パフォーマンスの目: ガーベジ談義

ガーベジ・コレクターが何をしているか知っていますか？

Jack Shirazi
Director
JavaPerformanceTuning.com

Kirk Pepperdine
CTO
JavaPerformanceTuning.com

2004年 5月 21日

アプリケーションが頻繁にメモリ不足エラーを起こしていますか？ ユーザーから応答時間が一定しないと言われていますか？ アプリケーションがかなり長期間応答無しになってしまいますか？ アプリケーションのパフォーマンスが落ちているように見えますか？ こうした質問のどれかに対する答えがイエスであれば、おそらくガーベジ・コレクションに問題が起きているのです。JavaPerformanceTuning.com の Jack Shirazi と Kirk Pepperdine の説明に耳を澄ましてください。ガーベジ・コレクションの問題をどのように特定すべきか、また皆さんの疑問・・つまり一体ガーベジ・コレクターは何をしているのか、と言う疑問に対する答えを彼らが説明してくれます。

多くの開発者にとって、メモリ管理というのはビジネス・ロジックの開発という本来の課題から見れば、面倒以外の何者でもありません。ただしそれは、ビジネス・ロジックが思ったように性能を発揮しないとか、テスト中に性能不十分なことが分かるまでは、の話です。一旦この問題が起きると、何が悪いのか、それはなぜ起きているのかを知る必要があり、また下にあるコンピューター・リソースと、特にメモリと自分のアプリケーションがどうやり取りするかを知る必要が出てきます。アプリケーションがどのようにメモリを利用しているかを理解するには、ガーベジ・コレクターの動作を観察するのが一番です。

なぜアプリケーションがつまづくのか？

Java 仮想マシンでのパフォーマンスに関して一番頭の痛い問題は、GC をアプリケーション・スレッドと並列的に実行する、という相互排他性です。ガーベジ・コレクターが仕事をするには、ガーベジ・コレクターが一定期間操作するヒープ空間 (メモリ) に対して、どのスレッドもアクセスしないようにする必要があります。この期間は、GC の「stop-the-world」フェーズとして知られています。その名前からも分かるように、ガーベジ・コレクターが仕事に没頭している間は、あなたのアプリケーションはきしみ音を立てて停止に追い込まれるというわけです。幸いこの休

止期間は普通、気が付かないほど短いのですが、勝手なタイミングに勝手な場所で、勝手な持続時間でアプリケーションが中断すると、とんでもない応答時間低下やスループット低下を引き起こすことは容易に想像できるでしょう。

ところが、GC はアプリケーションがつまずいたり止まったりする原因の一つにすぎません。ではこうした問題の原因が GC にある、とどのように結論づけられるのでしょうか。その質問に答えるには、ガーベジ・コレクターがどの程度熱心に仕事をしているかを測定する必要があります。そしてシステムに変更を加える間もその測定を継続することによって、加えた変更が望んだような成果をもたらしたかどうかを定量的に結論づけられるようになるのです。

メモリはどのくらい必要なのか？

システムにメモリを追加すればパフォーマンスの問題は解決する、と一般的には信じられています。この原則は JVM では普通の場合正しいのですが、多くありすぎるとパフォーマンスの害にもなり得るのです。ですから Java アプリケーションにとって必要なだけのメモリを用意し、それ以上は用意しないのが秘訣なのです。問題は、そのアプリケーションがどれだけメモリを必要とするかです。アプリケーションがつまずき気味であれば、ガーベジ・コレクションの振る舞いをよく観察し、必要以上にガーベジ・コレクションが働いていないかを調べる必要があります。こうした観察をすれば、私たちが加えようとしている変更が望むような効果を生むかどうかが分かるのです。

GC の動きを測る

GC ログを生成する標準的な方法は `-verbose:gc` フラグを使うものです。一旦これを設定すると、ガーベジ・コレクターは実行の度に何を行ったかの要約を生成し、普通はこれをコンソール (標準出力としてまたは、標準エラーとして) に書き込みます。多くの VM では、冗長な GC 出力をファイルに出力できるようなオプションをサポートしています。例えば Sun の 1.4 JVM では、`-Xloggc:filename` というスイッチを使って GC 出力をファイルに書き込みます。HP の JVM では `-Xverbosegc=file` スイッチを使います。この記事では、Sun 1.4.2 と IBM 1.4.1 JVM からキャプチャーした冗長 GC を調べることにします。

メモリ使用をモニターするためにこの方法を使う利点として一番大きいのは、アプリケーションのパフォーマンスへの影響が非常に少ないことです。残念ながらこの手法はログファイルが非常に大きくなり、維持管理のために JVM の再起動が必要なることもある等の点で、完全ではありません。とは言ってもこの手法は、生産環境でのみ症状を示すパフォーマンス問題を診断するものとして、やはり有用なものです。

GC の奥深くを見る

`-verbose:gc` フラグの出力結果は JVM ベンダーによって変わり、またガーベジ・コレクターの様々なオプションによってその実装特有の情報をレポートします。例えば IBM JVM からの出力は Sun JVM からの出力よりもはるかに冗長ですが、Sun の出力の方がツールでの読み取りには適しています。とは言っても、どの GC ログで分かるのもほとんどは基本的な情報で、どれだけのメモリが消費されたか、どれだけ回復されたか、GC のサイクルにどの程度時間を要したか、その他ガーベジ・コレクション中に収集した他のアクションの情報などです。こうした基本的な測定の結果から、何が起きているかを詳しく理解できるような詳細情報を取り出すことができます。以下はこれから計算する統計値です。

- 対象とするランタイムの継続時間
- コレクションの合計回数
- コレクションの頻度
- コレクションに要した、最長時間
- コレクションに要した合計時間
- コレクションに要した平均時間
- コレクションの平均時間間隔
- 割り当てられた総バイト数
- 毎秒毎コレクション当たり割り当てられた平均バイト数
- 回復された合計バイト数
- 毎秒毎コレクション当たり回復された平均バイト数

休止時間を理解することで、アプリケーションが応答しなくなる原因として GC が部分的要因なのか全要因なのかが理解できるようになります。そのための一方法としては、冗長 GC ログにある GC の動作を、システムが収集した他のログ (Web サーバー・ログでのリクエスト・バックログなど) と関連付けてみることです。GC による休止が最長になれば、ほぼ間違いなく全体的なシステム応答が明確に低下するはずなので、いつ応答時間が低下するかを知っていることは重要です。つまりその低下と GC の動作をアプリケーションのスループットと関連付けられるのです。

競合の可能性のある点としてもう一つは、ヒープ・メモリの割り付け・回復の速度で、これはチャーン (churn) として知られています。ほとんど即座に解放されるようなオブジェクトを多数生成するアプリケーションはよく、チャーンの問題を起こします。高速のチャーンはガーベジ・コレクターに大きな負荷となり、より多くのメモリ資源競合を引き起こします。これが長時間の休止や、恐ろしい `OutOfMemoryError` につながるのです。

アプリケーションがこうした問題のどれかを抱えているかどうかを調べるには、対象にしている合計実行時間の中で GC が占める合計時間を測定してみることです。この計算をしてみると、GC が過剰労働をしていないかが分かってきます。ではこうした判断ができるように、公式を導き出してみましょう。

Sun の GC ログ・レコード

リスト 1 は Sun 1.4.2_03 JVM をデフォルトの mark-and-sweep コレクターを `-Xloggc:filename` で実行した時に生成されるログ・レコードの例です。ご覧の通り各エントリーは非常に簡潔なレコードで、各パスが何をしたかの記録になっています。

リスト 1. -Xloggc:filename フラグを使った GC ログ・レコード

```
69.713: [GC 11536K->11044K(12016K), 0.0032621 secs]
69.717: [Full GC 11044K->5143K(12016K), 0.1429698 secs]
69.865: [GC 5958K->5338K(11628K), 0.0021492 secs]
69.872: [GC 6169K->5418K(11628K), 0.0021718 secs]
69.878: [GC 6248K->5588K(11628K), 0.0029761 secs]
69.886: [GC 6404K->5657K(11628K), 0.0017877 secs]
```

最初に気が付くのは、どのログ・レコードも角カッコで囲まれて出力されていることでしょう。並列コレクター等の他の GC アルゴリズムでは、一部の値をもっと細かい情報ビットにまで分解します。そうすると、調査対象の値は幾重もの角カッコの中に収まった詳細情報で置き換えられ、ツールでの冗長 GC 出力の処理が容易になります。

リスト 1 にある、69.713 というタグが付いたレコードから見てみましょう。このタグは JVM が起動してからの時間を秒とミリ秒で表したタイムスタンプです。このレコードの場合には、JVM はこの GC サイクルの開始までに 69.713 秒実行していたわけです。フィールドは左から右の順に、実行されたコレクションのタイプ、GC 前のヒープ利用、GC 後のヒープ利用、合計ヒープ容量、そして GC イベントの持続時間です。この記述から、最初の GC イベントはマイナー・コレクションだと分かります。GC の実行前には 11536 Kb のヒープ空間が使用されていました。終了時には 11044 Kb が使用され、ヒープ容量は 12016 Kb、全コレクションに 0.0032621 秒かかっています。次のイベントはフル GC で、69.717 秒つまりその前のマイナーGC イベントの開始から 0.003 秒後に起きています。注意して欲しいのですが、マイナーGC イベントの開始時間にそのマイナーGC イベントの持続時間を加えてみると、このマイナーGC イベントはフル GC の開始前 1ms 以内に終了していることが分かります。このことから、マイナー・コレクションが十分な空間の回復に失敗し、この失敗がフル GC を引き起こしているのだ、と結論できます。アプリケーションにとってこれは、0.1462319 秒持続する単一のイベントに見えます。では他の値をどう計算すればよいか、その定義を見て行きましょう。

GC ログ・レコードのパラメーター化

各 GC ログ・レコードにある値をパラメーター化することから解析作業を始めます。

```
R(n) = T(n): [ <GC> HB->HE(HC), D]
```

n	リスト中のレコードのインデックス (1 が最初で m が最後)
R(n)	GC レコード
T(n)	n 番目の GC が起きた時間
HB	GC 前に使用されていたヒープ量
HE	GC 後に使用されているヒープ量
HC	ヒープ空間の総量
D	GC サイクルの持続時間

こうした定義を使うことで、先に説明した値を計算するための公式を導き出すことができます。

基本的な値

最初に計算するのは、ログから分かる全体的な持続時間です。どのレコードも考慮するのであれば、最後のレコードのタイムスタンプを見さえすれば良いことになります。リスト 1 は全ログ・レコードの一部にすぎないので、最後のタイムスタンプから最初のタイムスタンプを引く必要があります。この例に関してはこの数字で十分正確ですが、完璧を期すには最後の GC の持続時間を加える必要があります。こうする理由は、タイムスタンプは GC の開始時に取られますが、レコードはタイムスタンプが記録された後に起こったものを表すからです。

残りの値はレコード中にある適当な値の和を取ることで計算できます。面白い点として、回復されたバイト数はレコード中の測定値間の関係を調べることで計算しますが、割り付けられたバイト数は連続したレコード間の測定値の関係を調べることで計算する、ということに注意してください。例えば、タイムスタンプ値 69.872 と 69.878 のレコードの対を考えると、若い方に割り付けられたバイト数を計算するには、最初のレコードの GC 後に消費されているメモリ量から、２

番目のレコードの GC 前に消費されているバイト数を引きます。つまり `6248 Kb - 5418 Kb = 830 Kb` ということになります。他の値のための公式を下の表 1 に示します。

GC による休止で最も長いものを見つけるには、持続時間を調べ、`D(n)` (レコード n での持続時間) の最大値を探せば良いわけです。

表 1. 統計値公式

統計値	計算 (時間単位は秒に調整)
実行持続時間	`RT = (T(M) + D(M)) - T(1)`
マイナー・コレクションの総数	`TMC = Sum( R(n))` where `GC(n) =GC`
フル・コレクションの総数	`TFC = Sum( R(n))` where `GC(n) =Full`
コレクション頻度 (マイナー)	`CFM = TMC / RT`
コレクション頻度 (フル)	`CFF = TFC / RT`
コレクション時間 (マイナーで最長の時間)	`MAX(D(n))` for all`n` where `GC(n) =GC`
コレクション時間 (フルで最長の時間)	`MAX(D(n))` for all`n` where `GC(n) =Full`
マイナー・コレクション時間 (合計)	`TTMC = Sum(D(n))` for all`n` where `GC(n) =GC`
フル・コレクション時間 (合計)	`TTFC Sum(D(n))` for all`n` where `GC(n) =Full`
コレクション時間 (合計)	`TTC = TTMC + TTFC`
マイナー・コレクション時間 (平均)	`ATMC = TTMC / RT`
フル・コレクション時間 (平均)	`ATFC = TTFC / RT`
コレクション時間 (平均)	`ATC = TTC / RT`
コレクション間隔 (平均)	`Sum( T(n+1) - T(n)) / (TMC + TFC)` for all`n`
割り付けバイト数 (合計)	`TBA = Sum(HB(n+1) - HE(n))` for all`n`
割り付けバイト数 (毎秒)	`TBA / RT`
割り付けバイト数 (コレクション毎)	`TBA / (TMC + TFC)`
マイナー・コレクションでの回復バイト数 (合計)	`BRM = Sum(HB(n) - HE(n))` where `GC(n) =GC`
フル・コレクションでの回復バイト数 (合計)	`BRF = Sum(HB(n) - HE(n))` where `GC(n) =Full`
回復バイト数 (合計)	`BRT = BRM + BRF`
回復バイト数 (マイナー・コレクション毎)	`BRPM = BRM / TMC`
回復バイト数 (フル・コレクション毎)	`BRPF = BRF / TMF`
回復バイト数 (マイナー毎秒)	`BRP = BRM / TTMC`
回復バイト数 (フル毎秒)	`BRF = BRF / TTFC`

この公式から分かるように、多くの場合フル GC とマイナーGC は別に考える必要があります。マイナーGC は基本的にフル GC とは異なり、一般的に言って、フル GC よりも少なくとも一桁高速です。これはリスト 1 をちょっと見れば分かります。マイナーで一番長いものでもフルよりも 50 倍早いのです。

下に挙げた表 2 は、表 1 の公式をリスト 1 の値に適用したものです。

表 2. Sun の GC ログの調査

統計値	計算（時間単位は秒に調整）
実行持続時間	(69.886 + 0.0017877) - 69.713 = 0.1747877
マイナー・コレクションの総数	5
フル・コレクションの総数	1
コレクション頻度（マイナー）	5 / 0.1747877 = 28.6 per second
コレクション頻度（フル）	1 / 0.1747877 = 5.27 per second
コレクション時間（マイナーで最長の時間）	0.0032621
コレクション時間（フルで最長の時間）	0.1429698
マイナー・コレクション時間（合計）	0.0123469
フル・コレクション時間（合計）	0.1429698
コレクション時間（合計）	0.1553167
マイナー・コレクション時間（平均）	7.1%
フル・コレクション時間（平均）	81.8%
コレクション時間（平均）	88.9%
コレクション間隔（平均）	.173/5=0.0346
割り付けバイト数（合計）	3292
割り付けバイト数（毎秒）	18834 Kb/second
割り付けバイト数（コレクション毎）	549 Kb
マイナー・コレクションでの回復バイト数（合計）	3270 Kb
フル・コレクションでの回復バイト数（合計）	5901 Kb
回復バイト数（合計）	5901 + 3270 = 9171 Kb
回復バイト数（マイナー・コレクション毎）	3270/5 = 654
回復バイト数（フル・コレクション毎）	5901/1 = 5901
回復バイト数（マイナー毎秒）	3270/0.0123469 = 264843 Kb/second
回復バイト数（フル毎秒）	5901/0.1429698 = 41274K/second

表 2 には、一見単純なログから得られた、かなりの情報が含まれています。直面している問題によっては、ある情報が他の情報よりも重要と言うこともあり得るので、こうした値全てを計算する必要はないかも知れません。アプリケーションが長期間応答しなくなるという場合には、GC の持続時間と GC の回数に着目すべきでしょう。

IBM ログ・レコード

Sun のログとは異なり、IBM のログは非常に冗長です。とは言え提供されている情報を完全に理解するには、やはり案内が必要です。リスト 2 はそうした `verbose:gc` ログファイルの一つから抜粋したものです。

リスト 2. IBM JVM の verbose:gc 出力

```
<AF[31]: Allocation Failure. need 528 bytes, 969 ms since last AF>
```

```
<AF[31]: managing allocation failure, action=1 (0/97133320) (1082224/1824504)>
  <GC(31): GC cycle started Wed Feb 25 23:08:41 2004
  <GC(31): freed 36259000 bytes, 37% free (37341224/98957824), in 569 ms>
  <GC(31): mark: 532 ms, sweep: 37 ms, compact: 0 ms>
  <GC(31): refs: soft 0 (age >= 32), weak 0, final 2, phantom 0>
<AF[31]: managing allocation failure, action=3 (37341224/98957824)>
  <GC(31): need to expand mark bits for 116324864-byte heap>
  <GC(31): expanded mark bits by 270336 to 1818624 bytes>
  <GC(31): need to expand alloc bits for 116324864-byte heap>
  <GC(31): expanded alloc bits by 270336 to 1818624 bytes>
  <GC(31): need to expand FR bits for 116324864-byte heap>
  <GC(31): expanded FR bits by 544768 to 3637248 bytes>
  <GC(31): expanded heap by 17367040 to 116324864 bytes, 47% free, ratio:0.417>
<AF[31]: completed in 812 ms>
<AF[32]: Allocation Failure. need 528 bytes, 1765 ms since last AF>
<AF[32]: managing allocation failure, action=1 (0/115394264) (930600/930600)>
  <GC(32): GC cycle started Wed Feb 25 23:08:43 2004
  <GC(32): freed 54489184 bytes, 47% free (55419784/116324864), in 326 ms>
  <GC(32): mark: 292 ms, sweep: 34 ms, compact: 0 ms>
  <GC(32): refs: soft 0 (age >= 32), weak 0, final 0, phantom 0>
<AF[32]: completed in 328 ms>
<AF[33]: Allocation Failure. need 528 bytes, 1686 ms since last AF>
<AF[33]: managing allocation failure, action=1 (0/115510592) (814272/814272)>
  <GC(33): GC cycle started Wed Feb 25 23:08:45 2004
  <GC(33): freed 56382392 bytes, 49% free (57196664/116324864), in 323 ms>
  <GC(33): mark: 285 ms, sweep: 38 ms, compact: 0 ms>
  <GC(33): refs: soft 0 (age >= 32), weak 0, final 18, phantom 0>
<AF[33]: completed in 324 ms>
```

リスト 2 には 3 つの GC ログ・レコードがあります。完全に説明して欲しいという要望はお断りする代わりに、Sam Borman 著による素晴らしい記事「Sensible Sanitation」（ 参考文献 ）を紹介しておきます。ここでの目的のためには Sun JVM でのログから得た情報と同じ種類の情報を引き出す必要があります。ありがたいことに、こうした計算のいくつかが既に準備されているのです。例えば`AF[31]`（イベント 31 に対する割り当て失敗）を見れば、GC の間隔、回復されたメモリ量、イベントの持続時間が分かります。こうした数字から、必要な他の値も計算できます。

数字が何を意味するか

目的とするところによって、こうした数字をどう見るかも大きく変わってきます。多くのサーバー・アプリケーションでは、目的は休止時間の減少に行き着きます。つまりフル・コレクションの持続時間と回数を減少させるということです。次回はこの情報をどのように利用し、正にこの問題で苦しんでいる実際のアプリケーションをどう調整したかについて説明します。

著者について

Jack Shirazi

Jack ShiraziはJavaPerformanceTuning.comのディレクターであり、Java Performance Tuning, 2nd Edition（O'Reilly刊）の著者でもあります。

Kirk Pepperdine

Kirk PepperdineはJava Performance Tuning.comのCTO（Chief Technical Officer）であり、過去15年間、オブジェクト技術やパフォーマンス調整に注力してきました。Ant Developer's Handbook（MacMillan刊）の共著者でもあります。