今まで知らなかった 5 つの事項: JVM のためのコマンドライン・フラグ

JVM のパフォーマンスと Java ランタイムを微調整する

Ted Neward
Principal
Neward & Associates

2010年 8月 24日

Java™ 仮想マシンには何百というコマンドライン・オプションが用意されています。経験豊富な Java 開発者であれば、これらのオプションを使って Java ランタイムを調整することができます。コンパイラーのパフォーマンスをモニタリングしてログに記録する方法、明示的なガーベッジ・コレクション (`System.gc();`) を無効にする方法、JRE を拡張する方法などを学びましょう。

このシリーズの他の記事を見る

この連載について

皆さんは自分が Java プログラミングについて知っていると思うかもしれません。しかし実際には、ほとんどの開発者は Java プラットフォームの表面的な部分しか扱っておらず、当面の作業を完了するために十分なことしか学んでいません。この連載では、Ted Neward が Java プラットフォームのコア機能を深く掘り下げ、非常に厄介な Java プログラミングの難題の解決にも役立つ、ほとんど知られていない事実を紹介します。

ほとんどの Java 開発者が当たり前のことと思っている Java アプリケーションの機能やパフォーマンスは、その裏で JVM がフル稼働することで実現されています。それにもかかわらず、JVM が何をどのように実行しているのかを本当に理解している人は、ごく一部の人に限られます。JVM は、オブジェクトの割り当てとガーベッジ・コレクション、スレッドの生成、ファイルのオープンとクローズ、Java バイトコードの解釈ならびに JIT コンパイル、あるいはそのいずれか等々を行っているのです。

JVM を理解していないと、アプリケーションで最高のパフォーマンスが実現できないだけでなく、JVM に何か問題が発生した場合、その問題の修復が非常に困難になります。

この連載、「今まで知らなかった 5 つの事項」の今回の記事では、Java 仮想マシンのパフォーマンスを診断したり調整したりするために使用することができる JVM のコマンドライン・フラグをいくつか紹介します。

1. DisableExplicitGC

私は今まで何度経験したかわからないほどですが、アプリケーションのパフォーマンスの問題で相談にのってくれと頼まれ、コード全体に対して簡単に `grep` を実行してみると、リスト 1 に示すようなコードが見つかる場合があります。これは Java に当初から存在する、パフォーマンスに関するアンチパターンと言えるものです。

リスト 1. System.gc();

```
// We just released a bunch of objects, so tell the stupid
// garbage collector to collect them already!
System.gc();
```

ガーベッジ・コレクションを明示的に実行するコードを作成するのは、極めて愚かなことであり、狂犬病の闘犬と一緒に電話ボックスに閉じ込もるようなものです。ガーベッジ・コレクションを呼び出した場合の厳密な動作は実装に依存しますが、皆さんの JVM が世代別ガーベッジ・コレクション (ほとんどの場合、`System.gc();`) を実行しているとすると、VM は (不必要であるにもかかわらず) ヒープを強制的に「フル・スイープ」します。フル・スイープは通常の GC 操作よりも数桁多く処理時間がかかるのが一般的であり、とにかく非常に非効率です。

ただし、私の言うことを真に受けないでください。Sun のエンジニア達は、この特定の人的エラーによる問題のためだけに JVM フラグを用意してくれました。つまり `-XX:+DisableExplicitGC` フラグを使うと `System.gc()` 呼び出しが自動的に空命令に変更され、`System.gc()` が JVM の実行全体に効果的か有害かを、コードを実行した上で判断することができます。

2. HeapDumpOnOutOfMemoryError

皆さんにも経験があると思いますが、何度試しても JVM が異常終了して `OutOfMemoryError` をスローしてしまうのに、何が問題かを調べるためにデバッガーを設定してエラーをキャッチしようとしても、それがどうしてもできない、という場合があります。このように、時々発生する問題や非確定的な問題によって、開発者は気が狂いそうになります。

使う側の危険負担

必ずしも JVM のすべてのコマンドライン・フラグが、Sun/Oracle 以外のどの VM でもサポートされているわけではありません。あるフラグがサポートされているかどうかを知るための最善の方法は、当然のことながら、実際にそのフラグを試し、動作するかどうかを調べてみることです。ただし、そうしたフラグが厳密にはサポートされていないにもかかわらず、そのフラグを使用する場合には、使用する側が完全な責任を負うことになります。そうしたフラグのいずれかによって、皆さんのコードやデータ、サーバーなどがどうなったとしても、私も Sun/Oracle も IBM® も責任は負いません。万一に備えて、それらのフラグをまず (本番環境ではない) 仮想環境で試してみることをお勧めします。

そうした場合には、まさに JVM が異常終了しようとしている時に、ヒープのスナップショットを取りたいものです。それをしてくれるのが `-XX:+HeapDumpOnOutOfMemoryError` コマンドです。

このコマンドを実行すると、JVM に対し、「ヒープ・ダンプのスナップショット」を取って処理用のファイルに保存するよう、命令することになります。ファイルへの保存には通常 `jhat` ユーティリティーを使用します (このユーティリティーについては以前の記事で紹介しました)。

ファイルの保存先の実際のパスの指定には、-XX:+HeapDumpOnOutOfMemoryError に対応する `-XX:HeapDumpPath` フラグを使います (どこにファイルが保存されるかによらず、そこにファイルシステムや Java プロセスが書き込むために必要なアクセス権が設定されているかどうかを確認する必要があります)。

3. bootclasspath

定期的な操作として、既存の JRE に用意されたクラスパスとは少し異なるクラスパスにクラスを配置したり、JRE を何らかの方法で拡張するクラスパスにクラスを配置したりすることは有効です。(例えば新しい Java Crypto API を提供する JRE など。) JRE を拡張しようとする場合には、カスタムの実装が、(`rt.jar` の中にある `java.lang.Object` とその関連ファイルをすべてロードする) ブートストラップ `ClassLoader` によってロードできる状態でなければなりません。

`rt.jar` を開き、その中にカスタム実装や新しいパッケージを入れることは可能ですが、そうすると厳密には、皆さんが JDK をダウンロードした時に同意した使用許諾条件に違反することになります。

このような場合には、JVM 自体の `-Xbootclasspath` オプションと、それと同類の `-Xbootclasspath/p`、`-Xbootclasspath/a` を使います。

`-Xbootclasspath` を使うと、完全なブート・クラスパスを設定することができます。ブート・クラスパスには通常、`rt.jar` への参照と、JDK には含まれているけれども `rt.jar` に含まれていない多数の JAR ファイルを配置しなければなりません。`-Xbootclasspath/p` によって既存のブート・クラスパスの前に値が追加され、`-Xbootclasspath/a` によって既存のブート・クラスパスの後ろに値が追加されます。

例えば、既存の `java.lang.Integer` を変更し、それを `mods` というサブディレクトリーに配置したとすると、`-Xbootclasspath/a mods` パラメーターによって新しい Integer はデフォルトの `Integer` の前に配置されます。

4. verbose

`-verbose` は簡単に使用できる便利な診断ユーティリティーであり、ほとんどあらゆるタイプの Java アプリケーションに使用することができます。このフラグには、`gc`、`class`、`jni` という 3 つのサブフラグがあります。

JVM のガーベッジ・コレクションが正常に動作してないのではないか、またパフォーマンス低下の原因になっているのではないか、といったことを判断するために、開発者は通常まず始めに `gc` を使用します。残念なことに、`gc` の出力の解釈には注意が必要であり、このトピックに関して何冊もの本が書かれているほどです。さらに悪いことに、コマンドラインに表示される出力は Java のリリースによって、あるいは JVM のバージョンによって異なり、正確な解釈が一層難しくなっています。

一般的に、ガーベッジ・コレクションが (エンタープライズ・クラスの VM のほとんどがそうであるように) 世代別コレクションである場合には、フル・スイープの GC パスであることを示す何らかの可視フラグが表示されます。Sun の JVM では、このフラグは `[Full GC ...]` として GC の出力行の先頭に表示されます。

`class` は、`ClassLoader` を診断したい場合や、クラスのミスマッチによる競合を診断したい場合に非常に役に立ちます。class によって、あるクラスがいつロードされたかだけでなく、そのクラスがどこからロードされたのかも、JAR ファイルへのパスを含めてレポートされます (ただし、そのクラスは JAR からロードされたものであることが前提です) 。

`jni` は、JNI とネイティブ・ライブラリーを扱う場合以外、ほとんど使い道がありません。jni をオンにすると、さまざまな JNI イベントが出力されます (例えば、いつネイティブ・ライブラリーがロードされ、いつメソッドがバインドされたか、など)。この場合も、出力は Java のリリースごと、あるいは JVM のバージョンごとに異なります。

5. コマンドラインの -X

ここまでは、JVM に用意されている中で、私のお気に入りのコマンドライン・オプションをいくつか紹介しました。しかし他にもオプションは多数あり、皆さん自身もそうしたオプションを見つけることができます。コマンドラインの引数 `-X` を実行すると、JVM に用意されている非標準オプション (ただしほとんどは安全なオプション) を一覧表示することができます。それらの一部を以下に挙げます。

- `-Xint` を使うと、JVM がインタープリター・モードで実行されます (これは、JIT コンパイラーが実際にコードに影響するかどうかをテストしたり、あるいは JIT コンパイラーにバグがあるかどうかを検証したりする場合に便利です)。
- `-Xloggc:` は `-verbose:gc` と同じことを行いますが、ログをコマンドライン・ウィンドウに表示する代わりに、ファイルに記録します。

JVM のコマンドライン・オプションは時々変更されるため、定期的に調べてみるとよいでしょう。そうすることで、夜遅くまでモニターを見つめながら過ごすのと、午後 5 時に帰宅して配偶者や子供達と楽しい食卓を囲むのとの違いほど大きな差が生ずるかもしれません (あるいは Mass Effect 2 (訳注: 2010年に発売された、宇宙を舞台にしたロール・プレイング・ゲーム) で敵を虐殺するのとの違いほどの差かもしれませんが、それは皆さんの好みによります)。

まとめ

コマンドライン・フラグは本番環境で永久に使うためのものではありません。実際、皆さんが JVM のガーベッジ・コレクションを調整するために使う羽目になる (かもしれない) フラグを別にすると、非標準のコマンドライン・フラグはどれも、実際に本番環境で使うためのものではありません。しかし、他の方法ではまったく調べる手段がない、仮想マシンの内部動作を確認するためのツールとして、コマンドライン・フラグはとても貴重です。

この連載、「今まで知らなかった 5 つの事項」の次回の記事では、日常的な Java ツールについて説明します。

著者について

Ted Neward

Ted Neward has written over 250 articles and a dozen books across many different technologies, including .NET, iOS, Java, Android, and JavaScript. He resides in Seattle with his wife, two kids, nine laptops, fourteen mobile devices, and two cats. Email him if you're interested in having him or his company work with you.