



运行这段代码会产生如下输出：

```
1  org.shelajev.throwaway.jdktools.OhMyMemory
2  java.lang.OutOfMemoryError: Java heap space
3  Dumping heap to java_pid5644.hprof ...
4  Heap dump file created [73169721 bytes in 0.645 secs]
5  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
6  at java.util.HashMap.resize(HashMap.java:703)
7  at java.util.HashMap.putVal(HashMap.java:662)
8  at java.util.HashMap.put(HashMap.java:611)
9  at org.shelajev.throwaway.jdktools.OhMyMemory.main(OhMyMemory.java:24)
10 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
11 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
12 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
13 at java.lang.reflect.Method.invoke(Method.java:483)
14 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
15 We have accumulated 393217 entries
```

不错，我们现在有一个可供分析的文件了。我们对这个文件执行jhat开始进行分析，jhat 会分析这个文件开启一个 http 服务器供我们查看结果。

```
1  $ jhat java_pid5644.hprof
2  Reading from java_pid5644.hprof...
3  Dump file created Thu Aug 14 14:48:19 EEST 2014
4  Snapshot read, resolving...
5  Resolving 1581103 objects...
6  Chasing references, expect 316 dots...
7  Eliminating duplicate references.....
8  Snapshot resolved.
9  Started HTTP server on port 7000
10 Server is ready.
```

可以通过访问 <http://localhost:7000> 来查看 dump 的数据。

# All Classes (excluding platform)

## Package com.intellij.rt.execution.application

[class com.intellij.rt.execution.application.AppMain](#) [0x7bc5ad190]  
[class com.intellij.rt.execution.application.AppMain\\$1](#) [0x7bc20f190]

## Package org.shelajev.throwaway.jdktools

[class org.shelajev.throwaway.jdktools.OhMyMemory](#) [0x7bc5b0ab0]  
[class org.shelajev.throwaway.jdktools.OhMyMemory\\$1](#) [0x7bc5ad4e0]

## 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](#)

在那个页面我们可以通过堆信息的柱状图了解究竟是什么耗尽了内存。

# Heap Histogram

[All Classes \(excluding platform\)](#)

<a href="#">Class</a>	<a href="#">Instance Count</a>	<a href="#">Total Size</a>
<a href="#">class [C</a>	395237	20345988
<a href="#">class java.util.HashMap\$Node</a>	393567	11019876
<a href="#">class java.lang.String</a>	395213	4742556
<a href="#">class [Ljava.util.HashMap\$Node;</a>	34	4202912
<a href="#">class java.lang.Integer</a>	393345	1573380
<a href="#">class [B</a>	462	122370
<a href="#">class [Ljava.lang.Object;</a>	600	52048
<a href="#">class java.lang.Class</a>	582	48888
<a href="#">class [Ljava.lang.String;</a>	96	9344

现在我们可以清晰地看到拥有 393567 结点的 HashMap 就是导致程序崩溃的元凶。虽然有更多可以检查内存分布使用情况和堆分析的工具，但是jhat 是内置的，是分析的一个好的开端。

## 3、jmap

jmap 是一个内存映射工具，它提供了另外一种不需要引发 OutOfMemoryErrors 就可以获取堆 dump 文件的方法。我们稍微修改一下上面的程序看一下效果。

```
1  public class OhMyMemory {
2
3  private static Map map = new HashMap<>();
4  }
```

```

5 public static void main(String[] args) {
6     Runtime.getRuntime().addShutdownHook(
7         new Thread() {
8             @Override
9             public void run() {
10                 try {
11                     System.out.println("Enter something, so I'll release the process");
12                     System.in.read();
13                     System.out.println("We have accumulated " + map.size() + " entries");
14                 }
15                 catch (IOException e) {
16                     e.printStackTrace();
17                 }
18             }
19         }
20     );
21
22     for(int i = 0; i < 10000; i++) {
23         map.put(Integer.toBinaryString(i), i);
24     }
25 }
26 }

```

注意，现在我们不要消耗大量的内存，只是比较早结束并在进程关闭钩子里等待不让 JVM 退出。这样就允许我们用 jmap 连接这个进程获取珍贵的内存 dump。

因此你可以用 jmap 的两个功能来实现，获取堆统计信息和触发一个堆 dump。因此，当执行：

jmap -heap 1354（这里 1354 是上面程序运行的进程号），就可以获取一个很好的内存使用统计信息：

```

1 $ jmap -heap 1354
2 Attaching to process ID 1354, please wait...
3 Debugger attached successfully.
4 Server compiler detected.
5 JVM version is 25.0-b70
6
7 using thread-local object allocation.
8 Parallel GC with 4 thread(s)
9
10 Heap Configuration:
11   MinHeapFreeRatio      = 40
12   MaxHeapFreeRatio      = 70
13   MaxHeapSize           = 67108864 (64.0MB)
14   NewSize                = 1572864 (1.5MB)
15   MaxNewSize             = 22020096 (21.0MB)
16   OldSize                = 45088768 (43.0MB)
17   NewRatio               = 2
18   SurvivorRatio          = 8
19   MetaspaceSize          = 21807104 (20.796875MB)
20   CompressedClassSpaceSize = 1073741824 (1024.0MB)
21   MaxMetaspaceSize       = 1759218604415 MB
22   G1HeapRegionSize       = 0 (0.0MB)
23
24 Heap Usage:
25 PS Young Generation
26 Eden Space:
27   capacity = 1048576 (1.0MB)
28   used     = 628184 (0.5990829467773438MB)
29   free     = 420392 (0.40091705322265625MB)
30   59.908294677734375% used
31 From Space:
32   capacity = 524288 (0.5MB)
33   used     = 491568 (0.4687957763671875MB)
34   free     = 32720 (0.0312042236328125MB)
35   93.7591552734375% used
36 To Space:
37   capacity = 524288 (0.5MB)
38   used     = 0 (0.0MB)
39   free     = 524288 (0.5MB)
40   0.0% used
41 PS Old Generation
42   capacity = 45088768 (43.0MB)
43   used     = 884736 (0.84375MB)
44   free     = 44204032 (42.15625MB)
45   1.9622093023255813% used
46
47 981 interned Strings occupying 64824 bytes.
48
49 $ jmap -dump:live,format=b,file=heap.bin 1354
50 Dumping heap to /Users/shelajev/workspace_idea/throwaway/heap.bin ...
51 Heap dump file created

```

jmap 还可以简单地触发当前堆 dump，之后可以随意进行分析。你可以像下面例子中的那样，传一个 -dump 参数给 jmap。

现在有了 dump 得到的文件 heap.bin，就可以用你喜欢的内存分析工具来分析。

## 4、jps

jps 是显示 Java 程序系统进程（PID）最常用的工具。它与平台无关，非常好用。想象一下我们启动了上面的程序，然后想用 jmap 连接它。这个时候我们需要程序的 PID，jps 正好的派上用场。

```

1 $ jps -mlv
2 5911 com.intellij.rt.execution.application.AppMain org.shelajev.throwaway.jdktools.OhMyMemory -Xmx64m -Didea.launcher.port=7535 -Didea.launcher.bin.path=/A
3 5544 -Dfile.encoding=UTF-8 -ea -Dsun.io.useCanonCaches=false -Djava.net.preferIPv4Stack=true -Djsse.enableSNIExtension=false -XX:+UseConcMarkSweepGC -XX:S
4 5930 sun.tools.jps.Jps -mlvV -Dapplication.home=/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home -Xms8m

```

我们发现大多数情况下，“-mlv” 参数组合起来最好用。它会打印main方法的参数、完整包名、JVM 相关参数。这样你就可以在一大堆相似的进程中找到你想要的那个。

现在有了 dump 得到的文件 heap.bin，就可以用你喜欢的内存分析工具来分析。

## 5、jstack

jstack 是一个生成指定 JVM 进程的线程堆栈工具。当你程序一直在那里转圈圈，而你想找到线程到底做了什么导致死锁，那么 jstack 最适合。

jstack 只有几个参数选项，如果你拿不准，把它们都加上。如果后面发现有些信息对你意义不大时可以调整参数限制它的输出。

-F 选项可以用来强制 dump，这在进程挂起时非常有用，-l 选项可以打印同步和锁的信息。

```
1 $ jstack -F -l 9153
2 Attaching to process ID 9153, please wait...
3 Debugger attached successfully.
4 Server compiler detected.
5 JVM version is 25.0-b70
6 Deadlock Detection:
7
8 No deadlocks found.
9 ...
```

上面的输出虽然看起来简单，但是它包含了每个线程的状态和它当前的堆栈的信息。

jstack 非常有用，我们在日常工作中使用非常频繁，特别是我们负责启动停止应用服务器的测试引擎。测试工作往往不顺利，jstack 可以让我们知道 JVM 内部的运行状态且没有什么负面的影响。

— Neeme Praks (ZeroTurnaround资深产品工程师)

---

## 还有其它的吗？

今天我们介绍了 JDK 发行预装的超棒工具。相信我，将来某天你肯定会用到它们中的一些。所以，如果你有时间，你可以翻一翻它们的官方文档。

试着在不同的场景使用并爱上它们。

如果你想学一些超棒的非 JDK 附带的工具，可以看看 JRebel，它可以让你马上看到代码的改动效果，还可以看到我们新的产品 XRebel，它可以像X光眼镜一样扫描你的 web 应用。

如果你知道开发最佳实践中至关重要的小工具，在本文末尾发表评论或者在 twitter上@shelajev 分享一下这个工具的细节。

---

## Bonus Section: References

奖励环节：参考

下面是一个更加完整的 JDK 工具可用列表。虽然这不是一个完整的列表，为了节省篇幅，我们省掉了加密、web-services 相关的工具等。谢谢 manpagez.com 提供的资源。

- [jar](#) — 一个创建和管理 jar 文件的工具。
- [java](#) — Java 应用启动器。在这篇文章里，开发和部署都是用的这个启动器。
- [javac](#) — Java 编译器。
- [javadoc](#) — API 文档生成器。
- [javah](#) — native 本地方法中用于生成 C 语言头文件和源文件。
- [javap](#) — class 文件反编译器。
- [jcmd](#) — JVM 命令行诊断工具，可发送诊断命令请求到 JVM 中。
- [jconsole](#) — 一个兼容 JMX 的监控 JVM 的图形化工具。可以监控本地和远程 JVM，也可以监控和管理单独的一个应用。
- [jdb](#) — Java 调试器。
- [jps](#) — JVM 进程查看工具，列出了系统运行的所有 hotspot JVM 进程。
- [jstat](#) — JVM 状态监控工具。它可以收集和打印指定的 JVM 进程性能状态。
- [jhat](#) — 堆 dump 信息的浏览器，启动一个 web 服务器来显示你用诸如 jmap -dump 得到的堆 dump 信息。
- [jmap](#) — Java 内存映射工具，打印指定进程、核心文件、远程调试服务器共享内存映射或者堆内存详细信息。
- [jsadebugd](#) — Java 服务调试守护进程—依附到一个 Java 进程或核心文件并且担当一个调试服务器的作用。
- [jstack](#) — Java 堆栈信息工具——打印指定进程或核心文件或者远程调试服务器的线程堆栈。
- [jjs](#) — 运行 Nashorn 命令行脚本 shell。
- [jrunscript](#) — Java 脚本运行工具。不过你要心里有数，这实际上是一个还没支持的测试功能。未来的 JDK 版本里面可能会移除它。