Hewlett Packard Enterprise

Java パフォーマンス・チューニング

パフォーマンス・チューニングのルールや Java パフォーマンス・ツールの使い方を説明します。また、HP-UX のカーネル・パラメータやネットワーク・パラメータのレベルでのチューニング方法も解説します。

《連載期間：2004 年 3 月～2004 年 8 月》

―目次―

第 1 回

基本的ルール、パフォーマンス・ツールの使い方

2004 年 3 月

連載の第 1 回では、パフォーマンス・チューニングにおける基本的ルールや、HPE が提供する各種の Java パフォーマンス・ツールの使い方を説明します。なお、今後の連載では、JVM レベルに留まらず、HP-UX のカーネル・パラメータやネットワーク・パラメータのレベルでのチューニング方法も解説します。また、より高度なチューニング技法として、JVM のガーベジ・コレクションやスレッド競合に注目する方法も紹介する予定です。

パフォーマンス・チューニングのルール

パフォーマンス・チューニングにはいくつかの基本ルールがあります。これらのルールは、どのようなシステムに対しても適用できる一般的なものと言えるでしょう。パフォーマンス・チューニングについて技術的な詳細を論じる前に、まずはこれらの普遍的なルールをいくつか紹介します。

ルール 1：　システムのパフォーマンスは、互いに依存する多数の要素の影響を受ける

1 つのシステムの内部では、数多くのコンポーネントが協調して動作しており、互いに複雑な依存関係を結んでいます。そのため、パフォーマンスのボトルネックがどの場所にあるのかを一目見ただけで特定するのは、きわめて困難です。

また、こうした複雑な依存関係によって、次のルールも導かれます。

ルール 2：　パフォーマンス・チューニングは、常にトレード・オフを伴う

システムのパフォーマンス・チューニングには、OS の動作メカニズムをはじめ、パフォーマンスに影響を与える依存関係の理解が必要です。ある部分にチューニングを施せば、他の部分で性能の劣化を招く可能性もあるのです。例えば、メモリ利用率を向上させるためのチューニングによって、ファイル・システムのパフォーマンスが低下するようなケースも考えられます。よって、アプリケーションの要件や、メモリとファイル・システムの相互作用を理解し、トレード・オフの最適なポイントを見極めることが重要となります。

また、ボトルネックが存在する場合には、システムに備わるリソースと、それに対する要求のバランスを上手に維持しなくてはなりません。ボトルネックを解消するためには、リソースを増加させるか、あるいは要求を減らすかのいずれかしかないのです。そうした観点でも、システム内部のリソースの使用状況を詳細に観察することが大きな意味を持ちます。

さらに、システムの挙動を観察する上では、次のルールが役に立つでしょう。

ルール 3：　システムに影響を与えずにシステムを観察することは不可能である

システムを観察することは、それ自体がシステムのリソースを消費する行為でもあります。もちろん、使用するツールの種類によってその消費量は異なってきます。例えば、GUI 版の glance である「gpm」は、glance や vmstat といったテキスト・ベースのツールに比べて、はるかに大きいメモリ（5MB 以上）を消費します。

また、こうしたツールはリソースを消費するだけでなく、システムの挙動そのものをも変えてしまうことに注意してください。十分なメモリを持たないシステムでは、ツールのメモリ消費によって OS のスラッシング（過度なスワッピング）が発生するかもしれません。こうなると、システムをありのままの姿で観察することはもはや不可能です。

どの程度までならシステムに与える影響を許容できるのでしょうか。これは、ツールを選択する上で重要なポイントとなります。また、状況に応じて様々なツールを使い分けることが必要となるでしょう。これは、次のルールで言い表すことができます。

ルール 4：　チューニングに必要な全てのデータを提供できる万能ツールは存在しない

どのような状況でも問題を解決できる完璧なツールは存在しません。よって、パフォーマンス・チューニングでは、数多くのツールの使い方を習得せざるを得ないのです。また、複数のツールを使用することで、得られた情報をクロス・チェックすることができ、ツールが本当に正しい情報を提供しているか確認することも可能となります。

チューニングの手順

パフォーマンスの問題を速やかに解決しなくてはならない状況では、計測や分析に時間をかけず、推測に基づく安易な対応を取ってしまいがちです。しかし大抵の場合、こうした対応は状況をさらに悪化させかねません。場当たり的な対応をする代わりに、以下に示す一定の手順に従ってパフォーマンス・チューニングを実施することが望ましいでしょう。図 1 は、この手順を図示したものです。

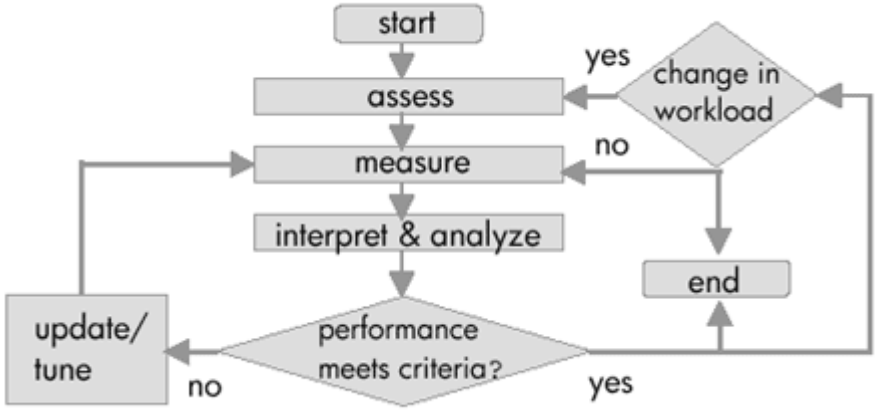

図 1：パフォーマンス・チューニングの手順

評価（Assess）

最初に実施する「評価」の段階では、以下に示す各項目について調査を進めていきます。これらの項目を明らかにすることで、何をどこまでチューニングするのかといった基本的な前提条件があぶり出されます。

- システムの構成
- アプリケーションの設計
- パフォーマンスの目標
- ピーク時の負荷の大きさと期間
- システムやアプリケーションに施された変更内容

- パフォーマンスの問題が発生する期間

この段階で最初に実施すべき作業は、ユーザ・リクエストが処理される上で通過する全てのマシンの構成を列挙することです。具体的には、以下の内容についてまとめます。

- マシンの CPU 数とクロックスピード（CPU 数は glance や gpm で確認可能）
- メインメモリ容量（同じく glance や gpm で確認）
- ディスク容量（使用領域と未使用領域のサイズ。df コマンドで確認）
- OS のチューニング可能なパラメータの値（HPjconfig ツールで確認）
- OS に適用されるべきパッチ（HPjconfig で確認）
- JVM のバージョン（「java -version」コマンドで確認）
- JVM に指定しているオプション（-Xms や-Xmx など）

例えば、「java -version」コマンドを実行すると、JVM のバージョン情報が以下のように得られます。

```
java version "1.3.1.01-release"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1.01- release-010816-12:37)
Java HotSpot(TM) Server VM (build 1.3.1 1.3.1.01-release-010816-
13:34-PA_RISC2.0 PA2.0, mixed mode)
```

これらのバージョン情報は、Java プログラムのパフォーマンスを評価する上で欠かせない基本情報となります。問題の内容によっては、最新バージョンの JVM にアップグレードしたり、OS に最新のパッチを適用したり、JVM 起動時に適切なオプションを指定したりといった簡単な方法で解決できることもあります。

一般的には、最新バージョンの JVM にアップグレードを行い、OS に最新のパッチ・セットを適用することが推奨されています。OS のパッチについては、HPE が提供する「HPjconfig」を利用することで、適切なパッチが適用されているかをチェックすることが可能です。なお、同ツールについては、次回の連載で詳しく説明する予定です。

測定（Measurement)

パフォーマンス・チューニングにおける次の段階は「測定」です。この段階では、実際のシステムを動作させることで、アプリケーションのパフォーマンスを測定します。

HP-UX 上で動作する Java アプリケーションのパフォーマンスを測定する手段としては、「sar」や「top」といった標準的なツールに加えて、以下の手段を利用することができます。

- パフォーマンス測定ツール「glance」および「gpm」
- JVM オプション「-Xverbosegc」（分析ツール「HPjtune」）
- JVM オプション「-Xeprof」（分析ツール「HPjmeter」）
- 「kill -3」によるスタックトレースの取得

これらの手段を利用した情報の収集には、かなりの手間と時間がかかる場合もあります。しかし、この測定の作業と次に説明する分析の作業は、実際にシステムに手を加える前に完了させておくことがポイントです。

分析（Analyze）

測定を終えた後は、収集した大量のデータを「分析」する段階に進みます。とはいえ、それぞれのツールが出力するデータの形態はまちまちであり、それらを正確に解釈することは容易な作業ではありません。冒頭で述べたとおり、1 つのシステムは複雑に依存し合う多数の要素で構成されているのです。

収集したデータの分析方法については、次回以降の本連載で詳しく解説する予定です。

特定（Identify）

分析の作業によって、システムのボトルネックはどこに存在するのかを「特定」することが可能になります。いったんボトルネックを見極めることができれば、パフォーマンスを改善するためにどのようなチューニングを施せばよいのかが明らかになります。もっとも、1 つのボトルネックを解消すると、今度は別の場所が新たなボトルネックとなる場合も少なくありません。

変更（Tune）

ボトルネックを特定したならば、その後に取るべき道は 2 つあります。それは、システムのリソースを増加させるか、あるいはリソースへの要求を減らすか、いずれかの「変更」を施すことです。どちらにせよ、変更の対象は 1 箇所に限定しなくてはなりません。2 箇所以上に変更を施すと、どれがパフォーマンスに影響を及ぼしたのか分からなくなってしまいます。

また実施した変更については、随時ノートなどに記録するようにします。変更の履歴を残すことで、後々の混乱を避けることができるはずです。

以上、連載第 1 回では、Java パフォーマンス・チューニングにおける基本的なルールとチューニング手順について説明しました。次回以降は、ツールを利用したチューニング作業の実際を解説します。

第 2 回

ガベージ・コレクション

2004 年 4 月

Java 言語では、Java オブジェクトに対するメモリ領域の割り当てや解放が JVM によって自動的に行われ、「ガベージ・コレクション」と呼ばれます。ガベージ・コレクションは、Java プログラムのパフォーマンスに決定的な影響を与えるため、その振る舞いを把握することがチューニング作業においてきわめて重要となります。そこで今回は、Java のガベージ・コレクションの役割を説明し、ログの記録方法などを解説します。

ガベージ・コレクションを知る

Java オブジェクトとヒープ

Java プログラムの実行中には、Java オブジェクトが生成されます。Java オブジェクトは、Java プログラムの起動時に生成される場合もあれば、実行中に必要に応じて生成されることもあります。いずれの場合も、JVM（Java 仮想マシン）の内部で新しい Java オブジェクトが作成されると、「ヒープ」と呼ばれるメモリ領域に、同オブジェクトを格納するための領域が割り当てられていきます。

このヒープのサイズは、JVM の起動時にオプションで指定することができます。例えば、以下のコマンドは、ヒープの最大値を 240 MB に制限して JVM を起動する例です。

```
$ java -Xmx240m <クラス名>
```

この場合、もし JVM 内部で大量の Java オブジェクトが生成され、そのサイズが 240MB を超えると、JVM は例外「OutOfMemoryException」を発して Java プログラムの実行を停止します。

このように、全ての Java オブジェクトは、JVM のヒープの内部に順次格納されていく仕組みです。

ガベージ・コレクションの役割

Java 言語では、ヒープ上のメモリ領域を Java オブジェクトに割り当てたり、もしくは割り当て済みの領域を解放したりする処理を、プログラムコードに明示的に記述する必要がありません。これは、C や C++などの言語との大きな違いです。C や C++では、プログラムが使用するメモリ領域の割り当てや、使用済みの領域の解放を、プログラムコードから明示的に指示しなくてはなりません。

Java 言語では、こうしたメモリ管理が JVM によって自動的に行われます。つまり JVM は、Java プログラムのどこからも参照されなくなった不要な Java オブジェクトを見つけ出し、そのメモリ領域を自動的に解放します。こうした Java オブジェクトの削除処理は「ガベージ・コレクション」と呼ばれます。JVM 内部では、このガベージ・コレクションが独立したスレッドとして定期的に動作しています。ガベージ・コレクションが動作する頻度は、JVM のヒープ・サイズやヒープに対する需要、その他の様々な要因によって変化します。

ガベージ・コレクションによるパフォーマンスへの影響

ここで注意すべき点は、JVM がガベージ・コレクションを実行している間、他の全てのスレッド（Java プログラムを実行中のスレッドや Java バイト・コードをコンパイル中のスレッドなど）の実行が停止してしまうことです。ガベージ・コレクションは、Java プログラムのパフォーマンスに決定的な影響を与えるため、その振る舞いを把握することがきわめて重要となります。

図 1 は、8 個の CPU を搭載したマシン上で 1 つの JVM を実行し、ガベージ・コレクションの動作による CPU 負荷状態への影響を示したものです。同図に示されるように、ガベージ・コレクションによって高負荷状態となる CPU は 1 個だけであり、他の CPU はほぼアイドル状態となっています。つまり、ガベージ・コレクションは、Java プログラム全体をスローダウンさせる大きな要因となるわけです。

ガベージ・コレクションの挙動をコントロールするためには、JVM 起動時のオプションを利用し、ヒープ・サイズ等を明示的に指定する必要があります。これについて詳しくは本連載で追って説明しますが、まずは最初の一歩として、JVM 内部でどの程度の頻度でガベージ・コレクションが発生しているかを把握することから始めます。

HP-UX の場合、ガベージ・コレクションが頻繁に発生しているかどうかは、glance や gpm を利用してその手がかりを得ることができます。例えば、gpm の画面上に図 2 のような突発的な CPU 負荷上昇のパターンが見られる場合は、ガベージ・コレク

ションが必要以上に頻繁に発生している可能性が高いと言えます。また同図を詳しく見ると、ユーザー時間が断続的に大きく落ち込んでおり、ガベージ・コレクションによって CPU リソースが奪われていることが分かります。

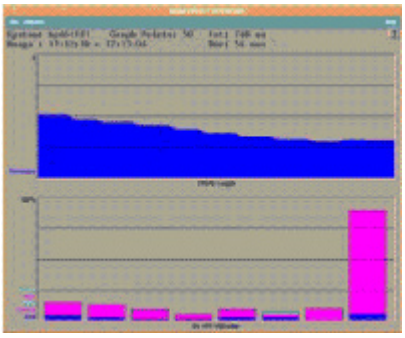

図 1：CPU 負荷状態への影響

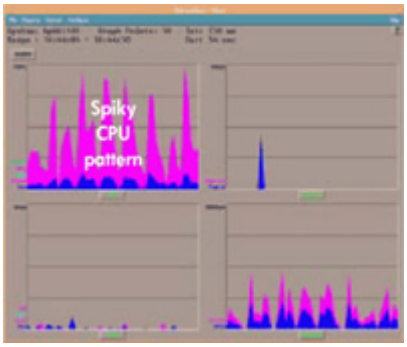

図 2：突発的な CPU 負荷上昇のパターン

ガベージ・コレクションを分析する

JVM オプション「-Xverbosegc」の活用

ガベージ・コレクションが実際にどのタイミングで発生しているかを知るには、以下のように JVM のオプションとして「-Xverbosegc」を指定します。

```
$ java -Xverbosegc:file=myfile.out ＜クラス名＞
```

これにより、「myfile.out」という名前のファイルにガベージ・コレクションのログを記録することができます。このファイル名は、自由に変更することができます。ちなみに、このオプションの指定によるパフォーマンスの低下は、ディスクへのファイル出力を除けば非常に小さなものです。よって、必要に応じて、実運用システムにも使用することができます。

このオプション指定によって出力されるログには、以下のような内容が記録されます。

```
<GC: -1 21.040560 1 824 1 13369144 0 13369344 0 1703936 1703936 01206120 50331648 4793888 4793888 4980736 0.124737 >< GC: -1 24.040994 2 80 1 13369344 0 13369344 1703936 1703936 17039361206120 5058264 50331648 5032840 5032840 5242880 1.896397 >< GC: 2 28.602708 1 216 1 1109352 0 13369344 1703936 0 1703936 50582646780520 50331648 5242688 5242688 5242880 0.870284 >< GC: -1 45.500549 3
```

```
656 32 13369136 0 13369344 0 805256 17039366780520 6780520 50331648 8720800 8720800 8916992
0.055405 >
```

このログには、ガベージ・コレクションが実行されたタイミングやその範囲など、様々な情報を表す 19 のカラムが含まれています。以下のコマンドを実行することで、各カラムの意味を説明するヘルプメッセージを表示できます。

```
$ java -Xverbosegc:help
```

GC のログを見やすく加工する

上記例のように、ガベージ・コレクションのログ内容をそのまま読んで理解することは困難です。そこで HPE では、同ログを見やすく整形するためのツール「processVerboseGC.awk」を提供しています。

ツールをダウンロードしたのち、以下のコマンドを実行します。

```
$ cat myfile.out |awk -f processVerboseGC.awk > output.txt
```

これにより、以下のような読みやすい形態にログを変換することができます。

```
GC: Full GC required - reason: Old generation expanded on last scavenge

GC: Full 1.985317 s since last: 1.985317 s gc time: 96 ms
eden: 1834928->0/3670016          survivor: 120576->0/262144
tenure: 2          old: 3204992->2356088/3928064

GC: Scav 2.190710 s since last: 0.205393 s gc time: 4 ms
eden: 3669968->0/3670016          survivor: 0->120720/262144
tenure: 32          old: 2356088->2356088/3928064
```

このログは、Java プログラムの実行中に発生した 3 回のガベージ・コレクションを順番に示しています。その発生時刻は、Java プログラムの開始時から経過した時間として、文字列「Full」もしくは「Scav」に続いて表示されています。また、その後には「eden」や「survivor」、「tenure」、「old」といった文字列が表示されていますが、これらの単語の意味は本連載で追って説明する予定です。

Full GC の発生に注目する

文字列「Full」は、「Full ガベージ・コレクション（Full GC）」の発生を表します。一方、文字列「Scav」は、「Scavenge ガベージ・コレクション（Scavenge GC）」を表します。次回説明するように、前者の GC は後者に比べて長い処理時間を要します。よって、Full GC が頻繁に発生している場合（例えば 1 分間に 1 回以上）は、何らかの対策を講じなくてはなりません。

また、Java アプリケーションによっては、開発者が記述した Java プログラムあるいはライブラリによって、明示的にガベージ・コレクションを呼び出している場合があります。その際には、上記ファイルに以下のようなログが記録されます。

```
Full GC required - reason : call to System.gc() or Runtime.gc()
```

こうした明示的なガベージ・コレクション実行は、パフォーマンスの観点からは可能な限り避けなくてはなりません。よって、Java プログラムから「System.gc()」もしくは「Runtime.gc()」と記述されているコードを取り除くか、もしくは JVM オプションとして以下のように「-XX:+DisableExplicitGC」を指定し、明示的なガベージ・コレクション実行を抑制します。

```
$ java -XX:+DisableExplicitGC ＜クラス名＞
```

GC ログの内容を観察する

さて、上記ログの内容について、もう少し詳しく見ていきましょう。先に述べたとおり、「Full」あるいは「Scav」のすぐ後に続く数字は、Java プログラム開始後の経過時間です。この時間（単位は秒）を見れば、個々の GC イベントがどの時点で発生したのかを知ることができます。また、文字列「since last」の後には、直前の GC から経過した時間を表す数字が記されます。文字列「gc time」の後には、この GC イベントの完了までに要した時間が表示されます。

これらの時間を注意深く観察することで、JVM 内部におけるガベージ・コレクションの挙動を把握することができます。正常に動作する JVM の場合、Full GC の回数は Scavenge GC よりも少なくなります。また、Scavenge GC は、0.5 秒以上の間隔を空けて数秒ごとに実行され、通常は 1 回あたり 3〜400ms 以内で完了します。

もし Scavenge GC に 1 秒以上の時間が費やされている場合は、Java プログラムを実行しているスレッドがその間完全に停止していることになり、パフォーマンスの問題を抱えていることが分かります。これを解決するには、JVM のオプションを指定して、ヒープ内部の各領域の利用度に応じたヒープ・サイズを指定します。この方法について詳しくは次回説明します。

Full GC は、ヒープ・サイズによっては数秒を要することもあります。通常は、Full GC が 1 秒以内に終了し、Scavenge GC と比較して少ない頻度で発生するように JVM を設定するようにします。例えば、Full GC が 5〜10 分おきに発生し、Scavenge GC は 5 から 10 秒おきに発生するような間隔であれば、その JVM の振る舞いは正常であると言えます。なお現在、HPE では、ログ情報をもとにグラフィカルに GC の分析を行うためのツール「HPjtune」を提供しています。HPjtune を利用することによって、GC に関連する問題点を容易に発見することができます。

以上、今回は、Java 言語におけるガベージ・コレクションの役割を解説し、その発生頻度をログに記録する方法を説明しました。次回は、JVM のヒープ・メモリの構造を説明し、パフォーマンスを最大限に引き出すためのチューニング方法を掘り下げる予定です。

第 3 回

ヒープ・メモリ管理

2004 年 5 月

今回は、Java におけるヒープ・メモリ管理の詳細を説明します。JVM のヒープ・メモリの中で、新しいオブジェクトと古いオブジェクトがどのように配置されるかを理解することで、ヒープ・メモリが有効に利用されているか否かを判断することができます。また、JVM が出力するガベージ・コレクションのログを解析し、オプションの指定によってヒープ・メモリのサイズを適切にチューニングする方法を紹介します。

Java ヒープ・メモリの構造

Java におけるガベージ・コレクションのメカニズムを理解するには、まずヒープ・メモリの構造を知っておく必要があります。

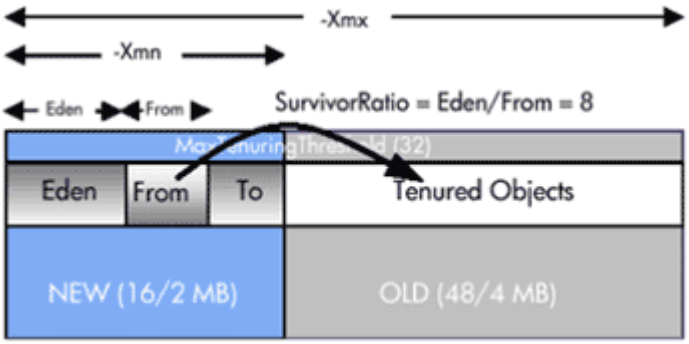

図 1 : JVM ヒープ・メモリの構造

図 1 は、JVM におけるヒープ・メモリの構造を示したものです。この図が示すように、ヒープ・メモリの中には新しいオブジェクトを格納する「NEW」領域と、古いオブジェクトを格納する「OLD」領域が存在します。生まれてすぐに不要となる短命なオブジェクトは NEW 領域でその一生を過ごし、長時間存在するオブジェクトは OLD 領域に留まることになります。また、図 1 には示されていませんが、ヒープ・メモリには JVM にロードされたクラスの置き場所として利用される「Permanent」領域も存在します。

JVM では、「Scavenge GC」と「Full GC」という 2 種類のガベージ・コレクションが実行されます。Scavenge GC は NEW 領域のみを対象とした短時間で終了するガベージ・コレクションであり、頻繁に実施されます。一方、Full GC は NEW と OLD 両方の領域を対象とした大がかりなガベージ・コレクションであり、比較的低い頻度で実施されます。こうした理由から、ヒープ・メモリ全体がオブジェクトの世代別に分割されています。

JVM のデフォルト設定では、NEW 領域の起動時のサイズが 2MB、最大サイズが 16MB にセットされています。また、OLD 領域の起動時のサイズは 4MB、最大サイズは 48MB です。これらのサイズは、以下の JVM オプションを用いることで変更できます。

オプション名	内容
-Xms	ヒープ・メモリ全体の起動時のサイズ
-Xmx	ヒープ・メモリ全体の最大サイズ
-Xmn	NEW 領域のサイズ
-XX:SurvivorRatio=<n>	Eden 領域のサイズを From または To 領域のサイズで割った値（From と To 領域は同じサイズ）

ここで、最初の 3 種類のオプションについては、指定するサイズをメガバイト単位で指定します。例えば「-Xmn256m」は、NEW 領域に 256MB を割り当てることを意味します。これらのオプションの使用方法については、以下に詳しく説明します。

世代別ガベージ・コレクションのメカニズム

図 1 に示されるとおり、NEW 領域内はさらに「Eden」「From」「To」という 3 つの領域に分割されています。これらのうち、Eden 領域は、新しいオブジェクトが作成された際に最初に配置されるメモリ領域です。よって同領域は、時間が経つとともに新しいオブジェクトで埋め尽くされていきます。

Eden 領域が満杯になると、前述した Scavenge GC が実行されます。このとき、まだ使用中のオブジェクト（すなわち、他のオブジェクトから参照されているオブジェクト）については Eden 領域から To 領域へと移動され、参照されていない不要なオブジェクトは破棄されます。これにより、Eden 領域全体がクリーンアップされ、再び新しいオブジェクトを受け入れ可能になります（図 2）。

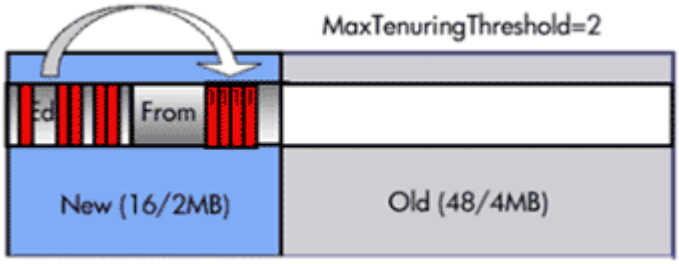

図 2：Scavenge GC による Eden 領域から To 領域へのオブジェクト移動

ここで、図 2 の To 領域に移動されたオブジェクトに「1」という数値が振られていることに注目してください。これは Scavenge GC によってオブジェクトが移動した回数を表しています。この数値の意味については追って説明します。

続いて、Eden 領域が再度満杯になり、2 回目の Scavenge GC が実行される状況を考えます。このとき、前回の GC 時の From 領域と To 領域は互いに入れ替わることになります。つまり、前回の GC でオブジェクトの移動先とされた To 領域は、次回の GC では From 領域として扱われるわけです。そして、この From 領域と Eden 領域にある使用中のオブジェクトが、再び To 領域に移動されます（図 3）。

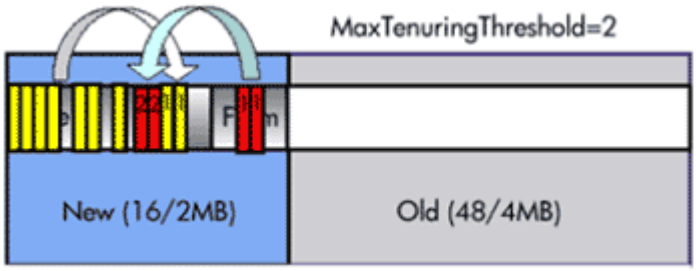

図 3：2 回目の Scavenge GC によるオブジェクト移動

このように、Scavenge GC では、オブジェクトが不要になるまでの間、From 領域と To 領域の間でオブジェクトの移動を繰り返します。そして、この移動の際には、オブジェクトに振られた移動回数の数値がカウントアップされます。図 3 を見れば、From 領域から To 領域へ移動したオブジェクトの移動回数が「2」に増えていることお分かりいただけるでしょう。Scavenge GC では、この移動回数が「MaxTenuringThreshold」と呼ばれるしきい値を上回るオブジェクトについて、OLD 領域への移動を行います。

JVM における MaxTenuringThreshold のデフォルト値は 32 に設定されています。よって新しいオブジェクトは、最大で 32 回まで Scavenge GC の対象となり、その間を From 領域と To 領域の中で過ごします。この回数を超えて生き延びたオブジェクトは、「寿命の長いオブジェクト（tenured object）」として OLD 領域に移動される仕組みです。

Scavenge GC のチューニング

ここまでの説明によって、前回説明したオプション-Xverbosegc によるガベージ・コレクション・ログの内容をより理解できるはずです。例えば、ログ中の以下のような出力を考えてみます。

```
eden: 1834928->0/3670016
```

このログは、以下のように解釈します。

- GC 前、Eden 領域の消費サイズは「1834928」バイトであった
- GC 後、Eden 領域の消費サイズは「0」バイトであった（つまり全オブジェクトが移動もしくは破棄された）
- GC 後、Eden 領域のサイズは「3670016」バイトであった

From 領域および To 領域についても、これと同様の方法でログの解釈が可能です。

```
survivor: 120576->0/262144
```

「survivor」とは、From 領域と To 領域両方を指します。ここでもし、上記ログのように GC 後の From/To 領域の消費サイズが「0」となった場合は注意が必要です。これはすなわち、オブジェクトが From 領域と To 領域の間を行き来せず、すぐに OLD 領域に移動してしまっていることを表します。このような状況では、OLD 領域は短命なオブジェクトですぐに埋まり、Full GC が頻発してしまいます。これはオーバーフローと呼ばれ、MaxTenuringThreshold 値の低い状態で一連の GC が発生している状況を見つけることで検出できます。

NEW 領域のサイズ調節

MaxTenuringThreshhold の値は、ガベージ・コレクションが進行するに従い調整されるため、プログラム実行中は常に変化する可能性があります。NEW 領域が小さすぎ、オブジェクトを短期間しか保持できない状況では、MaxTenuringThreshhold の値が低下しオブジェクトは NEW 領域から OLD 領域へ過度に移動しやすくなります。これは避けるべき状況です。そこで、MaxTenuringThreshhold を可能なかぎり 32 に近づけ、Full GC の回数を減らすことを目的として、以下のようなチューニングを実施します。

NEW 領域を拡大するには、-Xmn オプションを用いて同領域のサイズを指定します。これにより、NEW 領域内の全ての領域（Eden、From、To）が拡張されます。同オプションは、以下のように使用します。

```
$ java -Xmn160m -Xmx480m -Xms480m ＜クラス名＞
```

最大ヒープ・サイズ（-Xmx で指定）に対する NEW 領域の割合は、1/3 から 1/2 が推奨される値です。NEW 領域をより積極的に拡張したい場合は、1/2 程度のサイズを指定します。
また、From 領域と To 領域のサイズは、Eden 領域のサイズに対する比率を指定して設定できます。具体的には、オプション-XX:SurvivorRatio を以下のように使用します。

```
$ java -Xmn120m -Xmx480m -Xms480m -XX:SurvivorRatio=8 ＜クラス名＞
```

これにより、Eden 領域 のサイズは From/To 領域のサイズの 8 倍となります（From 領域と To 領域のサイズは同じです）。上記の例の場合、NEW 領域は 96MB（8×12MB）の Eden 領域と、それぞれ 12MB の From/To 領域に分割され、これらの合計は 120MB となります。ちなみに、-XX:SurvivorRatio の値はデフォルトのままにしておくことが推奨されています。この値は JDK 1.3.1 では 8 に設定されています。

チューニング結果の解析

オプション-Xverbosegc を指定すると、非常に大量のログが出力されます。このログを MS Excel などの表計算ソフトに取り込むのも 1 つのテクニックです。これにより、時間の経過とともに NEW 領域と OLD 領域が増減する様子をグラフに描くことが可能です。また、表計算ソフトを利用する以外にも、HPE が提供する Java ベースのチューニング・ツール HPjtune を使用することもできます。HPjtune は、以下の URL から入手することができます。

HPjtune ダウンロード

ここでは簡単な例として、Java ベンチマークプログラムの 1 つである「SPEC JBB2000」実行時の GC による OLD 領域のサイズ変化を観察してみます。

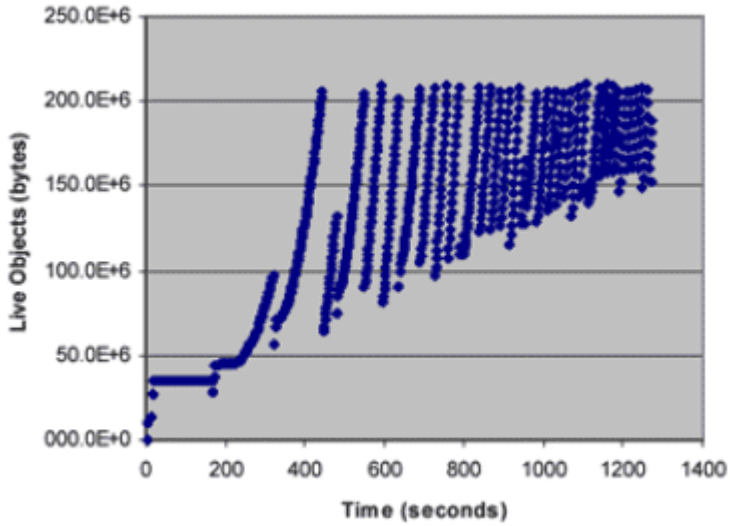

図 4：SPEC JBB2000 実行時の OLD 領域のサイズ変化（チューニング前）

図 4 は、ヒープ・サイズのチューニングを行わずに SPEC JBB2000 を実行した結果です。同図を見れば、GC が頻発して OLD 領域が急激に増加していることが分かります。これは NEW 領域が小さすぎるためにオーバーフローが発生し、OLD 領域がオブジェクトであふれている状況を示しています。また、Full GC のたびに同領域が大幅に縮小していることも分かります。つまり、短命なオブジェクトまでもが OLD 領域に移動されていると推測できます（なお、OLD 領域のサイズが増加傾向にあるのは、SPEC JBB2000 の特性によるものです）。

一方、図 5 は、JVM のオプション指定により、ヒープ・サイズのチューニングを実施した後の結果です。

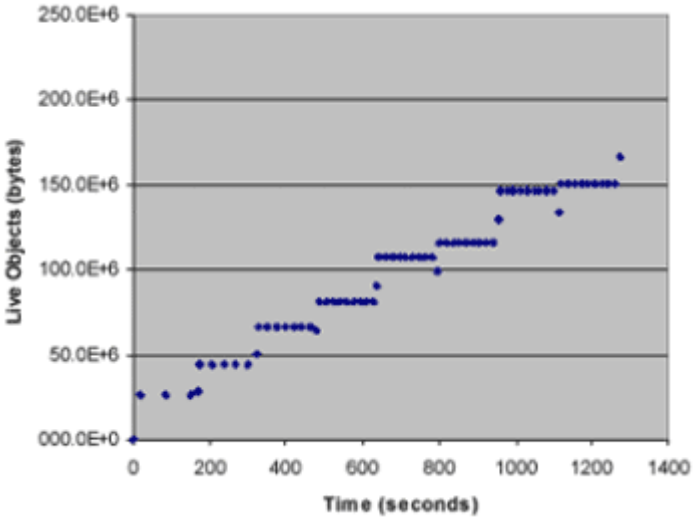

図 5：SPEC JBB2000 実行時の OLD 領域のサイズ変化（チューニング後）

図 4 と比較すると、GC の回数も減少し、OLD 領域の急激な増加も見られません。これは、NEW 領域や OLD 領域の拡張、および SurvivorRatio の変更などを実施したことにより、ヒープ・メモリが理想的な利用状況に調整されたことを示しています。

以上、今回は Java におけるガベージ・コレクションのメカニズムを解説し、JVM オプションの指定によるチューニング方法を説明しました。次回は、JVM のスレッドによるリソース競合の解消について解説する予定です。

第 4 回

マルチスレッドによるリソース競合

2004 年 6 月

今回は、リソース競合の問題を解消するためのチューニング手法として、Glance と HPjmeter という 2 つのツールと、HP JVM のスタックトレース機能を利用する方法を紹介します。

マルチスレッド利用の注意点

Java 言語の特長は、プログラム内で簡単にマルチスレッドを利用できる点にあります。しかしこれは、いわば諸刃の剣です。なぜなら、マルチスレッド・プログラミングに習熟していないプログラマがスレッドをむやみに利用すると、問題を生じることが多々あるからです。

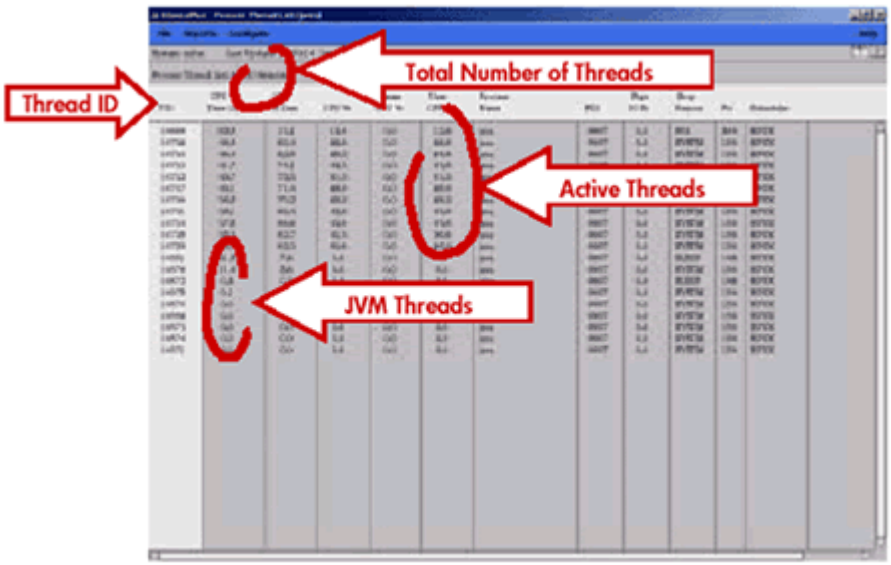

図 1：Glance/gpm による Java スレッドのモニタリング

図 1 は、HP-UX に付属するツール Glance/gpm を利用して、実行中の JVM プロセス内で動作する全てのスレッドを表示した例です。ここでは、JVM が内部的に使用する 11 のスレッドに加えて、Java アプリケーションの実行にともない生成された複数のスレッドが表示されています。こうしたスレッドを多用する Java アプリケーションのプログラミングでは、スレッドを無制限に生成しないよう配慮する必要があります。また、以下の 2 つのポイントに気をつけなくてはなりません。

【OS の上限数を越えるスレッドを作成しない】

大半の JVM では、OS のスレッドを利用して Java スレッドを実装しています。よって、Java スレッドをあまりに大量に生成すると、OS が定める 1 プロセスあたりのスレッド数の制限を超えてしまうのです。こうした場合、Java プログラムの実行中に OutOfMemory エラーが発生したり、「スレッドが多すぎる」と指摘するメッセージが表示されたりします。

もっとも、この問題への対処は比較的簡単です。HP-UX の場合、HP が提供するツール「HPjconfig」を利用することで、カーネル・パラメータ「max_thread_prod」に設定すべき推奨値を算出できます。この値をもとに、管理ツール「SAM」を利用して同パラメータの変更を行います。HPjconfig は、以下の Web サイトからダウンロードできます。

HPjconfig

【ロックを長時間保持しない】
複数のスレッド間でリソースを共有するアプリケーションでは、あるスレッドがリソースのロックを長い間保持してしまうと、他のスレッドの処理が停止してしまいます。この状態をロック競合といいます。ロック競合の起こりやすさは、ロックを獲得しようとするスレッドの数と、獲得の頻度によって決まります。ロック競合が過大に発生すると、スレッドは有意義な作業を進めることができず、ロックの解放待ちに大半の時間を費やしてしまいます。

この 2 つのうち、Java のパフォーマンス・チューニングでは、ロック競合の軽減が重要なポイントになります。ロック競合はアプリケーションの設計のまずさに起因して発生するため、設計変更によってその大半は解消できます。そこで以下、このロック競合問題の検出と解決の方法について説明します。

Glance による解析

ロック競合問題に対処するための最初の手掛かりは、Glance/gpm の出力より得られます。

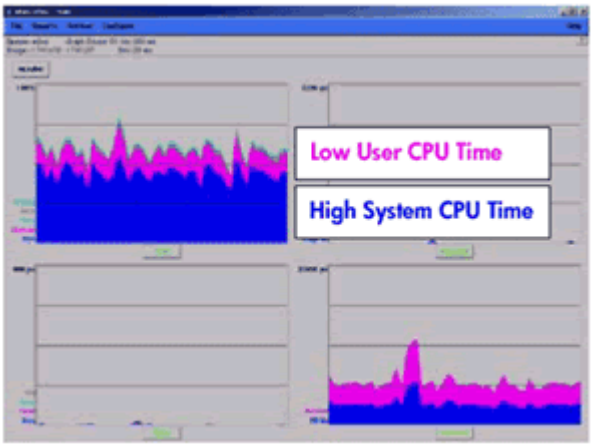

図 2：CPU 時間の割合が示すロック競合の兆候

図 2 は、Java アプリケーションによるロック競合が発生している状態で Glance/gpm による計測を行った例です。これを見れば、ほとんどの CPU 時間が青色の「System」（OS コード）によって消費され、紫色の「User」（アプリケーション・コード）の割合が低くなっていることがわかります。しかし本来、Java アプリケーションの動作が正常であれば、CPU 時間の大半を User が占めていなくてはなりません。これは、ロック競合の多発を示すひとつの指標となります。

ここで再び Glance/gpm を利用し、JVM プロセスによって呼ばれているシステム・コールを観察してみます。

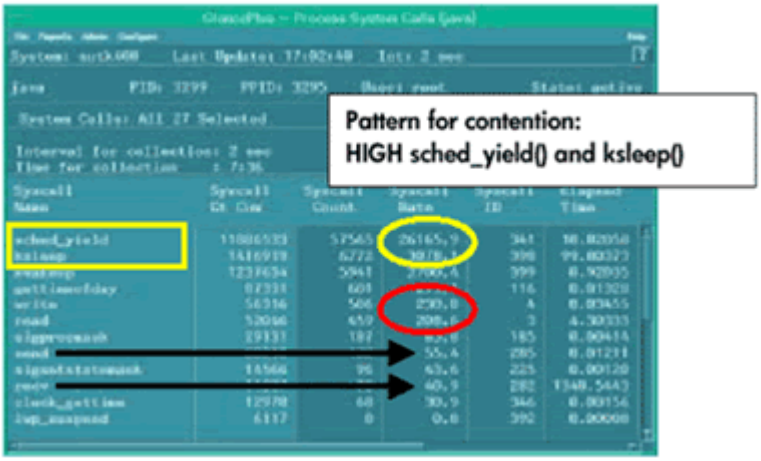

図 3：ロック競合を示すシステム・コールの呼び出し回数

ここでは、「Cumulative system call count」の項目に注目します。すると、「sched_yield」や「ksleep」、「kwakeup」などの特定のシステム・コールがかなりの頻度で呼び出されていることが分かります。また、これらに比較して、図 3 の矢印で示される「send」と「recv」の呼び出し回数はかなり低くなっています。この 2 つのシステム・コールは、Java アプリケーションのネットワーク処理を司るため、本来であればもっと高い頻度で呼び出されなくてはなりません。つまり、アプリケーション設計の不具合によるロック競合の影響で、呼び出される回数が低く抑えられているのです。

ちなみに、ネットワーク機能を利用する Java アプリケーションでは、図 3 のような状況がしばしば発生します。その原因は、ネットワーク・コネクションのオープンにともなうスレッドの大量生成にあります。つまり、数百や数千といった多数の TCP ソケットをオープンすると、同じ数だけスレッドが生成されてしまい、オーバーヘッドがきわめて高くなってしまうのです。この問題は、JDK 1.4 で導入された非同期 I/O の利用により、スレッド数を減らすことで解決できます。ただし、本連載の範囲を超えるため、その詳細については省略します。

HP JVM によるスタックトレース出力

HP-UX 対応の Java 仮想マシン「HP JVM」は、Java のスレッドの挙動を分析するための、非常にシンプルで強力な手段を提供しています。その使い方はとても簡単で、動作中の JVM プロセスに対して SIGQUIT シグナルを送信するだけです。これにより、JVM の動作には影響を与えることなく、その時点で存在する全てのスレッドの詳細情報を記したスタックトレースを取得することができます。
まずは、以下のコマンドを実行し、JVM プロセスのプロセス ID を特定します。

```
# ps -ef | grep java
```

つづいて、以下のコマンドを実行し、JVM プロセス対して SIGQUIT シグナルを送ります（コマンドの実行にはスーパーユーザー権限が必要な場合もあります）。

```
# kill -SIGQUIT ＜JVM のプロセス ID＞
```

これにより、標準出力に以下のようなスタックトレースが出力されます。場合によっては大量のスタックトレースが表示されるため、そのときは JVM の起動時に標準出力からファイルへのリダイレクトを指定しておけばよいでしょう。

```
"Worker Thread 17" prio=9 tid=0x1310b70 nid=41 lwp_id=14165 suspended[0x1194d000..0x11948478]
at fields.FieldPropertiesLibraryLoader.forClass(FieldPropertiesLibraryLoader.java:67)
- waiting to lock <0x3ca45848> (a java.lang.Object)
at fields.FieldsServiceImpl.getFpl(FieldsServiceImpl.java:75)
at fields.FieldsServiceImpl.getFpl(FieldsServiceImpl.java:64)
at base.core.BaseObject.getFpl(BaseObject.java:2930)
at base.core.BaseObject.getFieldProperties(BaseObject.java:2661)
at core.BaseObject.getFieldProperties(BaseObject.java:2670)
at fields.FieldProperties.getFieldsInGroup(FieldProperties.java:1157)
at fields.FieldProperties.getFieldsInGroup(FieldProperties.java:1107)
```

この例では、あるスレッドがスリープ状態にあり、オブジェクトのロックの解放を待ち続けている様子が示されています。こうしたスタックトレースを調べていくことで、ロックを長時間保持しているスレッドを見つけ出し、JVM 全体のスローダウンの原因を突き止めることができます。

HP JVM におけるこうしたスタックトレース解析の欠点は、多数のスレッドを含む Java アプリケーションにおいて大量のスタックトレースが生成されてしまうことです。その出力を綿密にチェックしていくのは非常な労力を要します。そこで、こうした作業を簡素化するために、次に紹介する HPjmeter を合わせて利用します。

HPjmeter によるロック競合の解析

HP JVM のバージョン 1.3.1 以降では、同 JVM 独自の拡張プロファイリング機能を利用できます。同機能によって収集されるデータを、HPE が無償で提供するプロファイリング・ツール HPjmeter で解析することで、ロック競合に関する正確な情報が得られます。
拡張プロファイリング機能を利用するには、以下のようにオプションを指定して JVM を起動します。

```
$ java -Xeprof:file=myfile.eprof ＜クラス名＞
```

これにより出力されるファイル（上記例では myfile.eprof）を分析するためのツールが、HPjmeter です。同ツールは、以下の Web サイトよりダウンロード可能です

HPjmeter

```
$ java -Xmn120m -Xmx480m -Xms480m -XX:SurvivorRatio=8 ＜クラス名＞
```

HPjmeter は Java アプリケーションであるため、JVM が動作するあらゆるプラットフォームで動作します。同ツールは、上述の拡張プロファイリング機能による出力ファイルだけでなく、Java の標準のプロファイリングによる出力ファイルも読み込むことができます。ちなみに HPjmeter は、スレッド挙動の解析だけでなく、その他の数多くの解析機能を備えています。これらについて詳しくは、上記 Web サイトのチュートリアルを参照してください。

HPjmeter は、JVM 内で動作するスレッドの状態と生存期間をグラフィカルに表示します。上述の-Xeprof オプションで得られた出力ファイルをロードし、Metric メニューの Thread Histogram を選択すると、図 4 の画面が表示されます。

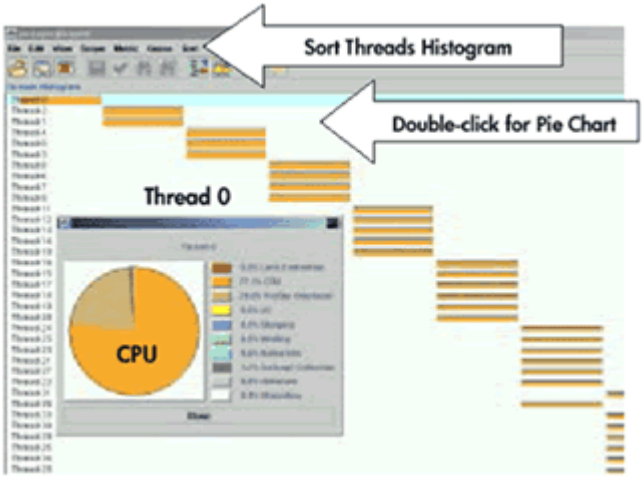

図 4：HPjmeter によるスレッド生存期間の表示

この画面を見れば、全てのスレッドの生存期間を一度に確認することができます。また、個々のスレッドを表すバーをダブルクリックすることで、それをさらに詳しく分析できます。例えば図 4 では、スレッド 0 の詳細を表示し、その存在期間の 77.1%が CPU による実行にあてられていることを確認しています。また、残りの期間は（21.4%）がプロファイリングによるオーバーヘッドであり、実行待ちやロック競合の割合はほとんどないことが分かります。これにより、スレッド 0 が正常な状態で動作していることが分かります。

このように、HP JVM と HPjmeter の組み合わせを用いることで、Java アプリケーションのスローダウンを招くスレッドの問題について、きわめて正確な情報を得られます。また HPjmeter は、他にも多くの場面で威力を発揮します。例えば、CPU を大量に消費するメソッドを調べたり、その実時間を追跡したりできます。これについては次回詳しく紹介する予定です。

スレッド優先度のチューニング

最後に、スレッドの優先度によるチューニングについて説明します。
HP JVM では、Java のスレッドを「LWP」と呼ばれる HP-UX のカーネル・スレッドに対応づけることで実装しています。このカーネル・スレッドの優先度の初期値は、JVM プロセスの優先度をもとに決定されます。そして通常は、CPU 時間を消費するにしたがい、次第にその優先度が低くなっていきます。OS は、スレッドの優先度に基づいてスケジューリングを実施するため、JVM 以外のスレッドを優先して実行するようになります。

そこで、マシン上では他に重要なアプリケーションを動作させていない場合は、JVM プロセスに高い優先度を与えることで Java アプリケーションのパフォーマンスを改善できます。具体的には、スーパーユーザー権限を用いて nice もしくは renice コマンドを利用するか、Glance/gpm の renice 機能を利用します。以下は、nice コマンドの使用例です。

```
# nice --20 java ＜クラス名＞
```

一方、renice コマンドを利用する際には、「ps -ef」コマンドや Glance/gpm の Process list 画面で得られる JVM のプロセス ID を指定します。例えば、プロセス ID が 1234 であるとすると、以下のように入力します。

```
# renice -20 1234
```

なお、nice と renice において指定するオプション「--20」と「-20」は、それぞれハイフンの数が異なることに注意してください。

以上のコマンドを利用することで、JVM プロセス内の全てのスレッドの優先度が上昇します。同一マシン上の他のプロセスの優先度は相対的に低くなり、Java アプリケーションにより多くの CPU 時間が割り当てられることになります。もっとも、この方法には、JVM 以外のプロセスの性能が低くなるというトレードオフがあり、全ての環境に適用できるとは限りません。

以上、今回は、マルチスレッドの挙動に着目したチューニング技法を紹介しました。次回はリソースを消費するメソッド・コールの分析方法を説明する予定です。

第 5 回

メモリ・リークの発見

2004 年 7 月

今回は、Java プログラムにおけるメモリ・リークの発見方法について説明します。

リソースを大量に消費するメソッド・コール

Java パフォーマンス・チューニングのポイントの 1 つは、実行時間やリソースの大半を消費しているメソッドを見つけ出すことです。前回の連載で紹介したツール HPjmeter は、こうした分析作業に最適な機能を備えています。

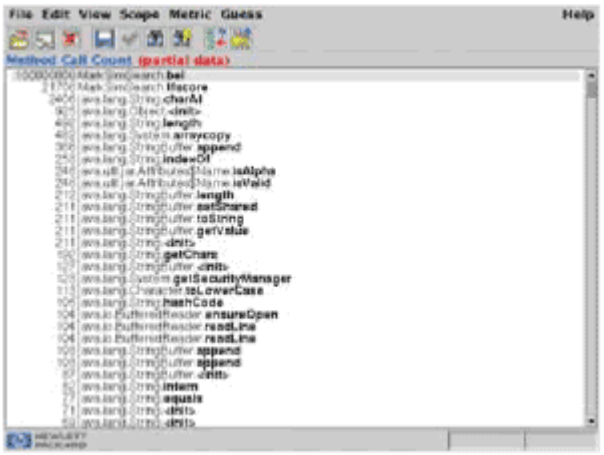

図 1：HPjmeter によるメソッド呼び出し回数の表示

図 1 は、HPjmeter を利用し、Java プログラム中のメソッドを呼び出し回数が多い順に一覧表示した画面です。この例では、「bel」メソッドの実行回数が他のメソッドに比べて非常に多いことが示されています。

また、HPjmeter に備わるもうひとつの計測項目である「Exclusive CPU Method Time（CPU 占有時間）」を使用すれば、Java プログラムの各メソッドの実行時間を知ることができます（図 2）。

図 2：HPjmeter によるメソッド実行時間の表示

メモリ・リークの原因を探る

Java プログラムの開発においてもっとも注意すべきことのひとつとして、JVM のヒープ・メモリ不足があります。Java プログラムの実行中に JVM のヒープ・メモリが不足すると、OutOfMemory エラーが発生し、プログラムの実行を継続できなくなったり JVM が異常終了したりします。

本連載の第 2 回「ガベージ・コレクション」で説明したとおり、Java オブジェクトが必要とするメモリ領域は、JVM のヒープ・メモリから自動的に割り当てられます。また、不要になった（どこからも参照されなくなった）Java オブジェクトはガベージ・コレクションによって収集され、そのメモリ領域は自動的に解放されます。よって、メモリ領域の確保や解放をプログラマが意識する必要がありません。そのため、Java では、C や C++のようなにメモリ・リークが多発することはなくなりました。

たとえば C++の場合、以下のようなコードによって、メモリ・リークが発生します。

```
ptr = new LargeObjectType();
//ptr を用いた処理
ptr = null;
```

ここでは、変数 ptr に C++オブジェクトのポインタを割り当て、使用後に null をセットしています。しかし、正しくは「delete ptr」と記述し、C++オブジェクトのメモリ領域を解放しなくてはなりません。こうした delete し忘れが、C++におけるメモリ・リークのおもな発生原因です。Java では、この delete に相当する作業を JVM のガベージ・コレクタが実行してくるため、こうしたタイプのメモリ・リークは起こらなくなりました。

メモリ・リテンションとは

しかし、Java においても、メモリ・リークが完全になくなったわけではありません。たとえば、Java オブジェクト A のオブジェクト変数 foo が、Java オブジェクト B を参照しているケースを考えます。この場合、たとえ B がプログラムの処理上は不要になったとしても、foo から B への参照が存在するかぎり、B はガベージ・コレクションの対象とはなりません。このように、プログラマが予想しないところでオブジェクトの参照が残ってしまうことを、「メモリ・リテンション」と呼びます。とくに、コレクシ

ョンや配列を参照元とするメモリ・リテンションが多発すると、解放されない Java オブジェクトが継続的に増加し、メモリ・リーク状態に陥ります。

メモリ・リークの発見の方法

こうした Java プログラムのメモリ・リークを発見するには、たとえば米 Sitraka の「JProbe」など市販のプロファイラ・ツールがよく利用されます。また、HP-UX であれば、Glance/gpm の「Memory Regions」画面がメモリ・リークの発見に大変役に立ちます。具体的には、監視対象とする JVM プロセス（java という名前のプロセス）を選択し、同プロセスが使用するすべてのメモリ領域のレポートを生成します。これにより、以下のような画面が表示されます。

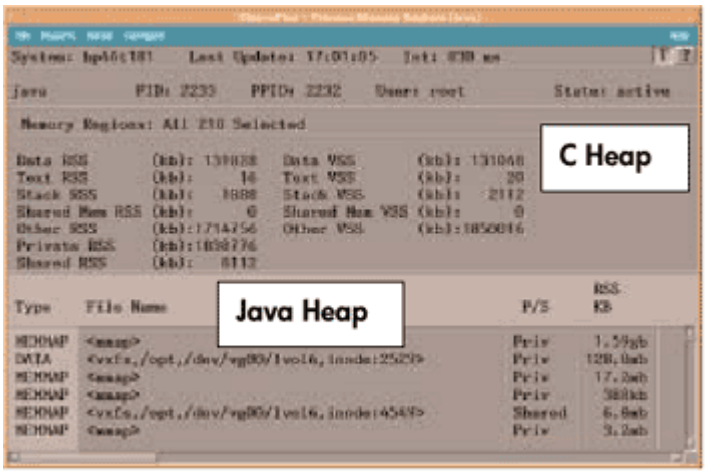

図 3：Glance/gpm による JVM プロセスの消費メモリの表示

図 3 において、Data RSS（resident set size）および Data VSS（virtual set size）のそれぞれの値が、JVM を実装する C プログラムが使用するメモリ・サイズを示します。一方、Other RSS と Other VSS、そして Private RSS は、JVM 内部のヒープ・メモリを含んでいます。これらの領域のうちいずれかに継続的な増加が見られる場合、その Java プログラムはメモリ・リテンションの問題を抱えているとみて間違いないでしょう。

また、JVM のメモリ・リークの原因は、メモリ・リテンション以外にも考えられます。たとえば、Java アプリケーションの中には、C もしくは C++プログラムを内部で呼び出しているものも少なくありません。それらの C もしくは C++プログラムにおいてメモリ・リークが発生しているケースもよくあります。

いずれにせよ、Glance/gpm を利用し、Data RSS や Data VSS などの値の振る舞いを分析することが、メモリ・リークの発見において重要なポイントとなります。

adviser mode の活用

小規模なメモリ・リークについては、Java プログラムを非常に長い期間に継続して運用してはじめて明らかになることがあります。そうした場合は、上記のように画面上で監視し続けるのは現実的ではありません。そこで、メモリ・リークの有無を厳密にテストするときは、Glance/gpm の「adviser mode」を活用します。
adviser mode とは、Glance/gpm の GUI を起動せず、バッチ形式で動作させるモードのことです。これにより、長期間にわたって収集した大量のデータをファイルに出力し、後ほどゆっくり分析することができます。
以下は、Glance/gpm の adviser mode を利用して 5 秒ごとにサンプルを採取するコマンドの例です。

```
$ /opt/perf/bin/glance ?adviser_only ?syntax adviser_commands -j 1
```

ここでは指定されている「adviser_commands ファイル」には、計測に使用するコマンドを以下のような書式で記述しておきます。

```
PRINT "----- ", gbl_stattime, " (proc name, pid, cpu, negnice cpu, VSS, RSS, thread #, I/O)"

PROCESS LOOP {
    if proc_proc_name == "java" then {
        PRINT proc_proc_name|8|0,
            proc_proc_id|8|0,
            proc_cpu_total_util|8|2,
            proc_cpu_nnice_time|8|2,
            proc_mem_virt,
            proc_mem_res,
            proc_thread_count,
            proc_io_byte_rate
    }
}
```

上述のコマンドを実行すると、メモリ計測の結果が以下のようにファイルに出力されます。

```
----- 20:15:37 (proc name, pid, cpu, negnice cpu, VSS, RSS, thread #, I/O)
java 6390 93.33 0.00 787.9mb 389.7mb 2438 0.0
java 6370 88.66 0.00 788.9mb 308.3mb 2444 0.0
----- 20:15:38 (proc name, pid, cpu, negnice cpu, VSS, RSS, thread #, I/O)
java 6390 80.00 0.00 793.8mb 389.7mb 2482 0.0
java 6370 74.66 0.00 794.7mb 308.3mb 2488 0.0
＜以下略＞
```

ここでは、VSS のサイズが継続的に増加していることがわかります。
メモリ・リークを検出する手段としては、Glance/gpm 以外にも以下のような方法があります。

- JVM オプション-Xverboseegc から得られるログを解析する
- HPjmeter によるメモリ・リークの解析
- gdb（gnu debugger）を用いる

以上、今回は Java プログラムによるメソッド・コールの調査、そしてメモリ・リークの発見について説明しました。次回は引き続き、HPjmeter によるメモリ・リークの解析について詳しく説明します。

最終回

メモリ・リーク解析と HotSpot JVM

2004 年 8 月

今回は、HPjmeter に備わる Compare 機能を活用し、一定時間が経過したあとの残存オブジェクトを洗い出し、メモリ・リークの原因を探る方法を説明します。また後半では、HotSpot JVM ではパフォーマンスを改善できないケースを明らかにし、その対処方法を学びます。

HPjmeter によるメモリ・リーク解析

まず前半では、HPE が提供するプロファイリング・ツール HPjmeter を利用したメモリ・リーク解析の方法を解説します。

前回説明したとおり、Java オブジェクトが必要とするメモリ領域は、JVM のヒープ・メモリから自動的に割り当てられます。また、不要になった（どこからも参照されなくなった）Java オブジェクトはガベージ・コレクションによって収集され、そのメモリ領域は自動的に解放されます。よって、メモリ領域の確保や解放をプログラマが意識する必要がありません。そのため、Java では、C や C++のようなにメモリ・リークが多発することはなくなりました。

しかし、Java においても、メモリ・リークが完全になくなったわけではありません。たとえば、Java オブジェクト A のオブジェクト変数 foo が、Java オブジェクト B を参照しているケースを考えます。この場合、たとえ B がプログラムの処理上は不要になったとしても、foo から B への参照が存在するかぎり、B はガベージ・コレクションの対象とはなりません。このように、プログラマが予想しないところでオブジェクトの参照が残ってしまうことを、「メモリ・リテンション」と呼びます。とくに、コレクションや配列を参照元とするメモリ・リテンションが多発すると、解放されない Java オブジェクトが継続的に増加し、メモリ・リーク状態に陥ります。

メモリ・リーク解析の 1 つの方法として、ガベージ・コレクションを強制実行したあとにプログラムを終了させる方法があります。これを実施するには、プログラムを終了させる部分で以下のように記述します。

```
System.gc()
System.runFinalization()
System.gc()
System.exit()
```

このとき、HP-JVM の起動のオプションには「-Xrunhprof:heap=all,cutoff=0」を追加しておきます。これにより出力されるプロファイル・データを HPjmeter で解析すれば、プログラムが終了した時点で残存するオブジェクトの状態を観察できます。

また、この方法を使えば、一定期間中の残存オブジェクトの増減を知ることも可能です。たとえば、プログラムが 1 時間動作する際に残存するオブジェクトを観察したいとします。そのためには、プログラムを 2 回実行し、2 回目は 1 回目より 1 時間長く動作させます。そして、HPjmeter の Compare 機能を利用して、得られた 2 つのプロファイル・データを比較します。

ここでは、プログラムを 1 時間実行したときと、2 時間実行したときの、それぞれのプロファイル・データを HPjmeter でオープンします。具体的には、以下のメニュー操作を実施します。

1. [File]→[Open]を選択し、2 つのファイルを続けて開きます
2. [File]→[Compare]を選択すると、2 つのデータの概要が表示されます
3. [Metric]→[Residual Objects(Count)]を選択します

以上の操作を行うと、以下の画面が表示されます。

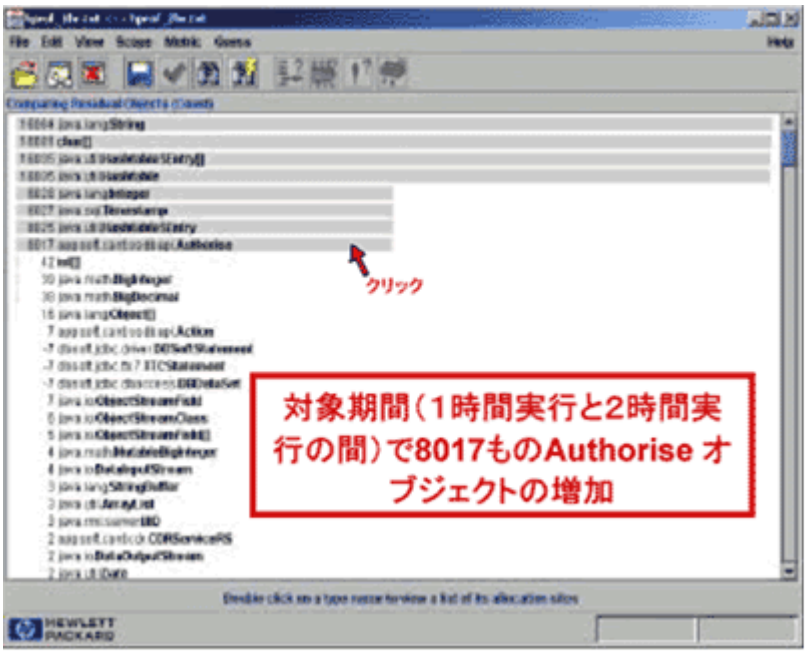

図 1：HPjmeter の Compare 機能による比較

図 1 の比較結果を見ると、1 時間目から 2 時間目の間に、Authorize オブジェクトが 8017 個も増加していることがわかり、メモリ・リークの発生が疑われます。そこで、この Authorize オブジェクトをさらに詳しく解析してみます。まずは、あとで検索しやすくするため、同オブジェクトの行をマークしておきます。

4. Authorize オブジェクトの行をクリックし、[Edit]→[Mark to Find]を選択してマークします

参照グラフツリーの解析

続いて、以下の操作を行い、2 時間実行したプロファイル・データをもとにして Authorize オブジェクトの解析を行います。

5. [File]→＜プロファイル・データのファイル名＞を選択します
6. [Metric]→[Reference Graph Tree]を選択します

これにより、2 時間実行した時点で、JVM 内の全オブジェクトが互いにどのように参照し合っているのかを示すグラフツリーが表示されます。この参照グラフツリーの中から、Authorize オブジェクトのツリーを検索します。

7. [Edit]→[File]を選択します

このとき、検索結果を一度にすべて表示するか、いずれか 1 つを表示するかを選択するダイアログが現れます。Authorize オブジェクトの数が多いので、ここでは[Find Any]を選択します。これにより表示されるツリーを確認すれば、多数の Authorize オブジェクトの参照元となっているオブジェクトがどれかを知ることができます。

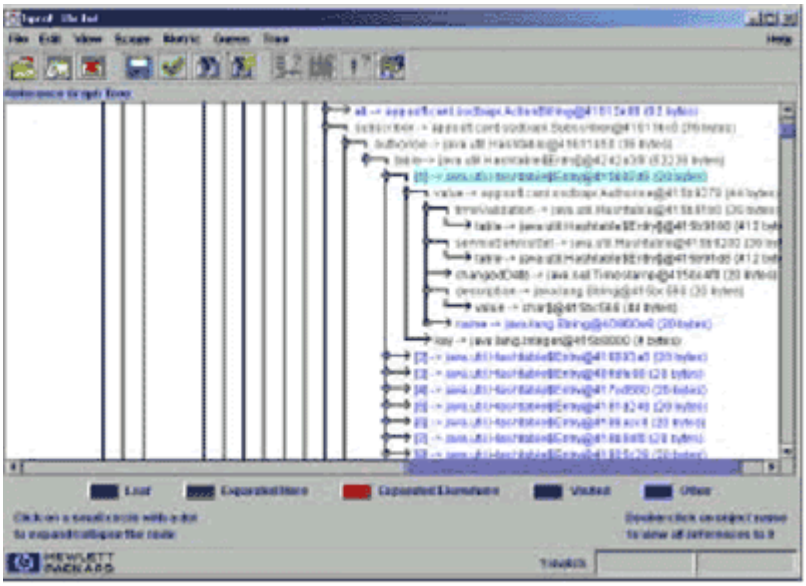

図 2：Authorize オブジェクトの参照グラフツリーの表示

また、HPjmeter 1.2.1 以降では、メモリ・リークの疑いのあるオブジェクトを検出する機能が追加されました。ただし同機能は、「参照が 1 つしかないオブジェクトはメモリ・リークが疑われる」という簡単な経験則に基づいています。そのため、必ずしもそれがメモリ・リークを見つけられるとは限りませんが、着目すべき部分を知る手段としては有効です。この機能を利用するには、以下の操作を行います。

8. [Guess]→[Memory Leaks]を選択します

これにより、メモリ・リークの疑いのあるオブジェクトの ID とサイズ（バイト数）が表示されます。これを先ほどの参照グラフツリーと比較すれば、同じオブジェクトであることがわかります。

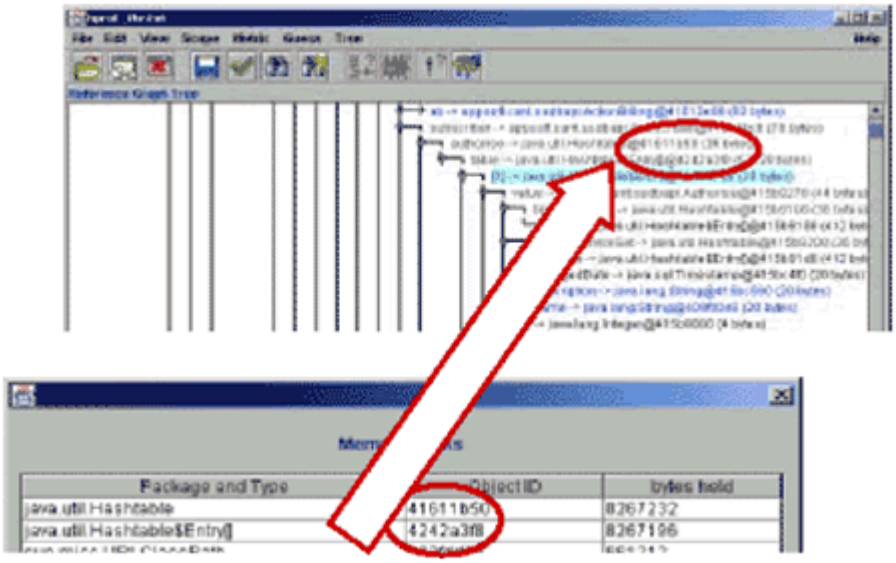

図 3：メモリ・リークの疑いのあるオブジェクトと比較

こうした一連の操作によって、メモリ・リークの可能性の高いオブジェクトが洗い出されたので、このあとは同オブジェクトに対象を絞ってさらなる解析を進めることとなります。このように、HPjmeter を利用することで、メモリ・リーク解析を効率的に実施することができます。

HotSpot JVM の特性を知る

後半では、Java パフォーマンス・チューニングの仕上げとして、HotSpot JVM に固有のふるまいを学びます。HotSpot JVM ではパフォーマンスを改善できないケースを明らかにし、その対象方法を説明します。

アプリケーションのチューニングの効果を比較するためにしばしば作成されるのが、簡単なベンチマーク・プログラムです。しかし、HotSpot ベースの JVM では、ベンチマークの設計のまずさが誤解を生む原因となりかねません。たとえば、ベンチマークによっては、旧型の JIT（Just-In-Time）コンパイラを備えた Classic JVM の方が高速になることもあるのです。

まずは、以下のベンチマークを見てください。

```
public class SimpleBenchmark {
        public static void main(String[] argv) {
                int value=0;

                // Record the start time.
                long start= System.currentTimeMillis();

                // Repeatedly executes feature to measure performance.
                for (int i=0; i<100000000; i++) {
                        // Replace line with your favorite computation.
                        value +=i;
                }
```

```
            // Record the finish time.
            long finish= System.currentTimeMillis();

            // Now report how long test ran.
            System.out.print ("Time spent = " +
            Long.toString(finish - start) + " ms\n");
        }
}
```

ご覧のとおり、ループを 10 億回繰り返し、その経過時間を計るという簡単なベンチマークです。このプログラムをコンパイルし、デフォルトの HotSport JVM で実行すると、以下のような結果が得られます。

```
$ /opt/java1.3/bin/java SimpleBenchmark
Time spent = 24168 ms
```

一方、JVM の起動オプションに-classic を指定し、Classic JVM で実行してみます。

```
$ /opt/java1.3/bin/java -classic SimpleBenchmark
Time spent = 911 ms
```

このように、HotSpot JVM は Classic JVM よりも 25 倍も遅いという結果が得られます。

ここで、JIT コンパイルを一切行わず、すべてをインタプリタ実行するとどのような結果が得られるか、試してみましょう。HP JVM では、-Xint オプションを指定することで、純粋なインタプリタとして JVM を動作させることができます。その結果は以下の通りです。

```
$ /opt/java1.3/bin/java -Xint SimpleBenchmark
Time spent = 24100 ms
```

このように、上述した HotSpot JVM による結果は、インタプリタ実行時の結果とほぼ同じであることがわかります。

HotSpot JVM と Classic JVM

では、なぜこのような違いが表れるのでしょうか。その理由を探るために、HotSpot JVM と Classic JVM の差を比較してみます。

Classic JVM	JIT（Just-In-Time）コンパイラを搭載する。実行されるすべてのコードをコンパイルする
HotSpot JVM	HotSpot コンパイラを搭載する。アプリケーションにおいてもっとも実行頻度の高い部分のみコンパイルする

一般的なプログラムでは、コードの 20%の部分に実行時間の 80%が費やされるという、いわゆる 80/20 の法則が当てはまります。HotSpot JVM は、この 80/20 の法則に基づいて設計されています。つまり、同 JVM では最初はアプリケーションをインタプリタ・モードで実行し、コードの実行頻度の解析を行います。これにより、「HotSpot」すなわち実行頻度の高い部分を特定したならば、その部分についてのみバイナリ・コードへのコンパイルやインライン展開などの最適化を実施します。これに対し、Classic JVM では、実行されるすべてのコードをコンパイルするため、コンパイル時間がオーバーヘッドとなります。

本来であれば、HotSpot JVM の方が Classic JVM よりも優れたパフォーマンスを実現できるはずです。しかし、上述したベンチマークの例では、ループを含むメソッドが 1 回しか呼び出されないため、インタプリタ・モードのままループを実行してしまうのです。

こうした場合は、HP JVM の起動オプションとして-XX:+UseOnStackReplacement を指定し、On Stack Replacement 機能を有効にします*。同機能を利用すると、メソッド呼び出しの最中でも、インタプリタ・モードからコンパイルされたバイナリ・コードの実行に切り替えられるようになります。

*この機能を有効にするためには、現時点ではパッチ (HPUX11.0:PHKL_24943, HP-UX11i PHKL_24751)が必要です。また同時に、コンパイラのセーフポイント機能も有効にする必要があります(-XX:+UseCompilerSafepoints)。詳細は SDK 1.3.1.02 以降のリリースノートを参照してください。

このオプションを指定して先のベンチマークを実行すると、以下のようになります。

```
$/opt/java1.3/bin/java
    -XX:+UseCompilerSafepoints
    -XX:+UseOnStackReplacement SimpleBenchmark
Time spent = 42 ms
```

このように、Classic JVM より優れた結果が得られることがわかります。

HotSpot 最適化にかかる時間

上記のベンチマークのように、比較的短い時間で終了するものは「マイクロ・ベンチマーク」と呼ばれます。しかし上述したとおり、HotSpot JVM では最適化をはじめる前にインタプリタ・モードでコードを実行し、HotSpot の分析に時間を費やします。そのため、マイクロ・ベンチマークでは、HotSpot JVM による最適化の効果をほとんど得ることができません。

HotSpot JVM において、こうしたマイクロ・ベンチマークで十分なパフォーマンスを得るには、ベンチマーク部分をメソッドとして切り出し、それを繰り返し呼び出すようにコードを修正します。以下は、上述のベンチマークを修正した例です。

```
public class HotSpotBenchmark {
        public static void runTest() {
                int value=0;

                // Repeatedly executes feature to measure performance.
                for (int i=0; i<100000000; i++) {
```

```
                // Replace line with your favorite computation.
                value +=i;
        }
    }

    public static void main(String[] argv) {
        // Run benchmark multiple times. This will allow us to
        // see when HotSpot begins executing compiled code.

        for (int i = 0; i < 8; i++) {

        // Record the start time.
        long start= System.currentTimeMillis();

        // Run benchmark test.
        runTest();

        // Record the finish time.
        long finish= System.currentTimeMillis();

        // Now report how long test ran.
        System.out.print ("Time spent = " +
            Long.toString(finish - start) + " ms\n");

        }
    }
}
```

このプログラムを HotSpot JVM で実行すると、以下のような結果が得られます。

```
$ /opt/java1.3/bin/java HotSpotBenchmark
Time spent = 23372 ms
Time spent = 23400 ms
Time spent = 23372 ms
Time spent = 11 ms
Time spent = 11 ms
Time spent = 11 ms
```

ここで、ベンチマーク部分のメソッドはインタプリタ・モードのままで 3 回実行されていることに注目してください。一般には、HotSpot JVM が最適化を終えるまでには 1〜2 分を要します。その最適化の結果、HotSpot JVM では Classic JVM の 80 倍の高速化を達成していることが分かります。

パフォーマンス・チューニングを目的としてベンチマークを作成する際には、それがアプリケーション全体のアーキテクチャを反映できているかを見直してください。HotSpot JVM の効果を高めるには、そのベンチマークが十分に長時間実行され、「HotSpot」部分が繰り返し呼び出されなくてはならないのです。

HP-UX
www.hpe.com/jp/hpux

© Copyright 2018 Hewlett Packard Enterprise Development LP.

本書の内容は、将来予告なく変更されることがあります。日本ヒューレット・パッカード製品およびサービスに対する保証については、当該製品およびサービスの保証規定書に記載されています。本書のいかなる内容も、新たな保証を追加するものではありません。日本ヒューレット・パッカードは、本書中の技術的あるいは校正上の誤り、脱字に対して、責任を負いかねますのでご了承ください。

Hewlett Packard Enterprise