JVM

**java内存区域及对象**

**一:几个计算机概念**

1. 计算机存储单位

从小到大 位Bit、字节Byte、千字节KB、兆MB、千兆G、TB 1byte = 8bit 其余相邻相差1024,即2的10次方

1. 计算机存储单元

寄存器：CPU的一部分,计算机中读写最快的存储元件,但是容量很少

内存: 独立的部件，是和CPU沟通的桥梁，存放了CPU的运算数据和外部存储器交换的数据，尽管速度很快，但是和寄存器还是有几个数量级的差别，因为寄存器在CPU之上

1. 内核空间和用户空间

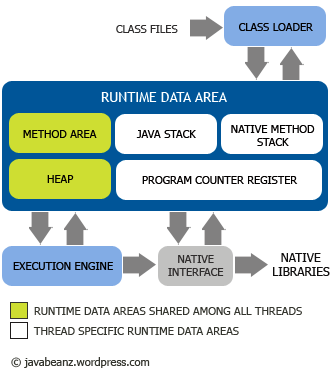
连接内存和寄存器的是地址总线，地址总线的宽度影响了物理地址的索引范围，因为总线宽度决定了处理器一次可以从寄存器或内存中获取多少个Bit，同时也决定了处理器最大可以寻址的地址空间。

内核空间主要是指操作系统运行时所使用的用于程序调度、虚拟内存的使用或者链接硬件资源的程序逻辑。区分内核空间和用户空间的目的主要是从系统的稳定性的角度考虑的。

1. 字长

CPU的主要技术之一，指的是CPU一次能处理的二进制数据位数bit，通常处理8位数据的是8位CPU，处理32位数据的是32位CPU，目前大多数是64位的，但是还是以32位字长运行

**二：运行时数据内存**



1. 线程独有的内存区域
2. 、program counter register(程序计数器)

这块内存区域很小，是当前线程锁执行字节码的行号指示器，字节码指示器通过改变计算器的值选取下一条需要执行的字节码，只对java方法起作用，如果执行的是native方法，这个寄存器是空的

1. 、java stack(虚拟机栈)

生命周期与线程相同。每个方法执行的时候会创建一个栈帧，用于存储局部变量、操作数栈、动态链接、方法出口等信息。每个方法从调用到执行完毕，都是对应一个栈帧在虚拟机从入栈到出栈的过程

1. 、native method stack(本地方法栈)

和虚拟机栈一样，只不过是为了java中的native方法服务的，虚拟机规范对这块区域不做强制要求，所以在HotSpot虚拟机中都没有这块区域，直接用虚拟机栈

1. 线程共享的内存区域
2. 、method area(方法区)

用于存储类信息，常量，静态变量 即时编译器编译后的代码等数据，在分代收集算法划分中，方法区≈永久代

1. 、heap(堆)

大多数应用，堆都是java虚拟机总内存中最大的一块，在虚拟机启动时创建，此内存的唯一目的就是存放对象实例，在分代收集算法中，堆可以细分成新生代和老年代，在细致划分包括Eden区、From Survivior区，To Survivior区

1. 、runtime constant pool(运行时常量池 1.7之后常量池被从方法区（永久代）（永久代在1.8后被移除用Metaspace元数据区）移到堆中)

方法区的一部分。用于存放编译期间生成的各种字面量和符号引用，这部分内容在类加载的时候进入方法区的运行时常量池中，另外翻译出来的直接引用也会存储在这个区域。这个区域还具备的特点就是动态性，并不要求只有编译期间的常量才会入运行时常量池，运行期间也可以往这个区域放入新的内容，String.intern()方法就是运用了这个特性

1. 直接内存

直接内存不是运行时内存的一部分，也不是java虚拟机规范定义的内存区域。但是这部分也被频繁使用，也可能会出现内存溢出问题。JDK 1.4新增了NIO，引入了一种基于通道和缓冲区的I/O方式，可以使用native函数库直接分配堆外内存，然后通过堆中的DirectByteBuffer对象作为这块内存的引用操作

**三：对象创建**

虚拟机创建对象步骤如下:

1. 虚拟机接收到一个new指令，会先检查这个指令参数能否在常量池中定位到一个类的符号引用，并检测这个符号引用代表的类是否已经被加载、解析和初始化，如果没有必须先完成类的初始化
2. 类加载完成后，虚拟机会为新生对象分配内存。对象所需内存在类加载之后便可以完全确定，但分配的方式有所区别
3. 、如果内存是规整的，采用的是指针碰撞法来给对象分配内存，其含义所有用过的内存在一边，空闲的内存在另外一边，中间放一个指针作为分界点的指示器，为对象分配内存即指示器往空闲区域移动和对象大小相等的距离。如果垃圾回收采用的是Serial、ParNew这种基于压缩算法的，虚拟机采用这种方式
4. 、如果内存是不规整的，已使用的内存和未使用的内存相互交错，虚拟机将采用空闲列表法为对象分配内存，即虚拟机维护了一份列表，记录了那块内存是可用的，在分配的时候会给对象实例分配一个足够大的内存。如果垃圾收集器选择的是CMS这种基于标记-清除算法的，虚拟机将采用这种分配方式
5. 内存分配结束之后，虚拟机会讲分配的内存空间初始化为零值（不包括对象头）。保证java代码在不需要给对象赋初始值就能直接访问，就能得到这些字段多代表的类型对应的零值
6. 对对象进行必要的设置，包含这个对象是哪个类的实例，如何找到类的元数据信息，对象的哈希值，对象的GC分代年龄等，这些信息存放在对象头中
7. 执行<init>方法，把对象按照程序员的意愿初始化，这样一个对象才会被真正的产生出来

**四：对象定位方式**

建立对象是为了使用对象，java程序使用栈上面的引用（reference）来操作堆上面的对象

Object obj = new Object();

new Object()之后包括两方面内容，一部分是类数据（比如类的Class对象）、一部分是实例数据

reference在虚拟机规范中只是一个指向new Object()的引用obj，并不知道如何定位，访问堆中的具体对象，访问方式也是由虚拟机决定的

1. 句柄访问。Java堆中划分一块句柄池，obj指向的是句柄的地址，句柄中包含类数据地址和实例数据地址
2. 指针访问。对象中存储类数据地址和实例数据地址，obj指向这个对象

HotSpot使用的是第二种方式

**内存泄漏和内存溢出、并行和并发、Minor GC和Full GC/Major GC、Client模式和Server模式的区别**

**一：内存溢出和内存泄漏**

1. 内存溢出：是指程序在申请内存的时候，没有足够大的空间可以分配了，在java程序中就会发生OOM（out of memory），可以在tomcat启动脚本内的JAVA\_OPS加上以下两个参数打印dump文件具体分析

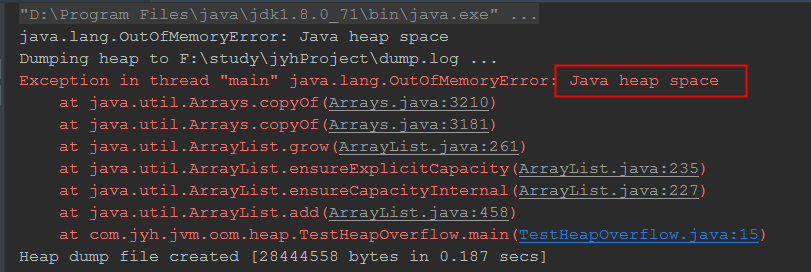
-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath=E:\dmpp\_dev\dmpp\dump.hprof

1. 堆溢出：保证对象不断创建并且对象不会被回收，到达堆限定的最大内存就会出现堆内存溢出。

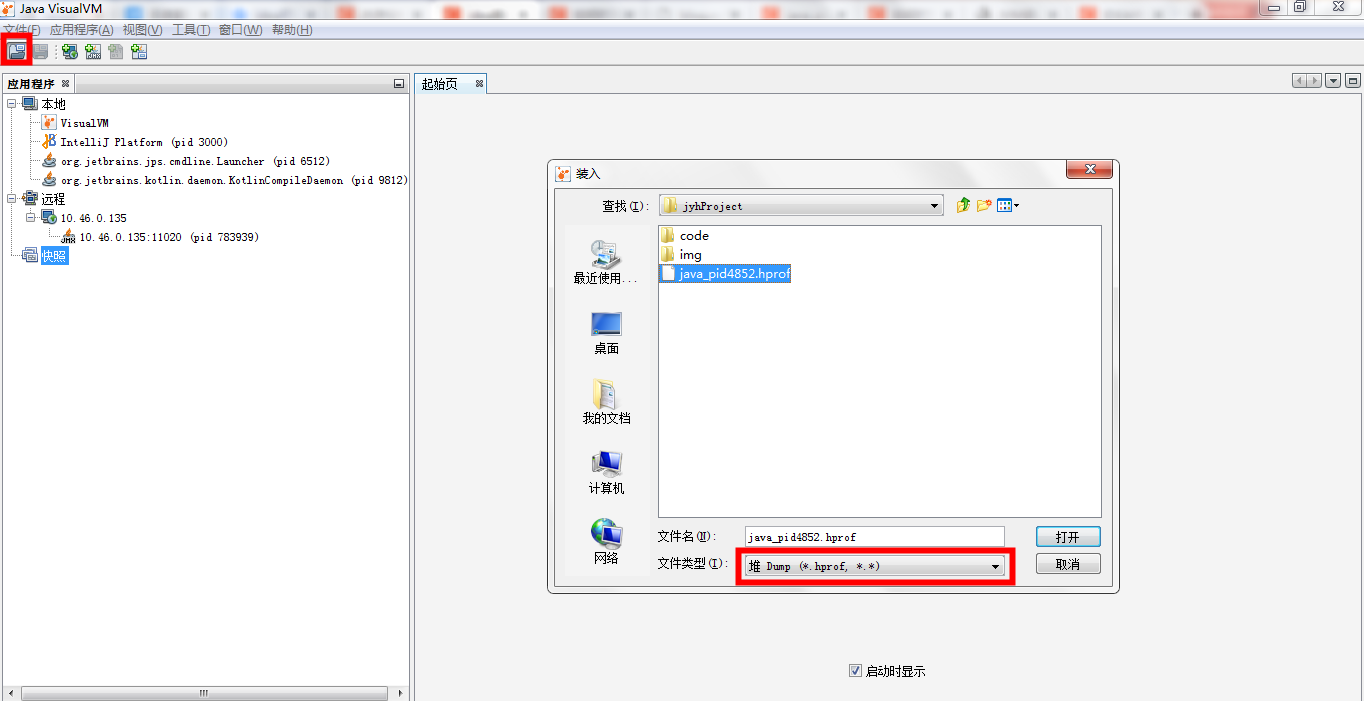
涉及JVM参数-Xmn20M（堆初始化大小） -Xmx20M（堆最大值大小）

测试案例：com.jyh.jvm.oom.heap.TestHeapOverflow。业务日志会直接显示是什么溢出

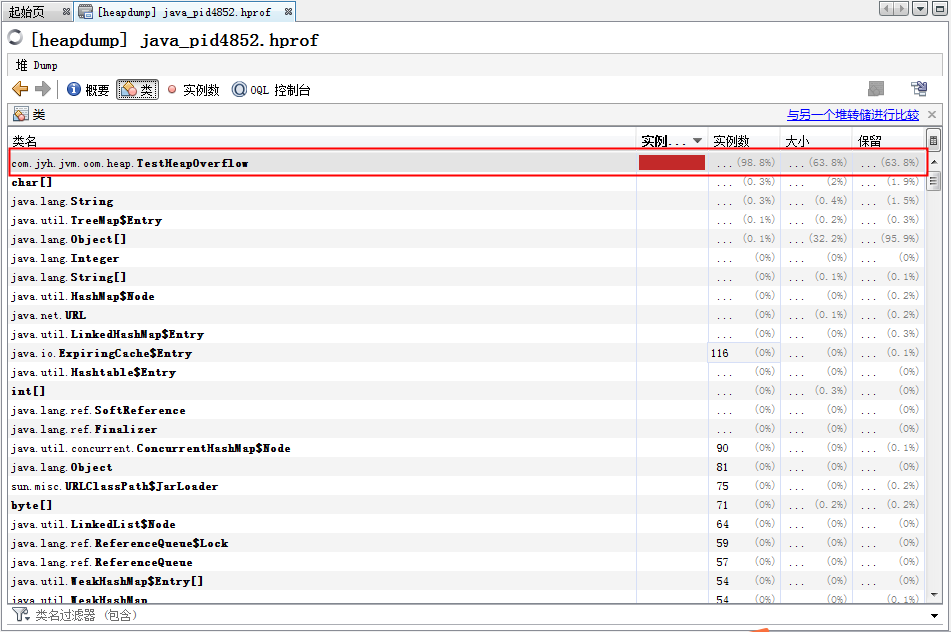


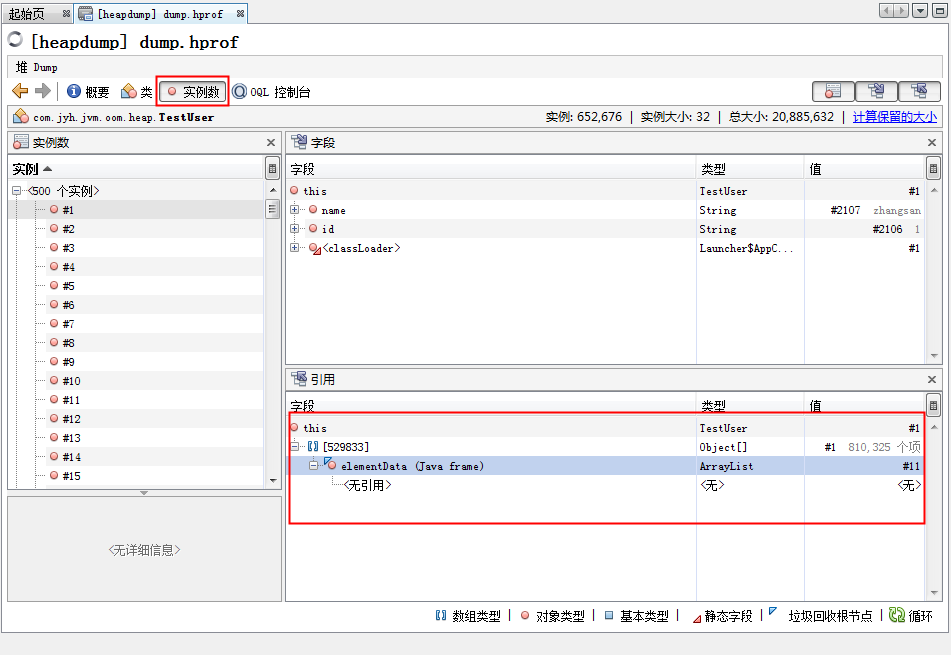
但是往往堆栈异常信息上反映不出来此时堆上面是被什么对象撑爆的，此时可以设置HeapDumpOnOutMemoryError参数后会打印类似于java\_pidxxxx.hprof的堆内存快照文件，可以使用以下三种方式打开快照：

1. Eclipse可以集成插件Memory Analyzer（MAT）
2. IDEA可以集成插件JProfiler
3. 使用jdk自带的jvisualvm,使用截图如下





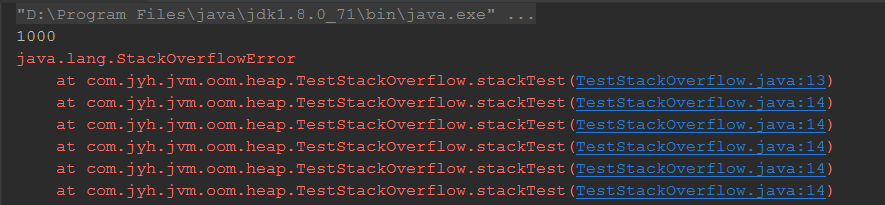




基本上可以通过上图看出是哪个大对象占用太多的内存一直没有释放导致堆溢出，而且可以通过最后一张图的引用看对象的树形结构，找出这个对象在哪里被引用的，这时候需要结合业务分析这个大对象产生的场景

1. 栈溢出：
2. 如果线程请求的栈深度过深（方法调用的深度过深）（一个线程的栈深度太深，导致该线程的栈超过了-Xss128k的配置）

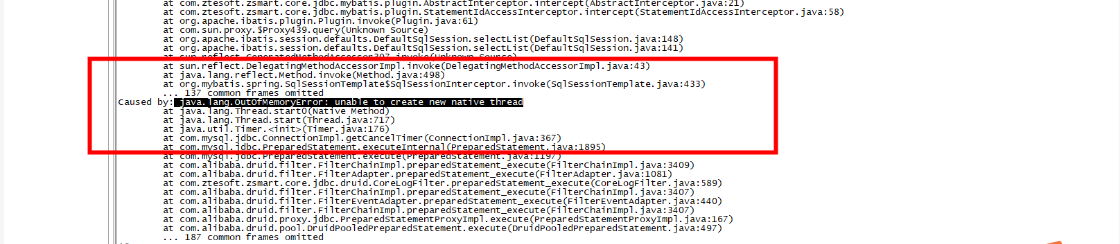
测试案例：com.jyh.jvm.oom.stack.TestStackOverflow



1. 如果不断的创建线程，每次创建一个线程都会创建一个栈空间，在服务器内存一定的情况下，并且堆方法区内存设定好之后

栈的总内存是固定的，线程数超过一定数量就会导致栈内存溢出，可以适当的减小栈空间的配置，能达到多创建线程的目的。

以下是生产环境发生的一次内存泄漏：不能创建线程



原因一般是两点：系统内存耗尽，无法为新线程分配内存；创建线程数超过了操作系统的限制

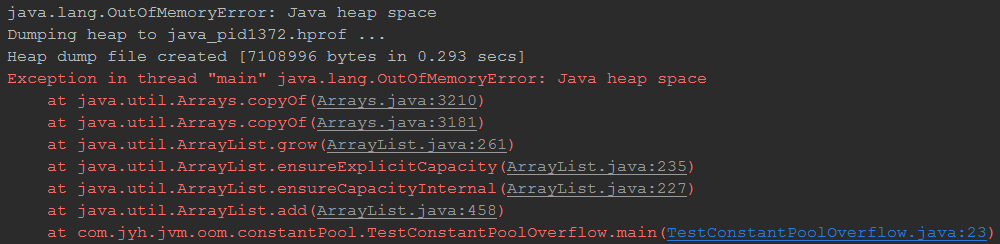
解决办法：ulimit -u查看某用户当前操作系统的最大允许进程数，cat /proc/sys/kernel/pid\_max 可以查看当前系统允许的最大线程数，在/etc/sysctl.conf下面kernel.pid\_max配置项可以配置操作系统允许的最大线程数，可以通过ls /proc/<pid>/fd | wc -l查看当前进程下的线程数，最终是通过改操作系统最大限制线程数解决的，但同时生产环境的-Xss是没有配置的，默认是1M,可以降低这个值，达到多创建线程数的目的

1. 方法区和运行常量池溢出（-XX:PermSize=10M -XX:MaxPermSize=10M）

可以使用String.intern()这个方法往常量池不断加常量，最终会导致java.lang.OutOfMemoryError: PermGen space

由于本地安装的是jdk1.8，常量池从1.7后就移到堆中了，永久代在1.8后也被Metaspace代替了，所以这里即使循环不断往常量池加常量，也不会出现PermGen space 。这里将使用一个案例证明常量池在1.7中被移到了堆中

测试案例：com.jyh.jvm.oom.constantPool.TestConstantPoolOverflow该案例不断往常量池加常量，最终导致堆溢出



1. 内存泄漏：指程序在申请到一块内存后，一直无法释放掉这块内存，即分配的对象可达但无用。内存泄漏一般都是内存中有一个大对象一直释放不掉，最终会存放在老年区，在java中可以通过配置以下参数打印GC日志，监控FGC下对象的前后内存

-XX:+DisableExplicitGC(可以使程序中的System.gc()方法失效，但是对于使用NIO直接利用直接内存来说，慎用，这块内存不会自动垃圾回收)

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

-XX:+PrintClassHistogramBeforeFullGC

-XX:+PrintClassHistogramAfterFullGC

-Xloggc:E:\dmpp\_dev\dmpp\gc.log

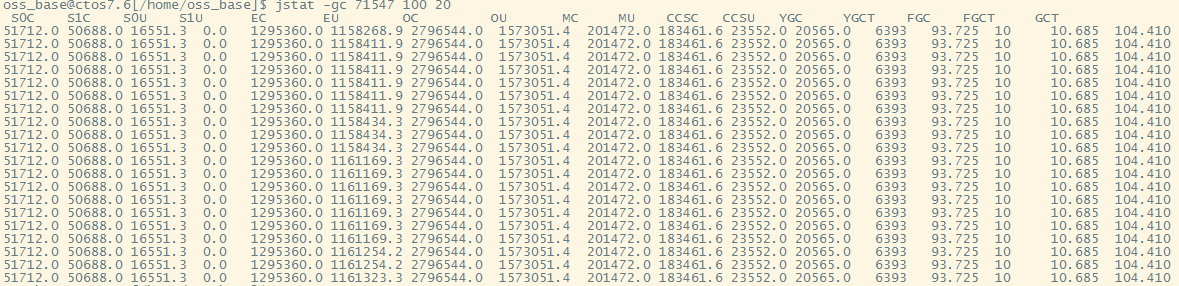
**二：并行和并发**

1. 并行（Parallel）：多个处理器同时处理多个任务，这是物理上同时发生的。在垃圾收集器上下文中，这里指的是多条垃圾收集线程同时运行，但用户线程此时仍然处于等待状态
2. 并发（Concurrent）：一个处理器同时处理多个任务，这是逻辑上同时发生的。在垃圾收集器上下文中，垃圾收集器线程和用户线程同时运行（不一定是并行的，可能是交替运行），用户程序继续运行，垃圾收集程序在另外一个cpu

**三：Minor GC和Major GC**

1. Minor GC（YGC）：发生在新生代堆中。频率比较高，触发条件是Eden区空间不足，触发时清空Eden和Survivor区中未存活的所有对象，将这部分区的存活的对象copy到老年代区
2. Major GC（Full GC/FGC）：old空间不足，perm区空间不足，调用System.gc(),dump live的内存信息（jmap -dump:live）,都会发生FGC,触发时清空所有整个堆中所有未存活的对象和卸载的class信息

**指令：jstat -gc <pid>**



S0C：新生代中第一个Survivor(幸存区)的容量（单位字节）

S1C：新生代中第二个Survivor(幸存区)的容量

S0U：新生代中第一个Survivor(幸存区)使用的容量

S1U：新生代中第二个Survivor(幸存区)使用的容量

EC：新生代中Eden（伊甸园区）容量

EU：新生代中Eden（伊甸园区）使用过的容量

OC：老年代中Old区的容量

OU：老年代中Old区使用过的容量

MC：元数据空间容量（jdk1.8后废除了持久代，使用了Metaspace来代替，1.7以前是PC，在HotSpot中，method area等同于持久代）

MU：元数据空间使用过的容量

CCSC：压缩类型空间容量（属于元数据区）

CCSU：压缩类型空间使用容量

YGC: 从应用程序开始到采样的时候年轻代区发生GC的次数

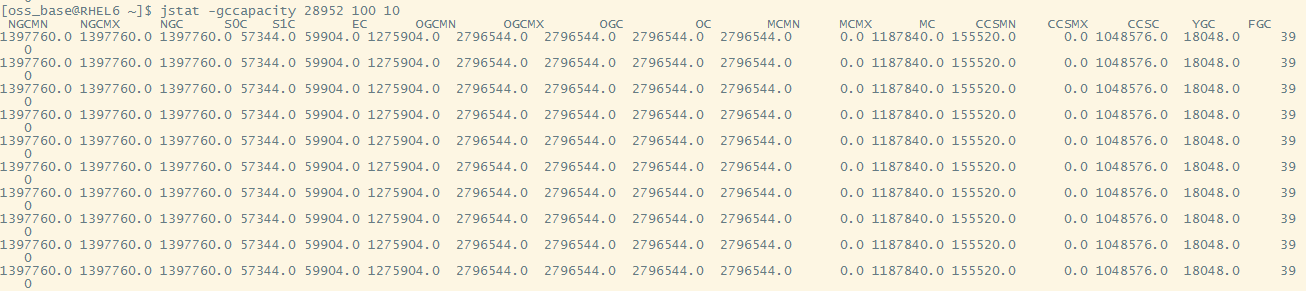
YGCT：从应用程序开始到采样的时候年轻代区发生GC的总时间(s)

FGC：从应用程序开始到采样的时候整个堆发生GC的次数

FGCT：从应用程序开始到采样的时候整个堆发生GC的总时间(s)

GCT：YGC和FGC发生的时间之和

**指令：jstat -gccapacity <pid>**



NGCMN：年轻代初始化（最小）容量（字节）

NGCMX：年轻代最大容量

NGC：年轻代当前容量

OGCMN：老年代初始化（最小）容量

OGCMX：老年代最大容量

OGC：老年代当前容量（和OC差不多）

MCMN：元数据区初始化（最小）容量（1.8之前是PGCMN）

MCMX：元数据区最大容量（1.8之前是PGCMX）

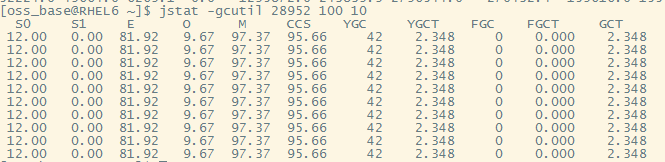
MC：元数据区当前容量（1.8之前是PGC）

CCSMN：压缩类型区初始化（最小）容量

CCSMX：压缩类型区最大容量

CCSC：压缩类型区当前容量

**指令：jstat -gcutil <pid>**



S0：年轻代中第一个survivor区使用率

S1：年轻代中第二个survivor区使用率

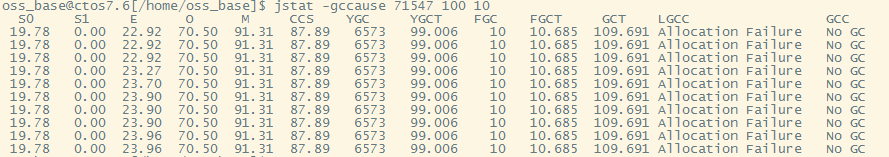
E：年轻代中Eden区使用率

O：老年代使用率

M：元数据区使用率

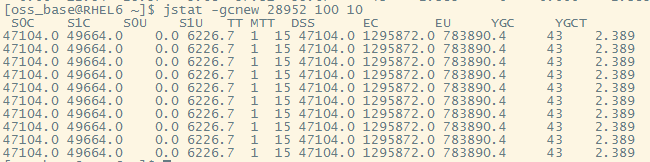
CCS：压缩类型使用率（属于元数据区）

**指令：jstat -gccause <pid>**



LGCC：描述YGC和FGC发生的原因

**指令：jstat -gcnew <pid>**

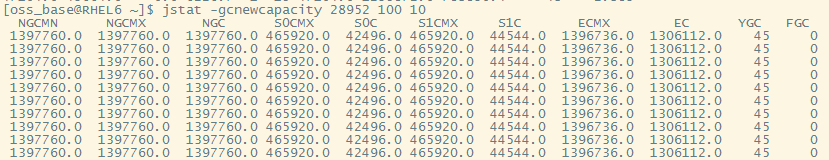


TT：持有次数限制

MTT：最大持有次数限制

DSS：当前需要Survivor的容量（Eden已满）

**指令：jstat -gcnewcapacity <pid>**

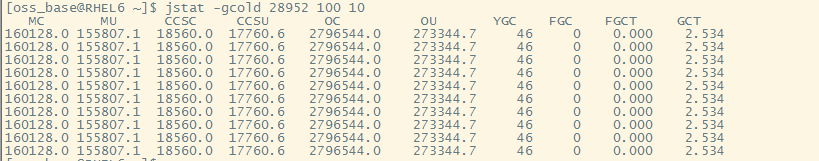


S0CMX：第一个Survivor区的最大容量

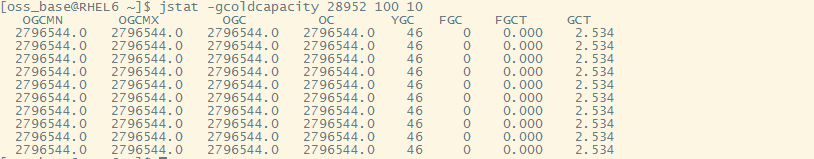
S1CMX：第二个Survivor区的最大容量

ECMX：Eden区最大容量

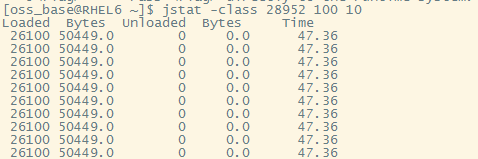
**指令：jstat -gcold <pid>**



**指令：jstat -gcoldcapacity <pid>**



**指令：jstat -class <pid>**



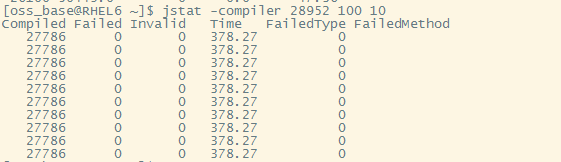
Loaded：装载类数量

Bytes：装载类大小

Unloaded：卸载类 数量

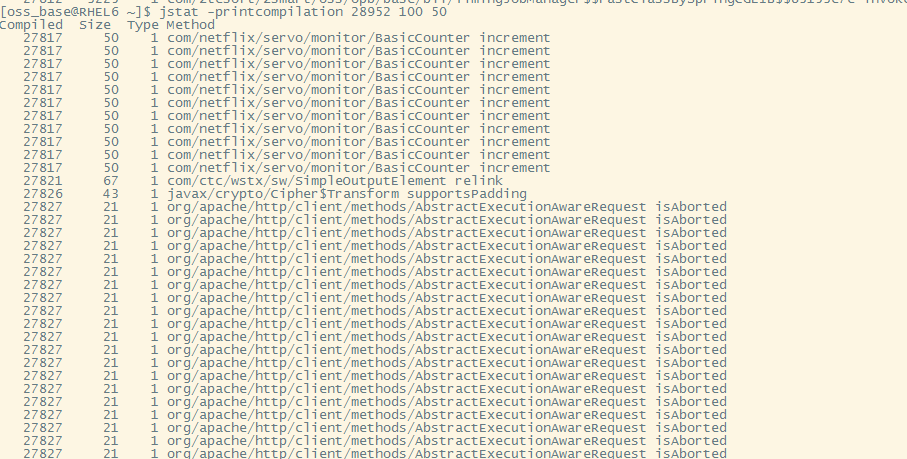
Time：耗费总时间

**指令：jstat -compiler <pid>**



Compiled：JIT编译过的方法数

**指令：jstat -printcompilation <pid>** 输出被JIT编译过的方法



**四：Client模式和Server模式的区别**

部分商用虚拟机中，Java程序最初是通过解释器对.class文件进行解释执行的，当虚拟机发现某个方法或代码块运行地特别频繁的时候，就会把这些代码认定为热点代码Hot Spot Code（这也是我们使用的虚拟机HotSpot名称的由来）。为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器叫做即时编译器（Just In Time Compiler，即JIT编译器）。JIT编译器并不是虚拟机必需的部分，Java虚拟机规范并没有要求要有JIT编译器的存在，更没有限定或指导JIT编译器应该如何去实现。但是，JIT编译器性能的好坏、代码优化程度的高低却是衡量一款商用虚拟机优秀与否的最关键指标之一。

解释器和编译器各有优势：

1. 当程序需要迅速启动和执行的时候，解释器可以省去编译的时间，立即执行
2. 在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率

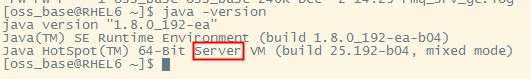
我们使用的HotSpot中内置了两个JIT编译器，即C1编译器和C2编译器，默认采用的是解释器和一个编辑器配合的方式进行工作。HotSpot在启动的时候会根据自身版本以及宿主机器的硬件性能自动选择运行模式，比如会检测宿主机器是否为服务器、比如J2SE会检测主机是否有至少2个CPU和至少2GB的内存。

1、如果是，则虚拟机会以Server模式运行，该模式与C2编译器共同运行，更注重编译的质量，启动速度慢，但是运行效率高，适合用在服务器环境下，针对生产环境进行了优化

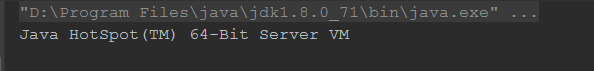
2、如果不是，则虚拟机会以Client模式运行，该模式与C1编译器共同运行，更注重编译的速度，启动速度快，更适合用在客户端的版本下，针对GUI进行了优化

可以使用以下方式查看jvm是server模式还是client模式

1. 使用java -version查看本地或者服务器上java虚拟机

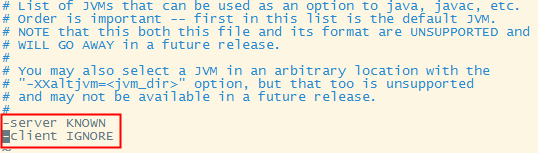


1. 查看本地安装的idea自带的虚拟机版本

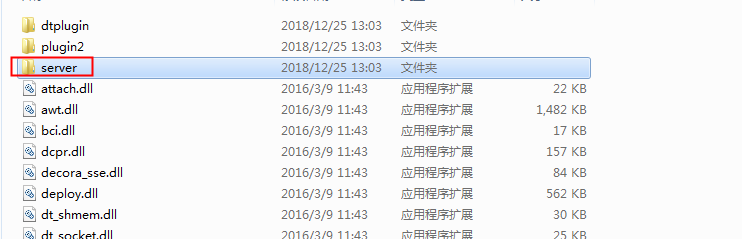
System.out.println(System.getProperty("java.vm.name"));

可以通过修改修改相关配置文件修改虚拟机运行模式

1. 64位jdk：JAVA\_HOME/jre/lib/amd64/jvm.cfg



64位jre只支持Server模式，因为在JAVA\_HOME/jre/bin/目录下只有server目录，没有client目录



1. 32位jdk：JAVA\_HOME/jre/lib/i386/jvm.cfg
2. IDEA：D:\Program Files\IntelliJ IDEA 2018.3\jre64\lib\amd64\jvm.cfg

说明：支持server模式和client模式的jvm.cfg配置文件中-Server和-Client的值都是KNOWN,更换模式修改顺序即可，同时要保证在JAVA\_HOME/jre/bin下有server和client文件夹才行

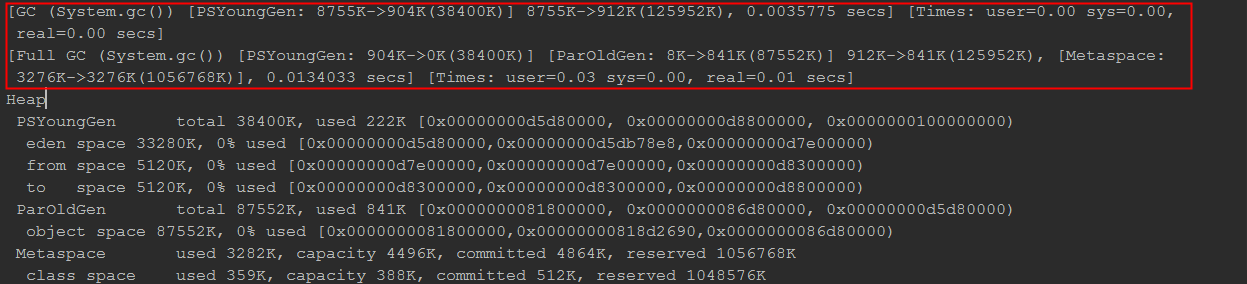
**Java垃圾回收（GC）机制详解**

**一：如何查找需要回收的对象**

要回收的垃圾是不可能被任何途径使用的对象，一般有以下方法查找未使用的对象

1. 引用计数法：给对象添加一个引用计数器，没当一个地方引用了一次+1，当引用失效时引用-1，当计数器为0时，此时这个对象将被回收，不过虚拟机不是用这种方式查找回收对象的，解决不了回收相互引用的对象

测试案例：com.jyh.jvm.gc.ReferenceCountingGC 相互引用的对象在调用System.gc()时释放了对象



1. 可达性分析法：这个算法的基本思想是通过一系列被称为“GC Roots”对象作为起始点，从这些节点往下寻找，搜索过的路径称为引用链，当GC Roots到对象没有任何引用链，则称GC Roots到对象不可达，此时证明对象不可用，将被回收。以下四种可以作为GC Roots
2. 、虚拟机栈中引用的对象
3. 、方法区中静态属性引用的对象
4. 、方法区（1.7之前，1.7后常量在堆）中常量引用的对象
5. 、 本地方法栈JNI（native方法）引用的对象

**二：四种引用状态**

1. 强引用

类似于Object obj=new Object()这样子的代码是一种强引用，如果该引用存在，垃圾回收器永远不会回收该对象

1. 软引用

描述有些还有用但是非必须的对象。在系统将要发生内存溢出之前，垃圾回收器会将这些对象进行二次回收。如果这次还没有足够的内存才会报内存溢出。Java中用类SoftReference表示软引用

1. 弱引用

描述非必须的对象。这部分对象总是会被垃圾回收器回收，不管内存是否充足。Java中用类WeakReference表示

1. 虚引用

唯一的作用是在被垃圾回收器回收的时候，返回一个被虚引用关联的对象通知。和该对象什么时候被回收的毫无关系。Java中用类PhantomReference表示

**三：方法区回收**

方法区主要回收废弃的常量和无用的类

常量回收比较简单，只要判断该常量没有被引用就可以了

类如果要被回收，需要满足以下三个条件：

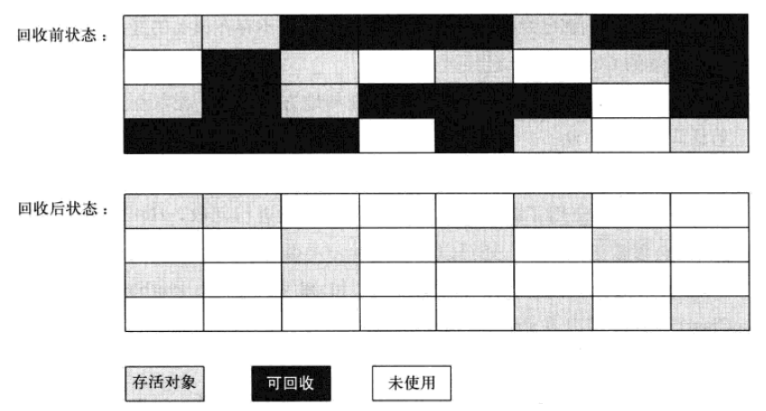
1. 该类的所有实例都被回收，即堆中没有该类的任何实例
2. 该类的加载器ClassLoader已经被回收
3. 该类对应的java.lang.Class对象没有被任何一处地方引用，没有任何地方通过反射获取该类

**四：垃圾回收算法**

垃圾回收思想：

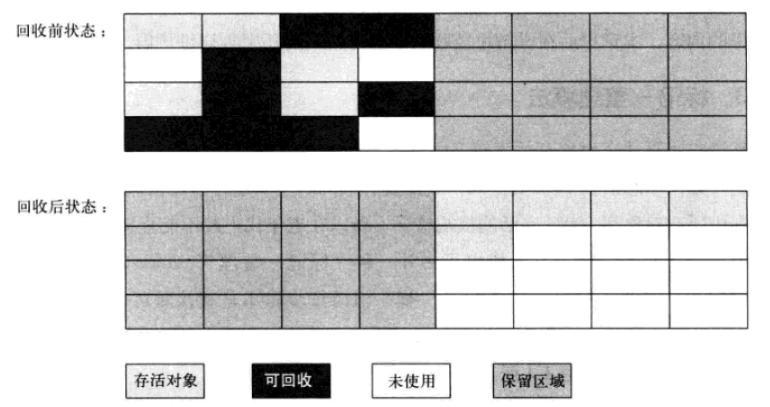
1. 标记-清除（Mark-Sweep）算法

分为两个阶段，先标记所有需要回收的对象，标记完成后统一收回所有的对象。该算法标记和清除效率都不高，而且回收后容易导致大量的不连续的内存碎片，当创建一个较大的对象时，将触发GC



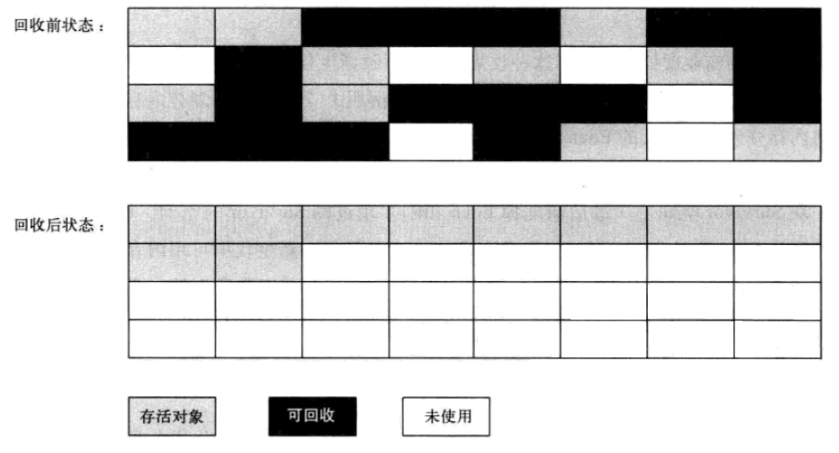
1. 复制（Copying）算法

复制算法为了解决效率出现的，将可用内存分为两半，每次只用其中一块，当另一块内存用完了，将可用对象全部复制到空闲的那块，将用完的内存全部清理掉。每次只要清理一半的内存，而且不用考虑内存碎片过多的问题,现在的商用虚拟机都采用这种算法回收新生代，但是1:1的比例代价太高，Hotspot将新生代划分成一个Eden区和两个Survivor。每次回收将Eden区和一个Survivor区中存活的对象复制到另外一个survivor区中，Hotspot默认Eden:Survivor = 8:1，每次新生代中可用的内存占整个新生代的90%



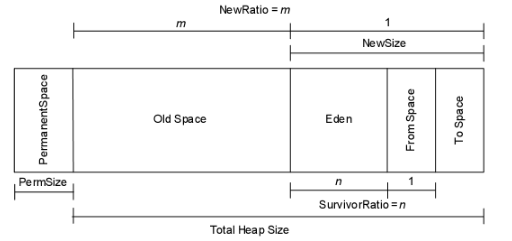
1. 标记整理（Mark-Compact）算法

对于对象存活率较高的内存空间，采用复制算法效率比较低，例如老年代。标记整理算法与标记清除算法差不多，只不过将所有存活的对象移到一端，然后直接清理掉边界以外的所有内存

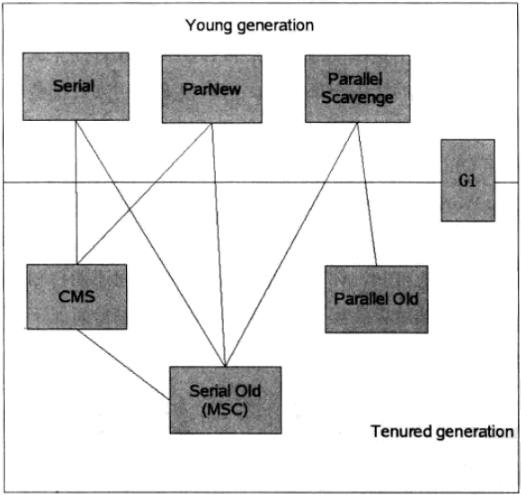


1. 分代收集算法

现在的商用虚拟机基本上都采用这种分代收集算法，一般新生代采用的是复制算法，老年代这种对象存活率比较高的空间采用标记清除算法或者标记整理算法



垃圾收集器



上图是Hotspot虚拟机包含的所有垃圾收集器，Young generation是作用在新生代的垃圾收集器，Tenured generation是作用在老年代的垃圾收集器，通过线条连接在一起的说明两者之间可以配合使用

1. Serial收集器

最基本，发展历史最悠久的垃圾收集器，是一个采用复制算法的单线程收集器，它只会使用一个CPU或者一个线程回收垃圾，而且其进行垃圾回收的时候会暂停其他所有线程（用户线程会全部暂停），Client模式下新生代默认此种垃圾回收器。对于桌面应用时，分配给虚拟机的一般内存不大，一次回收几十兆或者上百兆的内存停顿在几十ms或者几百ms内，只要不是太频繁，完全可以接受

1. ParNew收集器

是Serial多线程版本，可以允许使用多条线程完成垃圾回收工作，使用的也是复制算法，一般是server模式下首选的新生代垃圾回收器，因为他是除了Serial收集器之外唯一可以CMS搭配的垃圾收集器，CMS是第一次让垃圾收集器线程和用户线程同时运行的一种垃圾收集器。ParNew默认开启的线程数与CPU数量相同，如果CPU数量过多，可以使用-XX:ParallelGCThreads参数来限制垃圾收集器的线程数

1. Parallel收集器（吞吐量优先收集器）

是一种新生代垃圾收集器，使用复制算法，多线程并行处理；不同于其他类似于CMS等收集器是为了减少用户线程的停顿时间，Parallel的目标是达到一个可控制的吞吐量（CPU执行用户线程时间和CPU总消耗的比值，即 吞吐量 = 用户代码执行时间/用户代码执行时间+垃圾收集时间），Parallel是Server模式下默认垃圾收集器

1. 、停顿时间短适用于有非常多的用户交互的程序，高吞吐量是提高CPU的效率，适合用于不怎么交互的后台任务
2. 、-XX:MaxGCPauseMillis：设置停顿时间长短，也不是越小越好，GC停顿时间缩短是牺牲吞吐量和新生代内存大小为代价的
3. 、-XX:GCTimeRatio：设置吞吐量大小
4. 、-XX:+UseAdaptiveSizePolicy：开关参数，打开之后，不需要指定新生代Eden、Survivor参数等具体细节，虚拟机会根据当前的系统自动调节，以达到最合适的停顿时间或最大吞吐量
5. Serial Old收集器

Serial的老年代版本，同样是一个单线程收集器，采用标记—整理算法，主要意义是给Client模式下虚拟机使用

1. Parallel Old收集器

Parallel的老年代版本，多线程收集器，采用标记—整理算法，在注重吞吐量和CPU敏感资源的场合下可以使用Parallel + Parallel Old组合

1. CMS收集器

CMS是一种以获得最短停顿时间为目标的垃圾收集器，垃圾收集器线程与用户线程同时运行，一般应用于互联网站或者B/S系统的服务端上，这类应用注重服务的响应速度，希望停顿时间短，给用户带来更好的体验，采用标记—整理算法

1. G1收集器

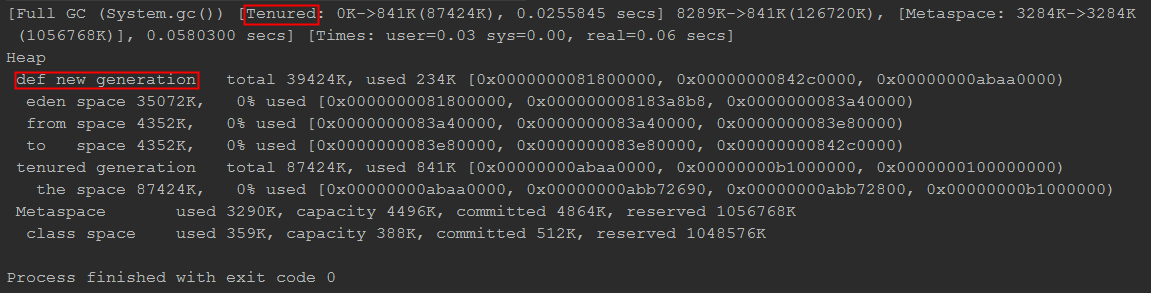
G1（Garbage-First）收集器是当今收集器技术发展的最前沿成果之一，JDK 7 Update 4后开始进入商用。在G1收集器之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1收集器不再是这样，使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region的集合。G1收集器跟踪各个Region里面的垃圾堆积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也是Garbage-First名称的由来）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

垃圾收集器组合

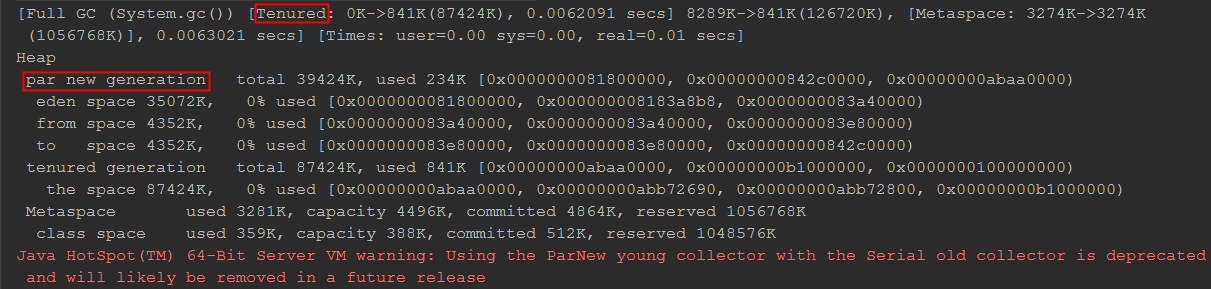
|  |  |  |  |
| --- | --- | --- | --- |
| GC组合 | Minor GC | Full GC | 描述 |
| -XX:+UseSerialGC | Serial收集器串行回收 | Serial Old收集器串行回收 | 该选项可以手动指定Serial + Serial Old组合执行内存回收 |
| -XX:+UseParNewGC | ParNew收集器并行回收 | Serial Old收集器串行回收 | 该选项可以手动指定ParNew+Serial Old组合执行内存回收 |
| -XX:+UseParallelGC 或者  -XX:+UseParallelOldGC | Parallel收集器并行回收 | Parallel Old收集器串行回收 | 该选项可以手动指定Parallel+Parallel Old组合执行内存回收 |
| -XX:+UseParallelGC 和  -XX:-UseParallelOldGC | Parallel收集器并行回收 | Serial Old收集器并行回收 | 该选项可以手动指定Parallel+Serial Old组合执行内存回收 |
| -XX:+UseConcMarkSweepGC | ParNew收集器并行回收 | CMS收集器回收  备用Serial Old收集器串行回收 | 该选项指定ParNew+CMS+Serial Old组合回收，优先使用ParNew + CMS,出现ConcurrentMode Fail或者Promotion Fail时，将采用ParNew+Serial Old |
| -XX:+UseConcMarkSweepGC和  -XX:-UseParNewGC | Serial收集器串行回收 |
| -XX:+UseG1GC | G1收集器并发、并发执行内存回收 | | 暂无 |

GC日志（使用了jdk1.8的版本，不同版本的jvm垃圾收集器可能不太一样）

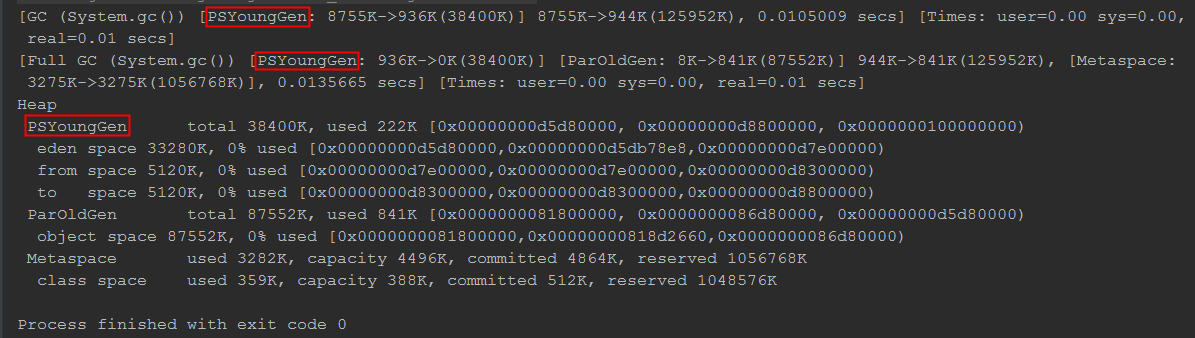
1. 使用-XX:+PrintGCDetails 和 -XX:+UseSerialGC（Serial + Serial Old）



1. 使用-XX:+PrintGCDetails 和 -XX:+UseParNewGC（ParNew + Serial Old）

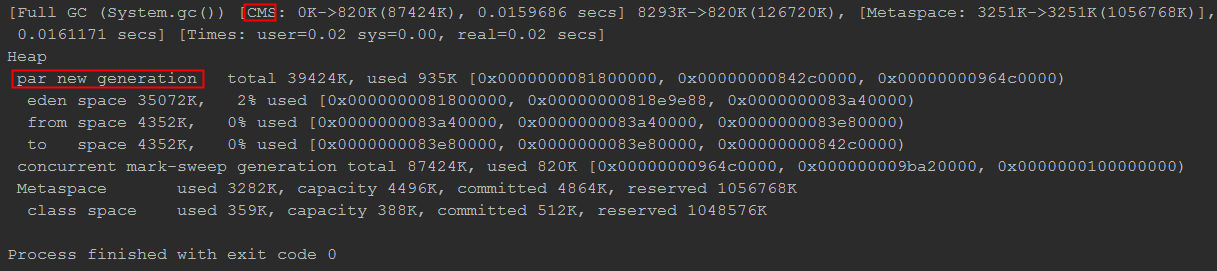


1. 使用-XX:+PrintGCDetails 和 -XX:+UseParallelGC（Parallel + Parallel Old）或者-XX:+UseParallelOldGC（Parallel + Parallel Old）

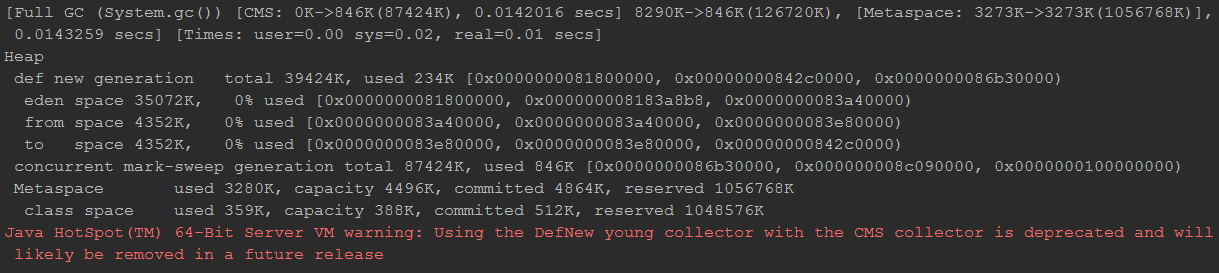


4、使用-XX:+PrintGCDetails 和 -XX:+UseParallelGC 和 -XX:-UseParallelOldGC

5、使用-XX:+PrintGCDetails 和 -XX:+UseConcMarkSweepGC（ParNew + CMS）



6、使用-XX:+PrintGCDetails 和 -XX:+UseConcMarkSweepGC 和 -XX:-UseParNewGC（Serial + CMS）



根据以上截图总结下共性：

1. 上图中GC和Full GC代表是停顿类型。如果是Full GC，说明本次GC停止了其他所有的工作线程。其中Full GC（System.gc()）代表是方法调用了System.gc()方法触发了Full GC
2. 后面的Tenured、PSYoungGen、CMS代表的是不同的老年代垃圾收集器，依次是Serial Old、Parallel Old、CMS。在Heap下面一行的开头可以看出新生代垃圾收集器，它们分别是def new generation（Serial）、par new generation（ParNew）、PSYoungGen（Parallel）
3. 其中ParNew + Serial Old和Serial + CMS组合有告警，在以后的jvm版本可能会被移除（当前使用的是1.8）
4. 以第一个为例：
5. 、[Tenured: 0K->841K(87424K), 0.0254556 secs] 7591K->841K(126720K)【该内存GC前使用的空间->经过GC后使用的空间（该内存总空间），这次GC花费的时长】GC前java堆使用内存->GC后java堆使用内存（java堆总内存），后面的是
6. 、 [Metaspace: 3275K->3275K(1056768K)], 0.0479850 secs] 元数据空间GC前后内存使用情况
7. 、[Times: user=0.00 sys=0.00, real=0.05 secs] 则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的钟墙时间。后面两个的区别是，钟墙时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间所以如果user或sys超过real是完全正常的
8. Heap具体介绍各个年代区域的内存情况

触发GC的时机

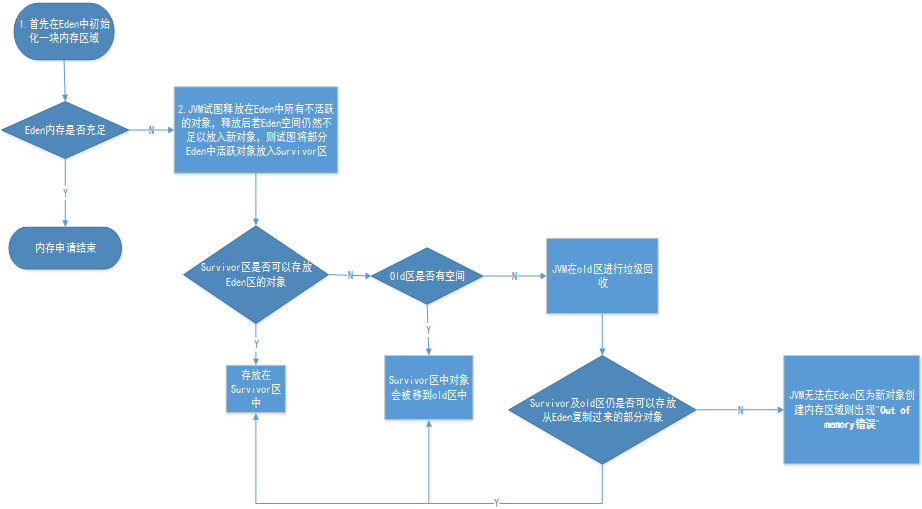
1. 当年轻代或者老年代满了，jvm无法为对象分配内存的时候，就会触发一次GC去回收掉那些已经不在使用的对象了
2. 手动调用一次System.gc()
3. 程序运行的时候有一条低优先级的GC线程，它是一条守护线程，当这条线程处于运行状态的时候，自然就触发了一次GC。

**内存分配原则**

**TLAB**

内存分配的动作，可以按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。哪个线程需要分配内存，就在哪个线程的TLAB上分配。虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。这么做的目的之一，也是为了并发创建一个对象时，保证创建对象的线程安全性。TLAB比较小，直接在TLAB上分配内存的方式称为快速分配方式，而TLAB大小不够，导致内存被分配在Eden区的内存分配方式称为慢速分配方式。

**JVM内存申请过程**



**对象优先分配在Eden 区上**

使用案例：com.jyh.jvm.gc.EdenAllocationTest

jvm配置参数如下：

-verbose:gc（打印GC详细日志）

-XX:+PrintGCDetails（在控制台打印GC详细日志）

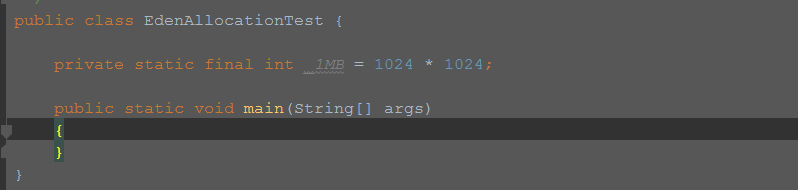
-Xms20M（堆内存初始化最小值）

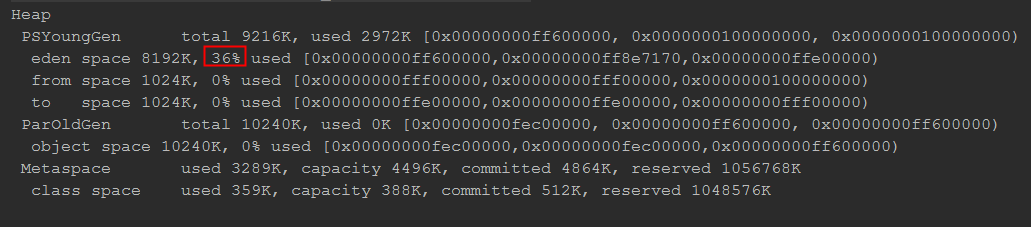
-Xmx20M（堆内存最大值）

-Xmn10M（年轻代大小）

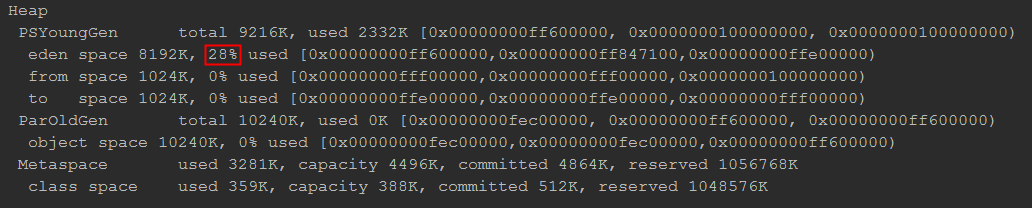
-XX:SurvivorRatio=8（Eden : survivor1 : survivor2 = 8 : 1 : 1）

1. 不分配任何对象



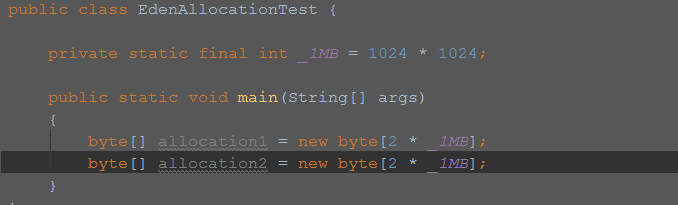


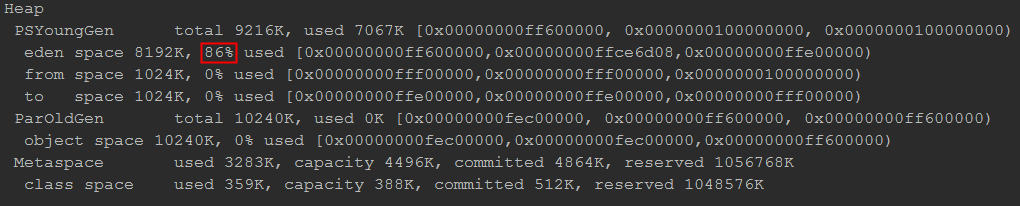
在没有显示分配对象的情况下，TLAB和EdenAllocationTest这个对象占据了eden区36%的内存了，其中TLAB占据多少内存可以加上-XX:-UseTLAB（默认是使用的）来对比一下测试出来



前后对比可以看出该线程的TLAB占据了Eden区的8%（8M \* 8% = 0.64M）

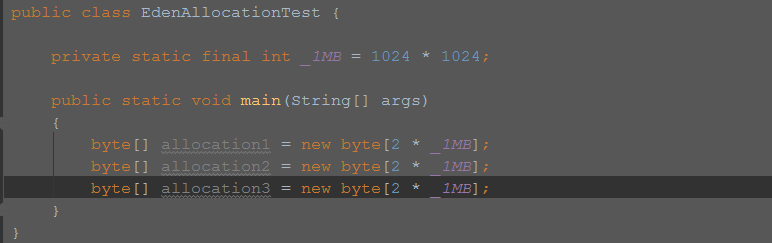
1. 先分配两个2M对象

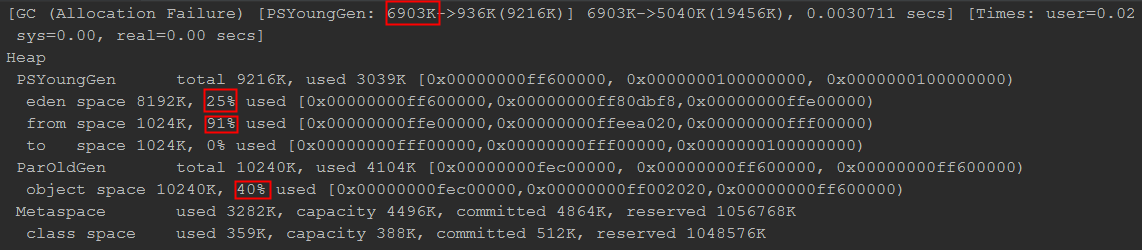




此时分配了两个2M对象，一共4M占据Eden区50%（Eden一共8M），加上一开始的36%，总共86%

1. 在分配一个2M对象

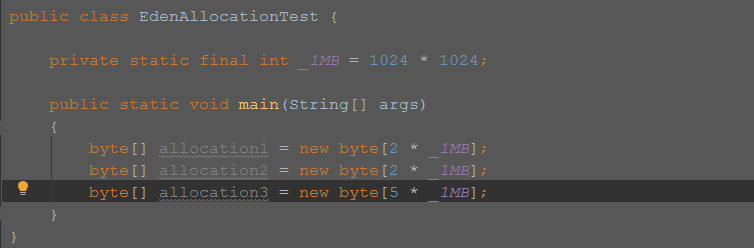


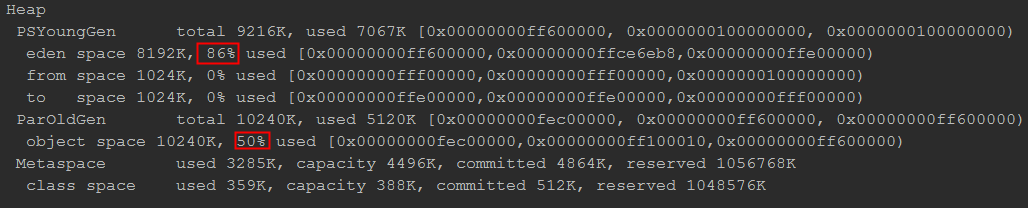


该日志显示分配失败了一次，发生了一次Minor GC（YGC）。在发生YGC之前年轻代已经使用了6903k（4096k(4M) + 2807k）包括分配的4M对象和TLAB和EdenAllocationTest这个对象所占的2807k，此时在分配一个2M的对象过来，年轻代一共可以分配是9M(Eden 8M(已用6.88M) + survivor 1M(已用0M)),该对象既进不了Eden区，也进不了Survivor区，而且前两个2M的对象引用还一直存在不会被回收，此时会发生一次YGC将Eden区中未回收的4M的对象复制到老年代，所以老年代使用率变成了40%，之后在重新调整年轻代，新进来的2M对象直接进入Eden区，回收后还剩余的不足1M的对象放入了From Survivor区。

**大对象直接分配到老年代中**

把上面的第三个步骤的修改成在分配一个5M的对象





上图中可以看出大对象直接进入了老年代，没有发生YGC，一般当一个对象>=Eden区的一半的时候将被视为大对象

同时可以设置参数-XX:PretenureSizeThreshold = 3145728（1024\*1024\*3 = 3M），当一个对象大于这个数值也被认为是大对象。这样做的目的是避免在eden和两个survivor区发生大量的内存复制，影响性能

**其他几条原则**

1. 长期存活的对象将进入老年代。Eden区中的对象在进行一次YGC没有被回收之后，对象年龄加1，当到达设置的-XX:MaxTenuringThreshold值的时候，将进入老年代
2. Survivor空间中相同年龄的对象大小总和大于Survivor空间的一半，年龄大于或者等于该年龄的对象就可以直接进入老年代，无需关注-XX:MaxTenuringThreshold设置的值

**Metaspace**

**历史背景**

Jvm内存分布中有一块method area（可以等同于持久代），主要包括

1. 、虚拟机加载的类信息
2. 、常量池
3. 、静态变量
4. 、即时编译后的代码

Java7之后常量池从方法区移到了堆中

Java8持久代被永久移除，取而代之的是Metaspace

**为什么移除永久代**

外因：Hotspot是为了融合JRockit而努力，Rockit用户没有永久代

内因：永久代的大小受jvm内存大小的限制-XX:PermSize和-XX:MaxPermSize大小限制，这两个参数又受jvm设定的大小限制，比起Matespace使用的内存不在虚拟机中，使用的本地内存，可能会导致永久代更容易出现溢出问题

**Metaspace**

Metaspace并不在虚拟机内存中而是使用本地内存，理论上取决于操作系统的内存大小

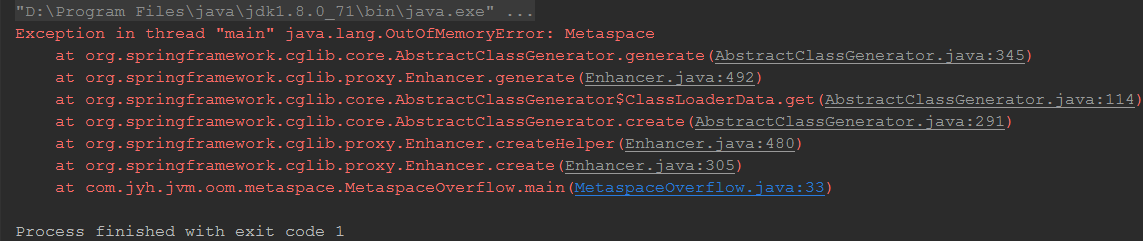
测试案例：com.jyh.jvm.oom.metaspace.MetaspaceOverflow（通过cglib无线生成类）

虚拟机参数：

-XX:MetaspaceSize=8m

-XX:MaxMetaspaceSize=128m

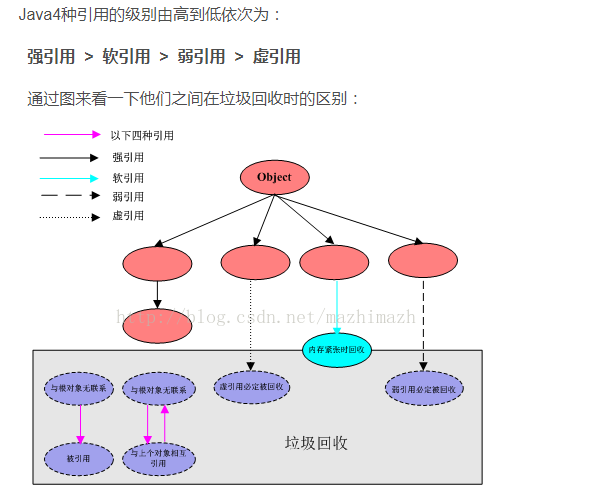
-XX:+PrintFlagsInitial(打印虚拟机初始化参数)



**Metaspace参数**

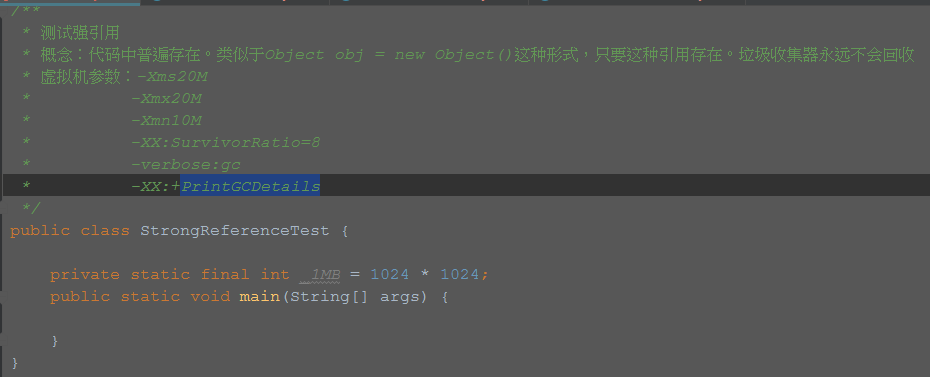
|  |  |
| --- | --- |
| MetaspaceSize | 初始化的Metaspace大小，控制Metaspace发生GC的阈值。GC后，动态增加或者降低MetaspaceSize，默认情况下，这个值大小根据不同的平台在12M到20M之间浮动 |
| MaxMetaspaceSize | 限制Metaspace增长上限，防止因为某些情况导致Metaspace无限使用本地内存，影响到其他程序，默认为4096M |
| MinMetaspaceFreeRatio | 当进行过Metaspace GC之后，会计算当前Metaspace的空闲空间比，如果空闲比小于这个参数，那么虚拟机增长Metaspace的大小，默认为40，即40% |
| MaxMetaspaceFreeRatio | 当进行过Metaspace GC之后，会计算当前Metaspace的空闲空间比，如果空闲比大于这个参数，那么虚拟机会释放部分Metaspace空间，默认为70，即70% |
| MaxMetaspaceExpanison | Metaspace增长时的最大幅度，默认值为5M |
| MinMetaspaceExpanison | Metaspace增长时的最小幅度，默认为330KB |
|  |  |

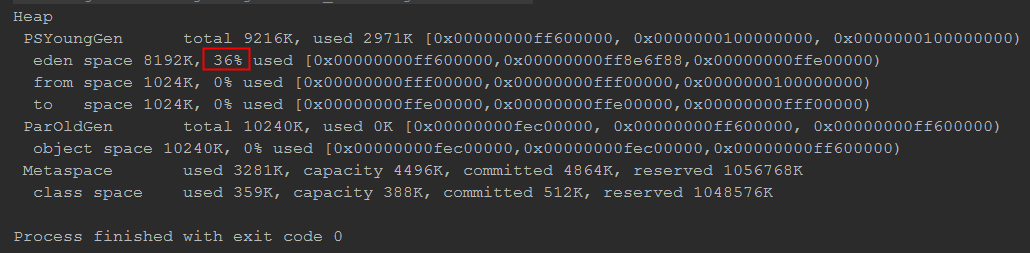
**四种引用状态**



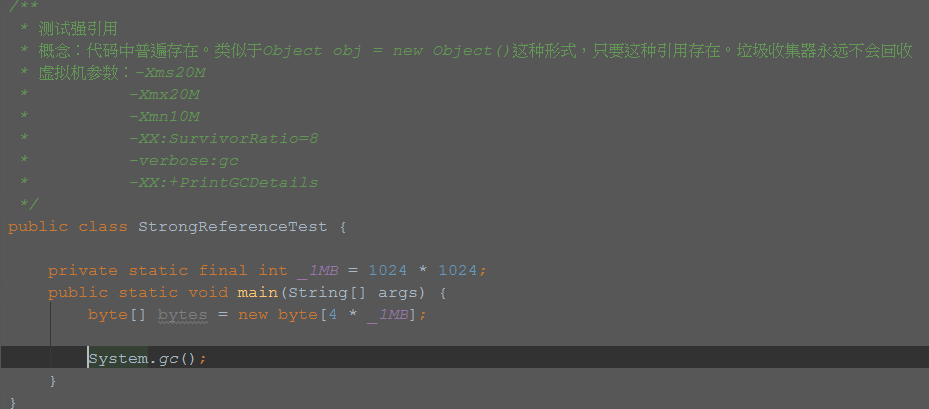
**强引用**

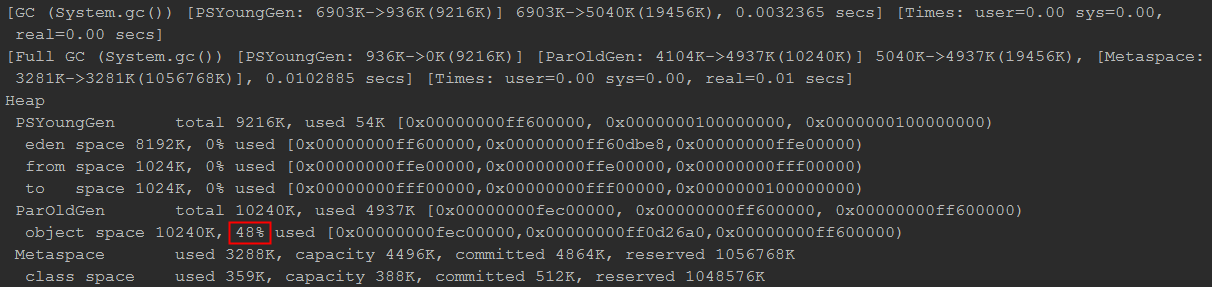
验证要点：当一个对象到GC Roots没有任何引用链相连，则证明这个对象不可用的（要被回收）



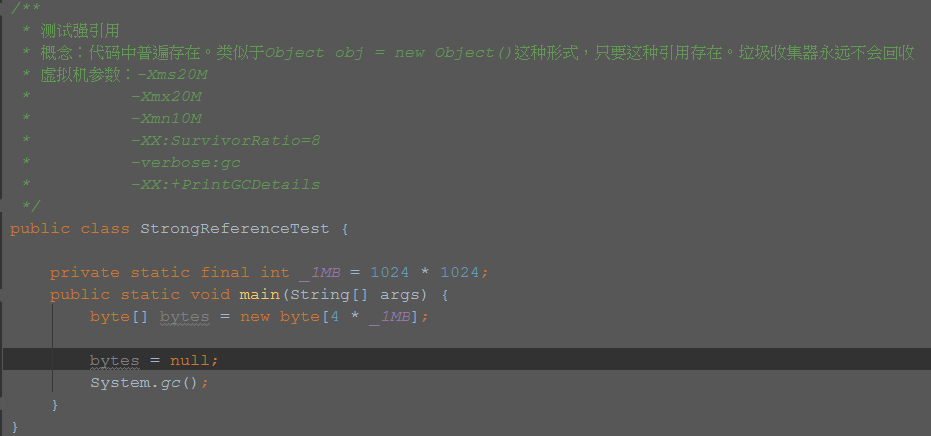


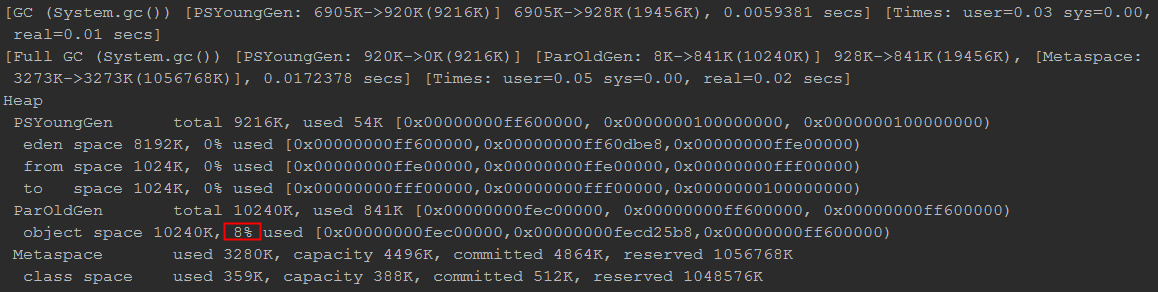
在没有任何对象的情况下，虚拟机启动的时候就会加载一部分数据到内存中，占用了新生代Eden区的36%





放入一个4M的对象，触发一次FGC，可以看到该对象没有被回收，被移到了老年代

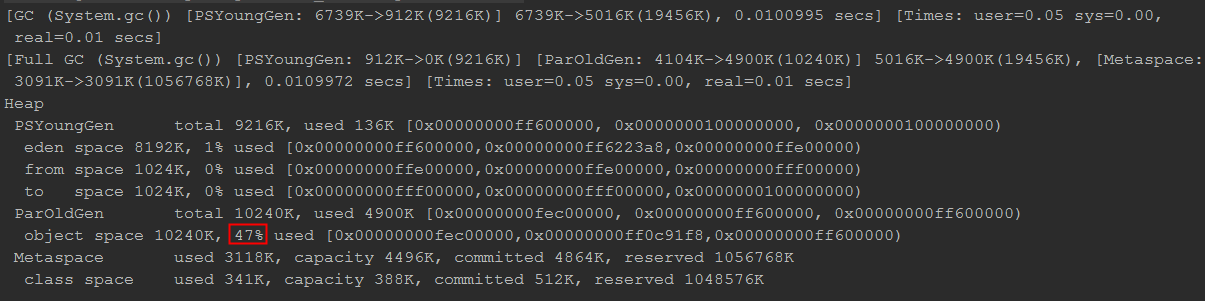




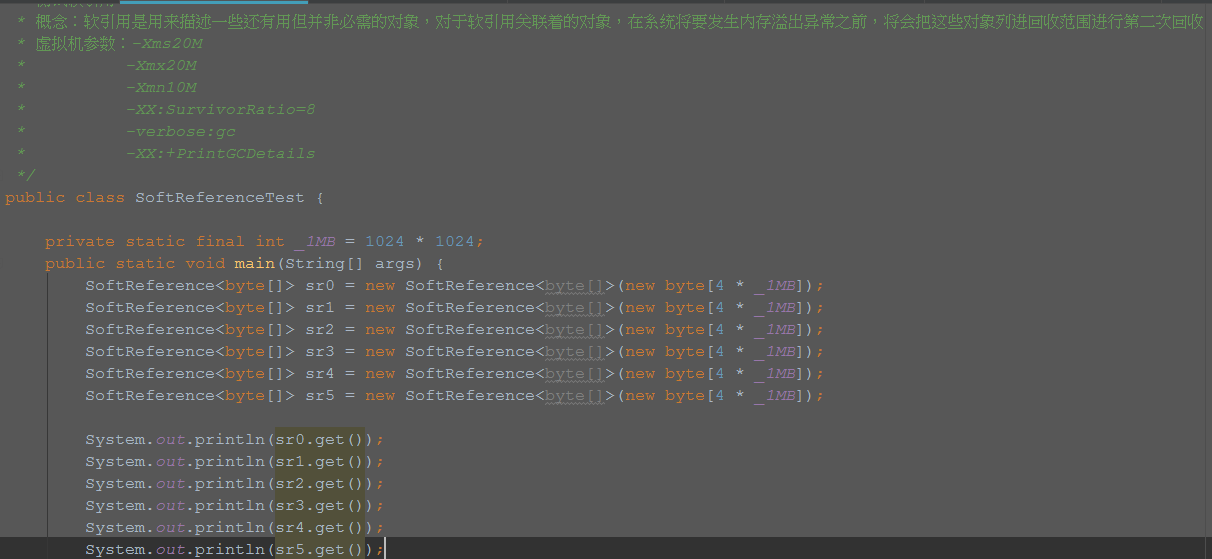
将该对象的引用置为null，此时触发一次GC，该4M对象被回收了

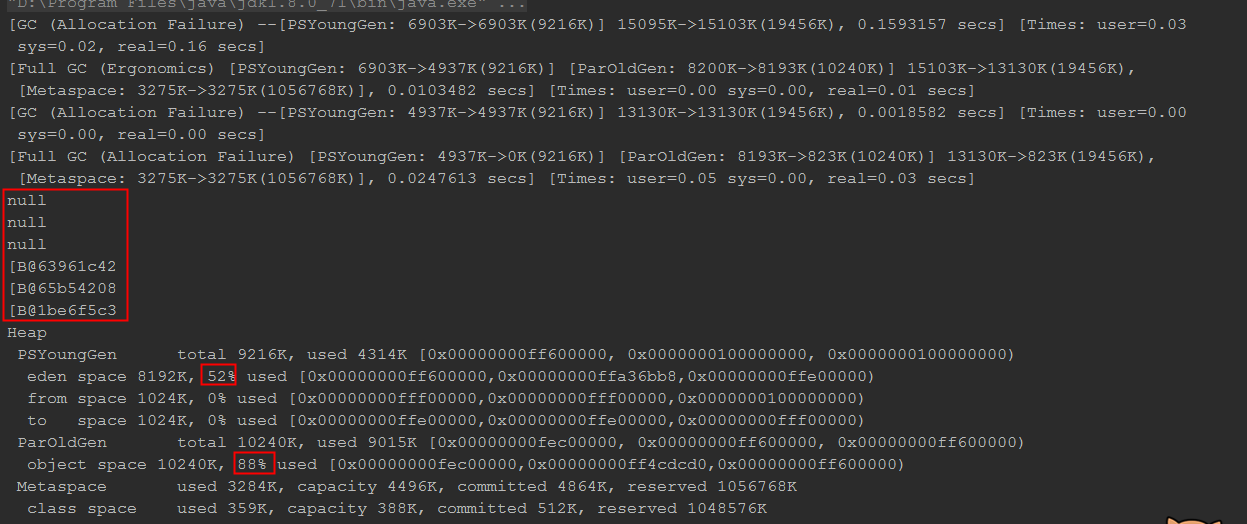
**软引用**





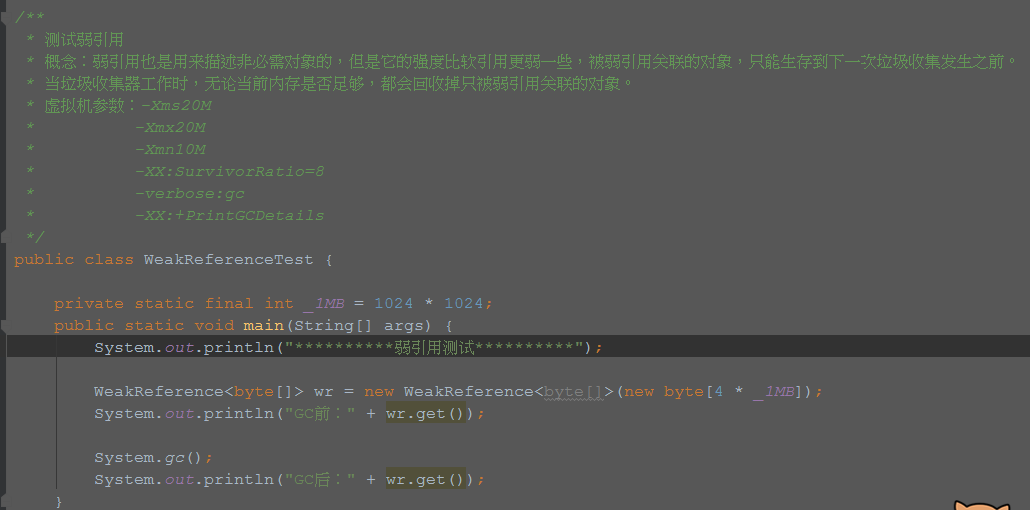
在没有内存溢出的场景下，软引用和强引用一样，只要该软引用存在，对象就不会被回收

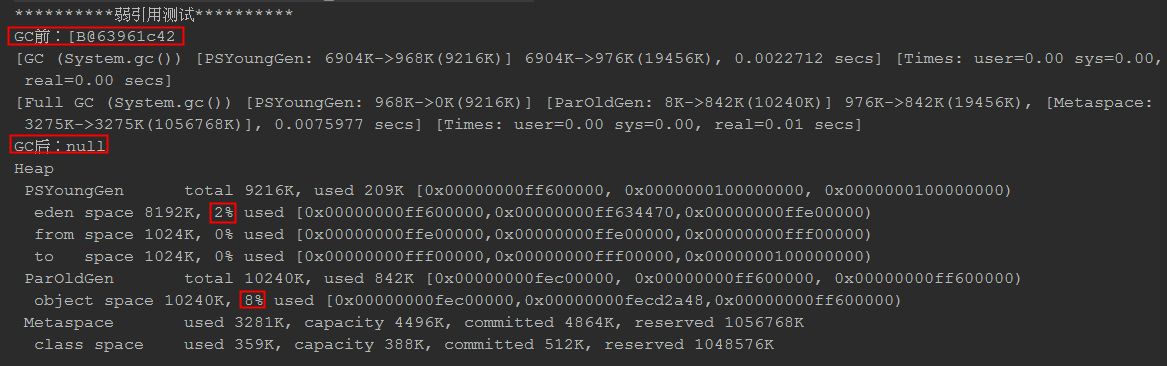




可以看出在内存溢出之前，即使软引用存在，对象也会被回收，最上面的三个对象被回收，剩余的三个两个在老年代，一个在新生代

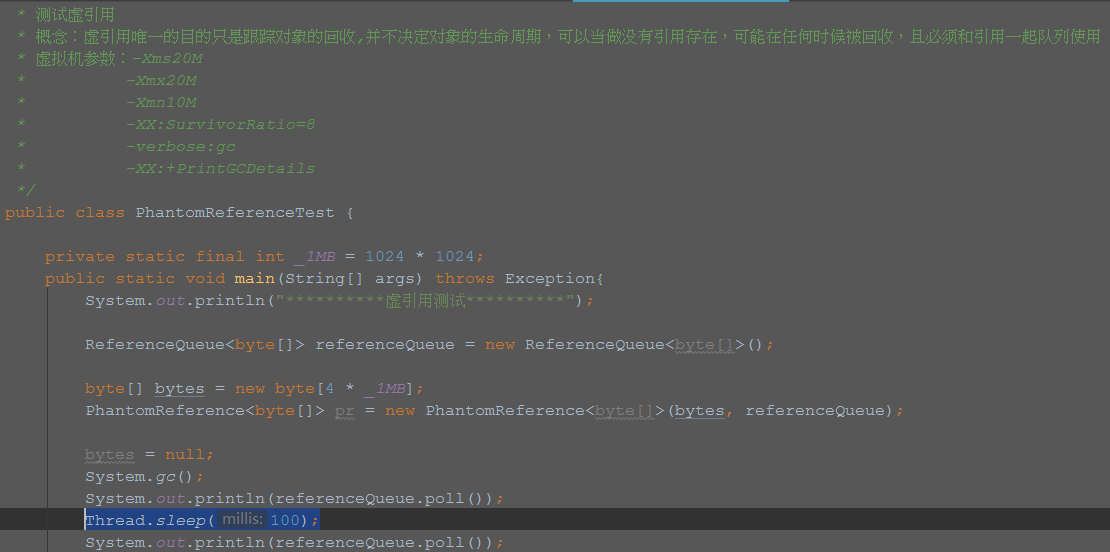
**弱引用**

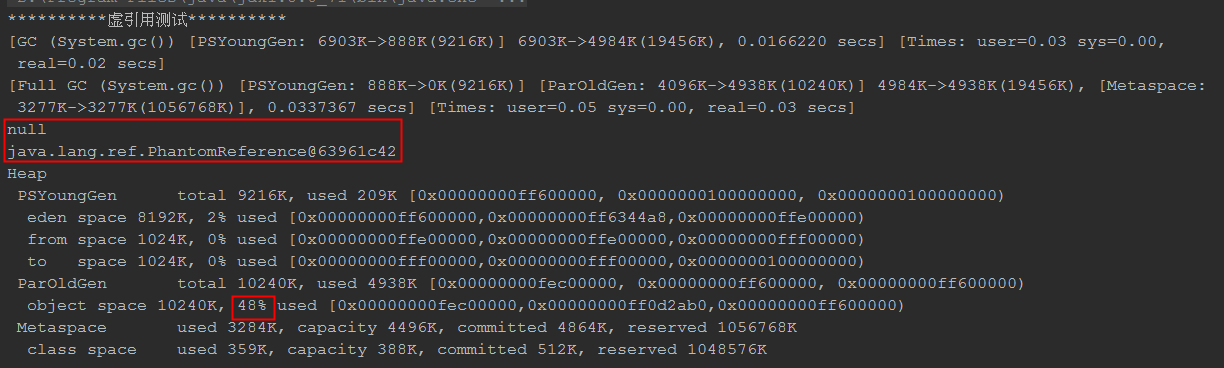




弱引用关联的对象，即使内存充足，只要发生垃圾回收，该对象也会被回收掉

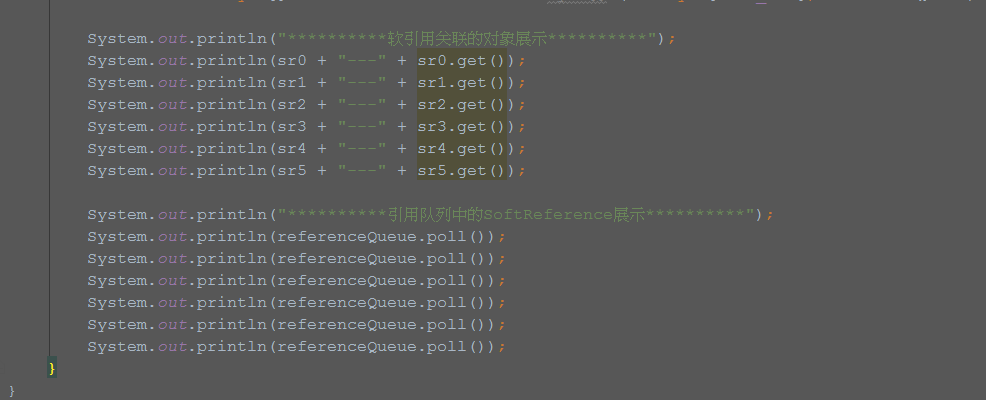
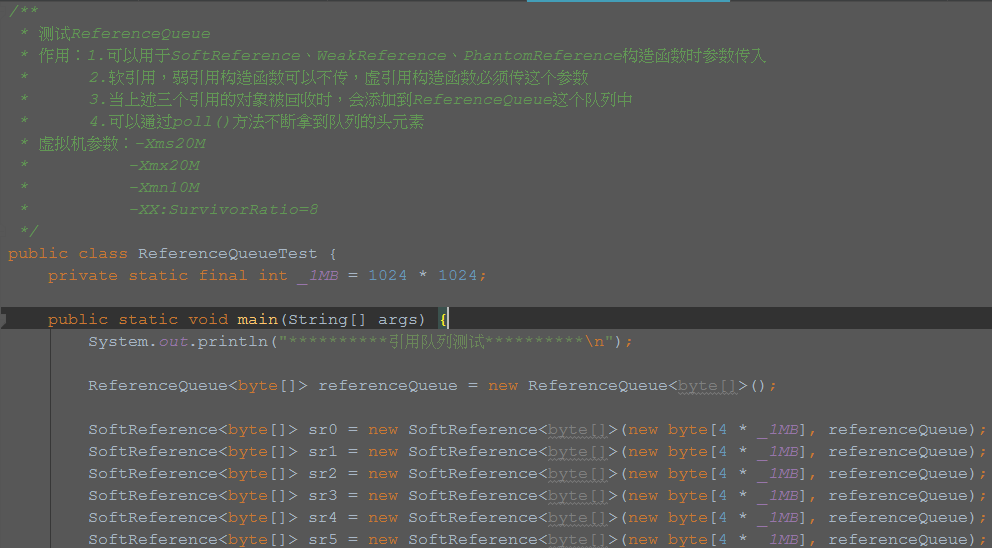
**虚引用**

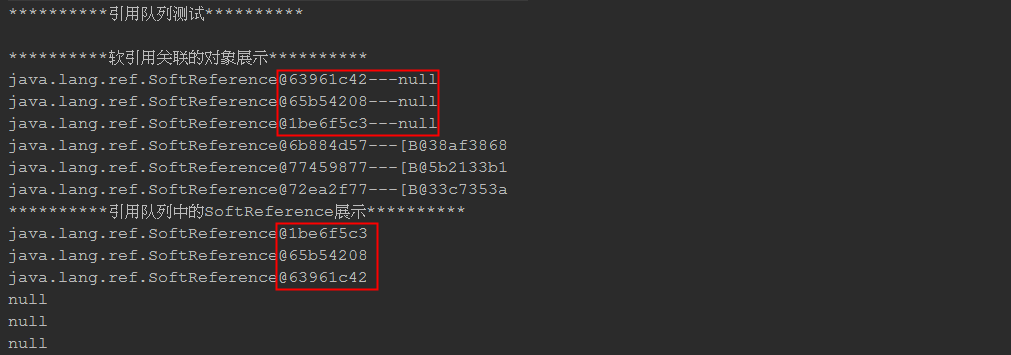




其作用是跟踪垃圾回收

**ReferenceQueue**

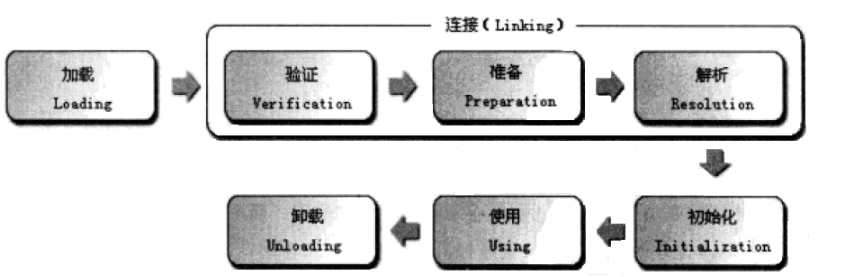




可以看出开始的三个软引用被回收了，且放入了ReferenceQueue的队列中了。

**Java类加载机制**

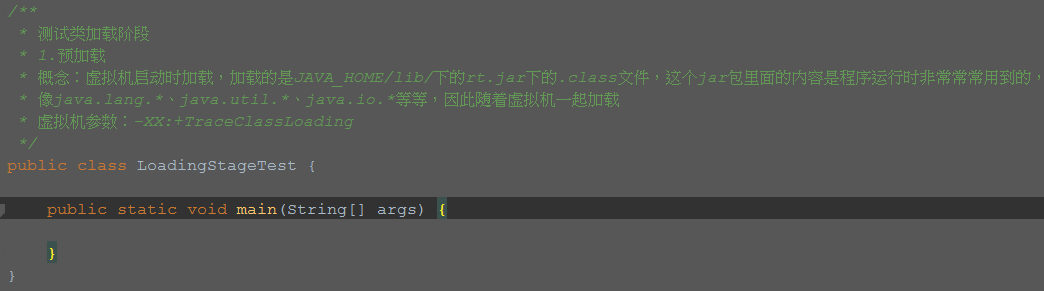
**类使用的7个阶段**

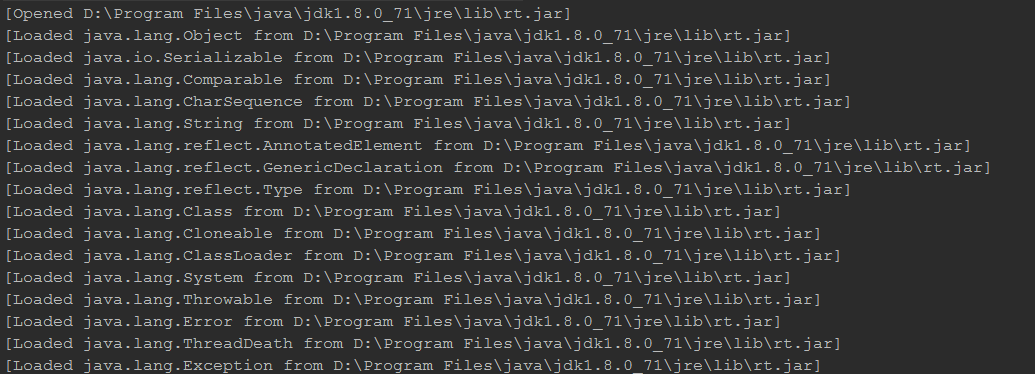


其中加载、验证、准备、初始化、卸载的顺序是一定的，解析可能在初始化之后，为了支持java的动态绑定。

**加载loading**

1. 预加载





1. 运行时加载

虚拟机在用到一个class的时候，会去内存中先查看一下这个.class文件有没有被加载，没有的话，按照类的全限定名来加载这个类

1. 、获取.class文件的二进制流
2. 、从zip包中获取，这就是以后jar、ear、war格式的基础
3. 、从网络中获取，典型应用就是Applet
4. 、运行时计算生成，典型应用就是动态代理技术
5. 、由其他文件生成，典型应用就是JSP，即由JSP生成对应的.class文件
6. 、从数据库中读取，这种场景比较少见
7. 、将类信息、静态变量、字节码、常量这些.class文件的内容放到方法区
8. 、在内存中生成一个.class文件的Class对象，可以用这个对象作为这个方法区各种数据的访问入口。一般这个对象在堆中，但是HotSpot是放在方法区中的

**验证**

这一阶段主要保证.class文件的字节流包含的信息符合当前虚拟机的要求，并不会危害虚拟机自身的安全。

包括以下几个阶段：

1. 、文件格式验证

该阶段主要验证.class文件的版本号与当前虚拟机的jdk版本号是否兼容，高版本的.class文件无法在低版本的虚拟机上运行，即只能向上兼容。其中.class文件的第5至8的4个字节是该文件的主次版本号。如果不兼容运行，将会抛出java.lang.UnsupportedClassVersionError

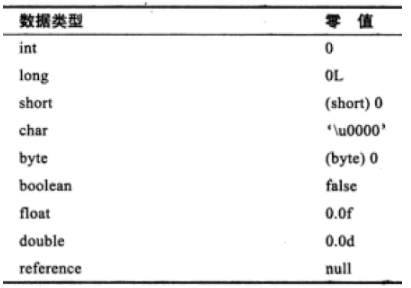
1. 、元数据验证
2. 、字节码验证
3. 、符号引用验证

**准备**

这一阶段是正式为类变量分配内存并设置其初始化值，这些变量使用的内存在方法区被分配

1. 、这时候分配的是类变量，不是实例变量，实例变量将会随着对象实例化的时候一起在堆中分配
2. 、这个阶段初始化是指不被final修饰的变量，比如private static int a = 123，a在准备阶段之后的值是0，不是123，123是在初始化的时候进行的动作。但是，如果private static final int a = 123,用了final修饰，那么在准备阶段a的值就是123了

各个数据零值如下：

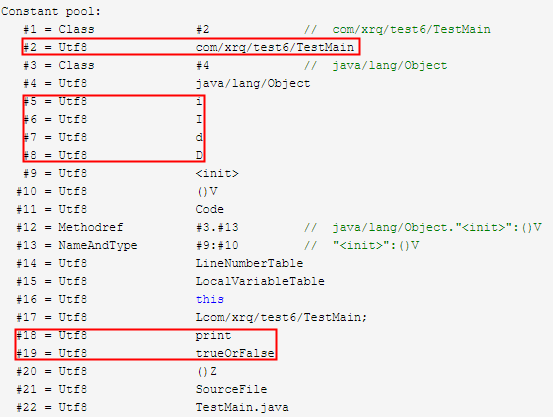


**解析**

解析阶段是虚拟机将常量池内的符号引用替换成直接引用的过程

符号引用是编译方面原理方面的概念：

1. 、类和接口的全限定名
2. 、字段的名称和描述符
3. 、方法的名称和描述符



Utf8代表的就是符号引用，上述标红的就是符号引用，分别代表的类的全限定名，字段的名称，方法的名称。符号引用和虚拟机的内存布局没有关系，引用的目标未必加入了虚拟机中。

**初始化**

初始化做的事就是给static变量（final修饰的除外，在准备阶段已经赋值完毕）赋予用户指定的值以及执行静态代码块

多线程场景下，虚拟机会保证类的初始化被正确的加锁，同步

Java虚拟机规定了5种情况必须立即对类进行初始化

1. 使用new实例化一个对象、读取（getstatic）或者设置一个类的静态字段(putstatic)（final修饰除外）、调用一个类的静态方法(invokestatic)
2. 使用java.lang.reflect包中的方法对类进行发射调用的时候
3. 初始化一个类的时候，发现其父类还没有进行初始化的时候,需要先初始化其父类
4. 虚拟机启动的时候，会先初始化用户指定包含main()方法的那个类
5. 但是用动态语言支持时，如果一个java.lang.invoke.MethodHandle实例后解析结果REF\_putStatic,REF\_getStatic,REF\_invokeStatic的方法句柄时，当改方法句柄对应的类没有初始化时，需要初始化该类。（动态语言支持详情请查看）

上述前四种称为主动引用，除了上述的场景引用类的方式都不会触发类的初始化，称为被动引用

1.子类引用父类静态字段，不会导致子类初始化

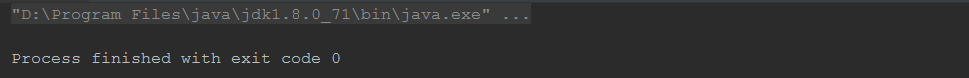




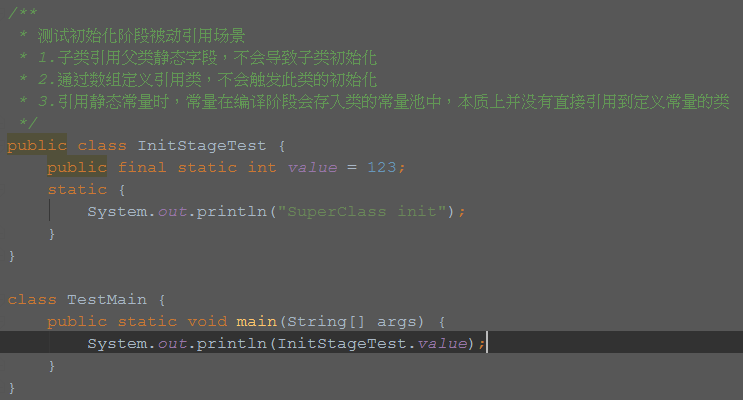
结果可以看到，符合类必须初始化第一条，获取一个类的静态变量。

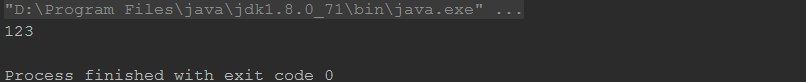
2.通过数组定义引用类，不会触发该类的初始化





1. 引用静态常量时，常量在编译阶段会存入类的常量池中，本质上并没有直接引用到定义常量的类（即引用被final修饰的类变量不会触发类的初始化）

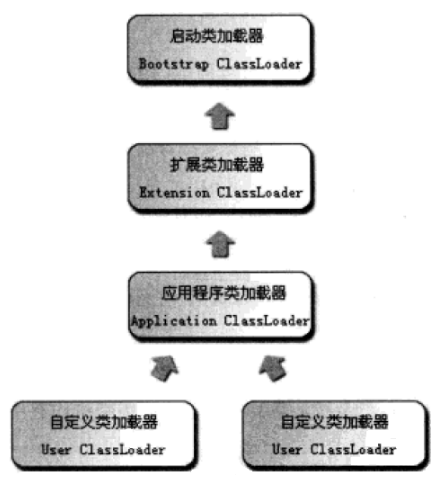




可以看到静态方法没有执行，所以类并没有初始化

**类加载器**

概念：虚拟机设计团队把类加载阶段张的"通过一个类的全限定名来获取此类的二进制字节流"这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为"类加载器"。



1. 启动类加载器Bootstrap ClassLoader

它用来加载 Java 的核心库(JAVA\_HOME/lib下的类库)，是用原生C++代码来实现的，并不继承自java.lang.ClassLoader，java应用无法直接使用。

1. 扩展类加载器Extension ClassLoader（sun.misc.Launcher$ExtClassLoader）

它用来加载 Java 的扩展库(JAVA\_HOME/lib/ext或者被java.ext.dirs系统变量指定所指定的路径中所有类库) 开发者可以直接使用扩展类加载器。

System.out.println(System.getProperty("java.ext.dirs"));



1. 应用程序类加载器Application ClassLoader（也被称为系统类加载器sun.misc.Launcher$AppClassLoader）

System.out.println(ClassLoader.getSystemClassLoader());



可以看到jdk认为Application ClassLoader是系统类加载器

其父类加载器：System.out.println(ClassLoader.getSystemClassLoader().getParent());



父类的父类：System.out.println(ClassLoader.getSystemClassLoader().getParent().getParent());



null是由于Bootstrap classloader是由C++实现的，如果ClassLoader得到的是null,就是启动类加载器

1. 自定义类加载器