# 2. 一次 Java 内存泄漏排查过程，涨姿势

在一些物理内存为8g的服务器上，主要运行一个Java服务，系统内存分配如下：Java服务的JVM堆大小设置为6g，一个监控进程占用大约 600m，Linux自身使用大约800m。从表面上，物理内存应该是足够使用的；但实际运行的情况是，会发生大量使用SWAP(说明物理内存不够使用 了)，如下图所示。由于SWAP和GC同时发生会致使JVM严重卡顿，所以我们要追问：内存究竟去哪儿了？

要分析这个问题，理解JVM和操作系统之间的内存关系非常重要。接下来主要就Linux与JVM之间的内存关系进行一些分析。

一、Linux与进程内存模型

JVM以一个进程（Process）的身份运行在Linux系统上，了解Linux与进程的内存关系，是理解JVM与Linux内存的关系的基础。下图给出了硬件、系统、进程三个层面的内存之间的概要关系。

从硬件上看，Linux系统的内存空间由两个部分构成：物理内存和SWAP（位于磁盘）。物理内存是Linux活动时使用的主要内存区域；当物理内存不够使用时，Linux会把一部分暂时不用的内存数据放到磁盘上的SWAP中去，以便腾出更多的可用内存空间；而当需要使用位于SWAP的数据时，必须 先将其换回到内存中。

从Linux系统上看，除了引导系统的BIN区，整个内存空间主要被分成两个部分：内核内存（Kernel space）、用户内存（User space）。

内核内存是Linux自身使用的内存空间，主要提供给程序调度、内存分配、连接硬件资源等程序逻辑使用。用户内存是提供给各个进程主要空间，Linux给各个进程提供相同的虚拟内存空间；这使得进程之间相互独立，互不干扰。实现的方法是采用虚拟内存技术：给每一个进程一定虚拟内存空间，而只有当虚拟内存实 际被使用时，才分配物理内存。如下图所示，对于32的Linux系统来说，一般将0～3G的虚拟内存空间分配做为用户空间，将3～4G的虚拟内存空间分配 为内核空间；64位系统的划分情况是类似的。

从进程的角度来看，进程能直接访问的用户内存（虚拟内存空间）被划分为5个部分：代码区、数据区、堆区、栈区、未使用区。代码区中存放应用程序的机器代码，运行过程中代码不能被修改，具有只读和固定大小的特点。数据区中存放了应用程序中的全局数据，静态数据和一些常量字符串等，其大小也是固定的。堆是运行时程序动态申请的空间，属于程序运行时直接申请、释放的内存资源。栈区用来存放函数的传入参数、临时变量，以及返回地址等数据。未使用区是分配新内 存空间的预备区域。

二、进程与JVM内存空间

JVM本质就是一个进程，因此其内存空间（也称之为运行时数据区，注意与JMM的区别）也有进程的一般特点。但是，JVM又不是一个普通的进程，其在内存空间上有许多崭新的特点，主要原因有两 个：1.JVM将许多本来属于操作系统管理范畴的东西，移植到了JVM内部，目的在于减少系统调用的次数；2. Java NIO，目的在于减少用于读写IO的系统调用的开销。 JVM进程与普通进程内存模型比较如下图:

需要说明的是，这个模型的并不是JVM内存使用的精确模型，更侧重于从操作系统的角度而省略了一些JVM的内部细节（尽管也很重要）。下面从用户内存和内核内存两个方面讲解JVM进程的内存特点。

**1.用户内存**

上图特别强调了JVM进程模型的代码区和数据区指的是JVM自身的，而非Java程序的。普通进程栈区，在JVM一般仅仅用做线程栈。JVM的堆区和普通进程的差别是最大的，下面具体详细说明：

首先是永久代。永久代本质上是Java程序的代码区和数据区。Java程序中类（class），会被加载到整个区域的不同数据结构中去，包括常量池、域、方法数据、方法体、构造函数、以及类中的专用方法、实例初始化、接口初始化等。这个区域对于操作系统来说，是堆的一个部分；而对于Java程序来 说，这是容纳程序本身及静态资源的空间，使得JVM能够解释执行Java程序。

其次是新生代和老年代。新生代和老年代才是Java程序真正使用的堆空间，主要用于内存对象的存储；但是其管理方式和普通进程有本质的区别。

普通进程在运行时给内存对象分配空间时，比如C++执行new操作时，会触发一次分配内存空间的系统调用，由操作系统的线程根据对象的大小分配好空间后返 回；同时，程序释放对象时，比如C++执行delete操作时，也会触发一次系统调用，通知操作系统对象所占用的空间已经可以回收。

JVM对内存的使用和一般进程不同。JVM向操作系统申请一整段内存区域（具体大小可以在JVM参数调节）作为Java程序的堆（分为新生代和老年代）； 当Java程序申请内存空间，比如执行new操作，JVM将在这段空间中按所需大小分配给Java程序，并且Java程序不负责通知JVM何时可以释放这 个对象的空间，垃圾对象内存空间的回收由JVM进行。

JVM的内存管理方式的优点是显而易见的，包括：第一，减少系统调用的次数，JVM在给Java程序分配内存空间时不需要操作系统干预，仅仅在 Java堆大小变化时需要向操作系统申请内存或通知回收，而普通程序每次内存空间的分配回收都需要系统调用参与；第二，减少内存泄漏，普通程序没有（或者 没有及时）通知操作系统内存空间的释放是内存泄漏的重要原因之一，而由JVM统一管理，可以避免程序员带来的内存泄漏问题。

最后是未使用区，未使用区是分配新内存空间的预备区域。对于普通进程来说，这个区域被可用于堆和栈空间的申请及释放，每次堆内存分配都会使用这个区 域，因此大小变动频繁；对于JVM进程来说，调整堆大小及线程栈时会使用该区域，而堆大小一般较少调整，因此大小相对稳定。操作系统会动态调整这个区域的 大小，并且这个区域通常并没有被分配实际的物理内存，只是允许进程在这个区域申请堆或栈空间。

**2.内核内存**

应用程序通常不直接和内核内存打交道，内核内存由操作系统进行管理和使用；不过随着Linux对性能的关注及改进，一些新的特性使得应用程序可以使 用内核内存，或者是映射到内核空间。Java NIO正是在这种背景下诞生的，其充分利用了Linux系统的新特性，提升了Java程序的IO性能。

上图给出了Java NIO使用的内核内存在linux系统中的分布情况。nio buffer主要包括：nio使用各种channel时所使用的ByteBuffer、Java程序主动使用 ByteBuffer.allocateDirector申请分配的Buffer。而在PageCache里面，nio使用的内存主要包 括：FileChannel.map方式打开文件占用mapped、FileChannel.transferTo和 FileChannel.transferFrom所需要的Cache（图中标示 nio file）。

通过JMX可以监控到NIO Buffer和 mapped 的使用情况，如下图所示。不过，FileChannel的实现是通过系统调用使用原生的PageCache，过程对于Java是透明的，无法监控到这部分内存的使用大小。

Linux和Java NIO在内核内存上开辟空间给程序使用，主要是减少不要的复制，以减少IO操作系统调用的开销。例如，将磁盘文件的数据发送网卡，使用普通方法和NIO时，数据流动比较下图所示：

将数据在内核内存和用户内存之间拷贝是比较消耗资源和时间的事情，而从上图我们可以看到，通过NIO的方式减少了2次内核内存和用户内存之间的数据拷贝。这是Java NIO高性能的重要机制之一（另一个是异步非阻塞）。

从上面可以看出，内核内存对于Java程序性能也非常重要，因此，在划分系统内存使用时候，一定要给内核留出一定可用空间。

三、案例分析

**1.内存分配问题**

通过上面的分析，省略比较小的区域，可以总结JVM占用的内存：  
JVM内存 ≈ Java永久代 ＋ Java堆(新生代和老年代) ＋ 线程栈＋ Java NIO

回到文章开头提出的问题，原来的内存分配是：6g(java堆) ＋ 600m(监控) ＋ 800m(系统)，剩余大约600m内存未分配。

现在分析这600m内存的分配情况：

1. Linux保留大约200m，这部分是Linux正常运行的需要，
2. Java服务的线程数量是160个，JVM默认的线程栈大小是1m，因此使用160m内存，
3. Java NIO buffer，通过JMX查到最多占用了200m，
4. Java服务使用NIO大量读写文件，需要使用PageCache，正如前面分析，这个暂时不好定量估算大小。

前三项加起来已经560m，因此可以断定Linux物理内存不够使用。

细心的人会发现，引言中给出两个服务器，一个SWAP最多占用了2.16g，另外一个SWAP最多占用了871m；但是，似乎我们的内存缺口没有那么大。事实上，这是由于SWAP和GC同时进行造成的，从下图可以看到，SWAP的使用和长时间的GC在同一时刻发生。

SWAP和GC同时发生会导致GC时间很长，JVM严重卡顿，极端的情况下会导致服务崩溃。原因如下：JVM进行GC时，时需要对相应堆分区的已用 内存进行遍历；假如GC的时候，有堆的一部分内容被交换到SWAP中，遍历到这部分的时候就需要将其交换回内存，同时由于内存空间不足，就需要把内存中堆 的另外一部分换到SWAP中去；于是在遍历堆分区的过程中，(极端情况下)会把整个堆分区轮流往SWAP写一遍。Linux对SWAP的回收是滞后的，我 们就会看到大量SWAP占用。上述问题，可以通过减少堆大小，或者增加物理内存解决。

因此，我们得出一个结论：部署Java服务的Linux系统，在内存分配上，需要避免SWAP的使用；具体如何分配需要综合考虑不同场景下JVM对Java永久代 、Java堆(新生代和老年代)、线程栈、Java NIO所使用内存的需求。

**2.内存泄漏问题**

另一个案例是，8g内存的服务器，Linux使用800m，监控进程使用600m，堆大小设置4g；系统可用内存有2.5g左右，但是也发生了大量的SWAP占用。

分析这个问题如下：

1. 在这个场景中， Java永久代 、Java堆(新生代和老年代)、线程栈所用内存基本是固定的，因此，占用内存过多的原因就定位在Java NIO上。
2. 根据前面的模型，Java NIO使用的内存主要分布在Linux内核内存的System区和PageCache区。查看监控的记录，如下图，我们可以看到发生SWAP之前，也就是 物理内存不够使用的时候，PageCache急剧缩小。因此，可以定位在System区的Java NIO Buffer发生内存泄漏。
3. 由于NIO的DirectByteBuffer需要在GC的后期被回收，因此连续申请DirectByteBuffer的程序，通常需要调用 System.gc()，避免长时间不发生FullGC导致引用在old区的DirectByteBuffer内存泄漏。分析到此，可以推断有两种可能的 原因：第一，Java程序没有在必要的时候调用System.gc()；第二，System.gc()被禁用。
4. 最后是要排查JVM启动参数和Java程序的DirectByteBuffer使用情况。在本例中，查看JVM启动参数，发现启用了-XX:+DisableExplicitGC导致System.gc()被禁用。

四、总结

本文详细分析了Linux与JVM的内存关系，比较了一般进程与JVM进程使用内存的异同点，理解这些特性将对Linux系统内存分配、JVM调优、Java程序优化有帮助。限于篇幅关系仅仅列举两个案例，希望起到抛砖引玉的作用。

*http://www.open-open.com/lib/view/open1420814127390.html*

# 2. 一次 Java 内存泄漏排查过程，涨姿势

人人都会犯错，但一些错误是如此的荒谬，我想不通怎么会有人犯这种错误。更没想到的是，这种事竟发生在了我们身上。当然，这种东西只有事后才能发现真相。接下来，我将讲述一系列最近在我们一个应用上犯过的这种错误。最有意思的是，一开始的迹象揭示的问题，与实际发生的问题完全不同。

## 在一个凄凉的午夜

午夜刚过，我就被一条来自监控系统的警报吵醒了。Adventory，我们的 PPC （以点击次数收费）广告系统中一个负责索引广告的应用，很明显连续重启了好几次。在云端的环境里，实例的重启是很正常的，也不会触发报警，但这次实例重启的次数在短时间内超过了阈值。我打开了笔记本电脑，一头扎进项目的日志里。

## 一定是网络的问题

我看到服务在连接 ZooKeeper 时发生了数次超时。我们使用 ZooKeeper（ZK）协调多个实例间的索引操作，并依赖它实现鲁棒性。很显然，一次 Zookeeper 失败会阻止索引操作的继续运行，不过它应该不会导致整个系统挂掉。而且，这种情况非常罕见（这是我第一次遇到 ZK 在生产环境挂掉），我觉得这个问题可能不太容易搞定。于是我把 ZooKeeper 的值班人员喊醒了，让他们看看发生了什么。

同时，我检查了我们的配置，发现 ZooKeeper 连接的超时时间是秒级的。很明显，ZooKeeper 全挂了，由于其他服务也在使用它，这意味着问题非常严重。我给其他几个团队发了消息，他们显然还不知道这事儿。

ZooKeeper 团队的同事回复我了，在他看来，系统运行一切正常。由于其他用户看起来没有受到影响，我慢慢意识到不是 ZooKeeper 的问题。日志里明显是网络超时，于是我把负责网络的同事叫醒了。

负责网络的团队检查了他们的监控，没有发现任何异常。由于单个网段，甚至单个节点，都有可能和剩余的其他节点断开连接，他们检查了我们系统实例所在的几台机器，没有发现异常。其间，我尝试了其他几种思路，不过都行不通，我也到了自己智力的极限。时间已经很晚了（或者说很早了），同时，跟我的尝试没有任何关系，重启变得不那么频繁了。由于这个服务仅仅负责数据的刷新，并不会影响到数据的可用性，我们决定把问题放到上午再说。

## 一定是 GC 的问题

有时候把难题放一放，睡一觉，等脑子清醒了再去解决是一个好主意。没人知道当时发生了什么，服务表现的非常怪异。突然间，我想到了什么。Java 服务表现怪异的主要根源是什么？当然是垃圾回收。

为了应对目前这种情况的发生，我们一直打印着 GC 的日志。我马上把 GC 日志下载了下来，然后打开 Censum开始分析日志。我还没仔细看，就发现了一个恐怖的情况：每15分钟发生一次 full GC，每次 GC 引发长达 20 秒的服务停顿。怪不得连接 ZooKeeper 超时了，即使 ZooKeeper 和网络都没有问题。

这些停顿也解释了为什么整个服务一直是死掉的，而不是超时之后只打一条错误日志。我们的服务运行在 Marathon 上，它定时检查每个实例的健康状态，如果某个端点在一段时间内没有响应，Marathon 就重启那个服务。

知道原因之后，问题就解决一半了，因此我相信这个问题很快就能解决。为了解释后面的推理，我需要说明一下 Adventory 是如何工作的，它不像你们那种标准的微服务。

Adventory 是用来把我们的广告索引到 ElasticSearch (ES) 的。这需要两个步骤。第一步是获取所需的数据。到目前为止，这个服务从其他几个系统中接收通过 Hermes 发来的事件。数据保存到 MongoDB 集群中。数据量最多每秒几百个请求，每个操作都特别轻量，因此即便触发一些内存的回收，也耗费不了多少资源。第二步就是数据的索引。这个操作定时执行（大概两分钟执行一次），把所有 MongoDB 集群存储的数据通过 RxJava 收集到一个流中，组合为非范式的记录，发送给 ElasticSearch。这部分操作类似离线的批处理任务，而不是一个服务。

由于经常需要对数据做大量的更新，维护索引就不太值得，所以每执行一次定时任务，整个索引都会重建一次。这意味着一整块数据都要经过这个系统，从而引发大量的内存回收。尽管使用了流的方式，我们也被迫把堆加到了 12 GB 这么大。由于堆是如此巨大（而且目前被全力支持），我们的 GC 选择了 G1。

我以前处理过的服务中，也会回收大量生命周期很短的对象。有了那些经验，我同时增加了 -XX:G1NewSizePercent 和 -XX:G1MaxNewSizePercent 的默认值，这样新生代会变得更大，young GC 就可以处理更多的数据，而不用把它们送到老年代。Censum 也显示有很多过早提升。这和一段时间之后发生的 full GC 也是一致的。不幸的是，这些设置没有起到任何作用。

接下来我想，或许生产者制造数据太快了，消费者来不及消费，导致这些记录在它们被处理前就被回收了。我尝试减小生产数据的线程数量，降低数据产生的速度，同时保持消费者发送给 ES 的数据池大小不变。这主要是使用背压（backpressure）机制，不过它也没有起到作用。

## 一定是内存泄漏

这时，一个当时头脑还保持冷静的同事，建议我们应该做一开始就做的事情：检查堆中的数据。我们准备了一个开发环境的实例，拥有和线上实例相同的数据量，堆的大小也大致相同。把它连接到 jnisualvm ，分析内存的样本，我们可以看到堆中对象的大致数量和大小。大眼一看，可以发现我们域中Ad对象的数量高的不正常，并且在索引的过程中一直在增长，一直增长到我们处理的广告的数量级别。但是……这不应该啊。毕竟，我们通过 RX 把这些数据整理成流，就是为了防止把所有的数据都加载到内存里。

随着怀疑越来越强，我检查了这部分代码。它们是两年前写的，之后就没有再被仔细的检查过。果不其然，我们实际上把所有的数据都加载到了内存里。这当然不是故意的。由于当时对 RxJava 的理解不够全面，我们想让代码以一种特殊的方式并行运行。为了从 RX 的主工作流中剥离出来一些工作，我们决定用一个单独的 executor 跑 CompetableFuture。但是，我们因此就需要等待所有的 CompetableFuture 都工作完……通过存储他们的引用，然后调用 join()。这导致一直到索引完成，所有的 future 的引用，以及它们引用到的数据，都保持着生存的状态。这阻止了垃圾收集器及时的把它们清理掉。

## 真有这么糟糕吗？

当然这是一个很愚蠢的错误，对于发现得这么晚，我们也很恶心。我甚至想起很久之前，关于这个应用需要 12 GB 的堆的问题，曾有个简短的讨论。12 GB 的堆，确实有点大了。但是另一方面，这些代码已经运行了将近两年了，没有发生过任何问题。我们可以在当时相对容易的修复它，然而如果是两年前，这可能需要我们花费更多的时间，而且相对于节省几个 G 的内存，当时我们有很多更重要的工作。

因此，虽然从纯技术的角度来说，这个问题如此长时间没解决确实很丢人，然而从战略性的角度来看，或许留着这个浪费内存的问题不管，是更务实的选择。当然，另一个考虑就是这个问题一旦发生，会造成什么影响。我们几乎没有对用户造成任何影响，不过结果有可能更糟糕。软件工程就是权衡利弊，决定不同任务的优先级也不例外。

## 还是不行

有了更多使用 RX 的经验之后，我们可以很简单的解决 ComplerableFurue 的问题。重写代码，只使用 RX；在重写的过程中，升级到 RX2；真正的流式处理数据，而不是在内存里收集它们。这些改动通过 code review 之后，部署到开发环境进行测试。让我们吃惊的是，应用所需的内存丝毫没有减少。内存抽样显示，相较之前，内存中广告对象的数量有所减少。而且对象的数量现在不会一直增长，有时也会下降，因此他们不是全部在内存里收集的。还是老问题，看起来这些数据仍然没有真正的被归集成流。

## 那现在是怎么回事？

相关的关键词刚才已经提到了：背压。当数据被流式处理，生产者和消费者的速度不同是很正常的。如果生产者比消费者快，并且不能把速度降下来，它就会一直生产越来越多的数据，消费者无法以同样的速度处理掉他们。现象就是未处理数据的缓存不断增长，而这就是我们应用中真正发生的。背压就是一套机制，它允许一个较慢的消费者告诉较快的生产者去降速。

我们的索引系统没有背压的概念，这在之前没什么问题，反正我们把整个索引都保存到内存里了。一旦我们解决了之前的问题，开始真正的流式处理数据，缺少背压的问题就变得很明显了。

这个模式我在解决性能问题时见过很多次了：解决一个问题时会浮现另一个你甚至没有听说过的问题，因为其他问题把它隐藏起来了。如果你的房子经常被淹，你不会注意到它有火灾隐患。

## 修复由修复引起的问题

在 RxJava 2 里，原来的 Observable 类被拆成了不支持背压的 Observable 和支持背压的 Flowable。幸运的是，有一些简单的办法，可以开箱即用的把不支持背压的 Observable 改造成支持背压的 Flowable。其中包含从非响应式的资源比如 Iterable 创建 Flowable。把这些 Flowable 融合起来可以生成同样支持背压的 Flowable，因此只要快速解决一个点，整个系统就有了背压的支持。

有了这个改动之后，我们把堆从 12 GB 减少到了 3 GB ，同时让系统保持和之前同样的速度。我们仍然每隔数小时就会有一次暂停长达 2 秒的 full GC，不过这比我们之前见到的 20 秒的暂停（还有系统崩溃）要好多了。

## 再次优化 GC

但是，故事到此还没有结束。检查 GC 的日志，我们注意到大量的过早提升，占到 70%。尽管性能已经可以接受了，我们也尝试去解决这个问题，希望也许可以同时解决 full GC 的问题。

如果一个对象的生命周期很短，但是它仍然晋升到了老年代，我们就把这种现象叫做过早提升（premature tenuring）（或者叫过早升级）。老年代里的对象通常都比较大，使用与新生代不同的 GC 算法，而这些过早提升的对象占据了老年代的空间，所以它们会影响 GC 的性能。因此，我们想竭力避免过早提升。

我们的应用在索引的过程中会产生大量短生命周期的对象，因此一些过早提升是正常的，但是不应该如此严重。当应用产生大量短生命周期的对象时，能想到的第一件事就是简单的增加新生代的空间。默认情况下，G1 的 GC 可以自动的调整新生代的空间，允许新生代使用堆内存的 5% 至 60%。我注意到运行的应用里，新生代和老年代的比例一直在一个很宽的幅度里变化，不过我依然动手修改了两个参数：-XX:G1NewSizePercent=40 和 -XX:G1MaxNewSizePercent=90看看会发生什么。这没起作用，甚至让事情变得更糟糕了，应用一启动就触发了 full GC。我也尝试了其他的比例，不过最好的情况就是只增加 G1MaxNewSizePercent而不修改最小值。这起了作用，大概和默认值的表现差不多，也没有变好。

尝试了很多办法后，也没有取得什么成就，我就放弃了，然后给 Kirk Pepperdine 发了封邮件。他是位很知名的 Java 性能专家，我碰巧在 Allegro 举办的 Devoxx 会议的训练课程里认识了他。通过查看 GC 的日志以及几封邮件的交流，Kirk 建议试试设置 -XX:G1MixedGCLiveThresholdPercent=100。这个设置应该会强制 G1 GC 在 mixed GC 时不去考虑它们被填充了多少，而是强制清理所有的老年代，因此也同时清理了从新生代过早提升的对象。这应该会阻止老年代被填满从而产生一次 full GC。然而，在运行一段时间以后，我们再次惊讶的发现了一次 full GC。Kirk 推断说他在其他应用里也见到过这种情况，它是 G1 GC 的一个 bug：mixed GC 显然没有清理所有的垃圾，让它们一直堆积直到产生 full GC。他说他已经把这个问题通知了 Oracle，不过他们坚称我们观察到的这个现象不是一个 bug，而是正常的。

## 结论

我们最后做的就是把应用的内存调大了一点点（从 3 GB 到 4 GB），然后 full GC 就消失了。我们仍然观察到大量的过早提升，不过既然性能是没问题的，我们就不在乎这些了。一个我们可以尝试的选项是转换到 GMS（Concurrent Mark Sweep）GC，不过由于它已经被废弃了，我们还是尽量不去使用它。

那么这个故事的寓意是什么呢？首先，性能问题很容易让你误入歧途。一开始看起来是 ZooKeeper 或者 网络的问题，最后发现是我们代码的问题。即使意识到了这一点，我首先采取的措施也没有考虑周全。为了防止 full GC，我在检查到底发生了什么之前就开始调优 GC。这是一个常见的陷阱，因此记住：即使你有一个直觉去做什么，先检查一下到底发生了什么，再检查一遍，防止浪费时间去错误的问题。

第二条，性能问题太难解决了。我们的代码有良好的测试覆盖率，而且运行的特别好，但是它也没有满足性能的要求，它在开始的时候就没有清晰的定义好。性能问题直到部署之后很久才浮现出来。由于通常很难真实的再现你的生产环境，你经常被迫在生产环境测试性能，即使那听起来非常糟糕。

第三条，解决一个问题有可能引发另一个潜在问题的浮现，强迫你不断挖的比你预想的更深。我们没有背压的事实足以中断这个系统，但是直到我们解决了内存泄漏的问题后，它才浮现。

http://www.importnew.com/29591.html

dd