# 介绍

对象的内存分配，从概念上讲，应该都是在堆上分配（而实际上也有可能经过即时编译后被拆散为标量类型并间接地在栈上分配）。在经典分代的设计下，新生对象通常会分配在新生代中，少数情况下（例如对象大小超过一定阈值）也可能会直接分配在老年代。对象分配的规则并不是固定的，《Java虚拟机规范》并未规定新对象的创建和存储细节，这取决于虚拟机当前使用的是哪一种垃圾收集器，以及虚拟机中与内存相关的参数的设定。

本次使用HotSpot虚拟机，以客户端模式运行。由于并未指定收集器组合，因此，本节验证的实际是使用Serial加Serial Old客户端默认收集器组合下的内存分配和回收的策略，这种配置和收集器组合也许是开发人员做研发时的默认组合（其实现在研发时很多也默认用服务端虚拟机了），但在生产环境中一般不会这样用。

# 对象优先在Eden分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC。

HotSpot虚拟机提供了-XX：+PrintGCDetails这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前的内存各区域分配情况。在实际的问题排查中，收集器日志常会打印到文件后通过工具进行分析。

在代码的testAllocation()方法中，尝试分配三个2MB大小和一个4MB大小的对象，在运行时通过-Xms20M、-Xmx20M、-Xmn10M这三个参数限制了Java堆大小为20MB，不可扩展，其中10MB分配给新生代，剩下的10MB分配给老年代。-XX：Survivor-Ratio=8决定了新生代中Eden区与一个Survivor区的空间比例是8∶1，从输出的结果也清晰地看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为9216KB（Eden区+1个Survivor区的总容量）。

|  |
| --- |
| private static final int \_1MB = 1024 \* 1024;  /\*\*  \* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8  \*/  public static void testAllocation() {  byte[] allocation1, allocation2, allocation3, allocation4;  allocation1 = new byte[2 \* \_1MB];  allocation2 = new byte[2 \* \_1MB];  allocation3 = new byte[2 \* \_1MB];  allocation4 = new byte[4 \* \_1MB]; // 出现一次Minor GC  } |

运行结果：

|  |
| --- |
| [GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426 secs] [Times: user=0.00 sys=0.0  Heap  def new generation total 9216K, used 4326K [0x029d0000, 0x033d0000, 0x033d0000)  eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)  from space 1024K, 14% used [0x032d0000, 0x032f5370, 0x033d0000)  to space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)  tenured generation total 10240K, used 6144K [0x033d0000, 0x03dd0000, 0x03dd0000)  the space 10240K, 60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)  compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)  the space 12288K, 17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)  No shared spaces configured. |

执行testAllocation()中分配allocation4对象的语句时会发生一次Minor GC，这次回收的结果是新生代6651KB变为148KB，而总内存占用量则几乎没有减少（因为allocation1、2、3三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。

产生这次垃圾收集的原因是为allocation4分配内存时，发现Eden已经被占用了6MB，剩余空间已不足以分配allocation4所需的4MB内存，因此发生Minor GC。垃圾收集期间虚拟机又发现已有的三个2MB大小的对象全部无法放入Survivor空间（Survivor空间只有1MB大小），所以只好通过分配担保机制提前转移到老年代去。

这次收集结束后，4MB的allocation4对象顺利分配在Eden中。因此程序执行完的结果是Eden占用4MB（被allocation4占用），Survivor空闲，老年代被占用6MB（被allocation1、2、3占用）。通过GC日志可以证实这一点。

# 大对象直接进入老年代

大对象就是指需要大量连续内存空间的Java对象，最典型的大对象便是那种很长的字符串，或者元素数量很庞大的数组，本节例子中的byte[]数组就是典型的大对象。大对象对虚拟机的内存分配来说就是一个不折不扣的坏消息，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，我们写程序的时候应注意避免。

在Java虚拟机中要避免大对象的原因是，在分配空间时，它容易导致内存明明还有不少空间时就提前触发垃圾收集，以获取足够的连续空间才能安置好它们，而当复制对象时，大对象就意味着高额的内存复制开销。

HotSpot虚拟机提供了-XX：PretenureSizeThreshold参数，指定大于该设置值的对象直接在老年代分配，这样做的目的就是避免在Eden区及两个Survivor区之间来回复制，产生大量的内存复制操作。

|  |
| --- |
| private static final int \_1MB = 1024 \* 1024;  /\*\*  \* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8  \* -XX:PretenureSizeThreshold=3145728  \*/  public static void testPretenureSizeThreshold() {  byte[] allocation;  allocation = new byte[4 \* \_1MB]; //直接分配在老年代中  } |

运行结果：

|  |
| --- |
| Heap  def new generation total 9216K, used 671K [0x029d0000, 0x033d0000, 0x033d0000)  eden space 8192K, 8% used [0x029d0000, 0x02a77e98, 0x031d0000)  from space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)  to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)  tenured generation total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)  the space 10240K, 40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)  compacting perm gen total 12288K, used 2107K [0x03dd0000, 0x049d0000, 0x07dd0000)  the space 12288K, 17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)  No shared spaces configured. |

执行代码中的testPretenureSizeThreshold()方法后，我们看到Eden空间几乎没有被使用，而老年代的10MB空间被使用了40%，也就是4MB的allocation对象直接就分配在老年代中，这是因为-XX：PretenureSizeThreshold被设置为3MB（就是3145728，这个参数不能与-Xmx之类的参数一样直接写3MB），因此超过3MB的对象都会直接在老年代进行分配。

注意　-XX:PretenureSizeThreshold参数只对Serial和ParNew两款新生代收集器有效，HotSpot的其他新生代收集器，如Parallel Scavenge并不支持这个参数。如果必须使用此参数进行调优，可考虑ParNew加CMS的收集器组合。

# 长期存活的对象将进入老年代

HotSpot虚拟机中多数收集器都采用了分代收集来管理堆内存，那内存回收时就必须能决策哪些存活对象应当放在新生代，哪些存活对象放在老年代中。为做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器，存储在对象头中。

对象通常在Eden区里诞生，如果经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，该对象会被移动到Survivor空间中，并且将其对象年龄设为1岁。对象在Survivor区中每熬过一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15），就会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX：MaxTenuringThreshold设置。

可以试试分别以-XX:MaxTenuringThreshold=1和-XX:MaxTenuringThreshold=15两种设置来执行代码中的testTenuringThreshold()方法，此方法中allocation1对象需要256KB内存，Survivor空间可以容纳。

当-XX：MaxTenuringThreshold=1时，allocation1对象在第二次GC发生时进入老年代，新生代已使用的内存在垃圾收集以后非常干净地变成0KB。

而当-XX：MaxTenuringThreshold=15时，第二次GC发生后，allocation1对象则还留在新生代Survivor空间，这时候新生代仍然有404KB被占用。

|  |
| --- |
| private static final int \_1MB = 1024 \* 1024;  /\*\*  \* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:Survivor  Ratio=8 -XX:MaxTenuringThreshold=1  \* -XX:+PrintTenuringDistribution  \*/  @SuppressWarnings("unused")  public static void testTenuringThreshold() {  byte[] allocation1, allocation2, allocation3;  allocation1 = new byte[\_1MB / 4]; // 什么时候进入老年代决定于XX:MaxTenuring  Threshold设置  allocation2 = new byte[4 \* \_1MB];  allocation3 = new byte[4 \* \_1MB];  allocation3 = null;  allocation3 = new byte[4 \* \_1MB];  } |

以-XX：MaxTenuringThreshold=1参数来运行的结果：

|  |
| --- |
| [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 1 (max 1)- age 1: 414664 bytes, 414664 total  : 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs] [Times: user=0.02 sys=0.00, real=0.0  [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 1 (max 1)  : 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs] [Times: user=0.00 sys=0.00, real=0.00  Heap  def new generation total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)  eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)  from space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)  to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)  tenured generation total 10240K, used 4500K [0x033d0000, 0x03dd0000, 0x03dd0000)  the space 10240K, 43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)  com\pacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)  the space 12288K, 17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)  No shared spaces configured. |

以-XX：MaxTenuringThreshold=15参数来运行的结果：

|  |
| --- |
| [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 15 (max 15)- age 1: 414664 bytes, 414664 total  : 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs] [Times: user=0.00 sys=0.00, real=0.0  [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 15 (max 15)- age 2: 414520 bytes, 414520 total  : 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs] [Times: user=0.00 sys=0.00, real=0.0  Heap  def new generation total 9216K, used 4582K [0x029d0000, 0x033d0000, 0x033d0000)  eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)  from space 1024K, 39% used [0x031d0000, 0x03235338, 0x032d0000)  to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)  tenured generation total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)  the space 10240K, 40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)  compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)  the space 12288K, 17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)  No shared spaces configured. |

# 动态对象年龄判定

为了能更好地适应不同程序的内存状况，HotSpot虚拟机并不是永远要求对象的年龄必须达到-XX：MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到-XX：MaxTenuringThreshold中要求的年龄。

执行代码中的testTenuringThreshold2()方法，并将设置-XX：MaxTenuring-Threshold=15，发现运行结果中Survivor占用仍然为0%，而老年代比预期增加了6%，也就是说allocation1、allocation2对象都直接进入了老年代，并没有等到15岁的临界年龄。因为这两个对象加起来已经到达了512KB，并且它们是同年龄的，满足同年对象达到Survivor空间一半的规则。我们只要注释掉其中一个对象的new操作，就会发现另外一个就不会晋升到老年代了。

|  |
| --- |
| private static final int \_1MB = 1024 \* 1024;  /\*\*  \* VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8  -XX:MaxTenuringThreshold=15  \* -XX:+PrintTenuringDistribution  \*/  @SuppressWarnings("unused")  public static void testTenuringThreshold2() {  byte[] allocation1, allocation2, allocation3, allocation4;  allocation1 = new byte[\_1MB / 4]; // allocation1+allocation2大于survivo空间一半  allocation2 = new byte[\_1MB / 4];  allocation3 = new byte[4 \* \_1MB];  allocation4 = new byte[4 \* \_1MB];  allocation4 = null;  allocation4 = new byte[4 \* \_1MB];  } |

运行结果：

|  |
| --- |
| [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 1 (max 15)- age 1: 676824 bytes, 676824 total  : 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs] [Times: user=0.00 sys=0.01, real=0.0  [GC [DefNew  Desired Survivor size 524288 bytes, new threshold 15 (max 15)  : 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs] [Times: user=0.00 sys=0.00, real=0.00  Heap  def new generation total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)  eden space 8192K, 51% used [0x029d0000, 0x02de4828, 0x031d0000)  from space 1024K, 0% used [0x031d0000, 0x031d0000, 0x032d0000)  to space 1024K, 0% used [0x032d0000, 0x032d0000, 0x033d0000)  tenured generation total 10240K, used 4756K [0x033d0000, 0x03dd0000, 0x03dd0000)  the space 10240K, 46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)  compacting perm gen total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)  the space 12288K, 17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)  No shared spaces configured. |

# 空间分配担保

在发生Minor GC之前，虚拟机必须先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那这一次Minor GC可以确保是安全的。

如果不成立，则虚拟机会先查看-XX：HandlePromotionFailure参数的设置值是否允许担保失败（Handle Promotion Failure）；如果允许，那会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者-XX：HandlePromotionFailure设置不允许冒险，那这时就要改为进行一次Full GC。

解释一下“冒险”是冒了什么风险：前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况——最极端的情况就是内存回收后新生代中所有对象都存活，需要老年代进行分配担保，把Survivor无法容纳的对象直接送入老年代，这与生活中贷款担保类似。

老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，但一共有多少对象会在这次回收中活下来在实际完成内存回收之前是无法明确知道的，所以只能取之前每一次回收晋升到老年代对象容量的平均大小作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。

取历史平均值来比较其实仍然是一种赌概率的解决办法，也就是说假如某次Minor GC存活后的对象突增，远远高于历史平均值的话，依然会导致担保失败。如果出现了担保失败，那就只好老老实实地重新发起一次Full GC，这样停顿时间就很长了。

虽然担保失败时绕的圈子是最大的，但通常情况下都还是会将-XX：HandlePromotionFailure开关打开，避免Full GC过于频繁。参见代码,以JDK 6 Update 24之前的HotSpot运行测试代码。

|  |
| --- |
| private static final int \_1MB = 1024 \* 1024;  /\*\*  \* VM参数：-Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:-Handle  PromotionFailure  \*/  @SuppressWarnings("unused")  public static void testHandlePromotion() {  byte[] allocation1, allocation2, allocation3, allocation4, allocation5, alloca-tion6, allocation7;  allocation1 = new byte[2 \* \_1MB];  allocation2 = new byte[2 \* \_1MB];  allocation3 = new byte[2 \* \_1MB];  allocation1 = null;  allocation4 = new byte[2 \* \_1MB];  allocation5 = new byte[2 \* \_1MB];  allocation6 = new byte[2 \* \_1MB];  allocation4 = null;  allocation5 = null;  allocation6 = null;  allocation7 = new byte[2 \* \_1MB];  } |

以-XX：HandlePromotionFailure=false参数来运行的结果：

|  |
| --- |
| [GC [DefNew: 6651K->148K(9216K), 0.0078936 secs] 6651K->4244K(19456K), 0.0079192 secs] [Times: user=0.00 sys=0.0  [GC [DefNew: 6378K->6378K(9216K), 0.0000206 secs][Tenured: 4096K->4244K(10240K), 0.0042901 secs] 10474K->4244K(1 |

以-XX：HandlePromotionFailure=true参数来运行的结果：

|  |
| --- |
| [GC [DefNew: 6651K->148K(9216K), 0.0054913 secs] 6651K->4244K(19456K), 0.0055327 secs] [Times: user=0.00 sys=0.0  [GC [DefNew: 6378K->148K(9216K), 0.0006584 secs] 10474K->4244K(19456K), 0.0006857 secs] [Times: user=0.00 sys=0. |

在JDK 6 Update 24之后，这个测试结果就有了差异，-XX：HandlePromotionFailure参数不会再影响到虚拟机的空间分配担保策略，观察OpenJDK中的源码变化，虽然源码中还定义了-XX：HandlePromotionFailure参数，但是在实际虚拟机中已经不会再使用它。

JDK 6 Update 24之后的规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小，就会进行Minor GC，否则将进行Full GC。

HotSpot中空间分配检查的代码片段:

|  |
| --- |
| bool TenuredGeneration::promotion\_attempt\_is\_safe(size\_t  max\_promotion\_in\_bytes) const {  // 老年代最大可用的连续空间  size\_t available = max\_contiguous\_available();  // 每次晋升到老年代的平均大小  size\_t av\_promo = (size\_t)gc\_stats()->avg\_promoted()->padded\_average();  // 老年代可用空间是否大于平均晋升大小，或者老年代可用空间是否大于当此GC时新生代所有对象容量  bool res = (available >= av\_promo) || (available >= max\_promotion\_in\_bytes);  return res;  } |