JVM

JVM是运行在操作系统之上的，与硬件没有直接的交互。

JVM体系结构：

1. Class files
2. 类装载器
3. 运行时数据区
4. 执行引擎
5. 本地方法接口

# JVM运行时数据区



### 1.程序计数器

　　占据一块较小的内存空间，可以看做当前线程所执行的字节码的行号指示器。在虚拟机概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，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。

　　由于jvm的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此未来线程切换后能恢复到正确的执行位置，**每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们成这类内存区域为“线程私有”的内存。**

　　如果线程正在执行的是一个Java方法，这个计数器记录的则是正在执行的虚拟机字节码指令的地址；

　　如果正在执行的是Native方法，这个计数器则为空（undefined）。

**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。**

### 2.Java虚拟机栈

**线程私有**，生命周期和线程相同，虚拟机栈描述的是Java方法执行的内存模型，**每个方法在执行的同时都会创建一个栈帧  用于存储局部变量表，操作数栈，动态链接，方法出口等信息。每一个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。**

　　局部变量表存放了编译期可知的各种**基本类型数据**（boolean、byte、char、short、int、float、long、double）、**对象引用**、returnAddress类型（指向了一条字节码指令的地址）。

　　其中64位长度的long和double类型的数据会占用2个局部变量表空间（slot），其余的数据类型只占用1个。局部变量表所需的内存空间在编译期完成分配，当进入一个方法时，这个方法所需要在栈帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对此区域规定了两种异常状况：**如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出Stack OverflowError异常；如果虚拟机栈可以动态扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。**

### 3.本地方法栈

　　本地方法栈与虚拟机栈所发挥的作用非常相似，他们之间的区别不过是**虚拟机栈为虚拟机执行Java方法**（字节码）服务，**而本地方法栈则为虚拟机中使用到的native方法服务**。在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机直接把本地方法栈和虚拟机栈合二为一，与虚拟机栈一样也会抛出Stack OverflowError异常和OutOfMemoryError异常。

### 4.Java堆

 　　对于大多数应用来说，**堆空间是jvm内存中最大的一块。**Java堆是被**所有线程共享**，虚拟机启动时创建，此内存区域唯一的目的就是**存放对象实例**，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配，但是随着JIT编译器的发展和逃逸分析技术逐渐成熟，栈上分配，标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也就变得不那么绝对了。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为“GC堆”。从内存回收角度看，由于现在收集器基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。从内存分配的角度来看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区。不过无论如何划分，都与存放内容无关，无论哪个区域，存储的都仍然是对象实例，进一步划分的目的是为了更好的回收内存，或者更快的分配内存。（**如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常**。）

### 5.方法区（也有人叫永久代）

和堆一样所有线程共享，主要用于存储已被jvm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。

　　（在JDK1.7发布的HotSpot中，已经把字符串常量池移除方法区了。）

### 6.常量池

　　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、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。

　　Java虚拟机对class文件每一部分的格式都有严格规定，每一个字节用于存储哪种数据都必须符合规范才会被jvm认可。但对于运行时常量池，Java虚拟机规范没做任何细节要求。

　　运行时常量池有个重要特性是动态性，Java语言不要求常量一定只在编译期才能产生，也就是并非预置入class文件中常量池的内容才能进入方法区的运行时常量池，运行期间也有可能将新的常量放入池中，这种特性使用最多的是String类的intern()方法。

既然运行时常量池是方法区的一部分，自然受到方法区内存的限制。当常量池无法再申请到内存时会抛出outOfMemeryError异常。

# 对象的创建

### 2.1 当虚拟机遇到一条New指令时：会进行如下步骤

1、检查指令的参数（即工作中我们New的对象），能否在常量池中找到它的符号引用。

2、如果存在，检查符号引用代表的类是否被加载、解析、初始化过。（如果没有则执行类的加载-----相关加载过程参考《[Jvm类的加载机制](https://www.cnblogs.com/chenpt/p/9777367.html" \t "https://www.cnblogs.com/chenpt/p/_blank)》）。

3、加载通过后，虚拟机将为新生对象分配内存。（所需内存大小在类加载完成后便可确定）

### 2.2 两种内存分配的方式：

**指针碰撞**：假设Java堆中的内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边。中间放着一个指针作为分界点的指示器，分配内存就仅仅是把指针往空闲空间那边挪动一段与对象大小相等的距离。这种方式则属于指针碰撞。

**空闲列表**：如果堆中的内存并不是规整的，已使用的内存和空闲内存相互交错，显然无法使用指针碰撞。虚拟机就必须维护一个列表，记录哪些内存是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新记录表上的数据。这种方式属于空闲列表。

具体选择哪种分配方式由Java堆决定，而Java堆是否规整，则有GC收集器决定。因此使用Serial、ParNew等带Compact过程的收集器时，系统采用的分配算法是指针碰撞。而使用CMS这种基于Mark-Sweep算法的收集器时，通常采用的空闲列表。

### 2.3如何保证分配内存时线程的安全性？

1、对分配内存的动作进行同步处理（实际上虚拟机采用CAS配上失败重试的机制保证了更新操作的原子性）

2、把分配内存的动作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））。

**类加载器**

类装载器：负责加载class文件。

第一类是虚拟机自带的类加载器，有三种：

1：启动类加载器（Bootstrap）C++

2：扩展类加载器（Extension）Java

3：应用程序类加载器（App）Java ，也叫系统类加载器，加载当前应用的classpath的所有类

第二类是用户自定义加载器，Java.lang.ClassLoader的子类，用户可以定制类的加载方式

**类加载器的双亲委派机制**

**沙箱机制（防止恶意代码对java的破坏）**

比如自己写了一个java.lang.String

主要是为了保证java的沙箱机制

双亲委派:某个特定的类加载器在接到加载类请求的时候，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成加载任务，就返回成功，只有父类加载器无法完成加载任务的时候，才自己去加载。

**执行引擎**

负责解释命令，提交操作系统执行。

**Native Interface本地方法接口**

Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI接口调用其他语言来实现对操作系统的访问。本地接口的作用是融合不同编程语言为Java所用，它的初衷是融合C/C++程序，Java在内存中专门开辟了一块区域处理标记为Native的代码，它的具体做法是Native Method Stack中等级Native方法，在Execution Engine执行的时候加载Native libraries（JNI）

目前该方法使用的越来越少，除非是与硬件有关的应用，比如通过Java程序驱动打印机或者Java系统管理生产设备，在企业级应用中已经比较少见。业务现在的异构领域间通信技术很发达，比如可以使用Socker通信，也可以使用WebService等。

**Native Method Stack 本地方法栈**

他的具体做法是在Native Method Stack 中等级Native方法，在Execution Engine执行的时候加载本地方法库。

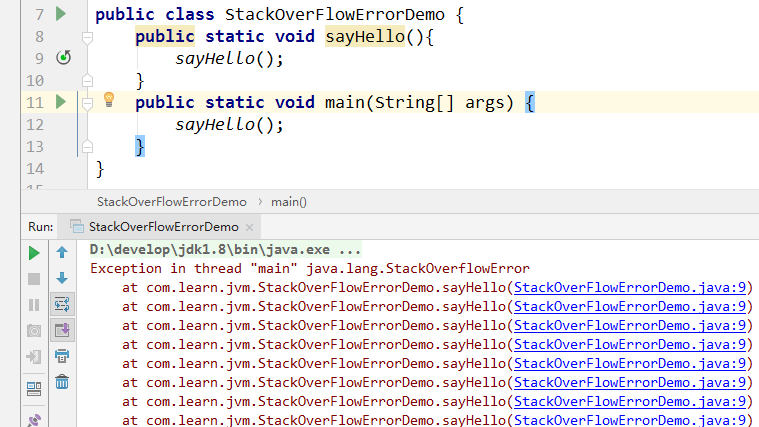
**PC寄存器**

每一个线程都有一个程序计数器，是线程私有的，就是一个指针，指向方法区中的方法字节码（用来存储指向下一条指令代码），由执行引擎读取下一条指令，是一个非常小的内存空间，几乎可以忽略不计。

**栈**

也叫栈内存，主管Java程序的运行，在创建线程的时候创建，它的生命周期是跟随着线程的生命周期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收的问题，只要线程一结束，该栈就Over，生命周期和线程一致，是线程私有的。基本类型中的变量、实例方法、引用类型变量都是在函数的栈内存中分配。

**StackOverFlowError**



**方法区**

1、方法区是线程共享的，通常用来保存装载的类的元结构信息

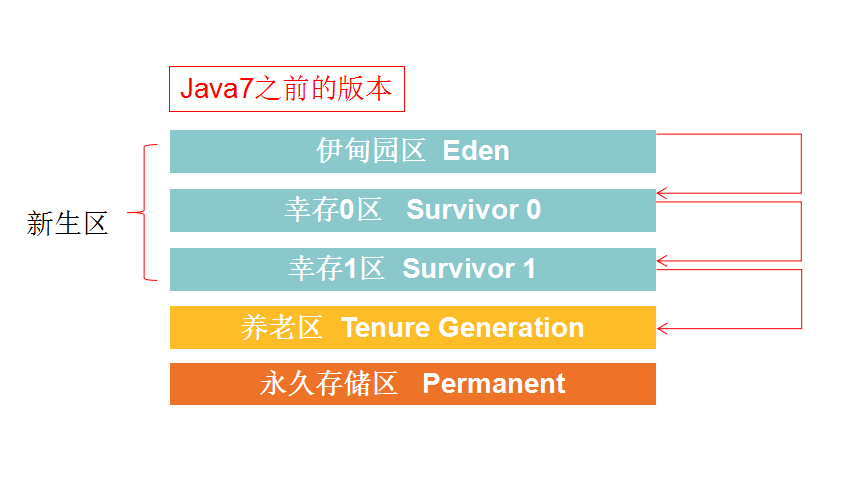
比如：运行时常量池+静态变量+常量+字段+方法字节码+在类/实例/接口 初始化用到的特殊方法。

1. 通常和永久区管理在一起（Java7之前），但是具体的跟JVM的实现和版本有关。

**Heap堆**

一个JVM实例只存在一个堆内存，堆内存的大小是可以调节的。类加载器读取了类文件之后，需要把类、方法、常变量放到堆内存中，保存所有类型的真实信息，以方便执行器执行。

堆内存在逻辑上分为三种：新生区，养老区，永久区



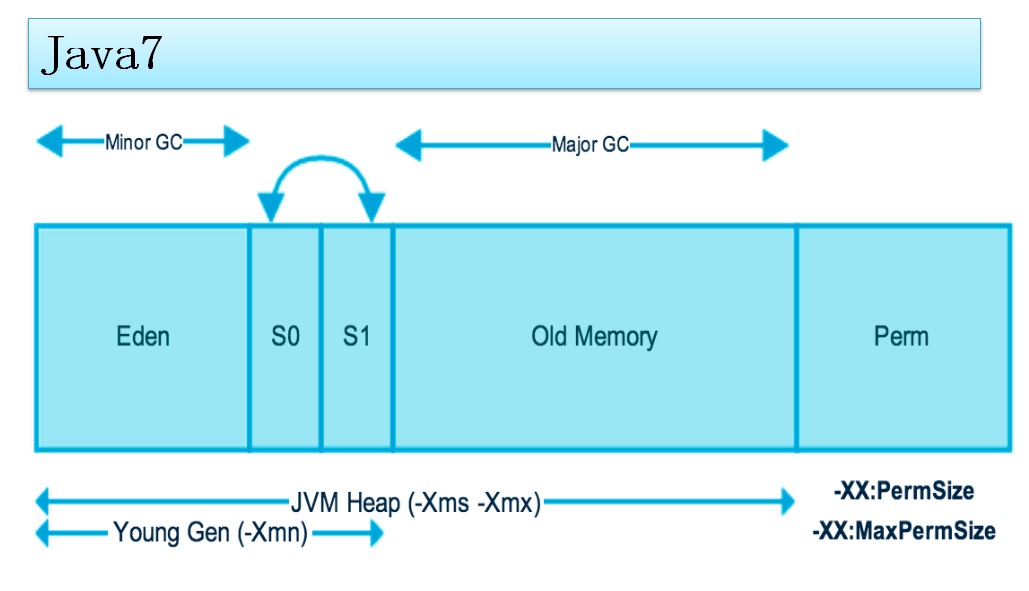
**新生区**

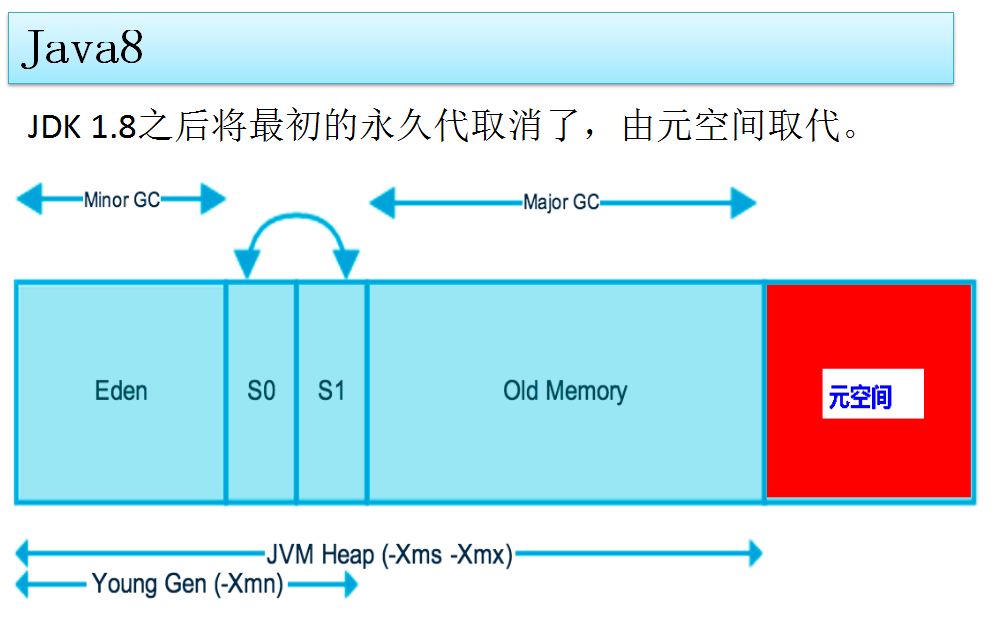
新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分： 伊甸区（Eden space）和幸存者区（Survivor pace） ，所有的类都是在伊甸区被new出来的。幸存区有两个： 0区（Survivor 0 space）和1区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收(Minor GC)，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存0区.若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区。那如果1区也满了呢？再移动到养老区。若养老区也满了，那么这个时候将产生MajorGC（FullGC），进行养老区的内存清理。若养老区执行了Full GC之后发现依然无法进行对象的保存，就会产生OOM异常“OutOfMemoryError”。

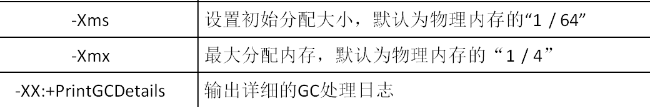
如果出现java.lang.OutOfMemoryError: Java heap space异常，说明Java虚拟机的堆内存不够。原因有：

（1）Java虚拟机的堆内存设置不够，可以通过参数-Xms、-Xmx来调整。

（2）代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在被引用）







# 常用的垃圾回收算法

一、引用计数。有对象引用加一，没有对象引用减一，到0的时候回收

缺点：

每次对对象赋值时均要维护引用计数器，并且计数器本身也有一定的消耗

比较难处理循环引用

JVM的实现一般不采用这种方式。

二、复制算法。赋值算法是在年轻代引用。复制之后有交换，谁空谁是to

1、eden,SurvivorFrom复制到SurvivorTo ，年龄+1

首先，当Eden区满的时候会触发第一次GC,把活着的对象拷贝到SurvivorFrom区域，当Eden区再次触发GC的时候会扫描Eden区和From区域，对这两个区域进行垃圾回收，经过这次回收之后还存活的对象，则直接复制到To区域（如果有对象的年龄已经达到了老年的标准，则复制到老年区域），同时把这些对象的年龄+1

2、清空Eden，SurvivorFrom

然后清空Eden和SurvivorFrom区域中的对象，也即复制之后有交换，谁空谁是To区域

1. SurvivorTo区域和SurvivorFrom区域互换

最后，SurvivorTo和SurvivorFrom互换，原来SurvivorTo成为下一次GC时的SurvivorFron区域。部分对象会在From和To区域中复制来复制去，如此交换15次（由参数MaxTenuringThreshold决定，这个参数的默认值是15），如果最终还是存活，就存入到老年代中。

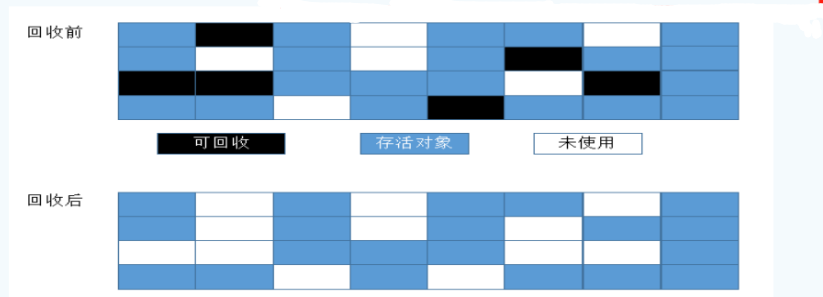
优缺点：优点，没有产出内存碎片。缺点，空间耗费。大对象复制起来耗费时间。

1. 标记清除

先标记，后清除。

优点：对象的标记，没有大面积复制，但是会产生内存碎片。

分为标记和清除两个阶段，先标记出需要回收的对象，然后统一回收这些对象，形如：



四、标记清除整理

原理：1、标记，与标记清除一样。2、压缩，再次扫描，并且往一端滑动存活对象。

缺点：移动对象需要成本。优点：没有内存碎片，可以利用bump

复制算法用于新生代，标记清除和标记清除整理用于养老区。



# 大厂面试

## 一、JVM垃圾回收的时候如何确定垃圾？是否知道什么是GC Roots?

什么是垃圾？ 内存中已经不再被使用到的空间就是垃圾。

要进行垃圾回收，如何判断一个对象是否可以被回收？

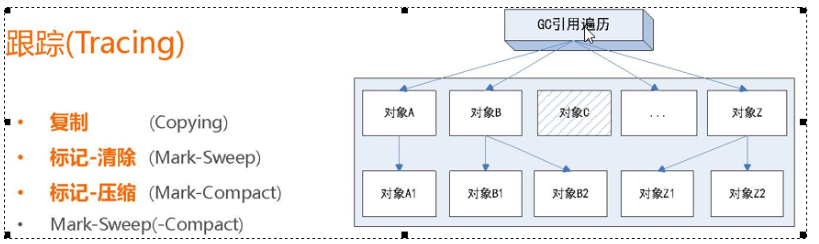
1、引用计数法：

Java中，引用和对象是有关联的。如果要操作对象，则必须用引用进行。

一个简单的办法就是通过引用计数来判断一个对象是否可以被回收，简单来说，给对象中添加一个引用计数器，每当有一个地方引用它，计数器值加1，没当有一个引用失效的时，计数器值减1。任何时刻计数器值为0的对象就是不可能再被使用，那么这个对象就是可回收对象。该算法存在但是目前无人使用，原因是无法解决循环引用的问题。

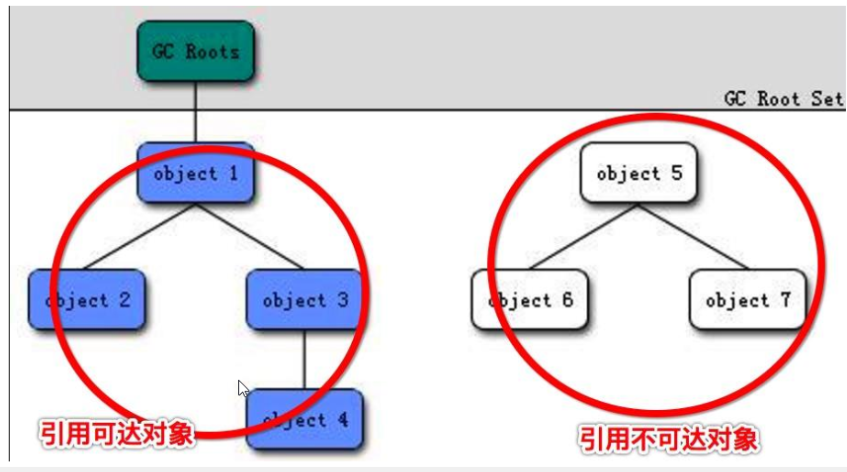
1. 枚举根节点做可达性分析（根搜索路径）

为了解决引用计数法的循环引用问题，Java使用了可达性分析办法。



所谓“GC Roots”或者说tracing GC的“跟集合” 就是**一组必须活跃的引用**。

基本思路就是通过一系列名为 “GC Roots”的对象作为起始点，从这个被称为DC Roots的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连接时，则说明此对象不可用。也即给定一个集合的引用作为跟出发，通过引用关系遍历对象图，能被遍历到的（可达到）对象就判定为存活，没有遍历到的就判定为死亡。



对象5、6、7没有从GC Root开始，所以他们会判断为引用不可达对象？

那些对象可以作为GC Root对象？总共是4种。

1、虚拟机栈（栈帧中局部变量区，也叫局部变量表）中引用的对象。

2、方法区中的类静态属性引用的对象。

3、方法区中常量引用的对象。

4、本地方法栈中JNI(Native方法)引用的对象。

Person p = new Person();

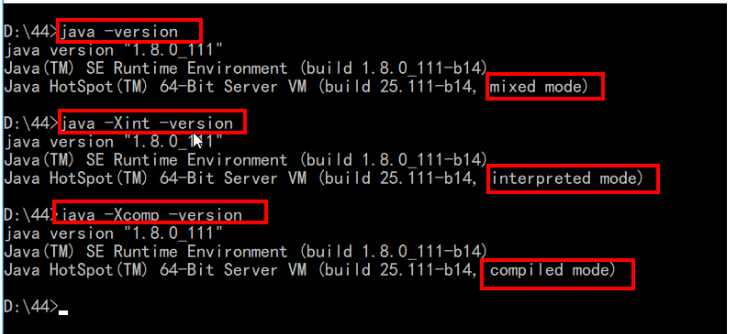
左边叫一个引用，右边称为一个实例对象。

## 二、JVM参数调优和参数配置，JVM系统默认值。

JVM的参数类型：标配参数，X参数，XX参数

标配参数：-version -help java -showversion

X参数：-Xint 解释执行 -Xcomp第一次使用就编译成本地代码 -Xmixed混合模式



XX参数：

**1、Boolean类型**

公式： -XX +或者- 某个属性值 +表示开启 -表示关闭

**如何查看一个正在运行中的Java程序，它的某个JVM参数是否开启？具体的值是多少？**



-XX: + PrintGCDeatils

-XX: +UseSerialGC

举例：是否打印GC的收集细节 jps -l 查看进程编号 jinfo -flag PrintGCDetails 1111进程

是否使用串行垃圾回收器

**2、KV设值类型**

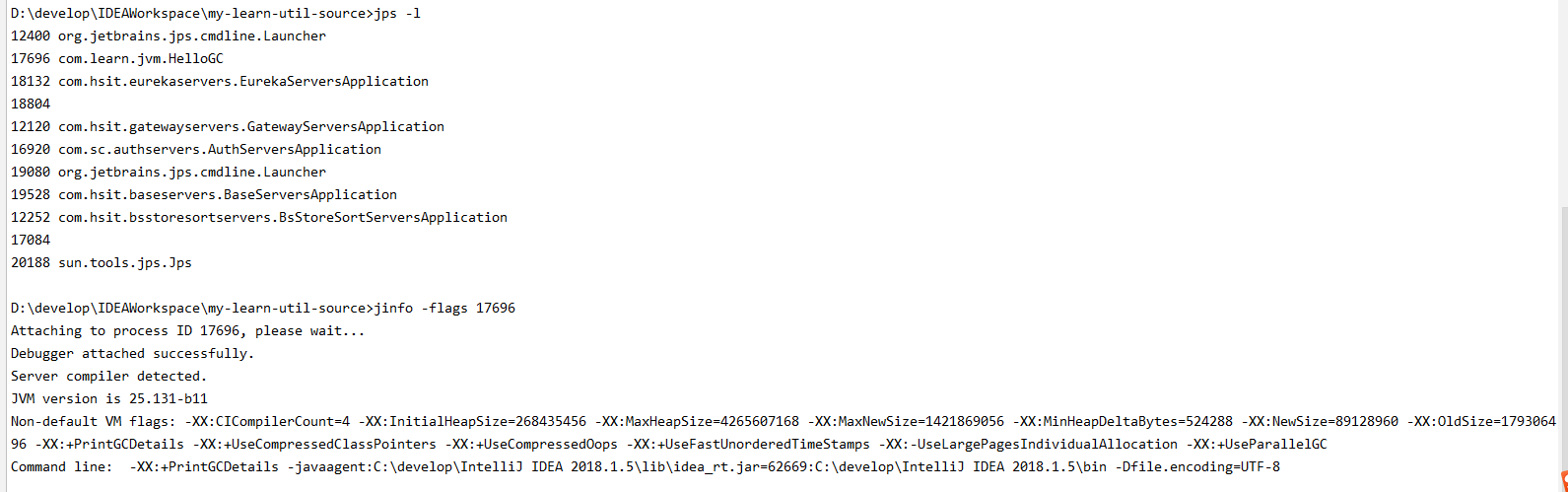
公式： -XX:属性值key = 属性值value

举例：-XX:MetaspaceSize=128m

-XX:MaxTenuringThreshould=15

**jinfo查看某个java进程所有jvm根据机器性能自带的参数**

**常见的jvm参数：<https://cloud.tencent.com/developer/article/1525732>**



Non-default VM flags :初始化加载，系统做的

Command line：人为加的

**另外两个参数 -Xms -Xmx**

-Xms等价于： -XX:InitialHeapSize

-Xmx等价于： -XX:MaxHeapSize

-Xms 默认值是系统内存的64分之1 -Xmx:默认是系统的4分之1

**-XX:MetaspaceSize**

**-XX:+PrintGCDetails**

### -XX:PrintFlagsInitial

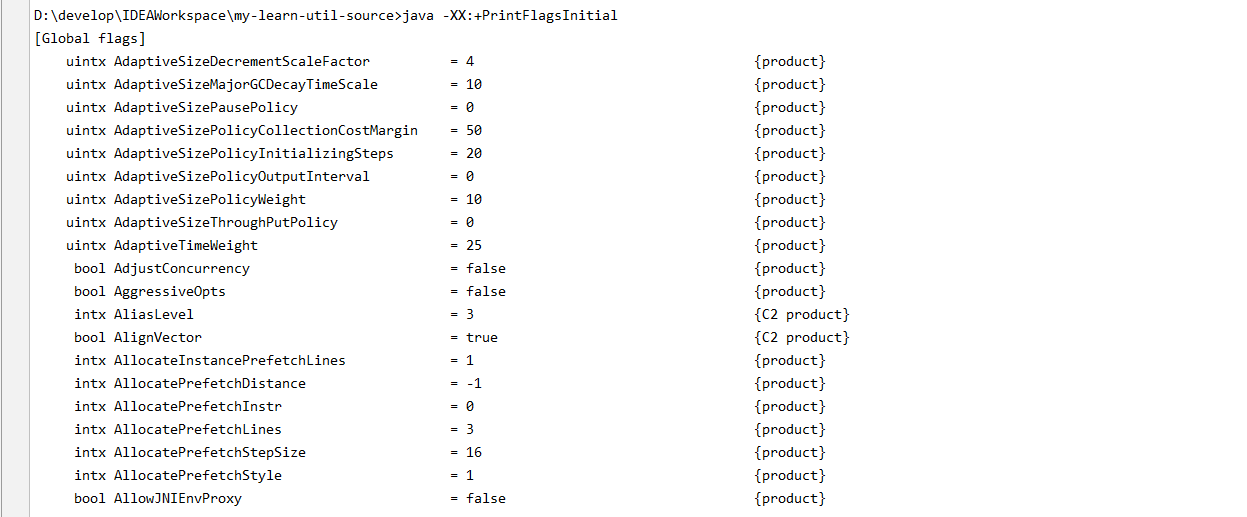
**盘点家底查看JVM默认值**

-XX:PrintFlagsInitial 主要是查看初始默认值，程序不需要启动

java -XX:+PrintFlagsInitial -version

java -XX:+PrintFlagsInitial

特别多



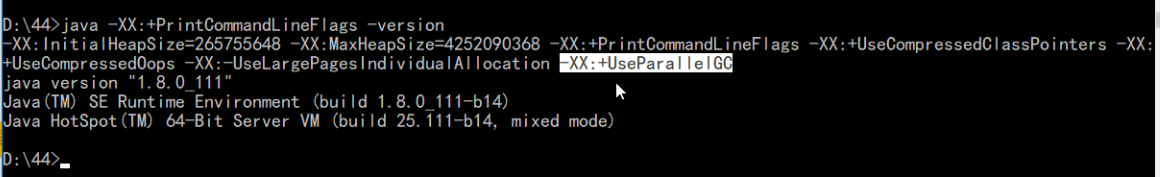
-XX:+PrintFlagsFinal主要查看修改更新

=表示没改过

:= 人为改过或者Jvm自己去修改过的

-XX:+PrintCommandLineFlags

最后一个是查看默认的垃圾回收器。



### -XX:SurvivorRatio

设置新生代中eden和 s0、s1区空间的比例，默认是8:1:1

默认：

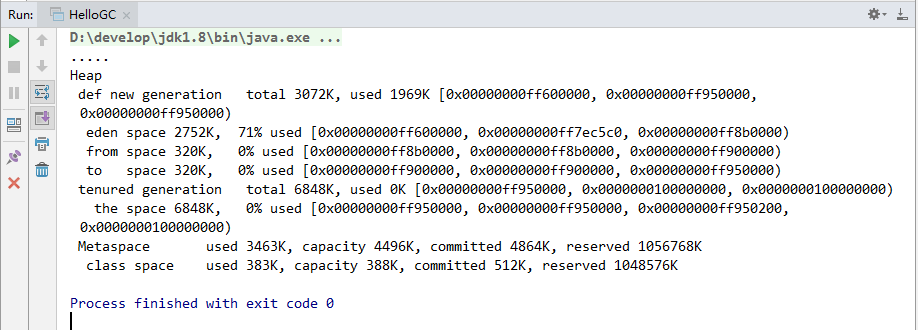
-XX:SurvivorRatio=8,Eden:S0:S1=8:1:1

-XX:SurvivorRatio=4,Eden:S0:S1=4:1:1

-XX:SurvivorRatio值就是设置eden区的比例占多少，s0和s1相同

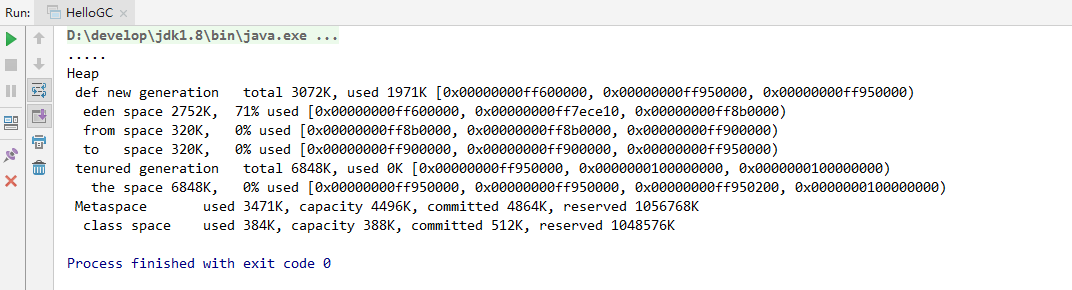
例子：

-XX:+PrintGCDetails -XX:+UseSerialGC -Xmx10m -Xmx10m

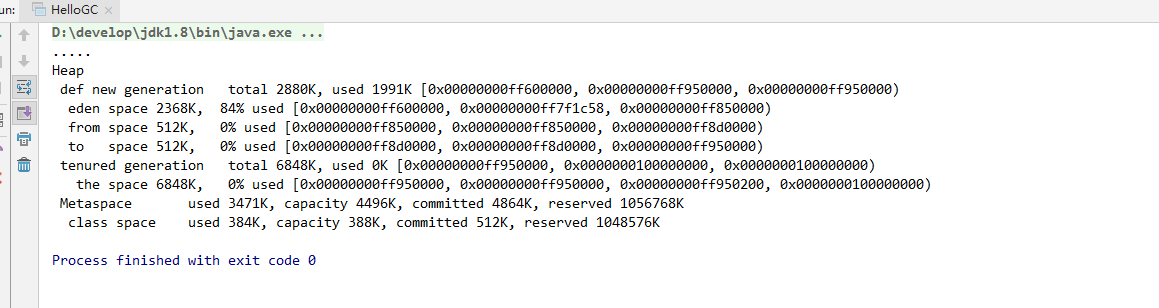


下面和上面是一样的：

-XX:+PrintGCDetails -XX:+UseSerialGC -Xmx10m -Xmx10m -XX:SurvivorRatio=8



-XX:+PrintGCDetails -XX:+UseSerialGC -Xmx10m -Xmx10m -XX:SurvivorRatio=4



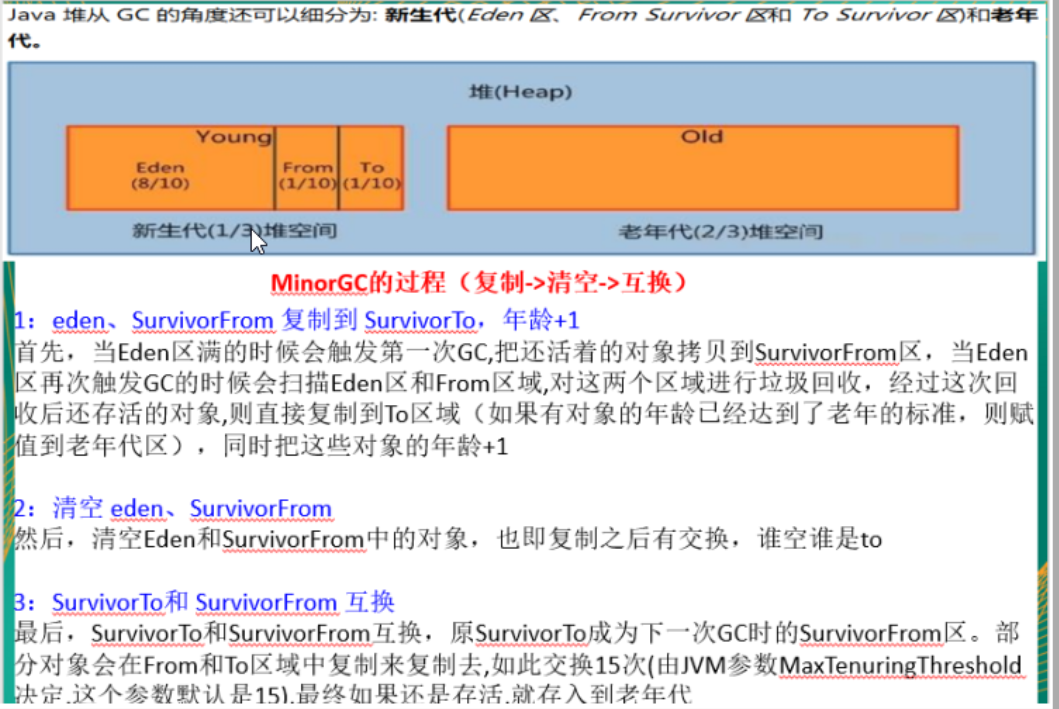
### -XX:NewRatio

配置年轻代与老年代在堆结构的占比

NewRatio值就是设置老年代的占比，剩下的1给新生代。

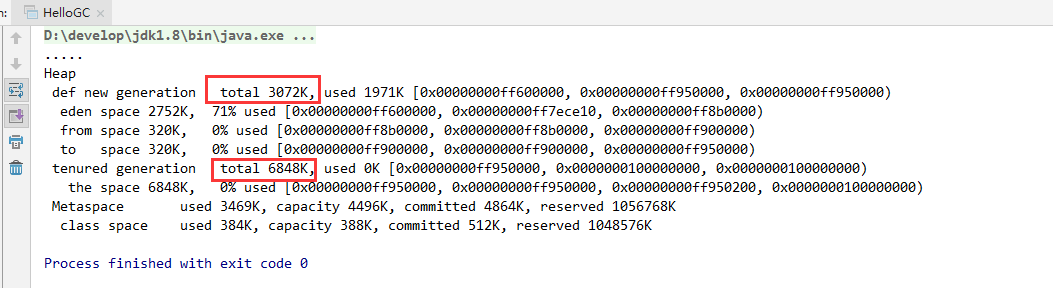
默认：-XX:NewRatio=2 新生代占1，老年代占2，年轻代占整个堆内存的1/3

假如设置：-XX:NewRatio=4 新生代占1，老年代占4，年轻代占整个堆内存的1/5

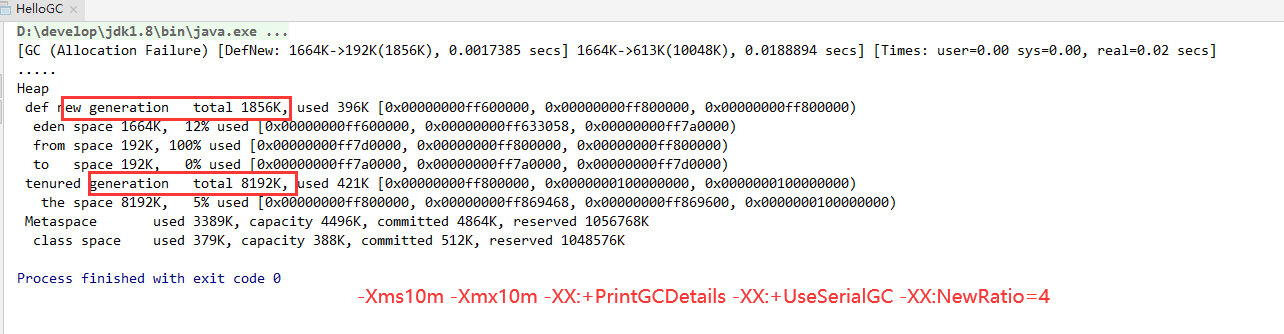


默认：

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseSerialGC -XX:NewRatio=2



-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseSerialGC -XX:NewRatio=4



### -XX:MaxTenuringThreshold

设置垃圾的最大年龄 ，默认是15次 也就是要逃过15次垃圾回收

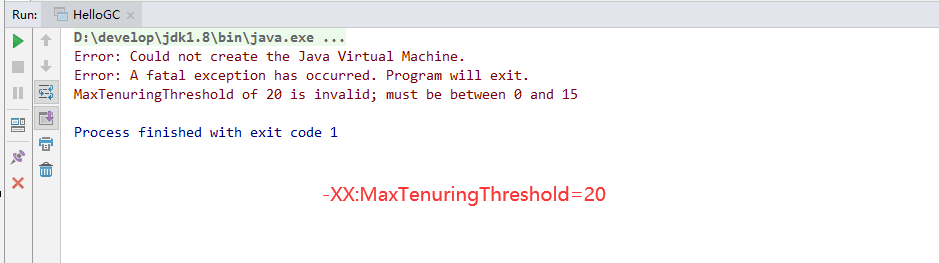
查看默认进入老年代的年龄，jinfo -flag MaxTenuringThreshold 进程编号

这个参数设置为0,则年轻代对象不经过Survivor区，直接进入老年代。对于老年代比较多的应用，可以提高效率。如果将这个值设置为一个比较大的值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代的存活时间，增加在年轻代即被回收的概率。



这个参数只能设置在0-15之间

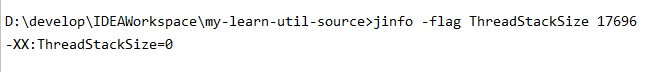
若 -XX:MaxTenuringThreshold=20



**-Xms**  初始内存大小，默认是1/64 等价于 -XX:InitialHeapSize

**-Xmx** 最大内存分配，默认是1/4 等价于 -XX:MaxHeapSize

**-Xss** 设置单个线程栈的大小，一般默认是512k - 1024k 等价于 -XX:ThreadStackSize



如果这个值是0，则代表使用的是系统默认的栈空间大小。

栈管运行，堆管存储。

**-Xmn** 设置新生区，年轻代的大小，一般使用默认值即可

**-XX:MetaspaceSize**  设置元空间的大小

元空间的本质和永久代类似，都是对JVM规范中方法区的实现，不过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，元空间的大小仅仅受到本地内存的限制。

-Xms10m -Xmx10m -XX:MetaspaceSize=1024m -XX:PrintFlagsFinal

**-XX:+PrintGCDetails**

**-XX:SurvivorRatio**

**-XX:NewRatio**

**-XX:MaxTenuringThreshould**

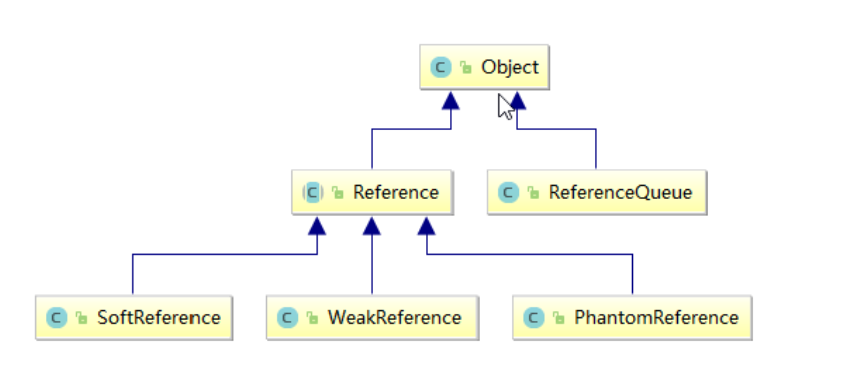
# 4、强引用、软引用、弱引用、虚引用分别是什么？

Book b = new Book();

左边在栈里面叫引用，右边在堆里面叫对象。

引用的整体架构：

java.lang.ref



## 强引用

当内存不足的时候，JVM开始垃圾回收，对于强引用对象，就算是出现了OOM也不会对该对象进行回收，死都不回收。

强引用是我们最常见的普通对象引用，只要还有强引用指向一个对象，就能表名对象还“活着”，垃圾回收器不会碰这种对象。在Java中最常见的就是强引用，把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被抢引用变量引用的时候，它处于可达状态，它是不可能被垃圾回收机制回收的，即时该对象以后永远都不会被用到，JVM也不会回收。因此，强引用是造成Java内存泄露的主要原因之一。

对于一个普通对象，如果没有其他的引用关系，只要超过了引用的作用域或者显示的将相应（强）引用赋值为null，一般就认为是可以被垃圾收集的了（当然具体回收的时机还要看垃圾收集的策略）。



## 软引用

强引用是死了都不收，软引用是内存足够的前提下不收，内存不够的时候会回收。来保证尽量不发生OOM

软引用是相对强引用弱化了一些的引用，需要用java.lang.ref.SoftReference类来实现，可以让对象豁免一些垃圾回收器。

当系统内存足够时，软引用对象不会被回收。

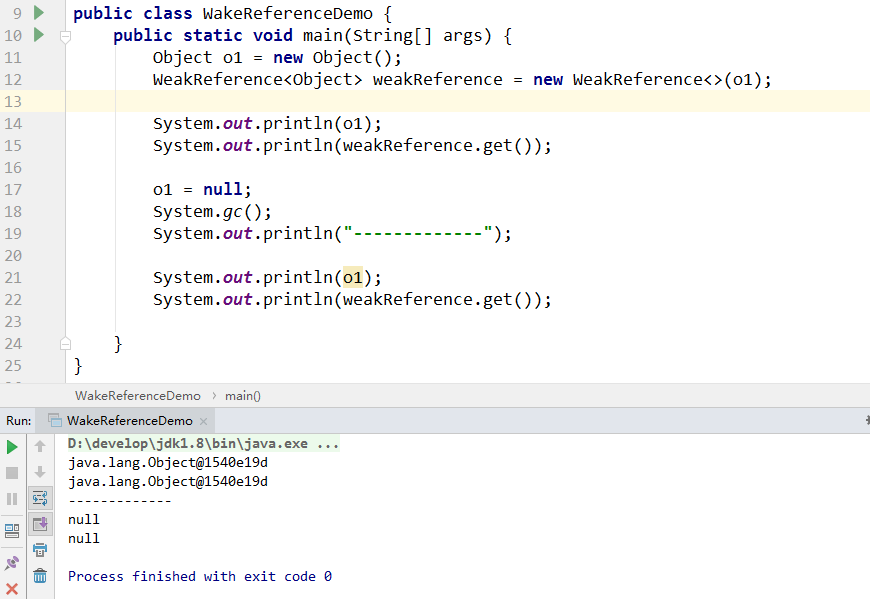
当系统内存不足时，软引用对选哪个会被回收。

软引用通常在对内存铭感的程序中，比如高速缓存就有应道软引用。

## 弱引用

不管内存是否充足，一律回收。

只要垃圾回收机制一运行，不管JVM的内存空间是否足够，都会回收该对象占用的内存。



**场景：读取大量图片**

**如果每次读取图片都要从硬盘读取则速度很慢。**

**如果一次性全部加载到内存中会爆出内存溢出。**

**使用软引用：**

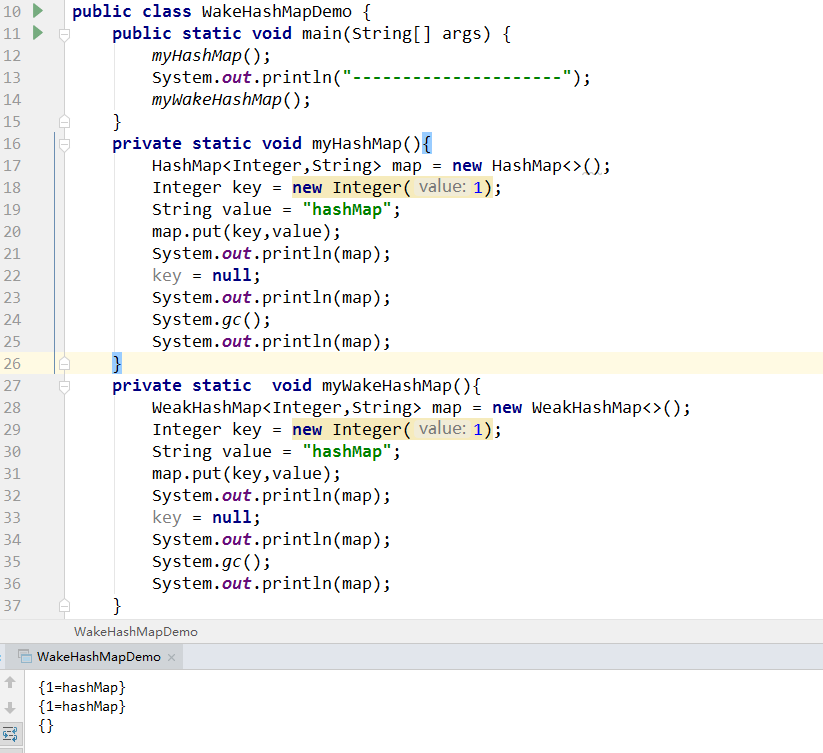
**用一个HashMap来保存图片路径和相应图片对象关联的软引用之间的映射关系，在内存不足的时候，JVM会自动回收这些缓存图片所占据的空间，从而有效的避免了OOM的问题。**

**Map<String,SoftReference<Bitmap>> imageCache = new HashMap<String,SoftReference<Bitmap>>();**

## WakeHashMap

缓存清空。

当kay被置空的时候，value自动被垃圾回收。



## 虚引用

也称为幽灵引用。

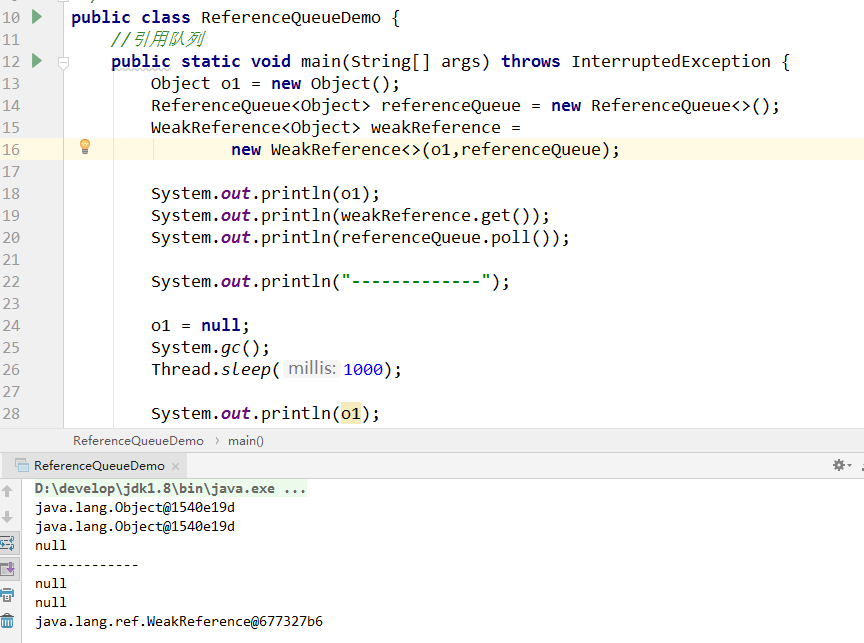
需要使用java.lang.ref.PhantomReference类来实现。

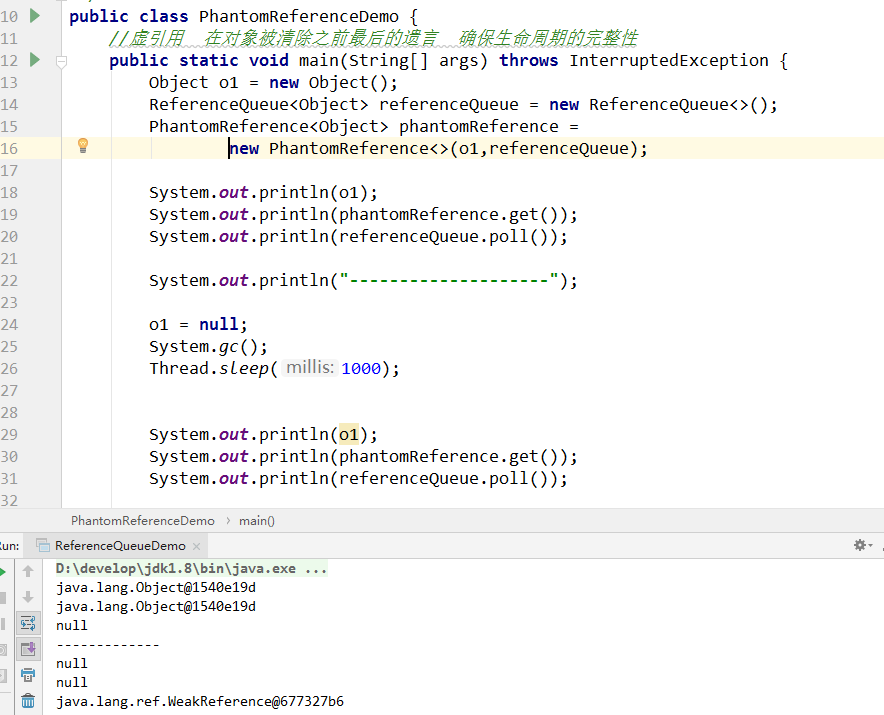
形同虚设，与其他几种引用都不太，虚引用不会决定对象的生命周期。如果一个对象仅支持虚引用，那么它就和没有任何引用引用，在任何时候都可能被垃圾回收器回收，它不能单独使用，也不能通过它访问对象，虚引用必须和引用队列联合使用。

虚引用的主要作用就是跟踪对象被垃圾回收的状态。仅仅是提供了一种确保对象被finalize以后，做某些事情的机制。

PhantomReference的get方法总是返回null，因此无法访问对应的引用对象。其意义在于说明一个对象已经进入finalization阶段，可以被gc回收，用来实现比finalization机制更灵活的回收操作。

换句话说，设置虚引用关联的唯一目的，就是在这个对象被回收器回收的时候收到一个系统通知或者后续添加进一步的处理。Java技术允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做的必要清理工作。



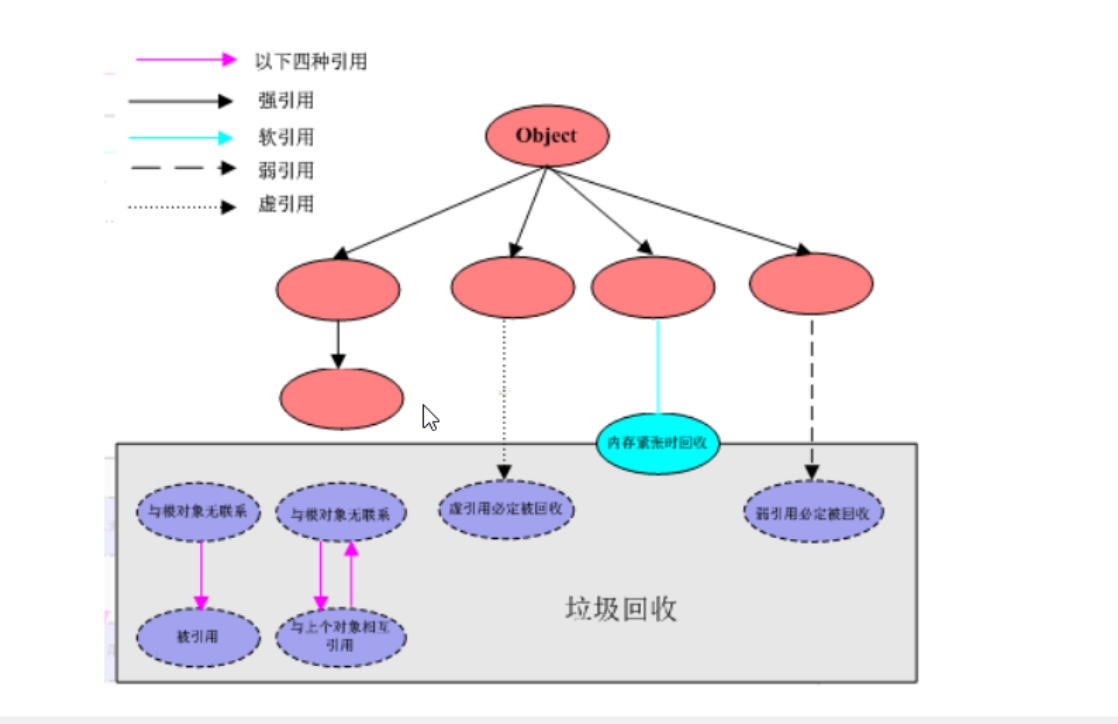


ReferenceQueue是用来配合引用工作的，没有ReferenceQueue一样可以运行.

创建引用的时候可以指定关联的队列，当GC释放对象内存的时候，会将引用加入到引用队列，如果程序发现某个虚引用已经被加入到引用队列，那么久可以在锁引用的对象内存被回收之前才去必要的行动，这相当于是一种通知机制。

当关联的引用队列中有数据的时候，意味着引用指向的堆内存中的对象被回收，通过这种方式，JVM允许我们在对象被销毁后，做一些想做的事情。

## GC Roots和四大引用小总结



# 5、谈谈你对OOM的认识

## java.lang.StackOverFlowError

栈内存溢出，递归调用容易导致栈内存溢出。栈空间被撑爆。

不想听别人啰嗦就自己把活干利索。

## java.lang.OutOfMemoryError :java heap sapce

堆内存不够了。

一直在new对象。

-Xms10m -Xmx10m

while(true){

str +=str + new Random().nexInt(1111);

}

byte[] byt = new byte[80\*1024\*1024];

新建一个超大对象。

## java.lang.OutOfMemoryError :GC overhead limit exceeded

GC过头导致超过了最大极限。

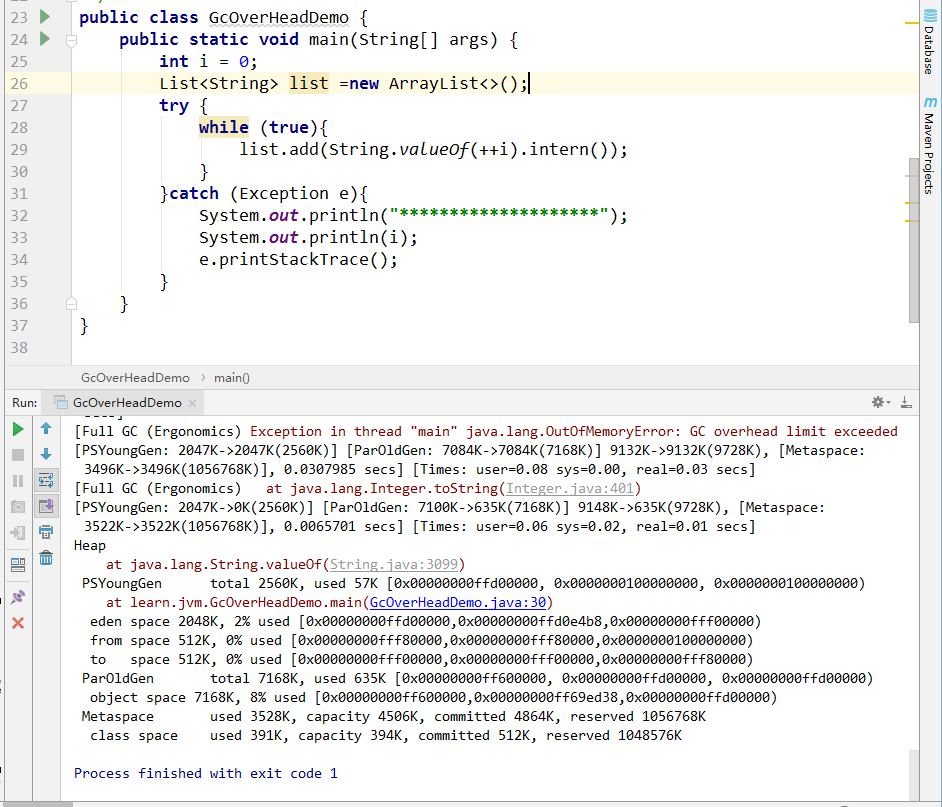
参数配置演示：

-Xms10m -Xmx10m -XX:maxDirectMemorySize=5m

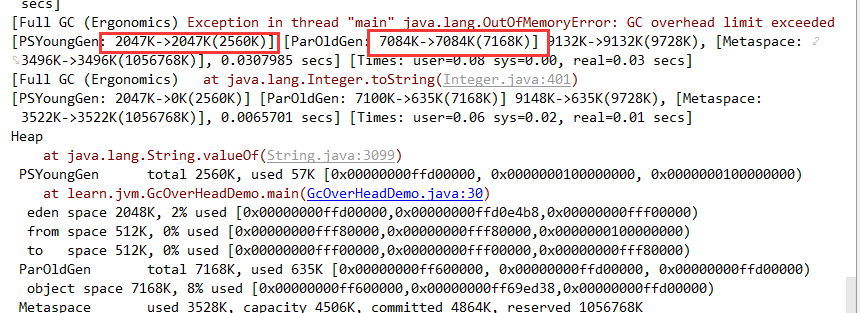
GC回收时间过长时会抛出OutOfMemoryError。过长的定义是：超出98%的时间用来做GC，并且回收了不到2%的堆内存，连续多次GC,都只回收了不到2%的极端情况下才会抛出。加入不抛出GC overhead limit 错误会发生什么情况呢？那就是GC清理的这些内存很快被再次填满，迫使GC再次执行，恶性循环。

现象就是CPU的使用率一直是100%，但是GC却没有任何的效果。

//Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded



一直在进行Full GC 收不收的效果是一样的。



## java.lang.OutOfMemoryError : Direct buffer memory

直接内存挂掉。

MetaSapce不在虚拟机中，而是使用本地内存。

这个异常的出现源于两种技术， netty，底层是一个NIO.

虚拟机参数演示配置：

-Xmx10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5m

导致原因：

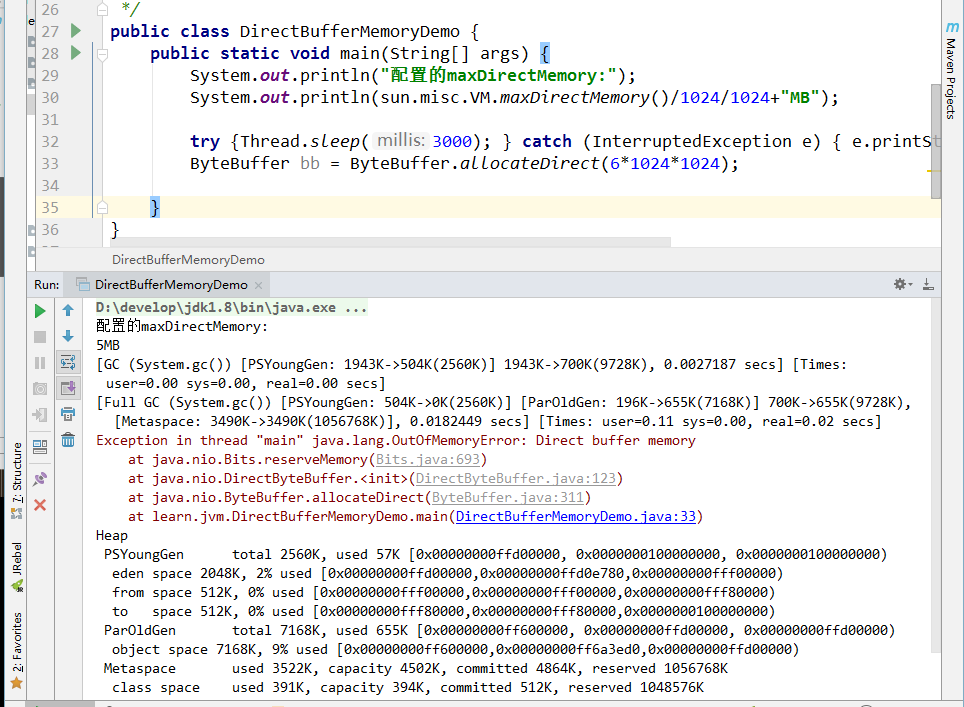
写NIO程序经常使用到ByteBuffer来读取或写入数据，这是一种基于通道Channel与缓冲区Buffer的IO方式。它可以使用Native函数库直接分配堆外面的内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作，这样能在一些场景中显著提升性能，因为避免了在Java堆和Native对中来回复制数据。

ByteBuffer.allocate(capablity)第一种方式是分配JVM堆内存，属于GC管辖范围，由于需要拷贝所以速度相对较慢。

ByteBuffer.allocateDirect(capability)第一种方式是分配OS内存，不属于GC管辖范畴，由于不需要内存拷贝所以速度相对较快。

但是如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行GC,DirectByteBuffer对象们就不会被回收，这时候堆内存充足，但是本地内存可能已经使用光了，再次尝试分配本地内存的时候就会出现OutOfMemoryError，程序直接崩溃。

以前的对象都是分配在JVM的堆内存中，有些对象如果是分配在了操作系统的本地内存中，GC不会回收本地内存中的对象，如果堆内存足够，GC也根本不会触发。本地内存用光的时候，程序也会爆出OOM。



## java.lang.OutOfMemoryError : unable to create new native thread

不能在创建本地线程了。

高并发请求服务器的时候，会出现以下异常：

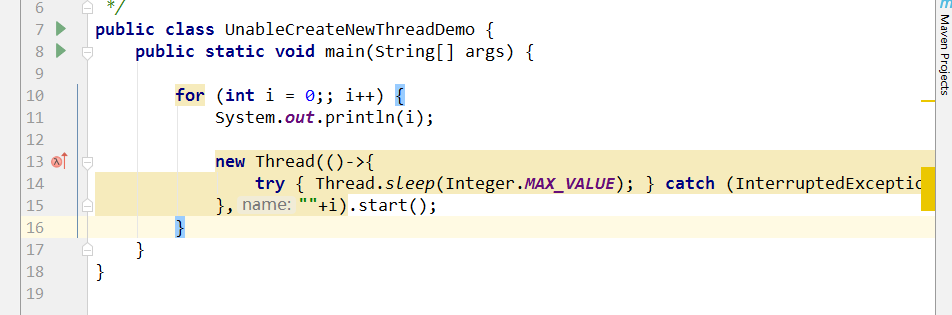
准确的讲该native thread异常与对应的平台有关。

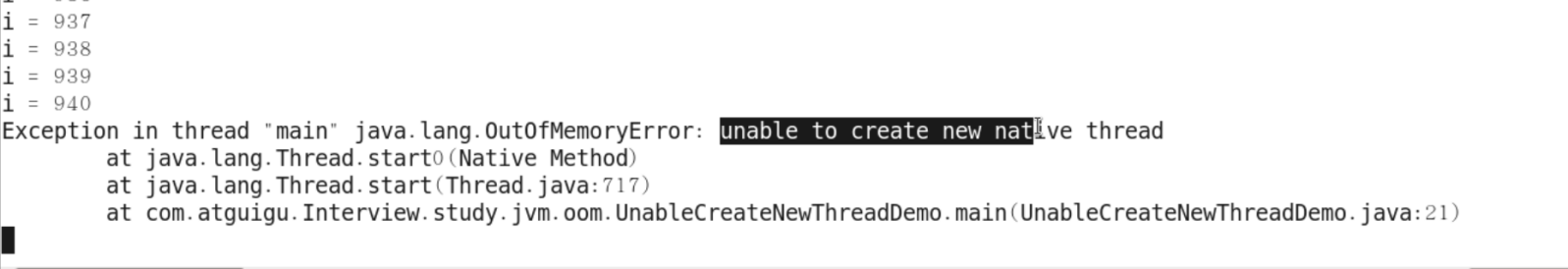
导致原因：

1. 你的应用创建了太多线程，一个应用进程创建多个线程，超过系统承载极限。
2. 你的服务器部允许你的应用程序创建这么多线程，Linux系统默认允许单个进程可以创建的线程数是1024个，你的应用创建超过这个数量，就会报异常。

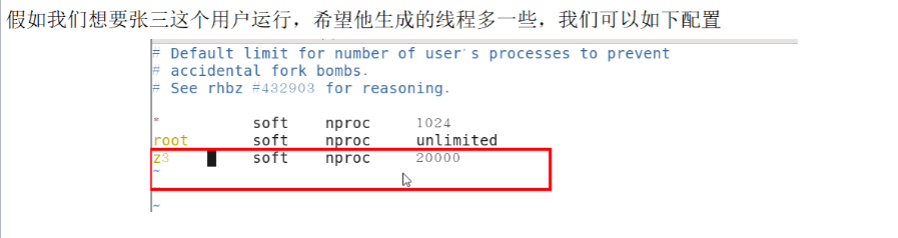
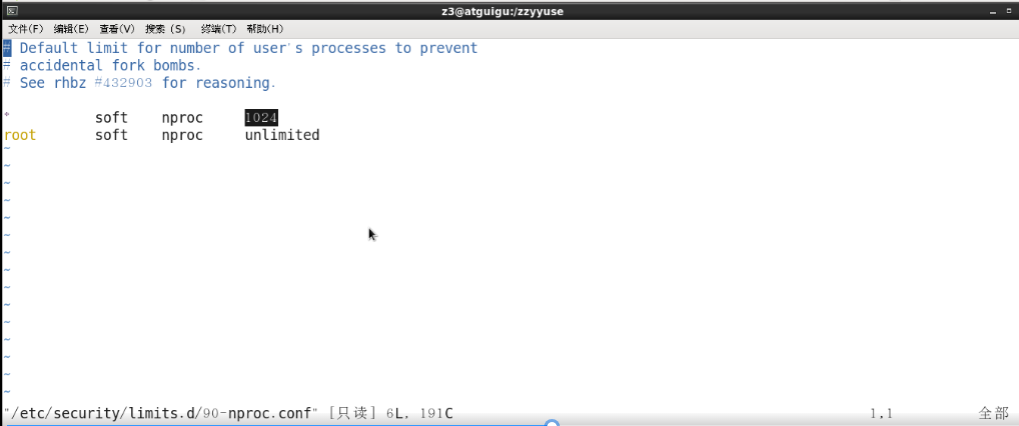
解决办法：

1. 想办法降低应用程序创建线程的数量，分析应用是否真的有必要创建这么多线程，如果不是，改代码将线程数降到最低。
2. 对于有的应用来说，确实需要创建很多线程，远远超过默认的1024个，可以修改Linux服务器配置，扩大Linux默认限制。









## java.lang.OutOfMemoryError : Metaspace

JVM参数演示：

-XX:MetaspaceSize=8m -XX:MaxMetaspaceSize=8m

Java8之后使用元空间替代了永久代。

Metaspace是方法区在HotSpot中的实现，它与持久代的最大区别在于，MetaSpace并不是在虚拟机内存中而是使用本地内存，也即在java8中，classe metadata(the virtual machines internal presentation of Java class)被存储在叫做metaspace的Native Memory

存放的信息如下：

虚拟机加载的类信息。

常量池

静态变量

即时编译后的代码

模拟metaspace空间溢出，不断生成类往元空间灌，类占据的空间总是会超过metaspace指定的空间大小。



# GC垃圾回收算法和垃圾收集器的关系？分别是什么。

垃圾回收算法：引用计数，复制拷贝，标记清除，标记整理。

复制一般是新生代，标清和标整更多的是用在老年代。

GC算法（引用计数/复制/标清/标整）是内存回收的方法论，垃圾收集器就是算法的落地实现。

因为目前为止还没有完美的收集器出现，更是没有万能的收集器，只是针对具体应用，使用最合适的收集器，进行分代收集。

4种主要的垃圾收集器：

Serial 串行回收

Parallel 并行回收

CMS 并发标记清除

G1

## Serial 串行垃圾回收器

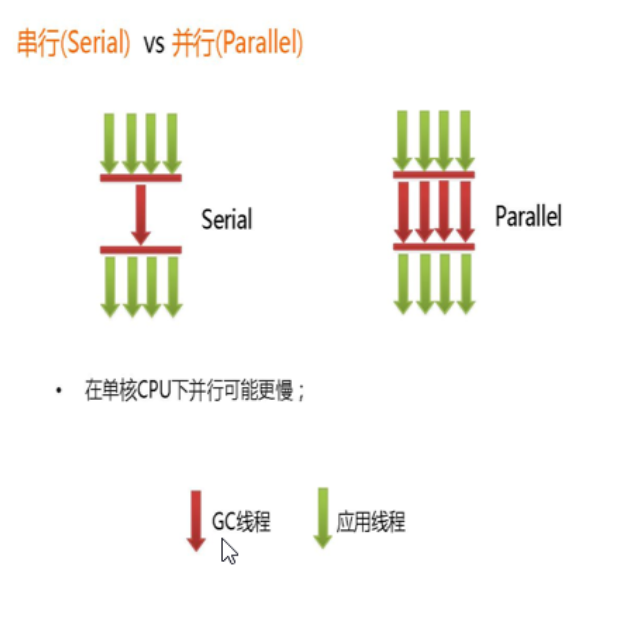
它为单线程环境设计并且只有一个线程进行垃圾回收，会暂停所有的用户线程，所以不适合服务器环境。

进行回收工作测线程只有一个。

## Parallel 并行垃圾回收器

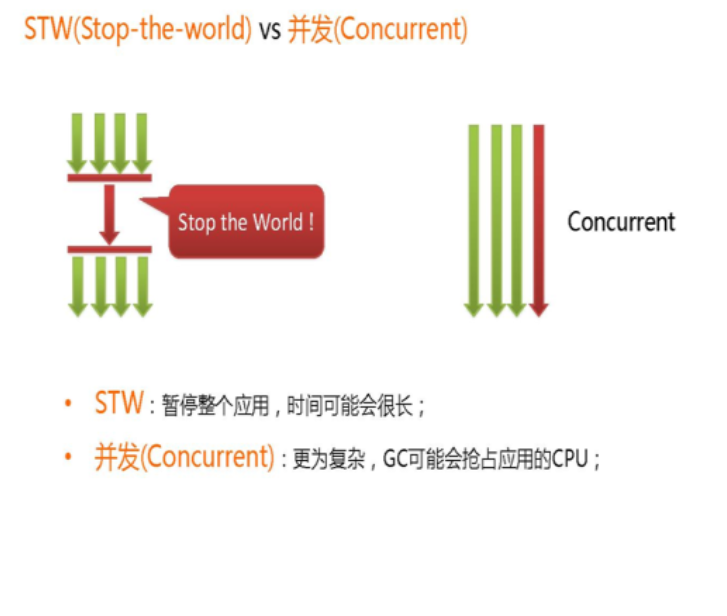
多个垃圾收集线程并行工作，此时用户线程是暂停的，适用于科学计算/大数据处理首台处理等弱交互场景。

进行回收工作测线程有多个。



## CMS并发标记清除垃圾回收器

用户线程和垃圾收集线程同时执行（不一定是并行，可能是交替执行）不需要停顿用户线程，互联网公司做用它，适用于对响应时间有要求的场景。



## G1 garbage first

G1垃圾回收器将堆内存分割成不同的区域然后并发的对其进行垃圾回收。

# 怎么查看服务器默认的收集器是哪个？生产上如何配置垃圾收集器？

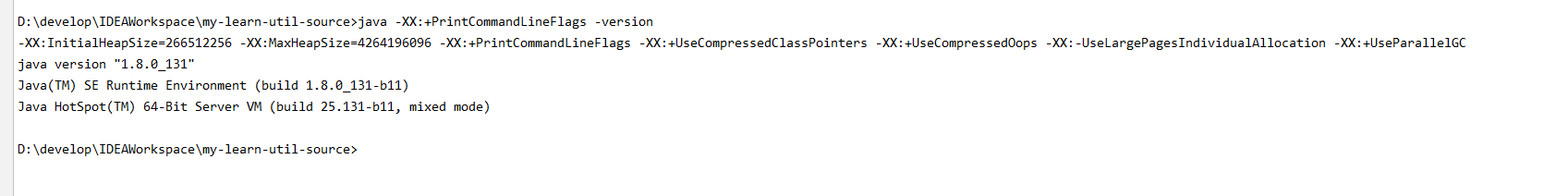
7种垃圾回收

## 默认的垃圾收集器

java8用默认用的是并行垃圾回收器

java -XX:+PrintCommandLineFlags -version

-XX:+UseParallelGC



JVM默认的垃圾收集器有哪些？

JavaGC回收的类型主要有以下几种：

UseSerialGC，UseParallelGC，UseConcMarkSweepGC，UseParNewGC，UseParallelOldGC，UseG1GC

还有一种UseSerialOldGC。已经不常用了，所以没有。

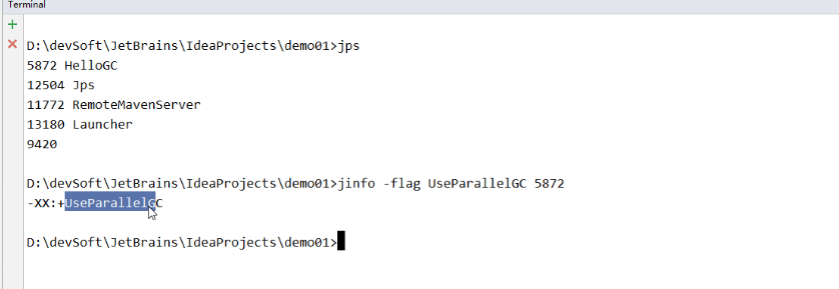
演示：

设置了JVM参数

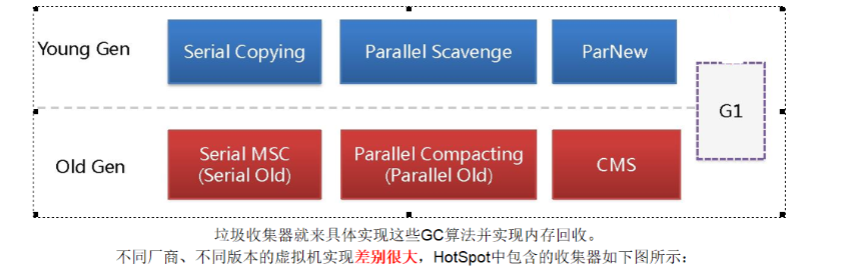
-XX:+UseSerialGC

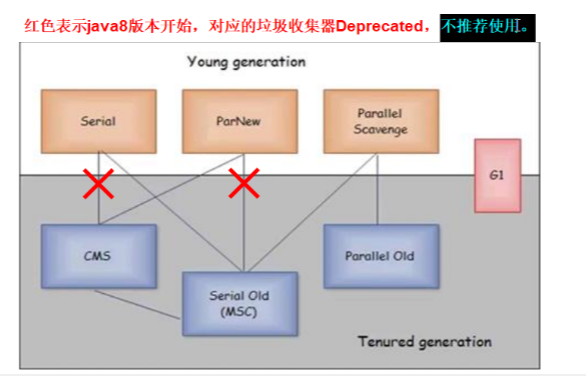


没有设置：



## 垃圾收集器





部分参数预先说明

DefNew Default New Generation 默认新生代使用的垃圾回收期

Tenured Old 老年代使用的垃圾回收器

ParNew Parallel New Generation 新生代使用并行回收

PSYoungGen Parallel Scavenge 新生区使用并行垃圾回收器

ParOldGen Parallel Old Generation 老年代使用并行垃圾回收器

## JVM的Server/Client模式是什么意思？

适用范围：

只需要掌握Server模式即可，Client模式基本不会用。

操作系统：

32位Windows系统 不论硬件如何默认都使用Client的JVM模式

32位其他操作系统，2GB内存同时有2个cpu以上用Server模式，低于改配置还是Client模式

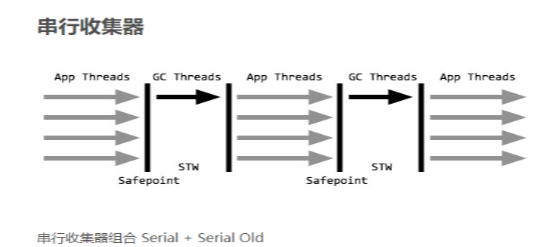
64位 only server模式

## 1、Serial收集器 新生代老年代都是串行

串行GC Serial / Serial Copying

串行收集器：Serial收集器

一个单线程的收集器，在进行垃圾收集的时候，必须暂停其他所有的工作线程直到它收集结束。



串行收集器是最古老，最稳定、效率最该的收集器，只使用一个线程去回收，但它在收集过程中可能会产生较长的停顿（Stop-The-World 状态） 虽然在收集垃圾过程中需要暂停其他的工作线程，但是它简单高效，对于限定单个CPU环境来说，没有线程交互的开销可以获得最高的单线程垃圾收集效率。因此Serial垃圾收集器依然是Java虚拟机运行在Client模式下默认的新生代垃圾回收器。

对应的JVM参数是： -XX:+UseSerialGC

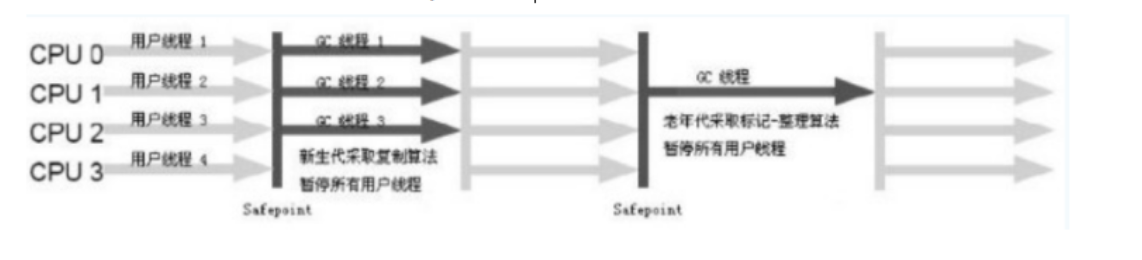
开启之后会使用Serial(Young区) + SerialOld（Old区域） 的收集器组合

表示：新生代、老年代都会使用串行回收器收集，新生代使用复制算法，老年代使用标记整理算法。

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseSerialGC

## 2、ParNew 收集器 新生代并行 老年代串行

使用多线程进行垃圾回收，在垃圾收集的时候，会stop-the-world 暂停当代其他所有工作线程直到收集结束。



ParNew收集器其实就是Serial收集器新生代的并行多线程版本，最常见的应用场景是配合老年代CMS GC工作，其余的行为和Serial收集器完全是一样的，ParNew垃圾收集器在垃圾收集的过程中同样也要暂停所有其他的工作线程。它是很多java虚拟机运行在Server模式下新生代的默认垃圾收集器。

常用参数：

-XX:+UseParNewGC 启用ParNew收集器，只影响新生代的收集，不影响老年代。

开启上述参数之后，会使用ParNew(Young区)+Serial Old的收集器组合，新生代用复制算法，老年代用标记整理算法。

但是：ParNew + Tenured这样的搭配，在java8中已经不再推荐，Java HotSpot虚拟机会报一个warning:

Using the ParNew young collection with the Serial conllection is deprecated and will likely be removed in future release

备注：启动多少个回收线程。

-XX:ParallelGCThreads 限制线程数量，默认开启和CPU数目相同的线程数

-Xms10m -Xmx10m -XX:PrintGCDetails -XX:+UseParNewGC

## Paraller GC 默认 新生代老年代都并行

使用复制算法，串行收集器在新生代和老年代并行收集。俗称，吞吐量优先收集器。

重点关注的是：

可控制的吞吐量（Thought=运行时用户代码时间/（运行时用户代码时间+垃圾收集时间）），也即比如程序运行100分钟，垃圾收集1分钟，吞吐量即时99%。高吞吐量意味着高效利用CPU时间。它多用于在后台运算而不需要太多交互的任务。

除此之外。自适应调节策略也是ParallelScavenge收集器与ParNew收集器的一个重要区别。

（自适应调节策略：虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最适合的停顿时间-XX:MaxGCPauseMillis）或者最大吞吐量。

常用的JVM参数：

-XX:UseParallelGC或者-XX:UseParallelOldGC 互相激活 使用ParallelScanvenge收集器

该参数开启之后：新生代使用的是复制算法。老年代使用标记-整理算法。

-XX:ParallelGCThreads = 数字N 表示启动多少个GC线程。

CPU>8 N = 5/8

cpu<8 N = 实际个数

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseParallelGC

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseParallelOldGC

## 4、ParallelOld 收集器 老年代并行

Parallel Old正是为了在老年代同样提供吞吐量优先的垃圾收集器，如果系统对吞吐量要求比较高，JDK1.8之后可以优先考虑新生代Parallel Scavenge和老年代Parallel Old收集器的搭配策略。

JVM常用参数：

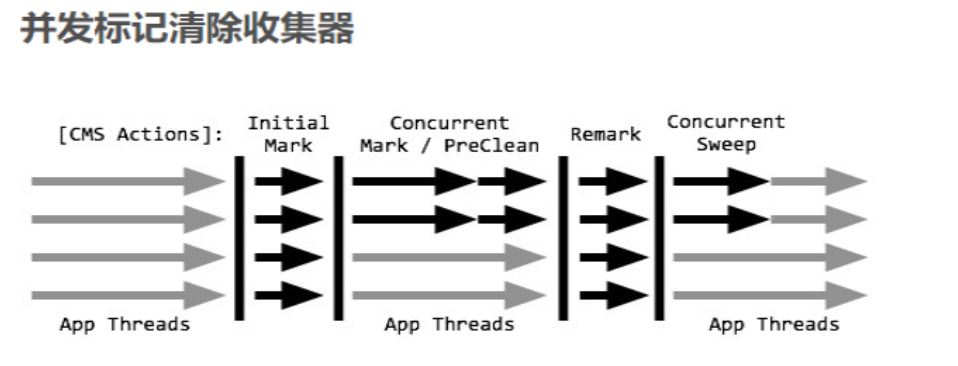
-XX:+UseParallelOldGC 改参数设置后，新生代Parallel + 老年代Parallel Old

## 5、CMS收集器

CMS收集器（Concurrent Mark Sweep 并发标记清除） 是一种获取最短回收停顿时间为目标的收集器。

适用于在互联网站或者B/S服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。CMS非常适合堆内存大，CPU核数多的服务器端应用上，也是G1出现之前大型应用的首选收集器。

ConcurrentMarkSweep并发标记清除，并发收集低停顿，与用户线程一起执行。



开启该收集器的JVM参数 -XX:UseConcMarkSweepGC 开启该参数后会自动将 -XX:+UseParNewGC打开，使用的是 ParNew(Young区) + Serial Old 的收集器组合，Serial Old 将作为 CMS出租时的后备收集器

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC

有四步过程：

**初始标记**： CMS initial mark 会停 只是标记一下GC Root能直接关联的对象，速度很快，需要暂停所有工作线程。

**并发标记**：CMS concurrent mark 和用户线程一起 不停 进行GC Root的跟踪过程，和用户线程一起工作，不需要暂停工作线程。主要标记过程，标记全部对象。

**重新标记**： CMS remark 会停 修正在并发标记期间，因为用户程序继续运行而导致标记产生变动的一部分对象标记的记录，仍然需要暂停所有的工作线程。由于并发标记时，用户线程仍然在运行，因此在正式清除前，再做修正。

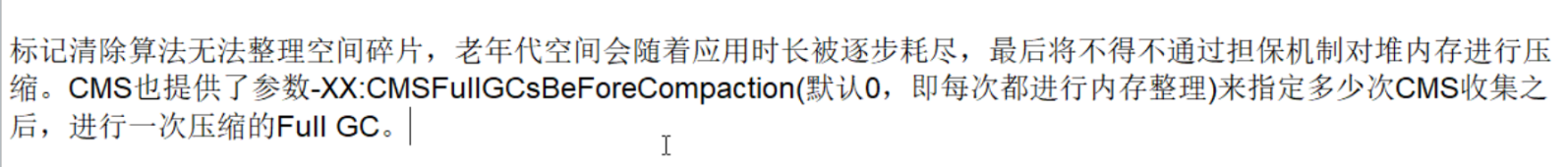
**并发清除**：CMS concurrnet sweep 和用户线程一起 清除GC Root不可达对象，和用户线程一起工作，不需要暂停工作线程。基于标记结果，直接清理对象。由于耗时最长的并发标记和清除过程中，垃圾收集线程可以和用户在一起并发工作。所以总体上来说。CMS收集器的内存回收和用户线程是一起并发的执行。

优缺点：

优：并发收集低停顿

缺：并发执行，CPU资源压力大

采用的标记清除算法会导致大量碎片



## 6、Serial Old

老年区 单线程 使用标记整理算法。

## 如何选择垃圾收集器？

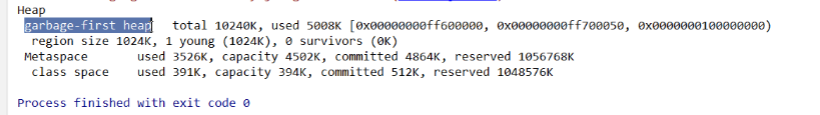




# G1垃圾收集器是什么？

参数演示配置：

-Xms10m -Xmx10m -XX:+PrintGCDetails -XX:+PrintCommandLineFlags -XX:+UseG1GC



## 以前收集器的特点

年轻代和老年代是各自独立并且连续的内存块；

年轻代收集使用单eden + S0 + S进行复制算法；

老年代收集必须扫描整个老年代区域；

都是尽可能少而快速地执行GC为设计原则。

## G1收集器是什么？

G1 Garbage-First 收集器，是一款面向服务端应用的收集器。

应用在多处理器和大容量内存环境中，在实现高吞吐量的同事，尽可能的满足垃圾收集器暂停时间的要求。它的特性：

像CMS收集器一样，能与应用程序线程并发执行；

整理空闲空间更快；

需要更多的时间来预测GC停顿时间；

不希望牺牲大量的吞吐性能；

不需要更大的Java Heap。

G1收集器的目标是替代CMS收集器，比CMS优秀的地方在于：

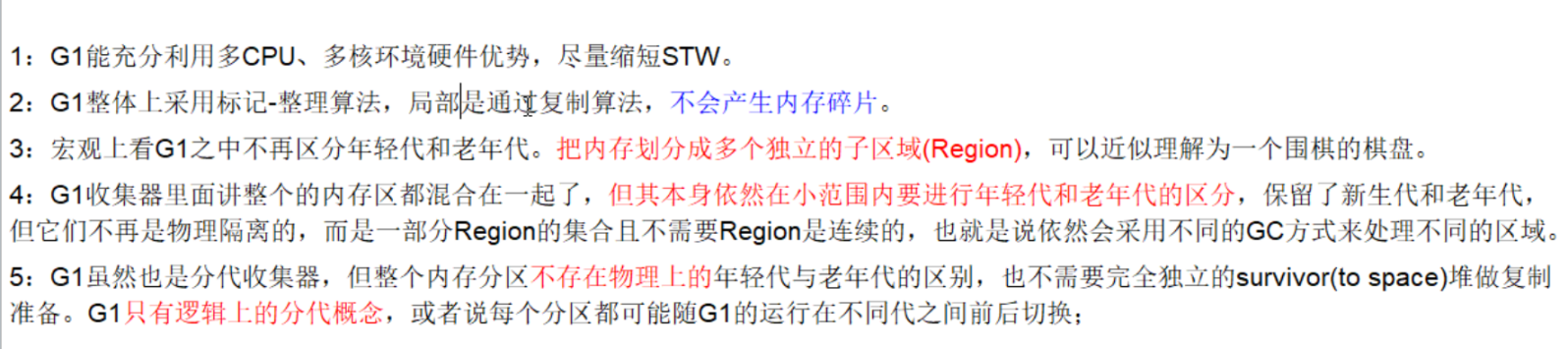
G1是一个有整理内存过程的垃圾收集器，不会产生很多内存碎片，G1的Stop the World 更可控，G1在停顿时间上添加了预测机制，用户可以指定期望停顿时间。

CMS垃圾收集器虽然减少了暂停应用程序的运行时间，但是还是存在着内存碎片问题。于是，为了去除内存碎片问题，同时又保留CMS垃圾收集器低暂停时间的有点，Java7发布了一个新的垃圾收集器，G1。

G1是在2012年才在jdk1.7u4中可用。Oracle官方计划在jdk9中将G1变成默认的垃圾收集器替代CMS。它是一款面向服务端应用的收集器，主要应用在多CPU和大内存的服务器环境中，极大的减少垃圾收集的停顿时间，全面提升服务器性能，逐步替换java8以前的CMS收集器。

主要改变是Eden，Survivoe 和 Tenured等内存区域不再是连续的了，而是变成了一个个大小一样的region , 每个region从1M到32M不等，一个region可能属于Eden，Surivivor或者Tenured内存区域。

## G1收集器的特点



## G1收集器的底层原理

### 化整为零

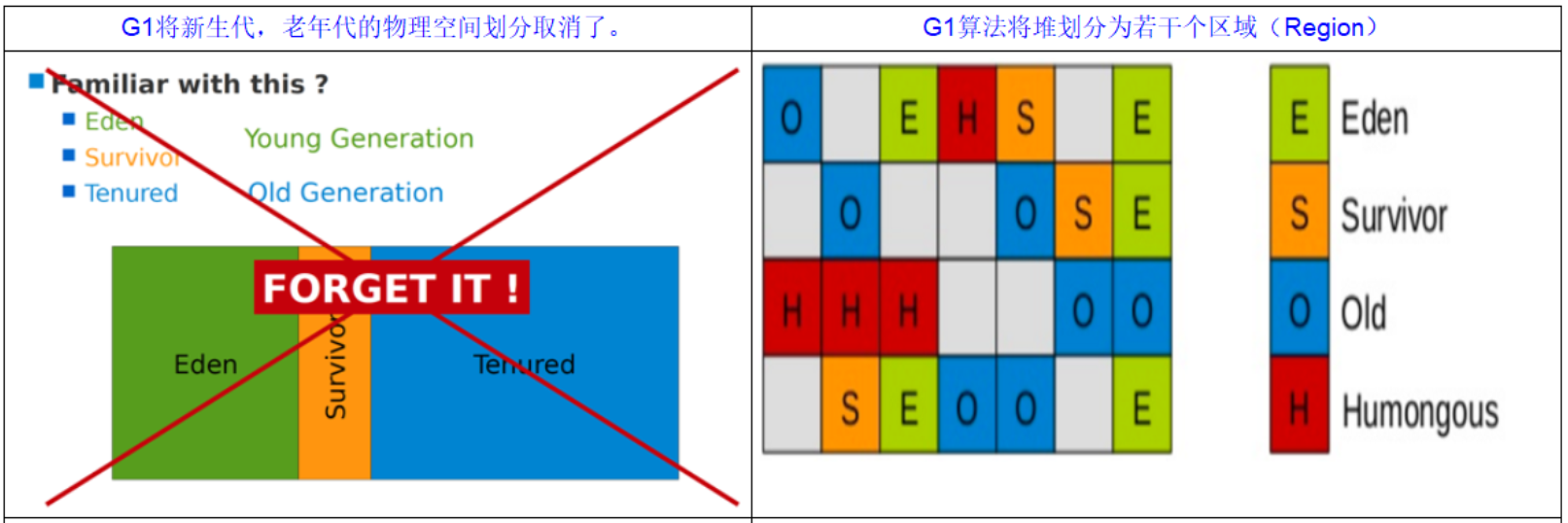
Region [区域] 最大的好处就是化整为零，避免全部内存扫描，只需要按照区域来进行扫描即可。

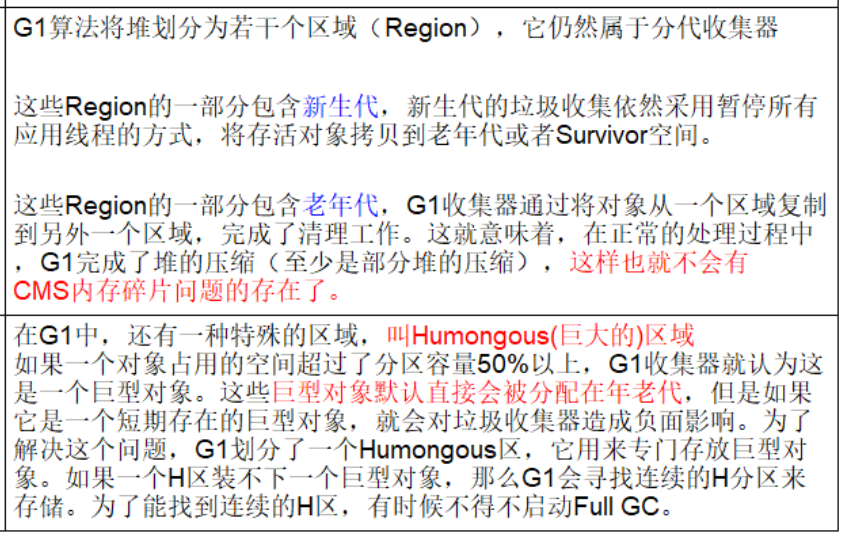
区域化内存划片Region，整体编了一些不连续的内存区域，避免了全内存臊面操作。核心思想是整个堆内存分成大小相同的子区域，在JVM启动的时候会自动设置这些子区域的大小，在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可，每个分区也不会固定地位某个代服务，可以按需在年轻代和老年代之间切换。启动时可以通过参数 -XX:G1HeapRegionSize=n 可以指定分区大小（1MB-32MB之间，并且必须是2的幂），默认将整堆划分为2048个分区。

大小范围在1MB-32MB之间，最多能设置2048个区域，也即能够支持的最大内存为：

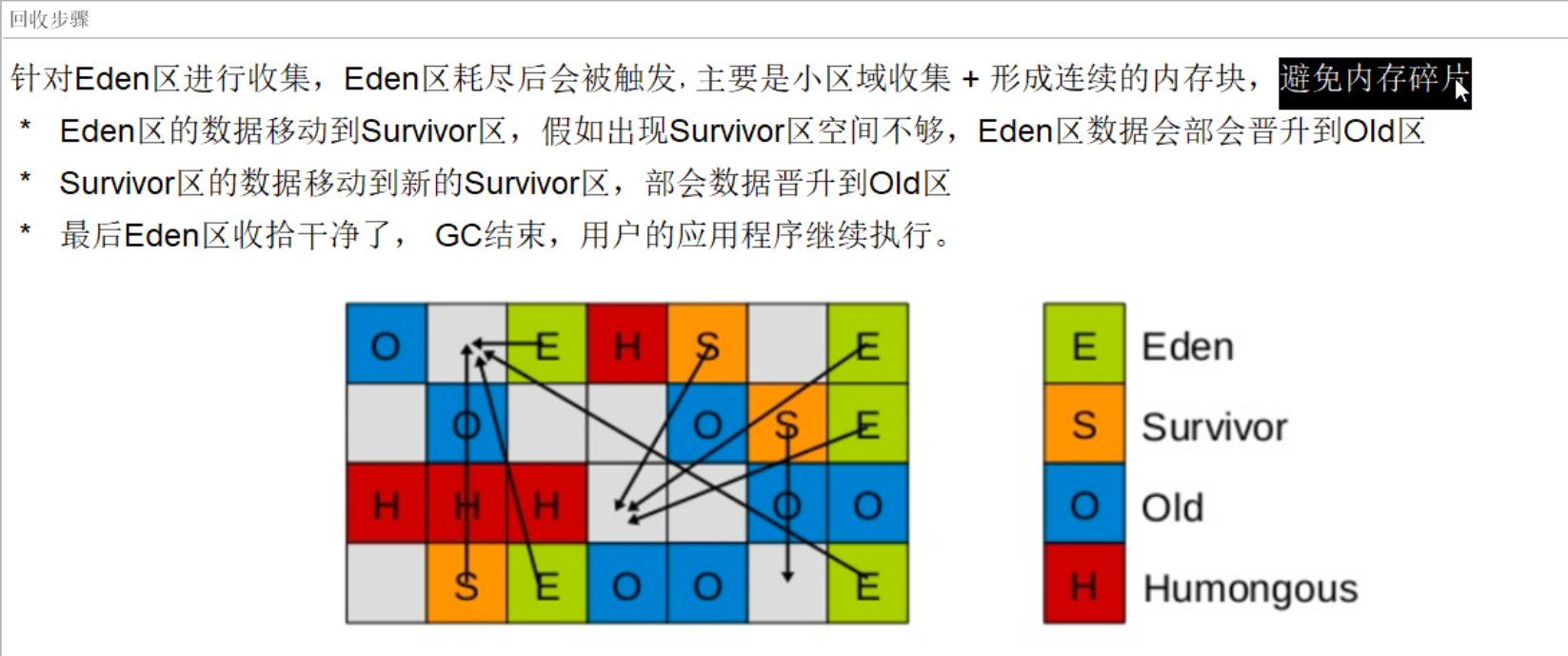
32MB \* 2048 = 64GB

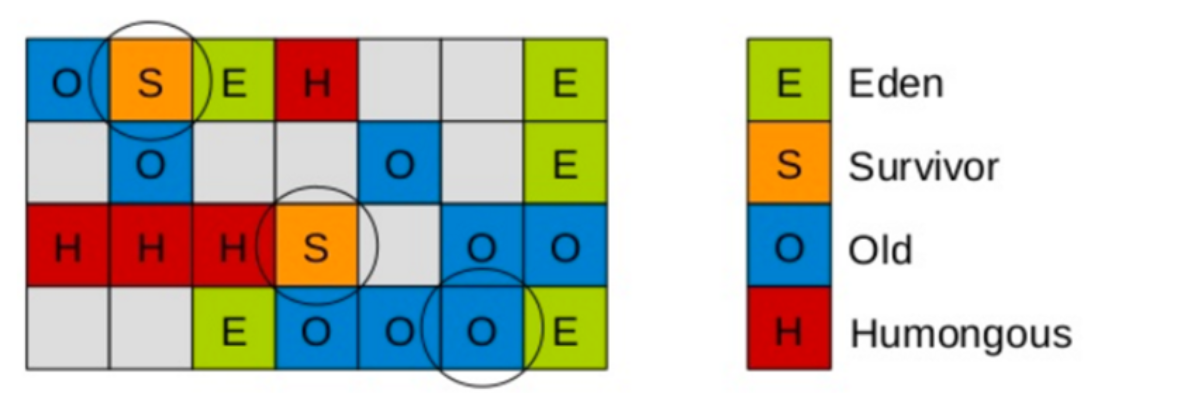
G1将新生代、老年代的物理空间划分取消了。





### 回收步骤







## 常用配置参数

-XX:+UseG1GC

-XX:G1HeapRegionSize=n 设置G1区域的大小。值是2的幂，范围是1M到32M。目标是根据最小的Java堆大小划分出约2048个区域。

-XX:MaxGCPauseMillis=n 设置GC停顿时间，这个是软目标，JVM尽可能停顿小于这个时间。

-XX:InitiatingHeapOccupancyPercent=n 堆占用了多少之后就触发GC,默认是45。

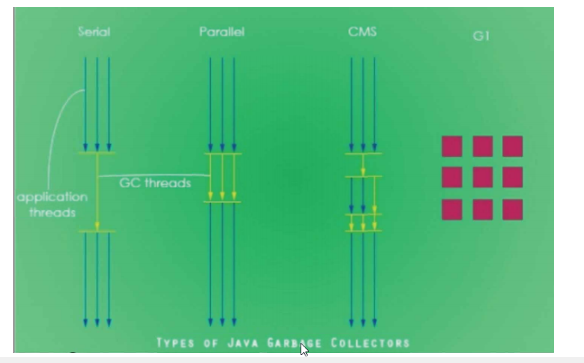
-XX:ConcGCThreads=n 并发GC使用的线程数。

-XX:G1ReservePa+ercent=n 设置作为空闲空间的预留内存百分比，以降低目标空间溢出的风险，默认值是10% 。

## 和CMS相比的优势

G1没有内存碎片。

G1可以精确控制停顿。把整个堆划分为多个固定大小的区域，每次根据运行停顿的时间去收集垃圾最多的区域。



## JVM + GC + SpringBoot +微服务的生产部署和参数调优

实际工作中，如何结合springboot进行调优？

1. 开发微服务工程
2. maven进行clean 和 package
3. 要求微服务启动的时候，同时配置JVM和GC的调优参数
4. java -jar xxxx.war

公式： java -server jvm参数 -jar 上面jar/war包的名字

举例： java -server -Xms1024m -Xmx1024m -XX:+UseG1GC -jar springboot2019-1.0-SNAPSHOT.war

jps -l

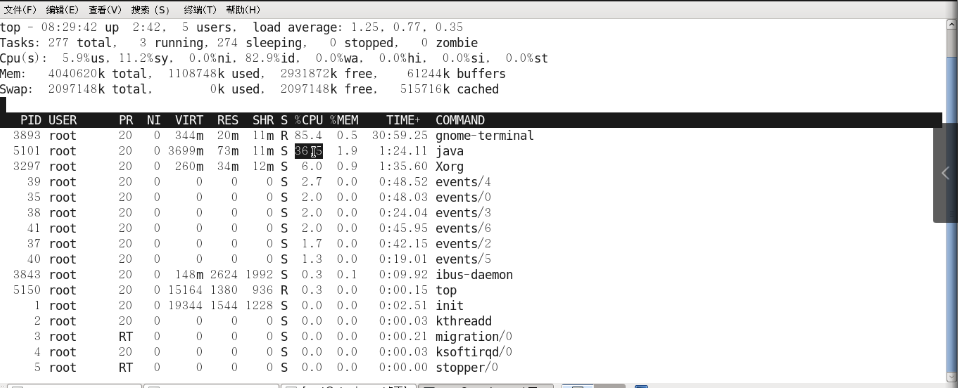
jinfo -flags 7900

# 生成环境服务器变慢，诊断思路和性能评估怎么做？

## 整机：top

uptime就是一个系统性能精简版。

top -Hp

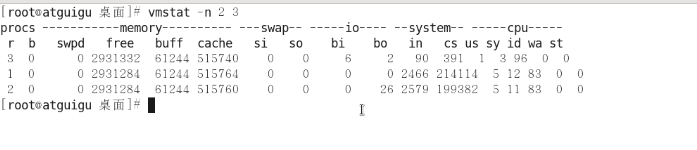


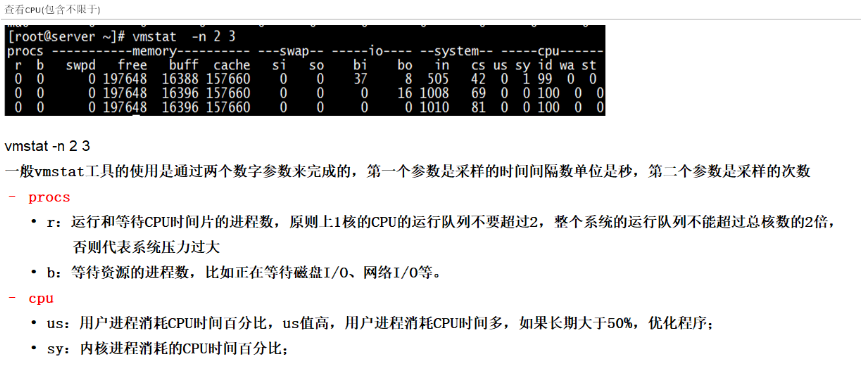
load average 负载均衡 三个值，一分钟五分钟十五分钟系统的平均负载值，加起来除以三，超过60%，就说明压力重。

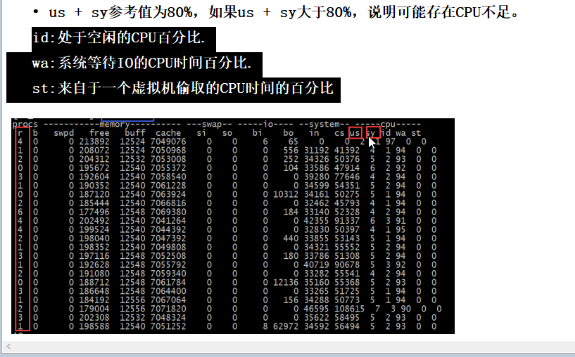
## CPU：vmstat

查看CPU（包含不限于）

vmstat -n 2 3 每两秒采样一次 共计采样三次

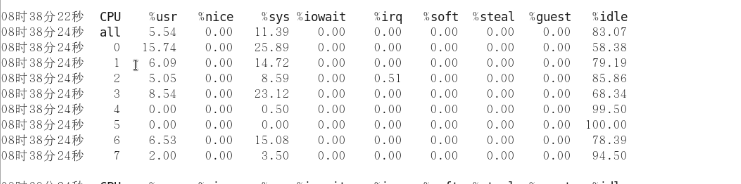






查看额外

查看所有cpu核信息： mpstat -P ALL 2

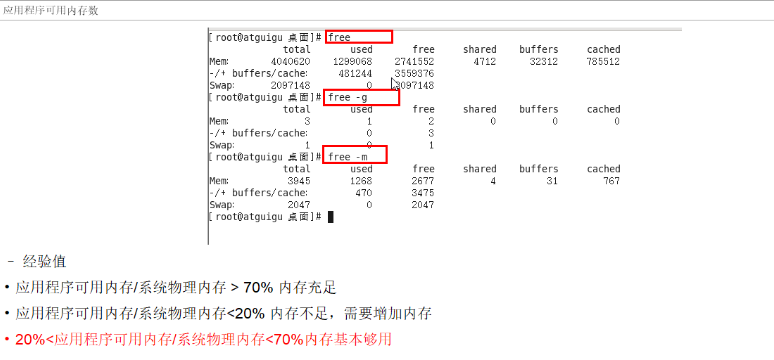


每个进程使用cpu的用量分解信息： pidstat -u 1 -p 进程编号



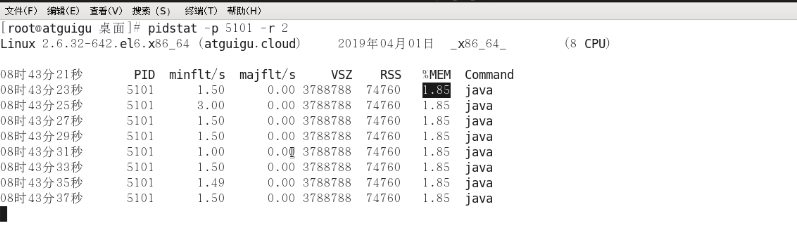
## 内存：free

查看应用程序可用内存数量



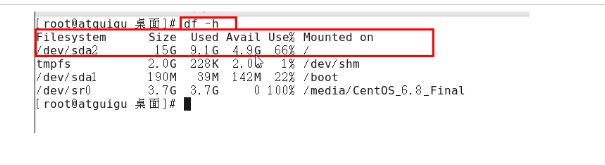
查看额外

pidstat -p 进程好 -r 采样时间秒数



## 硬盘：df

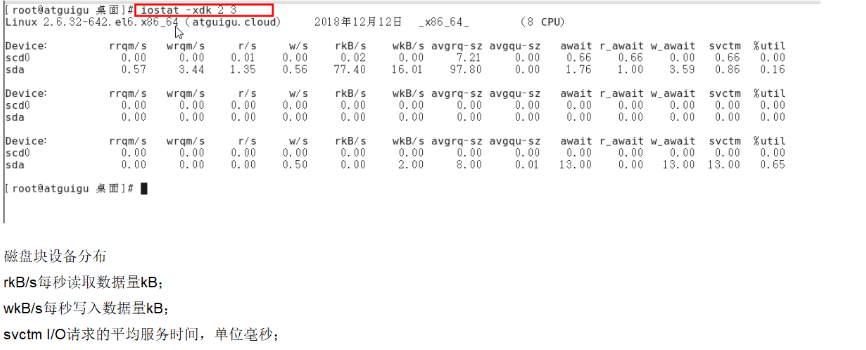
df -h 查看磁盘剩余空间

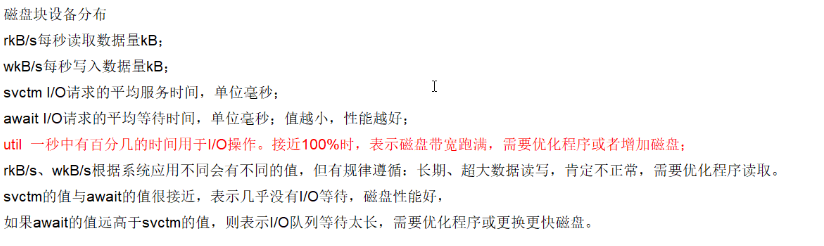


## 磁盘IO：iostat

大表存储查询。

磁盘IO性能评估：iostart -xdk 2 3



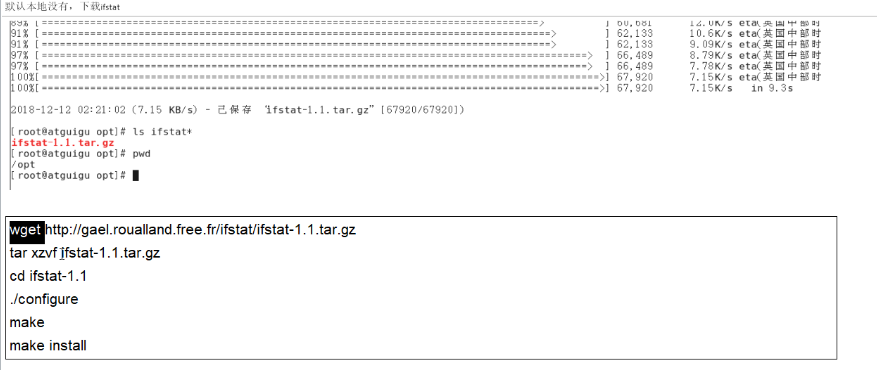


查看额外：

pidstat -d 采样时间间隔秒数 -p 进程号

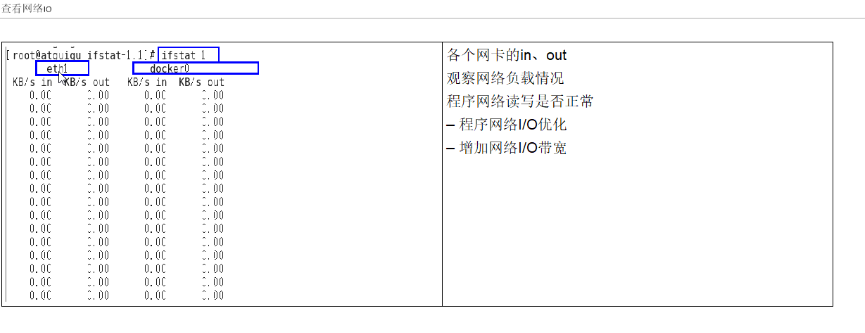
## 网络IO：ifstat

假设本地没有，下载ifstat



查看网络IO

ifstat 1



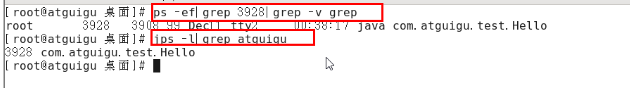
# 假如生产环境出现CPU占用过高，分析思路和定位方法。

结合Linux命令和JDK命令一块分析

进程是5101

1、先使用top命令找到CPU占比最高的是什么命令，也能找到进程编号。

2、使用ps -ef进一步定位 如果这个命令就是java命令，库使用jps命令，得知是一个什么样的后台程序导致的问题。

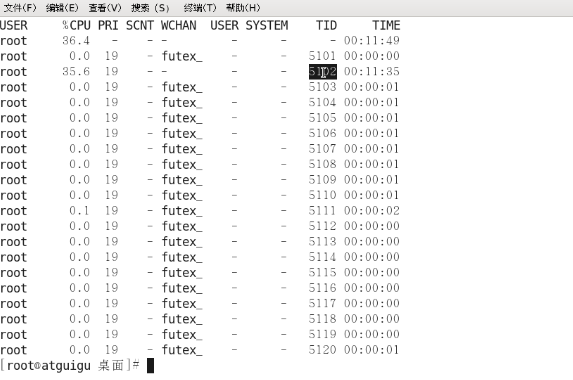


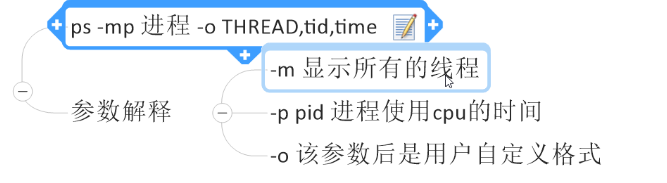
3、定位到具体的线程或者代码

ps -mp 进程编号 -o THREAD,tid,time

ps -mp 5101 -o THREAD,tid,time

发现5102线程占用CPU最高。





4、将需要的线程ID转换为16进制格式

线程在内存当中是16进制的。

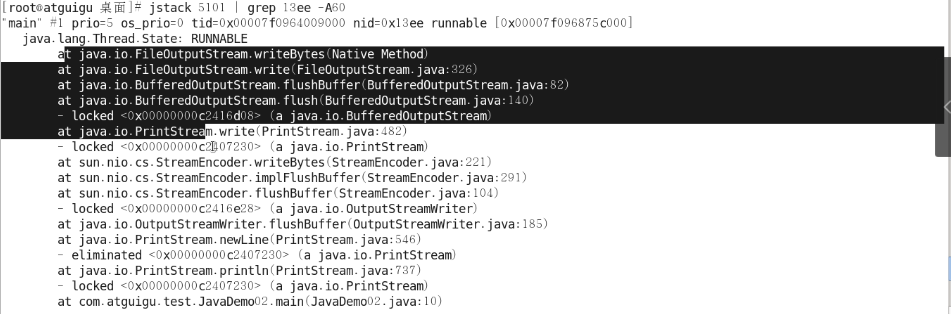
5102就转换成了13ee

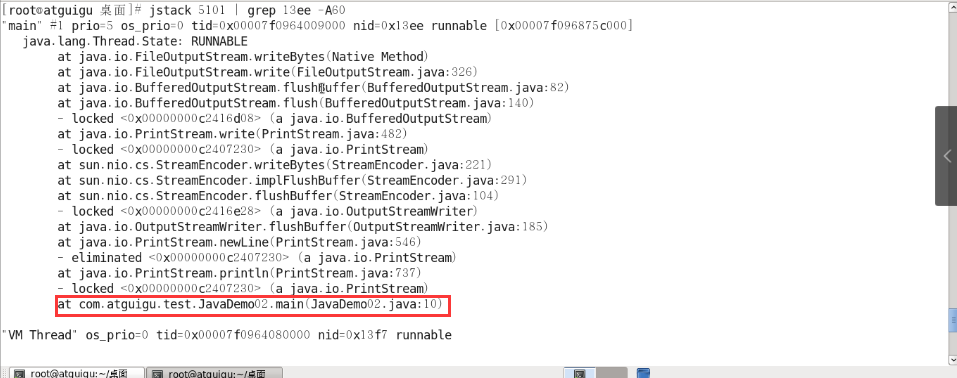
printf “%x\n” 有问题的线程ID

5、jstack 进程ID | grep tid(16进制线程ID小写英文)-A60

jstack 5101 | grep tid(13ee)-A60

前60行





1. 对于JDK自带的JVM监控和性能分析工具用过哪些，一般是怎么用的？

# GitHub

找到优秀的源码，深度的框架学习，学习其他高手的代码，自己共享贡献

## 常用词

watch：会持续受到这个项目的动态

fork：复制某个项目到自己的GitHub仓库

start：点赞理解支持

clone:将项目下载到本地

follow：关注感兴趣的作者，会收到他们的动态

## in关键字限制搜索范围

公式： xxx关键字in:name或description或redame

项目名称包含 项目描述包含 rademe文件包含

举例：搜索项目名称或者rademe文件中包含秒杀的项目：seckill in:name,rademe

## starts 或 fork数量关键字查找

公式：

xxx关键字 starts 通配符 :> 或者 :>=

区间数字 数字1 .. 数字2

点赞数大于5000的一个SpringBoot项目

springboot starts:>=5000

## awesome加强搜索

公式： awesome 关键字

一般是用来收集学习、工具、书籍类相关的项目

搜索优秀的和redis相关的项目，包括框架、教程等 可以使用： awesome redis

## 给别指出关键代码的行数

地址 #L 行号



## 项目内搜索

小写英语字母t

https://help.github.com/en/srticles/using-keyboard-shortcuts

## 某区域的大佬

北京地区 java方向的用户： location:baijing language:java