深入java虚拟机

群组地址:

http://hllvm.group.iteye.com/

整理和发表jvm相关文章的专栏，本专栏分为JVM基础，JVM调优和JVM调优实战

目 录

1.1 JVM调优总结（一）-- 一些概念

发表时间: 2010-11-08

数据类型

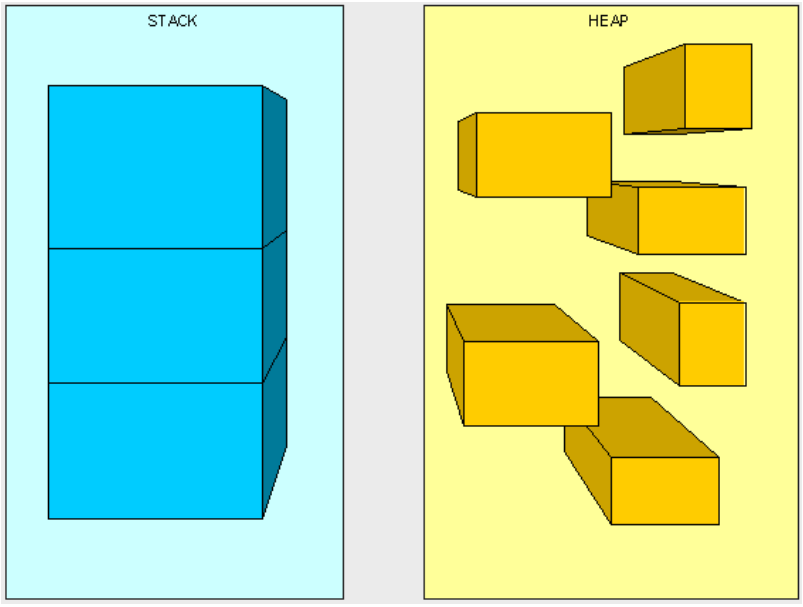
Java虚拟机中，数据类型可以分为两类：**基本类型**和**引用类型**。基本类型的变量保存原始值，即：他代表的值就是数值本身；而引用类型的变量保存引用值。“引用值”代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

基本类型包括：byte,short,int,long,char,float,double,Boolean,returnAddress

引用类型包括：**类类型**，**接口类型**和**数组**。

堆与栈

堆和栈是程序运行的关键，很有必要把他们的关系说清楚。



栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

第一，从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。

第二，堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。

第三，栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。

第四，面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

在Java中，Main函数就是栈的起始点，也是程序的起始点。

程序要运行总是有一个起点的。同C语言一样，java中的Main就是那个起点。无论什么java程序，找到main就找到了程序执行的入口：）

堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

Java中的参数传递时传值呢？还是传引用？

要说明这个问题，先要明确两点：

1. **不要试图与C进行类比，Java中没有指针的概念**

2. **程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。**不会直接传对象本身。

明确以上两点后。Java在方法调用传递参数时，因为没有指针，所以**它都是进行传值调用**（这点可以参考C的传值调用）。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过-Xss来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点。

1.2 JVM调优总结（二）-一些概念

发表时间: 2010-11-08

Java对象的大小

基本数据的类型的大小是固定的，这里就不多说了。对于非基本类型的Java对象，其大小就值得商榷。

在Java中，**一个空Object对象的大小是8byte**，这个大小只是保存堆中一个没有任何属性的对象的大小。看下面语句：

```
Object ob = new Object();
```

这样在程序中完成了一个Java对象的生命，但是它所占的空间为：**4byte+8byte**。4byte是上面部分所说的Java栈中保存引用的所需要的空间。而那8byte则是Java堆中对象的信息。因为所有的Java非基本类型的对象都需要默认继承Object对象，因此不论什么样的Java对象，其大小都必须是大于8byte。

有了Object对象的大小，我们就可以计算其他对象的大小了。

```
Class NewObject {
    int count;
    boolean flag;
    Object ob;
}
```

其大小为：空对象大小(8byte)+int大小(4byte)+Boolean大小(1byte)+空Object引用的大小(4byte)=17byte。但是因为Java在对对象内存分配时都是以8的整数倍来分，因此大于17byte的最接近8的整数倍的是24，因此此对象的大小为24byte。

这里需要注意一下**基本类型的包装类型的大小**。因为这种包装类型已经成为对象了，因此需要把他们作为对象来看待。包装类型的大小至少是12byte（声明一个空Object至少需要的空间），而且12byte没有包含任何有效信息，同时，因为Java对象大小是8的整数倍，因此**一个基本类型包装类的大小至少是16byte**。这个内存占用是很恐怖的，它是使用基本类型的N倍（N>2），有些类型的内存占用更是夸张（随便想下就知道了）。因此，可能的话应尽量少使用包装类。在JDK5.0以后，因为加入了自动类型装换，因此，Java虚拟机会在存储方面进行相应的优化。

引用类型

对象引用类型分为**强引用、软引用、弱引用和虚引用。**

强引用:就是我们一般声明对象是时虚拟机生成的引用，强引用环境下，垃圾回收时需要严格判断当前对象是否被强引用，如果被强引用，则不会被垃圾回收

软引用:软引用一般被做为缓存来使用。与强引用的区别是，软引用在垃圾回收时，虚拟机会根据当前系统的剩余内存来决定是否对软引用进行回收。如果剩余内存比较紧张，则虚拟机会回收软引用所引用的空间；如果剩余内存相对富裕，则不会进行回收。换句话说，虚拟机在发生OutOfMemory时，肯定是没有软引用存在的。

弱引用:弱引用与软引用类似，都是作为缓存来使用。但与软引用不同，弱引用在进行垃圾回收时，是一定会被回收掉的，因此其生命周期只存在于一个垃圾回收周期内。

强引用不用说，我们系统一般在使用时都是用的强引用。而“软引用”和“弱引用”比较少见。他们一般被作为缓存使用，而且一般是在内存大小比较受限的情况下做为缓存。因为如果内存足够大的话，可以直接使用强引用作为缓存即可，同时可控性更高。因而，他们常见的是被使用在桌面应用系统的缓存。

1.3 JVM调优总结（三）-基本垃圾回收算法

发表时间: 2010-11-08

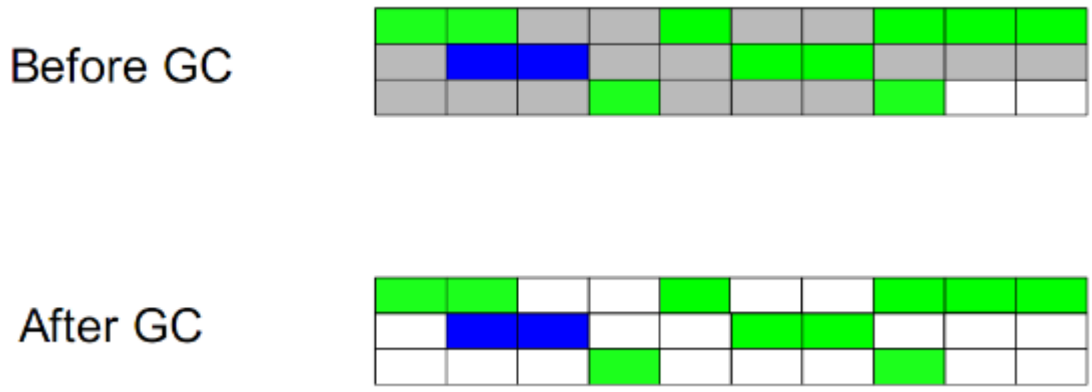
可以从不同的的角度去划分垃圾回收算法：

按照基本回收策略分

引用计数（Reference Counting）:

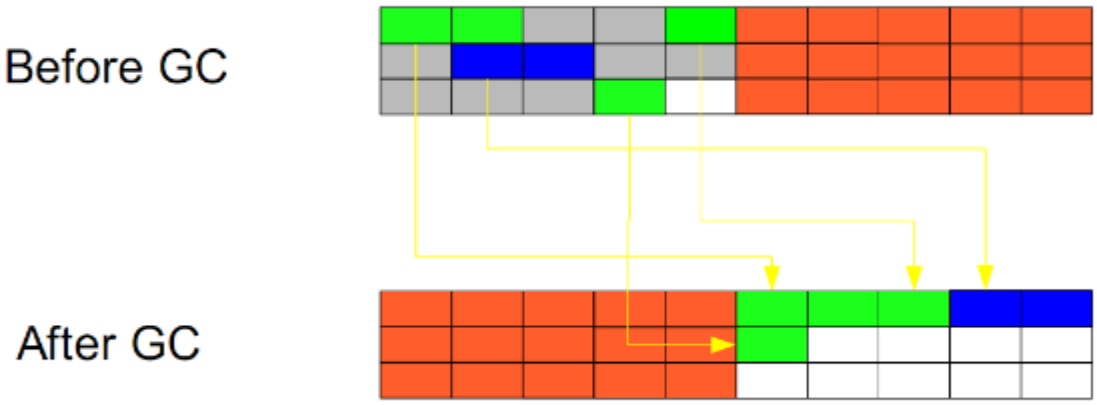
比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

标记-清除（Mark-Sweep）:



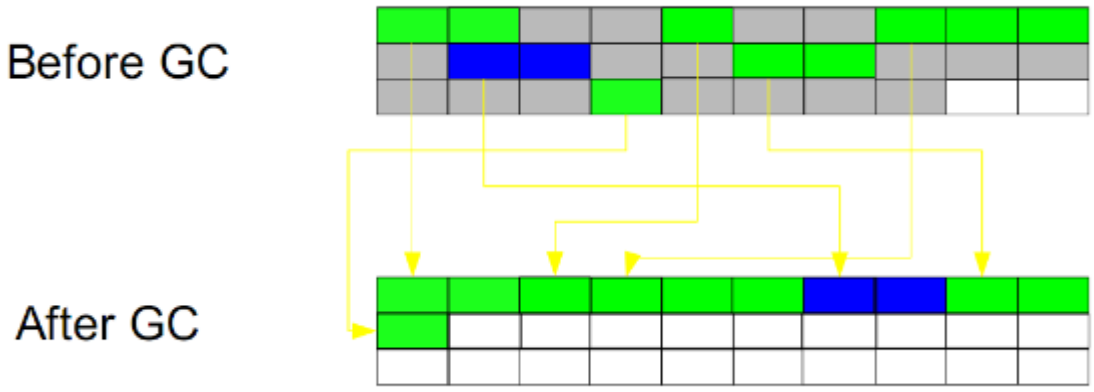
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）:



此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

标记-整理（Mark-Compact）：



此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

按分区对待的方式分

增量收集（Incremental Collecting）:实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

分代收集（Generational Collecting）:基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

按系统线程分

串行收集:串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。

并行收集:并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上CPU数目越多，越能体现出并行收集器的优势。

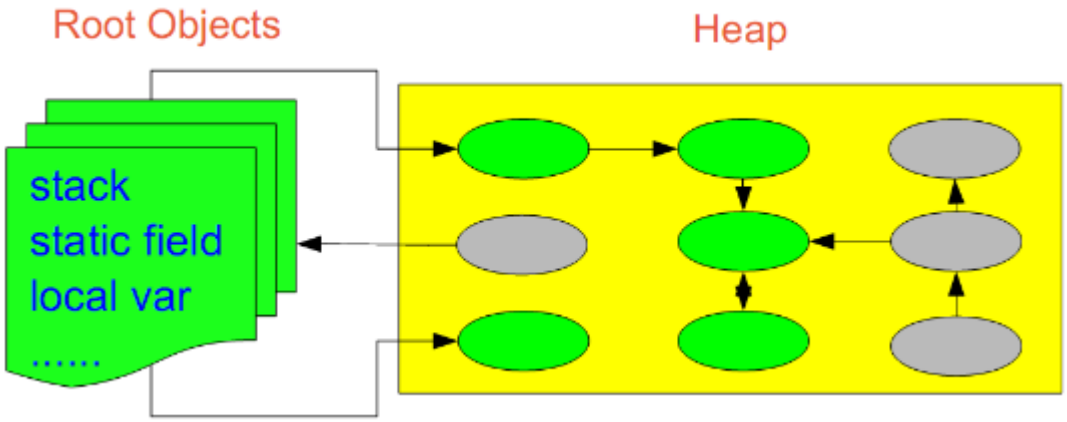
并发收集:相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

1.4 JVM调优总结（四）-垃圾回收面临的问题

发表时间: 2010-11-08

如何区分垃圾

上面说到的“引用计数”法，通过统计控制生成对象和删除对象时的引用数来判断。垃圾回收程序收集计数为0的对象即可。但是这种方法无法解决循环引用。所以，后来实现的垃圾判断算法中，都是从程序运行的根节点出发，遍历整个对象引用，查找存活的对象。那么在这种方式的实现中，**垃圾回收从哪儿开始的呢**？即，从哪儿开始查找哪些对象是正在被当前系统使用的。上面分析的堆和栈的区别，其中栈是真正进行程序执行地方，所以要获取哪些对象正在被使用，则需要从Java栈开始。同时，一个栈是与一个线程对应的，因此，如果有多个线程的话，则必须对这些线程对应的所有的栈进行检查。



同时，除了栈外，还有系统运行时的寄存器等，也是存储程序运行数据的。这样，以栈或寄存器中的引用为起点，我们可以找到堆中的对象，又从这些对象找到对堆中其他对象的引用，这种引用逐步扩展，最终以null引用或者基本类型结束，这样就形成了一颗以Java栈中引用所对应的对象为根节点的一颗对象树，如果栈中有多个引用，则最终会形成多颗对象树。在这些对象树上的对象，都是当前系统运行所需要的对象，不能被垃圾回收。而其他剩余对象，则可以视为无法被引用到的对象，可以被当做垃圾进行回收。

因此，**垃圾回收的起点是一些根对象（java栈, 静态变量, 寄存器...）**。而最简单的Java栈就是Java程序执行的main函数。这种回收方式，也是上面提到的“标记-清除”的回收方式

如何处理碎片

由于不同Java对象存活时间是不一定的，因此，在程序运行一段时间以后，如果不进行内存整理，就会出现零散的内存碎片。碎片最直接的问题就是会导致无法分配大块的内存空间，以及程序运行效率降低。所以，在上面提到的基本垃圾回收算法中，“复制”方式和“标记-整理”方式，都可以解决碎片的问题。

如何解决同时存在的对象创建和对象回收问题

垃圾回收线程是回收内存的，而程序运行线程则是消耗（或分配）内存的，**一个回收内存，一个分配内存**，从这点看，两者是矛盾的。因此，在现有的垃圾回收方式中，要进行垃圾回收前，一般都需要暂停整个应用（即：暂停内存的分配），然后进行垃圾回收，回收完成后再继续应用。这种实现方式是最直接，而且最有效的解决二者矛盾的方式。

但是**这种方式有一个很明显的弊端，就是当堆空间持续增大时，垃圾回收的时间也将会相应的持续增大，对应应用暂停的时间也会相应的增大。**一些对相应时间要求很高的应用，比如最大暂停时间要求是几百毫秒，那么当堆空间大于几个G时，就很有可能超过这个限制，在这种情况下，垃圾回收将会成为系统运行的一个瓶颈。为解决这种矛盾，有了**并发垃圾回收算法**，使用这种算法，垃圾回收线程与程序运行线程同时运行。在这种方式下，解决了暂停的问题，但是因为需要在新生成对象的同时又要回收对象，算法复杂性会大大增加，系统的处理能力也会相应降低，同时，“碎片”问题将会比较难解决。

1.5 JVM调优总结（五）-分代垃圾回收详述1

发表时间: 2010-11-08

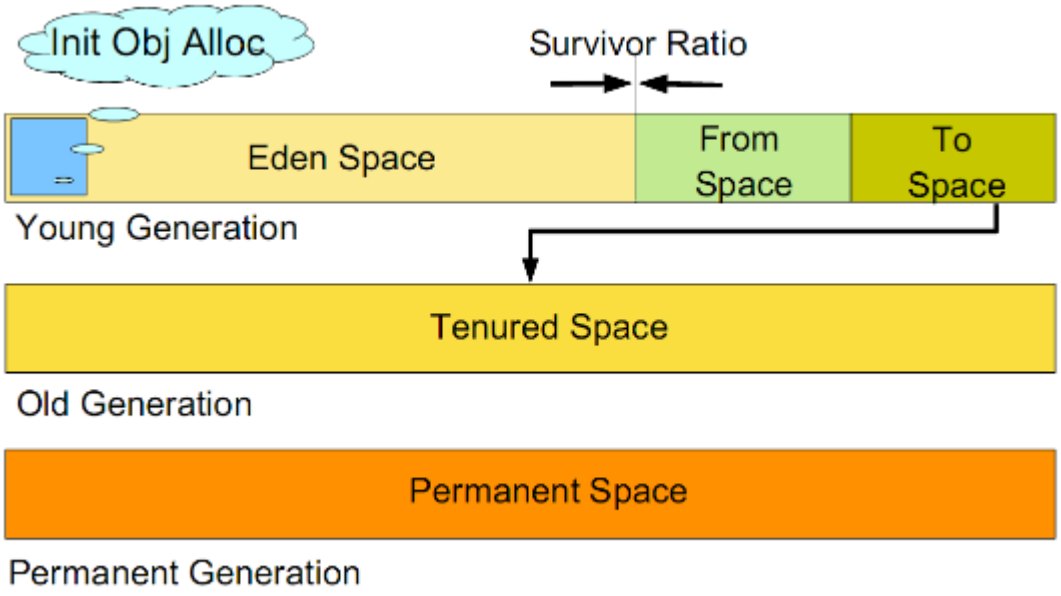
为什么要分代

分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的。**因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

在Java程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如Http请求中的Session对象、线程、Socket连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

如何分代



如图所示：

虚拟机中的共划分为三个代：**年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）**。其中持久代主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

年轻代:

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。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Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor去也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来 对象，和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor去过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代:

在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

持久代:

用于存放静态文件，如今Java类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些class，例如Hibernate等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=<N>进行设置。

什么情况下触发垃圾回收

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：**Scavenge GC**和**Full GC**。

Scavenge GC

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

Full GC

对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个对进行回收，所以比Scavenge GC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节。有如下原因可能导致Full GC：

· 年老代（Tenured）被写满

· 持久代（Perm）被写满

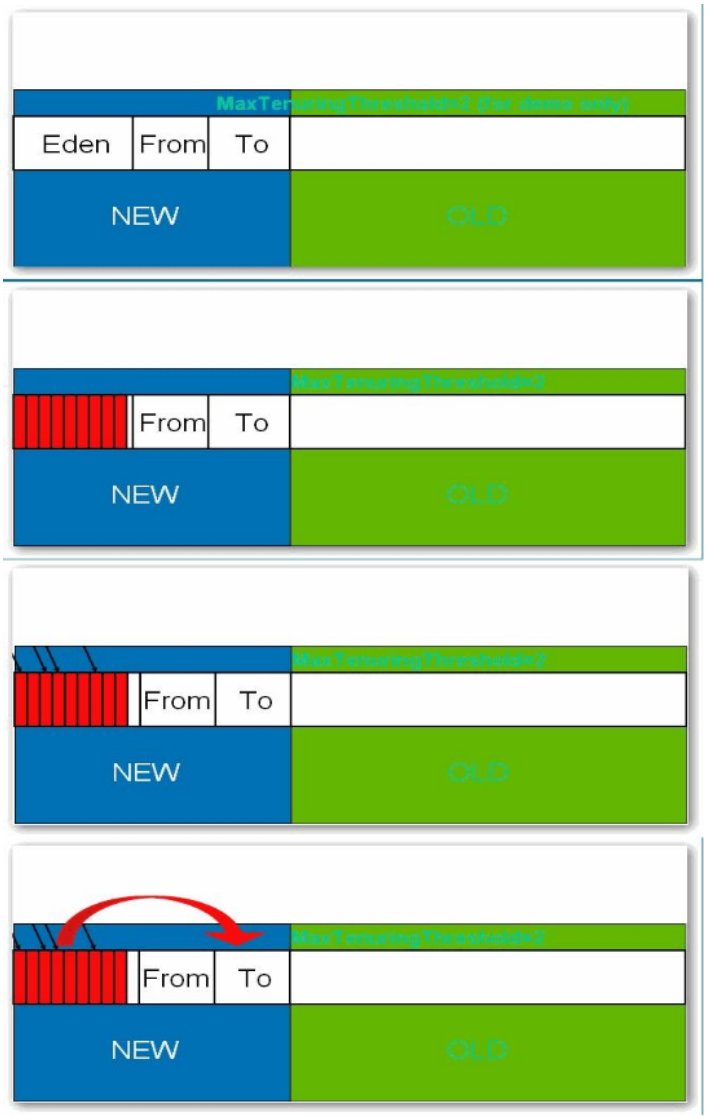
· System.gc()被显示调用

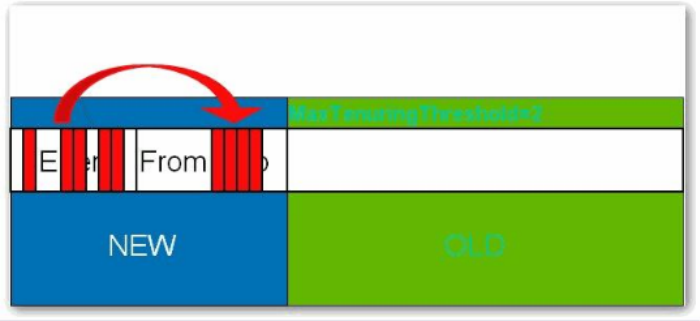
·上一次GC之后Heap的各域分配策略动态变化

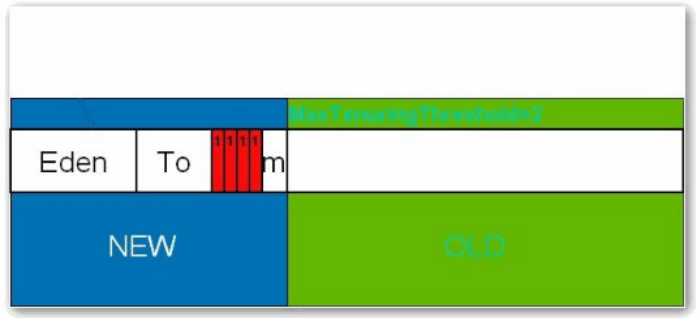
1.6 JVM调优总结（六）-分代垃圾回收详述2

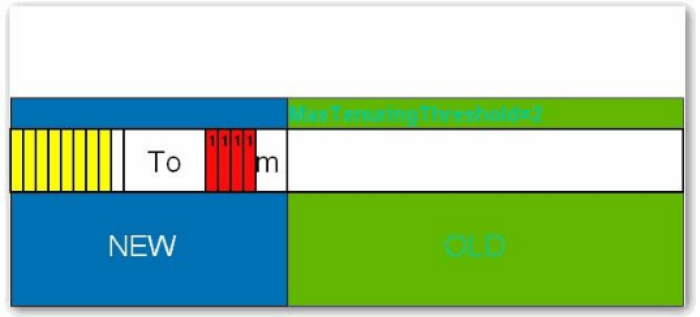
发表时间: 2010-11-08

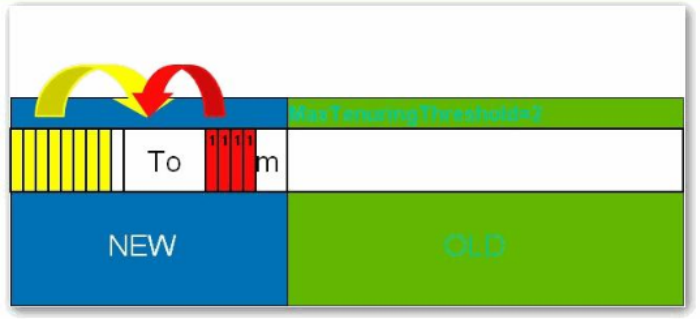
分代垃圾回收流程示意

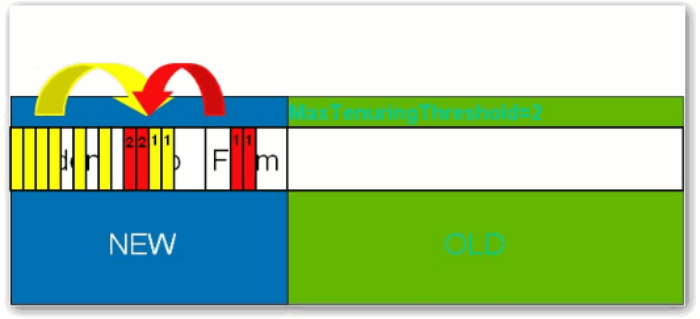


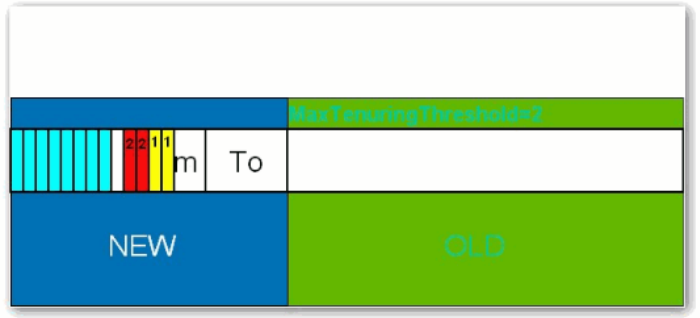

From
NEW
OLD

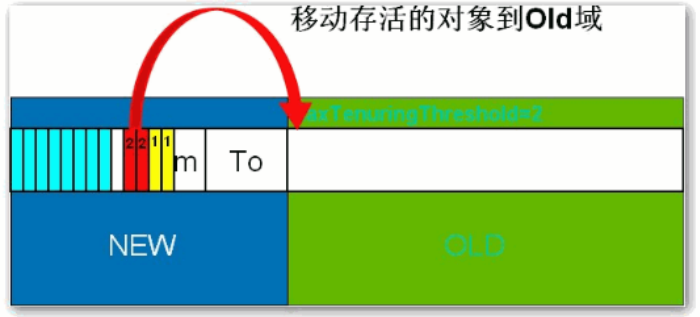

Eden
To
NEW
OLD

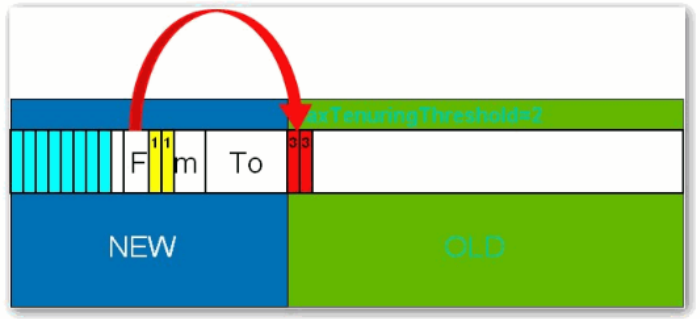

To
NEW
OLD

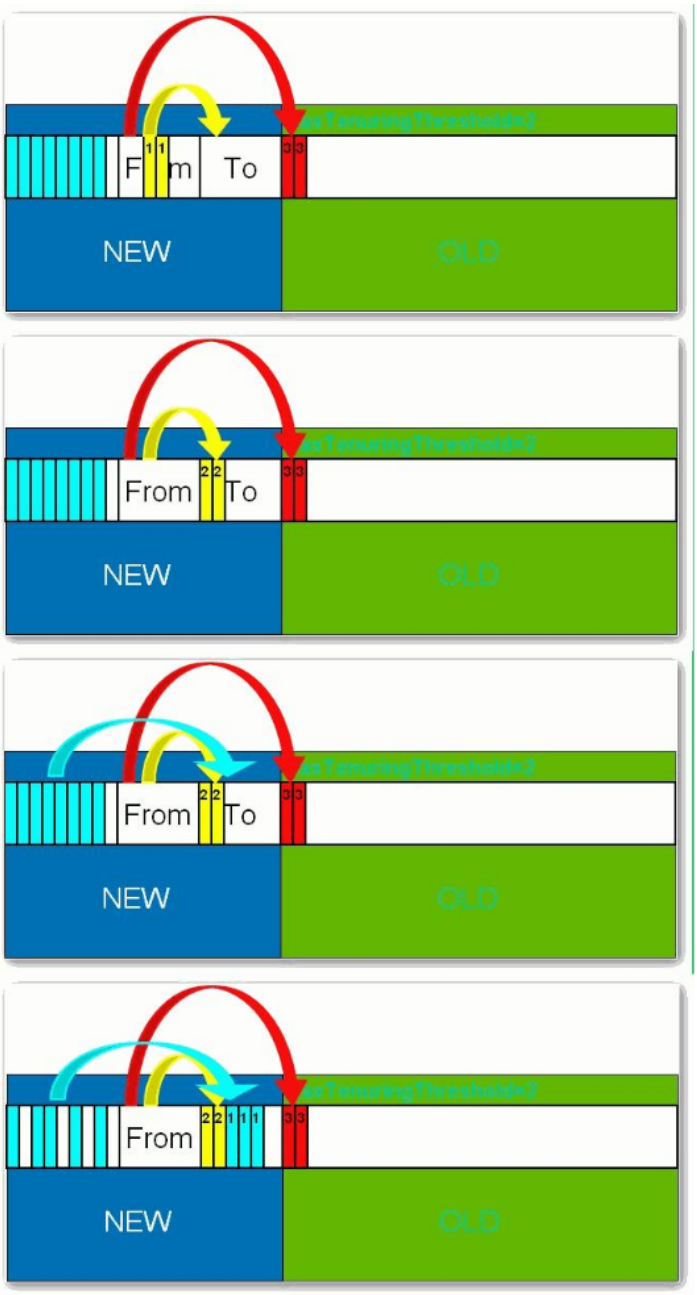

To
NEW
OLD


MaxTenuringThreshold=2
2 2 1 1
F
1 1
m
NEW
OLD


MaxTenuringThreshold=2
2 2 1 1
m
To
NEW
OLD

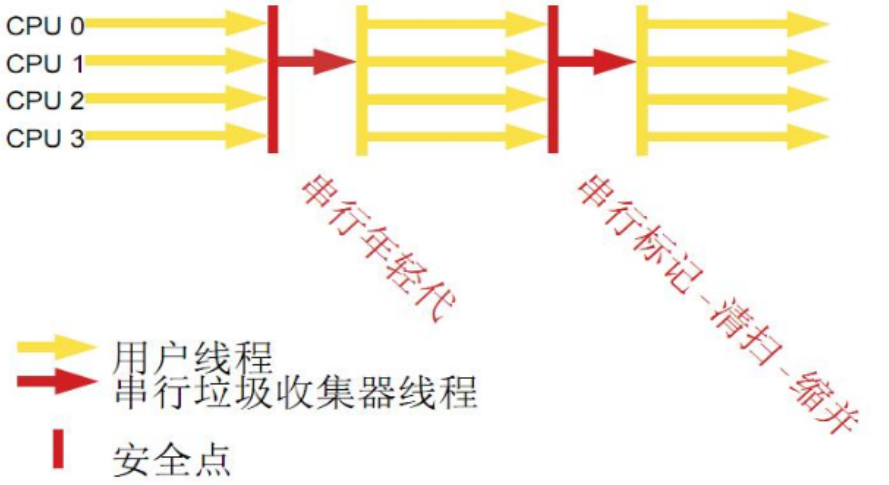

移动存活的对象到Old域
MaxTenuringThreshold=2
2 2 1 1
m
To
NEW
OLD


MaxTenuringThreshold=2
F
1 1
m
To
3 3
NEW
OLD



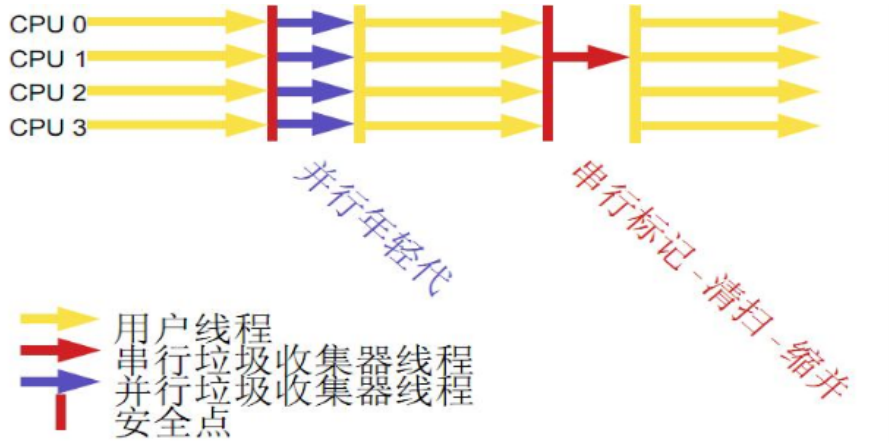
选择合适的垃圾收集算法

串行收集器



用单线程处理所有垃圾回收工作，因为无需多线程交互，所以效率比较高。但是，也无法使用多处理器的优势，所以此收集器适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。可以使用-XX:+UseSerialGC打开。

并行收集器



对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。一般在多线程多处理器机器上使用。使用-XX:+UseParallelGC.打开。并行收集器在J2SE5.0第六6更新上引入，在Java SE6.0中进行了增强--可以对年老

代进行并行收集。如果年老代不使用并发收集的话，默认是使用单线程进行垃圾回收，因此会制约扩展能力。使用-XX:+UseParallelOldGC打开。

使用-XX:ParallelGCThreads=<N>设置并行垃圾回收的线程数。此值可以设置与机器处理器数量相等。

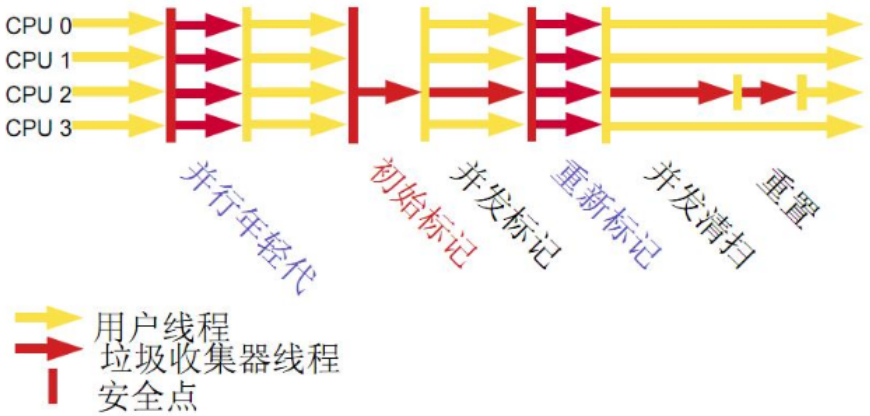
此收集器可以进行如下配置：

最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=<N>指定。<N>为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。

吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=<N>来设定，公式为1/（1+N）。例如，-XX:GCTimeRatio=19时，表示5%的时间用于垃圾回收。默认情况为99，即1%的时间用于垃圾回收。

并发收集器

可以保证大部分工作都并发进行（应用不停止），垃圾回收只暂停很少的时间，此收集器适合对响应时间要求比较高的中、大规模应用。使用-XX:+UseConcMarkSweepGC打开。



并发收集器主要减少年老代的暂停时间，他在应用不停止的情况下使用独立的垃圾回收线程，跟踪可达对象。在每个年老代垃圾回收周期中，在收集初期并发收集器 会对整个应用进行简短的暂停，在收集中还会再暂停一次。第二次暂停会比第一次稍长，在此过程中多个线程同时进行垃圾回收工作。

并发收集器使用处理器换来短暂的停顿时间。在一个N个处理器的系统上，并发收集部分使用K/N个可用处理器进行回收，一般情况下1<=K<=N/4。

在只有一个处理器的主机上使用并发收集器，设置为incremental mode模式也可获得较短的停顿时间。

浮动垃圾：由于在应用运行的同时进行垃圾回收，所以有些垃圾可能在垃圾回收进行完成时产生，这样就造成了“Floating Garbage”，这些垃圾需要在下次垃圾回收周期时才能回收掉。所以，并发收集器一般需要20%的预留空间用于这些浮动垃圾。

Concurrent Mode Failure：并发收集器在应用运行时进行收集，所以需要保证堆在垃圾回收的这段时间有足够的空间供程序使用，否则，垃圾回收还未完成，堆空间先满了。这种情况下将会发生“并发模式失败”，此时整个应用将会暂停，进行垃圾回收。

启动并发收集器：因为并发收集在应用运行时进行收集，所以必须保证收集完成之前有足够的内存空间供程序使用，否则会出现“Concurrent Mode Failure”。通过设置-XX:CMSInitiatingOccupancyFraction=<N>指定还有多少剩余堆时开始执行并发收集

小结

串行处理器：

--适用情况：数据量比较小（100M左右）；单处理器下并且对响应时间无要求的应用。
--缺点：只能用于小型应用

并行处理器：

--适用情况：“对吞吐量有高要求”，多CPU、对应用响应时间无要求的中、大型应用。举例：后台处理、科学计算。
--缺点：垃圾收集过程中应用响应时间可能加长

并发处理器：

--适用情况：“对响应时间有高要求”，多CPU、对应用响应时间有较高要求的中、大型应用。举例：Web服务器/应用服务器、电信交换、集成开发环境。

1.7 JVM调优总结（七）-典型配置举例1

发表时间: 2010-11-08

以下配置主要针对分代垃圾回收算法而言。

堆大小设置

年轻代的设置很关键

JVM中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制。在Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

典型设置：

java **-Xmx3550m -Xms3550m -Xmn2g** –Xss128k

-Xmx3550m：设置JVM最大可用内存为3550M。

-Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g：设置年轻代大小为2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

java -Xmx3550m -Xms3550m -Xss128k **-XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0**

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m:设置持久代大小为16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

回收器选择

JVM给了三种选择：**串行收集器、并行收集器、并发收集器**，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k **-XX:+UseParallelGC -XX:ParallelGCThreads=20**

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。**此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。**

-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 **-XX:+UseParallelOldGC**

-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC **-XX:MaxGCPauseMillis=100**

-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

n java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100 **-XX:+UseAdaptiveSizePolicy**

-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 **-XX:+UseConcMarkSweepGC -XX:+UseParNewGC**

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC: 设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC **-XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection**

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

辅助信息

JVM提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

-XX:+PrintGC：输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails：输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

-XX:+PrintGCTimeStamps -XX:+PrintGC：PrintGCTimeStamps可与上面两个混合使用
输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

-XX:+PrintGCApplicationConcurrentTime：打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用。输出形式：Application time: 0.5291524 seconds

-XX:+PrintGCApplicationStoppedTime：打印垃圾回收期间程序暂停的时间。可与上面混合使用。输出形式：Total time for which application threads were stopped: 0.0468229 seconds

-XX:PrintHeapAtGC: 打印GC前后的详细堆栈信息。输出形式：

34.702: [GC {Heap before gc invocations=7:

def new generation total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000)

eden space 49152K, 99% used [0x1ebd0000, 0x21bce430, 0x21bd0000)

from space 6144K, 55% used [0x221d0000, 0x22527e10, 0x227d0000)

to space 6144K, 0% used [0x21bd0000, 0x21bd0000, 0x221d0000)

tenured generation total 69632K, used 2696K [0x227d0000, 0x26bd0000, 0x26bd0000)

the space 69632K, 3% used [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)

compacting perm gen total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)

  the space 8192K, 35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)

ro space 8192K, 66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)

rw space 12288K, 46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)

34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs] 55264K->6615K(124928K)Heap after gc invocations=8:

def new generation total 55296K, used 3433K [0x1ebd0000, 0x227d0000, 0x227d0000)

eden space 49152K, 0% used [0x1ebd0000, 0x1ebd0000, 0x21bd0000)

```
  from space 6144K,  55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)
  to   space 6144K,   0% used [0x221d0000, 0x221d0000, 0x227d0000)
tenured generation   total 69632K, used 3182K [0x227d0000, 0x26bd0000, 0x26bd0000)
the space 69632K,   4% used [0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)
compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)
  the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)
  ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)
  rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)
}
, 0.0757599 secs]
```

-Xloggc:filename:与上面几个配合使用，把相关日志信息记录到文件以便分析。

1.8 JVM调优总结（八）-典型配置举例2

发表时间: 2010-11-08

常见配置汇总

堆设置

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小

收集器设置

-XX:+UseSerialGC:设置串行收集器

-XX:+UseParallelGC:设置并行收集器

-XX:+UseParalledlOldGC:设置并行年老代收集器

-XX:+UseConcMarkSweepGC:设置并发收集器

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

并行收集器设置

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间

-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

并发收集器设置

-XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

调优总结

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

1. 并发垃圾收集信息

2. 持久代并发收集次数

3. 传统GC信息

4. 花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用

一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

1. **-XX:+UseCMSCompactAtFullCollection：**使用并发收集器时，开启对年老代的压缩。

2. **-XX:CMSFullGCsBeforeCompaction=0：**上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

1.9 JVM调优总结（九）-新一代的垃圾回收算法

发表时间: 2010-11-08

垃圾回收的瓶颈

传统分代垃圾回收方式，已经在一定程度上把垃圾回收给应用带来的负担降到了最小，把应用的吞吐量推到了一个极限。但是他无法解决的一个问题，就是Full GC所带来的应用暂停。在一些对实时性要求很高的应用场景下，GC暂停所带来的请求堆积和请求失败是无法接受的。这类应用可能要求请求的返回时间在几百甚至几十毫秒以内，如果分代垃圾回收方式要达到这个指标，只能把最大堆的设置限制在一个相对较小范围内，但是这样有限制了应用本身的处理能力，同样也是不可接收的。

分代垃圾回收方式确实也考虑了实时性要求而提供了并发回收器，支持最大暂停时间的设置，但是受限于分代垃圾回收的内存划分模型，其效果也不是很理想。

为了达到实时性的要求（其实Java语言最初的设计也是在嵌入式系统上的），一种新垃圾回收方式呼之欲出，它既支持短的暂停时间，又支持大的内存空间分配。可以很好的解决传统分代方式带来的问题。

增量收集的演进

增量收集的方式在理论上可以解决传统分代方式带来的问题。增量收集把对堆空间划分成一系列内存块，使用时，先使用其中一部分（不会全部用完），垃圾收集时把之前用掉的部分中的存活对象再放到后面没有用的空间中，这样可以实现一直边使用边收集的效果，避免了传统分代方式整个使用完了再暂停的回收的情况。

当然，传统分代收集方式也提供了并发收集，但是他有一个很致命的地方，就是把整个堆做为一个内存块，这样一方面会造成碎片（无法压缩），另一方面他的每次收集都是对整个堆的收集，无法进行选择，在暂停时间的控制上还是很弱。而增量方式，通过内存空间的分块，恰恰可以解决上面问题。

Garbage Firest（G1）

这部分的内容主要参考这里，这篇文章算是对G1算法论文的解读。我也没加什么东西了。

目标

从设计目标看G1完全是为了大型应用而准备的。

支持很大的堆

高吞吐量

--支持多CPU和垃圾回收线程

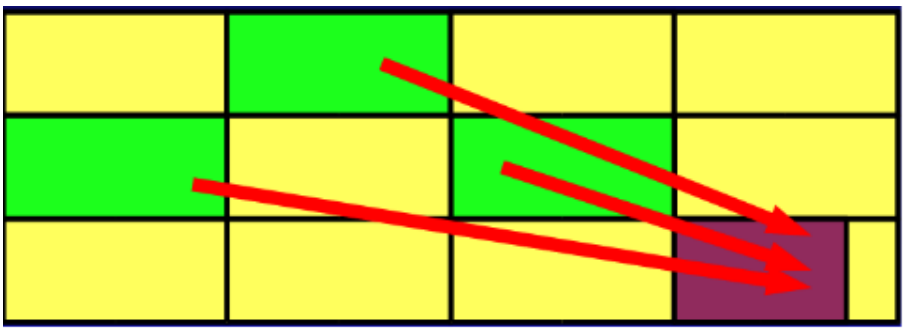
--在主线程暂停的情况下，使用并行收集

--在主线程运行的情况下，使用并发收集

实时目标：可配置在N毫秒内最多只占用M毫秒的时间进行垃圾回收

当然G1要达到实时性的要求，相对传统的分代回收算法，在性能上会有一些损失。

算法详解



G1可谓博采众家之长，力求到达一种完美。他吸取了增量收集优点，把整个堆划分为一个一个等大小的区域（region）。内存的回收和划分都以region为单位；同时，他也吸取了CMS的特点，把这个垃圾回收过程分为几个阶段，分散一个垃圾回收过程；而且，G1也认同分代垃圾回收的思想，认为不同对象的生命周期不同，可以采取不同收集方式，因此，它也支持分代的垃圾回收。为了达到对回收时间的可预计性，G1在扫描了region以后，对其中的活跃对象的大小进行排序，首先会收集那些活跃对象小的region，以便快速回收空间（要复制的活跃对象少了），因为活跃对象小，里面可以认为多数都是垃圾，所以这种方式被称为Garbage First（G1）的垃圾回收算法，即：垃圾优先的回收。

回收步骤：

初始标记（Initial Marking）

G1对于每个region都保存了两个标识用的bitmap，一个为previous marking bitmap，一个为next marking bitmap，bitmap中包含了一个bit的地址信息来指向对象的起始点。

开始Initial Marking之前，首先并发的清空next marking bitmap，然后停止所有应用线程，并扫描标识出每个region中root可直接访问到的对象，将region中top的值放入next top at mark start（TAMS）中，之后恢复所有应用线程。

触发这个步骤执行的条件为：

G1定义了一个JVM Heap大小的百分比的阀值，称为h，另外还有一个H，H的值为(1-h)*Heap Size，目前这个h的值是固定的，后续G1也许会将其改为动态的，根据jvm的运行情况来动态的调整，在分代方式下，G1还定义了一个u以及soft limit，soft limit的值为H-u*Heap Size，当Heap中使用的内存超过了soft limit值时，就会在一次clean up执行完毕后在应用允许的GC暂停时间范围内尽快的执行此步骤；

在pure方式下，G1将marking与clean up组成一个环，以便clean up能充分的使用marking的信息，当clean up开始回收时，首先回收能够带来最多内存空间的regions，当经过多次的clean up，回收到没多少空间的regions时，G1重新初始化一个新的marking与clean up构成的环。

并发标记（Concurrent Marking）

按照之前Initial Marking扫描到的对象进行遍历，以识别这些对象的下层对象的活跃状态，对于在此期间应用线程并发修改的对象的以来关系则记录到remembered set logs中，新创建的对象则放入比top值更高的地址区间中，这些新创建的对象默认状态即为活跃的，同时修改top值。

最终标记暂停（Final Marking Pause）

当应用线程的remembered set logs未满时，是不会放入filled RS buffers中的，在这样的情况下，这些remebered set logs中记录的card的修改就会被更新了，因此需要这一步，这一步要做的就是把应用线程中存在的remembered set logs的内容进行处理，并相应的修改remembered sets，这一步需要暂停应用，并行的运行。

存活对象计算及清除（Live Data Counting and Cleanup）

值得注意的是，在G1中，并不是说Final Marking Pause执行完了，就肯定执行Cleanup这步的，由于这步需要暂停应用，G1为了能够达到准实时的要求，需要根据用户指定的最大的GC造成的暂停时间来合理的规划什么时候执行Cleanup，另外还有几种情况也是会触发这个步骤的执行的：

G1采用的是复制方法来进行收集，必须保证每次的”to space”的空间都是够的，因此G1采取的策略是当已经使用的内存空间达到了H时，就执行Cleanup这个步骤；

对于full-young和partially-young的分代模式的G1而言，则还有情况会触发Cleanup的执行，full-young模式下，G1根据应用可接受的暂停时间、回收young regions需要消耗的时间来估算出一个yound regions的数量值，当JVM中分配对象的young regions的数量达到此值时，Cleanup就会执行；partially-young模式下，则会尽量频繁的在应用可接受的暂停时间范围内执行Cleanup，并最大限度的去执行non-young regions的Cleanup。

展望

以后JVM的调优或许跟多需要针对G1算法进行调优了。

1.10 JVM调优总结（十）-调优方法

发表时间: 2010-11-08

JVM调优工具

Jconsole，jProfile，VisualVM

Jconsole : jdk自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。详细说明参考这里

JProfiler：商业软件，需要付费。功能强大。详细说明参考这里

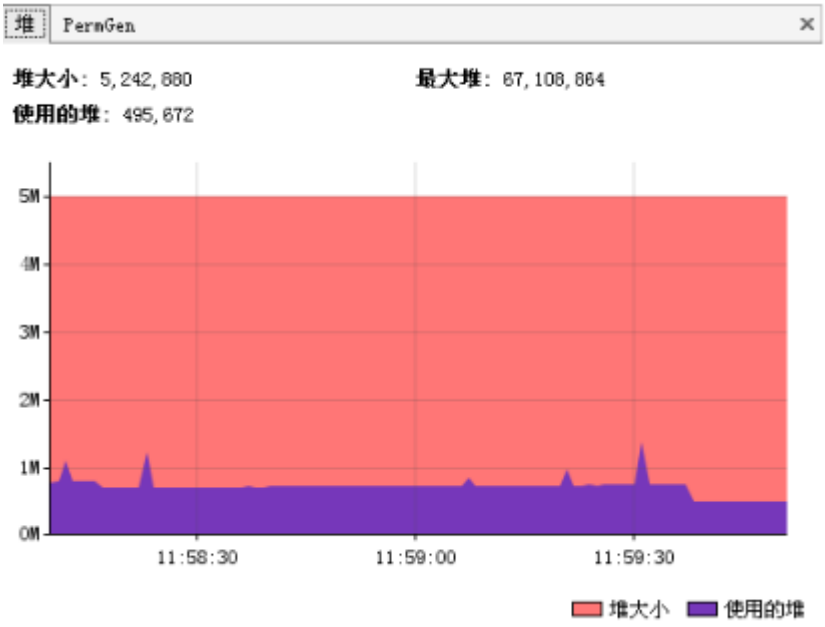
VisualVM：JDK自带，功能强大，与JProfiler类似。推荐。

如何调优

观察内存释放情况、集合类检查、对象树

上面这些调优工具都提供了强大的功能，但是总的来说一般分为以下几类功能

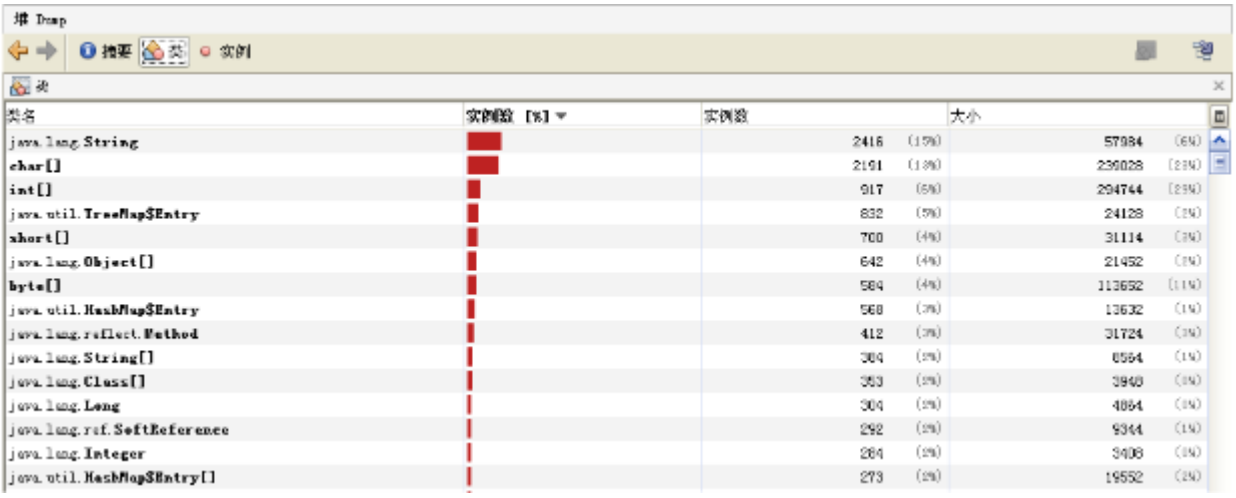
堆信息查看



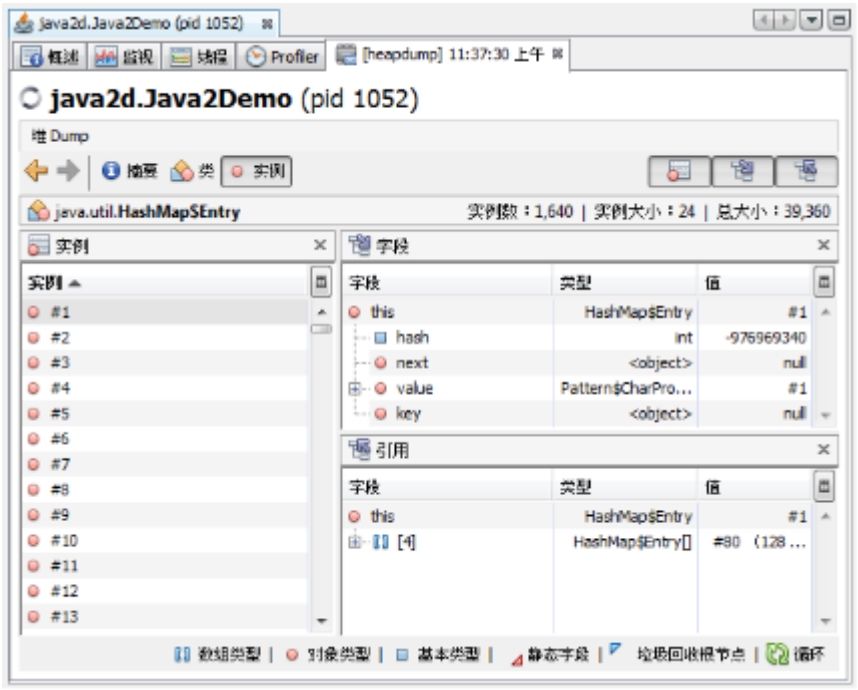
可查看堆空间大小分配（年轻代、年老代、持久代分配）

提供即时的垃圾回收功能

垃圾监控（长时间监控回收情况）



查看堆内类、对象信息查看：数量、类型等



对象引用情况查看

有了堆信息查看方面的功能，我们一般可以顺利解决以下问题：

--年老代年轻代大小划分是否合理

--内存泄漏

--垃圾回收算法设置是否合理

线程监控

线程信息监控：系统线程数量。

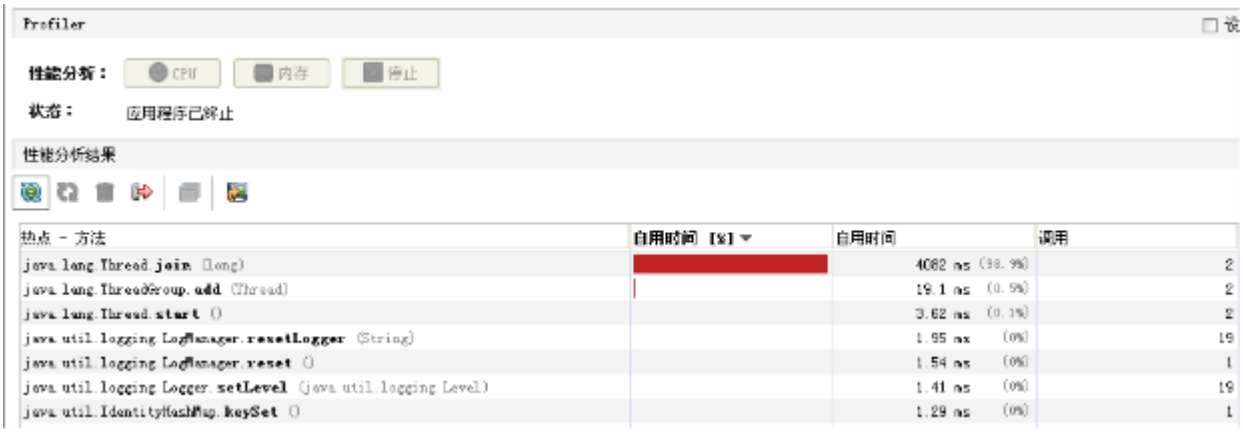
线程状态监控：各个线程都处在什么样的状态下

```
"Finalizer" daemon prio=8 tid=0x02ad0400 nid=0xd18 in Object.wait() [0x02c9f000..0x02c9fc94]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22ed2b78> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        - locked <0x22ed2b78> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

   Locked ownable synchronizers:
        - None

"Reference Handler" daemon prio=10 tid=0x02ac8000 nid=0xb08 in Object.wait() [0x02c4f000..0x02c4fd14]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x22ed2c00> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:485)
        at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
        - locked <0x22ed2c00> (a java.lang.ref.Reference$Lock)

   Locked ownable synchronizers:
        - None

"main" prio=6 tid=0x00306800 nid=0xc00 runnable [0x0093f000..0x0093fa54]
   java.lang.Thread.State: RUNNABLE
        at CPUUserTest.longUseCpu(CPUUserTest.java:9)
        at CPUUserTest.main(CPUUserTest.java:31)

   Locked ownable synchronizers:
        - None
```

Dump线程详细信息：查看线程内部运行情况

死锁检查

热点分析



CPU热点：检查系统哪些方法占用的大量CPU时间

内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

快照

快照是系统运行到某一时刻的一个定格。在我们进行调优的时候，不可能用眼睛去跟踪所有系统变化，依赖快照功能，我们就可以进行系统两个不同运行时刻，对象（或类、线程等）的不同，以便快速找到问题

举例说，我要检查系统进行垃圾回收以后，是否还有该收回的对象被遗漏下来的了。那么，我可以在进行垃圾回收前后，分别进行一次堆情况的快照，然后对比两次快照的对象情况。

内存泄漏检查

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

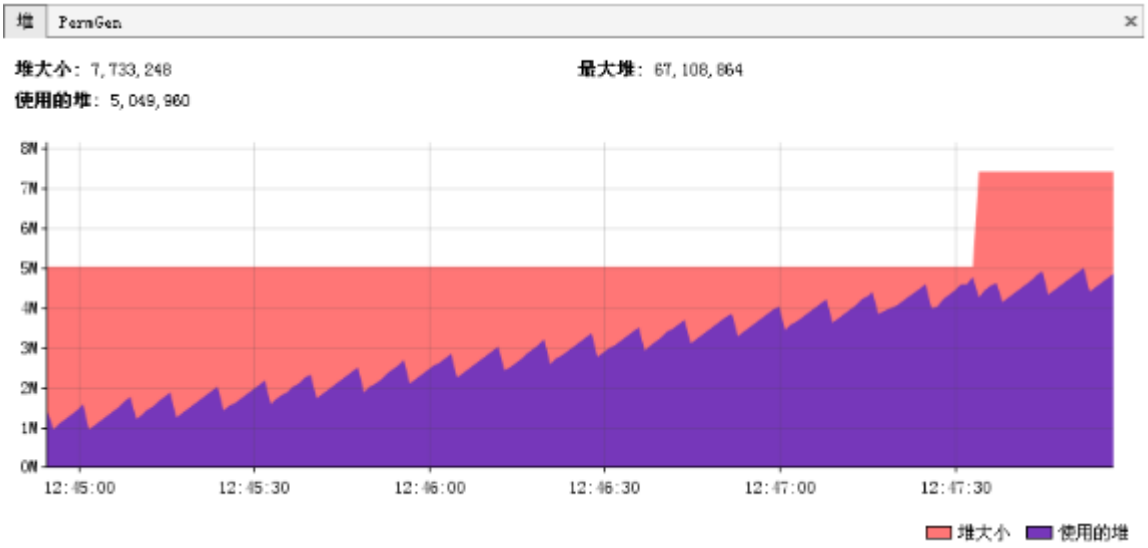
内存泄漏对系统危害比较大，因为他可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

年老代堆空间被占满

异常： java.lang.OutOfMemoryError: Java heap space

说明：



这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）

解决：

这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

持久代被占满

异常：java.lang.OutOfMemoryError: PermGen space

说明：

Perm空间被占满。无法为新的class分配存储空间而引发的异常。这个异常以前是没有的，但是在Java反射大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致Perm区被占满。

更可怕的是，不同的classLoader即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有N个classLoader那么他将会被加载N次。因此，某些情况下，这个问题基本视为无解。当然，存在大量classLoader和大量反射类的情况其实也不多。

解决：

1. -XX:MaxPermSize=16m
2. 换用JDK。比如JRocket。

堆栈溢出

异常：java.lang.StackOverflowError

说明：这个就不多说了，一般就是递归没返回，或者循环调用造成

线程堆栈满

异常：Fatal: Stack size too small

说明：java中一个线程的空间大小是有限制的。JDK5.0以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

解决：增加线程栈大小。-Xss2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

系统内存被占满

异常：java.lang.OutOfMemoryError: unable to create new native thread

说明：

这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。

分配给Java虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。

解决：

1. 重新设计系统减少线程数量。

2. 线程数量不能减少的情况下，通过-Xss减小单个线程大小。以便能生产更多的线程。

1.11 JVM调优总结（十二）-参考资料

发表时间: 2010-11-08

能整理出上面一些东西，也是因为站在巨人的肩上。下面是一些参考资料，供大家学习，大家有更好的，可以继续完善：）

· Java 理论与实践: 垃圾收集简史

· Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning

· Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1

· Hotspot memory management whitepaper

· Java Tuning White Paper

· Diagnosing a Garbage Collection problem

· Java HotSpot VM Options

· A Collection of JVM Options

· Garbage-First Garbage Collection

· Frequently Asked Questions about Garbage Collection in the HotspotTM JavaTM Virtual Machine

· JProfiler试用手记

· Java6 JVM参数选项大全

· 《深入Java虚拟机》。虽然过去了很多年，但这本书依旧是经典。

这里是本系列的最后一篇了，很高兴大家能够喜欢这系列的文章。期间也提了很多问题，其中有些是我之前没有想到的或者考虑欠妥的，感谢提出这些问题的朋友，我也学到的不少东西。

1.12 JVM 几个重要的参数

发表时间: 2010-11-08

<本文提供的设置仅仅是在高压力， 多CPU， 高内存环境下设置>

最近对JVM的参数重新看了下， 把应用的JVM参数调整了下。 几个重要的参数

-server -Xmx3g -Xms3g -XX:MaxPermSize=128m
-XX:NewRatio=1 eden/old 的比例
-XX:SurvivorRatio=8 s/e的比例
-XX:+UseParallelGC
-XX:ParallelGCThreads=8
-XX:+UseParallelOldGC 这个是JAVA 6出现的参数选项
-XX:LargePageSizeInBytes=128m 内存页的大小， 不可设置过大， 会影响Perm的大小。
-XX:+UseFastAccessorMethods 原始类型的快速优化
-XX:+DisableExplicitGC 关闭System.gc()

另外 -Xss 是线程栈的大小， 这个参数需要严格的测试， 一般小的应用， 如果栈不是很深， 应该是128k够用的， 不过，我们的应用调用深度比较大， 还需要做详细的测试。 这个选项对性能的影响比较大。 建议使用256K的大小.

例子:

-server -Xmx3g -Xms3g -Xmn=1g -XX:MaxPermSize=128m -Xss256k -XX:MaxTenuringThreshold=10 -XX:+DisableExplicitGC -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+AggressiveOpts -XX:+UseBiasedLocking

-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCTimeStamps -XX:+PrintGCDetails 打印参数

==========

另外对于大内存设置的要求:

Linux :

Large page support is included in 2.6 kernel. Some vendors have backported the code to their 2.4 based releases. To check if your system can support large page memory, try the following:

```
# cat /proc/meminfo | grep Huge
HugePages_Total: 0
HugePages_Free: 0
Hugepagesize: 2048 kB
#
```

If the output shows the three "Huge" variables then your system can support large page memory, but it needs to be configured. If the command doesn't print out anything, then large page support is not available. To configure the system to use large page memory, one must log in as root, then:

1. **Increase SHMMAX value. It must be larger than the Java heap size. On a system with 4 GB of physical RAM (or less) the following will make all the memory sharable:**

   ```
   # echo 4294967295 > /proc/sys/kernel/shmmax
   ```

2. **Specify the number of large pages. In the following example 3 GB of a 4 GB system are reserved for large pages (assuming a large page size of 2048k, then 3g = 3 x 1024m = 3072m = 3072 * 1024k = 3145728k, and 3145728k / 2048k = 1536):**

   ```
   # echo 1536 > /proc/sys/vm/nr_hugepages
   ```

Note the /proc values will reset after reboot so you may want to set them in an init script (e.g. rc.local or sysctl.conf).

==========

这个设置， 目前观察下来的结果是EDEN区域收集明显速度比较快， 最多几个ms, 但是，对于FGC， 大约需要0。9， 但是发生时间非常的长， 应该是影响不大。 但是对于非web应用的中间件服务, 这个设置很要不得, 可能导致很严重延迟效果. 因此, CMS必然需要被使用, 下面是CMS的重要参数介绍

关于CMS的设置:

使用CMS的前提条件是你有比较的长生命对象， 比如有200M以上的OLD堆占用。 那么这个威力非常猛， 可以极大的提高的FGC的收集能力。 如果你的OLD占用非常的少， 别用了， 绝对降低你性能， 因为CMS收集有2个STOP WORLD的行为。 OLD少的清情况， 根据我的测试， 使用并行收集参数会比较好。

-XX:+UseConcMarkSweepGC 使用CMS内存收集

-XX:+AggressiveHeap 特别说明下：(我感觉对于做java cache应用有帮助)

- **试图是使用大量的物理内存**
- **长时间大内存使用的优化，能检查计算资源（内存， 处理器数量）**
- **至少需要256MB内存**
- **大量的CPU / 内存，（在1.4.1在4CPU的机器上已经显示有提升）**

-XX:+UseParNewGC 允许多线程收集新生代

-XX:+CMSParallelRemarkEnabled 降低标记停顿

-XX+UseCMSCompactAtFullCollection 在FULL GC的时候， 压缩内存， CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。

压力测试下合适结果：

-server -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -XX:MaxTenuringThreshold=31 -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods

由于Jdk1.5.09及之前的bug, 因此， CMS下的GC， 在这些版本的表现是十分糟糕的。 需要另外2个参数来控制cms的启动时间：

-XX:+UseCMSInitiatingOccupancyOnly 仅仅使用手动定义初始化定义开始CMS收集

-XX:CMSInitiatingOccupancyFraction=70 CMS堆上， 使用70%后开始CMS收集。

使用CMS的好处是用尽量少的新生代、，我的经验值是128M - 256M， 然后老生代利用CMS并行收集， 这样能保证系统低延迟的吞吐效率。 实际上cms的收集停顿时间非常的短，2G的内存， 大约20 - 80ms的应用程序停顿时间。

==========系统情况介绍= =

这个例子是测试系统12小时运行后的情况：

$uname -a

2.4.21-51.EL3.customsmp #1 SMP Fri Jun 27 10:44:12 CST 2008 i686 i686 i386 GNU/Linux

$ free -m

	total	used	free	shared	buffers	cached
Mem:	3995	3910	85	0	162	1267
-/+ buffers/cache:		2479	1515			
Swap:	2047	0	2047			

$ jstat -gcutil 23959 1000

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
59.06	0.00	45.77	44.45	56.88	15204	324.023	66	1.668	325.691
0.00	39.66	27.53	44.73	56.88	15205	324.046	66	1.668	325.715
53.42	0.00	22.80	44.73	56.88	15206	324.073	66	1.668	325.741
0.00	44.90	13.73	44.76	56.88	15207	324.094	66	1.668	325.762
51.70	0.00	19.03	44.76	56.88	15208	324.118	66	1.668	325.786
0.00	61.62	19.44	44.98	56.88	15209	324.148	66	1.668	325.816
53.03	0.00	14.00	45.09	56.88	15210	324.172	66	1.668	325.840
53.03	0.00	87.87	45.09	56.88	15210	324.172	66	1.668	325.840
0.00	50.49	72.00	45.22	56.88	15211	324.198	66	1.668	325.866

GC参数配置：

JAVA_OPTS=" -server -XX:+PrintGCApplicationStoppedTime -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -XX:MaxTenuringThreshold=31 -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70 "

实际上我们可以看到并行young gc执行时间是： 324.198s / 15211 = 20ms, cms的执行时间是1.668 / 66 = 25ms. 当然严格来说， 这么算是不对的， 世界停顿的时间要比这是数据稍微大5-10ms. 对我们来说如果不输出日志， 对我们是有参考意义的。

32位系统下， 设置成2G， 非常危险， 除非你确定你的应用占用的native内存很少， 不然可能导致jvm直接crash。

-XX:+AggressiveOpts 加快编译

-XX:+UseBiasedLocking 锁机制的性能改善。

1.13 慢慢琢磨JVM——恭喜JavaEye重新开张

发表时间: 2010-11-26 关键字: jvm

1 JVM简介

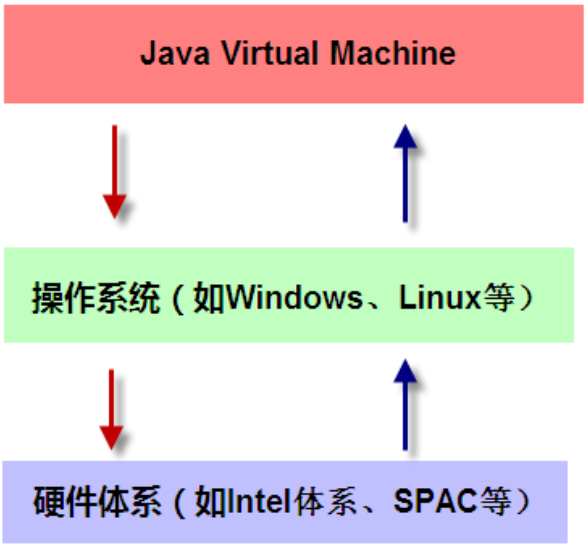
JVM是我们Javaer的最基本功底了，刚开始学Java的时候，一般都是从“Hello World”开始的，然后会写个复杂点class，然后再找一些开源框架，比如Spring，Hibernate等等，再然后就开发企业级的应用，比如网站、企业内部应用、实时交易系统等等，直到某一天突然发现做的系统咋就这么慢呢，而且时不时还来个内存溢出什么的，今天是交易系统报了StackOverflowError，明天是网站系统报了个OutOfMemoryError，这种错误又很难重现，只有分析Javacore和dump文件，运气好点还能分析出个结果，运行遭的点，就直接去庙里烧香吧！每天接客户的电话都是战战兢兢的，生怕再出什么幺蛾子了。我想Java做的久一点的都有这样的经历，那这些问题的最终根结是在哪呢？—— JVM。

JVM全称是Java Virtual Machine，Java虚拟机，也就是在计算机上再虚拟一个计算机，这和我们使用VMWare不一样，那个虚拟的东西你是可以看到的，这个JVM你是看不到的，它存在内存中。我们知道计算机的基本构成是：运算器、控制器、存储器、输入和输出设备，那这个JVM也是有这成套的元素，运算器是当然是交给硬件CPU还处理了，只是为了适应“一次编译，随处运行”的情况，需要做一个翻译动作，于是就用了JVM自己的命令集，这与汇编的命令集有点类似，每一种汇编命令集针对一个系列的CPU，比如8086系列的汇编也是可以用在8088上的，但是就不能跑在8051上，而JVM的命令集则是可以到处运行的，因为JVM做了翻译，根据不同的CPU，翻译成不同的机器语言。

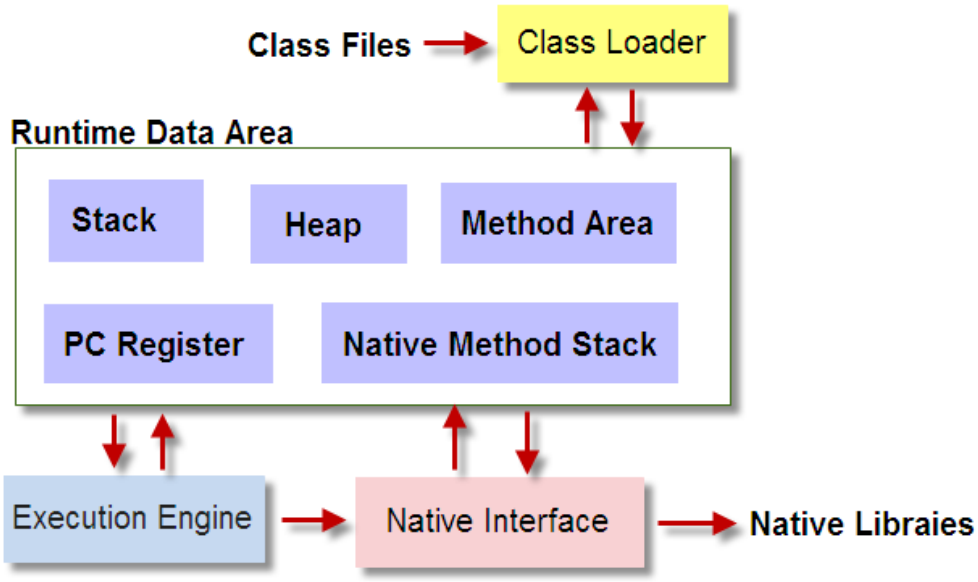
JVM中我们最需要深入理解的就是它的存储部分，存储？硬盘？NO，NO， JVM是一个内存中的虚拟机，那它的存储就是内存了，我们写的所有类、常量、变量、方法都在内存中，这决定着我们程序运行的是否健壮、是否高效，接下来的部分就是重点介绍之。

2 JVM的组成部分

我们先把JVM这个虚拟机画出来，如下图所示：



从这个图中可以看到，JVM是运行在操作系统之上的，它与硬件没有直接的交互。我们再来看下JVM有哪些组成部分，如下图所示：



该图参考了网上广为流传的JVM构成图，大家看这个图，整个JVM分为四部分：

q Class Loader 类加载器

类加载器的作用是加载类文件到内存，比如编写一个HelloWord.java程序，然后通过javac编译成class文件，那怎么才能加载到内存中被执行呢？Class Loader承担的就是这个责任，那不可能随便建立一个.class文件就能

被加载的，Class Loader加载的class文件是有格式要求，在《JVM Specification》中式这样定义Class文件的结构：

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

需要详细了解的话，可以仔细阅读《JVM Specification》的第四章“The class File Format”，这里不再详细说明。

友情提示：Class Loader只管加载，只要符合文件结构就加载，至于说能不能运行，则不是它负责的，那是由Execution Engine负责的。

q Execution Engine 执行引擎

执行引擎也叫做解释器(Interpreter)，负责解释命令，提交操作系统执行。

q Native Interface本地接口

本地接口的作用是融合不同的编程语言为Java所用，它的初衷是融合C/C++程序，Java诞生的时候是C/C++横行的时候，要想立足，必须有一个聪明的、睿智的调用C/C++程序，于是就在内存中专门开辟了一块区域处理标记为native的代码，它的具体做法是Native Method Stack中登记native方法，在Execution Engine执行时加载native libraies。目前该方法使用的是越来越少了，除非是与硬件有关的应用，比如通过Java程序驱动打印机，或者Java系统管理生产设备，在企业级应用中已经比较少见，因为现在的异构领域间的通信很发达，比如可以使用Socket通信，也可以使用Web Service等等，不多做介绍。

q Runtime data area运行数据区

运行数据区是整个JVM的重点。我们所有写的程序都被加载到这里，之后才开始运行，Java生态系统如此的繁荣，得益于该区域的优良自治，下一章节详细介绍之。

整个JVM框架由加载器加载文件，然后执行器在内存中处理数据，需要与异构系统交互是可以通过本地接口进行，瞧，一个完整的系统诞生了！

2 JVM的内存管理

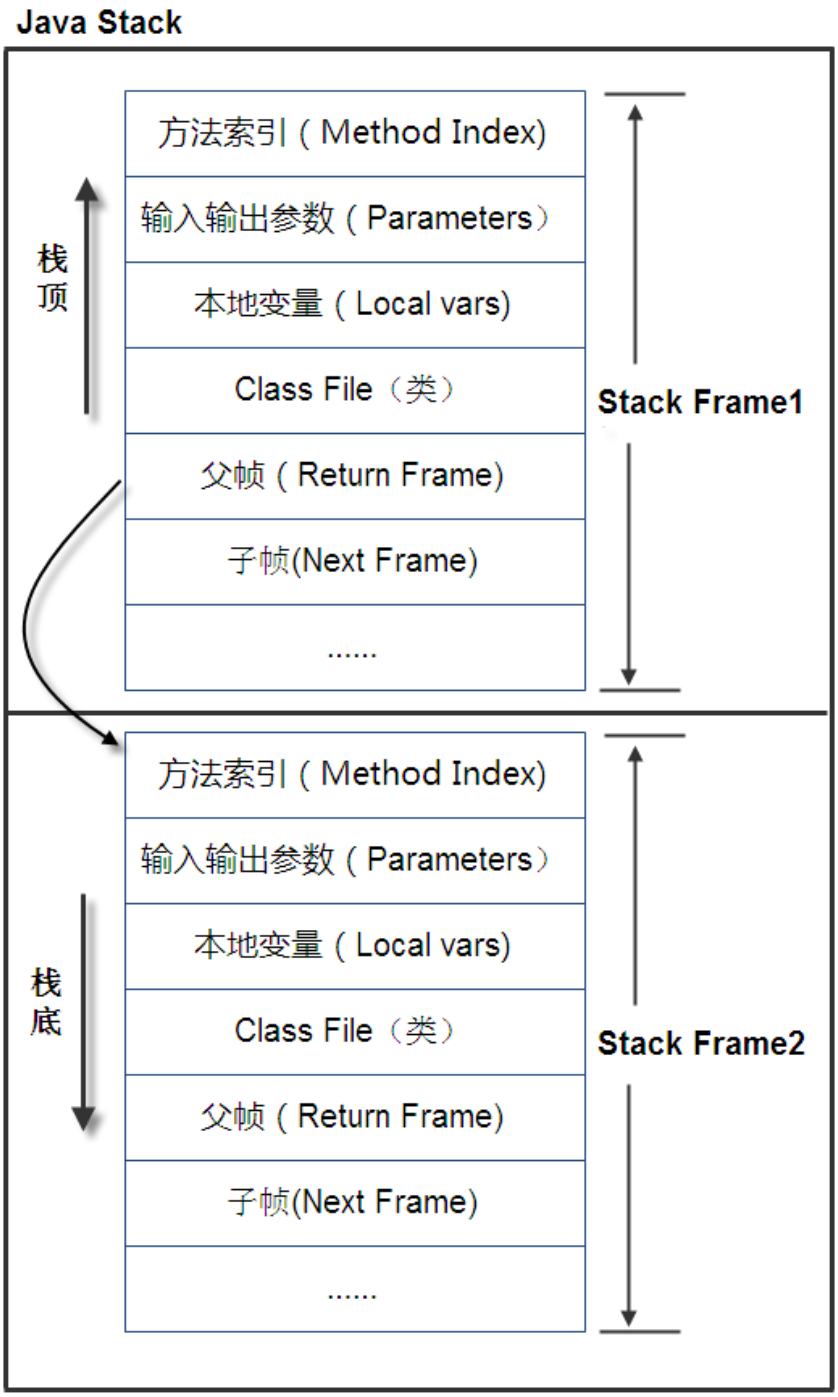
所有的数据和程序都是在运行数据区存放，它包括以下几部分：

q Stack 栈

栈也叫栈内存，是Java程序的运行区，是在线程创建时创建，它的生命期是跟随线程的生命期，线程结束栈内存也就释放，对于栈来说不存在垃圾回收问题，只要线程一结束，该栈就Over。问题出来了：栈中存的是那些数据呢？又什么是格式呢？

栈中的数据都是以栈帧（Stack Frame）的格式存在，栈帧是一个内存区块，是一个数据集，是一个有关方法(Method)和运行期数据的数据集，当一个方法A被调用时就产生了一个栈帧F1，并被压入到栈中，A方法又调用了B方法，于是产生栈帧F2也被压入栈，执行完毕后，先弹出F2栈帧，再弹出F1栈帧，遵循“先进后出”原则。

那栈帧中到底存在着什么数据呢？栈帧中主要保存3类数据：本地变量（Local Variables），包括输入参数和输出参数以及方法内的变量；栈操作（Operand Stack），记录出栈、入栈的操作；栈帧数据（Frame Data），包括类文件、方法等等。光说比较枯燥，我们画个图来理解一下Java栈，如下图所示：



图示在一个栈中有两个栈帧，栈帧2是最先被调用的方法，先入栈，然后方法2又调用了方法1，栈帧1处于栈顶的位置，栈帧2处于栈底，执行完毕后，依次弹出栈帧1和栈帧2，线程结束，栈释放。

q Heap 堆内存

一个JVM实例只存在一个堆类存，堆内存的大小是可以调节的。类加载器读取了类文件后，需要把类、方法、常变量放到堆内存中，以方便执行器执行，堆内存分为三部分：

Permanent Space 永久存储区

永久存储区是一个常驻内存区域，用于存放JDK自身所携带的Class,Interface的元数据，也就是说它存储的是运行环境必须的类信息，被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭JVM才会释放此区域所占用的内存。

Young Generation Space 新生区

新生区是类的诞生、成长、消亡的区域，一个类在这里产生，应用，最后被垃圾回收器收集，结束生命。新生区又分为两部分：伊甸区（Eden space）和幸存者区（Survivor pace），所有的类都是在伊甸区被new出来的。幸存区有两个： 0区（Survivor 0 space）和1区（Survivor 1 space）。当伊甸园的空间用完时，程序又需要创建对象，JVM的垃圾回收器将对伊甸园区进行垃圾回收，将伊甸园区中的不再被其他对象所引用的对象进行销毁。然后将伊甸园中的剩余对象移动到幸存0区。若幸存0区也满了，再对该区进行垃圾回收，然后移动到1区。那如果1区也满了呢？再移动到养老区。

Tenure generation space养老区

养老区用于保存从新生区筛选出来的JAVA对象，一般池对象都在这个区域活跃。 三个区的示意图如下：

q Method Area 方法区

方法区是被所有线程共享，该区域保存所有字段和方法字节码，以及一些特殊方法如构造函数，接口代码也在此定义。

q PC Register 程序计数器

每个线程都有一个程序计数器，就是一个指针，指向方法区中的方法字节码，由执行引擎读取下一条指令。

q Native Method Stack 本地方法栈

3 JVM相关问题

问：堆和栈有什么区别

答：堆是存放对象的，但是对象内的临时变量是存在栈内存中，如例子中的methodVar是在运行期存放到栈中的。

栈是跟随线程的，有线程就有栈，堆是跟随JVM的，有JVM就有堆内存。

问：堆内存中到底存在着什么东西？

答：对象，包括对象变量以及对象方法。

问：类变量和实例变量有什么区别？

答：静态变量是类变量，非静态变量是实例变量，直白的说，有static修饰的变量是静态变量，没有static修饰的变量是实例变量。静态变量存在方法区中，实例变量存在堆内存中。

问：我听说类变量是在JVM启动时就初始化好的，和你这说的不同呀！

答：那你是道听途说，信我的，没错。

问：Java的方法（函数）到底是传值还是传址？

答：都不是，是以传值的方式传递地址，具体的说原生数据类型传递的值，引用类型传递的地址。对于原始数据类型，JVM的处理方法是从Method Area或Heap中拷贝到Stack，然后运行frame中的方法，运行完毕后再把变量指拷贝回去。

问：为什么会产生OutOfMemory产生？

答：一句话：Heap内存中没有足够的可用内存了。这句话要好好理解，不是说Heap没有内存了，是说新申请内存的对象大于Heap空闲内存，比如现在Heap还空闲1M，但是新申请的内存需要1.1M，于是就会报OutOfMemory了，可能以后的对象申请的内存都只要0.9M，于是就只出现一次OutOfMemory，GC也正常了，看起来像偶发事件，就是这么回事。　　但如果此时GC没有回收就会产生挂起情况，系统不响应了。

问：我产生的对象不多呀，为什么还会产生OutOfMemory？

答：你继承层次忒多了，Heap中 产生的对象是先产生 父类，然后才产生子类，明白不？

问：OutOfMemory错误分几种？

答：分两种，分别是“OutOfMemoryError:java heap size”和”OutOfMemoryError: PermGen space”，两种都是内存溢出，heap size是说申请不到新的内存了，这个很常见，检查应用或调整堆内存大小。

“PermGen space”是因为永久存储区满了，这个也很常见，一般在热发布的环境中出现，是因为每次发布应用系统都不重启，久而久之永久存储区中的死对象太多导致新对象无法申请内存，一般重新启动一下即可。

问：为什么会产生StackOverflowError？

答：因为一个线程把Stack内存全部耗尽了，一般是递归函数造成的。

问：一个机器上可以看多个JVM吗？JVM之间可以互访吗？

答：可以多个JVM，只要机器承受得了。JVM之间是不可以互访，你不能在A-JVM中访问B-JVM的Heap内存，这是不可能的。在以前老版本的JVM中，会出现A-JVM Crack后影响到B-JVM，现在版本非常少见。

问：为什么Java要采用垃圾回收机制，而不采用C/C++的显式内存管理？

答：为了简单，内存管理不是每个程序员都能折腾好的。

问：为什么你没有详细介绍垃圾回收机制？

答：垃圾回收机制每个JVM都不同，JVM Specification只是定义了要自动释放内存，也就是说它只定义了垃圾回收的抽象方法，具体怎么实现各个厂商都不同，算法各异，这东西实在没必要深入。

问：JVM中到底哪些区域是共享的？哪些是私有的？

答：Heap和Method Area是共享的，其他都是私有的，

问：什么是JIT，你怎么没说？

答：JIT是指Just In Time，有的文档把JIT作为JVM的一个部件来介绍，有的是作为执行引擎的一部分来介绍，这都能理解。Java刚诞生的时候是一个解释性语言，别嘘，即使编译成了字节码（byte code）也是针对JVM的，它需要再次翻译成原生代码(native code)才能被机器执行，于是效率的担忧就提出来了。Sun为了解决该问题提出了一套新的机制，好，你想编译成原生代码，没问题，我在JVM上提供一个工具，把字节码编译成原生码，下次你来访问的时候直接访问原生码就成了，于是JIT就诞生了，就这么回事。

问：JVM还有哪些部分是你没有提到的？

答：JVM是一个异常复杂的东西，写一本砖头书都不为过，还有几个要说明的：

常量池（constant pool）：按照顺序存放程序中的常量，并且进行索引编号的区域。比如int i =100，这个100就放在常量池中。

安全管理器（Security Manager）：提供Java运行期的安全控制，防止恶意攻击，比如指定读取文件，写入文件权限，网络访问，创建进程等等，Class Loader在Security Manager认证通过后才能加载class文件的。

方法索引表（Methods table），记录的是每个method的地址信息，Stack和Heap中的地址指针其实是指向Methods table地址。

问：为什么不建议在程序中显式的生命System.gc()？

答：因为显式声明是做堆内存全扫描，也就是Full GC，是需要停止所有的活动的（Stop The World Collection），你的应用能承受这个吗？

问：JVM有哪些调整参数？

答：非常多，自己去找，堆内存、栈内存的大小都可以定义，甚至是堆内存的三个部分、新生代的各个比例都能调整。

拷贝过来后格式都变了，就附上PDF文件。

欢迎重拍，别手下留情，我喜欢~~~~

2.1 java线程安全总结

发表时间: 2010-11-12

最近想将java基础的一些东西都整理整理，写下来，这是对知识的总结，也是一种乐趣。已经拟好了提纲，大概分为这几个主题： java线程安全，java垃圾收集，java并发包详细介绍，java profile和jvm性能调优 。慢慢写吧。本人jameswxx原创文章，转载请注明出处，我费了很多心血，多谢了。关于java线程安全，网上有很多资料，我只想从自己的角度总结对这方面的考虑，有时候写东西是很痛苦的，知道一些东西，但想用文字说清楚，却不是那么容易。我认为要认识java线程安全，必须了解两个主要的点：java的内存模型，java的线程同步机制。特别是内存模型，java的线程同步机制很大程度上都是基于内存模型而设定的。后面我还会写java并发包的文章，详细总结如何利用java并发包编写高效安全的多线程并发程序。暂时写得比较仓促，后面会慢慢补充完善。

浅谈java内存模型

不同的平台，内存模型是不一样的，但是jvm的内存模型规范是统一的。其实java的多线程并发问题最终都会反映在java的内存模型上，所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改。总结java的内存模型，要解决两个主要的问题：可见性和有序性。我们都知道计算机有高速缓存的存在，处理器并不是每次处理数据都是取内存的。JVM定义了自己的内存模型，屏蔽了底层平台内存管理细节，对于java开发人员，要清楚在jvm内存模型的基础上，如果解决多线程的可见性和有序性。

那么，何谓可见性？ 多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。Java内存模型（JMM）规定了jvm有主内存，主内存是多个线程共享的。当new一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制的。当线程操作某个对象时，执行顺序如下：

(1) 从主存复制变量到当前工作内存 (read and load)

(2) 执行代码，改变共享变量值 (use and assign)

(3) 用工作内存数据刷新主存相关内容 (store and write)

JVM规范定义了线程对主存的操作指令：read，load，use，assign，store，write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享变量，那么其他线程应该能够看到这个被修改后的值，这就是多线程的可见性问题。

那么，什么是有序性呢 ？线程在引用变量时不能直接从主内存中引用,如果线程工作内存中没有该变量,则会从主内存中拷贝一个副本到工作内存中,这个过程为read-load,完成后线程会引用该副本。当同一线程再度引用该字段时,有可能重新从主存中获取变量副本(read-load-use),也有可能直接引用原来的副本(use),也就是说 read,load,use顺序可以由JVM实现系统决定。

线程不能直接为主存中中字段赋值，它会将值指定给工作内存中的变量副本(assign),完成后这个变量副本会同步到主存储区(store-write)，至于何时同步过去，根据JVM实现系统决定.有该字段,则会从主内存中将该字段赋值到工作内存中,这个过程为read-load,完成后线程会引用该变量副本，当同一线程多次重复对字段赋值时,比如：

```
for(int i=0;i<10;i++)
 a++;
```

线程有可能只对工作内存中的副本进行赋值,只到最后一次赋值后才同步到主存储区，所以assign,store,weite顺序可以由JVM实现系统决定。假设有一个共享变量x，线程a执行x=x+1。从上面的描述中可以知道x=x+1并不是一个原子操作，它的执行过程如下：

1 从主存中读取变量x副本到工作内存

2 给x加1

3 将x加1后的值写回主 存

如果另外一个线程b执行x=x-1，执行过程如下：

1 从主存中读取变量x副本到工作内存

2 给x减1

3 将x减1后的值写回主存

那么显然，最终的x的值是不可靠的。假设x现在为10，线程a加1，线程b减1，从表面上看，似乎最终x还是为10，但是多线程情况下会有这种情况发生：

1：线程a从主存读取x副本到工作内存，工作内存中x值为10

2：线程b从主存读取x副本到工作内存，工作内存中x值为10

3：线程a将工作内存中x加1，工作内存中x值为11

4：线程a将x提交主存中，主存中x为11

5：线程b将工作内存中x值减1，工作内存中x值为9

6：线程b将x提交到中主存中，主存中x为9

同样，x有可能为11，如果x是一个银行账户，线程a存款，线程b扣款，显然这样是有严重问题的，要解决这个问题，必须保证线程a和线程b是有序执行的，并且每个线程执行的加1或减1是一个原子操作。看看下面代码：

```
public class Account {

    private int balance;

    public Account(int balance) {
        this.balance = balance;
    }

    public int getBalance() {
        return balance;
```

```
    }

    public void add(int num) {
        balance = balance + num;
    }

    public void withdraw(int num) {
        balance = balance - num;
    }

    public static void main(String[] args) throws InterruptedException {
        Account account = new Account(1000);
        Thread a = new Thread(new AddThread(account, 20), "add");
        Thread b = new Thread(new WithdrawThread(account, 20), "withdraw");
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println(account.getBalance());
    }

    static class AddThread implements Runnable {
        Account account;
        int     amount;

        public AddThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 200000; i++) {
                account.add(amount);
            }
        }
    }
```

```
    static class WithdrawThread implements Runnable {
        Account account;
        int     amount;

        public WithdrawThread(Account account, int amount) {
            this.account = account;
            this.amount = amount;
        }

        public void run() {
            for (int i = 0; i < 100000; i++) {
                account.withdraw(amount);
            }
        }
    }
}
```

第一次执行结果为10200，第二次执行结果为1060，每次执行的结果都是不确定的，因为线程的执行顺序是不可预见的。这是java同步产生的根源，synchronized关键字保证了多个线程对于同步块是互斥的，synchronized作为一种同步手段，解决java多线程的执行有序性和内存可见性，而volatile关键字之解决多线程的内存可见性问题。后面将会详细介绍。

synchronized关键字

上面说了，java用synchronized关键字做为多线程并发环境的执行有序性的保证手段之一。当一段代码会修改共享变量，这一段代码成为互斥区或临界区，为了保证共享变量的正确性，synchronized标示了临界区。典型的用法如下：

```
synchronized(锁){
     临界区代码
}
```

为了保证银行账户的安全，可以操作账户的方法如下：

```
public synchronized void add(int num) {
     balance = balance + num;
}
public synchronized void withdraw(int num) {
     balance = balance - num;
}
```

刚才不是说了synchronized的用法是这样的吗：

```
synchronized(锁){
临界区代码
}
```

那么对于public synchronized void add(int num)这种情况，意味着什么呢？其实这种情况，锁就是这个方法所在的对象。同理，如果方法是public static synchronized void add(int num)，那么锁就是这个方法所在的class。

理论上，每个对象都可以做为锁，但一个对象做为锁时，应该被多个线程共享，这样才显得有意义，在并发环境下，一个没有共享的对象作为锁是没有意义的。假如有这样的代码：

```
public class ThreadTest{
  public void test(){
     Object lock=new Object();
     synchronized (lock){
        //do something
     }
  }
}
```

lock变量作为一个锁存在根本没有意义，因为它根本不是共享对象，每个线程进来都会执行Object lock=new Object();每个线程都有自己的lock，根本不存在锁竞争。

每个锁对象都有两个队列，一个是就绪队列，一个是阻塞队列，就绪队列存储了将要获得锁的线程，阻塞队列存储了被

阻塞的线程，当一个被线程被唤醒(notify)后，才会进入到就绪队列，等待cpu的调度。当一开始线程a第一次执行account.add方法时，jvm会检查锁对象account的就绪队列是否已经有线程在等待，如果有则表明account的锁已经被占用了，由于是第一次运行，account的就绪队列为空，所以线程a获得了锁，执行account.add方法。如果恰好在这个时候，线程b要执行account.withdraw方法，因为线程a已经获得了锁还没有释放，所以线程b要进入account的就绪队列，等到得到锁后才可以执行。

一个线程执行临界区代码过程如下：

1 获得同步锁

2 清空工作内存

3 从主存拷贝变量副本到工作内存

4 对这些变量计算

5 将变量从工作内存写回到主存

6 释放锁

可见，synchronized既保证了多线程的并发有序性，又保证了多线程的内存可见性。

生产者/消费者模式

生产者/消费者模式其实是一种很经典的线程同步模型，很多时候，并不是光保证多个线程对某共享资源操作的互斥性就够了，往往多个线程之间都是有协作的。

假设有这样一种情况，有一个桌子，桌子上面有一个盘子，盘子里只能放一颗鸡蛋，A专门往盘子里放鸡蛋，如果盘子里有鸡蛋，则一直等到盘子里没鸡蛋，B专门从盘子里拿鸡蛋，如果盘子里没鸡蛋，则等待直到盘子里有鸡蛋。其实盘子就是一个互斥区，每次往盘子放鸡蛋应该都是互斥的，A的等待其实就是主动放弃锁，B等待时还要提醒A放鸡蛋。

如何让线程主动释放锁

很简单，调用锁的wait()方法就好。wait方法是从Object来的，所以任意对象都有这个方法。看这个代码片段：

```
Object lock=new Object();//声明了一个对象作为锁
   synchronized (lock) {
       balance = balance - num;
       //这里放弃了同步锁，好不容易得到，又放弃了
       lock.wait();
}
```

如果一个线程获得了锁lock，进入了同步块，执行lock.wait()，那么这个线程会进入到lock的阻塞队列。如果调用lock.notify()则会通知阻塞队列的某个线程进入就绪队列。

声明一个盘子，只能放一个鸡蛋

```
package com.jameswxx.synctest;
public class Plate{
  List<Object> eggs=new ArrayList<Object>();
  public synchronized  Object getEgg(){
     if(eggs.size()==0){
        try{
            wait();
        }catch(InterruptedException e){
        }
     }

    Object egg=eggs.get(0);
    eggs.clear();//清空盘子
    notify();//唤醒阻塞队列的某线程到就绪队列
    return egg;
}

 public synchronized  void putEgg(Object egg){
    If(eggs.size()>0){
      try{
         wait();
      }catch(InterruptedException e){
      }
    }
    eggs.add(egg);//往盘子里放鸡蛋
    notify();//唤醒阻塞队列的某线程到就绪队列
  }
}
```

声明一个Plate对象为plate，被线程A和线程B共享，A专门放鸡蛋，B专门拿鸡蛋。假设

1 开始，A调用plate.putEgg方法，此时eggs.size()为0，因此顺利将鸡蛋放到盘子，还执行了notify()方法，唤醒锁的阻塞队列的线程，此时阻塞队列还没有线程。

2 又有一个A线程对象调用plate.putEgg方法，此时eggs.size()不为0，调用wait()方法，自己进入了锁对象的阻塞队列。

3 此时，来了一个B线程对象，调用plate.getEgg方法，eggs.size()不为0，顺利的拿到了一个鸡蛋，还执行了notify()方法，

唤醒锁的阻塞队列的线程，此时阻塞队列有一个A线程对象，唤醒后，它进入到就绪队列，就绪队列也就它一个，因此马上得到锁，开始往盘子里放鸡蛋，此时盘子是空的，因此放鸡蛋成功。

4 假设接着来了线程A，就重复2；假设来料线程B，就重复3。

整个过程都保证了放鸡蛋，拿鸡蛋，放鸡蛋，拿鸡蛋。

volatile关键字

volatile是java提供的一种同步手段，只不过它是轻量级的同步，为什么这么说，因为volatile只能保证多线程的内存可见性，不能保证多线程的执行有序性。而最彻底的同步要保证有序性和可见性，例如synchronized。任何被volatile修饰的变量，都不拷贝副本到工作内存，任何修改都及时写在主存。因此对于Valatile修饰的变量的修改，所有线程马上就能看到，但是volatile不能保证对变量的修改是有序的。什么意思呢？假如有这样的代码：

```
public class VolatileTest{
  public volatile int a;
  public void add(int count){
       a=a+count;
  }
}
```

当一个VolatileTest对象被多个线程共享，a的值不一定是正确的，因为a=a+count包含了好几步操作，而此时多个线程的执行是无序的，因为没有任何机制来保证多个线程的执行有序性和原子性。volatile存在的意义是，任何线程对a的修改，都会马上被其他线程读取到，因为直接操作主存，没有线程对工作内存和主存的同步。所以，volatile的使用场景是有限的，在有限的一些情形下可以使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全,必须同时满足下面两个条件:

1)对变量的写操作不依赖于当前值。

2)该变量没有包含在具有其他变量的不变式中

volatile只保证了可见性，所以Volatile适合直接赋值的场景，如

```
public class VolatileTest{
  public volatile int a;
  public void setA(int a){
      this.a=a;
  }
}
```

在没有volatile声明时，多线程环境下，a的最终值不一定是正确的，因为this.a=a;涉及到给a赋值和将a同步回主存的步骤，这个顺序可能被打乱。如果用volatile声明了，读取主存副本到工作内存和同步a到主存的步骤，相当于是一个原子操作。所以简单来说，volatile适合这种场景：一个变量被多个线程共享，线程直接给这个变量赋值。这是一种很简单的同步场景，这时候使用volatile的开销将会非常小。

3.1 通过Java/JMX得到full GC次数？

发表时间: 2010-12-16

今天有个同事问如何能通过JMX获取到某个Java进程的full GC次数：

引用

hi,问个问题，怎们在java中获取到full gc的次数呢？
我现在用jmx的那个得到了gc次数，不过不能细化出来full gc的次数

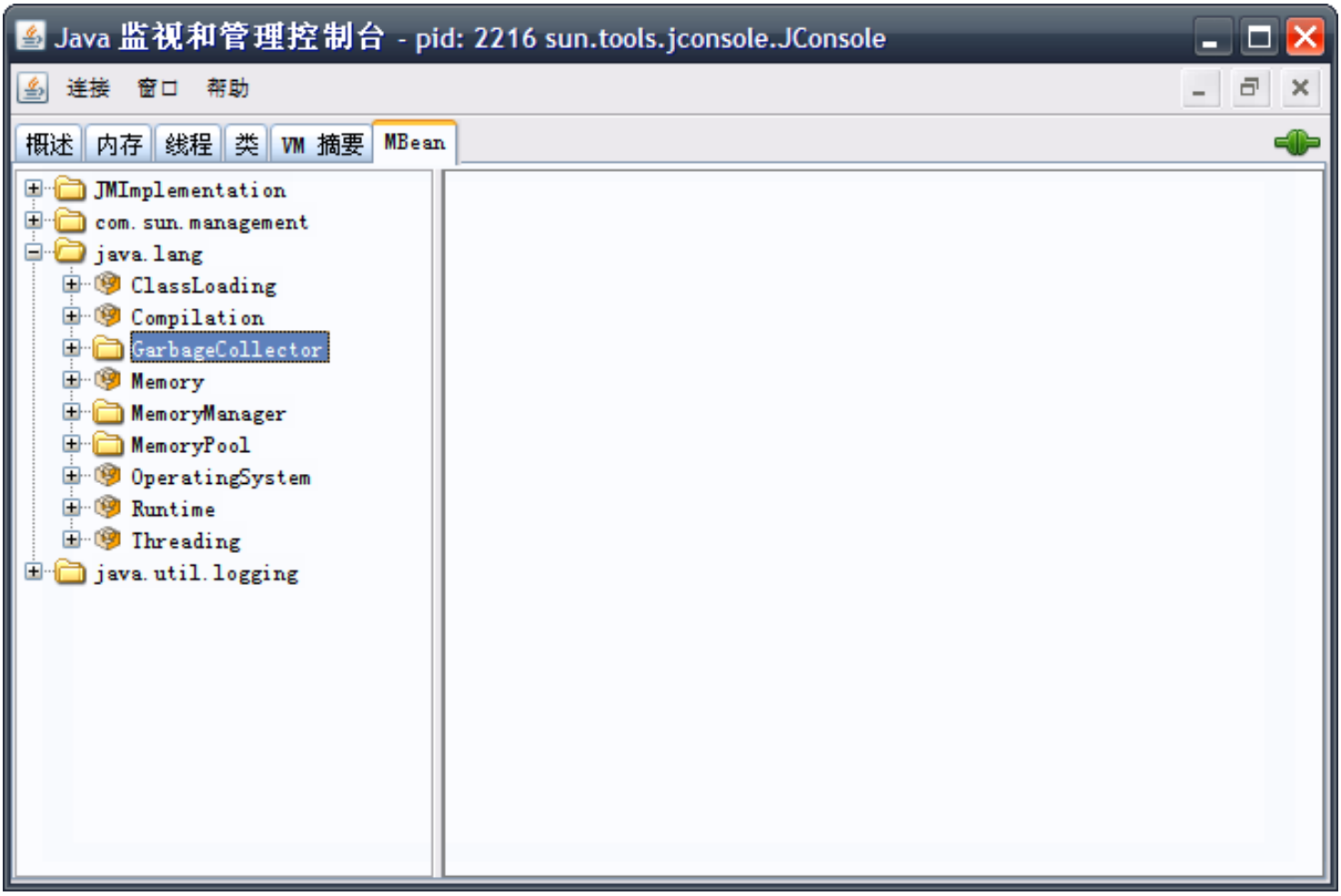
```
for (final GarbageCollectorMXBean garbageCollector
        : ManagementFactory.getGarbageCollectorMXBeans()) {
    gcCounts += garbageCollector.getCollectionCount();
}
```

你比如我现在是这样拿次数的

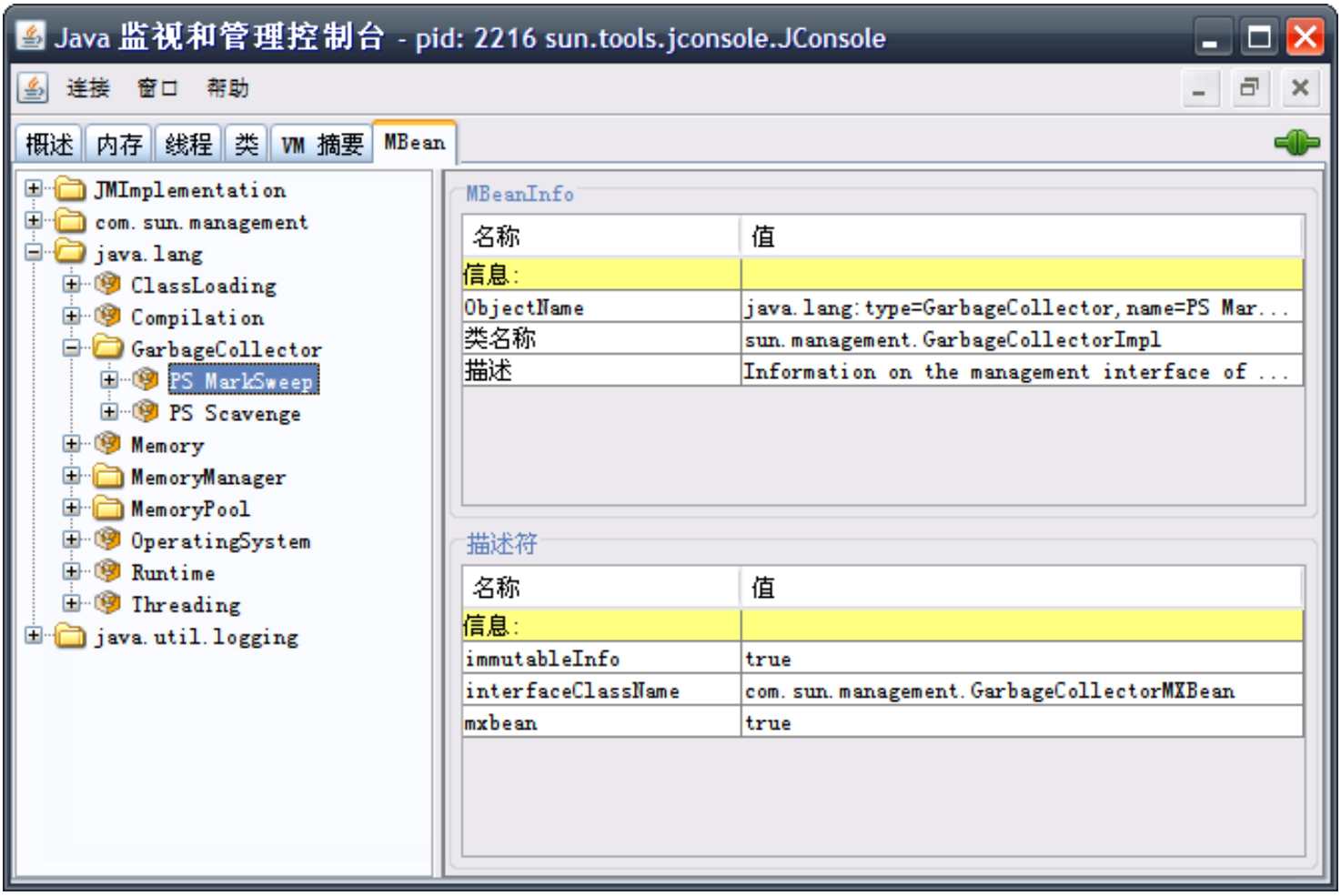
我回答说因为full GC概念只有在分代式GC的上下文中才存在，而JVM并不强制要求GC使用分代式实现，所以JMX提供的标准MXBean API里不提供“full GC次数”这样的方法也正常。
既然“full GC”本来就是非常平台相关的概念，那就hack一点，用平台相关的代码来解决问题好了。这些GC的MXBean都是有名字的，而主流的JVM的GC名字相对稳定，非要通过JMX得到full GC次数的话，用名字来判断一下就好了。

举个例子来看看。通过JDK 6自带的JConsole工具来查看相关的MXBean的话，可以看到，

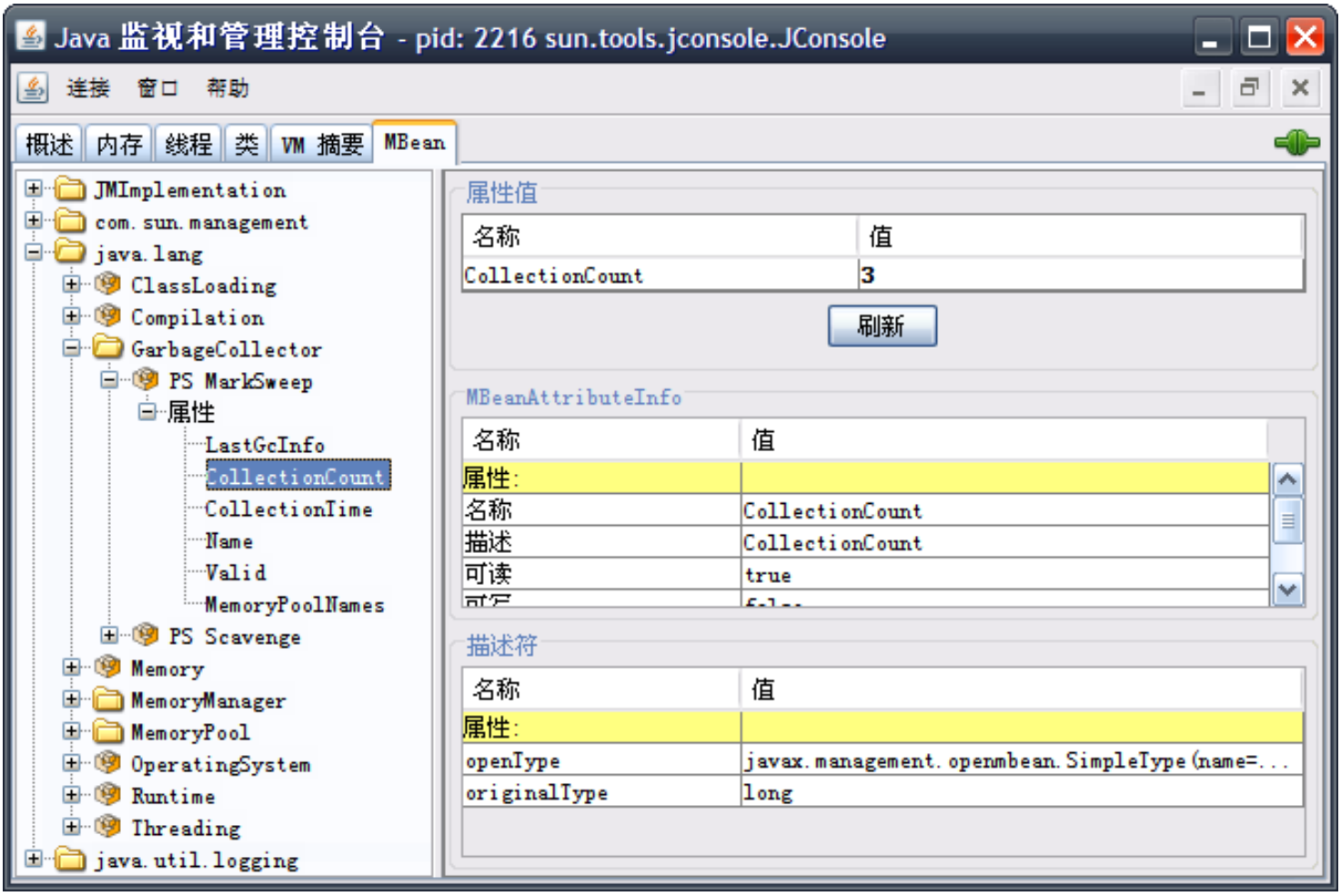
GC的MXBean在这个位置：



这个例子是用server模式启动JConsole的，使用的是ParallelScavenge GC，它的年老代对应的收集器在这里：



该收集器的总收集次数在此，这也就是full GC的次数：



于是只要知道我们用的JVM提供的GC MXBean的名字与分代的关系，就可以知道full GC的次数了。

Java代码写起来冗长，这帖就不用Java来写例子了，反正API是一样的，意思能表达清楚就OK。

用一个Groovy脚本简单演示一下适用于Oracle (Sun) HotSpot与Oracle (BEA) JRockit的GC统计程序：

```
import java.lang.management.ManagementFactory

printGCStats = {
  def youngGenCollectorNames = [
    // Oracle (Sun) HotSpot
    // -XX:+UseSerialGC
    'Copy',
    // -XX:+UseParNewGC
    'ParNew',
    // -XX:+UseParallelGC
    'PS Scavenge',

    // Oracle (BEA) JRockit
    // -XgcPrio:pausetime
```

```
  'Garbage collection optimized for short pausetimes Young Collector',
  // -XgcPrio:throughput
  'Garbage collection optimized for throughput Young Collector',
  // -XgcPrio:deterministic
  'Garbage collection optimized for deterministic pausetimes Young Collector'
]

def oldGenCollectorNames = [
  // Oracle (Sun) HotSpot
  // -XX:+UseSerialGC
  'MarkSweepCompact',
  // -XX:+UseParallelGC and (-XX:+UseParallelOldGC or -XX:+UseParallelOldGCCompacting)
  'PS MarkSweep',
  // -XX:+UseConcMarkSweepGC
  'ConcurrentMarkSweep',

  // Oracle (BEA) JRockit
  // -XgcPrio:pausetime
  'Garbage collection optimized for short pausetimes Old Collector',
  // -XgcPrio:throughput
  'Garbage collection optimized for throughput Old Collector',
  // -XgcPrio:deterministic
  'Garbage collection optimized for deterministic pausetimes Old Collector'
]

R: {
  ManagementFactory.garbageCollectorMXBeans.each {
    def name  = it.name
    def count = it.collectionCount
    def gcType;
    switch (name) {
    case youngGenCollectorNames:
      gcType = 'Minor Collection'
      break
    case oldGenCollectorNames:
      gcType = 'Major Collection'
      break
```

```
      default:
        gcType = 'Unknown Collection Type'
        break
      }
      println "$count <- $gcType: $name"
    }
  }
}

printGCStats()
```

执行可以看到类似这样的输出：

```
5 <- Minor Collection: Copy
0 <- Major Collection: MarkSweepCompact
```

↑这是用client模式的HotSpot执行得到的；

```
0 <- Minor Collection: Garbage collection optimized for throughput Young Collector
0 <- Major Collection: Garbage collection optimized for throughput Old Collector
```

↑这是用JRockit R28在32位Windows上的默认模式得到的。

通过上述方法，要包装起来方便以后使用的话也很简单，例如下面Groovy程序：

```
import java.lang.management.ManagementFactory

class GCStats {
  static final List<String> YoungGenCollectorNames = [
    // Oracle (Sun) HotSpot
    // -XX:+UseSerialGC
    'Copy',
    // -XX:+UseParNewGC
    'ParNew',
    // -XX:+UseParallelGC
```

```
    'PS Scavenge',

    // Oracle (BEA) JRockit
    // -XgcPrio:pausetime
    'Garbage collection optimized for short pausetimes Young Collector',
    // -XgcPrio:throughput
    'Garbage collection optimized for throughput Young Collector',
    // -XgcPrio:deterministic
    'Garbage collection optimized for deterministic pausetimes Young Collector'
]

static final List<String> OldGenCollectorNames = [
    // Oracle (Sun) HotSpot
    // -XX:+UseSerialGC
    'MarkSweepCompact',
    // -XX:+UseParallelGC and (-XX:+UseParallelOldGC or -XX:+UseParallelOldGCCompacting)
    'PS MarkSweep',
    // -XX:+UseConcMarkSweepGC
    'ConcurrentMarkSweep',

    // Oracle (BEA) JRockit
    // -XgcPrio:pausetime
    'Garbage collection optimized for short pausetimes Old Collector',
    // -XgcPrio:throughput
    'Garbage collection optimized for throughput Old Collector',
    // -XgcPrio:deterministic
    'Garbage collection optimized for deterministic pausetimes Old Collector'
]

static int getYoungGCCount() {
    ManagementFactory.garbageCollectorMXBeans.inject(0) { youngGCCount, gc ->
        if (YoungGenCollectorNames.contains(gc.name))
            youngGCCount + gc.collectionCount
        else
            youngGCCount
    }
}
```

```
  static int getFullGCCount() {
    ManagementFactory.garbageCollectorMXBeans.inject(0) { fullGCCount, gc ->
      if (OldGenCollectorNames.contains(gc.name))
        fullGCCount + gc.collectionCount
      else
        fullGCCount
    }
  }
}
```

用的时候：

```
D:\>\sdk\groovy-1.7.2\bin\groovysh
Groovy Shell (1.7.2, JVM: 1.6.0_20)
Type 'help' or '\h' for help.
------------------------------------------------
groovy:000> GCStats.fullGCCount
===> 0
groovy:000> System.gc()
===> null
groovy:000> GCStats.fullGCCount
===> 1
groovy:000> System.gc()
===> null
groovy:000> System.gc()
===> null
groovy:000> GCStats.fullGCCount
===> 3
groovy:000> GCStats.youngGCCount
===> 9
groovy:000> GCStats.youngGCCount
===> 9
groovy:000> GCStats.youngGCCount
===> 9
groovy:000> System.gc()
```

```
===> null
groovy:000> GCStats.youngGCCount
===> 9
groovy:000> GCStats.fullGCCount
===> 4
groovy:000> quit
```

这是在Sun JDK 6 update 20上跑的。顺带一提，如果这是跑在JRockit上的话，那full GC的次数就不会增加——因为JRockit里System.gc()默认是触发young GC的；请不要因为Sun HotSpot的默认行为而认为System.gc()总是会触发full GC的。

关于JMX的MXBean的使用，也可以参考下面两篇文档：
Groovy and JMX
Monitoring the JVM Heap with JRuby

3.2 如何更快的启动eclipse

发表时间: 2011-02-22

总是感觉自己的elipse启动比别人的慢，开始以为是装的插件太多（pydev，GAE，scala……）或者是导入的项目有点大。后来把-Xloggc:gc.log这个配置加上去看看启动的日志，吓了一跳，一次启动做了9次fullgc。和jboss服务器一样，肯定可以优化一下配置来更少的full gc来节约启动时间。

第一次优化：把-Xms（初始化堆大小） -Xmx（JVM最大堆大小）设置为一样大小512m，避免GC后JVM重新分配内存。但是重启eclipse的时候full gc的次数并没有减少，而且启动的时候GC全部变成了full gc，日志如下：

```
3.308: [Full GC 3.308: [Tenured: 0K->19530K(262144K), 0.1515426 secs] 172215K->19530K(498112K), [Perm : 16383K->16383K(16384K)], 0.1516281 secs] [Times: user=0.14 sys=0.00, real=0.15 secs]

8.472: [Full GC 8.472: [Tenured: 19530K->34170K(262144K), 0.2060534 secs] 145021K->34170K(498112K), [Perm : 20479K->20479K(20480K)], 0.2061412 secs] [Times: user=0.19 sys=0.00, real=0.21 secs]

9.027: [Full GC 9.027: [Tenured: 34170K->35855K(262144K), 0.1790415 secs] 54259K->35855K(498112K), [Perm : 24575K->24575K(24576K)], 0.1791281 secs] [Times: user=0.19 sys=0.00, real=0.18 secs]

10.004: [Full GC 10.004: [Tenured: 35855K->44735K(262144K), 0.2850547 secs] 81210K->44735K(498112K), [Perm : 28671K->28646K(28672K)], 0.2851505 secs] [Times: user=0.28 sys=0.00, real=0.28 secs]

10.725: [Full GC 10.725: [Tenured: 44735K->49542K(262144K), 0.2657311 secs] 71680K->49542K(498112K), [Perm : 32759K->32759K(32768K)], 0.2658216 secs] [Times: user=0.25 sys=0.00, real=0.27 secs]

12.057: [Full GC 12.057: [Tenured: 49542K->64706K(262144K), 0.3637080 secs] 179985K->64706K(498112K), [Perm : 36863K->36863K(36864K)], 0.3637938 secs] [Times: user=0.37 sys=0.00, real=0.36 secs]

12.788: [Full GC 12.788: [Tenured: 64706K->65640K(262144K), 0.3229940 secs] 87100K->65640K(498112K), [Perm : 40959K->40959K(40960K)], 0.3230836 secs] [Times: user=0.31 sys=0.00, real=0.32 secs]

13.652: [Full GC 13.652: [Tenured: 65640K->70639K(262144K), 0.4553435 secs] 116918K->70639K(498112K), [Perm : 45055K->44963K(45056K)], 0.4554289 secs] [Times: user=0.45 sys=0.00, real=0.46 secs]
```

14.679: [Full GC 14.679: [Tenured: 70639K->72308K(262144K), 0.4009647 secs] 122313K->72308K(498112K), [Perm : 49151K->49151K(49152K)], 0.4010552 secs] [Times: user=0.38 sys=0.00, real=0.40 secs]

从日志中分析可以看出：触发full gc的罪魁祸首是Perm，这个没有设置，所以继续优化！

第二次优化：-XX:PermSize=64m -XX:MaxPermSize=64m，把持久化的初始化大小和最大大小设置为一样。Full gc消失了，来了24次minor gc。

0.689: [GC 0.689: [DefNew: 32256K->2724K(36288K), 0.0108873 secs] 32256K->2724K(520256K), 0.0109685 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

1.020: [GC 1.020: [DefNew: 34980K->3090K(36288K), 0.0159294 secs] 34980K->5812K(520256K), 0.0159941 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

1.451: [GC 1.451: [DefNew: 35346K->2612K(36288K), 0.0131000 secs] 38068K->8344K(520256K), 0.0131866 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

2.670: [GC 2.674: [DefNew: 34868K->4032K(36288K), 0.0338445 secs] 40600K->14881K(520256K), 0.0357554 secs] [Times: user=0.03 sys=0.02, real=0.04 secs]

3.537: [GC 3.537: [DefNew: 36280K->4032K(36288K), 0.0297593 secs] 47129K->19882K(520256K), 0.0298390 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]

3.595: [GC 3.595: [DefNew: 36223K->74K(36288K), 0.0121076 secs] 52074K->19924K(520256K), 0.0122015 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

4.108: [GC 4.108: [DefNew: 32330K->1755K(36288K), 0.0071144 secs] 52180K->21605K(520256K), 0.0071898 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

7.550: [GC 7.550: [DefNew: 34011K->4032K(36288K), 0.0460676 secs] 53861K->35250K(520256K), 0.0461438 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]

8.818: [GC 8.818: [DefNew: 36288K->4032K(36288K), 0.0352634 secs] 67506K->38332K(520256K), 0.0353470 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]

9.926: [GC 9.926: [DefNew: 36288K->4032K(36288K), 0.0410570 secs] 70588K->45524K(520256K), 0.0411413 secs] [Times: user=0.03 sys=0.02, real=0.04 secs]

10.332: [GC 10.332: [DefNew: 36288K->4031K(36288K), 0.0325734 secs] 77780K->52292K(520256K), 0.0326496 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]

10.583: [GC 10.583: [DefNew: 36287K->4031K(36288K), 0.0250005 secs] 84548K->57151K(520256K), 0.0250791 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

10.765: [GC 10.765: [DefNew: 36213K->4032K(36288K), 0.0691980 secs] 89333K->72388K(520256K), 0.0692885 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]

10.977: [GC 10.977: [DefNew: 36288K->4031K(36288K), 0.0426303 secs] 104644K->81872K(520256K), 0.0427115 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]

11.211: [GC 11.211: [DefNew: 36287K->4032K(36288K), 0.0550659 secs] 114128K->91896K(520256K), 0.0551464 secs] [Times: user=0.03 sys=0.02, real=0.06 secs]

11.641: [GC 11.641: [DefNew: 36288K->3147K(36288K), 0.0295076 secs] 124152K->93474K(520256K), 0.0296096 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]

12.591: [GC 12.591: [DefNew: 35403K->2274K(36288K), 0.0241671 secs] 125730K->95722K(520256K), 0.0242549 secs] [Times: user=0.02 sys=0.02, real=0.02 secs]

12.896: [GC 12.896: [DefNew: 34530K->3023K(36288K), 0.0193394 secs] 127978K->98567K(520256K), 0.0194275 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

13.249: [GC 13.249: [DefNew: 35280K->939K(36288K), 0.0161462 secs] 130824K->99419K(520256K), 0.0162313 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]

13.919: [GC 13.919: [DefNew: 33195K->2070K(36288K), 0.0124033 secs] 131675K->100550K(520256K), 0.0125083 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

14.396: [GC 14.396: [DefNew: 34326K->4032K(36288K), 0.0204527 secs] 132806K->104239K(520256K), 0.0205335 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

14.554: [GC 14.554: [DefNew: 36288K->4031K(36288K), 0.0554755 secs] 136495K->114252K(520256K), 0.0555567 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]

14.735: [GC 14.735: [DefNew: 36287K->4031K(36288K), 0.0728643 secs] 146508K->129069K(520256K), 0.0729860 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]

14.954: [GC 14.954: [DefNew: 36287K->4032K(36288K), 0.0529429 secs] 161325K->137308K(520256K), 0.0530283 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]

15.308: [GC 15.308: [DefNew: 36288K->1126K(36288K), 0.0192389 secs] 169564K->138221K(520256K), 0.0193313 secs]

从日志中分析可以看出：频繁的minor gc是由新生代没有设置自动分配造成的。

第三次优化：-Xmn256m 设置新生代大小为256M。好了，就4次minor gc。完成任务。日志如下：

3.592: [GC 3.592: [DefNew: 209792K->19904K(235968K), 0.0765218 secs] 209792K->19904K(498112K), 0.0766072 secs] [Times: user=0.06 sys=0.02, real=0.08 secs]

10.457: [GC 10.457: [DefNew: 229696K->26176K(235968K), 0.1996293 secs] 229696K->58203K(498112K), 0.1997121 secs] [Times: user=0.17 sys=0.03, real=0.20 secs]

12.862: [GC 12.862: [DefNew: 235968K->17131K(235968K), 0.1315169 secs] 267995K->74647K(498112K), 0.1315965 secs] [Times: user=0.14 sys=0.00, real=0.13 secs]

14.465: [GC 14.465: [DefNew: 226923K->26176K(235968K), 0.1363962 secs] 284439K->101396K(498112K), 0.1364835 secs]

最后的配置如下：

```
-Xmn128m

-Xms512m

-Xmx512m

-XX:PermSize=64m

-XX:MaxPermSize=64m

-verbose:gc

-XX:+PrintGCTimeStamps
```

```
-XX:+PrintGCDetails

-Xloggc:gc.log
```

4.1 JVM内存管理：深入Java内存区域与OOM

发表时间: 2010-11-08

Java与C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

概述：

对于从事C、C++程序开发的开发人员来说，在内存管理领域，他们即是拥有最高权力的皇帝又是执行最基础工作的劳动人民——拥有每一个对象的“所有权”，又担负着每一个对象生命开始到终结的维护责任。

对于Java程序员来说，不需要在为每一个new操作去写配对的delete/free，不容易出现内容泄漏和内存溢出错误，看起来由JVM管理内存一切都很美好。不过，也正是因为Java程序员把内存控制的权力交给了JVM，一旦出现泄漏和溢出，如果不了解JVM是怎样使用内存的，那排查错误将会是一件非常困难的事情。

VM运行时数据区域

JVM执行Java程序的过程中，会使用到各种数据区域，这些区域有各自的用途、创建和销毁时间。根据《Java虚拟机规范（第二版）》（下文称VM Spec）的规定，JVM包括下列几个运行时数据区域：

1.程序计数器（Program Counter Register）：

每一个Java线程都有一个程序计数器来用于保存程序执行到当前方法的哪一个指令，对于非Native方法，这个区域记录的是正在执行的VM原语的地址，如果正在执行的是Natvie方法，这个区域则为空（undefined）。此内存区域是唯一一个在VM Spec中没有规定任何OutOfMemoryError情况的区域。

2.Java虚拟机栈（Java Virtual Machine Stacks）

与程序计数器一样，VM栈的生命周期也是与线程相同。VM栈描述的是Java方法调用的内存模型：每个方法被执行的时候，都会同时创建一个帧（Frame）用于存储本地变量表、操作栈、动态链接、方法出入口等信息。每一个方法的调用至完成，就意味着一个帧在VM栈中的入栈至出栈的过程。在后文中，我们将着重讨论VM栈中本地变量表部分。

经常有人把Java内存简单的区分为堆内存（Heap）和栈内存（Stack），实际中的区域远比这种观点复杂，这样划分只是说明与变量定义密切相关的内存区域是这两块。其中所指的“堆”后面会专门描述，而所指的“栈”就是VM栈中各个帧的本地变量表部分。本地变量表存放了编译期可知的各种标量类型（boolean、byte、char、short、int、float、long、double）、对象引用（不是对象本身，仅仅是一个引用指针）、方法返回地址等。其中long和double会占用2个本地变量空间（32bit），其余占用1个。本地变量表在进入方法时进行分配，当进入一个方法时，这个方法需要在帧中分配多大的本地变量是一件完全确定的事情，在方法运行期间不改变本地变量表的大小。

在VM Spec中对这个区域规定了2中异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果VM栈可以动态扩展（VM Spec中允许固定长度的VM栈），当扩展时无法申请到足够内存则抛出OutOfMemoryError异常。

3.本地方法栈（Native Method Stacks）

本地方法栈与VM栈所发挥作用是类似的，只不过VM栈为虚拟机运行VM原语服务，而本地方法栈是为虚拟机使用到的Native方法服务。它的实现的语言、方式与结构并没有强制规定，甚至有的虚拟机（譬如Sun Hotspot虚拟机）直接就把本地方法栈和VM栈合二为一。和VM栈一样，这个区域也会抛出StackOverflowError和OutOfMemoryError异常。

4.Java堆（Java Heap）

对于绝大多数应用来说，Java堆是虚拟机管理最大的一块内存。Java堆是被所有线程共享的，在虚拟机启动时创建。Java堆的唯一目的就是存放对象实例，绝大部分的对象实例都在这里分配。这一点在VM Spec中的描述是：所有的实例以及数组都在堆上分配（原文：The heap is the runtime data area from which memory for all class instances and arrays is allocated），但是在逃逸分析和标量替换优化技术出现后，VM Spec的描述就显得并不那么准确了。

Java堆内还有更细致的划分：新生代、老年代，再细致一点的：eden、from survivor、to survivor，甚至更细粒度的本地线程分配缓冲（TLAB）等，无论对Java堆如何划分，目的都是为了更好的回收内存，或者更快的分配内存，在本章中我们仅仅针对内存区域的作用进行讨论，Java堆中的上述各个区域的细节，可参见本文第二章《JVM内存管理：深入垃圾收集器与内存分配策略》。

根据VM Spec的要求，Java堆可以处于物理上不连续的内存空间，它逻辑上是连续的即可，就像我们的磁盘空间一样。实现时可以选择实现成固定大小的，也可以是可扩展的，不过当前所有商业的虚拟机都是按照

可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中无法分配内存，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

5.方法区（Method Area）

叫“方法区”可能认识它的人还不太多，如果叫永久代（Permanent Generation）它的粉丝也许就多了。它还有个别名叫做Non-Heap（非堆），但是VM Spec上则描述方法区为堆的一个逻辑部分（原文：the method area is logically part of the heap），这个名字的问题还真容易令人产生误解，我们在这里就不纠结了。

方法区中存放了每个Class的结构信息，包括常量池、字段描述、方法描述等等。VM Space描述中对这个区域的限制非常宽松，除了和Java堆一样不需要连续的内存，也可以选择固定大小或者可扩展外，甚至可以选择不实现垃圾收集。相对来说，垃圾收集行为在这个区域是相对比较少发生的，但并不是某些描述那样永久代不会发生GC（至少对当前主流的商业JVM实现来说是如此），这里的GC主要是对常量池的回收和对类的卸载，虽然回收的“成绩”一般也比较差强人意，尤其是类卸载，条件相当苛刻。

6.运行时常量池（Runtime Constant Pool）

Class文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量表(constant_pool table)，用于存放编译期已可知的常量，这部分内容将在类加载后进入方法区（永久代）存放。但是Java语言并不要求常量一定只有编译期预置入Class的常量表的内容才能进入方法区常量池，运行期间也可将新内容放入常量池（最典型的String.intern()方法）。

运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法在申请到内存时会抛出OutOfMemoryError异常。

7.本机直接内存（Direct Memory）

直接内存并不是虚拟机运行时数据区的一部分，它根本就是本机内存而不是VM直接管理的区域。但是这部分内存也会导致OutOfMemoryError异常出现，因此我们放到这里一起描述。

在JDK1.4中新加入了NIO类，引入一种基于渠道与缓冲区的I/O方式，它可以通过本机Native函数库直接分配本机内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java对和本机堆中来回复制数据。

显然本机直接内存的分配不会受到Java堆大小的限制，但是即然是内存那肯定还是要受到本机物理内存（包括SWAP区或者Windows虚拟内存）的限制的，一般服务器管理员配置JVM参数时，会根据实际内存设置-

Xmx等参数信息，但经常忽略掉直接内存，使得各个内存区域总和大于物理内存限制（包括物理的和操作系统级的限制），而导致动态扩展时出现OutOfMemoryError异常。

实战OutOfMemoryError

上述区域中，除了程序计数器，其他在VM Spec中都描述了产生OutOfMemoryError（下称OOM）的情形，那我们就实战模拟一下，通过几段简单的代码，令对应的区域产生OOM异常以便加深认识，同时初步介绍一些与内存相关的虚拟机参数。下文的代码都是基于Sun Hotspot虚拟机1.6版的实现，对于不同公司的不同版本的虚拟机，参数与程序运行结果可能结果会有所差别。

Java堆

Java堆存放的是对象实例，因此只要不断建立对象，并且保证GC Roots到对象之间有可达路径即可产生OOM异常。测试中限制Java堆大小为20M，不可扩展，通过参数-XX:+HeapDumpOnOutOfMemoryError让虚拟机在出现OOM异常的时候Dump出内存映像以便分析。（关于Dump映像文件分析方面的内容，可参见本文第三章《JVM内存管理：深入JVM内存异常分析与调优》。）

清单1：Java堆OOM测试

```
/**
 * VM Args：-Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
 * @author zzm
 */
public class HeapOOM {

    static class OOMObject {
    }
```

```
    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();

        while (true) {
            list.add(new OOMObject());
        }
    }
}
```

运行结果：

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3404.hprof ...
Heap dump file created [22045981 bytes in 0.663 secs]
```

VM栈和本地方法栈

Hotspot虚拟机并不区分VM栈和本地方法栈，因此-Xoss参数实际上是无效的，栈容量只由-Xss参数设定。关于VM栈和本地方法栈在VM Spec描述了两种异常：StackOverflowError与OutOfMemoryError，当栈空间无法继续分配分配时，到底是内存太小还是栈太大其实某种意义上是对同一件事情的两种描述而已，在笔者的实验中，对于单线程应用尝试下面3种方法均无法让虚拟机产生OOM，全部尝试结果都是获得SOF异常。

1.使用-Xss参数削减栈内存容量。结果：抛出SOF异常时的堆栈深度相应缩小。

2.定义大量的本地变量，增大此方法对应帧的长度。结果：抛出SOF异常时的堆栈深度相应缩小。

3.创建几个定义很多本地变量的复杂对象，打开逃逸分析和标量替换选项，使得JIT编译器允许对象拆分后在栈中分配。结果：实际效果同第二点。

清单2：VM栈和本地方法栈OOM测试（仅作为第1点测试程序）

```
/**
 * VM Args：-Xss128k
 * @author zzm
 */
public class JavaVMStackSOF {

    private int stackLength = 1;

    public void stackLeak() {
        stackLength++;
        stackLeak();
    }

    public static void main(String[] args) throws Throwable {
        JavaVMStackSOF oom = new JavaVMStackSOF();
        try {
            oom.stackLeak();
```

```
            } catch (Throwable e) {
                System.out.println("stack length:" + oom.stackLength);
                throw e;
            }
        }
    }
```

运行结果：

```
stack length:2402
Exception in thread "main" java.lang.StackOverflowError
        at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:20)
        at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)
        at org.fenixsoft.oom.JavaVMStackSOF.stackLeak(JavaVMStackSOF.java:21)
```

如果在多线程环境下，不断建立线程倒是可以产生OOM异常，但是基本上这个异常和VM栈空间够不够关系没有直接关系，甚至是给每个线程的VM栈分配的内存越多反而越容易产生这个OOM异常。

原因其实很好理解，操作系统分配给每个进程的内存是有限制的，譬如32位Windows限制为2G，Java堆和方法区的大小JVM有参数可以限制最大值，那剩余的内存为2G（操作系统限制）-Xmx（最大堆）-MaxPermSize（最大方法区），程序计数器消耗内存很小，可以忽略掉，那虚拟机进程本身耗费的内存不计算的话，剩下的内存就供每一个线程的VM栈和本地方法栈瓜分了，那自然每个线程中VM栈分配内存越多，就越容易把剩下的内存耗尽。

清单3：创建线程导致OOM异常

```
/**
 * VM Args：-Xss2M （这时候不妨设大些）
 * @author zzm
 */
public class JavaVMStackOOM {

    private void dontStop() {
        while (true) {
        }
    }

    public void stackLeakByThread() {
        while (true) {
            Thread thread = new Thread(new Runnable() {
                @Override
                public void run() {
                    dontStop();
                }
            });
            thread.start();
```

```
            }
        }

        public static void main(String[] args) throws Throwable {
            JavaVMStackOOM oom = new JavaVMStackOOM();
            oom.stackLeakByThread();
        }
    }
```

特别提示一下，如果读者要运行上面这段代码，记得要存盘当前工作，上述代码执行时有很大令操作系统卡死的风险。

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create
new native thread
```

运行时常量池

要在常量池里添加内容，最简单的就是使用String.intern()这个Native方法。由于常量池分配在方法区内，我们只需要通过-XX:PermSize和-XX:MaxPermSize限制方法区大小即可限制常量池容量。实现代码如下：

清单4：运行时常量池导致的OOM异常

```
/**
 * VM Args：-XX:PermSize=10M -XX:MaxPermSize=10M
 * @author zzm
 */
public class RuntimeConstantPoolOOM {

    public static void main(String[] args) {
        // 使用List保持着常量池引用，压制Full GC回收常量池行为
        List<String> list = new ArrayList<String>();
        // 10M的PermSize在integer范围内足够产生OOM了
        int i = 0;
        while (true) {
            list.add(String.valueOf(i++).intern());
        }
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
    at java.lang.String.intern(Native Method)
    at
org.fenixsoft.oom.RuntimeConstantPoolOOM.main(RuntimeConstantPoolOOM.java:18)
```

方法区

上文讲过，方法区用于存放Class相关信息，所以这个区域的测试我们借助CGLib直接操作字节码动态生成大量的Class，值得注意的是，这里我们这个例子中模拟的场景其实经常会在实际应用中出现：当前很多主流框架，如Spring、Hibernate对类进行增强时，都会使用到CGLib这类字节码技术，当增强的类越多，就需要越大的方法区用于保证动态生成的Class可以加载入内存。

清单5：借助CGLib使得方法区出现OOM异常

```
/**
 * VM Args： -XX:PermSize=10M -XX:MaxPermSize=10M
 * @author zzm
 */
public class JavaMethodAreaOOM {

    public static void main(String[] args) {
        while (true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                public Object intercept(Object obj, Method method,
Object[] args, MethodProxy proxy) throws Throwable {
```

```
                    return proxy.invokeSuper(obj, args);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject {

    }
}
```

运行结果：

```
Caused by: java.lang.OutOfMemoryError: PermGen space
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClassCond(ClassLoader.java:632)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:616)
    ... 8 more
```

本机直接内存

DirectMemory容量可通过-XX:MaxDirectMemorySize指定，不指定的话默认与Java堆（-Xmx指定）一样，下文代码越过了DirectByteBuffer，直接通过反射获取Unsafe实例进行内存分配（Unsafe类的getUnsafe()方法限制了只有引导类加载器才会返回实例，也就是基本上只有rt.jar里面的类的才能使用），因为DirectByteBuffer也会抛OOM异常，但抛出异常时实际上并没有真正向操作系统申请分配内存，而是通过计算得知无法分配既会抛出，真正申请分配的方法是unsafe.allocateMemory()。

```
/**
 * VM Args：-Xmx20M -XX:MaxDirectMemorySize=10M
 * @author zzm
 */
public class DirectMemoryOOM {

    private static final int _1MB = 1024 * 1024;

    public static void main(String[] args) throws Exception {
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while (true) {
            unsafe.allocateMemory(_1MB);
        }
    }
}
```

运行结果：

```
Exception in thread "main" java.lang.OutOfMemoryError
    at sun.misc.Unsafe.allocateMemory(Native Method)
    at
org.fenixsoft.oom.DirectMemoryOOM.main(DirectMemoryOOM.java:20)
```

总结

到此为止，我们弄清楚虚拟机里面的内存是如何划分的，哪部分区域，什么样的代码、操作可能导致OOM异常。虽然Java有垃圾收集机制，但OOM仍然离我们并不遥远，本章内容我们只是知道各个区域OOM异常出现的原因，下一章我们将看看Java垃圾收集机制为了避免OOM异常出现，做出了什么样的努力。

4.2 JVM内存管理：深入垃圾收集器与内存分配策略

发表时间: 2010-11-08

Java与C++之间有一堵由内存动态分配和垃圾收集技术所围成的高墙，墙外面的人想进去，墙里面的人却想出来。

概述：

说起垃圾收集（Garbage Collection，下文简称GC），大部分人都把这项技术当做Java语言的伴生产物。事实上GC的历史远远比Java来得久远，在1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期，人们就在思考GC需要完成的3件事情：哪些内存需要回收？什么时候回收？怎么样回收？

经过半个世纪的发展，目前的内存分配策略与垃圾回收技术已经相当成熟，一切看起来都进入“自动化”的时代，那为什么我们还要去了解GC和内存分配？答案很简单：当需要排查各种内存溢出、泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术有必要的监控、调节手段。

把时间从1960年拨回现在，回到我们熟悉的Java语言。本文第一章中介绍了Java内存运行时区域的各个部分，其中程序计数器、VM栈、本地方法栈三个区域随线程而生，随线程而灭；栈中的帧随着方法进入、退出而有条不紊的进行着出栈入栈操作；每一个帧中分配多少内存基本上是在Class文件生成时就已知的（可能会由JIT动态晚期编译进行一些优化，但大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收具备很高的确定性，因此在这几个区域不需要过多考虑回收的问题。而Java堆和方法区（包括运行时常量池）则不一样，我们必须等到程序实际运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，我们本文后续讨论中的“内存”分配与回收仅仅指这一部分内存。

对象已死？

在堆里面存放着Java世界中几乎所有的对象，在回收前首先要确定这些对象之中哪些还在存活，哪些已经“死去”了，即不可能再被任何途径使用的对象。

引用计数算法（Reference Counting）

最初的想法，也是很多教科书判断对象是否存活的算法是这样的：给对象中添加一个引用计数器，当有一个地方引用它，计数器加1，当引用失效，计数器减1，任何时刻计数器为0的对象就是不可能再被使用的。

客观的说，引用计数算法实现简单，判定效率很高，在大部分情况下它都是一个不错的算法，但引用计数算法无法解决对象循环引用的问题。举个简单的例子：对象A和B分别有字段b、a，令A.b=B和B.a=A，除此之

外这2个对象再无任何引用，那实际上这2个对象已经不可能再被访问，但是引用计数算法却无法回收他们。

根搜索算法（GC Roots Tracing）

在实际生产的语言中（Java、C#、甚至包括前面提到的Lisp），都是使用根搜索算法判定对象是否存活。算法基本思路就是通过一系列的称为“GC Roots”的点作为起始进行向下搜索，当一个对象到GC Roots没有任何引用链（Reference Chain）相连，则证明此对象是不可用的。在Java语言中，GC Roots包括：

1.在VM栈（帧中的本地变量）中的引用
2.方法区中的静态引用
3.JNI（即一般说的Native方法）中的引用

生存还是死亡？

判定一个对象死亡，至少经历两次标记过程：如果对象在进行根搜索后，发现没有与GC Roots相连接的引用链，那它将会被第一次标记，并在稍后执行他的finalize()方法（如果它有的话）。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束。这点是必须的，否则一个对象在finalize()方法执行缓慢，甚至有死循环什么的将会很容易导致整个系统崩溃。finalize()方法是对象最后一次逃脱死亡命运的机会，稍后GC将进行第二次规模稍小的标记，如果在finalize()中对象成功拯救自己（只要重新建立到GC Roots的连接即可，譬如把自己赋值到某个引用上），那在第二次标记时它将被移除出“即将回收”的集合，如果对象这时候还没有逃脱，那基本上它就真的离死不远了。

需要特别说明的是，这里对finalize()方法的描述可能带点悲情的艺术加工，并不代表笔者鼓励大家去使用这个方法来拯救对象。相反，笔者建议大家尽量避免使用它，这个不是C/C++里面的析构函数，它运行代价高昂，不确定性大，无法保证各个对象的调用顺序。需要关闭外部资源之类的事情，基本上它能做的使用try-finally可以做的更好。

关于方法区

方法区即后文提到的永久代，很多人认为永久代是没有GC的，《Java虚拟机规范》中确实说过可以不要求虚拟机在这区实现GC，而且这区GC的“性价比”一般比较低：在堆中，尤其是在新生代，常规应用进行一次GC可以一般可以回收70%~95%的空间，而永久代的GC效率远小于此。虽然VM Spec不要求，但当前生产中的商业JVM都有实现永久代的GC，主要回收两部分内容：废弃常量与无用类。这两点回收思想与Java堆中的对象回收很类似，都是搜索是否存在引用，常量的相对很简单，与对象类似的判定即可。而类的回收则比较苛刻，需要满足下面3个条件：

1.该类所有的实例都已经被GC，也就是JVM中不存在该Class的任何实例。
2.加载该类的ClassLoader已经被GC。

3.该类对应的java.lang.Class 对象没有在任何地方被引用，如不能在任何地方通过反射访问该类的方法。

是否对类进行回收可使用-XX:+ClassUnloading参数进行控制，还可以使用-verbose:class或者-XX:+TraceClassLoading、-XX:+TraceClassUnLoading查看类加载、卸载信息。

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JVM具备类卸载的支持以保证永久代不会溢出。

垃圾收集算法

在这节里不打算大量讨论算法实现，只是简单的介绍一下基本思想以及发展过程。最基础的搜集算法是“标记 - 清除算法”（Mark-Sweep），如它的名字一样，算法分层“标记”和“清除”两个阶段，首先标记出所有需要回收的对象，然后回收所有需要回收的对象，整个过程其实前一节讲对象标记判定的时候已经基本介绍完了。说它是最基础的收集算法原因是后续的收集算法都是基于这种思路并优化其缺点得到的。它的主要缺点有两个，一是效率问题，标记和清理两个过程效率都不高，二是空间问题，标记清理之后会产生大量不连续的内存碎片，空间碎片太多可能会导致后续使用中无法找到足够的连续内存而提前触发另一次的垃圾搜集动作。

为了解决效率问题，一种称为“复制”（Copying）的搜集算法出现，它将可用内存划分为两块，每次只使用其中的一块，当半区内存用完了，仅将还存活的对象复制到另外一块上面，然后就把原来整块内存空间一次过清理掉。这样使得每次内存回收都是对整个半区的回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存就可以了，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，未免太高了一点。

现在的商业虚拟机中都是用了这一种收集算法来回收新生代，IBM有专门研究表明新生代中的对象98%是朝生夕死的，所以并不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的eden空间和2块较少的survivor空间，每次使用eden和其中一块survivor，当回收时将eden和survivor还存活的对象一次过拷贝到另外一块survivor空间上，然后清理掉eden和用过的survivor。Sun Hotspot虚拟机默认eden和survivor的大小比例是8:1，也就是每次只有10%的内存是“浪费”的。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有10%以内的对象存活，当survivor空间不够用时，需要依赖其他内存（譬如老年代）进行分配担保（Handle Promotion）。

复制收集算法在对象存活率高的时候，效率有所下降。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保用于应付半区内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。因此人们提出另外一种“标记 - 整理”（Mark-Compact）算法，标记过程仍然一样，但后续步骤不是进行直接清理，而是令所有存活的对象一端移动，然后直接清理掉这端边界以外的内存。

当前商业虚拟机的垃圾收集都是采用“分代收集”（Generational Collecting）算法，这种算法并没有什

么新的思想出现，只是根据对象不同的存活周期将内存划分为几块。一般是把Java堆分作新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法，譬如新生代每次GC都有大批对象死去，只有少量存活，那就选用复制算法只需要付出少量存活对象的复制成本就可以完成收集。

垃圾收集器

垃圾收集器就是收集算法的具体实现，不同的虚拟机会提供不同的垃圾收集器。并且提供参数供用户根据自己的应用特点和要求组合各个年代所使用的收集器。本文讨论的收集器基于Sun Hotspot虚拟机1.6版。

图1.Sun JVM1.6的垃圾收集器

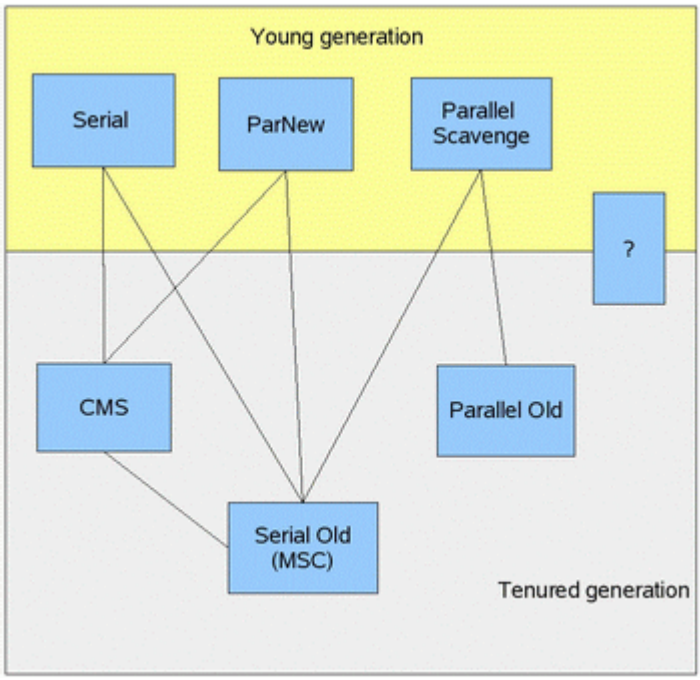


图1展示了1.6中提供的6种作用于不同年代的收集器，两个收集器之间存在连线的话就说明它们可以搭配使用。在介绍着些收集器之前，我们先明确一个观点：没有最好的收集器，也没有万能的收集器，只有最合适的收集器。

1.Serial收集器

单线程收集器，收集时会暂停所有工作线程（我们将这件事情称之为Stop The World，下称STW），使用复制收集算法，虚拟机运行在Client模式时的默认新生代收集器。

2.ParNew收集器

ParNew收集器就是Serial的多线程版本，除了使用多条收集线程外，其余行为包括算法、STW、对象分配规则、回收策略等都与Serial收集器一摸一样。对应的这种收集器是虚拟机运行在Server模式的默认新生代收集器，在单CPU的环境中，ParNew收集器并不会比Serial收集器有更好的效果。

3.Parallel Scavenge收集器

Parallel Scavenge收集器（下称PS收集器）也是一个多线程收集器，也是使用复制算法，但它的对象分配规则与回收策略都与ParNew收集器有所不同，它是以吞吐量最大化（即GC时间占总运行时间最小）为目标的收集器实现，它允许较长时间的STW换取总吞吐量最大化。

4.Serial Old收集器

Serial Old是单线程收集器，使用标记 - 整理算法，是老年代的收集器，上面三种都是使用在新生代收集器。

5.Parallel Old收集器

老年代版本吞吐量优先收集器，使用多线程和标记 - 整理算法，JVM 1.6提供，在此之前，新生代使用了PS收集器的话，老年代除Serial Old外别无选择，因为PS无法与CMS收集器配合工作。

6.CMS（Concurrent Mark Sweep）收集器

CMS是一种以最短停顿时间为目标的收集器，使用CMS并不能达到GC效率最高（总体GC时间最小），但它能尽可能降低GC时服务的停顿时间，这一点对于实时或者高交互性应用（譬如证券交易）来说至关重要，这类应用对于长时间STW一般是不可容忍的。CMS收集器使用的是标记 - 清除算法，也就是说它在运行期间会产生空间碎片，所以虚拟机提供了参数开启CMS收集结束后再进行一次内存压缩。

内存分配与回收策略

了解GC其中很重要一点就是了解JVM的内存分配策略：即对象在哪里分配和对象什么时候回收。

关于对象在哪里分配，往大方向讲，主要就在堆上分配，但也可能经过JIT进行逃逸分析后进行标量替换拆散为原子类型在栈上分配，也可能分配在DirectMemory中（详见本文第一章）。往细节处讲，对象主要分配在新生代eden上，也可能会直接老年代中，分配的细节决定于当前使用的垃圾收集器类型与VM相关参数设置。我们可以通过下面代码来验证一下Serial收集器（ParNew收集器的规则与之完全一致）的内存分配和回收的策略。读者看完Serial收集器的分析后，不妨自己根据JVM参数文档写一些程序去实践一下其它几种收集器的分配策略。

清单1：内存分配测试代码

```
public class YoungGenGC {

        private static final int _1MB = 1024 * 1024;

        public static void main(String[] args) {
                // testAllocation();
                testHandlePromotion();
                // testPretenureSizeThreshold();
```

```
        // testTenuringThreshold();
        // testTenuringThreshold2();
    }

    /**
     * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
     */
    @SuppressWarnings("unused")
    public static void testAllocation() {
        byte[] allocation1, allocation2, allocation3, allocation4;
        allocation1 = new byte[2 * _1MB];
        allocation2 = new byte[2 * _1MB];
        allocation3 = new byte[2 * _1MB];
        allocation4 = new byte[4 * _1MB];  // 出现一次Minor GC
    }

    /**
     * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
     * -XX:PretenureSizeThreshold=3145728
     */
    @SuppressWarnings("unused")
    public static void testPretenureSizeThreshold() {
        byte[] allocation;
        allocation = new byte[4 * _1MB];  //直接分配在老年代中
    }

    /**
     * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
     * -XX:+PrintTenuringDistribution
     */
    @SuppressWarnings("unused")
    public static void testTenuringThreshold() {
        byte[] allocation1, allocation2, allocation3;
        allocation1 = new byte[_1MB / 4];  // 什么时候进入老年代决定于XX:MaxTenuringThres
        allocation2 = new byte[4 * _1MB];
        allocation3 = new byte[4 * _1MB];
        allocation3 = null;
```

```
                allocation3 = new byte[4 * _1MB];
        }

        /**
         * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
         * -XX:+PrintTenuringDistribution
         */
        @SuppressWarnings("unused")
        public static void testTenuringThreshold2() {
                byte[] allocation1, allocation2, allocation3, allocation4;
                allocation1 = new byte[_1MB / 4];   // allocation1+allocation2大于survivo空间一半
                allocation2 = new byte[_1MB / 4];
                allocation3 = new byte[4 * _1MB];
                allocation4 = new byte[4 * _1MB];
                allocation4 = null;
                allocation4 = new byte[4 * _1MB];
        }

        /**
         * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
         */
        @SuppressWarnings("unused")
        public static void testHandlePromotion() {
                byte[] allocation1, allocation2, allocation3, allocation4, allocation5, allocat
                allocation1 = new byte[2 * _1MB];
                allocation2 = new byte[2 * _1MB];
                allocation3 = new byte[2 * _1MB];
                allocation1 = null;
                allocation4 = new byte[2 * _1MB];
                allocation5 = new byte[2 * _1MB];
                allocation6 = new byte[2 * _1MB];
                allocation4 = null;
                allocation5 = null;
                allocation6 = null;
                allocation7 = new byte[2 * _1MB];
        }
}
```

规则一：通常情况下，对象在eden中分配。当eden无法分配时，触发一次Minor GC。

执行testAllocation()方法后输出了GC日志以及内存分配状况。-Xms20M -Xmx20M -Xmn10M这3个参数确定了Java堆大小为20M，不可扩展，其中10M分配给新生代，剩下的10M即为老年代。-XX:SurvivorRatio=8决定了新生代中eden与survivor的空间比例是1：8，从输出的结果也清晰的看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为9216K（eden+1个survivor）。

我们也注意到在执行testAllocation()时出现了一次Minor GC，GC的结果是新生代6651K变为148K，而总占用内存则几乎没有减少（因为几乎没有可回收的对象）。这次GC是发生的原因是为allocation4分配内存的时候，eden已经被占用了6M，剩余空间已不足分配allocation4所需的4M内存，因此发生Minor GC。GC期间虚拟机发现已有的3个2M大小的对象全部无法放入survivor空间（survivor空间只有1M大小），所以直接转移到老年代去。GC后4M的allocation4对象分配在eden中。

清单2：testAllocation()方法输出结果

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4326K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  14% used [0x032d0000, 0x032f5370, 0x033d0000)
  to   space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
tenured generation   total 10240K, used 6144K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

规则二：配置了PretenureSizeThreshold的情况下，对象大于设置值将直接在老年代分配。

执行testPretenureSizeThreshold()方法后，我们看到eden空间几乎没有被使用，而老年代的10M控件被使用了40%，也就是4M的allocation对象直接就分配在老年代中，则是因为PretenureSizeThreshold被设置为3M，因此超过3M的对象都会直接从老年代分配。

清单3：

```
Heap
def new generation   total 9216K, used 671K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   8% used [0x029d0000, 0x02a77e98, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
compacting perm gen  total 12288K, used 2107K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)
No shared spaces configured.
```

规则三：在eden经过GC后存活，并且survivor能容纳的对象，将移动到survivor空间内，如果对象在survivor中继续熬过若干次回收（默认为15次）将会被移动到老年代中。回收次数由MaxTenuringThreshold设置。

分别以-XX:MaxTenuringThreshold=1和-XX:MaxTenuringThreshold=15两种设置来执行testTenuringThreshold()，方法中allocation1对象需要256K内存，survivor空间可以容纳。当MaxTenuringThreshold=1时，allocation1对象在第二次GC发生时进入老年代，新生代已使用的内存GC后非常干净的变成0KB。而MaxTenuringThreshold=15时，第二次GC发生后，allocation1对象则还留在新生代survivor空间，这时候新生代仍然有404KB被占用。

清单4：
MaxTenuringThreshold=1

```
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 1 (max 1)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs] [Times: user=0.02
sys=0.00, real=0.02 secs]
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 1 (max 1)
: 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
```

```
tenured generation   total 10240K, used 4500K [0x033d0000, 0x03dd0000, 0x03dd0000)
  the space 10240K,  43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.

MaxTenuringThreshold=15
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 15 (max 15)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 15 (max 15)
- age   2:     414520 bytes,     414520 total
: 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4582K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  39% used [0x031d0000, 0x03235338, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)
  the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
  the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

规则四：如果在survivor空间中相同年龄所有对象大小的累计值大于survivor空间的一半，大于或等于个年龄的对象就可以直接进入老年代，无需达到MaxTenuringThreshold中要求的年龄。

执行testTenuringThreshold2()方法，并将设置-XX:MaxTenuringThreshold=15，发现运行结果中survivor占用仍然为0%，而老年代比预期增加了6%，也就是说allocation1、allocation2对象都直接进入了老年代，而没有等待到15岁的临界年龄。因为这2个对象加起来已经到达了512K，并且它们是同年的，满足同年对象达到survivor空间的一半规则。我们只要注释掉其中一个对象new操作，就会发现另外一个就不会晋升到老年代中去了。

清单5：

```
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 1 (max 15)
- age   1:     676824 bytes,     676824 total
: 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs] [Times: user=0.00
sys=0.01, real=0.01 secs]
[GC [DefNew
Desired survivor size 524288 bytes, new threshold 15 (max 15)
: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
Heap
def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
tenured generation   total 10240K, used 4756K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)
compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

规则五：在Minor GC触发时，会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间，如果大于，改为直接进行一次Full GC，如果小于则查看HandlePromotionFailure设置看看是否允许担保失败，如果允许，那仍然进行Minor GC，如果不允许，则也要改为进行一次Full GC。

前面提到过，新生代才有复制收集算法，但为了内存利用率，只使用其中一个survivor空间来作为轮换备份，因此当出现大量对象在GC后仍然存活的情况（最极端就是GC后所有对象都存活），就需要老年代进行分配担保，把survivor无法容纳的对象直接放入老年代。与生活中贷款担保类似，老年代要进行这样的担保，前提就是老年代本身还有容纳这些对象的剩余空间，一共有多少对象在GC之前是无法明确知道的，所以取之前每一次GC晋升到老年代对象容量的平均值与老年代的剩余空间进行比较决定是否进行Full GC来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说如果某次Minor GC存活后的对象突增，大大高于平均值的话，依然会导致担保失败，这样就只好在失败后重新进行一次Full GC。虽然担保失败时做的绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure打开，避免Full GC过于频繁。

清单6：

```
HandlePromotionFailure = false
```

[GC [DefNew: 6651K->148K(9216K), 0.0078936 secs] 6651K->4244K(19456K), 0.0079192 secs] [Times: user=0.00 sys=0.02, real=0.02 secs]
[GC [DefNew: 6378K->6378K(9216K), 0.0000206 secs][Tenured: 4096K->4244K(10240K), 0.0042901 secs] 10474K->4244K(19456K), [Perm : 2104K->2104K(12288K)], 0.0043613 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

HandlePromotionFailure = true

[GC [DefNew: 6651K->148K(9216K), 0.0054913 secs] 6651K->4244K(19456K), 0.0055327 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 6378K->148K(9216K), 0.0006584 secs] 10474K->4244K(19456K), 0.0006857 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

总结

本章介绍了垃圾收集的算法、6款主要的垃圾收集器，以及通过代码实例具体介绍了新生代串行收集器对内存分配及回收的影响。

GC在很多时候都是系统并发度的决定性因素，虚拟机之所以提供多种不同的收集器，提供大量的调节参数，是因为只有根据实际应用需求、实现方式选择最优的收集方式才能获取最好的性能。没有固定收集器、参数组合，也没有最优的调优方法，虚拟机也没有什么必然的行为。笔者看过一些文章，撇开具体场景去谈论老年代达到92%会触发Full GC（92%应当来自CMS收集器触发的默认临界点）、98%时间在进行垃圾收集系统会抛出OOM异常（98%应该来自parallel收集器收集时间比率的默认临界点）其实意义并不太大。因此学习GC如果要到实践调优阶段，必须了解每个具体收集器的行为、优势劣势、调节参数。

4.3 深入理解JVM

发表时间: 2010-11-08

1 Java技术与Java虚拟机

说起Java，人们首先想到的是Java编程语言，然而事实上，Java是一种技术，它由四方面组成: Java编程语言、Java类文件格式、Java虚拟机和Java应用程序接口(Java API)。它们的关系如下图所示：

图1 Java四个方面的关系

运行期环境代表着Java平台，开发人员编写Java代码(.java文件)，然后将之编译成字节码(.class文件)。最后字节码被装入内存，一旦字节码进入虚拟机，它就会被解释器解释执行，或者是被即时代码发生器有选择的转换成机器码执行。从上图也可以看出Java平台由Java虚拟机和 Java应用程序接口搭建，Java语言则是进入这个平台的通道，用Java语言编写并编译的程序可以运行在这个平台上。这个平台的结构如下图所示：

在Java平台的结构中, 可以看出，Java虚拟机(JVM) 处在核心的位置，是程序与底层操作系统和硬件无关的关键。它的下方是移植接口，移植接口由两部分组成：适配器和Java操作系统, 其中依赖于平台的部分称为适配器；JVM 通过移植接口在具体的平台和操作系统上实现；在JVM 的上方是Java的基本类库和扩展类库以及它们的API， 利用Java API编写的应用程序(application) 和小程序(Java applet) 可以在任何Java平台上运行而无需考虑底层平台, 就是因为有Java虚拟机(JVM)实现了程序与操作系统的分离，从而实现了Java 的平台无关性。

那么到底什么是Java虚拟机(JVM)呢？通常我们谈论JVM时，我们的意思可能是：

1. 对JVM规范的的比较抽象的说明；
2. 对JVM的具体实现；
3. 在程序运行期间所生成的一个JVM实例。

对JVM规范的的抽象说明是一些概念的集合，它们已经在书《The Java Virtual Machine Specification》（《Java虚拟机规范》）中被详细地描述了；对JVM的具体实现要么是软件，要么是软件和硬件的组合，它已经被许多生产厂商所实现，并存在于多种平台之上；运行Java程序的任务由JVM的运行期实例单个承担。在本文中我们所讨论的Java虚拟机(JVM)主要针对第三种情况而言。它可以被看成一个想象中的机器，在实际的计算机上通过软件模拟来实现，有自己想象中的硬件，如处理器、堆栈、寄存器等，还有自己相应的指令系统。

JVM在它的生存周期中有一个明确的任务，那就是运行Java程序，因此当Java程序启动的时候，就产生JVM的一个实例；当程序运行结束的时候，该实例也跟着消失了。下面我们从JVM的体系结构和它的运行过程这两个方面来对它进行比较深入的研究。

2 Java虚拟机的体系结构

刚才已经提到，JVM可以由不同的厂商来实现。由于厂商的不同必然导致JVM在实现上的一些不同，然而JVM还是可以实现跨平台的特性，这就要归功于设计JVM时的体系结构了。

我们知道，一个JVM实例的行为不光是它自己的事，还涉及到它的子系统、存储区域、数据类型和指令这些部分，它们描述了JVM的一个抽象的内部体系结构，其目的不光规定实现JVM时它内部的体系结构，更重要的是提供了一种方式，用于严格定义实现时的外部行为。每个JVM都有两种机制，一个是装载具有合适名称的类(类或是接口)，叫做类装载子系统；另外的一个负责执行包含在已装载的类或接口中的指令，叫做运行引擎。每个JVM又包括方法区、堆、 Java栈、程序计数器和本地方法栈这五个部分，这几个部分和类装载机制与运行引擎机制一起组成的体系结构图为：

图3　JVM的体系结构

JVM的每个实例都有一个它自己的方法域和一个堆，运行于JVM内的所有的线程都共享这些区域；当虚拟机装载类文件的时候，它解析其中的二进制数据所包含的类信息，并把它们放到方法域中；当程序运行的时候，JVM把程序初始化的所有对象置于堆上；而每个线程创建的时候，都会拥有自己的程序计数器和 Java栈，其中程序计数器中的值指向下一条即将被执行的指令，线程的Java栈则存储为该线程调用Java方法的状态；本地方法调用的状态被存储在本地方法栈，该方法栈依赖于具体的实现。

下面分别对这几个部分进行说明。

执行引擎处于JVM的核心位置，在Java虚拟机规范中，它的行为是由指令集所决定的。尽管对于每条指令，规范很详细地说明了当JVM执行字节码遇到指令时，它的实现应该做什么，但对于怎么做却言之甚少。Java虚拟机支持大约248个字节码。每个字节码执行一种基本的CPU运算,例如,把一个整数加到寄存器,子程序转移等。Java指令集相当于Java程序的汇编语言。

Java指令集中的指令包含一个单字节的操作符,用于指定要执行的操作,还有0个或多个操作数,提供操作所需的参数或数据。许多指令没有操作数,仅由一个单字节的操作符构成。

虚拟机的内层循环的执行过程如下：

```
do{
```

```
取一个操作符字节;
根据操作符的值执行一个动作;
}while(程序未结束)
```

由于指令系统的简单性,使得虚拟机执行的过程十分简单,从而有利于提高执行的效率。指令中操作数的数量和大小是由操作符决定的。如果操作数比一个字节大,那么它存储的顺序是高位字节优先。例如,一个16位的参数存放时占用两个字节,其值为:

第一个字节*256+第二个字节字节码。

指令流一般只是字节对齐的。指令tableswitch和lookup是例外,在这两条指令内部要求强制的4字节边界对齐。

对于本地方法接口，实现JVM并不要求一定要有它的支持，甚至可以完全没有。Sun公司实现Java本地接口(JNI)是出于可移植性的考虑，当然我们也可以设计出其它的本地接口来代替Sun公司的JNI。但是这些设计与实现是比较复杂的事情，需要确保垃圾回收器不会将那些正在被本地方法调用的对象释放掉。

Java的堆是一个运行时数据区,类的实例(对象)从中分配空间，它的管理是由垃圾回收来负责的:不给程序员显式释放对象的能力。Java不规定具体使用的垃圾回收算法,可以根据系统的需求使用各种各样的算法。

Java方法区与传统语言中的编译后代码或是Unix进程中的正文段类似。它保存方法代码(编译后的java代码)和符号表。在当前的Java实现中,方法代码不包括在垃圾回收堆中,但计划在将来的版本中实现。每个类文件包含了一个Java类或一个Java界面的编译后的代码。可以说类文件是 Java语言的执行代码文件。为了保证类文件的平台无关性,Java虚拟机规范中对类文件的格式也作了详细的说明。其具体细节请参考Sun公司的Java 虚拟机规范。

Java虚拟机的寄存器用于保存机器的运行状态,与微处理器中的某些专用寄存器类似。Java虚拟机的寄存器有四种:

1. pc: Java程序计数器；
2. optop: 指向操作数栈顶端的指针；
3. frame: 指向当前执行方法的执行环境的指针；。
4. vars: 指向当前执行方法的局部变量区第一个变量的指针。

在上述体系结构图中，我们所说的是第一种，即程序计数器，每个线程一旦被创建就拥有了自己的程序计数器。当线程执行Java方法的时候，它包含该线程正在被执行的指令的地址。但是若线程执行的是一个本地的方法，那么程序计数器的值就不会被定义。

Java虚拟机的栈有三个区域:局部变量区、运行环境区、操作数区。

局部变量区

每个Java方法使用一个固定大小的局部变量集。它们按照与vars寄存器的字偏移量来寻址。局部变量都是32位的。长整数和双精度浮点数占据了两个局部变量的空间,却按照第一个局部变量的索引来寻址。(例如,一个具有索引n的局部变量,如果是一个双精度浮点数,那么它实际占据了索引n和n+1所代表的存储空间)虚拟机规范并不要求在局部变量中的64位的值是64位对齐的。虚拟机提供了把局部变量中的值装载到操作数栈的指令,也提供了把操作数栈中的值写入局部变量的指令。

运行环境区

在运行环境中包含的信息用于动态链接,正常的方法返回以及异常捕捉。

动态链接

运行环境包括对指向当前类和当前方法的解释器符号表的指针,用于支持方法代码的动态链接。方法的class文件代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用,装载必要的类以解释还没有定义的符号,并把变量访问翻译成与这些变量运行时的存储结构相应的偏移地址。动态链接方法和变量使得方法中使用的其它类的变化不会影响到本程序的代码。

正常的方法返回

如果当前方法正常地结束了,在执行了一条具有正确类型的返回指令时,调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的寄存器,并把调用者的程序计数器增加一个恰当的数值,以跳过已执行过的方法调用指令,然后在调用者的执行环境中继续执行下去。

异常捕捉

异常情况在Java中被称作Error(错误)或Exception(异常),是Throwable类的子类,在程序中的原因是:①动态链接错,如无法找到所需的class文件。②运行时错,如对一个空指针的引用。程序使用了throw语句。

当异常发生时,Java虚拟机采取如下措施:

- 检查与当前方法相联系的catch子句表。每个catch子句包含其有效指令范围,能够处理的异常类型,以及处理异常的代码块地址。
- 与异常相匹配的catch子句应该符合下面的条件:造成异常的指令在其指令范围之内,发生的异常类型是其能处理的异常类型的子类型。如果找到了匹配的catch子句,那么系统转移到指定的异常处理块处执行;如果没有找到异常处理块,重复寻找匹配的catch子句的过程,直到当前方法的所有嵌套的 catch子句都被检查过。
- 由于虚拟机从第一个匹配的catch子句处继续执行,所以catch子句表中的顺序是很重要的。因为Java代码是结构化的,因此总可以把某个方法的所有的异常处理器都按序排列到一个表中,对任意可能的程序计数器的值,都可以用线性的顺序找到合适的异常处理块,以处理在该程序计数器值下发生的异常情况。

- 如果找不到匹配的catch子句,那么当前方法得到一个"未截获异常"的结果并返回到当前方法的调用者,好像异常刚刚在其调用者中发生一样。如果在调用者中仍然没有找到相应的异常处理块,那么这种错误将被传播下去。如果错误被传播到最顶层,那么系统将调用一个缺省的异常处理块。

操作数栈区

机器指令只从操作数栈中取操作数,对它们进行操作,并把结果返回到栈中。选择栈结构的原因是:在只有少量寄存器或非通用寄存器的机器(如 Intel486)上,也能够高效地模拟虚拟机的行为。操作数栈是32位的。它用于给方法传递参数,并从方法接收结果,也用于支持操作的参数,并保存操作的结果。例如,iadd指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加,并把结果压回到操作数栈中。

每个原始数据类型都有专门的指令对它们进行必须的操作。每个操作数在栈中需要一个存储位置,除了long和double型,它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如,压入两个int类型的数,如果把它们当作是一个long类型的数则是非法的。在Sun的虚拟机实现中,这个限制由字节码验证器强制实行。但是,有少数操作(操作符dupe和swap),用于对运行时数据区进行操作时是不考虑类型的。

本地方法栈，当一个线程调用本地方法时，它就不再受到虚拟机关于结构和安全限制方面的约束，它既可以访问虚拟机的运行期数据区，也可以使用本地处理器以及任何类型的栈。例如，本地栈是一个C语言的栈，那么当C程序调用C函数时，函数的参数以某种顺序被压入栈，结果则返回给调用函数。在实现Java虚拟机时，本地方法接口使用的是C语言的模型栈，那么它的本地方法栈的调度与使用则完全与C语言的栈相同。

3 Java虚拟机的运行过程

上面对虚拟机的各个部分进行了比较详细的说明，下面通过一个具体的例子来分析它的运行过程。

虚拟机通过调用某个指定类的方法main启动，传递给main一个字符串数组参数，使指定的类被装载，同时链接该类所使用的其它的类型，并且初始化它们。例如对于程序：

```
class HelloApp
{
 public static void main(String[] args)
 {
  System.out.println("Hello World!");
  for (int i = 0; i < args.length; i++ )
  {
   System.out.println(args[i]);
  }
 }
}
```

编译后在命令行模式下键入： java HelloApp run virtual machine

将通过调用HelloApp的方法main来启动java虚拟机，传递给main一个包含三个字符串"run"、"virtual"、"machine"的数组。现在我们略述虚拟机在执行HelloApp时可能采取的步骤。

开始试图执行类HelloApp的main方法，发现该类并没有被装载，也就是说虚拟机当前不包含该类的二进制代表，于是虚拟机使用 ClassLoader试图寻找这样的二进制代表。如果这个进程失败，则抛出一个异常。类被装载后同时在main方法被调用之前，必须对类 HelloApp与其它类型进行链接然后初始化。链接包含三个阶段：检验，准备和解析。检验检查被装载的主类的符号和语义，准备则创建类或接口的静态域以及把这些域初始化为标准的默认值，解析负责检查主类对其它类或接口的符号引用，在这一步它是可选的。类的初始化是对类中声明的静态初始化函数和静态域的初始化构造方法的执行。一个类在初始化之前它的父类必须被初始化。整个过程如下：

图4：虚拟机的运行过程

4 结束语

本文通过对JVM的体系结构的深入研究以及一个Java程序执行时虚拟机的运行过程的详细分析，意在剖析清楚Java虚拟机的机理。

深入java虚拟机

本群组维护者

RednaxelaFX night_stalker QM42977 IcyFenix fantasy

iteye群组:
http://www.iteye.com/groups
申请群组:
http://www.iteye.com/groups/new