

参考博客: <https://blog.csdn.net/yx0628/article/details/30091211>

一、JDK 命令行工具

1、jps: 虚拟机进程状况工具

jps (JVM Process Status Tool) 可以列出正在运行的虚拟机进程, 并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID。

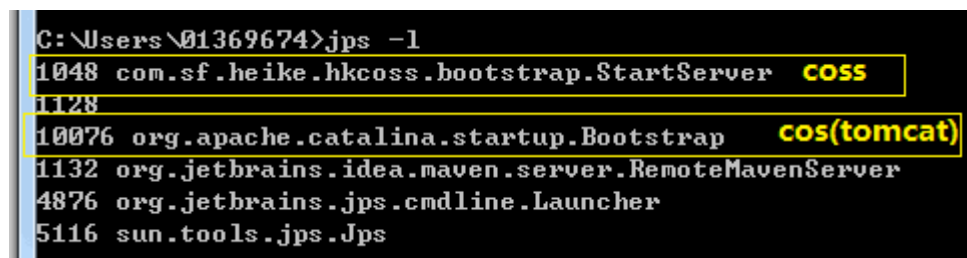
命令格式:

```
jps [options] [hostid]
```

jps的参数:

```
-q # 只输出LVMID, 省略主类的名称  
-m # 输出虚拟机进程启动时传递给主类main()函数的参数  
-l # 输出主类的全名, 如果进城执行的Jar包, 则输出Jar路径  
-v # 输出虚拟机进程启动时JVM参数
```

执行结果:



```
C:\Users\01369674>jps -l  
1048 com.sf.heike.hkcoss.bootstrap.StartServer COSS  
1128  
10076 org.apache.catalina.startup.Bootstrap cos(tomcat)  
1132 org.jetbrains.idea.maven.server.RemoteMavenServer  
4876 org.jetbrains.jps.cmdline.Launcher  
5116 sun.tools.jps.Jps
```

从图中看, `COSS` 和 `cos(tomcat)` 的进程码分别为1048和10076

2、jstat: 虚拟机统计信息监控工具

jstat(JVM Statistics Monitoring Tool) 是用于监视虚拟机各种运行状态信息的命令行工具, 它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据, 在没有GUI图形界面, 只提供了纯文本控制台环境的服务器上, 它将是运行期定位虚拟机性能问题的首选工具。

命令格式:

```
jstat [option vmid [interval[s|ms] [count]]]  
# 参数 interval 和 count 代表查询间隔和次数, 如果省略这两个参数, 说明只查询一次。  
  
jstat -gc 2764 250 20 # 每250毫秒查询一次进程2764垃圾收集情况, 一共查询20次
```

jstat 参数:

<code>-class</code>	# 监视类装载、卸载数量、总空间以及类装载所耗费的时间
<code>-gc</code>	# 监视Java堆状况，包括Eden区、两个Survivor区、老年代、永久代等的容量、已用空间、GC时间合计等信息
<code>-gccapacity</code>	# 监视内容与-gc基本相同，但输出主要关注Java堆各个区域使用到的最大、最小空间
<code>-gcutil</code>	# 监视内容与-gc基本相同，但输出主要关注已使用空间占总空间的百分比
<code>-gccause</code>	# 与-gcutil功能一样，但是会额外输出导致上一次GC产生的原因
<code>-gcnew</code>	# 监视新生代GC状况
<code>-gcnewcapacity</code>	# 监视内容与-gcnew基本相同，输出主要关注使用到的最大、最小空间
<code>-gcold</code>	# 监视老年代GC状况
<code>-gcoldcapacity</code>	# 监视内容与-gcold基本相同，输出主要关注使用到的最大、最小空间
<code>-gpermcapacity</code>	# 输出永久代使用到的最大、最小空间
<code>-compiler</code>	# 输出JIT编译器编译过的方法、耗时等信息
<code>-printcompilation</code>	# 输出已经被JIT编译的方法

执行结果：

- `jstat -class`

```
C:\Users\01369674>jstat -class 1048
Loaded Bytes Unloaded Bytes Time
6359 12767.8 0 0.0 3.57

C:\Users\01369674>jstat -class 10076
Loaded Bytes Unloaded Bytes Time
8399 17859.1 0 0.0 5.69
```

从图中看出：coss 工程装载了6359个class，占用了12767.8 bytes 空间，加载时间为 3.57s。启动 cos 的tomcat 装载了8399个class，占用了 17859.1 bytes，加载时间为 5.69。

- `jstat -gcutil 10076`

S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	GCT
0	67.43	98.35	41.59	-	-	9	0.232	0	0	0.23

查询结果放入表格后如上图，程序运行以来，一共进行了 9 次 young gc（YGC），用时0.232秒（YGCT），一共进行了0次full gc（FGC），用时0秒（FGCT），GC总耗时 0.232秒。S0、S1 是两个Survivor区所占空间，E 为新生代 Eden区所占空间，O表示老年代，M表示永久代（有些jvm版本为P）。

3、jinfo: Java 配置信息工具

jinfo（Configuration Info for Java）实时的查看和调整虚拟机的各项参数，使用jps命令的-v参数可以查看虚拟机启动时显式指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，除了找资料外，就只能使用jinfo的-flag选项进行查询了。

命令格式：

```
jinfo [option] pid
```

执行结果：

```
C:\Program Files\Java\jdk1.7.0_79\bin>jinfo -flags 1048
Attaching to process ID 1048, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 24.79-b02

-agentlib:jdwp=transport=dt_socket,address=127.0.0.1:54752,suspend=y,server=n -D
file.encoding=UTF-8
```

4、jmap: Java 内存映像工具

jmap（Memory Map for Java）命令用于生成堆转储快照（一般称为headdump或dump文件）。还可以查询finalize执行队列，Java堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。

命令格式：

```
jmap [option] vmid
```

jmap 选项：

```
-dump          # 生成Java堆转储快照。格式为：-dump:[live, ] format=b, file=<filename>, 其中live子参数说明是否只dump出存活的对象。
-finalizerinfo # 显示在F-Queue中等待Finalizer线程执行finalize方法的对象。只在Linux/Solaris平台下有效
-heap          # 显示Java堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在Linux/Solaris平台下有效
-histo        # 显示堆中对象统计信息，包括类、实例数量、合计容量
-permstat     # 以ClassLoader为统计口径显示永久代内存状态。只在Linux/Solaris平台下有效
-F            # 当虚拟机进程对-dump选项没有响应时，可使用这个选项强制生成dump快照，只在Linux/Solaris平台下有效
```

5、jhat: 虚拟机堆转储快照分析工具

jhat（JVM Heap Analysis Tool）与jmap搭配使用，分析jmap生成的堆转储快照。jhat内置了一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以在浏览器中查看。不过在实际中不使用jhat来分析dump，主要原因：1.一般不在部署应用的服务器上直接分析dump文件，而要复制到其他机器上进行分析，因为耗时且消耗硬件资源。2.jhat功能比较简陋，如VisualVM，Eclipse Memory Analyzer、IBM HeapAnalyzer等都更强大。

5、jstack: Java 堆栈跟踪工具

jstack（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照（一般称为threaddump或者javacore文件）。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情，或者等待什么资源。

6、HSDIS: JIT 生成代码反编译

HSDIS是一个Sun官方推荐的HotSpot虚拟机JIT编译代码的反汇编插件，它包含在HotSpot虚拟机的源码之中，但没有提供编译后的程序。它的作用是让HotSpot的-XX: +PrintAssembly指令调用它来把动态生成的本地代码还原为汇编代码输出，同时还生成大量非常有价值的注释。

二、JDK 可视化工具

1、JConsole: Java 监视与管理控制台

Jconsole (Java Monitoring and Management Console) 是一种基于JMX的可视化监视、管理工具。

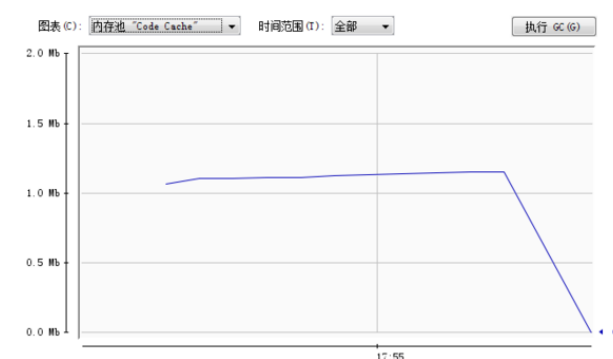
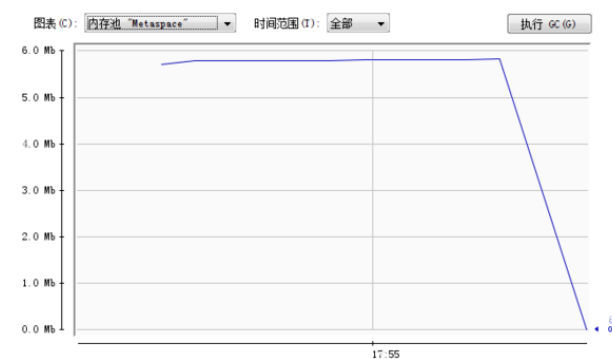
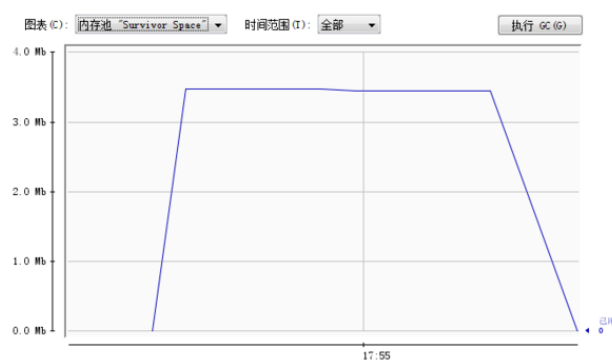
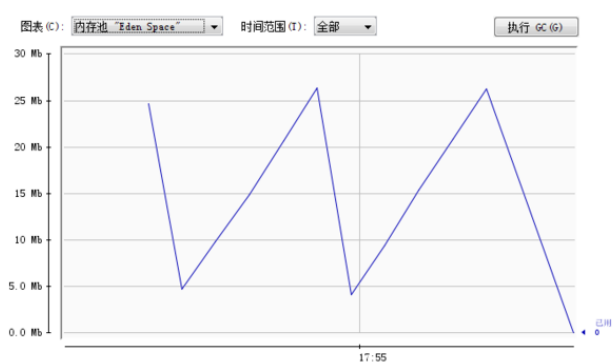
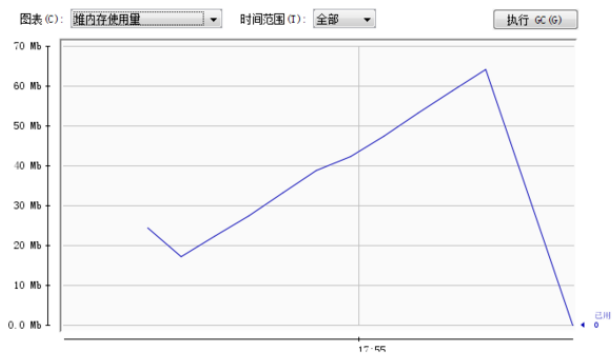
1.1 内存监控 (jstat)

```
/**
 * Jconsole 监视代码
 * 内存占位对象，一个 OOMObject 对象大约 64KB
 */
public class OOMObject {
    public byte[] placeholder = new byte[64*1024];

    public static void fillHeap(int num) throws InterruptedException {
        List<OOMObject> list = new ArrayList<>();
        for (int i = 0; i < num; i++) {
            //稍作延时，让监视曲线变化更明显
            Thread.sleep(50);
            list.add(new OOMObject());
        }
    }

    public static void main(String[] args) throws InterruptedException {
        fillHeap(1000);
    }
}
```

Jconsole 对堆不同区域的监控图：



2.2 线程监控 (jstack)

JConsole 的“线程”标签页相当于可视化jstack，可以监控线程的状态。前面提到线程长时间停顿的主要原因有：等待外部资源（数据库连接、网络资源、设备资源等）、死循环、锁等待（活锁和死锁）。下面通过代码演示一下这几种情况。

2.2.1 线程等待演示

```

public class BusyThread {

    /**
     * 线程死循环演示
     */
    public static void createBusyThread(){
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                int i=0;
                while(true){
                    i++;
                }
            }
        }, "testBusyThread");
        thread.start();
    }

    /**
     * 线程锁等待演示
     * @param lock
     */
    public static void createLockThread(final Object lock){
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (lock){
                    try{
                        lock.wait();
                    }catch (InterruptedException e){
                        e.printStackTrace();
                    }
                }
            }
        }, "testLockThread");
        thread.start();
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        br.readLine();
        createBusyThread();
        br.readLine();
        Object obj = new Object();
        createLockThread(obj);
    }

}

```

监控结果如下：

- main 线程：堆栈追踪显示，BufferedInputStream 在 readBytes 方法等待 System.in 输入。这时线程为 Runnable 状态，Runnable 状态的线程会被分配运行时间，但 readBytes 方法检查到流没有更新时会立即返

回执行令牌，这种等待只消耗很小的 CPU 资源。

名称: main

状态: RUNNABLE

总阻止数: 0, 总等待数: 0

堆栈跟踪:

```
java.io.FileInputStream.readBytes(Native Method)
java.io.FileInputStream.read(FileInputStream.java:255)
java.io.BufferedInputStream.read1(BufferedInputStream.java:284)
java.io.BufferedInputStream.read(BufferedInputStream.java:345)
    - 已锁定 java.io.BufferedInputStream@3da5b6
sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
    - 已锁定 java.io.InputStreamReader@8637ac
java.io.InputStreamReader.read(InputStreamReader.java:184)
java.io.BufferedReader.fill(BufferedReader.java:161)
java.io.BufferedReader.readLine(BufferedReader.java:324)
```

- testBusyThread 线程: 堆栈追踪看到线程一直在 BusyThread.java 24 行停留, 即停留在while(true)死循环语句中。这时线程为Runnable 状态, 而且没有归还线程执行令牌的动作, 会在空循环上用尽全部执行时间知道线程切换, 这种等待会消耗较多的 CPU 资源。

名称: testBusyThread

状态: RUNNABLE

总阻止数: 0, 总等待数: 0

堆栈跟踪:

```
com.susu.study.jvm.BusyThread$1.run(BusyThread.java:24)
java.lang.Thread.run(Thread.java:745)
```

- testLockThread 线程: 堆栈追踪显示 testLockThread 线程在等待 lock 对象的 notify 或 notifyAll 方法的出现, 线程这时候处于 WAITING 状态, 在被唤醒前不会被分配执行时间。

名称: testLockThread

状态: java.lang.Object@401211上的WAITING

总阻止数: 0, 总等待数: 1

堆栈跟踪:

```
java.lang.Object.wait(Native Method)
java.lang.Object.wait(Object.java:502)
com.susu.study.jvm.BusyThread$2.run(BusyThread.java:41)
java.lang.Thread.run(Thread.java:745)
```

2.2.2 死锁代码演示

下面代码有导致死锁的概率, 大概运行 2-3 次会遇到线程死锁。造成死锁的原因是 Integer.valueOf() 方法基于减少对象创建次数和节省内存的考虑, [-128,127] 的数字会被缓存, 代码中 100 次 Integer.valueOf(2),Integer.valueOf(1) 一共只返回了两个不同的对象。

```

public class SynAddRunnable implements Runnable {
    private int a,b;
    public SynAddRunnable(int a,int b){
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        synchronized (Integer.valueOf(a)){
            synchronized (Integer.valueOf(b)){
                System.out.println(a+b);
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new Thread(new SynAddRunnable(1,2)).start();
            new Thread(new SynAddRunnable(2,1)).start();
        }
    }
}

```

出现线程死锁后，点击 JConsole 线程监控页面的检测死锁，可以看到死锁的详情。

- 线程90：状态为 BLOCKED 等待 Integer@1d619db 对象，这个对象的持有者是线程57

线程	死锁
Thread-199	名称: Thread-90 状态: java.lang.Integer@1d619db上的BLOCKED, 拥有者: Thread-57 总阻止数: 1, 总等待数: 0 堆栈跟踪: com.susu.study.jvm.SynAddRunnable.run(SynAddRunnable.java:19) - 已锁定 java.lang.Integer@1520934 java.lang.Thread.run(Thread.java:745)
Thread-57	
Thread-90	

- 线程57：状态为 BLOCKED 等待 Integer@1520934 对象，这个对象的持有者是线程90

线程	死锁
Thread-199	名称: Thread-57 状态: java.lang.Integer@1520934上的BLOCKED, 拥有者: Thread-90 总阻止数: 2, 总等待数: 0 堆栈跟踪: com.susu.study.jvm.SynAddRunnable.run(SynAddRunnable.java:19) - 已锁定 java.lang.Integer@1d619db java.lang.Thread.run(Thread.java:745)
Thread-57	
Thread-90	

两个线程互相等待对方释放资源，产生了死锁，导致线程无法继续执行。

2、Visual VM：多合一故障处理工具

Visual VM 的功能更加强大，除了JConsole的运行监控功能外，还提供了故障处理、性能分析等功能。

自动更新插件时，发现默认下载地址已经迁移，需要查找当前版本的下载地址并修改设置。

Visual VM 插件下载地址：<https://visualvm.github.io/pluginscenters.html>

书上介绍了 heapdump、Profiler、BTrace 工具，未在自动安装插件后找到对应标签，待学习。