目录

一、Java内存区域

1. 运行时数据区域

1.1 程序计数器

当前线程所执行的字节码的行号指示器。

1.2 Java虚拟机栈

线程私有，与线程具有相同生命周期。用于存储**局部变量表、操作数栈、动态链**

表、方法出口等信息。

局部变量表存放内容：

- （1）基本数据类型（boolean、byte、char、short、int、float、long、double）
- （2）对象引用（区别于符号引用，符号引用存放在常量池）
- （3）returnAddress类型（指向一条字节码指令的地址）

64位长度的long和double类型数据占用2个局部变量空间（slot），其余占用1个slot。

两种异常：

- （1）StackOverflowError：线程请求的栈深度>虚拟机允许的深度
- （2）OutOfMemoryError: 动态扩展时无法申请到足够内存

1.3 本地方法栈（Native Method Stack）

与虚拟机栈类似，区别是Native Method Stack服务于Native方法，而虚拟机栈服务于Java方法。

1.4 Java堆（Java Heap）

所有线程共享，存放**对象实例、数组**。

垃圾收集器管理的主要区域，也称“GC堆（Garbage Collected Heap）”

包含新生代（Eden空间、From Survivor空间、To Survivor空间）、老生代。

可划分出多个线程私有的分配缓冲区（Thread Local Allocation Buffer，TLAB）。

物理上可以不连续，逻辑上连续。

可扩展：-Xmx和-Xms控制。-Xmx最大堆内存大小，-Xms初始堆内存大小。

当堆中没有可用内存完成实例分配，并且也无法再扩展时——OutOfMemoryError

1.5 方法区（别名Non-Heap）

也是所有线程共享，用于存储**已被虚拟机加载的类信息、常量、静态变量、即时编**

译器编译后的代码等数据。

也称“永久代（Permanent Generation）”，但本质上并不等价。

永久代有-XX:MaxPermSize的上限。

1.6 运行时常量池（Runtime Constant Pool）

属于方法区的一部分。

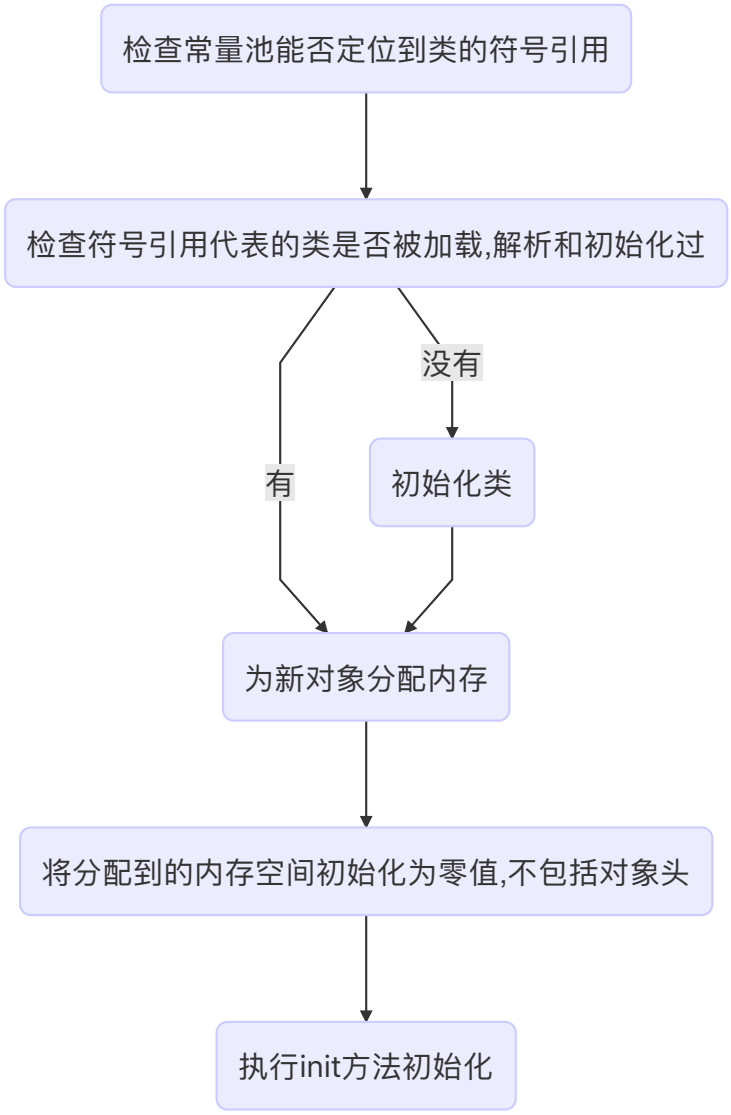
1.7 直接内存（Direct Memory）

JDK1.4新加入的NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可使用Native函数库直接分配**堆外内存**。不受Java堆大小（-Xmx）限制，从而可能造成各个内存区域总和大于物理内存限制而造成动态扩展时出现OutOfMemoryError。

小结：

- 1、2、3三种内存区域是各个线程私有的
- 4、5是所有线程共有的
- 6是5的一部分
- 7不是虚拟机运行时数据区的一部分，属于虚拟机的内存区域外的其他物理内存

2. 对象的创建



为新对象分配内存方式：

- 指针碰撞：堆中内存规整
- 空闲列表：堆中内存不规整

3. 对象的内存布局

对象在内存中存储的布局：

- 对象头（Header）
- 实例数据（Instance Data）——真正的有效数据

- 对齐补充（Padding）——占位符

对象头：

- 自身运行时数据

  包括哈希码、GC分代年龄、锁状态标志、偏向线程ID、偏向时间戳等

- 指针类型

  虚拟机通过此指针确定其属于哪个类的实例。

二、JVM参数

所有线程共享的内存主要有两块：堆内存和方法区。

其中堆内存分为两块：新生代Young generation（Eden区、From Survivor区、To Survivor区）、老年代Tenured generation。

方法区有人也称之“永久代”，但是它们并不等同。方法区是JVM的规范，而永久代是该规范的一种实现方式。从jdk1.7开始已经逐步去除“永久代”，在jdk8中取而代之的是“元空间”（Metaspace）。

元空间与永久代之间最大的区别在于：**元空间并不在虚拟机中，而是使用本地内存**。因此，默认情况下，元空间的大小仅受本地内存限制

下面是JVM的一些主要参数：

1. 基本参数

参数	描述
-XX:+	打开
-XX:-	关闭

2. 内存大小配置参数

参数	描述
-Xms	初始堆内存大小

-Xmx	最大堆内存大小
-Xmn	年轻代内存大小
-Xss	线程私有的虚拟机栈大小
-XX:MaxPermSize=64m	永久代最大值
-XX:PermSize	永久代初始值
-XX:MetaspaceSize	元空间初始大小
-XX:MaxMetaspaceSize	元空间最大值
-XX:MaxDirectMemorySize	直接内存大小，默认与Java堆最大值（-Xmx）一样

3. JVM调试参数

参数	描述
-verbose:gc	记录GC运行及运行时间
-XX:+PrintGCDetails	记录GC运行时的详细数据信息，以及在进程结束时打印当前的内存各区域分配情况。
-XX:+PrintGCTimeStamps	打印垃圾收集时间戳
-Xloggc:{gcLogPath}	gc日志存放路径
-XX:+PrintHeapAtGC	打印GC时堆的更详细的信息
-XX:+HeapDumpOnOutOfMemoryError	在内存溢出的时候生成Heap dump文件
-verbose:class、-XX:+TraceClassLoading	查看类加载信息(要求Product版虚拟机)
-XX:+TraceClassUnLoading	查看类卸载信息(要求FastDebug版虚拟机)

示例：GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -Xloggc:$azkaban_dir/logs/webserver.gc.log -

XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1 -XX:GCLogFileSize=512M"

4. 垃圾收集器

参数	描述
-XX:+UseSerialG	使用Serial+Serial Old的收集器组合进行内存回收。
-XX:+UseParNewGC	使用ParNew+Serial Old的收集器组合进行内存回收。
-XX:+UseConcMarkSweepGC	使用ParNew+CMS+Serial Old的收集器组合进行内存回收。Serial Old作为出现Concurrent Mode Failure失败后的后备收集器使用。
-XX:+UseParallelGC	使用Parallel Scavenge+Serial Old(PS Mark Sweep)收集器组合进行内存回收。
-XX:+UseParallelOldGC	使用Parallel Scavenge+Parallel Old收集器组合进行内存回收。

5. JVM调优参数

参数	描述
-XX:SurvivorRatio	新生代中Eden区域和Survivor区域（单个Survivor）的容量比值，默认为8
-XX:NewRatio	堆内存中老年代和新生代的容量比值。例：NewRatio=2表明Old:New=2:1
-XX:PretenureSizeThreshold	直接晋升到老年代的对象大小，大于该值的对象直接在老年代分配。
-XX:MaxTenuringThreshold	对象在新生代中能存活的最大年龄。
-XX:+UseAdaptiveSizePolicy	动态调整Java堆中各个区域的大小以及进入老年代的年龄（限Parallel Scaverge收集器）
-XX:+HandlePromot	允许老年代分配担保失败，开启后可以冒险YGC。

ionFailure	
-XX:ParallelGCThreads	设置并行GC时进行内存回收的线程数
-XX:GCTimeRatio	用于限制最大GC时间占总时间的比例。默认为99，即允许1%的GC时间。（限Parallel Scaverge收集器）
-XX:MaxGCPauseMillis	设置GC的最大停顿时间（限Parallel Scaverge收集器）
-XX:+CMSInitialingOccupancyFraction	设置CMS收集器在老年代空间被使用多少后触发Full GC。默认值是68，即68%。（限CMS收集器）
-XX:+UseCMSCompactionAtFullCollection	设置CMS在完成垃圾收集后进行一次内存碎片整理。（限CMS收集器）
-XX:+CMSFullGCsBeforeCompaction	设置CMS执行多少次GC后，下次GC时进行一次内存碎片整理，默认为0。即每次都整理。
-Xnoclassgc	不回收无用类

6. 其它

JVM调试参数，可用于远程调试 -Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000

关于GCTimeRatio值的具体算法：

GC时间占总时间的比例由下列公式计算得出：

$$\frac{1}{1+GCTimeRatio}$$

三、垃圾收集器与内存分配策略

内存回收与分配重点关注的是**堆内存**和**方法区内存**（程序计数器占用小，虚拟机栈和本地方法栈随线程有相同的生命周期）。

1. 判断对象是否存活?

1.1 引用计数算法

为每个对象设置一个引用计数器，被引用时加1，引用失效时减1。

优势：实现简单，效率高。

致命缺陷：无法解决对象相互引用的问题——会导致对象的引用虽然存在，但是已经不可能再被使用，却无法被回收。

1.2 可达性分析算法

对象到GC Roots没有引用链，则回收。

GC Roots包括：

- （1）Java虚拟机栈中引用的对象。
- （2）方法区中类静态属性引用的对象。
- （3）方法去中常量引用的对象。
- （4）本地方法栈中Native方法（JNI）引用的对象。

1.3 关于引用

JDK1.2之后，Java对引用进行了扩充。

- （1）强引用：Object obj = new Object()，不会被jvm回收
- （2）软引用：在内存溢出异常发生之前才被强制回收。
- （3）弱引用：延迟到下次垃圾回收之前再被回收。
- （4）虚引用：仅为了在被回收时收到一个系统通知。

1.4 finalize方法：

在对象被JVM回收之前，有一个低优先级的线程去执行。只有覆盖了finalize方法，才会执行，且只会执行一次。

1.5 回收方法区：

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。

2. 垃圾收集算法

2.1 标记-清除算法（Mark-Sweep）

- 效率问题：标记和清除两个过程的效率都不高。
- 空间问题：清除后剩余空间零散不连续，无法为大的对象分配内存。

2.2 复制算法

将内存分为两个半区，将区A中的存活对象全部复制到B区的连续空间，然后清理A中所有空间。

缺点：内存实际空间减半。在对象存活率较高时需要进行较多的复制操作。

实际应用：将堆内存分为新生代和老年代。由于新生代中的对象98%都是可回收的，故将新生代又划分为Eden空间和两块较小的Survivor空间，默认Eden:Survivor（单个）=8:1

这样每次垃圾收集时，将Eden和S1中的存活对象复制到S2，然后清空Eden和S1区。

为了防止S2中空间不足以存储Eden和S1的所有剩余存活对象，提供老年代作为保障（Handle Promotion：分配担保）。

2.3 标记-整理算法（将存活对象移动到一起）

将标记后的存活对象进行移动，清除剩余对象。

应用：老年代

2.4 分代收集

- 新生代：存活率低，使用复制算法
- 老年代：存活率高，使用“标记-整理”或“标记-清除”算法

3. 垃圾收集器

垃圾收集中的并行与并发：

- 并行（Parallel）：多条垃圾收集线程
- 并发（Concurrent）：用户线程与垃圾收集线程同时执行

3.1 Serial收集器

单线程、Stop the World.

Client模式下的默认新生代收集器（一个Client分配给JVM管理的内存一般不会很大，收集时间一般很快）。简单高效（相对于其他单线程收集器）

主要参数：

-XX:SurvivoRatio：新生代中Eden区域和Survivor区域（单个Survivor）的容量比值，默认为8

-XX:PretenureSizeThreshold：直接晋升到老年代的对象大小，大于该值的对象直接在老年代分配。

-XX:HandlePromotionFailure：允许老年代分配担保失败，开启后可以冒险YGC。

3.2 ParNew收集器

Serial的多线程版本（多条垃圾收集线程）。

Server模式下首选的新生代收集器。

除Serial外，只有它能与CMS收集器配合工作。

3.3 Parallel Scavenge收集器

新生代、复制算法、多线程。

注重吞吐量，适合后台进程。

-XX:MacGCPauseMillis：最大垃圾收集停顿时间

-XX:GCTimeRatio：吞吐量大小

GCTimeRatio=99，意味着允许最大垃圾收集时间占比为1/(1+99)=1%，GCTimeRatio=用户代码运行时间/GC时间。

-XX:+UseAdaptiveSizePolicy：动态自适应调整JVM参数（-Xmn、SurvivorRatio等）

3.4 Serial Old收集器

Serial的老年代版本，使用“标准—整理”算法，适合client端。

3.5 Parallel Old收集器

Parallel Scavenge的老年代版本

3.6 CMS收集器（Concurrent Mark Sweep）

老年代、GC系统停顿时间最短

“标记-清除”

四个阶段：

- （1）初始标记：找GC Roots
- （2）并发标记：找引用链
- （3）重新标记：找变更
- （4）并发清除：清除

（1）（3）：stop the world

（2）（4）：与用户线程并发

缺陷：

- 对cpu资源敏感：垃圾收集线程数和每个收集线程占用cpu的时间受cpu数量影响
- 无法处理“浮动垃圾”

  即并发清除阶段用户线程又产生的对象。 -XX:CMSInitiatingOccupancyFranction：老年代被使用的百分比，达到时触发GC（如果等老年代占满了再GC，则GC时并发产生的对象可能就获取不到存储空间）

  CMSInitiatingOccupancyFranction过高会导致大量Concurrent Mode Failure，即老年代预留的内存无法满足程序需要。
- 内存空间碎片

  -XX:+UseCMSCompactAtFullCollection：FullGC时启动内存碎片的合并整理

-XX:CMSFullGCsBeforeCompaction：执行多少次不压缩的FullGC后压缩一次，默认为0。即每次FullGC时都合并整理内存碎片。

3.7 G1收集器

最前沿成果。削弱新生代与老年代概念，将整个堆划分为独立的不同Region。根据各Region的回收价值，确定优先列表。

从整体来看：“标记-整理” 算法

从局部（两个Region之间）来看：“复制”算法

4. 内存分配与回收策略

4.1 优先在Eden区分配（如果启动本地线程分配缓冲TLAB-Thread Local Allocation Buffer，则优先在TLAB）

如果Eden区满，则触发一次Minor GC(也称Young GC)

-XX:+PrintGCDetails

- 在JVM发生垃圾收集时打印内存回收日志
- 在进程退出时输出当前各区域的内存分配情况

在JDK8中，PermGen（永久代）被Metaspace（元空间）取代了。

4.2 大对象直接进入老年代

-XX:PretenureSizeThreshold：直接进入老年代的对象大小

4.3 长期存活的对象将进入老年代

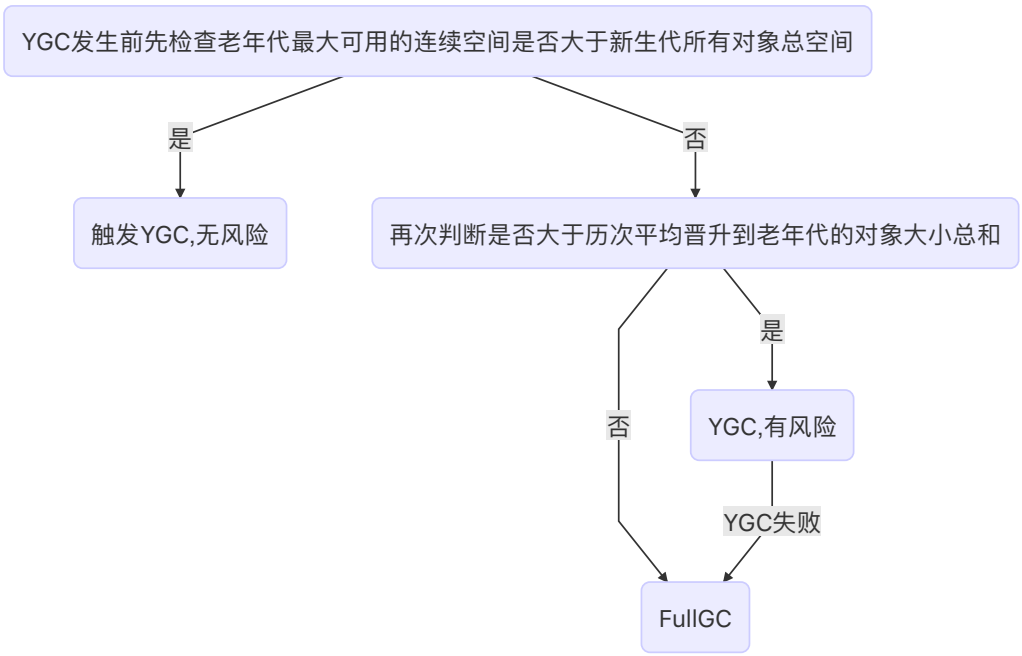
-XX:MaxTenuringThreshold：设置对象在新生代中能存活的最大年龄，默认15

-XX:+PrintTenuringDistribution：打印老年代内的各年龄对象内存分配情况

4.4 动态对象年龄判定

若Survivor中相同年龄的所有对象大小总和超过Survivor的一半，则年龄大于或等于该年龄的对象就可以直接进入老年代。

4.5 空间分配担保



5. 垃圾回收和GC实战

5.1 YGC测试代码：

```
/**
 * @author zni.feng
 */
import java.lang.management.ManagementFactory;

public class YGCTest {
        /**
         * VM参数:  -verbose:gc -XX:+UseSerialGC  -Xms20M -Xmx20M -
Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
         */
        private static final int _1MB= 1024 * 1024;

        public static void main(String[] args) {

System.out.println(ManagementFactory.getRuntimeMXBean().getInputArg
uments());//打印JVM参数
                byte[] allocation1, allocation2, allocation3,
```

```
allocation4;
                allocation1 = new byte[2*_1MB];
                allocation2 = new byte[2*_1MB];
                allocation3 = new byte[2*_1MB];
                allocation4 = new byte[4*_1MB]; //出现一次YGC
                System.out.println("exit");
        }

}
```

测试结果：

```
[-verbose:gc, -XX:+UseSerialGC, -Xms20M, -Xmx20M, -Xmn10M, -
XX:+PrintGCDetails, -XX:SurvivorRatio=8, -Dfile.encoding=UTF-8]
[GC (Allocation Failure) [DefNew: 6815K->282K(9216K), 0.0077121
secs] 6815K->6426K(19456K), 0.0077785 secs] [Times: user=0.01
sys=0.01, real=0.01 secs]
exit
Heap
 def new generation   total 9216K, used 4620K [0x00000007bec00000,
0x00000007bf600000, 0x00000007bf600000)
  eden space 8192K,  52% used [0x00000007bec00000,
0x00000007bf03c8d8, 0x00000007bf400000)
  from space 1024K,  27% used [0x00000007bf500000,
0x00000007bf546800, 0x00000007bf600000)
  to   space 1024K,   0% used [0x00000007bf400000,
0x00000007bf400000, 0x00000007bf500000)
 tenured generation   total 10240K, used 6144K [0x00000007bf600000,
0x00000007c0000000, 0x00000007c0000000)
   the space 10240K,  60% used [0x00000007bf600000,
0x00000007bfc00030, 0x00000007bfc00200, 0x00000007c0000000)
 Metaspace       used 2695K, capacity 4486K, committed 4864K,
reserved 1056768K
  class space    used 294K, capacity 386K, committed 512K, reserved
1048576K
```

测试结果分析：

从结果可以看到，发生了一次YGC，GC后新生代（DefNew：Default New Generaion）从6815K降到了282K，即基本上全部回收，而整个堆内存的大小并没有显著减小（从6815K降到了6426K），这是因为allocation1、allocation2、allocation3对象仍存在（强引用），并没有被真正回收，只是从新生代进入了老年代（Survivor区不足以容纳6M的对象）。并且，从程序结束时的各区域内存分配情况可以看到，老年代占用了约6M（tenured generation total 10240K,used

6144K）；新生代Eden区占用了52%，约4M（total 8192K，即8M），是用来存放allocation4的对象。

5.2 大对象直接进入老年代

```
/**
 * @author zni.feng
 */
public class PretenureSizeThresholdTest {
        /**
         * VM参数: -verbose:gc -XX:+UseSerialGC  -Xms20M -Xmx20M -
Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
         * -XX:PretenureSizeThreshold=3145728
         * (3145728=3*1024*1024=3MB)
         */

        private static final int _1MB=1024*1024;
        public static void main(String[] args) {
                byte[] allocation;
                allocation = new byte[4*_1MB];

        }

}
```

测试结果：

```
Heap
 def new generation   total 9216K, used 835K [0x00000007bec00000,
0x00000007bf600000, 0x00000007bf600000)
  eden space 8192K,  10% used [0x00000007bec00000,
0x00000007becd0f90, 0x00000007bf400000)
  from space 1024K,   0% used [0x00000007bf400000,
0x00000007bf400000, 0x00000007bf500000)
  to   space 1024K,   0% used [0x00000007bf500000,
0x00000007bf500000, 0x00000007bf600000)
 tenured generation   total 10240K, used 4096K [0x00000007bf600000,
0x00000007c0000000, 0x00000007c0000000)
   the space 10240K,  40% used [0x00000007bf600000,
0x00000007bfa00010, 0x00000007bfa00200, 0x00000007c0000000)
 Metaspace       used 2621K, capacity 4486K, committed 4864K,
reserved 1056768K
  class space    used 286K, capacity 386K, committed 512K, reserved
1048576K
```

测试结果分析：可以看到，allocation对象直接进入了老年代。

四、JDK自带的JVM性能监控命令和工具

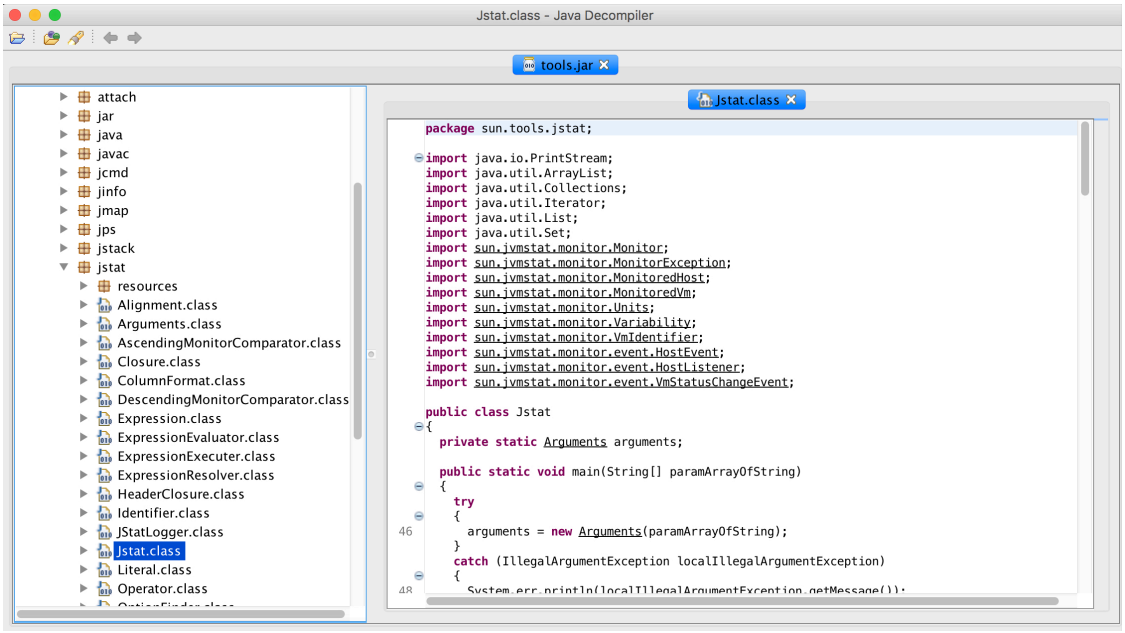
本章节介绍的性能监控命令和工具都是JDK自带的，执行目录都在$JAVA_HOME/bin下。 需要说明下，我机子使用的jdk版本是1.8.0_131，不同版本的jdk可能包含的内容稍有不同。

1. JDK的命令行工具

JDK用于性能监控的主要命令行工具有jps,jstat,jinfo,jmap,jstack，都在$JAVA_HOME/bin目录下，如下图所示：

```
hadoop@hzbxs-ds-bigdata-19:~/java-current/bin$ ls -lh
total 236K
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 appletviewer
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 apt
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 extcheck
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 idlj
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jar
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jarsigner
lrwxrwxrwx 1 root root   15 Aug 25 23:12 java -> ../jre/bin/java
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 javac
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 javadoc
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 javah
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 javap
-rwxr-xr-x 1 root root 2.8K Aug 25 01:21 java-rmi.cgi
lrwxrwxrwx 1 root root   17 Oct  2  2013 javaws -> ../jre/bin/javaws
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jcmd
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jconsole
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jdb
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jhat
-rwxr-xr-x 1 root root 6.4K Aug 25 23:12 jinfo
-rwxr-xr-x 1 root root 6.4K Aug 25 23:12 jmap
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jps
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jrunscript
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jsadebugd
-rwxr-xr-x 1 root root 6.4K Aug 25 23:12 jstack
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jstat
-rwxr-xr-x 1 root root 6.3K Aug 25 23:12 jstatd
```

可以看到，这些工具的体积都很小。事实上它们都只是对jdk/lib/tool.jar类库的一层简单包装而已。真正的代码实现都在tool.jar中：



对tool.jar进行反编译可以看到，这些命令行工具的具体实现都在sun/tools目录下，想了解这些jdk命令工具的具体实现可以从这里去查。

常用命令工具基本功能：

命令	功能
jps	JVM Process Status Tool, 显示系统内所有的HotSpot虚拟机进程
jstat	JVM Statistics Monitoring Tool，用于收集HotSpot虚拟机各方面的运行数据
jinfo	Configure Info for Java，显示虚拟机配置信息
jmap	Memory Map for Java，生成虚拟机的内存存储快照（heapdump文件）
jstack	Stack Trace for Java，显示虚拟机的线程快照

下面列举的是各个命令工具个人觉得比较有用的参数（建议在linux下操作，很多参数只限linux环境使用）：

1.1 jps 虚拟机进程状况工具

命令格式： jps [option] [hostid]

备注：hostid是因为jdk1.6之后，可以基于JMX管理服务监控远程计算机的java进程信息，如果在本机执行可以忽略这个参数。

option	作用
-m	输出进程启动时传递给主类main()函数的参数
-l	输出主类全名，如果执行的是Jar包，输出Jar路径
-v	输出虚拟机进程启动时的JVM参数

示例：

```
hadoop@hzbxs-ds-bigdata-19:~$ jps -lmv | grep azkaban
63077 azkaban.execapp.AzkabanExecutorServer -conf bin/../conf -javaagent:bin/../sentry-javaagent-home/sentry-javaagent-premain-2.0.0.jar -Dsentry_collector_libpath=bin/../lib -Dlog4j.configuration=bin/../conf/log4j.properties -Xmx3G -Dcom.sun.management.jmxremote -Djava.io.tmpdir=/tmp -Dexecutorport=12321 -Dserverpath=/home/hadoop/exec -Dlog4j.log.dir=bin/../logs -Dcom.netease.appname=AzkabanExecutorServer -Djava.library.path=/home/hadoop/hadoop-current/lib/native
```

即运行的主类为azkaban.webapp.AzkabanWebserver，输入的参数为-conf bin/../conf。后面则是jvm的启动参数。

1.2 jstat 虚拟机统计信息监视工具

命令格式： jstat [option] [vmid] [interval] [count]

其中

- vmid在本地就等于pid
- interval为采样间隔时间，默认单位ms
- count是总计采样次数。缺省时则为持续采样。

option	作用
-class	监视类装载、卸载数量、总空间以及类装载所耗费的时间
-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	与-gc基本相同，主要关注各个区域使用到的最大、最小空间
-gcutil	与-gc基本相同，主要关注已使用空间与总空间的百分比

-gccause	与-gcutil相同，额外输出导致上一次GC产生的原因
-gcnew	监视新生代GC状况
-gcold	监视老生代GC状况

示例：

```
hadoop@hzbxs-ds-bigdata-19:~$ jstat -gcutil 63077 1000 3
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
 25.00   0.00  87.13  35.47  48.40   4470   24.285     4    0.945   25.229
 25.00   0.00  88.43  35.47  48.40   4470   24.285     4    0.945   25.229
 25.00   0.00  89.74  35.47  48.40   4470   24.285     4    0.945   25.229
```

其中-gcutil输出的是各内存区域已使用空间占总空间的百分比，每隔250s采样一次，共采样4次。各参数代表的含义如下：

- S0：S0区内存使用百分比
- S1：S1区内存使用百分比
- E：Eden区内存使用百分比
- O：老年代内存使用百分比
- P： 永久代内存使用百分比
- YGC：从进程启动到现在总计发生YGC次数
- YGCT：从进程启动到现在总计发生YGC的时间
- FGC：从进程启动到现在总计发生FullGC的次数
- FGCT：从进程启动到现在总计发生FullGC的时间
- GCT：从进程启动到现在总计发生GC的时间，等于YGCT+FGCT

1.3 jinfo Java配置信息工具

命令格式：jinfo [option] pid

option	作用
-flags	输出JVM的启动参数

-sysprops	输出Java进程的所有系统参数

缺省option时，同时输出以上两者。

jinfo -flag ThreadStackSize pid 打印进程的默认线程栈大小

小坑： 使用时可能发生如下异常：

```
Attaching to process ID 63077, please wait...
Exception in thread "main" java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:622)
        at sun.tools.jinfo.JInfo.runTool(JInfo.java:97)
        at sun.tools.jinfo.JInfo.main(JInfo.java:71)
Caused by: sun.jvm.hotspot.runtime.VMVersionMismatchException: Supported versions are 23.25-b01. Target VM is 24.151-b01
        at sun.jvm.hotspot.runtime.VM.checkVMVersion(VM.java:234)
        at sun.jvm.hotspot.runtime.VM.<init>(VM.java:297)
        at sun.jvm.hotspot.runtime.VM.initialize(VM.java:367)
        at sun.jvm.hotspot.bugspot.BugSpotAgent.setupVM(BugSpotAgent.java:598)
        at sun.jvm.hotspot.bugspot.BugSpotAgent.go(BugSpotAgent.java:493)
        at sun.jvm.hotspot.bugspot.BugSpotAgent.attach(BugSpotAgent.java:331)
        at sun.jvm.hotspot.tools.Tool.start(Tool.java:163)
        at sun.jvm.hotspot.tools.JInfo.main(JInfo.java:128)
        ... 6 more
```

原因是，azkaban executor进程在启动脚本里设置了jdk是1.7，而该云主机默认使用的jdk是1.6。解决方式是使用jdk1.7的jinfo命令。进入jdk1.7对应的bin目录下，使用./jinfo pid命令即可解决该问题。

1.4 jmap Java内存映像工具

命令格式：jmap [option] vmid

option	作用
-dump	生成Java堆转储快照，示例：jmap -dump:live,format=b,file=heap.bin <pid>
-finalizerinfo	显示在F-Queue中等待Finalizer线程执行finalize方法的对象
-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
-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

-F	当虚拟机进程对-dump没有响应时，可使用此选项强制生成dump快照

示例：jmap -histo 32839 | less

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:         95712       14400664  <constMethodKlass>
   2:         95712       12263120  <methodKlass>
   3:          7576        9530360  <constantPoolKlass>
   4:         33028        6246200  [B
   5:          7571        5995600  <instanceKlassKlass>
   6:         60611        5852912  [C
   7:          6134        5044416  <constantPoolCacheKlass>
   8:          3469        1927200  [I
   9:          3151        1634376  <methodDataKlass>
  10:         55869        1340856  java.lang.String
  11:           229         933472  [Ljava.nio.ByteBuffer;
  12:         28062         897984  java.util.HashMap$Entry
  13:          8124         783896  java.lang.Class
  14:         14867         750840  [Ljava.lang.Object;
  15:         11843         722856  [S
  16:         12645         716360  [[I
  17:         20764         498336  java.util.Date
  18:         20596         494304  java.util.LinkedList$Node
  19:         20105         482520  azkaban.metric.inmemoryemitter.InMemoryHistoryNode
  20:          5931         474480  java.lang.reflect.Method
  21:         11407         365024  java.util.concurrent.ConcurrentHashMap$HashEntry
  22:          1779         322872  [Ljava.util.HashMap$Entry;
  23:         12838         308112  com.netease.sentry.javaagent.thirdparty.javassist.bytecode.Utf8Info
  24:          9403         300896  java.util.Hashtable$Entry
  25:           536         287296  <objArrayKlassKlass>
  26:         12853         277432  [Ljava.lang.Class;
  27:         16946         271136  java.lang.Object
  28:         10828         259872  java.util.ArrayList
  29:          4703         188120  java.util.LinkedHashMap$Entry
  30:          1343         150416  java.net.SocksSocketImpl
```

jmap -histo:live 44437 查看live状态的实例对象

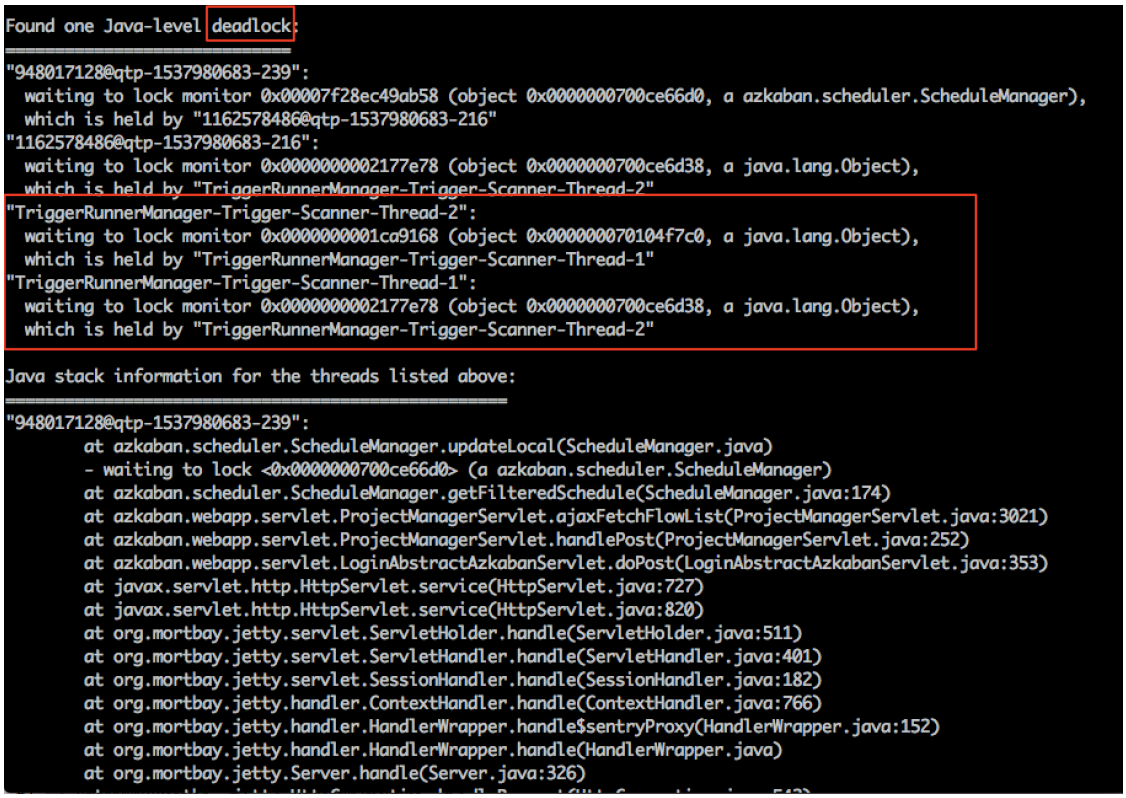
1.5 jstack 堆栈跟踪工具

用于生成虚拟机当前时刻的线程快照，可以分析定位线程出现长时间停顿的原因，以及进程假死、线程死锁等现象。

命令格式： jstack [option] vmid

option	作用
-F	当正常输入的请求不被响应时，强制输出线程堆栈
-l	除堆栈外，显示关于锁的附加信息
-m	如果调用本地方法的话，可以显示C/C++的堆栈

示例：Mammut4.7版本测试过程中，Azakban Webserver出现过一次线程死锁导致Webserver假死：



2. JDK的可视化工具

JDK提供了两种非常强大的可视化性能监视工具——JConsole和VisualVM，个人认为非常好用，在定位性能问题时可以与MAT结合进行分析。基本上它们集成了所有命令行工具的功能，且操作非常简单，这里就不细讲，有兴趣的同学自己可以去尝试一下。启动姿势分别是jdk/bin/jconsole和jdk/bin/jvisualvm。

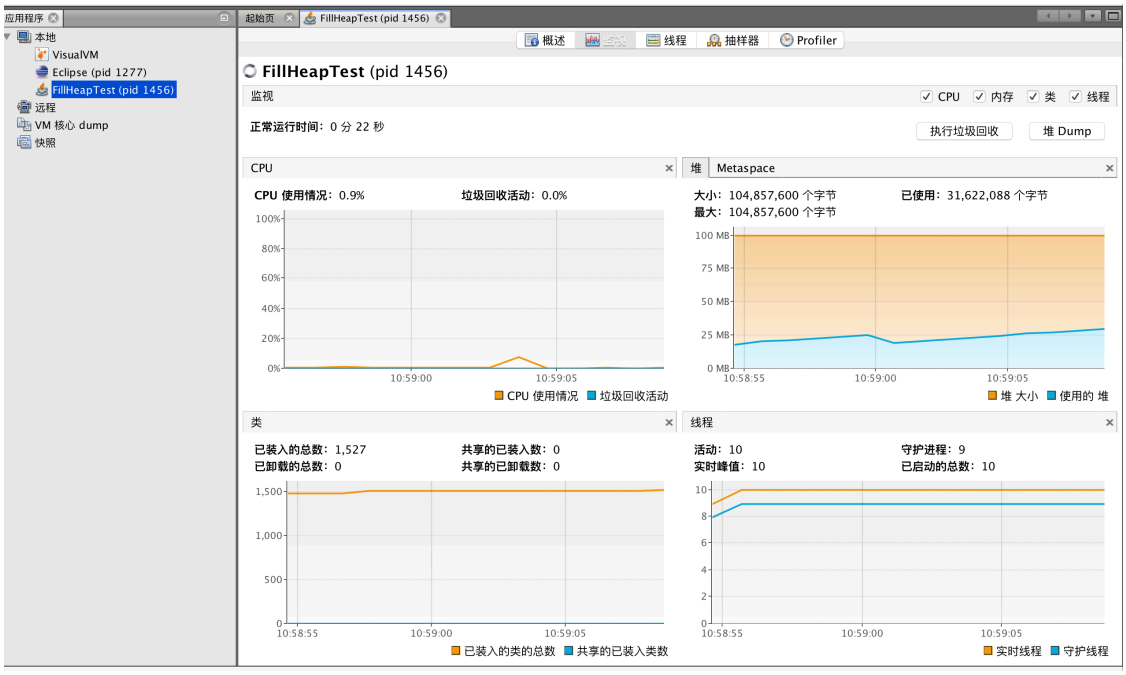
2.1 JConsole

这边是对我自己写的一个堆对象填充测试类FillHeapTest.java的Eden区监控截图：

2.2 VisualVM

比JConsole更强大，提供了故障处理和编写在线调试代码的功能（不需要重启服务就可以打印一些调试信息）。提供方法级的程序运行性能分析，找出被调用最多、运行时间最长的方法。

同样是对FillHeapTest.java的监控：


应用程序
本地
VisualVM
Eclipse (pid 1277)
FillHeapTest (pid 1456)
远程
VM 核心 dump
快照
起始页
FillHeapTest (pid 1456)
概述
线程
抽样器
Profiler
FillHeapTest (pid 1456)
监视
CPU
内存
类
线程
正常运行时间：0 分 22 秒
执行垃圾回收
堆 Dump
CPU
CPU 使用情况：0.9%
垃圾回收活动：0.0%
100%
80%
60%
40%
20%
0%
10:59:00
10:59:05
CPU 使用情况
垃圾回收活动
堆
Metaspace
大小：104,857,600 个字节
已使用：31,622,088 个字节
最大：104,857,600 个字节
100 MB
75 MB
50 MB
25 MB
0 MB
10:58:55
10:59:00
10:59:05
堆 大小
使用的 堆
类
已装入的总数：1,527
共享的已装入数：0
已卸载的总数：0
共享的已卸载数：0
1,500
1,000
500
0
10:58:55
10:59:00
10:59:05
已装入的类的总数
共享的已装入类数
线程
活动：10
守护进程：9
实时峰值：10
已启动的总数：10
10
8
6
4
2
0
10:58:55
10:59:00
10:59:05
实时线程
守护线程