに対してメソッドstartまで呼んでおくとよい。

配列アクセス

Javaでは、配列の要素へのアクセスがCと比べて重い処理となっている。その理由はいくつかあり、インスタンスである配列の要素へのアクセスの際にnullかどうかのチェックが必要なのも、その1つである。もしnullであれば、例外を発生させなければならない。このnullチェックは、環境によってはOSの機能で行え、オーバーヘッドとはならない場合も多い。また、ご存じのとおり、Javaでは添え字の範囲チェックが行われる。添え字がゼロ未満、または配列長以上である場合は例外が発生する。

これらのチェックはいずれも、JITコンパイラがデータ・フロー解析やloop versioningによって取り除いてくれることがある。「配列の境界チェックを一切行わない」という設定を受け付けるJITやAOTコンパイラもあるかもしれない。それはすでにJVMの仕様を逸脱しているし、範囲外アクセスによるクラッシュを招くかもしれないが。

配列のアクセス・コストを減らすためにプログラマーができることはそう多くはないが、指針はある。まず、チェックが不要であることをJITコンパイラやAOTコンパイラが読み切れるようなコードを書くことである。例えば、リスト8に示すコード中のarray[i]において、iが配列の境界を越えることはありえるだろうか。答えは「ありえない」である。もしJITやAOTコンパイラがそこまで解析してくれれば、境界チェックは省かれるかもしれない。しかし、仮にリスト8内の変数sizeがローカル変数ではなくインスタンス変数、もしくはstatic変数だったらどうだろうか。ほかのスレッドから変更される可能性が生まれる。forループの実行中に、もしほかのスレッドがsizeの値を増やしたとすると、iの値が配列サイズ以上になって例外が発生するかもしれない。そうした危険がある以上、JITやAOTコンパイラとしては境界チェックを省くわけにはいかなくなる。

この例は決してやさしくはないが、人間にとってわかりやすく、明らかに最適化できそうなコードは、コンパイラにとっても解析が容易であることが多い。わかりやすいコードを書くことが、結局はコンパイラを助けることになる。

プログラマーにできることは、ほかにもあるにはある。例えば、配列の長さがたかが知れている場合は、そもそも配列ではなくスカラ（配列でない）変数にするという方法もある。つまり、a[3]の代わりにa0、a1、a2の3つのスカラ変数を使うのである。もっとも、可読性を損なってまでこの方法を採用すべきかは、慎重に検討せねばなるまい。

多次元配列の生成とアクセス

多次元配列についての処理が重いのは、1次元配列のアクセスが重いという理由だけによるものではない。

正確に言うと、Javaに多次元配列はない。2次元配列のように見えるもの（a[][]）は1次元配列の配列であり、3次元配列に見えるもの（a[][][]）は1次元配列の配列の配列である（66ページの図5）。しかし、ここでは便宜上、「配列の配列の…配列」を多次元配列と呼ぶことにする。多次元配列は以下のようにして生成できる。

リスト7：インスタンスの生成回数を減らすことによる最適化

```
①最適化前のコード
void aMethod() {
   byte[] buffer = new byte[SIZE];
   bufferを使った目的の処理
}

②最適化後のコード
static byte[] buffer = new byte[SIZE];
synchronized void aMethod() {
   bufferを使った目的の処理
}
```

リスト8：配列arrayへのアクセス

```
int size = ...;
int[] array = new int[size];
for (int i = 0; i < size; i++) {
   array[i] = ...;
}
```

```
int[][] array = new int[3][4];
```

これは、以下のコードと等価である（厳密に言うと、ヒープの空き不足で例外が発生した場合の状態は異なる）。つまり、4つのインスタンスが生成される。

```
int[][] array = new int[3][];
array[0] = new int[4];
array[1] = new int[4];
array[2] = new int[4];
```

このように、多次元配列を生成すると、ただでさえ重いインスタンスの生成が何度も行われることになる。例えば、new int[5]、new int[5][5]、new int[5][5][5]ではそれぞれいくつのインスタンスが生成されるのかを考えてみてほしい。答えは6、31、156である。new int[5][5][5][5]に至っては、781ものインスタンスが生成される。

また、多次元配列はアクセスも重い。Cでは添え字を基にアクセス先のアドレスを計算してから要素にアクセスできるが、Javaでは最悪の場合、1次元配列へのアクセスを次元数の回数繰り返すことになる。そうなると、メモリ上で離れた地点をとびとびにアクセスすることでキャッシュ・ミスを起こしかねないうえに、配列アクセスにおける境界チェックなどのオーバーヘッドが次元数倍になる。もっとも、同じ要素への繰り返しのアクセスや隣接する要素への順次アクセスなどにより、JITやAOTコンパイラがオーバーヘッドを削減できる場合もある。

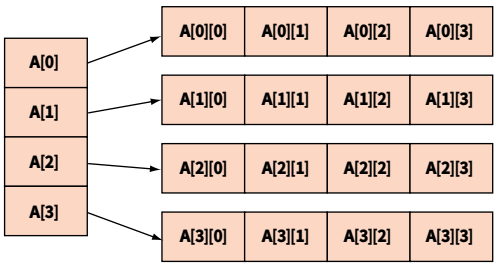
では、プログラマーに何ができるかと言え

図5：多次元配列のメモリ・レイアウト

C、C++、FORTRANの2次元配列

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]
A[3][0]	A[3][1]	A[3][2]	A[3][3]

Javaの2次元配列



リスト9：2次元配列から1次元配列への書き換え

①2次元配列A、Bの乗算

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      double sum = 0;
      for (k = 0; k < n; k++) {
         sum += A[i][k] * B[k][j];
      }
      C[i][j] = sum;
   }
}
```

②①のA、B、Cを1次元配列に書き換える

```
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
      double sum = 0;
      for (k = 0; k < n; k++) {
         sum += A[n*i + k] * B[n*k + j];
      }
      C[n*i + j] = sum;
   }
}
```

ば、次元数の削減という方法が挙げられる。例えば、a[100][2][5]という配列があったとしよう。この配列の代わりに、要素数は変えずにa[100][10]という配列を用意する。そして、a[i][j][k]とアクセスしていたところを、a[i][5*j+k]とアクセスするのである。もっと極端に、a[1000]をa[10*i+5*j+k]とアクセスしてもよい。

実際、筆者はこの方法の効果を調べてみた。行列積で2次元配列を1次元配列に書き換え、実行時間の違いを見てみたのである。元のコードは前述のリスト1-①と同じものだ（**リスト9**-①）。AとBを乗じた結果をCに格納するという処理である。配列A、B、Cを2次元配列から1次元配列に書き換える（リスト9-②）。

行列のサイズnを500とした場合に、筆者の環境（IBM JDK 1.3.0）での実行時間は以下のとおりであった（単位は秒）。1次元配列に書き換えたことで添え字の計算のための乗算と加算が増えているにもかかわらず、実行時間は短くなった。

```
2次元配列:9.96
1次元配列:9.27
```

多次元配列のアクセスが重いという問題は、Javaを数値計算や高性能計算に使おうというコミュニティ「Java Grande Forum」で議論されている。その結果、多次元配列の新しいAPI（例:クラスdouble Array2D）と、そのAPI中のクラスと対応づけられるa[i,j]という形式の新しい文法が考えられた。それぞれすでに「Java Specification Request（JSR）」として提案され、仕様策定手続きである「Java Community Process（JCP）」に沿って標準化の作業が行われている。ただし、もし仮に採用されるとしても、2001年にリリースが予定されているJDK 1.4（Java 2 SDK 1.4。開発コード名はJ2SE Merlin）には間に合わないので、当分先の話である。

変数の種類とアクセス・コスト

Javaの変数にはいくつかの種類がある。JVMの仕様書によれば、クラス変数、インスタンス変数、配列の要素、メソッド引数、コンストラクタ引数、例外ハンドラ引数、ローカル変数の7種類である。ただ、実際のところ、JVM内ではメソッド引数、コンストラクタ引数、例外ハンドラ引数はローカル変数の一種となっているので、実質的にはクラス変数、インスタンス変数、ローカル変数、配列の要素の4種類と考えられる。

変数アクセスのためのバイト・コード命令についても、これら4種に対応するものがそれぞれ用意されている。例えば、クラス変数の読み出しであればgetstatic命令、といった具合である。

変数の種類が違えば、読み書きのコストも違ってくる。どのくらい違うものなのかを調べるために実験を行った。以下に示すコードにおいて、int型変数のi、n、sumがそれぞれローカル変数、インスタンス変数、クラス変数の場合に、各変数へのアクセス時間がどうなるのかを計ってみた。

```
sum = 0;
for (i = 0; i < n; i++) {
   sum += i;
}
```

なお、JITコンパイラによる最適化でループ・ボディ（sum += i）の処理が削除されてしまうことのないよう、計測コードのうしろに変数sumを使用する文を記述した。また、繰り返し回数nを1億とし、筆者の環境（Pentium II/333MHz、Linux）でIBM JDK 1.3.0とSun JDK 1.3.0を使って計測した。

結果は表3のとおりである。すべての場合においてローカル変数が最も速かった。インスタンス変数とクラス変数を比較すると、Sunインタープリタを除き、インスタンス変数のほうが良い結果を示している。ただし、インスタンス変数とクラス変数のどちらへのアクセスが重いかは実行環境やコードによるので、一概には判断できない。

プログラマーとしては、「ローカル変数は比較的アクセス・コストが低い」ということを知っておけばよい。すると、高速化の方法としては、例えば頻繁にアクセスする変数はローカル変数にしておいたり、インスタンス変数やクラス変数をいったんローカル変数に代入してから頻繁にアクセスしたり、といったことが考えられる。しかし、こうした書き換えは、やはりコードの理解のしやすさや可読性を損ねることが少なくないし、JITコンパイラがインスタンス変数やクラス変数をローカル変数として扱うといった最適化を行ってくれる場合もある。損得をよく考えて採用すべきであろう。

では、なぜ、こういったアクセス・コストの違いが出てくるのだろうか。そこで、今度はアクセス・コストの観点から上記3種の変数を比較してみる。

インスタンス変数では、アクセスの際に対

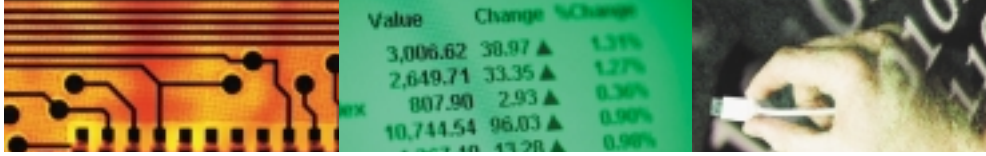


表3：変数アクセス（単位はミリ秒）

	ローカル変数	インスタンス変数	クラス変数
IBM JITコンパイラ	620	920	1,220
IBMインタープリタ	19,370	47,450	49,850
Sun JITコンパイラ	2,120	2,470	2,740
Sunインタープリタ	20,460	63,170	47,230

象インスタンスがnullかどうかのチェックが必要となる。もっとも、nullチェックにOSの機能を使える場合、これはオーバーヘッドとはならないし、JITやAOTコンパイラがチェックを省いてくれることもある。バージョン1.2.2までのWindows用JDKのようにClassic VMを使用している場合には、メモリ（ハンドル）の内容を見てそこに書いてあるアドレスを参照するという間接参照が行われ、オーバーヘッドとなる。

一方、クラス変数へのアクセスではnullチェックは不要である。また、インスタンス変数とは異なり、JITコンパイルの時点で変数の格納先アドレスを確定できるので、実行時にアドレス計算が要らないという長所がある。もっとも、アドレス計算はもともと大したオーバーヘッドではない。

逆に短所としては、クラス変数へのアクセスが、ごく稀にそのクラスの初期化を引き起こしうるという点が挙げられる（図6）。クラスの初期化とは、「static initializer (static {… })」の実行やクラス変数の初期化を指す。JVM仕様はクラスの初期化が行われるタイミングを厳密に定めており、そのうちの1つがクラス変数へのアクセスなのである。ロード後に1度だけ行われるべき初期化をどのように効率良く実装するかは、インタープリタやJITコンパイラ、AOTコンパイラを問わず、Javaの実行系における共通の課題なのだ。クラス変数へのアクセスのたびに初期化済みかどうかをチェックするという方法は、いかにもむだである。その実装方法によっては、2度目以降のアクセスにもある程度のオーバーヘッドが課せられてしまうことになる。

最後はローカル変数である。ローカル変数には上述したオーバーヘッド、すなわちインスタンス変数やクラス変数のようなオーバーヘッドがない。さらにもう1つ良い点がある。JIT、AOTコンパイラの立場から見ると、Java言語やJVMの仕様を侵すことなく安全にレジスタに割り付けられるのはローカル変数だけである。詳細な理屈は省くが、クラス変数とインスタンス変数は自スレッドだけでなく他のスレッドからもアクセスされるおそれがあるため、レジスタに割り付けることが難しい。インスタンス変数をローカル変数と同様にレジスタに割り付けようとする最適化を行うコンパイラもあるが、常にうまくいくとはかぎらない。

複数のスレッドから共有される可能性があるというクラス変数、そしてインスタンス変数の性質は、JIT、AOTコンパイラによるその他の最適化を妨げかねない（図7）。配列アクセスに関する説明の個所で述べたように、ローカル変数でないばかりに省けるはずの配列境界チェックを省けなくなることもありうる。

ほかにも、同期、ロックを省くチャンスが減ることもありうる。以下のコードを見てほしい。

```
Vector vec = new Vector();
……
vec.addElement(…);
vec.addElement(…);
vec.addElement(…);
……
```

ここで注意していただきたいのは、クラスVectorのaddElementはsynchronizedメソッドであるということだ。そのため、このコードを実行するスレッドは、メソッドaddElementを呼び出すたびにVectorのインスタンスのロックを取得し、呼び出しから戻る際に解放する。しかし、このロックの取得、解放は実は必要ではない。vecはローカル変数なので、vecが指すインスタンスを他のスレッドが参照することはありえないからである。複数のスレッドによるVectorのインスタンス操作を防ぐことが目的のロックなので、1つのスレッドからしか参照できないことが明らかなインスタンスをロックしてもむだである。JITやAOTコンパイラがこうしたむだなロックを省いてくれることは十分に考えられる。

ところが、変数vecがローカル変数でなかったらどうだろうか。vecが指すインスタンスを他のスレッドが参照してしまう可能性が出てくる。すると、ロックはむだとはならないので、コンパイラとしてはロックを省くわけにはいかなくなる。

このように、変数の種類は、コンパイラによる最適化に対してさまざまな影響を与えうる。プログラマーとしては、JITやAOTコンパイラはローカル変数についての最適化を施しやすいと覚えておいてほしい。

図6：クラス変数をアクセスした際の処理の流れ

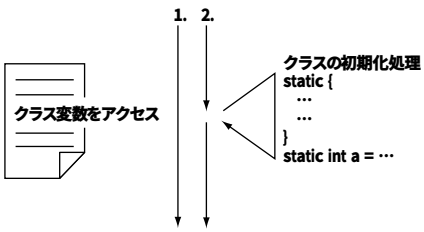
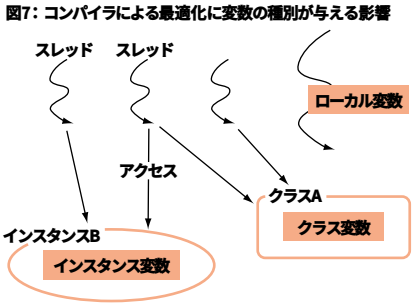


図7：コンパイラによる最適化に変数の種別が与える影響

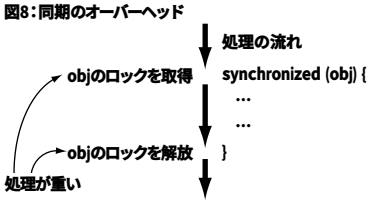


インスタンス変数とクラス変数は複数のスレッドからアクセスされうる
ローカル変数は1つのスレッドからしかアクセスされない

同期処理

同期、すなわちインスタンスに対するロックの取得、解放処理は比較的重い処理である（68ページの図8）。高性能を目標としたJVMやコンパイラの開発者は皆、「同期処理を軽くしよう、可能なら削除しよう」

図8：同期のオーバーヘッド



リスト10：同期処理

```
①個別にロックを取得、解放する
synchronized (obj) { 処理A }
処理B
synchronized (obj) { 処理C }

②複数回の同期を一度にまとめる
synchronized (obj) {
    処理A
    処理B
    処理C
}
```

リスト11：アクセス修飾子のコスト計測用コード

```
①呼び出されるメソッド
int callee(int value) {
   return value + 1;
}

②呼び出し側のコード
int n = 30000000;
int sum = 0;
for (int i = 0; i < n; i++) {
   sum += callee(i);
}
```

と努力している。

同期処理に関してプログラマーができることは、synchronizedブロックへの突入とsynchronizedメソッドの呼び出し回数、つまり同期の回数を最小限にとどめることである。

とはいえ、複数スレッド間の排他制御がどうしても必要な局面は残る。それでもさらに同期回数を減らしたければ、複数の同期を一度にまとめるという方法もある。具体的には、何度もロックを取得、解放する（リスト10-①）代わりに、複数回の同期を一度にまとめてしまうのである（リスト10-②）。

処理A、B、Cを含むメソッド自体をsynchronizedにしてしまう方法も考えられる。ただし、この方法は同期のコストを削減する代わりに、ロックを長時間保持し続けることで他のスレッドの動作チャンスを奪いかねないということを知っておくべきである。また、リスト10-②の例では、本来は排他制御する必要のない処理Bについてもロックを保持したまま処理するので、コードを見た別のプログラマーが、排他制御が必要なものと勘違いする危険もある。

同期に関する問題は、何もプログラマーによって書かれるコード上だけに存在するわけではない。Javaの標準クラス・ライブラリは、すでに多量のsynchronizedメソッドを含んでしまっている。その例としてJDK 1.3.0のクラスjava.util.Vectorのメソッドを見てみると、45個あるpublicメソッドのうち、実に30ものメソッドがsynchronizedメソッドとなっている。排他に無頓着なプログラムの場合でも、複数のスレッドが走った際に問題が起きないよう、最大限安全なように書かれているのである。1つのスレッドしか走らないプログラムでは、それこそ、これらのsynchronized指定はまったくのむだでしかない。

プログラマーとしては、このようなむだな同期はJIT、AOTコンパイラが削除してくれることを期待したい。そのためにできることは、変数の種類についての説明の中で述べたように、むだであるということをコンパイラが解析しやすいようにコードを書くことくらいである。

同期のコストをぎりぎりまで切り詰めたい場合は、標準ライブラリの中のクラスをそのまま使うのではなく、代替クラスを自分で用意するという手段もある。例えば、Vector.javaを基にsynchronizedの記述を削除した代替クラスを用意して、それを使うのである。この場合、当然であるが、複数スレッドからアクセスした場合に問題が起きないことを保証するのはプログラマーの責任となる。必要ならば排他制御は自分で行わなければならない。

メソッドのアクセス制御と呼び出しコスト

メソッドのアクセス修飾子（access modifier）であるpublic、protected、privateやfinal、static宣言は、メソッドの呼び出しコストに影響を与える。ここでは、コストがどのくらい違うものなのかを見てみよう。リスト11-①のメソッドcalleeを3,000万回呼び出し、所要時間を計測してみた。

メソッドcalleeのアクセス修飾子をさまざまに変え、筆者の環境でIBM JDK 1.3.0とSun JDK 1.3.0で実験した結果が表4である。表中で「static」とあるのは「public static」、「final」とあるのは「public final」とした場合のものだ。また、「interface」とあるのは、インタフェース経由でpublicメソッドを呼び出したときの結果である。

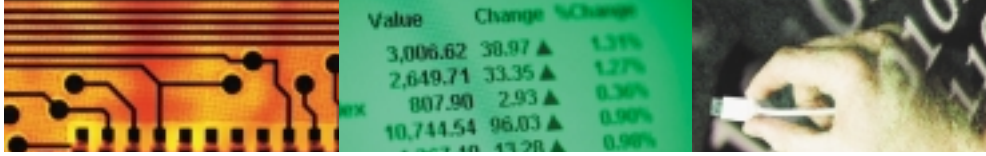
表4を見ると、インスタンス・メソッド（static以外）の中では、どの環境でもprivateメソッドの呼び出しが速い。また、finalを付けることで多くの環境で速くなっている。一方、クラス・メソッド（static）は、JITコンパイラではprivateメソッドとほぼ同等であり、インタープリタではprivateメソッドよりもさらに若干速いという結果となった。

この結果から導ける指針は、「アクセス修飾子は極力狭く、privateに近くして、可能ならばfinalを付ける」ということである。また、インスタンス・メソッドをクラス・メソッドに変更してしまうというかたちでの高速化も考えられる。しかし、それは設計の問題でもあるため、慎重に検討すべきであろう。

なぜ、privateでないインスタンス・メソッド（public、protected）の呼び出しコストが高いのかは、オーバーライドと動的呼び出し（virtual invocation、dynamic dispatch）に関係がある。例えば、クラスAがあったとして、A型の変数に対してインスタンス・メソッドfooを呼び出すコードを書いたとする。もしAのサブクラスがfooをオーバーライドしていた場合、その呼び出しで実際に呼び

表4：メソッド呼び出し（単位はミリ秒）

	public	protected	private	final	static	interface
IBM JITコンパイラ	840	840	380	380	380	380
IBMインタープリタ	20,030	20,020	18,680	20,040	15,100	20,000
Sun JITコンパイラ	1,550	1,720	380	450	350	1,540
Sunインタープリタ	16,160	16,160	15,890	15,890	14,800	17,150



出されるメソッドは対象インスタンスのクラスによって決まる。そのクラスは実行時にならないとわからないので、実行時に、呼び出すべきメソッドを決定するためのオーバーヘッドがある。例えば、単純な実装では、実行時にクラスの階層をたどって調べたり、表をひいたりするので、その処理がオーバーヘッドとなる。

それに対し、privateメソッドはオーバーライドされようがないので、プログラマーがコードを書いた時点で、どのメソッドが呼び出されるかが決まっている。この点、finalメソッドやクラス・メソッドもprivateメソッドと同様である。実行時になってみないと呼び出すべきメソッドが決まらないpublicやprotectedと、コードの字面で決まるprivateやfinal、staticとの違いは、表4の結果にもよく表れている。

また、動的呼び出しは、上記のようなオーバーヘッドがあるだけでなく、JITやAOTコンパイラによる各種最適化も制限してしまう。例えば、実行時にならないと呼び出すべきメソッドが決まらないため、インライン展開が制限される。インライン展開（inlining）とは、呼び出し元メソッドの中に呼び出し先メソッドの処理を展開してしまうことで、メソッド呼び出しのオーバーヘッドを削減するという、コンパイラによる最適化のことだ。インライン展開の本当のいちばんのメリットは、呼び出し元と呼び出し先をまたいでの大変な解析を行わずして、呼び出し元、呼び出し先をまたいで他の最適化を施せるという点である。

動的呼び出しに起因するこれらの問題は、Javaにかぎらず古くからオブジェクト指向言語に共通しており、これまでも多くの研究がなされてきた。JavaのJIT、AOTコンパイラやインタープリタにも、それらの成果が取り入れられている。

ただし、Javaの性質が原因で、そうした既存のテクニックを単純には使えない場合も多い。IBMのJITコンパイラは、クラス階層解析を行うことで、呼び出すべきメソッドをJITコンパイルの時点でなるべく1つまで絞ろうとする。その時点でメソッドがオーバーライドされているか否かを解析するのである。もし、実際に呼び出されるメソッドを1つにまで見切ることができれば、動的呼び出しの問題は回避できるわけだ。しかしここで、Javaのとある性質がこのテクニックの障害となる。Javaでは、プログラムの実行中に、必要となったクラスが順次JVM上に動的にロードされていく。そのため、JITコンパイルの時点ではオーバーライドされていないと判定できたメソッドが、動的ロードによってオーバーライドされてしまうかもしれない。そこでIBMのJITコンパイラは、動的ロードによってメソッドがオーバーライドされた場合には、コンパイル結果のコードを書き換えて対処する。

このように、同期処理やメソッド呼び出しのコンパイル手法は今もなお研究が盛んで、実行系の開発元ごとに差別化を図れる部分である。Javaの実行系によって性能が異なったり、新しいバージョンで大きく性能が変わったりする部分なので、Javaプログラマーとしては手もとの環境での性能が普遍的なものだとは考えずに、注意深く動向を観察していくことが必要だろう。

例外に関するコスト

ここでは、例外に関するコードの書き方、すなわちtry～catchブロックやメソッドのthrows宣言、例外のthrowが性能に与える影響について見ていこう。

まず、Javaではtry～catchを記述しただけではペナルティとはならないのが普通である。これは、例外に関する記述だけで性能が低下するC++とは異なる点だ。

『最適化に関するJavaの特徴』の節で、配列コピーの終了を繰り返し回数で判定するか、範囲外アクセスの例外で判定するかについて比較した。例外を使った方法の（1）と（2）の違いは、（1）ではtry～catchがループの外にあるのに対し、（2）ではループ内に書かれている点であった。もし、try～catchの記述が何らかのペナルティとなるのであれば、その分（2）のほうが性能は悪くなるはずである。しかし、実験結果では、インタープリタにおける（1）と（2）の性能は同じであり、JITコンパイラを使った場合はJITの種類によって優劣が入れ替わった。IBMのJITコンパイラでの結果は説明できていないが、少なくともそれ以外のすべての実行系では、try～catchの記述はペナルティとはなっていない。

try～catchに対し、例外のthrowはそれなりに重い処理である。例外はクラスjava.lang.Throwableか、もしくはそのサブクラスのインスタンスなので、例外がthrowされるとインスタンス生成が起こる。コンパイラがインスタンス生成を省いてくれる可能性があるとはいえ、あらゆる場合に期待できるわけではない。また、例外がthrowされると、次に実行をどこに移したらよいのかを決めるためにJVMは表を調べるが、これも重い処理である。

これらのペナルティを削減するための最適化も考えられてはいるが、実行系やコンパイラの開発者としては、「例外はあくまで例外的な状況にだけthrowされるのが普通だろう」と考えがちである。ごくたまに例外的に使われる機能の高速化に力を注いでも効果は低いというのは、本稿ですでに述べたとおりである。例外のthrow時の処理の高速化は、JVMやJIT開発者の間での優先順位が高いとは言えず、期待薄である。

try～catchの使用は性能に影響を与えにくいとはいえ、例外がプログラムの性能にまったく影響を与えないというわけではない。例外をthrowしうる処理がコード中にあるだけで、JITやAOTコンパイラによるある種の最適化は制限されてしまうのである。プログラマーにできるのは例外をthrowする可能性のあるコードを書かないことだが、残念ながらそれはほとんど無理である。

例えば以下のコードがあったとする。

```
処理A
例外をthrowするかもしれない処理B
処理C
```

プロセッサを効率良く動かすために、JITやAOTコンパイラとしては処理Cの一部を処理Bの前に移動させたいという状況はよくある。しかし、Java言語の仕様では、処理Bが例外をthrowした場合に処理Cの一部がすでに行われてしまっているということは許されない。そのため、コードの移動がかなり制限されるのである。

困ったことに、例外をthrowしうる処理は何もthrowと書かれたコードだけでなく、どんなJavaプログラムの中にもあふれている。インスタンス変数へのアクセスはNullPointerExceptionをthrowしうるし、配列の要素アクセスはArrayIndexOutOfBoundsExceptionを発生させる可能性がある。メソッドを呼び出すと、そのメソッドの中から例外がthrowされるかもしれない。

先に述べたクラスの動的ロードも含めて、Javaには、JITやAOTコンパイラによる最適化の障害となる仕様や性質が数多くある。仕様によっては、それを変更しようという動きもなくはないが、厳しい仕様を満たしつつ、どこまで最適化できるかということが、コンパイラ開発者の間ではチャレンジとなっているのである。

Javaコンパイラの違い

ここで言及するのは、JITやAOTコンパイラではなく、Javaソース・コード(.java)をクラス・ファイル(.class)に変換するJavaコンパイラである。Javaコンパイラと聞いてまず思い浮かぶのは、JDKに含まれているjavacであろう。だが、世の中にはjavac以外にも多くのJavaコンパイラが存在する。例えばIBMの「Jikes」は、C++で書かれた、高速さと言語仕様への忠実さが特徴のJavaコンパイラである。また「Kopi Java Compiler(KJC)」は、javacと同様にJava言語自身で書かれたJavaコンパイラで、トランスバーチャル・テクノロジーズのJVMである「Kaffe 1.0.5」のデフォルトのJavaコンパイラとなっている。

コンパイラごとに異なるのは、何もコンパイルにかかる時間ばかりではない。コンパイラの種類やバージョンによって、コンパイル結果のバイト・コード命令列が異なってくる。

例えばリスト12では、メソッドcallerがメソッドcalleeを呼び出している。privateメソッドであるcalleeがオーバーライドされることはありえないので、Javaコンパイラがcalleeの処理をcaller中にインライン展開してしまっても問題はない。

各種Javaコンパイラがこの最適化を行うかどうかを調べたところ、JDK 1.1.8のjavacだけが行い、JDK 1.2.2と1.3.0のjavac、およびJikes 1.0.5は行わなかった。定数伝搬など他の2、3の最適化手法についても調べたところ、インライン展開と同じ結果になった。

Javaコンパイラが最適化を行わないことが別段問題にはならない場合も多い。JITやAOTコンパイラを使う場合は、インライン展開などの最適化はいずれにせよJIT、AOTが行うので、バイト・コードの時点で最適化されていようがいまいが性能には影響がないことが多いのである。しかし、インタープリタや、コンパイルの速さを目標とした、あまり最適化を行わないJITコンパイラでは、Javaコンパイラがどの程度の最適化を行ったかが性能に影響するだろう。

Javaプログラマーとしては、使用するJavaコンパイラによって生成されるバイト・コード命令列が違うことと、それがインタープリタでの性能に影響を与える可能性があることを知っておきたい。

その他の最適化

前節までに挙げたもの以外にも、最適化のためのTIPSは数限りなく存在する。そのうち、どのようなアプリケーションにおいても検討する価値があるものを以下にいくつか紹介する。

ネーティブ・メソッド化

ネーティブ・メソッド化とは、Javaで記述してあるメソッドをCやC++で書き直すことである。CコンパイラがJIT、AOTコンパイラよりも高品質のコードを生成すれば、そのプログラムは速くなるだろう。しかし、JITがインライン展開できなくなるなど、性能面で損をする可能性もある。また、アプレットでネーティブ・コードを使ってしまうと、Webブラウザさえ用意すればどこでも実行できるというメリットが消えてしまう。損得をよく検討すべきである。

なお、Javaで書かれたメソッドをCのネーティブ・メソッドに変換するタイプのAOTコンパイラもあるので、そういったものを使う方法も考えられる。

リスト12:Javaコンパイラのインライン展開テスト用コード

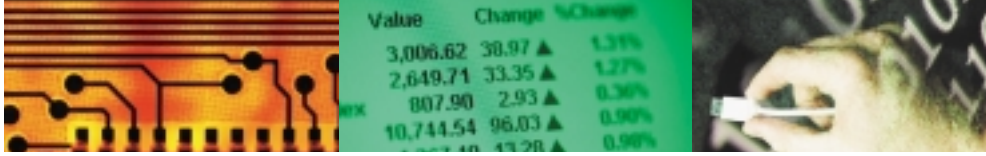
```
class InliningTest {
   public void caller() {
      System.out.println(callee());
   }
   private int callee() {
      return 5;
   }
}
```

メモリの節約

プログラムの消費メモリを節約することで、データがプロセッサのキャッシュに載るようになったり、GCの頻度が減ったり、OSによるページング回数が減ったりして、性能が向上する可能性がある。

メモリの節約に関して、プログラマーとしてできることには、生成するインスタンスの削減や、型の変更などがある。例えば、int型の配列を使っていても、もし値が−32,768から32,767の範囲に収まるならshort型に変更できる。そうすると、要素当たりの消費メモリが4バイトから2バイトと半分になり、メモリの節約となる。ただし、整数の演算はJVM中ではint型で行われることが基本なので、int型とshort型の変換が頻繁に必要になるかもしれない。

場合によっては、クラスjava.lang.Systemのメソッドgcをプログラムの中から呼ぶことも有効である。メソッドgcを呼ばなくてもGCは走るので、本来はわざわざ呼ぶ必要はない。しかし、例えば、入力に対してすばやく反応しないといけない局面でGCが動いていては困るといった場合には、



余裕のあるうちにメソッドgcを呼んでおくことで、大事な局面でGCが動いている可能性を低くできるかもしれない。もっとも、この問題については、GCの実装方法によって解決しようという努力がJVMの開発側で続けられている(incremental GC)。

クラスSystemのメソッドarraycopy

配列の内容をコピーする方法として最も基本的なのは次のコードであろう。

```
int[] src, dst;
   …
for (int i = 0; i < n; i++) {
   dst[i] = src[i];
}
```

配列のコピーには専用のメソッドarraycopyも用意されている。それを使うと上記のコードは次のように書ける。

```
System.arraycopy(src, 0, dst, 0, n);
```

メソッドarraycopyは、(少なくともJDK 1.3.0までは)ネーティブ・メソッドである。

どちらの方法が速いかは、主に配列の長さによって決まる。現在のところ、非常に大きな配列のコピーではメソッドarraycopyのほうが速いが、短い配列ではarraycopyを使わずにJavaで書いたほうが速い。どのくらいの配列長で逆転が起こるかは、JVM、JITやAOTコンパイラなどに依存するので、一概には決まらない。

1つだけ言えるのは、Javaで書く方法はJIT、AOTコンパイラの今後の状況によって速くなる可能性があるが、ネーティブ・メソッドのほうが速くなる可能性は低いということである。というのも、ネーティブ・メソッドで使われるCコンパイラやライブラリにおいて、配列コピーに関する進歩は期待できないからである。

入出力のバッファリング

『最適化に求められる知識』の例3に挙げたように、OSの入出力機能を利用する場合は、バッファ、つまりBufferedOutputStreamおよびBufferedInputStreamのインスタンスを用意することが非常に有効である。例3では実行時間は27分の1に短縮された。これは、OSの機能を直接呼ぶ回数を減らすテクニックである。OSの入出力機能には、例えばファイル入出力やネットワーク経由の通信が該当する。

また、バッファの大きさを変更すると、入出力性能、スループットが変わる。あまり小さいとバッファの意味がないし、逆に大きすぎるとプロセッサのキャッシュ・ヒット率が下がって性能が落ちる。バッファの大きさはBufferedOutputStream、BufferedInputStreamのコンストラクタで指定できる。なお、バッファのデフォルトの大きさを調べたところ、JDK 1.3.0、1.2.2、1.1.8ではBufferedOutputStreamで512バイト、BufferedInputStreamで2,048バイトとなっていた。

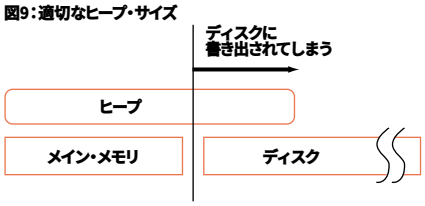
JVM起動時のオプションとパラメータ

コードの書き方からは外れるが、JVMを起動する際のオプションも軽視できないものだ。Sun JDK 1.3.0であれば、HotSpot Client VMとHotSpot Server VMの選択はオプション(-hotspotと-server)で行えるし、HotSpot VMのセールスポイントの1つであるincremental GCは-Xincgcオプションを付けないと有効にならない。

オプションの中でも、特にあらゆるプログラマーに関係が深いのはヒープ・サイズの指定である。ヒープとはインスタンスが置かれる領域のことで、例えば「-Xmx128m -Xms32m」という指定は初期ヒープ・サイズを32MBに、最大ヒープ・サイズを128MBにするという意味である。

ヒープ・サイズの設定は、もちろん、どれだけたくさんのインスタンスを生成できるかを決めるものだが、それだけではない。最大ヒープ・サイズが大きすぎるとヒープ全体がメモリに載らず、ディスクへの書き出し(ページング)が起きてしまいかねない(図9)。ページングが起きると、プログラム性能は大きく低下してしまう。それならむしろ、メモリからヒープがあふれる前にGCが走ったほうがマシである。

図9:適切なヒープ・サイズ



ヒープをメモリに載せ続けるためには、最大ヒープ・サイズをメモリの量より少ない適当な値に設定する必要がある。最適な設定値は、JVMと同時に走るOSや他のプログラムがどのくらいのメモリを消費するかによるだろう。また、最大ヒープ・サイズを指定しなかった場合に、一定の値にするのではなく搭載メモリ量に応じて適当に設定してくれるJVMもある。

＊ ＊ ＊

以上、最適化とは何かから始まり、Javaに固有のTIPSに至るまで、Javaプログラムの最適化に要求される考え方や知識をひととおり眺めてきた。こまごましたノウハウはたくさんあるものの、最も普遍的ではずしてはならないのは、アルゴリズムとデータ構造についての理解である。これによって、プログラムの性能は根本的に変わってくる。データ構造に関する理解とは、例えば目的の処理に対してjava.util.ArrayListとjava.util.LinkedListのどちらを使うべきかを判断できる素養である。

次に重要なのは、環境についてよく知ることである。効率の良いコードを書くためには、コンピュータ・アーキテクチャ、特にプロセッサやメモリ階層についての理解が欠かせない。JVMを知ることももちろん役に立つし、コンパイラを知ることで、コードの書き方と性能の関係が見えてくる。

あとは、本当に性能を向上させたいのなら、きちんと計測し、プロファイリングを行うことである。

Javaはまだ実行系の研究開発が盛んに行われている最中なので、状況は刻々と変化し続けている。したがって、過去の知識や結果に固執することなく、柔軟に対応し判断することを心がけてほしい。

特別リポート

最新JVM、JITコンパイラの性能を評価する

首藤一幸

各ベンダーがしのぎを削ってきたJavaプラットフォームの性能は、JDK 1.3.0のリリースとともに1段階引き上げられたと考えてよい。しかしながら、今やJVMやJITコンパイラも単に高速なだけでは許されない時代だ。アプリケーションのすばやい起動や操作への反応の良さなど、実際の使用感が問われ始めているのである。JDK 1.3.0は、そうした問題に対する、IBMやサンの現時点での回答である。本リポートでは、JDK 1.3.0をはじめとするJVM、JITコンパイラをいくつか選び出し、実際の計測を通じてその使用感と性能を評価してみたい。

新しいJavaプラットフォーム

現在、JITコンパイラの性能でトップを走っているのはIBMとサンである。その両社が先ごろJDK 1.3.0をリリースした。もちろん、標準クラス・ライブラリが拡充されているが、それだけではなく、JVMやJITコンパイラにも手が加えられている。

特にサンは、Windows用JDK 1.2.2まで採用してきたClassic VM（JVMのリファレンス実装。性能よりも仕様どおりの実装を目的としたJVM）とシマンテック製JITコンパイラという組み合わせをやめ、HotSpot VMを投入してきた。これまでも、HotSpot VM 1.0と1.0.1は、JDKへのプラグインというかたちで別途提供されてきたが、今回新しいHotSpot VMがSun JDK 1.3.0の標準JVMとなったのである。

サンによると、新しいHotSpot VMの特徴は次のとおりである。

- 適応的コンパイラ（adaptive compiler）
- 改良されたガーベジ・コレクション（GC）
- スレッド間の同期（thread synchronization）

適応的コンパイルとは、プログラムの実行状況に応じたコンパイルを行うことを指す。HotSpot VMは、プログラム中で頻繁に実行される部分、すなわちhot spotを検出し、そこを集中的に最適化する。

同時にガーベジ・コレクタも、より高速かつ効率的になったという。新しいガーベジ・コレクタはインクリメンタルGC（incremental GC）をサポートしている。これまでの単純な方法では、GCが不要インスタンスを回収しきるまでプログラムの実行が中断されてしまい、それによってプログラムの反応が鈍くなるという問題があった。一方、インクリメンタルGCでは、ガーベジ・コレクタが処理を途中で中断できるので、プログラム本体の実行を妨げにくい。ただし、この機能はJVM起動時に-Xincgcオプションを指定して初めて有効になる。また、上記のメリットと引き換えに、GC全体の性能がおよそ10%低下するとのことである。

もう1点、スレッド間の同期が軽く、速くなったことで、「マルチプロセッサ・マシンの複数プロセッサを生かしやすくなった」とサンは主張している。

サンは、JDK 1.3.0に2種類のHotSpot VMを用意した。HotSpot Client VMとHotSpot Server VM 2.0がそれである（HotSpot Client VM 2.0とは呼ばないようだ。サンはServer VMにのみ「2.0」を付けている）。この2つは、それぞれ異なった特性と目的を持っている。

まず、HotSpot Server VM 2.0は従来のHotSpot Performance Engine 1.0、1.0.1の後継という位置づけであり、プログラムのピーク性能が起動時間の短さよりも重要である場合、つまりサーバ・アプリケーション向けとされている。それに対してHotSpot Client VMは、すばやい起動がピーク性能と同じくらい重要な場合、すなわちクライアント・アプリケーション向けの特性となっている。

前述したとおり、Sun JDK 1.3.0ではHotSpot Client VMが標準のJVMとなっている。JDK 1.2.2までと同様のClassic VMも付属しているものの、Classic VM向けのJITコンパイラは用意されていない。またWindows用のSun JDK 1.3.0では、HotSpot Server VMは別途インストールする必要がある。サンのWebページから入手してインストールすれば、HotSpot Server VMとClient VMのどちらをデフォルトで起動させるのかを設定できる。さらに、Linux用Sun JDK 1.3.0のベータ版も公開されており、こちらには3種類すべてのJVMが付属している。

3種類のJVMのうち、どれを使うかはJVM起動時のオプションでも指定可能だ。HotSpot Client VMを使いたい場合は-hotspotオプションを、HotSpot Server VMを使うには-serverオプションを、何らかの理由でClassic VMを使いたい場合は-classicオプションを指定する。

従来のHotSpot Performance Engine 1.xとHotSpot Server VM 2.0の最大の違いは、やはりバイト・コードの実行性能だという。標準的なベンチマーク・テストでは「2.0は1.0.1よりも30%速い」とサンは主張している。そこで筆者は実際にどの程度向上したのかを計測してみた。その結果については、本リポートの後半で紹介する。

かたや、IBMのJVMはサンのClassic VMを基にしたものだ。とはいえ、大きな改造が施され、インタープリタもClassic VMとはもはや別物である。Classic VMで性能上の問題となっていた点もいくつか解決されている。オブジェクト・ハンドルの除去といった、結果的にサンのHotSpot VMと同じ改良もなされてきた。また、ヒープ・サイズの自動調整というような独自の機能も追加されている。Classic VMではオプション（-Xmxまたは-mx）を指定しなかった場合の最大ヒープ・サイズは固定であったのに対し、Windows用IBM JDK 1.1.8では物理メモリの半分のサイズに自動調整される。ヒープの拡張方法も改良され、縮小もサポートしているとのことである。

IBMのJITコンパイラは、JDK 1.1.8までのものと1.3.0のものとでは大きく異なっている。バージョン番号で言うと、Linux用JDK 1.1.8付属のものが3.0、Windows用JDK 1.1.8付属のものが3.5、JDK 1.3.0付属のものは3.6となっている。Windows用のIBM JDK 1.2.2は単独ではリリースされなかったが、アプリケーション・サーバの「WebSphere」などには付属しており、そのJITコンパイラのバージョンは3.6のようだ。ベンチマーク結果に関しても、やはり3.0、3.5、3.6とバージョンを追うごとにかなり向上している。

サンはHotSpot VMの特徴の1つとして適応的コンパイルを挙げているが、これは何もHotSpot VMだけにかぎったことではない。IBMのJITコンパイラも当然、短いコンパイル時間で最大の効果を上げるためにhot spotの検出には力を注いでいる。IBMの場合、サンのように“ HotSpot ”という宣伝文句を前面には出していないものの、JVMやJITにおける技術的課題は両社に共通なのである。

2つの要求：使用感とピーク性能

JITコンパイラに関する昨今の大きな問題は、「アプリケーションのピーク性能は向上したものの、コンパイルに時間を取られて起動が遅くなり、使用感が損なわれた」というものであった。JVM内蔵のWebブラウザを起動しただけで20秒もかかるという現象を経験した方も多いのではなかろうか。試しにJITコンパイラを削除してしまうと、Webブラウザの起動時間は数秒にまで短縮されるはずだ。

JVMは、起動されただけで数十から数百のクラスをロードし、そのメソッドを呼び出す。メソッドが呼ばれると、通常はそのメソッドについてJITコンパイラが呼び出されることになる。その時点でコンパイルが行われるかどうかはJITコンパイラや呼び出されたメソッドしだいだが、アプリケーション自体が起動するまでの間に相当数のメソッドがコンパイルされることは確かだろう。そのコンパイル時間が起動を遅くしているのである。

アプリケーションが起動するまでの間にJavaの

メソッドが呼ばれる回数は、JDKがバージョンを重ねるごとに着実に増えてしまっている。これには、JDK 1.2の時点で、それまでネーティブ・コードで行われていたJVM内の処理のいくつかがJavaコードに置き換えられたことも大きく影響している。

起動までの間にロードされるクラスの数は、JDK 1.1.8では50弱であったが、JDK 1.2.2では150を超え、JDK 1.3では200を大きく上回るに至った。ロードされたクラスの数とJITコンパイルの時間が直接比例するわけではないが、アプリケーションの起動中にJITコンパイラが動作する局面が増えたことは疑いのない事実だ。ただでさえロードされるクラスの数が増えて起動時間が長くなってしまったところに、JITコンパイラの動作が時間を消費してしまっては、アプリケーションの使用感は悪くなる一方である。

アプリケーションの起動や反応を速くするためには、JITコンパイルの対象とするメソッドの数を減らし、JITコンパイル中の最適化に費やす時間を短くすればよい。極端な話、メソッドを1つもコンパイルしなければ、JITコンパイラは時間を消費しない。しかし、単純にコンパイル対象と最適化に費やす時間を減らしたのでは、アプリケーションのピーク性能は下がってしまう。

使用感を良くするためのアプローチは、サンとIBMで異なっている。サンは、使用感とピーク性能という2つの要求を満たすために、特性の異なる2種類のJVM、すなわちHotSpot Server VMとHot Spot Client VMを用意した。前者は、起動時間を長くしてしまっても、JITコンパイルに時間をかけてピーク性能を高くすることのほうを優先する。一方、後者はすばやい起動も重視している。かたやIBMは、単一のJVMとJITコンパイラで2つの要求を同時に満たすという方針を採用している。「1つで両方を満たせる」とはIBM側開発者の談である。

つまり、サンのアプローチでは、ピーク性能優先（HotSpot Server VM）の場合、すばやい起動をあきらめる代わりに、JITコンパイラが最適化に時間を費やせるようにする。一方、IBMのアプローチはそれを許さない。起動時間を短く保ちつつ、なおかつピーク性能を追求するのである。よって、性能だけを追求すればよいHotSpot Server VMのほうが、同Client VMやIBMのJITコンパイラよりも、デザインは比較的易しくなる。

性能評価の方法

それでは、IBMとサンのJDK 1.3.0を中心に、各種JITコンパイラの性能を評価してみたい。ピーク性能と使用感の両方を調べるために、本リポートでは以下の2種類の実験を行った。

- SPEC JVM98
- 一太郎Arkの起動時間の計測

JVMやJITコンパイラの性能評価に広く用いられているSPEC JVM98でピーク性能を、実用アプリケーションの一太郎Arkをどれだけ速く起動できるかをもって使用感を測る。

SPEC JVM98

Standard Performance Evaluation Corporation（SPEC）は、性能評価のための基準やベンチマークを確立、整備している組織である。SPECを知らなくても、SPECintやSPECfpというベンチマーク名はプレスリリースなどでご存じの方も多いのではなかろうか。例えば今年3月9日のPentium III/1GHz発表の際にも、「SPECint2000の結果が410、SPECfp2000は284」というようにベンチマーク結果が発表されている。

先に触れたとおり、SPEC JVM98はJVM、JITコンパイラのベンチマークである。1998年8月19日にリリースされ、ライセンス料は100ドル（研究教育機関の場合は50ドル）となっている。SPEC JVM98の評価結果は、入出力やメモリ管理といったOSの性質のほか、当然ながらプロセッサの処理性能を反映したものとなる。

SPEC JVM98は8種類のプログラムから構成され、そのうちの5種が実アプリケーションを基にしている。足し算100億回といったマイクロベンチマークではなく、この点はSPECの他のベンチマークと同じである。SPEC JVM98に対する批判もないわけではないが、現実のアプリケーションに近い状況で測定できるため、多くのJVM、JIT開発者がSPEC JVM98を性能の指標として用いている。サンも、HotSpot Server VMの性能評価方法としてSPEC JVM98を推奨している。

SPEC JVM98は、多量のファイル入出力やネットワーク経由の通信、数十から数千のスレッドといった、いわゆるサーバ側アプリケーションで要求される性能は評価しない。これらはOSの性能に大きく左右されるため、JVMやJITコンパイラを中心に評価するというSPEC JVM98の目的に合致しないのである。サーバ側での性能を評価するベンチマークとしてはチャット・サーバを模したVolano Markがよく用いられているが、SPECも今年6月5日、新しいベンチマーク「Java Business Benchmark 2000（SPEC JBB2000）」を発表している。

SPEC JVM98を構成する8種類のプログラムは以下のとおりである。

_200_check：JVMの機能をチェックするプログラム。性能の指標としては使われない。

_201_compress：LZW法（Lempel-Ziv法の亜種）の圧縮プログラム。SPEC CPU95の129.compressベンチマークをJavaに移植したもの。

_202_jess：NASAのCLIPSエキスパート・システムのJava版。推論エンジンは、実行にしたがって大きくなるルール・セットを探索していく。

_209_db：IBMによって書かれた、データ管理のベンチマーク。メモリ上の住所データベースの上で複数の関数（つまり追加や削除、検索、ソート）を実行していく。

_213_javac：JDK 1.0.2のJavaコンパイラ。

_222_mpegaudio：MPEG audio Layer-3（いわゆるMP3）のデータを伸長するアプリケーション。4MBの音楽データを扱う。

_227_mtrt：レイ・トレーシングのプログラム。340KBの入力ファイルから恐竜を描く。2つのスレッドが走る。

_228_jack：パーサ・ジェネレータ。現在のJava CCの初期版である。jack自身の生成処理を行う。

一太郎Ark

一太郎Arkは、ジャストシステムによって開発された、いわゆるワープロである。多言語の混在が可能なことや文書フォーマットとしてXHTMLを採用したことも特徴であるが、何よりもアプリケーション自体がJava言語で記述されていることが最大の特徴である。

Javaで書かれた実用レベルのオフィス・アプリケーションの例として、本リポートでは一太郎Arkを、JVM、JITごとに異なる使用感を測るためのベンチマークとして利用する。具体的には、一太郎Arkが起動し終えるまでにかかる時間を計測し、その短さを使用感の目安とする。

性能評価の環境

今回の評価は、次のマシン上で行った。

- プロセッサ：AMD K6-2/400MHz 1個
- メモリ：64MB
- OS：Linux 2.2.14とWindows NT 4.0（Build 1381、SP4）

同じx86プロセッサといえども、Pentium II/IIIとK6-2とでは性能評価で違う傾向を示すことも珍しくない。そのため、今回の結果はあくまでK6-2での傾向でしかないことに注意してほしい。また、SPEC JVM98の_227_mtrtプログラムでは2つのスレッドが走るため、プロセッサが2個以上あればマルチプロセッサの性能を測れるのだが、今回使用したマシンはプロセッサを1個しか持たない。

以下は、本リポートで評価対象としたJITコンパイラである。基本的に、IBM JDK 1.3.0と1.1.8、Sun JDK 1.3.0と1.2.2に付属するものを対象とした。その概要を、まずはWindows用から紹介しよう。

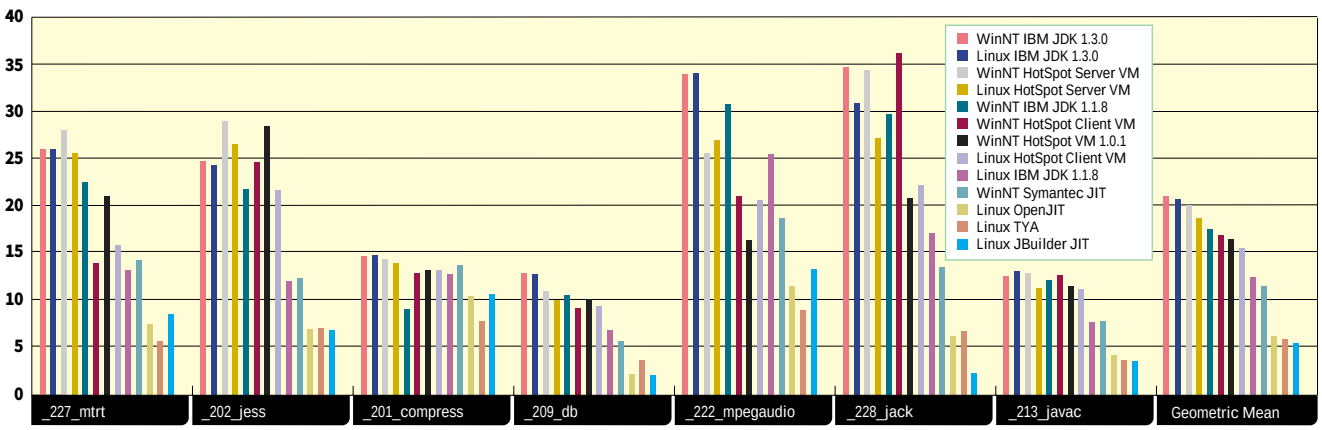
WinNT HotSpot Client VM：Sun JDK 1.3.0に付属しているJVM、JITである。

WinNT HotSpot Server VM：JDKとは別にサンのWebページで提供されているHotSpot Server VM 2.0。Sun JDK 1.3.0や1.2.2で使用できるが、

表A：SPEC JVM98の結果（JITコンパイラ）

	_227_mtrt	_202_jess	_201_compress	_209_db	_222_mpegaudio	_228_jack	_213_javac	Geometric Mean
WinNT IBM JDK 1.3.0	25.8	24.8	14.4	12.4	33.4	34.6	12.5	20.8
Linux IBM JDK 1.3.0	25.8	24.1	14.5	12.4	33.5	30.2	12.9	20.4
WinNT HotSpot Server VM	27.2	28.7	14.1	10.8	25.3	34.1	12.7	20.0
Linux HotSpot Server VM	25.2	26.1	13.8	9.92	26.5	26.8	11.1	18.4
WinNT IBM JDK 1.1.8	22.2	21.4	8.95	10.4	30.3	29.2	12.0	17.3
WinNT HotSpot Client VM	13.7	24.8	12.7	9.04	20.7	35.6	12.5	16.7
WinNT HotSpot VM 1.0.1	21.1	28.0	13.0	10.0	16.1	20.5	11.4	16.2
Linux HotSpot Client VM	15.6	21.3	13.0	9.23	20.3	21.9	11.0	15.3
Linux IBM JDK 1.1.8	13.0	11.9	12.6	6.79	25.1	16.9	7.65	12.3
WinNT Symantec JIT	14.1	12.2	13.5	5.64	18.4	13.3	7.72	11.4
Linux OpenJIT	7.40	6.93	10.3	2.16	11.0	6.12	4.19	6.12
Linux TYA	5.65	6.96	7.75	3.62	8.81	6.67	3.65	5.86
Linux JBuilder JIT	8.41	6.83	10.5	2.11	13.1	2.33	3.55	5.43

図A：SPEC JVM98の結果（JITコンパイラ）



今回はSun JDK 1.3.0とともに使用した。

WinNT HotSpot VM 1.0.1：HotSpot VMの旧バージョンであり、正式な名称はHotSpot Performance Engine 1.0.1。上記のHotSpot Server VM 2.0は、これの後継にあたる。このHotSpot VMは、リファレンス実装という、一般に配布されているものとは異なるSun JDK 1.2.2を必要とする。このSun JDK 1.2.2とHotSpot Performance Engine 1.0.1のどちらも、現在は配布されていない。

WinNT IBM JDK 1.3.0：IBMが配布しているWindows用のJDK 1.3.0。付属するJITコンパイラのバージョンは3.6である。「java -fullversion」として表示される日付は「20000622」である。ただし、IBMは、OSとバージョンがサンと競合するJDKをリリースできない。Sun JDKのソース・コードを基にしているためだ。この問題を回避するためか、このJDKはWebSphereの一部「IBM Cross Platform Technologies」として配布されている。

WinNT Symantec JIT：Sun JDK 1.2.2標準のシマンテック製JITコンパイラ。「jdk-1_2_2_005-win.exe」として配布されているものを用いた。

WinNT IBM JDK 1.1.8：IBMのJDK 1.1.8。同社製のJITコンパイラ3.5が付属している。「java -fullversion」と入力した際に表示される日付は「20000516」である。なお、IBMのJDKは、アーカイブのファイル名がそのままで中身のバージョンが変わるので、要注意である。

続いては、Linux用のJITコンパイラである。

Linux HotSpot Client VM：サンが現在ベータ版として公開しているLinux用のSun JDK 1.3.0。Windows用と同じくHotSpot Client VMが標準のJVM、JITとなっている。現時点では、あくまでベータ版であるという点に注意されたい。

Linux HotSpot Server VM：HotSpot Server VMは、Windows用ではJDKとは別に配布されているが、Linux用のJDK 1.3.0 Betaには付属している。

Linux IBM JDK 1.3.0：IBMが配布しているLinux用のJDK 1.3.0。付属するJITコンパイラのバージョンは3.6である。「java -version」に対して表示される日付は「20000623」である。

Linux IBM JDK 1.1.8：IBM JDK 1.1.8。付属するJITコンパイラのバージョンは3.0である。「java -fullversion」に対して表示される日付は「20000515」である。

Linux JBuilder JIT 1.2.15：インプライズ製のJITコンパイラ。JDK 1.2.2で使用できる。Linux用JDK 1.2.2には、サンが配布しているものとBlackdown Porting Team（ブラックダウン）が移植しているものの2種類があるが、どちらでも使える。今回はブラックダウンのJDK 1.2.2 RC4を用いた。

Linux TYA 1.7v2：アルブレヒト・クライネ氏によって開発されたJITコンパイラ。x86上のLinux、FreeBSD、JDK 1.1.x、1.2.xに対応している。JDKのソース・コードを一度も見ずに開発したという。「今後はメンテナンスだけを行う」というアナウンスが今年の1月にあった。

Linux OpenJIT 1.1.10：C、C++で書かれている多くのJITコンパイラとは異なり、大部分がJava言語で書かれたJITコンパイラ。Solaris 2（SPARC）、LinuxとFreeBSD（x86）に対応している。

性能評価の結果

まずはSPEC JVM98の結果から示そう。表Aは、ベンチマーク・プログラムごとのJITコンパイラのスコアである（数値が大きいほど良い）。また、図Aは表Aのスコアをグラフ化したものだ（Geometric Meanの良い順に左から並べている）。

これを見ると、IBM JDK 1.3.0とHotSpot Server VMのスコア（Geometric Mean）がほぼ同じで最も良い。ただし、注意していただきたいのは、ベンチマーク・プログラムの種類によってスコアの優劣が入れ替わっていることである。例えば、_222_mpegaudioではIBM JDK 1.3.0のスコアが最も高いが、_202_jessではHotSpot Server VMのほうが高いスコアを出している。

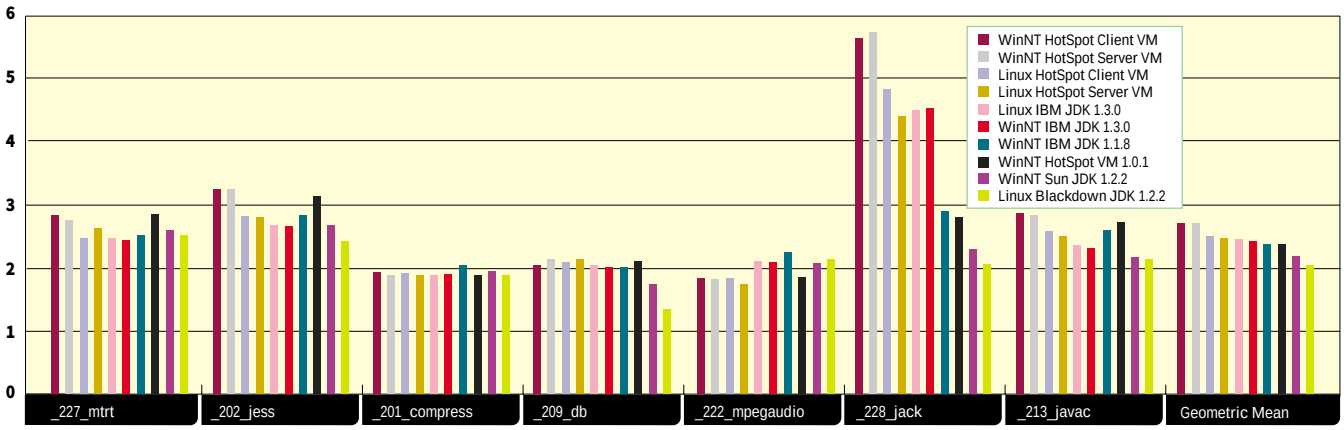
また、同じHotSpot Server VMでも、_222_mpegaudioを除くすべてのベンチマーク・プログラムで、Linux用のベータ版よりもWindows用のほうが良いスコアを出している。Linux用の正式版ではWindows用に追いつくことを期待したい。

もう1点、注目してほしいことがある。それは、IBM JDK 1.3.0とほぼ同じスコアを出しているのがHotSpot Server VMであるということだ。HotSpot Client VMのスコアはその約8割となっている。JITコンパイルに多くの時間を割いてもよいHotSpot Server VMに対し、IBM JDK 1.3.0はアプリケー

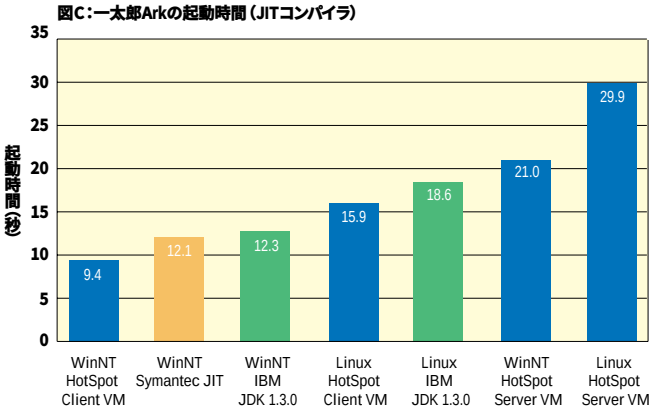
表B:SPEC JVM98の結果(インタープリタ)

	_227_mtrt	_202_jess	_201_compress	_209_db	_222_mpegaudio	_228_jack	_213_javac	Geometric Mean
WinNT HotSpot Client VM	2.88	3.31	1.97	2.08	1.86	5.69	2.92	2.76
WinNT HotSpot Server VM	2.80	3.30	1.92	2.18	1.85	5.78	2.88	2.75
Linux HotSpot Client VM	2.51	2.87	1.95	2.12	1.87	4.71	2.62	2.54
Linux HotSpot Server VM	2.68	2.86	1.91	2.17	1.83	4.33	2.54	2.51
Linux IBM JDK 1.3.0	2.51	2.73	1.91	2.07	2.14	4.41	2.40	2.50
WinNT IBM JDK 1.3.0	2.50	2.70	1.93	2.04	2.07	4.48	2.36	2.48
WinNT IBM JDK 1.1.8	2.57	2.89	2.08	2.04	2.29	2.92	2.64	2.47
WinNT HotSpot VM 1.0.1	2.90	3.19	1.91	2.15	1.89	2.85	2.77	2.47
WinNT Sun JDK 1.2.2	2.65	2.73	1.98	1.77	2.11	2.34	2.20	2.23
Linux Blackdown JDK 1.2.2	2.57	2.47	1.91	1.37	2.18	2.09	2.17	2.07

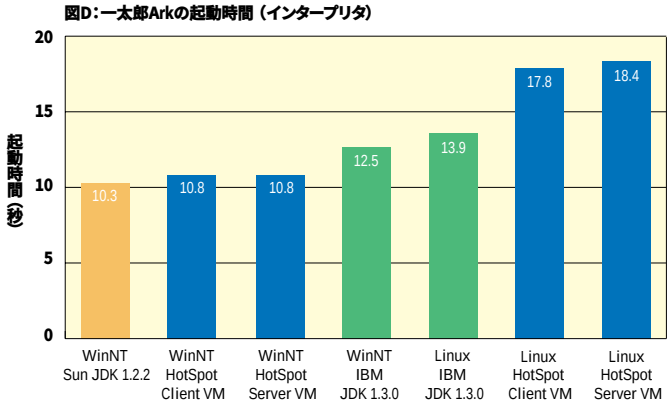
図B:SPEC JVM98の結果(インタープリタ)



図C:一太郎Arkの起動時間(JITコンパイラ)



図D:一太郎Arkの起動時間(インタープリタ)



ションの使用感をも追求してコンパイル時間を節約している。使用感、コンパイル時間を比較してみないと結論は出せないが、IBM JDK 1.3.0がいかに健闘しているか、である。

なお、参考までにインタープリタを使用した場合の結果も示しておこう(表B、図B)。IBMとサンの優劣の傾向はJITコンパイラの場合と似ている。例えば、_228_jackではサンが良い結果を出し、_222_mpegaudioではIBMが勝っている。

続いては、一太郎Arkを利用した計測の結果である。計測は、一太郎Arkが起動し終えて使用できる状態になるまでにかかる時間をストップウォッチで計ることで行った。起動のすばやさをもって使用感の1つの目安とするのである。もちろん、起動が速いからといって入力への反応が速いとはかぎらないし、目で見て手で計ったので結果の精度は期待できない。また、この実験はSPEC JVM98よりもOSの影響を大きく受けうるので、Linuxでの結果とWindowsでの結果を単純に比較するのは危険である。ここでの結果はあくまで目安としてとらえていただきたい。

計測の対象としたのは、主にIBMとサンのJDK 1.3.0である。また、従来のJVM、JITコンパイラの代表として、シマンテック製JITコンパイラが付属しているWindows用Sun JDK 1.2.2も評価した。

計測結果は図Cのとおりだ。Windows用のHotSpot Client VMが9.4秒と最も良い結果を出し、確かにSun JDK 1.2.2よりも速く起動している。それとは逆に、当然ではあるが、Windows用のHotSpot Server VMはHotSpot Client VMの倍以上の21.0秒かかっている。また、インタープリタでの結果(図D)と見比べたときに興味深いのは、他のJITコンパイラがインタープリタのときよりも多くの起動時間を要しているのに対し、Windows用HotSpot Client VMのJITは10.8秒から9.4秒と逆に起動時間を短縮している点である。

SPEC JVM98の結果が最も良かった2者、すなわちIBM JDK 1.3.0とHotSpot Server VMを比較すると、IBM JDK 1.3.0のほうが速く起動している。ピーク性能と同時に使用感も重視するというIBMの方針が結果に出たかたちである。

戦いは終わらない

研究目的のJVM、JITコンパイラならともかく、実用および製品レベルのJVM、JITでは今、IBMとサンが激しいトップ争いを演じている。ベンチマークのスコアさえ良ければよいという時代は、もはや過去のものとなった。

「A社のスコアは18.7、B社は10.8」というように、ベンチマークのスコアは比較しやすいがゆえに、唯一絶対の評価基準であるかのようにひとり歩きしがちである。しかし、今やJVM、JITの開発側は、ユーザーの本当のメリットを考えて、実際の使用感なども考慮しながら研究開発を行っているのである。したがって、利用者の側も、ベンチマーク結果の数値に踊らされることなく、本当に自分に必要なものを判断していくことが大切だろう。