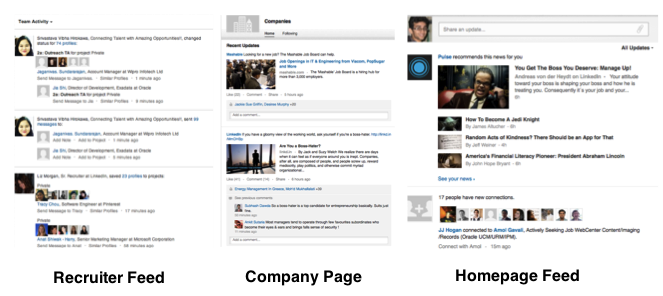
高吞吐低延迟Java应用的垃圾回收优化

高性能应用构成了现代网络的支柱。LinkedIn有许多内部高吞吐量服务来满足每秒数千次的用户请求。要优化用户体验，低延迟地响应这些请求非常重要。

比如说，用户经常用到的一个功能是了解动态信息——不断更新的专业活动和内容的列表。动态信息在LinkedIn随处可见，包括公司页面，学校页面以及最重要的主页。基础动态信息数据平台为我们的经济图谱(会员，公司，群组等等)中各种实体的更新建立索引，它必须高吞吐低延迟地实现相关的更新。

[](http://www.importnew.com/?attachment_id=11339)

*图1 LinkedIn 动态信息*

这些高吞吐低延迟的Java应用转变为产品，开发人员必须确保应用开发周期的每个阶段一致的性能。确定优化垃圾回收(Garbage Collection,GC)的设置对达到这些指标非常关键。

本文章通过一系列步骤来明确需求并优化GC，目标读者是为实现应用的高吞吐低延迟，对使用系统方法优化GC感兴趣的开发人员。文章中的方法来自于LinkedIn构建下一代动态信息数据平台过程。这些方法包括但不局限于以下几点：[并发标记清除](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf" \t "_blank)(Concurrent Mark Sweep,CMS)和[G1](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html" \t "_blank)垃圾回收器的CPU和内存开销，避免长期存活对象引起的持续GC周期，优化GC线程任务分配使性能提升，以及GC停顿时间可预测所需的OS设置。

优化GC的正确时机？

GC运行随着代码级的优化和工作负载而发生变化。因此在一个已实施性能优化的接近完成的代码库上调整GC非常重要。但是在端到端的基本原型上进行初步分析也很有必要，该原型系统使用存根代码并模拟了可代表产品环境的工作负载。这样可以捕捉该架构延迟和吞吐量的真实边界，进而决定是否纵向或横向扩展。

在下一代动态信息数据平台的原型阶段，几乎实现了所有端到端的功能，并且模拟了当前产品基础架构所服务的查询负载。从中我们获得了多种用来衡量应用性能的工作负载特征和足够长时间运行情况下的GC特征。

优化GC的步骤

下面是为满足高吞吐，低延迟需求优化GC的总体步骤。也包括在动态信息数据平台原型实施的具体细节。可以看到在ParNew/CMS有最好的性能，但我们也实验了G1垃圾回收器。

1.理解GC基础知识

理解GC工作机制非常重要，因为需要调整大量的参数。Oracle的Hotspot JVM 内存管理[白皮书](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf" \t "_blank)是开始学习Hotspot JVM GC算法非常好的资料。了解G1垃圾回收器，请查看该[论文](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html" \t "_blank)。

2. 仔细考量GC需求

为降低应用性能的GC开销，可以优化GC的一些特征。吞吐量、延迟等这些GC特征应该长时间测试运行观察，确保特征数据来自于应用程序的处理对象数量发生变化的多个GC周期。

* Stop-the-world回收器回收垃圾时会暂停应用线程。停顿的时长和频率不应该对应用遵守SLA产生不利的影响。
* 并发GC算法与应用线程竞争CPU周期。这个开销不应该影响应用吞吐量。
* 不压缩GC算法会引起堆碎片化，导致full GC长时间Stop-the-world停顿。
* 垃圾回收工作需要占用内存。一些GC算法产生更高的内存占用。如果应用程序需要较大的堆空间，要确保GC的内存开销不能太大。
* 清晰地了解GC日志和常用的JVM参数对简单调整GC运行很有必要。GC运行随着代码复杂度增长或者工作特性变化而改变。

我们使用Linux OS的Hotspot [Java7u51](http://www.oracle.com/technetwork/java/javase/7u51-relnotes-2085002.html" \t "_blank)，32GB堆内存，6GB新生代(young generation)和-XX:CMSInitiatingOccupancyFraction值为70(老年代GC触发时其空间占用率)开始实验。设置较大的堆内存用来维持长期存活对象的对象缓存。一旦这个缓存被填充，提升到老年代的对象比例显著下降。

使用初始的GC配置，每三秒发生一次80ms的新生代GC停顿，超过百分之99.9的应用延迟100ms。这样的GC很可能适合于SLA不太严格要求延迟的许多应用。然而，我们的目标是尽可能降低百分之99.9应用的延迟，为此GC优化是必不可少的。

3.理解GC指标

优化之前要先衡量。了解GC日志的详细细节(使用这些选项：-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime)可以对该应用的GC特征有总体的把握。

LinkedIn的内部监控和报表系统，[inGraphs](https://engineering.linkedin.com/32/eric-intern-origin-ingraphs" \t "_blank)和[Naarad](https://github.com/linkedin/naarad" \t "_blank)，生成了各种有用的指标可视化图形，比如GC停顿时间百分比，一次停顿最大持续时间，长时间内GC频率。除了Naarad，有很多开源工具比如[gclogviewer](https://code.google.com/p/gclogviewer/" \t "_blank)可以从GC日志创建可视化图形。

在这个阶段，需要确定GC频率和停顿时长是否影响应用满足延迟性需求的能力。

4.降低GC频率

在分代GC算法中，降低回收频率可以通过：(1)降低对象分配/提升率；(2)增加代空间的大小。

在Hotspot JVM中，新生代GC停顿时间取决于一次垃圾回收后对象的数量，而不是新生代自身的大小。增加新生代大小对于应用性能的影响需要仔细评估：

* 如果更多的数据存活而且被复制到survivor区域，或者每次垃圾回收更多的数据提升到老年代，增加新生代大小可能导致更长的新生代GC停顿。
* 另一方面，如果每次垃圾回收后存活对象数量不会大幅增加，停顿时间可能不会延长。在这种情况下，减少GC频率可能使应用总体延迟降低和(或)吞吐量增加。

对于大部分为短期存活对象的应用，仅仅需要控制前面所说的参数。对于创建长期存活对象的应用，就需要注意，被提升的对象可能很长时间都不能被老年代GC周期回收。如果老年代GC触发阈值(老年代空间占用率百分比)比较低，应用将陷入不断的GC周期。设置高的GC触发阈值可避免这一问题。

由于我们的应用在堆中维持了长期存活对象的较大缓存，将老年代GC触发阈值设置为-XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSInitiatingOccupancyOnly。我们也试图增加新生代大小来减少新生代回收频率，但是并没有采用，因为这增加了应用延迟。

5.缩短GC停顿时间

减少新生代大小可以缩短新生代GC停顿时间，因为这样被复制到survivor区域或者被提升的数据更少。但是，正如前面提到的，我们要观察减少新生代大小和由此导致的GC频率增加对于整体应用吞吐量和延迟的影响。新生代GC停顿时间也依赖于tenuring threshold(提升阈值)和空间大小(见第6步)。

使用CMS尝试最小化堆碎片和与之关联的老年代垃圾回收full GC停顿时间。通过控制对象提升比例和减小-XX:CMSInitiatingOccupancyFraction的值使老年代GC在低阈值时触发。所有选项的细节调整和他们相关的权衡，请查看[Web Services的Java 垃圾回收](http://engineering.linkedin.com/26/tuning-java-garbage-collection-web-services" \t "_blank)和[Java 垃圾回收精粹](http://mechanical-sympathy.blogspot.com/2013/07/java-garbage-collection-distilled.html" \t "_blank)。

我们观察到Eden区域的大部分新生代被回收，几乎没有对象在survivor区域死亡，所以我们将tenuring threshold从8降低到2(使用选项：-XX:MaxTenuringThreshold=2),为的是缩短新生代垃圾回收消耗在数据复制上的时间。

我们也注意到新生代回收停顿时间随着老年代空间占用率上升而延长。这意味着来自老年代的压力使得对象提升花费更多的时间。为解决这个问题，将总的堆内存大小增加到40GB，减小-XX:CMSInitiatingOccupancyFraction的值到80，更快地开始老年代回收。尽管-XX:CMSInitiatingOccupancyFraction的值减小了，增大堆内存可以避免不断的老年代GC。在本阶段，我们获得了70ms新生代回收停顿和百分之99.9延迟80ms。

6.优化GC工作线程的任务分配

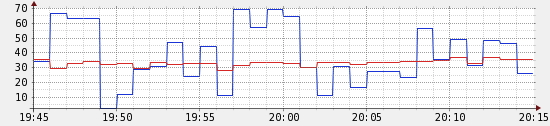
进一步缩短新生代停顿时间，我们决定研究优化与GC线程绑定任务的选项。

-XX:ParGCCardsPerStrideChunk 选项控制GC工作线程的任务粒度，可以帮助不使用[补丁](http://www.importnew.com/%5bhttp:/bugs.java.com/bugdatabase/view_bug.do?bug_id=7068625%5d(http://bugs.java.com/bugdatabase/view_bug.do?bug_id=7068625))而获得最佳性能，这个补丁用来优化新生代垃圾回收的[卡表扫描时间](http://blog.ragozin.info/2012/03/secret-hotspot-option-improving-gc.html" \t "_blank)。有趣的是新生代GC时间随着老年代空间的增加而延长。将这个选项值设为32678，新生代回收停顿时间降低到平均50ms。此时百分之99.9应用延迟60ms。

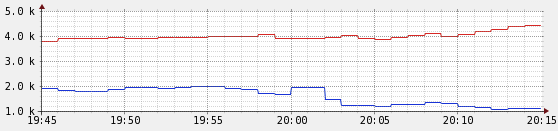
也有其他选项将任务映射到GC线程，如果OS允许的话，-XX:+BindGCTaskThreadsToCPUs选项绑定GC线程到个别的CPU核。-XX:+UseGCTaskAffinity使用affinity参数将任务分配给GC工作线程。然而，我们的应用并没有从这些选项发现任何益处。实际上，一些调查显示这些选项在Linux系统不起作用[1,2]。

7.了解GC的CPU和内存开销

并发GC通常会增加CPU的使用。我们观察了运行良好的CMS默认设置，并发GC和G1垃圾回收器共同工作引起的CPU使用增加显著降低了应用的吞吐量和延迟。与CMS相比，G1可能占用了应用更多的内存开销。对于低吞吐量的非计算密集型应用，GC的高CPU使用率可能不需要担心。

[](http://www.importnew.com/?attachment_id=11340)

*图2 ParNew/CMS和G1的CPU使用百分数%：相对来说CPU使用率变化明显的节点使用G1  
选项-XX:G1RSetUpdatingPauseTimePercent=20*

[](http://www.importnew.com/?attachment_id=11338)

*图3 ParNew/CMS和G1每秒服务的请求数：吞吐量较低的节点使用G1  
选项-XX:G1RSetUpdatingPauseTimePercent=20*

8.为GC优化系统内存和I/O管理

通常来说，GC停顿发生在(1)低用户时间，高系统时间和高时钟时间和(2)低用户时间，低系统时间和高时钟时间。这意味着基础的进程/OS设置存在问题。情况(1)可能说明Linux从JVM偷页，情况(2)可能说明清除磁盘缓存时Linux启动GC线程，等待I/O时线程陷入内核。在这些情况下如何设置参数可以参考[该PPT](http://www.slideshare.net/cuonghuutran/gc-andpagescanattacksbylinux" \t "_blank)。

为避免运行时性能损失，启动应用时使用JVM选项-XX:+AlwaysPreTouch访问和清零页面。设置vm.swappiness为零，除非在绝对必要时，OS不会交换页面。

可能你会使用mlock将JVM页pin在内存中，使OS不换出页面。但是，如果系统用尽了所有的内存和交换空间，OS通过kill进程来回收内存。通常情况下，Linux内核会选择高驻留内存占用但还没有长时间运行的进程([OOM情况下killing进程的工作流](https://www.kernel.org/doc/gorman/html/understand/understand016.html" \t "_blank))。对我们而言，这个进程很有可能就是我们的应用程序。一个服务具备优雅降级(适度退化)的特点会更好，服务突然故障预示着不太好的可操作性——因此，我们没有使用mlock而是vm.swappiness避免可能的交换惩罚。

LinkedIn动态信息数据平台的GC优化

对于该平台原型系统，我们使用Hotspot JVM的两个算法优化垃圾回收：

* 新生代垃圾回收使用ParNew，老年代垃圾回收使用CMS。
* 新生代和老年代使用G1。G1用来解决堆大小为6GB或者更大时存在的低于0.5秒稳定的、可预测停顿时间的问题。在我们用G1实验过程中，尽管调整了各种参数，但没有得到像ParNew/CMS一样的GC性能或停顿时间的可预测值。我们查询了使用G1发生内存泄漏相关的一个bug[3]，但还不能确定根本原因。

使用ParNew/CMS，应用每三秒40-60ms的新生代停顿和每小时一个CMS周期。JVM选项如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | // JVM sizing options  -server -Xms40g -Xmx40g -XX:MaxDirectMemorySize=4096m -XX:PermSize=256m -XX:MaxPermSize=256m  // Young generation options  -XX:NewSize=6g -XX:MaxNewSize=6g -XX:+UseParNewGC -XX:MaxTenuringThreshold=2 -XX:SurvivorRatio=8 -XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=32768  // Old generation  options  -XX:+UseConcMarkSweepGC -XX:CMSParallelRemarkEnabled -XX:+ParallelRefProcEnabled -XX:+CMSClassUnloadingEnabled  -XX:CMSInitiatingOccupancyFraction=80 -XX:+UseCMSInitiatingOccupancyOnly  // Other options  -XX:+AlwaysPreTouch -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -XX:-OmitStackTraceInFastThrow |

使用这些选项，对于几千次读请求的吞吐量，应用百分之99.9的延迟降低到60ms。

参考：

[1] -XX:+BindGCTaskThreadsToCPUs似乎在Linux系统上不起作用，因为hotspot/src/os/linux/vm/os\_linux.cpp的distribute\_processes方法在JDK7或JDK8没有实现。  
[2] -XX:+UseGCTaskAffinity选项在JDK7和JDK8的所有平台似乎都不起作用，因为任务的affinity属性永远被设置为sentinel\_worker = (uint) -1。源码见hotspot/src/share/vm/gc\_implementation/parallelScavenge/{gcTaskManager.cpp，gcTaskThread.cpp, gcTaskManager.cpp}。  
[3] G1存在一些内存泄露的bug，可能Java7u51没有修改。这个[bug](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=2212435" \t "_blank)仅在Java 8修正了。