讲堂 □ 深入拆解 Java 虚拟机 □ 文章详情

30 | Java虚拟机的监控及诊断工具（命令行篇）

2018-09-28 郑雨迪

30 | Java虚拟机的监控及诊断工具（命令行篇）
朗读人：郑雨迪　10′59″ | 5.04M

今天，我们来一起了解一下 JDK 中用于监控及诊断工具。本篇中我将使用刚刚发布的 Java 11 版本的工具进行示范。

jps

你可能用过`ps`命令，打印所有正在运行的进程的相关信息。JDK 中的`jps`命令（帮助文档）沿用了同样的概念：它将打印所有正在运行的 Java 进程的相关信息。

在默认情况下，`jps`的输出信息包括 Java 进程的进程 ID 以及主类名。我们还可以通过追加参数，来打印额外的信息。例如，`-l`将打印模块名以及包名；`-v`将打印传递给 Java 虚拟机的参数（如`-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`）；`-m`将打印传递给主类的参数。

具体的示例如下所示：

□ 复制代码

```
$ jps -mlv
18331 org.example.Foo Hello World
```

```
18332 jdk.jcmd/sun.tools.jps.Jps -mlv -Dapplication.home=/Library,
```

需要注意的是，如果某 Java 进程关闭了默认开启的`UsePerfData`参数（即使用参数`-XX:-UsePerfData`），那么`jps`命令（以及下面介绍的`jstat`）将无法探知该 Java 进程。

当获得 Java 进程的进程 ID 之后，我们便可以调用接下来介绍的各项监控及诊断工具了。

jstat

`jstat`命令（帮助文档）可用来打印目标 Java 进程的性能数据。它包括多条子命令，如下所示：

复制代码

```
$ jstat -options
-class
-compiler
-gc
-gccapacity
-gccause
-gcmetacapacity
-gcnew
-gcnewcapacity
-gcold
-gcoldcapacity
-gcutil
-printcompilation
```

在这些子命令中，`-class`将打印类加载相关的数据，`-compiler`和`-printcompilation`将打印即时编译相关的数据。剩下的都是以`-gc`为前缀的子命令，它们将打印垃圾回收相关的数据。

默认情况下，`jstat`只会打印一次性能数据。我们可以将它配置为每隔一段时间打印一次，直至目标 Java 进程终止，或者达到我们所配置的最大打印次数。具体示例如下所示：

复制代码

```
# Usage: jstat -outputOptions [-t] [-hlines] VMID [interval [coun
$ jstat -gc 22126 1s 4
S0C    S1C    S0U    S1U      EC       EU        OC         OU
17472,0 17472,0  0,0    0,0   139904,0 47146,4   349568,0   21321
17472,0 17472,0 420,6   0,0   139904,0 11178,4   349568,0   21321
```

```
17472,0 17472,0  0,0   403,9  139904,0 139538,4  349568,0   21323
17472,0 17472,0  0,0    0,0   139904,0   0,0     349568,0   21326
```

> 当监控本地环境的 Java 进程时，VMID 可以简单理解为 PID。如果需要监控远程环境的 Java 进程，你可以参考 jstat 的帮助文档。

在上面这个示例中，22126 进程是一个使用了 CMS 垃圾回收器的 Java 进程。我们利用`jstat`的`-gc`子命令，来打印该进程垃圾回收相关的数据。命令最后的`1s 4`表示每隔 1 秒打印一次，共打印 4 次。

在`-gc`子命令的输出中，前四列分别为两个 Survivor 区的容量（Capacity）和已使用量（Utility）。我们可以看到，这两个 Survivor 区的容量相等，而且始终有一个 Survivor 区的内存使用量为 0。

当使用默认的 G1 GC 时，输出结果则有另一些特征：

复制代码

```
$ jstat -gc 22208 1s
S0C    S1C    S0U    S1U      EC       EU        OC         OU
0,0   16384,0  0,0   16384,0 210944,0 192512,0  133120,0    5332,
0,0   16384,0  0,0   16384,0 210944,0 83968,0   133120,0    5749,
0,0    0,0    0,0    0,0   71680,0  18432,0   45056,0    20285,1
0,0   2048,0  0,0   2048,0 69632,0  28672,0   45056,0    18608,1
...
```

在上面这个示例中，`jstat`每隔 1s 便会打印垃圾回收的信息，并且不断重复下去。

你可能已经留意到，`S0C`和`S0U`始终为 0，而且另一个 Survivor 区的容量（S1C）可能会下降至 0。

这是因为，当使用 G1 GC 时，Java 虚拟机不再设置 Eden 区、Survivor 区，老年代区的内存边界，而是将堆划分为若干个等长内存区域。

每个内存区域都可以作为 Eden 区、Survivor 区以及老年代区中的任一种，并且可以在不同区域类型之间来回切换。（参考链接）

换句话说，逻辑上我们只有一个 Survivor 区。当需要迁移 Survivor 区中的数据时（即 Copying GC），我们只需另外申请一个或多个内存区域，作为新的 Survivor 区。

因此，Java 虚拟机决定在使用 G1 GC 时，将所有 Survivor 内存区域的总容量以及已使用量存放至 S1C 和 S1U 中，而 S0C 和 S0U 则被设置为 0。

当发生垃圾回收时，Java 虚拟机可能出现 Survivor 内存区域内的对象全被回收或晋升的现象。

在这种情况下，Java 虚拟机会将这块内存区域回收，并标记为可分配的状态。这样子做的结果是，堆中可能完全没有 Survivor 内存区域，因而相应的 S1C 和 S1U 将会是 0。

`jstat`还有一个非常有用的参数`-t`，它将在每行数据之前打印目标 Java 进程的启动时间。例如，在下面这个示例中，第一列代表该 Java 进程已经启动了 10.7 秒。

复制代码

```
$ jstat -gc -t 22407
Timestamp        S0C    S1C    S0U    S1U      EC       EU
           10,7  0,0    0,0    0,0    0,0    55296,0  45056,0   34
```

我们可以比较 Java 进程的启动时间以及总 GC 时间（GCT 列），或者两次测量的间隔时间以及总 GC 时间的增量，来得出 GC 时间占运行时间的比例。

如果该比例超过 20%，则说明目前堆的压力较大；如果该比例超过 90%，则说明堆里几乎没有可用空间，随时都可能抛出 OOM 异常。

`jstat`还可以用来判断是否出现内存泄漏。在长时间运行的 Java 程序中，我们可以运行`jstat`命令连续获取多行性能数据，并取这几行数据中 OU 列（即已占用的老年代内存）的最小值。

然后，我们每隔一段较长的时间重复一次上述操作，来获得多组 OU 最小值。如果这些值呈上涨趋势，则说明该 Java 程序的老年代内存已使用量在不断上涨，这意味着无法回收的对象在不断增加，因此很有可能存在内存泄漏。

> 上面没有涉及的列（或者其他子命令的输出），你可以查阅帮助文档了解具体含义。至于文档中漏掉的 CGC 和 CGCT，它们分别代表并发 GC Stop-The-World 的次数和时间。

jmap

在这种情况下，我们便可以请`jmap`命令（帮助文档）出马，分析 Java 虚拟机堆中的对象。

`jmap`同样包括多条子命令。

1. `-clstats`，该子命令将打印被加载类的信息。
2. `-finalizerinfo`，该子命令将打印所有待 finalize 的对象。

3. -histo，该子命令将统计各个类的实例数目以及占用内存，并按照内存使用量从多至少的顺序排列。此外，-histo:live只统计堆中的存活对象。
4. -dump，该子命令将导出 Java 虚拟机堆的快照。同样，-dump:live只保存堆中的存活对象。

我们通常会利用jmap -dump:live,format=b,file=filename.bin命令，将堆中所有存活对象导出至一个文件之中。

这里format=b将使jmap导出与hprof（在 Java 9 中已被移除）、-XX:+HeapDumpAfterFullGC、-XX:+HeapDumpOnOutOfMemoryError格式一致的文件。这种格式的文件可以被其他 GUI 工具查看，具体我会在下一篇中进行演示。

下面我贴了一段-histo子命令的输出：

复制代码

```
$ jmap -histo 22574
 num     #instances         #bytes  class name (module)
-------------------------------------------------------
   1:        500004       20000160  org.python.core.PyComplex
   2:        570866       18267712  org.python.core.PyFloat
   3:        360295       18027024  [B (java.base@11)
   4:        339394       11429680  [Lorg.python.core.PyObject;
   5:        308637       11194264  [Ljava.lang.Object; (java.bas
   6:        301378        9291664  [I (java.base@11)
   7:        225103        9004120  java.math.BigInteger (java.ba
   8:        507362        8117792  org.python.core.PySequence$1
   9:        285009        6840216  org.python.core.PyLong
  10:        282908        6789792  java.lang.String (java.base@1
  ...
2281:             1             16  traceback$py
2282:             1             16  unicodedata$py
Total       5151277      167944400
```

由于jmap将访问堆中的所有对象，为了保证在此过程中不被应用线程干扰，jmap需要借助安全点机制，让所有线程停留在不改变堆中数据的状态。

也就是说，由jmap导出的堆快照必定是安全点位置的。这可能导致基于该堆快照的分析结果存在偏差。举个例子，假设在编译生成的机器码中，某些对象的生命周期在两个安全点之间，那么:live选项将无法探知到这些对象。

另外，如果某个线程长时间无法跑到安全点，jmap将一直等下去。上一小节的jstat则不同。这是因为垃圾回收器会主动将jstat所需要的摘要数据保存至固定位置之中，而jstat只需直接读取即可。

关于这种长时间等待的情况，你可以通过下面这段程序来复现：

复制代码

```
// 暂停时间较长，约为二三十秒，可酌情调整。
// CTRL+C 的 SIGINT 信号无法停止，需要 SIGKILL。
static double sum = 0;

public static void main(String[] args) {
  for (int i = 0; i < 0x77777777; i++) { // counted loop
    sum += Math.log(i); // Math.log is an intrinsic
  }
}
```

jmap（以及接下来的jinfo、jstack和jcmd）依赖于 Java 虚拟机的Attach API，因此只能监控本地 Java 进程。

一旦开启 Java 虚拟机参数DisableAttachMechanism（即使用参数-XX:+DisableAttachMechanism），基于 Attach API 的命令将无法执行。反过来说，如果你不想被其他进程监控，那么你需要开启该参数。

jinfo

jinfo命令（帮助文档）可用来查看目标 Java 进程的参数，如传递给 Java 虚拟机的-X（即输出中的 jvm_args）、-XX参数（即输出中的 VM Flags），以及可在 Java 层面通过System.getProperty获取的-D参数（即输出中的 System Properties）。

具体的示例如下所示：

复制代码

```
$ jinfo 31185
Java System Properties:

gopherProxySet=false
awt.toolkit=sun.lwawt.macosx.LWCToolkit
java.specification.version=11
sun.cpu.isalist=
```

```
sun.jnu.encoding=UTF-8
...

VM Flags:
-XX:CICompilerCount=4 -XX:ConcGCThreads=3 -XX:G1ConcRefinementThr

VM Arguments:
jvm_args: -Xlog:gc -Xmx1024m
java_command: org.example.Foo
java_class_path (initial): .
Launcher Type: SUN_STANDARD
```

`jinfo`还可以用来修改目标 Java 进程的“manageable”虚拟机参数。

举个例子，我们可以使用`jinfo -flag +HeapDumpAfterFullGC <PID>`命令，开启`<PID>`所指定的 Java 进程的`HeapDumpAfterFullGC`参数。

你可以通过下述命令查看其他 "manageable" 虚拟机参数：

复制代码

```
$ java -XX:+PrintFlagsFinal -version | grep manageable
     intx CMSAbortablePrecleanWaitMillis           = 100
     intx CMSTriggerInterval                       = -1
     intx CMSWaitDuration                          = 2000
     bool HeapDumpAfterFullGC                      = false
     bool HeapDumpBeforeFullGC                     = false
     bool HeapDumpOnOutOfMemoryError               = false
    ccstr HeapDumpPath                             =
    uintx MaxHeapFreeRatio                         = 70
    uintx MinHeapFreeRatio                         = 40
     bool PrintClassHistogram                      = false
     bool PrintConcurrentLocks                     = false
java version "11" 2018-09-25
Java(TM) SE Runtime Environment 18.9 (build 11+28)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11+28, mixed mode)
```

jstack

jstack命令（帮助文档）可以用来打印目标 Java 进程中各个线程的栈轨迹，以及这些线程所持有的锁。

jstack的其中一个应用场景便是死锁检测。这里我用jstack获取一个已经死锁了的 Java 程序的栈信息。具体输出如下所示：

复制代码

```
$ jstack 31634
...

"Thread-0" #12 prio=5 os_prio=31 cpu=1.32ms elapsed=34.24s tid=0x(
   java.lang.Thread.State: BLOCKED (on object monitor)
 at DeadLock.foo(DeadLock.java:18)
 - waiting to lock <0x000000061ff904c0> (a java.lang.Object)
 - locked <0x000000061ff904b0> (a java.lang.Object)
 at DeadLock$$Lambda$1/0x0000000800060840.run(Unknown Source)
 at java.lang.Thread.run(java.base@11/Thread.java:834)

"Thread-1" #13 prio=5 os_prio=31 cpu=1.43ms elapsed=34.24s tid=0x(
   java.lang.Thread.State: BLOCKED (on object monitor)
 at DeadLock.bar(DeadLock.java:33)
 - waiting to lock <0x000000061ff904b0> (a java.lang.Object)
 - locked <0x000000061ff904c0> (a java.lang.Object)
 at DeadLock$$Lambda$2/0x0000000800063040.run(Unknown Source)
 at java.lang.Thread.run(java.base@11/Thread.java:834)

...

JNI global refs: 6, weak refs: 0



Found one Java-level deadlock:
=============================
"Thread-0":
  waiting to lock monitor 0x00007fb083015900 (object 0x000000061f
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor 0x00007fb083015800 (object 0x000000061f
```

```
  which is held by "Thread-0"

Java stack information for the threads listed above:
===================================================
"Thread-0":
 at DeadLock.foo(DeadLock.java:18)
 - waiting to lock <0x000000061ff904c0> (a java.lang.Object)
 - locked <0x000000061ff904b0> (a java.lang.Object)
 at DeadLock$$Lambda$1/0x0000000800060840.run(Unknown Source)
 at java.lang.Thread.run(java.base@11/Thread.java:834)
"Thread-1":
 at DeadLock.bar(DeadLock.java:33)
 - waiting to lock <0x000000061ff904b0> (a java.lang.Object)
 - locked <0x000000061ff904c0> (a java.lang.Object)
 at DeadLock$$Lambda$2/0x0000000800063040.run(Unknown Source)
 at java.lang.Thread.run(java.base@11/Thread.java:834)

Found 1 deadlock.
```

我们可以看到，`jstack`不仅会打印线程的栈轨迹、线程状态（BLOCKED）、持有的锁（locked ...）以及正在请求的锁（waiting to lock ...），而且还会分析出具体的死锁。

jcmd

你还可以直接使用`jcmd`命令（帮助文档），来替代前面除了`jstat`之外的所有命令。具体的替换规则你可以参考下表。

至于`jstat`的功能，虽然`jcmd`复制了`jstat`的部分代码，并支持通过`PerfCounter.print`子命令来打印所有的 Performance Counter，但是它没有保留`jstat`的输出格式，也没有重复打印的功能。因此，感兴趣的同学可以自行整理。

另外，我们将在下一篇中介绍`jcmd`中 Java Flight Recorder 相关的子命令。

总结与实践

今天我介绍了 JDK 中用于监控及诊断的命令行工具。我们再来回顾一下。

1. `jps`将打印所有正在运行的 Java 进程。

2. jstat允许用户查看目标 Java 进程的类加载、即时编译以及垃圾回收相关的信息。它常用于检测垃圾回收问题以及内存泄漏问题。
3. jmap允许用户统计目标 Java 进程的堆中存放的 Java 对象，并将它们导出成二进制文件。
4. jinfo将打印目标 Java 进程的配置参数，并能够改动其中 manageabe 的参数。
5. jstack将打印目标 Java 进程中各个线程的栈轨迹、线程状态、锁状况等信息。它还将自动检测死锁。
6. jcmd则是一把瑞士军刀，可以用来实现前面除了jstat之外所有命令的功能。

今天的实践环节，你可以探索jcmd中的下述功能，看看有没有适合你项目的监控项：

复制代码

```
Compiler.CodeHeap_Analytics
Compiler.codecache
Compiler.codelist
Compiler.directives_add
Compiler.directives_clear
Compiler.directives_print
Compiler.directives_remove
Compiler.queue
GC.class_histogram
GC.class_stats
GC.finalizer_info
GC.heap_dump
GC.heap_info
GC.run
GC.run_finalization
VM.class_hierarchy
VM.classloader_stats
VM.classloaders
VM.command_line
VM.dynlibs
VM.flags
VM.info
VM.log
VM.metaspace
VM.native_memory
```

```
VM.print_touched_methods

VM.set_flag

VM.stringtable

VM.symboltable

VM.system_properties

VM.systemdictionary

VM.unlock_commercial_features

VM.uptime

VM.version
```

精选留言

jimforcode 0

老师讲的好像和jdk11 没啥关系吧

2018-09-28

godtrue 0

嘿嘿，就喜欢这样的简单拿来主义，随学随用。老师能否深入讲一下这些命令的底层实现，对应的信息都是怎么获取到的？都是从哪里获取到的？如果说都是从JVM中感觉范围有点大，往细了讲是从JVM的什么地方获取的呢？

2018-09-28

Axis 0

Jdk11下开源了jfr但是没有jmc这个工具查看性能文件 是为什么？

2018-09-28