从实际案例聊聊Java应用的GC优化

From: 录录 美团技术团队 2017-12-28

当Java程序性能达不到既定目标，且其他优化手段都已经穷尽时，通常需要调整垃圾回收器来进一步提高性能，称为GC优化。但GC算法复杂，影响GC性能的参数众多，且参数调整又依赖于应用各自的特点，这些因素很大程度上增加了GC优化的难度。

即便如此，GC调优也不是无章可循，仍然有一些通用的思考方法。本篇会介绍这些通用的GC优化策略和相关实践案例，主要包括如下内容：

优化前准备: 简单回顾JVM相关知识、介绍GC优化的一些通用策略。
优化方法: 介绍调优的一般流程：明确优化目标→优化→跟踪优化结果。
优化案例: 简述笔者所在团队遇到的GC问题以及优化方案。

优化前的准备

GC优化需知

为了更好地理解本篇所介绍的内容，你需要了解如下内容。

1. GC相关基础知识，包括但不限于：

a) GC工作原理。
b) 理解新生代、老年代、晋升等术语含义。
c) 可以看懂GC日志。

2. GC优化不能解决一切性能问题，它是最后的调优手段。

如果对第一点中提及的知识点不是很熟悉，可以先阅读小结-JVM基础回顾；如果已经很熟悉，可以跳过该节直接往下阅读。

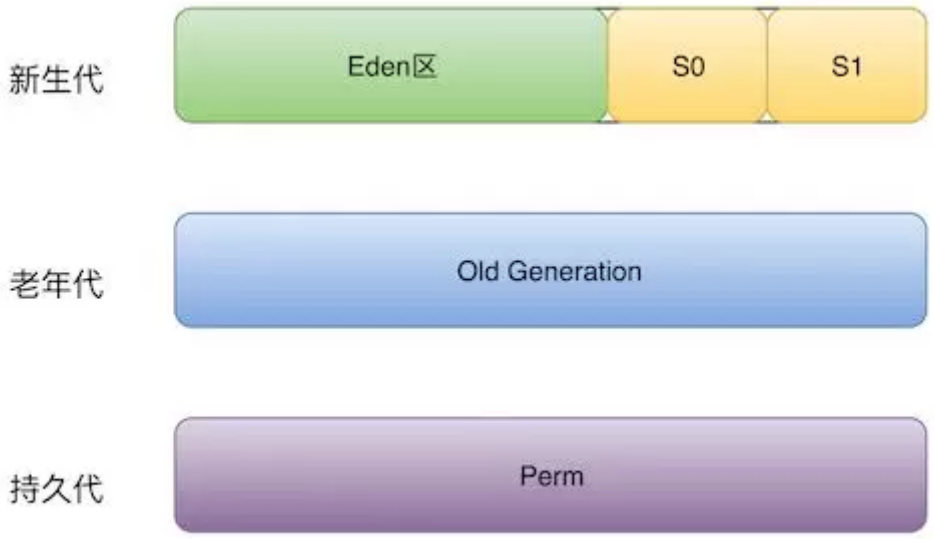
JVM基础回顾

JVM内存结构

简单介绍一下JVM内存结构和常见的垃圾回收器。

当代主流虚拟机（Hotspot VM）的垃圾回收都采用“分代回收”的算法。“分代回收”是基于这样一个事实：对象的生命周期不同，所以针对不同生命周期的对象可以采取不同的回收方式，以便提高回收效率。

Hotspot VM将堆划分为不同的物理区，就是“分代”思想的体现。如图所示，JVM堆主要由新生代、老年代、永久代构成。



1. 新生代（Young Generation）：大多数对象在新生代中被创建，其中很多对象的生命周期很短。每次新生代的垃圾回收（又称Minor GC）后只有少量对象存活，所以选用复制算法，只需要少量的复制成本就可以完成回收。

新生代内又分三个区：一个Eden区，两个Survivor区（一般而言），大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到两个Survivor区（中的一个）。当这个Survivor区满时，此区的存活且不满足“晋升”条件的对象将被复制到另外一个Survivor区。

对象每经历一次Minor GC，年龄加1，达到“晋升年龄阈值”后，被放到老年代，这个过程也称为“晋升”。显然，“晋升年龄阈值”的大小直接影响着对象在新生代中的停留时间，在Serial和ParNew GC两种回收器中，“晋升年龄阈值”通过参数MaxTenuringThreshold设定，默认值为15。

2. 老年代（Old Generation）：在新生代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代，该区域中对象存活率高。老年代的垃圾回收（又称Major GC）通常使用“标记-清理”或“标记-整理”算法。整堆包括新生代和老年代的垃圾回收称为Full GC（HotSpot VM里，除了CMS之外，其它能收集老年代的GC都会同时收集整个GC堆，包括新生代）。

3. 永久代（Perm Generation）：主要存放元数据，例如Class、Method的元信息，与垃圾回收要回收的Java对象关系不大。相对于新生代和年老代来说，该区域的划分对垃圾回收影响比较小。

常见垃圾回收器

不同的垃圾回收器，适用于不同的场景。常用的垃圾回收器：

- 串行（Serial）回收器是单线程的一个回收器，简单、易实现、效率高。
- 并行（ParNew）回收器是Serial的多线程版，可以充分的利用CPU资源，减少回收的时间。
- 吞吐量优先（Parallel Scavenge）回收器，侧重于吞吐量的控制。

- 并发标记清除（CMS，Concurrent Mark Sweep）回收器是一种以获取最短回收停顿时间为目标的回收器，该回收器是基于“标记-清除”算法实现的。

GC日志

每一种回收器的日志格式都是由其自身的实现决定的，换而言之，每种回收器的日志格式都可以不一样。但虚拟机设计者为了方便用户阅读，将各个回收器的日志都维持一定的共性。JavaGC日志 中简单介绍了这些共性。

参数基本策略

各分区的大小对GC的性能影响很大。如何将各分区调整到合适的大小，分析活跃数据的大小是很好的切入点。

活跃数据的大小是指，应用程序稳定运行时长期存活对象在堆中占用的空间大小，也就是Full GC后堆中老年代占用空间的大小。可以通过GC日志中Full GC之后老年代数据大小得出，比较准确的方法是在程序稳定后，多次获取GC数据，通过取平均值的方式计算活跃数据的大小。活跃数据和各分区之间的比例关系如下（见参考文献1）：

例如，根据GC日志获得老年代的活跃数据大小为300MB，那么各分区大小可以设为：

总堆：1200MB = 300MB × 4

新生代：450MB = 300MB × 1.5

老年代： 750MB = 1200MB - 450MB*

这部分设置仅仅是堆大小的初始值，后面的优化中，可能会调整这些值，具体情况取决于应用程序的特性和需求。

优化步骤

GC优化一般步骤可以概括为：确定目标、优化参数、验收结果。

确定目标

明确应用程序的系统需求是性能优化的基础，系统的需求是指应用程序运行时某方面的要求，譬如：

- 高可用，可用性达到几个9。
- 低延迟，请求必须多少毫秒内完成响应。
- 高吞吐，每秒完成多少次事务。

明确系统需求之所以重要，是因为上述性能指标间可能冲突。比如通常情况下，缩小延迟的代价是降低吞吐量或者消耗更多的内存或者两者同时发生。

由于笔者所在团队主要关注高可用和低延迟两项指标，所以接下来分析，如何量化GC时间和频率对于响应时间和可用性的影响。通过这个量化指标，可以计算出当前GC情况对服务的影响，也能评估出GC优化后对响应时间的收益，这两点对于低延迟服务很重要。

举例：假设单位时间T内发生一次持续25ms的GC，接口平均响应时间为50ms，且请求均匀到达，根据下图所示：

那么有(50ms+25ms)/T比例的请求会受GC影响，其中GC前的50ms内到达的请求都会增加25ms，GC期间的25ms内到达的请求，会增加0-25ms不等，如果时间T内发生N次GC，受GC影响请求占比=(接口响应时间+GC时间)×N/T 。可见无论降低单次GC时间还是降低GC次数N都可以有效减少GC对响应时间的影响。

优化

通过收集GC信息，结合系统需求，确定优化方案，例如选用合适的GC回收器、重新设置内存比例、调整JVM参数等。

进行调整后，将不同的优化方案分别应用到多台机器上，然后比较这些机器上GC的性能差异，有针对性的做出选择，再通过不断的试验和观察，找到最合适的参数。

验收优化结果

将修改应用到所有服务器，判断优化结果是否符合预期，总结相关经验。

接下来，我们通过三个案例来实践以上的优化流程和基本原则（本文中三个案例使用的垃圾回收器均为ParNew+CMS，CMS失败时Serial Old替补）。

GC优化案例

案例一：Major GC和Minor GC频繁

确定目标

服务情况：Minor GC每分钟100次，Major GC每4分钟一次，单次Minor GC耗时25ms，单次Major GC耗时200ms，接口响应时间50ms。

由于这个服务要求低延时高可用，结合上文中提到的GC对服务响应时间的影响，计算可知由于Minor GC的发生，12.5%的请求响应时间会增加，其中8.3%的请求响应时间会增加25ms，可见当前GC情况对响应时间影响较大。

（50ms+25ms）× 100次/60000ms = 12.5%，50ms × 100次/60000ms = 8.3%。

优化目标：降低TP99、TP90时间。

优化

首先优化Minor GC频繁问题。通常情况下，由于新生代空间较小，Eden区很快被填满，就会导致频繁Minor GC，因此可以通过增大新生代空间来降低Minor GC的频率。例如在相同的内存分配率的前提下，新生代中的Eden区增加一倍，Minor GC的次数就会减少一半。

这时很多人有这样的疑问，扩容Eden区虽然可以减少Minor GC的次数，但会增加单次Minor GC时间么？根据上面公式，如果单次Minor GC时间也增加，很难保证最后的优化效果。我们结合下面情况来分析，单次Minor GC时间主要受哪些因素影响？是否和新生代大小存在线性关系？

首先，单次Minor GC时间由以下两部分组成：T1（扫描新生代）和T2（复制存活对象到Survivor区）如下图。（注：这里为了简化问题，我们认为T1只扫描新生代判断对象是否存活的时间，其实该阶段还需要扫描部分老年代，后面案例中有详细描述。）

- 扩容前：新生代容量为R，假设对象A的存活时间为750ms，Minor GC间隔500ms，那么本次Minor GC时间= T1（扫描新生代R）+T2（复制对象A到S）。
- 扩容后：新生代容量为2R，对象A的生命周期为750ms，那么Minor GC间隔增加为1000ms，此时Minor GC对象A已不再存活，不需要把它复制到Survivor区，那么本次GC时间 = 2 × T1（扫描新生代R），没有T2复制时间。

可见，扩容后，Minor GC时增加了T1（扫描时间），但省去T2（复制对象）的时间，更重要的是对于虚拟机来说，复制对象的成本要远高于扫描成本，所以，单次Minor GC时间更多取决于GC后存活对象的数量，而非Eden区的大小。因此如果堆中短期对象很多，那么扩容新生代，单次

Minor GC时间不会显著增加。下面需要确认下服务中对象的生命周期分布情况：

通过上图GC日志中两处红色框标记内容可知：

1. new threshold = 2（动态年龄判断，对象的晋升年龄阈值为2），对象仅经历2次Minor GC后就晋升到老年代，这样老年代会迅速被填满，直接导致了频繁的Major GC。
2. Major GC后老年代使用空间为300Mb+，意味着此时绝大多数(86% = 2G/2.3G)的对象已经不再存活，也就是说生命周期长的对象占比很小。

由此可见，服务中存在大量短期临时对象，扩容新生代空间后，Minor GC频率降低，对象在新生代得到充分回收，只有生命周期长的对象才进入老年代。这样老年代增速变慢，Major GC频率自然也会降低。

优化结果

通过扩容新生代为为原来的三倍，单次Minor GC时间增加小于5ms，频率下降了60%，服务响应时间TP90，TP99都下降了10ms+，服务可用性得到提升。

调整前：

调整后：

小结

如何选择各分区大小应该依赖应用程序中对象生命周期的分布情况：如果应用存在大量的短期对象，应该选择较大的年轻代；如果存在相对较多的持久对象，老年代应该适当增大。

更多思考

关于上文中提到晋升年龄阈值为2，很多同学有疑问，为什么设置了MaxTenuringThreshold=15，对象仍然仅经历2次Minor GC，就晋升到

老年代？这里涉及到“动态年龄计算”的概念。

动态年龄计算：Hotspot遍历所有对象时，按照年龄从小到大对其所占用的大小进行累积，当累积的某个年龄大小超过了survivor区的一半时，取这个年龄和MaxTenuringThreshold中更小的一个值，作为新的晋升年龄阈值。在本案例中，调优前：Survivor区 = 64M，desired survivor = 32M，此时Survivor区中age<=2的对象累计大小为41M，41M大于32M，所以晋升年龄阈值被设置为2，下次Minor GC时将年龄超过2的对象被晋升到老年代。

JVM引入动态年龄计算，主要基于如下两点考虑：

1. 如果固定按照MaxTenuringThreshold设定的阈值作为晋升条件：

a）MaxTenuringThreshold设置的过大，原本应该晋升的对象一直停留在Survivor区，直到Survivor区溢出，一旦溢出发生，Eden+Svuvivor中对象将不再依据年龄全部提升到老年代，这样对象老化的机制就失效了。

b）MaxTenuringThreshold设置的过小，“过早晋升”即对象不能在新生代充分被回收，大量短期对象被晋升到老年代，老年代空间迅速增长，引起频繁的Major GC。分代回收失去了意义，严重影响GC性能。

2. 相同应用在不同时间的表现不同：特殊任务的执行或者流量成分的变化，都会导致对象的生命周期分布发生波动，那么固定的阈值设定，因为无法动态适应变化，会造成和上面相同的问题。

总结来说，为了更好的适应不同程序的内存情况，虚拟机并不总是要求对象年龄必须达到Maxtenuringthreshhold再晋级老年代。

案例二：请求高峰期发生GC，导致服务可用性下降

确定目标

GC日志显示，高峰期CMS在重标记（Remark）阶段耗时1.39s。Remark阶段是Stop-The-World（以下简称为STW）的，即在执行垃圾回收时，Java应用程序中除了垃圾回收器线程之外其他所有线程都被挂起，意味着在此期间，用户正常工作的线程全部被暂停下来，这是低延时服务不能接受的。本次优化目标是降低Remark时间。

优化

解决问题前，先回顾一下CMS的四个主要阶段，以及各个阶段的工作内容。下图展示了CMS各个阶段可以标记的对象，用不同颜色区分。

1. Init-mark初始标记(STW)，该阶段进行可达性分析，标记GC ROOT能直接关联到的对象，所以很快。
2. Concurrent-mark并发标记，由前阶段标记过的绿色对象出发，所有可到达的对象都在本阶段中标记。
3. Remark重标记(STW)，暂停所有用户线程，重新扫描堆中的对象，进行可达性分析，标记活着的对象。因为并发标记阶段是和用户线程并发执行的过程，所以该过程中可能有用户线程修改某些活跃对象的字段，指向了一个未标记过的对象，如下图中红色对象在并发标记开始时不可达，但是并行期间引用发生变化，变为对象可达，这个阶段需要重新标记出此类对象，防止在下一阶段被清理掉，这个过程也是需要STW的。特别需要注意一点，这个阶段是以新生代中对象为根来判断对象是否存活的。
4. 并发清理，进行并发的垃圾清理。

可见，Remark阶段主要是通过扫描堆来判断对象是否存活。那么准确判断对象是否存活，需要扫描哪些对象？CMS对老年代做回收，Remark阶段仅扫描老年代是否可行？结论是不可行，原因如下：

如果仅扫描老年代中对象，即以老年代中对象为根，判断对象是否存在引用，上图中，对象A因为引用存在新生代中，它在Remark阶段就不会被修正标记为可达，GC时会被错误回收。

新生代对象持有老年代中对象的引用，这种情况称为“跨代引用”。因它的存在，Remark阶段必须扫描整个堆来判断对象是否存活，包括图中灰色的不可达对象。

灰色对象已经不可达，但仍然需要扫描的原因：新生代GC和老年代的GC是各自分开独立进行的，只有Minor GC时才会使用根搜索算法，标记新生代对象是否可达，也就是说虽然一些对象已经不可达，但在Minor GC发生前

不会被标记为不可达，CMS也无法辨认哪些对象存活，只能全堆扫描（新生代+老年代）。由此可见堆中对象的数目影响了Remark阶段耗时。

分析GC日志可以得出同样的规律，Remark耗时>500ms时，新生代使用率都在75%以上。这样降低Remark阶段耗时问题转换成如何减少新生代对象数量。

新生代中对象的特点是“朝生夕灭”，这样如果Remark前执行一次Minor GC，大部分对象就会被回收。CMS就采用了这样的方式，在Remark前增加了一个可中断的并发预清理（CMS-concurrent-abortable-preclean），该阶段主要工作仍然是并发标记对象是否存活，只是这个过程可被中断。此阶段在Eden区使用超过2Mb时启动，直到Eden区空间使用率达到50%时中断，当然2Mb和50%都是默认的阈值，可以通过参数修改。如果此阶段执行时等到了Minor GC，那么上述灰色对象将被回收，Reamark阶段需要扫描的对象就少了。

除此之外CMS为了避免这个阶段没有等到Minor GC而陷入无限等待，提供了参数CMSMaxAbortablePrecleanTime，默认为5s，含义是如果可中断的预清理执行超过5s，不管发没发生Minor GC，都会中止此阶段，进入Remark。

根据GC日志红色标记2处显示，可中断的并发预清理执行了5.35s，超过了设置的5s被中断，期间没有等到Minor GC，所以Remark时新生代中仍然有很多对象。

对于这种情况，CMS提供CMSScavengeBeforeRemark参数，用来保证Remark前强制进行一次Minor GC。

优化结果

经过增加CMSScavengeBeforeRemark参数，单次执行时间>200ms的GC停顿消失，从监控上观察，GCtime和业务波动保持一致，不再有明显的毛刺。

小结

通过案例分析了解到，由于跨代引用的存在，CMS在Remark阶段必须扫描整个堆，同时为了避免扫描时新生代有很多对象，增加了可中断的预清理阶段用来等待Minor GC的发生。只是该阶段有时间限制，如果超时等不到

Minor GC，Remark时新生代仍然有很多对象，我们的调优策略是，通过参数强制Remark前进行一次Minor GC，从而降低Remark阶段的时间。

更多思考

案例中只涉及老年代GC，其实新生代GC存在同样的问题，即老年代可能持有新生代对象引用，所以Minor GC时也必须扫描老年代。

JVM是如何避免Minor GC时扫描全堆的?

经过统计信息显示，老年代持有新生代对象引用的情况不足1%，根据这一特性JVM引入了卡表（card table）来实现这一目的。如下图所示：

卡表的具体策略是将老年代的空间分成大小为512B的若干张卡（card）。卡表本身是单字节数组，数组中的每个元素对应着一张卡，当发生老年代引用新生代时，虚拟机将该卡对应的卡表元素设置为适当的值。如上图所示，卡表3被标记为脏（卡表还有另外的作用，标识并发标记阶段哪些块被修改过），之后Minor GC时通过扫描卡表就可以很快的识别哪些卡中存在老年代指向新生代的引用。这样虚拟机通过空间换时间的方式，避免了全堆扫描。

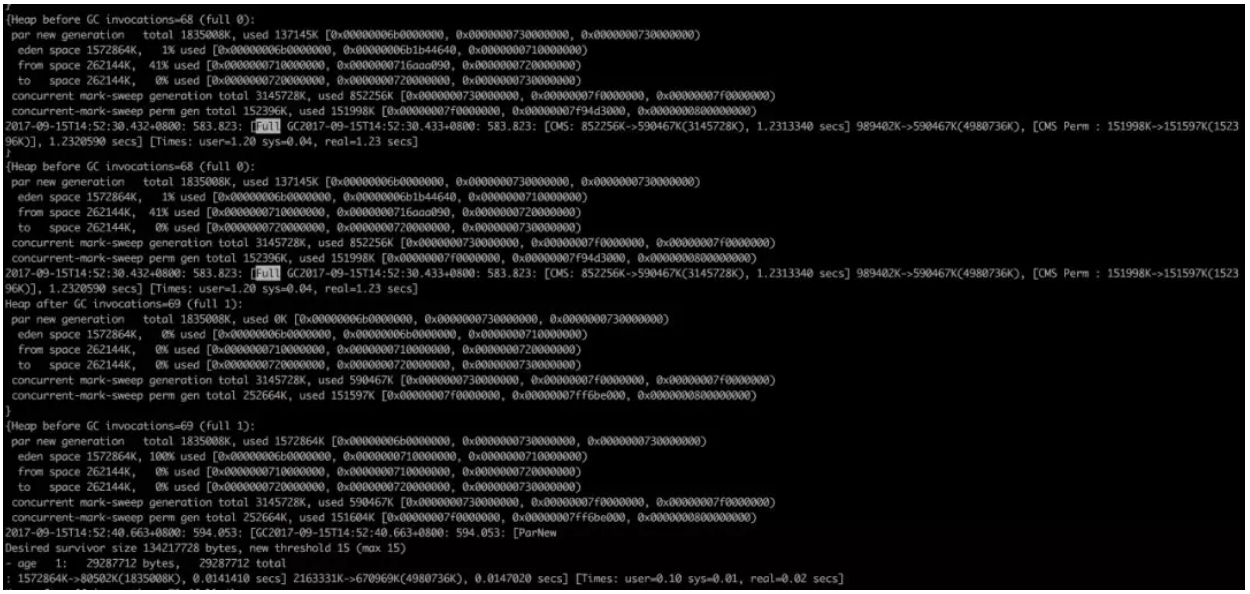
总结来说，CMS的设计聚焦在获取最短的时延，为此它“不遗余力”地做了很多工作，包括尽量让应用程序和GC线程并发、增加可中断的并发预清

理阶段、引入卡表等，虽然这些操作牺牲了一定吞吐量但获得了更短的回收停顿时间。

主案例三：发生Stop-The-World的GC

确定目标

GC日志如下图（在GC日志中，Full GC是用来说明这次垃圾回收的停顿类型，代表STW类型的GC，并不特指老年代GC），根据GC日志可知本次Full GC耗时1.23s。这个在线服务同样要求低时延高可用。本次优化目标是降低单次STW回收停顿时间，提高可用性。



优化

首先，什么时候可能会触发STW的Full GC呢?

1. Perm空间不足；
2. CMS GC时出现promotion failed和concurrent mode failure（concurrent mode failure发生的原因一般是CMS正在进行，但是由于老年代空间不足，需要尽快回收老年代里面的不再被使

用的对象，这时停止所有的线程，同时终止CMS，直接进行Serial Old GC）；

3. 统计得到的Young GC晋升到老年代的平均大小大于老年代的剩余空间；

4. 主动触发Full GC（执行jmap -histo:live [pid]）来避免碎片问题。

然后，我们来逐一分析一下：

- 排除原因2：如果是原因2中两种情况，日志中会有特殊标识，目前没有。
- 排除原因3：根据GC日志，当时老年代使用量仅为20%，也不存在大于2G的大对象产生。
- 排除原因4：因为当时没有相关命令执行。
- 锁定原因1：根据日志发现Full GC后，Perm区变大了，推断是由于永久带空间不足容量扩展导致的。

找到原因后解决方法有两种：

1. 通过把-XX:PermSize参数和-XX:MaxPermSize设置成一样，强制虚拟机在启动的时候就把永久带的容量固定下来，避免运行时自动扩容。

2. CMS默认情况下不会回收Perm区，通过参数CMSPermGenSweepingEnabled、CMSClassUnloadingEnabled，可以让CMS在Perm区容量不足时对其回收。

由于该服务没有生成大量动态类，回收Perm区收益不大，所以我们采用方案1，启动时将Perm区大小固定，避免进行动态扩容。

优化结果

调整参数后，服务不再有Perm区扩容导致的STW GC发生。

小结

对于性能要求很高的服务，建议将MaxPermSize和MinPermSize设置成一致（JDK8开始，Perm区完全消失，转而使用元空间。而元空间是直接存在内存中，不在JVM中），Xms和Xmx也设置为相同，这样可以减少内存自动扩容和收缩带来的性能损失。虚拟机启动的时候就会把参数中所设定的内存全部化为私有，即使扩容前有一部分内存不会被用户代码用到，这部分内存在虚拟机中被标识为虚拟内存，也不会交给其他进程使用。

总结

结合上述GC优化案例做个总结：

1. 首先再次声明，在进行GC优化之前，需要确认项目的架构和代码等已经没有优化空间。我们不能指望一个系统架构有缺陷或者代码层次优化没有穷尽的应用，通过GC优化令其性能达到一个质的飞跃。
2. 其次，通过上述分析，可以看出虚拟机内部已有很多优化来保证应用的稳定运行，所以不要为了调优而调优，不当的调优可能适得其反。
3. 最后，GC优化是一个系统而复杂的工作，没有万能的调优策略可以满足所有的性能指标。GC优化必须建立在我们深入理解各种垃圾回收器的基础上，才能有事半功倍的效果。

本文中案例均来北京业务安全中心（也称风控）对接服务的实践经验。同时感谢风控的小伙伴们，是他们专业负责的审阅，才让这篇文章更加完善。对于本文中涉及到的内容，欢迎大家指正和补充。

作者简介

录录，2016年加入美团点评，主要负责北京业务安全中心对接服务的后台研发工作。

招聘

美团点评北京业务安全中心致力于建设公司平台级业务安全基础设施、保障业务安全运行，工作涵盖交易秩序、帐号安全、爬虫防控等风控方向，基于千万级订单、千万级日活跃用户、亿级存量用户进行数据挖掘，实时处理每日百亿级流量，热诚期待各位开发、算法、策略产品经理人才加入。联系邮箱：tangyizhe#meituan.com。

参考文献

1. Scott O. Java Performance:The Definitive Guide. O'Reilly, 2014.
2. 周志明，深入理解Java虚拟机[M]，机械工业出版社，2013.
3. CMS垃圾回收机制.