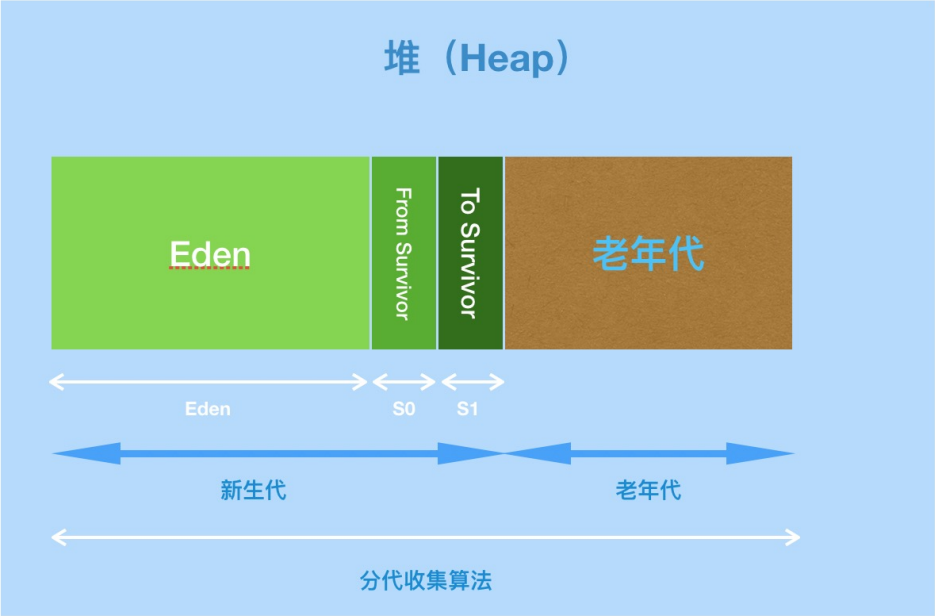
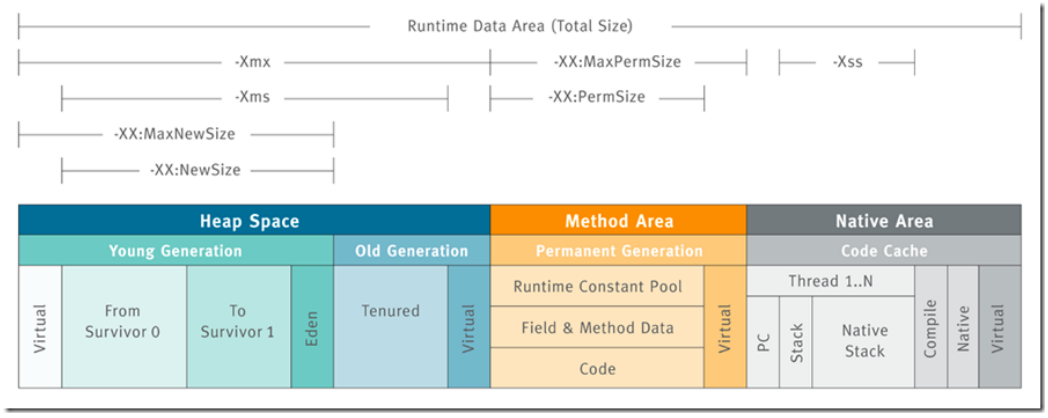
JAVA GC垃圾回收机制详解



图中展示了堆中三个区域：Eden、From Survivor、To Survivor。从图中可以也可以看到它们的大小比例，准确来说是：8:1:1。为什么要这样设计呢，本篇文章后续会给出解答，还是根据垃圾回收的具体情况来设计的。

还记得在设置JVM时，常用的类似-Xms和-Xmx等参数吗？对的它们就是用来说设置堆中各区域的大小的。



控制参数详解：

-Xms设置堆的最小空间大小。
-Xmx设置堆的最大空间大小。
-Xmn堆中新生代初始及最大大小（NewSize和MaxNewSize为其细化）。
-XX:NewSize设置新生代最小空间大小。
-XX:MaxNewSize设置新生代最大空间大小。
-XX:PermSize设置永久代最小空间大小。
-XX:MaxPermSize设置永久代最大空间大小。
-Xss设置每个线程的堆栈大小。

对照上面两个图，再来看这些参数是不是没有之前那么枯燥了，它们在图中都有了对应的位置。

有没有发现没有直接设置老年代空间大小的参数？我们通过简单的计算获得。

1、老年代空间大小=堆空间大小-年轻代大空间大小
对上面参数立即了，但记忆有困难？那么，以下几个助记词可能更好的帮你记忆和理解参数的含义。

Xmx（memory maximum）, Xms（memory startup）, Xmn（memory nursery/new）, Xss（stack size）。

对于参数的格式可以这样理解：

- -: 标准VM选项，VM规范的选项。
- -X: 非标准VM选项，不保证所有VM支持。
- -XX: 高级选项，高级特性，但属于不稳定的选项

1、GC概述

垃圾收集（Garbage Collection）通常被称为“GC”，由虚拟机“自动化”完成垃圾回收工作。

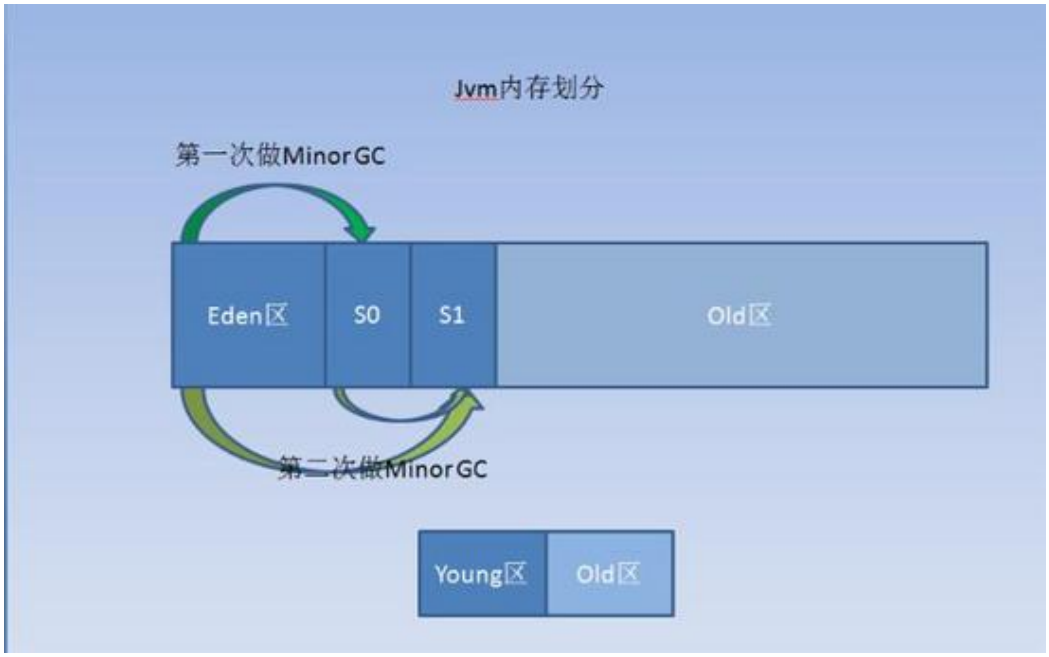
思考一个问题，既然GC会自动回收，开发人员为什么要学习GC和内存分配呢？为了能够配置上面的参数配置？参数配置又是为了什么？

“当需要排查各种内存溢出，内存泄露问题时，当垃圾成为系统达到更高并发量的瓶颈时，我们就需要对GC的自动回收实施必要的监控和调节。”

JVM中程序计数器、虚拟机栈、本地方法栈3个区域随线程而生随线程而灭。栈帧随着方法的进入和退出做入栈和出栈操作，实现了自动的内存清理。它们的内存分配和回收都具有确定性。

因此，**GC垃圾回收主要集中在堆和方法区，在程序运行期间，这部分内存的分配和使用都是动态的。**

2、 GC回收流程



（1）Eden区域是用来存放使用new或者newInstance等方式创建的对象，默认都是存放在Eden区，除非这个对象太大，或者超过了设定的阈值-XX:PretenureSizeThresold,这样的对象会被直接分配到Old区域。

（2）2个Survivor（幸存）区，一般称S0，S1，理论上他们是一样大的，解释一下，他们是如何工作的：

在不断创建对象的过程中，Eden区会满，这时候会开始做Young G也叫Minor GC，而Young空间的第一次GC就是找出Eden区中，幸存活着的对象，然后将这些对象，放到S0，或S1区中的其中一个， 假设第一次选择了S0，它会逐步将活着的对象拷贝到S0区域，但是如果S0区域满了，剩下活着的对象只能放old区域了，接下来要做的是，将Eden区域 清空，此时时候S1区域也是空的。

当第二次Eden区域满的时候，就将Eden区域中活着的对象+S0区域中活着的对象，迁移到S1中，如果S1放不下，就会将剩下的部门，放到Old区域中，只是这次对象来源区域增加了S0，最后会将Eden区+S0区域，清空

第三次和第四次依次类推，始终保证S0和S1有一个是空的，用来存储临时对象，用于交换空间的目的，反反复复多次没有被淘汰的对象，将会放入old区域中，默认是15次。具体的交换过程就和上图中的信息相似。

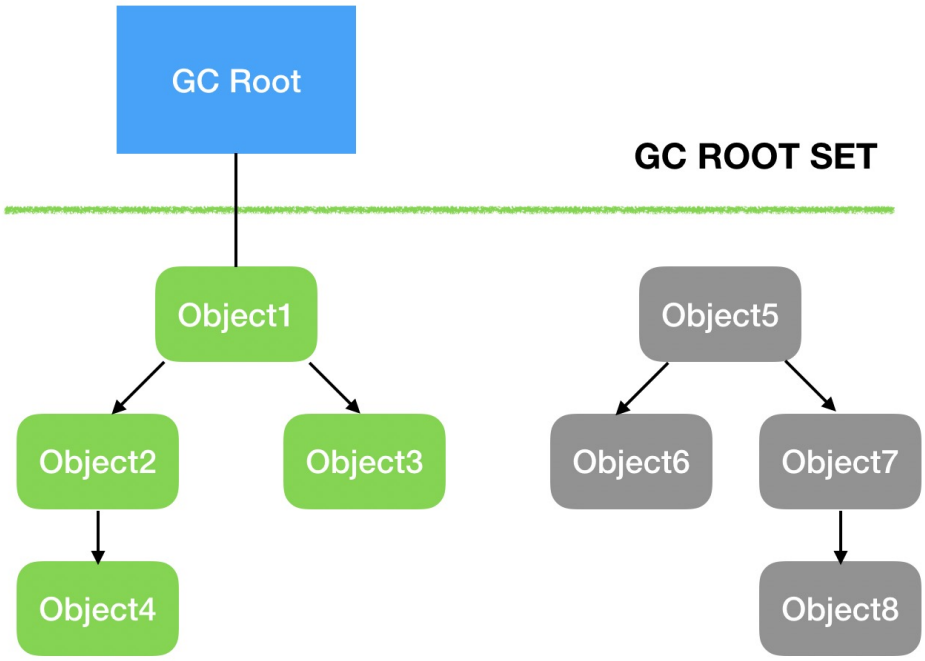
3、如何判断对象存活

判断对象常规有两种方法：**引用计数算法**和**可达性分析算法（Reachability Analysis）**。

引用计数算法：给对象添加一个引用计数器，每当有一个地方引用它时计数器加1，引用释放时计数减1，当计数器为0时可以回收。

引用计数算法实现简单，判断高效，在微软COM和Python语言等被广泛使用，但在主流的Java虚拟机中没有使用该方法，主要是因为无法解决对象相互循环引用的问题。

可达性分析算法：基本思想是通过一系列称为“GC Root”的对象（如系统类加载器、栈中的对象、处于激活状态的线程等）作为起点，基于对象引用关系，开始向下搜索，所走过的路径称为引用链，当一个对象到GC Root没有任何引用链相连，证明对象是不可用的。



上图中中绿色部分为存活对象，灰色部分为可回收对象。虽然灰色部分内部依旧有关联，但它们到GC Root是不可达的。

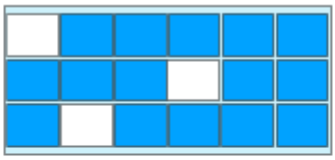
面试问题

面试官，说说Java GC都用了哪些算法？分别应用在什么地方？

答：复制算法、标记清除、标记整理.....

标记清除算法

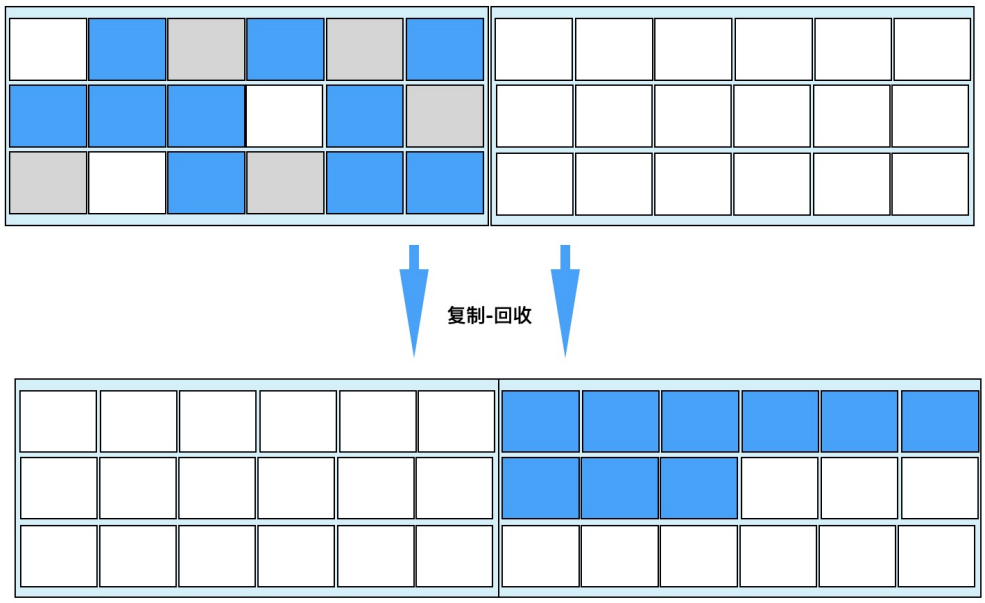
标记清除（Mark-Sweep）算法，包含“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。



主要缺点：一个是效率问题，标记和清除过程的效率都不高；另外是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

4、复制算法

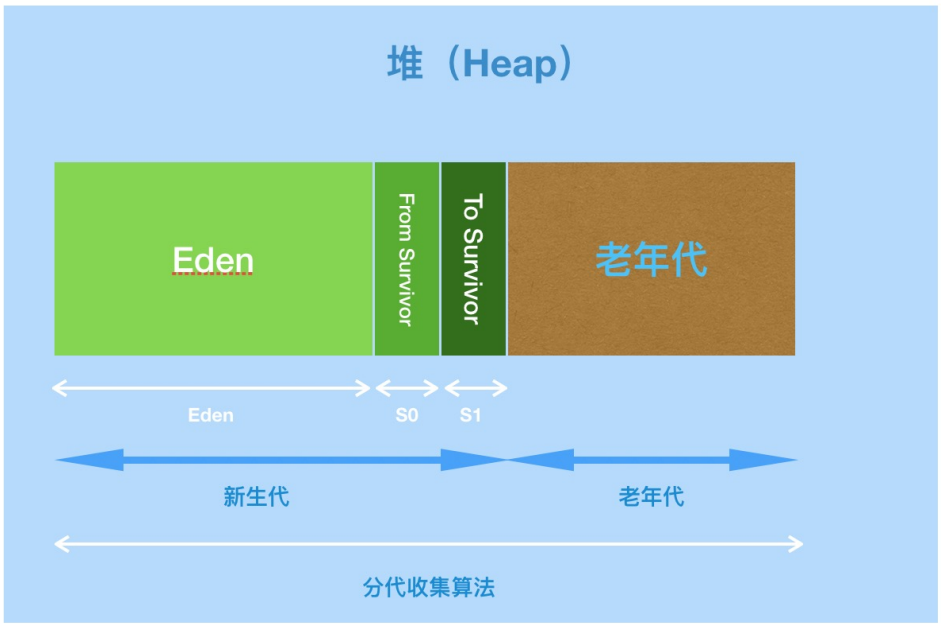
复制（Copying）算法：将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当一块内存用完了，就将还存活着的对象复制到另外一块上，然后清理掉前一块。



每次对半区内存回收时、内存分配时就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

缺点：将内存缩小为一半，性价比低，持续复制长生存期的对象则导致效率低下。

JVM堆中新生代便采用复制算法。回到最初推分配结构图。



在GC回收过程中，当Eden区满时，还存活的对象会被复制到其中一个Survivor区；当回收时，会将Eden和使用的Survivor区还存活的对象，复制到另外一个Survivor区，然后对Eden和用过的Survivor区进行清理。

如果另外一个Survivor区没有足够的内存存储时，则会进入老年代。

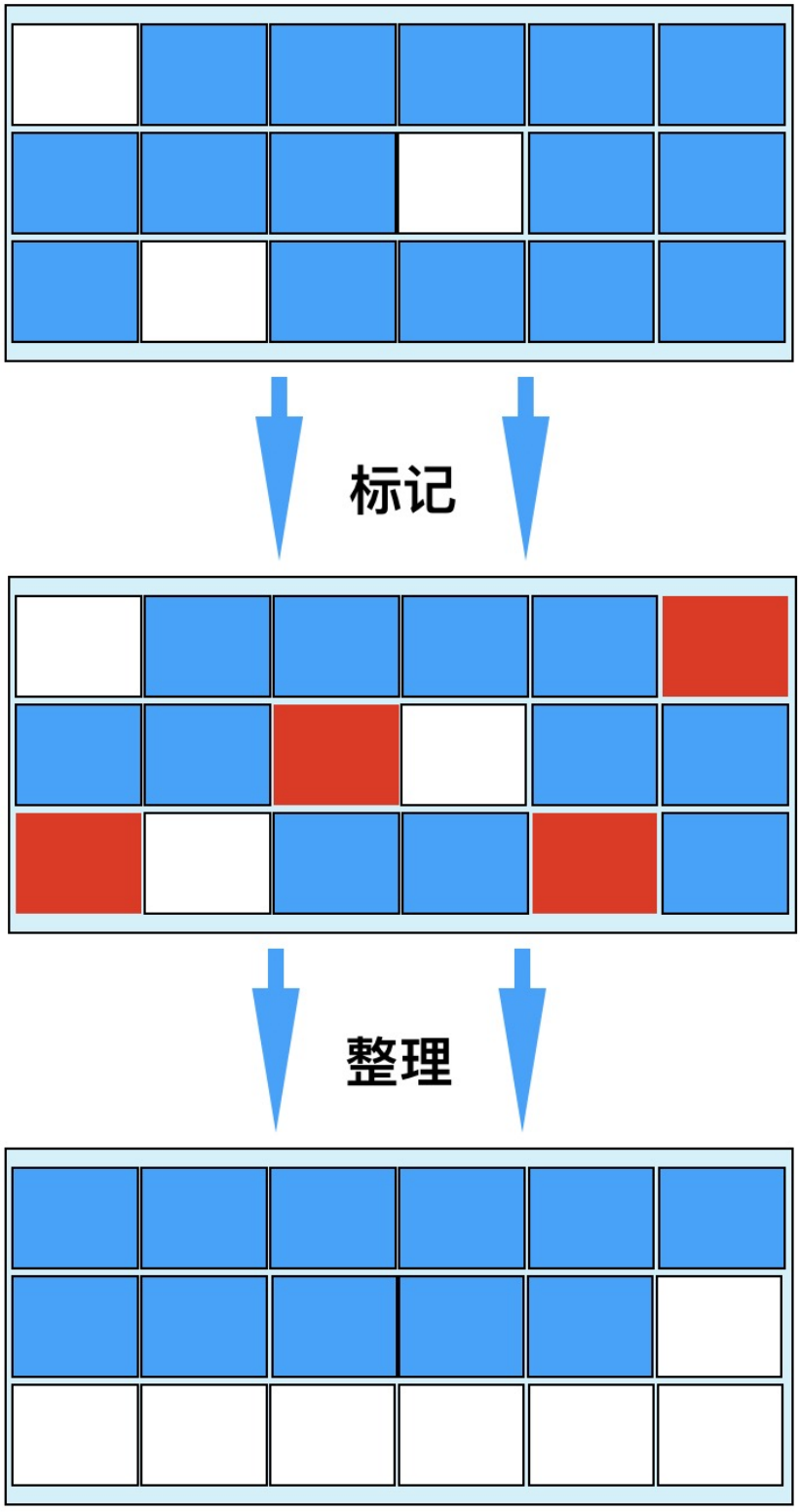
这里针对哪些对象会进入老年代有这样的机制：对象每经历一次复制，年龄加1，达到晋升年龄阈值后，该阈值默认为15，转移到老年代。

在这整个过程中，由于Eden中的对象属于像浮萍一样“瞬生瞬灭”的对象，所以并不需要1：1的比例来分配内存，而是采用了**8：1：1的比例来分配。**

而针对那些像“水熊虫”一样，历经多次清理依旧存活的对象，则会进入老年代，而老年的清理算法则采用下面要讲到的“标记整理算法”。

5、标记整理算法

标记整理（Mark-Compact）算法：标记过程与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。



6、分代收集算法

分代收集算法，基本思路：将Java的堆内存逻辑上分成两块，新生代和老年代，针对不同存活周期、不同大小的对象采取不同的垃圾回收策略。

而在新生代中大多数对象都是瞬间对象，只有少量对象存活，复制较少对象即可完成清理，因此采用复制算法。而针对老年代中的对象，存活率较高，又没有额外的担保内存，因此采用标记整理算法。

其实，回头看，分代收集算法就是对新生代和老年代算法从策略维度的规划而已。

7、常见问题锦集

1、对象进入Old区域有什么坏处？
old区域一般称为老年代，老年代与新生代不一样，年轻代，我们可以认为存活下来的对象很少，而老年代则相反，存活下来的对象很多，所以JVM的 堆内存，才是我们通常关注的主战场，因为这里面活着的对象非常多，所以发生一次FULL GC，来找出来所有存活的对象是非常耗时的，因此，我们应该尽量避免FULL GC的发生。

2、S0和S1一般多大，靠什么参数来控制，有什么变化？
一般来说很小，我们大概知道它与Young差不多相差一倍的比例，设置的的参数主要有两个：

-XX:SurvivorRatio=8

-XX:InitialSurvivorRatio=8

第一个参数是Eden和Survivor区域比重，注意是一个Survivor的的大小，如果将其设置为8，则说明Eden区是一个Survivor区的8倍，换句话说S0或S1空间是整个Young空间的1/10，剩余的80%由Eden区域来使用。

第二个参数是Young/S0的比值，当其设置为8时，表示s0或s1占整个Young空间的12.5%。

3、一个对象每次Minor Gc时，活着的对象都会在s0和s1区域转移，经过经过Minor GC多少次后，会进入Old区域呢？
默认是15次，参数设置-XX:MaxTenuringThreshold=15,计数器会在对象的头部记录它交换的次数

4、为什么发生FULL GC会带来很大的危害？
在发生FULL GC的时候，意味着JVM会安全的暂停所有正在执行的线程（Stop The World），来回收内存空间，在这个时间段内，所有除了回收垃圾的线程外，其他有关JAVA的程序，代码都会静止，反映到系统上，就会出现系统响应大幅度变慢，卡机等状态。

举个通俗易懂点的例子，就是在一个房间里，如果有一个人，不停的扔垃圾，然后有一个清洁工不停扫垃圾，这时候，我们的系统是OK的，因为基本不会 出现垃圾堆满房间的情景，而且因为清洁工可以对付过来，假设现在有10个人不停扔垃圾，那么就房间就会很快被堆满，这时候清洁工，由于工作不过来了，大声 吼一声，你们都暂停3分钟，别再扔了，我先把这个房间打扫完，你们才可以扔。

在这个场景中，一个人扔，一个人扫，就类似于Minor GC，这时候，并不会影响扔垃圾的人，然后一旦10个人同时仍，而且很快就没地方仍了，这时候，就会触发Full GC，然后JVM下令，你们暂时都别仍了，等我什么时候回收完垃圾了，你们在仍，现在大家清楚了吧，所谓的10个人，就是类似我们成千上百的java类， 在不停的执行任务，所谓的清洁工，就是我们的GC机制，所以，大家在平时编码的时候，一定注意尽量少造点垃圾对象，这样触发FULL GC的几率，才会变小。