**《深入理解Java虚拟机》笔记:**

一.Java内存区域:

1、程序计数器:

程序计数器是一块较小的内存空间,可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。线程私有。唯一一个没有任何OutOfMemoryError情况的区域。

2、Java虚拟机栈：

虚拟机栈描述的是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用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行结束完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型、对象引用和returnAddress类型。

线程私有。该区域有两种异常状况：如果线程请求的栈深度大于虚拟机栈所允许的深度，将抛出StackOverFlowError异常；如果虚拟机栈可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

3、本地方法栈：

和虚拟机栈发挥的作用相似，不过虚拟机栈为虚拟机执行Java方法服务，本地方法栈则为虚拟机执行Native方法服务。

4、堆：

Java堆是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块区域。在虚拟机启动时创建。次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以及数组都在这里分配内存。

如果堆中没有内存完成实例分配，并且无法扩展时，就会抛出OutOfMemoryError异常。

5、方法区：

方法区也是线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

该区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

6、运行时常量池：

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池。运行时常量池相对于Class文件常量池的另一个重要特性是具备动态性，运行期间也可能将新的常量放入池中，用的较多的是String类的intern（）方法。

二.Java对象:

1.对象的创建：

1.1、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的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过，如果没有，必须先执行相应的类加载过程。

1.2、在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定。（分配内存算法分为“指针碰撞”和“空闲列表”两种，选择哪种方式由Java堆是否完整决定，而Java堆是否完整又由所采用的垃圾收集器是否带有压缩整理功能决定。）

1.3、内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头）。

1.4、接下来虚拟机要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、对象的哈希码、对象的GC分代年龄等信息。这些信息都存放在对象头之中。

2.对象内存布局：

分为对象头、示例数据、对齐填充三部分。

对象头：一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等；另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

实例数据:对象真正存储的有效信息,也是代码中所定义的各种类型的字段内容。

对齐填充：并不是必然存在的，没有什么特殊含义，仅仅起着占位符的作用。由于对象的大小必须是8字节的整数倍，而对象头部分正好是8字节的倍数，因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

3、对象访问定位：

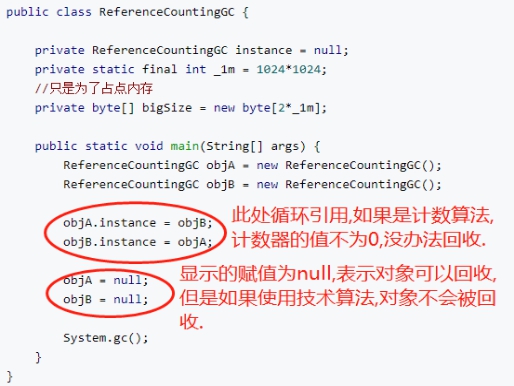
2.1、使用句柄访问：Java堆中会划分出一块内存用来作为句柄池，reference中存储的就是对象的句柄地址，句柄中包含了对象实例数据与类型数据的具体地址信息。使用句柄来访问的最大好处就是存储的句柄地址稳定，对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改。

2.2、使用直接指针访问：Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而reference中存储的直接就是对象地址。使用直接指针的最大好处就是速度更快，节省了一次指针定位的时间开销。虚拟机Sun HotSpot就是使用直接指针访问。

三.判断对象存活算法:

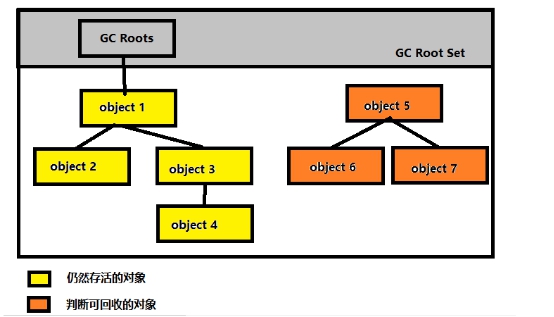
1.引用计数算法:

给对象添加一个引用计数器,每当一个地方引用它时,计数器值加1;当引用失效时,计数器值减1;任何时候计数器为0的对象就是不可能再被使用的.该算法没办法解决循环引用的问题.



2.可达性分析算法:

通过一系列的称为“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链（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 Roots到这个对象不可达）时，则证明此对象是不可用的。如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。(如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链,那它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要执行finalize()方法.当对象没有覆盖finalize()方法或者finalize()方法以及被JVM虚拟机调用过一次,此种情况下会被视为”没有必要执行”.finalize()方法最多会被自动调用一次.)



在Java语言中，可作为GC Roots的对象包括下面几种：

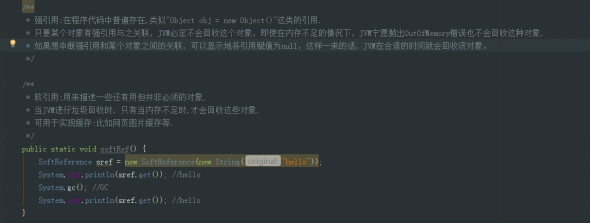
1. 虚拟机栈（栈帧中的本地变量表）中引用的对象。

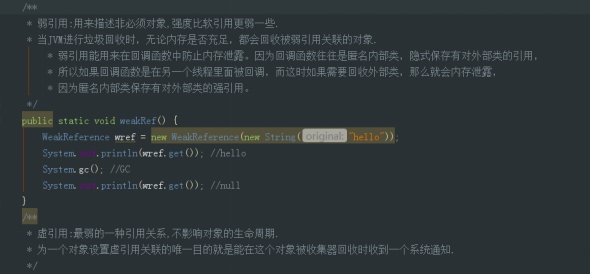
2. 方法区中类静态属性引用的对象。

3. 方法区中常量引用的对象。

4. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

四.Java四种引用:



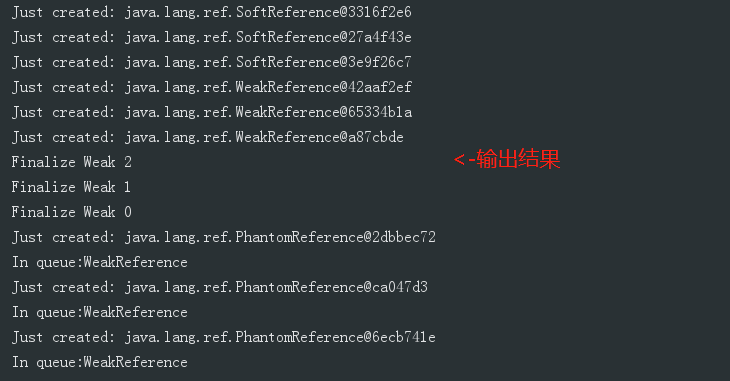




引用代码示例:







五.方法区回收:

永久代的垃圾回收主要两个部分:废弃常量和无用的类。

1.“废弃常量”条件:假如常量池中有”abc”,但是系统中并没有任何地方引用该字面量,必要情况下,该常量就废弃并被清理。常量池中的其他类(接口)、方法、字段的符号引用与此类似。

2.“无用的类”条件：该类所有的实例已经被回收；加载该类的ClassLoader已经被回收；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类的方法。

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的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

**六.垃圾收集算法:**

**1.标记-清除算法:**

首先标记所有需要回收的对象,在标记完成后统一回收所有被标记的对象。

标记过程：判断对象是否存活主要采用引用计数算法和可达性分析算法，Java采用可达性分析算法；要宣告一个对象死亡，至少要经历两次标记过程：如果对象进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被标记并且进行一次筛选，筛选条件是次对象是否执行finalize（）方法。

不足：1.效率问题，标记和清楚的效率都不高。

2.空间问题，标记清除之后会产生大量的不连续内存碎片，空间碎片太多可能会导致以后在程序运行过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一次垃圾收集动作。

**2.复制算法:**

它将可用内存按容量划分为大小相等的两块，每次都使用其中的一块。当这一块内存用完了，就将还存活的对象复制到另一块上面，然后再对已使用过的内存空间一次清理掉。这样使得每次都是对半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆栈指针，按顺序分配内存即可。

该算法主要用于回收新生代。内存划分时并不需要按照1:1的比例划分，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中的一块Survivor。当回收时，将Eden和Survivor中还存活的对象一次性的复制到另外一块Survivor上，然后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1。

**3.标记-整理算法:**

根据老年代的特点，有人提出了“标记-整理”算法，标记过程与“标记-清理”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理端边界外的内存。

**4.分代收集算法:**

当前商业虚拟机都采用“分代收集”算法。在新生代中，每次垃圾收集时都发现有大量对象死亡，只有少量存活，那就采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

**七.HotSpot算法实现:**

1.枚举根节点：

可作为GC Roots的节点主要在全局性的引用（例如常量和静态属性）与执行上下文（例如栈帧中的本地变量表）中。可达性分析对执行时间的敏感体现在GC停顿上，因为这项分析工作必须在一个能确保一致性的快照中进行--这里的“一致性”的意思是指在分析过程中整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断的变化的情况。所以GC进行时必须停顿所有的Java执行线程（Stop The World）。

虚拟机应当是有办法直接得知那些地方存放着对象引用，在Hot Spot的实现中，是使用一组成为OopMap的数据结构来达到这个目的的，在类加载完成时，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，也会在特定的位置记录下栈和寄存器中哪些位置是引用。

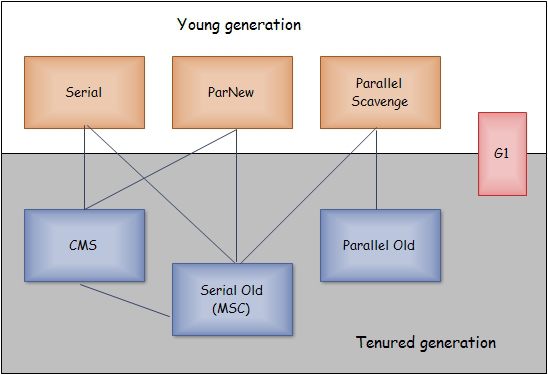
2.安全点：

HotSpot没有为每条指令都生成OopMap，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停顿下来开始GC，只有在到达安全点才能停顿。安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生Safepoint。

3.安全区域：

安全区域是指在一段代码片段之中,引用关系不会发生变化。可以吧Safe Region看做是被扩展了的Safepoint。

**八.垃圾收集器:**



1.Serial收集器:

最基本、发展历史最悠久的收集器，在jdk1.3.1之前是新生代的唯一选择。是一个单线程的收集器。在它进行垃圾回收时必须暂停其他所有的工作现场，知道它收集结束。对于运行在Clients模式下的虚拟机来说是一个很好的选择。



2.ParNew收集器：

是Serial收集器的多线程版本，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与CPU的数量相同。



3.Parallel Scavenge收集器：

是一个新生代的收集器，它也是采用复制算法的收集器，又是并行的多线程收集器。由于与吞吐量关系密切，又称为“吞吐量优先”收集器。该收集器有一个参数-XX:+UseAdaptiveSizePolicy值得关注，这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor的比例（-XX:SurvivorRatio）、晋升老年代对象大小（-XX:PretenureSizeThreshold）等细节参数，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）



4.Serial Old收集器：

是Serial收集器的老年代版本，也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在于给Clients模式下的虚拟机使用。



5.Parallel Old收集器：

是Parallel Scavenge收集器的老年代版本,使用多线程和”标记-整理”算法。



6.CMS收集器：

第一款真正意义上的并发收集器，第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。是一种以获取最短回收停顿时间为目标的收集器。是老年代收集器，基于“标记-清除”算法。

步骤：

1.初始标记（CMS initial mark）

2.并发标记（CMS concurrent mark）

3.重新标记（CMS remark）

4.并发清除（CMS concurrent sweep）

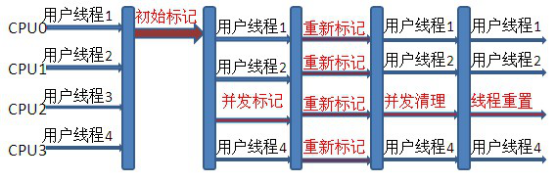
初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录。

缺点：

1.CMS收集器对CPU资源非常敏感。

2.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。

3.因为基于“标记-清除”算法实现，意味着收集结束时会有大量的空间碎片产生。



7.G1收集器：

G1收集器一款面向服务端应用的收集器。

特点：

1.并行与并发

2.分代收集

3.空间整合

4.可预测的停顿

步骤：

1.初始标记（Initial mark）

2.并发标记（Concurrent mark）

3.最终标记（Final Marking）

4.筛选回收（Live Data Counting and Evacuation）

并行与并发：

并行：指多条垃圾收集线程并行工作，但此时用户线程处于等待状态。

并发：指用户线程与垃圾收集线程同时执行（但并不一定是并行的，可能交替执行），用户程序在继续运行，而垃圾收集器程序运行于另一个CPU上。

吞吐量：

指CPU用于运行用户代码的时间与CPU总消耗时间的比值。

吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

**垃圾收集器参数总结：**

|  |  |
| --- | --- |
| **参数** | **描述** |
| **-XX:+UseSerialGC** | Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收 |
| **-XX:+UseParNewGC** | 打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收 |
| **-XX:+UseConcMarkSweepGC** | 使用ParNew + CMS +  Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。 |
| **-XX:+UseParallelGC** | Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge +  Serial Old的收集器组合进行回收 |
| **-XX:+UseParallelOldGC** | 使用Parallel Scavenge +  Parallel Old的收集器组合进行回收 |
| **-XX:SurvivorRatio** | 新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1 |
| **-XX:PretenureSizeThreshold** | 直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| **-XX:MaxTenuringThreshold** | 晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代 |
| **-XX:UseAdaptiveSizePolicy** | 动态调整java堆中各个区域的大小以及进入老年代的年龄 |
| **-XX:+HandlePromotionFailure** | 是否允许新生代收集担保，进行一次minor gc后, 另一块Survivor空间不足时，将直接会在老年代中保留 |
| **-XX:ParallelGCThreads** | 设置并行GC进行内存回收的线程数 |
| **-XX:GCTimeRatio** | GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效 |
| **-XX:MaxGCPauseMillis** | 设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效 |
| **-XX:CMSInitiatingOccupancyFraction** | 设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70 |
| **-XX:+UseCMSCompactAtFullCollection** | 由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效 |
| **-XX:+CMSFullGCBeforeCompaction** | 设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用 |
| **-XX:+DisableExplicitGC** | 是否关闭手动System.gc |

九．GC日志：

33.125:[GC  [DefNew:  3324k->152k(3712k),  0.0025925 secs] 3324k->152k(11904k),  0.0031680 secs]

最前面的数字33.125代表GC发生的时间,这个数字的含义是从Java虚拟机启动以来经过的秒数.

GC日志开头的[GC或者[Full GC说明了垃圾收集的停顿类型,不是用来区分新生代GC和老年代GC的,如果有Full说明GC发生了Stop-The-World.

[DefNew或者[Tenured或[Perm表示GC发生的区域,是与GC收集器密切相关的。Serial收集器中的新生代名为“Default New Generation”，所以显示“[DefNew”；ParNew收集器的新生代名为“[ParNew”，意为“Parallel New Generation”；Parallel Scavenge收集器的新生代名为“PSToungGen”。

后面方括号内部的“3324k->152k(3712k)”含义是”GC前该内存区域已使用容量->GC后该内存区域已使用容量(该内存区域总容量)”.

方括号外的”3324k->152k(11904k)”表示”GC前Java堆已使用容量->GC后Java堆已使用容量(Java堆总容量)”

再往后的”0.0031680 secs”表示该内存区域GC所占用的时间.

**十.内存分配策略:**

**1.对象优先在Eden分配**

**大多数情况下,对象在新生代Eden区分配,当Eden区没有足够内存进行分配时,虚拟机将会发起一次Minor GC。GC期间采用复制算法进行垃圾回收，存活的对象大小如果大于Survivor区内存大小，将不会进入Survivor区，而是通过分配担保机制转移到老年代。**

**2.大对象直接进入老年代**

**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符串以及数组。经常出现大对象容易导致内存还有不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”他们。虚拟机提供了-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区以及两个Survivor区之间发生大量的内存复制。**

**3.长期存活的对象将进入老年代**

**虚拟机给每个对象定义了一个对象年龄计数器。如果对象在Eden出生并经过第一次Minor GC后仍存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。对象在Survivor区中每熬过一个Minor GC，年龄就加1，当它的年龄增加到一定程度（默认是15岁），就会晋升到老年代。对象晋升老年代的年龄阙值可以通过参数-XX：MaxTenuringThreshold设置。**

**4.动态对象年龄判定：**

**为了更好的适应不同程序的内存情况，虚拟机并不是永远的要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，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，年龄大于或等于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。**

**5.空间分配担保：**

**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空间，如果条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure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内存连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次Minor GC，尽管是有风险的；如果小于或者参数设置为不允许，则进行一次Full GC。在JDK6 Update24之后，HandlePromotionFailure参数不会再影响到虚拟机的空间分配担保策略，规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行Minor GC，否则进行Full GC。**

**Minor GC与Major GC：**

**新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭特性，所以Minor GC非常频繁，一般回收速度也很快。**

**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。Major GC的速度一般会比Minor GC慢10倍以上。**

**十一.类文件结构:**

1、概念：

1.1.任何一个Class文件都对应唯一一个类或接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里面，譬如类或接口也可以通过类加载器直接生成。任意一个有效的类或接口所应当满足的格式成为“Class文件格式”，实际上它并不一定以磁盘文件的形式存在。

1.2.Class文件是一组以8字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件之中，中间没有任何分隔符，这使得Class文件之中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。当遇到需要占用8位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储。

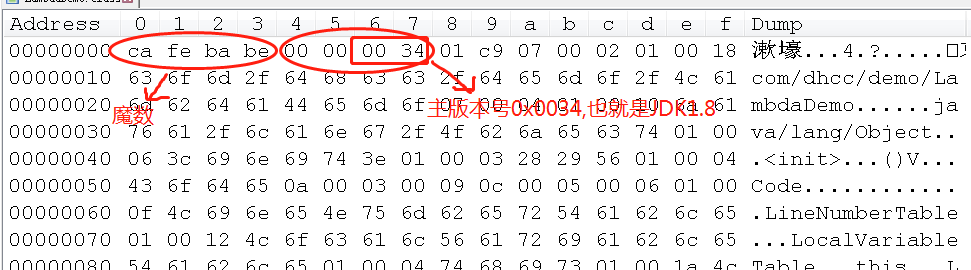
1.3.Class的存储数据结构只有两种类型：无符号数和表。

无符号数属于基本的数据类型，以u1、u2、u4、u8来分别表示1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成字符串值。

表是由多个无符号数或其他表作为数据项构成的复合数据类型，所有表都习惯性地以“\_info”来结尾。表用于描述层次关系的复合结构的数据，整个Class文件本质上就是一张表。

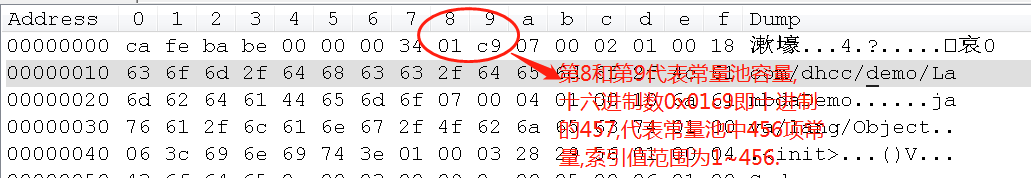
2、Class文件内容：

2.1、每个Class文件的头四个字节成为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件。Class文件的魔数是0xCAFEBABE。紧接着魔数的4个字节存储的是Class文件的版本号：第5和第6个字节是次版本号（Minor Version），第7和第8个字节是主版本号（Major Version）。Java的版本号是从45开始，JDK1.1之后的每个JDK大版本发布主版本号向上加1。



2.2、紧接着主次版本号之后的是常量池入口，常量池可理解为Class文件之中的资源仓库，它是Class文件结构中与其他项目关联最多的数据类型,也是占用Class文件空间最大的数据项目之一,同时它还是在Class文件中第一个出现的表数据类型项目。

由于常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项u2类型的数据，代表常量池容量计数值，与Java中语言习惯不一样的是，这个容量计数是从1开始。



常量池中主要存放两大类常量：字面量和符号引用。

字面量比较接近于Java语言层面的常量概念，如文本字符串、声明为final的常量值。

符号引用包括三类常量：

类和接口的全限定名、字段的名称和描述符、方法的名称和描述符。

**十二.虚拟机类加载机制:**

1、概念：

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析、初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

2、类加载的时机：

类从加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用、卸载7个阶段。其中验证、准备、解析3个部分统称为连接。

Java虚拟机规范中并没有对类加载过程的第一个阶段进行强制约束，这点可以交给虚拟机的具体实现来自由把握。但是对于初始化阶段，虚拟机规范则是严格规定了有且只有5种情况必须立即对类进行“初始化”（加载、验证、准备自然需要在此之前开始）：

1）、遇到new、getstatic、putstatic或invokestatic这4条指令时，如果类没有进行过初始化，则需要进行初始化。这4条指令的Java代码场景：使用new关键字实例化对象的时候，读取或设置一个类的静态字段（被final修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。

2）、使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发初始化。

3）、当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

4）、当虚拟机启动的时候，用户需要指定一个要执行的主类（包含main（）方法的类），虚拟机会先初始化这个主类。

5）、当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先进行初始化。

这5种场景中的行为称为对一个类进行主动引用，除此之外，所有引用类的方式都不会触发初始化，称为被动引用：

1）、通过子类引用父类的静态字段，不会导致子类初始化；

2）、通过数组定义来引用类，不会触发此类的初始化；

3）、常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

接口的加载过程与类加载过程稍有不同，接口中不能使用“static{}”语句块，但是编译器仍然会为接口生成“<clinit>()”类构造器，用于初始化接口中所定义的成员变量。接口与类真正有所区别的是前面描述的5种情况的第3种：当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父类接口全部都完成初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

3、类加载的过程：

3.1、加载：

在加载阶段，虚拟机需要完成3件事情：

1）、通过一个类的全限定名来获取定义此类的二进制字节流。

（该条并没有准确的指明要从哪里获取怎样获取，所有可以有以下情况：

1.1）、从ZIP包读取

1.2）、从网络获取，如Applet

1.3）、运行时计算生成，如动态代理技术

1.4）、由其他文件生成，如JSP应用

1.5）、从数据库读取，如一些中间件服务器（SAP Netweaver）

。。。。）

2）、将这个字节流所代表的的静态存储结构转化为方法区的运行时数据结构。（方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构。）

3）、在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。（并没有明确规定是在Java堆中，对于HotSpot虚拟机而言，Class对象比较特殊，它虽然是对象，但是存放在方法区里面。）

3.2、验证：

从整体上看，验证阶段分为4个阶段的校验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

1）、文件格式验证：

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理：

1.是否以魔数0xCAFEBABE开头。

2.主次版本号是否在当前虚拟机处理范围之内。

3.常量池的常量中是否有不被支持的常量类型。

4.指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量。

5.CONSTANT\_Utf8\_info型的常量中是否有不符合UTF8编码的数据。

6.Class文件中各个部分及文件本身是否有被删除的或附加的其他信息。

。。。。

该验证阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区被（也就是说该阶段在加载阶段时就已经开始验证了），格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区中进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。

2）、元数据验证：

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求：

1.这个类是否有父类。

2.这个类的父类是否继承了不允许被继承的类（被final修饰的类）。

3.如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法。

4.类中的字段、方法是否与父类产生矛盾。

。。。。

3）、字节码验证：

第三阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。在第二个阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件：

1.保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作。

2.保证跳转指令不会跳转到方法体以外的字节码指令上。

3.保证方法体中的类型转换是有效的。

。。。。

4）、符号引用验证：

最后一个阶段的验证发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段--解析阶段中发生。符号引用验证可以看做是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验：

1.符号引用中通过字符串描述的全限定名是否能找到对应的类。

2.在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段。

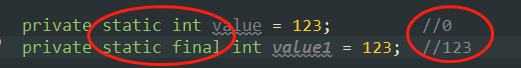
3.符号引用中的类、字段、方法的访问性是否可被当前类访问。

。。。。

3.3、准备：

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。

首先，这时候进行内存分配的仅包括类变量，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在堆内存中。其次，这里所说的初始值“通常情况”下是数据类型的零值，但是有一些特殊情况，如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值。



3.4、解析：

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时可以无歧义的定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用：直接引用是可以直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范中并未规定解析阶段发生的具体时间，虚拟机实现可以根据需要来判断到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

3.5、初始化：

类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>（）方法的过程。

4、类加载器：

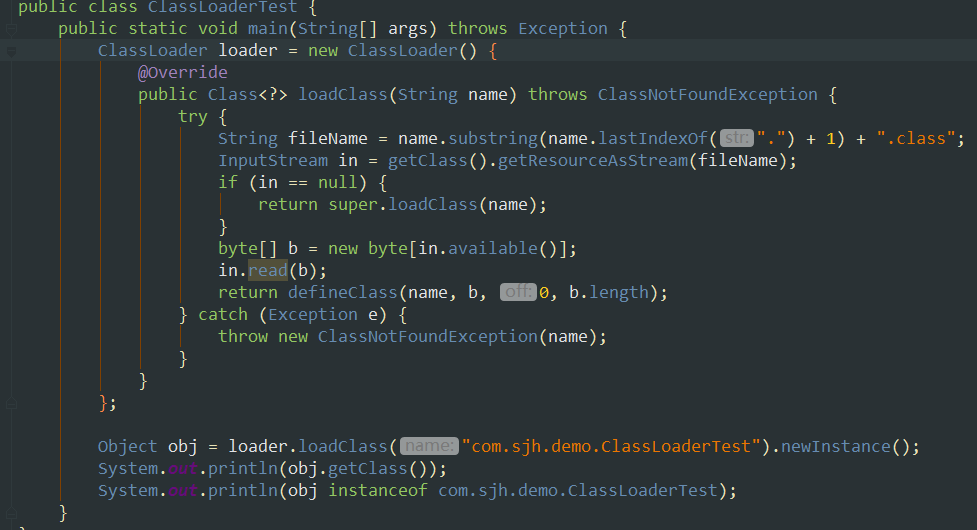
4.1、概念：

虚拟机设计团队把类加载阶段中的“通过一个类的全限定名去获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

4.2、类与类加载器（优秀博客：<https://blog.csdn.net/javazejian/article/details/73413292>）：

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就必定不相等。

这里所指的“相等”，包括代表类的Class对象的equals（）方法、isAssignableFrom（）方法、isInstance（）方法的返回结果，也包括使用instanceof关键字做对象所属关系判定等情况。



clipboard.png

4.3、双亲委派模型：

从Java虚拟机的角度来讲，只存在两种不同的类加载：一种是启动类加载器（Bootstrap ClassLoader）由C++语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些加载器由Java语言实现，独立于虚拟机外部，并且全部继承自抽象类java.lang.ClassLoader。

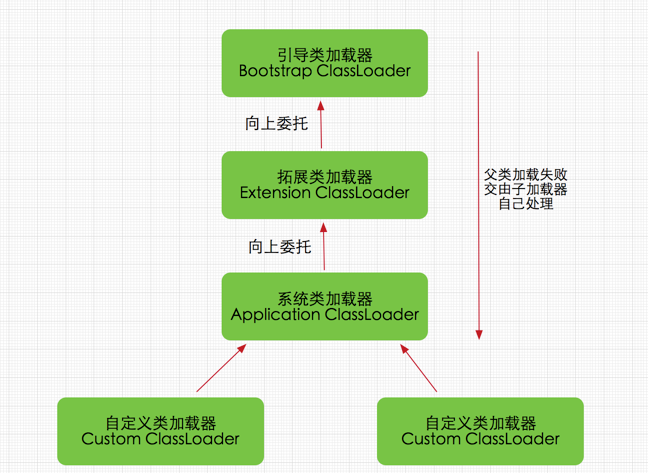
从Java开发人员的角度来看，可以细分为3种类加载器：

1）、启动类加载器（Bootstrap ClassLoader）：这个类加载器负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中，并且是虚拟机是别的类库加载到虚拟机内存中。启动类加载器无法被Java程序直接引用。

2）、扩展类加载器（Extension ClassLoader）：这个加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载<JAVA\_HOME>\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以使用扩展类加载器。

3）、应用程序类加载器（Application ClassLoader）：这个类加载器由sun.misc.Launcher$AppClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader（）方法的返回值，所以一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果程序中没有自定义自己的类加载，一般情况下这个就是程序中的默认加载器。

下图的类加载器之间的这种层次关系，称为类加载器的双亲委派模型。



双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求时（它的搜索范围内没有找到所需的类），子加载器才会尝试自己去加载。

**双亲委派模型的实现：**





正如loadClass（）方法所展示的，当类加载请求到来时，先从缓存中查找该类对象，如果存在直接返回，如果不存在则交给该类加载去的父加载器去加载，倘若没有父加载则交给顶级启动类加载器去加载，最后倘若仍没有找到，则使用findClass()方法去加载。在JDK1.2之前，在自定义类加载时，总会去继承ClassLoader类并重写loadClass方法，从而实现自定义的类加载类，但是在JDK1.2之后已不再建议用户去覆盖loadClass()方法，而是建议把自定义的类加载逻辑写在findClass()方法中，从前面的分析可知，findClass()方法是在loadClass()方法中被调用的，当loadClass()方法中父加载器加载失败后，则会调用自己的findClass()方法来完成类加载，这样就可以保证自定义的类加载器也符合双亲委托模式。需要注意的是ClassLoader类中并没有实现findClass()方法的具体代码逻辑，取而代之的是抛出ClassNotFoundException异常，同时应该知道的是findClass方法通常是和defineClass方法一起使用的。

例如自定义FileClassLoader ：





