java垃圾回收机制

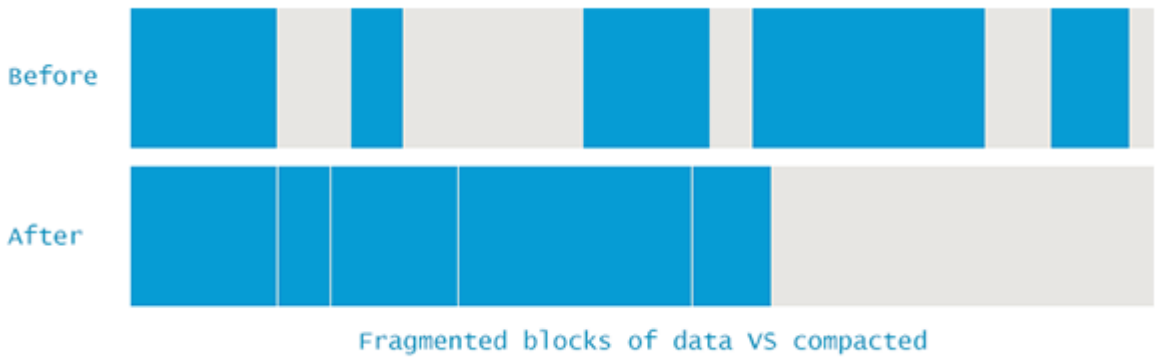
标记清除算法介绍最主要的理论算法之一，在实践过程中，为了真实情景需要，需要许多调整。举一个简单例子，我们检查JVM需要做的各种事情，以便我们安全地去创建对象。

清除压缩

当清除期间，JVM必须确保区域被不可达对象填充。这会(终将会)导致内存碎片化，同样会导致磁盘碎片化，由此产生两个问题：

- 写操作因为寻找下一个足够尺寸的空间变得耗费时间，这个写操作不再简单。
- 当创建新对象的时候，JVM分配一个连续的空间。如果内存碎片遍布每一个点，没有足够的空间容纳新创建的对象，分配就会发生错误。

为了避免上面的问题，JVM会确保碎片化不会失控。因此不会仅仅标记清除，垃圾回收期间，"内存整理"进程同时在工作。这个进程重新分配所有的可达对象让他们紧密排列，消除（或者减少）碎片。下面是示意图：



分代假设

正如我们之前提到，垃圾收集会引起应用彻底停顿。对象越多回收垃圾花费的时间越长，这是显而易见的。如果我们把使用一小块内存工作变成可能，那又会怎样呢？为了研究这种可行性，一些专家发现应用的大多数内存分为以下两种情况：

- 绝大多数对象很快就成为无用对象。
- 很少部分对象讲过很长时间存活下来。

上面的观察结果归类于新生代假设。基于这个假设：VM内存空间被划分为Young代 和Old代，后者有时也叫做Tenured。

众多算法在提升GC性能上已经取得进展，使得拥有这样一个独立易清除的内存区域变成现实。

这种方式虽说不上毫无问题。当垃圾收集器收集一个分代中的对象的时候，不同分代中的对象彼此相互引用的时，实际上被当作"GC roots"。

但是更更要的一点是，分代假设并不适用于一些应用。自此，因为那些“夭折”和“有可能永生”的对象,GC算法做了优化，JVM对那些期待更久生命的对象表现得友好。

内存空间

读者应该了解下图中java堆区里面的内存划分。不同内存区域的垃圾收集机制不辣么容易理解。应该注意，不同GC算法实现细节可能不同，然而它们的理念是一致的。

Eden

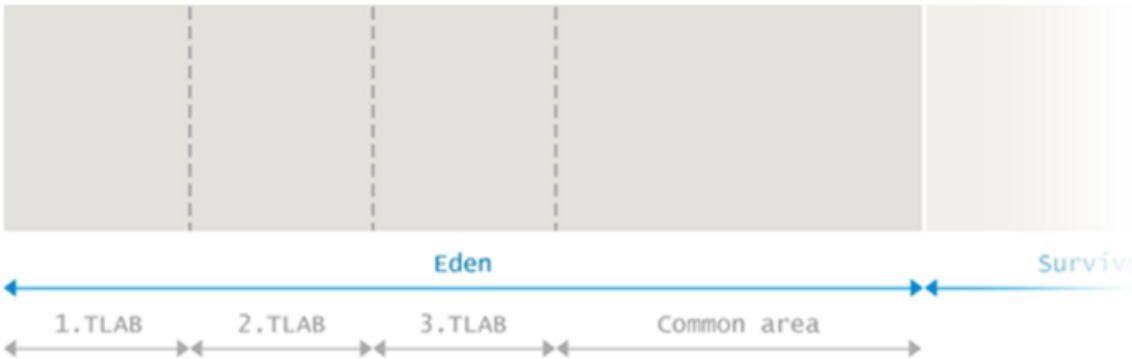
Eden区是对象被新创建的时候分配的内存区域。在这个区域中，多线程可以同时创建多个对象。在Eden区里，Eden 被分成一个或多个Thread Local Allocation Buffer (缩写：TLAB)。在这些缓存里，JVM允许线程在对应的缓存中分配绝大多数的对象，避免昂贵的多线程同步。

当TLAB中不能分配空间时（因为空间不足），JVM会移到共享的Eden区去分配，如果共享Eden空间也不足时，Young代中垃圾回收器去释放更多的空间。如果在GC之后还没有足够的空间以供使用，对象会在Old代中分配。

当Eden回收期间，GC把所有的可达对象标记为存活。

我们已经提前注意到，对象可以跨代关联，因此我们必须有一个快捷途径去检查其他代的对象引用Eden区中的对象。

从一开始就记录所有的分代引用是不可取的， JVM有自己的机制：卡标记。事实上，JVM仅仅标记Eden里面有可能Old代引用Eden代的对象的“脏”对象的位置，你可以在Nitsan的博客里了解更多的信息。

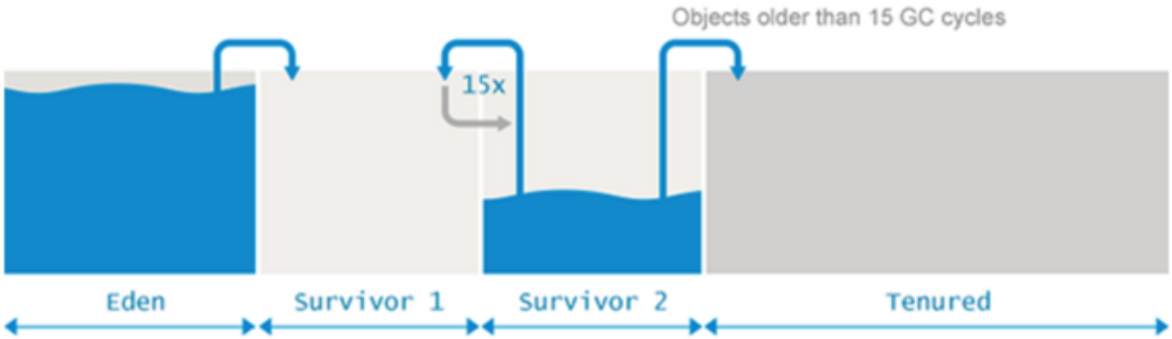


标记阶段完成之后，Eden区下面所有存活的对象被复制到Survivor下面的一块区域中。现在，Eden区被清空，重新分配新建对象。正如“标记-复制”的名称一样：存活的对象被标记，然后复制(不是移动)到Survivor区。

Survivor区

紧邻Eden区的下一个区域时两个叫做from和to的Survivor区。需要注意的一点，两块区域中的一块是空的。

Survivor中的空的区域会保存下一刻Young代中垃圾回收后的对象。Young代所有存活的对象（包括Eden区和Survior区中非空的from区域）被复制到Survivor区的"to"区域。在这之后，“to”区域存放所有对象，“from”区域清空。两者进行调换（译者注：即from变成to,to变成from）。



在两个区域进行数次复制存活对象操作，直到一些对象足够成熟（“old enough”）。记住这一点，基于分代假设，一些对象在数次GC之后存活下来，而且在很长一段时间内继续被引用。

这样的“tenured”对象会升级到Old代。这种情况发生的时候，对象不再从Survuvor区一个区域移动到另外一个区域，而是进入Old区，在成为不可达对象之前它们一直存在在old区中。

为了确定哪些对象“old enough”,需要为old区提供一种算法，GC记录幸存对象的详细信息。每代GC完成之后，那些依然存活的对象年龄进行增长。每当年龄超过设定的阀值之后，对象才会被升迁到old区。

实际上阀值被JVM动态设定，-XX:+MaxTenuringThreshold 除外，它设置最高限定。XX:+MaxTenuringThreshold=0表示跳过Survivor区的两个区域之间复制过程直接进入old区。jvm默认阀值是GC循环15次。在Hotspot是最大值。

Survivor区空间不足以容纳Young代所有存活对象的时候升迁操作被提前触发。

Old代

Old代的具体实现细节巨复杂。Old代通常被那些几乎不可能被当作垃圾的对象占据。

Old代GC触发的次数比Young代少。因此，Old代中的存活对象，不会发生标记复制过程。相反，这些对象保持最小碎片化。这种算法建立在不同维度之上。大体上，分为以下几步：

- 通过设置所有GC roots可达的对象标记位来标记所有可达对象。
- 删除所有的不可达对象。
- 通过复制对象并紧密排列在Old区的顶端压缩Old区的空间。

正如你看到上面描述的那样，Old代的GC必须明确处理压缩错左避免过多的碎片。

PermGen

JAVA 8之前中被称作“Permanent Generation”的特殊区域。这是以前存放例如class的metadata。而，Permgen还存储String之类的额外数据。实际上为JAVA开发者添加了许多麻烦，因为很难预测到底需要多少的空间。这些错误的预测结果表现形式为java.lang.OutOfMemoryError: Permgen space。除非是类似OutOfMemoryError的原因是真的是因为内存泄漏，解决这种问题的简单方法是增加permgen尺寸。下图中设置permgen尺寸的最大值为256M：

```
java -XX:MaxPermSize=256m
```

Metaspace

正如预测metadata是一件纷繁复杂的事情那样，JAVA 8移除了Permanent区，换作Metaspace。从那时起，绝大多数复杂的事情都被移到Java heap区。

类定义文件，现在都存入叫做“Metaspace”的区域中。他相当于本地内存的一块区域。理论上，Metaspace尺寸仅仅受限于JAVA进程可获得本地内存大小。将JAVA开发人员从仅仅在应用多增加一个类就造成java.lang.OutOfMemoryError: Permgen space的困境中解脱出来。需要注意的是这个看起来不受限制没有损失的空间-让Metaspace无限制的增长你会引起内存重交换或者/和本地内存分配失败。

某些场合你希望保护自己，你可以如下图所示限制Metaspace增长，Metaspace尺寸限制在265M：

```
java -XX:MaxMetaspaceSize=256m
```