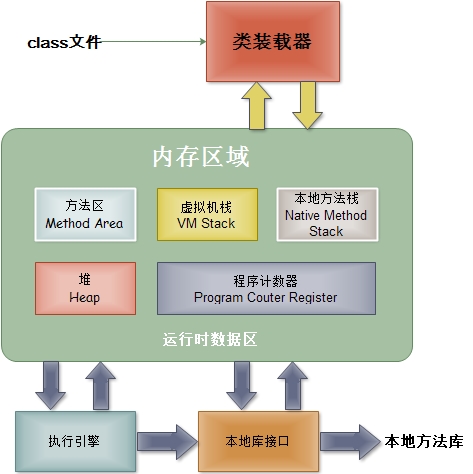
**JVM运行机制以及内存分配机制**



**垃圾回收器**

垃圾回收器（又称为gc）：是负责回收内存中无用的对象。

**类装载子系统**

编译好的.Class文件的系统，作用如下：

1.定位和导入二进制class文件

2.验证导入类的正确性

3.为类分配初始化内存

4.帮助解析符号引用

**执行引擎（Execution Engine）**

执行包在装载类的方法中的指令，也就是方法

**运行时数据区**

如上图：虚拟机内存或者Jvm内存，冲整个计算机内存中开辟一块内存存储Jvm需要用到的对象，变量等，运行区数据有分很多小区，分别为：Method Area（Non-Heap）（方法区）,Heap（堆）,Program Counter Register（程序计数器）, VM Stack（虚拟机栈，也有翻译成JAVA 方法栈的）,Native Method Stack （本地方法栈）。其中Method Area（方法区）和Heap（堆）是线程共享的，VM Stack，Native Method Stack 和Program Counter Register是非线程共享的。JVM初始运行的时候都会分配好Method Area（方法区）和Heap（堆），而JVM 每遇到一个线程，就为其分配一个程序计数器, 虚拟机栈和本地方法栈，当线程终止时，三者（虚拟机栈，本地方法栈和程序计数器）所占用的内存空间也会被释放掉。

1.程序计数器

当前线程执行字节码的位置指示器，线程是私有的，它的生命周期和线程相同。分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

2.虚拟机栈

Java虚拟机栈描述的是Java方法（区别于native的本地方法）执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动作链接、方法出口等信息。

线程私有，生命周期和线程相同，都有各个独立的计数器，各不影响。

每个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

Java虚拟机栈可能出现两种类型的异常：①线程请求的栈深度大于虚拟机允许的栈深度，将抛出StackOverflowError。②虚拟机栈空间可以动态扩展，当动态扩展是无法申请到足够的空间时，抛出OutOfMemory异常。

3.本地方法栈

本地方法栈和虚拟机栈基本类似，只不过Java虚拟机栈执行的是Java代码（字节码），本地方法栈中执行的是本地方法的服务。本地方法栈中也会抛出StackOverflowError和OutOfMemory异常。

4.方法区

方法区是线程共享的区域，用于存储已经被虚拟机加载的类信息，常量，静态变量和即时编译器（JIT）编译后的代码等数据。

运行时常量池是方法区的一部分，Class文件中除了有类的版本，字段，方法，接口等信息以外，还有一项信息是常量池用于存储编译器生成的各种字面量和符号引用，这部分信息将在类加载后存放到方法区的运行时常量池中。

5.堆

堆是Java虚拟机所管理的内存中最大的一块。堆是所有线程共享的一块区域，在虚拟机启动时创建。堆的唯一目的是存放对象实例，几乎所有的对象实例都在这里分配。

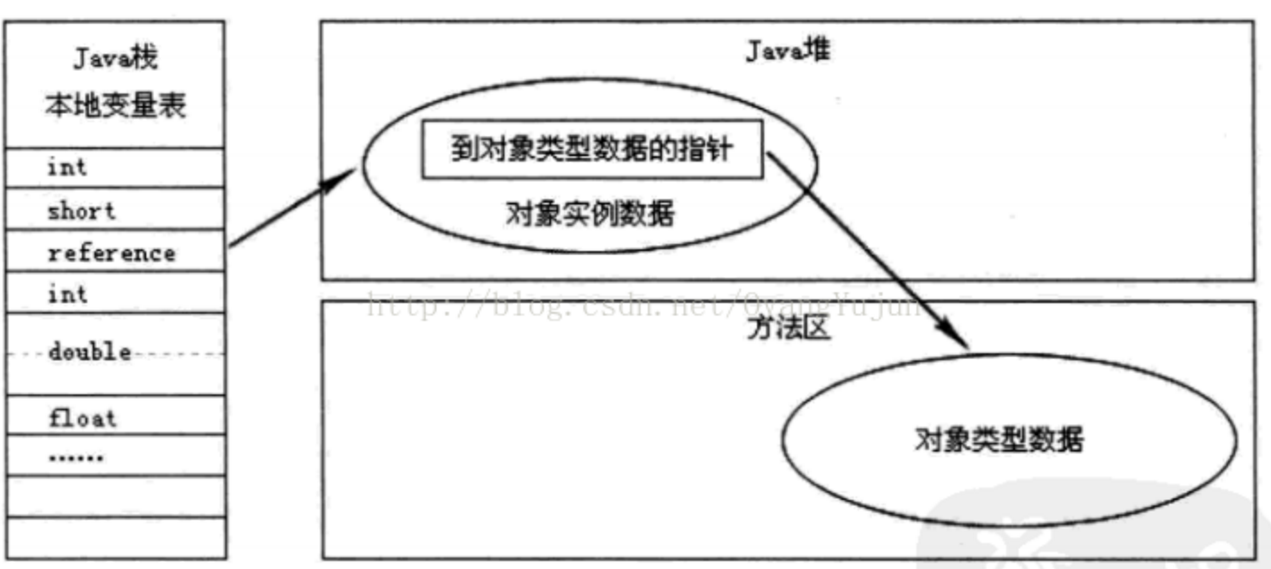
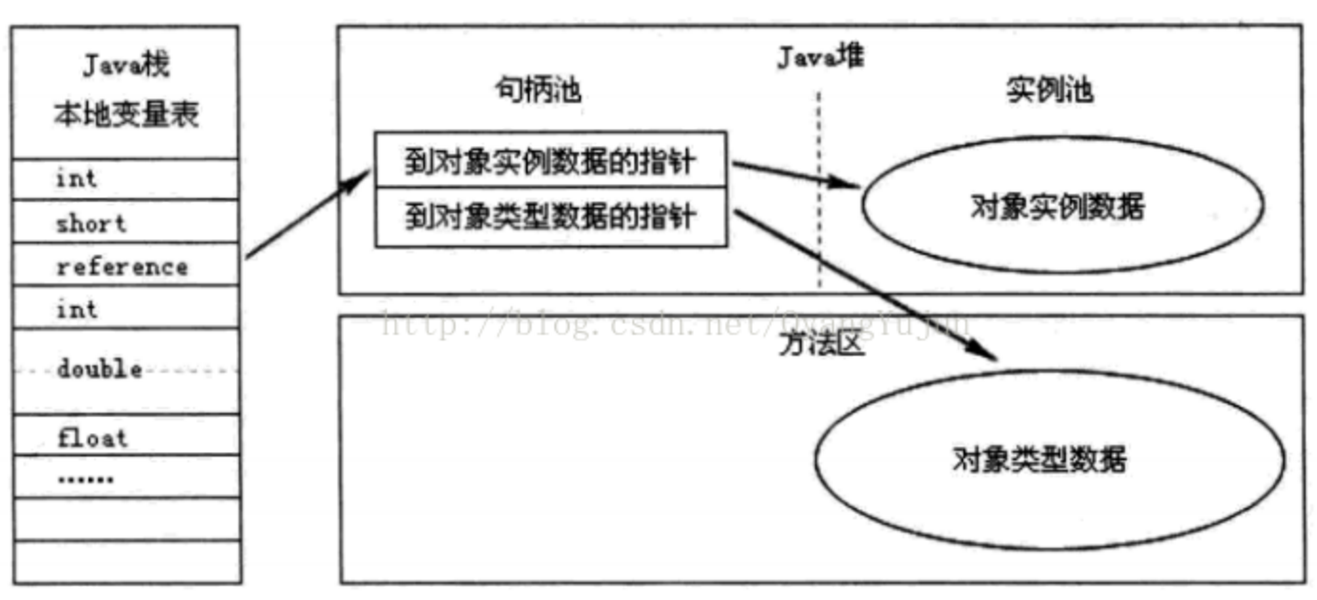
Java堆是垃圾收集器管理的主要区域，所以也称为“GC堆”。由于现在的垃圾收集器基本上都是采用分代收集算法，所以Java堆还可细分为:新生代和老生代。在细致一点可分为Eden空间，From Survivor空间，To Survivor空间。

6.对象访问

**Object obj = new Object();**

**假设这段代码出现在方法体中，那么“Object obj”部分的语义将会反映到Java栈的本地变量表中，作为一个reference类型的数据存在。而“new Object();”部分的语义将会反应到Java堆中，形成一块存储Object类型所有实例数据值（Instance Data）的结构化内存，根据具体类型以及虚拟机实现的对象分布的不同，这块内存的长度是不固定的。另外，在JAVA堆中还必须包含能查找到此对象内存数据的地址信息，这些类型数据则存储在方法区中。**

**由于reference类型在Java虚拟机中之规定了指向对象的引用，并没有规定这个引用要通过哪种方式去定位，以及访问到Java堆中的对象的具体位置，因此虚拟机实现的对象访问方式会有所不同。主流的访问方式有两种：句柄访问方式和直接指针。**



**JVM配置汇总**

**常见配置汇总**

**-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。**

**-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。**

**-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。**

**-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间**

**-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)**

**-XX:+PrintGC:打印GC日志**

**-XX:+PrintGCDetails:打印GC日志**

**-XX:+PrintGCTimeStamps:打印GC日志**

**-Xloggc:filename**

**-XX:+UseSerialGC:设置串行收集器**

**-XX:+UseParallelGC:设置并行收集器【公司内部在用】**

**-XX:+UseParalledlOldGC:设置并行年老代收集器**

**-XX:+UseConcMarkSweepGC:设置并发收集器**

**-Xms:初始堆大小**

**-Xmx:最大堆大小**

**-XX:NewSize=n:设置年轻代大小**

**-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4**

**-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5**

**-XX:MaxPermSize=n:设置持久代大小**

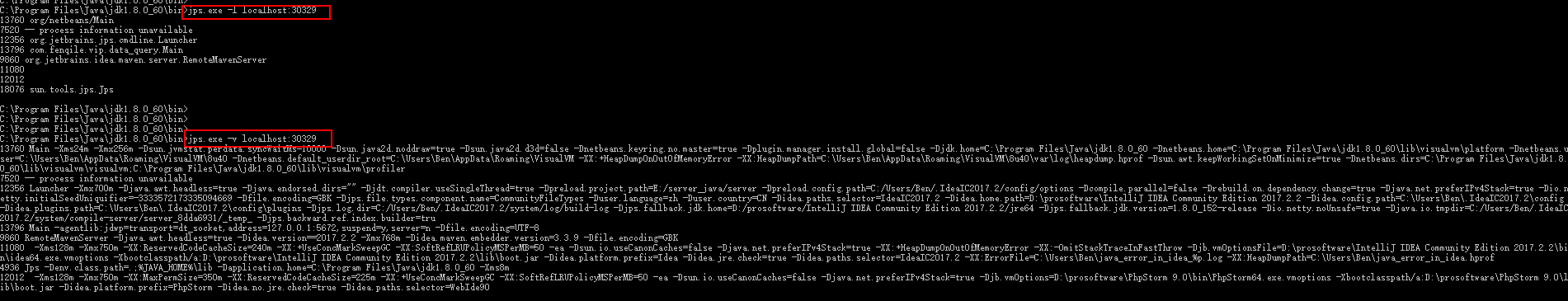
**JVM常用工具**

**Jps：虚拟机进程状况工具**

Jps格式：**jps [option] [hostid]**

|  |  |
| --- | --- |
| 选项 | 作用 |
| -q | 只输出lvmid，省略主类的名称 |
| -m | 输出虚拟机进程启动时2传递给主类main函数的参数 |
| -l | 输出主类的全名，如果进程执行的是jar，输出jar路径 |
| -v | 输出虚拟机进程时JVM参数 |

案例：



**Jstat：虚拟机统计信息监控工具**

Jstat用于监视虚拟机各种运行状态信息的命令行工具。他可以显示本地或远程虚拟机进程的类装载、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。

Jstat命令格式：**jstat [option vmid [interval[s|ms] [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

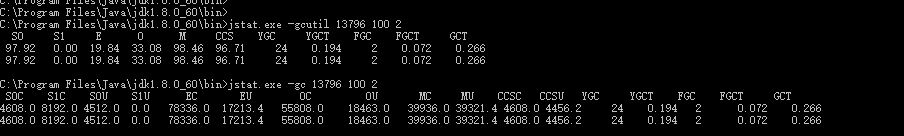
interval– 间隔时间，单位为秒或者毫秒

count — 打印次数，如果缺省则打印无数次

需注意远程和本地虚拟机经常VMID区别

Option分为3类：类装载、垃圾收集、运行期编译状况。

|  |  |
| --- | --- |
| 选项 | 作用 |
| -class | 监视类装载、卸载数量、总空间以及类装载耗费的时间 |
| -gc | 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 |
| -gccapacity | 监视内容与gc基本相同，但输出主要关注java堆各个区域使用到最大最小空间 |
| -gcutil | 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 |
| -gccause | 与gcutil功能一样，但是会额外输出导致上一次GC产生的原因 |
|  |  |
|  |  |



查询结果表示：这台机器的新生代Eden区（E，表示Eden）使用了19.84%的空间，两个Survivor区（S0\S1），老年代(O，表示Old)和永久代（M, 常驻内存区使用率）分别占用33.08%和98.46的空间。程序运行以来共发生Minor GC（YGC，表示Young GC）24次，总耗时0.194秒，发生Full GC（FGC）2次，总耗时0.072秒，所有GC总耗时(GCT，表示GC Time)为0.266秒

Jinfo：java配置信息工具

Jinfo的作用是实时的查看和调整虚拟机各项参数。使用jps命令-v参数可以查看虚拟机启东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系统默认值，除了找资料外，可以使用-flag来查找。



Jmap：java内存映像工具

Jmap命令用于生成堆转储快照（一般称为heapdump或dump文件）。

Jmap作用：查询finalize执行队列、java堆和永久代的详细信息，如空间使用率、当前用的哪种收集器等等。

Jmap命令格式：jmap [option] vmid

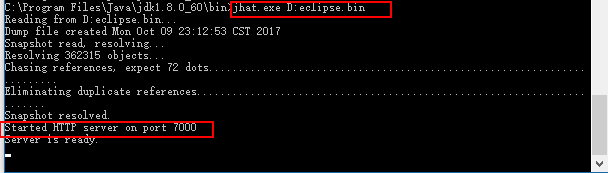
|  |  |
| --- | --- |
| 选项 | 作用 |
| -dump | 生成java堆转储快照。 |
| -finalizerinfo | 显示在F-Queue中等待Finalizer方法的对象。只在linux/Solaris平台下有效 |
| -heap | 显示java堆详细信息。如使用哪种回收器、参数配置、分代状况。只在linux/Solaris平台下有效 |
| -histo | 显示堆中对象统计信息。包括类、实例数量、合计容量 |
| -permstat | 以ClassLoader为统计口径显示永久代内存状态。只在linux/Solaris平台下有效 |
| -F | 当虚拟机进程对-dump选项没有响应，可以使用这个选项强制生成dump快照。只在linux/Solaris平台下有效 |

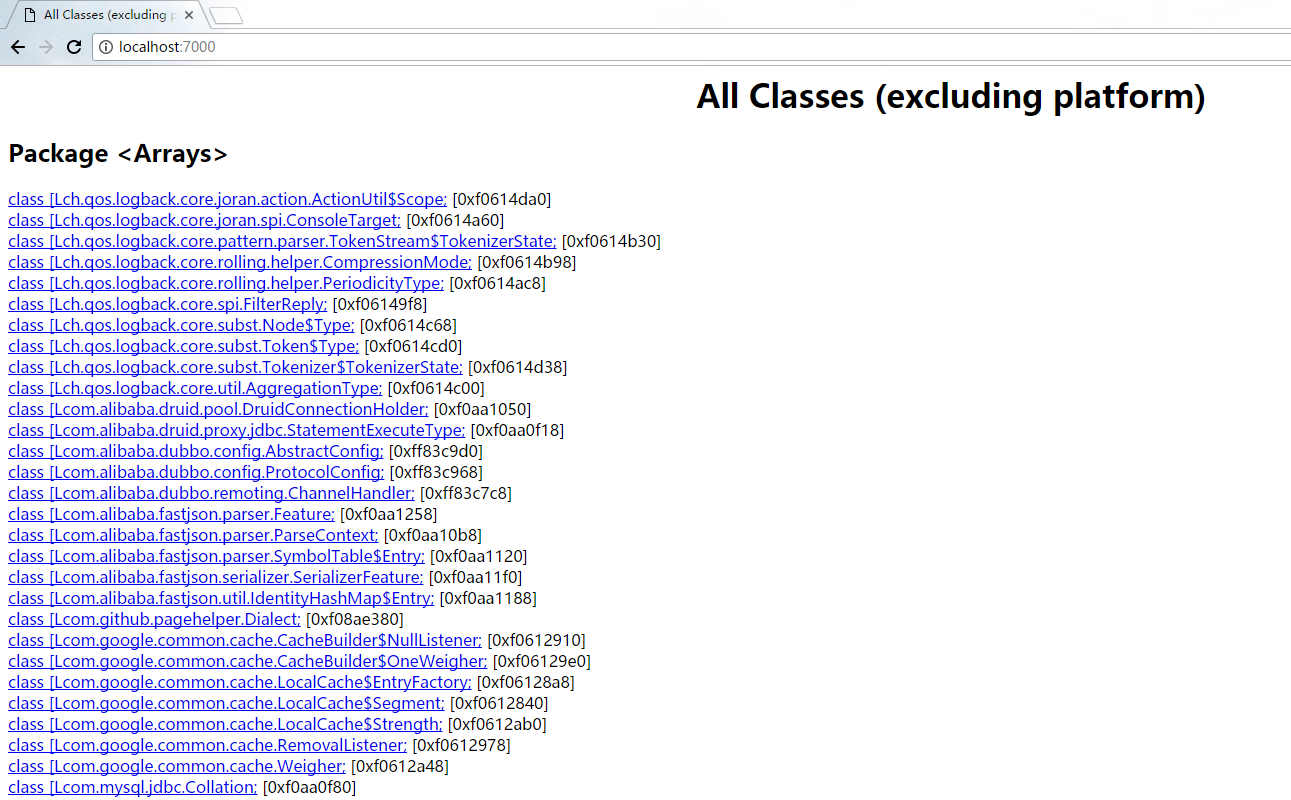
**Jhat：虚拟机堆转储快照分析工具**

Jhat内置一个微型http/html服务器，生成dump问价分析结果后，可以直接在浏览器中查看。

使用jhat分析jmap生成的内存快照文件，当屏幕显示“server is ready”的提示后，用户可以在浏览器输入<http://localhost:7000/>就可以看到分析结果。分析结果默认是以包为单位进行分组显示，分析内存泄漏问题主要会使用到其中的”Heap Histogram”（与jmap -histo功能一样）与OQL页签的功能，前者可以找到内存中国总睿量最大的对象，后者是标准的对象查询语言，使用类似sql语法对内存对象进行查询统计。

案例视图：





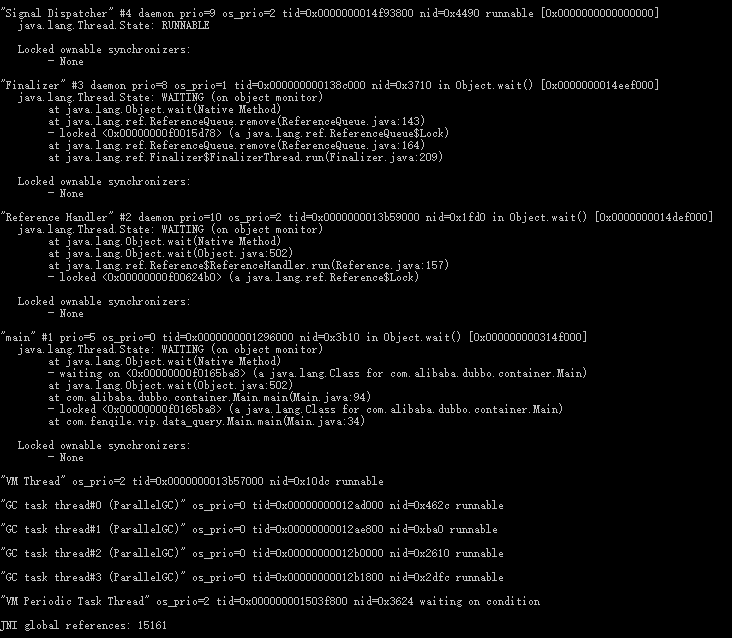
**Jstack：java堆栈跟踪工具**

Jstack命令用于生成虚拟机当前时刻的线程快照（threadump或者javacore文件）。线程快照就是当前虚拟机内每条线程正在执行的方法堆栈的集合吗，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都是导致线程尝试加停顿的常见原因。

Jstack命令格式：jstack [option] vmid

|  |  |
| --- | --- |
| 选项 | 作用 |
| -F | 当正常输出的请求不被响应时，强制输出线程堆栈 |
| -l | 除堆栈外，显示关于锁的附件信息 |
| -m | 如果调用到本地方法，可以显示C/C++堆栈 |

Jdk15中，java.lang.Thread类新增一个getAllStackTraces()方法用于获取 虚拟机中所有线程的StackTraceElement对象。使用这个方法可以通过简单的代码完成jstack的大部分功能，



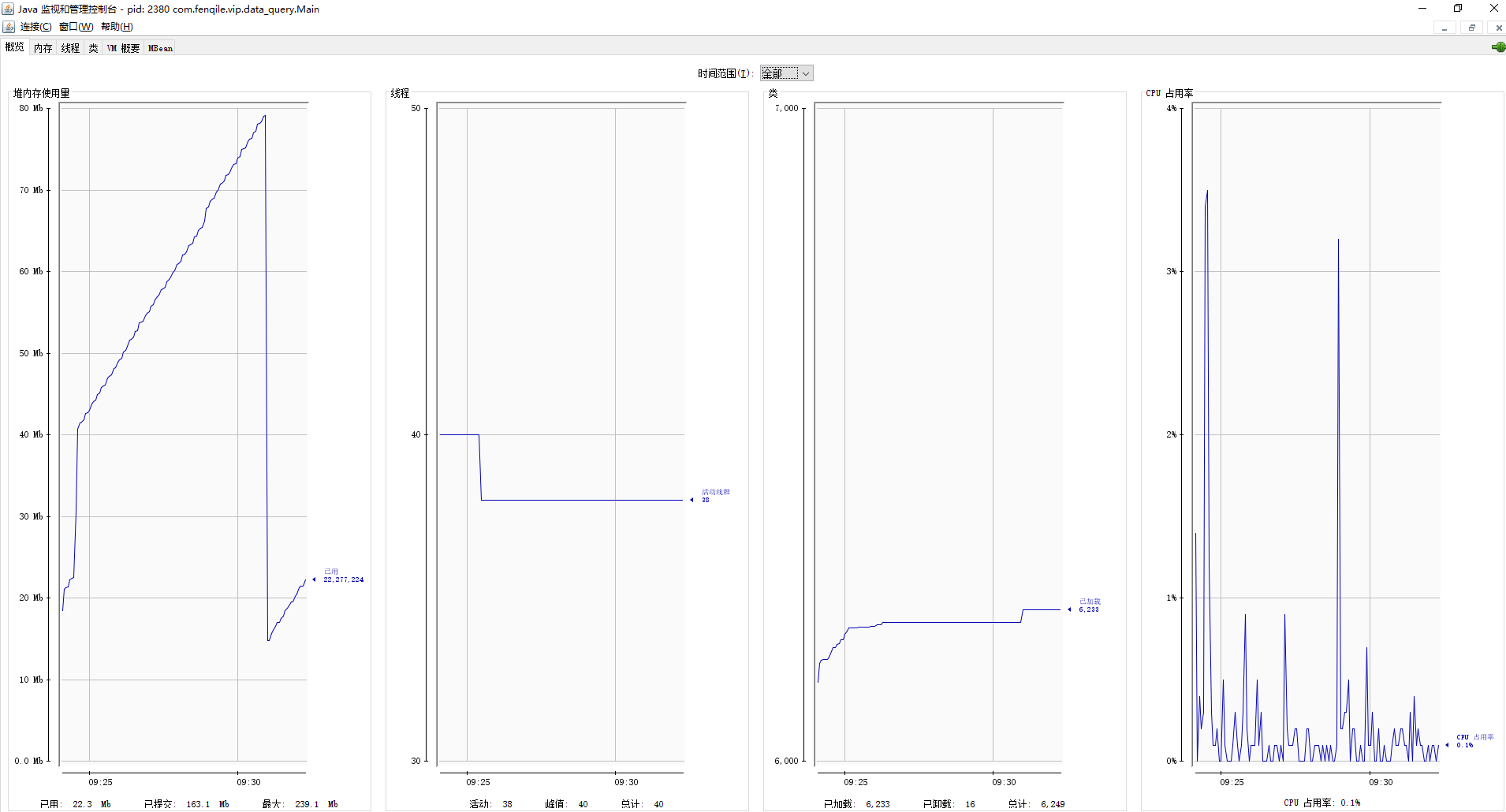
**Hsdis：jit生成代码反汇编**

Hsdis的作用是让HotSpot的-XX:+PritAssembly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，同时生成大量有价值的注释，这样我们可以通过输出代码来分析问题。

**JConsole：java监视与管理控制台**

JConsole是一种基于JMX的可视化监视管理工具。通过双击JDK/bin目录下的“jconsole.exe”启动。

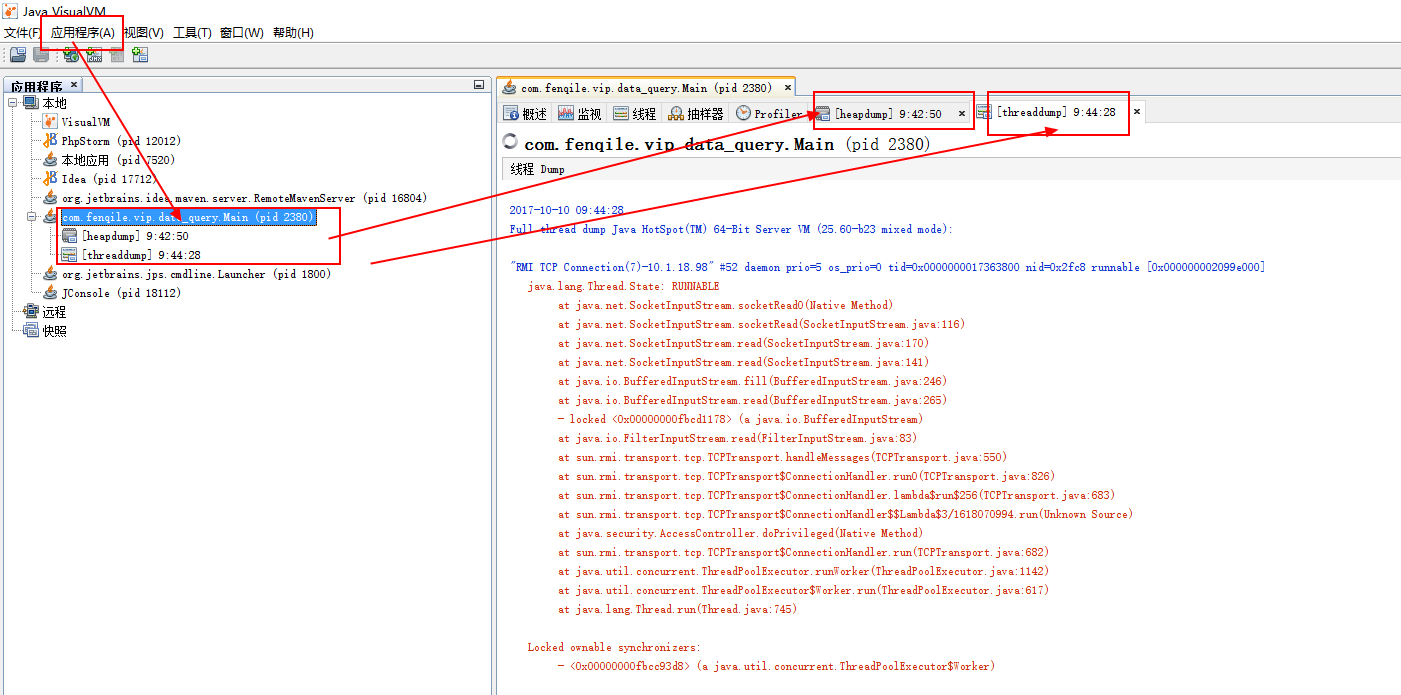
功能涵盖jps、jstat等工具直接视图化查看监控信息



**VisualVM：多合一故障处理工具**

VisualVM是目前为止随JDK发布的功能最强大的运行监视和故障处理程序，并且可以预见在未来一段时间内都是官方主力发展的虚拟机故障处理工具。

**①Visual GC插件**：监视每一部分需要的时间，方便我们进行调优。  
**②Btrace动态日志跟踪插件**：在VisualVM添加该插件，可以不停止目标程序运行的前提下，通过HotSpot虚拟机的HotSwap技术动态加入原本并不存在的调试代码。这项功能对实际生产中的程序很有意义，经常遇到程序出现问题，但缺少排查错误的一些必要信息，如方法参数，返回值等，开发时并没有打印到日志中，以至于不得不停掉服务，通过调试增量来加入日志代码以解决问题。

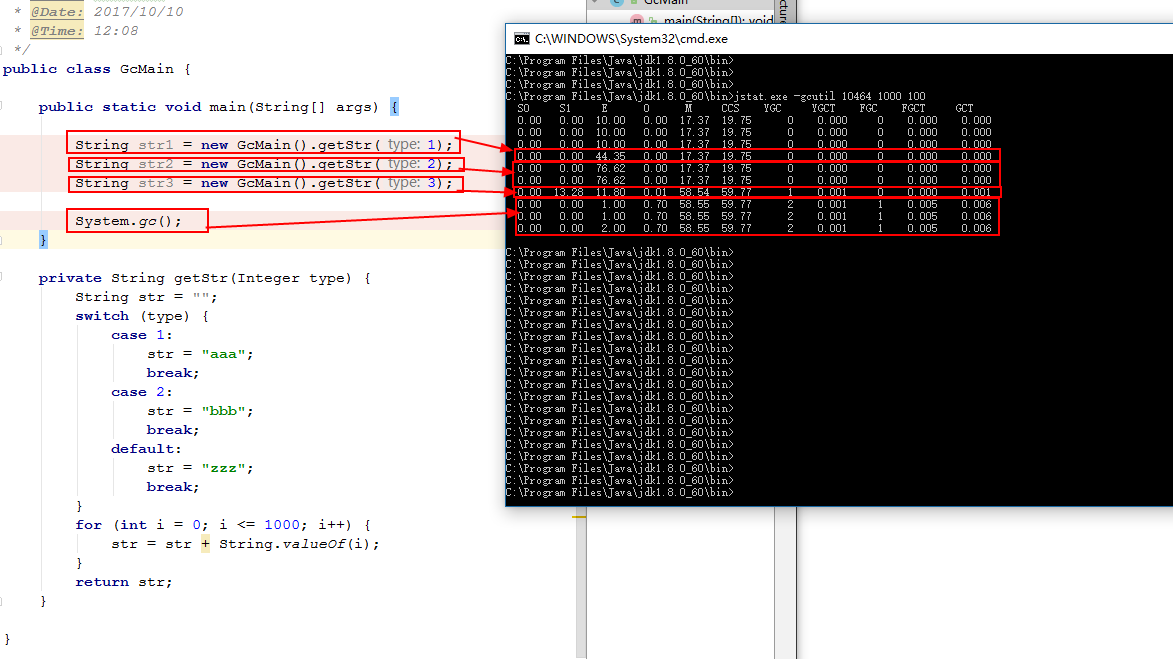


**以下GC案例地址：**https://github.com/ben201708/learn\_java/tree/master/out/production/learn\_java/com/gc

**YoungGC回收案例**

JVM配置如下：

-Xms128m -Xmx256m -XX:ReservedCodeCacheSize=240m -XX:+UseParallelGC -XX:SoftRefLRUPolicyMSPerMB=50 -ea -Dsun.io.useCanonCaches=false -Djava.net.preferIPv4Stack=true -XX:+HeapDumpOnOutOfMemoryError -XX:-OmitStackTraceInFastThrow



**Full GC回收案例**

JVM配置如下：（启动内存设置足够小，idea工具最小设置为5M，添加GC打印日志）

-Xms5m -Xmx5m -XX:+UseParallelGC -XX:+PrintGCDetails

