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（G1 GC 編）～

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

2023 年 6 月 9 日 初版
倉繁 泰三
小形 俊貴

1. はじめに

Java アプリケーションが Java 仮想マシン（Java Virtual Machine：以降、「JVM」と表記します）により実行される際、不要になった Java オブジェクトの解放は、JVM が提供するガーベジコレクターによって自動的に行われます。ガーベジコレクターによるメモリー解放処理であるガーベジコレクション（以降、「GC」と表記します）により、ユーザーはメモリー解放を意識する必要はなくなります。しかし、一方で、一般的なガーベジコレクターでは、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プログラムの修正に役立てることができます。

以前に掲載した『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）～』では、シリアル GC / パラレル GC におけるログの読み方、『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（CMS GC 編）～』では CMS GC におけるログの読み方について説明しました。
今回は、Garbage-First GC（以降、「G1 GC」と表記します）におけるログの読み方について説明します。

本記事は、FUJITSU Software Enterprise Application Platform の OpenJDK 11 を例に説明します。製品の詳細は下記より製品情報ページをご覧ください。

- FUJITSU Software Enterprise Application Platform

2. Java のガーベジコレクション(GC)ログの読み方(G1 GC)

2.1 G1 GC と GC ログの概要

G1 GC は、Java ヒープを領域(Region)として分割し使用します。各領域は、以下のどれか 4 つの領域種別に分類されます。

- Eden 領域
- Survivor 領域
- Old 領域
- Humongous 領域

各領域の種別は固定ではなく、Java プロセス生存中に、別の種別に変更されることがあります。

G1 GC は、以下から構成されています

- Young GC
- Concurrent Cycle
- Mixed GC
- Full GC

Young GC は Eden 領域・Survivor 領域の GC を、Concurrent Cycle は Eden 領域・Survivor 領域・Humongous 領域の GC を、Mixed GC は Eden 領域・Survivor 領域・Old 領域の GC を、Full GC はすべての領域の GC を行います。なお、例外として、Humongous 領域に割り当たっている大型のプリミティブタイプの配列については Young GC・Mixed GC で GC の対象となります。また、Concurrent Cycle では空になった領域の回収も実施します。
また、Full GC 以外の GC で発生するアプリケーションの停止時間は、停止目標時間に収まるように調整されます。停止目標時間のデフォルト値は 200ms ですが、オプション-XX:MaxGCPauseMillis を使用してチューニング可能です。

G1 GC の詳細は、『ガベージファースト・ガベージ・コレクタ（「Oracle Help Center」のページへ）』をご参照ください。

GC で行われる処理に費やした時間や、GC 前後でのヒープ使用量などは、「-Xlog」java 起動オプションを指定した際に出力される GC ログの内容を参照することで分かります。
-Xlog オプションの詳細は『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）』をご参照ください。
-Xlog オプションで採取できる GC ログには、-Xlog:gc オプションによる簡易的なログ（以降、「簡易ログ」と表記します）と、-Xlog:gc* オプションによる詳細なログ（以降、「詳細ログ」と表記します）の 2 種類があります。それぞれ、2.2、2.3 で、各ログのフォーマットと出力例を説明します。

2.2 簡易ログ（-Xlog:gc）

本節では、G1 GC で出力される簡易ログの形式を説明します。

2.2.1 Young GC / Mixed GC

【出力形式】

太字の部分が可変情報で、それ以外の部分は固定情報です。

```
[タイムスタンプ][info][gc] GC(GC 通し番号) To-space exhausted
[タイムスタンプ][info][gc] GC(GC 通し番号) Pause Young（ラベル）（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(Java ヒープサイズ) GC 時間
```

- 「To-space exhausted」のログは GC の際にオブジェクトのコピー先の領域が確保できなかった場合のみ表示されます。通常 Full GC の前にはこのログが表示されます。
- ラベルには以下の 4 種類があり、これらは実行される GC の分類を表します。
    - Normal
    - Concurrent Start
    - Prepare Mixed
    - Mixed

  Normal は通常の Young GC、Concurrent Start は Concurrent Cycle の直前に行われる Young GC、Prepare Mixed は Mixed GC の直前に行われる Young GC、Mixed は Mixed GC を表します。

【出力例】

```
[2.242s][info][gc] GC(3) Pause Young (Normal) (G1 Evacuation Pause) 212M->145M(2048M) 79.350ms
```

【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 2.242 秒後に、3 回目の GC として、通常の Young GC が発生しています。
- Java ヒープ使用量は 212MB から 145MB になり、GC 後の Java ヒープサイズは 2048MB です。
- この GC に費やした時間は、79.350ms です。

2.2.2 Concurrent Cycle

Concurrent Cycle でログとして出力されるフェーズは 2 種類存在します。アプリケーションを停止して処理を行う GC フェーズとアプリケーションを停止せずに処理を行う Marking フェーズです。GC フェーズは簡易ログと詳細ログの両方に出力され、Marking フェーズは詳細ログにのみ出力されます。

【出力形式】

太字の部分が可変情報で、それ以外の部分は固定情報です。

```
[タイムスタンプ][info][gc] GC(GC 通し番号) Concurrent Cycle
[タイムスタンプ][info][gc] GC(GC 通し番号) Pause GC フェーズ名 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(Java ヒープサイズ) GC 時間
[タイムスタンプ][info][gc] GC(GC 通し番号) Concurrent Cycle GC 時間
```


- Concurrent Cycle は時間を要するフェーズがあるため、同じ GC 通し番号のログが離れて出力されている場合があります。
- ログに表れる GC フェーズ名は、Remark、Cleanup の 2 種類です。
- Concurrent Cycle は中断される場合があります。GC フェーズのログは、Pause Remark フェーズが中断された場合は出力されません。Pause Cleanup フェーズが中断された場合は、Pause Remark フェーズのログだけが出力され、Pause Cleanup フェーズのログは出力されません。

【出力例】

```
[5.303s][info][gc] GC(25) Concurrent Cycle
[5.963s][info][gc] GC(25) Pause Remark 1310M->1202M(2048M) 8.205ms
[6.383s][info][gc] GC(25) Pause Cleanup 1202M->1202M(2048M) 0.809ms
[6.387s][info][gc] GC(25) Concurrent Cycle 1083.885ms
```

【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 5.303 秒後に Concurrent Cycle が開始されました。
- Pause から始まる GC フェーズは、アプリケーションを停止して処理を行います。GC フェーズごとの処理時間は以下のとおりです。
    - Pause Remark：8.205ms
    - Pause Cleanup：0.809ms
- Concurrent Cycle の期間は、1083.885ms でした。

2.2.3 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 Java ヒープ使用サイズ->GC 直後の Java ヒープ使用サイズ(Java ヒープサイズ) GC 時間
```

【出力例】

```
[229.661s][info][gc] GC(240) Pause Full (G1 Evacuation Pause) 2047M->1105M(2048M) 539.193ms
```

【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 229.661 秒後に Java ヒープ領域不足が原因で、240 回目の GC として、Full GC が発生しています。
- Java ヒープ使用量は 2047MB から 1105MB になり、GC 後の Java ヒープサイズは 2048MB です。
- この GC に費やした時間は、539.193ms です。

2.3 詳細ログ（-Xlog:gc*）

本節では、G1 GC で出力される詳細ログの形式を説明します。

2.3.1 Young GC / Mixed GC

【出力形式】

太字の部分が可変情報で、それ以外の部分は固定情報です。

```
タイムスタンプ][info][gc,start       ] GC(GC 通し番号) Pause Young (ラベル) (GC 発生理由)
[タイムスタンプ][info][gc,task        ] GC(GC 通し番号) Using GC スレッド数 workers of 最大 GC スレッド数 for evacuation
[タイムスタンプ][info][gc             ] GC(GC 通し番号) To-space exhausted
[タイムスタンプ][info][gc,phases      ] GC(GC 通し番号)   GC フェーズ名: GC 時間
[タイムスタンプ][info][gc,heap        ] GC(GC 通し番号) 領域名: GC 直前の使用領域数->GC 直後の使用領域数(GC 直後の領域数)
[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 直前の Metaspace サイズ)->GC 直後の Metaspace 使用サイズ(GC 直後の Metaspace サイズ) NonClass: GC 直前の使用サイズ(クラ
```



```
ス以外の情報を格納する Metaspace サイズ)->GC 直後の使用サイズ(クラス以外の情報を格納する Metaspace サイズ)
Class: GC 直前の使用サイズ(クラスの情報を格納する Metaspace サイズ)->GC 直後の使用サイズ(クラスの情報を格納す
る Metaspace サイズ)
[タイムスタンプ][info][gc              ] GC(GC 通し番号) Pause Young (ラベル) (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(Java 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
```

- 「To-space exhausted」のログは GC の際にオブジェクトのコピー先の領域が確保できなかった場合のみ表示されます。通常 Full GC の前にはこのログが表示されます。
- ログに表れる GC フェーズは、Pre Evacuate Collection Set、Evacuate Collection Set、Post Evacuate Collection Set、Other の 4 種類です。
- GC 直後の領域数は、Old 領域と Humongous 領域に対しては表示されません。
- Metaspace の情報では、(GC 直後の Metaspace サイズ)の部分までしか出力されず、領域の内訳情報が出力されない場合があります。
- UUU、SSS、RRR は、GC 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。

【出力例】

```
[784.757s][info][gc,start       ] GC(2830) Pause Young (Mixed) (G1 Evacuation Pause)
[784.757s][info][gc,task        ] GC(2830) Using 4 workers of 4 for evacuation
[784.788s][info][gc,phases      ] GC(2830)   Pre Evacuate Collection Set: 0.3ms
[784.788s][info][gc,phases      ] GC(2830)   Evacuate Collection Set: 29.6ms
[784.788s][info][gc,phases      ] GC(2830)   Post Evacuate Collection Set: 0.6ms
[784.788s][info][gc,phases      ] GC(2830)   Other: 0.7ms
[784.788s][info][gc,heap        ] GC(2830) Eden regions: 84->0(130)
[784.788s][info][gc,heap        ] GC(2830) Survivor regions: 18->13(13)
[784.788s][info][gc,heap        ] GC(2830) Old regions: 1281->1254
[784.788s][info][gc,heap        ] GC(2830) Humongous regions: 365->365
[784.788s][info][gc,metaspace   ] GC(2830) Metaspace: 26881K(28288K)->26881K(28288K) NonClass:
24103K(25088K)->24103K(25088K) Class: 2778K(3200K)->2778K(3200K)
[784.788s][info][gc             ] GC(2830) Pause Young (Mixed) (G1 Evacuation Pause) 1745M->1630M(2048M)
31.125ms
[784.788s][info][gc,cpu         ] GC(2830) User=0.12s Sys=0.00s Real=0.03s
```

【意味】

出力例のログからは以下の内容が読み取れます。

- 2830 回目の GC が発生しました。この GC は Mixed GC です。
- Mixed GC に利用可能な 4 つの GC スレッドのうち、4 つの GC スレッドを使用しました。
- GC フェーズごとの処理時間は以下のとおりです。
    - Pre Evacuate Collection Set: 0.3ms
    - Evacuate Collection Set: 29.6ms
    - Post Evacuate Collection Set: 0.6ms
    - Other: 0.7ms
- 各領域の個数は以下のとおりです。
    - Eden 領域: GC 前の使用数=84、GC 後の使用数=0、GC 後に使用可能な数=130
    - Survivor 領域: GC 前の使用数=18、GC 後の使用数=13、GC 後に使用可能な数=13
    - Old 領域: GC 前の使用数=1281、GC 後の使用数=1254
    - Humongous 領域: GC 前の使用数=365、GC 後の使用数=365
- Metaspace 領域の使用サイズは、26881KB から 26881KB になり、GC 後の Metaspace サイズは 28288KB です。
- Java ヒープ全体では、使用量が 1745MB から 1630MB になり、処理時間は 31.125ms でした。
- GC に要したユーザーCPU 時間は 0.12s、システム CPU 時間は 0.00s であり、実際の経過時間は 0.03s でした。

2.3.2 Concurrent Cycle

【出力形式】

太字の部分が可変情報で、それ以外の部分は固定情報です。

```
[タイムスタンプ][info][gc             ] GC(GC 通し番号) Concurrent Cycle
[タイムスタンプ][info][gc,marking     ] GC(GC 通し番号) Concurrent Marking フェーズ名
[タイムスタンプ][info][gc,marking     ] GC(GC 通し番号) Concurrent Marking フェーズ名 Marking 時間
[タイムスタンプ][info][gc,marking     ] GC(GC 通し番号) Concurrent Mark (タイムスタンプ)
[タイムスタンプ][info][gc,marking     ] GC(GC 通し番号) Concurrent Mark (Concurrent Mark 開始時のタイムスタンプ, タイムスタンプ) Concurrent Mark 時間
[タイムスタンプ][info][gc,task        ] GC(GC 通し番号) Using GC スレッド数 workers of 最大 GC スレッド数 for marking
[タイムスタンプ][info][gc,stringtable ] GC(GC 通し番号) Cleaned string and symbol table, strings: stringtable のエントリ数 processed, 削除した stringtable のエントリ数 removed, symbols: symbol table のエントリ数 processed, 削除した symbol table のエントリ数 removed
[タイムスタンプ][info][gc,start       ] GC(GC 通し番号) Pause GC フェーズ名
[タイムスタンプ][info][gc             ] GC(GC 通し番号) Pause GC フェーズ名 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(Java 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
[タイムスタンプ][info][gc             ] GC(GC 通し番号) Concurrent Cycle 処理時間
```

- Concurrent Cycle は時間を要するフェーズがあるため、同じ GC 通し番号のログが離れて出力されている場合があります。
- Concurrent Cycle は、Full GC の発生など、何らかの理由で中断される場合があります。その場合は一部のみログとして出力されます。
- ログに表れる Marking フェーズは Clear Claimed Marks、Scan Root Regions、Mark From Roots、Preclean、Rebuild Remembered Sets、Cleanup for Next Mark の 6 種類です。
- Concurrent Mark は Concurrent Mark From Roots フェーズと Concurrent Preclean フェーズとその関連処理から構成されるフェーズです。
- stringtable は、クラス名やメソッド名などを JVM 内で管理するためのテーブルに関する情報であり、Pause Remark で出力されます。
- ログに表れる GC フェーズは Remark、Cleanup の 2 種類です。GC 時間は、GC フェーズごとに出力されます。
- UUU、SSS、RRR は、GC 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。

【出力例】

説明の便宜のため、出力の冒頭に行番号を割り振っていますが、実際には出力されません。

```
1  [90.893s][info][gc             ] GC(45) Concurrent Cycle
2  [90.893s][info][gc,marking     ] GC(45) Concurrent Clear Claimed Marks
3  [90.893s][info][gc,marking     ] GC(45) Concurrent Clear Claimed Marks 0.054ms
4  [90.893s][info][gc,marking     ] GC(45) Concurrent Scan Root Regions
5  [90.899s][info][gc,marking     ] GC(45) Concurrent Scan Root Regions 5.549ms
6  [90.899s][info][gc,marking     ] GC(45) Concurrent Mark (90.899s)
7  [90.899s][info][gc,marking     ] GC(45) Concurrent Mark From Roots
8  [90.899s][info][gc,task        ] GC(45) Using 1 workers of 1 for marking
9  [91.875s][info][gc,marking     ] GC(45) Concurrent Mark From Roots 976.245ms
10 [91.875s][info][gc,marking     ] GC(45) Concurrent Preclean
11 [91.875s][info][gc,marking     ] GC(45) Concurrent Preclean 0.091ms
12 [91.875s][info][gc,marking     ] GC(45) Concurrent Mark (90.899s, 91.875s) 976.385ms
13 [91.876s][info][gc,start       ] GC(45) Pause Remark
14 [91.880s][info][gc,stringtable ] GC(45) Cleaned string and symbol table, strings: 13761 processed, 1 removed, symbols: 76801 processed, 1 removed
15 [91.880s][info][gc             ] GC(45) Pause Remark 1672M->1466M(2048M) 4.738ms
16 [91.880s][info][gc,cpu         ] GC(45) User=0.01s Sys=0.00s Real=0.00s
17 [91.880s][info][gc,marking     ] GC(45) Concurrent Rebuild Remembered Sets
18 [92.695s][info][gc,marking     ] GC(45) Concurrent Rebuild Remembered Sets 814.182ms
19 [92.695s][info][gc,start       ] GC(45) Pause Cleanup
20 [92.695s][info][gc             ] GC(45) Pause Cleanup 1467M->1467M(2048M) 0.552ms
21 [92.695s][info][gc,cpu         ] GC(45) User=0.00s Sys=0.00s Real=0.00s
22 [92.695s][info][gc,marking     ] GC(45) Concurrent Cleanup for Next Mark
23 [92.699s][info][gc,marking     ] GC(45) Concurrent Cleanup for Next Mark 3.935ms
24 [92.699s][info][gc             ] GC(45) Concurrent Cycle 1806.048ms
```


【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 90.893 秒後に、Concurrent Cycle の処理が開始されました。
- フェーズごとの処理時間は以下のとおりです。
    - Concurrent Clear Claimed Marks：0.054ms (行番号 3)
    - Concurrent Scan Root Regions：5.549ms (行番号 5)
    - Concurrent Mark From Roots：976.245ms (行番号 9)
    - Concurrent Preclean：0.091ms (行番号 11)
    - Pause Remark：4.738ms (行番号 15)
    - Concurrent Rebuild Remembered Sets：814.182ms (行番号 18)
    - Pause Cleanup：0.552ms (行番号 20)
    - Concurrent Cleanup for Next Mark：3.935ms (行番号 23)
- Pause Remark フェーズの処理時間 4.738ms と Pause Cleanup フェーズの処理時間 0.552ms の合計である 5.290ms はアプリケーションを停止しました。それ以外のフェーズはアプリケーションの処理と並列で GC が行われました。
- 「Concurrent Mark (90.899s, 91.875s) 976.385ms」のログは Concurrent Mark From Roots フェーズの処理時間と Concurrent Preclean の処理時間、その関連処理の処理時間の合計が 976.385ms であることを意味しています。
- Java ヒープ領域の使用量は、Pause Remark 前の 1672MB から、Pause Cleanup 後の 1467MB になり、Concurrent Cycle の期間は 1806.048ms でした。

2.3.3 Full GC

【出力形式】

太字の部分が可変情報で、それ以外の部分は固定情報です。

```
[タイムスタンプ][info][gc,task        ] GC(GC 通し番号) Using GC スレッド数 workers of 最大 GC スレッド数 for full compaction
[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 フェーズ名
[タイムスタンプ][info][gc,stringtable ] GC(GC 通し番号) Cleaned string and symbol table, strings: stringtable のエントリ数 processed, 削除した stringtable のエントリ数 removed, symbols: symbol table のエントリ数 processed, 削除した symbol table のエントリ数 removed
[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 フェーズ名 GC 時間
[タイムスタンプ][info][gc,heap        ] GC(GC 通し番号) 領域名: GC 直前の使用領域数->GC 直後の使用領域数(GC 直後の領域数)
[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 直前の Metaspace サイズ)->GC 直後の Metaspace 使用サイズ(GC 直後の Metaspace サイズ) NonClass: GC 直前の使用サイズ(クラス以外の情報を格納する Metaspace サイズ)->GC 直後の使用サイズ(クラス以外の情報を格納する Metaspace サイズ) Class: GC 直前の使用サイズ(クラスの情報を格納する Metaspace サイズ)->GC 直後の使用サイズ(クラスの情報を格納する 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(Java 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
```

- stringtable は、クラス名やメソッド名などを JVM 内で管理するためのテーブルに関する情報であり、GC フェーズ 1 で出力されます。
- GC 直後の領域数は、Old 領域と Humongous 領域に対しては表示されません。
- Metaspace の情報では、(GC 直後の Metaspace サイズ)の部分までしか出力されず、領域の内訳情報が出力されない場合があります。
- UUU、SSS、RRR は、GC 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。

【出力例】

```
[221.643s][info][gc,task        ] GC(202) Using 4 workers of 4 for full compaction
[221.643s][info][gc,start       ] GC(202) Pause Full (G1 Evacuation Pause)
[221.648s][info][gc,phases,start] GC(202) Phase 1: Mark live objects
[221.986s][info][gc,stringtable ] GC(202) Cleaned string and symbol table, strings: 14112 processed, 2143
removed, symbols: 76845 processed, 0 removed
[221.987s][info][gc,phases      ] GC(202) Phase 1: Mark live objects 338.759ms
[221.987s][info][gc,phases,start] GC(202) Phase 2: Prepare for compaction
[222.062s][info][gc,phases      ] GC(202) Phase 2: Prepare for compaction 75.822ms
[222.062s][info][gc,phases,start] GC(202) Phase 3: Adjust pointers
[222.225s][info][gc,phases      ] GC(202) Phase 3: Adjust pointers 162.466ms
[222.225s][info][gc,phases,start] GC(202) Phase 4: Compact heap
[222.305s][info][gc,phases      ] GC(202) Phase 4: Compact heap 79.639ms
[222.310s][info][gc,heap        ] GC(202) Eden regions: 0->0(363)
[222.310s][info][gc,heap        ] GC(202) Survivor regions: 0->0(13)
[222.310s][info][gc,heap        ] GC(202) Old regions: 1846->950
[222.310s][info][gc,heap        ] GC(202) Humongous regions: 202->158
[222.310s][info][gc,metaspace   ] GC(202) Metaspace: 27028K(28032K)->26110K(28032K) NonClass:
24159K(24832K)->23378K(24832K) Class: 2868K(3200K)->2731K(3200K)
[222.310s][info][gc             ] GC(202) Pause Full (G1 Evacuation Pause) 2047M->1105M(2048M) 667.417ms
[222.310s][info][gc,cpu         ] GC(202) User=2.52s Sys=0.01s Real=0.66s
```

【意味】

出力例のログからは以下の内容が読み取れます。

- Java ヒープ領域不足が原因で、202 回目の GC が発生しました。
- Full GC に利用可能な 4 つの GC スレッドのうち、4 つの GC スレッドを使用しました。
- GC フェーズごとの処理時間は以下のとおりです。
    - Mark live objects: 338.759ms
    - Prepare for compaction: 75.822ms
    - Adjust pointers: 162.466ms
    - Compact heap: 79.639ms
- 各領域の個数は以下のとおりです。
    - Eden 領域: GC 前の使用数=0、GC 後の使用数=0、GC 後に使用可能な数=363
    - Survivor 領域: GC 前の使用数=0、GC 後の使用数=0、GC 後に使用可能な数=13
    - Old 領域: GC 前の使用数=1846、GC 後の使用数=950
    - Humongous 領域: GC 前の使用数=202、GC 後の使用数=158
- Metaspace 領域の使用サイズは、27028KB から 26110KB になり、GC 後の Metaspace サイズは 28032KB です。
- Java ヒープ全体では、使用量が 2047MB から 1105MB になり、処理時間は 667.417ms でした。
- GC に要したユーザーCPU 時間は 2.52s、システム CPU 時間は 0.01s であり、実際の経過時間は 0.66s でした。

3. おわりに

今回は、G1 GC ログの読み方を紹介しました。GC に関する性能トラブルが発生している場合は、処理にかかった時間、GC 前後の Java ヒープの使用状況などをログから分析することで、チューニングやプログラムの修正に役立てることができます。

本記事を、G1 GC を用いたアプリケーションの安定稼働に向けて参考にしてください。