JAVA GC 与 JVM调优

关于JVM和内存相关，本博还有有 JVM深入浅出 和 Java Memory Leak详解 两篇文章，前者主要讲了JVM 内存模型的划分，后者主要讲了Java内存泄漏相关。

目前JAVA GC采用的是分代垃圾回收算法，Generational Collecting。

基于对象的生命周期，将JAVA堆分为年青代，年老代和持久代。对不同生命周期的对象采用不同的基本回收算法，从J2SE1.2就开始使用。

GC是只对JAVA堆中的对象进行垃圾回收，而JAVA堆中的引用都是从JAVA栈中来的。所以GC是从JAVA栈中找根结点，可能一个栈中有多个引用树。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GC ROOT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。

目前可作为GC Root的对象有：虚拟机栈中引用的对象（本地变量表）， 方法区中静态属性引用的对象，方法区中常量引用的对象和本地方法栈中引用的对象（Native对象）

使用堆存储空间对编程来说是个便利，也是一个负担，因为使用堆不当可能引起的致命问题：内存泄漏（Leak）或悬空引用（DanglingReference）。*内存泄漏*是指从堆上分配了空间，但在程序生命结束了还没有释放，而这些空间是被标记了被死掉的程序使用的，所以也不能再被其它程序使用，就成为了僵死的（zoombie），时间久了这些死尸没有回收“掩埋”，堆空间迟早会被耗尽，再有分配请求无法正常满足，结果必然导致系统不能正常工作。*悬空引用*就更危险了，悬空引用是指使用已经不再有效的实体，这些实体可能是使用之前自己把它释放掉了或者赋值给了别人，别人把它释放掉了。现在的系统基本都采取了存储空间的保护措施，悬空引用的也就是访问了分配给

别人的空间或者是空闲的空间，结果就会导致系统崩溃。下面来具体的来看一下引用的分类，到底有哪些类型的引用？每种引用都是做什么的呢？

Java中存在四种引用，每种引用如下：

1、 强引用， 只要引用存在，垃圾回收器永远不会回收

> Object obj = new Object(); //可直接通过obj取得对应的对象 如 obj.equels(new Object());
>
> 而这样 obj对象对后面new Object的一个强引用，只有当obj这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

2、 软引用，非必须引用，内存溢出之前进行回收，可以通过以下代码实现

> Object obj = new Object();
>
> SoftReference<Object> sf = newSoftReference<Object>(obj);
>
> obj = null;
>
> sf.get();//有时候会返回null

> 这时候sf是对obj的一个软引用，通过sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回null；
>
> 软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

3、 弱引用，第二次垃圾回收时回收，可以通过如下代码实现

> Object obj = new Object();
>
> WeakReference<Object> wf = newWeakReference<Object>(obj);
>
> obj = null;
>
> wf.get();//有时候会返回null

wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的isEnQueued方法返回对象是否被垃圾回收器

4、 虚引用（幽灵/幻影引用），垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

```
Object obj = new Object();

PhantomReference<Object> pf =new PhantomReference<Object>(obj);

obj=null;

pf.get();//永远返回null

pf.isEnQueued();//返回从内存中已经删除
```

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

一基本垃圾回收算法

1. 引用计数（Reference Counting）

比较古老的回收算法。注意不是类相互之间的引用，而是对象被引用的数目！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。

- 当创建了一个对象并把指向该对象的引用赋给一个变量时，这个对象的引用计数器就被置为1。

- 当对象的引用被赋给任意变量时，该对象的引用计数器加1。

- 当对象的引用超出了生存期或者被设置为新的值的时候，其引用计数器减1。

- 如果引用计数器为0，该对象就满足了垃圾回收的条件。

当代码出现下面的情形时，该算法将无法适应

a) ObjA.obj = ObjB

b) ObjB.obj = ObjA

这样的代码会产生如下引用情形 objA指向objB，而objB又指向objA，这样当其他所有的引用都消失了之后，objA和objB还有一个相互的引用，也就是说两个对象的引用计数器各为1，而实际上这两个对象都已经没有额外的引用，已经是垃圾了。

2. 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。

3. 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。

4. 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。

二Java堆区域划分

1. Young（年轻代）

年轻代分三个区。一个Eden区，两个Survivor区。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。由此可知，新生代通常存活时间较短，因此基于Copying算法来进行回收。

2. Tenured（年老代）

年老代存放从年轻代存活的对象。一般来说年老代存放的都是生命期较长的对象。年老代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记（Mark）算法来进行回收。

3. Perm（持久代）

用于放置最老的对象，这个区域不受-Xms和-Xmx参数影响，由the -XX:PermSize and -XX:MaxPermSize进行设置，在程序较稳定时可以将此参数设置一样的值，以防扩展区域造成时间和性能上的影响。当区间不够时，会抛出out of PermGen space ERROR。在JDK1.4之前，该区域不进行垃圾回收；之后，会默认也会被垃圾回收，但可以使用-noclassgc来关闭对该区域的垃圾回收。

三Java垃圾收集器

GC shortname	Generation	Command line parameter	Comment
Copy	Young	-XX:+UseSerialGC	The Copying collector

MarkSweepCompact	Tenured	-XX:+UseSerialGC	The Mark and Sweep Compactor
ConcurrentMarkSweep	Tenured	-XX:+UseConcMarkSweepGC	The Concurrent Mark and Sweep Compactor
ParNew	Young	-XX:+UseParNewGC	The parallel Young Generation Collector—can only be used with the Concurrent mark and sweep compactor.
PS Scavenge	Young	-XX:+UseParallelGC	The parallel object scavenger
PS MarkSweep	Tenured	-XX:+UseParallelOldGC	The parallel mark and sweep collector

从以上可以看出Young和Tenured分别有三种收集器，分别为串行，并行和并发。

- Serial 单线程
- Parallel 多GC线程并行, GC线程和App线程取一运行，即GC要Stop APP。
- Concurrent 多线程并发，GC线程和App线程可同时运行。(注: Young generation 没有CMS，取而代之的是可和CMS(Old)一起运行的ParNew)

四JVM如何调优

首先，选择合适的垃圾收集器

现在Serial的一般不使用了，默认的Collector是PS Scavenge和PS MarkSweep。所以Collector在Parallel和Concurrent之间选择。

如何选择Collector，先来看看衡量垃圾收集器的标准：吞吐量和时间延迟

> Ø Total Execution Time（程序运行总时间）= UsefulTime（程序运行时间）+ Paused Time（垃圾回收导致的暂停时间）
>
> Ø 吞吐量:表示没有将时间花在垃圾回收上的比例 = [Useful Time] / [Total Execution Time]
>
> Ø 时间延迟: 在程序运行期间暂停时间的平均值

吞吐量和时间延迟这两个参数，二者只能平衡，比如吞吐量可以通过扩大年轻代区域而得到改善，因为这将降低GC的频率，CPU用于运行程序的时间必然增多，但反过来，这会影响时间延迟，因为GC的时间与堆的大小成正比。对于实时程序和高交互性这类程序，更关心时间延迟，对于Web应用程序的Server端更关注吞吐量。

相同配置下，相对来说，并行Collector有因为有更多的CPU参与垃圾回收，故有更高的吞吐量参数，而并发Collector因为有APP继续运行故有更低的时间延迟参数。

下面就是做选择的时候了：

每个区域可以有不同类型的Collector，比如在年轻代中使用ParallelScavenge和在年老代中SerialOld collector就要使用参数: **java** -XX:+UseParallelGC

Switch	**Young Generation**	**Old Generation**
UseSerialGC	*Serial*	Serial Old (MSC)
UseParNewGC	ParNew	Serial Old (MSC)
UseConcMarkSweepGC	ParNew	*CMS (mostly used)* Serial Old (used when concurrent mode failure occurs)

+UseParallelGC	*Parallel Scavenge*	Serial Old
UseParallelOldGC	*Parallel Scavenge*	*Parallel Old*
+UseConcMarkSweepGC -UseParNewGC	*Serial*	*CMS* *Serial Old*

再次，需要调试配置Java堆各种区的大小

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。我在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。

五调优小结：

> 1. 堆区域不够大是导致GC的直接原因，所以要分配足够大的区域。试着分配调整50%-70%物理内存的大小，并记录不同的结果。
>
> 2. 将堆的最大和初始值设置一样，这将直接减少CPU再次分配区域的effort.
>
> 3. 设置合适大小的年轻代，需要设置的足够小以避免GC的Pause时间，还要足够大以此容纳足够的临时对象，正确使用NewSize和MaxNewSize参数，大约设置年轻代的大小为整个对区域的25%，还可以通过调整NewRatio参数来设置年轻代相对于年老代的比例。
>
> 4. 年轻代大小一定要少于整个堆区域的50%。
>
> 5. 只有在条件warrant的情况下尝试更复杂的选项，否则使用默认Collector。
>
> 6. 确保代码中将不需要的引用释放，特别是Collections中的对象。
>
> 7. 使用-verbose:gc,-XX:+PrintGC，-XX:+PrintGCDetails ，-XX:+PrintGCTimeStamps参数来监控GC Performance,理想情况下，要避免锯齿形图像，因为该图像表明每次GC之后都有大量的内存区域被释放。

- **-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), 0.0082960secs]

· -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 werestopped: 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 space49152K, 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。

除此之外，减少GC开销的措施

根据上述GC的机制,程序的运行会直接影响系统环境的变化,从而影响GC的触发。若不针对GC的特点进行设计和编码,就会出现内存驻留等一系列负面影响。为了避免这些影响,基本的原则就是尽可能地减少垃圾和减少GC过程中的开销。具体措施包括以下几个方面:

- 不要显式调用System.gc()，此函数建议JVM进行主GC,虽然只是建议而非一定,但很多情况下它会触发主GC,从而增加主GC的频率,也即增加了间歇性停顿的次数。
- 尽量减少临时对象的使用，临时对象在跳出函数调用后,会成为垃圾,少用临时变量就相当于减少了垃圾的产生,从而延长了出现上述第二个触

发条件出现的时间,减少了主GC的机会。

- 对象不用时最好显式置为Null，一般而言,为Null的对象都会被作为垃圾处理,所以将不用的对象显式地设为Null,有利于GC收集器判定垃圾,从而提高了GC的效率。
- 尽量使用StringBuffer,而不用String来累加字符串（详见blog另一篇文章JAVA中String与StringBuffer），由于String是固定长的字符串对象,累加String对象时,并非在一个String对象中扩增,而是重新创建新的String对象,如Str5=Str1+Str2+Str3+Str4,这条语句执行过程中会产生多个垃圾对象,因为对次作“+”操作时都必须创建新的String对象,但这些过渡对象对系统来说是没有实际意义的,只会增加更多的垃圾。避免这种情况可以改用StringBuffer来累加字符串,因StringBuffer是可变长的,它在原有基础上进行扩增,不会产生中间对象。
- 能用基本类型如Int,Long,就不用Integer,Long对象，基本类型变量占用的内存资源比相应对象占用的少得多,如果没有必要,最好使用基本变量。
- 尽量少用静态对象变量，静态变量属于全局变量,不会被GC回收,它们会一直占用内存。
- 分散对象创建或删除的时间，集中在短时间内大量创建新对象,特别是大对象,会导致突然需要大量内存,JVM在面临这种情况时,只能进行主GC,以回收内存或整合内存碎片,从而增加主 GC的频率。集中删除对象,道理也是一样的。它使得突然出现了大量的垃圾对象,空闲空间必然减少,从而大大增加了下一次创建新对象时强制主GC的机会。

六常见配置汇总

1. 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

2. 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

3. 垃圾回收统计信息

· -XX:+PrintGC

· -XX:+PrintGCDetails

· -XX:+PrintGCTimeStamps

· -Xloggc:filename

4. 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)

5. 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。

七检测工具

所谓“工欲善其事，必先利其器”，好的工具确能起到事半工倍的作用。用到的最多的就两个JConsole(JDK自带)，JVisualvm(JDK自带)和JProfiler（商业软件，有试用版本）。JConsole监控系统内存变化情况，如果有内存溢出的话，垃圾回收将会呈现锯齿状。发现问题以后，然后使用JProfiler，在小压力（或无压力）的情况下监控对象变化，定位内存溢出原因。

可以查看 JConsole的较详细使用， 而JVisualVM是JConcole的升级版，可参阅 JVisualVM的说明，使用很简单。

参考

http://www.softwareengineeringsolutions.com/blogs/2010/05/01/garbage-collection-in-java-part-3/