トラブルシューターの頭の中身

～7年間のJavaトラブルシュートサービスから

#jt12_b104

2012/04/04

Acroquest Technology株式会社

谷本 心 / 江里口 温

Acroquest Technology

自己紹介

- 谷本 心（Shin Tanimoto）
  Twitter : @cero_t
  Facebook : shin.tanimoto

- Acroquest Technology株式会社
  - フレームワーク開発
  - プロセス標準化
  - トラブルシュート
- 関西Javaエンジニアの会（関ジャバ）主催
- 日本Javaユーザグループ（JJUG）幹事

自己紹介

- 江里口 温（On Eriguchi）
  Twitter : @eripong
  Facebook : on.eriguchi

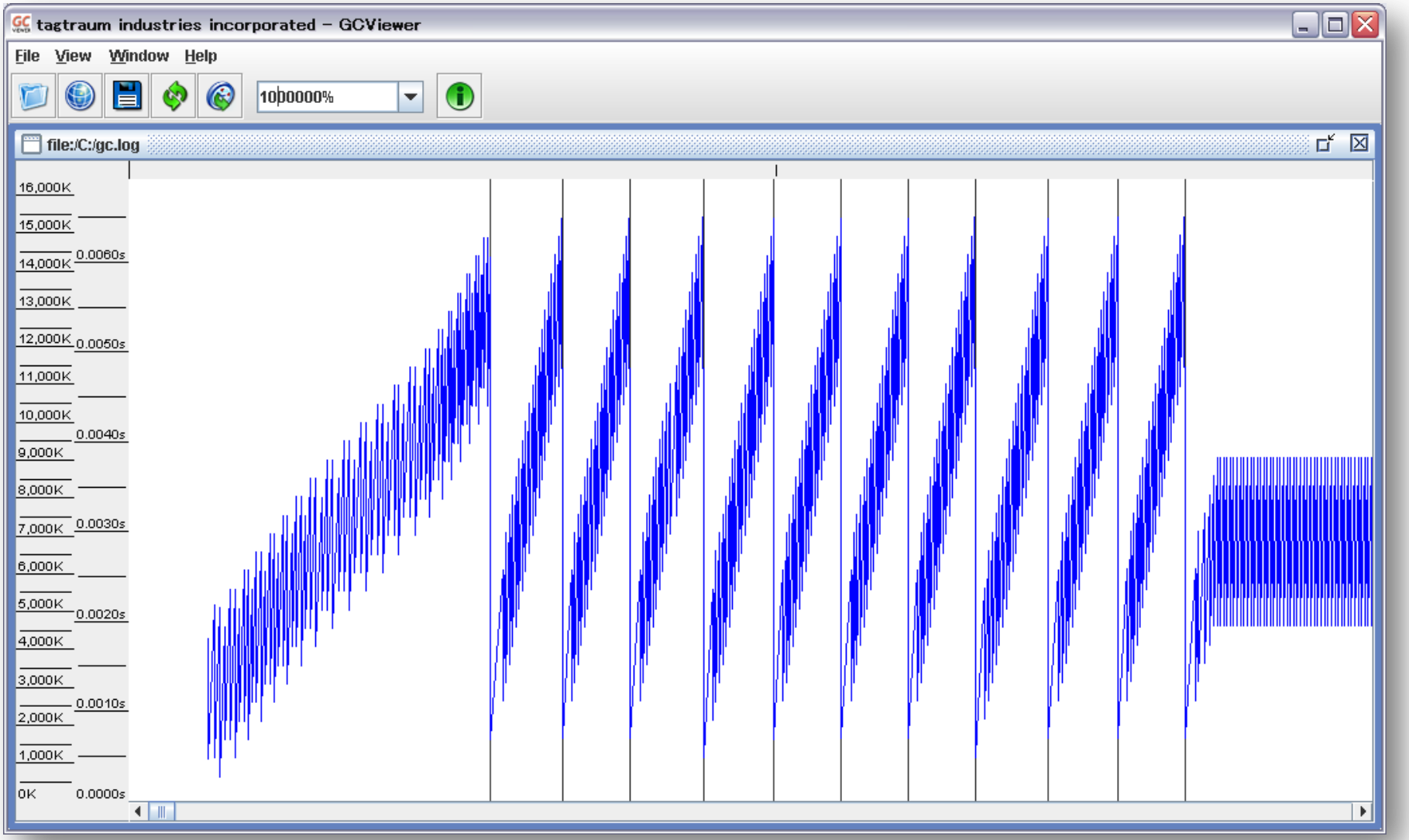
- Acroquest Technology株式会社
  - JaTS（JavaTrouble Shooting Service）
  - トラブルシュートツールENdoSnipeの開発

JaTSとは、こんなサービスです！

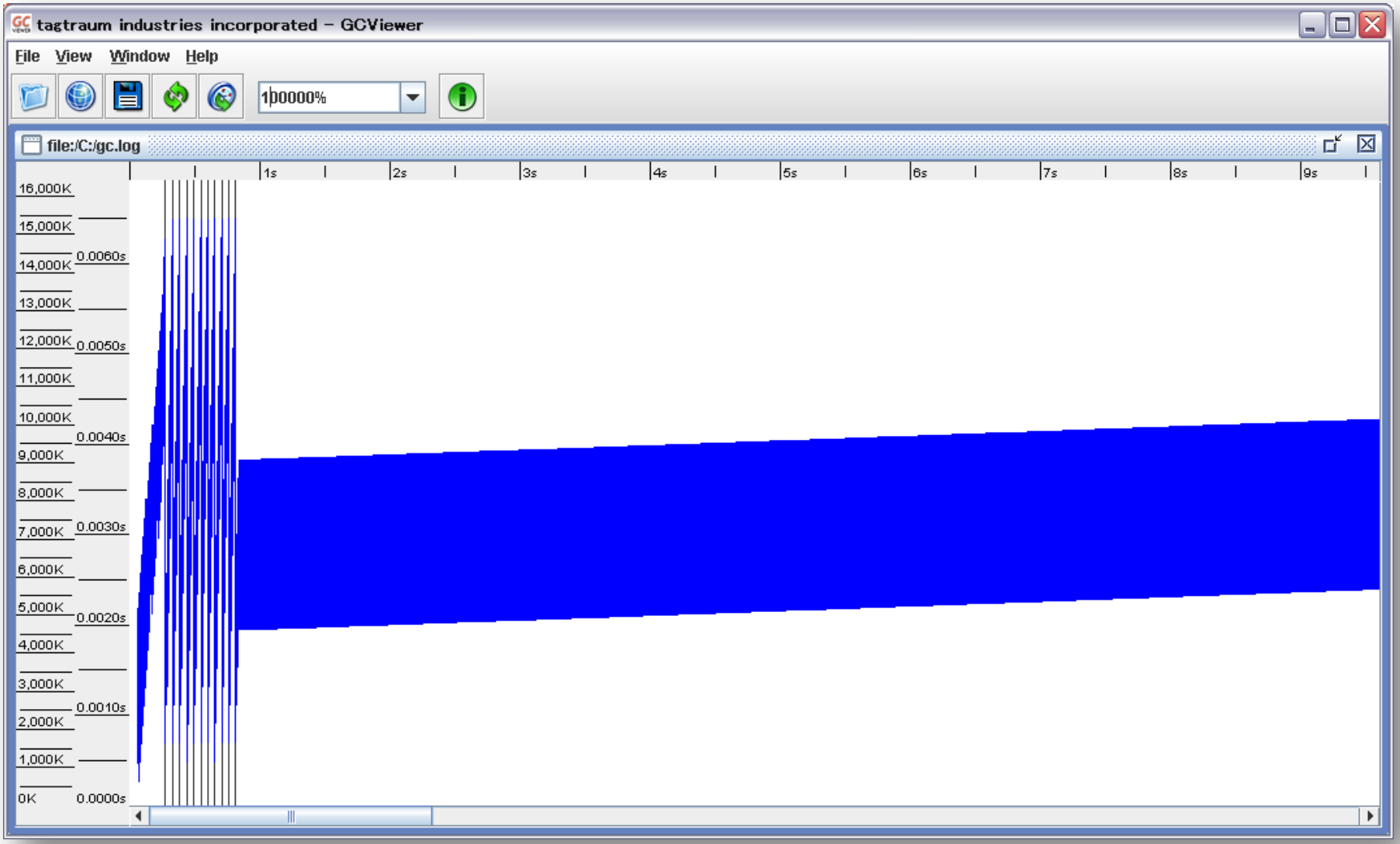
- 日本唯一（？）の期間内解決保障型トラブルシュートサービス

- 2004年のサービス開始以降、500件超の無敗記録を継続中！

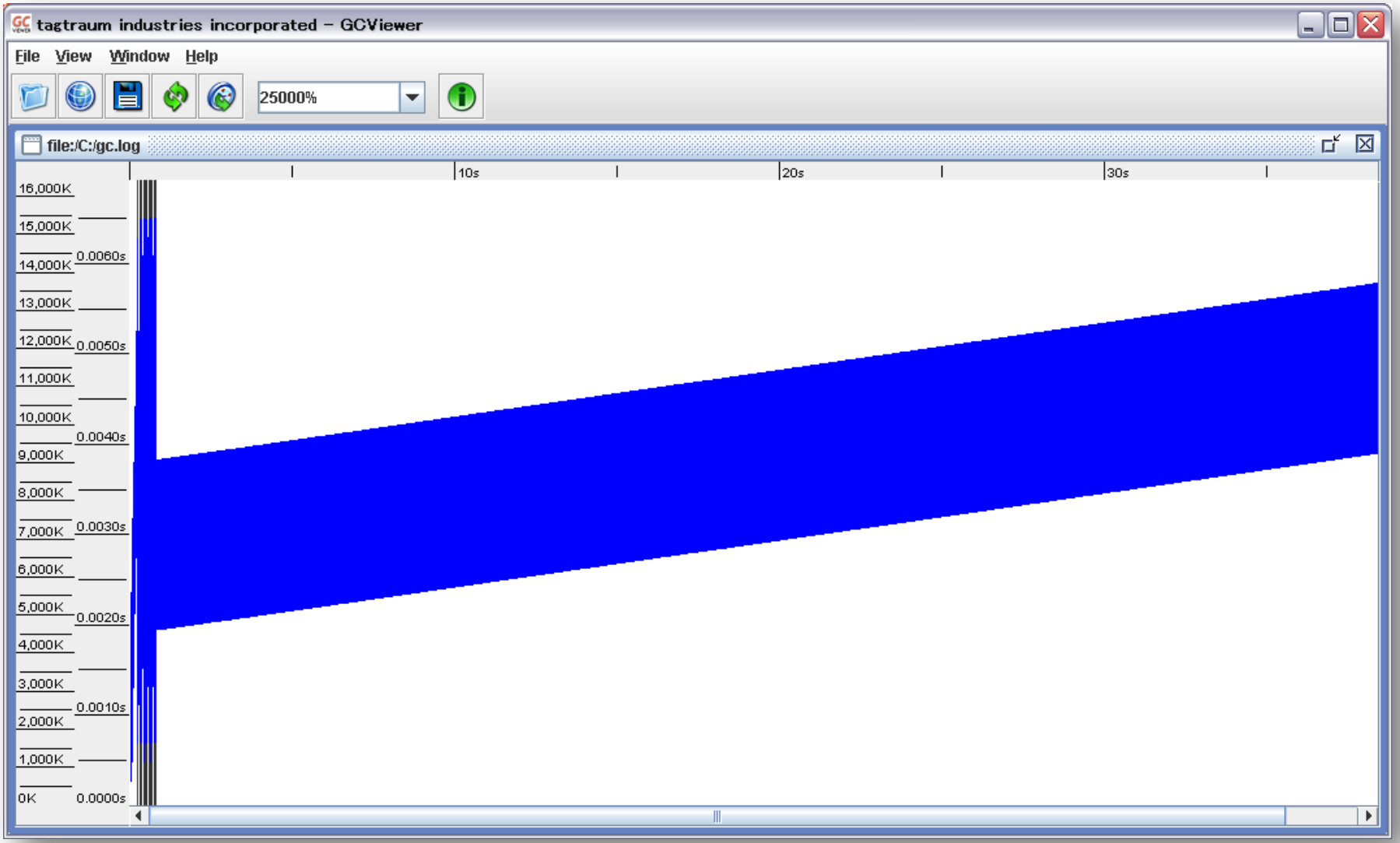
メモリリーク？


tagtraum industries incorporated – GCViewer
File View Window Help
10p0000%
file:/C:/gc.log
16,000K
15,000K
14,000K
13,000K
12,000K
11,000K
10,000K
9,000K
8,000K
7,000K
6,000K
5,000K
4,000K
3,000K
2,000K
1,000K
0K
0.0060s
0.0050s
0.0040s
0.0030s
0.0020s
0.0010s
0.0000s

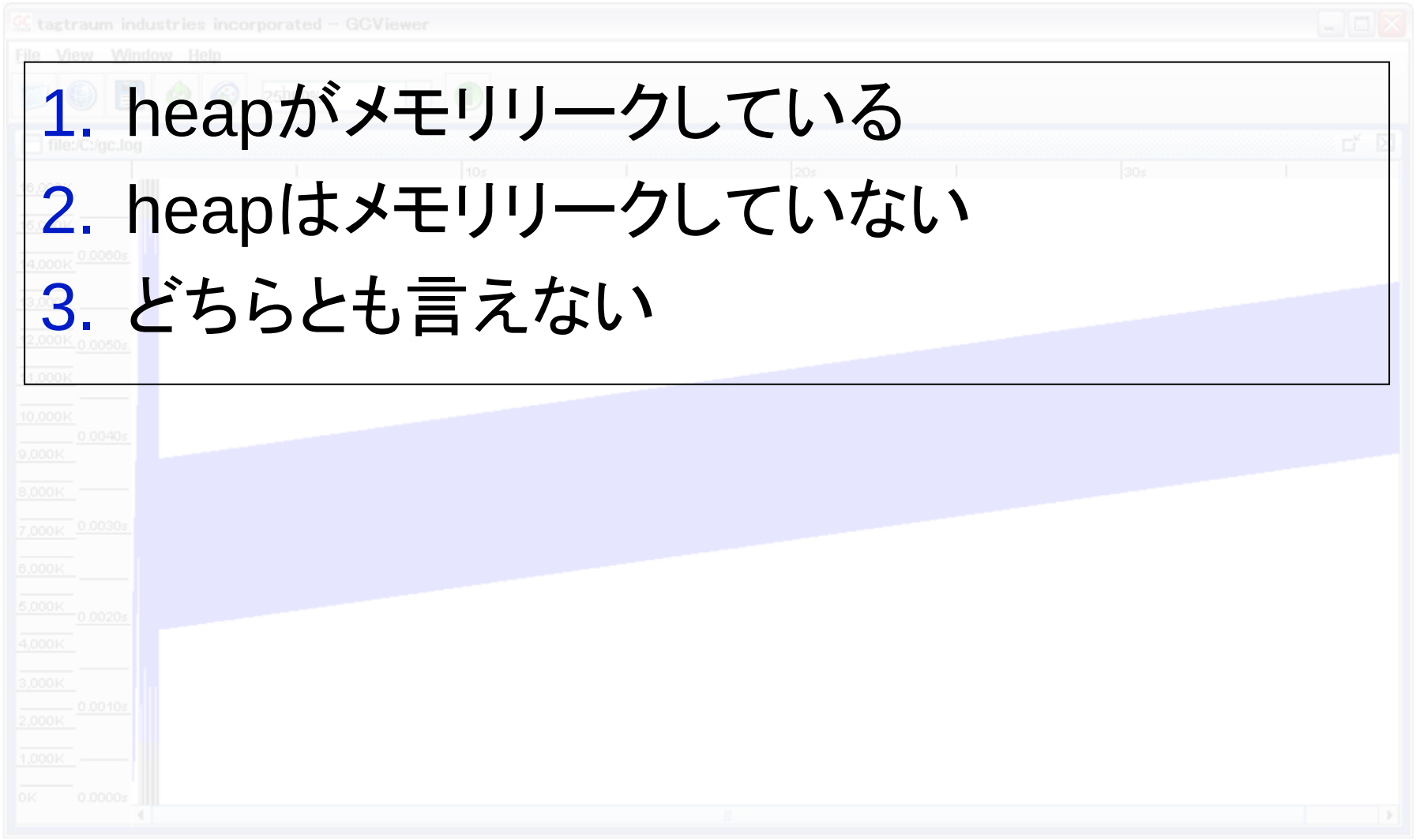
メモリリーク？


tagtraum industries incorporated – GCViewer
File View Window Help
100000%
file:/C:/gc.log
1s 2s 3s 4s 5s 6s 7s 8s 9s
16,000K
15,000K
14,000K
13,000K
12,000K
11,000K
10,000K
9,000K
8,000K
7,000K
6,000K
5,000K
4,000K
3,000K
2,000K
1,000K
0K
0.0060s
0.0050s
0.0040s
0.0030s
0.0020s
0.0010s
0.0000s

メモリリーク？

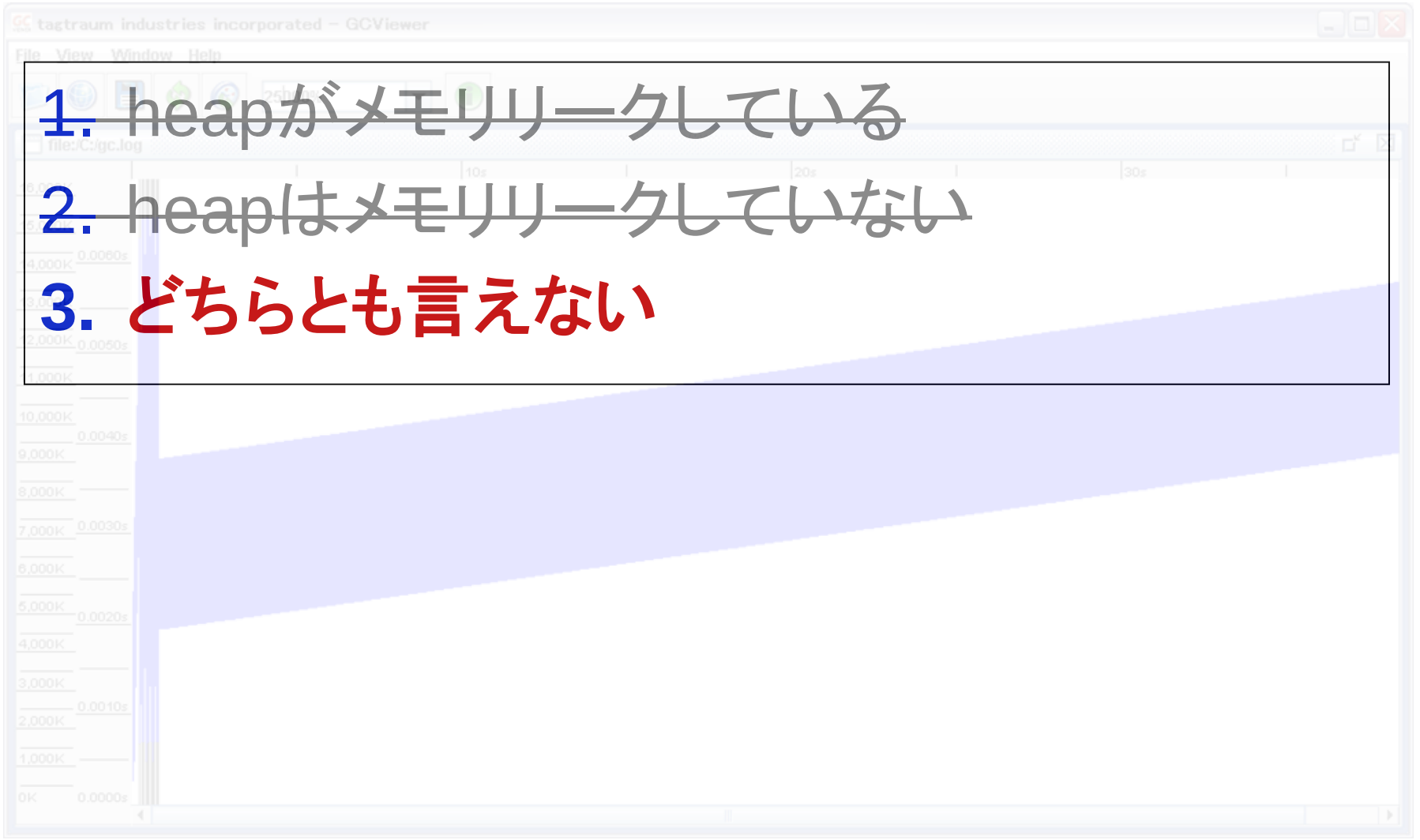

tagtraum industries incorporated – GCViewer
File View Window Help
25000%
file:/C:/gc.log
10s
20s
30s
16,000K
15,000K
14,000K
13,000K
12,000K
11,000K
10,000K
9,000K
8,000K
7,000K
6,000K
5,000K
4,000K
3,000K
2,000K
1,000K
0K
0.0060s
0.0050s
0.0040s
0.0030s
0.0020s
0.0010s
0.0000s

メモリリーク？



1. heapがメモリリークしている
2. heapはメモリリークしていない
3. どちらとも言えない

メモリリーク？



1. ~~heapがメモリリークしている~~
2. ~~heapはメモリリークしていない~~
3. どちらとも言えない

メモリリーク？

1. FullGCが実行されない限りは、heap使用量は右肩あがりになる。
2. FullGCを何度か実行させてみるべき。
   ① もっと長い時間テストする
   ② 手動でFullGCを発行させる
   ③ heap上限を低い値にする（例：-Xmx64m）

「思い込み」で調査してはいけない

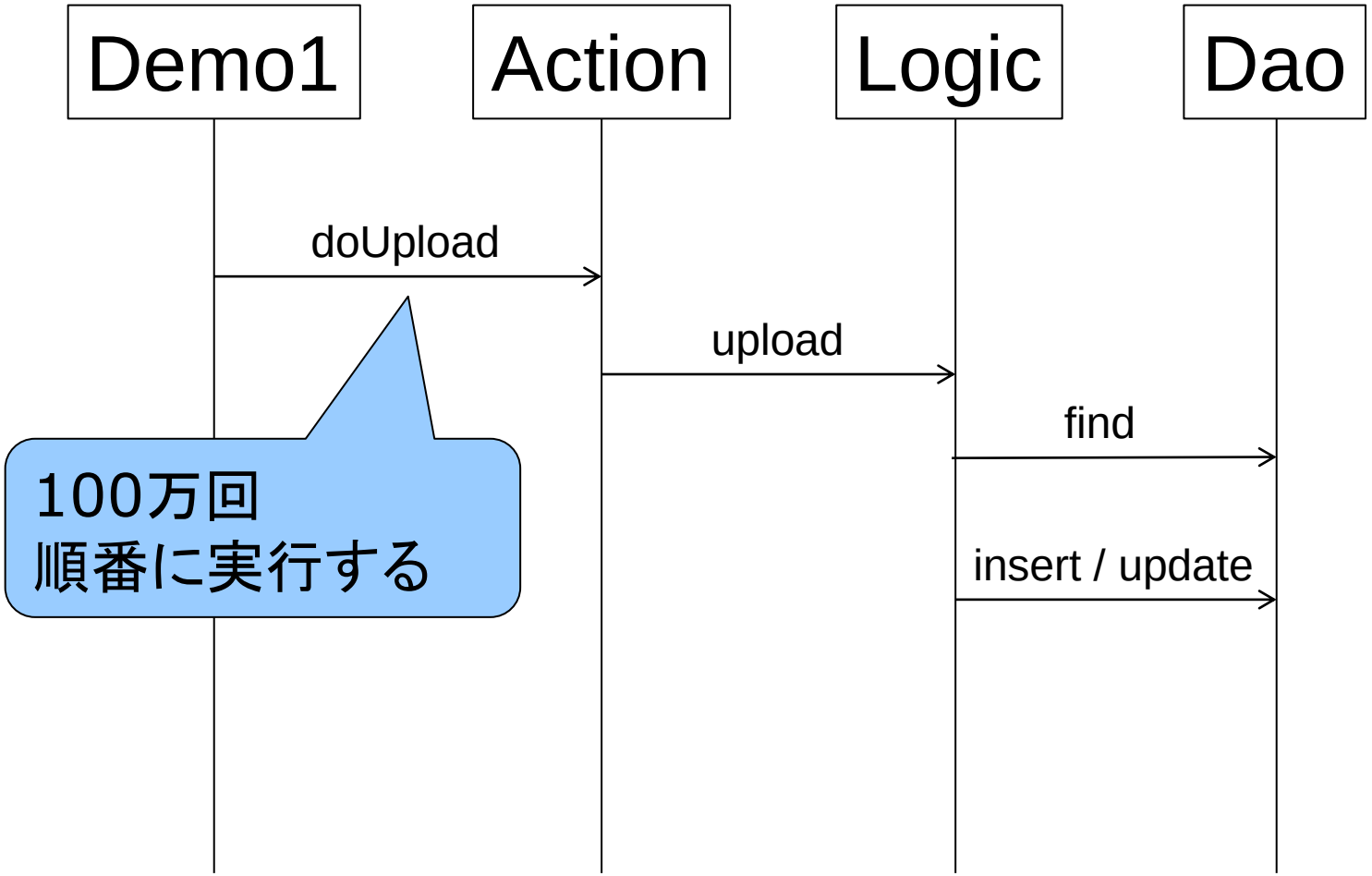
アジェンダ

1. 実演、トラブルシュート！
2. トラブルシューターの考え方
3. トラブルシューターの道具箱
4. トラブルシュート事例紹介

1. **実演、トラブルシュート！**
2. トラブルシューターの考え方
3. トラブルシューターの道具箱
4. トラブルシュート事例紹介

問題1
OutOfMemoryError: Java heap space

事例1：OutOfMemoryError: Java heap space



事例1：OutOfMemoryError: Java heap space

とりあえず、実行してみましょう。

あれ、実行すると例外が出ましたね。

```
C:¥javaone>java Demo1
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2786)
        at java.io.ByteArrayOutputStream.toByteArray(ByteArrayOutputStream.java:133)
        at Demo1.readFile(Demo1.java:33)
        at Demo1.main(Demo1.java:14)
```

問題箇所のソースを見る限り、
あまり問題なさそうなんですが・・・

こういう時は、どうしましょうか？

事例1：OutOfMemoryError: Java heap space

1. ファイル読み込み処理が怪しいから、
   やっぱりここを精査だ！
2. まだ何とも言えない。
   こういう時は、まずヒープダンプを取ろう！
3. その他

事例1：OutOfMemoryError: Java heap space

1. ~~ファイル読み込み処理が怪しいから、やっぱりここを精査だ！~~
2. **まだ何とも言えない。こういう時は、まずヒープダンプを取ろう！**
3. ~~その他~~

事例1：OutOfMemoryError: Java heap space

こういう時は、ヒープダンプを取りますね。

もし複数のトラブルシューターがいれば、
並行してスタックトレースが出ている箇所のソースを解析するんですけどね。

ヒープダンプですね、なるほど。

、、、で、どうやって取るんですか？

今回の場合ならOutOfMemoryErrorが
発生しているので、起動時の引数に
-XX:+HeapDumpOnOutOfMemoryError
というオプションを入れるのが良いでしょうね。

事例1：OutOfMemoryError: Java heap space

起動引数を入れて再実行してみましょう。

おっ、ダンプがファイルが出力されたようです。

```
C:¥javaone>java -XX:+HeapDumpOnOutOfMemoryError Demo1
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid1084.hprof ...
Heap dump file created [194545298 bytes in 2.248 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2786)
        at java.io.ByteArrayOutputStream.toByteArray(ByteArrayOutputStream.java:133)
        at Demo1.readFile(Demo1.java:33)
        at Demo1.main(Demo1.java:14)
```

このファイルは、どうやって解析するんでしょうか？

テキストエディタでおもむろに開いたりすると、
ダメですよね？

事例1：OutOfMemoryError: Java heap space

テキストエディタはあり得ないですね。

ダンプファイルを解析するツールはいくつかありますが
VisualVMを使うのが手っ取り早いでしょうね。

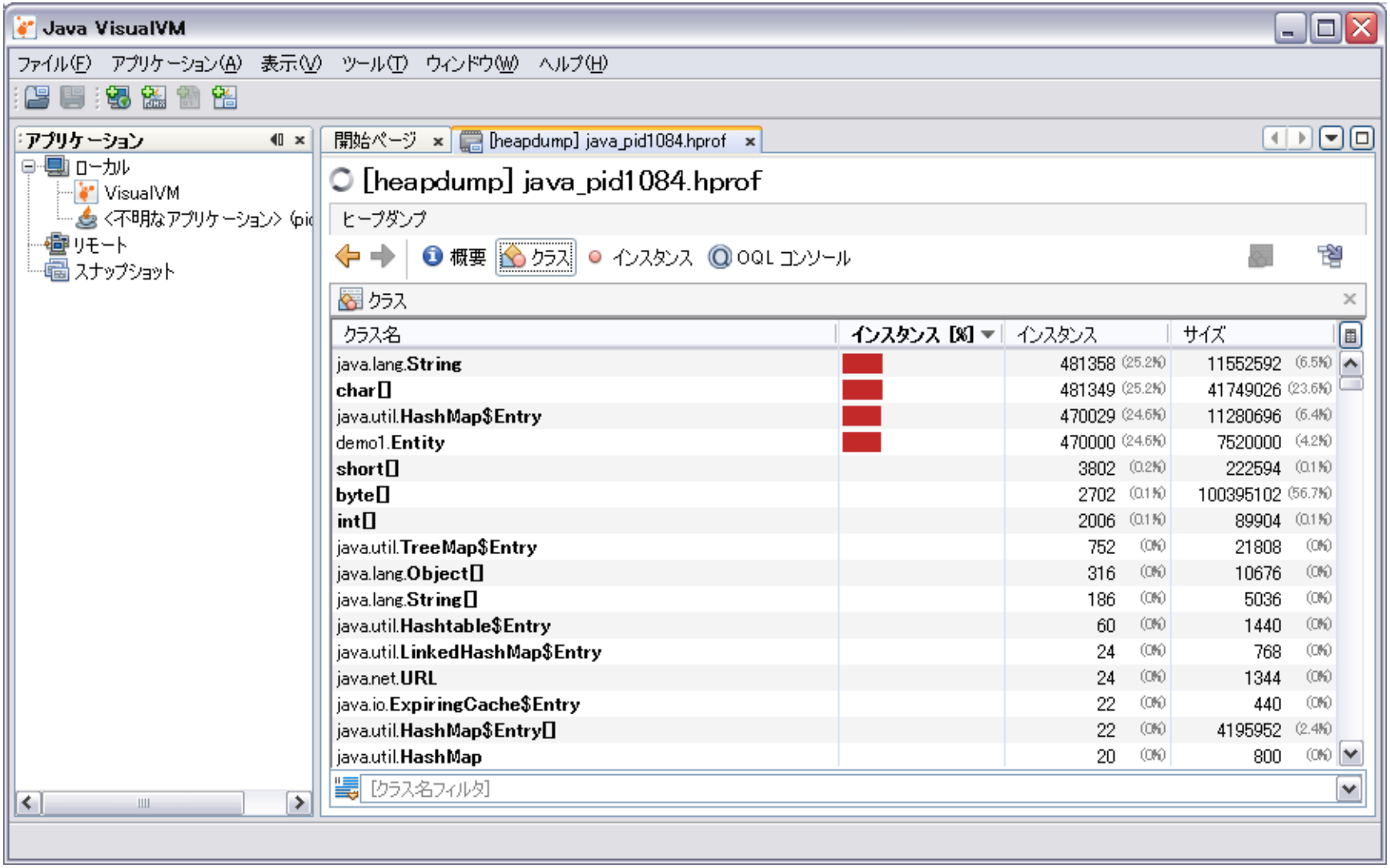
なるほど、VisualVMなら最近は標準で入ってますからね。

コマンドラインから jvisualvm と打ってみましょう。

オッケー、起動できました。
これでヒープダンプのファイルを読み込めるんですね。

早速、読み込んでみましょう。

事例1：OutOfMemoryError: Java heap space



事例1：OutOfMemoryError: Java heap space

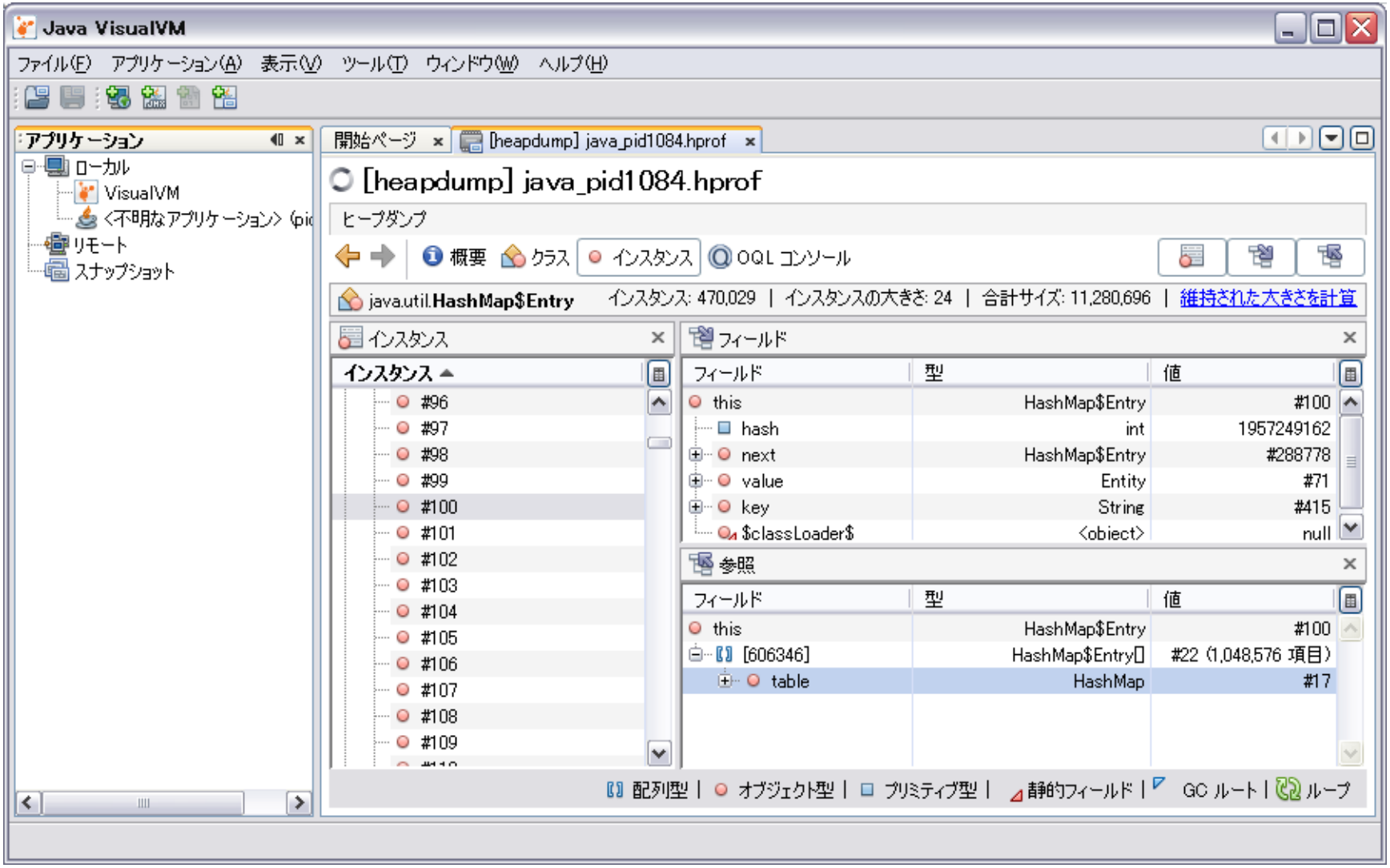
サイズで見れば byte[] が多いですね。
ただ、HashMap$Entryなどのインスタンス数も多いですね。

そうですね、HeapDumpを解析する時の観点は
「サイズ」と「オブジェクト数」の2つなのですが、
いまの場合、HashMap$Entryが47万もあるのは
ちょっと異常ですよね？

そうですね、
そんな大量オブジェクトを生成するようなプログラミングに
したつもりはありません。

では、このHashMap$Entryをダブルクリックして
インスタンスの詳細を見てみましょう。

事例1：OutOfMemoryError: Java heap space



事例1：OutOfMemoryError: Java heap space

親をたどって、HashMapクラスを開くと、
sizeというプロパティがあると思うんですが、
いくつになっていますか？

47万ですね。
要素数が47万のHashMapができたということですね。

そうですね。
では、このHashMapの親を辿って、
どのクラスが参照しているか見てみましょう。

辿ってみると・・・どうやら「Logic」というクラスの
「cache」というインスタンス変数のようですね。
何か不吉な名前ですが、開いてみましょう。

事例1：OutOfMemoryError: Java heap space

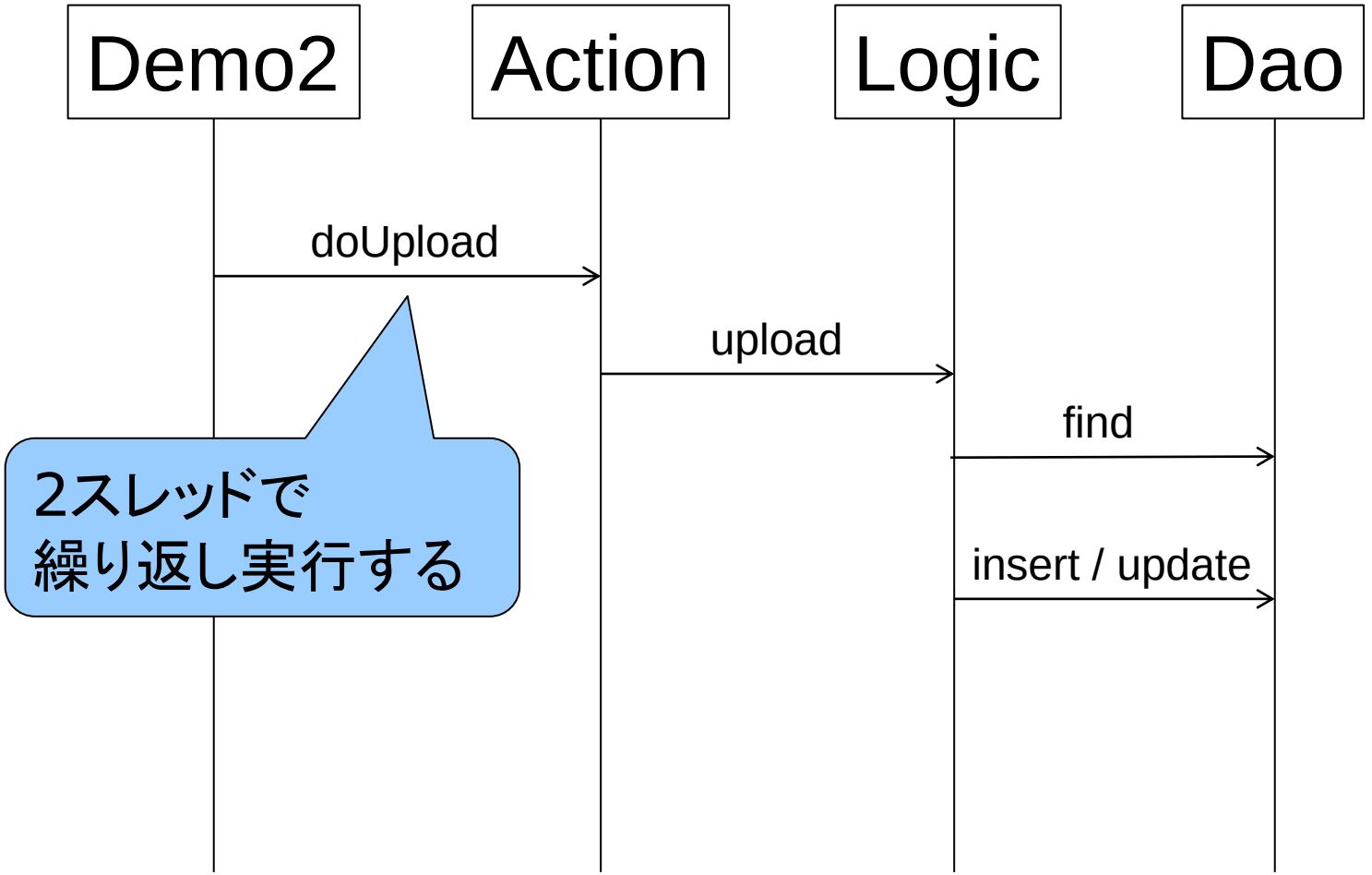
```
protected Entity find(String id) {
        String query = "select * from DEMO_TBL where id = " + id;
        if (cache.containsKey(query)) {
                return cache.get(query);
        }

        Entity result = dao.select(query);
        cache.put(query, result);

        return result;
}
```

あぁ、、、クエリのキャッシュを自前で実装しているんですが
追加しっぱなしで、クリアしてませんね。

この部分でメモリリークしているのは、間違いなさそうです。
キャッシュの方針を修正すれば、解決するでしょう。

問題2
処理が終わらない・・・

事例2：処理が終わらない・・・



事例2：処理が終わらない・・・

では、これも実行してみましょう。

あれ、処理が終わらないですね・・・

```
C:¥javaone>java Demo2
```

ほうほう。ちなみに、CPU使用率は、
どれぐらいになっていますか？

ほぼ100%で張り付いてますね。
これは、どういうことでしょうね？

事例2：処理が終わらない・・・

1. 無限ループの可能性が高いので、
   こういう時は、まずスレッドダンプを取ろう！
2. 無限ループの可能性が極めて高いので、
   ソースを見てforとwhileを精査しよう！
3. その他

事例2：処理が終わらない・・・

1. 無限ループの可能性が高いので、
   こういう時は、まずスレッドダンプを取ろう！
2. ~~無限ループの可能性が極めて高いので、~~
   ~~ソースを見てforとwhileを精査しよう！~~
3. ~~その他~~

事例2：処理が終わらない・・・

無限ループしている可能性がある場所を調べるには
スレッドダンプを取ってみるのが手っ取り早いですね。

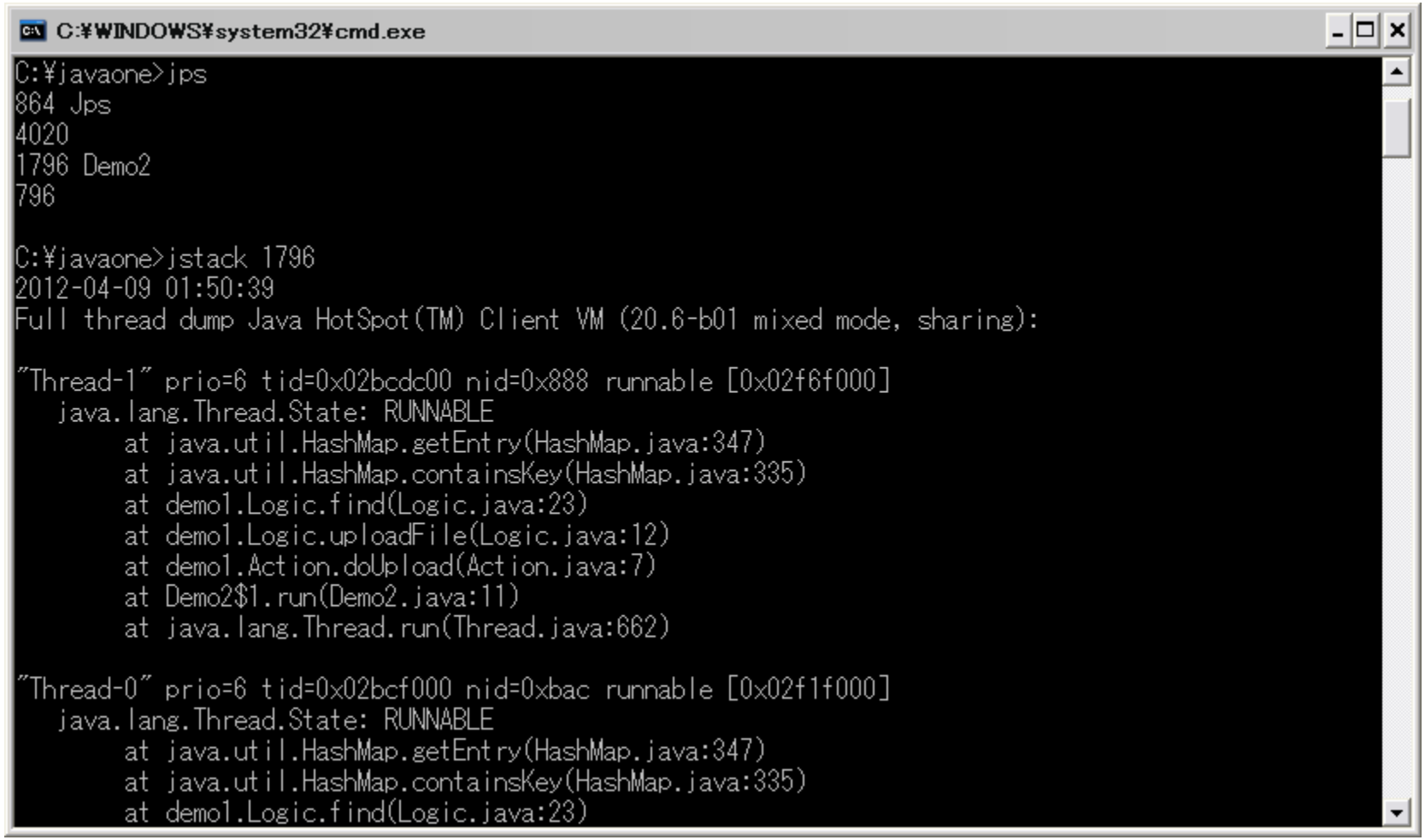
はいはい、スレッドダンプね。

えっと・・・どうやって取りましょうか？

いくつか方法はありますが、
今回は「jstack」コマンドが良いですね。
コマンドラインから jstack [pid] です。

pidを調べるには・・・jpsコマンドですね！
これぐらいなら僕も知ってます。

事例2：処理が終わらない・・・



事例2：処理が終わらない・・・

本当はスレッドのCPU使用率も取るべきなのですが
時間の都合で、ここでは取ったことにしますね。

スレッドごとのCPU使用率を確認するには、
Windowsの場合は、
パフォーマンスモニタ（perfmonコマンド）を
使うと良いですね。

それでCPU使用率を確認したところ、
このコンソールに出ているHashMap.getEntryを
呼び出しているスレッドのCPU使用率が100%だったと。

でも、なぜこんなことが起きるのでしょうか？

事例2：処理が終わらない・・・

HashMapは、複数スレッドで同時にputを行うと
構造が壊れてしまって、
その後、getする時に無限ループが起きるんですよね。

なるほど・・・そうなんですね！
では、ここでsynchronizedなどを入れて
同期処理すれば良いということですね。

せっかくなので、ちょっと「HashMap 無限ループ」で
Web検索してみましょうか。

おっ、出てきましたよ。

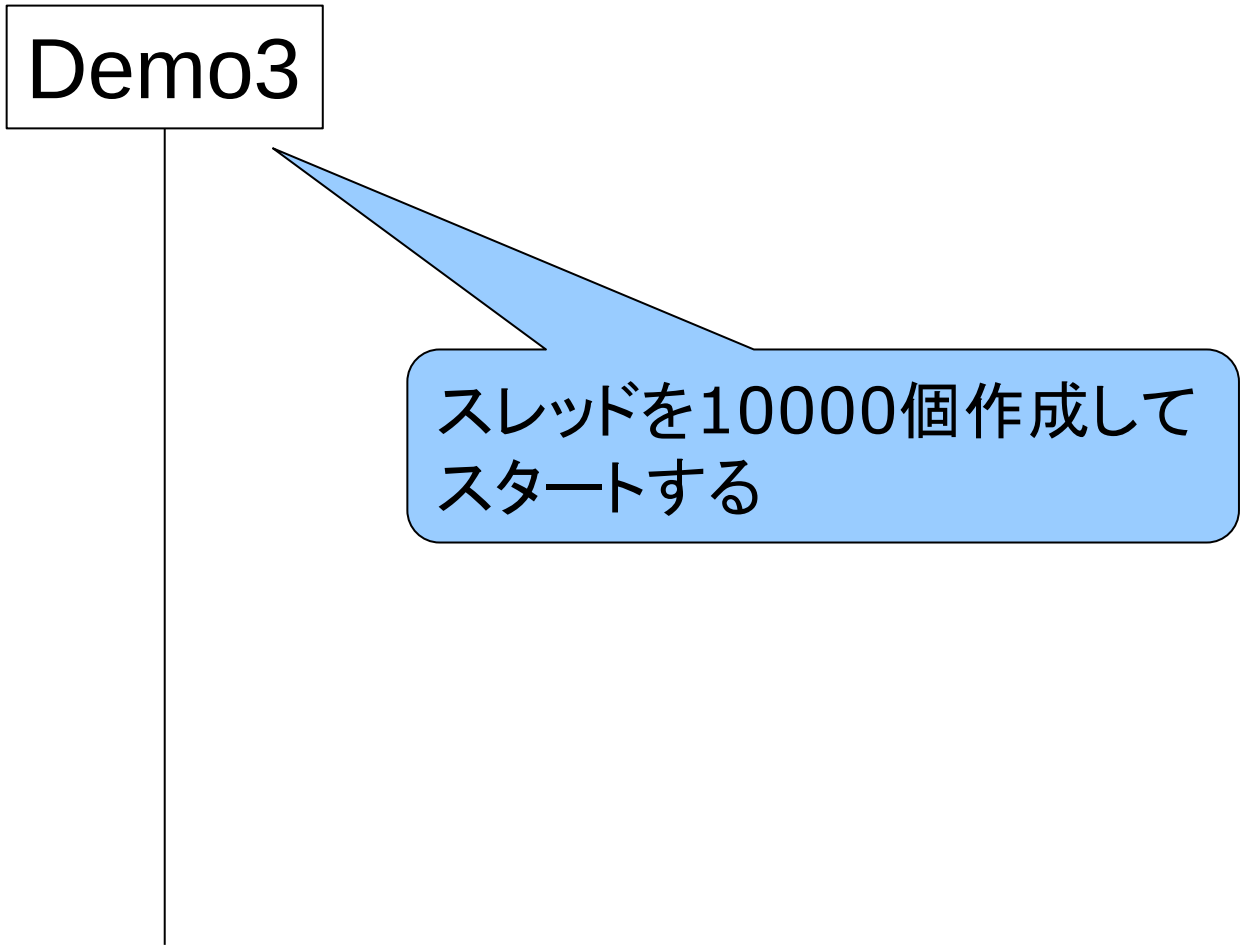
事例2：処理が終わらない・・・

とっても良さそうなサイトですね！

http://d.hatena.ne.jp/cero-t/20091126/1259254839

問題3
OutOfMemoryError: unable to create new native thread

事例3 : unable to create new native thread



事例3 : unable to create new native thread

10000スレッドを起こそうとすしたんですが、
5000ちょっとのスレッドを起こしたところで

OutOfMemoryError: unabe to create new native thread

というエラーが発生して止まってしまいました。

```
Th-5085
Th-5086
Th-5087
Th-5088
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
        at java.lang.Thread.start0(Native Method)
        at java.lang.Thread.start(Thread.java:640)
        at Demo3.main(Demo3.java:14)
#
# There is insufficient memory for the Java Runtime Environment to continue.
# Native memory allocation (malloc) failed to allocate 32756 bytes for ChunkPool::allocate
# An error report file with more information is saved as:
# C:¥javaone¥hs_err_pid1224.log

C:¥javaone>
```

事例3 : unable to create new native thread

OutOfMemoryErrorが発生しているので
こういう時は、ヒープダンプですね！

いや、ちょっと待ってください。
そもそもエラーメッセージには何て書いていますか？

はい、
Out Of Memo Error です！

いや、その後です。
native threadを作成できないと書いていますよね？

事例3 : unable to create new native thread

1. まずは仮想メモリを確認するところからだね。
2. こういう時はヒープダンプだってさっき聞いた。
3. OSのスレッド数制限を確認するべきだ。
4. その他

事例3 : unable to create new native thread

1. まずは仮想メモリを確認するところからだね。
2. ~~こういう時はヒープダンプだってさっき聞いた~~
3. OSのスレッド数制限を確認するべきだ。
4. ~~その他~~

事例3：unable to create new native thread

Windowsでは、スレッド数制限をしているところが
ちょっと見当たらなかったんですよね。
そういう意味で、OSの制限は特にないのかなと。

Linuxの場合は制限があるのですが
私もWindowsのOS制限は、ちょっと分かりません。

ただ、unable to create new native threadが
出る場合、たいていは仮想メモリ不足です。

では、その仮想メモリを確認してみましょうか。
どうやって確認しましょうかね？

Windowsなら、やはりパフォーマンスモニタですね。
コマンドラインから「perfmon」です。

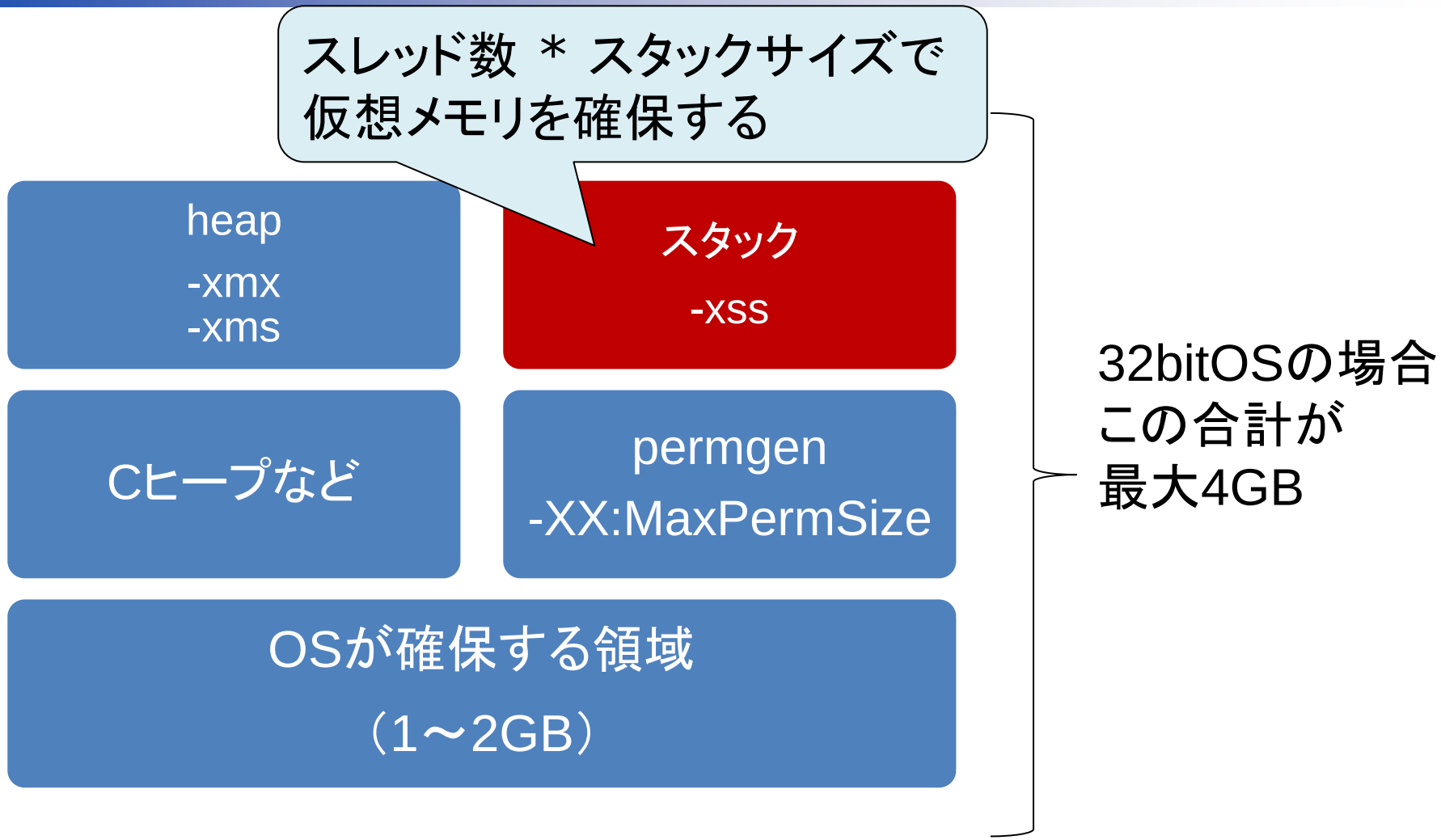
事例3 : unable to create new native thread

それで、カウンタを追加すれば良いんですね。
「Process」の「Virtual Bytes」が、その項目ですね。

これを確認しながら、Demo3を再実行してみたら
Virtual Bytesが2GBを超えたあたりで
OutOfMemoryErrorが発生していることが
分かりました。

なるほど、想定通りですね。
やはり仮想メモリ不足で間違いなさそうです。

事例3 : unable to create new native thread



事例3 : unable to create new native thread

- 対策
  - ① 64bit OSを利用する
    - 仮想メモリの上限がTB級まで上がる。
  - ② 1スレッドあたりのスタックのサイズを絞る
    - 例： -Xss128k
    - 元の4倍程度のスレッドを生成できるようになる。
  - ③ 3GBオプションを利用する（Windowsのみ）
    - OSが利用する領域を2GBから1GBに絞ることができる。
    - 元の1.5～2倍程度のスレッドを生成できるようになる。

事例3 : unable to create new native thread

そもそも10000スレッドを起こす必要があるのか、
という疑問はありますけどね。

ただ、それでも10000スレッド必要だと言う時には
仮想メモリの上限が大きい
64bit OSを利用すべきでしょうね。

JMeterなどの負荷試験ツールを動かす時に
大量スレッドを発生させたいことはあるんじゃないかな、
と思いました。

ひとまず今回のデモでは、-Xss128kで
正しく動くことだけ検証してみました。

1. 実演、トラブルシュート！
2. **トラブルシューターの考え方**
3. トラブルシューターの道具箱
4. トラブルシュート事例紹介

あなたの近くにいませんか？

1. 問題を見て、「ここが怪しい」と思ったら、
   一直線に突っ走る。
2. とりあえず片っ端からパラメータを修正する。
3. 複数の箇所を一気に修正して、
   動くかどうか試してみる。
4. 正しく動くようになったが、
   直った理由が説明できない。
5. かなりテンパっているから、口を挟みにくい。
6. どこから手を付けて良いのか分からず、
   思考停止している。

トラブルシュートの「4K」調査サイクル

トラブルシュートの「4K」調査サイクル

確認
- 問題の状況を確認し、ログ、リソースなどの情報を収集する。
- 問題を再現させる。

仮説
- 思いつく全ての原因を列挙する。
- 思い込みを排除する。

検証
- 仮説を順番に検証する。
- 可能性が高く、時間が掛からないものから検証する。

考察
- なぜ動かなかったのかを論理的に説明する。

1. 実演、トラブルシュート！
2. トラブルシューターの考え方
3. **トラブルシューターの道具箱**
4. トラブルシュート事例紹介

問題調査の対象

1. ハードウェアリソースの確認
2. Javaプロセスの確認
   - ① CPU使用率、メモリ使用状況の確認
   - ② Javaの動作状況の確認
     （スレッドダンプの取得）
   - ③ Javaのメモリ使用状況の確認
     （ヒープダンプの取得）
   - ④ デバッガ、トレーサでの確認
3. ネットワーク状況の確認
4. システムコールの監視

1. ハードウェアリソースの確認

1. Linux
   - top
   - vmstat
2. Windows
   - パフォーマンスモニタ
     （コントロールパネル – 管理ツール）

2－① CPU使用率、メモリ使用状況の確認

1. OS問わず
   - jstat
   - JConsole
   - VisualVM
   - JRockit Mission Control
   - JRCMD
   - GCログ
     - java -Xloggc:**gc.log** -XX:+PrintGCDetails

2 – ② スレッドダンプの取得

1. OS問わず
   - jstack
   - JConsole
   - VisualVM
   - JRockit Mission Control
2. Linux
   - kill -3 コマンド
3. Windows
   - Ctrl + Break

2－③ ヒープダンプの取得

1. OS問わず
   - jmap
   - OOME時の取得
     - -XX:+HeapDumpOnOutOfMemoryError
     - -XX:+HeapDumpOnCtrlBreak (Java5以前)
   - hprof
     - java -agentlib:hprof=format=b,file=xxx
   - JConsole
   - VisualVM
   - JRockit

２－④ デバッガ、トレーサ

1. Eclipse
2. YouDebug - http://youdebug.kenai.com/
3. BTrace - http://kenai.com/projects/btrace

3. ネットワーク状況の確認

1. ネットワークの状態確認
   - ① OS問わず
     - netstat
2. ポートを利用しているプロセスの確認
   - ① Windows
     - netstat –nao（PID）
     - netstat –noba（プロセス名）
   - ② Linux
     - netstat -nap
   - ③ Linux / OS X
     - lsof -i -P

4. システムコールの監視

1. Linux
   - strace
   - SystemTap
2. Solaris
   - truss
   - DTrace

5. その他

1. 環境変数
   - JAVA_TOOL_OPTIONS
     （全てのJavaプロセスにオプションを指定する）
2. 注目のツール
   - JRockit Flight Recorder
   - memleak

6. 私たちが作りました！

1. OS問わず

- ENdoSnipe

1. 実演、トラブルシュート！
2. トラブルシューターの考え方
3. トラブルシューターの道具箱
4. **トラブルシュート事例紹介**

トラブルシュートの事例紹介

1. システムが月に一度ダウン。
   - 有意義なサポートの使い方
2. 結合試験初日、システムが起動しなくなった。
   - フレームワークの使い方には要注意
3. Javaアプレットの画面が真っ白に。
   - 解決の秘策はアプレットの「ログ」
4. Swing + OpenSolarisでトラブル発生。
   - DTraceを使って無限ループを検出
   - x-windowのイベント解析ツールを自作
   - 解決にはBCIを利用

more...

1. **JaTS never ending story**

   http://bit.ly/jatsnovel

   (http://www.acroquest.co.jp/jats_novel/index.html)

2. **Java Trouble Shooting**

   http://bit.ly/acrojts

   (http://www.acroquest.co.jp/webworkshop/JavaTroubleshooting/index.html)

まとめ

- トラブルシューターとは・・・
  ① 確認、仮説、検証、考察の
  4Kサイクルで調査している。
  ② 用途に合わせてトラブルシュートツールを
  使い分けている。
  ③ 自分の仕事の一部を自動化するために、
  トラブルシュートツールを開発している。

ところで・・・

あなたの周りに
トラブルシューターはいますか？

Shameless Advertisement

- もしもトラブルシューターがいないなら・・・

① JaTSにお電話を！
  – ここにいる江里口が駆けつけます！

② ENdoSnipeのご検討を！
  – ここにいる江里口が導入サポートをします！

あなたもトラブルシューターを
目指してみませんか？

Acroquest Technology

Infrastructures Evolution