*上一次的学习笔记讲述的主要是是JVM中最基本的内容，比如说如何分区，以及各分区的作用，包含了Java 程序的什么信息等，最后融会贯通了各个分区在程序运行时如何交互。然后在多线程学习笔记中，我们结合volatile关键字，学习了JVM内存模型(JMM)，通过硬件的角度，了解了Java中线程的工作内存是如何与主存进行交互的，明白了三个重要的概念：原子性，可见性，顺序性。我们可以发现JVM的学习其实是和多线程息息相关的，两者在内存模型，线程运行时对应的其实就是JVM中的栈等等都是有密切联系的。*

*这次，我们来学习一下GC(Garbage Collection)垃圾回收机制，看看GC中都有哪些算法，如何判断哪些东西是垃圾的*。

1. **GC Root**

**从名字就可以知道，GC Root就是判断对象可达性的根节点，告诉了我们垃圾回收从哪个对象开始。只有知道了GC Root是哪个，我们才能利用后续的GC算法，根据GC Root判断对象是否可达，然后做对象的回收工作。GC Root(根节点，根对象)GC中相当重要的一点，那到底什么是GC Root？**

* **所有当前正在运行的Java线程中活跃的栈帧里指向堆中对象的引用，换句话说，就是当前正在执行的所有方法中的对象的引用**
* **静态变量修饰的对象的引用**
* **运行时常量池的引用类型常量(String)**
* **当前被加载的Java类**

**这里需要注意一点，上述提到的全部都是活跃的引用，不是对象本身。**

1. **GC算法**

* **Tracing GC**

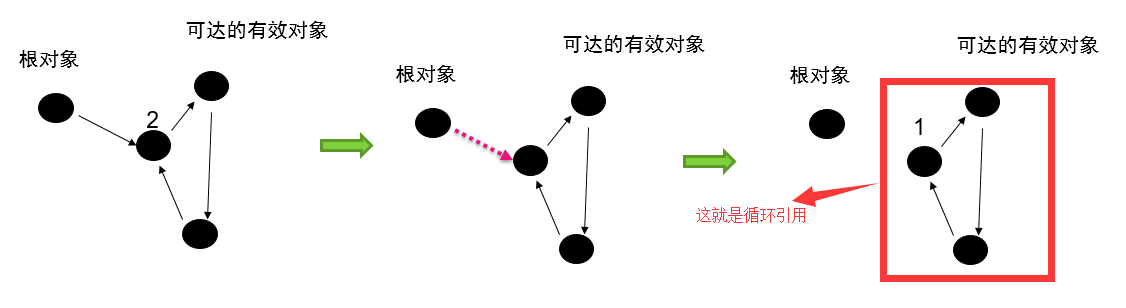
**其实就是通过GC Root的集合，通过对象之间的引用关系遍历对象关系图，确认对象的可达性。上述的GC Root就是Tracing GC的起点。**

* 引用**计数**法(没有被Java使用)

引用计数法非常简单粗暴，他就是直接统计某个对象当前还有多少个其他对象在**引用**他，只要有一个，此对象的引用就**++**一个**计数**。当一个引用消失，引用数量就**--**一个**计数**。如果**计数**为0，那么这个对象就会被回收。

虽然简单粗暴，但是引用计数法存在几个缺点：

1. 频繁的引用的去引用，伴随着频繁的加减，**影响性能**
2. 很难处理**循环引用**的情况。什么是循环引用？如下图：



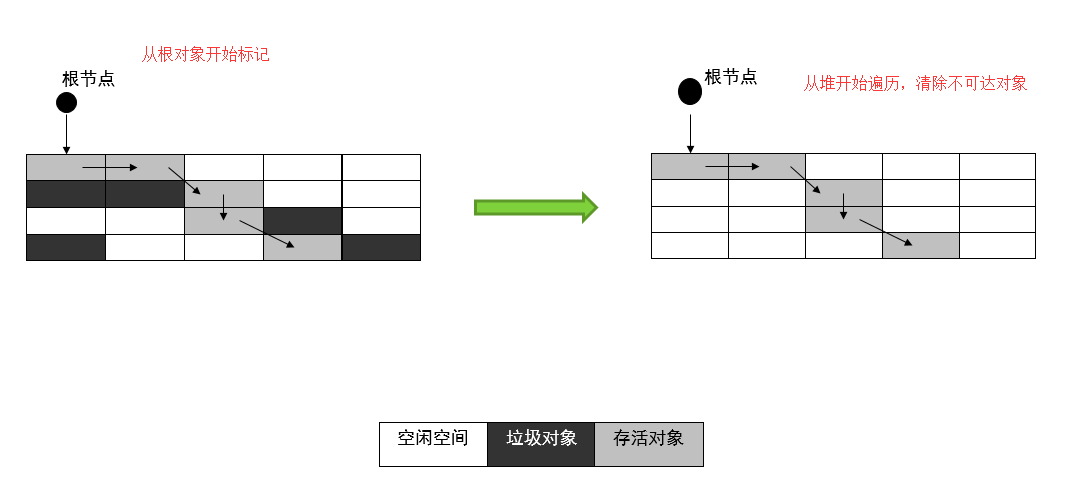
由于三个对象都有彼此的引用，它们明明已经从根对象不可达了，但是仍然没有被回收。

* 标记—清除

含义很明显，是通过先标记引用对象，后清除非引用对象来实现的。

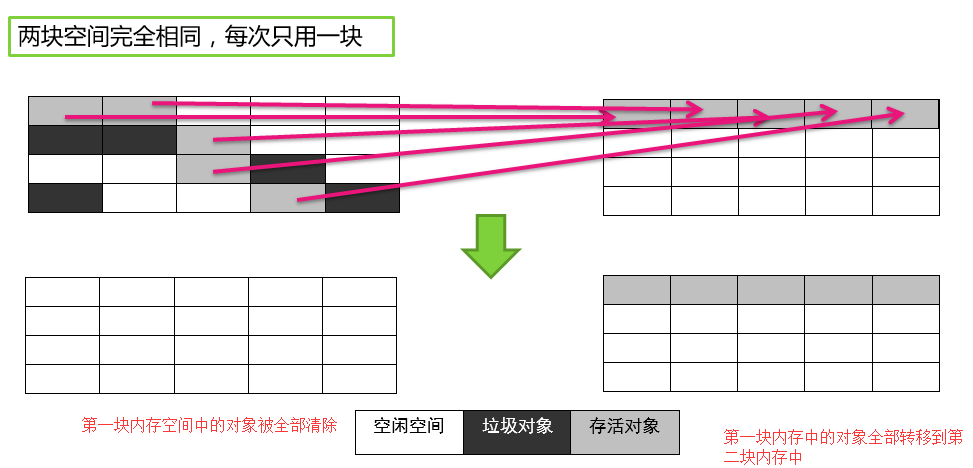
标记过程中，我们从根对象(根节点)出发，标记出所有可达(被引用)对象

清除过程中，遍历整个堆，将不可达的对象直接清除。



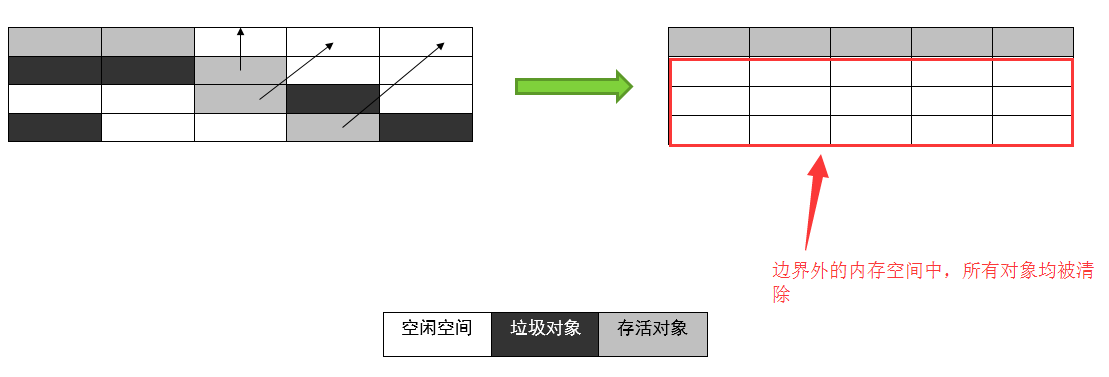
* 复制(压缩)

开辟一块大小相同的空间，遍历第一块空间，把里边所有的正在使用中的对象放在另一块空间中，然后把第一块内存空间中的所有对象都清除。



* 标记—压缩

标记—压缩就相当于是标记清除和复制的结合体，第一次先标记那些对象是正在使用中的对象，第二次通过复制压缩，把所有存活对象压缩到内存中的一端，然后清理边界外的所有对象。

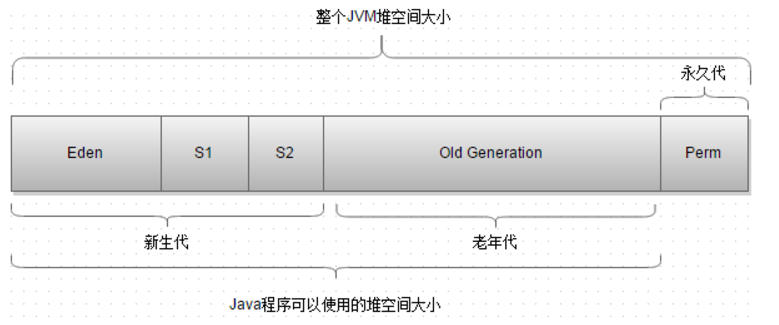


**2. 年代划分**

* 刚才描述了许多GC算法，有这么多GC算法，使用哪种GC是根据分代情况来决定的
* 根据对象的生命周期长短，我们把对象划分为：**新生代**，**老年代，永久带。**由于永久带在1.8中就没了，因此我们就先不说了。顾名思义，**生命周期短**的对象就是新生代，**生命周期长**的对象叫做老年代。
* 什么是新生代？什么是老年代？堆中是如何划分新生代和老年代的？

对象存在于堆中，刚**new**出来的对象大多属于**新生代**，在一次**GC发生之后（新生代发生的GC称为Minor GC，这个等下再讲）**，如果此对象**还存活**并且达到了**老年代的年龄**，这个对象就可以移步到老年代中。

* 新生代的构成



**如图**，s1和s2是新生代用于采用复制算法的(s1和s2也成为**survivor**区)，必须有一块为空，来进行from-to的拷贝。

**新生代**由一个**Eden(**伊甸)区和两个**survivor**区构成的，默认比例大小为8:1

1. 刚被创建的对象放在Eden区
2. Eden区第一次GC之后，幸存的对象移动到**其中一个**幸存者空间(**这是什么算法？Minor GC**)
3. 每次Eden区发生GC，都把幸存的的对象放到刚才的那个幸存者空间中去
4. 当其中一个幸存者空间满了，就把仍然存活的对象放到另一个幸存者空间里(这是GC之中的**压缩**算法，也叫Minor GC)，然后把前一个幸存者空间清空，因此**总有一个survivor空间是空的**
5. 重复以上步骤依然存活的对象，进入老年代

* 新生代对象如何转化成老年代对象？

1. **大对象**直接进入老年代(多大的对象算大对象？)
2. 新生代发生Minor GC之后仍然存活的对象

* 结合刚才描述到的GC算法来说，**少量对象存活**，使用压缩算法，**大量对象存活**，使用标记清除和标记压缩算法。

**新生代**采用**压缩**复制算法，**效率高，老年代**使用**标记清除**和**标记压缩**算法，**效率低**。

**3. 垃圾回收的起点**

栈是真正进行程序执行地方，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他对象的引用，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**栈中的引用的对象**, **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。

**4. 伴随GC所产生的问题**

* 如何处理**内存碎片**？

当程序执行了一段时间之后，经过了许多次的对象的创建和对象的收集。如果不进行内存碎片的**整理**，许多非常小的内存碎片就不能组合起来存放大一点的对象，会造成**内存浪费**。

* GC的过程中整个应用程序会**停止运行**，如何让对象的创建和对象的回收合理进行？

在一条线程中，垃圾收集不断地收集对象，程序运行的线程是不断生成对象从而消耗内存的，这个是很矛盾的。在你收集垃圾的时候，不断有新对象创建，那我怎么判断哪个对象是垃圾？我怎么从根节点遍历出对象的可达性？

因此，在GC过程中，整个应用停止运行了。

那如果我的堆非常大有很多很多对象，那GC进行的时候岂不是会浪费很多时间吗？

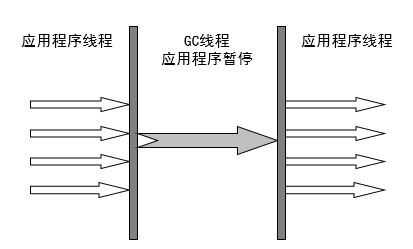
由此，并发收集就应运而生了。

当然，这个并发收集说的是**垃圾回收变为多线程**，而不是边收集垃圾边产生对象。

**5. 收集器的分类**

* 串行收集器(Serial)

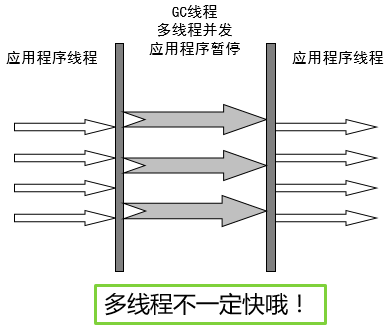
这是最古老的一种垃圾收集器，比较稳定。但是存在**Stop-The-World**，会造成应用的**停顿**，而且还是单线程进行垃圾回收，速度比较慢

。

* 并行收集器1(ParNew)

从名字就可以看出来，New代表了此收集器处理的是**新生代**对象的垃圾收集。

ParNew收集器在**新生代**采用**并行**，**老年代**采用**串行**，也就是第一种。



当然，多线程并不一定就比单线程要快，因为多个线程的启动和销毁也是会消耗性能的，在**单个CPU**上，我们还是推荐使用串行收集器。

* 并行收集器2(Parallel **Scavenge**)

这个也是个并行收集器，其实和ParNew差不多。新生代采用复制压缩算法，老年代采用标记-压缩算法。和ParNew收集器只在新生代采用并行相同，Parallel Scavenge在老年代是串行单线程的。

说道ParNew和Parallel Scavenge的区别，，Parallel更加关注吞吐量。怎么说这个吞吐量呢？

**吞吐量=代码执行时间/代码执行时间+GC垃圾收集时间**

由此可以看出，GC时间越短，吞吐量越大。但是GC所要执行完成的**任务量**肯定是**一致**的，如果缩短了每一次GC所执行的时间，那么GC执行的**次数**肯定就会**变多**。

* Serial Old

对比串行收集器，这个是收集老年代对象的，使用标记-压缩算法进行回收。

* Parallel Old

Parallel Scavenge的老年代版本，一般配合Parallel Scavenge使用。

* CMS收集器

1. 与前边的几个并行收集器不同，CMS收集器是**并发**收集器。**并行**收集器指的是在进行GC的时候**多线程的进行垃圾回收**，而并发收集器指的是**GC垃圾回收与应用线程(**创建对象线程)**同时**的执行。

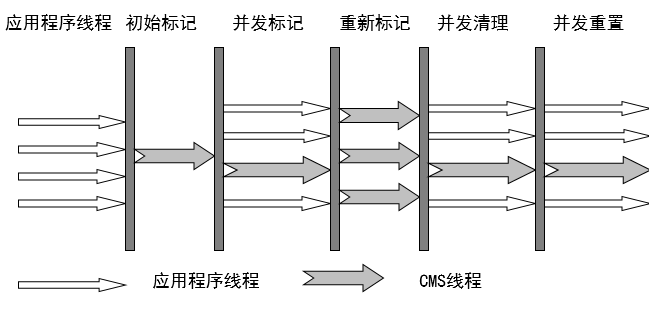
2. CMS收集器采用标记-清除的方法，来处理**老年代**区间的对象。由于CMS里GC和应用线程是同时进行的(新生代对象仍然是Stop-The-World，因为**新生代的垃圾回收时间短**)，因此CMS的标记过程十分复杂：

1.初始标记：从根对象(GC Roots)开始标记，可触及的对象会被标记。会造成用户线程短暂停顿，不过速度非常快。

2.并发标记：和用户线程一起标记

3.重新标记：独占的

4.清除：和用户线程一起



从上面我们可以看到，CMS虽说叫并发收集器，但是并发的部分也就只有**并发标记**和并**发清理**两部分而已。但是初始标记和重新标记的执行速度是很快的，会大量**消耗**时间的是**并发标记**的部分。把消耗大量时间的操作用并发来解决，节省了很多的时间。

下面我们说一下CMS收集器存在的几个问题：

1.吞吐量与性能：在应用程序运行时，需要分出50%的CPU去执行GC，因此系统的性能与吞吐量会下降。

2.清理不彻底：如果在并发清理的时候，应用程序还在运行，可能会有新的对象需要被回收，那么在这一次GC中，这个对象就不能被回收到了。

3.产生内存碎片：由于CMS是采用标记-清除的方式来处理垃圾的，所以不可避免的会产生内存碎片，导致内存利用率低。你可能会问，为啥不采用标记-压缩呢？因为在清理对象的同时应用程序还在创建对象，我们无法通过压缩的方式操作内存。

* G1垃圾收集器

前面已经说了好几个垃圾收集齐了，重点的是CMS垃圾收集器，接下来我们要说说G1垃圾收集器了，尤其是和CMS收集器来做一个对比。

1. 什么是G1垃圾收集器，和CMS有什么区别？

2. OopMap和RememberSet