Java 内存管理

与其它高级语言（如 c 和 c++）不太一样，在 Java 中我们基本上不会要显式的调用分配内存的函数，我们甚至不用关心，到底哪些程序指令需要分配内存哪些不需要分配内存。因为在 Java 中，分配内存和回收内存都由 JVM 帮你完成了，你很少会遇到像 c++程序中那样令人头疼的内存泄露问题。

虽然 Java 语言的这些特点很容易会“惯坏”开发人员，使得我们不需要太关心，到底我们的程序是怎么使用内存的，到底使用了多少内存。但是我们最好也应该了解 Java 是如何管理内存的，当你真的遇到 OutOfMemoryError 时，你不会奇怪的问，为什么 Java 也有内存泄露。你要快速的知道到底什么地方导致了 OutOfMemoryError，并能根据错误日志快速的定位出错原因。

本章首先将从操作系统层面简单介绍下物理内存的分配和 Java 运行时的内存分配之间的关系，也就是先明白 Java 中使用的内存与物理内存有何区别。其次介绍 Java 是如何使用从物理内存申请下来的内存，并如何来划分它们。后面还会介绍 Java 的核心技术：如何分配和回收内存。最后介绍一些例子如何解决 OutOfMemoryError，并提供一些处理这类问题的常用手段。

物理内存与虚拟内存

我们先看看物理内存，所谓物理内存就是我们通常所说的 RAM（随机存储器）。在计算机中，还有一个存储单元叫寄存器，它是用于存储计算单元在执行指令如浮点、整数等运输时的中间结果。寄存器的大小决定了一次计算可使用的最大数值。

连接处理器和 RAM 或者处理器与寄存器的是地址总线，这个地址总线的宽度影响了物理地址的索引范围，因为总线的宽度决定了处理器一次可以从寄存器或者内存中获取多少个 bit。同时也决定了处理器最大可以寻址的地址空间，如 32 位地址总线可以寻址的范围为：0x0000 0000~0xffff ffff。这个范围是 2^32=4 294 967 296 个内存位置，每个地址会引用一个字节，所以 32 位总线宽度可以由 4G 的内存空间。

通常情况下，地址总线和寄存器或者 RAM 有相同的位数，因为这样更容易传输数据，但是也有不一致的情况，如 x86 的 32 位寄存器宽度，它的物理地址可能有两种大小，分别是 32 位物理地址和 36 位物理地址，这个拥有 36 位物理地址的是 Pentium Pro 和更高型号。

除了我们在学校的编译原理的实践课或者你要开发硬件程序的驱动程序时需要直接通过程序访问存储器外，我们大部分情况下都是调用操作系统提供的接口来访问内存。在 Java 中，甚至不需要写和内存相关的代码。

不管是在 windows 或者 Linux 系统，我们要运行程序，都会要向操作系统先申请内存地址，通常操作系统管理内存的申请空间是按照进程来管理的，每个进程会拥有一段独立的地址空间，每个进程之间不会相互重合，操作系统也会保证，每个进程只能访问自己的内存空间。这主要是从程序的安全性来考虑的，也便于操作系统来管理物理内存。

其实上面所说的进程的内存空间的独立主要是指逻辑上独立，也就是这个独立是由操作系统来保证的，但是真正的物理空间是不是只能只有一个进程来使用？这个就不一定了，因为随着程序越来越庞大和多任务的设计，物理内存无法满足程序的需求，这种情况下就有了虚拟内存的出现。

虚拟内存出现使得多个进程在同时运行时可以共享物理内存，这里的共享只是空间上共享，在逻辑上，它们仍然是不能相互访问的。虚拟地址不但可以让进程可以共享物理内存、提供内存利用率，而且还能够扩展内存的地址空间。如一个虚拟地址可能被映射到一段物理内存、文件或者其它可以寻址的存储上。当一个进程在不活动的的情况下，操作

系统将这个物理内存中的数据移到一个磁盘文件中（也就是通常 windows 上的页面文件，或者 linux 上交换分区），而真正高效的物理内存，留给正在活动的程序使用。这种情况下，在我们重新唤醒一个很长时间没有使用的程序时，磁盘会吱吱作响，并且会有一个短暂的停顿得到印证，这时操作系统又会把磁盘上的数据重新交互到物理内存中，但是我们必须要避免这种情况经常出现，如果操作系统频繁的在交互物理内存的数据和磁盘数据时，效率将会非常的低，尤其在 linux 的服务器上，我们要关注 linux 中 swap 的分区的活跃度。如果 swap 分区被频繁使用时，系统将会非常缓慢，很可能意味着物理内存已经严重不足或者某些程序没有及时的释放内存。

内核空间与用户空间

一个计算机通常有一定大小的内存空间，如我使用的电脑是 4G 的地址空间，但是程序并不能完全使用这些地址空间，因为这些地址空间被划分为：内核空间和用户空间。我们的程序只能使用用户空间的内存，这里所说的使用是指我们的程序能够申请的内存空间，并不是程序真正访问的地址空间。

内核空间主要是指操作系统运行时所使用的用于程序调度、虚拟内存的使用或者连接硬件资源等程序逻辑。为何需要内存空间和用户空间的划分呢？很显然和前面所说的每个进程都独立的使用属于自己的内存一样，为了保证操作系统的稳定性，运行在操作系统中的用户程序，不能够访问操作系统所使用的内存空间。这也是从安全性上考虑，如访问硬件资源，只能由操作系统来发起，用户程序不允许直接访问硬件资源。如果用户程序需要访问硬件资源的话如网络连接等，可以调用操作提供的接口来实现，这个调用接口的过程也就是系统调用。每一次系统调用都会存在两个内存空间的切换，我通常的网络传输，也是一次系统调用，通过网络传输的数据先是从内核空间接收到远处主机的数据然后再从内核空间 cope 到用户空间，以供用户程序使用。这种从内核空间到用户空间数据 copy 很费时，虽然保住了程序运行的安全性和稳定性，但是也牺牲了一部分效率。但是现在已经出现了很多其它技术能够减少这种从内核空间到用户空间的数据 cope 的方式如：linux 提供了 sendfile 文件传输方式。

内核空间和用户空间的大小如何分配也是一个问题，是更多的分配给用户空间供用户程序使用呢？还是首先要保住内核有足够的空间来运行？这要平衡一下，如果是一台登陆服务器，那么很显然，要分配更多的内核空间，因为每一个登陆用户操作系统都会初始化一个用户进程，这个进程大部分显然都在内核空间里运行。当前的 windows 32 位操作系统中默认内核空间和用户空间的比例是 1:1（2G 的内核空间，2G 的用户空间），而 32 位 linux 系统中默认的比例是 1:3（1G 的内核空间，3G 的用户空间）。

Java 中哪些需要使用内存

Java 启动后也是作为一个进程运行在操作系统中，那么这个进程有哪些部分需要分配内存空间呢？

Java 堆

Java 堆是分配了对象的内存区域。大多数 Java SE 实现都拥有一个逻辑堆，但是一些专家级 Java 运行时拥有多个堆，比如实现 Java 实时规范（Real Time Specification for Java，RTSJ）的运行时。一个物理堆可被划分为多个逻辑扇区，具体取决用于管理堆内存的垃圾收集（GC）算法。这些扇区通常现为连续的本机内存块，这些内存块受 Java 内存管理器（包含垃圾收集器）控制。

堆的大小可以在 Java 命令行使用 -Xmx 和 -Xms 选项来控制（mx 表示堆的最大大小，ms 表示初始大小）。尽管逻辑堆（经常被使用的内存区域）可以根据堆上的对象数量和在 GC 上花费的时间而增大和缩小，但使用的本机内存大小保持不变，而且由 -Xmx 值（最大堆大小）指定。大部分 GC 算法依赖于被分配为连续的内存块的堆，因此不能在堆需要扩大时分配更多本机内存。所有堆内存必须预先保留。

保留本机内存与分配本机内存不同。当本机内存被保留时，无法使用物理内存或其它存储器作为备用内存。尽管保留地址空间块不会耗尽物理资源，但会阻止内存被用于其它用途。由保留从未使用的内存导致的泄漏与泄漏分配的内存一样严重。

当使用的堆区域缩小时，一些垃圾收集器会回收堆的一部分（释放堆的后备存储空间），从而减少使用的物理内存。

对于维护 Java 堆的内存管理系统，需要更多本机内存来维护它的状态。当进行垃圾收集时，必须分配数据结构来跟踪空闲存储空间和记录进度。这些数据结构的确切大小和性质因实现的不同而不同，但许多数据结构都与堆大小成正比。

线程

应用程序中的每个线程都需要内存来存储器堆栈（用于在调用函数时持有局部变量并维护状态的内存区域）。每个 Java 线程都需要堆栈空间来运行。根据实现的不同，Java 线程可以分为本机线程和 Java 堆栈。除了堆栈空间，每个线程还需要为线程本地存储（thread-local storage）和内部数据结构提供一些本机内存。

堆栈大小因 Java 实现和架构的不同而不同。一些实现支持为 Java 线程指定堆栈大小，其范围通常在 256KB 到 756KB 之间。

尽管每个线程使用的内存量非常小，但对于拥有数百个线程的应用程序来说，线程堆栈的总内存使用量可能非常大。如果运行的应用程序的线程数量比可用于处理它们的处理器数量多，效率通常很低，并且可能导致糟糕的性能和更高的内存占用。

类和类加载器

Java 应用程序由一些类组成，这些类定义对象结构和方法逻辑。Java 应用程序也使用 Java 运行时类库（比如 java。lang。String）中的类，也可以使用第三方库。这些类需要存储在内存中以备使用。

存储类的方式取决于具体实现。Sun JDK 使用永久生成（permanent generation，PermGen）堆区域。Java 5 的 IBM 实现会为每个类加载器分配本机内存块，并将类数据存储在其中。现代 Java 运行时拥有类共享等技术，这些技术可能需要将共享内存区域映射到地址空间。要理解这些分配机制如何影响您 Java 运行时的本机内存占用，您需要查阅该实现的技术文档。然而，一些普遍的事实会影响所有实现。

从最基本的层面来看，使用更多的类将需要使用更多内存。（这可能意味着您的本机内存使用量会增加，或者您必须明确地重新设置 PermGen 或共享类缓存等区域的大小，以装入所有类）。记住，不仅您的应用程序需要加载到内存中，框架、应用服务器、第三方库以及包含类的 Java 运行时也会按需加载并占用空间。

Java 运行时可以卸载类来回收空间，但是只有在非常严酷的条件下才会这样做。不能卸载单个类，而是卸载类加载器，随其加载的所有类都会被卸载。只有在以下情况下才能卸载类加载器：

- Java 堆不包含对表示该类加载器的 java。lang。ClassLoader 对象的引用。
- Java 堆不包含对表示类加载器加载的类的任何 java。lang。Class 对象的引用。
- 在 Java 堆上，该类加载器加载的任何类的所有对象都不再存活（被引用）。

需要注意的是，Java 运行时为所有 Java 应用程序创建的 3 个默认类加载器（ bootstrap、extension 和 application ）都不可能满足这些条件，因此，任何系统类（比如 java。lang。String）或通过应用程序类加载器加载的任何应用程序类都不能在运行时释放。

即使类加载器适当进行收集，运行时也只会将收集类加载器作为 GC 周期的一部分。一些实现只会在某些 GC 周期中卸载类加载器。

也可能在运行时生成类，而不用释放它。许多 JEE 应用程序使用 JavaServer Pages (JSP) 技术来生成 Web 页面。使用 JSP 会为执行的每个 jsp 页面生成一个类，并且这些类会在加载它们的类加载器的整个生存期中一直存在 —— 这个生存期通常是 Web 应用程序的生存期。

另一种生成类的常见方法是使用 Java 反射。反射的工作方式因 Java 实现的不同而不同，但 Sun 和 IBM 实现都使用了这种方法，我马上就会讲到。

当使用 java。lang。reflect API 时，Java 运行时必须将一个反射对象（比如 java。lang。reflect。Field）的方法连接到被反射到的对象或类。这可以通过使用 Java 本机接口（Java Native Interface，JNI）访问器来完成，这种方法需要的设置很少，但是速度缓慢。也可以在运行时为您想要反射到的每种对象类型动态构建一个类。后一种方法在设置上更慢，但运行速度更快，非常适合于经常反射到一个特定类的应用程序。

Java 运行时在最初几次反射到一个类时使用 JNI 方法，但当使用了若干次 JNI 方法之后，访问器会膨胀为字节码访问器，这涉及到构建类并通过新的类加载器进行加载。执行多次反射可能导致创建了许多访问器类和类加载器。保持对反射对象的引用会导致这些类一直存活，并继续占用空间。因为创建字节码访问器非常缓慢，所以 Java 运行时可以缓存这些访问器以备以后使用。一些应用程序和框架还会缓存反射对象，这进一步增加了它们的本机内存占用。

NIO

Java 1.4 中添加的新 I/O (NIO) 类引入了一种基于通道和缓冲区来执行 I/O 的新方式。就像 Java 堆上的内存支持 I/O 缓冲区一样，NIO 添加了对直接 ByteBuffer 的支持（使用 java。nio。ByteBuffer。allocateDirect() 方法进行分配），ByteBuffer 受本机内存而不是 Java 堆支持。直接 ByteBuffer 可以直接传递到本机操作系统库函数，以执行 I/O —— 这使这些函数在一些场景中要快得多，因为它们可以避免在 Java 堆与本机堆之间复制数据。

对于在何处存储直接 ByteBuffer 数据，很容易产生混淆。应用程序仍然在 Java 堆上使用一个对象来编排 I/O 操作，但持有该数据的缓冲区将保存在本机内存中，Java 堆对象仅包含对本机堆缓冲区的引用。非直接 ByteBuffer 将其数据保存在 Java 堆上的 byte[] 数组中。

直接 ByteBuffer 对象会自动清理本机缓冲区，但这个过程只能作为 Java 堆 GC 的一部分来执行，因此它们不会自动响应施加在本机堆上的压力。GC 仅在 Java 堆被填满，以至于无法为堆分配请求提供服务时发生，或者在 Java 应用程序中显示请求它发生（不建议采用这种方式，因为这可能导致性能问题）。

发生垃圾收集的情形可能是，本机堆被填满，并且一个或多个直接 ByteBuffers 适合于垃圾收集（并且可以被释放来腾出本机堆的空间），但 Java 堆几乎总是空的，所以不会发生垃圾收集。

JNI

JNI 支持本机代码（使用 C 和 C++ 等本机编译语言编写的应用程序）调用 Java 方法，反之亦然。Java 运行时本身极大地依赖于 JNI 代码来实现类库功能，比如文件和网络 I/O。JNI 应用程序可能通过 3 种方式增加 Java 运行时的本机内存占用：

JNI 应用程序的本机代码被编译到共享库中，或编译为加载到进程地址空间中的可执行文件。大型本机应用程序可能仅仅加载就会占用大量进程地址空间。

本机代码必须与 Java 运行时共享地址空间。任何本机代码分配或本机代码执行的内存映射都会耗用 Java 运行时的内

存。

某些 JNI 函数可能在它们的常规操作中使用本机内存。GetTypeArrayElements 和 GetTypeArrayRegion 函数可以将 Java 堆数据复制到本机内存缓冲区中，以供本机代码使用。是否复制数据依赖于运行时实现。（IBM Developer Kit for Java 5。0 和更高版本会进行本机复制）。通过这种方式访问大量 Java 堆数据可能会使用大量本机堆。

Java 内存结构

前面介绍了内存的不同形态：物理内存和虚拟内存，又介绍内存的使用形式：内核空间和用户空间，接着又介绍了 Java 有哪些组件需要使用内存。下面我们着重介绍 JVM 中是如何使用内存的。

JVM 划分内存结构是按照运行时数据的存储结构来划分的，JVM 在运行 Java 程序时，将它们划分成几种不同格式的数据，分别存储在不同的区域，这些数据统一称为运行时数据（Runtime Data）。运行时数据包括 Java 程序本身的数据信息和 JVM 运行 Java 程序需要的额外的数据信息，如要记录当前程序指令执行的指针又称为 pc 指针等。

在 Java 虚拟机规范中将 Java 运行时数据划分为 6 种，分别为：

- PC 寄存器数据
- Java 栈
- 堆
- 方法区
- 本地方法区
- 运行时常量池

PC 寄存器

PC 寄存器严格说是一个数据结构，它用于保存当前正常执行的程序的内存地址，同时 Java 程序是多线程执行的，所以不可能一直都是按照线性执行下去的，当有多个线程交叉执行时，必然那个被中断的线程的程序当前执行到哪条执行的内存地址就要保存下来，以便于它被恢复执行时再按照被中断时的指令地址继续执行下去，这很好理解，它就像一个记事员一样记录下哪个线程当前执行到哪条指令了。

但是 JVM 规范只定义了 Java 方法需要记录指针信息，而对于 native 方法，并没有要求记录执行的指针地址。

Java 栈

Java 栈总是和线程关联在一起的，每当创建一个线程时，JVM 就会为这个线程创建一个对应的 Java 栈，这个 Java 栈中又会含有多个栈帧（frames），这些栈帧是与每个方法关联起来的，每运行一个方法就是创建一个栈帧，每个栈帧会含有一些内部变量（在方法内定义的变量），操作栈和方法返回值等信息。

每当一个方法执行完成时这个栈帧就会弹出栈帧的元素作为这个方法的返回值，并清除这个栈帧，Java 栈的栈顶的栈帧就是当前正在执行的活动栈，也就是当前正在执行的方法，PC 寄存器也会指向这个地址，只有这个活动的栈帧的本地变量可以被操作栈使用，当这个栈帧中有调用另外一个方法时，与之对应一个新的栈帧又被创建，这个新创建的栈帧又被放到 Java 栈的顶部，变为当前的活动栈帧，同样现在只有这个栈帧的本地变量才能被使用，当这个栈帧中所有指令执行完成时这个栈帧移除出 Java 栈，刚才的那个栈帧又变为活动栈帧，前面的栈帧的返回值又变为这个栈帧的操

作栈中一个操作数，如果前面的栈帧没有返回值的话，那么当前的栈帧的操作栈的操作数没有变化。

由于 Java 栈是与 Java 线程对应起来的，这个数据不是线程共享的，所以我们不用关心，它的数据一致性问题也不会存在同步锁的问题。

堆

堆是存储 Java 对象的地方，它是 JVM 管理 Java 对象的核心存储区域，堆是 Java 程序员最应该关心的地方，因为他是我们的应用程序与内存关系最密切的存储区域。

每一个存储在堆中的 Java 对象都会是这个对象的类的一个拷贝，它会拷贝包括继承它父类的所有非静态属性。

堆是被所有 Java 线程所共享的，所以对它的访问需要注意同步问题，不管是方法和对应的属性都需要保证一致性。

方法区

JVM 方法区是用于存储 Class 结构信息的地方，如在前面一章中介绍的将一个 class 文件解析成 JVM 能识别的几个部分，这些不同的部分在这个 class 被加载到 JVM 时，会被存储在不同的数据结构中，其中如常量池、域、方法数据、方法体、构造函数、包括类中的专用方法、实例初始化、接口初始化都存储在这个区域。

方法区这个存储区域也是属于后面介绍的 Java 堆中的一部分，也就是我们通常所说的 Java 堆中的永久区。这个区域可以被所有的线程共享，并且它的分配区域的大小可以通过参数来设置。

这个方法区存储区域的大小一般在程序启动后的一段时间内是就是固定的了，JVM 运行一段时间后，需要加载的 Class 通常都已经加载到 JVM 中了，但是有一种情况是需要注意的，那就是在你的项目中如果存在对类的动态编译，而且是同样一个类的多次编译，那么要注意需要观察方法区的大小是否满足你的类存储。

方法区这个区域有点特殊，由于它不像其它 Java 堆一样会频繁被 GC 回收器回收，它的存储的信息相对比较稳定，但是它仍然是占用了 Java 堆的空间，所以仍然会被 JVM 的 GC 回收器来管理，在一些特殊的场合下，我们有时候通常需要缓存一块内容，这个内容也很少变动，但是如果把它置于 Java 堆中时它会不停的被 GC 回收器扫描，直到经过很长的时间后会进入 old 区。这种情况下，我们通常是能控制这个缓存区域中数据的生命周期的，我们不希望它被 JVM 内存管理，但是又希望它是在内存中。面对这种情况淘宝现在正在开发一种技术用于在 JVM 中分配另外一个内存存储区域，它不需要 GC 回收器来回收，但是和其它内存中对象一样来使用。

运行时常量池

JVM 规范中是这样定义运行时常量池这个数据结构的：Runtime Constant Pool 是代表运行时每个 class 文件中的常量表。它包括几种常量：编译期的数字常量、方法或者域的引用（在运行时解析）。runtime constant pool 的功能类似于传统编程语言的符号表，尽管它包含的数据比典型的符号表要丰富的多。每个 Runtime Constant pool 都是在 JVM 的 method area 中分配的，每个 class 或者 interface 的 constant pool 都是在 jvm 创建 class 或接口的时候创建的。

从上面的描述你可能有点迷惑，这个常量池与前面的方法区的常量池是否是一回事，答案是一回事。它是方法区的一部分，所以它的存储也是受方法区的规范约束，如果常量池无法分配，同样会报 OutOfMemoryError。

本地方法栈

本地方法栈是为 JVM 运行 Native 方法准备的空间，它和前面介绍的 Java 栈的作用是类似的，由于 Native 方法很多都是 C 语言实现的，所以它通常又叫 c 栈，除了我们的代码中包含的常规的 Native 方法会使用这个存储空间，在 JVM 利

用 JIT 技术时会将一些 Java 方法重新编译为 Native Code 代码，这些编译后的本地代码通常也是利用这个栈来跟踪方法的执行状态。

JVM 规范中没有对这个区域严格限制，它可以由不同的 JVM 实现者自由实现，但是它和其它存储区一样也会抛 OutOfMemoryError 和 StackOverflowError

Java 内存分配策略

在分析 Java 内存分配策略之前我们先介绍一下，通常情况下操作系统都是采用哪些策略来分配内存的？

通常的内存分配策略

我想大家都学过操作系统，在操作系统这门课上，将内存分配策略分为三种，分别是：

- 静态内存分配
- 栈内存分配
- 堆内存分配

静态存储分配是指在程序编译时就能确定每个数据在运行时刻的存储空间需求，因而在编译时就可以给它们分配固定的内存空间。这种分配策略要求程序代码中不允许有可变数据结构(比如可变数组)的存在，也不允许有嵌套或者递归的结构出现，因为它们都会导致编译程序无法计算准确的存储空间需求。

栈式存储分配也可称为动态存储分配，是由一个类似于堆栈的运行栈来实现的。和静态存储分配相反，在栈式存储方案中，程序对数据区的需求在编译时是完全未知的，只有到运行的时候才能够知道，但是规定在运行中进入一个程序模块时，必须知道该程序模块所需的数据区大小才能够为其分配内存。和我们在数据结构所熟知的栈一样，栈式存储分配按照先进后出的原则进行分配。

我们在编写程序时除了在编译时就能确定数据的存储空间和在程序入口处就能知道存储空间外，还有一种情况就是当程序真正运行到相应代码时才会知道空间大小，这种情况下我们就需要堆这种分配策略了。

这几种内存分配策略中，很明显堆这个分配策略是最自由的，但是这种分配测试对操作系统和内存管理程序来说是一种挑战，另外这个动态的内存分配是在程序运行时才执行的，它的运行效率也是比较差的。

Java 中内存分配

从前面的 Java 内存结构的分析中我们可知，Java 内存分配主要是基于两种，分别是：堆和栈。先来说说 Java 栈是如何分配的。

Java 栈的分配是和线程绑定在一起的，当我们创建一个线程时，很显然，JVM 就会为这个线程创建一个新的 Java 栈，一个线程的方法的调用和返回对应于这个 Java 栈的压栈和出栈。当线程激活一个 Java 方法，JVM 就会在线程的 Java 堆栈里新压入一个帧。这个帧自然成为了当前帧。在此方法执行期间，这个帧将用来保存参数、局部变量、中间计算过程和其它数据。

栈中主要存放一些基本类型的变量数据（int、short、 long、 byte、 float、 double、 boolean、 char）和对象句柄(引用)。存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是 确定的，缺乏灵活性。

如下面这段代码：

ToDo

每一个 Java 应用都唯一对应一个 JVM 实例，每一个实例唯一对应一个堆。应用程序在运行中所创建的所有类实例或数组都放在这个堆中，并由应用所有的线程共享。Java 中分配堆内存是自动初始化的，Java 中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在堆栈中分配，也就是说在建立一个对象时两个地方都分配内存，在堆中分配的内存实际建立这个对象，而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。

Java 的堆是一个运行时数据区，这些对象通过 new、newarray、 anewarray 和 multianewarray 等指令建立，它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的，Java 的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

如下所示堆是如何分配内存的：

ToDo

堆和栈的比较

上面的定义从编译原理的教材中总结而来，除静态存储分配之外，都显得很呆板和难以理解，下面撇开静态存储分配，集中比较堆和栈:

从堆和栈的功能和作用来通俗的比较，堆主要用来存放对象的，栈主要是用来执行程序的。而这种不同又主要是由于堆和栈的特点决定的

在编程中，例如 C/C++中，所有的方法调用都是通过栈来进行的，所有的局部变量、形式参数都是从栈中分配内存空间的。实际上也不是什么分配，只是从栈顶向上用就行，就好像工厂中的传送带(conveyor belt)一样，Stack Pointer 会自动指引你到放东西的位置，你所要做的只是把东西放下来就行。退出函数的时候，修改栈指针就可以把栈中的内容销毁。这样的模式速度最快， 当然要用来运行程序了。需要注意的是，在分配的时候，比如为一个即将要调用的程序模块分配数据区时，应事先知道这个数据区的大小，也就说是虽然分配是在程序运行时进行的，但是分配的大小多少是确定的、不变的，而这个"大小多少"是在编译时确定的，不是在运行时。

堆是应用程序在运行的时候请求操作系统给自己分配内存，由于操作系统管理内存分配，所以在分配和销毁时都要占用时间，因此用堆的效率非常低。但是堆的优点在于，编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间，因此，用堆保存数据时会得到更大的灵活性。事实上，面向对象的多态性，堆内存分配是必不可少的，因为多态变量所需的存储空间只有在运行时创建了对象之后才能确定。在 C++中，要求创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价:在堆里分配存储空间时会花掉更长的时间！这也正是导致我们刚才所说的效率低的原因，看来列宁同志说的好，人的优点往往也是人的缺点，人的缺点往往也是人的优点。

Java 内存回收策略

Java 语言和其它语言的一个很大不同之处就是 Java 开发人员不需要了解内存这个概念，因为在 Java 中没有什么语法和内存直接有联系，不像在 C 或 C++中有 malloc 这种语法直接操作内存。但是我们知道程序执行根据需要内存空间来支持，不然我们的那些数据存在哪里，那么 Java 语言没有提供直接操作内存的语法，那我们的数据又是如何申请内存的呢？就 Java 语言本身来说，通常显示的内存申请有两种：一种是静态内存分配；一种是动态内存分配。

静态内存分配和回收

Java 中静态内存分配是在 Java 被编译时就已经能够确定需要的内存空间，当程序被加载时系统就把内存一次性分配给它。这些内存不会在程序执行时发生变化，直到程序执行结束时内存才被回收掉。在 Java 中，静态内存分配的例子如：类中方法中的局部变量包括原生数据类型（int、long、char 等）和对象的引用。如下面这段代码：

```
    public void staticData(int arg){
          String s="String";
          long l=1;
          Long lg=1L;
          Object o = new Object();
          Integer i = 0;
    }
```

其中参数 arg、l 是原生的数据类型，s、o 和 i 是指向对象的引用。在 Javac 编译时就已经确定了这些变量的静态内存空间。其中 arg 会分配 4 个字节，long 会分配 8 个字节，String、Long、Object 和 Integer 是对象的类型，它们的引用会占用 4 个字节空间，所以这个方法占用的静态内存空间是 4+4+8+4+4+4=28 字节空间。

静态内存空间的回收是当这段代码运行结束时回收，根据前面一章的介绍，我们知道这些静态内存空间是在 Java 栈上分配的，当这个方法运行结束时，这个方法对应的栈帧也就是撤销，所以分配的静态内存空间也就回收了。

动态内存分配和回收

在前面的例子中变量 lg 和 i 存储的值虽然和 l 和 arg 变量一样，但是它们存储的位置是不一样的，后者是原生数据类型，它们存储在 Java 栈中，方法执行结束也就会消失，而前者是对象类型，它们存储在 Java 堆中，它们是可以被共享的，也不一定随着方法执行结束而消失。变量 l 和 lg 分配的内存空间大小显然也是不一样的，l 在 Java 栈中为它分配 8 个字节空间，而 lg 分配四个 4 个字节的地址指针空间，这个地址指针是指向这个对象在堆中的地址。很显然在堆中 long 类型数字 1 肯定不只 8 个字节的空间。所以 Long 代表的数字肯定表 long 类型占用的空间要大很多。

Java 中对象的内存空间分配是动态分配的，所谓的动态分配就是在程序执行时才知道要分配的存储空间，而不是在编译时就已经能够确定。lg 代表的 Long 对象，只有 JVM 在解析 Long 类时才知道这个类中有哪些 Field，这些 Field 都是哪些类型，然后在为这些 Field 分配相应的存储空间存储相应的值，而这个对象什么时候被回收也是不确定的。只有等到这个对象不再使用时才会被回收。

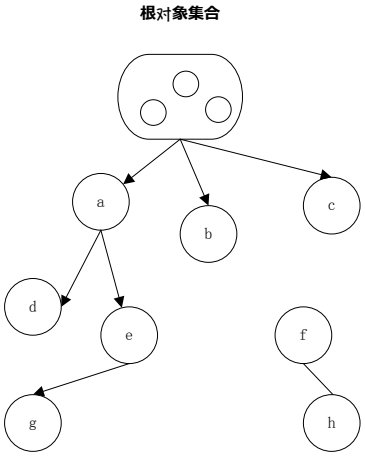
从前面的分析可知内存的分配是在对象创建时发生的，而内存的回收是在对象不再引用为前提的。这种动态内存的分配和回收是和 Java 中一些数据类型关联的，Java 程序员根本不需要关注内存的分配和回收，而只要关注这些数据类型的使用就行了。

那么如何确定这个对象什么时候不被使用，又如何来回收它们，这正是 JVM 的一个很重要的特性——垃圾收集器要解决的问题。

如何检测垃圾

垃圾收集器必须要能够完成两件事情：一个是能够正确的检测出垃圾对象；一个是能够释放垃圾对象占用的内存空间。其中如何检测出垃圾是垃圾收集器的关键所在。

从前面的分析已经知道只要是某个对象不再被其它活动对象引用，那么这个对象可以被回收了。这里所指的活动对象指的是能够被一个根对象集合所能够到达的对象。如下图所示：



活动对象

上图中除了 f 和 h 对象之外，其它都可以称之为活动对象，因为它们都可以被根对象集合所能够到达。h 对象虽然也被 f 对象引用但是 h 对象不能够被根对象集合达到，所以它们都是非活动对象，可以被垃圾收集器回收。

那么这个根对象集合又都是些什么呢？虽然根对象集合和 JVM 的具体实现也有关系，但是大体都会包含如下一些元素：

方法中局部变量区中对象的引用。如在前面的 staticData 方法中定义的 lg 和 o 等对象的引用就是作为一个根对象集合中的根对象，这些根对象时直接存储在栈帧中的局部变量区中。

Java 操作栈中的对象引用。有些对象是直接在操作中的持有的，所以操作栈肯定也包含根对象集合。

常量池中的对象引用。每个类都会包含一个常量池，这些常用池中也会包含有很多对象引用，如表示类名的字符串就是保存在堆中，那么常量池中只会持有这个字符串对象的引用。

本地方法中的持有的对象引用。有些对象被传入本地方法中，但是这些对象还没有被释放。

类的 Class 对象。当每个类被 JVM 加载时都会创建一个代表这个类的唯一数据类型的 Class 对象，而这个 Class 对象也同样存放在堆中，当这个类不在被使用时，在方法区中类数据和这个 Class 对象同样需要被回收。

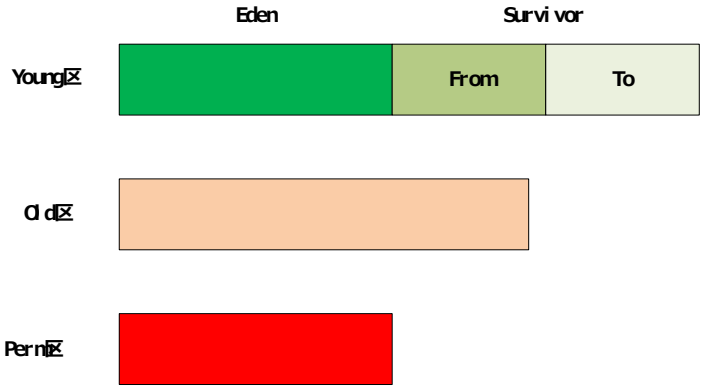
JVM 在做垃圾回收时会检查堆中的所有对象是否都会被这些根对象直接或者间接引用，能够被引用的对象就是活动对象，否则就可以被垃圾收集器回收。

基于按代的垃圾收集算法

经过这么长时间的发展垃圾收集算法已经发展了很多种，不同的垃圾各有优缺点，这里将主要介绍 hotspot 中使用的基于分代的垃圾收集方式。

该算法的设计思路是：通过把对象按照寿命长短来分组，分为年轻代和年老代，对象新创建的分在年轻代，如果对象经过几次回收后仍然存活，那么把这个对象再划分到年老代。年老代的收集频度不像年轻代那么平凡，这样就减少了每次垃圾收集所要扫描的对象的数量，从而提升了垃圾回收效率。

这种设计思路是把堆划分成若干个子堆，每个子堆对应一个年龄代，如下图所示：

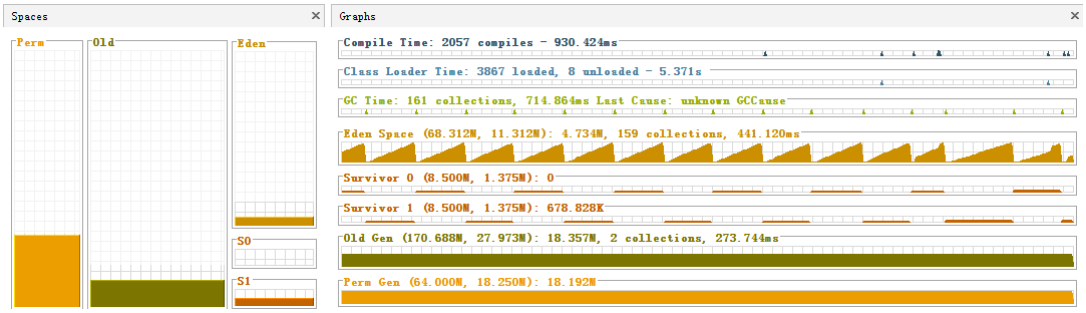


基于分代的堆结构

JVM 将整个堆划分为 Young 区、Old 区和 Perm 区，分别存放不同年龄的对象，这三个区存放的对象有如下区别：

- Young 区有分为 Eden 区和两个 Survivor 区，其中所有新创建的对象都在 Eden 区，当 Eden 区满后会触发 minor GC 将 Eden 区仍然存活的对象 copy 到其中一个 Survivor 区中，其中另外一个 Survivor 区中的存活对象也 copy 到这个 Survivor 中。以保证始终有一个 Survivor 区是空的。
- Old 区存放的是 Young 区的 Survivor 满后触发 minor GC 后仍然存活的对象，当 Eden 区满后会将对象存放到 Survivor 区中，如果 Survivor 区仍然存不下这些对象，GC 收集器会将这些对象直接存放到 Old 区。如果 Survivor 区中的对象足够老的话，对象也直接存放到 Old 区。如果 Old 区也满了话，将会触发 Full GC 回收整个堆内存。
- Perm 区主要存放的是类的 Class 对象，如果一个类频繁的被加载的话，也可能会导致 Perm 区满，Perm 区的垃圾回收也是有 Full GC 触发的。

Sun 的 JVM 中提供了一个 visualvm 工具其中有个 Visual GC 插件可以观察到 JVM 的不同代的垃圾回收情况，如下图所示：



Visual GC 插件

Visual GC 插件可以观察到每个代的当前内存大小，以及回收的次数等。

Sun 对堆中的不同代的大小也给出了建议，一般建议 Young 区的大小为整个堆的 1/4。而 Young 区中 Survivor 区一般设置为整个 Young 区的 1/8。

GC 收集器对这些区采用的垃圾收集算法也不一样，Hotspot 提供了三类垃圾收集算法可供配置，下面详细介绍这三类垃圾收集算法的区别和如何使用。这三类垃圾收集算法分别是：

- Serial Collector
- Parallel Collector
- CMS Collector

1. Serial Collector（串行）

Serial Collector 是 JVM 在 client 模式下默认的 GC 方式。可以通过 JVM 配置参数：-XX:+UseSerialGC 来指定 GC 使用串行收集算法来收集。我们指定所有的对象创建都是在 Young 区的 Eden 中创建。但是如果创建的对象超过的 Eden 区的总大小，或者超过了 PretenureSizeThreshold 配置参数配置的大小的话，就只能在 Old 区分配了，如 -XX:PretenureSizeThreshold= 30720，虽然这两种在实际使用中很少发生。

当 Eden 空间不足时就是触发 Minor GC，触发 Minor GC 时首先会检查之前每次 Minor GC 时晋升到 Old 区的平均对象大小是否大于 Old 区的剩余空间，如果大于的话则这次就将直接再触发 Full GC，如果小于的话，则再要看 HandlePromotionFailure 参数（-XX:-HandlePromotionFailure）设置的值。如果为 true 的话仅触发 Minor GC，否则再触发一次 Full GC，其实这个规则很好理解。如果每次晋升的对象大小都超过了 Old 区的剩余空间，那么说明当前的 Old 区的空间区已经经常不能满足新对象所占空间的大小，只有触发 Full GC 才能获得更多的内存空间。

如这个例子：

TODO

xulingbo 2012/3/9 9:12 PM
已设置格式: 正常, 空格 段前: 0 pt, 段后: 12 pt

当 Minor GC 时除了将 Eden 区的非活动对象回收掉以外，还会把一些老对象也 copy 到 Old 区中。这个老对象的定义是通过配置参数 MaxTenuringThreshold 来控制的，如-XX:MaxTenuringThreshold=10，则如果这个对象已经被 Minor GC

回收过 10 次后仍然存活，那么这个对象在这次 Minor GC 后直接放入 Old 区。还有一种情况，当这次 Minor GC 时 Survivor 区中的 To Space 放不下这些对象时，这些对象也将直接放入 Old 区。

如下这个例子：

TODO

如果 Old 区或者 Perm 区空间不足时，将会触发 Full GC，Full GC 会检查 Heap 堆中的所有对象，清楚所有垃圾对象，如果是 Perm 区的话会清除已经被卸载的 classloader 中加载的 class 的信息。

JVM 在做 GC 时由于是串行的，所以这些动作都是单线程完成的，JVM 的中的其它应用会全部停止。

2. Parallel Collector

Parallel GC 根据 Minor GC 和 Full GC 的不同有分为三种，分别是：ParNewGC、ParallelGC 和 ParallelOldGC

1) ParNewGC

可以通过-XX:+UseParNewGC 参数来指定，它的对象分配和回收策略与 Serial Collector 类似，只是回收的线程不是单线程而是多线程并行回收。Parallel Collector 中还有一个 UseAdaptiveSizePolicy 配置参数，这个参数是用来动态控制 Eden、From Space 和 To Space 的 TenuringThreshold 的大小，以便控制哪些对象经过多少次回收后可以直接放入 Old 区。

2) ParallelGC

Server 下默认 GC 方式，可以通过-XX:+UseParallelGC 参数来强制指定，并行回收的线程数可以通过-XX:ParallelGCThreads 来指定，这个数字的值有个计算公式，如果 CPU 和核数小于 8 的话，线程数可以和核心一样，如果大于 8 的话，值为：3+(cpu core*5)/8。

可以通过-Xmn 来控制 Young 区的大小，如-Xman10m，设置 Young 区的大小为 10m。Young 区内的 Eden、From Space 和 To Space 的大小控制可以通过 SurvivorRatio 参数来完成，如设置成这样-XX:SurvivorRatio=8，表示 Eden 区与 From Space 的大小为 8:1 如果 Young 区的总大小为 10m 的话，那么 Eden、s0 和 s1 的大小分别为 8m、1m 和 1m。但在默认情况下以-XX:InitialSurivivorRatio 设置的为准，这个值默认也为 8，表示的是 Young:s0 为 8:1。

当在 Eden 区中申请内存空间，Eden 区不够时，那么看当前申请的空间是否大于等于 Eden 的一半，如果大于则这次申请的空间直接在 Old 中分配，如果小于的话则触发 Minor GC，在触发 GC 之前首先会检查每次晋升到 Old 区的平均大小是否大于 Old 区的剩余空间，如大于则再触发 Full GC。在这次 GC 后仍然会按照这个规则重新检查一次。也就是如果上面这个规则满足的话，Full GC 会执行两次。

如下面这个例子：

TODO

在 Young 区的对象经过多次 GC 后有可能仍然存活，那么它们晋升到 Old 区的规则可以通过如下这些参数来控制：AlwaysTenure，默认 false，表示只要 Minor GC 时存活就晋升到旧生代；NeverTenure，默认 false，表示永远晋升到旧生代；上面两个都没设置的情况下，如设置 UseAdaptiveSizePolicy，启动时以 InitialTenuringThreshold 值作为

存活次数的阈值，在每次 GC 后会动态调整，如果不想使用 UseAdaptiveSizePolicy，则以 MaxTenuringThreshold 为准，不使用 UseAdaptiveSizePolicy 可以这样设置-XX:-UseAdaptiveSizePolicy。如果 Minor GC 时 To Space 不够的话，对象也将会直接放到 Old 区。

当 Old 或者 Perm 区空间不足时会触发 Full GC，如配置了参数 ScavengeBeforeFullGC 的话，在 Full GC 之前会先触发 Minor GC。

3) ParallelOldGC

可以通过-XX:+UseParallelOldGC 参数来强制指定，并行回收的线程数可以通过-XX:ParallelGCThreads 来指定，这个数字的值有个计算公式，如果 CPU 和核数小于 8 的话，线程数可以和核心一样，如果大于 8 的话，值为：3+(cpu core*5)/8。

它与 ParallelGC 有何不同呢？其实不同之处在于 Full GC 上，前者 Full GC 进行的动作为清空整个 Heap 堆中的垃圾对象，清楚 Perm 区中已经被卸载的 Class 信息，并行压缩。而后者是情况 Heap 堆中的部分垃圾对象，并进行部分的空间压缩。

GC 垃圾回收都是以多线程方式进行的，同样也将暂停所有应用。

3. CMS Collector

可通过-XX:+UseConcMarkSweepGC 来指定，并发的线程数默认为：(并行 GC 线程数+3)/4，也可通过 ParallelCMSThreads 指定。

CMS GC 与上面讨论的 GC 不太一样，它既不是上面所说的 Minor GC 也不是 Full GC，它是基于这两种之间的一种 GC，它的触发规则是检查 Old 区或者 Perm 区中的使用率，当达到一定比例时就会触发 CMS GC，触发时会回收 Old 区中的内存空间。这个比例可以通过 CMSInitiatingOccupancyFraction 参数来指定，默认是 92%，这个默认值是通过如下公式计算出来的((100 -MinHeapFreeRatio) +(double)(CMSTriggerRatio* MinHeapFreeRatio) / 100.0)/ 100.0，其中的 MinHeapFreeRatio 为 40、CMSTriggerRatio 为 80。如果让 Perm 区也使用 CMS GC 的话可以通过 -XX:+CMSClassUnloadingEnabled 来设定，Perm 区的比例默认值也是 92%，这个值可以通过 CMSInitiatingPermOccupancyFraction 设定。这个默认值也是通过一个公式计算出来的：((100 -MinHeapFreeRatio) +(double)(CMSTriggerPermRatio* MinHeapFreeRatio) / 100.0)/ 100.0，其中 MinHeapFreeRatio 为 40，CMSTriggerPermRatio 问哦 80。

CMS GC 触发时执行的回收只是 Old 区或者 Perm 区的垃圾对象，在回收时和前面所说的 Minor GC 和 Full GC 基本没有关系。

这个模式下的 Minor GC 触发规则和回收规则与 Serial Collector 基本是一致的，不同之处只是 GC 回收的线程是多线程而已。

触发 Full GC 的话是在这两种情况下发生：一种是 Eden 分配失败，Minor GC 后分配到 To Space，To Space 不够再分配的 Old 区，Old 区同样不够，则这时触发 Full GC。；另外一种情况是当 CMS GC 正在进行时向 Old 申请内存失败则会直接触发 Full GC。

这里还需要特别提醒一下的是在 Hotspot 1.6 中使用这种 GC 方式时在程序中显示的调用了 System.gc，且设置了

ExplicitGCInvokesConcurrent 参数。那么使用 NIO 时可能会引发内存泄露，这个内存泄露在后面再介绍。

CMS GC 何时执行 JVM 还会有一些时机选择，如当前的 CPU 是否繁忙等这些因素，因此它会有一个计算规则，并根据这个规程来动态调整。但是这也会给 JVM 带来另外的开销，如果去掉这个动态调整功能，禁止 JVM 自行触发 CMS GC 的话，可以通过配置参数-XX:+UseCMSInitiatingOccupancyOnly 来设置。

4. 组合使用这三种 GC

GC 组合	Young 区	Old 区
-XX:+UseSerialGC	串行 GC	串行 GC
-XX:+UseParallelGC	PSGC	并行 MSCGC
-XX:+UseParNewGC	并行 GC	串行 GC
-XX:+UseParallelOldGC	PSGC	并行 CompactingGC
-XX:+UseConcMarkSweepGC	ParNewGC	并发 GC 当出现 concurrentMode failure 时采用串行 GC
-XX:+UseConcMarkSweepGC -XX:-UseParNewGC	串行 GC	并发 GC 当出现 ConcurrentMode failure 或 promotionfailed 时则采用串行 GC
不支持的组合方式	1、-XX:+UseParNewGC–XX:+UseParallelOldGC 2、-XX:+UseParNewGC–XX:+UseSerialGC	

5. GC 参数列表集合

GC 方式	参数集合	
Heap 堆配置	-Xms/堆初始大小	
	–Xmx/堆最大值	
	–Xmn/Young 区大小	
	–XX:PermSize/Perm 区大小	
	–XX:MaxPermSize/Perm 区最大值	
Serial Collector（串行）	-XX:+UseSerialGC/GC 方式	
	-XX:SurvivorRatio/默认为 8，代表 eden:s0	
	-XX:MaxTenuringThreshold/默认为 15，代表对象在新生代经历多少次 MinorGC 后才晋升到 Old 区效率高当 Heap 过大时，应用暂停时间较长	
Parallel Collector（并行）	ParNewGC	-XX:+UseParNewGC/GC 方式
		-XX:SurvivorRatio/默认为 8，代表 eden:s0
		-XX:MaxTenuringThreshold/默认为 15
		-XX:+UseAdaptiveSizePolicy
	ParallelGC	-XX:+UseParallelGC/GC 方式
		-XX:ParallelGCThreads/并发线程数
		-XX:InitialSurivivorRatio/默认为 8,Young:s0 的比值
		-XX:+UseAdaptiveSizePolicy
		-XX:MaxTenuringThreshold/默认为 15
		-XX:+ScavengeBeforeFullGC/FullGC 前触发 MinorGC
	ParallelOldGC	-XX:+UseParallelOldGC/GC 方式

		其它同上
CMS Collector（并发）	-XX:+UseConcMarkSweepGC/GC 方式	
	-XX:ParallelCMSThreads/设置并发 CMS GC 时的线程数；	
	-XX:CMSInitiatingOccupancyFraction/当旧生代使用比率占到多少百分比时触发 CMS GC；	
	-XX:+UseCMSInitiatingOccupancyOnly/默认为 false，代表允许 hotspot 根据成本来决定什么时候执行 CMSGC；	
	-XX:+UseCMSCompactAtFullCollection/当 Full GC 时执行压缩；	
	-XX:CMSMaxAbortablePrecleanTime=5000/设置 preclean 步骤的超时时间，单位为毫秒；	
	-XX:+CMSClassUnloadingEnabled/PermGen 采用 CMS GC 回收。	

6. 三种 GC 优缺点对比

GC	优点	缺点
Serial Collector（串行）	适合内存有限的情况下 GC	回收慢
Parallel Collector（并行）	效率高	当 Heap 过大时，应用暂停时间较长
CMS Collector（并发）	Old 区回收暂停时间短	产生内存碎片、整个 GC 时间耗时较长、比较耗 CPU

内存问题分析

GC 日志分析

有时候内存溢出是我们可能并不知道何时会发生，但是当已经发生时，我们却可能并不知道原因，所以在 JVM 启动时就加上一些参数来控制，当 JVM 出问题时能记下一些当时的情况，还有就是记录下来 GC 的 log，我们可以观察 GC 的频度以及每次 GC 都回收了哪些内存。

GC 的 log 输出都有这些参数：

- -verbose:gc　可以辅劣输出一些详细的 GC 信息
- -XX:+PrintGCDetails　输出 GC 详细信息
- -XX:+PrintGCApplicationStoppedTime　输出 GC 造成应用暂停的时间
- -XX:+PrintGCDateStamps　GC 发生的时间信息
- -XX:+PrintHeapAtGC　在 GC 前后输出堆中各个区域的大小
- -Xloggc:[file]　将 GC 信息输出到单独的文件中

每种 GC 的 log 形式如下表所示：

GC 方式	log 日志形式
Serial Collector（串行）	[GC [DefNew: 11509K->1138K(14336K), 0.0110060 secs] 11509K->1138K(38912K), 0.0112610 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
	[Full GC [Tenured: 9216K->4210K(10240K), 0.0066570 secs] 16584K->4210K(19456K), [Perm : 1692K->1692K(16384K)], 0.0067070 secs][Times: user=0.00 sys=0.00, real=0.01 secs]

Parallel Collector（并行）	ParNewGC	[GC [ParNew: 11509K->1152K(14336K), 0.0129150 secs] 11509K->1152K(38912K), 0.0131890 secs] [Times: user=0.05 sys=0.02, real=0.02 secs]
		[GC [ASParNew: 7495K->120K(9216K), 0.0403410 secs] 7495K->7294K(19456K), 0.0406480 secs] [Times: user=0.06 sys=0.15, real=0.04 secs]
	ParallelGC	[GC [PSYoungGen: 11509K->1184K(14336K)] 11509K->1184K(38912K), 0.0113360 secs][Times: user=0.03 sys=0.01, real=0.01 secs]
		[Full GC [PSYoungGen: 1208K->0K(8960K)] [PSOldGen: 6144K->7282K(10240K)] 7352K->7282K(19200K) [PSPermGen: 1686K->1686K(16384K)], 0.0165880 secs] [Times: user=0.01 sys=0.01, real=0.02 secs]
	ParallelOldGC	[Full GC [PSYoungGen: 1224K->0K(8960K)] [ParOldGen: 6144K->7282K(10240K)] 7368K->7282K(19200K) [PSPermGen: 1686K->1685K(16384K)], 0.0223510 secs] [Times: user=0.02 sys=0.06, real=0.03 secs]
CMS Collector（并发）	[GC [1 CMS-initial-mark: 13433K(20480K)] 14465K(29696K), 0.0001830 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] [CMS-concurrent-mark: 0.004/0.004 secs] [Times: user=0.01 sys=0.00, real=0.01 secs] [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] CMS: abort precleandue to time [CMS-concurrent-abortable-preclean: 0.007/5.042 secs] [Times: user=0.00 sys=0.00, real=5.04 secs] [GC[YG occupancy: 3300 K (9216 K)][Rescan (parallel) , 0.0002740 secs] [weak refs processing, 0.0000090 secs] [1 CMS-remark: 13433K(20480K)] 16734K(29696K), 0.0003710 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] [CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] [CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]	

除去 CMS 的 log 与其它 GC 的 log 差别较大外，它们都可以抽象成如下这个格式：

[GC [<collector>: <starting occupancy1> -> <ending occupancy1>（total size1）, <pause time1> secs] <starting occupancy2> -> <ending occupancy2>(total size2), <pause time2> secs]

其中代表的意思分别如下所示：

- <collector>GC 收集器的名称
- <starting occupancy1> Young 区在 GC 前占用的内存
- <ending occupancy1> Young 区在 GC 后占用的内存
- <pause time1> Young 区局部收集时 jvm 暂停处理的时间
- <starting occupancy2> JVM Heap 在 GC 前占用的内存
- <ending occupancy2> JVM Heap 在 GC 后占用的内存
- <pause time2> GC 过程中 jvm 暂停处理的总时间

可以根据 log 日志来判断是否有内存在泄露，如果<ending occupancy1>-<starting occupancy1>=<ending

occupancy2>-<starting occupancy2>，则表明这次 GC 对象 100%被回收，没有对象进入到 Old 区或者 Perm 区，如果前面的值大于等号后面的值那么差值就是这次回收对象进入 Old 区或者 Perm 区的大小。如果随着时间的延长<ending occupancy2>的值一直在增长的话，而且 Full GC 很频繁，那么很可能就是内存泄露了。

除去 log 日志文件分析外，还可以直接通过 JVM 自带的一些工作直接分析如 jstat，使用格式如：jstat –gcutil [pid] [intervel] [count]，如下图所示：

如下面这个日志：

```
[junshan@tbskip027085 junshan]$ sudo /opt/taobao/java/bin/jstat -gcutil 29723 500 100
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
  0.00   4.05  30.56  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  32.79  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  35.96  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  38.23  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  40.45  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  43.07  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  46.16  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  49.05  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  50.86  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  54.49  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  57.27  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  59.62  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  63.03  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  65.81  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  68.51  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  71.16  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  74.38  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  76.94  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  79.27  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  82.74  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  85.85  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  88.35  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  91.05  39.49  45.22   5401   68.689    66   22.519   91.209
  0.00   4.05  93.45  39.49  45.22   5401   68.689    66   22.519   91.209
```

上面的参数是如下含义：

- S0　Heap 上的 Survivor space 0 区已使用空间的百分比
- S1　Heap 上的 Survivor space 1 区已使用空间的百分比
- E　Heap 上的 Eden space 区已使用空间的百分比
- O　Heap 上的 Old space 区已使用空间的百分比
- P　Perm space 区已使用空间的百分比
- YGC　从应用程序启动到采样时发生 Young GC 的次数
- YGCT　从应用程序启动到采样时 Young GC 所用的时间(单位秒)
- FGC　从应用程序启动到采样时发生 Full GC 的次数
- FGCT　从应用程序启动到采样时 Full GC 所用的时间(单位秒)
- GCT　从应用程序启动到采样时用于垃圾回收的总时间(单位秒)

堆快照文件分析

可通过命令 jmap –dump:format=b,file=[filename] [pid]来 Dump 下堆的内存快照。然后利用第三方工具来如 mat 来分析整个 Heap 的对象关联情况。具体分析看后面的实例介绍。

如果 OOM 那么可能导致 JVM 直接 Crash，可以通过参数：-XX:+HeapDumpOnOutOfMemoryError 来配置当 OOM 时记

录下内存快照，可以通过-XX:HeapDumpPath 来指定文件的路径，这个文件的名称的命名格式如 java_[pid].hprof

JVM Crash 日志分析

JVM 有时候也会因为一些原因而导致直接 Crash，因为 JVM 本身也是一个正在运行的程序，这个程序本身也会有很多情况直接出问题，如 JVM 本身也有一些 Bug，这些 Bug 可能会导致 JVM 异常退出。JVM 退出一般会在工作目录下产生一个 log 日志文件，也可以通过 JVM 参数来设定，如：-XX:ErrorFile=/tmp/log/hs_error_%p.log。

下面是一个日志格式文件：

```
    #
    # A fatal error has been detected by the Java Runtime Environment:
    #
    #  SIGSEGV (0xb) at pc=0x00002ab12ba7103a, pid=7748, tid=1363515712
    #
    # JRE version: 6.0_26-b03
    # Java VM: OpenJDK 64-Bit Server VM (20.0-b11-internal mixed mode linux-amd64 )
    # Problematic frame:
    # V  [libjvm.so+0x8bf03a]  jni_GetFieldID+0x22a
    #
    # If you would like to submit a bug report, please visit:
    #   http://java.sun.com/webapps/bugreport/crash.jsp
    #
    ---------------  T H R E A D  ---------------
    Current thread (0x00002aabd0ba5000):  JavaThread "http-0.0.0.0-7001-32" daemon [_thread_in_vm, id=8192,
stack(0x0000000051359000,0x000000005145a000)]

    siginfo:si_signo=SIGSEGV: si_errno=0, si_code=1 (SEGV_MAPERR), si_addr=0x0000000000000010

    Registers:
    ...
    Top of Stack: (sp=0x0000000051455620)
    ...
    Instructions: (pc=0x00002ab12ba7103a)
    0x00002ab12ba7101a:   01 00 00 48 8b 5f 10 48 8d 43 08 48 3b 47 18 0f
    0x00002ab12ba7102a:   87 53 02 00 00 48 89 47 10 48 89 13 48 83 c2 10
    0x00002ab12ba7103a:   48 8b 0a 48 89 d7 4c 89 fe 31 c0 ff 51 58 49 8b
    0x00002ab12ba7104a:   47 08 48 85 c0 0f 85 f7 01 00 00 48 c7 45 90 00
    Register to memory mapping:

    RAX=
    [error occurred during error reporting (printing register info), id 0xb]

    Stack: [0x0000000051359000,0x000000005145a000],  sp=0x0000000051455620,  free space=1009k
    Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
    V  [libjvm.so+0x8bf03a]  jni_GetFieldID+0x22a
    C  [libocijdbc10.so+0xcc81]  Java_oracle_jdbc_driver_T2CConnection_t2cDescribeError+0x205
    C  [libocijdbc10.so+0x878b]  Java_oracle_jdbc_driver_T2CConnection_t2cCreateState+0x193
    j  oracle.jdbc.driver.T2CConnection.t2cCreateState([BI[BI[BI[BISI[S[B[B)I+0
    j  oracle.jdbc.driver.T2CConnection.logon()V+589
    j
oracle.jdbc.driver.PhysicalConnection.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/St
ring;Ljava/util/Properties;Loracle/jdbc/driver/OracleDriverExtension;)V+370
    j
oracle.jdbc.driver.T2CConnection.<init>(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;
Ljava/util/Properties;Loracle/jdbc/driver/OracleDriverExtension;)V+10
    j
oracle.jdbc.driver.T2CDriverExtension.getConnection(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/
lang/String;Ljava/util/Properties;)Ljava/sql/Connection;+67
    j
oracle.jdbc.driver.OracleDriver.connect(Ljava/lang/String;Ljava/util/Properties;)Ljava/sql/Connection;+831
    ...
    ---------------  P R O C E S S  ---------------

    Java Threads: ( => current thread )
```

```
      0x000000004d11e800 JavaThread "IdleRemover" daemon [_thread_blocked, id=8432,
stack(0x00000000584ca000,0x00000000585cb000)]
    =>0x00002aabd0ba5000 JavaThread "http-0.0.0.0-7001-32" daemon [_thread_in_vm, id=8192,
stack(0x0000000051359000,0x000000005145a000)]
    ...
    VM state:not at safepoint (normal execution)

    VM Mutex/Monitor currently owned by a thread: None

    Heap
     par new generation   total 1474560K, used 1270275K [0x00002aaaae0f0000, 0x00002aab120f0000, 0x00002aab120f0000)
      eden space 1310720K,  89% used [0x00002aaaae0f0000, 0x00002aaaf5d634e0, 0x00002aaafe0f0000)
      from space 163840K,  57% used [0x00002aab080f0000, 0x00002aab0dcfd748, 0x00002aab120f0000)
      to   space 163840K,   0% used [0x00002aaafe0f0000, 0x00002aaafe0f0000, 0x00002aab080f0000)
     concurrent mark-sweep generation total 2555904K, used 888664K [0x00002aab120f0000, 0x00002aabae0f0000,
0x00002aabae0f0000)
     concurrent-mark-sweep perm gen total 262144K, used 107933K [0x00002aabae0f0000, 0x00002aabbe0f0000,
0x00002aabbe0f0000)

    Code Cache  [0x00002aaaab025000, 0x00002aaaabcd5000, 0x00002aaaae025000)
     total_blobs=3985 nmethods=3447 adapters=491 free_code_cache=37205440 largest_free_block=30336

    Dynamic libraries:
    40000000-40009000 r-xp 00000000 ca:06 224241                             /opt/taobao/install/jdk-1.6.0_26/bin/java
    ...
    ---------------  S Y S T E M  ---------------

    OS:Red Hat Enterprise Linux Server release 5.4 (Tikanga)
```

这个文件中的信息主要分为四种：退出原因分类、导致退出的 Thread 信息、退出时的 Process 状态信息、退出时与操作系统 System 相关信息。

JVM 退出一般有三种主要原因导致，如上面这个例子中是 SIGSEGV (0xb)，这三种分别如下：

1. EXCEPTION_ACCESS_VIOLATION

正在运行 JVM 自己的代码，而不是外部的 Java 代码或其它类库代码。这种情况很可能是 JVM 的自己的 Bug，遇到这种错误时，可以根据出错信息到 http://bugreport.sun.com/bugreport/index.jsp 地址去搜索一下以及发行的 Bug。

大部分情况下这个原因是由于 JVM 的内存回收导致的，所以可以观察 Process 部分的信息查看堆的内存占用情况。

2. SIGSEGV

JVM 正在执行本地或 JNI 的代码，出这种错误很可能是第三方的本地库有问题，可以通过 gbd 和 core 文件来分析出错原因。

3. EXCEPTION_STACK_OVERFLOW

这是个栈溢出的错误，注意 JVM 在执行 Java 线程时出现的栈溢出通常不会导致 JVM 退出，而是抛出 java.lang.StackOverflowError，但是在 Java 虚拟机中，Java 的代码和本地 C 或 C++代码公用相同的 Stack，这是如果出现栈溢出的话，就有可能直接导致 JVM 退出。建议将 JVM 的 Stack 的尺寸调大，主要设计两个参数：“-Xss”和“-XX:StackShadowPages=n”。

日志文件的 Thread 部分的信息对我们排查问题的原因最有帮助，这部分有两个关系信息包括 Machine Instructions 和 Thread Stack。Instructions 是当前系统执行的机器指令，这里显示的 16 进制，我们可以将它转成指令，可通过 udis86 工具来转换，可以在 http://udis86.sourceforge.net/ 下载到，安装在 linux 中，将上面的 16 进制数字 copy 到命令行中用如下方式执行：

```
[junshan@v024153.sqa.cm4 ~]$ echo "47 08 48 85 c0 0f 85 f7 01 00 00 48 c7 45 90 00"  | udcli -64 -x
0000000000000000 47084885        or [r8-0x7b], r9b
0000000000000004 c00f85          ror byte [rdi], 0x85
0000000000000007 f701000048c7    test dword [rcx], 0xc7480000
000000000000000d 4590            xchg r8d, eax
```

可以得到汇编指令形式，由于是 64 位机器，所以是 udcli -64 –x，如果是 32 的话，改成 udcli -32 –x。可以通过这个指令来判断当前正在执行什么指令导致了 Crash，如当前访问的寄存器地址，那么这个地址是否合法，如果是除法指令操作数是否合法等。

而 Stack 信息最直接，可以帮助我们看到到底在哪个库的哪行代码出错，如上面的错误信息中显示的是由于执行 Oracle 的 Java 驱动程序引起出错的。我们还可以通过生成的 core 文件来更详细的看出执行到那行代码出错的，如下所示：

```
$gdb /opt/taobao/java/bin/java /home/admin/tbskip/target/core.14595
GNU gdb Fedora (6.8-37.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
…
(gdb) bt
#0  0x000000320ea30265 in raise () from /lib64/libc.so.6
#1  0x000000320ea31d10 in abort () from /lib64/libc.so.6
#2  0x00002b4ba59d80e9 in os::abort () from /opt/taobao/install/jdk-1.6.0_26/jre/lib/amd64/server/libjvm.so
#3  0x00002b4ba59d1e0f in VMError::report_and_die () from
/opt/taobao/install/jdk-1.6.0_26/jre/lib/amd64/server/libjvm.so
…
```

通过 gdb 来调试 core 文件可以看到更详细的信息，还可以通过 frame n 和 info local 组合命令来更进一步查看这一行执行所包含的 local 变量值，但这只能是程序在使用-g 命名编译的结果，也就是编译后的程序包含 debug 信息。

日志文件的第三部分包含的是 Process 信息，这里详细列出了该程序产生的所有线程，以及线程正处于的状态，由于同一时刻只能是一个线程用于 CPU 使用权，所以你可以看到，其它所有的线程的状态都是_thread_blocked，而执行的正是那个出错线程。

这部分最有价值的部分就是记录下来了当前 JVM 的堆信息，如下所示：

```
Heap
 par new generation   total 1474560K, used 1270275K [0x00002aaaae0f0000, 0x00002aab120f0000, 0x00002aab120f0000)
  eden space 1310720K,  89% used [0x00002aaaae0f0000, 0x00002aaaf5d634e0, 0x00002aaafe0f0000)
  from space 163840K,  57% used [0x00002aab080f0000, 0x00002aab0dcfd748, 0x00002aab120f0000)
  to   space 163840K,   0% used [0x00002aaafe0f0000, 0x00002aaafe0f0000, 0x00002aab080f0000)
 concurrent mark-sweep generation total 2555904K, used 888664K [0x00002aab120f0000, 0x00002aabae0f0000,
0x00002aabae0f0000)
 concurrent-mark-sweep perm gen total 262144K, used 107933K [0x00002aabae0f0000, 0x00002aabbe0f0000,
0x00002aabbe0f0000)
```

每个分区当前所使用的空间大小，尤其是 Old 区的空间是否已经满了，可以判断出当前的 GC 是否正常。

还有一个信息也比较有价值那就是当前 JVM 的启动参数，如设置的堆大小和使用的 GC 方式等都可以从这里看出。

最后一部分是 System 信息，这部分主要记录了当前操作系统的状态信息，如操作系统的 CPU 信息和内存情况等。

实例 1

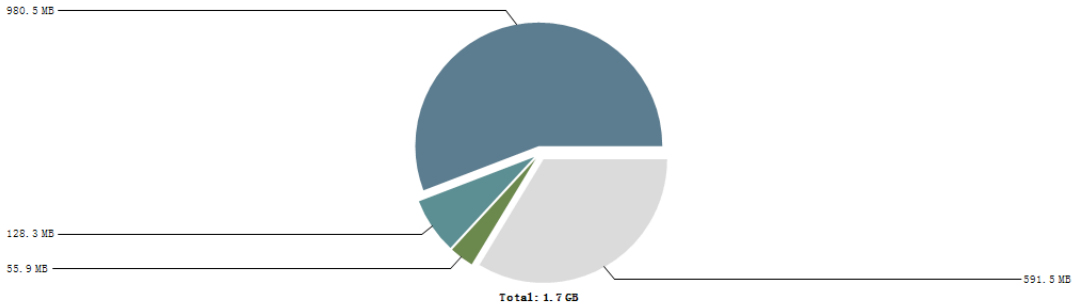
这里有一个 Java 内存泄露的实例，是淘宝一个系统中发生的，这个问题的原因也是我写的一个模板引擎导致的，这个模板引擎的介绍在后面的章节中再介绍。

当时的情况是系统 load 偏高达到了 6 左右，而平时基本在 1 左右，系统整个响应较慢，但是重启系统之后就恢复正常。于是查看 GC 情况发现 FGC 情况明显超出正常情况,并且 gc 过程中出现 concurrent mode failure。如下 log 所示：

```
     [GC 642473.656: [ParNew: 2184576K->2184576K(2403008K), 0.0000280 secs]642473.656:
[CMS2011-09-30T03:53:22.209+0800: 642473.656: [CMS-concurrent-abortable-preclean: 0.064/1.953 secs] [Times:
user=0.06 sys=0.00, real=1.95 secs]
     (concurrent mode failure): 1408475K->1422713K(1572864K), 6.0568470 secs] 3593051K->1422713K(3975872K), [CMS Perm :
77027K->77005K(200704K)], 6.0570590 secs] [Times: user=6.05 sys=0.00, real=6.06 secs]
```

说明在 Olde 区分配内存的时候出现分配失败的现象，而且整个内存占用得到了 6G 左右，超出了平时的 4G 于是得出可能是有内存泄露的问题。

通过 jmap –dump:format=b,file=[filename] [pid]命令 Dump 出 Heap，通过 mat 工具分析：



com.alibaba.webx.service.DefaultWebxServiceManager @ 0x7908d1318

Shallow Size: **104 B** Retained Size: **980.5 MB**

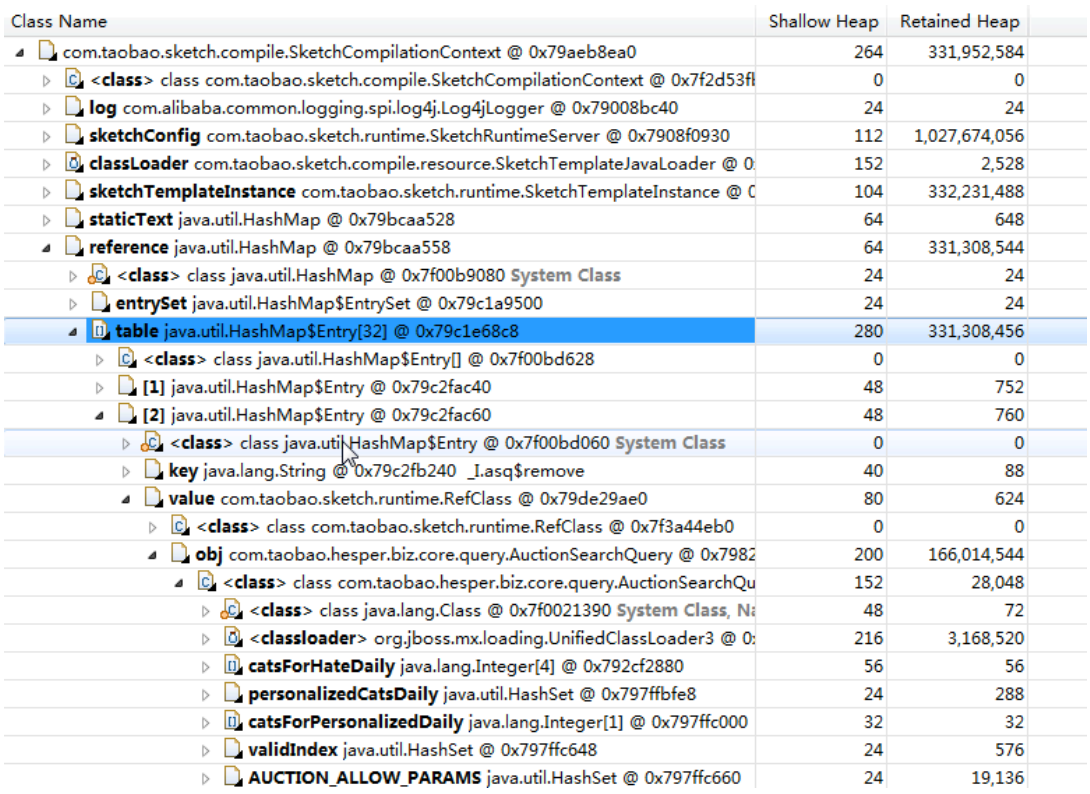
上图中最大的一个对象占用了九百多兆内存，这显然有问题。下面是 mat 给出的可能有问题的对象的说明，指出了 Map 集合占用了 55%的空间。

Problem Suspect 1

One instance of **"com.alibaba.webx.service.DefaultWebxServiceManager"** loaded by **"org.jboss.mx.loading.UnifiedClassLoader3 @ 0x7909290a0"** occupies **1,028,119,560 (55.83%)** bytes. The memory is accumulated in one instance of **"java.util.concurrent.ConcurrentHashMap$Segment[]"** loaded by **"<system class loader>".Keywords**
java.util.concurrent.ConcurrentHashMap$Segment[]
org.jboss.mx.loading.UnifiedClassLoader3 @ 0x7909290a0
com.alibaba.webx.service.DefaultWebxServiceManager

Details »

在看一下到底这个对象都持有了哪些对象。如下图所示：



Class Name	Shallow Heap	Retained Heap
com.taobao.sketch.compile.SketchCompilationContext @ 0x79aeb8ea0	264	331,952,584
<class> class com.taobao.sketch.compile.SketchCompilationContext @ 0x7f2d53f	0	0
log com.alibaba.common.logging.spi.log4j.Log4jLogger @ 0x79008bc40	24	24
sketchConfig com.taobao.sketch.runtime.SketchRuntimeServer @ 0x7908f0930	112	1,027,674,056
classLoader com.taobao.sketch.compile.resource.SketchTemplateJavaLoader @ 0	152	2,528
sketchTemplateInstance com.taobao.sketch.runtime.SketchTemplateInstance @ 0	104	332,231,488
staticText java.util.HashMap @ 0x79bcaa528	64	648
reference java.util.HashMap @ 0x79bcaa558	64	331,308,544
<class> class java.util.HashMap @ 0x7f00b9080 System Class	24	24
entrySet java.util.HashMap$EntrySet @ 0x79c1a9500	24	24
table java.util.HashMap$Entry[32] @ 0x79c1e68c8	280	331,308,456
<class> class java.util.HashMap$Entry[] @ 0x7f00bd628	0	0
[1] java.util.HashMap$Entry @ 0x79c2fac40	48	752
[2] java.util.HashMap$Entry @ 0x79c2fac60	48	760
<class> class java.util.HashMap$Entry @ 0x7f00bd060 System Class	0	0
key java.lang.String @ 0x79c2fb240 _I.asq$remove	40	88
value com.taobao.sketch.runtime.RefClass @ 0x79de29ae0	80	624
<class> class com.taobao.sketch.runtime.RefClass @ 0x7f3a44eb0	0	0
obj com.taobao.hesper.biz.core.query.AuctionSearchQuery @ 0x7982	200	166,014,544
<class> class com.taobao.hesper.biz.core.query.AuctionSearchQu	152	28,048
<class> class java.lang.Class @ 0x7f0021390 System Class, Na	48	72
<classloader> org.jboss.mx.loading.UnifiedClassLoader3 @ 0x	216	3,168,520
catsForHateDaily java.lang.Integer[4] @ 0x792cf2880	56	56
personalizedCatsDaily java.util.HashSet @ 0x797ffbfe8	24	288
catsForPersonalizedDaily java.lang.Integer[1] @ 0x797ffc000	32	32
validIndex java.util.HashSet @ 0x797ffc648	24	576
AUCTION_ALLOW_PARAMS java.util.HashSet @ 0x797ffc660	24	19,136

对象大小

上图中的第一列是类名，第二例 Shallow Heap 是表示这个对象本身的对象大小，所谓对象本身大小就是这个对象的一些 Field 中直接分配的存储空间，如这样定义 byte[] byte=new byte[1024]，那么这个 byte 属性的大小就直接包含在这个对象的 Shallow 中。而如果这个对象的某些属性是指向一个对象的话，那么所指向的那个对象的大小就计算在 Retained Heap 中。

上图中 SketchCompileContext 对象持有一个 Map 集合，这个 Map 集合所占用的空间很大，仔细查看后这个 Map 所持有的一个 DO 对象，这个对象的确是一个大对象，它的大小并没有超出我们的预期，仔细查看其它集合，没有发现所持有对象有什么不对的地方。但是仔细计算整个对象集合的大小发现虽然所有的对象都是应该存在的，但是比我们计算的正常的大小空间将近多了一倍左右，于是我们考虑到可能是持有了两份同样的对象。

按照这个思路我们仔细搜索了这个 Map 集合中的对象的数值发现，果然同样一个数值，有两个不同的 DO 对象对应，但是为什么有两个 DO 对象呢？按照正常情况应该单例啊，怎么会产生两份 DO 对象，后面仔细检查这个 DO 对象的业务逻辑，原来是这个 DO 要在每天凌晨 2 点要更新一次，更新后老对象会自动释放，但是我们这个新引擎是要保存住这些对象，以便于做编译优化，所以不能及时的释放掉这个更新后的老对象。所以导致这个大对象在内存中保存了两份。

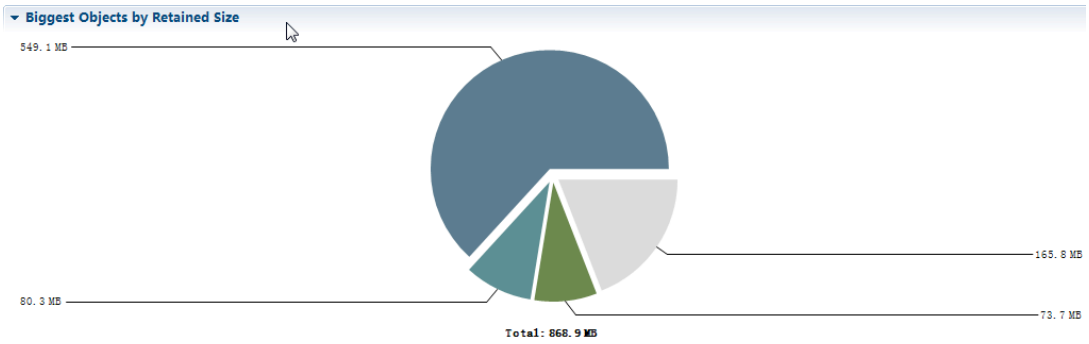
实例 2

这个例子和前面介绍的 CMS GC 回收方式的一个 JVM 的 bug 相关，淘宝的某应用，某天突然线上有部分机器会报警，Java 的内存使用非常严重，得到了 6G，超过了平时的 4G，而且有几台机器一段时间后导致 OOM、JVM 退出。当时第一反应是重启部分机器，保留几台有问题的机器来寻找原因。

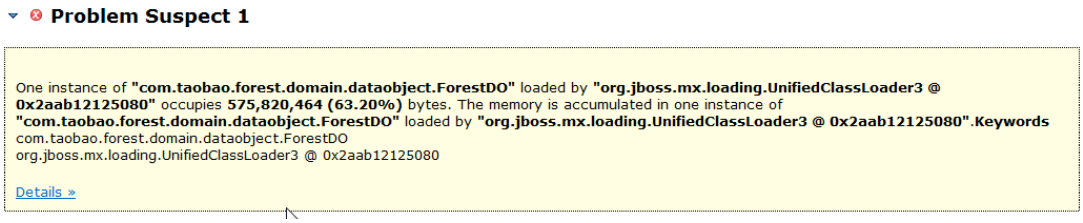
观察重启后的机器，发现应用恢复正常，但是发现 JVM 进程占用的内存一直在增长，可以大体推断出是 JVM 有内存泄露。然后检查最近是否有系统改动，是否是 Java 代码问题导致了内存泄露。最近一周 Java 代码改动的很少，而且代码 Review 也没有发现有内存泄露问题。

同时检查 GC 的日常，发现有问题的机器的 Full GC 没有异常，甚至 Full GC 比平时还少，CMS GC 也很少。从日常中没有发现可能有内存问题。

为了进一步确认 GC 是否正常于是 Dump 出 JVM 的 Heap，用 mat 分析堆文件，堆的使用情况如下图所示：



可以看出整个 Heap 只有不到 1G 的空间，而且从 Leak Suspects 给出的报告中占有最大空间的对象时一个 DO 对象，如下图：



而这个对象的大小也符合我们的预期，所以可以得出判断，不是 JVM 的堆内存有问题，但是既然 JVM 的堆占有的内存并不多，那么 Java 进程为什么占有了那么多内存呢？

于是想到了可能是堆外分配的内存有泄漏，从前面的分析中我们已经知道，JVM 除了堆需要内存外还有很多方面也需要在运行时使用内存，如 JVM 本身 JIT 编译需要内存，JVM 的栈也需要内存，还有 JNI 调用本地代码也需要内存，还有 NIO 方式也会使用 Direct Buffer 来申请内存。

从这些因素中我们推断可能是 Direct Buffer 导致的，因为在上次发布中引入过一个 Apache 的 Mina 包，这个包中肯定使用了 Direct Buffer。但是为什么 Direct Buffer 没有正常回收呢？很奇怪。

这时中间件的景升同学想到了可能是 JVM 的一个 bug，详见 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6919638。外部调用了 System.gc，且设置了-XX:+DisableExplicitGC，所以导致 System.gc()变成了空调用，而应用 GC 却很少，这里的 GC 包括 CMS GC 个 Full GC。所以 Direct Buffer 对象还没及时被回收，相应的 native memory 不能被释放。为了验证这一点，撒迦同学还写了一个工具可以检查当前 JVM 中 NIO direct memory 的使用情况，如下所示：

```
      [sajia@tbskip022053.cm4 ~]$ sjdirectmem `s pgrep java`
Attaching to process ID 3543, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version 20.1-b02. You may see unexpected re
sults.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NIO direct memory: (in bytes)
  reserved size = 163.417320 MB (171355480 bytes)
  max size      = 3936.000000 MB (4127195136 bytes)
    [sajia@tbskip018156.cm4 ~]$ sjdirectmem 25332
Attaching to process ID 25332, please wait...
WARNING: Hotspot VM version 20.0-b11-internal does not match with SA version 20.1-b02. You may see unexpected re
sults.
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11-internal
NIO direct memory: (in bytes)
  reserved size = 1953.929705 MB (2048843794 bytes)
  max size      = 3936.000000 MB (4127195136 bytes)
```

可以看出一段时间后 NIO direct memory 的确增长了很多，所以可以肯定的是 NIO direct memory 没有释放导致 Java 进程占用的内存持续增长。

这个问题的解决办法是去掉 -XX:+DisableExplicitGC，换上 -XX:+ExplicitGCInvokesConcurrent，使得外部的显示 System.gc 调用生效，这样即使 Java GC 不频繁时通过外部主动调用 System.gc 来回收 NIO direct memory 分配的内存。

总结