

内容

はじめに1
1. アプリケーションの監視1
2. ツール
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)2
2.2. JDK Mission Control(JMC)
2.3. jcmd
2.4. JConsole, jps, jstat
3. モニタリング
3.1. 稼働中 JVM の一覧と各実行パラメータを表示3
3.2. 稼働中 JVM をリアルタイム監視3
4. プロファイリング
4.1. VisualVM
4.2. JConsole
5. リモートでのモニタリング12
6. OS からの監視13
6.1. Windows
6.2. Linux

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.





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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 1



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



2. ツール

IVM 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。JDK7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
- 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。 jps
- 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 jstat 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。

フューチャオフィスコーデネイトサービス株式会社

Office

Coordinate Service Corporation

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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 2



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

jdk.jshell/jdk.internal.jshell.tool.JShellToolProvider -Dapplication.home=/us
 jdk.jshell.execution.RemoteExecutionControl -agentlib:jdwp=transport=dt_sock
 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 -1

128 jdk.jcmd/sun.tools.jcmd.JCmd -1

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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 3





<実行例>

jstat -gccause -t -h 20 748 100ms

Timestamp	S0	S1	E	0	М	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 F	Pause No GC
458.	3 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 F	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 F	Pause No GC
459.	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 F	Pause No GC
459.	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 F	Pause No GC
459.1	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 F	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	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 F	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 F	Pause No GC
(中略)														
Timestamp	S0	S1	E	0	М	CCS	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT LGCC	GCC
3305.3	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 Al	llocation No GC
3305.4	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 Al	llocation 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 Al	llocation 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 Al	llocation 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 Al	llocation No GC
3305.	3 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 Al	llocation No GC
3305.	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 Al	llocation No GC
3306.	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.	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 Al	llocation 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 Al	llocation 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/0x000000840084840.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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 4





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	0	М	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:/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.O.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)
```

https://docs.oracle.com/javase/jp/11/gctuning/introduction-garbage-collection-tuning.html

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 5

⁵ 公式なチューニングガイドは下記 URL



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> <u>C:¥Tools¥visualvm 21</u>

(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。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 6





(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。

崔 VisualVM 2.1	- 0	×
File Applications View Tools Win	ndow <u>H</u> elp	
: 🕒 📖 : 📽 🎎 🖬 😫 😫		
Applications × –	ualWr 🎂 JShell remote agent (pid 14040) 🗙 🎂 JShell (pid 2616) 🗙 🎂 JShell remote agent (pid 11624) 🗙 🗍	-
	🙀 Overview 👹 Monitor 🧮 Threads 🔬 Sampler 🚫 Profiler	
JShell (pid 4256)	U UShell remote agent (più 11024)	
- Pernote	Monitor Classes	reads
🌆 VM Coredumps	Uptime: 3 min 59 sec Perform GC Heap Du	пе
@millionalistics	OPU X Heep Metaspace	×
	1000- 10	d heap
	Classes X Threads	×
		4
	Total loaded classes Shared loaded classes Live threads Deemon th	nreads
L		

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。

VisualVM 2.1	- 🗆 X
File Applications View Tools Win	dow Help
8 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	
Applications × –	4040) 🎂 JShell (pid 2616) 🗙 🎂 JShell remote agent (pid 11624) 🗙 🎂 JShell remote agent (pid 12432) 🗙 4 🕨 🖛
E- E Local	🙀 Overview 😹 Monitor 🧮 Threads 🔬 Sampler 🕑 Profiler
🥐 VisualVM 🚣 JShell (pid 7516)	Shell remote agent (pid 12492)
JShell remote agent (pid 12492)	Monitor OPU Memory Classes Mithreads
Hemote	Listing firm 26 rac
	Perform GC Heap Dump
🔚 Snapshots	CPU X Heap Metaspace X
	200 Ma 505 505 505 1220 1
	Olieses X Threads X 2,000 10 10
	0 1320 1322 1324 Total loaded dasser 0 1320 1322 Total loaded dasser 0 10 10

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 7





VisualVM 2.1						X
File Applications View Tools Win	dow Help					
E E E E E E E E E E E E E E E E E E E						
Applications X	4040 👍 JShell (pid 2616) 🗙 📥 JShell remote agent (pid 11624) 🗙	🐣 JShell remote agent (p	d12492) 🗙	a .		
	Cherniew Monitor Threads (1) Sampler (2) Profiler	8				
M/laush/						
- 💩 JShell (pid 7516)	JShell remote agent (pid 12492)		_	_		
e Remote	Monitor		Memor	y Classes	Three	ds
	Uptime:10 min 20 sec		Per	form GC Hea	p Dump	
Snapshots						
	Heap Wetaspace					~
	Size: 592,445,472 B U Max: 2,118,123,544 B	lsed: 421,551,544 B			_	
	500 MB					
	400 MB				-	
	300 MB					
	200 MB -				_	
	100 MB-					
	0 MB	12.24	2.26	12,29		-
	10.40 10.44		0.20	📕 Heap size 📘	Used he	ap
L]						
Image: Second state of the second s	dow <u>H</u> elp					×
Applications × –	Start Page 🗙 📥 JShell remote agent (pid 4680) 🗙 📥 JShell remote	agent (pid 6408) 🗙		(
	🛐 Overview 🔛 Monitor 🧮 Threads 🖓 Sampler 🕑 Profiler					
JShell remote azent (pid 4680)	JShell remote agent (pid 6408)					
JShell (pid 3192)	Sampler			Г	Setting	5S
💩 JShell remote agent (pid 6408)					_	
- M Coredumps	Sample: 🕑 CPU 🧱 Memory 🔯 Stop					
	Status: memory sampling in progress					
E Snapshots						
	Heap histogram Per thread allocations					
	Results: U C2 🛆 Collected date: 📰 Snapshot	X		Perform GC	Heap Du	mp
	Name	Live Bytes		Live Objects		•
	ison lant. Object	1,166,166,0	00 B (97%)	2,416,347	(74.5%)	^
	int]	41942	40B (0.3%)	13,487	(0.4%)	
	i avautil.TreeMap\$Entry	3,927,7	50 B (0.3%)	98,194	(310	
		3,401,4	32 B (0.3%)	11,936	(0.4N)	
	java.io.HeapCharBuffer	2,848.2	56 B (0.1%)	89,007 35.797	01.19	
	iavalant String	1,136,7	50 B (0.1 %)	47,365	0.5%	
	isvaio.ObjectStreamClass	976,3	52 B (0.1 %)	9,388	(0.3%)	
	💫 iava.util.TreeMap\$KeyIterator	587,9	36 B (0%)	18,373	(0.5%)	
	😥 isveutil.Hash Map	453,6	96 B (0%)	9,452	(0.3N)	
	iavalant Long	420,5	28 B (0%)	17,522	(0.5%)	
	isva.util.TreeMap	408,7	20 B (0%)	8,515	(0.3%)	
	isve lang management. Threadinfo	400,7	2B (0%)	3,853	(0.1 N)	~
	in the second beaution of the second beaution	361.1		11.987	1112260	

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...

- ⑦ List 型のインスタンス変数 strArr を作る
- ④ 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) ⑦で作った strArr に⑦で作ったインスタンス(の参照)を追加していく
- (王) 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>();//⑦
   String str = "0".repeat(1000);
   public void biting() {
      for (long i=1; true; i++) {
          str = str + Long.toString(i);//④
          strArr.add(str);//🗇
          if(i%300==1) strArr = new ArrayList<String>();//①
      }
   }
}
(new MemoryBiter()).biting();
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. D. 8

フューチャオフィスコーデネイトサービス株式会社 Office Coordinate Service Corporation Future



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。

VisualVM 2.1		_		X
File Applications View Tools Win	dow Help			
	<u>-</u> +			
: 🖿 📖 : 🐼 🎆 🔛 🗠 🏸				
Applications × –	4040 🎂 JShell (pid 2616) 🗙 🎂 JShell remote agent (pid 11624) 🗙 🎂 JShell remote agent (pid 12492) 🗙			•
	📊 Overview 🚟 Monitor 🧮 Threads 🤬 Sampler 🕑 Profiler			
🚰 Visual VM 💩 JShell (pid 7516)	🔘 JShell remote agent (pid 12492)			
JShell remote spent (pid 12492)	Manitar 🗌 OPU 🔽 Memory	Classes		reads
- 🚰 Remote - 🔄 VM Coredumos	Lindiana 10 min 20 man		_	
- R Snapshots	Perf	orm GC	Heap Du	mp
- Snapshots				
- <u></u>	Heap Metaspace			×
	Size: 592,445,472 B Used: 421,561,544 B			
	Max: 2,118,123,544 B			

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。

VisualVM 2.1				— П X
File Applications View Tools Window	<u>H</u> eip			
Applications ×	Start Page 🗙 盠 JShell remote agent (pid 9176) 🗙			< ▶ ▼ □
E- E	📊 Overview 🗰 Monitor 🔚 Threads 🔐 Sample	er 🕑 Profiler 🔚 [heapdu	mp] 9:34:15 ×	
Mileur (internet)	O JShell remote errent (pid 9176)			
- Shell remote agent (pid 9176)	the Des			
[hespdump] 9:34:15	Heap Lump			
Remote	Summery +			0
- Im VM Coredumps				
- Snapshots	Heap		Environment	
Snapshots	Size:	1,174,930,649 B	System	Windows 10 (10.0)
	Classes:	2,450	Architecture:	amd64.64bit
	Classic adars	75	Java Home:	Carloos#Adoptopenack#uk-110103-Notspot
	GC Roots	2318	Java Name:	Open-DK 54-Eit Senier (M (11 01049 mixed mode)
	Objects Pending for Finalization:	0	Java Vendor:	AdoptOpenuDK
			JVM Uptime:	25 min 27 seo
	System Properties [show]		Classes by Size of Instance	a [view all]
		15.921 (02.710)		1179 079 613 P (00.00)
	isus land String	15,331 (23.7%)	isus long String	443 729 B (362 A)
	isvauti HashMap\$Node	4,366 (6.5%)	isvalant Object[]	320,840 B (0%)
	isvalang.Object	2,756 (4.1%)	isva.util.HashMap\$Node	192,104 B (0%)
	i ava Jang ref:WeakReference	2,453 (3.5%)	🟠 java.util.HashMap\$Node	137,296 B (0%)
	Instances by the [view all]		Dominators by Retained Siz	wiew all]
	() byte[#15924: 0.137.120 items	6.197.150 B (0.5%)	Retained sizes must be comp	uted first:
	() byte #15923 : 6,197,119 items	6,197,143 B (0.5%)	Compute Retained Sizer	
	byte[#15922 : 6,197,112 items	6,197,136 B (0.5%)	Compare received bices	
	8 byte #15921 : 6,197,105 items	6,197,129 B (0.5%)		
	byte[#15920 : 6,197,098 items	6,197,122 B (0.5%)		

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 9





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 Root⁶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。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 10



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。

VisualVM 2.1 File Applications View Tools Window	Help				- 0		×
Applications × –	Start Page 🗙 🧔 JShell remote agent (pid 3176) 🗙						
Local VisualVM JShell (pid 6912)	Overview Wenter Threads Sampler Profiler JShell remote agent (pid 9176)						
JShell remote agent (pid 9176)	Sampler] Settine	E S
- 🔤 VM Coredumps - 📽 "FR Snapshots - ஞ Snapshots	Sample: OPU Memory Stop Status: OPU sampling in progress						
	CPU samples Thread OPU time					hand Da	
	Nasures: U C view N N N View Conected data: I snapsnot	Total Time	~		Total Time (DPL	J)	mp T
		41	11,195 ms	(1.00%)	411,195 ms	(1.00%)	^
	Line *** JFluid Monitor thread *** Line *** Profiler Agent Special Execution Thread 6	41	11,195 ms 11,195 ms	(100%)	0.0 ms 362 ms	(−%) (1.00%)	
	main field with the leve oution Remote Execution Control main 0	38	89,868 ms 89,868 ms	(100%) (100%)	389,688 ms 389,688 ms	(100%)	
	[Akishellexecution.DilliforwardExecutionControlAndiO 0	36	89,868 ms 89,868 ms	(100%)	389,688 ms	(100%)	
		38	89,868 ms 89,868 ms 89,688 ms	(100%)	389,688 ms 389,688 ms 389,688 ms	(100%)	
	Wij idki shell assocition DirectExecutionControl invoke () Wij idki shell assocition DirectExecutionControl invoke ()	38	89,688 ms 89,688 ms	(100%)	389,688 ms 389,688 ms	(100%)	
	Yei ava lang reflect Method invoke 0 Yei (dikinena) reflect Delesting Method Accessoring Linvoke 0	38	89,688 ms 89,688 ms	(100%) (100%)	389,688 ms 389,688 ms	(100%)	
		36	89,688 ms 89,688 ms 89,688 ms	(100%) (100%) (100%)	389,688 ms 389,688 ms 389,688 ms	(100%)	
	PEPL 3Shelif119/kemory6Herbibing () O Self time O Self time O iava Jang Invoke Invokers3Holder InkToTargetMethod ()	36	89,688 ms 90,897 ms 15,729 ms	(100%) (49%) (29.7%)	389,688 ms 190,897 ms 115,729 ms	(100%) (49%) (29.7%)	
	ter bilder bild		50,838 ms 21,961 ms	(15.5%) (5.5%)	60,838 ms 21,961 ms	(15.5%) (5.5%)	~
	< >>						

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 11





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= r + F
- 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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 12





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。



(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



Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 13



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 - Tasks: %Cpu(s MiB Me MiB Sw	03:57 9 f ;): 29 ;m : (ap:	:14 up total, .2 us, 6249.3 2048.0	4:07 1 r 5.0 tota	7, 0 use running, sy, 0.6 al, 361 al. 204	ers, lo 7 sle 0 ni, 64 18.2 fre 18.0 fre	ad avera eping, .0 id, 2 e, 1664	ge: 1.2 0 stop 1.3 wa, 4.7 use 0.0 use	0, 1.0 ped, 0.0 d, d. 4	7, 0.82 1 zombie hi, 0.5 si, 0.0 st 966.4 buff/cache 011.9 avail Mem	
0.7.0				NTOT.		CHIP C	Nonu	0/44544	TTUS - COMMAND	
PID	USER	PR	NI	VIRI	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

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 14





(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

sar -r1 1秒毎にメモリの使用状況を表示(使用率:%memused)

\$ docker e root@62049 Linux 5.10	\$ 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 -p1 1 秒毎に CPU の使用状況を表示(使用率:%user)

root@6204913	Beefca:/#	sar -p 1						
Linux 5.10.1	l6.3-micro	soft-stand	ard-WSL2	(6204913ee	fca) 09/17	/21	_x86_64_	(4 CPU)
10.05.45	CDU	Vucon	V nico	Vevetor	Viewsit	%ctool	%idle	
10:05:45	CPU	/auser	76HICE	/system	/slowalc	/osteal	/siule	
10:05:46	all	37.50		4.25	4.50		53./5	
10:05:47	all	38.25		2.75			59.00	
10:05:48	all	39.15		2.74			58.10	
10:05:49	all	38.00		4.50			57.50	
10:05:50	all	36.18		4.27			59.55	
10:05:51	all	36.78		3.78			59.45	
10:05:52	all	39.65		3.49			56.86	
10:05:53	all	37.69		3.02			59.30	
10.05.54				7.55				

③ sar -d1 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)						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								
10:10:20	loop1								
10:10:20	sda								
10:10:20	sdb								
10:10:20	sdc								

④ sar -n DEV1 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) 09/17/21x86_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	10	16.00	16.00	0.88	0.88					
10:15:45	tun10									
10:15:45	sit0									
10:15:45	eth0									

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. p. 15

