Jvm 调优浅谈

1.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：它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。

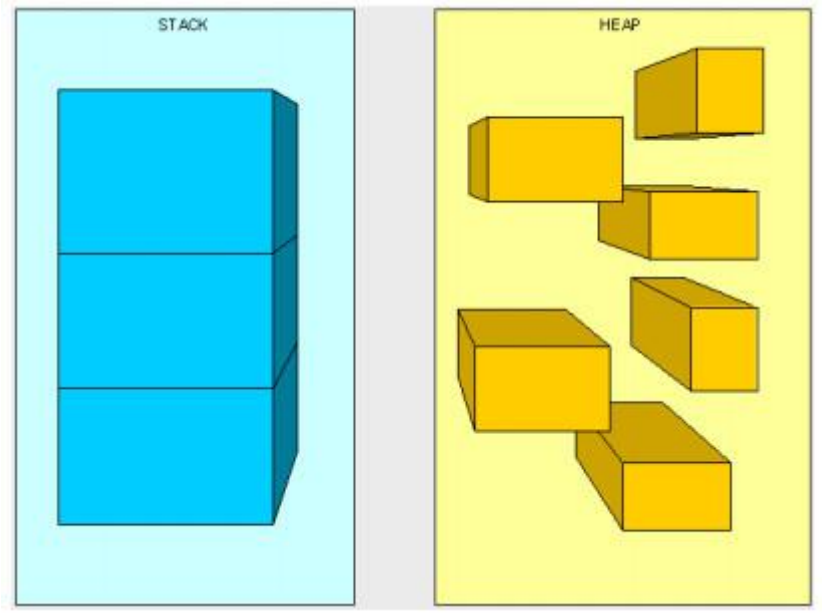
基本类型包括：byte、short、int、long、char、float、double、boolean、returnAddress？？

引用类型包括：类类型、接口类型和数组

byte	1B(8 位)	-128 ~ 127	0
short	2B(16 位)	-2^{15} ~ $2^{15}-1$	0
Int	4B(32 位)	-2^{31} ~ $2^{31}-1$	0
long	8B(64 位)	-2^{63} ~ $2^{63}-1$	0
char	2B(16 位)	0 ~ $2^{16}-1$	\U0000
float	4B(32 位)	1.4013E-45 ~3.4028E+38	0.0F
double	8B(64 位)	4.9E-324 ~1.7977E+308	0.0D
boolean	1B(8 位)	True, false	false

2.堆与栈

堆和栈是程序运行的关键，很有必要它他们的关系说清楚。



栈是运行时的单位，而堆是存储的单元。

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）相关的信息。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？

1. 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。

2.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。

3. 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。

4. 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，其实就是编写了数据结构，也编写了处理数据的逻辑。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 4byte 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，如果把它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，它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 中的参数传递是传值呢？还是传引用？

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。不会直接传递对象本身。

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，而且也简化了 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，而不是对象本身，即：修改的是堆中的数据。所以这个修改是可以保持的。

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，则还是一棵树（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的，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，当栈中存储的数据比较多时，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。

java 对象的大小

基本数据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，其大小就值得商讨。

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。看看下面语句：

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

这样在程序中完成了一个 **java** 对象的声明，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)+Boolea 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 的整数倍来分的，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，因此需要把它们作为对象来看待。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)，这些类型的内存占用更是夸张。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：软引用一般被作为缓存来使用。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，而且一般是在内存比较受限的情况下作为缓存。

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。

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

```
ReferenceQueue queue = new ReferenceQueue ();
PhantomReference pr = new PhantomReference
(object, queue);
```

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

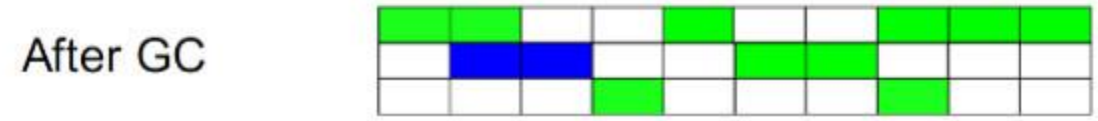
可以从不同的角度去划分垃圾回收算法：

按照基本回收策略分

引用计数（Reference Counting）

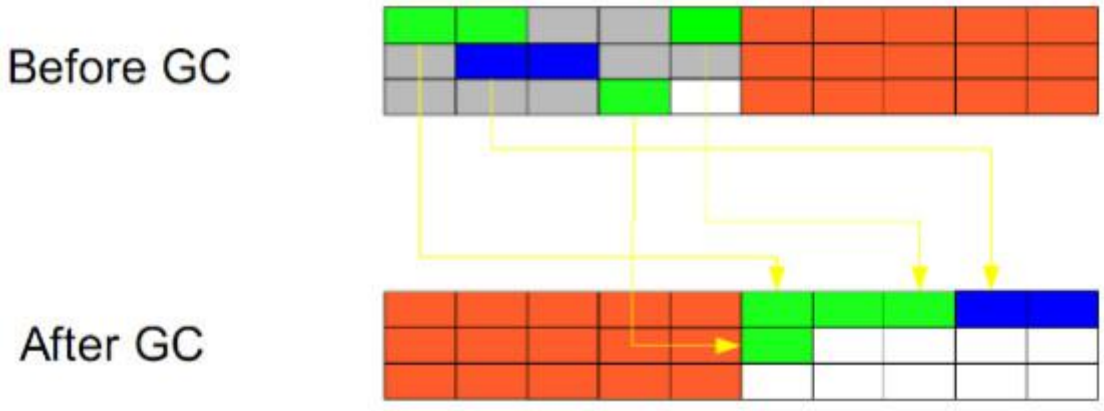
比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。

标记-清除（Mark-Sweep）



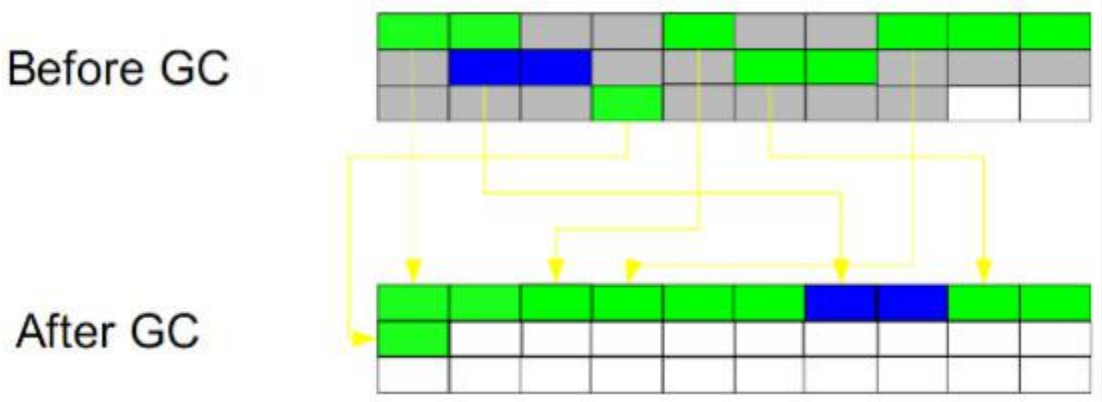
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

复制（Copying）



此算法把内存空间划分为两个相等的区域，每次只是用其中一个区域。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。此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，此算法的缺点也是比较明显的，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。也是分两个阶段，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”到堆中的其中一块，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。

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**年轻代、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

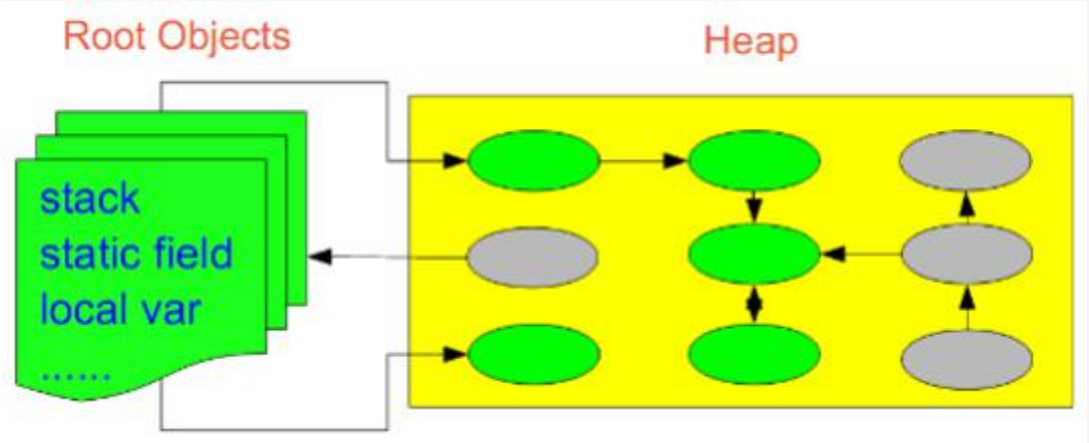
串行收集：串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。

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：GC 线程和应用线程大部分时间是并发执行，只是在初始标记（initial mark）和二次标记（remark）时需要 stop-the-world，这可以大大缩短停顿时间（pause time），所以适用于响应时间优先的应用，减少用户等待时间。由于 GC 是和应用线程并发执行，只有在多 CPU 场景下才能发挥其价值，在执行过程中还会产生新的垃圾 floating garbage，如果等空间满了再开始 GC，那这些新产生的垃圾就没地方放了，这时就会启动一次串行 GC，等待时间将会很长，所以要在空间还未满时就要启动 GC。mark 和 sweep 操作会引起很多碎片，所以间隔一段时间需要整理整个空间，否则遇到大对象，没有连续空间也会启动一次串行 GC。采用此收集器（如 tenured generation），收集频率不能大，否则会影响到 cpu 的利用率，进而影响吞吐量。。

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是的引用数来判断。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的地方，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它对象的引用，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，相应应用暂停的时间也会相应的增大**。一些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，以后研究的重点！！！！！！。

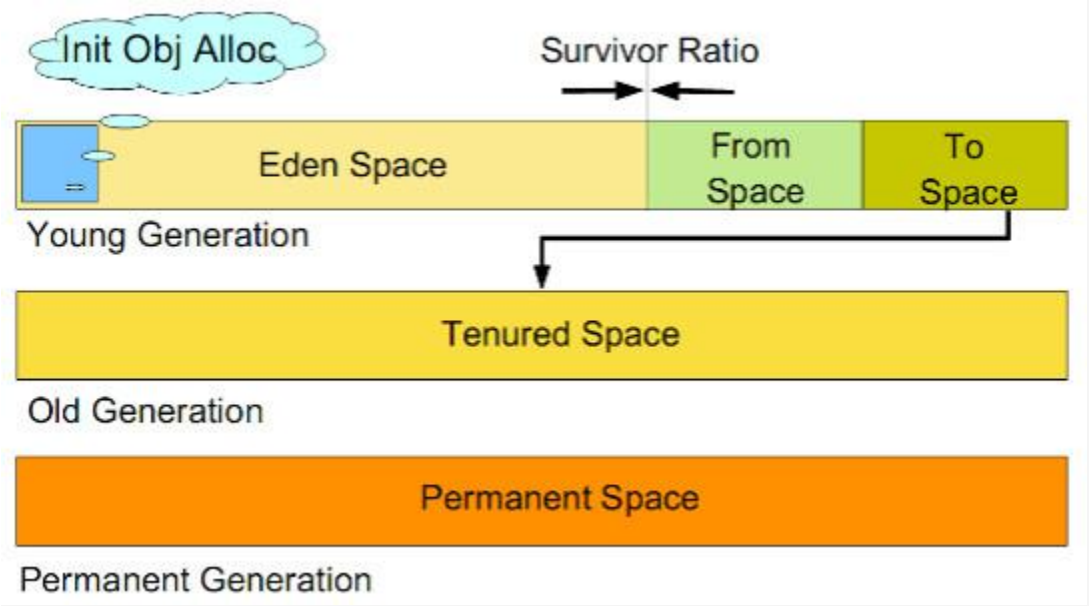
为什么要分代

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，不同声明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。

是想，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，但是它们依旧存在。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：

虚拟机中共划分了三个代：**年轻代（Young Generation）、年老代（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**分为三个区。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，此区的存活将被复制到另外一个 Survivor 区，当这个 Survivor 区也满

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 复制过来的对象和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 区过来的对象。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 **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 区能尽快空闲出来。

Full GC

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个堆进行回收，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 Full GC 的调节。

有如下原因可能导致 Full GC:

. **年老代**（Tenured）被写满

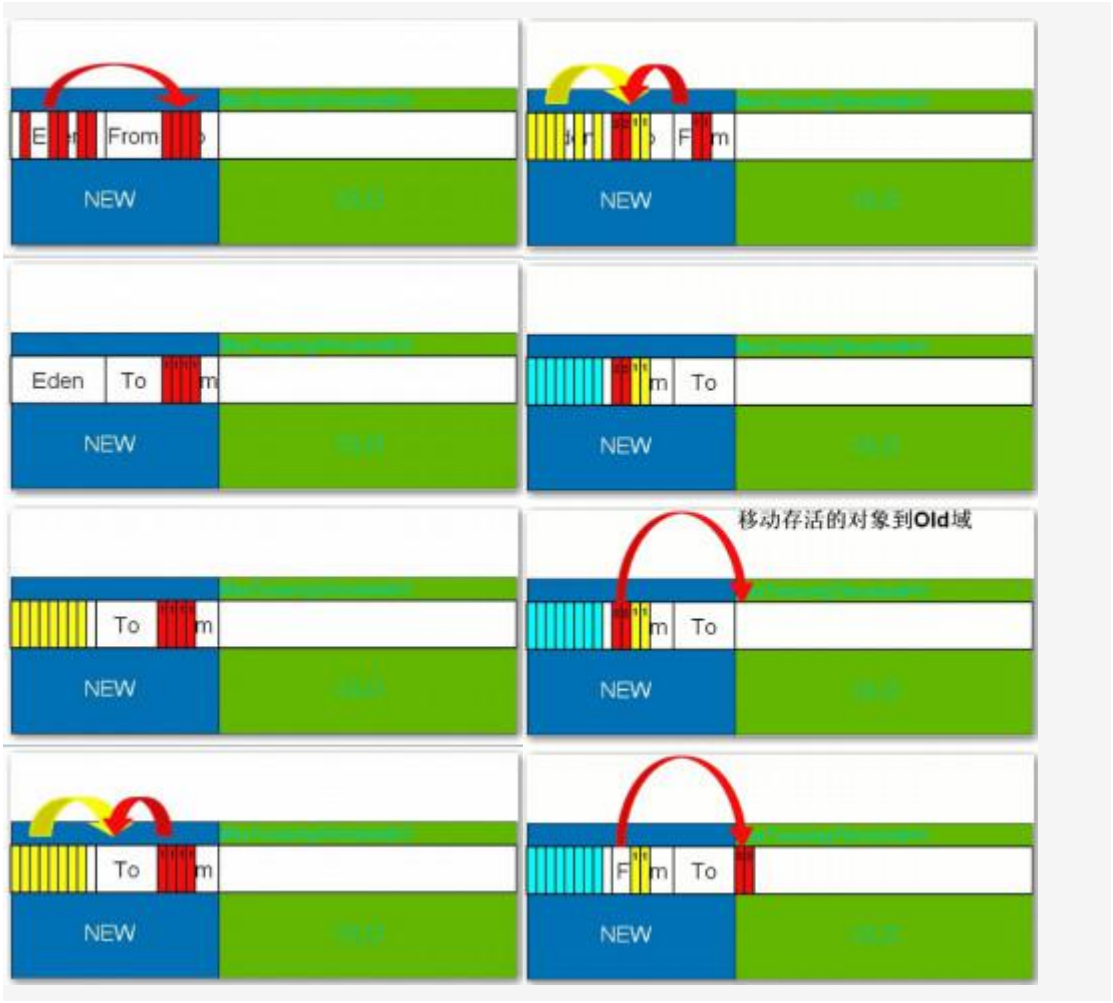
. 持久代（Perm）被写满

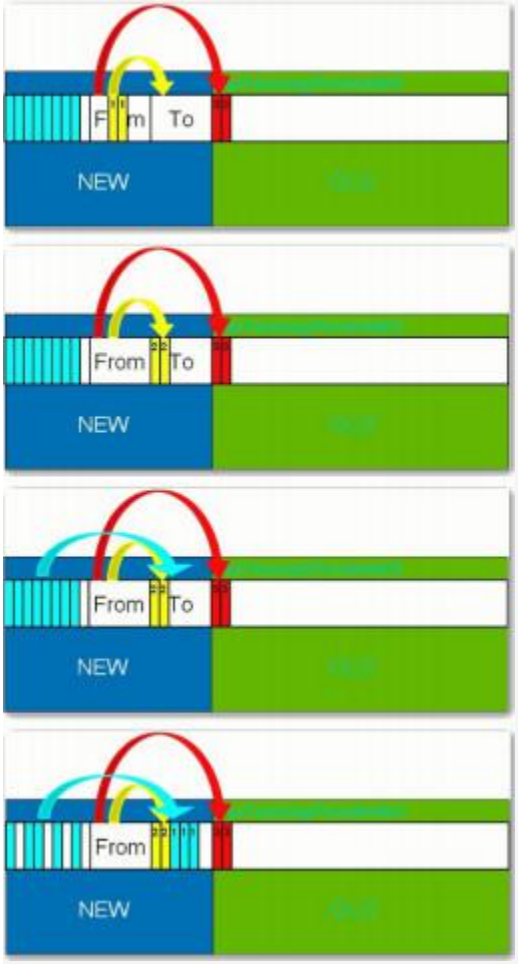
. System.gc()被显式调用

. 上一次 GC 之后 Heap 的各域分配策略动态变化

分代垃圾回收流程示意

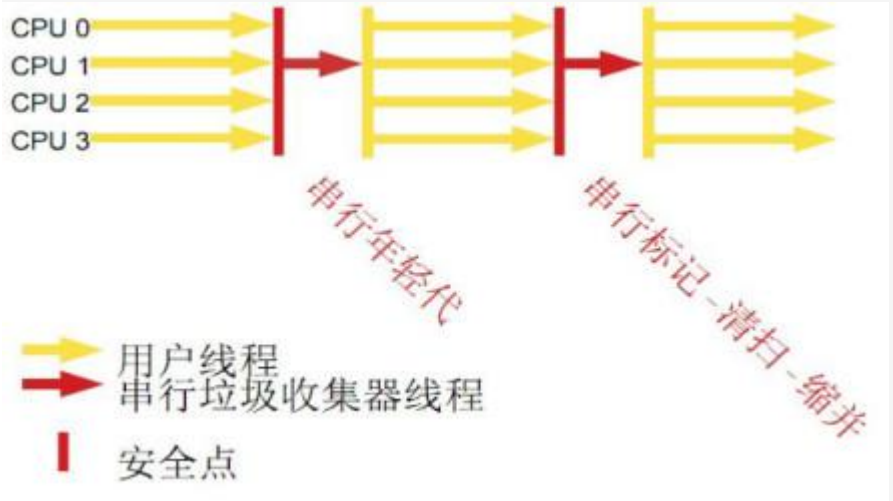
Java 架构学习群：895244712





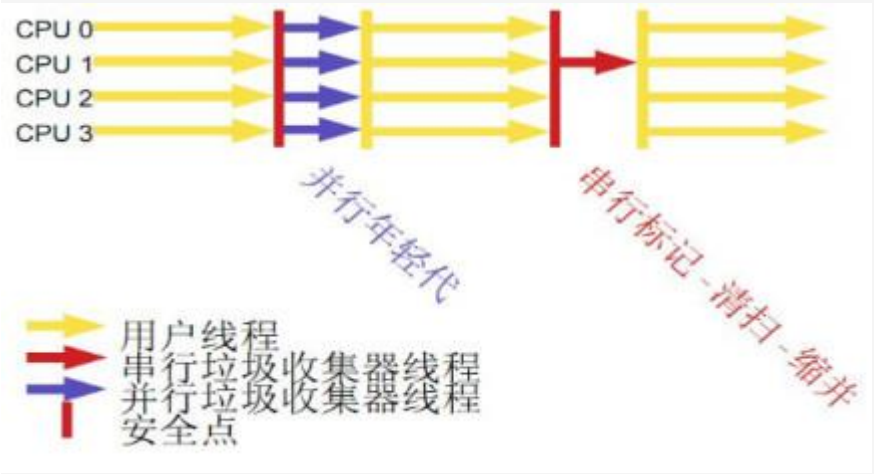
选择合适的垃圾收集算法

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，所有效率比较高。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**不使用并行收集的话，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。

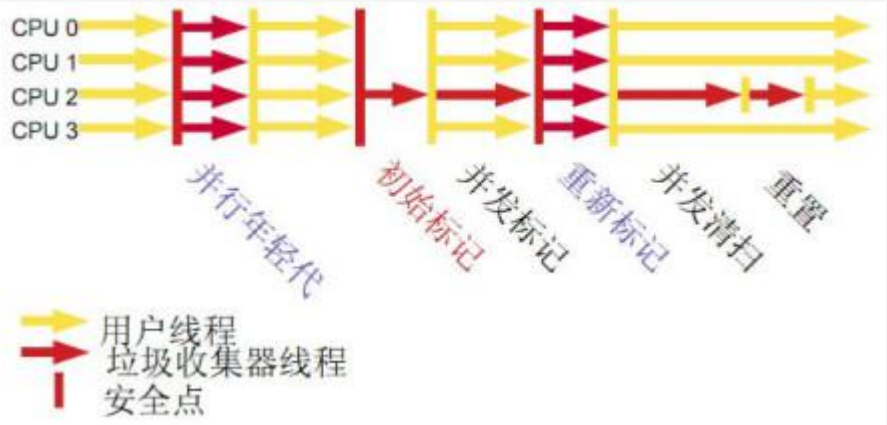
此收集器可以进行如下配置：

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，它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是开始执行并发收集。

小结

串行处理器：

-- 适用情况：数据量比较小（100M 左右），单处理器下并且对相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，对应用过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。

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-bit 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 位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 初始内存为 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。根据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，这样可以增加对象在**年轻代**的存活时间，增加在**年轻代**被回收的概率。

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的时候加入相应参数。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，适用于科学计算和后台处理等。

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。

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响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 -Xms3550 -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 -Xms3550 -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。

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，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。

JVM 会根据机器的硬件配置对每个内存代选择适合的回收算法，比如，如果机器多于 1 个核，会对年轻代选择并行算法，关于选择细节请参考 JVM 调优文档。

稍微解释下的是，并行算法是用多线程进行垃圾回收，回收期间会暂停程序的执行，而并发算法，也是多线程回收，但期间不停止应用执行。所以，并发算法适用于交互性高的一些程序。经过观察，并发算法会减少年轻代的大小，其实就是使用了一个大的年老代，这反过来跟并行算法相比吞吐量相对较低。

还有一个问题是，垃圾回收动作何时执行？

- 当年轻代内存满时，会引发一次普通 GC，该 GC 仅回收年轻代。需要强调的时，年轻代满是指 Eden 代满，Survivor 满不会引发 GC
- 当年老代满时会引发 Full GC，Full GC 将会同时回收年轻代、年老代
- 当永久代满时也会引发 Full GC，会导致 Class、Method 元信息的卸载

另一个问题是，何时会抛出 OutOfMemoryException，并不是内存被耗空的时候才抛出

- JVM98%的时间都花费在内存回收
- 每次回收的内存小于 2%

满足这两个条件将触发 OutOfMemoryException，这将会留给系统一个微小的间隙以做一些 Down 之前的操作，比如手动打印 Heap Dump。

二、内存泄漏及解决方法

1.系统崩溃前的一些现象：

- 每次垃圾回收的时间越来越长，由之前的 10ms 延长到 50ms 左右，FullGC 的时间也有之前的 0.5s 延长到 4、5s
- FullGC 的次数越来越多，最频繁时隔不到 1 分钟就进行一次 FullGC
- 年老代的内存越来越大并且每次 FullGC 后年老代没有内存被释放

之后系统会无法响应新的请求，逐渐到达 OutOfMemoryError 的临界值。

2.生成堆的 dump 文件

通过 JMX 的 MBean 生成当前的 Heap 信息，大小为一个 3G（整个堆的大小）的 hprof 文件，如果没有启动 JMX 可以通过 Java 的 jmap 命令来生成该文件。

3.分析 dump 文件

下面要考虑的是如何打开这个 3G 的堆信息文件，显然一般的 Window 系统没有这么大的内存，必须借助高配置的 **Linux**。当然我们可以借助 X-Window 把 Linux 上的图形导入到 Window。我们考虑用下面几种工具打开该文件：

1. Visual VM
2. IBM HeapAnalyzer
3. JDK 自带的 Hprof 工具

使用这些工具时为了确保加载速度，建议设置最大内存为 6G。使用后发现，这些工具都无法直观地观察到内存泄漏，Visual VM 虽能观察到对象大小，但看不到调用堆栈；HeapAnalyzer 虽然能看到调用堆栈，却无法正确打开一个 3G 的文件。因此，我们又选用了 Eclipse 专门的静态内存分析工具：Mat。

4.分析内存泄漏

通过 Mat 我们能清楚地看到，哪些对象被怀疑为内存泄漏，哪些对象占的空间最大及对象的调用关系。针对本案，在 ThreadLocal 中有很多的 JbpmContext 实例，经过调查是 JBPM 的 Context 没有关闭所致。

另，通过 Mat 或 JMX 我们还可以分析线程状态，可以观察到线程被阻塞在哪个对象上，从而判断系统的瓶颈。

5.回归问题

Q：为什么崩溃前垃圾回收的时间越来越长？

A:根据内存模型和垃圾回收算法，垃圾回收分两部分：内存标记、清除（复制），标记部分只要内存大小固定时间是不变的，变的是复制部分，因为每次垃圾回收都有一些回收不掉的内存，所以增加了复制量，导致时间延长。所以，垃圾回收的时间也可以作为判断内存泄漏的依据

Q：为什么 Full GC 的次数越来越多？

A：因此内存的积累，逐渐耗尽了年老代的内存，导致新对象分配没有更多的空间，从而导致频繁的垃圾回收

Q:为什么年老代占用的内存越来越大？

A:因为年轻代的内存无法被回收，越来越多地被 Copy 到年老代

三、性能调优

除了上述内存泄漏外，我们还发现 CPU 长期不足 3%，系统吞吐量不够，针对 8core×16G、64bit 的 Linux 服务器来说，是严重的资源浪费。

在 CPU 负载不足的同时，偶尔会有用户反映请求的时间过长，我们意识到必须对程序及 JVM 进行调优。从以下几个方面进行：

- 线程池：解决用户响应时间长的问题
- 连接池
- JVM 启动参数：调整各代的内存比例和垃圾回收算法，提高吞吐量
- 程序算法：改进程序逻辑算法提高性能

1.Java 线程池（java.util.concurrent.ThreadPoolExecutor）

大多数 JVM6 上的应用采用的线程池都是 JDK 自带的线程池，之所以把成熟的 Java 线程池进行罗嗦说明，是因为该线程池的行为与我们想象的有点出入。Java 线程池有几个重要的配置参数：

- corePoolSize：核心线程数（最新线程数）
- maximumPoolSize：最大线程数，超过这个数量的任务会被拒绝，用户可以通过 RejectedExecutionHandler 接口自定义处理方式
- keepAliveTime：线程保持活动的时间
- workQueue：工作队列，存放执行的任务

Java 线程池需要传入一个 Queue 参数（workQueue）用来存放执行的任务，而对 Queue 的不同选择，线程池有完全不同的行为：

- SynchronousQueue： 一个无容量的等待队列，一个线程的 insert 操作必须等待另一线程的 remove 操作，采用这个 Queue 线程池将会为每个任务分配一个新线程
- LinkedBlockingQueue ： 无界队列，采用该 Queue，线程池将忽略 maximumPoolSize 参数，仅用 corePoolSize 的线程处理所有的任务，未处理的任务便在 LinkedBlockingQueue 中排队
- ArrayBlockingQueue： 有界队列，在有界队列和 maximumPoolSize 的作用下，程序将很难被调优：更大的 Queue 和小的 maximumPoolSize 将导致 CPU 的低负载；小的 Queue 和大的池，Queue 就没起动应有的作用。

其实我们的要求很简单，希望线程池能跟连接池一样，能设置最小线程数、最大线程数，当最小数<任务<最大数时，应该分配新的线程处理；当任务>最大数时，应该等待有空闲线程再处理该任务。

但线程池的设计思路是，任务应该放到 Queue 中，当 Queue 放不下时再考虑用新线程处理，如果 Queue 满且无法派生新线程，就拒绝该任务。设计导致“先放等执行”、“放不下再执行”、“拒绝不等待”。所以，根据不同的 Queue 参数，要提高吞吐量不能一味地增大 maximumPoolSize。

当然，要达到我们的目标，必须对线程池进行一定的封装，幸运的是 ThreadPoolExecutor 中留了足够的自定义接口以帮助我们达到目标。我们封装的方式是：

- 以 SynchronousQueue 作为参数，使 maximumPoolSize 发挥作用，以防止线程被无限制的分配，同时可以通过提高 maximumPoolSize 来提高系统吞吐量
- 自定义一个 RejectedExecutionHandler，当线程数超过 maximumPoolSize 时进行处理，处理方式为隔一段时间检查线程池是否可以执行新 Task，如果可以把拒绝的 Task 重新放入到线程池，检查的时间依赖 keepAliveTime 的大小。

2.连接池（org.apache.commons.dbcp.BasicDataSource）

在使用 org.apache.commons.dbcp.BasicDataSource 的时候，因为之前采用了默认配置，所以当访问量大时，通过 JMX 观察到很多 Tomcat 线程都阻塞在 BasicDataSource 使用的 Apache ObjectPool 的锁上，直接原因当时是因为 BasicDataSource 连接池的最大连接数设置的太小，默认的 BasicDataSource 配置，仅使用 8 个最大连接。

我还观察到一个问题，当较长的时间不访问系统，比如 2 天，DB 上的 **MySQL** 会断掉所以的连接，导致连接池中缓存的连接不能用。为了解决这些问题，我们充分研究了 BasicDataSource，发现了一些优化的点：

- Mysql 默认支持 100 个链接，所以每个连接池的配置要根据集群中的机器数进行，如有 2 台服务器，可每个设置为 60
- initialSize：参数是一直打开的连接数
- minEvictableIdleTimeMillis：该参数设置每个连接的空闲时间，超过这个时间连接将被关闭
- timeBetweenEvictionRunsMillis：后台线程的运行周期，用来检测过期连接
- maxActive：最大能分配的连接数
- maxIdle：最大空闲数，当连接使用完毕后发现连接数大于 maxIdle，连接将被直接关闭。只有 initialSize < x < maxIdle 的连接将被定期检测是否超期。这个参数主要用来在峰值访问时提高吞吐量。
- initialSize 是如何保持的？经过研究代码发现，BasicDataSource 会关闭所有超期的连接，然后再打开 initialSize 数量的连接，这个特性与 minEvictableIdleTimeMillis、timeBetweenEvictionRunsMillis 一起保证了所有超期的 initialSize 连接都会被重新连接，从而避免了 Mysql 长时间无动作会断掉连接的问题。

3.JVM 参数

在 JVM 启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置 JVM 会工作的很好，但对一些配置很好的 Server 和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

- GC 的时间足够的小
- GC 的次数足够的少
- 发生 Full GC 的周期足够的长

前两个目前是相悖的，要想 GC 时间小必须要一个更小的堆，要保证 GC 次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对 JVM 堆的设置一般，可以通过-Xms -Xmx 限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值

（2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的

比率 NewRadio 来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize 来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize 设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

- 更大的年轻代必然导致更小的年老代，大的年轻代会延长普通 GC 的周期，但会增加每次 GC 的时间；小的年老代会导致更频繁的 Full GC
- 更小的年轻代必然导致更大年老代，小的年轻代会导致普通 GC 很频繁，但每次的 GC 时间会更短；大的年老代会减少 Full GC 的频率
- 如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着 Full GC 尽量少的原则，让年老代尽量缓存常用对象，JVM 的默认比例 1：2 也是这个道理（B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响 Full GC 的前提下，根据实际情况加大年轻代，比如可以把比例控制在 1：1。但应该给年老代至少预留 1/3 的增长空间

（4）在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法： **-XX:+UseParallelOldGC** ，默认为 Serial 收集

（5）线程堆栈的设置：每个线程默认会开启 1M 的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般 256K 就足用。理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

（4）可以通过下面的参数打 Heap Dump 信息

- -XX:HeapDumpPath
- -XX:+PrintGCDetails
- -XX:+PrintGCTimeStamps
- -Xloggc:/usr/aaa/dump/heap_trace.txt

通过下面参数可以控制 OutOfMemoryError 时打印堆的信息

- -XX:+HeapDumpOnOutOfMemoryError

请看一下一个时间的 Java 参数配置：（服务器：Linux 64Bit，8Core×16G）

JAVA_OPTS="$JAVA_OPTS -server -Xms3G -Xmx3G -Xss256k -XX:PermSize=128m -XX:MaxPermSize=128m -XX:+UseParallelOldGC -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/aaa/dump -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xloggc:/usr/aaa/dump/heap_trace.txt -XX:NewSize=1G -XX:MaxNewSize=1G"

经过观察该配置非常稳定，每次普通 GC 的时间在 10ms 左右，Full GC 基本不发生，或隔很长很长的时间才发生一次

通过分析 dump 文件可以发现，每个 1 小时都会发生一次 Full GC，经过多方求证，只要在 JVM 中开启了 JMX 服务，JMX 将会 1 小时执行一次 Full GC 以清除引用，关于这点请参考附件文档。

4.程序算法调优：本次不作为重点

调优方法

一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1、多数的 Java 应用不需要在服务器上进行 GC 优化；
2、多数导致 GC 问题的 Java 应用，都不是因为我们参数设置错误，而是代码问题；
3、在应用上线之前，先考虑将机器的 JVM 参数设置到最优（最适合）；
4、减少创建对象的数量；
5、减少使用全局变量和大对象；
6、GC 优化是到最后不得已才采用的手段；
7、在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多；

GC 优化的目的有两个（http://www.360doc.com/content/13/0305/10/15643_269388816.shtml）：

1、将转移到老年代的对象数量降低到最小；
2、减少 full GC 的执行时间；

为了达到上面的目的，一般地，你需要做的事情有：
1、减少使用全局变量和大对象；
2、调整新生代的大小到最合适；
3、设置老年代的大小为最合适；
4、选择合适的 GC 收集器；

在上面的 4 条方法中，用了几个“合适”，那究竟什么才算合适，一般的，请参考上面“收集器搭配”和“启动内存分配”两节中的建议。但这些建议不是万能的，需要根据您的机器和应用情况进行发展和变化，实际操作中，可以将两台机器分别设置成不同的 GC 参数，并且进行对比，选用那些确实提高了性能或减少了 GC 时间的参数。

真正熟练的使用 GC 调优，是建立在多次进行 GC 监控和调优的实战经验上的，进行监控和调优的一般步骤为：

1，监控 GC 的状态

使用各种 JVM 工具，查看当前日志，分析当前 JVM 参数设置，并且分析当前堆内存快照和 gc 日志，根据实际的各区域内存划分和 GC 执行时间，觉得是否进行优化；

2，分析结果，判断是否需要优化

如果各项参数设置合理，系统没有超时日志出现，GC 频率不高，GC 耗时不高，那么没有必要进行 GC 优化；如果 GC 时间超过 1-3 秒，或者频繁 GC，则必须优化；

注：**如果满足下面的指标，则一般不需要进行 GC：**

Minor GC 执行时间不到 50ms；

Minor GC 执行不频繁，约 10 秒一次；

Full GC 执行时间不到 1s；

Full GC 执行频率不算频繁，不低于 10 分钟 1 次；

3，调整 GC 类型和内存分配

如果内存分配过大或过小，或者采用的 GC 收集器比较慢，则应该优先调整这些参数，并且先找 1 台或几台机器进行 beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择；

4，不断的分析和调整

通过不断的试验和试错，分析并找到最合适的参数

5，全面应用参数

如果找到了最合适的参数，则将这些参数应用到所有服务器，并进行后续跟踪。

调优实例

上面的内容都是纸上谈兵，下面我们以一些真实例子来进行说明：

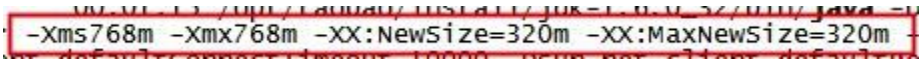
实例 1：

笔者昨日发现部分开发测试机器出现异常：java.lang.OutOfMemoryError: GC overhead limit exceeded，这个异常代表：

GC 为了释放很小的空间却耗费了太多的时间，其原因一般有两个：1，堆太小，2，有死循环或大对象；

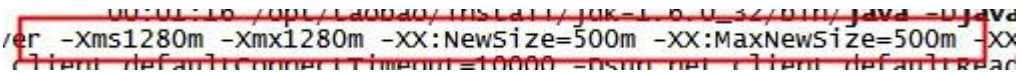
笔者首先排除了第 2 个原因，因为这个应用同时是在线上运行的，如果有问题，早就挂了。所以怀疑是这台机器中堆设置太小；

使用 ps -ef |grep "java"查看，发现：



该应用的堆区设置只有 768m，而机器内存有 2g，机器上只跑这一个 java 应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多；

笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，于是改成下面的情况：

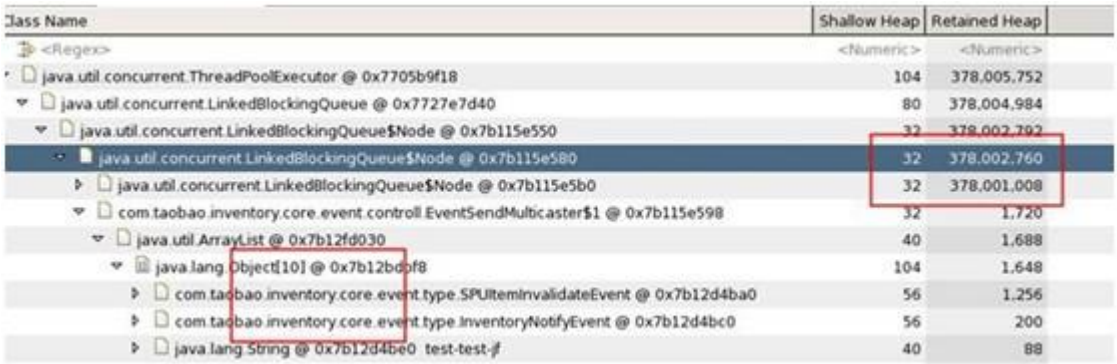


跟踪运行情况发现，相关异常没有再出现；

实例 2：（http://www.360doc.com/content/13/0305/10/15643_269388816.shtml）

一个服务系统，**经常出现卡顿，分析原因，发现 Full GC 时间太长：**
jstat -gcutil:
S0 S1 E O P YGC YGCT FGC FGCT GCT
12.16 0.00 5.18 63.78 20.32 54 2.047 5 6.946 8.993
分析上面的数据，发现 Young GC 执行了 54 次，耗时 2.047 秒，每次 Young GC 耗时 37ms，在正常范围，**而 Full GC 执行了 5 次，耗时 6.946 秒，每次平均 1.389s，数据显示出来的问题是：Full GC 耗时较长**，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为 1:9，这就是问题的原因：
1，新生代太小，导致对象提前进入老年代，触发老年代发生 Full GC；
2，老年代较大，进行 Full GC 时耗时较大；
优化的方法是调整 NewRatio 的值，调整到 4，发现 Full GC 没有再发生，只有 Young GC 在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是很有用的，但并不是对所有应用都要这么做）

实例 3：
一应用在性能测试过程中，发现内存占用率很高，Full GC 频繁，使用 sudo -u admin -H jmap -dump:format=b,file=文件名.hprof pid 来 dump 内存，生成 dump 文件，并使用 Eclipse 下的 mat 差距进行分析，发现：



从图中可以看出，这个线程存在问题，队列 LinkedBlockingQueue 所引用的大量对象并未释放，导致整个线程占用内存高达 378m，此时通知开发人员进行代码优化，将相关对象释放掉即可。