[译]Java8官方GC调优指南 --(三) 分代 - 掘金

juejin.cn/post/6844904053529378824

2020年1月28日

> 本套文章是Java8官方GC调优指南的全文翻译，点击查看原文,原文章名称《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》

3 Generations 分代

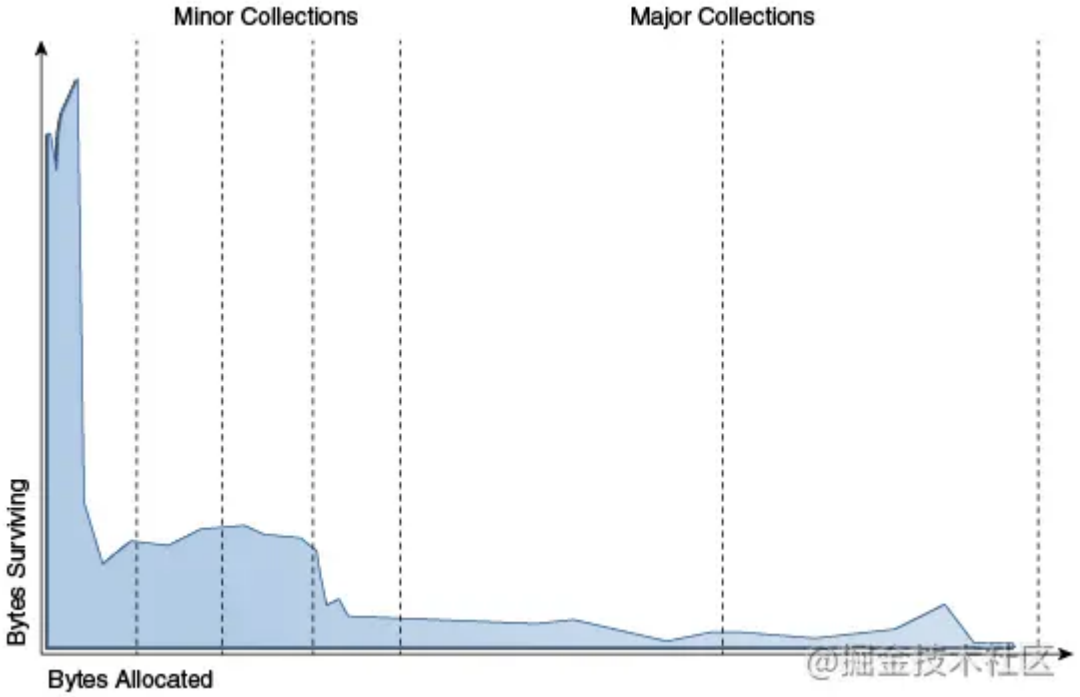
简介

Java SE Platform的一个优势就是它让开发者无需了解内存分配就能上手开发(**其实完全不懂也不行**)。 但是，当垃圾回收器已经成为Java程序的主要瓶颈时，了解其内部实现是非常有用的。

垃圾回收器会处理那些已经没有任何指针指向的对象。最简单的垃圾回收算法迭代所有可达的对象。剩余不可达的对象就是可回收对象。GC时间主要取决于存活对象的数量，如果是大型服务端应用，存活对象超多，最好别用这种回收器。

JVM使用分代回收策略(generational colletion)，这种策略包括了许多不同的垃圾回收算法。 最原始的垃圾回收检查heap中的所有存活对象，分代回收策略开发多个假设和观察属性去减少回收对象的工作量。 这些观察属性中最重要的就是弱代猜想，这个猜想标记了大多数对象只存活一小段时间。

下图中蓝色区域是存活的对象的典型分布。X轴是存活的对象数量，Y轴是对象使用的总字节数。图中左侧的垂直向下的 peak代表那些分配很快即可回收的对象。例如，**for循环中创建的对象，生命周期就是那一次循环，循环结束对象就可以被收集了**。



有些对象却寿命久的多，所以分布图一直平滑延伸到了右侧。这些对象从服务启动之后就分配了，直到整个进程退出。对于一些Java程序来说，它的对象存活分布图可能跟上图是完全不同的。不过很令人惊讶的是很多应用的分布图都和这个图相差无几。**专注于回收那些“存活时间短的对象”的回收策略才是最有效的**。

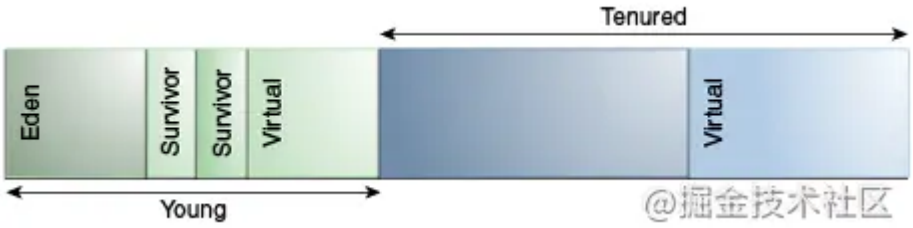
要让这个场景更加乐观，内存也要分代管理的。当某个代的内存满了，这个代就开始垃圾回收。大量的对象都是短命对象。当young generation满了，就触发一次minor collection。其他代的垃圾对象不会被回收。minor collection的最理想情况是，假定所有短命对象都在young区，那就直接回收了。minor collection的回收成本在于存活对象占总对象的比例；如果young区的对象全死了，那么回收起来会很快。一般情况下，存活对象中的一部分在每次 minor gc后会移动到tenured generation。最终，tenured generation 会被装满也就是整个heap都满了，触发major collection。Major collection通常会持续更久的时间，因为大量的对象都要被参与回收。

在Ergonomics章节提到，jvm会为不同的Java应用选择垃圾回收器动态的提供不错的性能。serial garbage collector和它的默认参数是给那些小型Java应用设计的。parallel 或者吞吐量 garbage collector是用来给中型应用设计的。heap size参数和一些附加的特性参数是为了给服务器级别的应用设计的。这些擦书大多数都工作的很好，但也不一定。(这不是废话么，以前听某厂大牛讲课说JVM原生的回收器就是垃圾，不过一般小厂也没那么大的堆。Hey Garbage Collector，why don't you collect yourself?) 下面是这个文档的中心准则：

如果垃圾回收器成为了系统瓶颈，你最好自定义每个分代的heap size。检查gc日志来寻找属于你的最佳参数。

> 笔者认为上面这句话诠释了GC调优的本质，也就是说，调优GC参数，需要根据程序的特点来调整堆大小，因为堆的大小一定程度决定了GC的停顿时间，还要了解所有可选的JVM性能参数，结合自己程序运行的物理环境，调整参数，观察GC日志来确保吞吐量和停顿时间符合要求。如何才能称这两个目标都符合要求呢？如何制定目标？这个就要看自己了，我们后面再详细说，先把本套文档看完。

下图表示了默认的分代策略(对于所有的回收器的使用，除了parallel collector和G1)



Java应用初始化时，最大的地址空间只是预定了一下但是并没有真实分配。完整的内存地址空间可以被分为young区和tenured区。

young区包含1个eden区和两个suvivor区。大多数object都在eden区分配。任意时刻都有一个survivor区是空的，eden区存活对象GC后会转移到这个survivor区；另一个survivor区是给下次复制收集来用的。对象会在两个s区来回复制，直到它们年龄足够老，可以晋升到tenured区。

Performance Considerations 性能因素

对于垃圾回收器的性能，有两个主要的测量方式：

- 吞吐量，除GC消耗时间之外的应用运行时间百分比。吞吐量包含内存分配时间。
- 停顿次数，GC触发STW时程序是无响应的。

> 萌新提示：只要触发了STW，程序就会暂停，必须等GC结束才能继续响应。

用户对于垃圾回收器有不同的需求。例如，某个策略：一个web服务最佳的优化目标是吞吐量优先，因为GC回收造成的停顿可能会被网络延迟掩盖。不过，在一个图形化程序中，即使是短暂的停顿也会影响用户体验。

还有一些用户对些其他的因素比较敏感。Footprint是一个进程的内存占用情况，通过内存页和缓存行来度量。在给定物理内存和CPU核数的操作系统上，Footprint决定了应用的可伸缩性。

Promptness(敏捷性)是分布式系统非常重要的一个性能因素，包括RMI，指的是对象死亡和内存可用之间的间隔时间。

通常，给特定的分代指定大小是在这些因素之间做权衡。例如，一个非常大的young区可能会提高吞吐量，但是也会提高footprint，promptness和停顿时间。young区停顿时间可以通过调小young区大小来降低。一个分代的大小不会影响其他分代的回收频率和停顿时间。

没有一个简单正确的方式来决定一个分代的大小。最好的选择来自于应用使用内存的方式和用户的需求。JVM的默认回收器和参数不一定是最优的，你可能需要通过参数调整来获取最佳的效果。具体可以查看章节Sizing the Generations。

Measurement 度量

使用特定于应用程序的指标可以很好的度量Throughput(吞吐量) 和 Footprint（内存占用）。例如，一个web服务的吞吐量可以通过客户端压力测试来获取，服务器的footprint可以通过pmap命令来查看。GC停顿信息也可以通过JVM输出日志来查看。

参数 `-verbose:gc` 可以在每次GC后输出heap和垃圾回收的信息。例如，这是一条服务器的GC日志：

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
复制代码
```

日志标明两次minor collection和一次major collection。箭头左右的两个数字(325407K->83000K)表示gc前后的内存空间。在minor collection之后，这个空间中有一些垃圾对象，但是内存空间并没有释放。这些对象要么在tenured generation，要么就是被tenured generation的对象引用了。

圆括号中的数字，776768K是已经提交的heap大小：操作系统已经分配给Java创建对象的空间大小。注意这个数字值包含一个survivor区的大小。除了垃圾回收本身，只有一个survivor区会被用来储存对象。

最后一项，0.2300771 secs表示执行GC花费的时间。

第三行的Full GC日志跟上面的Young GC差不多。

注意，`-verbose:gc`输出的日志格式在将来的release版本中可能会更改

`-XX:+PrintGCDetails`参数会多打些日志。下面是例子：

```
[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K(261184K),
0.0459067 secs]
复制代码
```

这表明minor GC回收了young区98%的空间，`DefNew: 64575K->959K(64576K)`执行消耗了`0.0457646 secs`。

全部heap的使用率降低到了大约51%左右(`196016K->133633K(261184K)`)，总时间会比minor gc时间稍微高一点。

注意，`-XX:+PrintGCDetails`输出的日志格式在将来的release版本中可能会更改

`-XX:+PrintGCTimeStamps`参数为每次GC增加了一个时间戳。这个在观察gc频率的时候会非常有用。

```
111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505 secs]111.042:
[Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K), 0.1293306
secs]
复制代码
```

这次回收发生在Java程序运行至111秒时。minor gc差不多在同一时刻开始执行。更多的，这个log还显示了Tenured gc。tenured generation的使用率减少到了10%(`18154K->2311K(24576K))`)，花费了`0.1293306 secs`，大约130毫秒。