**[Java虚拟机详解----JVM常见问题总结](http://www.cnblogs.com/smyhvae/p/4810168.html)**

**【声明】**

欢迎转载，但请保留文章原始出处→\_→

生命壹号：[http://www.cnblogs.com/smyhvae/](http://www.cnblogs.com/smyhvae/" \t "_blank)

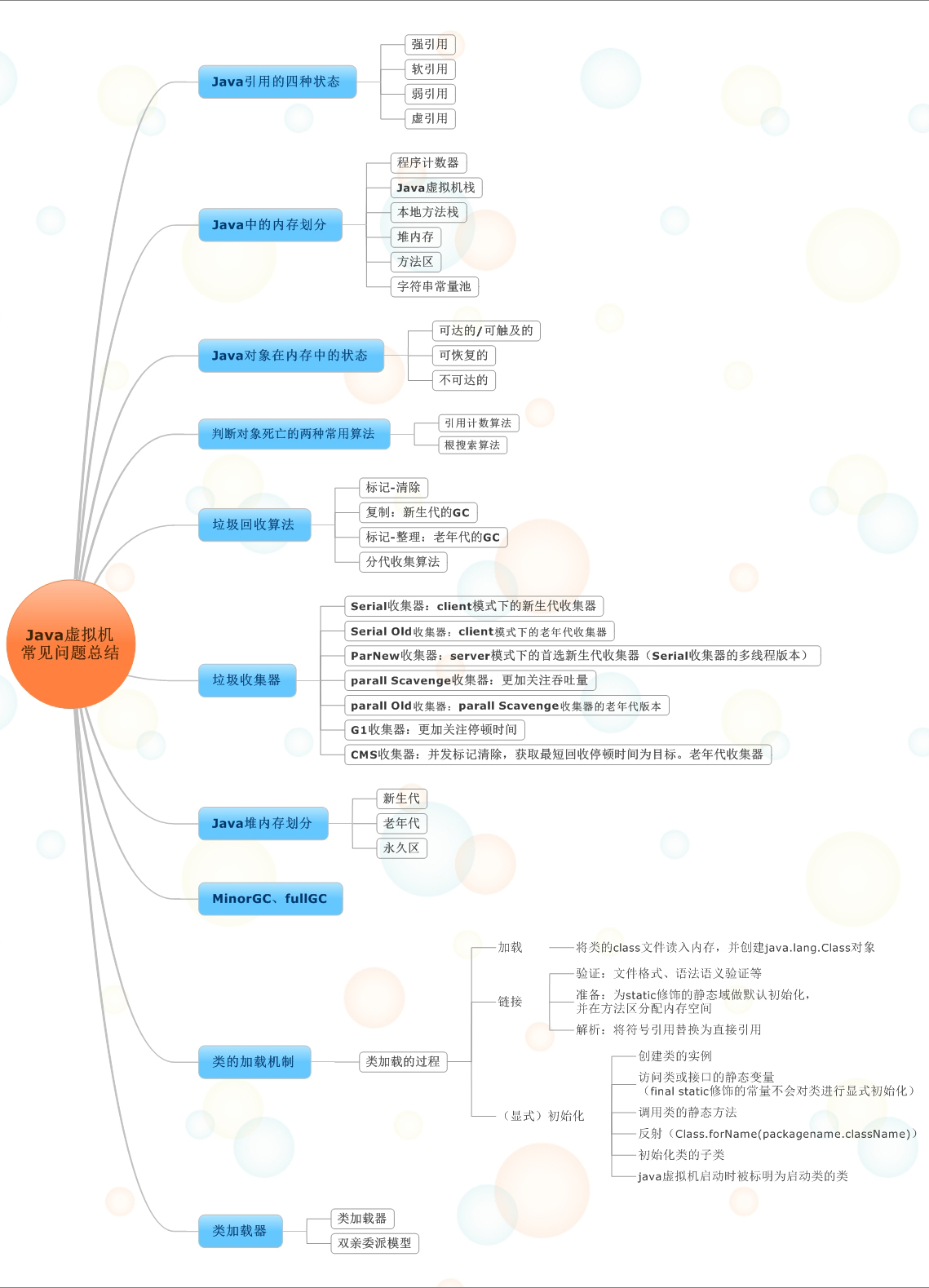
文章来源：<http://www.cnblogs.com/smyhvae/p/4810168.html>

【正文】

声明：本文只是做一个总结，有关jvm的详细知识可以参考本人之前的系列文章，尤其是那篇：[Java虚拟机详解04----GC算法和种类](http://www.cnblogs.com/smyhvae/p/4744233.html" \t "_blank)。那篇文章和本文是面试时的重点。

面试必问关键词：**JVM垃圾回收、类加载机制**。

先把本文的目录画一个思维导图：（图的源文件在本文末尾）



**一、Java引用的四种状态：**

**强引用：**

　　用的最广。我们平时写代码时，new一个Object存放在堆内存，然后用一个引用指向它，这就是强引用。

**如果一个对象具有强引用，那垃圾回收器绝不会回收它**。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**软引用：**

　　如果一个对象只具有软引用，则内存空间足够时，垃圾回收器就不会回收它；如果**内存空间不足了，就会回收这些对象的内存**。（备注：如果内存不足，随时有可能被回收。）

　　只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

**弱引用：**

　　弱引用与软引用的区别在于：**只具有弱引用的对象拥有更短暂的生命周期**。

**每次执行GC的时候**，一旦发现了只具有弱引用的对象，**不管当前内存空间足够与否，都会回收它的内存**。不过，由于垃圾回收器是一个优先级很低的线程，因此**不一定会很快发现那些只具有弱引用的对象**。

**虚引用：**

　　“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**。

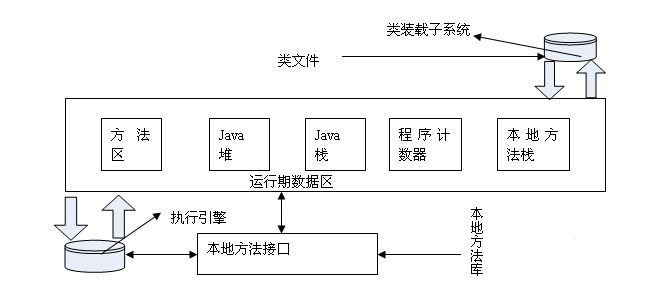
　　虚引用主要用来跟踪对象被垃圾回收器回收的活动。

注：关于各种引用的详解，可以参考这篇博客：

<http://zhangjunhd.blog.51cto.com/113473/53092>

**二、Java中的内存划分：**

Java程序在运行时，需要在内存中的分配空间。**为了提高运算效率，就对数据进行了不同空间的划分**，因为每一片区域都有特定的处理数据方式和内存管理方式。

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141609086-1837036503.jpg)

上面这张图就是jvm运行时的状态。具体划分为如下5个内存空间：（非常重要）

* 程序计数器：保证线程切换后能恢复到原来的执行位置
* 虚拟机栈：（栈内存）为虚拟机执行java方法服务：方法被调用时创建栈帧-->局部变量表->局部变量、对象引用
* 本地方法栈：为虚拟机执使用到的Native方法服务
* **堆内存：**存放**所有new出来的东西**
* **方法区：**存储被虚拟机加载的**类信息、常量、静态常量、静态方法**等。
* 运行时常量池（方法区的一部分）

**GC对它们的回收：**

内存区域中的**程序计数器、虚拟机栈、本地方法栈**这3个区域**随着线程而生，线程而灭**；**栈中的栈帧**随着方法的进入和退出而有条不紊地执行着出栈和入栈的操作，每个栈帧中分配多少内存基本是在类结构确定下来时就已知的。在这几个区域不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟着回收了。

**GC回收的主要对象**：而J**ava堆和方法区**则不同，一个接口中的多个实现类需要的内存可能不同，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，GC关注的也是这部分内存，后面的文章中如果涉及到“内存”分配与回收也仅指着一部分内存。

**1、程序计数器：（线程私有）**

每个线程拥有一个程序计数器，在线程创建时创建，

指向下一条指令的地址

执行本地方法时，其值为undefined

说的通俗一点，我们知道，Java是支持多线程的，程序先去执行A线程，执行到一半，然后就去执行B线程，然后又跑回来接着执行A线程，那程序是怎么记住A线程已经执行到哪里了呢？这就需要程序计数器了。因此**，为了线程切换后能够恢复到正确的执行位置，每条线程都有一个独立的程序计数器**，这块儿属于“线程私有”的内存。

**2、Java虚拟机栈：（线程私有）**

每个**方法被调用**的时候都会创建一个**栈帧**，用于存储局部变量表、操作栈、动态链接、方法出口等信息。局部变量表存放的是：编译期可知的基本数据类型、对象引用类型。

    每个方法被调用直到执行完成的过程，就对应着一个栈帧在虚拟机中从入栈到出栈的过程。

在Java虚拟机规范中，对这个区域规定了两种异常情况：

　　（1）如果线程请求的栈深度太深，超出了虚拟机所允许的深度，就会出现StackOverFlowError（比如无限递归。因为每一层栈帧都占用一定空间，而 Xss 规定了栈的最大空间，超出这个值就会报错）

　　（2）虚拟机栈可以动态扩展，如果扩展到无法申请足够的内存空间，会出现OOM

**3、本地方法栈：**

（1）本地方法栈与java虚拟机栈作用非常类似，其区别是：java虚拟机栈是为虚拟机执行java方法服务的，而本地方法栈则为虚拟机执使用到的**Native方法服务**。

（2）Java虚拟机没有对本地方法栈的使用和数据结构做强制规定，Sun HotSpot虚拟机就把java虚拟机栈和本地方法栈合二为一。

（3）本地方法栈也会抛出StackOverFlowError和OutOfMemoryError。

**4、Java堆：即堆内存（线程共享）**

（1）堆是java虚拟机所管理的内存区域中最大的一块，java堆是被所有线程共享的内存区域，在java虚拟机启动时创建，堆内存的唯一目的就是存放对象实例几乎所有的对象实例都在堆内存分配。

（2）**堆是GC管理的主要区域**，从垃圾回收的角度看，由于现在的垃圾收集器都是采用的**分代收集**算法，因此java堆还可以初步细分为**新生代和老年代**。

（3）Java虚拟机规定，堆可以处于物理上不连续的内存空间中，只要逻辑上连续的即可。在实现上既可以是固定的，也可以是可动态扩展的。如果在堆内存没有完成实例分配，并且堆大小也无法扩展，就会抛出OutOfMemoryError异常。

**5、方法区：（线程共享）**

（1）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。

（2）Sun HotSpot虚拟机把方法区叫做永久代（Permanent Generation），方法区中最终要的部分是运行时常量池。

**6、运行时常量池：**

（1）运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时就会抛出OutOfMemoryError异常。

注：关于本段的详细内容，可以参考本人的另外一篇博客：[Java虚拟机详解02----JVM内存结构](http://www.cnblogs.com/smyhvae/p/4748392.html" \t "_blank)

**三、Java对象在内存中的状态：**

**可达的/可触及的：**

　　Java对象被创建后，如果被一个或多个变量引用，那就是可达的。即从根节点可以触及到这个对象。

　　其实就是从根节点扫描，只要这个对象在引用链中，那就是可触及的。

**可恢复的：**

　　Java对象不再被任何变量引用就进入了可恢复状态。

　　在回收该对象之前，该对象的finalize()方法进行资源清理。如果**在finalize()方法中重新让变量引用该对象，则该对象再次变为可达状态**，否则该对象进入不可达状态

**不可达的：**

　　Java对象不被任何变量引用，且系统在调用对象的finalize()方法后依然没有使该对象变成可达状态（该对象依然没有被变量引用），那么该对象将变成不可达状态。

　　当Java对象处于不可达状态时，系统才会真正回收该对象所占有的资源。

**四、判断对象死亡的两种常用算法：**

    当对象不被引用的时候，这个对象就是死亡的，等待GC进行回收。

**1、引用计数算法：**

**概念：**

　　给对象中添加一个引用计数器，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0的对象就是不可能再被使用的。

**但是：**

　　主流的java虚拟机并没有选用引用计数算法来管理内存，其中最主要的原因是：**它很难解决对象之间相互循环引用的问题**。

**优点：**

　　算法的实现简单，判定效率也高，大部分情况下是一个不错的算法。很多地方应用到它

**缺点：**

引用和去引用伴随加法和减法，影响性能

致命的缺陷：对于**循环引用的对象**无法进行回收

**2、根搜索算法：**（jvm采用的算法）

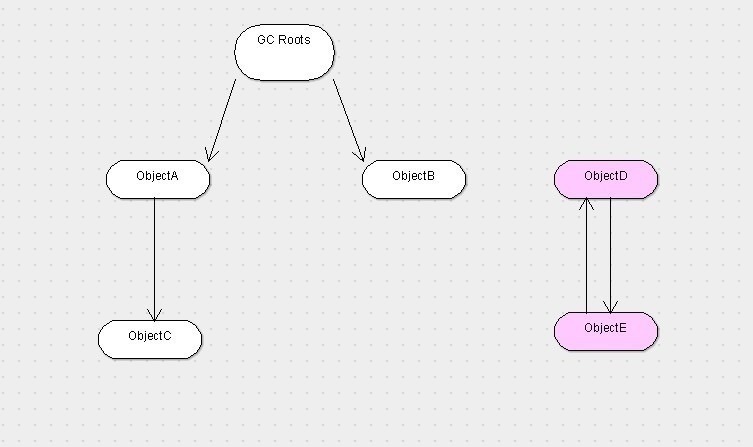
**概念：**

　　设立若干种根对象，当任何一个根对象（GC Root）到某一个对象均不可达时，则认为这个对象是可以被回收的。

**注：**这里提到，设立若干种根对象，当任何一个根对象到某一个对象均不可达时，则认为这个对象是可以被回收的。我们在后面介绍标记-清理算法/标记整理算法时，也会一直强调从根节点开始，对所有可达对象做一次标记，那什么叫做可达呢？

**可达性分析：**

　　从根（GC Roots）的对象作为起始点，开始向下搜索，搜索所走过的路径称为“**引用链**”，当一个对象到GC Roots没有任何引用链相连（用图论的概念来讲，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。

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141611679-1989614075.jpg)

如上图所示，ObjectD和ObjectE是互相关联的，但是由于GC roots到这两个对象不可达，所以最终D和E还是会被当做GC的对象，上图若是采用引用计数法，则A-E五个对象都不会被回收。

**根（GC Roots）：**

说到GC roots（GC根），在JAVA语言中，可以当做GC roots的对象有以下几种：

1、**栈**（栈帧中的本地变量表）**中引用的对象**。

2、方法区中的静态成员。

3、方法区中的常量引用的对象（全局变量）

4、本地方法栈中JNI（一般说的Native方法）引用的对象。

注：第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

在根搜索算法的基础上，现代虚拟机的实现当中，**垃圾搜集的算法**主要有三种，分别是**标记-清除算法**、**复制算法**、**标记-整理算法**。这三种算法都扩充了根搜索算法，不过它们理解起来还是非常好理解的。

**五、垃圾回收算法：**

**1、标记-清除算法：**

**概念：**

标记阶段：**先通过根节点，标记所有从根节点开始的可达对象**。因此，未被标记的对象就是未被引用的垃圾对象；

清除阶段：清除所有未被标记的对象。

**缺点：**

标记和清除的过程**效率不高**（标记和清除都需要从头遍历到尾）

标记清除后**会产生大量不连续的碎片**。

**2、复制算法：（新生代的GC）**

概念：

　　将原有的**内存空间分为两块**，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的**存活对象**复制到未使用的内存块中，然后清除正在使用的内存块中的所有对象。

**优点：**

这样使得每次都是对整个半区进行回收，内存分配时也就**不用考虑内存碎片**等情况

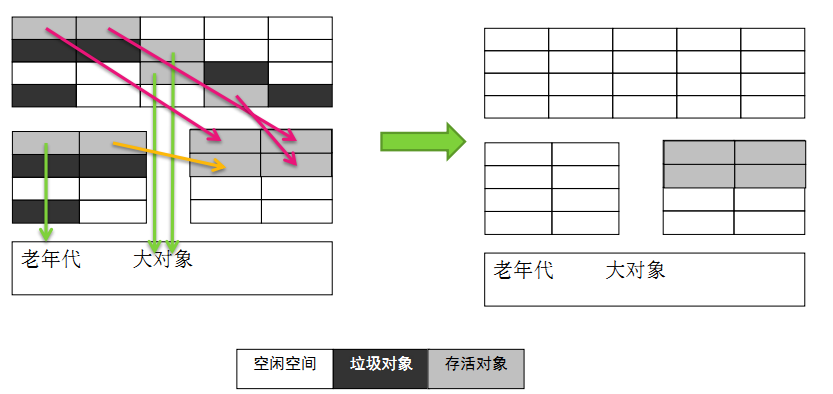
只要移动堆顶指针，按顺序分配内存即可，实现简单，**运行效率高**

**缺点：空间的浪费**

　　从以上描述不难看出，复制算法要想使用，最起码对象的存活率要非常低才行。

　　现在的商业虚拟机都采用这种收集算法来回收新生代，新生代中的对象98%都是“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**内存分为一块比较大的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和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是说，每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的空间会被浪费。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，**当Survivor空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代**。整个过程如下图所示：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141614148-131403768.png)

**3、标记-整理算法：（老年代的GC）**

    复制算法在对象**存活率高**的时候要进行较多的复制操作，效率将会降低，所以在老年代中一般不能直接选用这种算法。

**概念：**

标记阶段：先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象

整理阶段：**将将所有的存活对象压缩到内存的一端**；之后，清理边界外所有的空间

**优点：**

　　不会产生内存碎片。

**缺点：**

　　在标记的基础之上还需要进行对象的移动，成本相对较高，效率也不高。

它们的区别如下：（>表示前者要优于后者，=表示两者效果一样）

（1）效率：复制算法 > 标记/整理算法 > 标记/清除算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。

（2）内存整齐度：复制算法=标记/整理算法>标记/清除算法。

（3）内存利用率：标记/整理算法=标记/清除算法>复制算法。

注1：标记-整理算法不仅可以弥补标记-清除算法当中，内存区域分散的缺点，也消除了复制算法当中，内存减半的高额代价。

注2：可以看到标记/清除算法是比较落后的算法了，但是后两种算法却是在此基础上建立的。

注3：时间与空间不可兼得。

**4、分代收集算法：**

　　当前商业虚拟机的GC都是采用的“分代收集算法”，这并不是什么新的思想，**只是根据对象的存活周期的不同将内存划分为几块儿**。一般是把Java堆分为新生代和老年代：**短命对象归为新生代，长命对象归为老年代**。

* **存活率低：少量对象存活，适合复制算法**：在新生代中，每次GC时都发现有大批对象死去，只有少量存活（新生代中98%的对象都是“朝生夕死”），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成GC。
* 存活率高：大量对象存活，适合用标记-清理/标记-整理：在老年代中，因为对象存活率高、没有额外空间对他进行分配担保，就必须使用“标记-清理”/“标记-整理”算法进行GC。

注：**老年代的对象中，有一小部分是因为在新生代回收时，老年代做担保，进来的对象；绝大部分对象是因为很多次GC都没有被回收掉而进入老年代**。

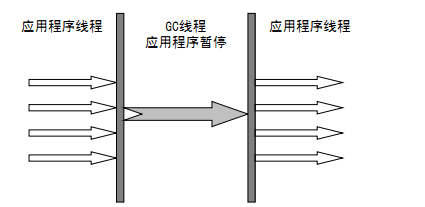
**六、垃圾收集器：**

如果说收集算法时内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们在对各种收集器进行比较，但并非为了挑出一个最好的收集器。因为直到现在位置还没有最好的收集器出现，更加没有万能的收集器，所以我们**选择的只是对具体应用最合适的收集器**。

**1、Serial收集器：（串行收集器）**

这个收集器是一个单线程的收集器，但它的单线程的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程（**Stop-The-World：将用户正常工作的线程全部暂停掉**），直到它收集结束。收集器的运行过程如下图所示：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141616039-1927452153.png)

上图中：

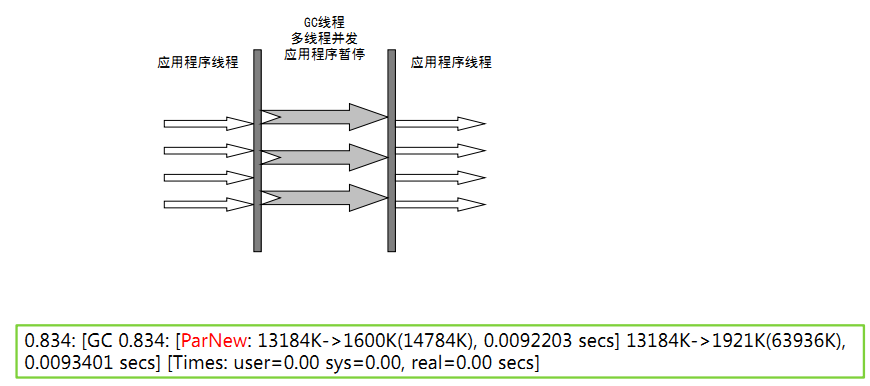
* 新生代采用复制算法，Stop-The-World
* 老年代采用标记-整理算法，Stop-The-World

当它进行GC工作的时候，虽然会造成Stop-The-World，但它存在有存在的原因：正是因为它的简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，没有线程交互的开销，专心做GC，自然可以获得最高的单线程手机效率。所以**Serial收集器对于运行在client模式下是一个很好的选择**（它依然是虚拟机运行在**client模式下**的默认**新生代**收集器）。

**2、ParNew收集器：**Serial收集器的多线程版本（使用多条线程进行GC）

　　ParNew收集器是Serial收集器的多线程版本。

**它是运行在server模式下的首选新生代收集器**，除了Serial收集器外，目前只有它能与CMS收集器配合工作。CMS收集器是一个被认为具有划时代意义的并发收集器，因此如果有一个垃圾收集器能和它一起搭配使用让其更加完美，那这个收集器必然也是一个不可或缺的部分了。收集器的运行过程如下图所示：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141617742-295619379.png)

上图中：

* 新生代采用复制算法，Stop-The-World
* 老年代采用标记-整理算法，Stop-The-World

**3、ParNew Scanvenge收集器**

　　类似ParNew，但更加**关注吞吐量。**目标是：达到一**个可控制吞吐量**的收集器。

**停顿时间和吞吐量不可能同时调优**。我们一方买希望停顿时间少，另外一方面希望吞吐量高，其实这是矛盾的。因为：在GC的时候，垃圾回收的工作总量是不变的，如果将停顿时间减少，那频率就会提高；既然频率提高了，说明就会频繁的进行GC，那吞吐量就会减少，性能就会降低。

**吞吐量**：CPU用于用户代码的时间/CPU总消耗时间的比值，即=运行用户代码的时间/(运行用户代码时间+垃圾收集时间)。比如，虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

**4、G1收集器：**

　　是当今收集器发展的最前言成果之一，知道jdk1.7，sun公司才认为它达到了足够成熟的商用程度。

**优点：**

　　它最大的优点是结合了空间整合，不会产生大量的碎片，也降低了进行gc的频率。

　　二是可以让使用者**明确指定指定停顿时间**。（可以指定一个最小时间，超过这个时间，就不会进行回收了）

它有了这么高效率的原因之一就是：对垃圾回收**进行了划分优先级的操作**，这种有优先级的区域回收方式保证了它的高效率。

如果你的应用追求停顿，那G1现在已经可以作为一个可尝试的选择；如果你的应用追求吞吐量，那G1并不会为你带来什么特别的好处。

注：以上所有的收集器当中，当执行GC时，都会stop the world，但是下面的CMS收集器却不会这样。

**5、CMS收集器：（老年代收集器）**

CMS收集器（Concurrent Mark Sweep：**并发标记清除**）是一种**以获取最短回收停顿时间为目标**的收集器。适合应用在互联网站或者B/S系统的服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。

**CMS收集器运行过程：**（着重实现了**标记**的过程）

（1）初始标记

　　根可以直接关联到的对象

　　速度快

（2）并发标记（和用户线程一起）

　　主要标记过程，标记全部对象

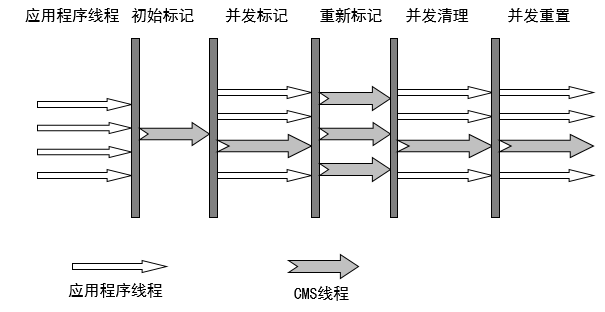
（3）重新标记

　　由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正

（4）并发清除（和用户线程一起）

　　基于标记结果，直接清理对象

整个过程如下图所示：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141620445-1626423736.png)

上图中，初始标记和重新标记时，需要stop the world。整个过程中耗时最长的是并发标记和并发清除，这两个过程都可以和用户线程一起工作。

**优点：**

　　并发收集，低停顿

**缺点：**

（1）导致用户的执行速度降低。

（2）无法处理浮动垃圾。因为它采用的是标记-清除算法。有可能有些垃圾在标记之后，需要等到下一次GC才会被回收。如果CMS运行期间无法满足程序需要，那么就会临时启用Serial Old收集器来重新进行老年代的手机。

（3）由于采用的是标记-清除算法，那么就会产生大量的碎片。往往会出现老年代还有很大的空间剩余，但是无法找到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

**疑问：既然标记-清除算法会造成内存空间的碎片化，CMS收集器为什么使用标记清除算法而不是使用标记整理算法：**

**答案：**

　　CMS收集器更加关注停顿，它在**做GC的时候是和用户线程一起工作的**（并发执行），如果使用标记整理算法的话，那么在清理的时候就会去移动可用对象的内存空间，那么**应用程序的线程就很有可能找不到应用对象在哪里**。

**七、Java堆内存划分：**

根据对象的存活率（年龄），Java对内存划分为3种：新生代、老年代、永久代：

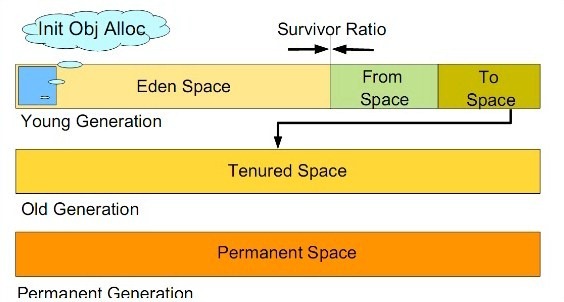
**1、新生代：**

比如我们在方法中去new一个对象，那这方法调用完毕后，对象就会被回收，这就是一个典型的新生代对象。

现在的商业虚拟机都采用这种收集算法来回收新生代，新生代中的对象98%都是“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**内存分为一块比较大的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和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是说，每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的空间会被浪费。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，**当Survivor空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代**。同时，**长期存活的对象将进入老年代（**虚拟机给每个对象定义一个年龄计数器）。

来看下面这张图：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141622289-1667125770.jpg)

**Minor GC和Full GC：**

GC分为两种：Minor GC和Full GC

**Minor GC：**

　　Minor GC是发生在新生代中的垃圾收集动作，采用的是复制算法。

对象在Eden和From区出生后，在经过一次Minor GC后，如果对象还存活，并且能够被to区所容纳，那么在使用复制算法时这些存活对象就会被复制到to区域，然后清理掉Eden区和from区，并将这些对象的年龄设置为1，以后对象在Survivor区每熬过一次Minor GC，就将对象的年龄+1，当对象的年龄达到某个值时（默认是15岁，可以通过参数 --XX:MaxTenuringThreshold设置），这些对象就会成为老年代。

但这也是不一定的，对于一些较大的对象（即需要分配一块较大的连续内存空间）则是直接进入老年代

**Full GC：**

　　Full GC是发生在老年代的垃圾收集动作，采用的是标记-清除/整理算法。

老年代里的对象几乎都是在Survivor区熬过来的，不会那么容易死掉。因此Full GC发生的次数不会有Minor GC那么频繁，并且做一次Full GC要比做一次Minor GC的时间要长。

另外，如果采用的是标记-清除算法的话会产生许多碎片，此后如果需要为较大的对象分配内存空间时，若无法找到足够的连续的内存空间，就会提前触发一次GC。

**2、老年代：**

    在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中。而且大对象直接进入老年代。

**3、永久代：**

    即方法区。

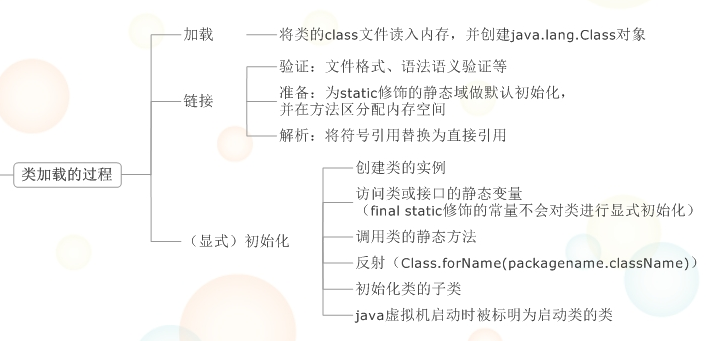
**八、类加载机制：**

    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、转换解析和初始化，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。

**类加载的过程：**

    包括加载、链接（含验证、准备、解析）、初始化

如下图所示：



**1、加载：**

　　类加载指的是将类的class文件读入内存，并为之创建一个java.lang.**Class对象**，作为方法区**这个类的数据访问的入口**。

也就是说，当程序中使用任何类时，系统都会为之建立一个java.lang.Class对象。具体包括以下三个部分：

（1）通过**类的全名**产生对应类的二进制数据流。（根据early load原理，如果没找到对应的类文件，只有在类实际使用时才会抛出错误）

（2）分析并将这些二进制数据流转换为方法区方法区特定的数据结构

（3）创建对应类的java.lang.Class对象，作为方法区的入口（有了对应的Class对象，并不意味着这个类已经完成了加载链接）

通过使用不同的类加载器，可以从不同来源加载类的二进制数据，通常有如下几种来源：

（1）从本地文件系统加载class文件，这是绝大部分程序的加载方式

（2）从jar包中加载class文件，这种方式也很常见，例如jdbc编程时用到的数据库驱动类就是放在jar包中，jvm可以从jar文件中直接加载该class文件

（3）通过网络加载class文件

（4）把一个Java源文件动态编译、并执行加载

**2、链接：**

    链接指的是将Java类的二进制文件合并到jvm的运行状态之中的过程。在链接之前，这个类必须被成功加载。

类的链接包括**验证、准备、解析**这三步。具体描述如下：

**2.1  验证：**

    验证是用来确保Java类的二进制表示在结构上是否完全正确（如文件格式、语法语义等）。如果验证过程出错的话，会抛出java.lang.VertifyError错误。

主要验证以下内容：

* 文件格式验证
* 元数据验证：语义验证
* 字节码验证

**2.2  准备：**

　　准备过程则是创建Java类中的**静态**域（static修饰的内容），并将这些域的值设置为**默认值**，同时在**方法区中分配内存空间**。准备过程并不会执行代码。

注意这里是做默认初始化，不是做显式初始化。例如：

public static int value = 12;

上面的代码中，在**准备阶段**，会给value的值设置为0（**默认初始化**）。在后面的**初始化阶段**才会给value的值设置为12（**显式初始化**）。

**2.3  解析：**

　　解析的过程就是确保这些被引用的类能被正确的找到（将符号引用替换为直接引用）。解析的过程可能会导致其它的Java类被加载。

**3、初始化：**

　　初始化阶段是类加载过程的最后一步。到了初始化阶段，才真正执行类中定义的Java程序代码（或者说是字节码）。

在以下几种情况中，会执行初始化过程：

（1）创建类的实例

（2）访问类或接口的静态变量（**特例：如果是用static final修饰的常量，那就不会对类进行显式初始化。static final 修改的变量则会做显式初始化**）

（3）调用类的静态方法

（4）反射（Class.forName(packagename.className)）

（5）初始化类的子类。注：子类初始化问题：满足主动调用，即**父类访问子类中的静态变量、方法，子类才会初始化**；否则仅父类初始化。

（6）java虚拟机启动时被标明为启动类的类

**代码举例1：**

我们对上面的第（5）种情况做一个代码举例。

(1)Father.java:

[复制代码](javascript:void(0);)

1 public class Father {

2

3 static {

4 System.out.println("\*\*\*\*\*\*\*father init");

5 }

6 public static int a = 1;

7 }

[复制代码](javascript:void(0);)

(2)Son.java:

1 public class Son extends Father {

2 static {

3 System.out.println("\*\*\*\*\*\*\*son init");

4 }

5 public static int b = 2;

6 }

(3)JavaTest.java:

1 public class JavaTest {

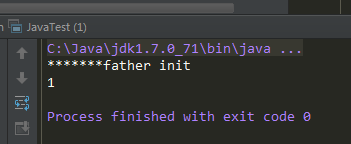
2 public static void main(String[] args) {

3 System.out.println(Son.a);

4 }

5 }

上面的测试类中，虽然用上了Son这个类，但是并没有调用子类里的成员，所以并不会对子类进行初始化。于是运行效果是：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141624148-1667561406.png)

如果把JavaTest.java改成下面这个样子：

1 public class JavaTest {

2 public static void main(String[] args) {

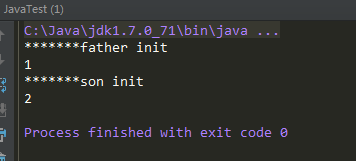
3 System.out.println(Son.a);

4 System.out.println(Son.b);

5 }

6 }

运行效果：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141625039-1480148250.png)

如果把JavaTest.java改成下面这个样子：

JavaTest.java:

1 public class JavaTest {

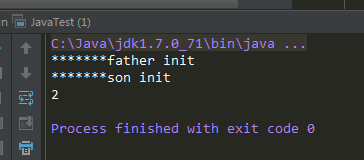
2 public static void main(String[] args) {

3 System.out.println(Son.b);

4 }

5 }

运行效果：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141626383-178038035.png)

**代码举例2：**

我们对上面的第（2）种情况做一个代码举例。即：如果是用static final修饰的常量，则不会进行显式初始化。代码举例如下：

(1)Father.java:

1 public class Father {

2 static {

3 System.out.println("\*\*\*\*\*\*\*father init");

4 }

5 public static int a = 1;

6 }

(2)Son.java:

[复制代码](javascript:void(0);)

1 public class Son extends Father {

2 static {

3 System.out.println("\*\*\*\*\*\*\*son init");

4 }

5

6 public static int b = 2;

7 **public static final int c = 3**;

8 }

[复制代码](javascript:void(0);)

这里面的变量c是一个静态常量。

(3)JavaTest.java:

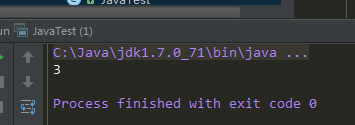
1 public class JavaTest {

2 public static void main(String[] args) {

3 System.out.println(Son.c);

4 }

5 }

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141627570-86669376.png)

上面的运行效果显示，**由于c是final static修饰的静态常量，所以根本就没有调用静态代码块里面的内容，也就是说，没有对这个类进行显式初始化**。

现在，保持Father.java的代码不变。将Son.java代码做如下修改：

[复制代码](javascript:void(0);)

1 public class Son extends Father {

2 static {

3 System.out.println("\*\*\*\*\*\*\*son init");

4 }

5

6 public static int b = 2;

7 **public static final int c = new Random().nextInt(3);**

8 }

[复制代码](javascript:void(0);)

JavaTest.java:

1 public class JavaTest {

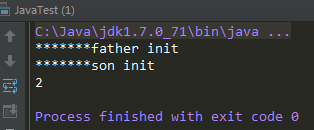
2 public static void main(String[] args) {

3 System.out.println(Son.c);

4 }

5 }

运行效果如下：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141628742-508669910.png)

**代码举例3：（很容易出错）**

我们来下面这段代码的运行结果是什么：

[复制代码](javascript:void(0);)

1 public class TestInstance {

2

3 **public static TestInstance instance = new TestInstance();**

4 public static int a;

5 public static int b = 0;

6

7 public TestInstance() {

8 a++;

9 b++;

10 }

11

12 public static void main(String[] args) {

13 System.out.println(TestInstance.a);

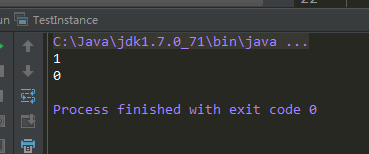
14 System.out.println(TestInstance.b);

15 }

16 }

[复制代码](javascript:void(0);)

运行结果：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141630664-2052544103.png)

之所以有这样的运行结果，这里涉及到类加载的顺序：

（1）在加载阶段，加载类的信息

（2）在链接的准备阶段给instance、a、b做默认初始化并分配空间，此时a和b的值都为0

（3）在初始化阶段，执行构造方法，此时a和b的值都为1

（4）在初始化阶段，给静态变量做显式初始化，此时b的值为0

我们改一下代码的执行顺序，改成下面这个样子：

[复制代码](javascript:void(0);)

1 public class TestInstance {

2

3 public static int a;

4 public static int b = 0;

5 **public static TestInstance instance = new TestInstance();**

6

7 public TestInstance() {

8 a++;

9 b++;

10 }

11

12 public static void main(String[] args) {

13 System.out.println(TestInstance.a);

14 System.out.println(TestInstance.b);

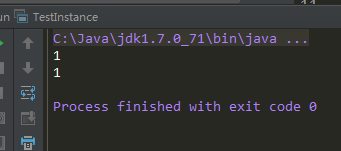
15

16 }

17 }

[复制代码](javascript:void(0);)

运行效果是：

[](http://images2015.cnblogs.com/blog/641601/201509/641601-20150915141631695-715996760.png)

之所以有这样的运行结果，这里涉及到类加载的顺序：

（1）在加载阶段，加载类的信息

（2）在链接的准备阶段给instance、a、b做默认初始化并分配空间，此时a和b的值都为0

（3）在初始化阶段，给静态变量做显式初始化，此时b的值仍为0

（4）在初始化阶段，执行构造方法，此时a和b的值都为1

注意，这里涉及到另外一个类似的知识点不要搞混了。知识点如下。

**知识点：类的初始化过程（重要）**

Student s = new Student();在内存中做了哪些事情?

* 加载Student.class文件进内存
* 在**栈内存**为s开辟空间
* 在**堆内存**为学生对象开辟空间
* 对学生对象的成员变量进行**默认初始化**
* 对学生对象的成员变量进行**显示初始化**
* 通过**构造方法**对学生对象的**成员变量赋值**
* 学生对象初始化完毕，把**对象地址赋值给s变量**

【思维导图文件下载地址】