# Java 内存泄露监控工具

Java 内存泄露监控工具（一） -- JVM监控工具介绍jstack, jconsole, jinfo, jmap, jdb, jsta

2010-03-17 13:20

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **jstack** -- 如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到 当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。目前只有在Solaris和Linux的JDK版本里面才有。  **jconsole** – jconsole是基于[Java](http://java.chinaitlab.com/) Management Extensions (JMX)的实时图形化监测工具，这个工具利用了内建到JVM里面的JMX指令来提供实时的性能和资源的监控，包括了[Java](http://java.chinaitlab.com/)程序的内存使用，Heap size, 线程的状态，类的分配状态和空间使用等等。  **jinfo** – jinfo可以从core文件里面知道崩溃的Java应用程序的配置信息，目前只有在Solaris和Linux的JDK版本里面才有。  **jmap** – jmap 可以从core文件或进程中获得内存的具体匹配情况，包括Heap size, Perm size等等，目前只有在Solaris和Linux的JDK版本里面才有。  **jdb** – jdb 用来对core文件和正在运行的Java进程进行实时地调试，里面包含了丰富的命令帮助您进行调试，它的功能和Sun studio里面所带的dbx非常相似，但 jdb是专门用来针对Java应用程序的。  **jstat** – jstat利用了JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对Heap size和垃圾回收状况的监控等等。  **jps** – jps是用来查看JVM里面所有进程的具体状态, 包括进程ID，进程启动的路径等等。  [**jstatd**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstatd.html) 启动jvm监控服务。它是一个基于rmi的应用，向远程机器提供本机jvm应用程序的信息。默认端口1099。 实例：jstatd -J-Djava.security.policy=my.policy  my.policy文件需要自己建立，内如如下： grant codebase "file:$JAVA\_HOME/lib/tools.jar" { permission java.security.AllPermission; }; 这是安全策略文件，因为jdk对jvm做了jaas的安全检测，所以我们必须设置一些策略，使得jstatd被允许作网络操作  上面的操作没有通过，出现：  Could not create remote object access denied (java.util.PropertyPermission java.rmi.server.ignoreSubClasses write) java.security.AccessControlException: access denied (java.util.PropertyPermission java.rmi.server.ignoreSubClasses write) at java.security.AccessControlContext.checkPermission(AccessControlContext.java:323) at java.security.AccessController.checkPermission(AccessController.java:546) at java.lang.SecurityManager.checkPermission(SecurityManager.java:532) at java.lang.System.setProperty(System.java:727) at sun.tools.jstatd.Jstatd.main(Jstatd.java:122)  create in your usr/java/bin the jstatd.all.policy file, with the content must be   1. grant codebase "file:${java.home}/../lib/tools.jar" { 2. permission java.security.AllPermission; 3. };   [**jps**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jps.html) 列出所有的jvm实例 实例： jps 列出本机所有的jvm实例  jps 192.168.0.77 列出远程服务器192.168.0.77机器所有的jvm实例，采用rmi协议，默认连接端口为1099 （前提是远程服务器提供jstatd服务）  输出内容如下： jones@jones:~/data/ebook/java/j2se/jdk\_gc$ jps 6286 Jps 6174  Jstat  **jconsole** 一个图形化界面，可以观察到java进程的gc，class，内存等信息。虽然比较直观，但是个人还是比较倾向于使用jstat命令（在最后一部分会对jstat作详细的介绍）。  [**jinfo**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jinfo.html)（linux下特有） 观察运行中的java程序的运行环境参数：参数包括Java System属性和JVM命令行参数 实例：jinfo 2083 其中2083就是java进程id号，可以用jps得到这个id号。 输出内容太多了，不在这里一一列举，大家可以自己尝试这个命令。  [**jstack**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstack.html)（linux下特有） 可以观察到jvm中当前所有线程的运行情况和线程当前状态 jstack 2083 输出内容如下： http://www.517sou.net/Attach/month_1001/lmfqgl_085350_1.png  [**jmap**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jmap.html)（linux下特有，也是很常用的一个命令） 观察运行中的jvm物理内存的占用情况。 参数如下： **-heap**：打印jvm heap的情况 **-histo：**打印jvm heap的直方图。其输出信息包括类名，对象数量，对象占用大小。 **-histo：live ：**同上，但是只答应存活对象的情况 **-permstat：**打印permanent generation heap情况  命令使用： jmap -heap 2083 可以观察到New Generation（Eden Space，From Space，To Space）,tenured generation,Perm Generation的内存使用情况 输出内容： http://www.517sou.net/Attach/month_1001/5rlawx_085353_2.png  jmap -histo 2083 ｜ jmap -histo:live 2083 可以观察heap中所有对象的情况（heap中所有生存的对象的情况）。包括对象数量和所占空间大小。 输出内容： http://www.517sou.net/Attach/month_1001/z9zlmt_085354_3.png 写个脚本，可以很快把占用heap最大的对象找出来，对付内存泄漏特别有效。  [**jstat**](http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jstat.html)最后要重点介绍下这个命令。 这是jdk命令中比较重要，也是相当实用的一个命令，可以观察到classloader，compiler，gc相关信息 具体参数如下： -class：统计class loader行为信息 -compile：统计编译行为信息 -gc：统计jdk gc时heap信息 -gccapacity：统计不同的generations（不知道怎么翻译好，包括新生区，老年区，permanent区）相应的heap容量情况 -gccause：统计gc的情况，（同-gcutil）和引起gc的事件 -gcnew：统计gc时，新生代的情况 -gcnewcapacity：统计gc时，新生代heap容量 -gcold：统计gc时，老年区的情况 -gcoldcapacity：统计gc时，老年区heap容量 -gcpermcapacity：统计gc时，permanent区heap容量 -gcutil：统计gc时，heap情况 -printcompilation：不知道干什么的，一直没用过。  一般比较常用的几个参数是： jstat -class 2083 1000 10 （每隔1秒监控一次，一共做10次） 输出内容含义如下：   |  |  | | --- | --- | | Loaded | Number of classes loaded. | | Bytes | Number of Kbytes loaded. | | Unloaded | Number of classes unloaded. | | Bytes | Number of Kbytes unloaded. | | Time | Time spent performing class load and unload operations. |   jstat -gc 2083 2000 20（每隔2秒监控一次，共做10） 输出内容含义如下：   |  |  | | --- | --- | | S0C | Current survivor space 0 capacity (KB). | | EC | Current eden space capacity (KB). | | EU | Eden space utilization (KB). | | OC | Current old space capacity (KB). | | OU | Old space utilization (KB). | | PC | Current permanent space capacity (KB). | | PU | Permanent space utilization (KB). | | YGC | Number of young generation GC Events. | | YGCT | Young generation garbage collection time. | | FGC | Number of full GC events. | | FGCT | Full garbage collection time. | | GCT | Total garbage collection time. |   输出内容： http://www.517sou.net/Attach/month_1001/spyqva_085355_4.png  如果能熟练运用这些命令，尤其是在linux下，那么完全可以代替jprofile等监控工具了，谁让它收费呢。呵呵。 用命令的好处就是速度快，并且辅助于其他命令，比如grep gawk sed等，可以组装多种符合自己需求的工具。  u              **jps的用法**  用来查看JVM里面所有进程的具体状态, 包括进程ID，进程启动的路径等等。与unix上的ps类似，用来显示本地的java进程，可以查看本地运行着几个java程序，并显示他们的进程号。    **[root@localhost ~]# jps**  25517 Jps  25444 Bootstrap |

u              **jstack的用法**

如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。目前只有在Solaris和Linux的JDK版本里面才有。

**[root@localhost bin]# jstack 25444**

Attaching to process ID 25917, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 1.5.0\_08-b03

Thread 25964: (state = BLOCKED)

Error occurred during stack walking:

sun.jvm.hotspot.debugger.DebuggerException: sun.jvm.hotspot.debugger.DebuggerException: get\_thread\_regs failed for a lwp

        at sun.jvm.hotspot.debugger.linux.LinuxDebuggerLocal$LinuxDebuggerLocalWorkerThread.execute(LinuxDebuggerLocal.java:134)

        at sun.jvm.hotspot.debugger.linux.LinuxDebuggerLocal.getThreadIntegerRegisterSet(LinuxDebuggerLocal.java:437)

        at sun.jvm.hotspot.debugger.linux.LinuxThread.getContext(LinuxThread.java:48)

        at

u              **jstat的用法**

用以判断JVM是否存在内存问题呢？如何判断JVM垃圾回收是否正常？一般的top指令基本上满足不了这样的需求，因为它主要监控的是总体的系统资源，很难定位到java应用程序。

Jstat是JDK自带的一个轻量级小工具。全称“Java Virtual Machine statistics monitoring tool”，它位于java的bin目录下，主要利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对Heap size和垃圾回收状况的监控。可见，Jstat是轻量级的、专门针对JVM的工具，非常适用。由于JVM内存设置较大，图中百分比变化不太明显

一个极强的监视VM内存工具。可以用来监视VM内存内的各种堆和非堆的大小及其内存使用量。

jstat工具特别强大，有众多的可选项，详细查看堆内各个部分的使用量，以及加载类的数量。使用时，需加上查看进程的进程id，和所选参数。

语法结构：

Usage: jstat -help|-options

       jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

参数解释：

Options — 选项，我们一般使用 -gcutil 查看gc情况

vmid    — VM的进程号，即当前运行的java进程号

interval– 间隔时间，单位为秒或者毫秒

count   — 打印次数，如果缺省则打印无数次

S0  — Heap上的 Survivor space 0 区已使用空间的百分比

S1  — Heap上的 Survivor space 1 区已使用空间的百分比

E   — Heap上的 Eden space 区已使用空间的百分比

O   — Heap上的 Old space 区已使用空间的百分比

P   — Perm space 区已使用空间的百分比

YGC — 从应用程序启动到采样时发生 Young GC 的次数

YGCT– 从应用程序启动到采样时 Young GC 所用的时间(单位秒)

FGC — 从应用程序启动到采样时发生 Full GC 的次数

FGCT– 从应用程序启动到采样时 Full GC 所用的时间(单位秒)

GCT — 从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

实例使用1：

**[root@localhost bin]# jstat -gcutil 25444**

  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT

 11.63   0.00   56.46  66.92  98.49 162    0.248    6      0.331    0.579

实例使用2：

**[root@localhost bin]# jstat -gcutil 25444 1000 5**

  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT

 73.54   0.00  99.04  67.52  98.49    166    0.252     6    0.331    0.583

 73.54   0.00  99.04  67.52  98.49    166    0.252     6    0.331    0.583

 73.54   0.00  99.04  67.52  98.49    166    0.252     6    0.331    0.583

 73.54   0.00  99.04  67.52  98.49    166    0.252     6    0.331    0.583

 73.54   0.00  99.04  67.52  98.49    166    0.252     6    0.331    0.583

我们可以看到，5次young gc之后，垃圾内存被从Eden space区(E)放入了Old space区(O)，并引起了百分比的变化，导致Survivor space使用的百分比从73.54%(S0)降到0%(S1)。有效释放了内存空间。绿框中，我们可以看到，一次full gc之后，Old space区(O)的内存被回收，从99.05%降到67.52%。

图中同时打印了young gc和full gc的总次数、总耗时。而，每次young gc消耗的时间，可以用相间隔的两行YGCT相减得到。每次full gc消耗的时间，可以用相隔的两行FGCT相减得到。例如红框中表示的第一行、第二行之间发生了1次young gc，消耗的时间为0.252-0.252＝0.0秒。

常驻内存区(P)的使用率，始终停留在98.49%左右，说明常驻内存没有突变，比较正常。

如果young gc和full gc能够正常发生，而且都能有效回收内存，常驻内存区变化不明显，则说明java内存释放情况正常，垃圾回收及时，java内存泄露的几率就会大大降低。但也不能说明一定没有内存泄露。

GCT 是YGCT 和FGCT的时间总和。

以上，介绍了Jstat按百分比查看gc情况的功能。其实，它还有功能，例如加载类信息统计功能、内存池信息统计功能等，那些是以绝对值的形式打印出来的，比较少用，在此就不做介绍。

**[root@localhost bin]# ps -ef | grep java**

root     25917     1  2 23:23 pts/2    00:00:05 /usr/local/jdk1.5/bin/java -Djava.endorsed.dirs=/usr/local/jakarta-tomcat-5.0.30/common/endorsed -classpath /usr/local/jdk1.5/lib/tools.jar:/usr/local/jakarta-tomcat-5.0.30/bin/bootstrap.jar:/usr/local/jakarta-tomcat-5.0.30/bin/commons-logging-api.jar -Dcatalina.base=/usr/local/jakarta-tomcat-5.0.30 -Dcatalina.home=/usr/local/jakarta-tomcat-5.0.30 -Djava.io.tmpdir=/usr/local/jakarta-tomcat-5.0.30/temp org.apache.catalina.startup.Bootstrap start

jstat -class pid:显示加载class的数量，及所占空间等信息。

实例使用3：

**[root@localhost bin]# jstat -class 25917**

Loaded  Bytes  Unloaded  Bytes     Time

2629    2916.8       29   24.6     0.90

jstat -compiler pid:显示VM实时编译的数量等信息。

实例使用4：

**[root@localhost bin]# jstat -compiler 25917**

Compiled Failed Invalid   Time   FailedType FailedMethod

     768      0       0   0.70            0

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。

**[root@localhost bin]# jstat -gccapacity 25917**

NGCMN       640.0

NGCMX       4992.0

NGC         832.0

S0C         64.0

S1C         64.0

EC          704.0

OGCMN       1408.0

OGCMX       60544.0

OGC         9504.0

OC          9504.0                  OC是old内纯的占用量

PGCMN       8192.0                  PGCMN显示的是最小perm的内存使用量

PGCMX       65536.0                 PGCMX显示的是perm的内存最大使用量

PGC         12800.0                 PGC是当前新生成的perm内存占用量

PC          12800.0                 PC是但前perm内存占用量

YGC         164

FGC         6

jstat -gcnew pid: new对象的信息

**[root@localhost bin]# jstat -gcnew 25917**

 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT

 64.0   64.0   47.4   0.0   2  15   32.0    704.0    145.7    168    0.254

jstat -gcnewcapacity pid: new对象的信息及其占用量

**[root@localhost bin]# jstat -gcnewcapacity 25917**

 NGCMN  NGCMX   NGC   S0CMX  S0C   S1CMX  S1C   ECMX    EC      YGC   FGC

640.0  4992.0  832.0 64.0   448.0 448.0  64.0   4096.0  704.0  168     6

jstat -gcold pid: old对象的信息。

**[root@localhost bin]# jstat -gcold 25917**

   PC       PU        OC          OU       YGC    FGC    FGCT     GCT

 12800.0  12617.6     9504.0      6561.3   169     6    0.335    0.591

jstat -gcoldcapacity pid:old对象的信息及其占用量。

**[root@localhost bin]# jstat -gcoldcapacity 25917**

OGCMN      OGCMX        OGC         OC       YGC   FGC    FGCT     GCT

1408.0     60544.0      9504.0      9504.0   169     6    0.335    0.591

jstat -gcpermcapacity pid: perm对象的信息及其占用量。

[root@localhost bin]# jstat -gcpermcapacity 25917

PGCMN      PGCMX       PGC         PC      YGC   FGC    FGCT     GCT

8192.0    65536.0    12800.0    12800.0   169     6    0.335    0.591

jstat -printcompilation pid:当前VM执行的信息。

**[root@localhost bin]# jstat -printcompilation -h3  25917 1000 5**

每1000毫秒打印一次，一共打印5次，还可以加上-h3每三行显示一下标题。

Compiled  Size  Type Method

     788     73    1 java/io/File <init>

     788     73    1 java/io/File <init>

     788     73    1 java/io/File <init>

Compiled  Size  Type Method

     788     73    1 java/io/File <init>

     788     73    1 java/io/File <init>

u              **jmap的用法**

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，甚至可以将VM 中的heap，以二进制输出成文本。使用方法 jmap -histo pid。如果连用SHELL jmap -histo pid>a.log可以将其保存到文本中去，在一段时间后，使用文本对比工具，可以对比出GC回收了哪些对象。jmap -dump:format=b,file=String 3024可以将3024进程的内存heap输出出来到String文件里。

**[root@localhost bin]# jmap -histo  25917**

Attaching to process ID 26221, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 1.5.0\_08-b03

Iterating over heap. This may take a while...

Unknown oop at 0xaa6e42d0

Oop's klass is null

Object Histogram:

Size    Count   Class description

-------------------------------------------------------

3722768 30467   \* ConstMethodKlass

1976480 25334   char[]

1907880 46994   \* SymbolKlass

1762088 2947    byte[]

1709536 30467   \* MethodKlass

1487816 2600    \* ConstantPoolKlass

1009576 2600    \* InstanceKlassKlass

904880  2199    \* ConstantPoolCacheKlass

741432  30893   java.lang.String

653576  4785    int[]

351760  4397    java.lang.reflect.Method

277824  2894    java.lang.Class

248704  3401    short[]

200888  4411    java.lang.Object[]

193656  4045    java.lang.Object[]

179744  5617    java.util.TreeMap$Entry

175688  1800    java.util.HashMap$Entry[]

165288  6887    java.util.HashMap$Entry

104736  3273    java.lang.ref.SoftReference

104136  4339    java.lang.ref.WeakReference

96096   3521    java.lang.String[]

86160   3590    java.util.Hashtable$Entry

85584   3566    java.util.ArrayList

83472   1206    java.util.Hashtable$Entry[]

82944   1728    java.beans.MethodDescriptor

80560   265     \* ObjArrayKlassKlass

69120   1728    java.util.HashMap

52464   3055    java.lang.Class[]

43040   1076    java.util.Hashtable

42496   664     org.apache.commons.modeler.AttributeInfo

37880   947     java.util.TreeMap

33896   557     javax.management.modelmbean.ModelMBeanAttributeInfo[]

33152   518     java.beans.PropertyDescriptor

616     11      org.springframework.aop.framework.ProxyFactory

608     19      java.util.PropertyPermission

608     38      org.springframework.beans.MutablePropertyValues

608     38      org.springframework.beans.factory.support.MethodOverrides

608     2       \* ArrayKlassKlass

608     38      org.springframework.beans.factory.config.ConstructorArgumentValues

608     4       org.apache.xerces.impl.XMLDTDScannerImpl

576     24      java.util.Stack

576     36      java.util.regex.Pattern$Category

576     24      org.apache.naming.NamingEntry

560     7       java.net.URL[]

552     23      sun.management.MappedMXBeanType$BasicMXBeanType

552     1       java.util.Locale[]

552     22      java.io.ObjectStreamField[]

544     17      java.util.Collections$SynchronizedMap

176     11      java.util.regex.Pattern$Ctype

8       1       sun.reflect.GeneratedMethodAccessor49

8       1       sun.reflect.GeneratedMethodAccessor6

8       1       sun.reflect.GeneratedConstructorAccessor10

Heap traversal took 12.003 seconds.

u              **jinfo的用法**

可以输出并修改运行时的java 进程的opts。用处比较简单，就是能输出并修改运行时的java进程的运行参数。用法是jinfo -opt  pid 如：查看2788的MaxPerm大小可以用  jinfo -flag MaxPermSize 2788。

u              **jconsole的用法**

jconsole:一个java GUI监视工具，可以以图表化的形式显示各种数据。并可通过远程连接监视远程的服务器VM。

用java写的GUI程序，用来监控VM，并可监控远程的VM，非常易用，而且功能非常强。命令行里打 jconsole，选则进程就可以了

不过我没有运行起来，老是报下面的错。会的朋友，帮忙看看。

**[root@localhost bin]# jconsole**

Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:

No X11 DISPLAY variable was set, but this program performed an operation which requires it.        at java.awt.GraphicsEnvironment.checkHeadless(GraphicsEnvironment.java:159)

        at java.awt.Window.<init>(Window.java:317)

        at java.awt.Frame.<init>(Frame.java:419)

        at javax.swing.JFrame.<init>(JFrame.java:194)

        at sun.tools.jconsole.JConsole.<init>(JConsole.java:65)

        at sun.tools.jconsole.JConsole$4.run(JConsole.java:666)

        at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)

        at java.awt.EventQueue.dispatchEvent(EventQueue.java:461)

        at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:242)

        at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:163)

        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)

        at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)

        at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)

u              **jdb的用法**

用来对core文件和正在运行的Java进程进行实时地调试，里面包含了丰富的命令帮助您进行调试，它的功能和Sun studio里面所带的dbx非常相似，但 jdb是专门用来针对Java应用程序的。