链滴

《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》简要读书笔记（更新中）

作者：seanlee

原文链接：https://ld246.com/article/1501406922713

来源网站：链滴

<h2 id="第一章走进java">第一章：走进Java</h2>
<h4 id="概述">概述</h4>
<h4 id="java技术体系">Java技术体系</h4>
<h4 id="java发展史">Java发展史</h4>
<h4 id="java虚拟机发展史">Java虚拟机发展史</h4>

1996年 JDK1.0，出现Sun Classic VM
HotSpot VM， 它是 Sun JDK 和 OpenJDK 中所带的虚拟机，最初并不是Sun开发
Sun Mobile- Embedded VM/ Meta- Circular VM
BEA JRockit/ IBM J9 VM JRockit曾号称世界上最快的java虚拟机，BEA公司发布.J9属于IBM主扶持的虚拟机
Azul VM/ BEA Liquid VM 我们平时所提及的“ 高性能Java虚拟机” 一般 是指 HotSpot、 JRocit、 J9这类在通用平台上运行的商用虚拟机，但其实 Azul VM和 BEA Liquid VM 这类特定硬件平台有的虚拟机才是“ 高性能” 的武器。
Apache Harmony/ Google Android Dalvik VM
Microsoft JVM 及其他

<h4 id="展望java技术的未来">展望JAVA技术的未来</h4>

模块化
混合语言
多核并行
进一步丰富语法
64位虚拟机

<h2 id="第二章java内存区域与内存溢出异常">第二章：Java内存区域与内存溢出异常</h2>
<h3 id="jvm运行时的数据区">JVM运行时的数据区</h3>
共有五个，线程共享的有 堆（Heap）、方法去（Method Area）， 线程私有的有 虚拟机栈（M Stack）、本地方法栈（Native Method Stack）和程序计数器。

程序计数器
程序计数器（Program Counter Register） 是一 块 较小的内存空间，它可以作是当前线程所执行的字节码的行号 指示器。 如果线程正在执行的是一个Java方法，这个计数器记的是正在执行的 虚拟机字节码指令的地址； 如果正在执行的是 Native 方法，这个 计数器值则为空（Undefined）。 此内存区域是唯一一个在 Java虚拟 机规范中没有规定任何OutOfMemoryError情况区域。
Java虚拟机栈
与程序计数器一样，虚拟机栈也是线程私有的，它的 生命周期与线程相同。拟机栈描述的是Java方法执行 的内存模型：每个方法在执行的同时都会创建一个栈 帧（ Stack Fram） 用于存储 局部 变 量表、操 作数栈、动态 链接、方法出口等息。 每一个方法 从调用直至执行完成的过程，就对应着一个栈帧在虚拟 机栈中入栈到出的过程。局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、 对象引用和returnAddress 类型。
在 Java 虚拟机规范中， 对这个区域定了两种异常 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；如果虚拟机 栈可以动态扩展（当前大部分的Java虚拟机都可动态 扩展， 只不过Java虚拟机规中也允许固定长度的虚拟 机 栈），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。
本地方法栈
本地方法栈与虚拟机栈所发挥的作用是非常相似的， 它们之间的区别不过是虚机栈为虚拟机执行Java方法（也就是 字节 码） 服务，而本地方法栈则为虚拟机使 用到的Native 方服务。
与 虚拟 机 栈 一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError异常。
Java堆 Java堆是虚拟机所管理的内存中最大的一块，是 被所有线程共享的一块内存区域，在虚机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例 都在这里分配内存。br>Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“ GC 堆” （Garbage Collected eap）。
从内存回收的角度来看，由于现在收集器 基本都采用分代收集算法，所以Java堆中还

以细分为： 新生代和老年代； 再细致 一点的有Eden空间、From Survivor 空间、 To Survivor 空间。
根据Java虚拟机规范的规定， Java堆可以 处于物理上不连续的内存空间中，只要逻辑上 是续的即可，就像我们的磁盘空间一样。在实现时， 既可以实现成固定大小的，也可以是可扩展的，不当前主流的虚拟机都是按照可扩展来实现的（通过- Xmx 和- Xms 控制）。如果在堆中没有内存完成例分配， 并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。
方法区 方法区用于存储已被虚拟机加载的类信息、 常量、静态 变量、即时编译器编后的 代码等数据。 虽然 Java 虚拟机规范把 方法区描述为堆的一个逻辑部分，但是它却一个别名叫做 Non- Heap（ 非 堆），目的应该是与堆区分开来。
很多人都更愿意把方法区称“ 永久代” （ Permanent Generation），本质上两者并不 等价，等价， 仅仅是因为 HotSpot 虚拟机 的设计团队选择把 GC分代收集扩展至 方法 区， 或者说 使用 永久代来实现方法区 而已， 这样HoSpot的垃圾收集器可以像管理Java堆一样管理这部分内存，能够省去专门为方法区编写内存管理代码工作，并非就如永久带就是就是进入了方法区。（Java 8已经将移除了永久代更改为元数区）
运行时常量池
Runtime Constant Pool， 该区域属于法区的一部分。 Class 文件中除了有类的版本、字段、方法、接等 描述信息外，常量池，用于存放编译期生成的各种字面量和符号引用，这部内容将在类加载后进入方法区的运行时常量池中存放。
运行时常量池相对于Class 文件常量池的另外 一个重要特征是具备动态性，Java语言并不 要求常量一定只有编译期才产生，也就是并非 预置Class文件中常量池的内容才能进入 方法区运行时 常量 池， 运行 期间 也可 能将 新的 常量 放入 池， 这种 特性 被 开发 人员 利用 得比 较多 的 便是 String 类 的 intern() 方法。 当常量池无法 再申请内存时会抛出 OutOfMemoryError 异常。

<h3 id="hotspot虚拟机对象探秘">HotSpot虚拟机对象探秘</h3>
以Java堆为例，探访对象的创建、内存布局和定位。

对象的创建

虚拟机遇到一条new指令时， 首先 将去 检查 这个 指令 的参数是否能在常量池中定位到一个类符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。 如果没有，那必须先执相应的类加载过程。
通过类加载之后，从Java堆中划出一块内存给新生的对象。内存的分配方式有多种，由Java堆是否规整决定，最终由采用的垃圾收集器决定。
在分配内存是考虑到并发和线程问题，除了同步处理保证原子性之外，有一种方法是地线程分配缓冲（Thread Local Allocation Buffer, TLAB）:是把内存分配 的动作按照线划分在不同的空间之中进行，哪个 线程 要 分配 内存， 就在哪个线程的TLAB上分配， 只有 TLAB 用并分配新的TLAB 时， 才需要同步锁定。 虚拟机是否使用TLAB，可以通过- XX:+/- UseTLAB 参数来定。
将内存空间初始化零值，如果使用了TLAB则可以在分配时初始化。
对对象进行必要的设置， 例如这个对象是哪个类的 实例、 如何才能找到类的元数据信息、对象哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头（ Object Header）之中。

<h3 id="对象的内存布局">对象的内存布局</h3>

在 HotSpot 虚拟机中，对象在内存中存储的布局可以分为3块区域： 对象头（ Header）、实例数据（ Instance Data）和对齐填充（ Padding）。
对象头：两部分：第一部分用于存储对象自身的运行时数据， 如哈希码（HashCode）、 GC分年龄、锁状态 标志、线程持有的锁、偏向线程ID、偏向时间戳等，32位和64位虚拟机中中分别为32和64位，官方成为“Mark Word”。第二部分类型指针，即对象指向它元数据的指针，虚拟机通过针来确定是哪个类的实例。
实例部分试对象真正存储的有效信息，程序中各种类型的字段内容。
对其填充并不是必然存在的，仅仅起着占位符的作用。保证对象的大小是8字节的倍数。

<h3 id="对象的访问定位">对象的访问定位</h3>
Java程序需要通过栈上的reference数据来操作堆上的 具体对象。 由于reference类型在Java虚机规范中只 规定了一个指向对象的引用，并没有定义这个引用应该 通过何种方式去定位、访问堆中对象的具体 位置， 所以对象访问方式也是取决于虚拟机实现而定 的。 目前 主流的访问方式有<stron>使用句柄和直接指针两种。

句柄：那么 Java 堆 中将 会 划分 出 一块 内存 来作为句柄池， <stron>reference中存储的就是对象的句柄 地址， 而句柄中包含了对象实例数据与类型数据 各自的具体地信息。
直接指针：如果使用直接指针访问， 那么Java 堆 对象的布局中就必须考虑如何放置访问类型 数的相关信息， 而reference中存储的直接就是 对象地址。
这两种对象访问方式各有优势， 使用句柄来访问的 最大好处就是reference中存储的是稳定的句地址，在 对象被移动（垃圾收集时移动对象是非常普遍的行为） 时只会改变句柄中的实例数据指针而reference本身 不需要修改。使用直接指针访问方式的最大好处就是 速度更快，它节省了一次指针位的时间开销，由于 对象的访问在Java中非常频繁，因此这类开销积少成多后也是一项非常可观的执成本。就本书讨论的主要虚拟机Sun HotSpot而言，它是使用第二种方式进行对象访问的，但从整个件开发的范围来看，各种 语言和框架使用句柄也很很常见。

<h3 id="实战-outofmemoryerror异常">实战 OutOfMemoryError异常</h3>

Java堆溢出 Java堆用于存储对象实例，只要不断的创建对象并且保证GC Roots 到 对象之间有达途径保证不被来及回收这些对象，就可以造成堆内存溢出。
虚拟机栈和本地方法栈溢出 对于 HotSpot 来说， 虽然- Xoss 参数（ 设置 本地 方法 栈 大小）在， 但 实际上 是 无效 的， 栈 容量 只 由- Xss 参数 设定。
如果 线程 请求 的 栈 深度 大于 虚 机 所 允许 的 最大 深度， 将 抛出 StackOverflowError 异常。
如果 虚拟 机 在 扩展 栈 时 无法申请 到 足够 的 内存 空间， 则 抛出 OutOfMemoryError 异常。
假如一台 计算机内存有2G，JM提供参数来控制Java 堆 和 方法 区 的 这 两部分 内存 的 最大值。 剩余 的 内存 为 2GB（ 操作系统限制） 减去 Xmx（ 最大 堆 容量）， 再 减去 MaxPermSize（ 最大 方法 区 容量）， 程序 计数器 耗 内存 很小， 可以 忽略 掉。 如果 虚拟 机 进程 本身 耗费 的 内存 不计 算在 内， 剩下 的 内存 就 虚拟 机 栈 和 本地 方法 栈 “ 瓜分” 了。 每个 线程 分配 到 的 栈 容量 越大， 可以 建立 的 线程量 自然 就 越少， 建立 线程 时 就 越 容易 把 剩下 的 内存 耗尽。
方法区和常量池溢出 String. intern()是一个Native方法，它的作用是：如果 字符串常量池中已包含一个等于此String对象的字符串， 则返回代表池中这个字符串的String对象；否则，将此String象包含的字符串添加到常量池中，并且返回 此String对象的引用。 在JDK 1. 6及之前的版本中，<spa> 由于常量池分配在永久代内，我们可以通过- XX:PermSize- XX:MaxPermSize 限制方法区大小， 从而间接限制其中常量池的容量。
本机直接内存溢出 DirectMemory容量可通过- XX: MaxDirectMemorySize 指定， 如果不指定 则默认与 Java堆最大值（- Xmx 指定）一样。

<h2 id="第三章垃圾收集器与内存分配策略">第三章：垃圾收集器与内存分配策略</h2>
垃圾收集（Garbage Collection，GC）,最开始诞生于1960的Lisp.
程序计数器、虚拟机栈本地方法栈随线程而生，随线程而灭，这三个区域不考虑垃圾回收。Java堆和方法区是主要进行垃圾收的区域。
<h3 id="判断对象是否已经死亡">判断对象是否已经死亡</h3>

引用计数算法（Reference Counting）
给对象中添加一个引用计数， 每当有一个地方引用它时， 计数器值就加1； 当引用失效时，计数器值就减1；任何时刻计数器为的对象就是不可能再被使用的。
优点:判定效率高 缺点：无法解决对象之间相互循环引用的问题

<blockquote>
举个 简单 的 例子， 请看 代码 清单 3- 1 中的 testGC() 方法： 对象 objA 和 objB 都有 字段 instance， 赋值 令 objA. instance= objB 及 objB. instance= objA， 除此之外， 这 两个 对象 再无 任

引用， 实际上 这 两个 对象 已经 不可能 再被 访问， 但是 它们 因为 互相 引用 着 对方， 导致 它们 的 引用 计数 都不 为 0， 于是 引用 计数 算法 无法 通知 GC 收集 器 回收 它们.
</blockquote>

可达性分析算法（Reachability Analysis）这个算法的基本思路就是通过系列的称为"GC Roots"的对象作为起始 点， 从这些节点开始向下搜索，搜索所走过的路径称为引用（ Reference Chain），当一个对象到GC Roots没有任何引用链相连（用 图论的话来说，就是从GC oots到这个对象不可达）时，则证明此对象是不可用的。

再谈引用（引用的定义及分类） 在 JDK 1. 2 以前，Java中的引用的定义很传统：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。
在 DK 1. 2之后， Java对引用的概念进行了扩充，将引用分为强引用（ Strong Reference）、软引用（ oft Reference）、 弱引用（ Weak Reference）、虚引用（Phantom Reference）4种，这4种引用 度依次逐渐减弱。

强引用：只要强引用还在永远不会被垃圾回收，定义方式通常为 A a = new A().
软引用:描述一些还有用但非必要的属性。在系统将要发生内存溢出 异常之 ，将会把这些对象列进回收范围之中进行第二次回收。如果 这次回收还没有足够的内存， 才会抛出 存溢出异常。
软引用：它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下 次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够， 都会回收掉只被弱 引用关联 对象。
虚引用：也成为幽灵引用或者幻影引用，最弱。为一个对象设置虚引用关 的唯一目的就是能在这个对象 被收集器回收时收到一个系统通知。
<h3 id="生存还是死亡">生存还是死亡</h3>
要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行 可达性分析后发现没有与 CRoots相连接的引用链，那它将会被第一次标记并且进行一次筛选， 筛选的条件是此对象是否有必 执行 finalize() 方法。 当对象没覆盖finalize() 方法， 或者finalize() 方法已经被虚拟机调用过，虚拟 将这两种情况都视为 “ 没有必要 执行” 。 如果这个对象被判定为有必要执行 finalize()方法， 那么 个对象将会放置在一个叫做F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的F nalizer线程去执行它。
finalize()方法是对象逃脱死亡命运的最后一次机会。
<h3 id="回收方法区">回收方法区</h3>
Java 虚拟机规范中确实说过可以不要求虚拟机在方法区实现垃圾收集， 而且在方法区中进行垃 收集的 “ 性价比” 一般比较低：在堆中，尤其是 在新生代中， 常规应用进行一次垃圾收集一般可以 收70%~95%的空间，而永久代的垃圾收集效率远低于此。
永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。 回收 废弃 量与回收Java堆中的对象非常类似。
如何判断一个类是否无用：

该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
加载该类的ClassLoader已经被回收。
该类对应的java. lang. Class 对象没有在任何地方被引用， 无法 在任何地方通过反射访问该类的 法。

<h3 id="垃圾收集算法">垃圾收集算法</h3>
<h4 id="31-标记-清除算法">3.1 标记-清除算法</h4>
最基础的收集算法是 “ 标记- 清除” （ Mark- Sweep）算法，如同它的 名字一样， 算法分为 “ 标记” 和 “ 清除” 两个阶段： 首先标记出 所有 需要回收的对象， 在标记完成后统一回收所有被标 的对象。

不足：它的主要不足有两个： 一个是效率问题， 标记和清除两个过程 的效率都不高；另一个是间 问题， 标记清除之后会产生大量 不连续 的内存碎片， 空间碎片太多可能会导致以后在程序运行程中需要 分配较大对象时， 无法找到足够的连续内存而不得不提前触发另一次 垃圾收集。
<h4 id="32-复制算法">3.2 复制算法</h4>
为了解决效率问题， 一种称为 “ 复制” （ Copying） 的收集算法出现 了， 它将可用内存按容划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回，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，按顺序分配内存即可，实现简，运行高效。只是这种算法的代价是内存缩小为了原来的一半，未免太高了一点。
现在的商业虚拟机都采用这种收集算法来回收新生代， IBM公司的专门研究表明，新生代中的对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大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。 当回收时，将Eden和Survivor 中 还存活着的对象 一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用的 Survivor 空间。HotSpot虚拟机默认的Eden和Survivor的大小为8:1
缺点：在对象存活率比较高时就要进行较多的复制操作，效率会变低。
<h4 id="33-标记-整理算法">3.3 标记-整理算法</h4>
（Mark-Compact）老年代一般采取该算法。
算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，是让所有存活的对象都向一端移动， 然后直接 清理掉端边界以外的内存。
<h4 id="34-分代收集算法">3.4 分代收集算法</h4>
当前商业虚拟机的垃圾收集都采用 “ 分 代 收集” （ Generational Collection）算法，这种算并没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发有大批对象死去，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收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用 “ 标记— 清理” 或者 “ 标记— 整理” 算法来进行回收。
<h3 id="hotspot的算法实现">HotSpot的算法实现</h3>
<h4 id="41枚举根节点">4.1枚举根节点</h4>
在HotSpot的实现中， 是使用一组称OopMap的数据结构 来达到这个目的的， 在类加载完成的候， HotSpot 就把对象内什么偏移量上是什么类型的数据计算出来， 在JIT编译过程中，也会在特定位置记录下栈和 寄存器 中哪些位置是引用。这样， GC在扫描时就可以直接得知 这些信息了。
<h4 id="42-安全点">4.2 安全点</h4>
在 OopMap的协助下，HotSpot可以快速且准确地完成 GC Roots 枚举，但一个很现实的问题之而来：可能导致 引用关系变化，或者说OopMap内容变化的指令非常多， 如果为每一条指令都生对应的OopMap，那将会需要大量的额外空间， 这样GC的空间成本将会变得很高。
实际上，HotSpot也的确没有为每条指令都生成 OopMap， 前面已经 提到，只是 在 “ 特定 的位置” 记录了 这些信息， 这些位置称为安全点 （ Safepoint） ，即 程序执行时并非在所有地方都能顿下来开始 GC，只有 在到达安全点时才能暂停。
<h4 id="43-安全区域">4.3 安全区域</h4>
Safepoint机制保证了程序执行时， 在不太长的时间内 就会遇到可进入 GC的Safepoint。
安全区域（Safe-Region）是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的意地方开始GC都是安全的。我们也可以把Safe-Region看做是被扩展了的Safepoint。
<h3 id="垃圾收集器">垃圾收集器</h3>
这里讨论的收集器基于JDK1.7Update14之后的HotSpot虚拟机。
<h4 id="51-serial收集器">5.1 Serial收集器</h4>
Serial 收集器是最基本、发展历史最悠久的收集器，曾经是虚拟机新生代集的唯一选择。该收集器为单线程收集器，它在工作时会暂停其它线程。 “Stop the world” 指的就这种情况。 优点：简单而高效 （ 与其他收集器的单线程 比） ， 对于限定单个CPU的环境来说，Seria收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集。
<h4 id="52-parnew收集器">5.2 ParNew收集器</h4>
ParNew收集器其实就是Serial收集器的多线程版本， 除了使用多条线程进行垃圾收集之外，其行为包括 Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策等都与Serial收集器完全一样， 在实现上，这两种收集器也共用了相当多的代码。
它是许多运行在Server模式下的虚拟机中首选的新生代 </s

an>收集器，其中有一个与性能无关但很重要的原因,原因 是，除了Serial收集器外，目前只有它能与MS收集器配合工作。
<blockquote>
在 JDK 1. 5 时期， HotSpot 推 出了 一 款 在 强 交互 应用 中 几乎 可 认为 有 划时代 意义 的 垾 收集 器—— CMS 收集 器（ Concurrent Mark Sweep， 本节 稍后 将 详细 介绍 这 款 收集 器） 这 款 收集 器 是 HotSpot 虚拟 机中 第一 款 真正 意义上 的 并发（ Concurrent） 收集 器， 它 第 次 实现 了 让 垃圾 收集 线程 与 用户 线程（ 基本上） 同时 工作， 用 前面 那个 例子 的 话来 说， 是 做 到了 在 你的 妈妈 打扫 房间 的 时候 你 还能 一边 往 地上 扔 纸屑。 不幸 的 是， CMS 作为 老 年代 的 收集 器， 却 无法 与 JDK 1. 4. 0 中 已经 存在 的 新生代 收集 器 Parallel Scavenge 配合 工 [1]， 所以 在 JDK 1. 5 中 使用 CMS 来 收集 老 年代 的 时候， 新生代 只能 选择 ParNew 或者 Seria 收集 器 中的 一个。
</blockquote>
<h4 id="53-parallel-scavenge收集器">5.3 Parallel Scavenge收集器</h4>
Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多程收集器。
Parallel Scavenge收集器的特点是它的关注点与其他收集器不同， CMS等收集器的关注点 是尽能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制吞吐量（Throughput）。
<h4 id="54-serial-old收集器">5.4 Serial Old收集器</h4>
Serial Old 是 Serial 收集 器 的 老年 代版本，它同样是一个单线程收集器， 使用 “ 标记- 整理” 算法。 这个收集器 的 主要意义也是在于给Client 模式下的 虚拟机使用。如果在Serve 模式下，那么它主要还有两大用途： 一种 用途 是在 JDK 1. 5 以及之前 的 版本中与 Parallel Scavenge收集器搭使用[1]， 另一种 用途就是作为CMS收集器的后备预案，在 并发收集发生Concurrent Mode Failur 时 使用。
<h4 id="parallel-old收集器">Parallel Old收集器</h4>
Parallel Old是Parallel Scavenge收集器 的老年代版本，使用多线程和 “ 标记- 整理” 算法。 个收集器是在JDK 1.6中 才开始提供的。
<h4 id="cmc收集器">CMC收集器</h4>
CMS（ Concurrent Mark Sweep）收集器 是 一种以获取最短回收停顿时间为目标的 收集器。 前很大一部分的Java 应用 集中在互联网站或者 B/ S 系统的服务 端上，这类应用尤其重视服务的响 速度， 希望系统停顿时间最短， 以给用户带来 较好的体验。
从名字（包含" Mark Sweep"） 上就可以 看出， CMS 收集器是基于 “ 标记— 清除” 算法实 的，它的运作过程相对于前面 几种收集器来说更复杂一些， 整个过程 分为 4 个 步骤， 包括：

初始 标记（ CMS initial mark）
并发 标记（ CMS concurrent mark）
重新 标记（ CMS remark）
并发 清除（ CMS concurrent sweep）

CMS是一款优秀的收集器，它的主要优点 在 名字上已经体现出来了： 并发收集、低 停顿，Sun 司的一些官方文档中也称之为 并发低停顿收集器（ Concurrent Low Pause Collector）。
缺点：

CMS 收集器对CPU资源非常敏感。
CMS 收集器无法处理浮动垃圾（ Floating Garbage），可能出现" Concurrent Mode Failure" 失败而导致 另一次Full GC的产生。
CMS 是一款基于 “ 标记— 清除” 算法 实现的收集器，收集结束时会有大量空间 碎片产生。空 碎片过多时，将会给大 对象分配带来很大麻烦， 往往会出现老年 代还有很大空间剩余， 但是无法找 足够大的连续空间来分配当前对象， 不得不 提前 触发 一次 Full GC。

<h4 id="57-g1收集器">5.7 G1收集器</h4>
G1（ Garbage- First 收集器是当今收集 器技术发展的最前沿成果之一，早在JDK 1. 7 刚刚确立 目目标， Sun 公司给出 的 JDK 1. 7 RoadMap 里面，它就被视为 JDK 1. 7 中HotSpot 虚拟机的一

重要进化 特征。
G1 是一 款 面向 服务 端 应用 的 垃圾 收集 器。 HotSpot 开发 团队 赋予 它的 使命 是 （ 在 比较 长期 的） 未来 可以 替换 掉 JDK 1. 5 中 发布 的 CMS 收集 器。 与其 他 GC 收集 器 相比， G1 具备 如下 特点。

并行与并发
分代收集
空间整合
可预测的停顿

<h4 id="58理解gc日志">5.8理解GC日志</h4>
虚拟机提供了- XX:+ PrintGCDetails 这个 收集器日志参数， 告诉虚拟机在发生垃圾 收集行为 打印内存回收日志， 并且在 进程退出的时候输出当前的内存各区域 分配情况。 在实际应用中， 内 回收 日志一般是打印到文件后通过日志工具 进行分析， 不过本实验的日志并不 多，直接阅读就能看 很清楚。
<h3 id="内存分配与回收策略">内存分配与回收策略</h3>
<h4 id="对象优先在eden分配">对象优先在Eden分配</h4>
新生代GC（ Minor GC）： 指发生在 新生代的垃圾收集动作， 因为Java 象 大多都具备朝生夕灭的 特性， 所以 Minor GC 非常 频繁，一般回收速度也比较快。
老年代GC（ Major GC/ Full GC）： 指发生在老年代的 GC，出现了Major GC， 经常会伴随至少一次的Minor GC（ 但非 绝对的，在 Parallel Scavenge 收集器 的收集策略里 有直接进行 Major GC 的 策略 选择 过程）。** Major GC的速度 一般 会 比 Minor GC慢10倍以上。 *
<h4 id="大对象直接进入老年代">大对象直接进入老年代</h4>
所谓的大对象是指， 需要大量连续内存 空间的 Java 对象， 最典型的大对象 就是 那种很长的字 串以及数组。 大对象对 虚拟机的内存分配来说就是一个坏消息，经常出现大对象容易导致内存还有 少空 间时就提前触发垃圾收集以获取足够的 连续 空间 来 “ 安置” 它们。
<h4 id="长期存活的对象进入老年代">长期存活的对象进入老年代</h4>
如果对象在Eden 出生并经过第一次 Minor GC 后仍然存活， 并且能被 Survivor 容纳的 话，将 移动到 Survivor 空间中， 并且 对象年龄设为1。 对象在 Survivor 区 中 每 “ 熬过” 一次 Minor G ，年龄就增加 1 岁， 当它的年龄增加到一定程度（ 默认 为 15 岁）， 就将会被晋升到老年代 中。 象晋升老年代的年龄阈值， 可以通过参数- XX: MaxTenuringThreshold 设置。
<h4 id="动态对象年龄判定">动态对象年龄判定</h4>
如果在 Survivor 空间中相同年龄所有对象大小的总和大于Survivor 空间的一半，年龄 大于或等 该年龄的对象就可以直接进入老年代， 无须等到 MaxTenuringThreshold 中 要求的年龄。
<h4 id="空间分配担保">空间分配担保</h4>
在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总 间，如果这个条件成立，那么MinorGC可以确保是安全的。如果不成立，则虚拟机会查看HandlePro otionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否 于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次 Minor G 是有风险的； 如果小于，或者HandlePromotionFailure 设置不允许冒险， 那这时也要改为进行一次 ull GC。
<h2 id="第四章虚拟机性能监控与故障处理">第四章：虚拟机性能监控与故障处理</h2>
给一个系统定位问题的时候，知识、 经验是关键基础，数据是依据，工具 是运用知识处理数据 手段。 这里说的数据包括： 运行日志、&nbs ;异常堆栈、 GC 日志、 线程 快照（ threaddump/ javacore 文件）、 堆 转储 照（ heapdump/ hprof 文件） 等。 经常使用适当的虚拟机监控和 分析的工具可以加快 们分析数据、 定位解决问题的速度。
<h2 id="第五章调优案例分析与实战">第五章：调优案例分析与实战</h2>
<h2 id="第六章类文件结构">第六章：类文件结构</h2>
<h2 id="第七章虚拟机加载类机制">第七章：虚拟机加载类机制</h2>
<h2 id="第八章虚拟机字节码执行引擎">第八章：虚拟机字节码执行引擎</h2>
<h2 id="第九章-类加载及执行子系统的案例与实战">第九章： 类加载及执行子系统的案例与实战</ 2>

<h2 id="第十章早期编译期优化">第十章：早期（编译期）优化</h2>
<h2 id="第十一章-晚期运行期优化">第十一章： 晚期（运行期）优化</h2>
<h2 id="第十二章java内存模型与线程">第十二章：Java内存模型与线程</h2>
<h2 id="第十三章线程安全与锁优化">第十三章：线程安全与锁优化</h2>