[**Java 内存区域和GC机制**](http://www.cnblogs.com/hnrainll/p/3410042.html)

目录

1. [Java垃圾回收概况](http://www.cnblogs.com/zhguang/p/3257367.html#introduction)
2. [Java内存区域](http://www.cnblogs.com/zhguang/p/3257367.html#memoryArea)
3. [Java对象的访问方式](http://www.cnblogs.com/zhguang/p/3257367.html#access)
4. [Java内存分配机制](http://www.cnblogs.com/zhguang/p/3257367.html#distribution)
5. [Java GC机制](http://www.cnblogs.com/zhguang/p/3257367.html#GCScheama)
6. [垃圾收集器](http://www.cnblogs.com/zhguang/p/3257367.html#GCparam)

**Java垃圾回收概况**

　　Java GC（Garbage Collection，垃圾收集，垃圾回收）机制，是Java与C++/C的主要区别之一，作为Java开发者，一般不需要专门编写内存回收和垃圾清理代 码，对内存泄露和溢出的问题，也不需要像C程序员那样战战兢兢。这是因为在Java虚拟机中，存在自动内存管理和垃圾清扫机制。概括地说，该机制对 JVM（Java Virtual Machine）中的内存进行标记，并确定哪些内存需要回收，根据一定的回收策略，自动的回收内存，永不停息（Nerver Stop）的保证JVM中的内存空间，放置出现内存泄露和溢出问题。

　　关于JVM，需要说明一下的是，目前使用最多的Sun公司的JDK中，自从 1999年的JDK1.2开始直至现在仍在广泛使用的JDK6，其中默认的虚拟机都是HotSpot。2009年，Oracle收购Sun，加上之前收购 的EBA公司，Oracle拥有3大虚拟机中的两个：JRockit和HotSpot，Oracle也表明了想要整合两大虚拟机的意图，但是目前在新发布 的JDK7中，默认的虚拟机仍然是HotSpot，因此本文中默认介绍的虚拟机都是HotSpot，相关机制也主要是指HotSpot的GC机制。

　　Java GC机制主要完成3件事：确定哪些内存需要回收，确定什么时候需要执行GC，如何执行GC。经过这么长时间的发展（事实上，在Java语言出现之前，就有 GC机制的存在，如Lisp语言），Java GC机制已经日臻完善，几乎可以自动的为我们做绝大多数的事情。然而，如果我们从事较大型的应用软件开发，曾经出现过内存优化的需求，就必定要研究 Java GC机制。

　　学习Java GC机制，可以帮助我们在日常工作中排查各种内存溢出或泄露问题，解决性能瓶颈，达到更高的并发量，写出更高效的程序。

　　我们将从4个方面学习Java GC机制，1，内存是如何分配的；2，如何保证内存不被错误回收（即：哪些内存需要回收）；3，在什么情况下执行GC以及执行GC的方式；4，如何监控和优化GC机制。

**Java内存区域**

　　了解Java GC机制，必须先清楚在JVM中内存区域的划分。在Java运行时的数据区里，由JVM管理的内存区域分为下图几个模块：

其中：

**1，程序计数器（Program Counter Register）**：程序计数器是一个比较小的内存区域，用于指示当前线程所执行的字节码执行到了第几行，可以理解为是当前线程的行号指示器。字节码解释器在工作时，会通过改变这个计数器的值来取下一条语句指令。

　　每个程序计数器只用来记录一个线程的行号，所以它是**线程私有**（一个线程就有一个程序计数器）的。

　　如果程序执行的是一个Java方法，则计数器记录的是正在执行的虚拟机字节码指令地址；如果正在执行的是一个本地（native，由C语言编写 完成）方法，则计数器的值为Undefined，由于程序计数器只是记录当前指令地址，所以不存在内存溢出的情况，因此，程序计数器也是所有JVM内存区 域中唯一一个没有定义OutOfMemoryError的区域。

**2，虚拟机栈（JVM Stack）**：一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。

　　局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占 用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定 好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。

　　虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多 数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，知道内存不足，此时，会抛出 OutOfMemoryError（内存溢出）。

　　每个线程对应着一个虚拟机栈，因此虚拟机栈也是线程私有的。

**3，本地方法栈（Native Method Statck）**：本地方法栈在作用，运行机制，异常类型等方面都与虚拟机栈相同，唯一的区别是：虚拟机栈是执行Java方法的，而本地方法栈是用来执行native方法的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。

　　本地方法栈也是线程私有的。

**4，堆区（Heap）**：堆区是理解Java GC机制最重要的区域，没有之一。在JVM所管理的内存中，堆区是最大的一块，堆区也是Java GC机制所管理的主要内存区域，堆区**由所有线程共享**，在虚拟机启动时创建。堆区的存在是为了**存储对象实例**，原则上讲，所有的对象都在堆区上分配内存（不过现代技术里，也不是这么绝对的，也有栈上直接分配的）。

　　一般的，根据Java虚拟机规范规定，堆内存需要在逻辑上是连续的（在物理上不需要），在实现时，可以是固定大小的，也可以是可扩展的，目前主 流的虚拟机都是可扩展的。如果在执行垃圾回收之后，仍没有足够的内存分配，也不能再扩展，将会抛出OutOfMemoryError:Java heap space异常。

　　关于堆区的内容还有很多，将在下节“**Java内存分配机制**”中详细介绍。

**5，方法区（Method Area）**：在Java虚拟机规范中，将方法区作为堆的一个逻辑部分来对待，但事实 上，方法区并不是堆（Non-Heap）；另外，不少人的博客中，将Java GC的分代收集机制分为3个代：青年代，老年代，永久代，这些作者将方法区定义为“永久代”，这是因为，对于之前的HotSpot Java虚拟机的实现方式中，将分代收集的思想扩展到了方法区，并将方法区设计成了永久代。不过，除HotSpot之外的多数虚拟机，并不将方法区当做永 久代，HotSpot本身，也计划取消永久代。本文中，由于笔者主要使用Oracle JDK6.0，因此仍将使用永久代一词。

　　方法区是各个线程共享的区域，用于存储已经被虚拟机加载的类信息（即加载类时需要加载的信息，包括版本、field、方法、接口等信息）、final常量、静态变量、编译器即时编译的代码等。

　　方法区在物理上也不需要是连续的，可以选择固定大小或可扩展大小，并且方法区比堆还多了一个限制：可以选择是否执行垃圾收集。一般的，方法区上 执行的垃圾收集是很少的，这也是方法区被称为永久代的原因之一（HotSpot），但这也不代表着在方法区上完全没有垃圾收集，其上的垃圾收集主要是针对 常量池的内存回收和对已加载类的卸载。

　　在方法区上进行垃圾收集，条件苛刻而且相当困难，效果也不令人满意，所以一般不做太多考虑，可以留作以后进一步深入研究时使用。

　　在方法区上定义了OutOfMemoryError:PermGen space异常，在内存不足时抛出。

　　运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存储编译期就生成的字面常量、符号引用、翻译出来的直接引用（符号引用就是编码是用字符串表示某个变量、接口的位置，直接引用就是根据符号引用翻译出来的地址，将在类链接阶段完成翻译）；运行时常量池除了存储编译期常量外，也可以存储在运行时间产生的常量（比如String类的intern()方法，作用是String维护了一个常量池，如果调用的字符“abc”已经在常量池中，则返回池中的字符串地址，否则，新建一个常量加入池中，并返回地址）。

**6，直接内存（Direct Memory）**：直接内存并不是JVM管理的内存，可以这样理解，直接内存，就是 JVM以外的机器内存，比如，你有4G的内存，JVM占用了1G，则其余的3G就是直接内存，JDK中有一种基于通道（Channel）和缓冲区 （Buffer）的内存分配方式，将由C语言实现的native函数库分配在直接内存中，用存储在JVM堆中的DirectByteBuffer来引用。 由于直接内存收到本机器内存的限制，所以也可能出现OutOfMemoryError的异常。

**Java对象的访问方式**

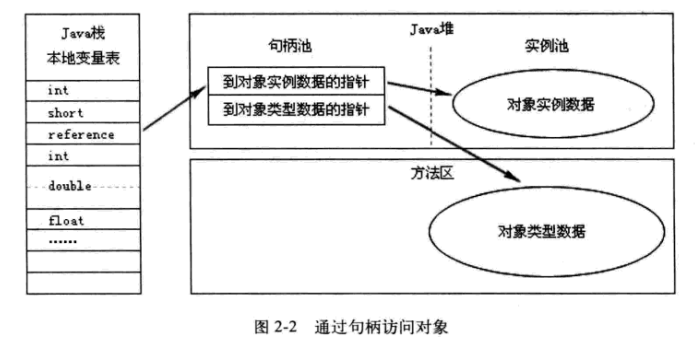
一般来说，一个Java的引用访问涉及到3个内存区域：JVM栈，堆，方法区。

　　以最简单的本地变量引用：Object obj = new Object()为例：

* Object obj表示一个本地引用，存储在JVM栈的本地变量表中，表示一个reference类型数据；
* new Object()作为实例对象数据存储在堆中；
* 堆中还记录了Object类的类型信息（接口、方法、field、对象类型等）的地址，这些地址所执行的数据存储在方法区中；

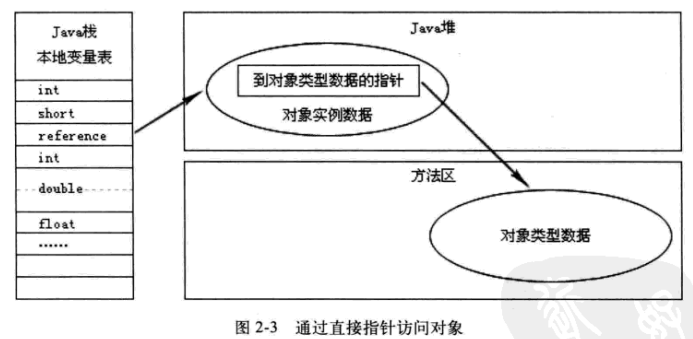
在Java虚拟机规范中，对于通过reference类型引用访问具体对象的方式并未做规定，目前主流的实现方式主要有两种：

1，通过句柄访问（图来自于《深入理解Java虚拟机：JVM高级特效与最佳实现》）：



通过句柄访问的实现方式中，JVM堆中会专门有一块区域用来作为句柄池，存储相关句柄所执行的实例数据地址（包括在堆中地址和在方法区中的地址）。这种实现方法由于用句柄表示地址，因此十分稳定。

2，通过直接指针访问：（图来自于《深入理解Java虚拟机：JVM高级特效与最佳实现》）

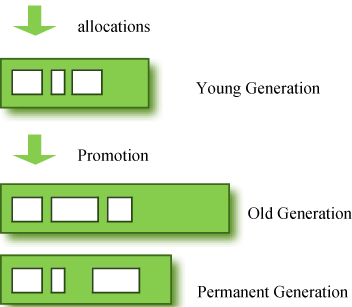


通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

**Java内存分配机制**

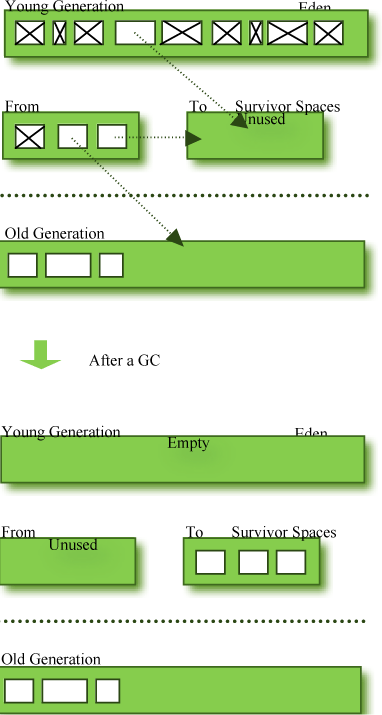
这里所说的内存分配，主要指的是在堆上的分配，一般的，对象的内存分配都是在堆上进行，但现代技术也支持将对象拆成标量类型（标量类型即原子类型，表示单个值，可以是基本类型或String等），然后在栈上分配，在栈上分配的很少见，我们这里不考虑。

　　Java内存分配和回收的机制概括的说，就是：分代分配，分代回收。对象将根据存活的时间被分为：年轻代（Young Generation）、年老代（Old Generation）、永久代（Permanent Generation，也就是方法区）。如下图（来源于《成为JavaGC专家part I》，http://www.importnew.com/1993.html）：



**年轻代（Young Generation）**：对象被创建时，内存的分配首先发生在年轻代（大对象可以直接 被创建在年老代），大部分的对象在创建后很快就不再使用，因此很快变得不可达，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消 亡的），这个GC机制被称为Minor GC或叫Young GC。注意，Minor GC并不代表年轻代内存不足，它事实上只表示在Eden区上的GC。

　　年轻代上的内存分配是这样的，年轻代可以分为3个区域：Eden区（伊甸园，亚当和夏娃偷吃禁果生娃娃的地方，用来表示内存首次分配的区域，再 贴切不过）和两个存活区（Survivor 0 、Survivor 1）。内存分配过程为（来源于《成为JavaGC专家part I》，http://www.importnew.com/1993.html）：



1. 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
2. 当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
3. 此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；
4. 当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。
5. 当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

　　从上面的过程可以看出，Eden区是连续的空间，且Survivor总有一个为空。经过一次GC和复制，一个Survivor中保存着当前还活 着的对象，而Eden区和另一个Survivor区的内容都不再需要了，可以直接清空，到下一次GC时，两个Survivor的角色再互换。因此，这种方 式分配内存和清理内存的效率都极高，这种垃圾回收的方式就是著名的**“停止-复制（Stop-and-copy）”清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中）**，这不代表着停止复制清理法很高效，其实，它也只在这种情况下高效，如果在老年代采用停止复制，则挺悲剧的。

　　在Eden区，HotSpot虚拟机使用了两种技术来加快内存分配。分别是bump-the-pointer和TLAB（Thread- Local Allocation Buffers），这两种技术的做法分别是：由于Eden区是连续的，因此bump-the-pointer技术的核心就是跟踪最后创建的一个对象，在对 象创建时，只需要检查最后一个对象后面是否有足够的内存即可，从而大大加快内存分配速度；而对于TLAB技术是对于多线程而言的，将Eden区分为若干 段，每个线程使用独立的一段，避免相互影响。TLAB结合bump-the-pointer技术，将保证每个线程都使用Eden区的一段，并快速的分配内 存。

**年老代（Old Generation）**：对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次 Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时， 将执行Major GC，也叫 Full GC。

　　可以使用-XX:+UseAdaptiveSizePolicy开关来控制是否采用动态控制策略，如果动态控制，则动态调整Java堆中各个区域的大小以及进入老年代的年龄。

　　如果对象比较大（比如长字符串或大数组），Young空间不足，则大对象会直接分配到老年代上（大对象可能触发提前GC，应少用，更应避免使用短命的大对象）。用-XX:PretenureSizeThreshold来控制直接升入老年代的对象大小，大于这个值的对象会直接分配在老年代上。

　　可能存在年老代对象引用新生代对象的情况，如果需要执行Young GC，则可能需要查询整个老年代以确定是否可以清理回收，这显然是低效的。解决的方法是，年老代中维护一个512 byte的块——”card table“，所有老年代对象引用新生代对象的记录都记录在这里。Young GC时，只要查这里即可，不用再去查全部老年代，因此性能大大提高。

**Java GC机制**

GC机制的基本算法是：分代收集，这个不用赘述。下面阐述每个分代的收集方法。

**年轻代：**

　　事实上，在上一节，已经介绍了新生代的主要垃圾回收方法，在新生代中，使用“停止-复制”算法进行清理，将新生代内存分为2部分，1部分 Eden区较大，1部分Survivor比较小，并被划分为两个等量的部分。每次进行清理时，将Eden区和一个Survivor中仍然存活的对象拷贝到 另一个Survivor中，然后清理掉Eden和刚才的Survivor。

　　这里也可以发现，停止复制算法中，用来复制的两部分并不总是相等的（传统的停止复制算法两部分内存相等，但新生代中使用1个大的Eden区和2个小的Survivor区来避免这个问题）

　　由于绝大部分的对象都是短命的，甚至存活不到Survivor中，所以，Eden区与Survivor的比例较大，HotSpot默认是 8:1，即分别占新生代的80%，10%，10%。如果一次回收中，Survivor+Eden中存活下来的内存超过了10%，则需要将一部分对象分配到 老年代。用-XX:SurvivorRatio参数来配置Eden区域Survivor区的容量比值，默认是8，代表Eden：Survivor1：Survivor2=8:1:1.

**老年代：**

　　老年代存储的对象比年轻代多得多，而且不乏大对象，对老年代进行内存清理时，如果使用停止-复制算法，则相当低效。一般，老年代用的算法是标记-整理算法，即：标记出仍然存活的对象（存在引用的），将所有存活的对象向一端移动，以保证内存的连续。

在发生Minor GC时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次Full GC，否则，就查看是否设 置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行MinorGC，此时可以容忍内存分配失败；如果不 允许，则仍然进行Full GC（这代表着如果设置-XX:+Handle PromotionFailure，则触发MinorGC就会同时触发Full GC，哪怕老年代还有很多内存，所以，最好不要这样做）。

**方法区（永久代）：**

　　永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的所有实例都已经被回收
2. 加载类的ClassLoader已经被回收
3. 类对象的Class对象没有被引用（即没有通过反射引用该类的地方）

永久代的回收并不是必须的，可以通过参数来设置是否对类进行回收。HotSpot提供-Xnoclassgc进行控制

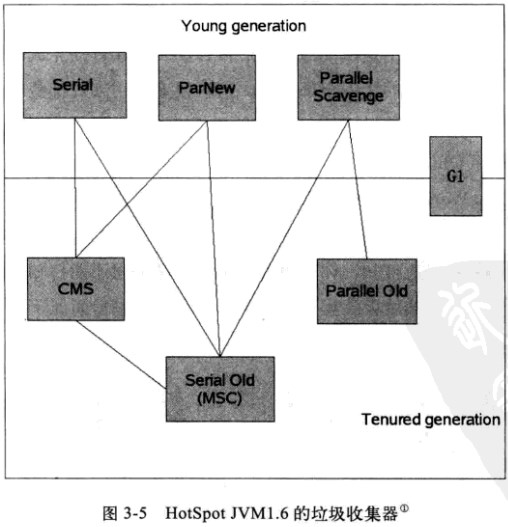
使用-verbose，-XX:+TraceClassLoading、-XX:+TraceClassUnLoading可以查看类加载和卸载信息

-verbose、-XX:+TraceClassLoading可以在Product版HotSpot中使用；

-XX:+TraceClassUnLoading需要fastdebug版HotSpot支持

**垃圾收集器**

在GC机制中，起重要作用的是垃圾收集器，垃圾收集器是GC的具体实现，Java虚拟机规范中对于垃圾收集器没有任何规定，所以不同厂商实现的垃圾 收集器各不相同，HotSpot 1.6版使用的垃圾收集器如下图（图来源于《深入理解Java虚拟机：JVM高级特效与最佳实现》，图中两个收集器之间有连线，说明它们可以配合使用）：



在介绍垃圾收集器之前，需要明确一点，就是在新生代采用的停止复制算法中，“停 止（Stop-the-world）”的意义是在回收内存时，需要暂停其他所 有线程的执行。这个是很低效的，现在的各种新生代收集器越来越优化这一点，但仍然只是将停止的时间变短，并未彻底取消停止。

* Serial收集器：新生代收集器，使用停止复制算法，使用一个线程进行GC，其它工作线程暂停。使用-XX:+UseSerialGC可以使用Serial+Serial Old模式运行进行内存回收（这也是虚拟机在Client模式下运行的默认值）
* ParNew收集器：新生代收集器，使用停止复制算法，Serial收集器的多线程版，用多个线程进行GC，其它工作线程暂停，关注缩短垃圾收集时间。使用-XX:+UseParNewGC开关来控制使用ParNew+Serial Old收集器组合收集内存；使用-XX:ParallelGCThreads来设置执行内存回收的线程数。
* Parallel Scavenge 收集器：新生代收集器，使用停止复制算法，关注CPU吞吐量，即运行用户代码的时间/总时间，比如：JVM运行100分钟，其中运行用户代码99分钟，垃 圾收集1分钟，则吞吐量是99%，这种收集器能最高效率的利用CPU，适合运行后台运算（关注缩短垃圾收集时间的收集器，如CMS，等待时间很少，所以适 合用户交互，提高用户体验）。使用-XX:+UseParallelGC开关控制使用 Parallel Scavenge+Serial Old收集器组合回收垃圾（这也是在Server模式下的默认值）；使用-XX:GCTimeRatio来设置用户执行时间占总时间的比例，默认99，即 1%的时间用来进行垃圾回收。使用-XX:MaxGCPauseMillis设置GC的最大停顿时间（这个参数只对Parallel Scavenge有效）
* Serial Old收集器：老年代收集器，单线程收集器，使用标记整理（整理的方法是Sweep（清理）和Compact（压缩），清理是将废弃的对象干掉，只留幸存 的对象，压缩是将移动对象，将空间填满保证内存分为2块，一块全是对象，一块空闲）算法，使用单线程进行GC，其它工作线程暂停（注意，在老年代中进行标 记整理算法清理，也需要暂停其它线程），在JDK1.5之前，Serial Old收集器与ParallelScavenge搭配使用。
* Parallel Old收集器：老年代收集器，多线程，多线程机制与Parallel Scavenge差不错，使用标记整理（与Serial Old不同，这里的整理是Summary（汇总）和Compact（压缩），汇总的意思就是将幸存的对象复制到预先准备好的区域，而不是像Sweep（清 理）那样清理废弃的对象）算法，在Parallel Old执行时，仍然需要暂停其它线程。Parallel Old在多核计算中很有用。Parallel Old出现后（JDK 1.6），与Parallel Scavenge配合有很好的效果，充分体现Parallel Scavenge收集器吞吐量优先的效果。使用-XX:+UseParallelOldGC开关控制使用Parallel Scavenge +Parallel Old组合收集器进行收集。
* CMS（Concurrent Mark Sweep）收集器：老年代收集器，致力于获取最短回收停顿时间，使用标记清除算法，多线程，优点是并发收集（用户线程可以和GC线程同时工作），停顿小。使用-XX:+UseConcMarkSweepGC进行ParNew+CMS+Serial Old进行内存回收，优先使用ParNew+CMS（原因见后面），当用户线程内存不足时，采用备用方案Serial Old收集。

CMS收集的方法是：先3次标记，再1次清除，3次标记中前两次是初始标记和重新标记（此时仍然需要停止（stop the world））， 初始标记（Initial Remark）是标记GC Roots能关联到的对象（即有引用的对象），停顿时间很短；并发标记（Concurrent remark）是执行GC Roots查找引用的过程，不需要用户线程停顿；重新标记（Remark）是在初始标记和并发标记期间，有标记变动的那部分仍需要标记，所以加上这一部分 标记的过程，停顿时间比并发标记小得多，但比初始标记稍长。在完成标记之后，就开始并发清除，不需要用户线程停顿。

所以在CMS清理过程中，只有初始标记和重新标记需要短暂停顿，并发标记和并发清除都不需要暂停用户线程，因此效率很高，很适合高交互的场合。

CMS也有缺点，它需要消耗额外的CPU和内存资源，在CPU和内存资源紧张，CPU较少时，会加重系统负担（CMS默认启动线程数为(CPU数量+3)/4）。

另外，在并发收集过程中，用户线程仍然在运行，仍然产生内存垃圾，所以可能产生“浮动垃圾”，本次无法清理，只能下一次Full GC才清理，因此在GC期间，需要预留足够的内存给用户线程使用。所以使用CMS的收集器并不是老年代满了才触发Full GC，而是在使用了一大半（默认68%，即2/3，使用-XX:CMSInitiatingOccupancyFraction来设置）的时候就要进行Full GC，如果用户线程消耗内存不是特别大，可以适当调高-XX:CMSInitiatingOccupancyFraction以降低GC次数，提高性能，如果预留的用户线程内存不够，则会触发Concurrent Mode Failure，此时，将触发备用方案：使用Serial Old 收集器进行收集，但这样停顿时间就长了，因此-XX:CMSInitiatingOccupancyFraction不宜设的过大。

还有，CMS采用的是标记清除算法，会导致内存碎片的产生，可以使用-XX：+UseCMSCompactAtFullCollection来设置是否在Full GC之后进行碎片整理，用-XX：CMSFullGCsBeforeCompaction来设置在执行多少次不压缩的Full GC之后，来一次带压缩的Full GC。

* G1收集器：在JDK1.7中正式发布，与现状的新生代、老年代概念有很大不同，目前使用较少，不做介绍。

注意并发（Concurrent）和并行（Parallel）的区别：

**并发**是指用户线程与GC线程同时执行（不一定是并行，可能交替，但总体上是在同时执行的），不需要停顿用户线程（其实在CMS中用户线程还是需要停顿的，只是非常短，GC线程在另一个CPU上执行）；

**并行**收集是指多个GC线程并行工作，但此时用户线程是暂停的；

所以，Serial和Parallel收集器都是并行的，而CMS收集器是并发的.

**关于JVM参数配置和内存调优实例，见我的下一篇博客（编写中：Java系列笔记(4) - JVM监控与调优），本来想写在同一篇博客里的，无奈内容太多，只好另起一篇。**

**说明：**

　　本文是Java系列笔记的第3篇，这篇文章写了很久，主要是Java内存和 GC机制相对复杂，难以理解，加上本人这段时间项目和生活中耗费的时间很多，所以进度缓慢。文中大多数笔记内容来源于我在网络上查到的博客和《深入理解 Java虚拟机：JVM高级特效与最佳实现》一书。

　　本人能力有限，如果有错漏，请留言指正。

**参考资料：**

《JAVA编程思想》，第5章；

《Java深度历险》，Java垃圾回收机制与引用类型；

《深入理解Java虚拟机：JVM高级特效与最佳实现》，第2-3章；

成为JavaGC专家Part II — 如何监控Java垃圾回收机制, http://www.importnew.com/2057.html

JDK5.0垃圾收集优化之--Don't Pause,http://calvin.iteye.com/blog/91905

【原】java内存区域理解-初步了解，http://iamzhongyong.iteye.com/blog/1333100