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

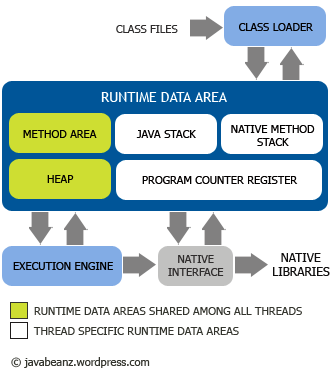
连接内存和寄存器的是地址总线，地址总线的宽度影响了物理地址的索引范围，因为总线宽度决定了处理器一次可以从寄存器或内存中获取多少个Bit，同时也决定了处理器最大可以寻址的地址空间。

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

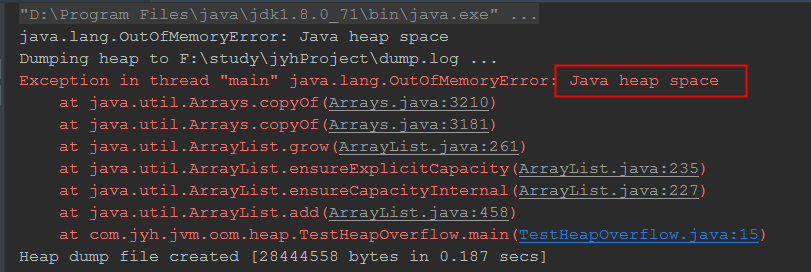
-XX:+HeapDumpOnOutOfMemoryError

-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。

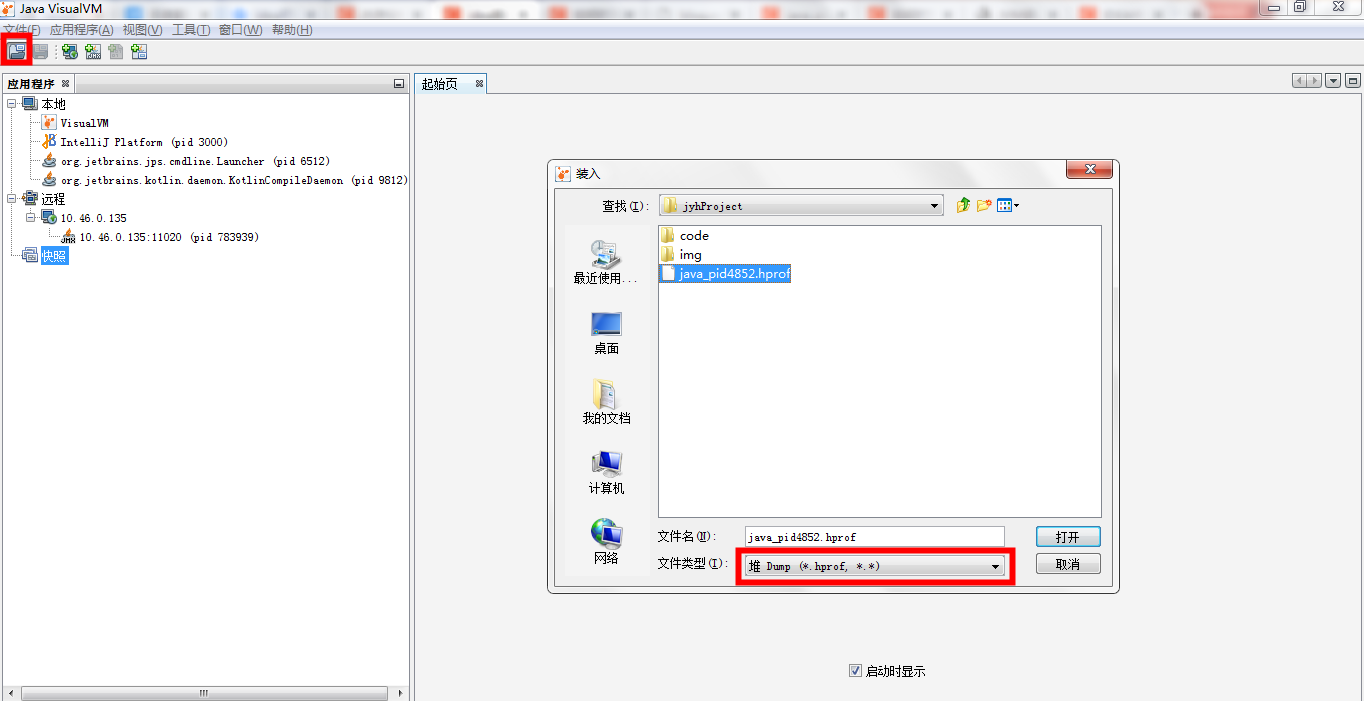
涉及JVM参数-Xmn20M（堆初始化大小） -Xmx20M（堆最大值大小）

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

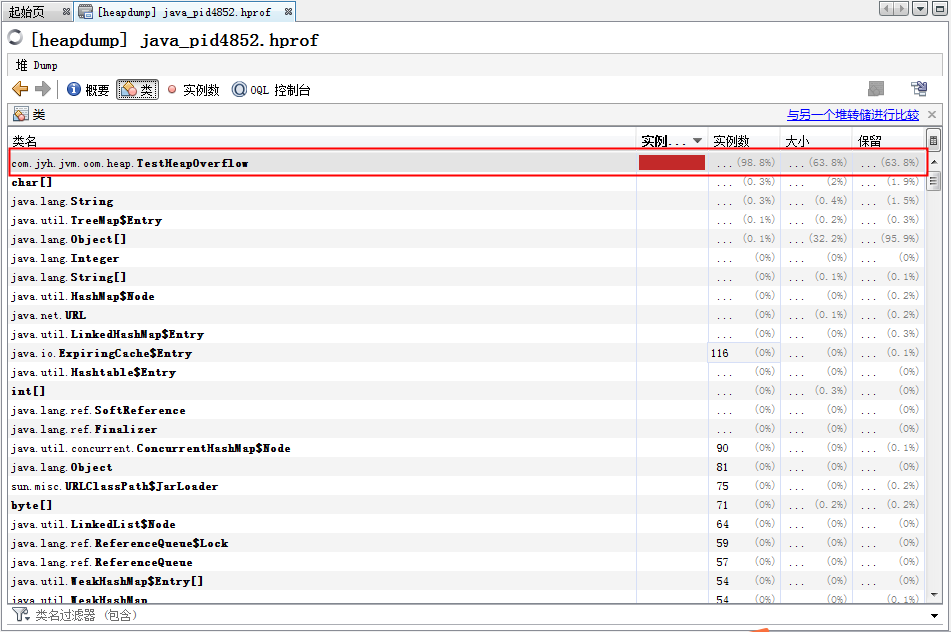


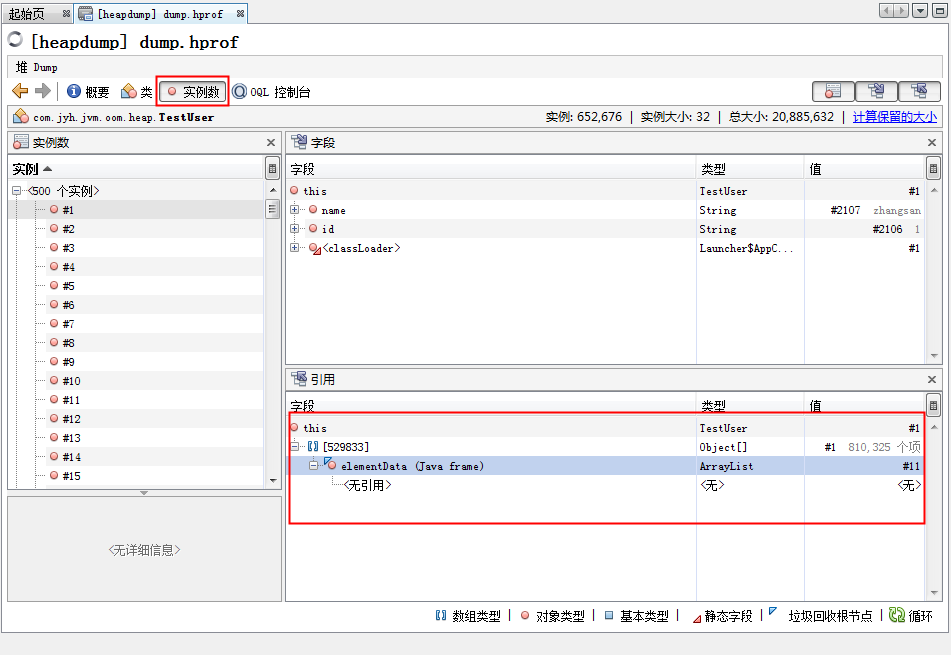
但是往往堆栈异常信息上反映不出来此时堆上面是被什么对象撑爆的，此时可以设置HeapDumpOnOutMemoryError参数后会打印类似于java\_pidxxxx.hprof的堆内存快照文件，可以使用以下三种方式打开快照：

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





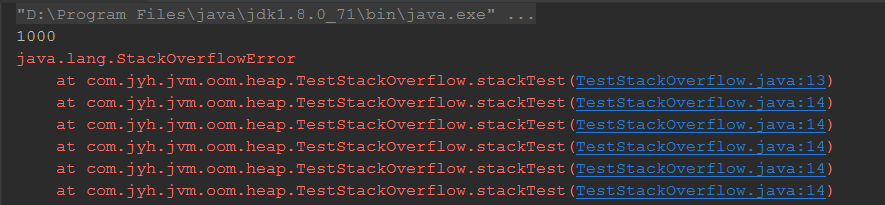




基本上可以通过上图看出是哪个大对象占用太多的内存一直没有释放导致堆溢出，而且可以通过最后一张图的引用看对象的树形结构，找出这个对象在哪里被引用的，这时候需要结合业务分析这个大对象产生的场景

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）

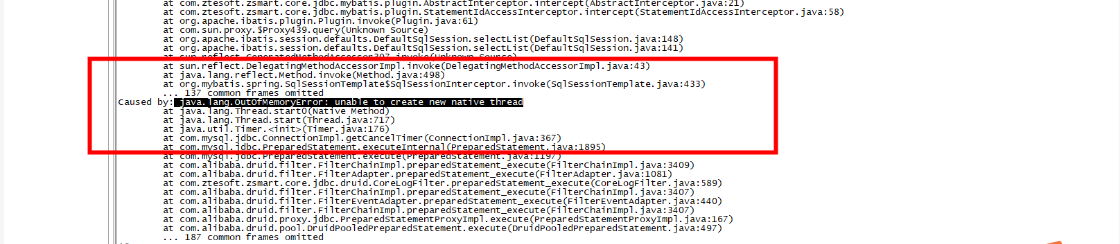
测试案例：com.jyh.jvm.oom.stack.TestStackOverflow



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

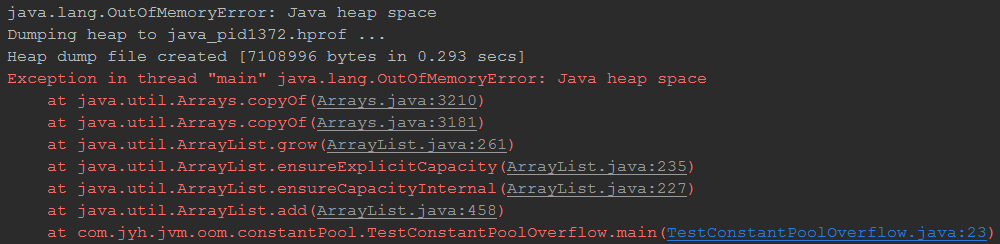
解决办法：ulimit -u查看某用户当前操作系统的最大允许进程数，cat /proc/sys/kernel/pid\_max 可以查看当前系统允许的最大线程数，在/etc/sysctl.conf下面kernel.pid\_max配置项可以配置操作系统允许的最大线程数，可以通过ls /proc/<pid>/fd | wc -l查看当前进程下的线程数，最终是通过改操作系统最大限制线程数解决的，但同时生产环境的-Xss是没有配置的，默认是1M,可以降低这个值，达到多创建线程数的目的

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

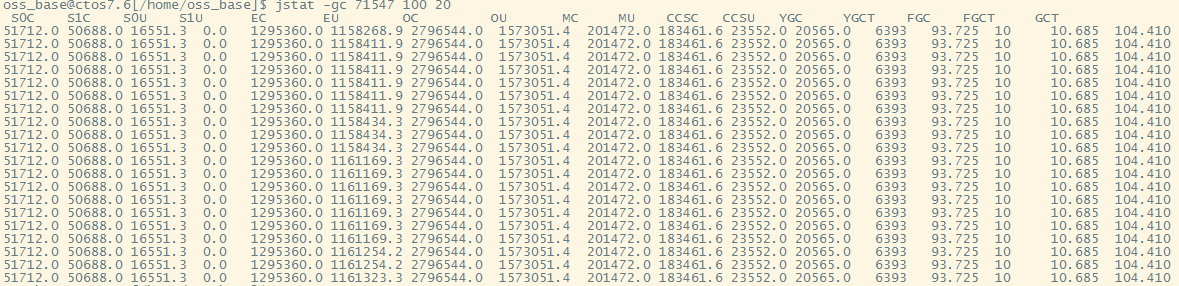
**二：并行和并发**

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

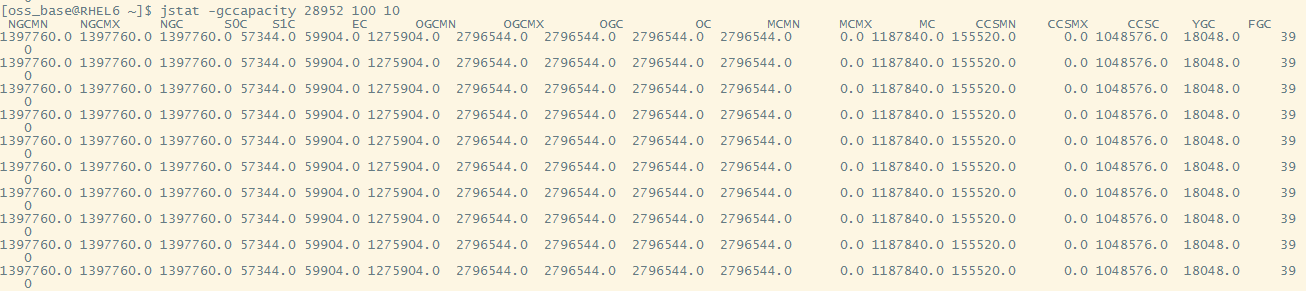
YGCT：从应用程序开始到采样的时候年轻代区发生GC的总时间(s)

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）

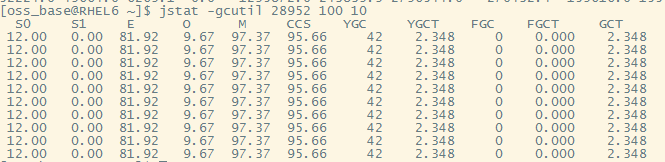
MC：元数据区当前容量（1.8之前是PGC）

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

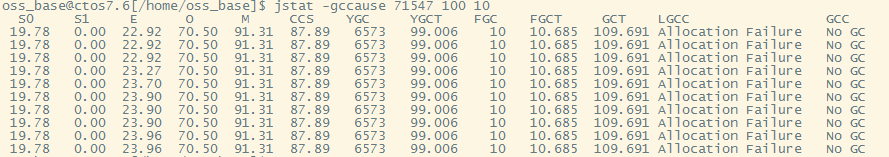
E：年轻代中Eden区使用率

O：老年代使用率

M：元数据区使用率

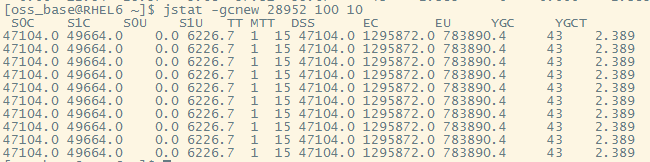
CCS：压缩类型使用率（属于元数据区）

**指令：jstat -gccause <pid>**



LGCC：描述YGC和FGC发生的原因

**指令：jstat -gcnew <pid>**

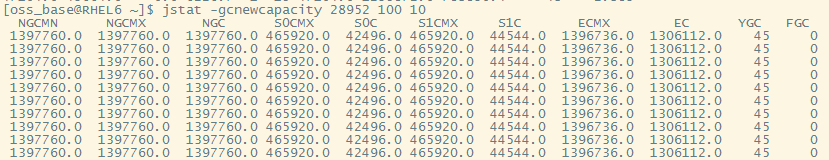


TT：持有次数限制

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>**

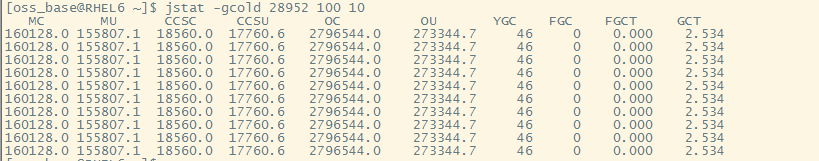


S0CMX：第一个Survivor区的最大容量

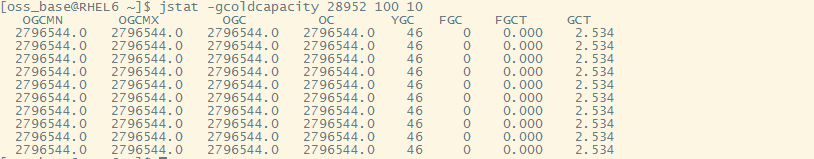
S1CMX：第二个Survivor区的最大容量

ECMX：Eden区最大容量

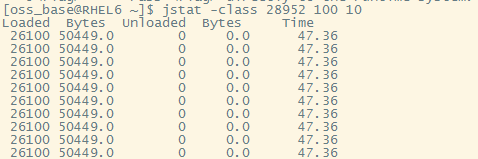
**指令：jstat -gcold <pid>**



**指令：jstat -gcoldcapacity <pid>**



**指令：jstat -class <pid>**



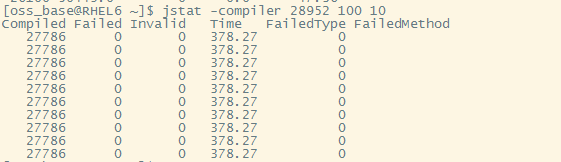
Loaded：装载类数量

Bytes：装载类大小

Unloaded：卸载类 数量

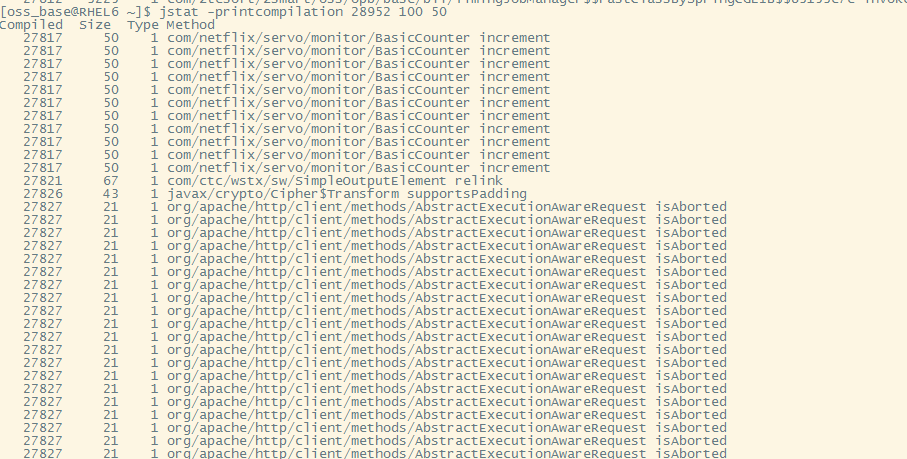
Time：耗费总时间

**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

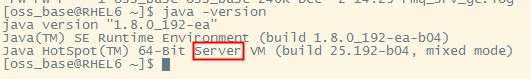
我们使用的HotSpot中内置了两个JIT编译器，即C1编译器和C2编译器，默认采用的是解释器和一个编辑器配合的方式进行工作。HotSpot在启动的时候会根据自身版本以及宿主机器的硬件性能自动选择运行模式，比如会检测宿主机器是否为服务器、比如J2SE会检测主机是否有至少2个CPU和至少2GB的内存。

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

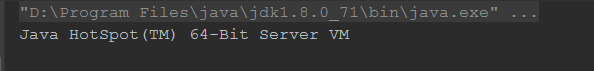
2、如果不是，则虚拟机会以Client模式运行，该模式与C1编译器共同运行，更注重编译的速度，启动速度快，更适合用在客户端的版本下，针对GUI进行了优化

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

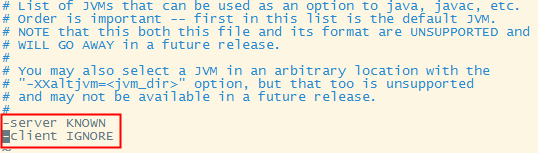


1. 查看本地安装的idea自带的虚拟机版本

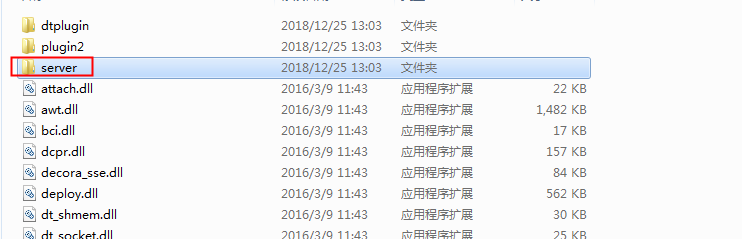
System.out.println(System.getProperty("java.vm.name"));

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

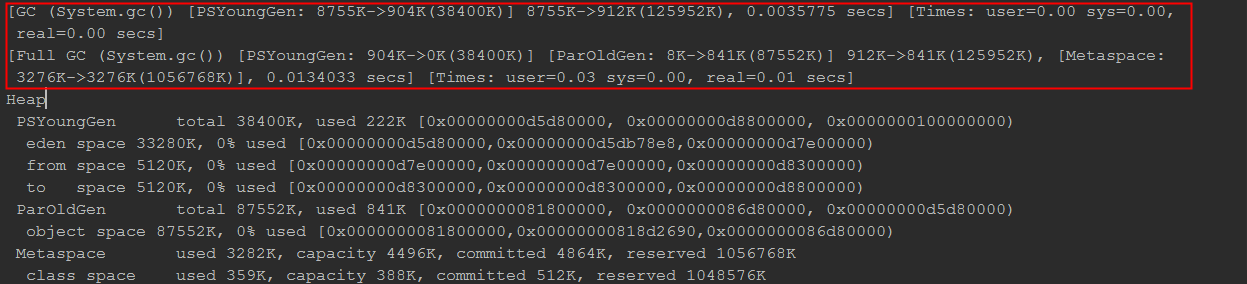
**Java垃圾回收（GC）机制详解**

**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：

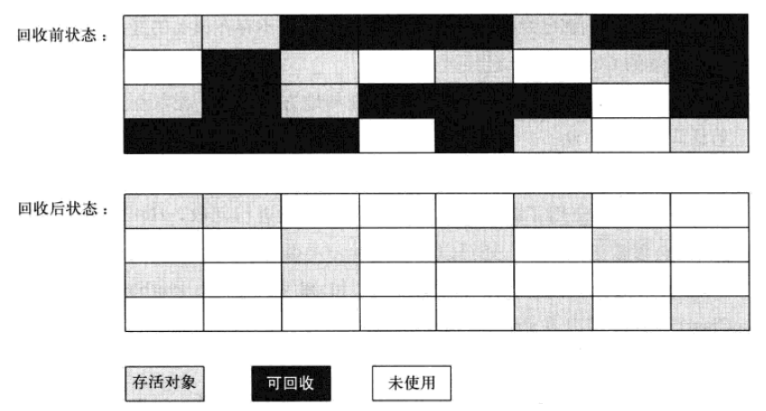
1. 该类的所有实例都被回收，即堆中没有该类的任何实例
2. 该类的加载器ClassLoader已经被回收
3. 该类对应的java.lang.Class对象没有被任何一处地方引用，没有任何地方通过反射获取该类

**四：垃圾回收算法**

垃圾回收思想：

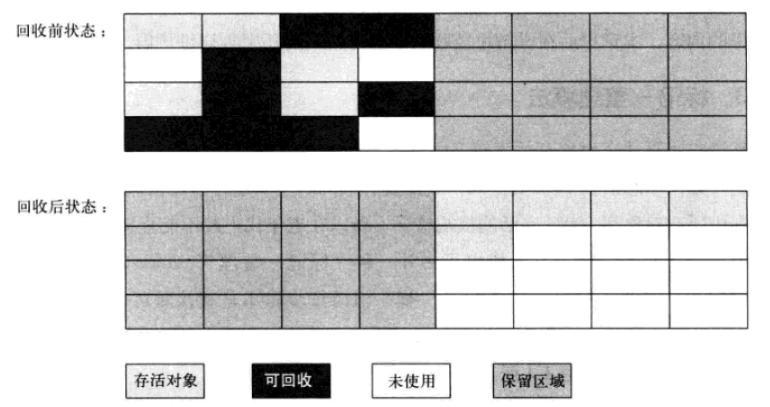
1. 标记-清除（Mark-Sweep）算法

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



1. 标记整理（Mark-Compact）算法
2. 分代收集算法

垃圾收集器

1. Serial收集器
2. ParNew收集器
3. Parallel收集器
4. Serial Old收集器
5. Parallel Old收集器
6. CMS收集器
7. G1收集器