富士通 Java ブログ

Java の性能トラブルを防止 ～Java のガーベジコレクション（GC）ログの見方を知る （シリアル GC / パラレル GC 編）～

シリアル GC/パラレル GC 編 | CMS GC 編 | G1 GC 編

2021 年 12 月 3 日 初版

笠井 輝美

1. はじめに

OpenJDK でアプリケーションを実行すると、Java 仮想マシン（Java Virtual Machine：以降、JVM）が不要になったオブジェクトを Java ヒープ上から削除し、再度利用可能にするためのガーベジコレクション（以降、GC）が行われます。GC によって JVM は Java ヒープを効率的に利用できますが、GC 処理中はアプリケーションの処理が停止するため、GC の長時間化や頻発は性能トラブルに直結します。

GC に関する性能トラブル調査には、GC ログが有用です。GC ログには、GC 処理に費やした時間や GC 前後のメモリー使用状況などが出力されるため、これらを分析することでチューニングやアプリケーション修正に役立てることができます。
GC ログの出力は、JDK 8 と JDK 9 で大きく変わりました。これまでフォーマットにばらつきがあった JVM の各種ログの出力が、「JVM 統合ロギングフレームワーク」に統合され、GC ログも採取方法やフォーマットが変わっています。
今回は、最初に JVM 統合ロギングフレームワークを簡単に紹介した後、GC ログの採取方法、読み方、分析方法について説明します。
なお、GC ログの内容は GC の種別によって異なります。今回は一番わかりやすいシリアル GC とパラレル GC を使用して説明します。
CMS GC や G1 GC に関しては、別記事にて解説を予定しています。

本記事は、デジタルトランスフォーメーションを支えるアプリケーションサーバー「FUJITSU Software Enterprise Application Platform」の OpenJDK 11 を例に説明します。

2. JVM 統合ロギングフレームワーク

JDK 8 までは、JVM の機能ごとにログの採取方法やフォーマットが異なっていましたが、ログを使用する機能が増え続けるなかで、統一したロギングフレームワークが必要となってきました。
そこで、OpenJDK コミュニティーの中で、「JEP 158: Unified JVM Logging」および、「JEP 271: Unified GC Logging」が提案されました。JEP 158 では、コマンドラインオプションの共通化、フォーマットの統一、ログレベルの導入、動的変更の仕組み、などが、ロギングフレームワークに組み込まれました。

JVM 統合ロギングフレームワークを使用するには、-Xlog オプションを指定します。-Xlog オプションの詳細については、「JVM 統合ロギングフレームワークを使用したロギングの有効化（「Oracle Help Center」のページへ）」を参照してください。

```
-Xlog[:[what][:[output][:[decorators][:output-options[,…]]]]
```

3. GC ログの採取方法

「JVM 統合ロギングフレームワーク」で GC ログを採取する場合は、-Xlog オプションを指定する必要があります。この章では「JVM 統合ロギングフレームワーク」による GC ログの採取方法について解説します。

3.1. Java のガーベジコレクション（GC）ログの出力オプション

-Xlog オプションで GC ログを採取する場合、以下のように、「what」の部分に「gc」を設定します。また、「output」には出力先を、「decorators」には出力フォーマットをそれぞれ設定できます。

```
-Xlog:gc[*][:[output][:[decorators]]]
```


オプションの例を以下の表で示します。目的に合わせて指定してください。

オプション	説明
-Xlog:gc	簡易的な GC 結果を出力します。このオプションは-verbose:gc と同値です。以降、この出力形式を「簡易ログ」と表記します。
-Xlog:gc*	タグに「gc」を含むすべての GC メッセージを出力します。以降、この出力形式を「詳細ログ」と表記します。
-Xlog:gc:<出力ファイルのパス>	「output」に出力ファイルのパスを指定することで、GC 結果をファイル出力します。
-Xlog:gc::uptime	「decorators」に「uptime」を指定することで、JVM 起動からの時間を含んだ GC 結果を出力します。-Xlog:gc のデフォルトフォーマットで、JDK 8 の-XX:+PrintGCTimeStamps と同じフォーマットです。 出力例：0.043s
-Xlog:gc::time	「decorators」に「time」を指定することで、日付時刻を含んだ GC 結果を出力します。JDK 8 の-XX:+PrintGCDateStamps と同じフォーマットです。 出力例：2021-10-18T18:21:26.393+0900
-Xlog:gc::none	「decorators」に「none」を指定することで、時刻情報を含まない GC 結果を出力します。JDK 8 の-verbose:gc のデフォルトも、時刻情報を含まない形式で出力されます。

3.2 非推奨オプション

以下の表に記載したオプションは、JDK 8 では利用できましたが、JDK 11 では非推奨となります。-Xlog:gc を利用してください。

オプション	説明
-XX:+PrintGCDetails	指定すると、警告文が出力され、自動的に-Xlog:gc*が指定されます。
-Xloggc:<出力ファイルのパス>	指定すると、警告文が出力され、自動的に-Xlog:gc:<出力ファイルのパス>が指定されます。

4. Java のガーベジコレクション（GC）ログの読み方（シリアル GC / パラレル GC）

この章では、シリアル GC とパラレル GC において GC ログの共通となる読み方について紹介します。

シリアル GC は単一スレッドで動作する軽量な GC です。CPU のコア数が少ない場合、または使用するヒープ量が小さい場合の動作に適しているため、主にデスクトップクライアント環境で動作するアプリケーションに適しています。シリアル GC を使用するには、-XX:+UseSerialGC オプションを指定します。

パラレル GC は GC 処理を複数のスレッドで動作させることで GC 実行時間を短縮するもので、主にスループット性能を重視するアプリケーションに適しています。パラレル GC を使用するには、-XX:+UseParallelGC オプションを指定します。パラレル GC 使用時、デフォルトでは Full GC も複数スレッドで並列実行されます（パラレルオールド GC）。
ただし、-XX:-UsePrallelOldGC オプションを指定することで、Full GC 処理を単一スレッドで実施させることもできます。

4.1. 簡易ログ（-Xlog:gc）

シリアル GC とパラレル GC では、GC 対象となる領域が世代別に分かれて管理されています。New 世代領域を対象にした GC を「New GC」、New 世代領域・Old 世代領域などを対象にした GC を「Full GC」と呼びます。なお、世代別領域については「世代別ガベージ・コレクション（「Oracle Help Center」のページへ）」を参照してください。

-Xlog:gc を指定した時の出力形式（簡易ログ）について、New GC と Full GC のそれぞれについて説明します。

4.1.1. New GC

出力形式

```
[タイムスタンプ][info][gc] GC(GC 通し番号) Pause Young (GC 発生理由) GC 直前のヒープ使用サイズ->GC 直後のヒープ使用サイズ(ヒープサイズ) GC 時間
```

出力例

```
[15.010s][info][gc] GC(28) Pause Young (Allocation Failure) 340M->292M(455M) 22.227ms
```

意味

出力例のログからは以下の内容が読み取れます。

- JMV 起動から 15.01 秒後に New 世代領域不足が原因で、28 回目の GC が発生しています。
- ヒープ使用量は 340MB から 292MB になり、GC 後のヒープサイズは 455MB です。
- この GC 処理に費やした時間は、22.227ms です。

4.1.2 Full GC

出力形式

```
[タイムスタンプ][info][gc] GC(GC 通し番号) Pause Full (GC 発生理由) GC 直前のヒープ使用サイズ->GC 直後のヒープ使用サイズ(ヒープサイズ) GC 時間
```

出力例

```
[15.561s][info][gc] GC(29) Pause Full (Ergonomics) 292M->284M(455M) 550.828ms
```

意味

出力例のログからは以下の内容が読み取れます。

- JVM 起動から 15.561 秒後に領域不足（注 1）が原因で、29 回目の GC が発生しています。
- ヒープ使用量は 292MB から 284MB になり、GC 後のヒープサイズは 455MB です。
- この GC 処理に費やした時間は、550.828ms です。

注 1 Ergonomics と表示された場合、New 世代領域が不足し実施した New GC で失敗、あるいは、Old 世代領域の空きが不足して今後の New GC に失敗すると判断されたため、Full GC が発生しています。JVM のエルゴノミクスについては、「付録. エルゴノミクス」を参照してください。

4.2. 詳細ログ（-Xlog:gc*）

-Xlog:gc*を指定した時の出力形式（詳細ログ）について、New GC と Full GC のそれぞれについて説明します。1 つの GC 処理に対して複数行の出力が行われます。

4.2.1. New GC

出力形式

```
[タイムスタンプ][info][gc,start       ] GC(GC 通し番号) Pause GC (GC 発生理由)
[タイムスタンプ][info][gc,heap        ] GC(GC 通し番号) 対象の世代領域名: GC 直前の使用サイズ->GC 直後の使用サイズ(世代領域サイズ)
[タイムスタンプ][info][gc,metaspace   ] GC(GC 通し番号) Metaspace: GC 直前の metaspace 使用サイズ->GC 直後の metaspace 使用サイズ(metaspace サイズ)
[タイムスタンプ][info][gc             ] GC(GC 通し番号) Pause Young (GC 発生理由) GC 直前の Java ヒープ使用サイズ->GC 直後の Java ヒープ使用サイズ(ヒープサイズ) GC 時間
[タイムスタンプ][info][gc,cpu         ] GC(GC 通し番号) User=UUUs Sys=SSSs Real=RRRs
```


注 2 UUU、SSS、RRR は、GC 処理で使用した時間を示す数値（秒）で、それぞれ、ユーザーCPU 時間、システム CPU 時間、経過時間を示します。

以下は、シリアル GC とパラレル GC の出力例ですが、世代領域名が異なる以外は、同じ内容になります。

出力例（シリアル GC）

```
[12.088s][info][gc,start       ] GC(18) Pause Young (Allocation Failure)
[12.144s][info][gc,heap        ] GC(18) DefNew: 157248K->17471K(157248K)
[12.144s][info][gc,heap        ] GC(18) Tenured: 280045K->285408K(349568K)
[12.144s][info][gc,metaspace   ] GC(18) Metaspace: 13272K->13272K(1060864K)
[12.144s][info][gc             ] GC(18) Pause Young (Allocation Failure) 427M->295M(494M) 56.154ms
[12.144s][info][gc,cpu         ] GC(18) User=0.09s Sys=0.00s Real=0.06s
```

意味

出力例のログからは以下の内容が読み取れます。

- New 世代領域不足が原因で、18 回目の GC が発生しました。
    - New 世代領域（DefNew）の使用量は、157,248KB から 17,471KB になり、GC 後の領域サイズは 157,248KB です。
    - Old 世代領域（Tenured）の使用量は、280,045KB から 285,408KB になり、GC 後の領域サイズは 349,568KB です。
    - Metaspace 領域の使用量は 13,272KB のままで、GC 後の領域サイズは 1,060,864KB です。
- ヒープ全体では、使用量が 427MB から 295MB になり、処理時間が 56.154ms でした。

出力例（パラレル GC）

```
[7.646s][info][gc,start       ] GC(16) Pause Young (Allocation Failure)
[7.682s][info][gc,heap        ] GC(16) PSYoungGen: 68032K->9280K(116736K)
[7.683s][info][gc,heap        ] GC(16) ParOldGen: 280578K->287666K(349696K)
[7.683s][info][gc,metaspace   ] GC(16) Metaspace: 13304K->13304K(1060864K)
[7.683s][info][gc             ] GC(16) Pause Young (Allocation Failure) 340M->289M(455M) 36.578ms
[7.683s][info][gc,cpu         ] GC(16) User=0.07s Sys=0.01s Real=0.04s
```

意味

出力例のログからは以下の内容が読み取れます。

- New 世代領域不足が原因で、16 回目の GC が発生しました。
    - New 世代領域（PSYoungGen）の使用量は、68,032KB から 9,280KB になり、GC 後の領域サイズは 116,736KB です。
    - Old 世代領域（ParOldGen）の使用量は、280,578KB から 287,666KB になり、GC 後の領域のサイズは 349,696KB です。
    - Metaspace 領域の使用量は 13,304KB のままで、GC 後の領域サイズは 1,060,864KB です。
- ヒープ全体では、使用量が 340MB から 289MB になり、処理時間が 36.578ms でした。

4.2.2. Full GC

出力形式

```
[タイムスタンプ][info][gc,start       ] GC(GC 通し番号) Pause Full (GC 発生理由)
[タイムスタンプ][info][gc,phases,start] GC(GC 通し番号) Phase GC フェーズの通し番号: GC フェーズの処理名
[タイムスタンプ][info][gc,phases      ] GC(GC 通し番号) Phase GC フェーズの通し番号: GC フェーズの処理名 GC フェーズの処理時間
[タイムスタンプ][info][gc,heap        ] GC(GC 通し番号) 対象の世代領域名: GC 直前の使用サイズ->GC 直後の使用サイズ(世代領域サイズ)
[タイムスタンプ][info][gc,metaspace   ] GC(GC 通し番号) Metaspace: GC 直前の metaspace 使用サイズ->GC 直後の metaspace 使用サイズ(metaspace サイズ)
[タイムスタンプ][info][gc             ] GC(GC 通し番号) Pause Full (GC 発生理由) GC 直前の Java ヒープ使用サイズ->GC 直後の Java ヒープ使用サイズ GC 時間(ヒープサイズ)
[タイムスタンプ][info][gc,cpu         ] GC(GC 通し番号) User=UUUs Sys=SSSs Real=RRRs
```


注 3 UUU、SSS、RRR は、GC 処理で使用した時間を示す数値（秒）で、それぞれ、ユーザーCPU 時間、システム CPU 時間、経過時間を示します。

注 4 シリアル GC では、CPU 時間情報は、GC 発生理由が System.gc()の場合に出力されます。

以下は、シリアル GC とパラレル GC の出力例ですが、世代領域名および GC フェーズが異なります。

出力例（シリアル GC）

```
[12.533s][info][gc,start       ] GC(20) Pause Full (Allocation Failure)
[12.533s][info][gc,phases,start] GC(20) Phase 1: Mark live objects
[12.741s][info][gc,phases      ] GC(20) Phase 1: Mark live objects 207.725ms
[12.741s][info][gc,phases,start] GC(20) Phase 2: Compute new object addresses
[12.895s][info][gc,phases      ] GC(20) Phase 2: Compute new object addresses 154.720ms
[12.895s][info][gc,phases,start] GC(20) Phase 3: Adjust pointers
[13.033s][info][gc,phases      ] GC(20) Phase 3: Adjust pointers 137.146ms
[13.033s][info][gc,phases,start] GC(20) Phase 4: Move objects
[13.050s][info][gc,phases      ] GC(20) Phase 4: Move objects 17.793ms
[13.051s][info][gc             ] GC(20) Pause Full (Allocation Failure) 432M->241M(494M) 518.108ms
```

意味

出力例のログからは以下の内容が読み取れます。

- 領域不足が原因で、20 回目の GC が発生しました。
- フェーズごとの処理時間は以下のとおりです。
    - Mark live objects：207.725ms
    - Compute new object addresses：154.72ms
    - Adjust pointers：173.146ms
    - Move objects：17.793ms
- ヒープ全体では、使用量が 432MB から 241MB になり、処理時間が 518.108ms でした。

出力例（パラレル GC）

```
[7.683s][info][gc,start       ] GC(17) Pause Full (Ergonomics)
[7.683s][info][gc,phases,start] GC(17) Marking Phase
[8.025s][info][gc,phases      ] GC(17) Marking Phase 341.989ms
[8.025s][info][gc,phases,start] GC(17) Summary Phase
[8.025s][info][gc,phases      ] GC(17) Summary Phase 0.017ms
[8.025s][info][gc,phases,start] GC(17) Adjust Roots
[8.036s][info][gc,phases      ] GC(17) Adjust Roots 11.168ms
[8.036s][info][gc,phases,start] GC(17) Compaction Phase
[8.546s][info][gc,phases      ] GC(17) Compaction Phase 509.993ms
[8.546s][info][gc,phases,start] GC(17) Post Compact
[8.548s][info][gc,phases      ] GC(17) Post Compact 1.806ms
[8.548s][info][gc,heap        ] GC(17) PSYoungGen: 9280K->0K(116736K)
[8.548s][info][gc,heap        ] GC(17) ParOldGen: 287666K->229676K(349696K)
[8.548s][info][gc,metaspace   ] GC(17) Metaspace: 13304K->13304K(1060864K)
[8.548s][info][gc             ] GC(17) Pause Full (Ergonomics) 289M->224M(455M) 865.247ms
[8.548s][info][gc,cpu         ] GC(17) User=1.67s Sys=0.02s Real=0.86s
```

意味

出力例のログからは以下の内容が読み取れます。

- 領域不足（注 5）が原因で、17 回目の GC が発生しました。
- フェーズごとの処理時間は以下のとおりです。
    - Marking Phase：341.989ms
    - Summary Phase：0.017ms
    - Adjust Roots：11.168ms
    - Compaction Phase：509.993ms
    - Post Compact：1.806ms

- New 世代領域（PSYoungGen）の使用量は、9,280KB から 0KB になり、GC 後の領域サイズは 116,736KB です。
- Old 世代領域（ParOldGen）の使用量は、287,666KB から 229,676KB になり、GC 後の領域サイズは 349,696KB です。
- Metaspace 領域の使用量は、13,304KB のままで、GC 後の領域サイズは 1,062,912KB です。
- ヒープ全体では、使用量が 289MB から 224MB になり、処理時間が 865.247ms でした。

注 5 記載されている「Ergonomics」については、「付録. エルゴノミクス」を参照してください。

出力例（パラレル GC、パラレルオールド GC を無効にした場合）

```
[7.220s][info][gc,start       ] GC(17) Pause Full (Ergonomics)
[7.220s][info][gc,phases,start] GC(17) Phase 1: Mark live objects
[7.454s][info][gc,phases      ] GC(17) Phase 1: Mark live objects 233.053ms
[7.454s][info][gc,phases,start] GC(17) Phase 2: Compute new object addresses
[7.557s][info][gc,phases      ] GC(17) Phase 2: Compute new object addresses 103.607ms
[7.557s][info][gc,phases,start] GC(17) Phase 3: Adjust pointers
[7.702s][info][gc,phases      ] GC(17) Phase 3: Adjust pointers 145.089ms
[7.702s][info][gc,phases,start] GC(17) Phase 4: Move objects
[7.726s][info][gc,phases      ] GC(17) Phase 4: Move objects 23.405ms
[7.726s][info][gc,heap        ] GC(17) PSYoungGen: 9248K->0K(116736K)
[7.726s][info][gc,heap        ] GC(17) PSOldGen: 287351K->247386K(349696K)
[7.726s][info][gc,metaspace   ] GC(17) Metaspace: 13304K->13304K(1062912K)
[7.726s][info][gc             ] GC(17) Pause Full (Ergonomics) 289M->241M(455M) 505.936ms
[7.726s][info][gc,cpu         ] GC(17) User=0.60s Sys=0.00s Real=0.51s
```

意味

出力例のログからは以下の内容が読み取れます。

- 領域不足（注 6）が原因で、17 回目の GC が発生しました。
- フェーズごとの処理時間は以下のとおりです。
    - Mark live objects：233.053ms
    - Compute new object addresses：103.607ms
    - Adjust pointers：145.089ms
    - Move objects：23.405ms
- New 領域（PSYoungGen）の使用量は、9,248KB から 0KB になり、GC 後の領域サイズは 116,736KB です。
- Old 領域（PSOldGen）の使用量は、287,351KB から 247,386KB になり、GC 後の領域サイズは 349,696KB です。
- Metaspace 領域の使用量は、13,304KB のままで、GC 後の領域サイズは 1,060,864KB です。
- ヒープ全体では、使用量が 289MB から 241MB になり、処理時間が 505.936ms でした。

注 6 記載されている「Ergonomics」については、「付録. エルゴノミクス」を参照してください。

5. Java のガーベジコレクション（GC）ログの分析

この章では、GC ログを分析するポイントについて解説します。

（1）長時間 GC 発生の確認

GC 処理時間が長い GC があるかどうかを確認します（以降の説明では、処理時間が長い GC のことを「長時間 GC」と表現します）。以下は、パラレル GC の Full GC で、長時間 GC が発生している例です。

GC ログに Pause と出力されている場合、GC 処理によりアプリケーションが停止することを示しています。以下の例では赤字で書かれている GC がそれぞれ 9 秒（9043.206ms）、11 秒（11992.944ms）の時間を費やしていることが読み取れます。

```
簡易ログ：
[20.294s][info][gc] GC(9) Pause Full (Ergonomics) 1092M->1063M(1678M) 9043.206ms

詳細ログ：
[55.586s][info][gc,start       ] GC(13) Pause Full (Ergonomics)
[55.586s][info][gc,phases,start] GC(13) Marking Phase
```



```
[62.247s][info][gc,phases      ] GC(13) Marking Phase 6660.840ms
[62.247s][info][gc,phases,start] GC(13) Summary Phase
[62.247s][info][gc,phases      ] GC(13) Summary Phase 0.068ms
[62.247s][info][gc,phases,start] GC(13) Adjust Roots
[62.264s][info][gc,phases      ] GC(13) Adjust Roots 17.037ms
[62.264s][info][gc,phases,start] GC(13) Compaction Phase
[66.001s][info][gc,phases      ] GC(13) Compaction Phase 3655.816ms
[66.133s][info][gc,phases,start] GC(13) Post Compact
[67.073s][info][gc,phases      ] GC(13) Post Compact 699.210ms
[67.362s][info][gc,heap        ] GC(13) PSYoungGen: 37216K->0K(465920K)
[67.407s][info][gc,heap        ] GC(13) ParOldGen: 1101894K->1107755K(1398272K)
[67.491s][info][gc,metaspace   ] GC(13) Metaspace: 13502K->13502K(1062912K)
[67.607s][info][gc             ] GC(13) Pause Full (Ergonomics) 1112M->1081M(1820M) 11992.944ms
[67.624s][info][gc,cpu         ] GC(13) User=6.00s Sys=0.07s Real=12.04s
```

GC 処理に許容される時間はアプリケーションに依存しますが、1 秒以上費やす GC はアプリケーションの性能に影響している可能性があります。

アプリケーションに、レスポンス遅延が発生している場合は、長時間 GC の発生時刻とレスポンス遅延発生時刻を照らし合わせます。それらが一致する場合、GC によってアプリケーションが長時間停止し、性能問題を引き起こしていると考えられます。このような場合は、長時間 GC の発生を防ぐように、Java ヒープサイズ等のチューニングが必要です。

（2）Full GC 後のヒープ使用量の確認

Full GC 後のヒープ使用量が増え続けている場合、メモリーリークが発生している可能性があります。複数回 Full GC が発生している GC ログから「GC 直後の Java ヒープ使用サイズ」をプロットしたグラフを作成すると、ヒープ使用量が増加傾向にあるかどうかが分かります。

以下は簡易ログにおける Full GC の出力例ですが、プロットする値となる「GC 直後の Java ヒープ使用サイズ」は赤字で書かれている値です。

```
[151.804s][info][gc] GC(390) Pause Full (Ergonomics) 484M->290M(489M) 626.136ms
[163.442s][info][gc] GC(425) Pause Full (Ergonomics) 352M->313M(458M) 649.544ms
[184.115s][info][gc] GC(501) Pause Full (Ergonomics) 468M->341M(472M) 886.016ms
[185.650s][info][gc] GC(502) Pause Full (Ergonomics) 434M->353M(472M) 1234.186ms
```

図 1 は、生存オブジェクトの量をプロットしてグラフを作成した例です。

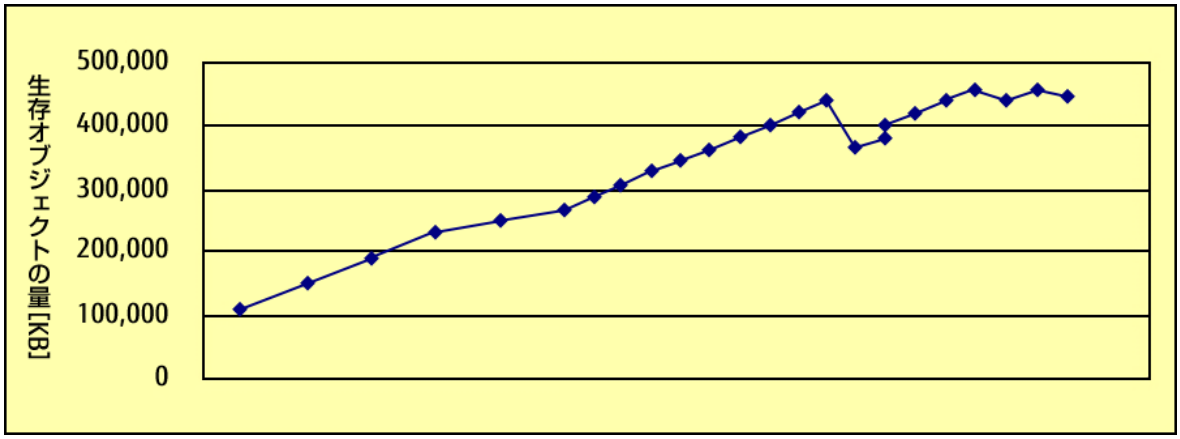

図 1 生存オブジェクト量が増加傾向にある例

このような傾向がみられる場合は、ヒープダンプやヒープヒストグラムを採取し、メモリーリークが発生しているかを特定します。ヒープダンプやヒープヒストグラムの採取方法やメモリーリークの特定方法は、別記事で解説を予定しています。

6. おわりに

今回は、GC ログの読み方を紹介して、GC に関わるトラブルの確認方法について説明しました。GC の頻発や長時間 GC、メモリーリークは、アプリケーションの停止やレスポンス悪化のトラブルにつながりますので、本記事を安定稼働の参考にしてください。

次回は、CMS GC、G1 GC のログの読み方を紹介します。

付録. エルゴノミクス

エルゴノミクスとは、JVM がパフォーマンスを向上させるために、GC の種別やヒープサイズを自動的に最適化する機能のことです。

エルゴノミクスでは-XX:+UseParallelGC などのオプションで明示的に指定しない限り、GC の種別は G1 GC が使用されます。-Xmx や-Xms などのオプションで明示的に指定しない限り、使用可能な CPU や物理メモリーから計算されたヒープサイズの最大値と最小値が自動的に設定されます。

エルゴノミクスは、内部でこの最大値と最小値の範囲でヒープサイズを自動で調整し、GC によるアプリケーション最大一時停止時間とスループット性能の目標を達成するように最適化します。

Full GC の GC ログの出力例では GC の発生理由として「Ergonomics」と表記されています。この表記の理由は、Full GC の前に実施された New 世代 GC によって Old 世代領域に移動したオブェクトサイズの統計情報を採取して、 この次に New 世代 GC を実施した場合に自動設定された Old 世代領域のサイズでは不足する可能性があると、JVM が自動的に判断した結果、Full GC が発生したためです。