

常用 JVM 性能监控和分析工具使用介绍

目录

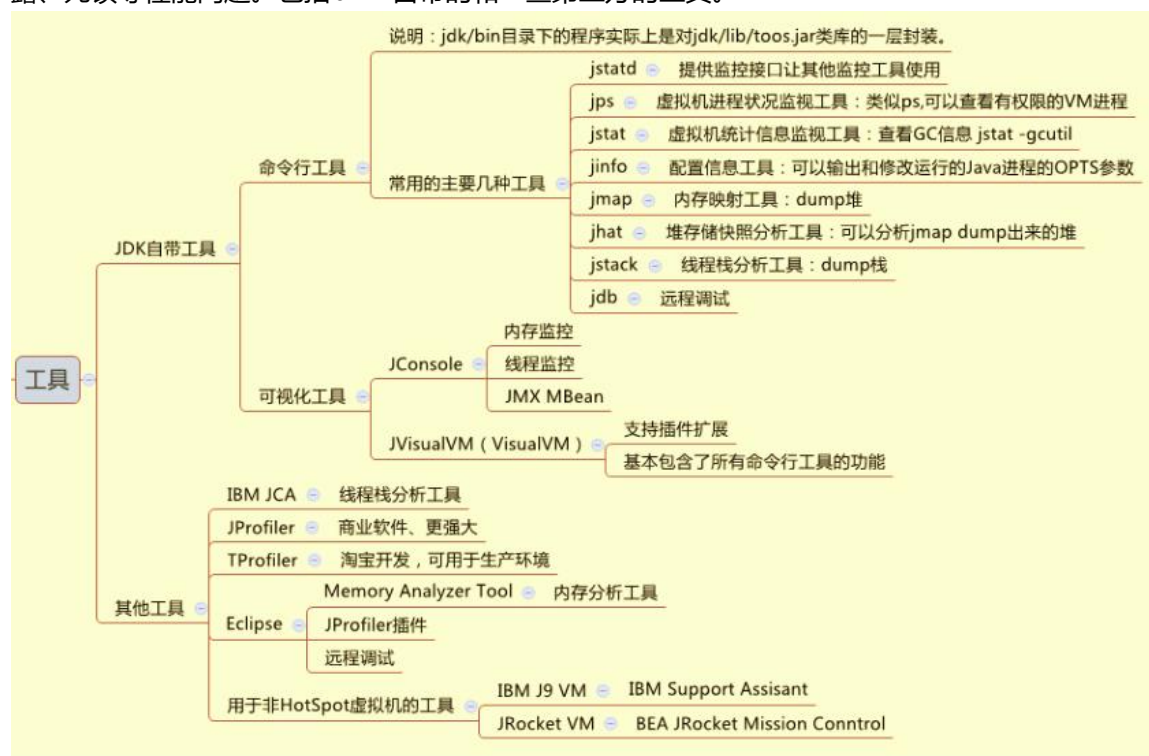
1. 简介	4
2. JDK 自带工具	4
2.1 jstatd	4
2.2 jps	5
2.3 jinfo	6
2.4 jstat	6
2.5 jstack	8
2.6 jmap	10
2.7 jhat	11
2.8 jcmd	13
2.9 jconsole	13
2.10 jvisualvm	13
3. GC 日志	19
4. 其他工具	20
4.1 IBM jca.jar	20
4.2 MAT	20
4.3 JProfiler	28
4.4 TProfiler	28
5. 一些概念和知识	28
5.1 GC Roots	28
5.2 Shallow Heap 和 Retained Heap	29
5.3 Java 中的内存泄露	30
5.4 饥饿和死锁	31
5.5 OQL	32
6. 一些问题	32
6.1 jvisualvm 无法远程内存采样	32
6.2 jmap 等工具报 InvocationTargetException 异常	34
6.3 jstat 遇到的一个问题	34
7. 案例分析	35

7.1	内存泄露	35
7.2	CPU 问题	35
7.3	线程死锁	35
7.4	ClassLoader 类加载泄露	35

常用 JVM 性能监控和分析工具使用介绍

1. 简介

该文档主要是介绍一些常用的 Java 虚拟机性能监控和分析的工具，帮助开发人员能够快速的定位内存泄露、死锁等性能问题。包括 JDK 自带的和一些第三方的工具。



2. JDK 自带工具

2.1 jstatd

1. 简介

jstatd 主要是用来在要监控的机器上创建监控接口，方便其他工具比如 jvisualvm 来使用。

2. 用法

```
usage: jstatd [-nr] [-p port] [-n rminame] [-J option]
```

1) 首先创建安全策略文件 jstatd.all.policy 内容如下：

```
1. grant codebase "file:${java.home}/../lib/tools.jar" {
2. permission java.security.AllPermission;
3. };
```

这里是所有权限，也就是所有用户创建的 JVM 进程都可以被监控。

2) 启动：

```
jstatd -J-Djava.security.policy=jstatd.all.policy
```

options

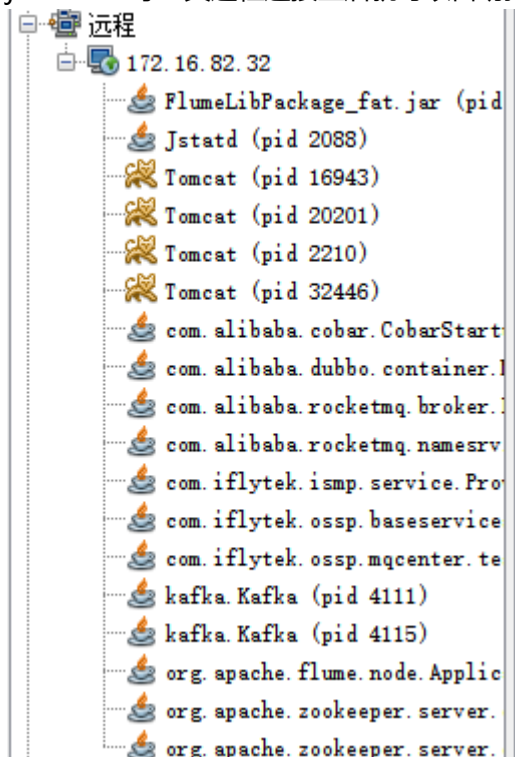
-nr 当一个存在的 RMI Registry 没有找到时，不尝试创建一个内部的 RMI Registry

-p port 端口号，默认为 1099

-n rminame 默认为 JStatRemoteHost；如果多个 jstatd 服务开始在同一台主机上，rminame 唯一确定一个 jstatd 服务

-J JVM 选项

jvisualvm 等工具远程连接上后就可以自动扫描出所有 jvm 进程。



2.2 jps

1. 简介

虚拟机进程状况监视工具：类似 ps, 可以查看有权限的 JVM 进程相关的信息。

2. 用法

```
usage:jps [-q] [-mlvV] [<hostid>][-J option]
```

options

- q 只显示 pid , 不显示 class 名称,jar 文件名和传递给 main 方法的参数
- m 输出传递给 main 方法的参数 , 在嵌入式 jvm 上可能是 null
- v 输出传递给 JVM 的参数
- V 输出通过 flag 是否传递给 JVM 的参数
- hostid 主机
- J JVM 选项

默认不跟任何参数就列出本地的所有 jvm 进程 , 如果需要列出远程服务器上的信息 , 远程服务器需要启动 jstatd。

2.3 jinfo

1. 简介

可以配置和修改运行中 JVM 进程的参数。

2. 用法

```
Usage:
  jinfo [option] <pid>
      (to connect to running process)
  jinfo [option] <executable <core>
      (to connect to a core file)
  jinfo [option] [server_id@]<remote server IP or hostname>
      (to connect to remote debug server)
```

optionas

- flag <name> 输出某个 VM 选项的配置
- flag [+|-]<name> 启用和禁用某个 JVM 选项
- flag <name>=<value> 设置某个 JVM 选项的值
- flags 打印 JVM 进程启动时跟的选项
- sysprops 打印 Java 系统属性
- <no option> 不跟选项则打印上面所有的信息

3. 示例

```
jinfo -flag MaxPermSize 4111
output:
-XX:MaxPermSize=85983232
```

2.4 jstat

1. 简介

JVM 统计监测工具。

2. 用法

```
jstat [generalOption|outputOptions vmid [interval[s|ms] [count]]]
```

vmid 是 Java 虚拟机 ID，在 Linux/Unix 系统上一般就是进程 ID。interval 是采样时间间隔。count 是采样数目。

比如 jstat 最常用用来输出的是 GC 信息：

```
jstat -gcutil 21711 1000 100
```

采样时间间隔为 1s，采样数为 100，输出格式：

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498
0.00	0.00	46.11	1.13	59.76	1447	11.020	3	0.478	11.498

S0，**S1** 两个 survival 区域的使用情况，不会同时 100%

E 年轻代区域使用占比

O 老年代区域使用占比

P 持久代区域使用占比

YGC Yong GC 次数

YGCT Yong GC 的累计时间

FGC Full GC 次数

FGCT Full GC 累计时间

GCT YGCT+FGCT

除了 gcutil 还有很多其他一些统计信息：

- ✧ jstat -class pid:显示加载 class 的数量，及所占空间等信息。
- ✧ jstat -compiler pid:显示 VM 实时编译的数量等信息。
- ✧ jstat -gc pid:可以显示 gc 的信息，查看 gc 的次数，及时间。其中最后五项，分别是 young gc 的次数，young gc 的时间，full gc 的次数，full gc 的时间，gc 的总时间。

- ✧ `jstat -gccapacity`:可以显示, VM 内存中三代 (young,old,perm) 对象的使用和占用大小, 如:
PGCMN 显示的是最小 perm 的内存使用量, PGCMX 显示的是 perm 的内存最大使用量, PGC 是当前新生成的 perm 内存占用量, PC 是但前 perm 内存占用量。其他的可以根据这个类推, OC 是 old 内纯的占用量。
- ✧ `jstat -gcnew pid:new` 对象的信息。
- ✧ `jstat -gcnewcapacity pid:new` 对象的信息及其占用量。
- ✧ `jstat -gcold pid:old` 对象的信息。
- ✧ `jstat -gcoldcapacity pid:old` 对象的信息及其占用量。
- ✧ `jstat -gcpermcapacity pid: perm` 对象的信息及其占用量。
- ✧ `jstat -util pid`:统计 gc 信息统计。
- ✧ `jstat -printcompilation pid`:当前 VM 执行的信息。

`jstat` 适合现场的排查故障或者作为监控的接口。

2.5 jstack

1. 简介

主要用来查看 JVM 进程的线程堆栈的信息, 最常用用法是 dump 线程转储文件, 然后集合一些第三方工具 (如 IBM 的 jca) 来分析线程死锁问题。jstack 的好处是通过堆栈信息可以直接定位到有问题的代码行。

2. 用法

```
Usage:
  jstack [-l] <pid>
    (to connect to running process)
  jstack -F [-m] [-l] <pid>
    (to connect to a hung process)
  jstack [-m] [-l] <executable> <core>
    (to connect to a core file)
  jstack [-m] [-l] [server_id@]<remote server IP or hostname>
    (to connect to a remote debug server)
```

options

- l 输出有关锁的附加信息
- F 当进程无响应(hung 住了)的时候强制的 dump
- m 同时输出 java 和本地的线程帧信息

3. 基本使用

```
jstack -l 4111 > thread.dump
```

输出格式


```

2014-12-17 18:22:08
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.55-b03 mixed mode):

3 "catalina-exec-1900" daemon prio=10 tid=0x00007f6e80654000 nid=0x4526 waiting on condition [0x00007f6de0290000]
3   java.lang.Thread.State: TIMED_WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000007264d8f08> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSynchronizer.java:2082)
    at java.util.concurrent.LinkedBlockingQueue.poll(LinkedBlockingQueue.java:467)
    at org.apache.tomcat.util.threads.TaskQueue.poll(TaskQueue.java:86)
    at org.apache.tomcat.util.threads.TaskQueue.poll(TaskQueue.java:32)
    at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
    at java.lang.Thread.run(Thread.java:745)

3   Locked ownable synchronizers:
    - None

```

"catalina-exec-1900" 是线程名，这里是 Tomcat 线程池里的线程。这提醒我们编码的时候最好给自己启动的线程一个有意义的名字，这样方便排查问题。

daemon 说明这个线程是守护线程，守护线程会随着主线程退出而退出。而对于非守护线程，进程必须要等所有非守护线程结束才能结束。

prio=10 线程的优先级

tid 线程 id

重点需要关注的是后面的线程状态，有以下几种情况：

- ✧ 死锁，**Deadlock (重点关注)**
- ✧ 执行中，Runnable 该状态表示线程具备所有运行条件，在运行队列中准备操作系统的调度，或者正在运行。
- ✧ 等待资源，**Waiting on condition (重点关注)**
- ✧ 等待获取监视器，**Waiting on monitor entry (重点关注)** 线程处于等待获取监视器的状态
- ✧ 对象等待中，in Object.wait() 或 TIMED_WAITING 线程获取到监视器后发现条件不满足调用 wait() 方法，放弃监视器。
- ✧ 暂停，Suspended
- ✧ 阻塞，**Blocked (重点关注)**
- ✧ 停止，Parked

注意，即使出现大量上面标红的需要重点关注的状态，也不代表程序一定有问题，比如 Tomcat 线程池空闲时期绝大部分线程都会处于 Waiting on conditon 状态，还是需要集合代码来分析。

一旦发现问题可以通过下面的详细堆栈信息顺藤摸瓜找出有问题的代码。虽然有时候不会直接死锁，但是严重的锁竞争还是会导致很差的性能，比如搜索 dump 文件发现大量的线程处于：**waiting to lock**

```

at com.mchange.v2.resourcepool.BasicResourcePool.checkInResource(BasicResourcePool.java:64)
- waiting to lock <0x0000000763c01dc0> (a com.mchange.v2.resourcepool.BasicResourcePool)

```

这时候可以尝试搜索 **locked <0x0000000763c01dc0>** 看到到底是哪个线程占用了锁。

一般来说手动的查找 dump 文件效率太低，通常会结合一些第三方工具来使用，比如 IBM 的一个 jca.jar.

2.6 jmap

1. 简介

打印出某个 java 进程（使用 pid）内存内的，所有‘对象’的情况（如：产生那些对象，及其数量）。可以输出所有内存中对象的工具。更常用的是将 VM 中的 heap 以二进制输出成文件（.hprof），结合 MAT 或 jhat 一起使用，能够以图像的形式直观的展示当前内存是否有问题。

2. 用法

Usage:

```
jmap [option] <pid>
      (to connect to running process)
jmap [option] <executable <core>
      (to connect to a core file)
jmap [option] [server_id@]<remote server IP or hostname>
      (to connect to remote debug server)
```

options

-heap 打印堆的一些基本信息

-histo[:live] 统计各个类的实例数和占用 bytes，live 这个子选项表示只输出存活的对象统计信息。-

hist:live 可以强制 FullGC。

num	#instances	#bytes	class name
1:	38240	4904368	<methodKlass>
2:	38240	4490168	<constMethodKlass>
3:	3070	3258024	<constantPoolKlass>
4:	3070	3071456	<instanceKlassKlass>
5:	2771	1973216	<constantPoolCacheKlass>
6:	17926	1829288	[C
7:	3050	1045448	<methodDataKlass>
8:	6063	903304	[B
9:	24958	598992	java.util.concurrent.ConcurrentSkipListMap\$Node
10:	24878	597072	java.lang.Double
11:	17834	428016	java.lang.String
12:	3322	393856	java.lang.Class
13:	4778	382240	kafka.network.RequestChannel\$Request
14:	4722	377760	kafka.server.KafkaApis\$DelayedFetch
15:	4722	339984	kafka.api.FetchRequest
16:	5496	324560	[Ljava.lang.Object;
17:	5972	312936	[[I
18:	12396	297504	java.util.concurrent.ConcurrentSkipListMap\$Index
19:	5048	274344	[S
20:	4222	135104	java.util.concurrent.ConcurrentHashMap\$HashEntry
21:	5373	128952	java.util.concurrent.atomic.AtomicLong

class name 说明：

B byte
C char
D double
F float
I int
J long
Z boolean
[数组, 如[I 表示 int[]]
[L+类名 其他对象

-permstat 打印进程的类加载器和类加载器加载的持久代对象信息, 输出: 类加载器名称、对象是否存活 (不可靠)、对象地址、父类加载器、已加载的类大小等信息

class_loader	classes	bytes	parent_loader	alive?	type
<bootstrap>	1396	8136416	null	live	<internal>
0x00000000d0377ea8	4	13744	null	dead	javax/management/remote/rmi/NoCallStackClassLoader@0
x00000000fb337560					
0x00000000d0377ef8	2	32128	null	dead	javax/management/remote/rmi/NoCallStackClassLoader@0
x00000000fb337560					
0x00000000d04b7450	0	0	null	dead	sun/misc/Launcher\$ExtClassLoader@0x00000000fafa04a8
0x00000000d0407fb8	0	0	0x00000000d0377e58	dead	java/util/ResourceBundle\$RBClassLoader@0x000
00000fb8350e8					
0x00000000d038bbf8	1	3256	null	dead	sun/reflect/DelegatingClassLoader@0x00000000fae4f960
0x00000000d0377e58	1876	12446304	0x00000000d04b7450	dead	sun/misc/Launcher\$AppClassLoader@0x0
000000fb00c968					

-finalizerinfo 打印正在等待回收的对象的信息

-dump:<dump-options> dump 堆转储文件

dump-options

live 只 dump 存活的对象, 默认是 dump 所有的对象

format=b dump 二进制格式文件, 即 jhat 和 jvisualvm 可以分析的 hprof 格式

file=<file> dump 出来文件的存储路径

-F 当进程无响应的时候, 强制 dump, 这时候-dump 不支持 live 子选项

-J JVM 选项, 如-J-d64 64 机器下执行的时候使用。

示例

dmp hprof 格式的堆转储文件。

```
jmap -dump:format=b,file=heap.hprof 3024
```

2.7 jhat

1. 简介

用来分析 jmap dump 下来的堆转储文件。

2. 用法

```
Usage: jhat [-stack <bool>] [-refs <bool>] [-port <port>] [-baseline <file>] [-debug <int>]
[-version] [-h|-help] <file>
```

```
jhat -port 7777 -J-Xmx512m <heap dump file>
```

浏览器输入地址:7777 可以看到 HTML 格式的分析结果，拖到页面最底部，可以看到有以下几个查询：

Other Queries

- [All classes including platform](#)
- [Show all members of the rootset](#)
- [Show instance counts for all classes \(including platform\)](#)
- [Show instance counts for all classes \(excluding platform\)](#)
- [Show heap histogram](#)
- [Show finalizer summary](#)
- [Execute Object Query Language \(OQL\) query](#)

依次的作用是：

✧ 显示堆中包含的所有类，点进去可以查看每个类的引用、被引用的情况。

Package <Default Package>

```
class Fastjson ASM DataVersion 8 [0x75f663f80]
class Fastjson ASM RemotingCommand 1 [0x75f613540]
class Fastjson ASM TopicConfigSerializeWrapper 7 [0x75f664018]
class Fastjson ASM TopicConfig 10 [0x75f663de0]
class Fastjson ASM Field DataVersion timestatmp 9 [0x75f663ee8]
class Fastjson ASM Field RemotingCommand code 2 [0x75f613438]
class Fastjson ASM Field RemotingCommand flag 5 [0x75f613270]
class Fastjson ASM Field RemotingCommand opaque 4 [0x75f613308]
class Fastjson ASM Field RemotingCommand remark 6 [0x75f6130f8]
class Fastjson ASM Field RemotingCommand version 3 [0x75f6133a0]
class Fastjson ASM Field TopicConfig perm 12 [0x75f663cb0]
class Fastjson ASM Field TopicConfig readQueueNums 14 [0x75f663b10]
class Fastjson ASM Field TopicConfig topicName 11 [0x75f663d48]
```

✧ 从根集能够引用到的对象

所谓根集就是 GC Roots

- ✧ 显示所有类的实例数，包括平台的类，就是 JDK 自己类。
- ✧ 显示不包括平台类的实例数
- ✧ 堆实例的分布表

Heap Histogram

All Classes (excluding platform)

Class	Instance Count	Total Size
class [I	309	63631068
class java.util.PriorityQueue\$Node	539632	19426752
class java.util.concurrent.locks.AbstractQueuedSynchronizer\$Node	271198	9763128
class [B	7045	4424321
class java.util.concurrent.ConcurrentLinkedQueue\$Node	275534	4408544
class java.util.concurrent.LinkedBlockingQueue\$Node	268224	4291584
class [C	24685	4071426
class [Ljava.lang.Object;	2516	1404112
class [Ljava.nio.channels.SelectionKey;	88	656960
class java.lang.Class	2272	345344
class java.lang.reflect.Field	3560	345320
class java.lang.String	18419	294704
class java.util.LinkedList\$ListNode	8725	279200
class com.alibaba.fastjson.parser.ParseContext	6397	204704
class java.util.LinkedList	8406	201744
class [S	3031	177664
class java.nio.DirectByteBuffer	2461	155043
class [Ljava.util.HashMap\$Entry;	912	131112

[I、[B 的含义参见 jmap 中的说明。

- ✧ 垃圾收集的信息，类似 jmap -finalizerinfo
- ✧ 执行 QQL 语句

2.8 jcmd

1. 简介

Native Memory Tracking

2. 用法

- 1) JVM 参数设置：-XX:NativeMemoryTracking=summary or -XX:NativeMemoryTracking=detail.
- 2) 设置 baseline：jcmd <pid> VM.native_memory baseline.
- 3) 跟踪内存变化：jcmd <pid> VM.native_memory detail.diff

2.9 jconsole

3. 简介

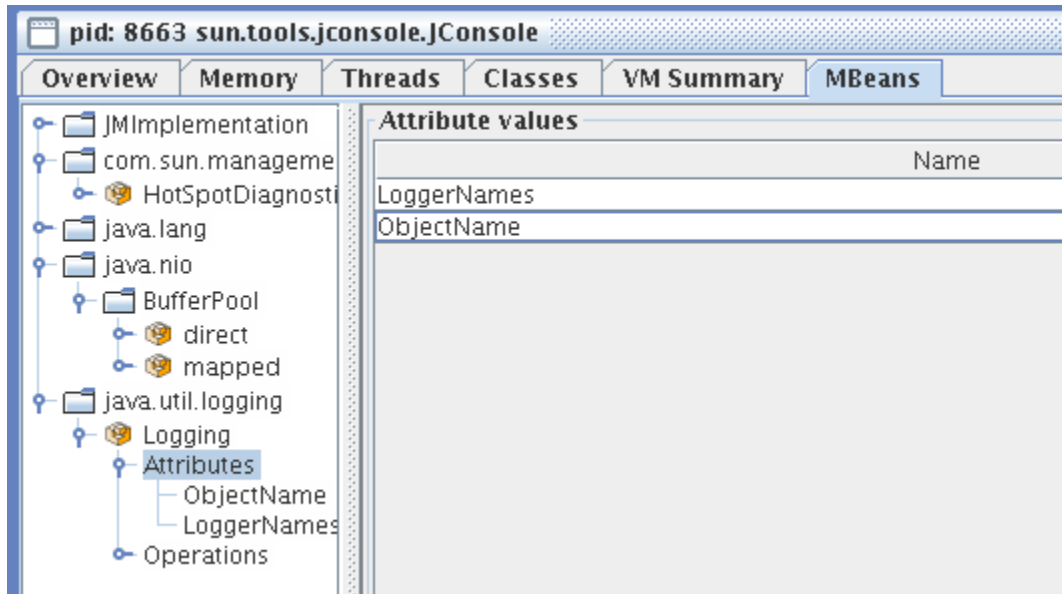
jconsole 主要是用来监视堆内存的使用情况，但其功能 jvisualvm 都支持或者通过安装插件支持。

2.10 jvisualvm

1. 简介

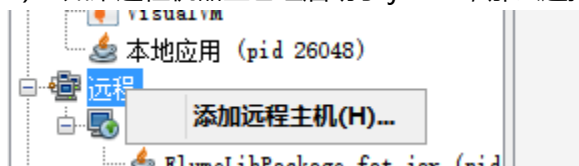
可视化的、集成的 JVM 性能监控和分析工具，支持插件扩展，基本可以包括其他所有工具的功能。

所以不做详细介绍，需要提一下就是它支持对 MBeans 对象的监控。



4. 启动

- 1) 如果远程机器上已经启动了 jstatd，那么选择**添加远程主机**就可以自动扫描出 JVM 进程

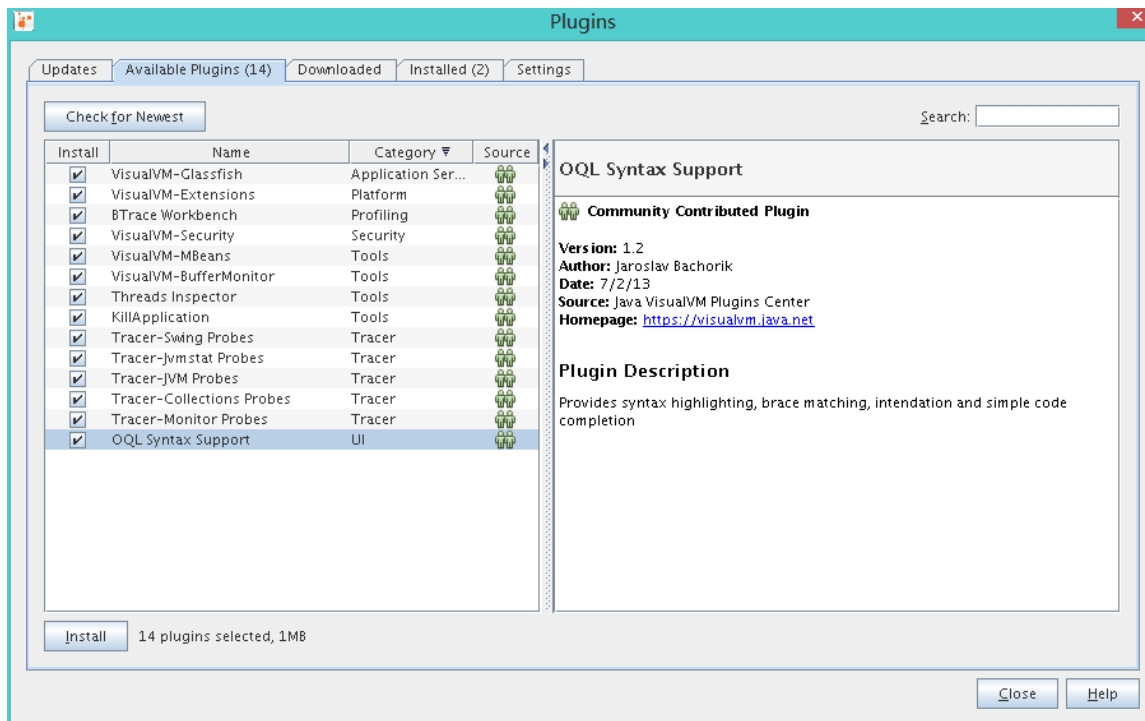


- 2) 如果是 Tomcat 也可以不启动 jstatd，直接在 **JAVA_OPTS** 里加上下面的参数：

```
-Dcom.sun.management.jmxremote=true -Dcom.sun.management.jmxremote.port=9991 -  
Djava.rmi.server.hostname=172.16.82.31 -Dcom.sun.management.jmxremote.ssl=false -  
Dcom.sun.management.jmxremote.authenticate=false
```

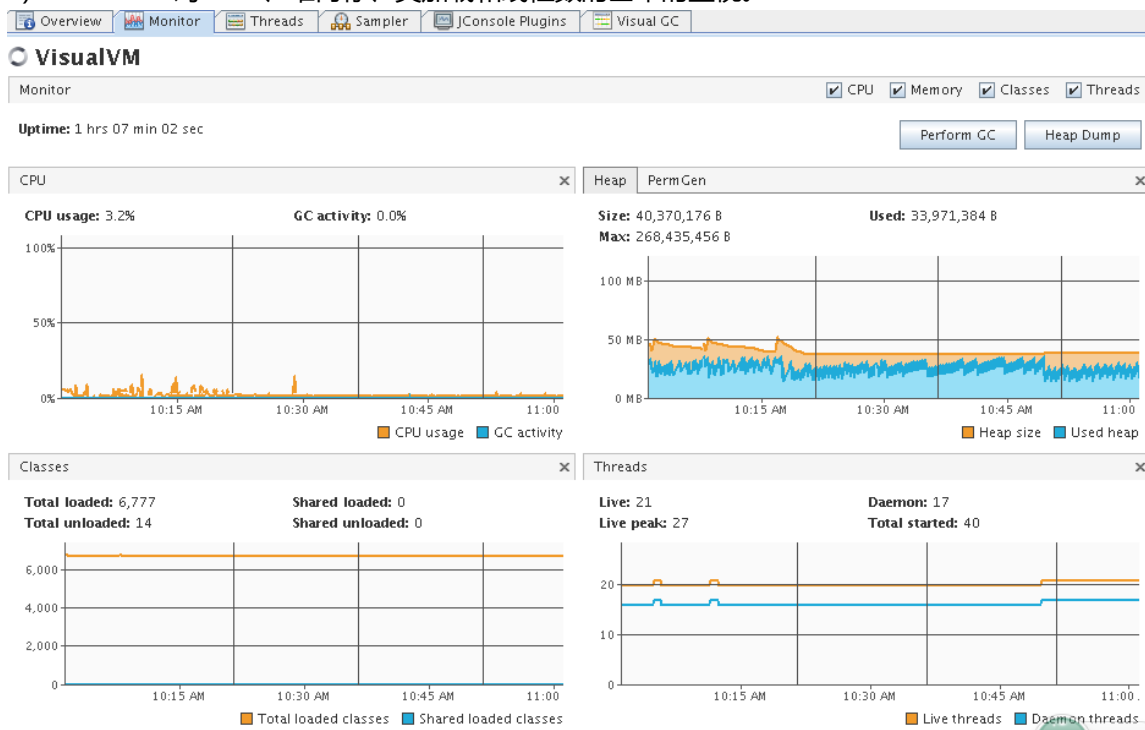
5. 插件

visualvm 的强大在于它支持插件扩展，通过插件它能包含几乎所有其他 JDK 自带工具的功能。常用的插件有 Visual GC 和 VisualVM JConsole



6. 基本使用

- 1) Overview JVM 进程的基本信息，jps 能看到的的信息这里都能看到。
- 2) Monitor 对 CPU、堆内存、类加载和线程数的基本的监视。

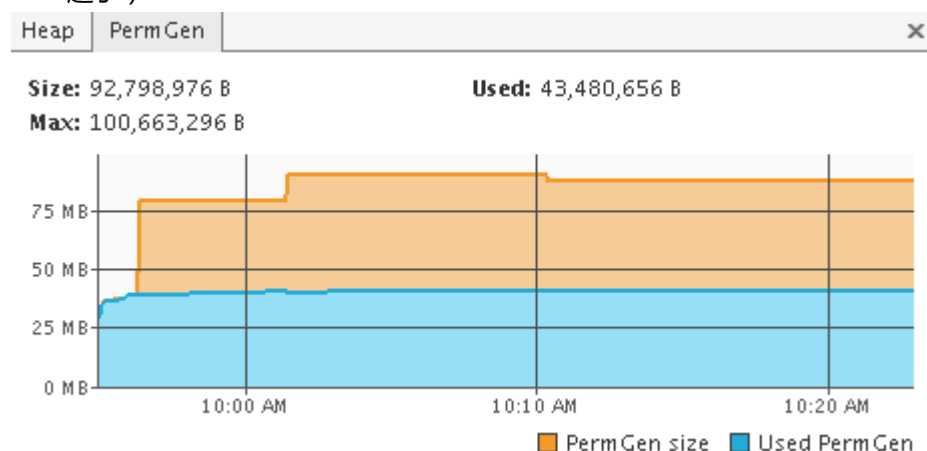


通过 Monitor 能够发现一些非常明显的问题：

- a) **CPU** 消耗特别大，或者 GC activity 异常，正常情况这个值应该很小，偶尔有毛刺。如果一直占比很大，或者毛刺很多说明 GC 的压力比较大。
- b) **Heap** 内存的曲线如果一直上涨不一定是内存泄露，要看发生 GC 的时候能不能回收，但如果增长的太快说明程序还是有问题的，内存使用有问题，瞬间耗光堆内存会导致进程僵死。还有如果曲线一直增长坡拉的很长，只有 FullGC 才能使其降下来，说明程序内存使用的也有问题，要不对象的生命周期太长了，要不就是分配的都是大对象，直接分配到老年代了。

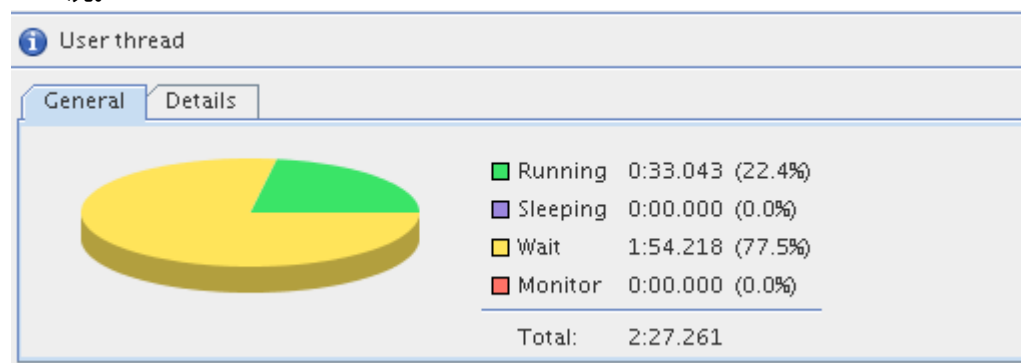
一般应用正常曲线应该如上图所示的那样：较长的时间上是平缓的，局部很多毛刺（Yong GC）。

- c) **PermGen** 的内存使用（蓝色的线）一直再涨，那肯定是内存泄露了（很大可能是类加载的代码出问题了）



- d) **Classes** 如果加载的类的数量再不停的涨，那 3) PermGen 的曲线肯定也再涨
- e) **Threads** 正常的应用这个曲线应该是平稳的，对于使用线程池的应用应该也不能波动太大，否则性能可能并不会太好。

- 3) **Threads** 页中能够看到所有线程的运行状况，在 Running,Sleeping,Wait 等各个状态上的时间占比情况。



General	Details
0:00.000: Started	
0:00.000: Wait	
0:01.231: Running	
0:03.224: Wait	
0:05.228: Running	
0:08.223: Wait	
0:24.219: Running	
0:25.220: Wait	

大量的线程在 Wait 或 Running 都不一定能说明问题，要结合应用的情况来分析，比如部署在 Tomcat 中的 Servlet 应用，峰值期间可能很多线程在 Running，但空闲期可能大部分线程池中的线程都处于 Wait 状态。最好结合 jstack 对 dump 的分析。

- 4) Sampler 采样器是 jvisualvm 中最重要的功能。利用它可以实时的观察 JVM 进程 CPU 和内存占用的变化情况，已经被谁占用了。

CPU 采用

CPU samples Thread CPU Time			
Snapshot		Thread Dump	
Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
org.openide.util.RequestProcessor\$Processor.run ()	73.9%	1,391 ms	0.000 ms
org.netbeans.core.TimableEventQueue.dispatchEvent ()	1.0%	192 ms	192 ms
org.openide.util.lookup.DelegatingStorage.<init> ()	5.3%	99.5 ms	99.5 ms
org.netbeans.swing.tabcontrol.TabbedContainer.paint ()	5.3%	99.3 ms	99.3 ms
org.openide.util.RequestProcessor\$TickTac.cancel ()	5.3%	99.2 ms	99.2 ms
org.netbeans.core.ui.warmup.MenuWarmUpTask\$NbWindowsAdapter.run ()	0.0%	0.000 ms	0.000 ms
org.openide.modules.Places.getUserDirectory ()	0.0%	0.000 ms	0.000 ms
org.openide.util.lookup.ProxyLookup.lookup ()	0.0%	0.000 ms	0.000 ms
org.openide.util.lookup.AbstractLookup.lookup ()	0.0%	0.000 ms	0.000 ms
org.openide.util.lookup.AbstractLookup.lookupItem ()	0.0%	0.000 ms	0.000 ms
org.openide.util.lookup.AbstractLookup.enterStorage ()	0.0%	0.000 ms	0.000 ms
org.netbeans.core.TimableEventQueue.done ()	0.0%	0.000 ms	0.000 ms

Self time：指方法自己消耗的 wall-clock 时间，不包括调用的其他方法中的耗时。

Self time(CPU)：指方法自己消耗的处理器时间，不包括 waiting、sleeping 等状态的时间。

Self time[%]：点击 Self time 或 Self time(CPU)这一栏就是其对应的占所总时间的百分比。

因为是采样，所以这几个值都是近似值。

内存采样

Heap histogram

PermGen histogram

Per thread allocations

Deltas

Snapshot

Perform GC

Heap Dump

Classes: +176

Instances: ~244,078

Bytes: ~18,584,608

Class Name	Bytes [%] ▼	Bytes	Instances
com.sun.tools.visualvm.sampler.memory.MemoryView\$DeltaClassInfo		+596,480	+18,640
com.sun.tools.visualvm.attach.HeapHistogramImpl\$ClassInfoImpl		+552,120	+13,803
long[]		+536,696	-114
java.util.HashMap\$Entry		+426,848	+13,339
javax.swing.text.AbstractDocument\$LeafElement		+422,200	+10,555
javax.swing.text.GapContent\$MarkData		+339,280	+8,482
java.lang.Double		+265,632	+11,068
javax.swing.text.GapContent\$StickyPosition		+249,480	+10,395
javax.swing.JLabel		+212,856	+543
java.lang.Object[]		+179,296	+604
java.util.HashMap\$Entry[]		+175,072	+2,369
javax.swing.JPanel		+154,112	+448
java.lang.Integer		+120,176	+7,511
java.util.HashMap		+105,168	+2,191
java.awt.GridBagConstraints		+76,496	+683
javax.swing.text.AbstractDocument\$AbstractElement[]		+71,824	-41
java.security.ProtectionDomain[]		+56,720	+1,312

除非有非常明显的内存泄露，不然 Heap histogram 中大部分情况占比最高的都是 byte[]、int[] 这些，并不一定有什么参考价值。Per thread allocations 的帮助可能会更大点。

- 5) Visual GC Plugin 如果通过 Monitor 发现 GC activity 比较异常，可以通过这个插件来实时观察堆内存中各个区域的变化（使用 jconsole 也可以）。

VisualVM



- 6) Profiler

在 Profiler 页签中，VisualVM 提供了程序运行期间方法级别的 CPU 执行时间分析和内存分析。选择“CPU”和“内存”一个按钮点击开始分析，然后切换到程序中对应用程序进行操作，VisualVM 会分析这段时间内 CPU 或内存的情况。但 Profiling 对程序的性能的影响比较大，不要在生产环境使用这个功能。

7) BTrace 插件

这个插件利用 HotSwap 技术能够在不停止应用的情况下，动态的向程序中加入原本并不存在的调试代码。

功能很强大，使用门槛比较高，项目地址：<https://kenai.com/projects/btrace>

8) 堆 dump 分析

jvisualvm 也可以分析 jmap dump 下来的 hprof 格式的堆内存。

3. GC 日志

GC 日志是排查 GC 方面最好的工具。

```
-XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:<filename>
```

其他参数：

-verbose.gc 开关可显示 GC 的操作内容。打开它，可以显示最忙和最空闲收集行为发生的时间、收集前后的内存大小、收集需要的时间等。

-XX:+ PrintHeapAtGC 了解堆的更详细的信息。

-XX:+ PrintGCApplicationStoppedTime 输出 GC 造成应用暂停的时间

-XX:+ PrintGCDateStamps GC 发生的日期

部分参数可以在运行时通过 jinfo 设置：

```
jinfo -flag +PrintGCTimeStamps <pid>  
jinfo -flag +PrintGCDetails <pid>
```

但 Xloggc,PrintGCApplicationStoppedTim, PrintHeapAtG 不行，Tomcat 默认 GC 日志输出到 catalina.out。

```

D:\>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:MaxTenuringThreshold=15 -X
[GC [DefNew: 6991K->686K(9216K), 0.0017849 secs] 6991K->686K(19456K), 0.0021369 secs] [Times: user=0.00 sys=0.00, re
[GC [DefNew: 7080K->686K(9216K), 0.0007505 secs] 7080K->686K(19456K), 0.0010383 secs] [Times: user=0.00 sys=0.00, re
[GC [DefNew: 2048K->0K(9216K), 0.0009062 secs] 2048K->0K(9216K), 0.0038406 secs] 2734K->2734K(10240K), 0.0000000 secs]
[Full GC [Tenured: 2734K->2718K(10240K), 0.0053720 secs] 2734K->2718K(19456K), 0.0000000 secs] 3210K->3202K(21248K), 0.0000000
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at ossp.demo.gc.TestGC.testAllocation(TestGC.java:61)
    at ossp.demo.gc.TestGC.main(TestGC.java:17)
Heap:
def new generation      total 9216K, used 220K [0x00000000f9a00000, 0x00000000fa400000, 0x00000000fa400000>
eden space 8192K,  2% used [0x00000000f9a00000, 0x00000000f9a371e8, 0x00000000fa200000>
from space 1024K,  0% used [0x00000000fa300000, 0x00000000fa300000, 0x00000000fa400000>
to space 1024K,  0% used [0x00000000fa200000, 0x00000000fa200000, 0x00000000fa300000>
tenured generation      total 10240K, used 2718K [0x00000000fa400000, 0x00000000fae00000, 0x00000000fae00000>
the space 10240K,    26% used [0x00000000fa400000, 0x00000000fa6a7b10, 0x00000000fa67c000, 0x00000000fae00000>
compacting perm gen      total 21248K, used 3210K [0x00000000fae00000, 0x00000000fc2c0000, 0x0000000010000000>
the space 21248K,    15% used [0x00000000fae00000, 0x00000000fb122e90, 0x00000000fb123000, 0x00000000fc2c0000>
No shared spaces configured.

```

一次Full GC Tenured指GC发生的区域是老年代

方括号内部的指GC前该内存区域已使用容量->GC后该内存区域已使用容量（该内存区域总容量）

方括号之外的指GC前Java堆已使用容量->GC后Java堆使用容量（Java堆总容量）

GC 日志分析工具：GCHisto, GCLogViewer, Hppjmeter, GCViewer, garbagecat

4. 其他工具

4.1 IBM jca.jar

一个非常简单的可视化线程堆栈分析工具，导入 jstack dump 出的文件就可以了。

下载地址：<http://public.dhe.ibm.com/software/websphere/appserv/support/tools/jca/jca457.jar>

The screenshot displays the IBM jca.jar tool interface. On the left, a list of threads is shown with columns for Name, State, Native ID, Method, and Stack. Most threads are in a 'Waiting' state. On the right, the 'Thread Status Analysis' section provides a summary of thread states:

Status	Number of Threads : 3039	Percentage
Deadlock	0	0 (%)
Runnable	47	2 (%)
Waiting on condition	2928	96 (%)
Waiting on monitor	0	0 (%)
Suspended	0	0 (%)
Object.wait()	38	1 (%)
Blocked	26	1 (%)
Parked	0	0 (%)

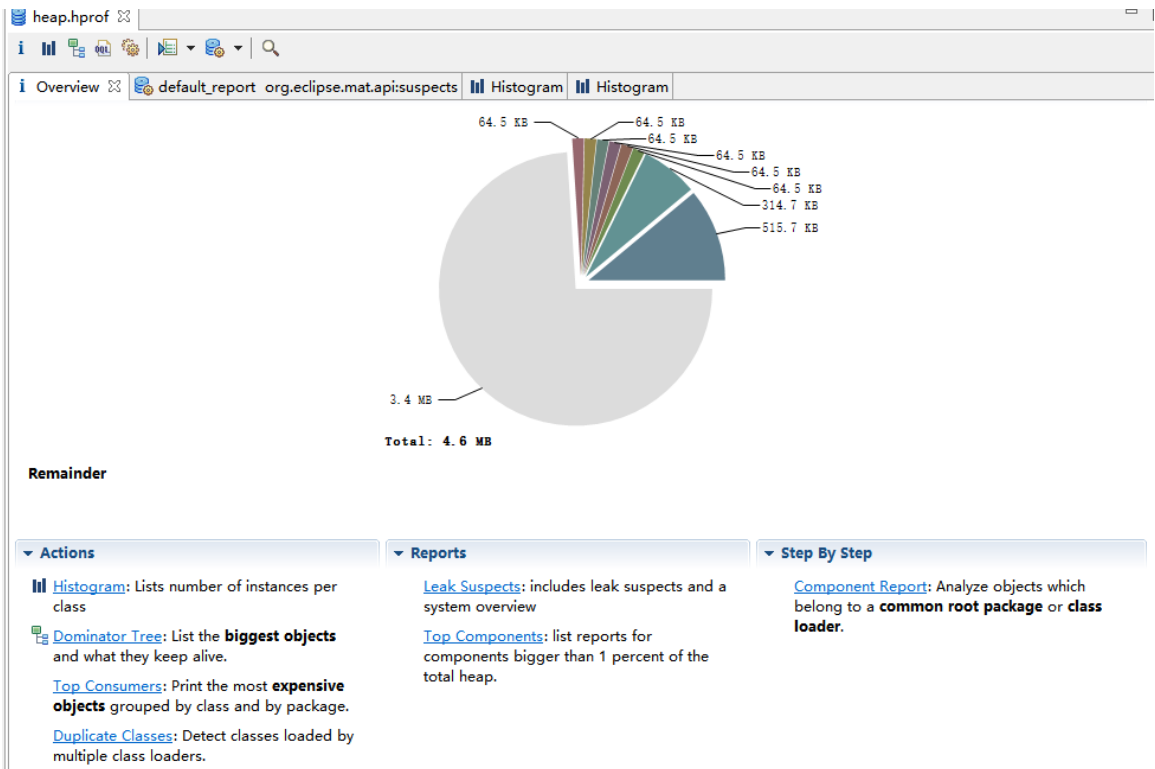
Below this, the 'Thread Method Analysis' section shows the methods being executed by the threads:

Method Name	Number of Threads : 3039	Percentage
sun.misc.Unsafe.park(Native Method)	2893	95 (%)
java.lang.Thread.sleep(Native Method)	56	2 (%)
java.lang.Object.wait(Native Method)	38	1 (%)
NO JAVA STACK	33	1 (%)
sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)	12	0 (%)

4.2 MAT

Eclipse Memory Analyzer 用来分析 Java 堆转储文件的工具。项目地址：<http://www.eclipse.org/mat/>

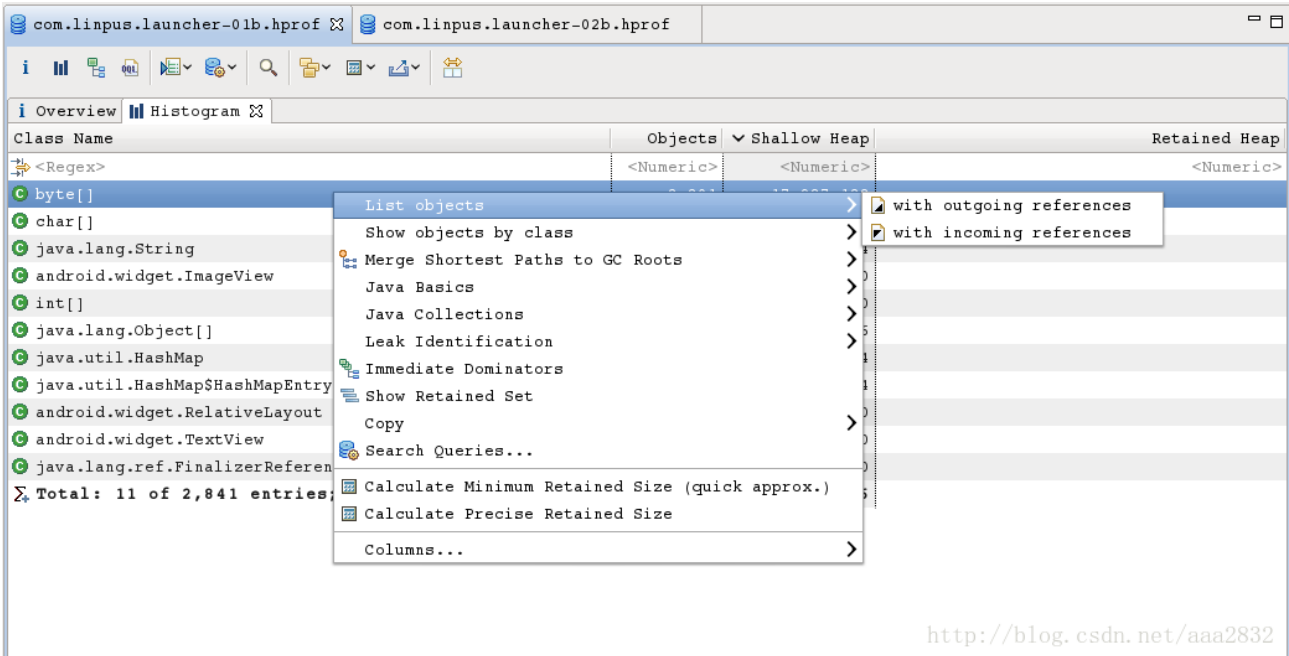
相比 jhat MAT 功能要强大很多，对信息提供了更直观的可视化描述。



主要功能：









1. Histogram

用的最多的功能是 Histogram,点击 Actions 下的 Histogram 项将得到 Histogram 结果：




它按类名将所有的实例对象列出来，可以点击表头进行排序,在表的第一行可以输入正则表达式来匹配结

果：

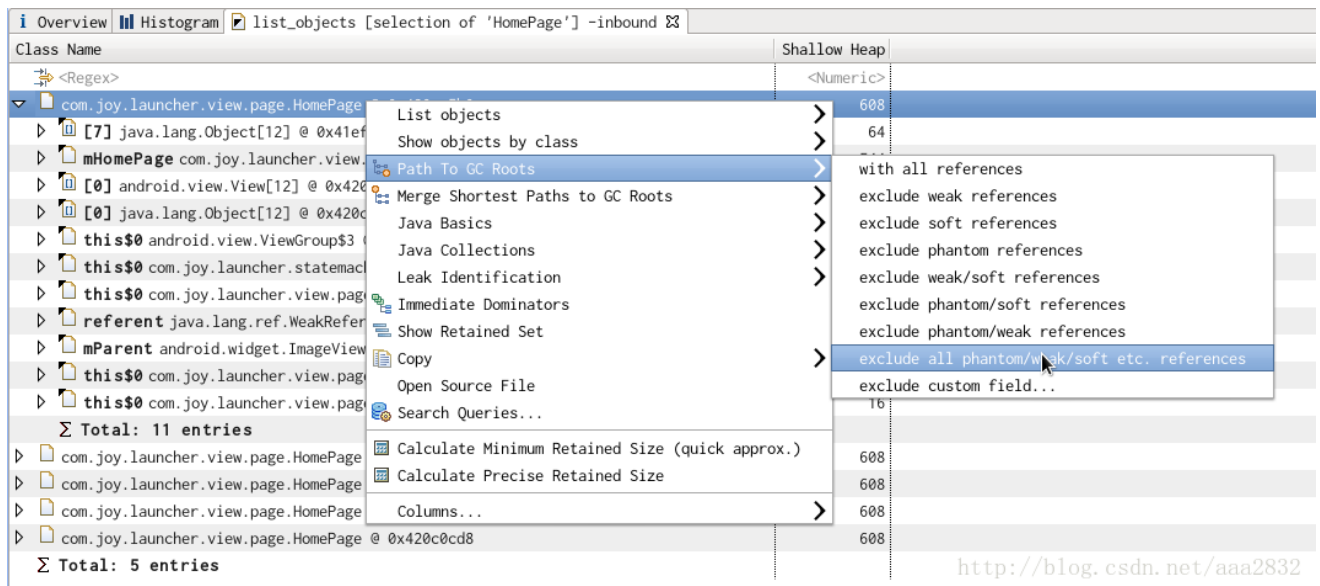
Overview	Histogram	
Class Name	Objects	Shallow Heap
 .*HomePage.*	<Numeric>	<Numeric>
 com.joy.launcher.view.page.HomePage	5	3,040
 com.joy.launcher.view.page.HomePageContainer	5	2,720
 com.joy.launcher.model.page.HomePageModel	5	120
 com.joy.launcher.view.page.HomePage\$2	5	80
 com.joy.launcher.view.page.HomePage\$1	5	80
 com.joy.launcher.model.page.HomePageModel\$HomePageType	2	32
 com.joy.launcher.model.page.HomePageModel\$HomePageType[]	1	24
Σ Total: 7 entries (3,840 filtered)		

在某一项上右键打开菜单选择 list objects ->with incoming refs 将列出该类的实例：

Overview	Histogram	list_objects [selection of 'HomePage'] -inbound
Class Name		Shallow Heap
 <Regex>		<Numeric>
com.joy.launcher.view.page.HomePage @ 0x420ca5b0		608
[7] java.lang.Object[12] @ 0x41ef4d20		64
mHomePage com.joy.launcher.view.page.HomePageContainer @ 0x420c9e40		544
[0] android.view.View[12] @ 0x420ca128		64
[0] java.lang.Object[12] @ 0x420ca548		64
this\$0 android.view.ViewGroup\$3 @ 0x420ca838		16
this\$0 com.joy.launcher.statemachine.wrapper.StatedViewGroup\$1 @ 0x420cac30		16
this\$0 com.joy.launcher.view.page.CellPage\$1 @ 0x420cae30		16
referent java.lang.ref.WeakReference @ 0x420caf10		24
mParent android.widget.ImageView @ 0x420caf50		432
this\$0 com.joy.launcher.view.page.HomePage\$1 @ 0x420cb290		16
this\$0 com.joy.launcher.view.page.HomePage\$2 @ 0x420cb2a0		16
Σ Total: 11 entries		
com.joy.launcher.view.page.HomePage @ 0x420c8da0		608
com.joy.launcher.view.page.HomePage @ 0x420c2ae8		608
com.joy.launcher.view.page.HomePage @ 0x420c1cb0		608
com.joy.launcher.view.page.HomePage @ 0x420c0cd8		608
Σ Total: 5 entries		

它展示了对象间的引用关系，比如展开后的第一个子项表示这个 HomePage(0x420ca5b0)被 HomePageContainer(0x420c9e40)中的 mHomePage 属性所引用。

快速找出某个实例没被释放的原因，可以右键 Path to GC Roots-->exclude all phantom/weak/soft etc. reference



得到的结果是：

Class Name	Shallow Heap
<Regex>	<Numeric>
com.joy.launcher.view.page.HomePage @ 0x420ca5b0	608
[7] java.lang.Object[12] @ 0x41ef4d20	64
mHomePage com.joy.launcher.view.page.HomePageContainer @ 0x420c9e40	544
[5] android.view.View[12] @ 0x41edefc0	64
mChildren com.joy.launcher.view.viewport.HomeViewport @ 0x41eded58	576
[1] java.lang.Object[135] @ 0x42399b50	552
array java.util.ArrayList @ 0x41f264f8	24
mListenerList com.joy.launcher.PreferenceManager @ 0x41f264e8	16
mInstance class com.joy.launcher.PreferenceManager @ 0x41f26428 System Class	8
Σ Total: 2 entries	

http://blog.csdn.net/aaa2832

从表中可以看出 PreferenceManager -> ... -> HomePage 这条线路就引用着这个 HomePage 实例。用这个方法可以快速找到某个对象的 GC Root。.

2. Histogram 对比

为查找内存泄漏，通常需要两个 Dump 结果作对比，打开 Navigator History 面板，将两个表的 Histogram 结果都添加到 Compare Basket 中去：

Overview Histogram

Class Name	Objects	Shallow Heap
. *HomePage.*	<Numeric>	<Numeric>
com.joy.launcher.view.page.HomePage	5	3,040
com.joy.launcher.view.page.HomePageContainer	5	2,720
com.joy.launcher.model.page.HomePageModel	5	120
com.joy.launcher.view.page.HomePage\$2	5	80
com.joy.launcher.view.page.HomePage\$1	5	80
com.joy.launcher.model.page.HomePageModel\$HomePageType	2	32
com.joy.launcher.model.page.HomePageModel\$HomePageType[]	1	24
Total: 7 entries (3.840 filtered)	28	6.096

LogCat Console Inspector Call Hierarchy Navigation History Compare Basket

OverviewPane

histogr Activate

Add to Compare Basket

Close

Close Branch

Remove from List

<http://blog.csdn.net/aaa2832>

添加好后，打开 Compare Basket 面板，得到结果：

Compare Basket

Results to be compared	Heap Dump
histogram	/tmp/android5836882618414665539.hprof
histogram	/tmp/android3901468162456371956.hprof

击右上角的!按钮，将得到对比结果：

Overview Histogram Compared Tables Compared Tables

Class Name	Objects #0	Objects #1	Shallow Heap #0	Shallow Heap #1
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
java.lang.String	17,231	17,281	413,544	414,744
char[]	15,910	15,961	1,519,576	1,529,496
java.util.HashMap\$HashMapEntry	7,731	7,760	185,544	186,240
int[]	5,666	5,697	322,280	323,640
java.util.HashMap	3,943	3,979	189,264	190,992
java.lang.ref.FinalizerReference	3,020	3,672	120,800	146,880
java.lang.Object[]	3,398	3,434	214,816	216,720
java.lang.Class	2,841	2,845	66,704	66,712
java.lang.Integer	2,495	2,497	39,920	39,952
Total: 9 of 2,845 entries; 2,836 more	115,891	117,762	23,530,856	23,623,800

<http://blog.csdn.net/aaa2832>

注意，上面这个对比结果不利于查找差异，可以调整对比选项：

Class Name	Objects #0	Objects #1-#0	Shallow Heap #0	Shallow Heap #1
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
byte[]	7,232,184	17,232,184	894,536	892,360
char[]	11,734	11,737	323,280	322,632
java.lang.String	13,470	13,443	167,152	167,152
int[]	3,237	3,237	159,744	159,744
android.widget.ImageView	384	384	136,440	136,440
java.util.HashMap\$HashMapEntry	5,685	5,685	88,100	88,100

再把对比的结果排序，就可得到直观的对比结果：

Class Name	Objects #0	Objects #1-#0	Shallow Heap #0	Shallow Heap #1-#0
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
byte[]	2,170	-123	21,305,704	+363,272
android.content.res.Configuration	96	+4	9,216	+384
android.content.res.Resources	3	+2	240	+160
android.util.LongSparseArray	13	+6	312	+144
android.util.DisplayMetrics	5	+2	320	+128
android.util.TypedValue	24	+3	960	+120
java.util.Locale	35	+4	840	+96
android.content.res.AssetManager	3	+2	120	+80
android.content.res.XmlBlock[]	3	+2	96	+64
android.content.res.StringBlock[]	3	+2	64	+48
android.view.ViewRootImpl\$W	1	+1	32	+32
android.util.SparseArray	24	+1	576	+24

也可以对比两个对象集合，方法与此类似，都是将两个 Dump 结果中的对象集合添加到 Compare Basket 中去对比。找出差异后用 Histogram 查询的方法找出 GC Root，定位到具体的某个对象上。

3. Dominator Tree

列出了堆中最大的对象，可以根据 Shallow Heap 和 Retained Heap 的大小来排序。第二层级的节点表示当被第一层级的节点所引用到的对象，当第一层级对象被回收时，这些对象也将被回收。这个工具可以帮助我们定位对象间的引用情况，垃圾回收时候的引用依赖关系。

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
sun.misc.Launcher\$AppClassLoader @ 0x7d7045610	80	810,768	48.96%
sun.util.resources.TimeZoneNames @ 0x7d75c3ef0	40	112,008	6.76%
class java.io.ObjectStreamClass\$Caches @ 0x7d7526898 System Class	16	75,408	4.55%
class java.lang.System @ 0x7d7000d10 System Class	32	35,824	2.16%
sun.security.provider.Sun @ 0x7d73c4ac8	104	31,352	1.89%
class sun.util.calendar.ZoneInfo @ 0x7d7566f98 System Class	72	29,112	1.76%
java.io.PrintStream @ 0x7d7020c10	32	25,064	1.51%
class java.nio.charset.Charset @ 0x7d7009740 System Class	40	20,128	1.22%
java.lang.Thread @ 0x7d745eca8 RMI TCP Connection(1)-192.168.57.46 Threa	104	18,448	1.11%
com.sun.jmx.mbeanserver.DefaultMXBeanMappingFactory\$CompositeMapping	48	16,712	1.01%
com.sun.jmx.mbeanserver.PerInterface @ 0x7d7319c20	40	15,704	0.95%
class sun.util.calendar.ZoneInfoFile @ 0x7d756a1d8 System Class	72	15,344	0.93%
class java.lang.Package @ 0x7d705a3e8 System Class	16	14,928	0.90%
class java.lang.ref.Finalizer @ 0x7d7001950 System Class	16	13,792	0.83%
class java.io.File @ 0x7d7026e40 System Class	32	13,352	0.81%
com.sun.jmx.mbeanserver.JmxMBeanServer @ 0x7d71e9780	40	11,008	0.66%
sun.misc.Launcher\$ExtClassLoader @ 0x7d7034c18	80	10,336	0.62%

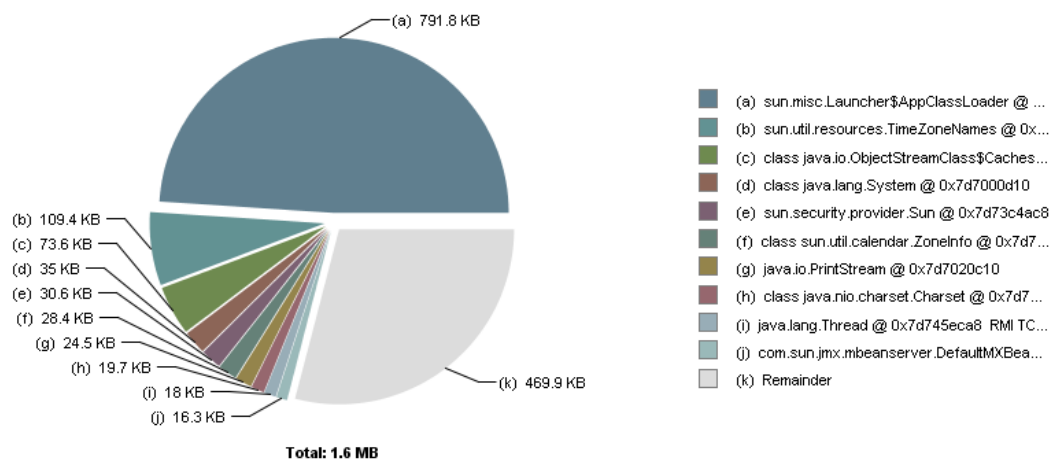
4. Top Consumers

根据类或包来分组的统计堆中最大的对象。

[Top Consumers](#)

Top Consumers

▼ Biggest Objects (Overview)



▼ Biggest Top-Level Dominator Classes

Label	Number of Objects	Used Heap Size	Retained Heap Size	Retained Heap, %
sun.misc.Launcher\$AppClassLoader	1	80	810,768	48.96%
java.lang.Class	1,244	12,552	377,016	22.77%
sun.util.resources.TimeZoneNames	1	40	112,008	6.76%
java.lang.String	762	18,288	59,208	3.58%
com.sun.jmx.mbeanserver.PerInterface	12	480	54,984	3.32%
sun.security.provider.Sun	1	104	31,352	1.89%
com.sun.jmx.mbeanserver.DefaultMXBeanMappingFactory\$CompositeMapping	7	336	28,736	1.74%
java.io.PrintStream	1	32	25,064	1.51%
java.lang.Thread	8	832	21,696	1.31%
Σ Total: 9 entries	2,037	32,744	1,520,832	

Package	Retained Heap	Retained Heap, %	# Top Dominators
<all>	1,656,016	100.00%	2,646
└─ sun	1,093,728	66.05%	458
└─ misc	846,248	51.10%	71
└─ Launcher\$AppClassLoader	810,776	48.96%	2
└─ util	170,584	10.30%	68
└─ resources	112,280	6.78%	9
└─ TimeZoneNames	112,008	6.76%	2
└─ calendar	45,616	2.75%	14
└─ ZoneInfo	29,112	1.76%	1
└─ Σ Total: 2 entries	157,896		23
└─ security	37,640	2.27%	30
└─ provider	32,896	1.99%	11
└─ Sun	31,368	1.89%	2
└─ rmi	19,080	1.15%	69
└─ Σ Total: 4 entries	1,073,552		238

5. Leak Suspects

报告一些可能存在内存泄露的可疑点。

▼ Problem Suspect 1

The classloader/component "**sun.misc.Launcher\$AppClassLoader @ 0x7d7045610**" occupies **810,768 (48.96%)** bytes. The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "<system class loader>".

Keywords

java.lang.Object[]

sun.misc.Launcher\$AppClassLoader @ 0x7d7045610

[Details »](#)

▼ Problem Suspect 2

1,244 instances of "**java.lang.Class**", loaded by "<system class loader>" occupy **377,016 (22.77%)** bytes.

Biggest instances:

- class java.io.ObjectStreamClass\$Caches @ 0x7d7526898 - 75,408 (4.55%) bytes.
- class java.lang.System @ 0x7d7000d10 - 35,824 (2.16%) bytes.
- class sun.util.calendar.ZoneInfo @ 0x7d7566f98 - 29,112 (1.76%) bytes.
- class java.nio.charset.Charset @ 0x7d7009740 - 20,128 (1.22%) bytes.

但注意：存在问题经常不代表内存泄露，尤其是 dump 文件很小的时候，一些系统的类会占很大部分，说明不了什么。一般应该结合出问题的前后的 dump 来分析，如果出现自己写的代码，可疑性更高点。

6. Duplicate Classes

用来检测一个类被多个 ClassLoader 加载了，ClassLoader 加载过多类通常是很隐蔽的一个内存泄露点。

Overview duplicate_classes

Class Name	Count	Defined Classes	No. of Instances
<Regex>	<Numeric>	<Numeric>	<Numeric>
Serializer_1	2		
com.alibaba.fastjson.util.ASMClassLoader @ 0x713c78e50		11	8
com.alibaba.fastjson.util.ASMClassLoader @ 0x7114a6968		3	3
Σ Total: 2 entries			
Serializer_2	2		
Serializer_3	2		
ch.qos.logback.classic.BasicConfigurator	2		
ch.qos.logback.classic.Level	2		
ch.qos.logback.classic.Logger	2		
ch.qos.logback.classic.LoggerContext	2		
ch.qos.logback.classic.PatternLayout	2		
ch.qos.logback.classic.boolex.JaninoEventEvaluator	2		
ch.qos.logback.classic.encoder.PatternLayoutEncoder	2		
ch.qos.logback.classic.filter.LevelFilter	2		

7. OQL 查询

支持 OQL 查询，进行一些更高级的分析。如：

```
SELECT toString(s) AS Value, s.@usedHeapSize AS "Shallow Size", s.@retainedHeapSize AS "Retained Size" FROM java.lang.String s
```

```
SELECT * FROM java.lang.String s WHERE toString(s) LIKE ".*172.16"
```

4.3 JProfiler

4.4 TProfiler

参看调研文档 https://192.168.75.168:8888/svn/YHZ_OSSP2.2/Trunk/Project/03.Study/tprofiler

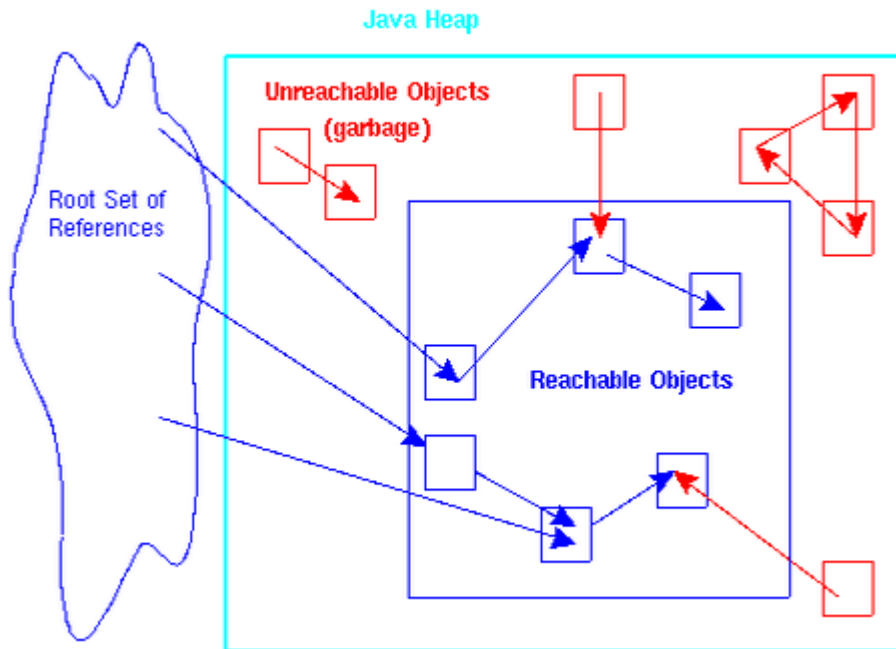
5. 一些概念和知识

5.1 GC Roots

现代垃圾收集器的算法基本都是一种叫可达性分析的算法，也就是通过一系列的称为 GC Roots 的对象作为起始点，从这些起始点开始向下搜索，搜索所走的路径称为引用链（Reference Chain），当一个对象到 GC Root 没有任何引用链相连的时候，就认为这个对象不可用。

GC Roots 对象包括下面几种：

- ❖ 虚拟机栈（栈帧中的本地变量表）引用的对象。
- ❖ 方法区中静态类属性引用的对象。
- ❖ 方法区中常量引用的对象。
- ❖ 本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。



5.2 Shallow Heap 和 Retained Heap

在使用 jhat 或 MAT 工具分析 Heap 的时候经常会看到这两个概念。

Shallow Heap

指对象自身所占用的内存大小，不包含其引用的对象所占的内存大小。

1、数组类型

数组元素对象所占内存的大小总和。

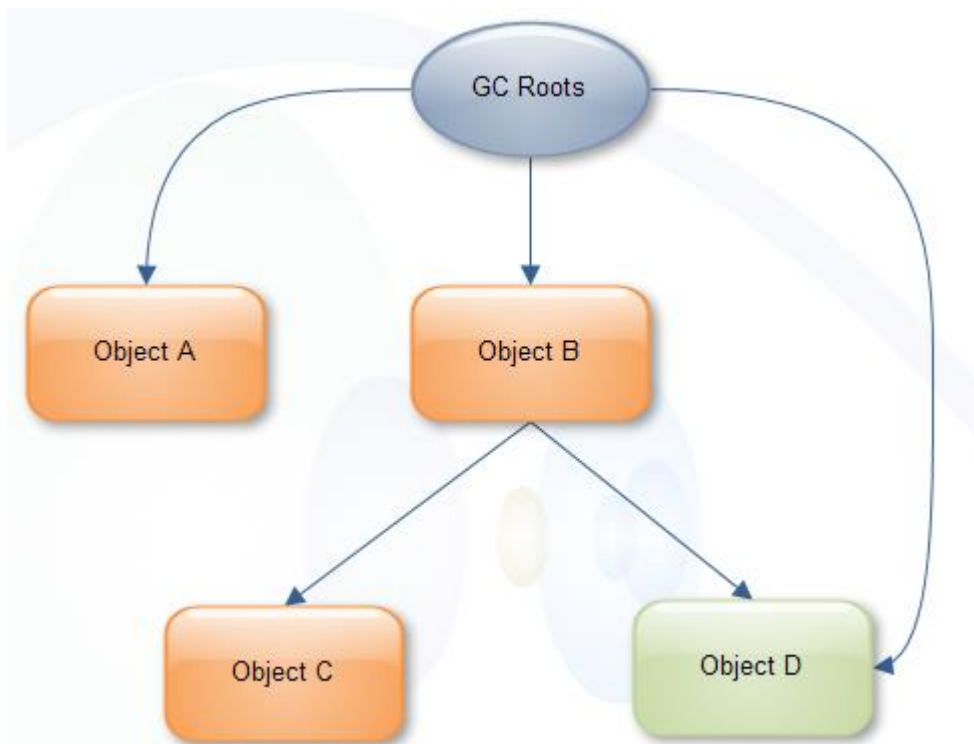
2、非数组类型

对象与它所有的成员变量大小的总和。当然这里面还会包括一些 java 语言特性的数据存储单元。

Retained Heap

换句话说，Retained Heap 是指当前对象被 GC 后，从 Heap 上总共能释放掉的内存。即当前对象大小 + 当前对象可直接或间接引用到的对象的大小总和。(间接引用的含义：A->B->C, C 就是间接引用)，不过要排除被 GC Roots 直接或间接引用的对象，因为他们暂时不会被当做垃圾被回收。

举例说明：



GC Roots 直接引用了 A 和 B 两个对象

A 对象的 Retained Size=A 对象的 Shallow Size

B 对象的 Retained Size=B 对象的 Shallow Size + C 对象的 Shallow Size

这里不包括 D 对象，因为 D 对象被 GC Roots 直接引用。

5.3 Java 中的内存泄露

严格来说只有**对象不会再被程序用到了（自己的代码无法再引用到了）**，但是**GC 又不能回收他们的情况才叫内存泄露**。但实际情况很多时候一些不太好的实践（或疏忽）会导致对象的生命周期变得很长甚至导致 OOM，也可以叫做宽泛意义上的“内存泄露”。

1. 静态字段保持了对对象的引用。

```
public class Leaker {  
    private static final Map<String, Object> CACHE = new HashMap<String, Object>();  
  
    // Keep adding until failure.  
    public static void addToCache(String key, Object value) { Leaker.CACHE.put(key, value); }  
}
```

即使 list 里的对象都会用到，但是不停的往里面放对象显然是有问题的。但严格来说这不能叫内存泄露。

2. 对很长的字符串进行String.intern()操作。

```
String str=readString();
```

```
str.intern();
```

intern 操作增加了到这个长字符串对象的引用链。

3. 没有关闭打开的流（文件、网络等）

```
try {  
    BufferedReader br = new BufferedReader(new FileReader(inputFile));  
    ...  
    ...  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

4. 没有关闭 TCP 连接

```
try {  
    Connection conn = ConnectionFactory.getConnection();  
    ...  
    ...  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

3 和 4 实际可能并不会导致内存出问题，而是连接问题（叫连接泄露），因为流或 TCP 连接一般都实现了 finalize 方法，GC 第一次没法回收，但 finalizer thread 会执行 finalize 方法，在后续的 GC 周期中对象还是会被回收的。

5. 垃圾收集器无法（或几乎）清理的地方

- 1) 本地方法（native methods）分配的内存区域
- 2) 类加载器加载大量的类。

6. 在一些 web 程序中将对象存储在应用程序或会话级别的对象中，导致对象生命周期很长。

```
getServletContext().setAttribute("SOME_MAP", map);  
session.setAttribute("SOME_MAP", map);
```

参看：<http://stackoverflow.com/questions/6470651/creating-a-memory-leak-with-java>

5.4 饥饿和死锁

饥饿：一个线程无限期的阻塞，无法进展。

死锁：一组线程等待一个不可能发生的条件从而都阻塞了。跟饿死不同，饿死可能只是一个线程。

5.5 OQL

一种对象查询语言，可以用类似 SQL 的语句查询 Java 堆 dump 文件。


参考：

<http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.mat.ui.help%2Freference%2Foglsyntax.html>

6. 一些问题

6.1 jvisualvm 无法远程内存采样

工具本身的限制，不支持远程的内存抽样。

 Tomcat (pid 20201)

抽样器

抽样：



CPU



内存



停止

状态：

抽样处于非活动状态

摘要

CPU 抽样：

可用。按“CPU”按钮开始收集性能数据。

内存抽样：

不可用。不支持远程应用程序。

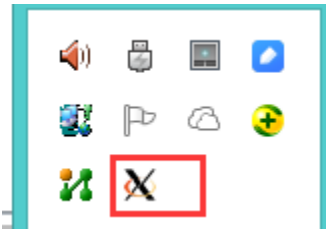
解决办法

使用 SSH+X Server 的方式，在服务器（Linux）上运行 jvisualvm，本地（Windows）显示界面。

SSH 使用 PuTTY 下载地址：<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>

X Server 使用 Xming 下载地址：<http://sourceforge.net/projects/xming/>

- 1) 本地（Windows）安装 Xming，一般默认安装就可以了，安装后在任务栏可以看到一个 X 图标说明 X Server 启动成功了。



- 2) 服务器（Linux）上不需要完成的安装 X11 System，但是需要安装 **xauth**

```
yum install xauth
```


3) 服务端配置 ssh 支持 X11 Forwarding

```
vim /etc/ssh/sshd_config
```

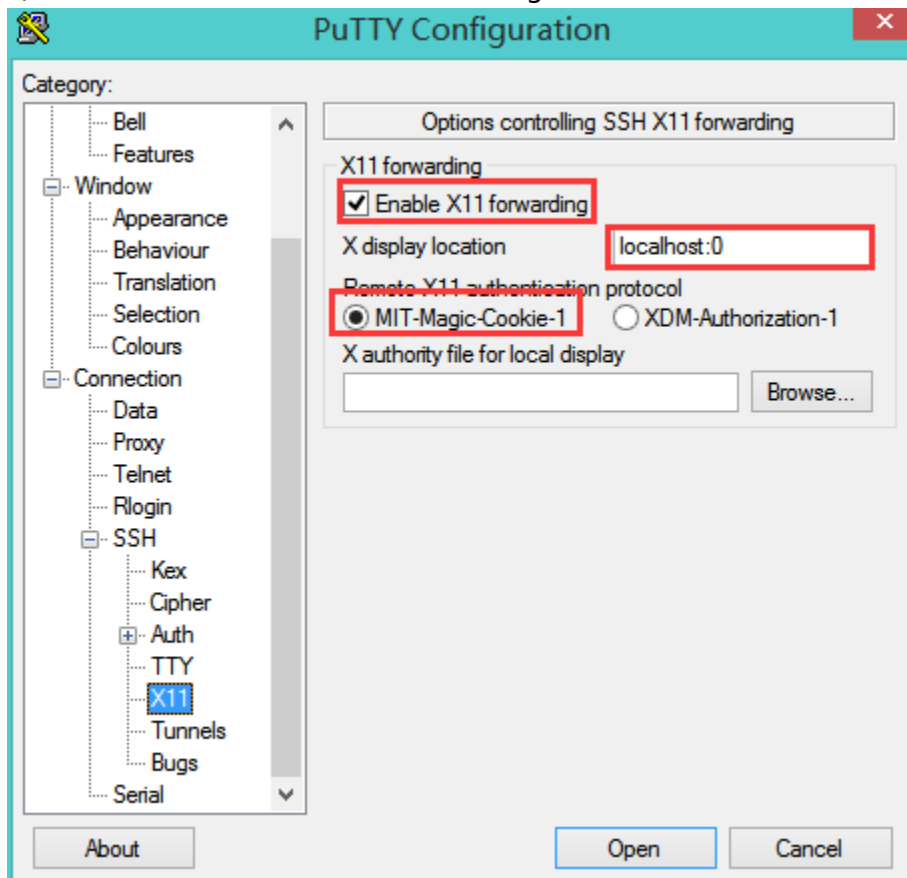
加上

```
X11Forwarding yes
```

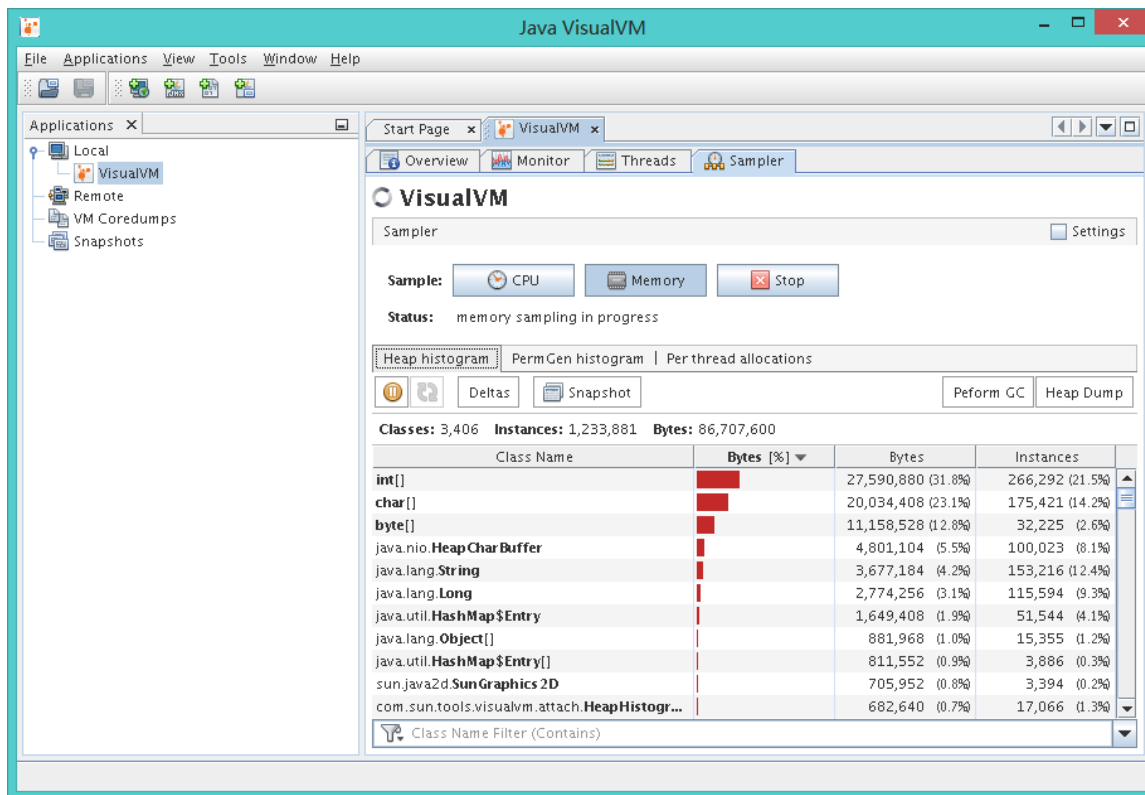
重启 sshd

```
service sshd restart
```

4) 本地启动 PuTTY, 配置 X11 Forwarding :



ssh 登录后启动 jvisualvm , 本地就出现了可视化界面。内存采样都很多通过远程连接不可用的功能都可以了。



6.2 jmap 等工具报 InvocationTargetException 异常

```
[iflyweb@h0082032 tomcat-inputbase-8083]$ jmap -heap 4111
Attaching to process ID 4111, please wait...
Exception in thread "main" java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:616)
    at sun.tools.jmap.JMap.runTool(JMap.java:196)
    at sun.tools.jmap.JMap.main(JMap.java:128)
Caused by: java.lang.RuntimeException: Type "nmethoBucket*", referenced in VMStructs::localHotSpotVMStructs i
n the remote VM, was not present in the remote VMStructs::localHotSpotVMTypes table (should have been caught i
n the debug build of that VM). Can not continue.
    at sun.jvm.hotspot.HotSpotTypeDataBase.lookupOrFail(HotSpotTypeDataBase.java:362)
    at sun.jvm.hotspot.HotSpotTypeDataBase.readVMStructs(HotSpotTypeDataBase.java:253)
    at sun.jvm.hotspot.HotSpotTypeDataBase.<init>(HotSpotTypeDataBase.java:87)
    at sun.jvm.hotspot.bugspot.BugSpotAgent.setupVM(BugSpotAgent.java:568)
    at sun.jvm.hotspot.bugspot.BugSpotAgent.go(BugSpotAgent.java:494)
    at sun.jvm.hotspot.bugspot.BugSpotAgent.attach(BugSpotAgent.java:332)
    at sun.jvm.hotspot.tools.Tool.start(Tool.java:163)
    at sun.jvm.hotspot.tools.HeapSummary.main(HeapSummary.java:39)
    ... 6 more
```

出现这个问题的原因是使用工具的版本和运行 JVM 进程的 JDK 版本不一致，用同一个版本 JDK 自带的工具就不会有问题了。

6.3 jstat 遇到的一个问题

<http://stackoverflow.com/questions/27918259/why-no-fullgc-but-old-gen-from-99-to-14-display-in-results-of-the-jstat-gcuti>

7. 案例分析

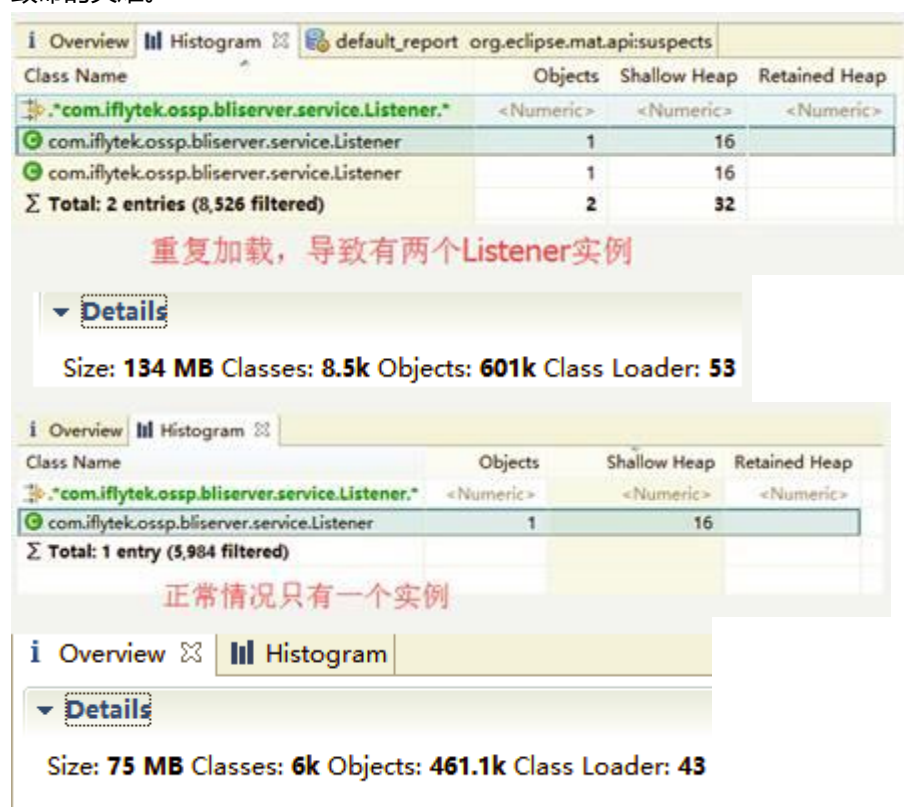
7.1 内存泄露

7.2 CPU 问题

7.3 线程死锁

7.4 ClassLoader 类加载泄露

Tomcat 的 server.xml 配置不对的时候会导致应用被多次加载，如果用到了 spring 及其他涉及到管理应用生命周期的组件时（如 ServletContextListener），要特别注意：周期性任务及单例模式的问题，这是个致命的灾难。



更严重的是因为两次加载的 ClassLoader 对象不同，即使是单例的对象在内存中也会存在两份实例，会导致加载的类多很多。

duplicate_classes			
Class Name	Count	Defined Classes	No. of Instances
com.iflytek.ossbliserver.service.Listener.	<Numeric>	<Numeric>	<Numeric>
com.iflytek.ossbliserver.service.Listener	2		
org.apache.catalina.loader.WebappClassLoader @ 0x7f56dc528		2,503	22,132
org.apache.catalina.loader.WebappClassLoader @ 0x7f55b88b8		2,502	22,132
Σ Total: 2 entries			

如果应用在 contextInitialized 中做了一些操作又没有打印日志的话，后果是出现问题而且更隐蔽难于排查。

具体问题和解决方法参见：<http://blog.csdn.net/jason5186/article/details/7317614>

官方文档对这个特别做了说明：

path	The <i>context path</i> of this web application, which is matched against the beginning of each request URI to select the appropriate web application for processing. All of the context paths within a particular <i>Host</i> must be unique. If you specify a context path of an empty string (""), you are defining the <i>default</i> web application for this Host, which will process all requests not assigned to other Contexts.
	This attribute must only be used when statically defining a Context in server.xml . In all other circumstances, the path will be inferred from the filenames used for either the .xml context file or the docBase.
	Even when statically defining a Context in server.xml , this attribute must not be set unless either the docBase is not located under the Host's appBase or both deployOnStartup and autoDeploy are false. If this rule is not followed, double deployment is likely to result.

<http://tomcat.apache.org/tomcat-7.0-doc/config/context.html>