Java モニタリング／プロファイリング

内容

Java モニタリング／プロファイリング

はじめに

システムの開発・運用の要求仕様として提示されたレスポンスタイムやスループット等の性能がでなかったり、改修や取扱データの変化により性能劣化することがあります。そういったときにモニタリング（観察）とプロファイリング（分析）が必要になります。ネットワーク上に構築されたシステムでは以下の構成要素を観察し、ボトルネックを見つけて原因等の分析を行います。

・ネットワーク機器（スィッチ／ルータ／負荷分散装置）
・SAN で接続されたストレージ
・DB サーバの機器とミドルウェア
・（複数の）AP サーバの機器（NIC、ディスク、OS）とアプリケーション

上記のシステム構成要素のうち、AP サーバ以外はそれぞれのコマンドや監視ソフトが必要になります。アプリケーション（特に自前で開発したもの）は独自のプロファイリングが必要になります。

1. アプリケーションの監視

Java 製の〔Web アプリケーション／Web サービス〕（以下 Web アプリ）はしばしばガベージコレクション（Garbage Collection…以下 GC）に関係するプロファイリングが必要になります。

Java は C++に類似した文法で設計されていますが「メモリへの直接アクセス」はできず、GC によるメモリ再利用を行います。GC は（JVM の実行時パラメータによりますが）以下のようにオブジェクトを管理します[1]。

① オブジェクトを若い世代（インスタンス化したばかり）、古い世代（GC 生き残り）に分け、順次、若い世代から古い世代に移す
② 若い世代の使用率が閾値を超えたら、若い世代の GC 及び連続領域の確保を行う
③ 若い世代の GC だけで十分な空きが得られないとき、古い世代を含めたヒープ全体の GC(Full GC)を行う

問題は主に以下の 2 段階で発生します。頻発するようであればプロファイリングし、対策を行います。

i Full GC でシステム停止状態（Stop-The-World）になる
（Java 9 以降のデフォルト G1GC では長時間の停止は起きづらいようです）
ii GC で解放されない領域が増加（メモリリーク）してヒープが確保できなくなり OutOfMemoryError が発生する

[1] https://docs.oracle.com/javase/jp/8/docs/technotes/guides/vm/gctuning/introduction.html

2. ツール

JVM の状態を監視するためのツールがいくつか用意されています。

2.1. VisualVM（オールインワン Java トラブルシューティングツール）

VisualVM は JDK Version 6 Update 7 ～ Version 8 に標準で同梱されていたツールです。Java のオープン化に伴い JDK から外され、独立したサイトで開発・管理されています。

ツールのサイトには、『VisualVM は、コマンドライン JDK ツールと軽量プロファイリング機能を統合したビジュアルツールです。』と書かれています。

https://visualvm.github.io/index.html

2.2. JDK Mission Control(JMC)

Mission Control は、Java 用のオープンソースの実稼働時間プロファイリングおよび診断ツールです。始まりは Appeal Virtual Machines 社／BEASystems 社が作った JRockitJVM で動作するツールでしたが、Oracle に買収され、同じく Oracle に買収された SunMicrosystems 社の HotSpotVM に JRockitJVM を統合した Java 7 以降、各種の JDK(AdoptOpenJDK, Oracle, Red Hat その他)で動作するようになっています。

https://github.com/openjdk/jmc

2.3. jcmd

JMC 同様に JRockit 向けに提供されていたコマンドです。類似機能を持つコマンドに jps、jstat, jstack、jinfo および jmap がありますが、Oracle のドキュメントでは jcmd の方を使うように推奨しています[2]。JDK 7 以降の JDK に同梱されています。

2.4. JConsole, jps, jstat

JConsole JMX 準拠のグラフィカルツール。ローカル JVM とリモート JVM の両方を監視できます。アプリケーションの監視と管理を行うこともできます

jps JVM プロセスステータスツール －HotSpot Java 仮想マシン[3]の一覧を表示します。

jstat JVM 統計データ監視ツール - HotSpot Java 仮想マシンに接続し、パフォーマンス統計データを収集および記録します。

※jps, jstat,は Java 6 で試験的・非推奨になっていますが、Java 16 では有用なツールとして紹介[4]されています。

---

[2] https://docs.oracle.com/javase/jp/8/docs/technotes/guides/troubleshoot/tooldescr025.html

[3] Java11 では、“OpenJ9”(IBM/Eclipse)等の明示がない JVM は HotSpot です

[4] https://docs.oracle.com/javase/jp/16/troubleshoot/diagnostic-tools.html#GUID-FC269C18-470F-441E-9564-7EEA182F8125

3. モニタリング

本番相当のデータを使った試験や本番運用時に発生する障害の種類は色々ありますが、現象の多くは CPU やメモリのボトルネックとして感知されます。初動は JVM を観察してボトルネックの発生個所を確認します。以下の例は Linux で実行していますが、Windows のコマンドプロンプトでもパス表示形式を除き同様の結果が得られます。実行例で使用した JDK は以下のものです。

＜Linux＞

```
root@f9730c0e1ea0:/# java -version
openjdk version "11.0.12" 2021-07-20
OpenJDK Runtime Environment 18.9 (build 11.0.12+7)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.12+7, mixed mode, sharing)
```

＜Windows＞

```
C:¥Users¥User>java -version
openjdk version "11.0.10" 2021-01-19
OpenJDK Runtime Environment AdoptOpenJDK (build 11.0.10+9)
OpenJDK 64-Bit Server VM AdoptOpenJDK (build 11.0.10+9, mixed mode)
```

3.1. 稼働中 JVM の一覧と各実行パラメータを表示

稼働中 JVM の一覧を表示し、監視対象のプロセスを特定します。

（1）jps -lv

＜実行例①＞途中行末は省略しています

```
root@f9730c0e1ea0:/# jps -lv
1 jdk.jshell/jdk.internal.jshell.tool.JShellToolProvider -Dapplication.home=/us
26 jdk.jshell.execution.RemoteExecutionControl -agentlib:jdwp=transport=dt_sock
109 jdk.jcmd/sun.tools.jps.Jps -Dapplication.home=/usr/local/openjdk-11 -Xms8m
```

※行先頭の数値がプロセスID(pid)

（2）jcmd -l

＜実行例②＞

```
root@f9730c0e1ea0:/# jcmd -l
128 jdk.jcmd/sun.tools.jcmd.JCmd -l
1 jdk.jshell/jdk.internal.jshell.tool.JShellToolProvider
26 jdk.jshell.execution.RemoteExecutionControl 46005
```

※行先頭の数値がプロセスID(pid)

3.2. 稼働中 JVM をリアルタイム監視

（1）jstat -gccause -t -h 20 704 100ms

-gccause :GC の内容を表示します

-t :タイムスタンプ（経過時間）を表示します

-h nn :nn 行毎に見出し行を出力します

pid :jps で出力した pid の一覧から監視対象を指定します

ms :表示間隔…監視対象が終了するか、ctrl＋ c 押下まで指定ミリ秒間隔で出力を続けます

＜実行例＞

jstat -gccause -t -h 20 748 100ms

```
Timestamp         S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT    CGC    CGCT     GCT    LGCC                 GCC
         458.7   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         458.8   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         458.9   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         459.0   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         459.1   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         459.2   0.00 100.00  81.67   1.59  34.50  37.86     69    0.519     0    0.000     0    0.000    0.519 G1 Evacuation Pause  No GC
         459.3   0.00 100.00  95.00   1.59  34.50  37.86     70    0.519     0    0.000     0    0.000    0.519 No GC                G1 Evacuation Pause
         459.4   0.00 100.00   0.00   1.59  34.50  37.86     70    0.525     0    0.000     0    0.000    0.525 G1 Evacuation Pause  No GC
         459.5   0.00 100.00   0.00   1.59  34.50  37.86     70    0.525     0    0.000     0    0.000    0.525 G1 Evacuation Pause  No GC
（中略）
Timestamp         S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT    CGC    CGCT     GCT    LGCC                 GCC
        3305.3   0.00 100.00   4.35  99.94  34.28  34.22  17179   51.465     0    0.000 34069   59.437  110.902 G1 Humongous Allocation No GC
        3305.4   0.00 100.00  14.29  99.94  34.28  34.22  17182   51.473     0    0.000 34075   59.447  110.920 G1 Humongous Allocation No GC
        3305.5   0.00 100.00   4.35  99.94  34.28  34.22  17185   51.485     0    0.000 34081   59.458  110.943 G1 Humongous Allocation No GC
        3305.6   0.00 100.00   9.09  99.95  34.28  34.22  17189   51.493     0    0.000 34086   59.466  110.958 G1 Humongous Allocation No GC
        3305.7   0.00 100.00   6.25  99.95  34.28  34.22  17195   51.511     0    0.000 34092   59.478  110.989 G1 Humongous Allocation No GC
        3305.8   0.00 100.00  16.67  99.95  34.28  34.22  17204   51.527     0    0.000 34098   59.486  111.012 G1 Humongous Allocation No GC
        3305.9   0.00   0.00   0.00  99.96  34.28  34.22  17220   51.557     2    0.022 34100   59.488  111.067 G1 Humongous Allocation No GC
        3306.0   0.00   0.00   0.00  99.99  34.28  34.22  17229   51.578     8    0.078 34100   59.488  111.144 No GC              G1 Humongous Allocation
        3306.1   0.00   0.00   0.00  99.99  34.28  34.22  17242   51.597    16    0.162 34100   59.488  111.247 No GC              G1 Humongous Allocation
        3306.2   0.00   0.00   0.00  99.96  34.28  34.22  17247   51.602    20    0.206 34100   59.488  111.296 G1 Humongous Allocation No GC
        3306.3   0.00   0.00   0.00  99.96  34.28  34.22  17247   51.602    20    0.206 34100   59.488  111.296 G1 Humongous Allocation No GC
```

※jstat のタイムスタンプ 3306.3 近辺で監視対象の JVM が出力したログ…FullGC でヒープの空きが確保できずに、OutOfMemoryError の発生を記録

```
|  Exception java.lang.OutOfMemoryError: Java heap space
|        at Unsafe.allocateUninitializedArray (Unsafe.java:1269)
|        at StringConcatFactory$MethodHandleInlineCopyStrategy.newArray (StringConcatFactory.java:1633)
|        at DirectMethodHandle$Holder.invokeStatic (DirectMethodHandle$Holder)
|        at LambdaForm$MH/0x0000000840084840.invoke (LambdaForm$MH)
|        at LambdaForm$MH/0x0000000840085440.invoke (LambdaForm$MH)
|        at Invokers$Holder.linkToTargetMethod (Invokers$Holder)
|        at (#4:2)
```

＜出力内容＞

Java11 はデフォルトの GC が G1 と呼ばれる方式です。(Java 9 以降のデフォルトがこの G1GC)

それ以前の CMS GC ではヒープのメモリを若い世代（Eden、Survivor 0、Survivor 1）と、古い(OLD)世代（Tenured)、メタデータ（メソッド領域等の変化が起こらい情報）に分けて管理しており、jstat の出力もこの単位になっています。

●ガベージ・コレクション統計データのサマリー

S0: Survivor 領域 0 の使用率(現在の容量に対するパーセンテージ)。

S1: Survivor 領域 1 の使用率(現在の容量に対するパーセンテージ)。

E: Eden 領域の使用率(現在の容量に対するパーセンテージ)。

O: Old 領域の使用率(現在の容量に対するパーセンテージ)。

M: メタスペースの使用率(現在の容量に対するパーセンテージ)。

CCS: 圧縮されたクラス領域の使用率(パーセンテージ)。

YGC: Young 世代の GC イベントの数。

YGCT: Young 世代のガベージ・コレクション時間。

FGC: フル GC イベントの数。

FGCT: フル・ガベージ・コレクションの時間。

GCT: ガベージ・コレクションの総時間。

LGCC: 最後のガベージ・コレクションの原因

GCC: 現在のガベージ・コレクションの原因

G1GC はヒープをリージョン単位で管理し、世代を論理セットで認識するようにしました。また、再利用可能な領域の回収をアプリケーションの実行と並行して行うため停止時間は気にしなくてもよくなっています。注意が必要なのは、以下の点です[5]。

i GC が発生したときに、Old 領域の使用率が下がっているか（十分な解放領域があるか）

ii GCC に“Humongous Allocation”（大型オブジェクトの割り当て）が頻発していないか（LGCC は無視）⇒ 頻発するならリージョンサイズの拡張を検討

iii FGC が発生していないか（メモリリークがないかプロファイリングが必要）

iv 確保できるメモリの量がオプション-Xmx で指定した最大ヒープ・サイズより小さい場合、Xmx の値に達する前に OutOfMemoryError になる ⇒ OS のコマンドでメモリの状態を確認

```
Timestamp         S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT    CGC    CGCT     GCT    LGCC                 GCC
          573.7   0.00 100.00  12.50  99.95  33.30  33.33   8993   32.832    49    2.640 17461   33.650   69.122 No GC                G1 Humongous Allocation
          573.8   0.00 100.00  12.50  99.96  33.30  33.33   8995   32.844    49    2.640 17466   33.656   69.140 G1 Humongous Allocation No GC
          573.9   0.00 100.00   8.33  99.96  33.30  33.33   8998   32.859    49    2.640 17471   33.665   69.164 G1 Humongous Allocation No GC
          574.0   0.00 100.00   0.00  99.96  33.30  33.33   9001   32.870    49    2.640 17477   33.678   69.189 G1 Humongous Allocation No GC
          574.1   0.00 100.00   0.32   7.87  33.30  33.33   9003   32.875    49    2.640 17482   33.687   69.202 G1 Humongous Allocation No GC
```

（２）jcmd <pid|main-class> PerfCounter.print

コマンド実行時の情報が出力されますが、jstat のような連続での出力はできません。

＜実行例＞

```
root@f9730c0e1ea0:/# jcmd 27 PerfCounter.print
27:
java.ci.totalTime=183880800
java.cls.loadedClasses=160
java.cls.sharedLoadedClasses=714
java.cls.sharedUnloadedClasses=0
java.cls.unloadedClasses=0
java.property.java.class.path=”.”
java.property.java.home=”/usr/local/openjdk-11”
java.property.java.library.path=”/usr/java/packages/lib:/usr/lib64:/lib64:/lib:/usr/lib”
java.property.java.version=”11.0.12”
java.property.java.vm.info=”mixed mode, sharing”
java.property.java.vm.name=”OpenJDK 64-Bit Server VM”
（略）
sun.gc.cause=”No GC”
sun.gc.collector.0.invocations=0
sun.gc.collector.0.lastEntryTime=0
sun.gc.collector.0.lastExitTime=0
sun.gc.collector.0.name=”G1 incremental collections”
sun.gc.collector.0.time=0
sun.gc.collector.1.invocations=0
sun.gc.collector.1.lastEntryTime=0
sun.gc.collector.1.lastExitTime=0
sun.gc.collector.1.name=”G1 stop-the-world full collections”
sun.gc.collector.1.time=0
sun.gc.collector.2.invocations=0
sun.gc.collector.2.lastEntryTime=0
（以下略）
```

[5] 公式なチューニングガイドは下記 URL
https://docs.oracle.com/javase/jp/11/gctuning/introduction-garbage-collection-tuning.html

4. プロファイリング

JVM の動作に異常があるときは、プロファイリングツールを使い起因となるクラスを探します。

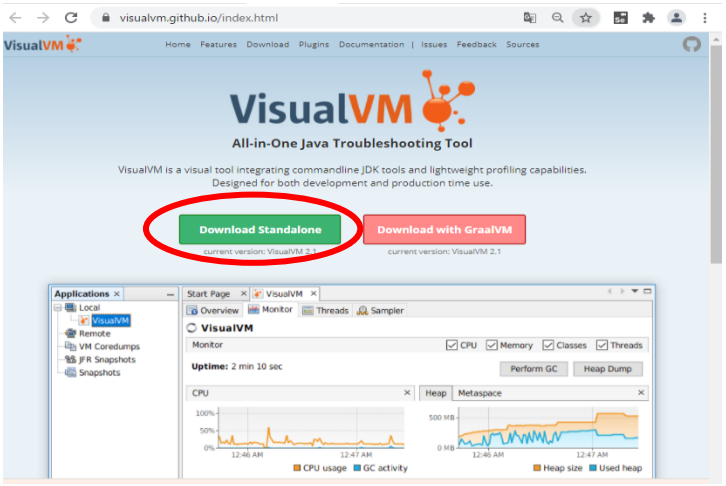
4.1. VisualVM

以下、VisualVM バージョン 2.1 で実行した内容です。

（1）VisualVM のダウンロード

https://visualvm.github.io/index.html

[Download Standalone]ボタンのリンク先から Zip ファイルのダウンロードができます。



解凍するだけで動作します。

※以降の説明は、次の path に解凍したものとします。

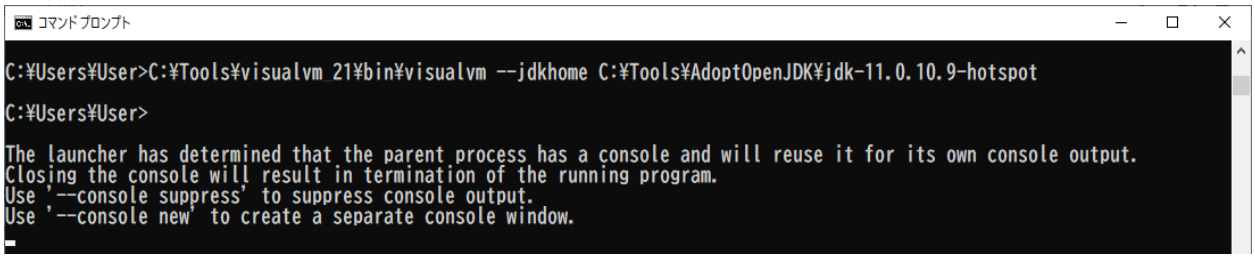
＜解凍先例＞ C:¥Tools¥visualvm_21

（2）起動

解凍した中の bin フォルダに入っている visualvm を実行します。パラメータには JDK の bin フォルダの直上のフォルダを指定してください。（パスに空白を含む場合は「"」でパスを囲みます）

＜コマンド例＞

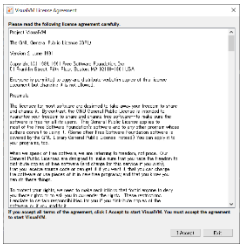
```
C:¥Tools¥visualvm_21¥bin¥visualvm --jdkhome C:¥Tools¥AdoptOpenJDK¥jdk-11.0.10.9-hotspot
```



※パラメータの JDK パスは、以下のように環境定義ファイルでも指定できます

etc¥visualvm.conf の visualvm_jdkhome=”C:¥Tools¥AdoptOpenJDK¥jdk-11.0.10.9-hotspot”

（3）ライセンス（初回起動時のみ）



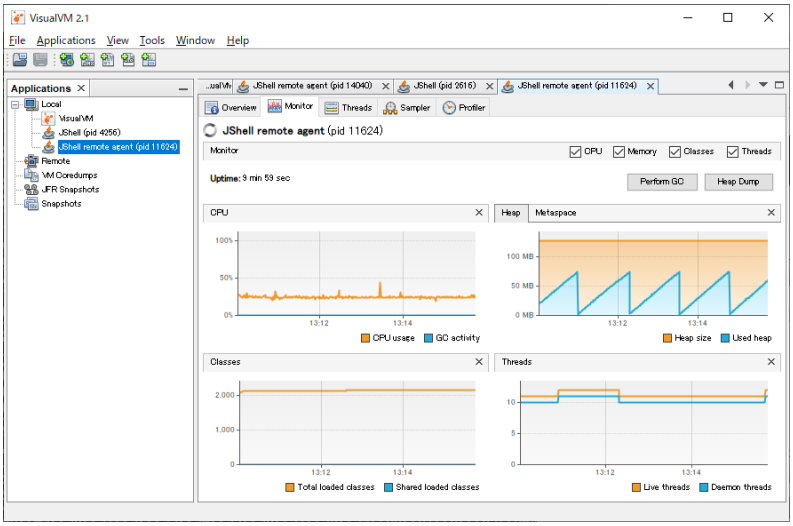
[I Accept]ボタンを押下してください。

（4）モニタリング

① 正常な実行状態

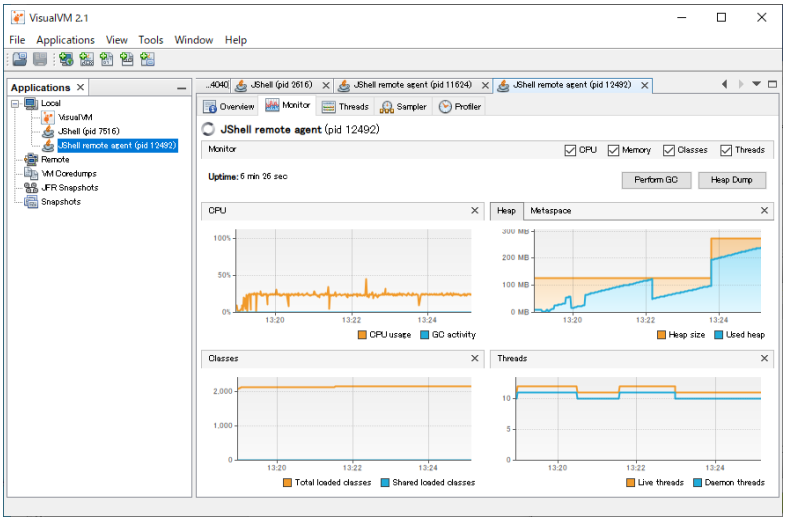
この例は GC の発生で Heap の表示が鋸状に上下していますが、上下が一定の幅に収まっており問題の兆候はありません。実運用の Web アプリは集中・分散するリクエストを並列処理するためきれいな波にはなりませんが、波の増大傾向がなければ問題ありません。

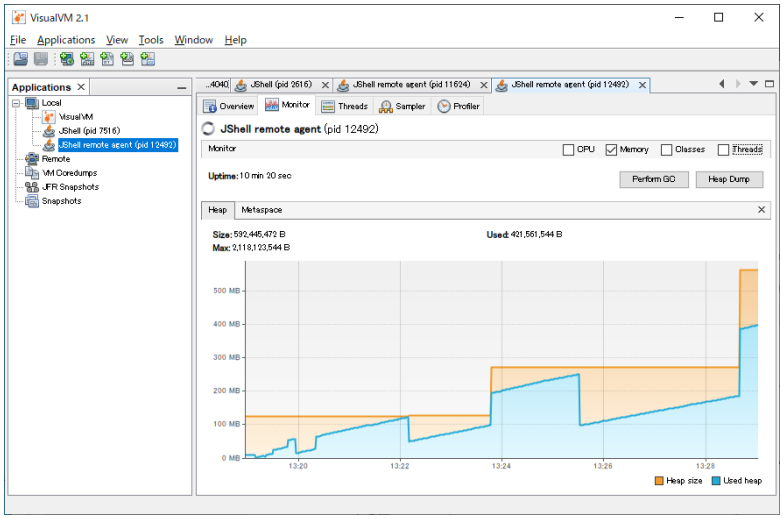
Classes の欄でロードされているクラスが増えた場合は同期して Used Heap が上がりますが、それも問題ありません。



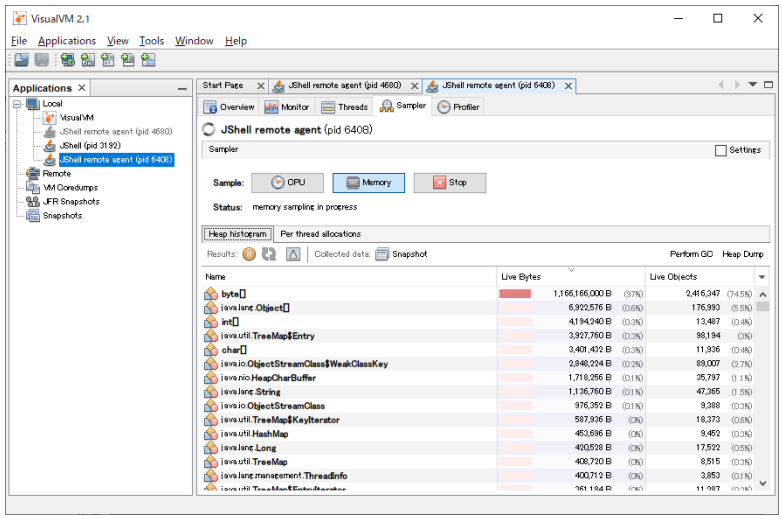
② 異常な実行状態

Classes のグラフに上昇がないのに Used Heap が右肩上がりになっています。GC で解放されずメモリリークしている可能性があります。





Heap Size がどんどん切り上がっているので、JVM のメモリ使用量が増加しています。上限に達すると OutOfMemoryError が発生します。



Sampler タブで Memory を選択すると、メモリを消費しているクラスを確認できます。

（５）実行状況の確認例

JShell を使って以下の内容の MemoryBiter クラスを作り、実行します。実行内容は...

㋐ List 型のインスタンス変数 strArr を作る

㋑ String の連結で新しいインスタンスを発生させる…このとき文字列を長くしていく

㋒ ㋐で作った strArr に㋑で作ったインスタンス（の参照）を追加していく

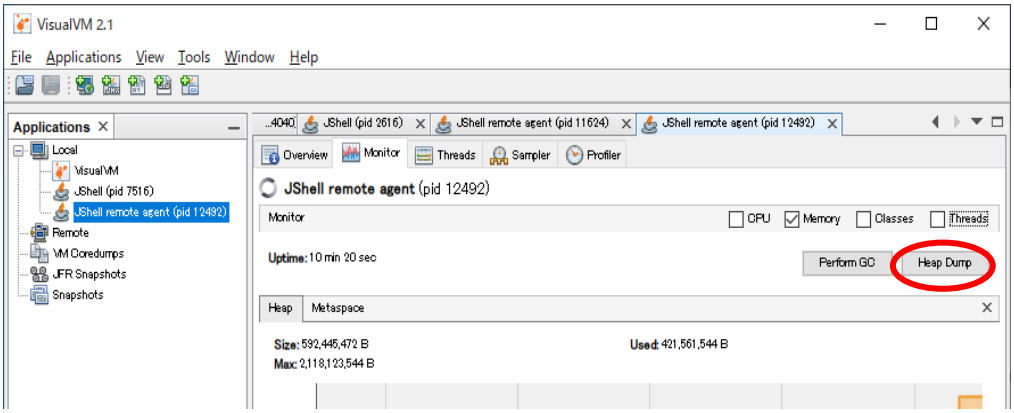
㋓ 300 ループに 1 回 strArr を割り当て直し、格納されていたインスタンスを GC の対象にする

```
public class MemoryBiter {
    List<String> strArr = new ArrayList<String>();//㋐
    String str = "0".repeat(1000);
    public void biting() {
        for (long i=1; true; i++) {
            str = str + Long.toString(i);//㋑
            strArr.add(str);//㋒
            if(i%300==1) strArr = new ArrayList<String>();//㋓
        }
    }
}
(new MemoryBiter()).biting();
```


① メモリリークの調査（ヒープダンプの取得）

[Monitor]画面の右上にある[Heap Dump]ボタンを押下します。

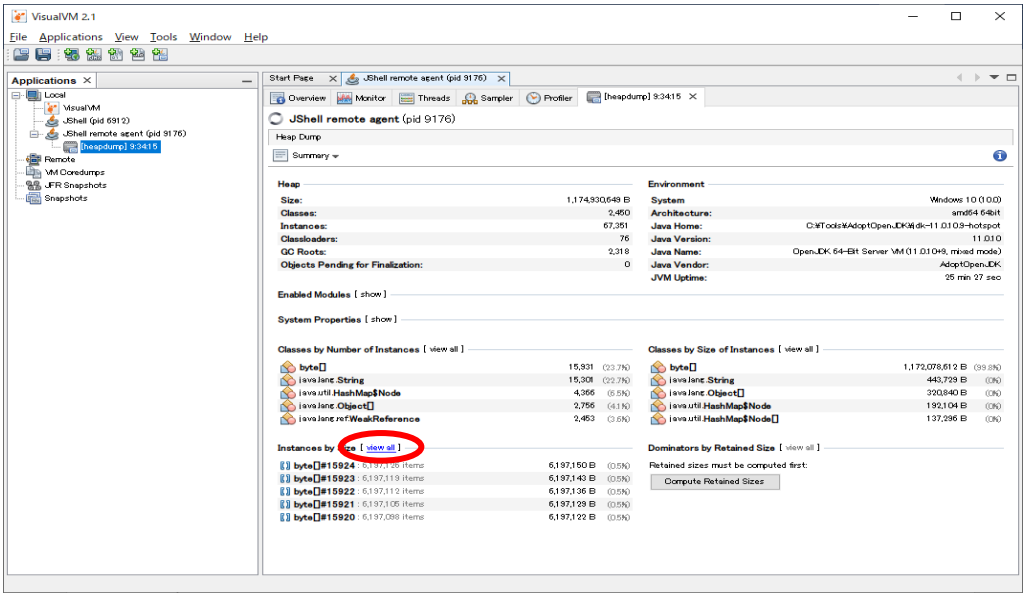
※ヒープダンプはメモリの内容を全てファイルに落とす間 JVM が動作を停止します。VisualVM はネットワークの先の JVM に接続できますが、ローカル環境で実行した方がよいでしょう。



② メモリ消費が大きなインスタンス

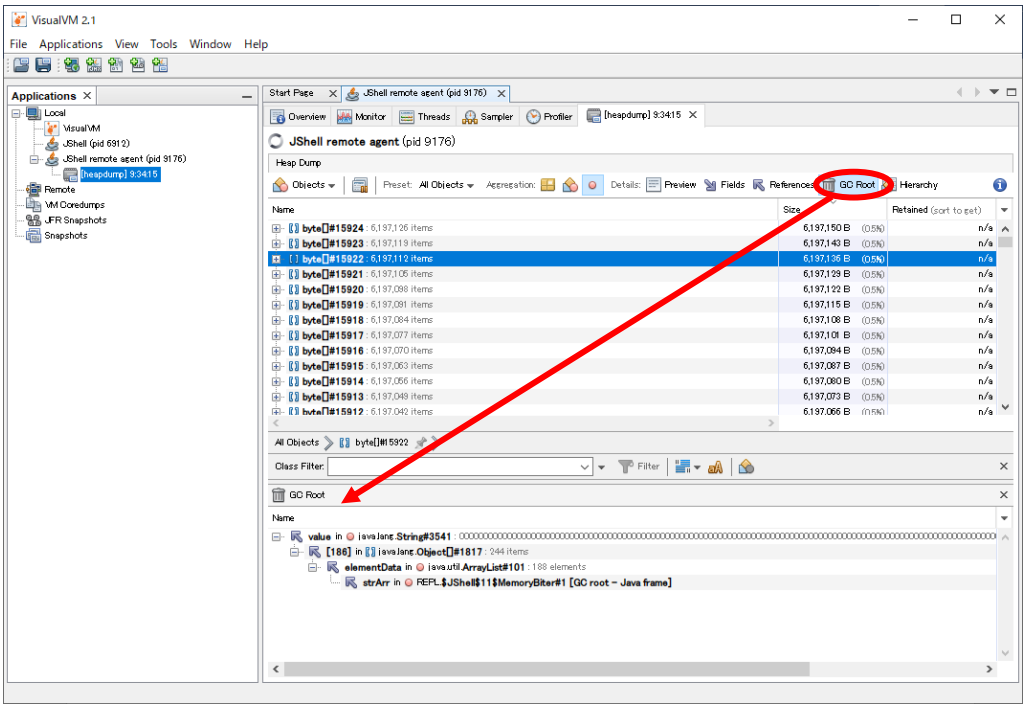
画面左の[Applications]からヒープダンプを開くと、Summary が表示されます。

●Summary の左下”Instances by Size [view all]”の viw all をクリックします。



③ インスタンス（オブジェクト）の作成元

Objects の一覧画面で［GC Root］を押下すると、選択されたオブジェクトの GC Root[6]からの一連の参照が階層で表示されます。この一番下に出ているのがオブジェクトの作成元です。



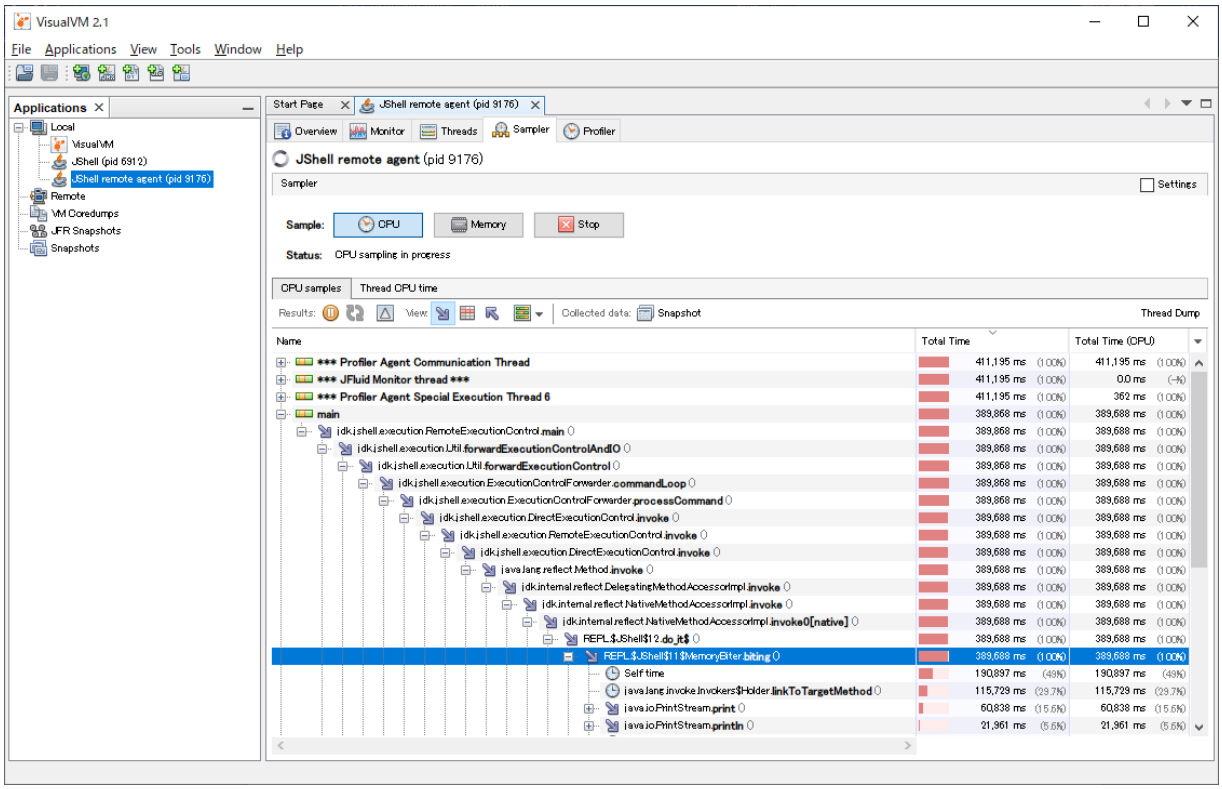
〔結果〕MemoryBiter クラスのフィールド strArr がインスタンスの参照を持っており、メモリリークの候補です。GC ルートが表示されないインスタンスは GC の対象（リークではない）になります。

[6] GC ルートは GC を行うときに参照が残っているか否かを調べるときの起点で、ヒープの外から参照できるオブジェクト…GC ルート以外のオブジェクトは GC ルートを経由しないと参照できません

④ CPU

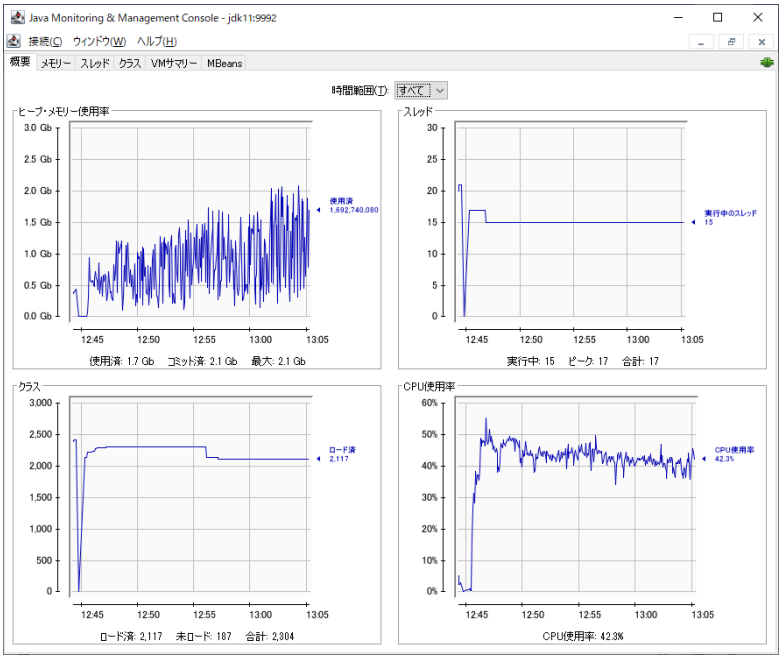
CPU Sampler を使います。VisualVM には Sampler と Profiler があり、類似したグラフを表示します。違いは、Sampler は定期的にスレッドダンプから数値を得るのに対して、Profiler はより厳密な数値を得るためにコードを一部変更して実行時に値を取得するようです。このため Profiler はアプリケーションの実行効率に影響を及ぼす可能性があります。したがって、初動は Sampler で全体の動作を確認します。

〔Total Time〕の大きなクラス.メソッドを掘り下げていくと、CPU を消費している箇所を見つけることができます。



4.2. JConsole

JConsole はアプリケーション内部へのドリルダウンはできませんが JDK 以外の追加のインストールが必要がないため、制限の多い環境では機能するかもしれません。



5. リモートでのモニタリング

Linux サーバを AP サーバにした場合 GUI(X-Window)は一般的に使わないため、VisualVM と JConsole は機能しません。このような環境では JVM 側に JMX(Java Management Extensions)用のポートを開けておくと Windows 等上のプロファイラーから接続することができます。

JMX のポートは Java アプリケーション(AP コンテナ等)の起動オプションに以下を指定します。

① com.sun.management.jmxremote.port=*ポート番号*

② com.sun.management.jmxremote.authenticate=false　　認証無

③ com.sun.management.jmxremote.ssl=false　　通信に SSL を使用しない

---

②：認証を行う場合は、パスワードファイルの作成等を行います

③：SSL を使う場合は証明書の作成と設置が必要です

＜JShell を使った実行例＞

```
jshell -R-Dcom.sun.management.jmxremote.port=9992 ¥
       -R-Dcom.sun.management.jmxremote.authenticate=false ¥
       -R-Dcom.sun.management.jmxremote.ssl=false
```

※オプションの先頭についている”-R”は JShell が立ち上げる JVM(RemoteExecutionControl)に渡すためのオプションなので JShell 以外では不要です

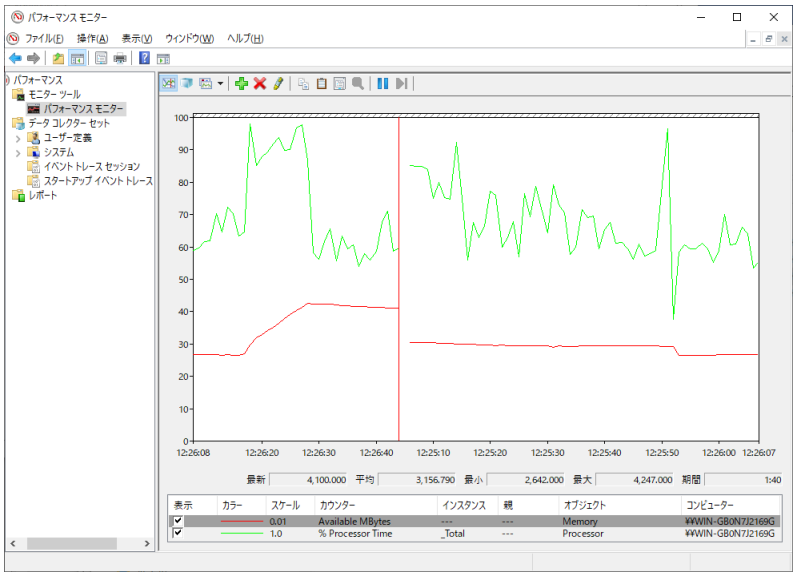
6. OS からの監視

アプリケーションの異常は CPU 使用率やメモリの逼迫等の形で OS に波及します。AP サーバであってもアプリケーション以外に多種のプロセスが走っているため、異常の初期状態は OS が感知する場合があります。

6.1. Windows

（1）パフォーマンスモニタ

特に JVM 以外のアプリケーションが同一 OS 配下で稼働している場合は、システム管理ツールー ＞ パフォーマンスモニタでサーバ全体のリソース消費状況も確認した方がよいでしょう。



《以下のカウンターを表示》

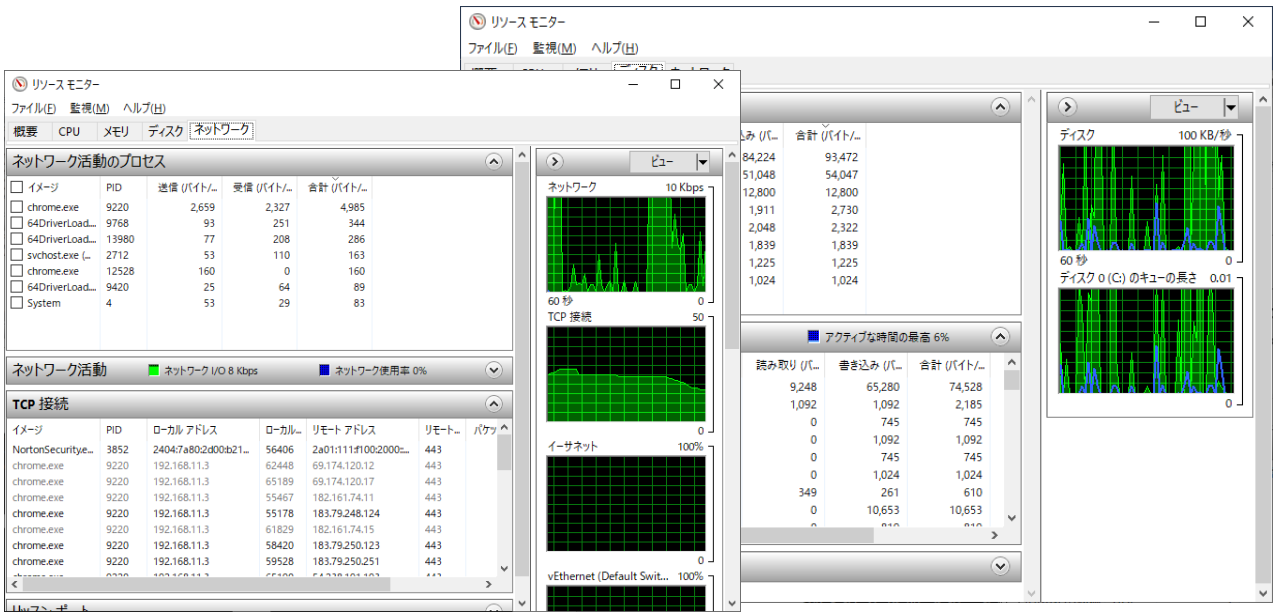
・Available Mbyte

（右クリック-「選択したカウンターのスケール設定」）

・% Process Time _Total

（2）システム管理ツールー ＞ リソースモニタ

バッチ処理がある場合はディスク、FTP 等がある場合はネットワークを監視



6.2. Linux

（1）top

実行中のプロセス数や Cpu、メモリの全体状況、プロセス毎の情報を見ることができます。

特に、以下の項目の推移を監視するのに便利です。

%CPU, %MEM …CPU（コア合算)、メモリのプロセス毎の使用率

```
top - 03:57:14 up  4:07,  0 users,  load average: 1.20, 1.07, 0.82
Tasks:   9 total,   1 running,   7 sleeping,   0 stopped,   1 zombie
%Cpu(s): 29.2 us,  5.0 sy,  0.0 ni, 64.0 id,  1.3 wa,  0.0 hi,  0.5 si,  0.0 st
MiB Mem :   6249.3 total,   3618.2 free,   1664.7 used,    966.4 buff/cache
MiB Swap:   2048.0 total,   2048.0 free,      0.0 used.   4011.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1141 root      20   0 4811928 735100  28328 S 121.3  11.5   8:25.04 java
 1121 root      20   0 5147944 214016  39356 S   3.7   3.3   0:43.08 jshell
    1 root      20   0 5145700 196088  38028 S   0.3   3.1   1:06.52 jshell
   26 root      20   0 4545660  52652  27836 S   0.0   0.8   0:28.06 java
   73 root      20   0    5988   3772   3256 S   0.0   0.1   0:00.19 bash
  103 root      20   0       0      0      0 Z   0.0   0.0   1:18.85 java
  191 root      20   0    6504   4324   3272 S   0.0   0.1   0:00.20 bash
  368 root      20   0    5988   3764   3252 S   0.0   0.1   0:00.11 bash
  928 root      20   0    8824   3572   3100 R   0.0   0.1   0:00.57 top
```

＜基本的な操作キー＞

x, b: ソート列、現在行を強調表示

＞:ソート列を右に移動

＜: ソート行を左に移動

q : コマンド終了

（2）sar

各種リソースの使用状況を時系列で見ることができます。

コマンドの形式 ＞ **sar** <*対象*> <サンプリング・インターバル（秒）> 終了は ctrl + c

※sar コマンドが「command not found」になる場合は、**sysstat** パッケージのインストールが必要です

① sar -r 1　1 秒毎にメモリの使用状況を表示(使用率：**%memused**)

```
$ docker exec -it 6204913eefca bash
root@6204913eefca:/# sar -r 1
Linux 5.10.16.3-microsoft-standard-WSL2 (6204913eefca)  09/17/21        _x86_64_        (4 CPU)

10:01:07    kbmemfree   kbavail kbmemused  %memused kbbuffers  kbcached  kbcommit   %commit  kbactive   kbinact   kbdirty
10:01:08      3078148   3459000   2321072     36.27     22044    922688   5840084     68.74    252012   2948096        64
10:01:09      3077888   3458740   2321328     36.27     22044    922688   5840084     68.74    252012   2948284        20
10:01:10      3077888   3458740   2321328     36.27     22044    922688   5840084     68.74    252012   2948284        20
10:01:11      3077888   3458740   2321328     36.27     22044    922688   5840084     68.74    252012   2948344        20
10:01:12      3077888   3458740   2321328     36.27     22044    922688   5840084     68.74    252012   2948344        20
10:01:13      3077888   3458740   2321328     36.27     22044    922688   5840084     68.74    252012   2948344        20
10:01:14      3077888   3458740   2321328     36.27     22052    922680   5840084     68.74    252012   2948344        20
10:01:15      3077888   3458740   2321320     36.27     22052    922688   5840084     68.74    252012   2948344        44
10:01:16      3077888   3458740   2321320     36.27     22052    922688   5840084     68.74    252012   2948388        44
```

② sar -p 1　1 秒毎に CPU の使用状況を表示(使用率：**%user**)

```
root@6204913eefca:/# sar -p 1
Linux 5.10.16.3-microsoft-standard-WSL2 (6204913eefca)  09/17/21        _x86_64_        (4 CPU)

10:05:45        CPU     %user     %nice   %system   %iowait    %steal     %idle
10:05:46        all     37.50      0.00      4.25      4.50      0.00     53.75
10:05:47        all     38.25      0.00      2.75      0.00      0.00     59.00
10:05:48        all     39.15      0.00      2.74      0.00      0.00     58.10
10:05:49        all     38.00      0.00      4.50      0.00      0.00     57.50
10:05:50        all     36.18      0.00      4.27      0.00      0.00     59.55
10:05:51        all     36.78      0.00      3.78      0.00      0.00     59.45
10:05:52        all     39.65      0.00      3.49      0.00      0.00     56.86
10:05:53        all     37.69      0.00      3.02      0.00      0.00     59.30
```

③ sar -d 1　1 秒毎にデバイスの使用(r:読込み、w:書込み)状況を表示(ビジー率：**%util**)

```
root@6204913eefca:/# sar -d 1
Linux 5.10.16.3-microsoft-standard-WSL2 (6204913eefca)  09/17/21        _x86_64_        (4 CPU)

10:10:19          DEV       tps     rkB/s     wkB/s     dkB/s   areq-sz    aqu-sz     await     %util
10:10:20        loop0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:10:20        loop1      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:10:20          sda      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:10:20          sdb      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:10:20          sdc      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
```

④ sar -n DEV 1　ネットワークインターフェース毎の転送量等(rx:受信、tx:送出)

```
root@6204913eefca:/# sar -n DEV 1
Linux 5.10.16.3-microsoft-standard-WSL2 (6204913eefca)  09/17/21        _x86_64_        (4 CPU)

10:15:44        IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s   %ifutil
10:15:45           lo     16.00     16.00      0.88      0.88      0.00      0.00      0.00      0.00
10:15:45        tunl0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:15:45         sit0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
10:15:45         eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00      0.00
```

