**自动内存管理机制**

**Java虚拟机栈(线程私有)**

虚拟机栈描述的是Java方法执行的内存模型:每个方法在执行的同时

都会创建一个栈帧(Stack Frame )用于存储局部变量表、操作数栈、动态链接、方法出口

等信息。每一个方法从调用直至执行完成的过程,就对应着一个栈帧在虚拟机栈中入栈到出

栈的过程。

局部变量表存放了编译期可知的各种基本数据类型(boolean、byte、char、short、int、

float、long、double)、对象引用(reference类型,它不等同于对象本身,可能是一个指向对

象起始地址的引用指针,也可能是指向一个代表对象的句柄或其他与此对象相关的位置)和

returnAddress类型(指向了一条字节码指令的地址)

**Java堆(线程共享)**

对于大多数应用来说,Java堆(Java Heap)是Java虚拟机所管理的内存中最大的一块。

Java堆是被所有线程共享的一块内存区域,在虚拟机启动时创建。此内存区域的唯一目的就

是存放对象实例,几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描

述是:所有的对象实例以及数组都要在堆上分配

Java堆是垃圾收集器管理的主要区域,因此很多时候也被称做“GC--Garbage

Collected Heap

根据Java虚拟机规范的规定,Java堆可以处于物理上不连续的内存空间中,只要逻辑上

是连续的即可,就像我们的磁盘空间一样。在实现时,既可以实现成固定大小的,也可以是

可扩展的,不过当前主流的虚拟机都是按照可扩展来实现的(通过-Xmx和-Xms控制)。如

果在堆中没有内存完成实例分配,并且堆也无法再扩展时,将会抛出OutOfMemoryError异

常。

**对象的创建**

虚拟机遇到一条new指令时,首先将去检查这个指令的参数是否能在常量池中定位到一

个类的符号引用,并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没

有,那必须先执行相应的类加载过程

假设Java堆中内存是绝对规整的,所有用过的内

存都放在一边,空闲的内存放在另一边,中间放着一个指针作为分界点的指示器,那所分配

内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离,这种分配方式称

为“指针碰撞”(Bump the Pointer)。如果Java堆中的内存并不是规整的,已使用的内存和空

闲的内存相互交错,那就没有办法简单地进行指针碰撞了,虚拟机就必须维护一个列表,记

录上哪些内存块是可用的,在分配的时候从列表中找到一块足够大的空间划分给对象实例,

并更新列表上的记录,这种分配方式称为“空闲列表”(Free

List)

除如何划分可用空间之外,还有另外一个需要考虑的问题是对象创建在虚拟机中是非常

频繁的行为,即使是仅仅修改一个指针所指向的位置,在并发情况下也并不是线程安全的,

可能出现正在给对象A分配内存,指针还没来得及修改,对象B又同时使用了原来的指针来

分配内存的情况。解决这个问题有两种方案,一种是对分配内存空间的动作进行同步处理

——实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性;另一种是把内存分

配的动作按照线程划分在不同的空间之中进行,即每个线程在Java堆中预先分配一小块内

存,称为本地线程分配缓冲(Thread Local Allocation Buffer,TLAB)。哪个线程要分配内

存,就在哪个线程的TLAB上分配,只有TLAB用完并分配新的TLAB时,才需要同步锁定。

虚拟机是否使用TLAB,可以通过-XX:+/-UseTLAB参数来设定。

**对象的访问定位**

建立对象是为了使用对象,我们的Java程序需要通过栈上的reference数据来操作堆上的

具体对象。



内存泄露：应用程序使用资源后没有及时释放，导致内存持有了不需要的资源

内存一处：内存已经不能满足正常使用了，堆栈已经达到了系统设置的最大值

**回收永久代（JAVA8被元数据区取代）**

在堆中,尤其是在新生代中,常规应用进行一次垃圾收集一般可以

回收70%~95%的空间,而永久代的垃圾收集效率远低于此。

要判定一个类是否是“无用的类”的条件则

相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”:

该类所有的实例都已经被回收,也就是Java堆中不存在该类的任何实例。

加载该类的ClassLoader已经被回收。

该类对应的java.lang.Class对象没有在任何地方被引用,无法在任何地方通过反射访问该

类的方法。

虚拟机可以对满足上述3个条件的无用类进行回收,这里说的仅仅是“可以”,而并不是

和对象一样,不使用了就必然会回收。是否对类进行回收,HotSpot虚拟机提供了-Xnoclassgc

参数进行控制,还可以使用-verbose:class以及-XX:+TraceClassLoading、-XX:

+TraceClassUnLoading查看类加载和卸载信息,其中-verbose:class和-XX:

+TraceClassLoading可以在Product版的虚拟机中使用,-XX:+TraceClassUnLoading参数需要

FastDebug版的虚拟机支持。

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁

自定义ClassLoader的场景都需要虚拟机具备类卸载的功能,以保证永久代不会溢出。

**自定义类加载器作用**

**1. 加密，防止反编译**

**2. 可以从外界源，如数据库，网络加载类**

**3. 依情况动态创建**

**大对象直接进入老年代**

**所谓的大对象是指,需要大量连续内存空间的Java对象,最典型的大对象就是那种很长**

**的字符串以及数组(笔者列出的例子中的byte[]数组就是典型的大对象)。大对象对虚拟机**

**的内存分配来说就是一个坏消息(替Java虚拟机抱怨一句,比遇到一个大对象更加坏的消息**

**就是遇到一群“朝生夕灭”的“短命大对象”,写程序的时候应当避免),经常出现大对象容易**

**导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。**

控制Full GC频率的关键是看应用中绝大多数对象能否符合“朝生夕灭”的原则，即大多数

对象的生存时间不应太长，尤其是不能有成批量的、长生存时间的大对象产生，这样才能保

障老年代空间的稳定。

在大多数网站形式的应用里，主要对象的生存周期都应该是请求级或者页面级的，会话

级和全局级的长生命对象相对很少。只要代码写得合理，应当都能实现在超大堆中正常使用

而没有Full GC，这样的话，使用超大堆内存时，网站响应速度才会比较有保证。

从实践经验的角度出发，除了Java堆和永久代之外，我们注意到下面这些区域还会占用

较多的内存，这里所有的内存总和受到操作系统进程最大内存的限制。

Direct Memory：可通过-XX：MaxDirectMemorySize调整大小，内存不足时抛出

OutOfMemoryError或者OutOfMemoryError：Direct buffer memory。

线程堆栈：可通过-Xss调整大小，内存不足时抛出StackOverflowError（纵向无法分配，

即无法分配新的栈帧）或者OutOfMemoryError：unable to create new native thread（横向无法

分配，即无法建立新的线程）。

Socket缓存区：每个Socket连接都Receive和Send两个缓存区，分别占大约37KB和25KB内

存，连接多的话这块内存占用也比较可观。如果无法分配，则可能会抛出IOException：Too

many open files异常

这是一个来自网络的案例：一个数字校园应用系统，运行在一台4个CPU的Solaris 10操

作系统上，中间件为GlassFish服务器。系统在做大并发压力测试的时候，发现请求响应时间

比较慢，通过操作系统的mpstat工具发现CPU使用率很高，并且系统占用绝大多数的CPU资

源的程序并不是应用系统本身。这是个不正常的现象，通常情况下用户应用的CPU占用率应

该占主要地位，才能说明系统是正常工作的。

通过Solaris 10的Dtrace脚本可以查看当前情况下哪些系统调用花费了最多的CPU资

源，Dtrace运行后发现最消耗CPU资源的竟然是“fork”系统调用。众所周知，“fork”系统调用

是Linux用来产生新进程的，在Java虚拟机中，用户编写的Java代码最多只有线程的概念，不

应当有进程的产生。

这是个非常异常的现象。通过本系统的开发人员，最终找到了答案：每个用户请求的处

理都需要执行一个外部shell脚本来获得系统的一些信息。执行这个shell脚本是通过Java的

Runtime.getRuntime（）.exec（）方法来调用的。这种调用方式可以达到目的，但是它在Java

虚拟机中是非常消耗资源的操作，即使外部命令本身能很快执行完毕，频繁调用时创建进程

的开销也非常可观。Java虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一

样环境变量的进程，再用这个新的进程去执行外部命令，最后再退出这个进程。如果频繁执

行这个操作，系统的消耗会很大，不仅是CPU，内存负担也很重。

用户根据建议去掉这个Shell脚本执行的语句，改为使用Java的API去获取这些信息后，

系统很快恢复了正常。

JVM内存划分:

堆区：（Eden->Survivor）新生代->(Tenured Gen)老年代

非堆区：代码缓存区、永久代、jvm stack(虚拟机栈)、本地方法栈

新生代GC频繁发生，很明显是由于虚拟机分配给新生代的空间太小而导致的，Eden区

加上一个Survivor区还不到35MB。因此很有必要使用-Xmn参数调整新生代的大小。

jvm内存调节宗旨：

1. 频繁gc则可能是相应的内存区域分配太小
2. 在gc时频繁扩容，则考虑将最大和最小值设为相等，这样启动后直接分配最大值，不需要运行时扩容了
3. GC后内存无变化，则肯能是程序手动调用System.gc(),加入参数-XX：

+DisableExplicitGC屏蔽掉System.gc（）