富士通 Java ブログ

Java メモリリークが原因の性能トラブルを分析する方法 ～ jcmd ツールの使い方 ～

2025 年 8 月 29 日 初版

桐山 卓弥

Java アプリケーションを実行していると、レスポンスが遅くなるなどの性能劣化のトラブルに直面することがあります。Java の性能劣化の要因は多岐にわたりますが、その中の 1 つに「メモリリーク」があります。
メモリリーク分析の方法には、jcmd でヒープダンプを採取する方法と JDK Flight Recorder（以降、JFR）を利用する方法があります。本稿では、メモリリークの概要について説明し、2 つの分析方法の特徴を紹介した後、jcmd による分析方法を説明します。JFR を使った方法については、別記事での解説を予定しています。

本稿では、「FUJITSU Software Enterprise Application Platform V2.0」に同梱している OpenJDK 21 を例に説明します。

1. Java のメモリリーク

開発者がメモリ管理を意識してコーディングする必要がある C や C++のような言語とは異なり、Java はガベージコレクション (以降、GC) が Java ヒープのメモリ領域を自動的に解放・再利用します。GC は、Java ヒープに存在する不要になったオブジェクトを自動的に回収する仕組みです。これにより、開発者は複雑なメモリ管理を意識せずにアプリケーションを開発することができます。

Java のメモリリークとは、アプリケーションによって参照を誤って保持し続けられたオブジェクトが GC によって回収されず、メモリ領域を占有し続ける状態を指します。GC はどこからも参照されなくなったオブジェクトを不要と判断して回収しますが、まだ参照されているオブジェクトは回収できません。メモリリークが放置されると、利用可能なメモリ領域が徐々に減少し、最終的には OutOfMemoryError (以降、OOME) を引き起こすことがあります。OOME が発生すると、アプリケーションが強制終了されるなど、システムの安定性を大きく損なうことになります。また、OOME に至らない場合でも、GC の実行頻度が増加し、アプリケーションのレスポンスが悪化する原因となります。

メモリリークの原因を特定するには、GC の対象にならず Java ヒープに残り続けているオブジェクトの参照関係を詳細に調査することが必要です。調査方法として伝統的に用いられてきたのが、ヒープダンプの解析です。ヒープダンプは、採取時点の Java ヒープ全体のメモリ状態をスナップショットとして取得するもので、どのオブジェクトがどこから参照されていてどれだけ Java ヒープに残っているか調べることができます。

しかし、ヒープダンプには採取時の負荷が大きい、時間経過による変化がわからないといった課題がありました。そこでメモリリーク分析の別の選択肢になるのが、新しいプロファイリングツールである JFR を使用したデータ採取です。JFR は OpenJDK に同梱されており、低い負荷で継続的にデータ採取できるという、ヒープダンプにはない利点があります。

次章では、最も簡単にヒープダンプを採取できる jcmd を使う方法と JFR を使ってデータ採取する方法の特徴について説明します。

2. jcmd と JFR の特徴

表 1 は jcmd でヒープダンプを採取する方法と JFR を使ってデータ採取する方法の特徴を比較したものです。

表 1: 分析方法の比較

方法	jcmd	JFR
アプリケーションへの負荷	高い	低い
出力ファイルサイズ	大きい	小さい
記録形式	ヒープダンプ	イベントと呼ばれる JFR 固有データ
記録内容	ヒープに含まれる 全オブジェクト、参照関係	一部のオブジェクト、参照関係、生成時のスタックトレース
記録タイミング	ヒープダンプ採取時点のみ	継続的な記録

JFR は低負荷で継続的にデータを採取できます。性能への影響を与えたくない本番環境でデータを採取したい場合は JFR を使った方法を選択して下さい。また、メモリリークがいつ起きるか予期できず、データ採取のタイミングがわからない場合も、JFR を使って長期的にデータを記録する方法が有効です。
jcmd でヒープダンプを採取するとアプリケーションに高負荷を与えますが、ヒープダンプを解析することで、ヒープの状況を詳細に調査することができます。負荷を気にせずに、メモリリークの原因となるオブジェクトの詳細な参照関係を調べたい場合は、jcmd でヒープダンプを採取する分析方法を選択してください。

なお、JDK8 以前は、ヒープダンプを採取するために jmap が使われることもありましたが、現在は jcmd に統合されており、こちらを使うことが推奨されています。詳細は［Enterprise Application Platform でのトラブルシューティング技法（第 1 回）：OpenJDK 11 の JDK ツールの概要を知る］の記事を参照してください。

次章では、jcmd によるメモリリーク分析の手順を説明します。JFR を使う手順については、別記事での説明を予定しています。

3. jcmd によるメモリリーク分析

本章では、メモリリークを意図的に引き起こすコードを例に、jcmd を使ってメモリリーク分析する手順について説明します。
メモリリークの原因となっているオブジェクトへの参照元を特定するために、jcmd を使用してクラスヒストグラムとヒープダンプを採取して分析します。

以下は、オブジェクトへの参照を誤って保持し続けてメモリリークを意図的に引き起こすコードの例です。

```
 1 package com.fujitsu.demo;
 2
 3 import java.util.ArrayList;
 4 import java.util.List;
 5
 6 public class MemLeak {
 7     List objects = new ArrayList<>();
 8
 9     public static void main(String[] args) {
10         MemLeak memLeak = new MemLeak();
11         memLeak.leak();
12     }
13
14     private void leak() {
15         while (true) {
16             LeakingObject o = new LeakingObject();
17             objects.add(new LeakingObject());
18             try {
19                 Thread.sleep(100);
20             } catch (InterruptedException e) {
21                 e.printStackTrace();
```



```
            }
        }
    }
}

class LeakingObject {
    public byte[] largeArray;
    private String data;

    public LeakingObject() {
        largeArray = new byte[1024];
        data = "This is LeakingObject";
    }
}
```

MemLeak クラスの leak メソッドでは、LeakingObject のインスタンスを無限ループ内で生成し、ArrayList クラスのインスタンスである objects に追加しています。ArrayList はリストに含まれる要素への参照を保持するため、objects に追加された LeakingObject クラスのインスタンスへの参照が残り続け、GC の回収対象となりません。その結果、ヒープメモリが徐々に消費され、最終的には OutOfMemoryError が発生する可能性があります。

このメモリリークを解決するためには、次の手順で分析する必要があります。

1. リークしているクラスが LeakingObject であることを特定
2. ArrayList クラスのインスタンスである objects が LeakingObject クラスのインスタンスへの参照を保持し続けていることを特定

「3.1. リーククラスの特定」、「3.2. オブジェクトの参照関係の追跡」で手順の詳細について説明します。

3.1. リーククラスの特定

Java ヒープでオブジェクトの数とメモリサイズが継続的に増加しているクラスを調べることで、リークしているクラスを特定できます。そのためには、時間をおいてクラスヒストグラムを複数回取得し、各クラスのオブジェクトの数とメモリサイズの増減を調べます。採取する間隔はアプリケーションによりますが、複数回の FullGC が発生する程度の時間を目途に調整してください。

以下のコマンドを実行することで、Java ヒープのクラスヒストグラムを出力できます。

```
jcmd <プロセス ID> GC.class_histogram
```

以下にクラスヒストグラムの出力例を示します。各行にクラスごとのオブジェクトが表示され、"#instances"列にはオブジェクト数が、"#bytes"列にはオブジェクトが使用しているメモリサイズが表示されます。クラスヒストグラムはオブジェクトが使用しているメモリサイズの順で表示されます。

1 回目に採取したヒストグラム

```
  num     #instances         #bytes  class name (module)
-------------------------------------------------------
   1:         13835        1168184  [B (java.base@21.0.5)
   2:         12655         303720  java.lang.String (java.base@21.0.5)
   3:          2309         280920  java.lang.Class (java.base@21.0.5)
   4:          2490         169784  [Ljava.lang.Object; (java.base@21.0.5)
～
  28:           335           8040  com.fujitsu.demo.LeakingObject
  29:           114           7296  java.util.concurrent.ConcurrentHashMap (java.base@21.0.5)
  30:           167           6680  java.lang.invoke.DirectMethodHandle (java.base@21.0.5)
```


2 回目に採取したヒストグラム

```
num     #instances         #bytes  class name (module)
-------------------------------------------------------
   1:         19656        7442056  [B (java.base@21.0.5)
   2:         12451         298824  java.lang.String (java.base@21.0.5)
   3:          2279         277360  java.lang.Class (java.base@21.0.5)
   4:          2362         202216  [Ljava.lang.Object; (java.base@21.0.5)
   5:          6377         153048  com.fujitsu.demo.LeakingObject
～
```

この例では、"byte"配列を示す"[B"と"com.fujitsu.demo.LeakingObject"のオブジェクト数とメモリサイズが増加していることがわかるため、リークしている可能性があります。次に、リークの原因となっている参照元を特定するために参照関係を調べます。

3.2. オブジェクトの参照関係の追跡

オブジェクトの参照を調べるために、実行中のアプリケーションのヒープダンプを採取します。

以下のコマンドを実行することで、ヒープダンプファイルを採取できます。

```
jcmd <プロセス ID> GC.heap_dump <ファイル名>
```

採取したヒープダンプを解析するためのツールは複数ありますが、本稿では、JDK Mission Control(以降、JMC)を使います。[Eclipse Mission Control（「Eclipse Adoptium」のページ）]から、使用する環境に合わせた JMC をダウンロードして展開します。 本稿では、バージョン 8.3.0 を使用して説明します。

Windows 環境で使用する場合は、以下の手順で JMC を起動します。

（手順 1）JMC を起動する Java を環境変数に設定

```
jcmd <プロセス ID> GC.heap_dump <ファイル名>
```

（手順 2）jmc.exe を実行

```
<JMC のインストールディレクトリ>¥JDK Mission Control¥jmc.exe
```

JMC でヒープダンプを読み込むときは、JMC を起動後、「ファイル」メニューの「ファイルを開く」から解析したいファイルを指定してください。ヒープダンプファイルを開くと、「JOverflow」ウインドウが開かれます。

オブジェクトの参照関係の追跡は、次の「調査するオブジェクトを選別する」、「選択したオブジェクトの参照元を調べる」、「詳細な参照関係を調べる」の順で実施します。

調査するオブジェクトを選別する

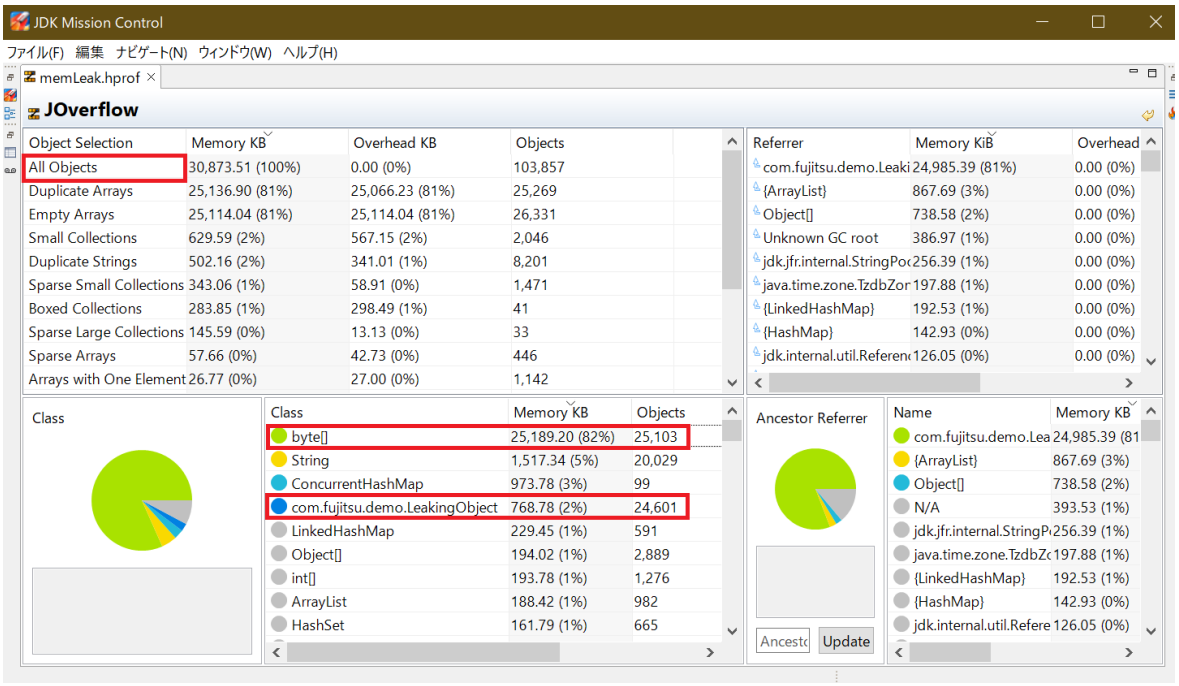

図 1 ヒープダンプに含まれるオブジェクト

左ウインドウの「All Objects」を選択することで、ヒープダンプに含まれるすべてのオブジェクトを表示できます。 参照関係を調べるオブジェクトの候補は、「3.1 リーククラスの特定」で特定したリーククラスのオブジェクトです。図 1 の例では、"byte¥[]"のオブジェクトが 25,103 個残存し、Java ヒープの 82%と大部分を占めていることがわかります。
また、"com.fujitsu.demo.LeakingObject"のオブジェクトが 24,601 個残存しています。クラスヒストグラムで継続的に増加していることを確認したオブジェクトが、ヒープダンプで Java ヒープの大部分を占めていることがわかりました。これらのオブジェクトが Java ヒープに多く残っている原因を特定するためにオブジェクトの参照元を調べます。

選択したオブジェクトの参照元を調べる

図 2 のように、表示されたオブジェクトから"byte¥[]"を選択すると、右下のウインドウに選択したオブジェクトの参照元の一覧が表示されます。参照元の一覧には以下が表示されます。

- オブジェクトへの参照を保持する変数
- オブジェクトへの参照を要素に持つ配列およびコレクション[（注 1）]

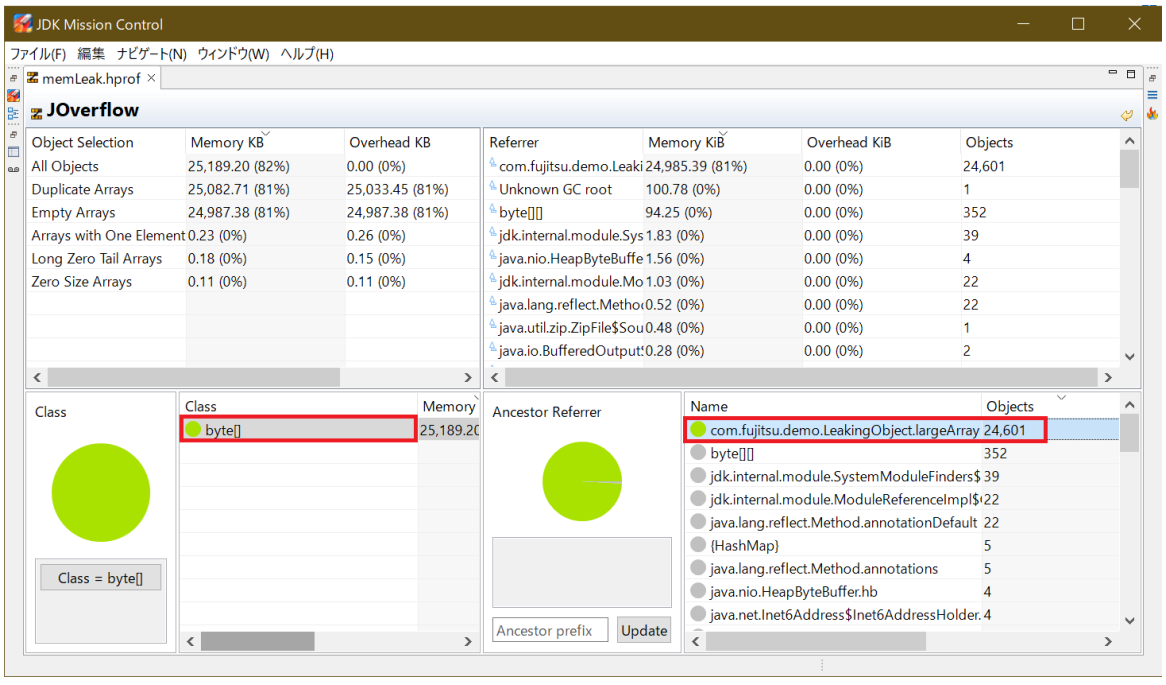

図 2 ヒープダンプに含まれる byte 配列の参照元

図 2 の例では、"byte¥[]"を選択した結果、"com.fujitsu.demo.LeakingObject.largeArray"から最も多くの 24,601 個の"byte¥[]"のオブジェクトが参照されていることがわかります。

（注 1）コレクションとは、複数のオブジェクトを格納、管理するためのフレームワークです。java.util.Collection の実装クラスとして提供されます。java.util.ArrayList などが該当します。

詳細な参照関係を調べる

次に、"byte¥[]"のオブジェクトを最も多く参照していた"com.fujitsu.demo.LeakingObject.largeArray"の参照元を調べます。"largeArray"はさらに別の変数や配列、コレクションから連鎖的に参照されている可能性があるからです。 図 2 の右下のウインドウの一覧から"com.fujitsu.demo.LeakingObject.largeArray"を選択することで、右上のウインドウに参照の連鎖が表示されます。

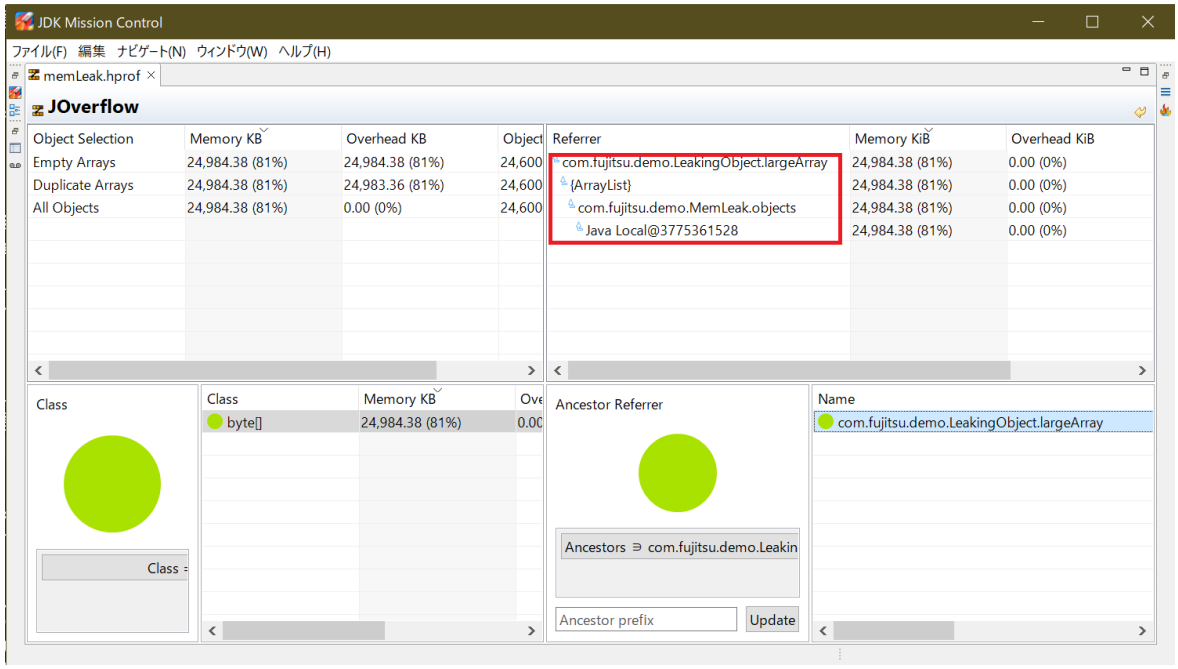

図 3 ヒープダンプに含まれる LeakingObject の参照元

図 3 の例では、"com.fujitsu.demo.LeakingObject.largeArray"への参照として、"{ArrayList}"が表示されています。これは、"java.util.ArrayList"を意味する"{ArrayList}"の要素として、参照が保持されていることを示します。
"{ArrayList}"への参照として、"com.fujitsu.demo.MemLeak.objects"が表示されています。これは、MemLeak クラスに属する objects フィールドが参照を保持していることを示します。 "com.fujitsu.demo.MemLeak.objects"への参照として、"Java Local"が表示されています。"Java Local"はローカル変数が参照を保持していることを示しますが、どのローカル変数を示しているかはヒープダンプだけではわからないので、コードを確認します。「1. Java のメモリリーク」のコードを確認すると、コードの 10 行目、MemLeak クラスの main メソッドで作成されたローカル変数 memLeak に MemLeak クラスのオブジェクトが格納されています。

以上から、"byte¥[]"のオブジェクトを最も多く参照している"com.fujitsu.demo.LeakingObject.largeArray"の参照をたどると、MemLeak クラスの main メソッドで作成されたローカル変数から参照されていることがわかりました。

クラスヒストグラムとヒープダンプを採取する時の注意

jcmd でクラスヒストグラムまたはヒープダンプを採取する時に、コマンドラインオプションに"-all"を指定しないでください。"-all"を指定すると、採取する前に GC が実行されなくなります。

おわりに

OpenJDK には、トラブルシューティングに役立つ便利なツールが豊富に用意されています。本稿では、Java アプリケーションにおけるメモリリークに焦点を当て、OpenJDK に付属している jcmd と外部ツール JMC を使って分析する手順を解説しました。本記事を安定稼働の参考にしてください。

次回は、JFR を使った分析方法を紹介します。