Javaプログラムへの動的な測定点設置方式の評価

堀川 隆
NEC システム基盤ソフトウエア開発本部

プログラム内への測定点設置は、性能測定（プロファイリング）の基本技術である。本論文では、Java プログラムに対して、source code を修正することなく測定点を設置する手法（Dynamic Instrumentation）について方式検討、試作、および性能評価を行った。その結果、早期の JVM から備えられているプロファイラ用インタフェース（JVMPI）はイベント通知のオーバーヘッドが大きく、hook を設置しないメソッドでもイベントが通知される点が問題となる可能性が高いこと、バイト・コードを動的に書き換える方式では、使用するツールに応じてクラス・ロードや HotSpot の動作時間は長くなるが、hook 対象メソッド以外では一切オーバーヘッドがかからないため、定常状態の性能測定に適していることを定量的に示すことができた。

A Study on Dynamic Instrumentation for Java Programs

Takashi Horikawa
System Platform Software Development Division, NEC

Some methods of probe instrumentation for java programs, the first step in the performance profiling, has been studied. Quantitative performance evaluation shows that the CPU overhead produced by JVMPI mechanism was mainly due to its event notification process, and thus the overhead will be inflicted even on methods other than measurement targets. While dynamic bytecode instrumentation produced a certain amount of CPU overhead depending on the instrumentation tool especially during the class load operation of JVM, there was no CPU overhead in the execution of non-probed methods and slightly bigger overhead than measurement target methods equipped with probes by source code modification method, and thus dynamic bytecode instrumentation is suitable for performance measurement for Java program execution in steady state, in which all methods are translated into native machine code by HotSpot compiler.

1．はじめに

作成したプログラムが期待した性能を発揮しない場合、対策の第一歩は性能上の問題点すなわち、プログラムのどの部分で処理に時間がかかっているのかを見つけ出すことである。実行時間に占める割合の大きい処理について改善を実施すれば効果は大きいが、そうでない処理を相手にすると、同じ労力をかけても全体に対する効果は小さくなってしまう。

プロファイラは上記の目的で使用するツールであり、プログラム実行の部分々で費やした時間を集計し提示する機能がある。この機能は、プログラムの実行性能を考える上で必須と考えられており、コンパイル時のオプション指定で使用可能となるもの、CPU のハードウエア機能を使用するもの、など、様々なプロファイラが存在している。

Java プログラムを対象としたプロファイリングでは、JVM（Java Virtual Machine）に備わっているプロファイラ用の機能を利用して測定点をプログラム内に設置するツールが一般的である。本稿では、この機能を利用したプロファイラを試作し、その性能（CPU オーバーヘッド）を比較した結果を報告する。

2．測定点設置方式と実装

ここでは、測定点設置（hook と呼ぶ）を、測定対象メソッドの入口と出口の２箇所で測定操作を行うメソッド（probe と呼ぶ）を call するように測定対象メソッドを変更する（図１）こと、と定義し、これを『probe を測定対象メソッドに hook する』と表現する。入口と出口に probe を hook することで、各々を通過した２つの時刻の差分（メソッド実行にかかった時間）を求める等の測定が可能となる。

動的な測定点設置とは、プログラムの実行形式（Java の場合はバイト・コード）に変更を加えることなく、プログラム（JVM）起動後に対象メソッドに probe を hook すること、と定義する。試作・評価した各方式の実装では、測定点設置対象となるクラス名・メソッド名をテキスト形式のファイルで指定可能な機能を持たせた。

各方式の性能を比較する際の基準とするため、動的でない測定点設置方式（source code 埋め込み方式）も測定対象としたので、その方式についても簡単に触れることにする。

2．1　Source code 埋め込み方式

Java プログラム（測定対象メソッド）の source code に probe を呼び出すステートメントを挿入する測定点設置方法である。この source code をコンパイルして作成されたバイト・コードを実行することで、probe による測定操作が行われる。

メソッド入口は１箇所のため hook 設置作業は容易な一方、出口については、メソッド途中に return 文が存在できるので、複数個所に hook を設置しなければならない場合もある。実務への適用を考えた場合、手作業による hook 設置は面倒であり、また、hook を入れるべき位置を見逃してしまう危険性もあるので、実用性は乏しいといえる。

2．2　JVMPI イベント通知機能利用方式

JVMPI（Java Virtual Machine Profiler Interface）[1] は、JVM（Java Virtual Machine）が提供するプロファイラのためのインタフェースであり、JVM が java プログラムを実行している際に発生するイベントをプロファイラ・エージェントに通知（エージェントに用意した関数を call）する機能を持っている。

この方式は、「メソッドに入る」と「メソッドから出る」というイベントの通知を受けて probe を call するものである。このイベント通知においては、対象となるメソッドの識別手段として JVM の管理する ID が通知されるのみであり、クラス名とメソッド名は陽には分からないようになっているため、「クラスのロード」イベント発生時にクラス名、メソッド名と ID の対応を覚えておく処理も必要となった。

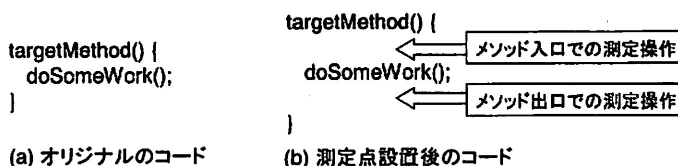

図1　メソッドへの測定点設置

2．3　バイト・コード書換え方式

これはバイト・コードを JVM の動作中に書換えることで測定点を設置する方式である。書換えのタイミングは、JVM がバイト・コードをロー

ドする時点とロード完了後の任意の時点の2種類が考えられるが、ここでは前者を評価対象とした。

JVMがロードしようとしているバイト・コードの内容を書換えるために用意されているメカニズムは主に2種類である。また、バイト・コードを書換えて対象メソッドに hook を設置するための手段としては、既に有力なツール（ライブラリ）が存在しているので、それを利用した。

2.3.1 メカニズム

ロード中のバイト・コードを書換えるためにJVMに用意されているメカニズムには下記がある。

1. JVMPIの「クラスファイルデータの設置準備完了」イベント通知機能
2. instrument API (java.lang.instrument) [2]

どちらも、JVMがクラスをロードしている途中でエージェントにバイト・コードを渡し、そこから受け取ったバイト・コードによりそのクラスのメモリ内部表現を構築する、という機能である。エージェントにて書換え操作を行い、書換えたバイト・コードを JVM に戻すことにより、ロード時の書換えが可能となる。

両者は、イベント通知先（書換え対象のバイト・コードを受ける）メソッドの実装方法が異なっている。JVMPI での通知先が共有ライブラリ中の関数（native な機械語命令）であるのに対し、instrument API での通知先は、Java プログラムとして記述されたメソッド（バイト・コード）である。このため、JVMPIはOSやCPUの種類毎にエージェントを用意する必要があるのに対し、instrument API はJavaの世界で完結しており、OSやCPUの種類には依存しない。

また、サポートする J2SE (JDK) のバージョンにも差異がある。JVMPIは、SUNの提供するJDK 1.2 の最終リリースの段階で用意されているのに対し、instrument API は、J2SE 1.5 で導入された新しいAPIである。

なお、instrument API には、ロード済みクラスのバイト・コードを書換える機能も用意されているが、今回の評価では対象外とした。

2.3.2 バイト・コード書換えライブラリ

バイト・コードの書換えをサポートするライブラリとして、ここでは、BCEL (Byte Code Engineering Library) [3] と Javassist[4] の2種類を使用した。BCEL は、Java クラスファイルのデータ構造を直接操作する用途に適している一方、Javassist はJava バイト・コードにあまり詳しくない開発者でもバイト・コード変換を実装できることを目的としたライブラリである。

関数の入口と出口に hook を設置する場合、BCEL では hook 位置に挿入する処理をバイト・コードとして与えるのに対し、Javassist ではコンパイラ機能を有しているため hook 位置に挿入する Java ステートメントを指定すれば良いようになっている。

2．4 動的hook設置方式の比較

各方式について主に測定オーバーヘッドの点から比較する。測定オーバーヘッドとは、hook を設置しない場合に比べて処理時間が増大すること、と考えられるが、probe メソッドで行う測定操作は全方式とも同じと考えられるので、ここで言及しない。

JVMPI 利用方式では、総てのメソッドについて「メソッドに入る」と「メソッドから出る」というイベントが通知される点が主なオーバーヘッドの原因となる。すなわち、JVM によるイベントの通知操作、および、エージェントの処理関数にて当該メソッドが hook 対象かどうかを判定する操作が、hook 対象でないメソッドについても行われるため、オーバーヘッドが大きくなると予想される。

バイト・コード書換え方式では上記の問題は発生しないが、hook 対象メソッドを有するクラスを JVM がロードしている最中に行う書換え操作がオーバーヘッドとなる。書換えライブラリの違いとしては、Javassist は hook 位置に挿入する Java ステートメントをコンパイルする処理が入るため、hook 設置処理の時間はBCELより長いと予想される。

3．性能評価

3．1 評価対象

試作したプロファイラは、筆者が開発しているイベント・トレース方式の性能（挙動）測定・分析ツール (mevalet) [5] を利用したサブ・ツールである。利用者が測定対象メソッド名をtext形式のファイルに記述してプロファイラに与えると、JVM機能を利用して hook を対象メソッドの入口と出口に設置する。対象メソッドに設置された hook の実行によりイベント・トレーサが記録するイベントを発生させるようになっているので、測定対象メソッドの性能指標は、得られたトレース・データの分析により求めることができる。

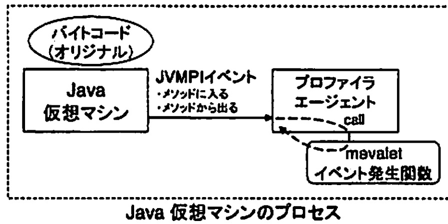

図2 JVMPIイベント通知機能利用方式

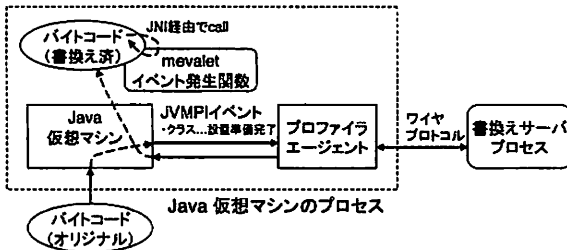

図3 バイトコード書換え方式1

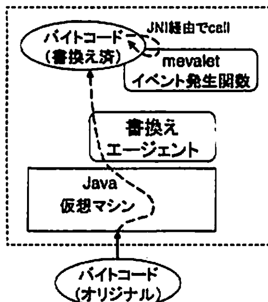

図4 バイトコード書換え方式2、3

設置する hook の機能は、イベント・トレーサの一機能として提供している「アプリケーションから記録するイベントを発生させるライブラリ関数（アセンブラで記述）」を call することである。バイト・コード書換え方式の場合、hook では JNI（Java Native Interface）経由でライブラリ関数を使う probe メソッドを call するようにした。

比較評価した hook 設置方式は5種類である。

① Source code 埋め込み方式
② JVMPI イベント通知機能利用方式（図2）
③ バイト・コード書換え方式1
(JVMPI + BCEL、図3)
④ バイト・コード書換え方式2
(instrument API + BCEL、図4)
⑤ バイト・コード書換え方式3
(instrument API + Javassist、図4)

比較の基準として、hook を設置しない場合についても測定を実施した。Source code 埋め込み方式については、probe メソッドを call するステートメントのないプログラムの実行時間、その他の方式では、hook 対象メソッドを定義するファイルとして空ファイルを使用して実行させたときの実行時間である。

実行時間測定で使用した時計は java の System.currentTimeMillis() メソッドによる時計である。分解能は1ミリ秒とされている。

Hook 設置対象プログラムは、tree 状に下位クラスのメソッドを call する人工的な CPU 浪費プログラム（図5）を用意した。クラス数は10000個である。簡単のため、Main を除く全クラスでは static な1個のメソッドを実装している。Hook 対象メソッドは、各クラスに1個記述されている static なメソッド全部、すなわち、10000個のメソッドである。

測定で使用したマシンの CPU は 3.0GHz の PentiumD、メモリは 1G バイト、OS は mevalet を適用した 2.6.9 ベースの商用 Linux, J2SE のバージョンは 1.5.0_08 である。

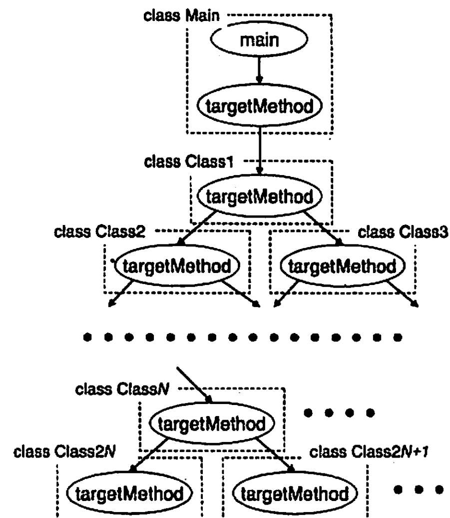

図5 測定対象プログラムの概要

3．2　評価項目と測定結果

Java プログラムの性能を考える上では、1) あるクラスが最初に使用される（そのクラスのあるメソッドが call される）場合、クラス（バイト・コード）を JVM にロードする操作が行われる、2) あるメソッドが何回か実行されると、そのメソッドは HotSpot 等の動的コンパイルにより機械語コード化され、実行が高速になる、という特性を考慮に入れる必要がある。そこで、本評価では、バイト・コードが機械語コード化されるまでの間（3.2.1）とそれ以降の定常状態（3.2.2）の２種類に分けて評価対象プログラムの実行時間を計測した。

なお、インライン展開やデッドコード除去等、ベンチマーク結果を歪める可能性のある最適化[6]は行わないよう、HotSpot はクライアント・コンパイラを使用した。

表１　定常状態になるまでの実行時間（Hook 設置時）

	①	②	③	④	⑤
1回目	372.3	1232.3	1009.8	912.1	2288.6
2回目	32.1	30.1	31.8	33.6	32.8
3回目	289.0	301.6	476.9	472.1	385.5
4回目	4.7	7.9	7.3	7.3	7.0
5回目	4.5	7.7	6.9	7.0	6.7
6回目	2.9	4.3	3.6	3.5	3.6
7回目	2.9	4.3	3.6	3.5	3.5
8回目	2.9	4.3	3.6	3.5	3.6
9回目	2.9	4.3	3.6	3.6	3.6
10回目	2.9	4.3	3.6	3.6	3.6

単位［μ秒］

表２　定常状態になるまでの実行時間（Hook 非設置時）

	①	②	③	④	⑤
1回目	355.6	1233.9	644.2	370.4	371.1
2回目	29.2	29.9	29.3	29.0	29.2
3回目	190.0	301.4	191.5	190.3	190.6
4回目	3.2	7.9	3.2	3.2	3.2
5回目	3.1	7.6	3.1	3.1	3.1
6回目	2.2	4.3	2.1	2.1	2.1
7回目	2.1	4.3	2.1	2.1	2.1
8回目	2.1	4.3	2.1	2.1	2.1
9回目	2.1	4.3	2.1	2.1	2.1
10回目	2.1	4.3	2.1	2.1	2.1

単位［μ秒］

表３　定常状態での実行時間（Hook 設置時）

	①	②	③	④	⑤
定常状態	2.869	4.306	3.550	3.550	3.558

単位［μ秒］

表４　定常状態での実行時間（Hook 非設置時）

	①	②	③	④	⑤
定常状態	2.114	4.287	2.114	2.115	2.117

単位［μ秒］

3.2.1　定常状態になるまでの実行時間

10000 個のメソッドを tree 状に call する動作１回を単位動作とし、その１回目から 10 回目までの動作時間を測定した。１メソッドの平均実行時間に換算した結果を表１および表２に示す。なお、各欄の値は、同じ測定を 10 回実施して得られた平均値である。実行時間の長い３回目までに関しては、測定値の平均からのずれは概ね±2%（最大で±5%）に収まっており、結果の再現性は高いといえる。

どの方式でも、クラス・ロードを行う１回目と HotSpot が動作すると考えられる３回目の動作時間が長くなっている。また、６回目以降は動作時間がほぼ一定となったため、この段階で機械語命令に変換された状態のコードを実行するようになったと考えられる。

3.2.2　定常状態での実行時間

10000 個のメソッドを tree 状に call する動作 100 回（1M 回のメソッド・コール）を１回の単位動作とし、クラス・ロード操作や HotSpot によるコンパイル操作の影響を受けない２回目（10000 回のメソッド call では 101 回目から 200 回目、以下同様）から 16 回目までの動作時間（15 個）を測定した。１メソッドの平均実行時間に換算した結果を表３および表４に示す。測定値の平均からのずれは最大±0.5%程度であり、結果の再現性は高いといえる。

3．3　考察

3.3.1　クラスのロード時間

表１と表２の結果において、１回目の実行時間がクラス・ロード処理を含む実行時間である。バイト・コード書換え方式（③④⑤）では、クラス・ロード時にバイト・コードが JVM からエージェントに送られて書換え操作が行われるため、source code 埋め込み方式（①）よりも１回目の実行時間が長くなっている。

JVMPI イベント通知機能利用方式（②）でもクラス・ロード時に大きなオーバーヘッドがかかっている。これはクラス名・メソッド名と JVM の管理する ID との対応付け処理の可能なタイミングが、バイト・コードのロード時であるため、その時点で hook 対象メソッドかどうかを判断する処理を行わせていることによるものと考えられる。JVMPI のイベント

通知（クラス・ロード）は、JVM起動直後から通知されるため、全部のクラスのロード操作についてイベント通知、および、その結果として上記処理が実行されるのに対し、instrumentation API では、main()メソッドの直前から書換えエージェントにバイト・コードが渡されるようになっており、立ち上がり時における基本的なクラスのロード操作は対象外となっている。この違いが、ロード時に書換え操作を行わない②でも③④⑤と同程度のオーバーヘッドが発生している原因と考えられる。

バイト・コード書換え方式では、表1と表2の1回目の実行時間の差が、バイト・コード書換え処理（hook対象メソッドに2個のhookを設置）の時間に相当する。コンパイル操作が行われるJavassistの方がBCELより長い（約3.5倍）、という定性的に予想された結果となっている。

また、表2の③④⑤と①の差はバイト・コードのロード操作に書換えエージェントが介入することによるオーバーヘッドと考えられる。同じJVMのコンテキストで動作するinstrument API（④⑤）の方が、別プロセスである書換えサーバとの間でバイト・コードの受け渡しが発生する③よりオーバーヘッドは少ない。これも定性的な予想と合致した結果である。

3.3.2 定常状態でのprobe実行時間

バイト・コード書換えによって、source code書換えと全く同じバイト・コードが作成されるのであれば、表3の①と③④⑤の実行時間は同じになるはずである。しかし、source code書換え（①）の方が実行時間は短い、という結果が得られた。これより、③④⑤のバイト・コード書換えでは、source code書換えによるhook設置とは異なるバイト・コードになっていると判断できる。また、表3の③④と⑤の実行時間には差が殆どなかったが、HotSpotが動作したときの時間（表1の3回目）には差が観測されたことから、hook設置後のバイト・コードは両ツールで異なっているものの、定常状態での実行時間はほぼ同じとなるコードに書き換えられていたと考えられる。

動的に設置したhookのオーバーヘッドを評価する際の基準は、source code書換え方式のprobeメソッド実行時間（表3と表4の差）と考えた。Source code書換え方式とバイト・コード書換え方式のprobeメソッドの実行時間には約0.7マイクロ秒の差が観測されたので、これが動的に設置したhookのオーバーヘッドと考えられる。このオーバーヘッドの要因についての詳細な分析は今後の課題と考えている。

プローブを使う立場からすると、このような定量評価により、オーバーヘッドの特性を把握しておくことが重要と考えている。

3.3.1 JVMPIイベント通知機能のオーバーヘッド

JVMPIのイベント通知機能利用方式では、hook対象メソッドが存在しない場合でも、メソッド入口と出口でイベント通知操作が行われる。Hookを設置した場合としない場合の定常状態実行時間では、JVMPIイベント通知機能利用方式（表2と表4の②）にのみ大きな差が見られなかったことから、この方式におけるオーバーヘッドの主要因は、probe関数ではなくイベント通知機能であることが分かった。従って、この方式は、特にプログラム中のごく一部のメソッドのみにhookを設置する場合、バイト・コード書き換え方式に比べてオーバーヘッドの面で著しく劣ってしまうことになる。

4．まとめ

Javaプログラムに対して動的に測定点を設置する方式について検討を行い、JVMPIイベント通知機能利用方式（JVMPI方式）とバイト・コード書換え方式によるツールを試作した。それらの性能評価を行った結果、JVMPI方式ではイベント通知のオーバーヘッドが主であり、測定点設置対象でないメソッドの入口・出口でも大きなオーバーヘッドが発生することが分かった。

バイト・コード書換え方式では、source code埋め込み方式に比べるとオーバーヘッドは大きくなったが、測定点設置対象でないメソッドではオーバーヘッドは発生しないことを示せた。このため、測定点設置対象メソッドを絞り込むことでオーバーヘッドは低減可能であり、有用な方式といえる。

JVMがロードしようとしているクラスのバイト・コードを書換えエージェントに渡すメカニズムとしてinstrument APIは、J2SE 1.5以降でないと使えないという欠点はあるが、Javaの世界で処理が完結するので扱い易く、また、ロード済みクラスのバイト・コードを書換える機能も提供されていることから今後の主流になるものと予想される。ツール作成が容易となるのに従い、今後は、このような基本機能を使ってどのようなデータを採取し、どのよ

うに使えばよいのか、といった利用技術の深耕が求められていくものと考えられる。

参考文献

[1] Java™ Virtual Machine Profiler Interface (JVMPI), http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmpi/jvmpi.html.

[2] 青柳　龍也, J2SE 5 解体新書　第14回　バイトコードの操作, JavaWorld, Vol.10, No.3, pp. 108-118, 2006.

[3] BCEL (Front Page), http://jakarta.apache.org/bcel/index.html.

[4] 千葉 滋, "Javassist・Java バイトコードを操作するクラスライブラリ・入門", JAVA PRESS, Vol.35, p.76, 2004.

[5] 堀川　隆, 性能基礎データ収集方法の比較検討, 情処研報, Vol.2004, No.83, 2004-EVA-10, pp.1-6, 第10回システム評価研究会 (2004年8月).

[6] Java の理論と実践: 動的コンパイルとパフォーマンス測定 コンパイル下でのベンチマークの危険性, http://www-06.ibm.com/jp/developerworks/java/050114/j_j-jtp12214.html.