Java の診断を IBM スタイルで: 第 2 回 IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer を使用したガーベッジ・コレクション

アプリケーション・パフォーマンスの向上、ガーベッジ・コレクションの最適化、そしてアプリケーションの問題解明を図る

Dr. Holly Cummins
Software Engineer
IBM

2013年 8月 22日
(初版 2007年 10月 09日)

IBM の新しいツールである IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer は、メモリーに関連する Java のパフォーマンス問題の診断と分析を支援することを目的に設計されています。全 4 回からなる連載の第 2 回となるこの記事では、このツールの入手方法と使用方法を説明し、このツールを使って一般的な問題を素早く診断する手順を実例で紹介します。
2013年 8月 22日 — 記事で取り上げるツールの名称を、現在の名称である「IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer」に変更するとともに、「参考文献」に新しい項目「Java Technology Community」を追加しました。

このシリーズの他の記事を見る

この連載について

連載「Java の診断を IBM スタイルで」では、Java アプリケーションの問題解決を支援し、アプリケーションのパフォーマンスを改善する新たな IBM のツールを取り上げます。毎回掲載される記事から、すぐに実行に移せる新しい知識を得られるはずです。

この連載の寄稿者はいずれも、Java アプリケーションでの問題解決支援ツールを作成するために新しく組織されたチームのメンバーです。さまざまなバックグラウンドを持つ著者たちが、さまざまなスキルとその専門の分野でチームに貢献しています。

記事に関するご意見・ご質問は、それぞれの記事を担当する著者にお寄せください。

アプリケーションでのガーベッジ・コレクション (GC) を詳細に調べたいと思う理由はいくつかあります。例えば、アプリケーションがメモリーを使い過ぎているのではないか、メモリー・リークが発生しているのではないか、あるいは長期間維持できるメモリー使用量であるかどうかな

ど、アプリケーションのメモリー使用パターンが懸念される場合です。また、アプリケーションの動作速度を上げられるかどうかを調べたいという場合もあります。ガーベッジ・コレクションはアプリケーションのパフォーマンスに大きな影響力を持ちます。GC が適切に構成されていないとリソースの消費量が多くなり、アプリケーションの処理速度が遅くなることは広く知れ渡っていますが、その逆も然りです。つまり、ガーベッジ・コレクションのパラメーターを賢く選択することで、アプリケーションを高速化することができます。

短時間の Java アプリケーションや、パフォーマンスがそれほど重要ではないアプリケーションでは GC を無視しても構いませんが、それ以外のアプリケーションではツールの使用によって、至って簡単に verbose GC のログから必要な情報を取得することができます。ツールでヒープ内の状況をグラフにすれば簡単にパターンを見分けられるようになるだけでなく、ツールに特定のパターンを検出させて調整のための推奨値を出させることも可能です。

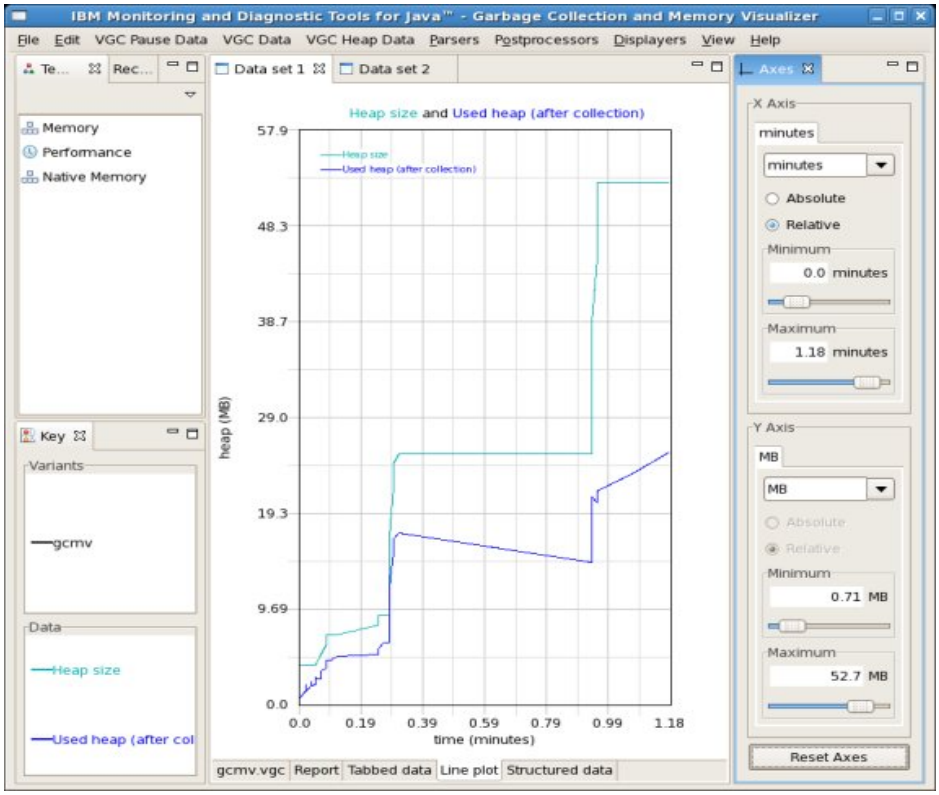
このトピックに関するスキルを磨いてください

このコンテンツは、皆さんのスキルを漸進的に磨いていくための Knowledge Path の一部です。次のリンクをご参照ください。 Monitor and diagnose Java applications

IBM の新しいツール・スイートに含まれる GC and Memory Visualizer は、verbose GC のログを分析し、メモリー管理の問題に関する詳細を理解できるようにするためのツールです。この記事では GC and Memory Visualizer の機能について説明するとともに、GC and Memory Visualizer がメモリー問題の診断に役立つ事例をいくつか紹介します。

GC and Memory Visualizer で処理可能なのは、Version 1.4.2 以降のすべての IBM JRE からのログです。また、IBM WebSphere Real Time からのログもグラフにすることができます。このツールでは、複数のログの同時比較、ログの特定エリアのズームイン、データのフィルタリング、そしてさまざまな単位での表示が可能です。図 1 に、GC and Memory Visualizer の表示例を記載します。

図 1. GC and Memory Visualizer の表示例



verbose GC のロギングを有効にする

分析用のログを生成するには、アプリケーションの verbose GC のロギングを有効にする必要があります。そのためには `-verbose:GC` 仮想マシン (VM) フラグを指定するか、または Version 5.0 以降であれば IBM VM に対して `-XverboseGClog:file` を指定してコマンドを実行します。コマンドの `file` には選択したログ・ファイルの名前を指定してください。`-XverboseGClog` オプションを使用できる場合には、このオプションを使用することをお勧めします。通常、verbose GC がアプリケーションのパフォーマンスに与える影響は比較的限られています。

GC and Memory Visualizer のダウンロードおよびインストール手順

IBM Support Assistant について

ISA (IBM Support Assistant) は、IBM ソフトウェア製品での問題の解決を支援する無料のソフトウェア保守サービス・ワークベンチです。ISA には、大量の IBM ドキュメンテーションを網羅し、結果をカテゴリー別に分類して検討できるようにする検索機能が備わっています。

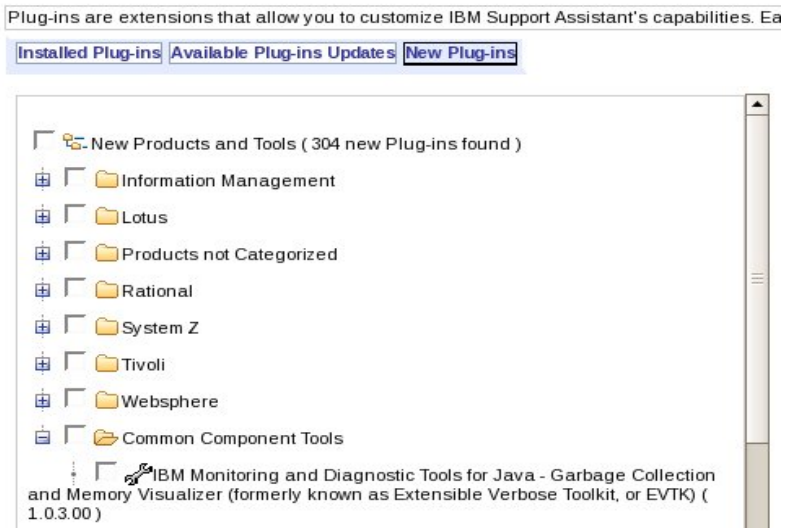
ISA には、製品サポート・ページやホーム・ページ、トラブルシューティング・ガイド、そしてフォーラムやニュースグループなどにリンクした製品情報機能もあります。さらに、ISA のサービス機能を使えばデスクトップからの情報を収集し、IBM への問題レポートを簡単に作成することができます。

ISA のツール・ワークベンチには、IBM 製品での問題の解決に役立つ問題判別ツールが用意されています。これらのツールは定期的に更新され、デスクトップでトラブルシューティン

グ・ツールおよび診断ツールを実行できるようにします。ISA のダウンロード・リンクについては、「参考文献」を参照してください。

GC and Memory Visualizer は IBM Support Assistant に含まれており、無料でダウンロードできるツールです。IBM Support Assistant をまだインストールしていない場合は、これをダウンロードするところから始めます (「参考文献」にリンクを記載)。IBM Support Assistant のインストールが完了したら、IBM Support Assistant に JVM が組み込まれた製品を使用していることを認識させます。そのためには、製品プラグインをインストールします (図 2 を参照)。製品プラグインは、IBM Support Assistant の Updater ページからダウンロードすることができます。例えば Others セクションの Developer kit for Java を選択することも、WebSphere セクションの WebSphere 製品を選択することもできます。

図 2. 製品プラグインのインストール

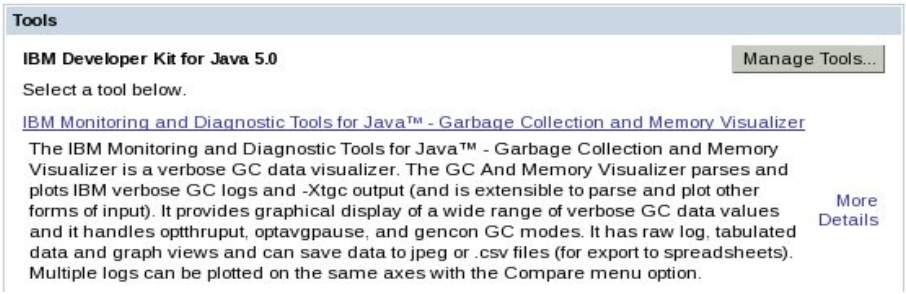


同時に GC and Memory Visualizer プラグインをインストールすることができます。GC and Memory Visualizer は、New Plug-ins タブの Common Component Tools セクションに見つかります (図 3 を参照)。

図 3. GC and Memory Visualizer のインストール

製品プラグインと GC and Memory Visualizer をインストールしたら、IBM Support Assistant を再起動する必要があります。これで、Tools ページで GC and Memory Visualizer を起動できるようになります (図 4 を参照)。

図 4. GC and Memory Visualizer の起動



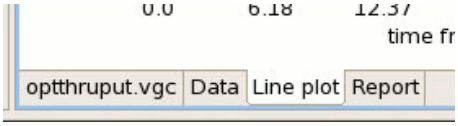
共通タスク

早速、GC and Memory Visualizer を使った基本的なログ分析タスクをいくつか実行してみましょう。

分析対象のログの表示

GC and Memory Visualizer で verbose GC のログを分析するには、まず GC and Memory Visualizer を起動して File メニューから Open File を選択します。すると、タブが 4 つある GC and Memory Visualizer エディターにログが開きます (図 5 を参照)。実行中のアプリケーションのログを開くこともできますが、GC and Memory Visualizer は表示を自動的に更新しません。表示を更新するには Reset Axes ボタンをクリックします。

図 5. GC and Memory Visualizer エディターのタブ



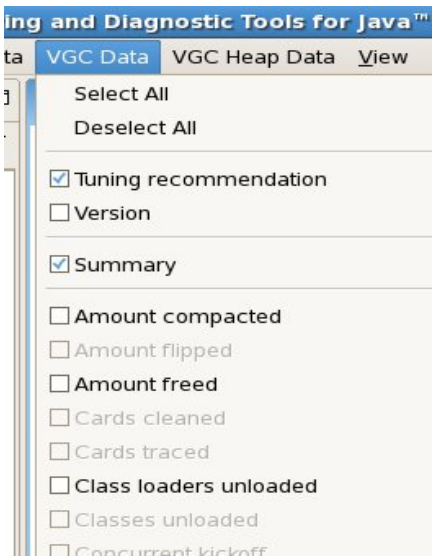
タブの内容は以下のとおりです。

- ファイル名のラベルが付いたタブには、ログ自体のテキストが表示されます。ログが大きい場合、GC and Memory Visualizer にはすべてのテキストが表示されるわけではありませんが、ログ全体が解析されたことに変わりはありません。
- Data タブには GC and Memory Visualizer が生成したデータがそのまま表示されます。このデータはスプレッドシートにカット・アンド・ペーストするのに適しています。
- Line plot タブにはグラフにされたデータが表示されます。
- Report タブには、データに関する GC and Memory Visualizer のレポートが表示されます。レポートの内容は、選択された各フィールドのサマリー、表形式でのログ全体のサマリー、そして一連の調整推奨値です。

VGC Data メニュー (図 6 を参照) には、表示可能なすべてのフィールドが示されます。薄いグレーで表示されたフィールドは、GC and Memory Visualizer が検索したけれども現行ログには見つからなかったフィールドに該当します。Summary フィールドが選択されていない場合、このフィール

ドを選択すると表形式のサマリーを有効にすることができます。同様に、Tuning recommendation を選択すると推奨値が表示されます。

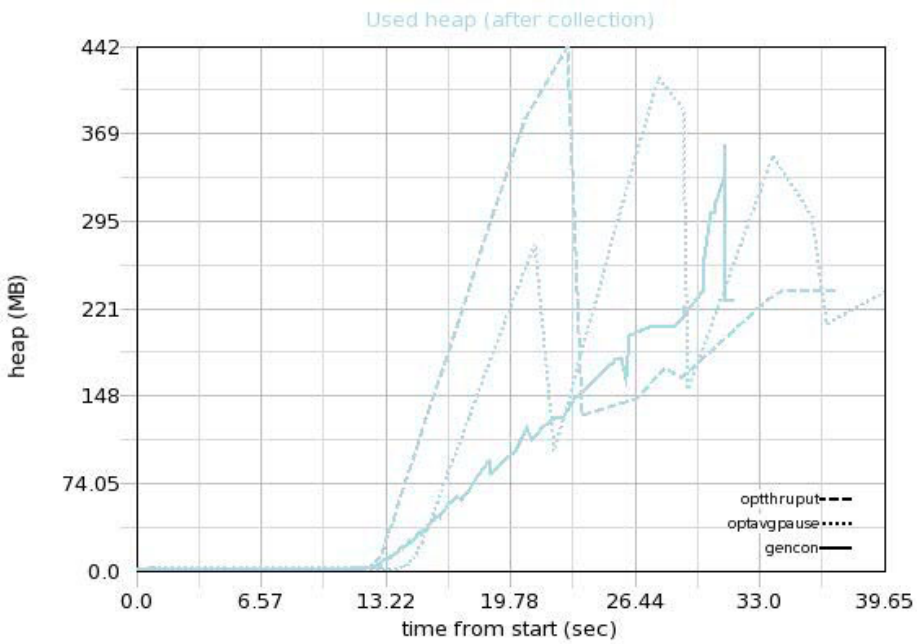
図 6. VGC Data メニュー



複数のファイルの比較

GC and Memory Visualizer では複数のファイルを並べて分析できるので、パフォーマンスの変化による影響を評価する際に重宝です。図 7 に、一定のワークロードを 3 種類の GC ポリシーで実行した場合のアプリケーションを示します (アプリケーション例として GC and Memory Visualizer 自体を使っています)。実線は gencon GC ポリシー、点線は optavgpause ポリシー、そして破線は optthruput ポリシーです。GC and Memory Visualizer が各線に付けるラベルはログ・ファイル名に基づいています (各種 GC ポリシーについての詳細は、参考文献に記載した「Java technology, IBM style: Garbage collection policies, Part 1」を参照してください)。

図 7. 3 種類のガーベッジ・コレクション・ポリシーそれぞれのヒープ使用量と一時停止時間

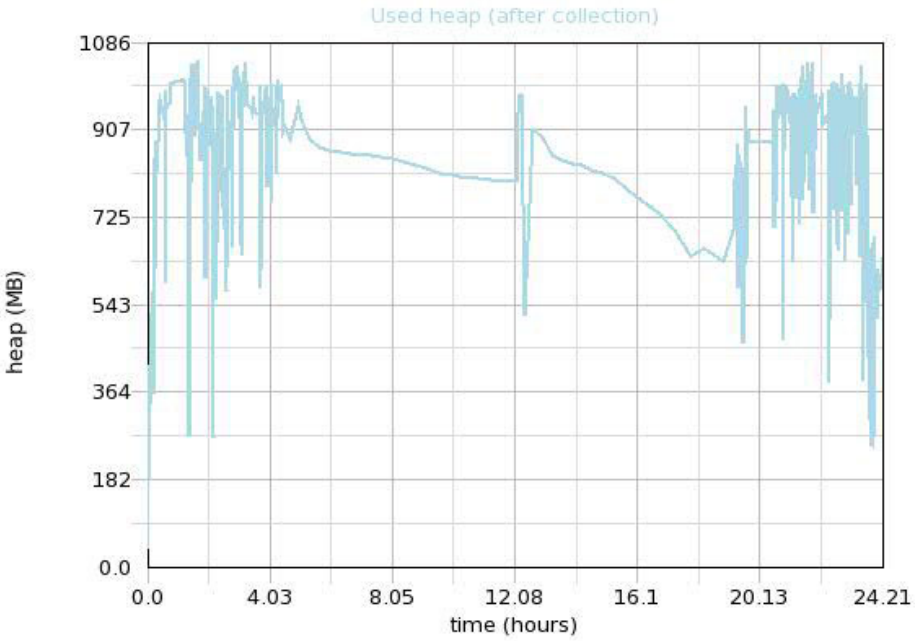


上記の例では、gencon モードがすべての基準で最も優れているのは明らかです。タスクを最も短時間で完了しているこのモードは、プロセスで使用するヒープも少なく、GC 一時停止時間は他と比べてはるかに短時間です。しかし gencon ポリシーはデフォルトではありません。デフォルトは、たいていの状況で gencon よりパフォーマンスが良いことから optthruput となっています。そうは言っても、この例が示すように optthruput のパフォーマンスが常に gencon ポリシーより勝るとは限らないため、異なる GC ポリシーを使うとアプリケーションの振る舞いがどのように変化するのかを調べる価値はあります。アプリケーションで使用する GC ポリシーを変更するなどといった極めて単純な変更が大きな改善をもたらすことはよくあります。

問題のある期間へのズームイン

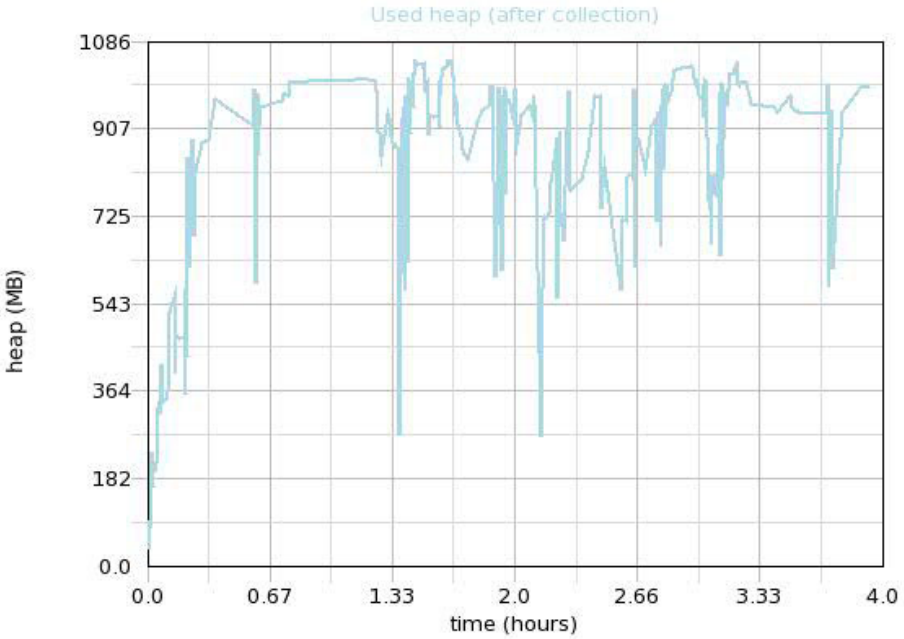
GC and Memory Visualizer ではログ内の特定の期間に焦点を絞ることができます。特定の期間にズームインすると、すべてのサマリー・データと推奨値がその期間のみを反映したものに変更されます。例えば、図 8 のログは、日中はビジーで夜間はアイドルになるアプリケーションのヒープ使用量を示しています。

図 8. 日中はビジーで夜間はアイドルになるアプリケーションのヒープ使用量



このログ全体としては GC オーバーヘッド (つまり、GC の所要時間) が約 5 パーセントなので、かなり良い成績です。ただしこの計算には、アプリケーションが何の処理も行っていないために GC の必要がない期間もだいぶ含まれています。特定の期間にズームインすることで、ビジー期間中のシステムの振る舞いをより正確にグラフに反映することができます (図 9 を参照)。

図 9. ビジー期間へのズームイン



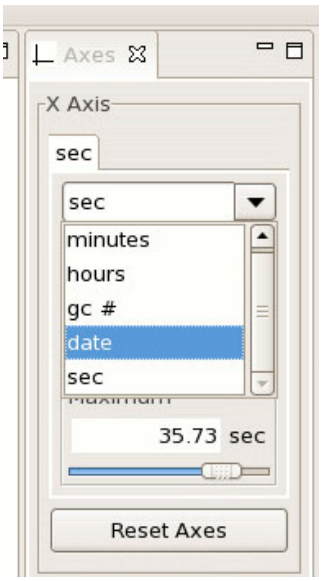
GC and Memory Visualizer では特定のデータ範囲に焦点を絞ることも可能です。例えば、非常に長い一時停止時間や 500MB を上回るヒープ使用量の期間にのみ関心があるとします。この場合、Y 軸の値を変更するという手段で、この種のフィルタリングを行うことができます。

単位の変更

GC and Memory Visualizer では表示単位を変更することができます。単位を変更すると、グラフの状態が変わるだけでなく、サマリー表と調整推奨値の単位も同じく変更されます。単位を変更するには、グラフの単位を右クリックするか、(View メニューから) Advanced Perspective を開きます。

デフォルトでは、時間 (X 軸の単位) は秒単位で表示されます。秒単位での表示は実行時間が短い場合には適していますが、長期間にわたるログには理想的とは言えません。別の単位に変更するには、右側のドロップダウン・メニューから目的の単位を選択してください (図 10 を参照)。選択可能な単位には、時間、分、日付、GC 番号 (単なるコレクションのシーケンス番号) があります。Normalize チェック・ボックスによって、ログの開始を基準とした時間 (正規化) を表示するか、または絶対時間 (非正規化) で表示するかが決まります。

図 10. 単位の変更

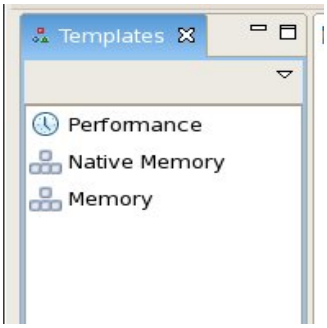


Y 軸の単位も同じく変更することができます。例えば、ヒープはデフォルトではメガバイトで表示されますが、これをギガバイトに変更することや、合計ヒープに占めるパーセンテージとして表示することもできます。

テンプレートの使用およびエクスポート

よくあることですが、いつの間にか同じフィールドの組み合わせを繰り返し表示していることに気付く場合があります。GC and Memory Visualizer ではテンプレートを使って、フィールドの組み合わせを後で使用できるように保存することができます。Templates ビューは、ウィンドウの左上端にあります (図 11 を参照)。

図 11. Templates ビュー



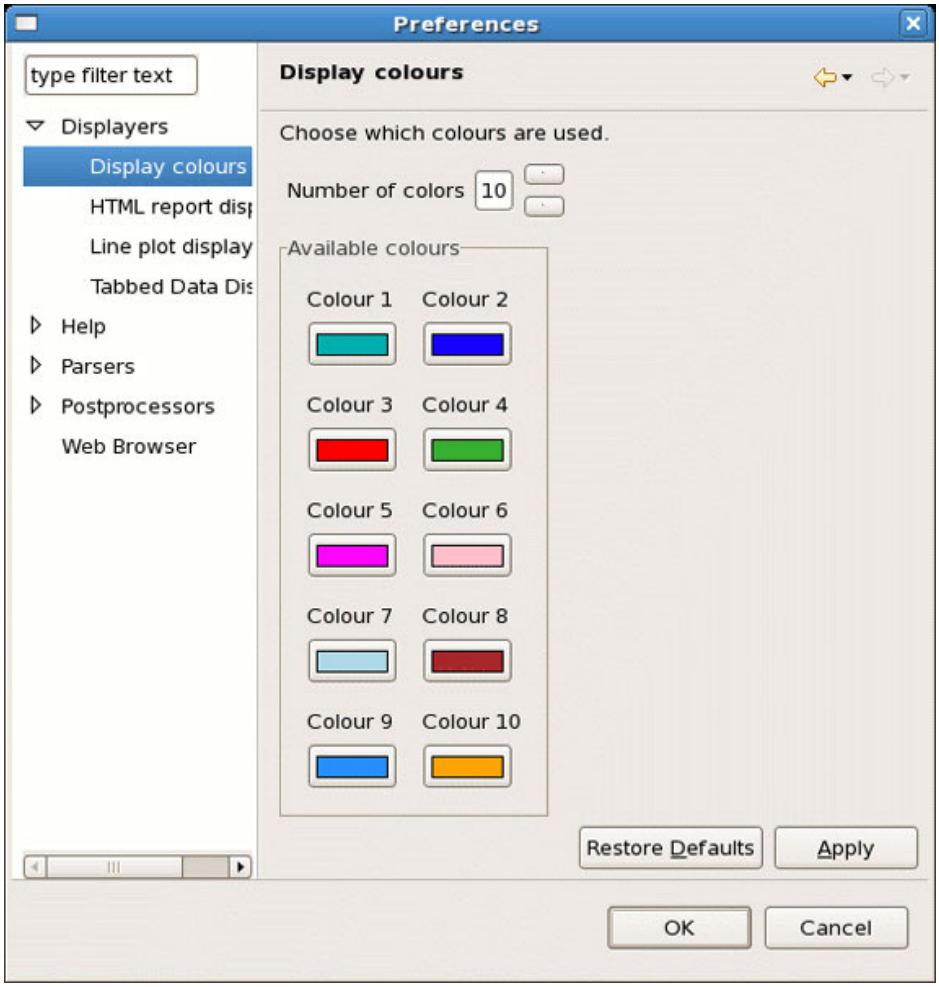
現行のデータ・セットにテンプレートを適用するには、該当するテンプレートをダブルクリックします。GC and Memory Visualizer には事前定義されたテンプレートがいくつか用意されています。そのうち、Heap テンプレートはアプリケーションのメモリー使用量および要件を評価する場合に有効です。Pauses テンプレートは、GC に関連すると思われるパフォーマンス上の問題を診断する上での第一歩となります。

テンプレートをエクスポートするには、View メニューを表示するか、または Templates ビュー内を右クリックして Export current settings as template を選択します。テンプレート名を入力すると、GC and Memory Visualizer はテンプレートを今後使用できるように Templates ビューに保存します。

色の変更

GC and Memory Visualizer がグラフに使用する色は変更することができます。View メニューの Preferences 項目をクリックし、Displayers カテゴリーに含まれる Display colours ページに進んでください (図 12 を参照)。

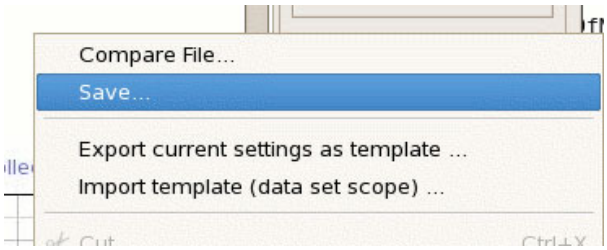
図 12. 色の設定ページ



出力の保存

GC and Memory Visualizer の出力はいずれも、メイン・パネル内を右クリックし、表示されるコンテキスト・メニューから Save を選択することで保存することができます (図 13 を参照)。折れ線グラフは JPEG 画像として、レポートは HTML として、未処理データは CSV ファイルとして保存することができます。この記事に記載しているグラフは、Line plot タブから保存したものです。

図 13. 保存



推奨値の使用

GC and Memory Visualizer は、verbose GC のログに関する興味深い特徴のサマリーと併せて、その特徴に対する調整推奨値を提供します。このサマリーと推奨値は、Report タブのレポートに表示されます。

なぜ、調整に介入して手作業で調整を行う必要に迫られるかと言えば、ガーベッジ・コレクターは元々、かなりの自律的な調整動作によってそれ自体のパフォーマンスを最適化しようとしますが、ユーザーの優先順位、そしてユーザーがガイダンスなしで進んで行うトレードオフは識別できないからです。あらゆるワークロードとあらゆる状況に共通して最適な唯一の構成というものはありません。ユーザーにできる最も簡単な調整は、ポリシーを指定し、ガーベッジ・コレクターにスループットと一時停止時間のどちらが重要であるかを伝えることです。もう少し冒険したいという方、あるいはどうしても最適なパフォーマンスを実現したいというのであれば、ヒープ・サイズを固定してみたり、新しい世代領域のヒープ・サイズを変更、あるいはさらに増やして最大にしてみることもできます。

事例研究: メモリー・リークを診断する

verbose GC のログを調べる主な理由の 1 つは、アプリケーションのメモリー使用量を検討し、どのような形であれ問題となっていないことを確実にするためです。例えばアプリケーションが予期される以上のメモリーを使用している可能性がある場合は、verbose GC の出力からアプリケーションのフットプリントがおおよそわかります。メモリー・リークは直接の原因にはならないとは言え、はるかに重大な問題です。Java プラットフォームの GC 機能により、Java アプリケーションではオブジェクトへのすべての参照がなくなるまでそのオブジェクトを解放しないとしても、メモリー・リークは発生しないようになっています。しかし、アプリケーションが誤ってオブジェクト参照を保持してしまうと、ガーベッジ・コレクターがまだ参照されているオブジェクトを収集できないため、アプリケーションはメモリー・リークに対して脆弱になってしまいます。

ソフト参照と弱参照

ソフト参照とは、ヒープ内の空きスペースが小さい場合にクリアできる参照のことです。ソフト参照はメモリーの状況に依存したキャッシングを行う上で役に立ちます。一方、弱参照とは他にその参照先オブジェクトに対する参照がない場合にクリアされる参照のことで、メタデータをオブジェクトに関連付けるマップにも、リスナーのリストを管理する際にも非常に有益です。当然のことながら、マップ内にあるすべてのものを永遠に維持するつもりなら、弱参照はふさわしくありません。

通常、メモリー・リークはかなり簡単に診断することができます。まず、アプリケーションで verbose GC を有効にしてある程度の期間実行し、それから使用されたヒープ (コレクション後) の量を GC and Memory Visualizer でグラフ化してください。アプリケーションの初期化が行われていたり、アプリケーションのワークロードが増大したりすれば、当然、アプリケーションのメモリー使用量も増加します。メモリー・リークの可能性が考えられるのは、アプリケーションのメモリー所要量が増加する明らかな理由もないのに、使用されたヒープの量を示す線が上昇している場合です。GC and Memory Visualizer はこのパターンを探し、リークの可能性を検出した場合には調整推奨値にコメントを追加します。

verbose GC はリークが進行中であることを示すことはできますが、メモリー・リークの原因となっているオブジェクトを特定することはできません。該当するオブジェクトは、場合によってはコードを検査するだけで突き止められます。ハッシュ・マップとその他のコレクションをよく調べて、変化のないものがあるかどうか、そのすべてにオブジェクトを追加するメカニズムと同時にオブジェクトを削除するメカニズムがあるかどうかを調べてください。また、キャッシュ対象という点でアプリケーションが寛容になり過ぎていないかも調べます。オブジェクト・プーリングもメモリー・リークの原因となることがあります。

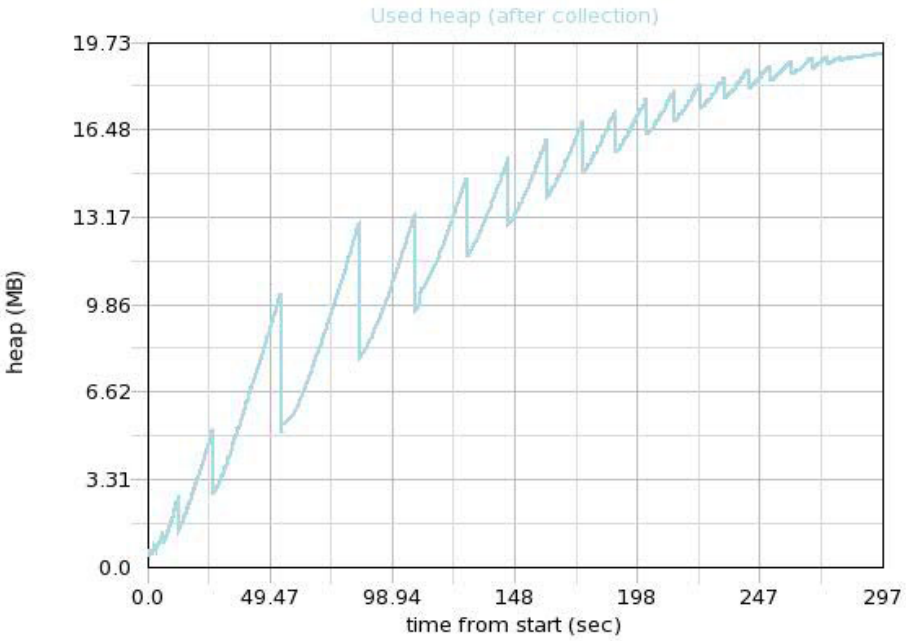
以下の例が示すように、弱参照とソフト参照はメモリー・リークを修正する強力なツールです (弱参照とソフト参照についての詳細は囲み記事を、いつ、どのように使うかについての説明は「参考文献」のリンクを参照してください)。リスト 1 のアプリケーションには明らかにメモリー・リークがあり、マップに追加することはあってもマップから削除することは決してありません。

リスト 1. メモリー・リークが激しい Java クラス

```
public class Leaker
{
 private Map things = new HashMap();

 public void leak() {
  while (true) {
   things.put(new Date(), new Leak());
  }
 }

 private class Leak
 {
  private Object data;

  public Leak() {
   data = new Object();
  }
 }
}
```

図 14 は、このアプリケーションのヒープ使用量です。ヒープ使用量の下落は、ヒープが圧縮された時点を示します。このログは、JVM がメモリー不足になった時点で終了します。

図 14. 大幅なメモリー・リークを伴うアプリケーションでのヒープ使用量



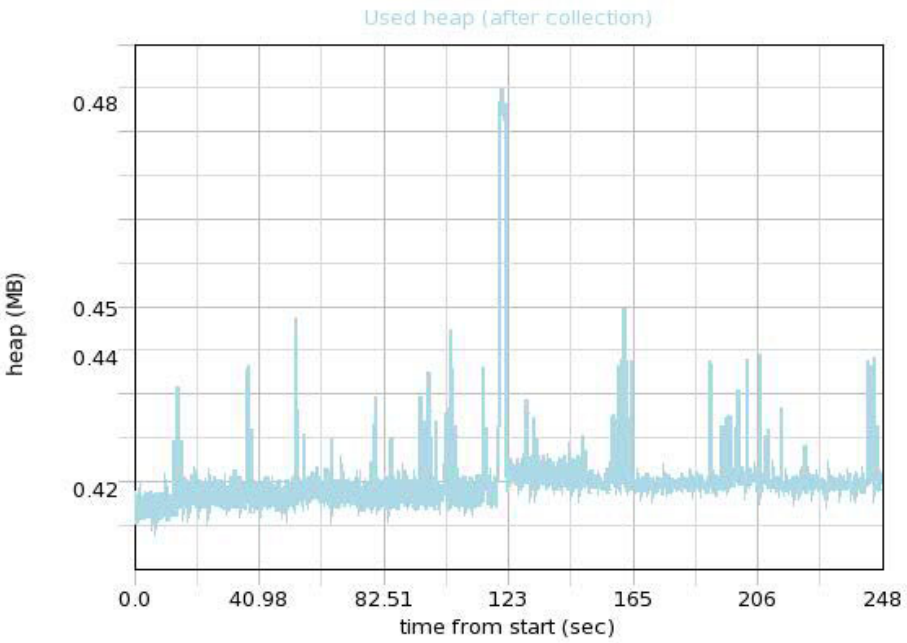
弱参照を使用してリークを防ぐ

この問題は、`WeakHashMap` (リスト 2 を参照) に切り替えることで即座に解決できます。この改善されたヒープ使用量を図 15 に示します。この場合、ヒープ使用量が 1MB を上回ることはないため、アプリケーションはいつまでも動作し続けます。

リスト 2. Leaker クラスの単純な修正によるメモリー・リークの防止

```
private Map things = new WeakHashMap();
```

図 15. WeakHashMap によって改善された潜在的なメモリー・リークを伴うアプリケーション



その一方で、弱参照では修正しきれないメモリー・リークもあります。例えば、図 15 のマップがリスト 3 に示すようなリンク・リストだったとしたらどうなるでしょう。

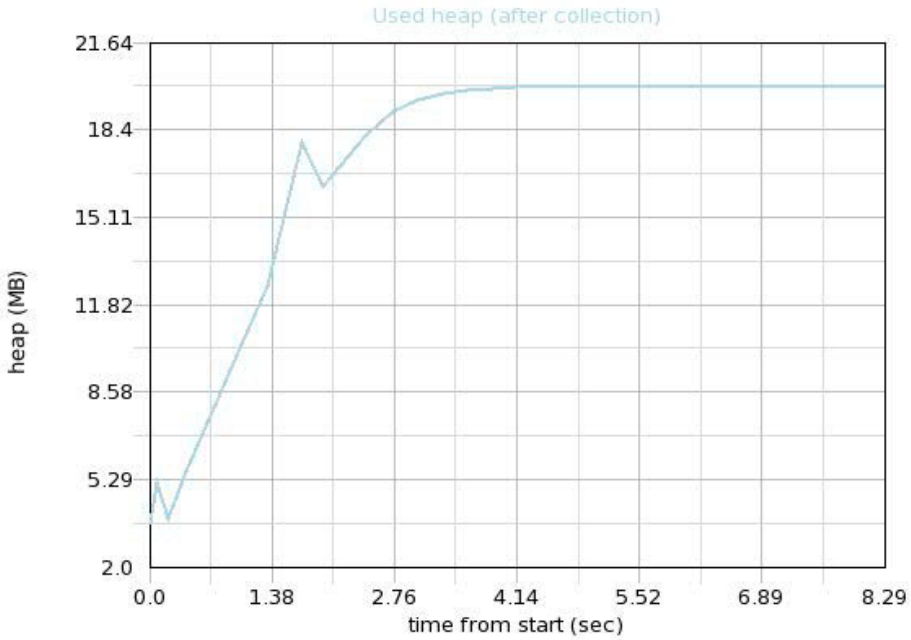
リスト 3. メモリー・リークを再びもたらす Leaker クラスの変更

```
public class Leaker
{
 private Map things = new WeakHashMap();

 public void leak() {
  Object previousThing = null;
  while (true) {
   final Leak thing = new Leak(previousThing);
   things.put(new Date(), thing);
   previousThing = thing;
  }
 }

 private class Leak
 {
  private Object data;

  public Leak(Object thing) {
```

```
    /* Make a linked list */
    data = thing;
   }
  }
}
```

弱参照は、オブジェクトに弱参照以外の参照がない場合にオブジェクトを収集するようガーベッジ・コレクターに指示します。しかしマップに含まれるそれぞれのオブジェクトは前のオブジェクトを参照するため、弱参照がクリアされずにアプリケーションはたちまちメモリー不足となってしまいます (図 16 を参照)。

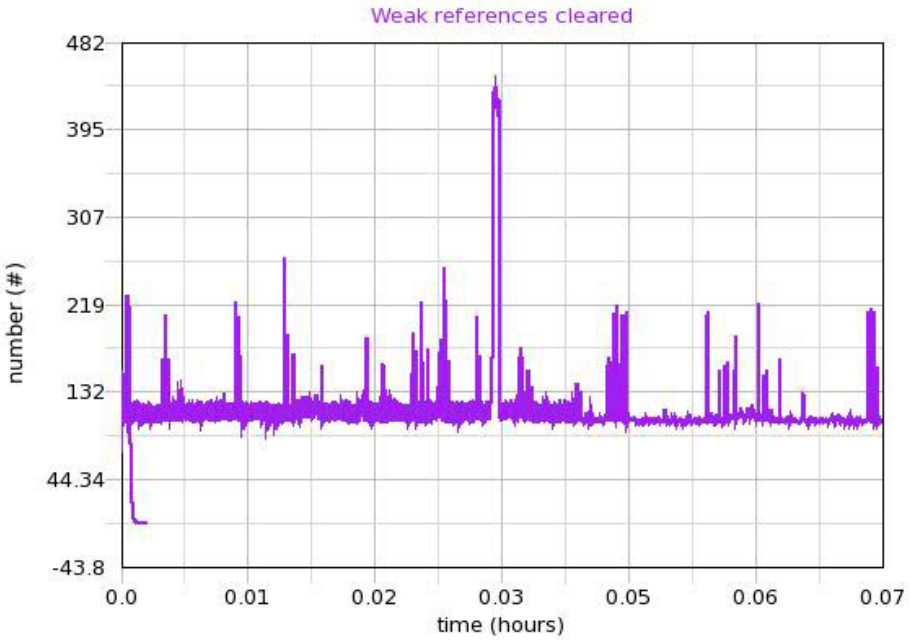
図 16. WeakHashMap では改善されないメモリー・リークを伴うアプリケーションでのヒープ使用量



弱参照が期待どおりに機能していることを確実にする

この問題は、クリアされた弱参照を GC and Memory Visualizer でグラフ化すると確認することができます (図 17 を参照)。リストにリンクが追加される前は数多くの弱参照がクリアされていましたが、リンクの追加後には、その数はゼロになっています (アプリケーションの新しいバージョンは X 軸上のゼロ点あたりにある極めて短い線で、前のバージョンはそれよりも長く高い線です)。このグラフを見ると、弱参照が機能していないことは明らかです。

図 17. 潜在的なメモリー・リークがあるアプリケーションの 2 つのバージョンでの弱参照のクリア数



この場合の解決策は、リンク・リスト内のリンクも弱参照に変更することです。リスト 4 に示すコードの変更を実装すると、弱参照の数は大幅に増え、ヒープ使用量は再び最小限になります。

リスト 4. 参照が必要以上長く保持されないようにするための弱参照の追加導入

```
private class Leak
{
 private WeakReference reference;

 public Leak(Object thing) {
  this.reference = new WeakReference(thing);
  /*
          * We can get back our object from the reference with
          * reference.get(), but we should always check it for null.
          */
 }
}
```

クリアされる弱参照の数を GC and Memory Visualizer で調べることで、弱参照を用いた再設計が実際に有効であることを簡単に確認することができます。

コードの検査で簡単にメモリー・リークを発見できない場合は、アプリケーションのダンプを取得して分析し、参照のサイズが増加しているオブジェクトを見つけ出す必要があります。メモリー・リークを検出して修正する方法についての記事は、「参考文献」を参照してください。

verbose GC のログは、アプリケーションの拡張性を評価する上でも役立ちます。例えば、大量のデータを処理するように意図されているアプリケーションであるにもかかわらず、テストでわずかな量のデータを処理する際に大量のメモリーを使用するのであれば、そのアプリケーションはおそらく期待するようには拡張できないはずです。

事例研究: ヒープのサイズを変更する

開発者の多くは、verbose GC のデータを使用してヒープに最適なサイズを選択します。ヒープが小さすぎてアプリケーションが必要とするデータが収まりきらないと、アプリケーションはメモリーを使い果たし、`OutOfMemoryError` により強制終了する結果となります。ヒープにアプリケーション・データ用のスペースはあっても予備のスペースが十分になければ、ガーベッジ・コレクターは新しい割り当てのためのスペースをヒープに確保するために多くの時間を費やすことになり、それによってアプリケーションのパフォーマンスは損なわれることになります。ヒープ・サイズが大きすぎるためにアプリケーションのパフォーマンスが影響されることは通常ありませんが、過剰なヒープ・サイズは無駄が多く、GC 一時停止時間を長びかせるおそれがあります。一般に、マシンで動作するアプリケーションは 1 つだけではないので、単一の Java アプリケーションに必要以上のメモリーを与えないようにメモリーを再配布するのが妥当な場合もあります。

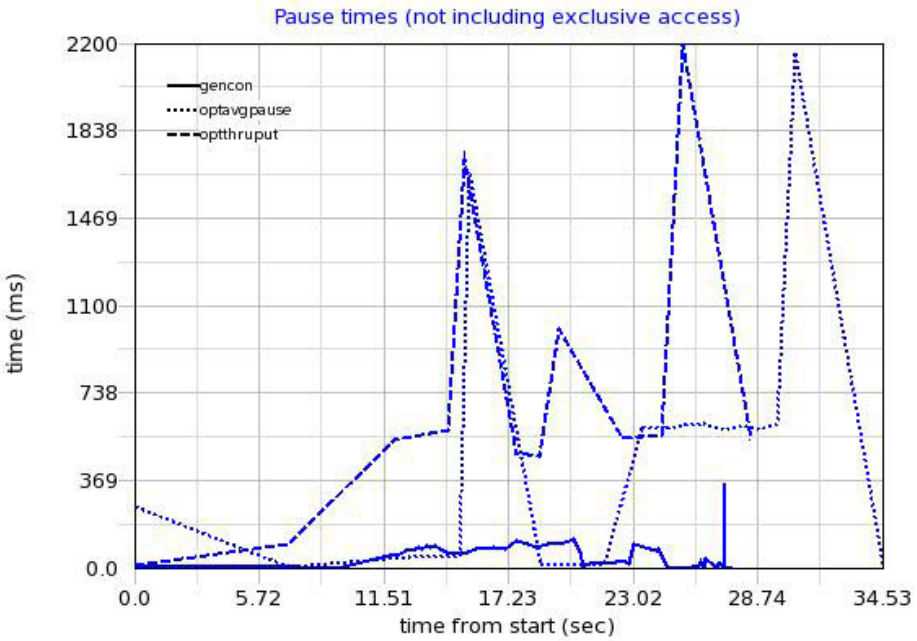
ガーベッジ・コレクターはヒープのサイズを最適化しようと試みますが、マシンで使用可能な物理メモリーの半分以上を使用しようとすることはしません。また、ヒープを最適なサイズにまで増やすには時間がかかる上、アプリケーション占有率の減少によってヒープが縮小される場合もあります。ヒープ・サイズの変動はアプリケーションの速度低下の原因となるだけでなく、同じシステムで動作している他のアプリケーションに物理メモリーが必要ないのであれば無駄なことです。アプリケーションのメモリー要件が十分明確な場合は、ヒープ・サイズを固定することで簡単にパフォーマンスを最適化することができます。

ヒープに理想的なサイズは一通りではありません。通常は、ヒープが大きくなればなるほど、アプリケーションのパフォーマンスは向上するため、ヒープのサイズの決定にはアプリケーションの要件と物理メモリーに対するその他の要求とのトレードオフが関わってきます。試行錯誤として妥当な線は、ヒープのサイズを実データの 2 倍以上にすることです。ヒープをこれほどまで大きくすることが可能ではない場合、あるいはふさわしくないという場合には、gencon ポリシーを使うとほぼ間違いありません。このポリシーは、ヒープ・サイズが制約されているような場合には、optthruput ポリシーより有効に機能するからです。仮に可能であったとしても、マシンが仮想メモリーを使用しないと対応できないようなヒープ・サイズにすることは避けてください。仮想メモリーを使用すると、パフォーマンスが大幅に劣化します。

固定ヒープ・サイズの利点

図 18 に示しているのは、ヒープ・サイズを 500MB に固定し、コマンドライン・オプションに `-Xms500m -Xmx500m` を指定した JVM で、図 7 と同じワークロードを実行した場合の一時停止時間です。一時停止時間だけを基準にすると、ヒープ・サイズを固定したことによって状況は悪化しているように見えます。平均一時停止時間はすべてのポリシーで増加しており、GC に占める時間の割合は変わっていません。けれども合計一時停止時間 (レポート・ビューの表の 4 番目の列) は実際、顕著に減少しています。合計と平均の一時停止時間が矛盾しているように思えるのは、小さなヒープでのコレクションは極めて短時間で完了することが理由となっています。ヒープのサイズが変動する場合、ヒープが小さいうちは JVM が非常に短時間のコレクションを何度も実行することから、平均一時停止時間は短くなるというわけです。そうなると、最終的なパフォーマンス・メトリックは JVM が作業を完了するまでに費やした時間 (線の長さ) ということになります。ヒープを固定した場合のパフォーマンスの向上は、gencon ポリシーでは 13 パーセント、optavgpause ポリシーでは 15 パーセント、optthruput ポリシーでは 30 パーセントという結果となっています。

図 18. 固定ヒープ・サイズで動作するアプリケーションでの一時停止時間



この例はもちろん、30 秒以上実行する Java プログラムの場合で、初期の実行時間のほとんどは JVM が最適なヒープ・サイズを見つけるために費やされます。一般に長時間のプログラムでは、ヒープ・サイズを固定してもこのように劇的な向上はもたらされません。ワークロードが明確になっていない場合は、ヒープ・サイズを固定すると、小さすぎるヒープで JVM が実行させられるおそれがあるため、ヒープ・サイズを固定するのは賢い選択ではないかもしれません。ワークロードがどれだけ安定しているか、そして JVM が許容されている以上のメモリーを必要とする可能性がどれほどあるかを見極めるには、verbose GC の出力を使用することができます。

事例研究: verbose GC のログからアプリケーションのスループットを概算する

アプリケーションのパフォーマンスを最適化するには GC を調整します。では、アプリケーションのパフォーマンスがどれだけ優れているかを判断するにはどうすればいいでしょうか。ベンチマークには明確なパフォーマンス・メトリックがありますが、ガーベッジ・コレクターを調整してベンチマークを最適化し、そのガーベッジ・コレクターの構成が別のアプリケーションに対しても最善の結果を生むと前提するのはあまりにも無分別です。アプリケーションはそれぞれに異なるため、ガーベッジ・コレクターに最適な構成も 1 通りではありません (最適な構成が 1 通りしかないとしたら、ガーベッジ・コレクターをその構成で出荷すればいいわけで、調整の必要はなくなります)。ベンチマークとは異なり、すべてのアプリケーションがそのパフォーマンスの状態を示すレポートを提供するとは限りません。

アプリケーションがそのようなレポートを提供しない場合には、verbose GC のログ自体がパフォーマンスの状態を判断する上でとても有効な手掛かりとなります。ただし、verbose GC のログはアプリケーションのパフォーマンスを評価する出発点として適切ではあるものの、レポートされた一時停止時間を基準として評価するのは明らかに誤っています。前述したヒープ・サイズを固定する例でのように、アプリケーションの処理速度は調整前より速くなったとしても、GC の

オーバーヘッドが変わらなかったり、さらには平均一時停止時間が増加したりする場合もあります。GC に時間をかけ過ぎるアプリケーションは明らかにパフォーマンス・ヒットのように見えますが、オブジェクトの配列方法によっては、GC に長い時間を費やすことでアプリケーションのパフォーマンスが向上する場合もあります。

GC によってアプリケーションが高速化する理由

優れた GC によってアプリケーションのパフォーマンスを向上できることは事実です。gencon モードやコンパクションなどによってオブジェクトが密に配置されると空きリストを検索する必要がなくなるため、新しいオブジェクトを割り当てる時間が、はるかに短縮されます。これは verbose GC のログでは測定されませんが、短時間でのオブジェクト割り当てはアプリケーションに極めて有効です。同じようなタイミングで使用されるオブジェクトが互いに近接するようにオブジェクトを配置すれば (局所配置と言います)、オブジェクトへのアクセスも断然速くなります。賢いガーベッジ・コレクターはオブジェクトへのアクセス速度を最大限にできるようにオブジェクトを再配置します。

GC 一時停止時間がアプリケーションに与える影響を分析する代わりに、生成されるガーベッジの量を調べてください。アプリケーションのパフォーマンスを最もよく表す指標の 1 つは、アプリケーションが生成しているガーベッジの量です。ガーベッジはアプリケーションの処理による副次作用なので、生成されるガーベッジが多いほど、アプリケーションは多くの処理を行っていることになります。生成されるガーベッジはすべて収集されるため、ガーベッジの生成量はすなわちガーベッジ・コレクターの収集量でもあります。

収集されたガーベッジの量をグラフ化するには、GC and Memory Visualizer の VGC Data メニューから Amount freed を選択します。すると、Report タブのグラフに実行中に収集されたガーベッジの平均量および合計量に関する統計が示されます。平均解放量はパフォーマンスの指標として十分ではありません。占有率が安定している場合、コレクションごとの解放量もかなり安定しているはずだからです。一方、アプリケーションのパフォーマンスが優れていれば、より短い時間でより多くの作業が処理されるため、コレクションの頻度も増えることになります。したがって、一定期間のパフォーマンスを示す指標としては、合計解放量のほうが適切です。収集したログが一定の期間をカバーしていない場合、設定した期間にズームインすれば、その期間に限った合計解放量を表示することができます。

さらに有効な指標となるのは、GC のレートです。比較している複数のログが同じ期間を対象としていないとしても、GC のレートが持つ意味は変わらないからです。このレートは、Report タブの上部にある表に記載されます (表が表示されていない場合は、VGC Data メニューの Summary を有効に設定してください)。GC のレートが高いということは、アプリケーションが短期間でより多くの作業を処理していること、つまり優れたパフォーマンスを意味します。

ヒープ・サイズを固定した先ほどの例を考えてみてください。先ほどの例では、GC がアプリケーションにどれほど有効であるかに関して、平均一時停止時間から得られる印象は、ほとんど当てになりませんでした。けれども GC のレートを見てみると、ヒープ・サイズを固定して実行するとレートが高くなっていることがわかります。一例として、図 19 に示す 2 通りで実行した optthruput を比較すると、ヒープ・サイズを前もって設定した場合にはレートが 12 パーセント増しになります。

図 19. コレクション・レートのサマリー・ビュー

Summary

Variant	fixedoptthruput	optthruput
Mean interval between collections (sec)	2.37	0.11
Largest memory request (bytes)	30190432	38273040
Forced collection count	0	0
Number of collections	13	346
Rate of garbage collection	94.506 MB/sec	84.061 MB/sec
Mean garbage collection pause (ms)	710	34.36

GC のレートはガーベッジ生成のレートとしても考えられます。一見すると、ガーベッジの生成は望ましくないことで、最小限にとどめなければならないように思えます。大量のガーベッジを生成するアプリケーションはより多くの負担をガーベッジ・コレクターにかけるため、ガーベッジの生成量が少ないアプリケーションよりもパフォーマンスが劣る可能性が高いことは事実ですが、常にそうであるとも限りません。例えば、オブジェト・プーリングはアプリケーションが生成するガーベッジの量を減らしますが、ガーベッジ・コレクションのパフォーマンスに深刻な影響を与えるおそれもあります (その理由についての詳細は、「参考文献」に記載したパフォーマンスに関する都市伝説の記事を参照してください)。さらに一般的なこととして、破棄されるはずのオブジェクト参照を維持すると生成されるガーベッジの量は減りますが、GC には悪影響を与えがちです。変数のスコープを適切に宣言してインスタンス変数の使用を減らせば、このような類の維持されるオブジェクトを減らすことができます。

アプリケーションの負荷が低い場合、つまり処理対象の作業が十分にない場合には、入ってくる作業がないと GC のレートが減少するため、GC のレートは有効なパフォーマンス指標にはなりません。例えば、すべてのクライアントが切断されたサーバーは大量のガーベッジを生成しませんが、だからと言ってサーバーを調整しなければならないことにはなりません。アプリケーションの負荷が低いことによる良い点は、いずれにせよ、調整はそれほど必要ないということです。個別のトランザクションを高速化することが目的であれば、GC ログをトランザクションの期間に絞り込むと適切な情報を得ることができます。

アプリケーションの応答時間を概算する

アプリケーションのスループットよりアプリケーションの応答時間に重点を置いている場合、verbose GC の一時停止時間が応答時間の指標としてふさわしいと思いがちです。しかし一時停止時間を応答時間の判断基準にできる場合は限られています。たとえ判断基準になるとしても全面的に適用できるわけではないので、一時停止時間から推測する際には十分慎重にならなければなりません。アプリケーションの負荷が低い状態の場合、最大一時停止時間に関連するのは最大応答時間ですが、平均応答時間に関しては通常、スループットに比例します。したがって、一時停止時間が長いポリシー (optthruput など) の平均応答時間は、一時停止時間が短いポリシー (optavgpause など) の場合よりも実際には短くなる可能性があります。アプリケーションが過負荷状態の場合には、一時停止時間の重要性はさらに低くなります。なぜなら、処理はサービス・キューに入れられる可能性があるため、応答時間は主にキューの長さよって決まりますが、このキューの長さはアプリケーションのスループットによって左右されるからです。

まとめ

verbose GC のログを検討することで、アプリケーションの特性をより深く理解する結果になることがよくあります。さらに、アプリケーションのメモリー使用量に関する重大な問題を未然に検出し、パフォーマンスの向上につなげられる場合もあります。IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer は、verbose GC に提供された情報を最大限活用する上での強力なツールです。

著者について

Dr. Holly Cummins

Holly Cummins は、IBM United Kingdom の Java Technology Centre に在籍する開発者で、EVTK の作成者でもあります。これまで 6 年間 IBM に勤務している彼女は、量子計算の博士号とソフトウェア・エンジニアリングの修士号を持っています。ブルーベリー、グリュイエール・チーズ、無名の写植言語、そして非実用的な靴に目がない彼女は、家事の手腕に関してはお粗末で、キッチンのテーブルには郵便物が山積みになっています。