# 排查Java内存问题

**原文出处：**

<http://www.infoq.com/cn/articles/Troubleshooting-Java-Memory-Issues?utm_campaign=rightbar_v2&utm_source=infoq&utm_medium=articles_link&utm_content=link_text>

## 核心要点

* 排查Java的内存问题可能会非常困难，但是正确的方法和适当的工具能够极大地简化这一过程；
* Java HotSpot JVM会报告各种OutOfMemoryError信息，清晰地理解这些错误信息非常重要，在我们的工具箱中有各种诊断和排查问题的工具，它们能够帮助我们诊断并找到这些问题的根本原因；
* 在本文中，我们会介绍各种诊断工具，在解决内存问题的时候，它们是非常有用的，包括：
  + HeapDumpOnOutOfMemoryError和PrintClassHistogram JVM选项
  + Eclipse MAT
  + Java VisualVM
  + JConsole
  + jhat
  + YourKit
  + jmap
  + jcmd
  + Java Flight Recorder和Java Mission Control
  + GC Logs
  + NMT
  + 原生内存泄露探测工具，比如dbx、libumem、valgrind和purify等。

对于一个Java进程来说，会有多个内存池或空间——Java堆、Metaspace、PermGen（在Java 8之前的版本中）以及原生堆。

每个内存池都可能会遇到自己的内存问题，比如不正常的内存增加、应用变慢或者内存泄露，每种形式的问题最终都会以各自空间OutOfMemoryError的形式体现出来。

在本文中，我们会尝试理解这些OutOfMemoryError错误信息的含义以及分析和解决这些问题要收集哪些诊断数据，另外还会研究一些用来收集和分析数据的工具，它们有助于解决这些内存问题。本文的关注点在于如何处理这些内存问题以及如何在生产环境中避免出现这些问题。

Java HotSpot VM所报告的OutOfMemoryError信息能够清楚地表明哪块内存区域正在耗尽。接下来，让我们仔细看一下各种OutOfMemoryError信息，理解其含义并探索导致它们出现的原因，最后介绍如何排查和解决这些问题。

## OutOfMemoryError: Java Heap Space

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at java.util.Arrays.copyOfRange(Unknown Source)

at java.lang.String.<init>(Unknown Source)

at java.io.BufferedReader.readLine(Unknown Source)

at java.io.BufferedReader.readLine(Unknown Source)

at com.abc.ABCParser.dump(ABCParser.java:23)

at com.abc.ABCParser.mainABCParser.java:59)

这个信息表示JVM在Java堆上已经没有空闲的空间，JVM无法继续执行程序了。这种错误最常见的原因就是指定的最大Java堆空间已经不足以容纳所有的存活对象了。要检查Java堆空间是否足以容纳JVM中所有存活的对象，一种简单的方式就是检查GC日志。

688995.775: [Full GC [PSYoungGen: 46400K->0K(471552K)] [ParOldGen: 1002121K->304673K(1036288K)] 1048

521K->304673K(1507840K) [PSPermGen: 253230K->253230K(1048576K)], 0.3402350 secs] [Times: user=1.48

sys=0.00, real=0.34 secs]

从上面的日志条目我们可以看到在Full GC之后，堆的占用从1GB（1048521K）降低到了305MB（304673K），这意味着分配给堆的1.5GB（1507840K）足以容纳存活的数据集。

现在，我们看一下如下的GC活动：

20.343: [Full GC (Ergonomics) [PSYoungGen: 12799K->12799K(14848K)] [ParOldGen: 33905K->33905K(34304K)] 46705K- >46705K(49152K), [Metaspace: 2921K->2921K(1056768K)], 0.4595734 secs] [Times: user=1.17 sys=0.00, real=0.46 secs]

...... <snip> several Full GCs </snip> ......

22.640: [Full GC (Ergonomics) [PSYoungGen: 12799K->12799K(14848K)] [ParOldGen: 33911K->33911K(34304K)] 46711K- >46711K(49152K), [Metaspace: 2921K->2921K(1056768K)], 0.4648764 secs] [Times: user=1.11 sys=0.00, real=0.46 secs]

23.108: [Full GC (Ergonomics) [PSYoungGen: 12799K->12799K(14848K)] [ParOldGen: 33913K->33913K(34304K)] 46713K- >46713K(49152K), [Metaspace: 2921K->2921 K(1056768K)], 0.4380009 secs] [Times: user=1.05 sys=0.00, real=0.44 secs]

23.550: [Full GC (Ergonomics) [PSYoungGen: 12799K->12799K(14848K)] [ParOldGen: 33914K->33914K(34304K)] 46714K- >46714K(49152K), [Metaspace: 2921K->2921 K(1056768K)], 0.4767477 secs] [Times: user=1.15 sys=0.00, real=0.48 secs]

24.029: [Full GC (Ergonomics) [PSYoungGen: 12799K->12799K(14848K)] [ParOldGen: 33915K->33915K(34304K)] 46715K- >46715K(49152K), [Metaspace: 2921K->2921 K(1056768K)], 0.4191135 secs] [Times: user=1.12 sys=0.00, real=0.42 secs] Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded at oom.main(oom.java:15)

从转储的“Full GC”频率信息我们可以看到，这里存在多次连续的Full GC，它会试图回收Java堆中的空间，但是堆已经完全满了，GC并没有释放任何空间。这种频率的Full GC会对应用的性能带来负面的影响，会让应用变慢。这个样例表明应用所需的堆超出了指定的Java堆的大小。增加堆的大小会有助于避免full GC并且能够规避OutOfMemoryError。Java堆的大小可以通过-Xmx JVM选项来指定：

java –Xmx1024m –Xms1024m Test

OutOfMemoryError可能也是应用存在内存泄露的一个标志。内存泄露通常难以察觉，尤其是缓慢的内存泄露。如果应用无意间持有了堆中对象的引用，会造成内存的泄露，这会导致对象无法被垃圾回收。随着时间的推移，在堆中这些无意被持有的对象可能会随之增加，最终填满整个Java堆空间，导致频繁的垃圾收集，最终程序会因为OutOfMemoryError错误而终止。

请注意，最好始终启用GC日志，即便在生产环境也如此，在出现内存问题时，这样有助于探测和排查。如下的选项能够用来开启GC日志：

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-XX:+PrintGCDateStamps

-Xloggc:<gc log file>

探测内存泄露的第一步就是监控应用的存活集合（live-set）。存活集合指的是full GC之后的Java堆。如果应用达到稳定状态和稳定负载之后，存活集合依然在不断增长，这表明可能会存在内存泄露。堆的使用情况可以通过Java VisualVM、Java Mission Control和JConsole这样的工具来进行监控，也可以从GC日志中进行抽取。

## Java堆：诊断数据的收集

在这一部分中，我们将会讨论要收集哪些诊断数据以解决Java堆上的OutOfMemoryErrors问题，有些工具能够帮助我们收集所需的诊断数据。

### 堆转储

在解决内存泄露问题时，堆转储（dump）是最为重要的数据。堆转储可以通过jcmd、jmap、JConsole和HeapDumpOnOutOfMemoryError JVM配置项来收集，如下所示：

* jcmd <process id/main class> GC.heap\_dump filename=heapdump.dmp
* jmap -dump:format=b,file=snapshot.jmap pid
* JConsole工具，使用Mbean HotSpotDiagnostic
* -XX:+HeapDumpOnOutOfMemoryError

java -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xmx20m -XX:+HeapDumpOnOutOfMemoryError oom

0.402: [GC (Allocation Failure) [PSYoungGen: 5564K->489K(6144K)] 5564K->3944K(19968K), 0.0196154 secs] [Times: user=0.05 sys=0.00, real=0.02 secs]

0.435: [GC (Allocation Failure) [PSYoungGen: 6000K->496K(6144K)] 9456K->8729K(19968K), 0.0257773 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

0.469: [GC (Allocation Failure) [PSYoungGen: 5760K->512K(6144K)] 13994K->13965K(19968K), 0.0282133 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

0.499: [Full GC (Ergonomics) [PSYoungGen: 512K->0K(6144K)] [ParOldGen: 13453K->12173K(13824K)] 13965K-

>12173K(19968K), [Metaspace: 2922K->2922K(1056768K)], 0.6941054 secs] [Times: user=1.45 sys=0.00, real=0.69 secs] 1.205: [Full GC (Ergonomics) [PSYoungGen: 5632K->2559K(6144K)] [ParOldGen: 12173K->13369K(13824K)] 17805K-

>15929K(19968K), [Metaspace: 2922K->2922K(1056768K)], 0.3933345 secs] [Times: user=0.69 sys=0.00, real=0.39 secs]

1.606: [Full GC (Ergonomics) [PSYoungGen: 4773K->4743K(6144K)] [ParOldGen: 13369K->13369K(13824K)] 18143K-

>18113K(19968K), [Metaspace: 2922K->2922K(1056768K)], 0.3009828 secs] [Times: user=0.72 sys=0.00, real=0.30 secs]

1.911: [Full GC (Allocation Failure) [PSYoungGen: 4743K->4743K(6144K)] [ParOldGen: 13369K->13357K(13824K)] 18113K-

>18101K(19968K), [Metaspace: 2922K->2922K(1056768K)], 0.6486744 secs] [Times: user=1.43 sys=0.00, real=0.65 secs]

java.lang.OutOfMemoryError: Java heap space

Dumping heap to java\_pid26504.hprof ...

Heap dump file created [30451751 bytes in 0.510 secs] Exception in thread "main" java.lang.OutOfMemoryError: Java heap space

at java.util.Arrays.copyOf(Arrays.java:3210)

at java.util.Arrays.copyOf(Arrays.java:3181)

at java.util.ArrayList.grow(ArrayList.java:261)

at java.util.ArrayList.ensureExplicitCapacity(ArrayList.java:235)

at java.util.ArrayList.ensureCapacityInternal(ArrayList.java:227)

at java.util.ArrayList.add(ArrayList.java:458)

at oom.main(oom.java:14)

请注意，并行垃圾收集器可能会连续地调用Full GC以便于释放堆上的空间，即便这种尝试的收益很小、堆空间几乎已被充满时，它可能也会这样做。为了避免这种情况的发生，我们可以调节-XX:GCTimeLimit 和-XX:GCHeapFreeLimit的值。

GCTimeLimit能够设置一个上限，指定GC时间所占总时间的百分比。它的默认值是98%。减少这个值会降低垃圾收集所允许花费的时间。GCHeapFreeLimit设置了一个下限，它指定了垃圾收集后应该有多大的空闲区域，这是一个相对于堆的总小大的百分比。它的默认值是2%。增加这个值意味着在GC后要回收更大的堆空间。如果五次连续的Full GC都不能保持GC的成本低于GCTimeLimit并且无法释放 GCHeapFreeLimit所要求的空间的话，将会抛出OutOfMemoryError。

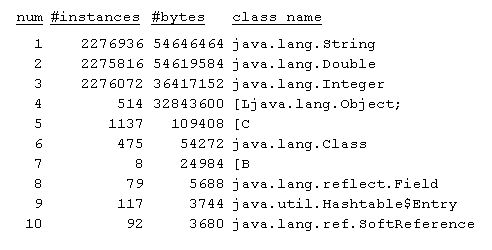
例如，将GCHeapFreeLimit设置为8%的话，如果连续五次垃圾收集无法回收至少8%的堆空间并且超出了GCTimeLimit设置的值，这样能够帮助垃圾收集器避免连续调用Full GC的情况出现。

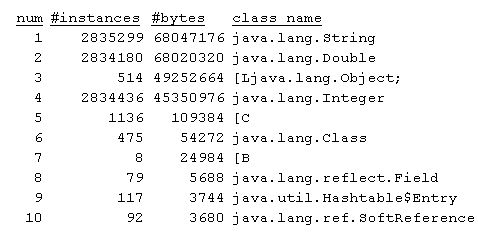
### 堆直方图

有时，我们需要快速查看堆中不断增长的内容是什么，绕过使用内存分析工具收集和分析堆转储的漫长处理路径。堆直方图能够为我们快速展现堆中的对象，并对比这些直方图，帮助我们找到Java堆中增长最快的是哪些对象。

* -XX:+PrintClassHistogram以及Control+Break
* jcmd <process id/main class> GC.class\_histogram filename=Myheaphistogram
* jmap -histo pid
* jmap -histo <java> core\_file

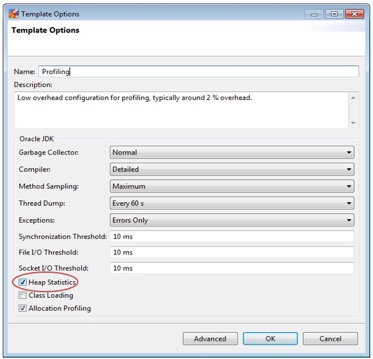
下面的示例输出显示String、Double、Integer和Object[]的实例占据了Java堆中大多数的空间，并且随着时间的流逝数量在不断增长，这意味着它们可能会导致内存泄露：

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/4331-1520876873293.png)

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/11.1-1520876872698.png)

## Java飞行记录

将飞行记录（Flight Recordings）启用堆分析功能能够帮助我们解决内存泄露的问题，它会展现堆中的对象以及随着时间推移，哪些对象增长最快。要启用堆分析功能，你可以使用Java Mission Control并选中“Heap Statistics”，这个选项可以通过“Window->Flight Recording Template Manager”找到，如下所示：

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/842-1520876872871.jpg)

或者手动编辑.jfc文件，将heap-statistics-enabled设置为true。

<event path="vm/gc/detailed/object\_count">

<setting name="enabled" control="heap-statistics-enabled">true</setting>

<setting name="period">everyChunk</setting>

</event>

飞行记录可以通过如下的方式来创建：

* JVM Flight Recorder选项，比如：

-XX:+UnlockCommercialFeatures -XX:+FlightRecorder  
-XX:StartFlightRecording=delay=20s,duration=60s,name=MyRecording,

filename=C:\TEMP\myrecording.jfr,settings=profile

* Java诊断命令：jcmd

jcmd 7060 JFR.start name=MyRecording settings=profile delay=20s duration=2m filename=c:\TEMP\myrecording.jfr

* Java任务控制（Java Mission Control）

飞行记录器只能帮我们确定哪种类型的对象出现了泄露，但是想要找到是什么原因导致了这些对象泄露，我们还需要堆转储。

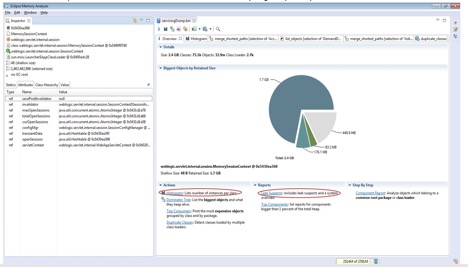
## Java堆：分析诊断数据

### 堆转储分析

堆转储可以使用如下的工具进行分析：

* [Eclipse MAT](http://www.eclipse.org/mat/)（内存分析工具，Memory Analyzer Tool）是一个社区开发的分析堆转储的工具。它提供了一些很棒的特性，包括：
  + 可疑的泄漏点：它能探测堆转储中可疑的泄露点，报告持续占有大量内存的对象；
  + 直方图：列出每个类的对象数量、浅大小（shallow）以及这些对象所持有的堆。直方图中的对象可以很容易地使用正则表达式进行排序和过滤。这样有助于放大并集中我们怀疑存在泄露的对象。它还能够对比两个堆转储的直方图，展示每个类在实例数量方面的差异。这样能够帮助我们查找Java堆中增长最快的对象，并进一步探查确定在堆中持有这些对象的根；
  + 不可达的对象：MAT有一个非常棒的功能，那就是它允许在它的工作集对象中包含或排除不可达/死对象。如果你不想查看不可达的对象，也就是那些会在下一次GC周期中收集掉的对象，只关心可达的对象，那么这个特性是非常便利的；
  + 重复的类：展现由多个类加载器所加载的重复的类；
  + 到GC根的路径：能够展示到GC根（JVM本身保持存活的对象）的引用链，这些GC根负责持有堆中的对象；
  + OQL：我们可以使用对象查询语言（Object Query Language）来探查堆转储中的对象。它丰富了OQL的基础设施，能够编写复杂的查询，帮助我们深入了解转储的内部。
* [Java VisualVM](https://visualvm.github.io/)：监控、分析和排查Java语言的一站式工具。它可以作为JDK工具的一部分来使用，也可以从GitHub上下载。它所提供的特性之一就是堆转储分析。它能够为正在监控的应用创建堆转储，也可以加载和解析它们。从堆转储中，它可以展现类的直方图、类的实例，也能查找特定实例的GC根；
* [jhat](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jhat.html)命令工具（在<jdk>/bin文件夹中）提供了堆转储分析的功能，它能够在任意的浏览器中展现堆转储中的对象。默认情况下，Web服务器会在7000端口启动。jhat支持范围广泛的预定义查询和对象查询语言，以便于探查堆转储中的对象；
* Java任务控制（Java Mission Control）的[JOverflow](http://hirt.se/blog/?p=611)插件：这是一个实验性的插件，能够让Java任务控制执行简单的堆转储分析并报告哪里可能存在内存浪费；
* [Yourkit](https://www.yourkit.com/java/profiler/features/)是一个商业的Java profiler，它有一个堆转储分析器，具备其他工具所提供的几乎所有特性。除此之外，YourKit还提供了：
  + 可达性的范围（reachability scope）：它不仅能够列出可达和不可达的对象，还能按照它们的可达性范围显示它们的分布，也就是，强可达、弱/软可达或不可达；
  + 内存探查：YourKit内置了一组全面的查询，而不是使用ad-hoc查询功能，YourKit的查询能够探查内存，查找反模式并为常见的内存问题分析产生原因和提供解决方案。

我使用Eclipse MAT较多，我发现在分析堆转储时，它是非常有用的。

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/733-1520876873033.jpg)

MAT有一些高级的特性，包括直方图以及与其他的直方图进行对比的功能。这样的话，就能清晰地看出内存中哪些内容在增长并且能够看到Java堆中占据空间最大的是什么内容。我非常喜欢的一个特性是“Merge Shortest Paths to GC Roots（合并到GC Root的最短路径）”，它能够帮助我们查找无意中所持有的对象的跟踪痕迹。比如，在下面的引用链中，ThreadLocalDateFormat对象被ThreadLocalMap$Entry对象的“value”字段所持有。只有当ThreadLocalMap$Entry从ThreadLocalMap中移除之后，ThreadLocalDateFormat才能被回收。

weblogic.work.ExecuteThread @ 0x6996963a8 [ACTIVE] ExecuteThread: '203' for queue: 'weblogic.kernel.Default (self-tuning)' Busy Monitor, Thread| 1 | 176 | 40 | 10,536

'- threadLocals java.lang.ThreadLocal$ThreadLocalMap @ 0x69c2b5fe0 | 1 | 24 | 40 | 7,560

'- table java.lang.ThreadLocal$ThreadLocalMap$Entry[256] @ 0x6a0de2e40 | 1 | 1,040 | 40 | 7,536

'- [116] java.lang.ThreadLocal$ThreadLocalMap$Entry @ 0x69c2ba050 | 1 | 32 | 40 | 1,088

'- value weblogic.utils.string.ThreadLocalDateFormat @ 0x69c23c418 | 1 | 40 | 40 | 1,056

通过这种方式，我们可以看到堆中增长最快的罪魁祸首，并且看到内存中哪里出现了泄露。

### Java任务控制

Java任务控制可以在JDK的<jdk>/bin文件夹中找到。启用Heap Statistics功能之后所收集到的飞行记录能够极大地帮助我们解决内存泄露问题。我们可以在Memory->Object Statistics中查看对象的分析信息。这个视图将会展现对象的直方图，包括每个对象类型所占据的堆的百分比。它能够展现堆中增长最快的对象，在大多数情况下，也就直接对应了内存泄露的对象。

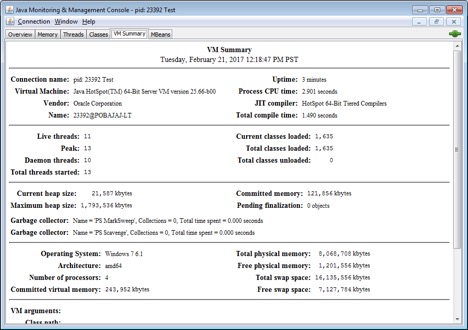
## 终结器所导致的OutOfMemoryError

滥用终结器（finalizer）可能也会造成OutOfMemoryError。带有终结器的对象（也就是含有finalize()方法）会延迟它们所占有空间的回收。在回收这些实例并释放其堆空间之前，终结器线程（finalizer thread）需要调用它们的finalize()方法。如果终结者线程的处理速度比不上要终结对象的增加速度（添加到终结者队列中以便于调用其finalize()方法）的话，那么即便终结器队列中的对象都有资格进行回收，JVM可能也会出现OutOfMemoryError。因此，非常重要的一点就是确保不要因为大量对象等待（pending）终结而造成内存耗尽。

我们可以使用如下的工具来监控等待终结的对象数量：

* JConsole

我们可以连接JConsole到一个运行中的进程，然后在VM Summary页面查看等待终结的对象数量，如下图所示。

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/554-1520876873162.jpg)

* jmap – finalizerinfo

D:\tests\GC\_WeakReferences>jmap -finalizerinfo 29456

Attaching to process ID 29456, please wait...

Debugger attached successfully. Server compiler detected.

JVM version is 25.122-b08

Number of objects pending for finalization: 10

* 堆转储

几乎所有的堆转储分析工具都能详细给出等待终结的对象信息。

Java VisualVM的输出

Date taken: Fri Jan 06 14:48:54 PST 2017

File: D:\tests\java\_pid19908.hprof

File size: 11.3 MB

Total bytes: 10,359,516

Total classes: 466

Total instances: 105,182

Classloaders: 2

GC roots: 419

Number of objects pending for finalization: 2

# OutOfMemoryError: PermGen Space

java.lang.OutOfMemoryError: PermGen space

我们知道，从Java 8之后，PermGen已经移除掉了。如果读者运行的是Java 8以上的版本，那么这一小节可以直接略过。

在Java 7及以前，PermGen（“永久代，permanent generation”的缩写）用来存储类定义以及它们的元数据。在这个内存区域中，PermGen意料之外的增长以及OutOfMemoryError意味着类没有按照预期卸载，或者所指定的PermGen空间太小，无法容纳所有要加载的类和它们的元数据。

要确保PermGen的大小能够满足应用的需求，我们需要监控它的使用情况并使用如下的JVM选项进行相应的配置：

        –XX:PermSize=n –XX:MaxPermSize=m

# OutOfMemoryError: Metaspace

MetaSpace的OutOfMemoryError输出样例如下所示：

java.lang.OutOfMemoryError: Metaspace

从Java 8开始，类元数据存储到了Metaspace中。Metaspace并不是Java堆的一部分，它是分配在原生内存上的。所以，它仅仅受到机器可用原生内存数量的限制。但是，Metaspace也可以通过 MaxMetaspaceSize参数来设置它的大小。

如果Metaspace的使用接近MaxMetaspaceSize的最大限制，那么我们就会遇到OutOfMemoryError。与其他的区域类似，这种错误可能是因为没有足够的Metaspace，或者存在类加载器/类泄露。如果出现了后者的情况，我们需要借助诊断工具，解决Metaspace中的内存泄露。

# OutOfMemoryError: Compressed class space

java.lang.OutOfMemoryError: Compressed class space

如果启用了UseCompressedClassesPointers的话（打开UseCompressedOops的话之后，会默认启用），那么原生内存上会有两个独立的区域用来存储类和它们的元数据。启用UseCompressedClassesPointers之后，64位的类指针会使用32位的值来表示，压缩的类指针会存储在压缩类空间（compressed class space）中。默认情况下，压缩类空间的大小是1GB并且可以通过CompressedClassSpaceSize进行配置。

MaxMetaspaceSize能够为这两个区域设置一个总的提交（committed）空间大小，即压缩类空间和类元数据的提交空间。

启用UseCompressedClassesPointers之后，在GC日志中会进行采样输出。在Metaspace所报告的提交和保留（reserved）空间中包含了压缩类空间的提交和预留空间。

Metaspace used 2921K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 288K, capacity 386K, committed 512K, reserved 1048576K

## PermGen和Metaspace：数据收集和分析工具

PermGen和Metaspace所占据的空间可以使用Java任务控制、Java VisualVM和JConsole进行监控。GC能够帮助我们理解Full GC前后PermGen/Metaspace的使用情况，也能看到是否存在因为PermGen/Metaspace充满而导致的Full GC。

另外非常重要的一点在于确保类按照预期进行了卸载。类的加载和卸载可以通过启用下面的参数来进行跟踪：

-XX:+TraceClassUnloading –XX:+TraceClassLoading

在将应用从开发环境提升到生产环境时，需要注意应用程序有可能会被无意地改变一些JVM可选参数，从而带来不良的后果。其中有个选项就是-Xnoclassgc，它会让JVM在垃圾收集的时候不去卸载类。现在，如果应用需要加载大量的类，或者在运行期有些类变得不可达了，需要加载另外一组新类，应用恰好是在–Xnoclassgc模式下运行的，那么它有可能达到PermGen/Metaspace的最大容量，就会出现OutOfMemoryError。因此，如果你不确定这个选项为何要设置的话，那么最好将其移除，让垃圾收集器在这些类能够回收的时候将其卸载掉。

加载的类和它们所占用的内存可以通过Native Memory Tracker（NMT）来进行跟踪。我们将会在下面的“OutOfMemoryError: Native Memory”小节详细讨论这个工具。

需要注意，在使用并发标记清除收集器（Concurrent MarkSweep Collector，CMS）时，需要启用如下的选项，从而确保CMS并发收集周期能够将类卸载掉：–XX:+CMSClassUnloadingEnabled

在Java 7中，这个标记默认是关闭的，而在Java 8中它默认就是启用的。

### jmap

“jmap –permstat”会展现类加载器的统计数据，比如类加载器、类加载器所加载的类的数量以及这些类加载已死亡还是尚在存活。它还会告诉我们PermGen中interned字符串的总数，以及所加载的类及其元数据所占用的字节数。如果我们要确定是什么内容占满了PermGen，那么这些信息是非常有用的。如下是一个示例的输出，展现了所有的统计信息。在列表的最后一行我们能够看到有一个总数的概述。

$ jmap -permstat 29620

Attaching to process ID 29620, please wait...

Debugger attached successfully. Client compiler detected.

JVM version is 24.85-b06

12674 intern Strings occupying 1082616 bytes. finding class loader instances ..

done. computing per loader stat ..done. please wait.. computing liveness.........................................done.

class\_loader classes bytes parent\_loader alive? type

<bootstrap> 1846 5321080 null live <internal>

0xd0bf3828 0 0 null live sun/misc/Launcher$ExtClassLoader@0xd8c98c78

0xd0d2f370 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0c99280 1 1440 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0b71d90 0 0 0xd0b5b9c0 live java/util/ResourceBundle$RBClassLoader@0xd8d042e8

0xd0d2f4c0 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0b5bf98 1 920 0xd0b5bf38 dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0c99248 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f488 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0b5bf38 6 11832 0xd0b5b9c0 dead sun/reflect/misc/MethodUtil@0xd8e8e560

0xd0d2f338 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f418 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f3a8 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0b5b9c0 317 1397448 0xd0bf3828 live sun/misc/Launcher$AppClassLoader@0xd8cb83d8

0xd0d2f300 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f3e0 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0ec3968 1 1440 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0e0a248 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0c99210 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f450 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0d2f4f8 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

0xd0e0a280 1 904 null dead sun/reflect/DelegatingClassLoader@0xd8c22f50

total = 22 2186 6746816 N/A alive=4, dead=18 N/A

从Java 8开始，jmap –clstats <pid>命令能够打印类加载器及其存活性的类似信息，不过它所展现的是Metaspace中已加载的类的数量和大小，而不再是PermGen。

jmap -clstats 26240

Attaching to process ID 26240, please wait...

Debugger attached successfully. Server compiler detected. JVM version is 25.66-b00 finding class loader instances ..done. computing per loader stat ..done. please wait.. computing liveness.liveness analysis may be inaccurate ...

class\_loader classes bytes parent\_loader alive? type

<bootstrap> 513 950353 null live <internal>

0x0000000084e066d0 8 24416 0x0000000084e06740 live sun/misc/Launcher$AppClassLoader@0x0000000016bef6a0

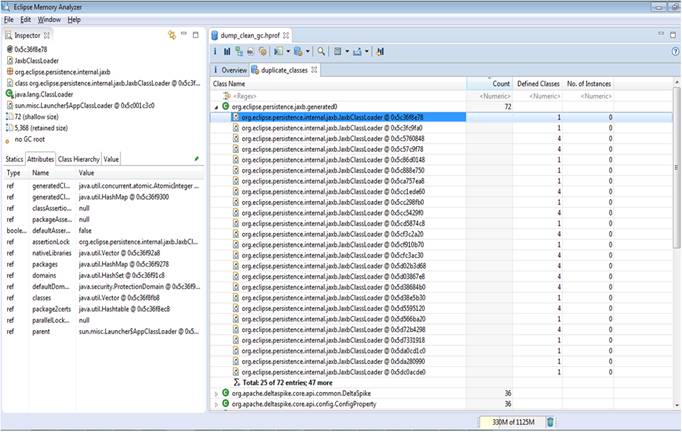
0x0000000084e06740 0 0 null live sun/misc/Launcher$ExtClassLoader@0x0000000016befa48

0x0000000084ea18f0 0 0 0x0000000084e066d0 dead java/util/ResourceBundle$RBClassLoader@0x0000000016c33930

total = 4 521 974769 N/A alive=3, dead=1 N/A

### 堆转储

正如我们在前面的章节所提到的，Eclipse MAT、jhat、Java VisualVM、JOverflow JMC插件和Yourkit这些工具都能分析堆转储文件，从而分析排查OutOfMemoryError。在解决PermGen和Metaspace的内存问题时，堆转储同样是有用的。Eclipse MAT提供了一个非常好的特性叫做“Duplicate Classes”，它能够列出被不同的类加载实例多次加载的类。由不同的类加载器加载数量有限的重复类可能是应用设计的一部分，但是，如果它们的数量随着时间推移不断增长的话，那么这就是一个危险的信号，需要进行调查。应用服务器托管多个应用时，它们运行在同一个JVM中，如果多次卸载和重新部署应用的话，经常会遇到这种状况。如果被卸载的应用没有释放所有它创建的类加载器的引用，JVM就不能卸载这些类加载器所加载的类，而新部署的应用会使用新的类加载器实例重新加载这些类。

[](https://s3.amazonaws.com/infoq.content.live.0/articles/Troubleshooting-Java-Memory-Issues/zh/resources/475-1520876873423.jpg)

这个快照显示JaxbClassLoader加载了类的重复副本，这是因为应用在为每个XML进行Java类绑定的时候，不恰当地创建了新的JAXBContext实例。

### jcmd

jcmd <pid/classname> GC.class\_stats能够提供被加载类的更详细信息，借助它，我们能够看到Metaspace每个类所占据的空间，如下面的示例输出所示。

jcmd 2752 GC.class\_stats 2752:

Index Super InstBytes KlassBytes annotations CpAll MethodCount Bytecodes MethodAll ROAll RWAll Total ClassName

1 357 821632 536 0 352 2 13 616 184 1448 1632 java.lang.ref.WeakReference

2 -1 295272 480 0 0 0 0 0 24 584 608 [Ljava.lang.Object;

3 -1 214552 480 0 0 0 0 0 24 584 608 [C

4 -1 120400 480 0 0 0 0 0 24 584 608 [B

5 35 78912 624 0 8712 94 4623 26032 12136 24312 36448 java.lang.String

6 35 67112 648 0 19384 130 4973 25536 16552 30792 47344 java.lang.Class

7 9 24680 560 0 384 1 10 496 232 1432 1664 java.util.LinkedHashMap$Entry

8 -1 13216 480 0 0 0 0 0 48 584 632 [Ljava.lang.String;

9 35 12032 560 0 1296 7 149 1520 880 2808 3688 java.util.HashMap$Node

10 -1 8416 480 0 0 0 0 0 32 584 616 [Ljava.util.HashMap$Node;

11 -1 6512 480 0 0 0 0 0 24 584 608 [I

12 358 5688 720 0 5816 44 1696 8808 5920 10136 16056 java.lang.reflect.Field

13 319 4096 568 0 4464 55 3260 11496 7696 9664 17360 java.lang.Integer

14 357 3840 536 0 584 3 56 496 344 1448 1792 java.lang.ref.SoftReference

15 35 3840 584 0 1424 8 240 1432 1048 2712 3760 java.util.Hashtable$Entry

16 35 2632 736 368 8512 74 2015 13512 8784 15592 24376 java.lang.Thread

17 35 2496 504 0 9016 42 2766 9392 6984 12736 19720 java.net.URL

18 35 2368 568 0 1344 8 223 1408 1024 2616 3640 java.util.concurrent.ConcurrentHashMap$Node

…<snip>…

577 35 0 544 0 1736 3 136 616 640 2504 3144 sun.util.locale.provider.SPILocaleProviderAdapter$1

578 35 0 496 0 2736 8 482 1688 1328 3848 5176 sun.util.locale.provider.TimeZoneNameUtility

579 35 0 528 0 776 3 35 472 424 1608 2032 sun.util.resources.LocaleData$1

580 442 0 608 0 1704 10 290 1808 1176 3176 4352 sun.util.resources.OpenListResourceBundle

581 580 0 608 0 760 5 70 792 464 1848 2312 sun.util.resources.TimeZoneNamesBundle

1724488 357208 1536 1117792 7754 311937 1527952 1014880 2181776 3196656 Total

53.9% 11.2% 0.0% 35.0% - 9.8% 47.8% 31.7% 68.3% 100.0%

Index Super InstBytes KlassBytes annotations CpAll MethodCount Bytecodes MethodAll ROAll RWAll Total ClassName

从这个输出中，我们可以看到所加载类的名称（**ClassName**）、每个类所占据的字节（**KlassBytes**）、每个类的实例所占据的字节（InstBytes）、每个类中方法的数量（**MethodCount**）、字节码所占据的空间（**ByteCodes)**）等等。

需要注意的是，在Java 8中，这个诊断命令需要Java进程使用‑XX:+UnlockDiagnosticVMOptions选项启动。

jcmd 33984 GC.class\_stats 33984:

GC.class\_stats command requires -XX:+UnlockDiagnosticVMOptions

在Java 9中，该诊断命令不需要-XX:+UnlockDiagnosticVMOption。

# OutOfMemoryError: Native Memory

原生内存出现OutOfMemoryError的一些示例如下所示：  
**因为没有足够交换空间（swap space）所引起的OutOfMemoryError：**

# A fatal error has been detected by the Java Runtime Environment:

#

# java.lang.OutOfMemoryError: requested 32756 bytes for ChunkPool::allocate. Out of swap space?

#

# Internal Error (allocation.cpp:166), pid=2290, tid=27 # Error: ChunkPool::allocate

**因为没有足够进程内存所导致的OutOfMemoryError：**

# A fatal error has been detected by the Java Runtime Environment:

#

# java.lang.OutOfMemoryError : unable to create new native Thread

这些错误清楚地表明JVM不能分配原生内存，这可能是因为进程本身消耗掉了所有的原生内存，也可能是系统中的其他进程在消耗原生内存。在使用“pmap”（或其他原生内存映射工具）监控原生堆的使用之后，我们可以恰当地配置Java堆、线程数以及栈的大小，确保有足够的空间留给原生堆，如果我们发现原生堆的使用在持续增长，最终会出现OutOfMemoryError，那么这可能提示我们遇到了原生内存的泄露。

## 64位JVM上的原生堆OutOfMemoryError

在32位JVM中，进程大小的上限是4GB，所以在32位Java进程中更容易出现原生内存耗尽的情况。但是，在64位JVM中，对内存的使用是没有限制的，从技术上讲，我们可能认为永远也不会遇到原生堆耗尽的情况，但事实并非如此，原生堆遇到OutOfMemoryErrors的情况并不少见。这是因为64位JVM默认会启用一个名为CompressedOops的特性，该特性的实现会决定要将Java堆放到地址空间的什么位置。Java堆的位置可能会对原生内存的最大容量形成限制。在下面的内存地图中，Java堆在8GB的地址边界上进行了分配，剩下了大约4GB留给原生堆。如果应用需要在原生内存分配大量空间，超出了4GB的话，即便系统还有大量的内存可用，它依然会抛出原生堆OutOfMemoryError。

0000000100000000 8K r-x-- /sw/.es-base/sparc/pkg/jdk-1.7.0\_60/bin/sparcv9/java

0000000100100000 8K rwx-- /sw/.es-base/sparc/pkg/jdk-1.7.0\_60/bin/sparcv9/java

0000000100102000 56K rwx-- [ heap ]

0000000100110000 2624K rwx-- [ heap ] <--- native Heap

00000001FB000000 24576K rw--- [ anon ] <--- Java Heap starts here

0000000200000000 1396736K rw--- [ anon ]

0000000600000000 700416K rw--- [ anon ]

这个问题可以通过-XX:HeapBaseMinAddress=n选项来解决，它能指定Java堆的起始地址。将它的设置成一个更高的地址将会为原生堆留出更多的空间。

关于如何诊断、排查和解决该问题，请参阅[该文](https://blogs.oracle.com/poonam/entry/running_on_a_64bit_platform)了解更详细的信息。

## 原生堆：诊断工具

让我们看一下内存泄露的探查工具，它们能够帮助我们找到原生内存泄露的原因。

### 原生内存跟踪

JVM有一个强大的特性叫做原生内存跟踪（Native Memory Tracking，NMT），它在JVM内部用来跟踪原生内存。需要注意的是，它无法跟踪JVM之外或原生库分配的内存。通过下面两个简单的步骤，我们就可以监控JVM的原生内存使用情况：

* 以启用NMT的方式启动进程。输出级别可以设置为“summary”或“detail”级别：
  + -XX:NativeMemoryTracking=summary
  + -XX:NativeMemoryTracking=detail
* 使用jcmd来获取原生内存使用的细节：
  + jcmd <pid> VM.native\_memory

NMT输出的样例：

d:\tests>jcmd 90172 VM.native\_memory 90172:

Native Memory Tracking:

Total: reserved=3431296KB, committed=2132244KB

- Java Heap (reserved=2017280KB, committed=2017280KB)

(mmap: reserved=2017280KB, committed=2017280KB)

- Class (reserved=1062088KB, committed=10184KB)

(classes #411)

(malloc=5320KB #190)

(mmap: reserved=1056768KB, committed=4864KB)

- Thread (reserved=15423KB, committed=15423KB)

(thread #16)

(stack: reserved=15360KB, committed=15360KB)

(malloc=45KB #81)

(arena=18KB #30)

- Code (reserved=249658KB, committed=2594KB)

(malloc=58KB #348)

(mmap: reserved=249600KB, committed=2536KB)

- GC (reserved=79628KB, committed=79544KB)

(malloc=5772KB #118)

(mmap: reserved=73856KB, committed=73772KB)

- Compiler (reserved=138KB, committed=138KB)

(malloc=8KB #41)

(arena=131KB #3)

- Internal (reserved=5380KB, committed=5380KB)

(malloc=5316KB #1357)

(mmap: reserved=64KB, committed=64KB)

- Symbol (reserved=1367KB, committed=1367KB)

(malloc=911KB #112)

(arena=456KB #1)

- Native Memory Tracking (reserved=118KB, committed=118KB)

(malloc=66KB #1040)

(tracking overhead=52KB)

- Arena Chunk (reserved=217KB, committed=217KB)

(malloc=217KB)

关于使用jcmd命令访问NMT数据的细节以及如何阅读它的输出，可以参见[该文](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr007.html)。

### 原生内存泄露探查工具

对于JVM外部的原生内存泄露，我们需要依赖原生内存泄露工具来进行探查和解决。原生工具能够帮助我们解决JVM之外的原生内存泄露问题，这样的工具包括[dbx](https://docs.oracle.com/cd/E18659_01/html/821-1380/toc.html)、[libumem](https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks005.html#CIHEHJAG)、[valgrind](http://valgrind.org/gallery/linux_mag.html)以及[purify](https://teamblue.unicomsi.com/products/purifyplus/)等。

## 总结

排查内存问题可能会非常困难和棘手，但是正确的方法和适当的工具能够极大地简化这一过程。我们看到，Java HotSpot JVM会报告各种OutOfMemoryError信息，清晰地理解这些错误信息非常重要，在工具集中有各种诊断和排查工具，帮助我们诊断和根治这些问题。

## 关于作者

**Poonam Parhar**, 目前在Oracle担任JVM支持工程师，她的主要职责是解决客户针对JRockit和HotSpot JVM的问题。她乐于调试和解决问题，主要关注于如何提升JVM的可服务性（serviceability）和可支持性（supportability）。她解决过HotSpot JVM中很多复杂的问题，热衷于改善调试工具和产品的可服务性，从而更容易地排查和定位JVM中垃圾收集器相关的问题。在帮助客户和Java社区的过程中，她通过[博客](https://blogs.oracle.com/poonam/)分享了她的工作经验和知识。