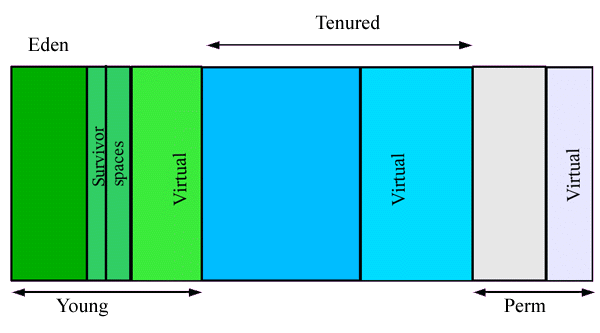
## 我对Java内存的认识（原创）\_nigelzeng\_新浪博客

http://blog.sina.com.cn/s/blog\_68158ebf0100wp83.html

**转载请注明出处：**<http://blog.sina.com.cn/s/blog_68158ebf0100wp83.html>

**一、Java内存的构成**

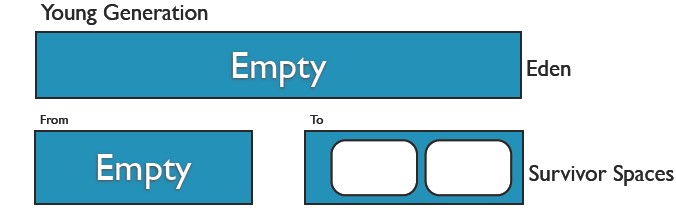
    先上一个官方java document里的图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68158ebf0100wp83&url=http://s6.sinaimg.cn/orignal/68158ebfgad7160602115)  
    由上图可知，整块区域分为Young Generation、Tenured Generation、Permanent Generation。

详细解释一下Young区：

    Young区又分为：Eden、Survivor Space。

    Survivor Space又分为 To Survivor、 From Survivor，如下图所示：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68158ebf0100wp83&url=http://s9.sinaimg.cn/orignal/68158ebfgad71e7acd748)

Java内存分为 堆内存（heap）和 Permanent区。

1、Java堆内存（heap）：

    --是 JVM 用于分配 Java 对象的内存，包含活动对象和不可用对象

    --堆大小通常是在服务器启动时使用 java 命令中的 –Xms(最小) –Xmx（最大）标志来定义。

2、Permanent区：

    --指内存的永久保存区域

    --是Sun JDK和HP JDK用来加载类(class)和Meta信息的专门的内存区

    --这个区域不归属Java堆内存(heap)范围

    --Class在被Loader时就会被放到此，如果Java应用很大，例如类(class)很多，那么建议增大这个区域的大小来满足加载这些类的内存需求

    --通过–XX:PermSize=\*\*\*M –XX:MaxPermSize=\*\*\*M调整

这里还有一个本地内存的概念：

·本地内存(native memory)：

    --是 JVM 用于其内部操作的本地内存（非Java内存）

    --JNI 代码和第三方本地模块（例如，本地 JDBC 驱动程序）也使用本地内存

    --最大本地内存大小取决于以下因素：操作系统进程内存大小限制、已经指定用于 Java 堆的内存

也就是说，整个物理机的内存可以说由以下部分构成：

物理内存 = Java 内存 ＋ 本地内存 ＋ 操作系统保留的内存

**二、垃圾回收（Garbage Collection，GC）**

**1、为什么要垃圾回收**

    --JVM自动检测和释放不再使用的内存。

    --Java 运行时JVM会执行 GC，这样程序员不再需要显式释放对象。

**2、垃圾回收（GC）的分类**

    --Minor GC

    --Full GC

**3、垃圾回收（GC）的产生过程**

     1）新生成的对象在Eden区完成内存分配

     2）当Eden区满了，再创建对象，会因为申请不到空间，触发minorGC，进行young(eden+1survivor)区的垃圾回收。（为什么是eden+1survivor：两个survivor中始终有一个survivor是空的，空的那个被标记成To Survivor）

     3）minorGC时，Eden不能被回收的对象被放入到空的survivor（也就是放到To Survivor，同时Eden肯定会被清空），另一个survivor（From Survivor）里不能被GC回收的对象也会被放入这个survivor（To Survivor），始终保证一个survivor是空的。（MinorGC完成之后，To Survivor 和 From Survivor的标记互换）

     4）当做第3步的时候，如果发现存放对象的那个survivor满了，则这些对象被copy到old区，或者survivor区没有满，但是有些对象已经足够Old（通过XX:MaxTenuringThreshold参数来设置），也被放入Old区

     5）当Old区被放满的之后，进行完整的垃圾回收，即 Full GC

     6）Full GC时，整理的是Old Generation里的对象，把存活的对象放入到Permanent Generation里。

**4、垃圾回收的回收器**

  --串行（–XX:+UseSerialGC ）

    Out of Box算法，年轻代串行复制，年老代串行标记整理，主要用于桌面应用

  --并行（–XX:+UseParallelGC ）

     年轻代暂停应用程序，多个垃圾收集线程并行的复制收集，年老代暂停应用程序，与串行收集器一样，单垃圾收集线程标记整理。JDK 6.0启用该算法后，默认启用了-XX:+UseParallelOldGC，性能大为提高

  --并发(Concurrent Low Pause Collector)( –XX:+UseConcMarkSweepGC )

     启用该参数，默认启用了-XX:+UseParNewGC；简单的说，并发是指用户线程与垃圾收集线程并发，程序在继续运行，而垃圾收集程序运行于其他CPU上。

**三、Java内存的调优参数**

-Xmx1024m：

    设置JVM最大可用内存为1024M。

-Xms1024m：

    设置JVM促使内存为1024M。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn512m：

    设置年轻代大小为512M。（持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。）

-Xss128k：

    设置每个线程的堆栈大小。这个值可以根据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。

-XX:NewRatio=4

    设置年轻代（包括Eden和两个Survivor区）与年老代的比值（总的大小是Xms的值）。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。

    举个例子，-Xms 设置为 1024m，-Xmx 也设置为 1024m的情况下：

      ·年轻代 = 1024M/5 = 204.8M

      ·年老代 = 1024M/5\*4 = 819.2M

    如果-Xms和-Xmx的值设置的不一样，可以添加 -XX:MinHeapFreeRatio=<minimum> 和 -XX:MaxHeapFreeRatio=<maximum> 参数，使内存的大小能够在 大于 -Xms 和 小于 -Xmx 之间的范围内自动调整，所以内存中会有Virtual的空间（我是这样理解的，不是太清楚，这里需要大家指教）

英文原文如下：[http://java.sun.com/docs/hotspot/gc1.4.2/#3. Sizing the Generations|outline](http://java.sun.com/docs/hotspot/gc1.4.2/#3.%20Sizing%20the%20Generations|outline)

    By default, the virtual machine grows or shrinks the heap at each collection to try to keep the proportion of free space to live objects at each collection within a specific range. This target range is set as a percentage by the parameters -XX:MinHeapFreeRatio=<minimum> and -XX:MaxHeapFreeRatio=<maximum>, and the total size is bounded below by -Xms and above by -Xmx .

-XX:SurvivorRatio=4：

    设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

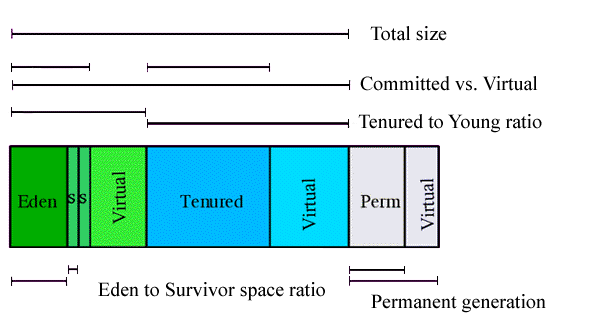
-XX:MaxPermSize=16m:

    设置持久代大小为16m。

-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。

总结如下图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68158ebf0100wp83&url=http://s15.sinaimg.cn/orignal/68158ebfgad725e5e506e)

**四、内存分配中会出现的错误**

关于内存最常见的错误应该是这两个：

  -- 内存溢出 Out Of Memory（OOM）

  -- 内存泄露 Memory Leak （ML）

**1、内存溢出**

    内存溢出发生在这种状况下：Java内存完成Minor GC 之后想要把还存活的对象放到 Old区 里，但是这时Old区 已经满了，同时 Permanent区也已经放不下存活的对象。这时就会产生 OOM 错误。

**2、内存泄露**

    在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是有被引用的，即在有向树形图中，存在树枝通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

    找到一个例子：

    “这里引用一个常看到的例子，在下面的代码中，循环申请Object对象，并将所申请的对象放入一个Vector中，如果仅仅释放对象本身，但因为Vector仍然引用该对象，所以这个对象对GC来说是不可回收的。因此，如果对象加入到Vector后，还必须从Vector中删除，最简单的方法就是将Vector对象设置为null。

1. Vector v = **new** Vector(10);
2. **for** (**int** i = 1; i < 100; i++)
3. {
4. Object o = **new** Object();
5. v.add(o);
6. o = **null**;
7. }//此时，所有的Object对象都没有被释放，因为变量v引用这些对象。

实际上这些对象已经是无用的，但还被引用，GC就无能为力了(事实上GC认为它还有用)，这一点是导致内存泄漏最重要的原因。”

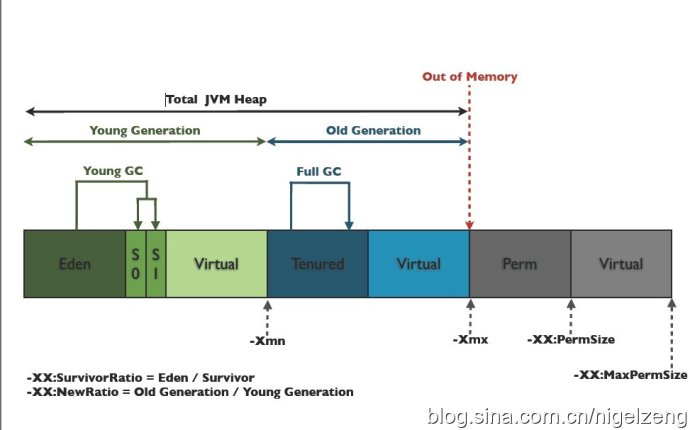
**3、补充一个：PermGen space Error**

    因为 GC 不会在主程序运行期对PermGen space进行清理，所以如果应用中有很CLASS需要Load的话，就很可能出现PermGen space错误。

    另外如果WEB APP下使用了大量的第三方jar, 其大小超过了 jvm 默认的大小那么也会产生此错误信息了。

五、总结

   上面4点的内容可以跟下面这个图来进行融合：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=68158ebf0100wp83&url=http://s1.sinaimg.cn/orignal/68158ebfg77beac7b6000)

## Java 8: 从永久代（PermGen）到元空间（Metaspace） - 炎黄的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/zhyhang/article/details/17246223

正如大家所知，[JDK 8 Early Access](http://jdk8.java.net/)版已经提供下载。这使开发者可以体验Java8的新特性。其中之一，是Oracle从JDK7发布以来就一直[宣称](http://javaeesupportpatterns.blogspot.com/2011/10/java-7-features-permgen-removal.html)的要完全移除永久代空间。例如，字符串内部池，已经在JDK7中从永久代中移除。JDK8的发布将宣告它的终结。这篇文章将会分享到目前为止对 PermGen 继任者：**Metaspace**的了解。我们将通过运行一个存在类元数据对象“泄漏”的程序，来对比HotSpot1.7与HotSpot1.8（b75，译者注：翻译文章时已经到b118）的运行时行为。待Java 8 正式发布之后，才会提供最终的规范，优化参数和相关文档。

元空间（Metaspace）：

一种新的内存空间的诞生

    JDK8 HotSpot JVM 使用本地内存来存储类元数据信息并称之为：**元空间（Metaspace）**；这与[Oracle JRockit](http://javaeesupportpatterns.blogspot.com/2012/02/java-heap-space-jrockit-vm.html) 和[IBM JVM’s](http://javaeesupportpatterns.blogspot.com/2012/02/java-heap-space-ibm-vm.html)很相似。这将是一个好消息：意味着不会再有[java.lang.OutOfMemoryError: PermGen](http://javaeesupportpatterns.blogspot.com/2011/02/outofmemoryerror-permgen-patterns-part1.html)问题，也不再需要你进行调优及监控内存空间的使用……但请等等，这么说还为时过早。在默认情况下，这些改变是透明的，接下来我们的展示将使你知道仍然要关注类元数据内存的占用。请一定要牢记，这个新特性也不能神奇地消除类和类加载器导致的内存泄漏。你需求使用不同的方法以及遵守新的命名约定来追踪这些问题。我推荐大家阅读有关PermGen移除总结和[Jon对此的评论](http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-September/006679.html)。

    总结如下：

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）。

Metaspace 内存分配模型

* 大部分类元数据都在本地内存中分配。
* 用于描述类元数据的“klasses”已经被移除。

Metaspace 容量

* 默认情况下，类元数据只受可用的本地内存限制（容量取决于是32位或是64位操作系统的可用虚拟内存大小）。
* 新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。

Metaspace 垃圾回收

* 对于僵死的类及类加载器的垃圾回收将在元数据使用达到“MaxMetaspaceSize”参数的设定值时进行。
* 适时地监控和调整元空间对于减小垃圾回收频率和减少延时是很有必要的。持续的元空间垃圾回收说明，可能存在类、类加载器导致的内存泄漏或是大小设置不合适。

Java 堆内存的影响

* 一些杂项数据已经移到Java堆空间中。升级到JDK8之后，会发现Java堆 空间有所增长。

Metaspace 监控

* 元空间的使用情况可以从HotSpot1.8的详细GC日志输出中得到。
* Jstat 和 JVisualVM两个工具，在我们使用b75版本进行测试时，已经更新了，但是还是能看到老的PermGen空间的出现。

    前面已经从理论上充分说明，下面让我们通过“泄漏”程序进行新内存空间的观察……

PermGen vs. Metaspace 运行时比较

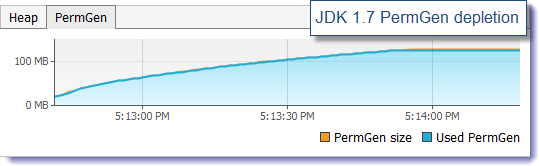
    为了更好地理解Metaspace内存空间的运行时行为，我们建立了一个类元数据泄漏程序。可以从此处下载[源代码](https://docs.google.com/file/d/0B6UjfNcYT7yGbjVLNlpJRWlKUWc/edit?usp=sharing)。

    将进行以下几种场景的测试：

* 使用JDK1.7运行Java程序，监控并耗尽设定的128MB大小的PermGen内存空间。
* 使用JDK1.8 (b75)运行Java程序，监控新Metaspace内存空间的动态增长和垃圾回收过程。
* 使用JDK1.8 (b75)运行Java程序，模拟耗尽通过“MaxMetaspaceSize”参数设定的128MB大小的Metaspace内存空间。

JDK 1.7 @64-bit – PermGen 耗尽测试

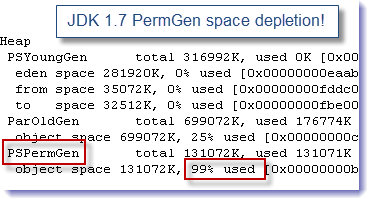
* Java程序中包括5万次可配置迭代过程
* Java堆大小为1024 MB
* Java的PermGen空间为128 MB（-XX:MaxPermSize=128m）

[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/JDK7_JVisualVM_PermGen_depletion.png)

    可以从上面的JVisualVM的截图看出：当加载超过3万个类之后，PermGen被耗尽。我们也能通过程序和GC的输出观察耗尽的过程。

**[html]** [view plaincopy](http://blog.csdn.net/zhyhang/article/details/17246223)[在CODE上查看代码片](https://code.csdn.net/snippets/112240)

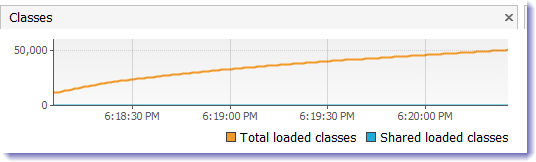
1. Class metadata leak simulator
2. Author: Pierre-Hugues Charbonneau
3. http://javaeesupportpatterns.blogspot.com
4. ERROR: java.lang.OutOfMemoryError: PermGen space

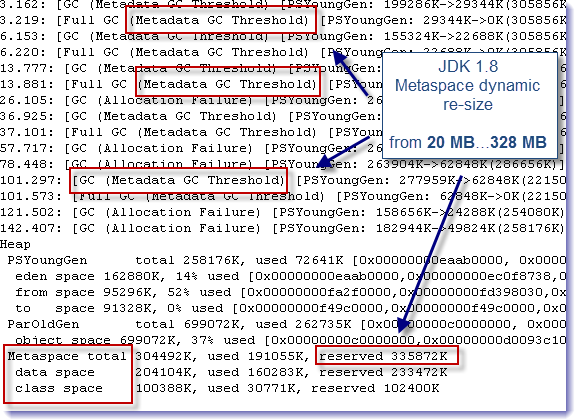
[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/JDK7_GC_PermGen_depletion.png)

    下面我们使用HotSpot JDK 1.8 JRE来执行程序。

JDK 1.8 @64-bit – Metaspace大小动态调整测试

* Java程序中包括5万次可配置迭代过程
* Java堆大小为1024 MB
* Java的Metaspace空间：不受限制 （默认）

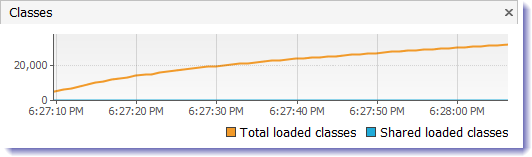
[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/Java8_Metaspace_classes.png)

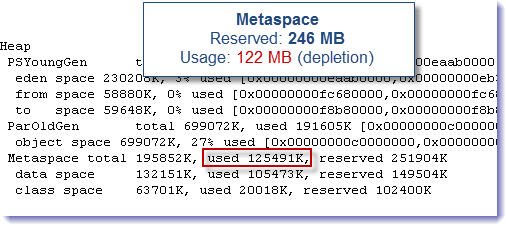
[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/Java8_Metaspace_dynamic_resize.png)

    从上面的截图可以看到详细的GC输出日志，JVM Metaspace进行了动态扩展，本地内存的使用由20MB增长到328MB，以满足程序中不断增长的类数据内存占用需求。我们也能观察到JVM的垃圾回收事件—试图销毁僵死的类或类加载器对象。但是，由于我们程序的泄漏，JVM别无选择只能动态扩展Metaspace内存空间。程序能够运行5万次迭代，加载超过5万个类，而没有出现OOM事件。下面继续进行最后的一个测试场景：

JDK 1.8 @64-bit – Metaspace depletion

* Java程序中包括5万次可配置迭代过程
* Java堆大小为1024 MB
* Java的Metaspace空间：128MB（-XX:MaxMetaspaceSize=128m）

[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/Java8_Metaspace_classes_2.png)

[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/JDK8_GC_Metaspace_depletion.png)

    可以从上面的JVisualVM的截图看出：当加载超过3万个类之后，Metaspace被耗尽；与JDK1.7运行时非常相似。我们也能通过程序和GC的输出观察耗尽的过程。另一个有趣的现象是，保留的原生内存占用量是设定的最大大小两倍之多。这可能表明，如果可能的话，可微调元空间容量大小策略，来避免本地内存的浪费。

    从Java程序的输出中看到如下异常。

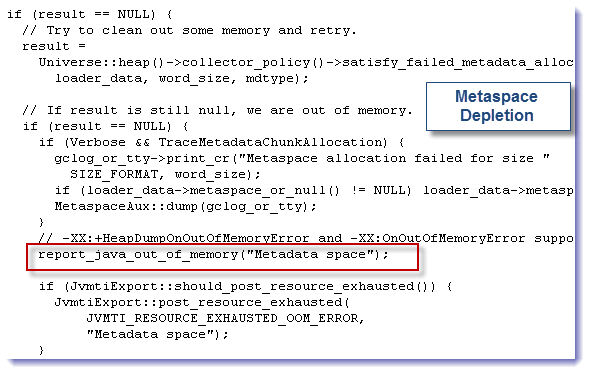
**[html]** [view plaincopy](http://blog.csdn.net/zhyhang/article/details/17246223)[在CODE上查看代码片](https://code.csdn.net/snippets/112240)

1. Class metadata leak simulator
2. Author: Pierre-Hugues Charbonneau
3. http://javaeesupportpatterns.blogspot.com
4. ERROR: java.lang.OutOfMemoryError: Metadata space

    完成！

    正如预期的那样，像运行JDK1.7基线时一样，限定128 MB大小元空间时，并不能让程序完成50万次迭代。一种新的OOM错误被JVM抛出。上述OOM事件，是由于元空间内存分配失败由JVM抛出的。

#metaspace.cpp

[](http://www.javacodegeeks.com/wp-content/uploads/2013/02/Java8_Metaspace_CPP_OOM.png)

结束语

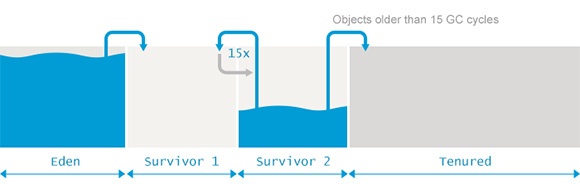
    我希望您能喜欢这篇较早的对Java8元空间的分析和实验文章。目前的观察有力地表明，适当的监控和调优是必须的，以此来避免诸如，过度的在元空间中的GC，或像我们测试场景下触发OOM的条件。后面的文章中可能包括，性能上的比较，以确定这一新特性是否有潜在的性能上的提升。

**参考：**[Java 8: From PermGen to Metaspace](http://javaeesupportpatterns.blogspot.com/2013/02/java-8-from-permgen-to-metaspace.html) from our[JCG partner](http://www.javacodegeeks.com/p/jcg.html) Pierre-Hugues Charbonneau at the[Java EE Support Patterns & Java Tutorial](http://javaeesupportpatterns.blogspot.com/)blog.

[原文地址](http://www.javacodegeeks.com/2013/02/java-8-from-permgen-to-metaspace.html)

### java堆

java堆 新生代（eden, survivor space0, survivor space2,）和老生代



1）新生成的对象在Eden区完成内存分配；

2）当Eden区满了，再创建对象，触发minorGC，进行young(eden+1survivor)区的垃圾回收；

3）minorGC时，Eden不能被回收的对象被放入到空的survivor，同时Eden被清空；

4）当做第3步的时候，如果发现存放对象的那个survivor满了，则这些对象被copy到old区，或者survivor区没有满，但是有些对象已经足够Old（通过XX:MaxTenuringThreshold参数来设置），也被放入Old区

5）当Old区被放满的之后，进行完整的垃圾回收，即 Full GC

6）Full GC时，整理的是Old Generation里的对象，把存活的对象放入到Permanent Generation里。

## JAVA中堆和栈的区别 - huozhicheng的专栏 - 博客频道 - CSDN.NET

http://blog.csdn.net/huozhicheng/article/details/6575960

在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。

当在一段代码块定义一个变量时，Java就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。

堆内存用来存放由new创建的对象和数组。

在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。

在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。

引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。

java中变量在内存中的分配

1、类变量（static修饰的变量）：在程序加载时系统就为它在堆中开辟了内存，堆中的内存地址存放于栈以便于高速访问。静态变量的生命周期--一直持续到整个"系统"关闭

2、实例变量：当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量（比如说类实例），然后根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的"物理位置"。 实例变量的生命周期--当实例变量的引用丢失后，将被GC（垃圾回收器）列入可回收“名单”中，但并不是马上就释放堆中内存

3、局部变量：局部变量，由声明在某方法，或某代码段里（比如for循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放

附：java的内存机制

Java 把内存划分成两种：一种是栈内存，另一种是堆内存。在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配，当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量分配的内存空间，该内存空间可以立即被另作它用。

　　堆内存用来存放由 new 创建的对象和数组，在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或者对象之后，还可以在栈中定义一个特殊的变量，让栈中的这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或者对象，引用变量就相当于是为数组或者对象起的一个名称。引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序运行到使用 new 产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才变为垃圾，不能在被使用，但仍然占据内存空间不放，在随后的一个不确定的时间被垃圾回收器收走（释放掉）。

　　这也是 Java 比较占内存的原因，实际上，栈中的变量指向堆内存中的变量，这就是 Java 中的指针！