从JVM分代模型到垃圾回收机制

1.分享概要

本次分享儒猿专栏**《从 0 开始带你成为JVM实战高手》**中JVM垃圾回收机制。本次分享会先从JVM堆内存为什么要分代存放对象开始，分析对象流转的规律以及解析相应的JVM参数。

再从触发一次新生代垃圾回收Minor GC入手，分析GC前的各种校验机制，看看GC后可能出现哪些结果，最后站在JVM整体上分析如何优化JVM参数，让系统更加高效的运行。

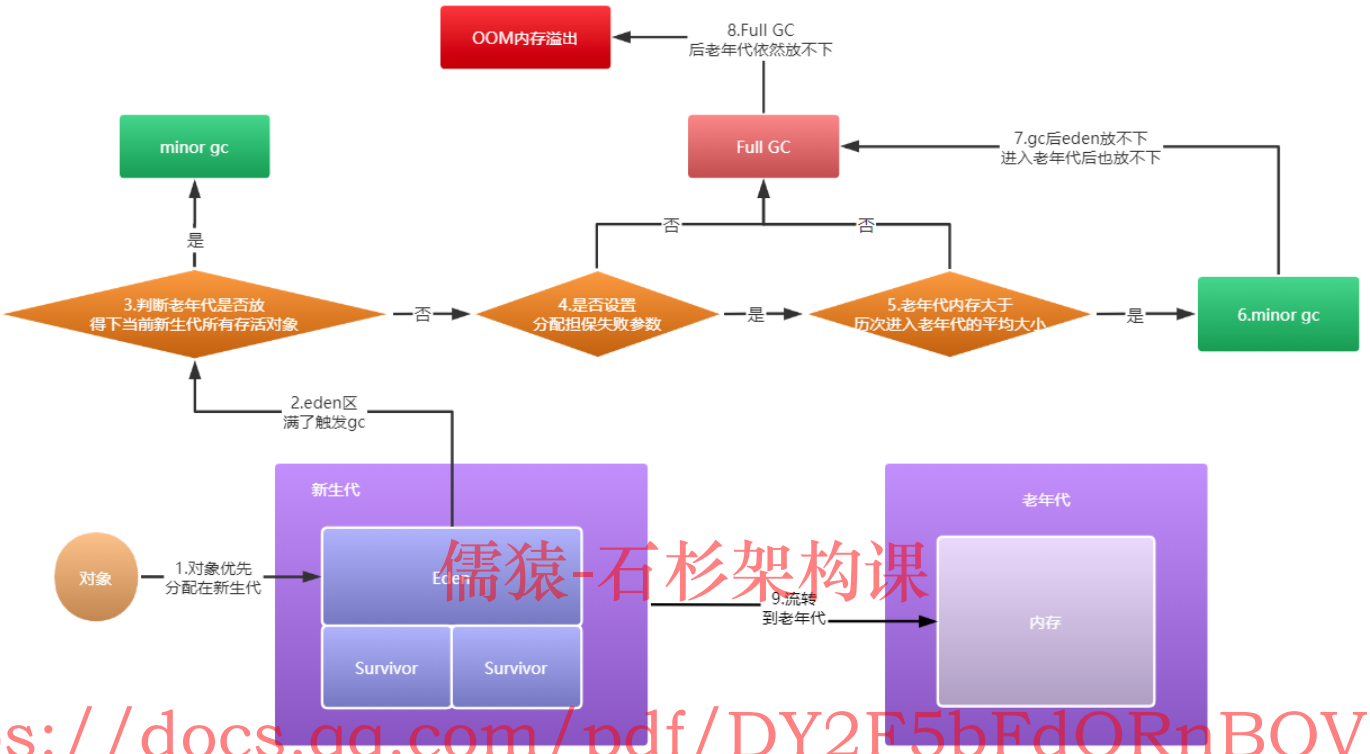
在开始分享前，我们先思考下面的一些面试题：

1.java对象在JVM中是如何分配和流转的?

2.什么时候会在Minor GC前触发一次 Full GC?

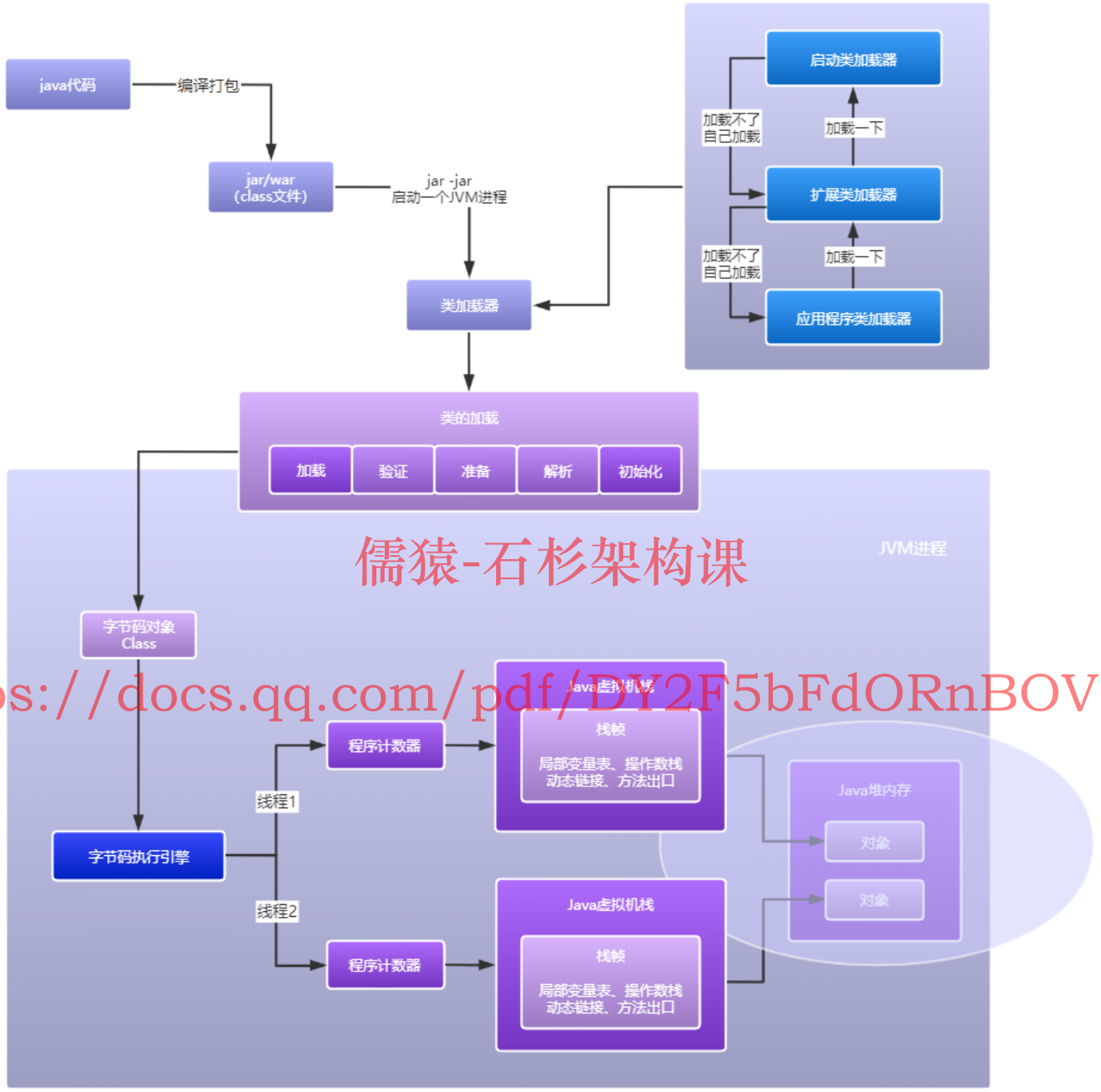
3.Minor GC后触发后对应哪几种回收结果?

本次Minor GC执行的核心逻辑如下图所示：



2.从分代模型到对象分配流转

上一篇文章我们从java代码如何在JVM中执行时，了解到会结合JVM中的方法区、程序计数器、JVM栈、堆内存这些数据结构来完成Java代码的整体运行，而JVM这块最重要的一个环节就是了解了堆内存这个数据结构，我们大概看下当下的重心转移到哪了：

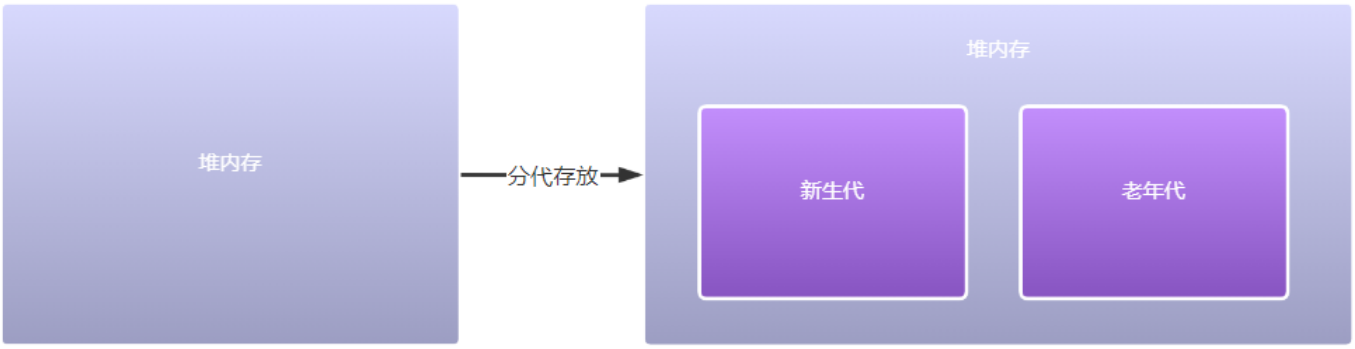


一般我们只要new一个对象，就会在堆内存中开辟内存并创建一个对象，而堆内存中的对象一般主要分为两类：

一类是生命周期比较短的，比如在方法中创建一个对象，方法执行完、相应JVM栈中的的栈帧销毁，堆内存对象没有被引用就成垃圾就可以回收了；

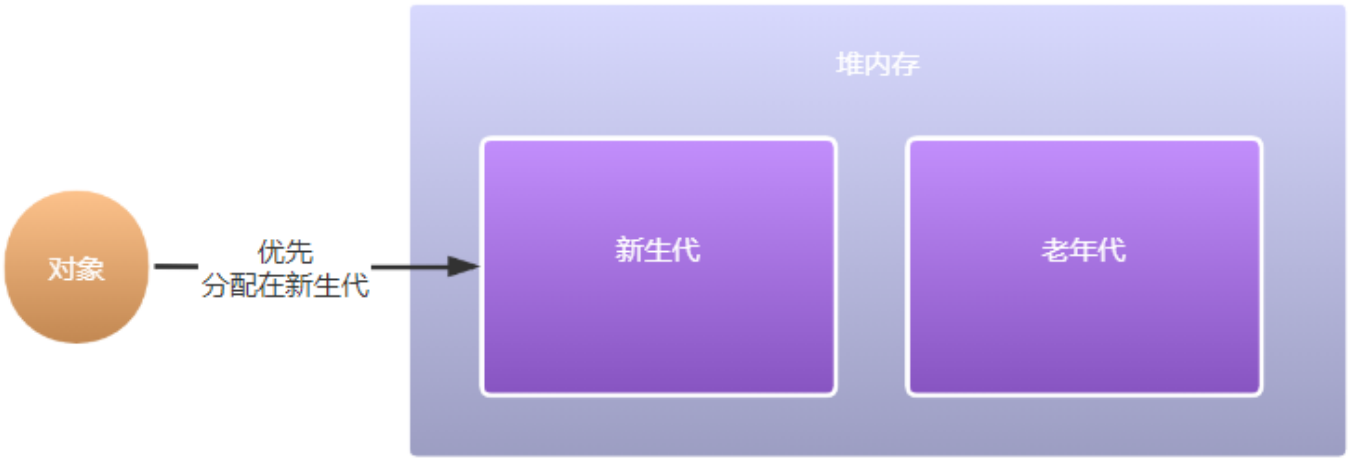
另一类是生命周期比较长的对象，比如单例模式创建的对象，生命周期就比较长了。

正是因为有这两种生命周期的对象，JVM的堆内存划分为了两个部分分开存放，分别为新生代和老年代，方便我们后续对这两类对象、用不同的方式处理和回收它们，如下图所示：



2.1 对象在堆内存中如何分配的

JVM规定，当我们在方法中创建一个对象时，对象优先是在堆内存中的新生代分配的，也就是说不管你创建的对象是生命周期长的还是生命周期短的，一开始都是存放在新生代中的，如下图所示：



2.2 对象在堆内存中如何流转的

那么现在问题来了，既然JVM堆内存根据对象不同生命周期的特点，划分了新生代和老年代两个区域，但是对象一开始又是首先在新生代分配内存，那么新生代中的对象在什么样的条件下才会进入老年

代呢，接下来我们看下几种常见的流转场景。

（1）年龄大的对象进入老年代

当新生代中的对象经过一次垃圾回收之后，发现没有被回收掉，比如前面说到的通过单例模式创建的对象，肯定存活时间就很长，简单的一次垃圾回收肯定就回收不掉。

每当新生代中的对象躲过一次垃圾回收，对象的年龄就会增加1岁，默认是当新生代的年龄超过15岁时就会流转到老年代中，可以通过JVM参数：–XX:MaxTenuringThreshold 调整。

（2）动态年龄判断

这个判断比较有意思，它假设现在新生代中有一批对象，这批对象已经占据新生代50%以上的内存了，那么其他对象就要和这批对象中、最大年龄的那个对象对比下年龄，如果比这批对象最大的那个对象年龄还大，那你就直接到老年代去吧。

比如有一批占据新生代50%以上的一批对象年龄有1岁、2岁、5岁、7岁，如果其他对象如8岁，那么这个8岁的对象因为已经大于这批对象最大年龄7岁，所以直接就会流转到老年代中。

（3）大对象

如果新生代中分配了一个比较大的对象，比如往往会出现那种大数组，这样的大对象放在新生代也是不合适的，也会直接流转到老年代，可以通过JVM参数：**–XX:PretenureSizeThreshold** 调整多大的对象进入新生代就会直接流转到老年代。

（4）新生代放不下

当一次新生代的垃圾回收之后，发现存活的对象太多了，导致新生代中自己都放不下了，此时也会将这批新生代放不下的对象流转到老年代中，此时老年代扮演的角色就像是为新生代内存的不足兜底的。

其实这一块对象的流转机制也比较好理解，年龄大的对象肯定是要让它快点进入老年代的，毕竟老年代划分出来的目的、就是为了存放那些生命周期比较长的对象，实至名归；

动态年龄判断也是考虑到哪些年龄比较小的对象已经占据了过多的空间了，年龄大迟早要去老年代的就先快点去吧；

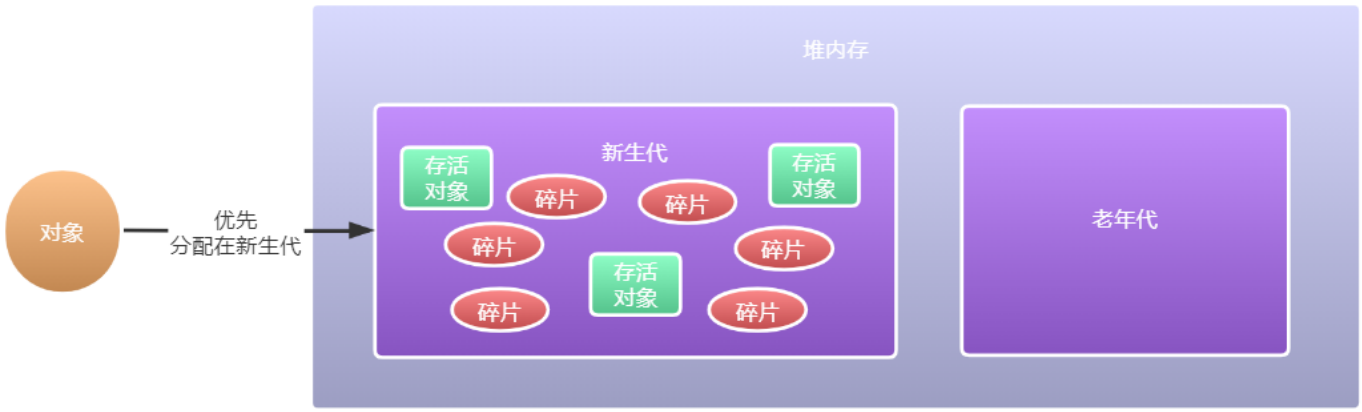
至于大对象以及新生代放不下的场景，更像是老年代作为新生代内存不够的一个备用内存，为新生代内存的不足兜底的。

3.从GC算法看JVM内存划分

因为堆内存中存在生命周期长和生命周期短的两类对象，所以才将堆内存划分出了新生代和老年代、分开来存放这两类对象，最终的目的当然是分而治之，用不同的方法回收这两个区域的垃圾对象。

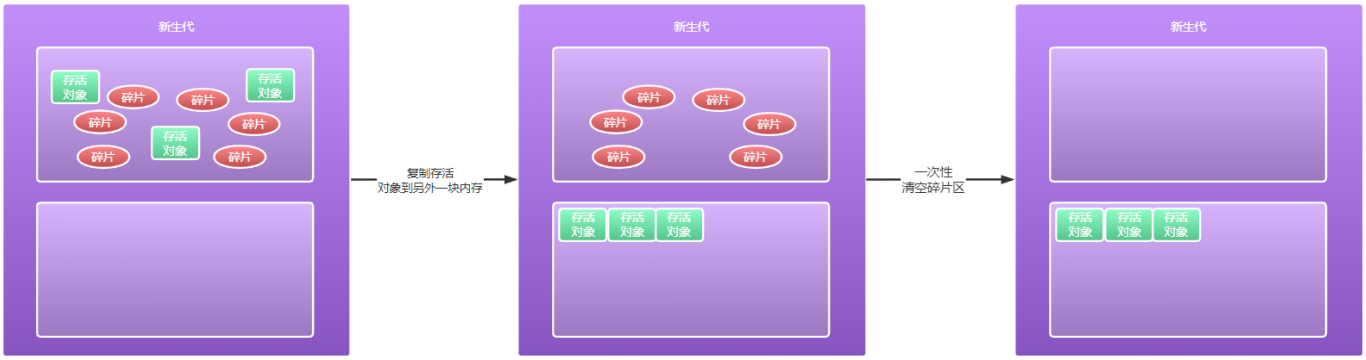
垃圾回收算法主要有三种，分别是标记清除、复制算法和标记整理算法。

对于新生代而言如果使用标记清除算法，就会导致大量的内存碎片，因为新生代中存放的都是生命周期比较短的对象，一次垃圾回收之后，大量的对象都会被回收掉，从而留下大量的内存碎片，后续如果有一个稍微大一点的对象想要进入新生代，可能都没有足够大的连续内存放的下，从而被迫流向老年代，如下图所示：



新生代比较适合使用复制算法，毕竟新生代里面大量的对象都是存活时间比较短的，一次GC之后可能都存活不到多少对象，复制的成本就很低了。

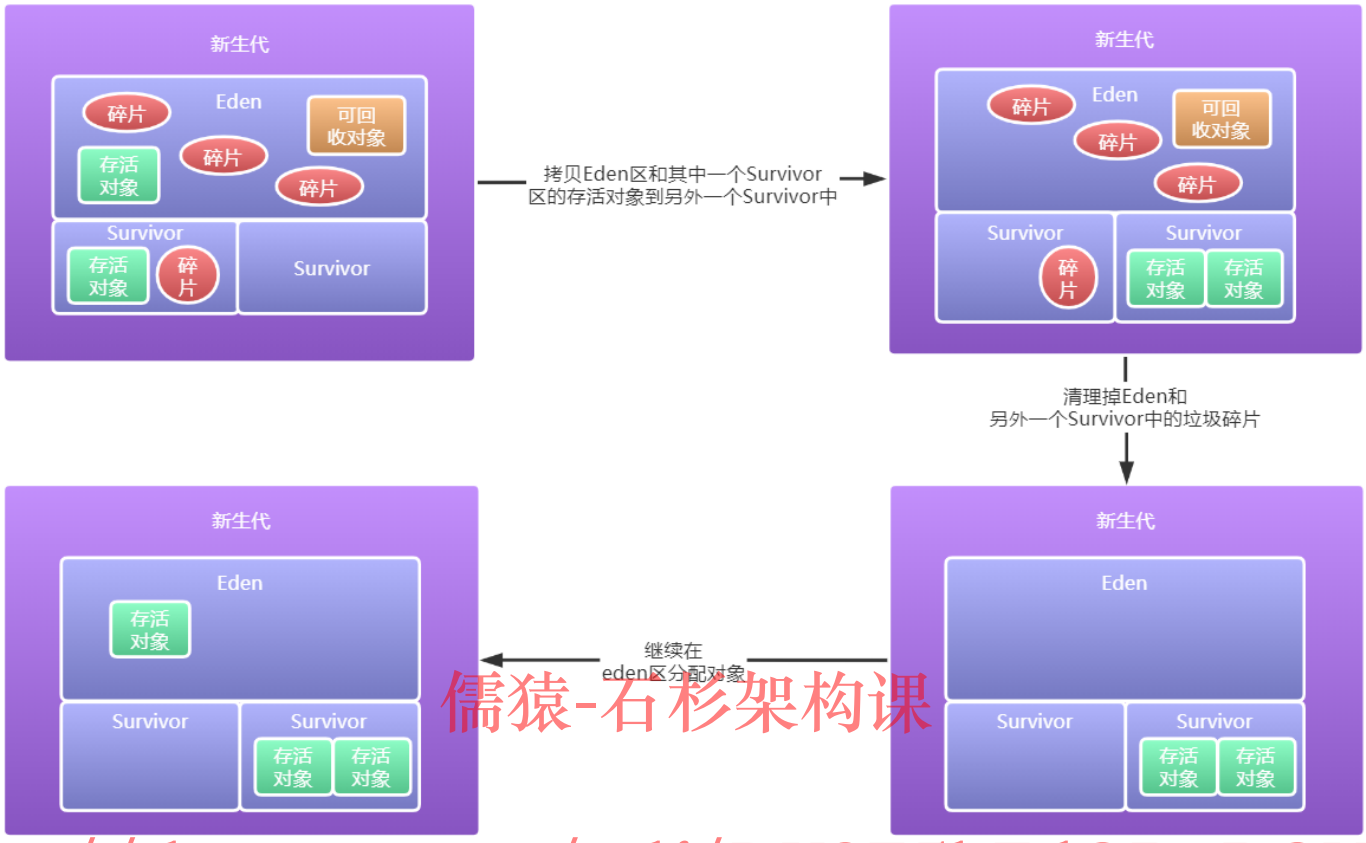
最初级的复制算法，会先将新生代内存划分为两块等同大小的内存，每次只在其中一块内存中分配对象，另外一半内存空闲，当其中一块内存满了时，直接将内存中还存活的对象拷贝到空闲的内存中、紧贴存放，然后将内存一次性清空，这样内存碎片就会很少了，如下图所示：



从上图我们可以看到，初级的复制算法的问题还是比较明显的，每次将近有一半的新生代内存都被浪费了，资源的使用率只有50%。

而经过改良后，最终在JVM中划分了三个区域：Eden、Survivor、Survivor，默认Eden和Survivor的占比为：8：1，可通过JVM参数： -XX:SurvivorRatio 调整比例，默认值就为：8。

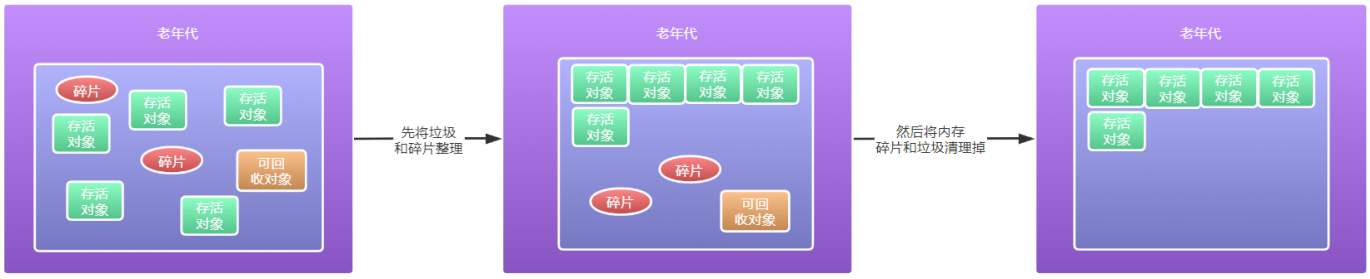
这样每次对象就会优先分配在Eden区，Eden区满了之后，触发新生代的垃圾回收，将Eden区和其中一个Survivor区中的存活对象一次性、复制到另外一个Survivor区中，然后一次性将Eden和Survivor中的内存清空，继续在Eden区中分配对象，如下图所示：



从上图我们可以看到，新生代中大多都是那些生命周期比较短的对象，一次垃圾回收之后可能存活对象就没多少了，所以移动这些存活对象的性能损耗也就比较小，但是如果老年代使用复制算法就不太适合了。

老年代对象的特点是存活时间比较长，如果要用复制算法的话，可能就会导致一次GC之后发现大量的对象都是存活的，此时你还要移动这么多的对象，性能损耗就非常大、非常耗时。

老年代更适合标记整理算法，先将存活对象移动整理起来，然后再一次性清理掉垃圾和内存碎片，如下图所示：



通过以上的分析，我们现在可以知道，JVM内存为什么划分出来了新生代和老年代，新生代存放生命周期较短的对象、采用复制算法进行垃圾回收，老年代存放生命周期较长的对象、采用标记整理算法进行垃圾回收，以及新生代对象流转到老年代的触发时机是什么都比较清晰了，下一步我们就看下GC时的一些核心机制。

4.Minor GC前–三道校验机制

当我们了解完JVM划分为新生代和老年代、他们之间对象的流转规则以及分别采用哪种复制算法，接下来就可以具体了解下GC时的一些细节了。

我们都知道对象优先分配到新生代中，基于复制算法对新生代内存划分的优化，我们又进一步的知道对象是优先分配在新生代中的Eden区，那什么时候会触发新生代的垃圾回收（Minor GC/Yong GC）呢?

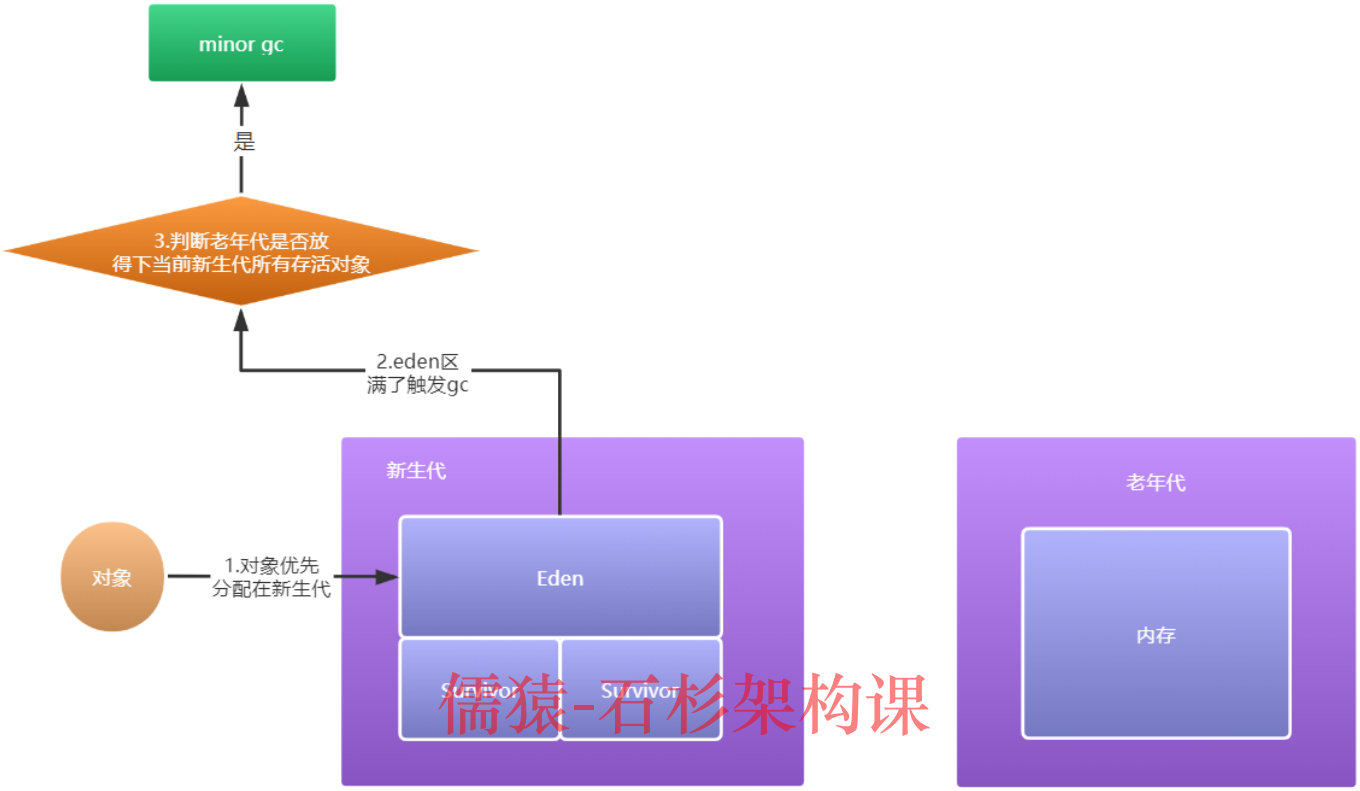
这个比较简单，就是当Eden区快要放不下的时候，就会触发新生代的垃圾回收，但是在真正执行Minor GC前，又会触发一系列的检查机制、看下当前能不能马上进行Minor GC，接下来我们继续看下这块内容。

（1）老年代能否放的下当前新生代所有存活对象

首先JVM会看一下老年代中的剩余内存，能不能放的下当前新生代中的所有存活对象。

因为运气最差的情况，就是当执行一次Minor GC之后，发现新生代中的所有对象都是存活的，此时可能Survivor区就放不下了，根据对象的流转规则，这些对象就会流转到老年代中，老年代得要保证能放的下。

如果老年代都能放得下新生代所有存活对象，就表示最坏的结果老年代这边都是能承受的，那就可以大胆的进行Minor GC了，如下图所示：

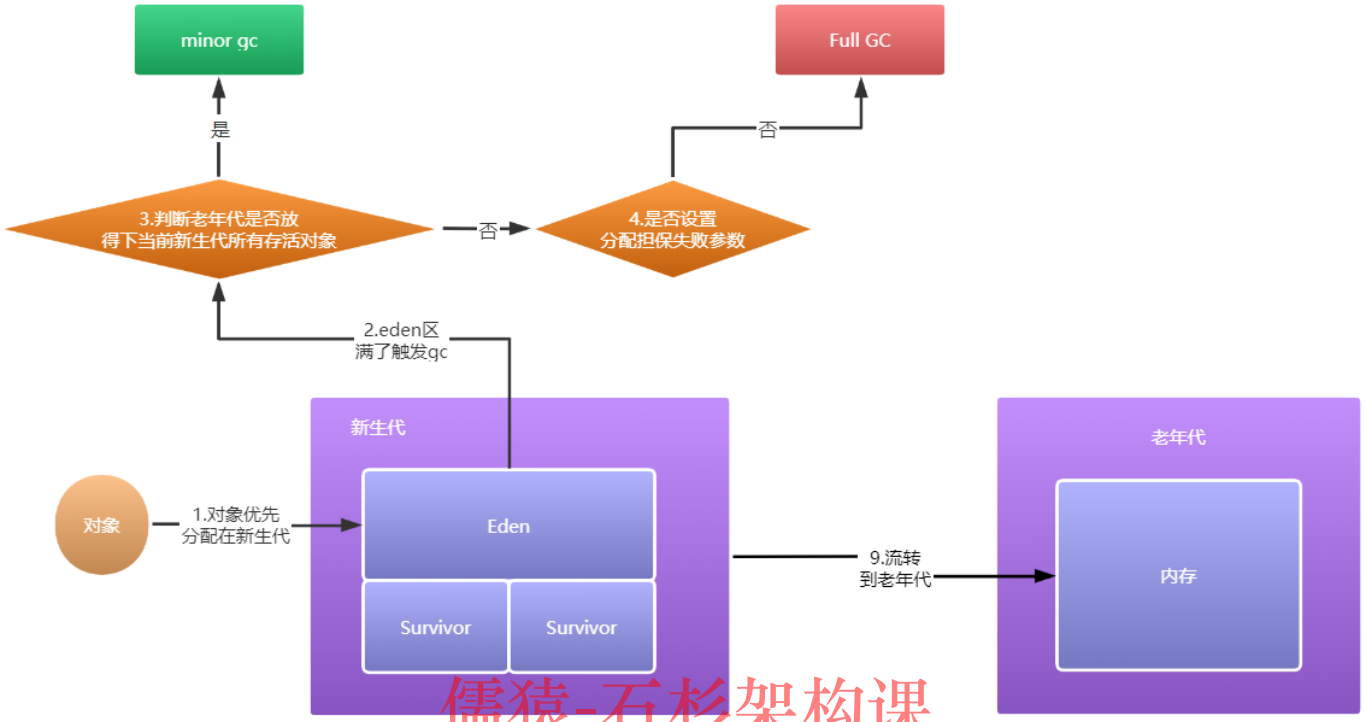


（2）是否开启分配担保失败

如果第一步的判断中，很不幸，当前老年代的可用内存比较小了，已经放不下当前新生代中所有存活的对象了，那就退一步，先问问JVM能不能跟我做一个担保。

就像你找银行借钱，银行问你能不能拿房子或其他东西做担保，同样、在JVM中就会看看分配担保的参数：-XX:-HandlePromotionFailure ，如果为false就表示没开启担保。

那就好比你回复银行说我没有什么东西可以担保的，银行当然就不会借钱给你，体现在JVM中当然就会担保失败，此时就会在Minor GC执行之前，先执行一次老年代的垃圾回收（Full GC），先让老年代腾出一些空间来，如下图所示：



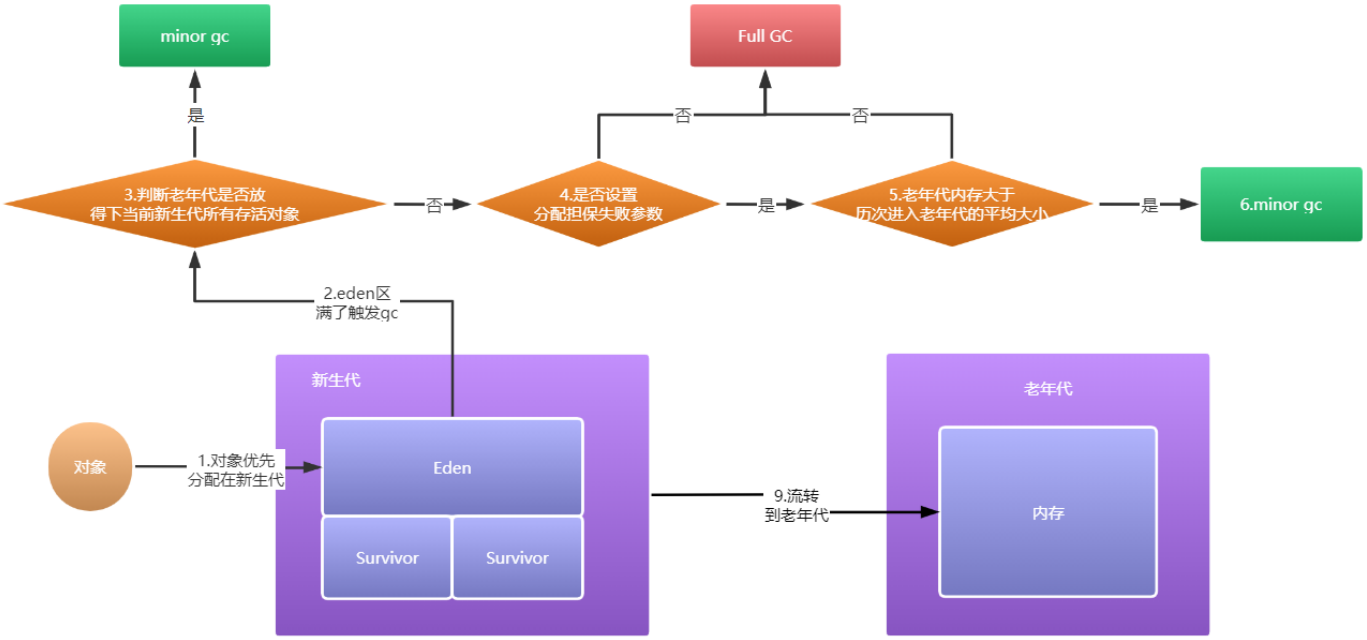
（3）历次进入老年代平均对象大小判断

紧接着上一步，如果JVM中开启了分配担保失败的配置，那么就好比你跟银行说我可以给你一点东西作为借钱的担保，此时银行就会看下你给的担保的东西够不够。

反应到JVM中就是，老年代它自己就会估摸一下，看下前面好几次进入老年代的每个对象中，他们的平均大小、我现在的剩余可用内存还能不能放的下，对吧，既然新生代中的、所有存活对象我不能一口气全放下，前面好几次进入老年代对象的平均大小能不能放的下我先看下。

如果能放的下，那好老年代就姑且赌一把：

Minor GC后，要是有对象进入老年代基本肯定也能放的下，所以含赌一把的成分的Minor GC勉强让它开始执行，如下图所示：



从以上分析我们可以看到，在Minor GC执行前，JVM也是做得够谨慎的，一不小心在Minor GC触发前就有两种情况触发一次老年代的垃圾回收。

当然这一切的目的，还是为了让无法预料到的新生代存活对象、进入老年代后还有位置可以存放，不然老年代都放不下，后果可是很严重的。

5.Minor GC后-四种垃圾回收结果

那么现在，经过上面的重重校验之后，终于可以开始执行Minor GC了。

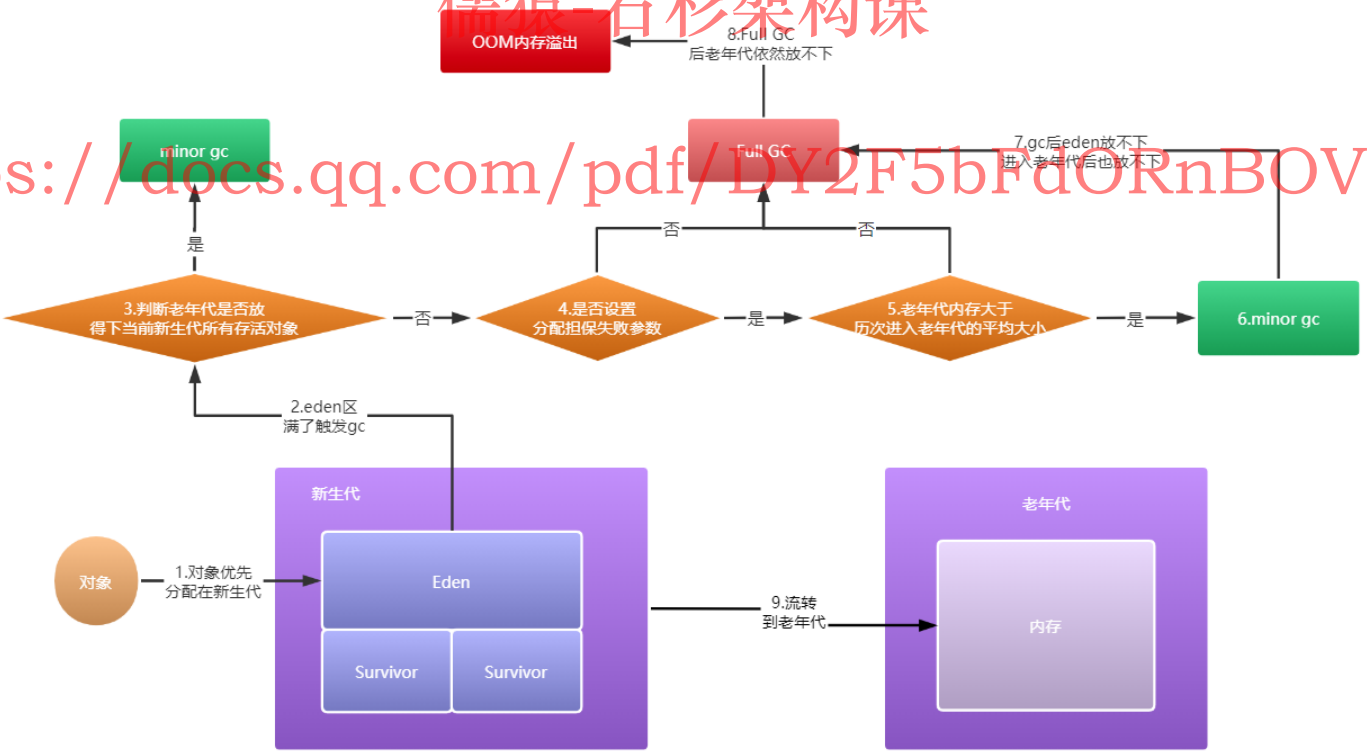
此时在新生代中，当然就通过复制算法，将Eden区以及其中一块有对象的Survivor区中的对象、拷贝到另外一块空的Survivor中，然后清空前面两块内存。

当然，这种回收结果即空闲的Survivor区能放的下所有存活对象，是我们最希望看到的一种回收结果结果。

如果Survivor区放不下存活的对象，此时就会流转到老年代中，此时如果老年代放的下，那也还行，只不过老年代对象会慢慢变多，老年代内存如果满了的话就会触发Full GC，而Full GC执行起来就会比较耗时了。

第三种回收结果，就是Survivor放不下，老年代也放不下，此时就会触发一次Full GC，先让老年代腾出一些空间来，然后再尝试着放在老年代，如果能放的下，就放进去。

最差的一种结果就是，老年代进行了一次Full GC，专门打扫了一些老年代空间后还放不下流转过来的新生代存活对象，此时JVM直接就会认为你这对象实在是太多了，直接就报OOM堆内存溢出错误了，如下图所示：



6.优化JVM参数,消灭Full GC

经过以上分析，我们知道Full GC的罪魁祸首、还是新生代中的对象频繁进入老年代，导致老年代内存不足，从而导致放不下新生代所有存活对象，此时就会导致老年代满了被迫触发Full GC，所以消灭Full GC关键中的关键，还是要避免新生代中的对象频繁进入老年代。

既然要避免新生代中的对象进入老年代，就要尽量让新生代中的存活对象在Survivor区中能放的下，大家发现没有，优化的核心不就是要控制对象的流转过程吗？而控制对象的流转过程，不就是严格控制我们之前说的那四个规则吗：对象年龄、平均年龄、大对象、GC后存活对象新生代放不下，对于这四种情况，我们具体看下如何优化。

优化-对象年龄：默认对象年龄达到15岁时，就会流转到老年代，可以适当将控制年龄的参数
-XX:MaxTenuringThreshold 调小一点，毕竟如果是新生代对象可能都躲不过几次GC就会被回收
掉了；对于那些多次GC都回收不掉，八成是属于生命周期比较长的对象，那就尽早让它进入到
老年代中吧，就不需要苦苦等到15岁了、还占内存。

优化-平均年龄：这一块主要是要适当调大一点Survivor的内存大小，可以通过调整Survivor和Eden的比例来控
制，毕竟如果Survivor内存比较小，很容易一批对象就会占到Survivor内存的50%，就会过快的
因为一个对象的年龄大于这批对象最大年龄，而触发进入老年代。

优化-大对象：　这一块，一般JVM参数设置大对象为1MB已经是很大了，除非是那种专门处理大数据量的系统，
我们可以为大对象的阈值设置几十MB，防止对象频繁进入老年代引发频繁的Full GC；但是对于
一般的系统而言1MB大小来衡量大对象已经足够了。

优化-新生代放不下：这块可以适当调大Survivor的内存，具体根据时机情况调整下，资源足够可以堆内存扩容，

资源不够可以增大新生代占堆内存的比例、或者增加Survivor占新生代的占比，尽量让每次

Minor GC后的对象都能在Survivor中能存放。

7.面试题剖析

1.java对象在JVM中是如何分配和流转的?

对象分配：对象优先分配在新生代中；

对象流转：对象优先分配在新生代中，当新生代中的对象年龄超过默认的15岁、对象年龄超过Survivor中，一批占50%内存以上的对象的最大年龄时、对象是个大对象、或者GC后新生代中的Survivor放不下了，这四种情况都会导致对象流转到老年代中。

2.什么时候会在Minor GC前触发一次 Full GC?

当Minor GC前，检查到发现老年代中的剩余内存已经放不下新生代中的所有存活对象的情况下，分配担保失败的参数没有打开、或者历次进入老年代对象的平均大小老年代都放不下了，这两种情况下就会触发一次Full GC。

3.Minor GC后触发后对应哪几种回收结果?

结果1：新生代Survivor放的下存活对象，直接放到Survivor中；

结果2：Survivor放不下，但是老年代可用内存足够大，能放的下；

结果3：Survivor放不下、老年代内存也放不下，老年代触发了一次Full GC，腾出一部分内存后，又能在老年

代放的下了；

结果4：老年代触发了一次Full GC之后，还是放不下新生代流转过来的对象，此时直接OOM报堆内存溢出错

误，这是最不幸的；
