JVM

Table of Contents

[JVM 1](#_Toc16487953)

[JVM是什么？ 1](#_Toc16487954)

[JVM内存分哪几个区？每个区的作用是什么？ 1](#_Toc16487955)

[讲下java垃圾回收机制，java中垃圾回收的方法有哪些？ 3](#_Toc16487956)

[标记-清除算法 4](#_Toc16487957)

[分代收集算法 5](#_Toc16487958)

[标记整理算法 5](#_Toc16487959)

[简述java内存分配原则，以及MinorGC，和FullGC 5](#_Toc16487960)

[java类加载过程 6](#_Toc16487961)

[简述类加载机制，或者问类加载器双亲委派模型机制？双亲委派可以被打破吗？ 7](#_Toc16487962)

[什么是类加载器，类加载器有哪些?具体的加载是哪个类呢？加载主要分为哪些模块来加载？重点重点重点，需要整理） 7](#_Toc16487963)

[创建对象的过程，new 出一个对象的过程？ 8](#_Toc16487964)

[类冲突问题有碰到过吗？NoClassDefFound和ClassNotFound区别？ 8](#_Toc16487965)

[定位CPU过高的情况 9](#_Toc16487966)

[JVM优化 9](#_Toc16487967)

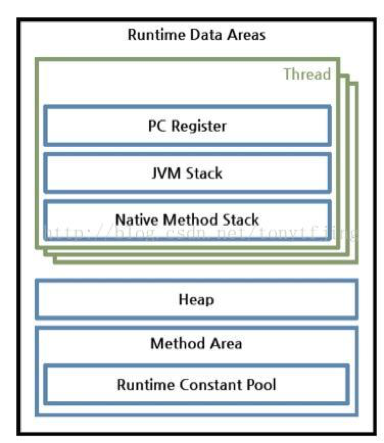
[CMS收集器 13](#_Toc16487968)

JVM是什么？

答：jvm是java虚拟机，是负责执行程序的，jvm本身不是夸平台的，但是对于开发者而言是跨平台的，因为sun公司在jvm帮我们考虑了跨平台，针对每个主流平台都设计了一套JVM。对于java开发者而言，无需关心。

JVM内存分哪几个区？每个区的作用是什么？

**答：JVM内存分5个区，程序计数器，java虚拟机栈，本地方法栈，堆，方法区**



|  |  |
| --- | --- |
| **程序计数器** | 是线程私有的，存放当前线程所执行字节码的行号指示器，记录正在执行的字节码指令的地址，字节码解释器工作时就是通过改变这个计数的值来选取下一条需要执行的字节码指令。循环，跳转，线程恢复等基础模型都需要依赖这个计数器来完成。 |
| **Java虚拟机栈** | 虚拟机栈也是线程私有的，每当创建一个线程时，JVM都会为这个线程创建一个栈，每运行一个方法，就会创建一个栈桢压入栈中，这个栈桢用于存储局部变量表，操作栈，方法返回值。每个方法从调用到执行结束，就对应一个栈桢在java栈中入栈到出栈的过程。 |
| **本地方法栈** | 也是线程私有，作用和java虚拟机栈差不多，只不过是为了JVM使用的本地方法服务的。 |
| **Java堆** | 不是线程私有，里面存储的是java对象的地方，是GC的主要区块。里面又分年轻代和老年代，年轻代又被划分为一个伊甸区，2个存活区from和to，比例默认是8比1比1。伊甸区是新对象分配内存的地方，伊甸区空间不足时会进行一次minorGC   * 在minorGC时，JVM使用复制算法将伊甸区和from区的存活对象复制到to区，并清除内存， * 在minorGC时, to区会把一些存活的足够久（默认是经历过15次minorGC）的对象移至老年代。 * minor GC完之后，这时from区和to区角色会进行互换。   对于年老代，因为对象存活率高，没有额外空间对它进行分配担保，就必须使用“标记-整理”来进行回收。通常会在年老代内存被占满时将会触发Full-GC,回收整个堆内存。Full GC会stop the word，影响性能.  为什么是8：1：1， |
| **方法区** | 也不是线程私有的，用于存储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，方法区还包含一个运行时常量池，用于存储编译期生成的各种字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。Java 1.8之前采用是永久带实现的，之后采用元空间实现，元空间可以利用外部的本地内存，更加的减少GC |

讲下java垃圾回收机制，java中垃圾回收的方法有哪些？

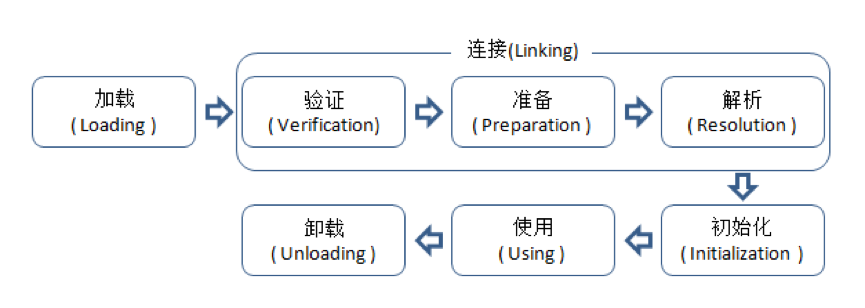
Java的垃圾回收机制的核心原理就是GC回收算法，GC中的垃圾回收算法，大致可分为可达性分析算法，标记-清除算法，标记-整理算法，分代回收算法，复制算法。

|  |  |
| --- | --- |
| **可达性分析算法** | 可达性分析算法的基本思路就是JVM内存中的对象是以树的形式管理的，我叫它为GCtree,GCTree的根节点叫做GC Roots,通常GC-Root为起始点，从这些节点开始向下搜索，搜索走过的路径叫引用链，当一个对象到GCRoot没有任何引用链相连时，则此对象不可用。  有哪些对象可以作为GC Roots呢？   * **虚拟机栈(栈桢中的本地变量表)中的引用的对象** * **本地方法栈中JNI（Native方法）的引用的对象** * **方法区中的类静态变量引用的对象** * **方法区中的常量引用的对象**   判断对象是否存活都与“引用”有关？  Java的四种引用，强弱软虚，用到的场景。   * **强：通常情况下都是，不会被回收 ，就算内存不足，宁愿抛出OOM的错误，也不会回收，所有有时为了收回强引用，就需要设置强引用对象为null** * **弱：内存紧张时会回收，可以用于缓存** * **软：检查到即回收，可以用于缓存** * **虚：几乎和没有引用一样，适用于跟踪对象的回收** |
| 标记-清除算法 | 分标记和清除两个阶段，首先标记标记处所有要回收的对象，在标记完成之后统一回收所有被标记的对象。  这个算法的核心是二次标记，  第一次标记：当GC线程遍历GC tree检测到无用对象的时候，并不是立马回收，只是先给它做个标记，  第二次标记， GC线程会去检查此对象是否有必要执行finalize()方法。当类重写了Object方法的finalize方法时，GC线程就会去执行finalize。如果对象被判定需要执行finalize()方法，那么它将会放置在一个叫F-Queue的队列里面挨个等待执行，如果对象想在finalize()中成功拯救自己，只要重新与GC Roots建立关联，比如把自己赋值给某个类对象或者对象的成员变量。执行了finalize方法，并且成功与引用链相连的话，在对该对象第二次标记的时候，它将从“即将回收”的集合中中被移除；如果这时它还没有建立关联，那么它这次真的就要GC了。  不过它有两点不足之处：  1.效率问题，标记和清除过程的效率不高（递归与全堆对象遍历）。  2.空间问题，标记清除后会产生大量不连续的内存碎片，碎片太多可能会导致以后在程序中需要分配占用较大连续空间的对象（如数组）时，无法找到足够的连续内存而不得不提前触发下另一次垃圾收集动作。  为了解决这些问题，“复制算法”应运而生。 |
| **复制算法** | 复制算法思路比较简单：将内存按容量划分为大小相等的两块，每次只使用其中的一块。当一块内存空间满了，就将还活着的对象复制到另外一块，然后再将之前那块内存空间彻底清空。    这种算法，解决了内存碎片的问题，但是代价太高，浪费一半的内存空间。  在jvm中的新生代使用的就是复制算法，不过对复制算法进行了优化，由于堆内存中的绝大部分对象都是朝生夕死的，所以完全没有必要按照1：1来设定，把内存分成了一块较大的伊甸区和两块较小的存活区（from区和to区），默认比例是8:1:1，每次都使用伊甸区和其中一块from区。当minorGC的时候，将伊甸区和from区里面的存活对象一次性复制到to区里面，最后对伊甸区和from区进行清算，里面的所有要GC的对象清除干净，之后from区和to区互换角色！  为什么比例是8:1:1呢？  首先这比例相比于复制算法，只有10%的空间浪费，另外存活区分为2块呢，为什么不是9:1呢，其实这就引出了另一个算法，**分代收集算法**。 |
| 分代收集算法 | Jvm把堆内存分为两块，一块内存叫就是之前提的新生代，还有另外一块内存叫老年代。我们创建一个对象，经过minorGC之后，这个对象就它的对象头里面会有一个GC分代标识，每经历一次MinorGC如果能活下来该标识+1，当加到默认15次后，GC会判定该对象是个存活时间久的对象，于是乎把它从“新生代”转移到“老年代”了，这就是为什么需要使用2个存活区，因为新生代的对象要满足minorGC15次才能进入到老年代，所以需要2个存活区通过复制算法将GC次数更多的对象进行收集。一个空的存活区用于复制算法里的空的区域，另一个存活区用于保存多次ygc没释放的对象。  由于老年代对象存活率极高，用不着复制算法这一套。于是有人提出了另外一种算法叫做“**标记-整理算法**”。 |
| 标记整理算法 | 标记-整理算法其实基本过程跟“标记-删除”算法差不多，只不过后续的步骤不是对无用对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理到端边界以外的内存。这样就完美解决了“标记-清除算法”内存碎片化的问题。  JAVA的GC回收算法基本就差不多了。我们的GC就是针对内存中的不同区域，采取合理的算法从而达到自动清理的效果。**新生代的对象大多数朝生暮死，只要复制少量的存活对象，就采用“复制算法”，老年代的对象存活率极高，就采用“标记-删除算法”或者“标记-整理算法”** |

简述java内存分配原则，以及MinorGC，和FullGC

* **java内存分配**
  1. 新对象优先在堆的Eden区分配.
  2. 大对象直接进入老年代.
  3. 长期存活的对象将直接进入老年代.
* **MinorGC，和FullGC**
  1. 当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor Gc通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生Gc的频率较高，回收速度比较快;
  2. Full Gc/Major GC 发生在老年代或者方法区，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

java类加载过程



LVPRI

答：类加载过程包括7个阶段：加载，验证，准备，解析，初始化，使用，卸载。

1.在加载阶段虚拟机需要完成以下三件事情：

* **通过一个类的全限定名来获取定义此类的二进制字节流。**（这里没有指明二进制字节流要从一个Class文件中获，可以从其他渠道，譬如：网络、动态生成、数据库等取）
* **将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。**
* **在内存中生成一个代表这个类的对象，作为方法区这个类的访问入口。**

2．**验证阶段负责确保Class文件的字节流是否合法。其中包括文件格式验证，元数据验证，字节码验证，符号引用验证。**

3.**准备阶段负责为类的静态变量分配内存并设置类的静态变量初始值（0值）。**对于final static变量的变量，初始值就是等号右边的值。

4.**解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。**解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

5. 类初始化阶段是类加载过程的最后一步，到了初始化阶段，**才真正开始执行类中定义的java程序代码（或者说是字节码），开始在内存中构造一个Class对象来表示该类，即执行类构造器<clinit>()的过程**。在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则程序员通过程序制定的计划去初始化静态变量和其他资源。

简述类加载机制，或者问类加载器双亲委派模型机制？双亲委派可以被打破吗？

<https://www.cnblogs.com/lanxuezaipiao/p/4138511.html>

类加载机制就是是双亲委派模型机制。

**当一个类加载器收到了类加载请求时，首先不会自己去尝试加载这个类，而是将其委派给父类加载器去加载，每一个层次的类加载器都是如此，因此所有的加载请求最终都会传送到顶层的启动类加载器，只有当父类加载器无法完成这个加载请求的时候，子类加载器才会尝试自己去加载。这种方式的好处是Java类随着它的类加载器一起具备了一种带有优先级的层次关系，可以避免重复加载类。**例如Object类，不管交给哪个类加载器加载，最终都会委派给模型最顶端的bootstrap类加载进行加载，如果没有双亲委派模型的话，各个子类加载器自行加载Object类，那系统将会出现多个不同的Object类，那程序将会混乱。

双亲委派可以被打破。

打破双亲委派机制则不仅要继承ClassLoader类，还要重写loadClass和findClass方法，默认的loadClass方法是实现了双亲委派机制的逻辑，即会先让父类加载器加载，当无法加载时才由自己加载。这里为了破坏双亲委派机制必须重写loadClass方法，即这里先尝试交由系统类加载器加载，加载失败才会由自己加载。它并没有优先交给父类加载器，这就打破了双亲委派机制。

什么是类加载器，类加载器有哪些?具体的加载是哪个类呢？加载主要分为哪些模块来加载？重点重点重点，需要整理）

答：类加载器是java运行时环境的一部分，**负责动态加载类到虚拟机内存中，第一次使用该类才会加载，类加载器包括**

* **启动类加载器，负责核心java库的加载**，存储在<JAVA\_HOME>/jre/lib目录中，具体加载的类由原生代码（如C语言）编写，不继承自java.lang.ClassLoader。
* **扩展类加载器，负责java的扩展类的加载**，存储在<JAVA\_HOME>/jre/lib/ext,或java.ext.dirs中指明的目录中加载 Java的扩展库。该类由sun.misc.Launcher$ExtClassLoader实现。
* **系统类加载器，负责将classpath里的java库进行加载**，该类由sun.misc.Launcher$AppClassLoader实现。
* 自定义类加载器。

创建对象的过程，new 出一个对象的过程？

1. 第一步，首先去检查这个指令的参数是否能在常量池中定位到一个的符号引用，并且检查这个符号引用代表的类是否已被加载，解析和初始化过。如果没有那必须先执行相应的类加载过程。
2. 第二步，如果该类已经加载，解析，初始化之后，就为这个新生的对象分配内存，分配内存又分为两种，一种是“指针碰撞”，一种是“空闲列表”，指针碰撞指的是java堆中的内存区域是完全规整的，所有用过的内存放在一边，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，那分配内存就仅仅是把那个指针向空闲内存空间那边挪动一段与对象大小相等的距离，这就是“指针碰撞”的内存分配方式，”空闲队列“指的是堆内存空间并不是规则的，虚拟机就维护了一种空闲列表中，记录那些内存可用，当分配内存的时候，就会在空闲列表中申请一个足够大的内存空间划分给对象实例，并更新列表上的记录，这就是“空闲列表”的内存分配方式。
3. 第三步，内存分配完成之后，虚拟机对分配的内存进行初始化。就是执行init方法。
4. 第四步，栈中的对象引用指向这块内存区域。

* 扩展：编译器的指令优化，会产生指令重排序，重排序通常是编译时或运行时环境为了优化程序性能而采取的对指令进行重新排序执行的一种手段。重排序分为两类：编译期重排序和运行期重排序，分别对应编译时和运行时环境。在单线程中不影响结果。但是多线程的情况下，指令重排序就会产生线程安全的问题。创建对象的过程，第三步和第四步就可能会乱序，如果是1243，而不是1234的话，在第四步执行完毕，第三部还未执行之前，就被其他线程抢占的话，线程二就会直接返回null的对象实例，顺理成章就会报错。

类冲突问题有碰到过吗？NoClassDefFound和ClassNotFound区别？

FDA

class类是没有找到jar包引起的，一个类引用了另外一个类，结果这个类的jar包没有被jvm加载，就会报错class not founded问题，noClassDefFound的一般是编译失败吧，有这个类文件，也加载了，但是该类中有错误引起的问题。

另外，noClassDefFound是一个错误，classnotfound是一个异常，错误和异常是不一样的，异常可以捕捉然后做处理 可以由开发者控制，错误不行。

还有一个导致ClassNotFoundException的原因就是：当一个类已经某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类。【亲身经历：由于备份了一个class文件A\_bak.class，但是里面的包名类名不变，把新的class拿过来A.class,会重复加载类，就会报该错】

noClassDefFound错误：ClassLoader实例加载类的时找不到类的定义。要查找的类在编译的时候是存在的，运行的时候new却找不到了。NoClassDefFoundError是有JVM引起的，所以不应该尝试捕捉这个错误。

**总结一下区别**

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| 从java.lang.Exception继承，是一个Exception类型 | 从java.lang.Error继承，是一个Error类型 |
| 当应用程序运行的过程中尝试使用类加载器去加载Class文件的时候，如果没有在classpath中查找到指定的类，就会抛出ClassNotFoundException。 | 当JVM在加载一个类的时候，如果这个类在编译时是可用的，但是在运行时找不到这个类的定义的时候，JVM就会抛出一个NoClassDefFoundError错误。比如当我们在new一个类的实例的时候，如果在运行是类找不到，则会抛出一个NoClassDefFoundError的错误。 |
| 一般在执行Class.forName()、ClassLoader.loadClass()或ClassLoader.findSystemClass()的时候抛出 | 由JVM的运行时系统抛出 |

定位CPU过高的情况

1. 执行top -c命令，找到cpu最高的进程id ，pid

2. 然后jstack PID，打印线程堆栈

3. top -H -p pid 定位CPU高的线程，并且通过printf “%x\n” nid，打印nid。

在打印的堆栈日志文件中，tid 和 nid 的含义：

Nid : 对应的 Linux 操作系统下的 tid 线程号，也就是前面转化的 16 进制数字

Tid: 这个应该是 jvm 的 jmm 内存规范中的唯一地址定位

4. 然后在线程堆栈中寻找nid。既可定位有问题的线程和堆栈。

JVM优化

**jvm调优的原则**，也是GC优化的目标就是减少Full GC，

**Full GC触发条件**：

* 1. System.gc()方法的调用，不建议使用
  2. 老年代空间不足，只有在新生代对象转入及创建为大对象、大数组时，老年代才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出错误：java.lang.OutOfMemoryError: Java heap space 。为避免以上两种状况引起的FullGC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。
     + 通过Minor GC后进入老年代的平均大小大于老年代的可用内存。如果发现统计数据说之前Minor GC的平均晋升大小比目前old gen剩余的空间大，则不会触发Minor GC而是转为触发full GC。
     + 由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小
  3. 永久代空间不足。永久代中存放的为一些class的信息等，当系统中要加载的类、反射的类和调用的方法较多时，永久代可能会被占满，在未配置为采用CMS GC的情况下会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出错误信息：java.lang.OutOfMemoryError: PermGen space 。为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

**三种 OutOfMemoryError：**

**OutOfMemoryError:Java heap space - 堆空间溢出**

原因：JVM 分配给堆内存的空间已经用满了。

问题定位

（1）使用 jmap 或 -XX:+HeapDumpOnOutOfMemoryError 获取堆快照。

（2）使用内存分析工具（visualvm、mat、jProfile 等）对堆快照文件进行分析。 （3）根据分析图，重点是确认内存中的对象是否是必要的，分清究竟是是内存泄漏（Memory Leak）还是内存溢出（Memory Overflow）。

内存泄露

内存泄漏是指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。内存泄漏随着被执行的次数越多-最终会导致内存溢出。

内存泄漏常见几个情况：

1.静态集合类

声明为静态（static）的 HashMap、Vector 等集合

比如hashMap的key值不是final的时候，就有可能产生内存泄露。

还有通俗来讲 A 中有 B，当前只把 B 设置为空，A 没有设置为空，回收时 B 无法回收-因被 A 引用。

2.监听器

监听器被注册后，释放对象时没有删除监听器，必须在该对象释放前取消注册

3.物理连接

DataSource.getConnection()建立链接，必须通过 close()关闭链接

4.内部类和外部模块等的引用

发现它的方式同内存溢出，可再加个实时观察

jstat -gcutil 7362 2500 70

5.单例模式

单例对象在被初始化后 将以静态变量的方式 在 JVM 的整个生命周期中存在

如果单例对象持有外部对象的引用 那么这个外部对象将不能被 jvm 正常回收

重点关注：

FGC — 从应用程序启动到采样时发生 Full GC 的次数。

FGCT — 从应用程序启动到采样时 Full GC 所用的时间（单位秒）。

FGC 次数越多，FGCT 所需时间越多-可非常有可能存在内存泄漏。

解决方案

（1） 检查程序，看是否有死循环或不必要地重复创建大量对象。有则改之。

（2） 扩大堆内存空间使用 -Xms 和 -Xmx 来控制堆内存空间大小。

**OutOfMemoryError:PermGen space - 方法区和运行时常量池溢出**

原因：

Perm 区主要用于存放 Class 和 Meta 信息的，Class 在被 Loader 时就会被放到 PermGen space，这个区域称为年老代。GC 在主程序运行期间不会对年老区进行清理，默认是 64M 大小。当程序程序中使用了大量的 jar 或 class，使 java 虚拟机装载类的空间不够，超过 64M 就会报这部分内存溢出了，需要通过加大内存分配，一般 128m 足够。

解决方案：

（1）扩大永久代空间

JDK7 以前使用 -XX:PermSize 和 -XX:MaxPermSize 来控制永久代大小。

JDK8 以后把原本放在永久代的字符串常量池移出, 放在 Java 堆中(元空间 Metaspace)中，元数据并不在虚拟机中，使用的是本地的内存。使用 -XX:MetaspaceSize 和 -XX:MaxMetaspaceSize 控制元空间大小。

注意：-XX:PermSize 一般设为 64M

（2）清理应用程序中 WEB-INF/lib 下的 jar，用不上的 jar 删除掉，多个应用公共的 jar 移动到 Tomcat 的 lib 目录，减少重复加载。

OutOfMemoryError:unable to create new native thread - 线程过多

**OutOfMemoryError: GC overhead limit exceeded**

原因：JDK6 新增错误类型，当 GC 为释放很小空间占用大量时间时抛出；一般是因为堆太小，导致异常的原因，没有足够的内存。

解决方案：

查看系统是否有使用大内存的代码或死循环； 通过添加 JVM 配置，来限制使用内存：

<jvm-arg>-XX:-UseGCOverheadLimit</jvm-arg>

**OutOfMemoryError：unable to create new native thread**

原因：线程过多

当发起一个线程的创建时，虚拟机会在 JVM 内存创建一个 Thread 对象同时创建一个操作系统线程，而这个系统线程的内存用的不是 JVMMemory，而是系统中剩下的内存： (MaxProcessMemory - JVMMemory - ReservedOsMemory) 结论：你给 JVM 内存越多，那么你能用来创建的系统线程的内存就会越少，越容易发生 java.lang.OutOfMemoryError: unable to create new native thread。

**GC 优化需要考虑的 JVM 参数**

栈中内存参数设置：

-Xss：设置每个线程的栈大小

堆中内存参数设置：

-Xms启动 JVM 时堆内存的大小

-Xmx堆内存最大限制新生代空间大小

-Xmn350M：设置新生代大小，-Xmn：至于这个参数则是对 -XX:newSize、-XX:MaxnewSize两个参数的同时配置，也就是说如果通过-Xmn来配置新生代的内存大小

-XX:NewRatio新生代和老年代的内存比：默认值2 ，新生代:老年代 = 1:2

-XX:NewSize新生代初始内存大小 -XX:MaxnewSize 新生代最大内存大小。

-XX:SurvivorRatio：Eden 区和 Survivor 区的内存比，默认值是8

GC 优化时最常用的参数是-Xms,-Xmx和-XX:NewRatio。-Xms和-Xmx参数通常是必须的，所以NewRatio的值将对 GC 性能产生重要的影响。

永久代内存参数设置：

有些人可能会问如何设置永久代内存大小，你可以用-XX:PermSize和-XX:MaxPermSize参数来进行设置，但是要记住，只有当出现OutOfMemoryError错误时你才需要去设置永久代内存

printGCDetail

CMS收集器

<https://www.jianshu.com/p/7d4110b70f7c>

<https://www.baidu.com/link?url=Ru0FiAiZv1hYT6Z6O2CKVl8WnLwKq7JAag6BxGkzzt-ZwHUOKcyu3VeV-85CFgG22M9bInNZ1mQ0Sr6gUx79V7oIS9XtIjUCfuyuW5-8mei&wd=&eqid=f2286ffb00aa3f60000000035d50a749>