

# 内存的自定义分配：

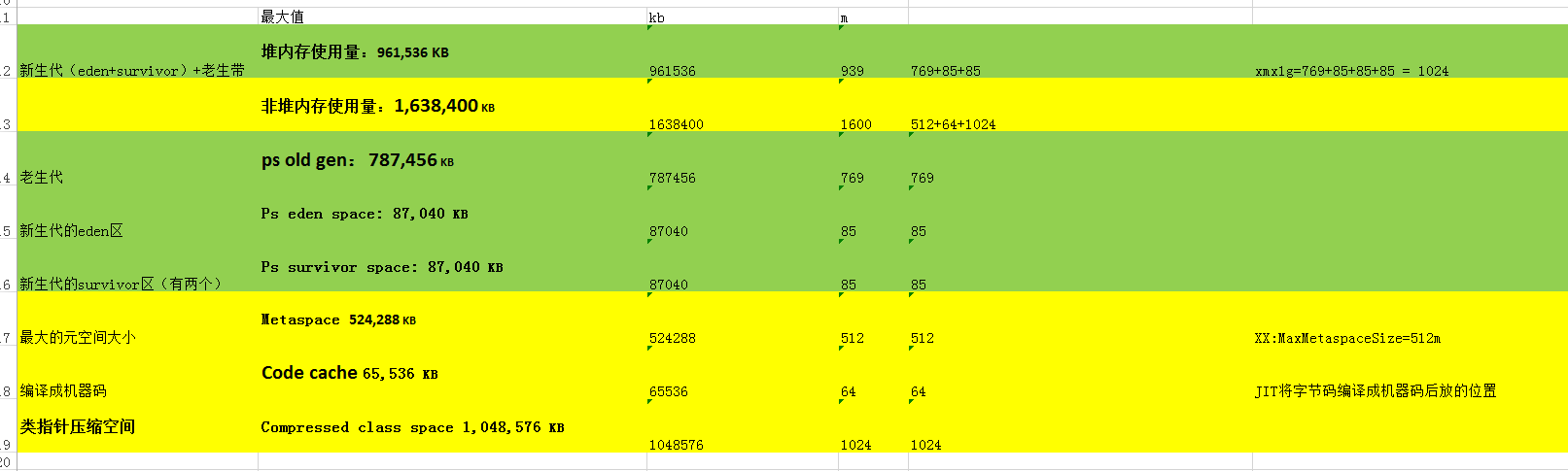
堆区：**通过new的方式创建的对象（类实例）所占用的内存空间**

非堆区：**代码、常量、外部访问（如文件访问流所占资源）等**

简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给自己用的

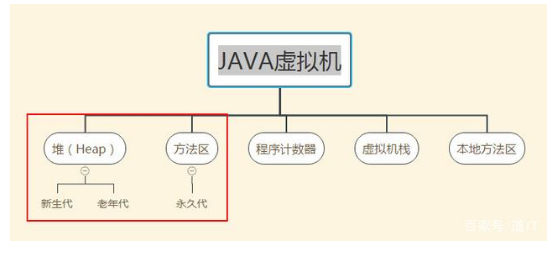
## 堆大小：

堆大小=年轻代（新生代）大小+年老代大小 =新生代（eden+survivor）+年老带



## 堆区的内存：-Xms 、-Xmx、-XX:newSize、-XX:MaxnewSize、-Xmn

1、-Xms ：表示java虚拟机**堆区内存初始内存分配的大小**，通常为操作系统可用内存的1/64大小即可，但仍需按照实际情况进行分配。有可能真的按照这样的一个规则分配时，设计出的软件还没有能够运行得起来就挂了。  
 2、-Xmx： 表示java虚拟机**堆区内存可被分配的最大上限**，通常为操作系统可用内存的1/4大小。但是开发过程中，通常会将 -Xms 与 -Xmx两个参数的配置相同的值，其目的是为了能够在java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小而浪费资源。



### 堆区的新生代、老生代

一般来讲对于堆区的内存分配只需要对上述两个参数进行合理配置即可，但是如果想要进行更加精细的分配还可以对堆区内存进一步的细化，那就要用到下面的三个参数了-XX:newSize、-XX:MaxnewSize、-Xmn。当然这源于对堆区的进一步细化分：新生代、中生代、老生代。

#### 新生代(Young Generation)：java中每新new一个对象所占用的内存空间就是新生代的空间

**1、**-XX:newSize=265m：表示**新生代初始内存的大小**，应该小于 -Xms的值；  
 **2、**-XX:MaxnewSize=1024m：表示**新生代可被分配的内存**的最大上限；当然这个值应该小于 -Xmx的值；  
 **3、**-Xmn250m：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，那么-XX:newSize = -XX:MaxnewSize = -Xmn，虽然会很方便，但需要注意的是这个参数是在JDK1.4版本以后才使用的。

**4、**-XX:SurvivorRatio=5

设置Eden和一个Suivior的比例，比如值为5，即Eden是To(S2)的比例是5，（From和To是一样大的），此时Eden占据Yong Generation的5/7

**5、**-XX:+PrintTenuringDistribution

这个参数用于显示每次Minor GC时Survivor区中各个年龄段的对象的大小。

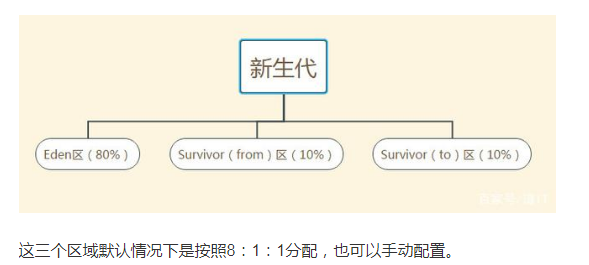
**6、**-XX:InitialTenuringThreshol和-XX:MaxTenuringThreshold

用于设置晋升到老年代的对象年龄的最小值和最大值，每个对象在坚持过一次Minor GC之后，年龄就加1

**7、-XX:NewRatio=4**:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

一般情况下，不允许-XX:Newratio值小于1，即Old要比Yong大

**8、-XX:MaxTenuringThreshold=0**：设置垃圾最大年龄。**如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代**。对于年老代比较多的应用，可以提高效率。**如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间**，增加在年轻代即被回收的概论



主要是用来存放新生的对象。一般占据堆的1/3空间。由于频繁创建对象，所以新生代会频繁触发MinorGC进行垃圾回收。

为了方便垃圾收集，新生代又分出了一个Eden区，两个 Survivor区。

JVM 每次只会使用 Eden区 和其中的一块 Survivor 区域来为对象服务，另一块Survivor区域是空的，用于垃圾回收。

Eden区：Java新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

ServivorTo：保留了一次MinorGC过程中的幸存者。

ServivorFrom：上一次GC的幸存者，作为这一次GC的被扫描者。

当JVM无法为新建对象分配内存空间的时候(Eden满了)，Minor GC被触发。因此新生代空间占用率越高，Minor GC越频繁。

**MinorGC**的过程：采用复制算法。

首先，把Eden和ServivorFrom区域中存活的对象复制到ServicorTo区域（如果有对象的年龄以及达到了老年的标准，一般是15，则赋值到老年代区）

同时把这些对象的年龄+1（如果ServicorTo不够位置了就放到老年区）

然后，清空Eden和ServicorFrom中的对象；最后，ServicorTo和ServicorFrom互换，原ServicorTo成为下一次GC时的ServicorFrom区。

#### 老生代（Old Generation）：当java垃圾回收机制对堆区进行资源回收后，中生代的被转移到老生代

**-XX:OldSize：**设置JVM启动分配的老年代内存大小（初始大小），类似于新生代内存的初始大小-XX:NewSize

在新生代每进行一次垃圾收集后，就会给存活的对象“加1岁”，当年龄达到一定数量的时候就会进入老年代（默认是15,可以通过-XX：MaxTenuringThreshold来设置）

另外，比较大的对象也会进入老年代，可以-XX：PretenureSizeThreshold进行设置。

如-XX：PretenureSizeThreshold3M，那么大于3M的对象就会直接就进入老年代。

老年代中存放的都是一些生命周期较长的对象或者特别大的对象

老年代的对象比较稳定，所以MajorGC不会频繁执行。

在进行MajorGC前一般都先进行了一次MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次MajorGC进行垃圾回收腾出空间。

MajorGC采用标记—清除算法：

首先扫描一次所有老年代，标记出存活的对象

然后回收没有标记的对象。

MajorGC的耗时比较长，因为要扫描再回收。MajorGC会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。

当老年代也满了装不下的时候，就会抛出OOM（Out of Memory）异常。

### 永久代（Permanent Generation）：方法区

**-XX:MaxPermSize=16m**:设置持久代大小为16m

**在非堆区的内存**

不同的Java虚拟机之间可能会进行类共享，因此持久代又分为只读区和读写区

指内存的永久保存区域，主要存放Class和Meta（元数据）的信息。

Class在被加载的时候被放入永久区域。它和和存放实例的区域不同，GC不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常。

在Java8中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。

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，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入java堆中. 这样可以加载多少类的元数据就不再由MaxPermSize控制, 而由系统的实际可用空间来控制。

**Major GC和Full GC区别**

Full GC：收集young gen、old gen、perm gen

Major GC：有时又叫old gc，只收集old gen

#### 为什么移除持久代

它的大小是在启动时固定好的——很难进行调优。-XX:MaxPermSize，设置成多少好呢？

HotSpot的内部类型也是Java对象：它可能会在Full GC中被移动，同时它对应用不透明，且是非强类型的，难以跟踪调试，还需要存储元数据的元数据信息（meta-metadata）。

简化Full GC：每一个回收器有专门的元数据迭代器。

可以在GC不进行暂停的情况下并发地释放类数据。

使得原来受限于持久代的一些改进未来有可能实现

根据上面的各种原因，永久代最终被移除，**方法区移至Metaspace，字符串常量移至Java Heap**

#### 移除持久代后，PermGen空间的状况

这部分内存空间将全部移除。

JVM的参数：PermSize 和 MaxPermSize 会被忽略并给出警告（如果在启用时设置了这两个参数）。**Java8设置这两个参数就无用了！！**

#### 元空间（Metaspace）

-XX:MetaspaceSize，初始空间大小，达到该值就会触发垃圾收集进行类型卸载（超过这个大小会进行Full GC来进行类的回收），同时GC会对该值进行调整：如果释放了大量的空间，就适当降低该值；如果释放了很少的空间，那么在不超过MaxMetaspaceSize时，适当提高该值。

-XX:MaxMetaspaceSize，最大空间，默认是没有限制的

-XX:MinMetaspaceFreeRatio，在GC之后，最小的Metaspace剩余空间容量的百分比，减少为分配空间所导致的垃圾收集   
-XX:MaxMetaspaceFreeRatio，在GC之后，最大的Metaspace剩余空间容量的百分比，减少为释放空间所导致的垃圾收集

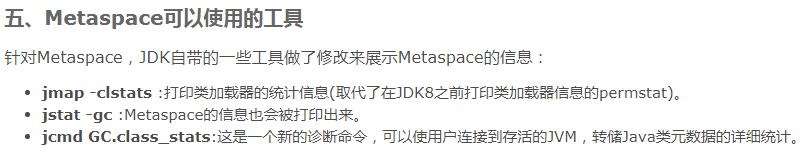
-verbose参数是为了获取类型加载和卸载的信息











从JDK 8开始，Java开始使用元空间取代永久代，元空间并不在虚拟机中，而是直接使用本地内存。

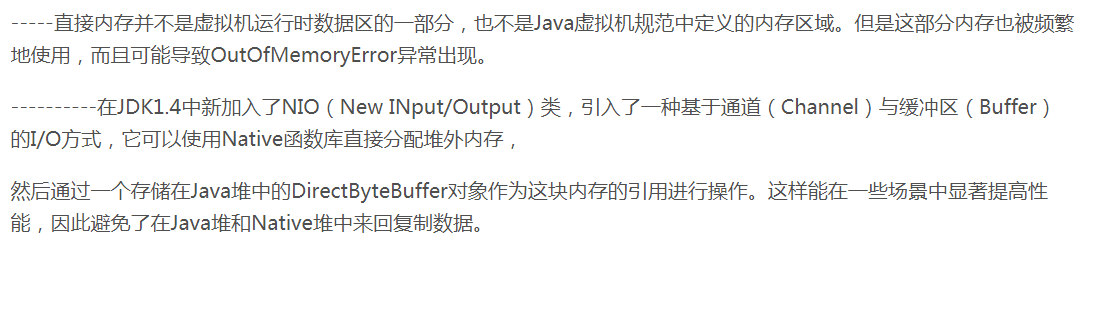
那么，默认情况下，**元空间的大小仅受本地内存限制。当然，也可以对元空间的大小手动的配置。**

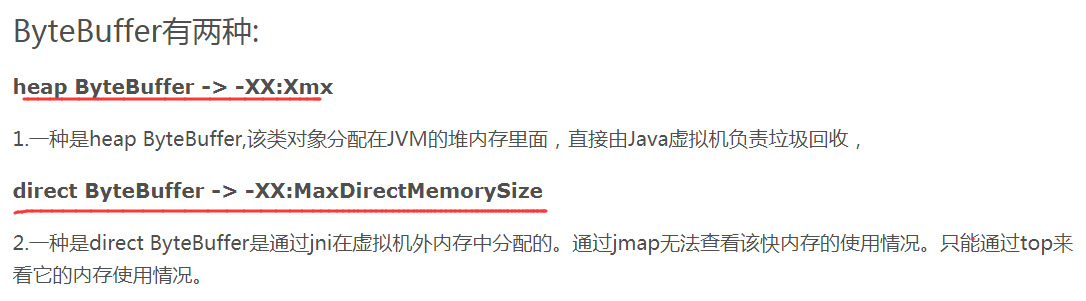
参考：https://www.cnblogs.com/duanxz/p/3520829.html

## 栈内存：

线程所持有的内存，栈空间每个线程是固定的

## DirectMemory内存：





**-XX:MaxDirectMemorySize**

是java nio引入的，直接以native的方式分配内存，不受jvm管理。这种方式是为了提高网络和文件IO的效率，避免多余的内存拷贝而出现的

属于JVM堆外内存，不受JVM堆内存大小的限制，

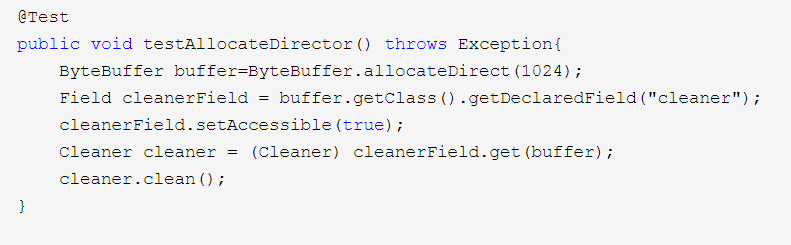
DirectMemory 默认的大小是等同于JVM最大堆，但是并不受其限制。论上说受限于 进程的虚拟地址空间大小，比如 32位的windows上，每个进程有4G的虚拟空间除去 2G为OS内核保留外，再减去 JVM堆的最大值，剩余的才是DirectMemory大小。通过 设置 JVM参数 -Xmx64M，即JVM最大堆为64M，然后执行以下程序可以证明DirectMemory不受JVM堆大小控制：



### 回收机制：

**https://blog.csdn.net/aitangyong/article/details/39403031**

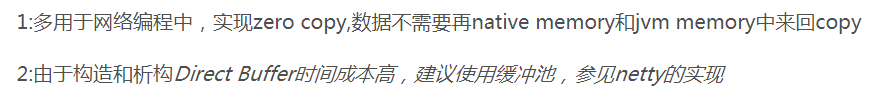
direct ByteBuffer通过full gc来回收内存的，direct ByteBuffer会自己检测情况而调用system.gc()，但是如果参数中使用了DisableExplicitGC那么就无法回收该快内存了，-XX:+DisableExplicitGC标志自动将System.gc()调用转换成一个空操作，就是应用中调用System.gc()会变成一个空操作。那么如果设置了就需要我们手动来回收内存了



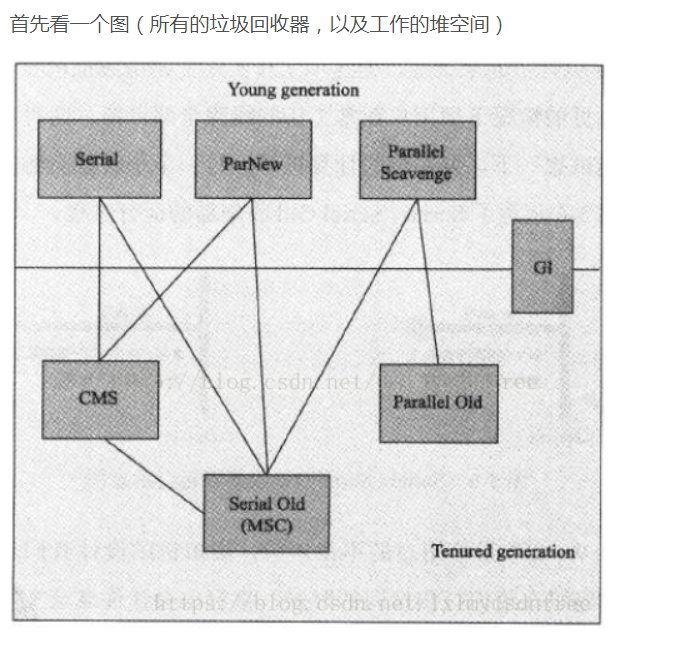
那么除了FULL GC还有别的能回收direct ByteBuffer吗？CMS GC会回收Direct ByteBuffer的内存,CMS主要是针对old space空间的垃圾回收。但是是Oracle JDK 6u32以后的版本



### 使用场景：



## 各个垃圾回收器：



对于我们默认（jdk1.7）使用parallel 垃圾回收器，old区就是使用的是parallel old, yong区使用的是parallel Scavenge



JVM给了三种选择：**串行收集器、并行收集器、并发收集器**，

**串行收集器：**只适用于小数据量的情况，默认情况下，JDK5.0以前都是使用串行收集器，

**并行收集器：-XX:+UseParallelGC，**选择垃圾收集器为并行收集器。**此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。**

**并发收集器：**

**-XX:ParallelGCThreads=20** 配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

-Xss128k 设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行调整。在相同物理内 存下，**减小这个值能生成更多的线程**。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右

**-XX:+UseParallelOldGC**：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

**-XX:MaxGCPauseMillis=100:**设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

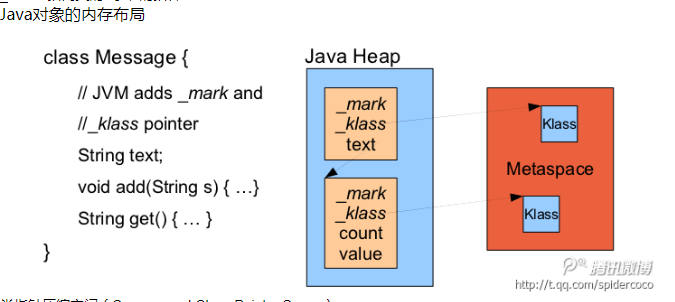
**-XX:+UseAdaptiveSizePolicy**：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开

# 类指针压缩空间：CompressedClassSpaceSize

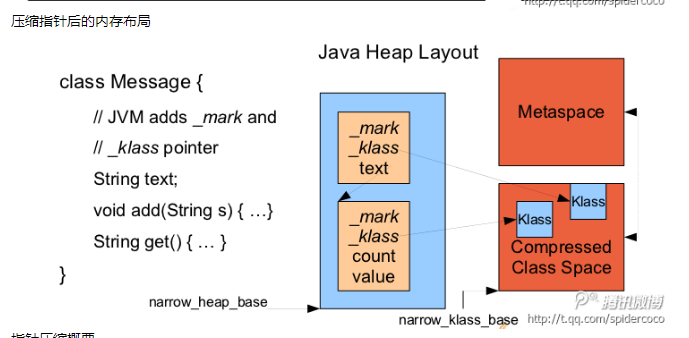
只有是64位平台上启用了类指针压缩才会存在这个区域。

对于64位平台，为了压缩JVM对象中的\_klass指针的大小，引入了类指针压缩空间（Compressed Class Pointer Space）。

## 32位？



## 64位压缩



## 指针压缩概要

* 64位平台上默认打开
* 使用-XX:+UseCompressedOops压缩对象指针 "oops"指的是普通对象指针("ordinary" object pointers)。 Java堆中对象指针会被压缩成32位。 使用堆基地址（如果堆在低26G内存中的话，基地址为0）
* 使用-XX:+UseCompressedClassPointers选项来压缩类指针
* 对象中指向类元数据的指针会被压缩成32位
* 类指针压缩空间会有一个基地址

## CompressedClassSpaceSize的调优

* 只有当-XX:+UseCompressedClassPointers开启了才有效
* -XX:CompressedClassSpaceSize=1G
* 由于这个大小在启动的时候就固定了的，因此最好设置得大点。
* 没有使用到的话不要进行设置
* JVM后续可能会让这个区可以动态的增长。不需要是连续的区域，只要从基地址可达就行；可能会将更多的类元信息放回到元空间中；未来会基于PredictedLoadedClassCount的值来自动的设置该空间的大小

# 系统的设置：

-Xms1g java虚拟机**堆区内存初始内存分配的大小**

-Xmx1g 表示java虚拟机**堆区内存可被分配的最大上限**

-Xmn250m **新生代内存限制**

-XX:MaxDirectMemorySize=512m 默认值是Runtime.getRuntime().maxMemory()，xmx如果设置值，则该值即为默认值，能通过top来看它的内存使用情况

-XX:MetaspaceSize=256m 元空间初始大小

-XX:MaxMetaspaceSize=512m 元空间最大大小

-XX:+UseParallelGC 选择垃圾收集器为并行收集器。**此配置仅对年轻代有效**

-XX:+UseParallelOldGC 配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

-XX:+AlwaysLockClassLoader

-XX:+DisableExplicitGC 禁止代码中显示调用GC，标志自动将System.gc()调用转换成一个空操作，就是应用中调用System.gc()会变成一个空操作。那么如果设置了就需要我们手动来回收内存了

-XX:+PrintGCDateStamps

-XX:+PrintGCDetails

-verbose:gc -Xloggc:%gcDir%/%srvName%\_gc.log

-XX:ErrorFile=%gcDir%/%srvName%\_error\_%p.log

-XX:HeapDumpPath=%gcDir%/%srvName%\_heapDump\_%p.hprof

-Dsun.zip.disableMemoryMapping=true

-javaagent:libs\\classReloader.jar

# 一些典型的配置：

JVM给了三种选择：**串行收集器、并行收集器、并发收集器**，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前[系统配置](http://java.sun.com/j2se/1.5.0/docs/guide/vm/server-class.html)进行判断。

1. **吞吐量优先**的并行收集器  
   如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。  
   **典型配置**：
   * java -Xmx3800m -Xms3800m -Xmn2g -Xss128k **-XX:+UseParallelGC -XX:ParallelGCThreads=20**  
     **-XX:+UseParallelGC**：选择垃圾收集器为并行收集器。**此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。  
     -XX:ParallelGCThreads=20**：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。
   * java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 **-XX:+UseParallelOldGC  
     -XX:+UseParallelOldGC**：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。
   * java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  **-XX:MaxGCPauseMillis=100  
     -XX:MaxGCPauseMillis=100:**设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。
   * java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100 **-XX:+UseAdaptiveSizePolicy  
     -XX:+UseAdaptiveSizePolicy**：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。
2. **响应时间优先**的并发收集器  
   如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。  
   **典型配置**：
   * java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 **-XX:+UseConcMarkSweepGC -XX:+UseParNewGC  
     -XX:+UseConcMarkSweepGC**：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。  
     **-XX:+UseParNewGC**:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。
   * java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC **-XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection**  
     **-XX:CMSFullGCsBeforeCompaction**：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。  
     **-XX:+UseCMSCompactAtFullCollection**：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

## 辅助信息 JVM提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

* + **-XX:+PrintGC**输出形式**：[GC 118250K->113543K(130112K), 0.0094143 secs]**

**[Full GC 121376K->10414K(130112K), 0.0650971 secs]**

* + **-XX:+PrintGCDetails**输出形式**：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]**

**[GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]**

* + **-XX:+PrintGCTimeStamps** -XX:+PrintGC：PrintGCTimeStamps可与上面两个混合使用  
    输出形式：**11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]**
  + **-XX:+PrintGCApplicationConcurrentTime:**打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用  
    输出形式：**Application time: 0.5291524 seconds**
  + **-XX:+PrintGCApplicationStoppedTime**：打印垃圾回收期间程序暂停的时间。可与上面混合使用  
    输出形式：**Total time for which application threads were stopped: 0.0468229 seconds**
  + **-XX:PrintHeapAtGC**:打印GC前后的详细堆栈信息  
    输出形式：  
    34.702: [GC {Heap before gc invocations=7:  
     def new generation   total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000)  
    **eden space 49152K,  99% used** [0x1ebd0000, 0x21bce430, 0x21bd0000)  
    **from space 6144K,  55% used** [0x221d0000, 0x22527e10, 0x227d0000)  
      to   space 6144K,   0% used [0x21bd0000, 0x21bd0000, 0x221d0000)  
     tenured generation   total 69632K, used 2696K [0x227d0000, 0x26bd0000, 0x26bd0000)  
    **the space 69632K,   3% used** [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)  
     compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
       the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
        ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
        rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
    34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs] 55264K->6615K(124928K)**Heap after gc invocations=8:** def new generation   total 55296K, used 3433K [0x1ebd0000, 0x227d0000, 0x227d0000)  
    **eden space 49152K,   0% used** [0x1ebd0000, 0x1ebd0000, 0x21bd0000)  
      from space 6144K,  55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)  
      to   space 6144K,   0% used [0x221d0000, 0x221d0000, 0x227d0000)  
     tenured generation   total 69632K, used 3182K [0x227d0000, 0x26bd0000, 0x26bd0000)  
    **the space 69632K,   4% used**[0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)  
     compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
       the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
        ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
        rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
    }  
    , 0.0757599 secs]
  + **-Xloggc:filename**:与上面几个配合使用，把相关日志信息记录到文件以便分析。

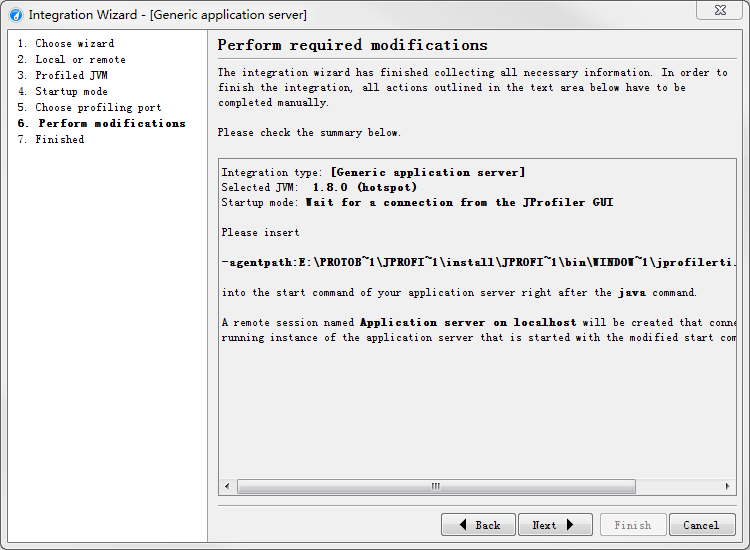
## 常见配置汇总

* + 收集器设置
    - **-XX:+UseSerialGC**:设置串行收集器
    - **-XX:+UseParallelGC**:设置并行收集器
    - **-XX:+UseParallelOldGC**:设置并行年老代收集器
    - **-XX:+UseConcMarkSweepGC**:设置并发收集器
  + 垃圾回收统计信息
    - **-XX:+PrintGC**
    - **-XX:+PrintGCDetails**
    - **-XX:+PrintGCDateStamps** 记录的是系统时间，可读性比较好
    - **-XX:+PrintGCTimeStamps** 这个选项记录的是jvm启动时间为起点的相对时间，可读性较差，不利于定位问题
    - **-Xloggc:filename**
  + 并行收集器设置
    - **-XX:ParallelGCThreads=n**:设置并行收集器收集时使用的CPU数。并行收集线程数。
    - **-XX:MaxGCPauseMillis=n**:设置并行收集最大暂停时间
    - **-XX:GCTimeRatio=n**:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)
  + 并发收集器设置
    - **-XX:+CMSIncrementalMode**:设置为增量模式。适用于单CPU情况。
    - **-XX:ParallelGCThreads=n**:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。
    - **-XX:ReservedCodeCacheSize** 设置(代码缓存)codecache的大小，比如我们jit编译的代码都是放在codecache里的，所以codecache如果满了的话，那带来的问题就是无法再jit编译了，而且还会去优化
    - **-XX:InitialCodeCacheSize** 初始大小

# 一些分析java内存的工具

JDK自带的jps, jstack, jmap, jconsole

# jprofile



-agentpath:E:\PROTOB~1\JPROFI~1\install\JPROFI~1\bin\WINDOW~1\jprofilerti.dll=port=8849

