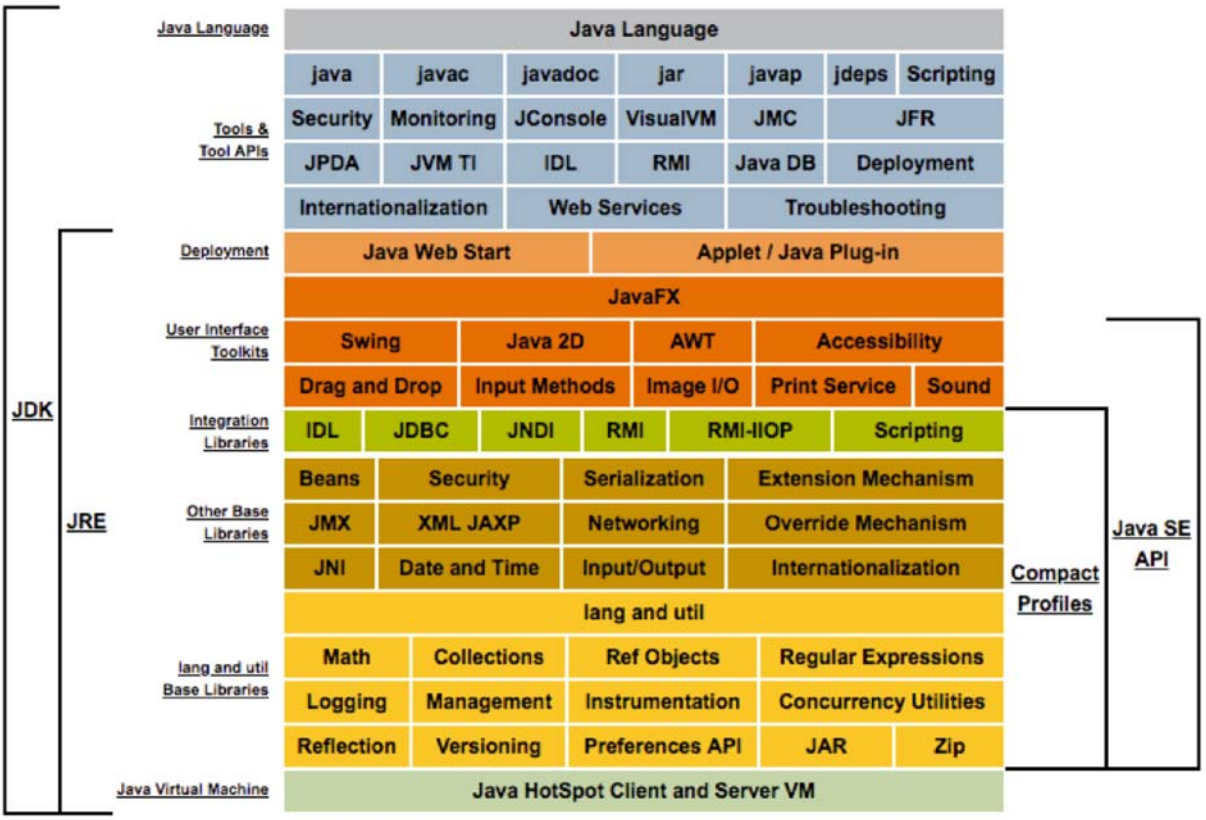
Java 高级篇： JVM 垃圾回收机制

整体了解 JDK & JVM

首先要对官方的 SDK 有点认识，同时要明白下面的概念：

- Java SE（Java Platform, Standard Edition）：它是 Java 的标准版，主要用于桌面应用开发，同时也是 Java 的基础，它包含 Java 语言基础、JDBC（Java 数据库连接性）操作、I/O（输出输出）操作、网络通信、多线程等技术。
- Java EE（Java Platform, Enterprise Edition）：它是 Java 的企业版本（javax..*），包含了 Servlet、JSP、JMS、JNDI 等的扩展。
- Java ME（Java Platform, Micro Edition）：一般是指 Java ME Embedded，Java 微型版本，一般做嵌入式开发用。
- JRE（Java Runtime Environment），Java 运行环境。
- JDK（Java Development Kit），Java 开发者工具集，是用来编译和执行 Java 程序必备的 Java 开发环境，现在我们一般说 JDK 就是指的 Oracle 的 Java SE。因为 Sun JDK 和 Open JDK、JRockit 都被 Orcale 收购了，一统了江湖。

我们看下 Oracle 官方的图，如下：



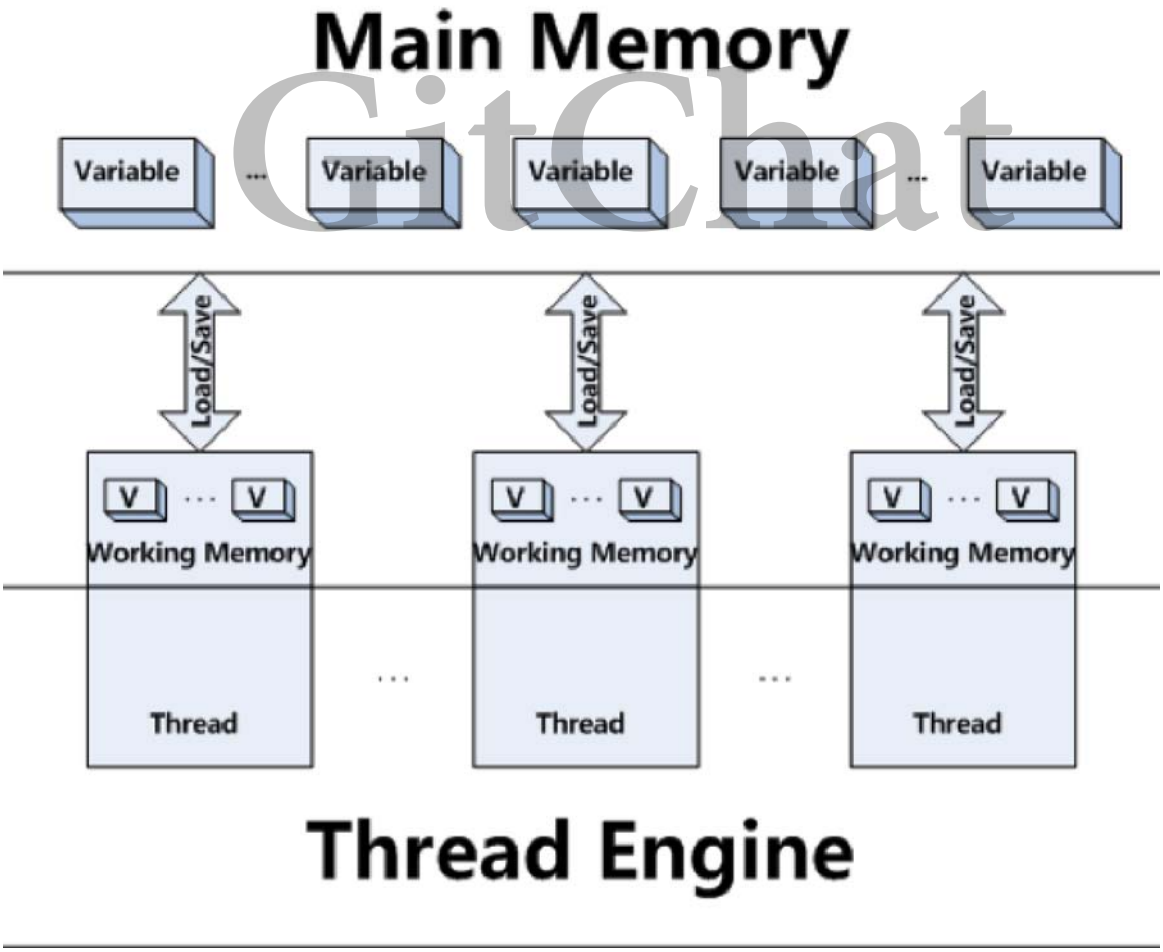
这样我们对 JDK、JRE，有了概念上的认识之后，我们来看下我们的 JVM（Java Virtual Machine）——Java 虚拟机，也就是 HotSport 了。

> Oracle 也出了 JVM 的规范。

Java 内存模型

Java 内存模型是理解 Java 多线程和 Java GC 必须要了解的抽象知识点，我们可以通过工具来更好的掌握 Java 内存模型。我给大家一个点：“我们通过不同的视角来理解内存模型”。我们可以通过不同的视角来理解内存模型。因为 JVM 类里面实际的内存操作远比我们想象得要复杂，因为这部分代码是 Oracle 官方的核心机密，没有对外公开，我们也只能通过官方文档及其 Jdk/bin 目录下面的工具来做到整体认识。

站在理解线程的视角看内存模型

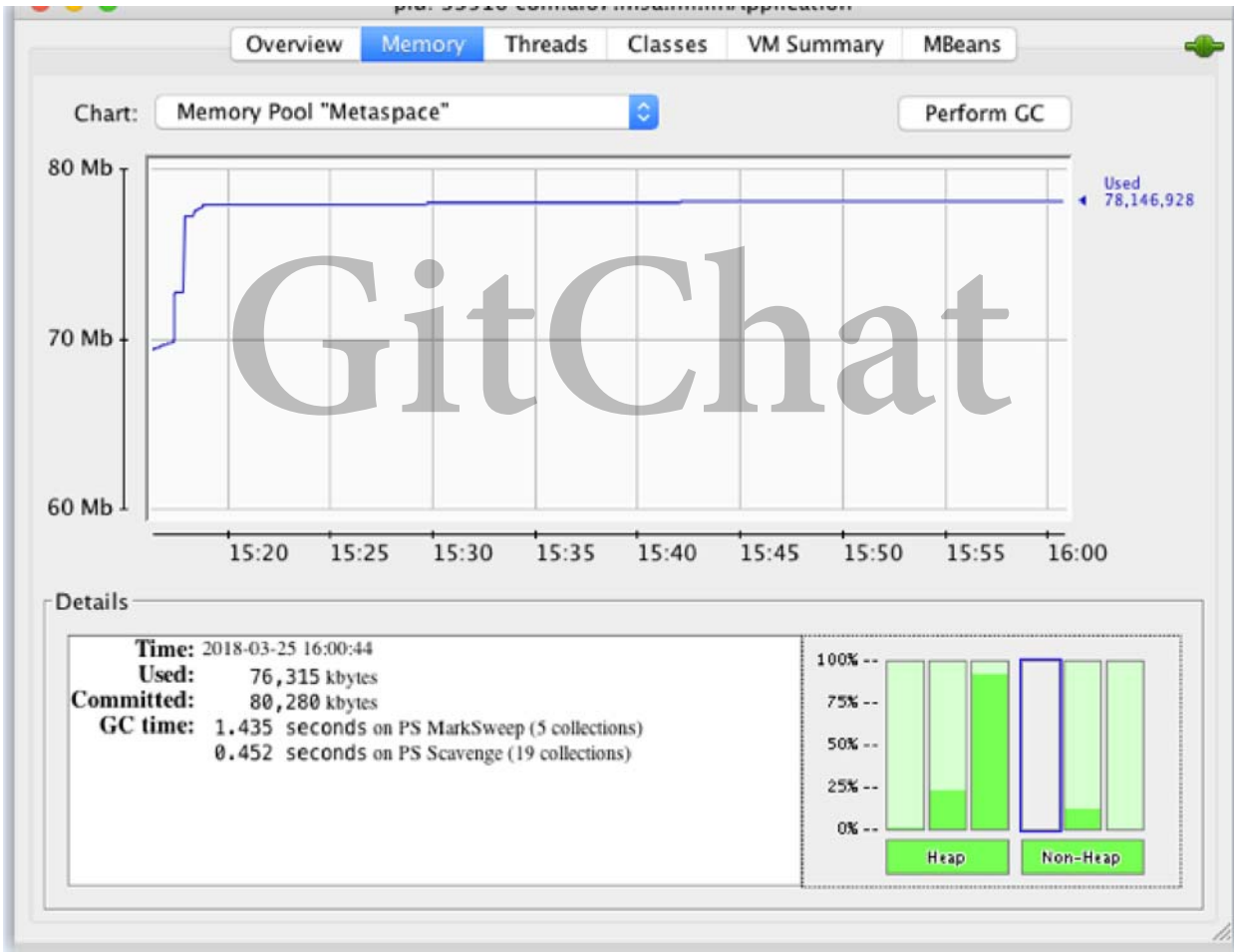


我们可以把 JVM 内存结构直接分成线程私有内存和共享主内存。这样我们就可以很好地理解多线程的很多问题如同步锁、lock、validate 关键字，及其 ThreadLocal。这部分内容如果还疑惑的可以看作者的另外一篇 Chat：Java 多线程与并发编程 · Java 工程师必知必会。

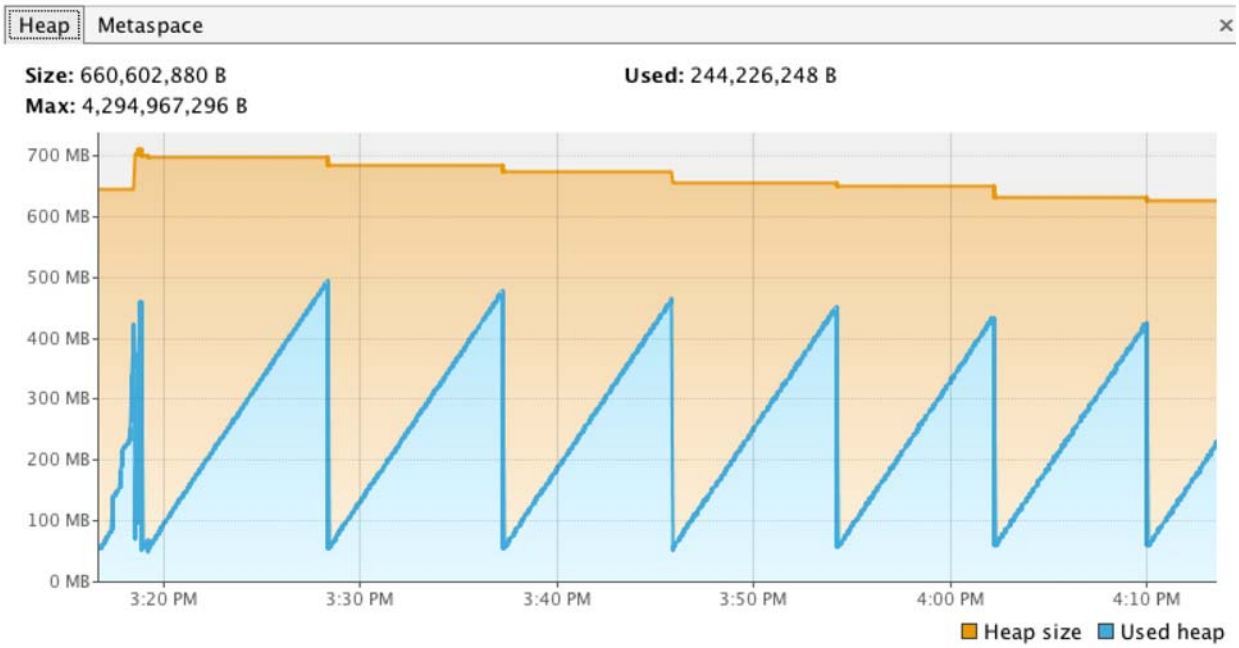
我们从内存设置的角度出发

我们可以将内存直接分位堆内存、非堆内存（JDK8 以后叫 Metaspace，元空间）和其它，三个大的类别。
我们来看一下 JConsole 和 JVisualVM。

java/bin/jconsole 打开以后界面如下：



java/bin/jvisualvm 打开以后界面如下：



利用 tools，也可以看到 Java 工具也是简单的将其分成堆和非堆（Metaspace）。

而其它是什么呢?

Other 指的是“直接内存”，如一些（IO/NIO），这些 JVM 控制不了（如果线程变多线程栈吃的内存也会变的非常大，不可设置）。

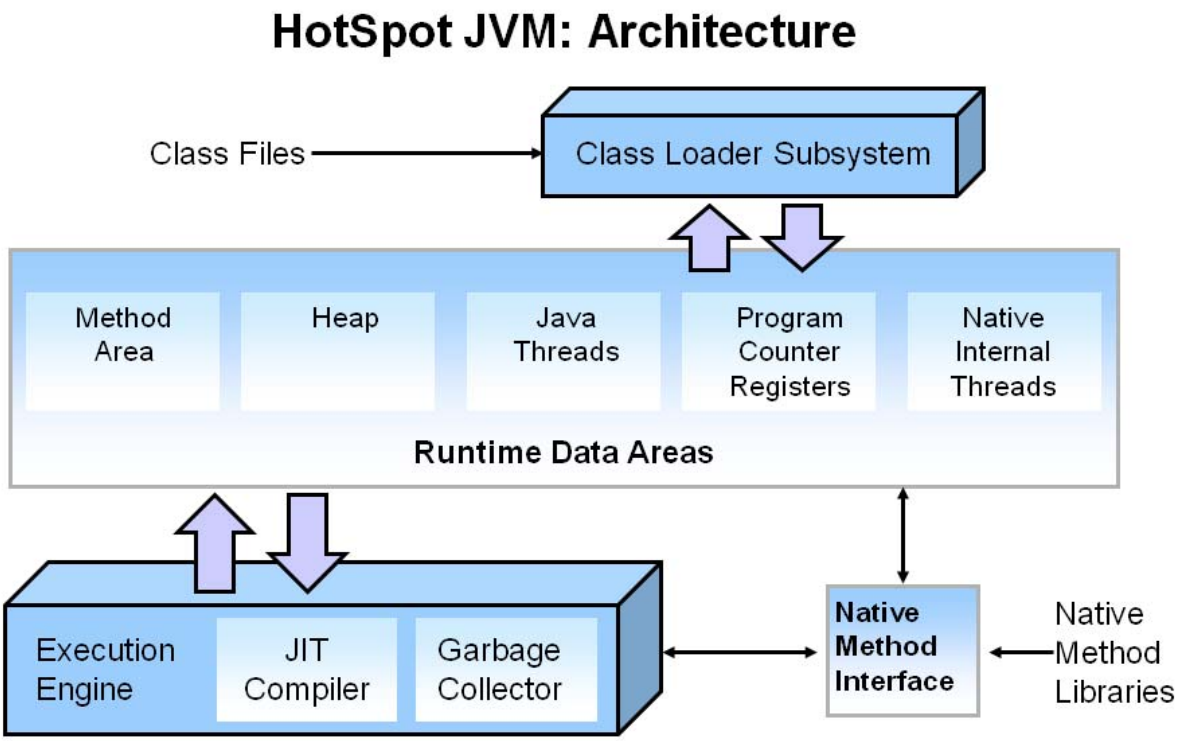
对应的 JVM 设置的参数是：

- Xmx4g：JVM 最大允许分配的堆内存，按需分配；
- Xms4g：JVM 初始分配的堆内存，一般和 Xmx 配置成一样以避免每次 gc 后 JVM 重新分配内存；
- XX：MetaspaceSize=64m 初始化元空间大小；
- XX：MaxMetaspaceSize=128m 最大化元空间大小。

Metaspace 建议大家不要设置，一般让 JVM 自己启动的时候动态扩容就好了，没必要自己去设置。如果不动态加载 class，当启动起来的时候，一般是很少有变化的。

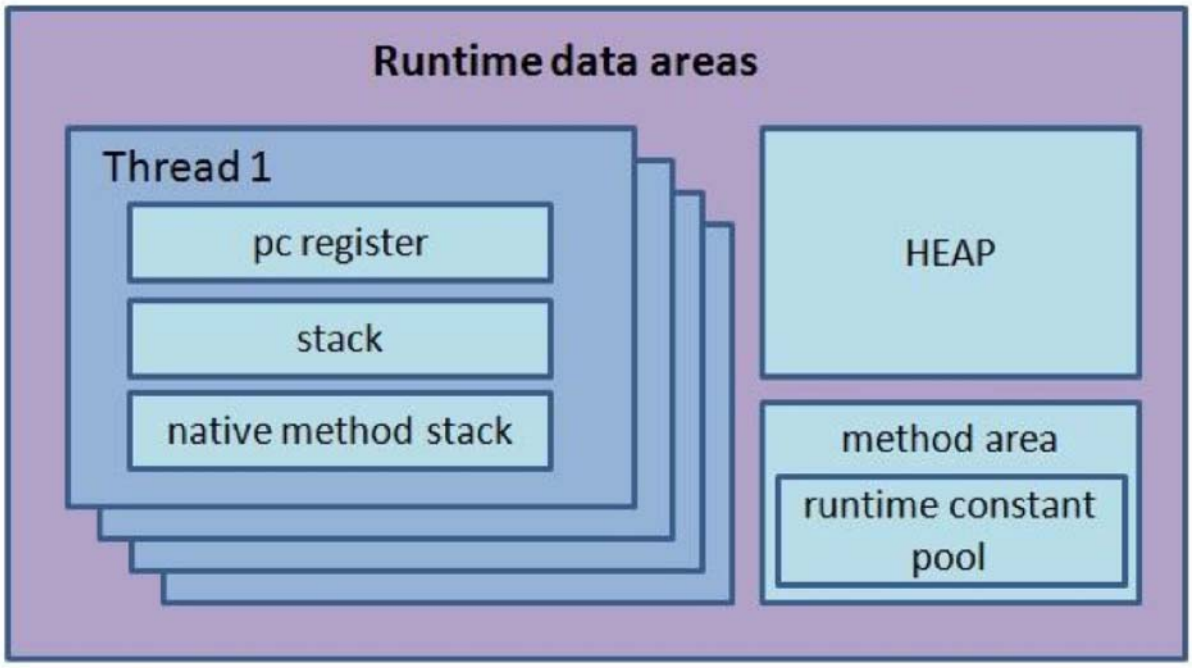
从这个角度我们可以认为我们的 JVM 内存的大小是堆+metaspace+io(运行时产生的大小)。

我们从 JVM 的运行期的视角来看



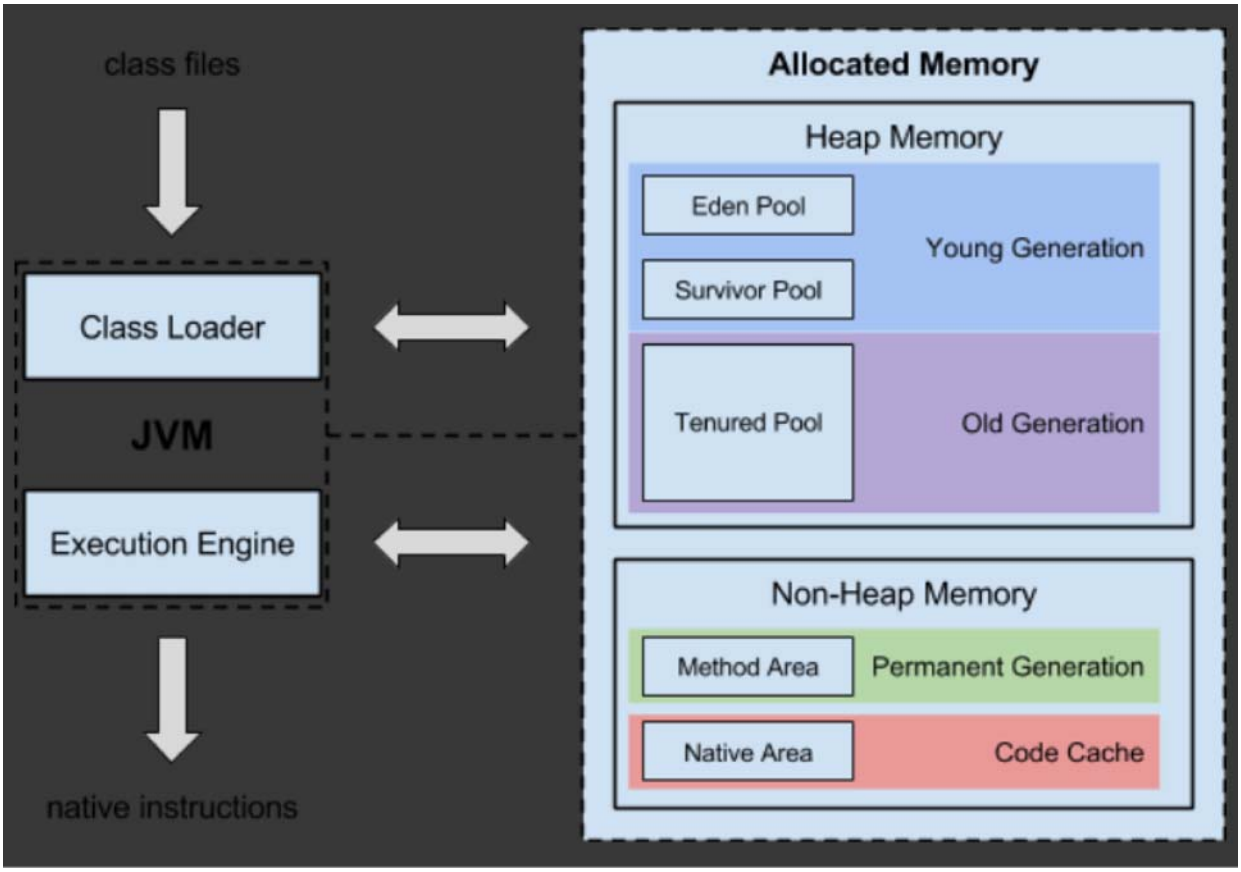
可以分为五大部分：方法区、堆、本地方法栈区、PC 计数器、线程栈。我们也可以看下面的图，PC 计数器和栈、本地方法栈，是随着当前的线程开始而开始，销毁而销毁的。

我们再通过下面这个图理解一下这五个区和线程的关系：

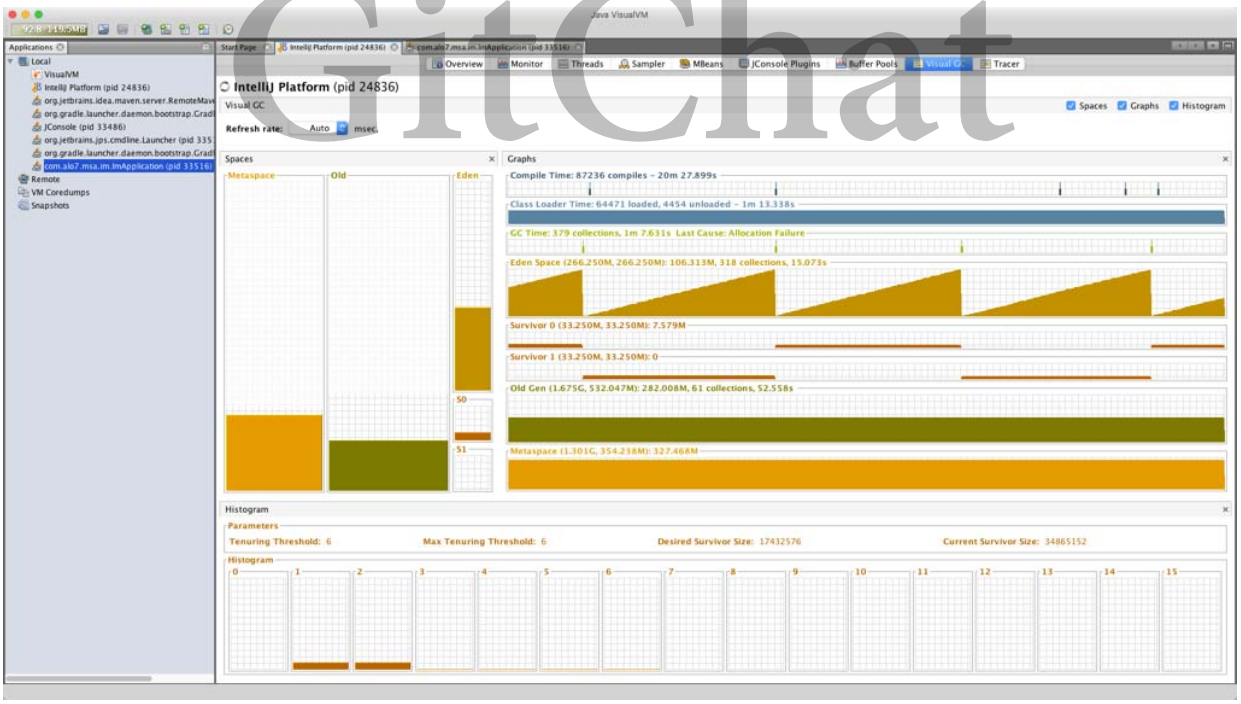


对应的 JVM 的参数为 Xss512k，用来设置每个线程的堆栈大小。

从垃圾回收机制的视角来看



全局分代收集器，我们通过 java/bin/jvisualvm 来观察一下：



通过 JVisualVM 我们可以看得出来：

- 内存直接被垃圾收集器切分了5个部分：metaspace（class 结构）（永久代）、Old（老年代）、新生代（一个 Eden（新对象创作的乐园，老外真会取名）、二个 Survivor Space））。
- 一个对象默认被交换了6次还没有回收掉就会被扔到老年区里面。
- `-XX:InitialTenuringThreshold=7` 通过这个来设置，交换几次丢到老年代。

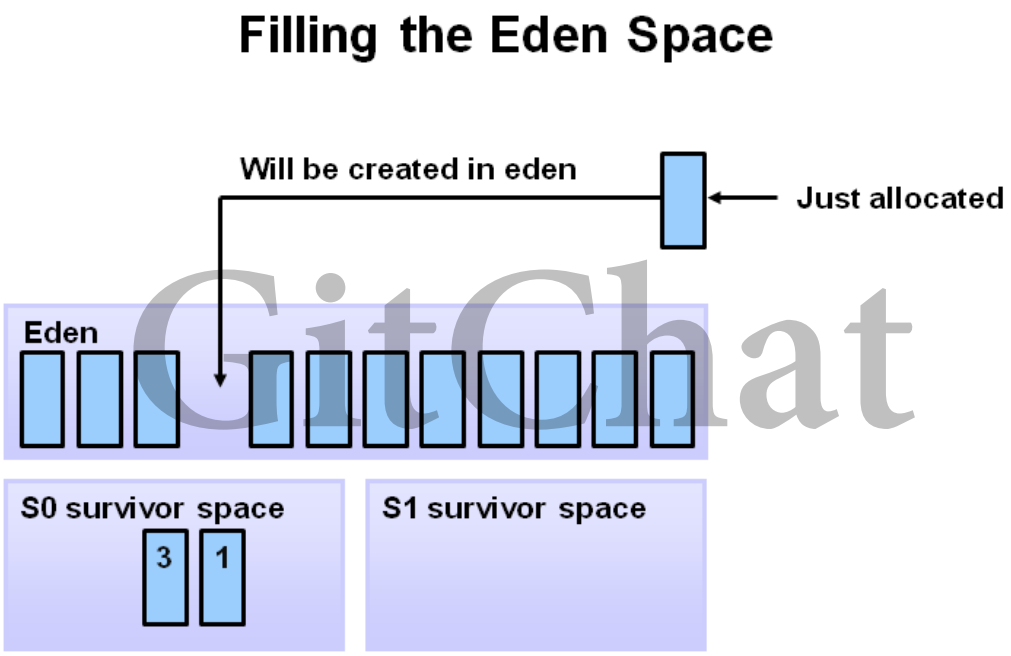
到此内存模型介绍完毕，给大家留一个思考题：递归和 for 循环分别影响的是哪块内存?

JVM 垃圾回收机制

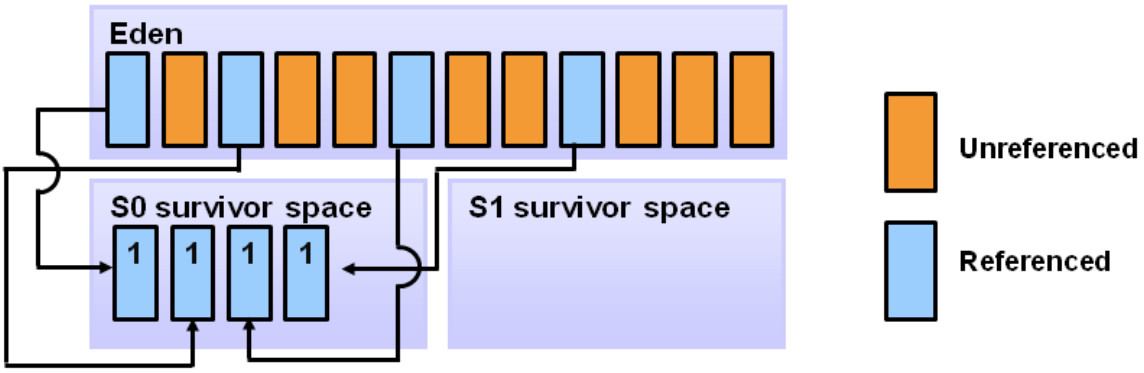
由于 Hotspot JVM 全局是采用的分代收集器，所以我们来看一下每个代区分别都有哪些可以配置的收集器，及其回收的一个过程。

分代回收的过程

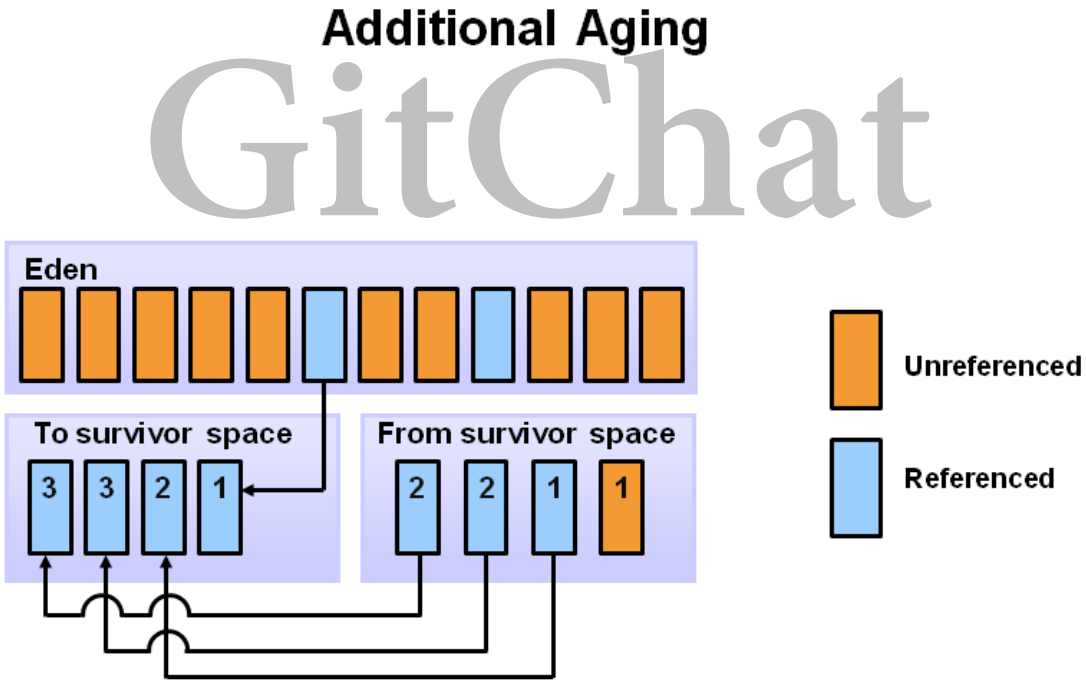
（1）分配新对象在新生代的 Eden 区。



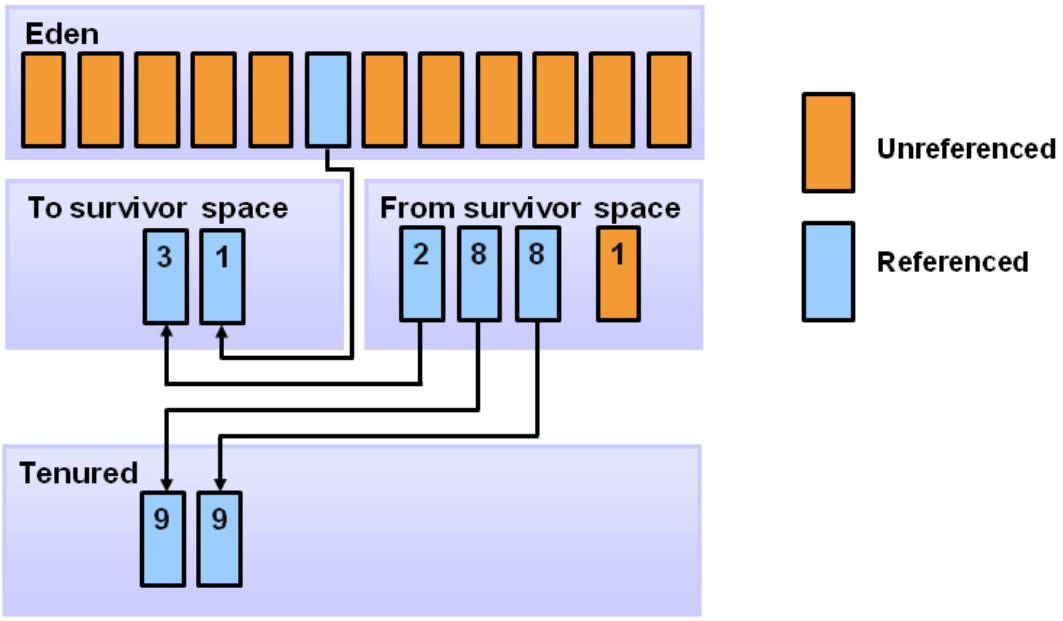
（2）当达到回收条件的时候，未有引用的会受到，有引用的放到新生代的 Survivor Space 区。



（3）等几次回收之后，来回再 Survivor Space 复制几回，并且标记上对象的年龄。



（4）当我们设置 `-xx：MaxTenuringThreshold=9` 的时候。当年龄达到9的时候 copy 到老年代



（5）然后循环此过程，当老年代达到一定值的时候触发老年 GC。

回收算法

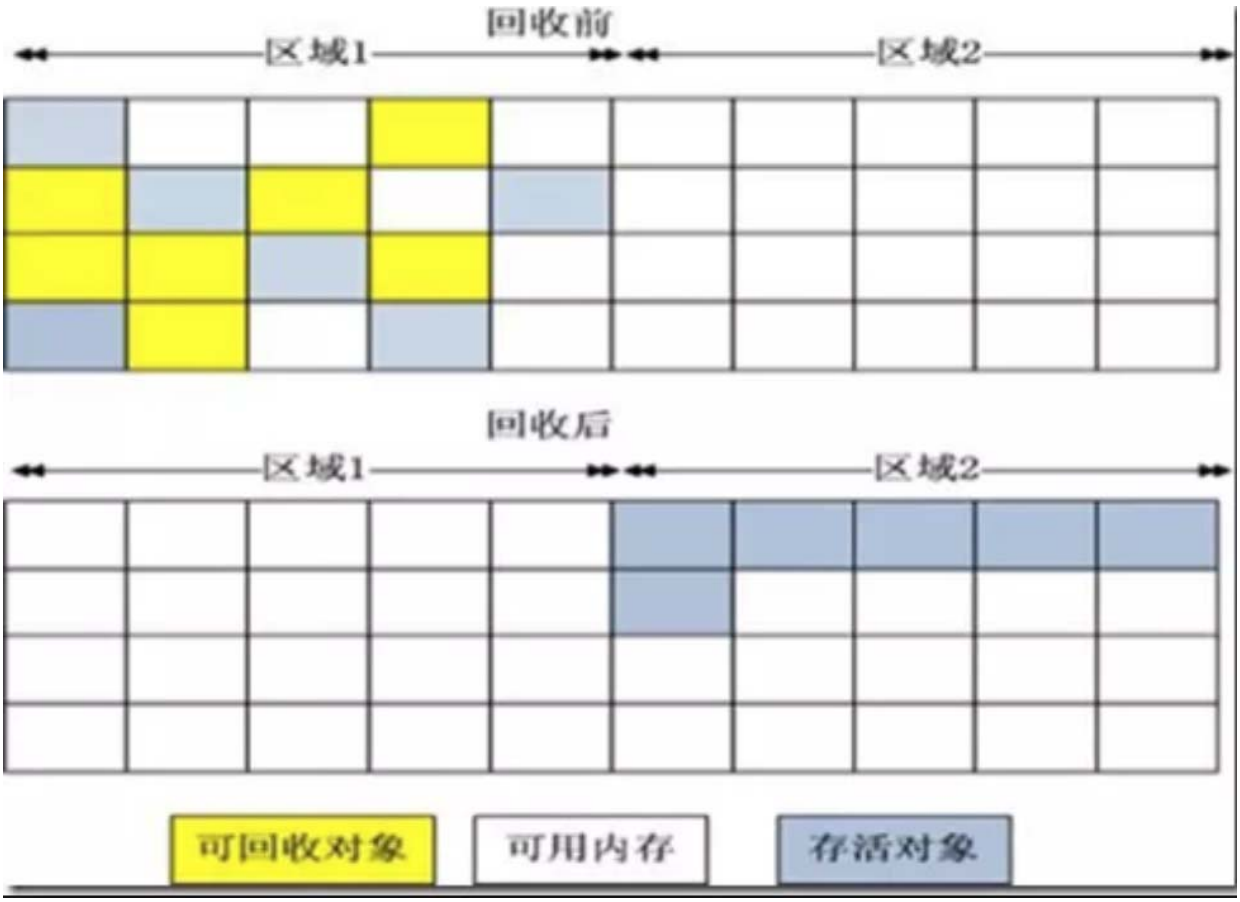
当发生垃圾回收的时候会用到三种算法。

（1）“标记-清除”（Mark-Sweep）算法。

如它的名字一样，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掉所有被标记的对象。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。

它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，可能会导致太多空间碎片，当程序在以后的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另一次垃圾收集动作。

（2）复制算法（Copying）



该算法思想有以下几点：

- 将现有的内存空间分为两快，每次只使用其中一块；
- 当其中一块时候完的时候，就将还存活的对象复制到另外一块上去；
- 再把已使用过的内存空间一次清理掉。

其优点有：

- 由于是每次都对整个半区进行内存回收，内存分配时不必考虑内存碎片问题；
- 只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

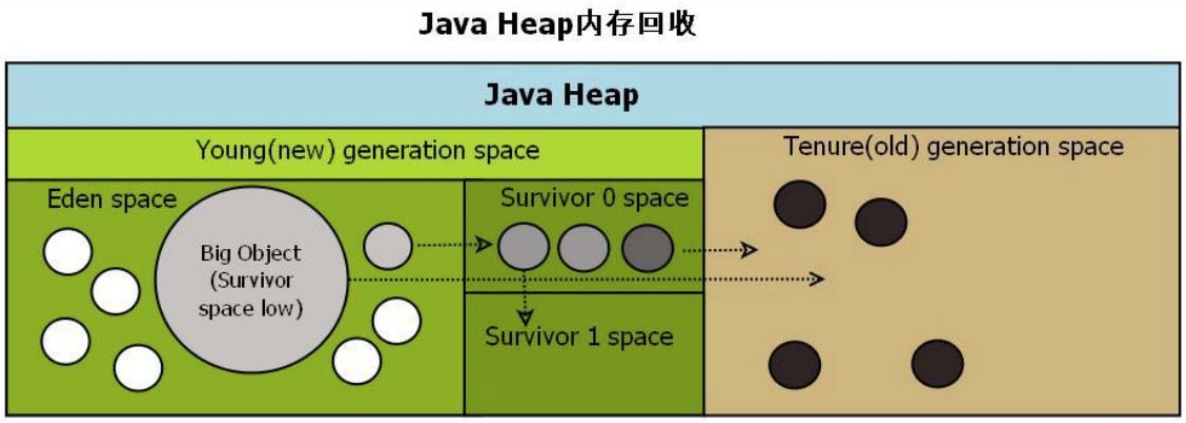
其缺点有：

- 内存减少为原来的一半，太浪费了；
- 对象存活率较高的时候就要执行较多的复制操作，效率变低；
- 如果不使用50%的对分策略，老年代需要考虑的空间担保策略。

演进：

并不需要根据1:1划分内存空间，而是将内存划分为一块较大的 Eden Space 和两块较小的 Survivor Space。

JavaHeap 内存回收模型如下图（当前商业虚拟机大多使用此算法回收新生代）：



1.当前商业虚拟机都采用这种算法回收Young Generation Space。
2.Eden和Survivor并不是根据1:1的比例来划分内存空间。（HotSpot虚拟机默认是8:1）
3.总有一块Survivor是空闲的。
4.当Survivor空间不够用的时候，需要依赖于Old Generation Space的空间担保。

（3）标记-整理算法（Mark-Compact）

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）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，而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（有点 copy 的意思，但是比 copy 省空间。比清理好的一点是没有碎片）。

注意：其实全局的分代回收思想也是一种算法。

收集器

在大的分代回收的思想下面，不同的代区可以选择不同的收集器，而不同的收集器在不同的代区又会用到上面不同的算法，我们来详细了解一下。

（1）Serial收集器

其特点有：

- Serial（串行）收集器是最基本、发展历史最悠久的串行收集器，JDK 1.5 之前默认都是此收集器，因为那时候 CPU 都是单核的。
- 单线程阻塞队列。

其优点有为简单而高效（与其他收集器的单线程相比），对于限定单个 CPU 的环境来说，Serial 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更高的单线程收集效率。

其缺点有：

- 它是一个单线程收集器，只会使用一个 CPU 或一条收集线程去完成垃圾收集工作；

- 它在进行垃圾收集时，必须暂停其他所有的工作线程，直至 Serial 收集器收集结束为止（Stop The World）。

其应用场景有：

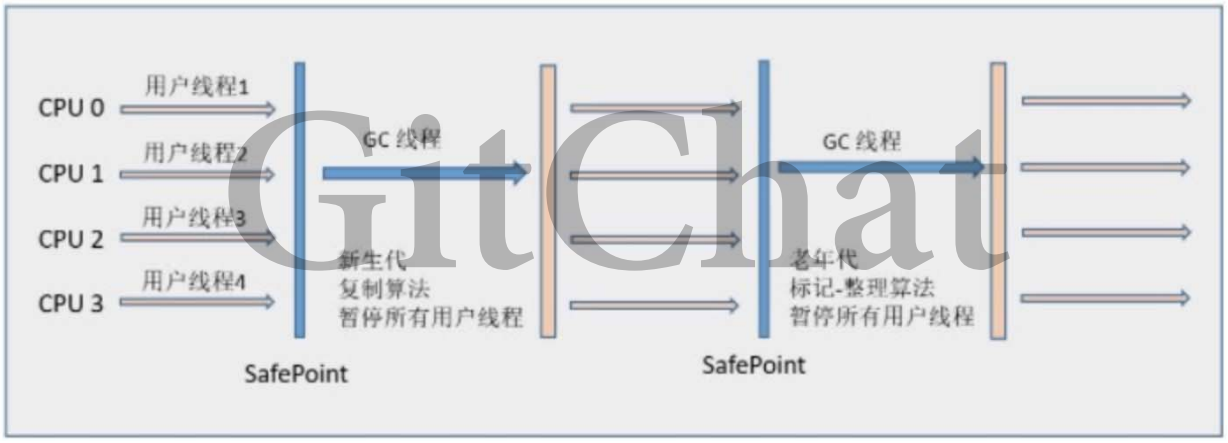
- HotSpot 虚拟机运行在 Client 模式下的默认的新生代收集器。
- 单 CPU 虚拟机里面。
- JDK 1.3.1 之前，是虚拟机新生代收集的唯一选择。JDK 1.5.0 之前老年代的唯一选择。
- 内存比较小的情况下，效率还是很高的。

对应的算法：复制算法（年轻代）、标记整理算法（老年代）。

内存模型块：年轻代、老年代叫（Serial Old），老年代没办法直接指定。

配置的参数： `+xx:UseSerialGC` （如果使用此配置默认年轻代，老年代采用 Serial Old）。

下图展示了 Serial 收集器（老年代采用 Serial Old 收集器）的运行过程：



ParNew 收集器

ParNew 收集器就是 Serial 收集器的多线程版本（即并发模式），除了使用多线程进行垃圾收集外，其余行为包括 Serial 收集器可用的所有控制参数、收集算法（复制算法）、Stop The World、对象分配规则、回收策略等与 Serial 收集器完全相同，两者共用了相当多的代码。

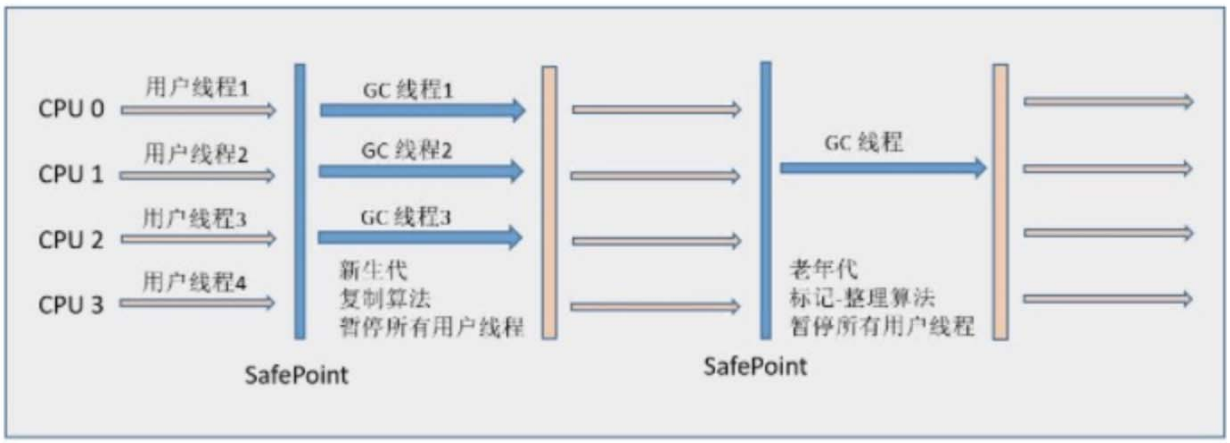
其缺点为只能用于新生代。

其特点为：ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器有更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证可以超越。在多 CPU 环境下，随着 CPU 的数量增加，它对于 GC 时系统资源的有效利用是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多的情况下可使用 `-XX:ParallerGCThreads` 参数设置。

配置的参数为：

- `+xx:UseParNewGC` （如果使用此配置默认年轻代，老年代采用 Serial Old）。
- `-XX:ParallerGCThreads=3` （多 CPU 情况下面开启多少个线程来回收）。

ParNew 收集器的工作过程如下图（老年代采用 Serial Old 收集器）：



Parallel Scavenge 并行收集器

其特点有：

- 并行的；
- Parallel Scavenge 收集器的目标是达到一个可控制的吞吐量（Throughput）；
- 自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别。

其吞吐量（Throughput），即CPU用于运行用户代码的时间与 CPU 总消耗时间的比值，即“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）”。

假设虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

其优点为：停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验。而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

其缺点为：Parallel Scavenge 收集器无法与 CMS 收集器配合使用。

其应用场景有：

- 新生代：复制算法。设置参数： `-XX:+UseParallelGC` 。
- 老年代：使用多线程和“标记-整理”算法。设置参数： `-XX:+UseParallelOldGC` 。

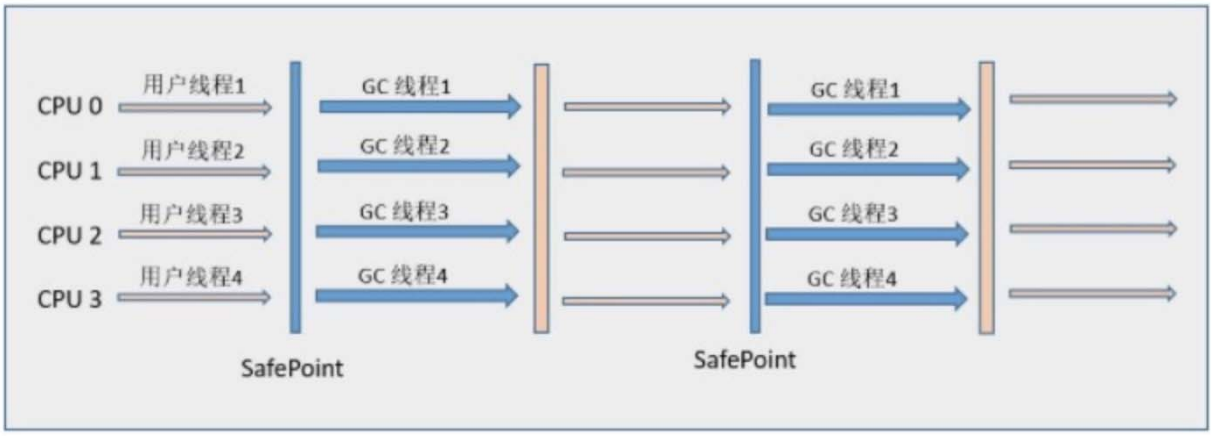
涉及到的参数设置有：

- `-XX:+UseAdaptiveSizePolicy` ，这是一个动态调整各个代区的内存大小的开关参数，打开参数后，就不需要手工指定新生代的大小（`-Xmn`）、Eden 和 Survivor 区的比例（`-XX:SurvivorRatio`）、晋升老年代对象年龄（`-XX:PretenureSizeThreshold`）等细节参数了，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，这种方式称为 GC 自适应的调节策略（GC Ergonomics）。

- `-XX:ParallelGCThreads=n`，并行 GC 线程数。
- `-XX:MaxGCpauseMillis=5`，默认 GC 最大停留时间。
- `-xx:GCTimeRatio`，GC 占用总时间的最大比率。

Parallel Scavenge/Parallel Old 收集器配合使用的流程图：



并发标记清理（Concurrent Mark-Sweep，CMS）垃圾收集器

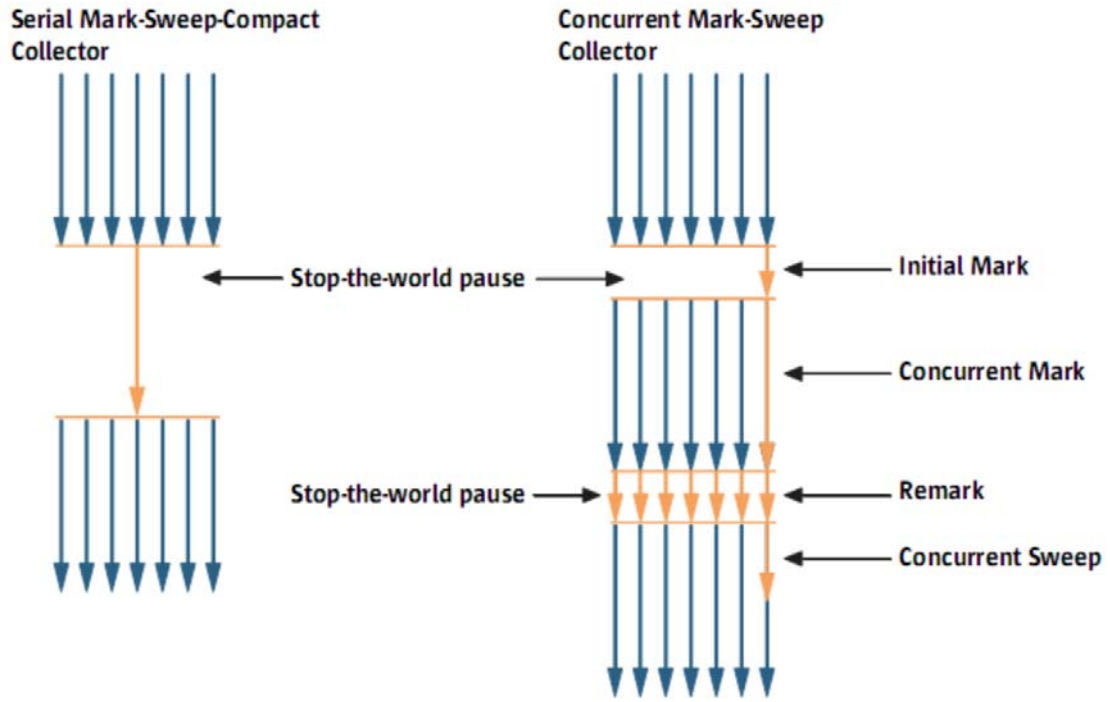
CMS （Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它非常符合那些集中在互联网站或者 B/S 系统的服务端上的 Java 应用，这些应用都非常重视服务的响应速度。从名字上（“Mark Sweep”）就可以看出它是基于“标记-清除”算法实现的。

CMS 收集器工作的整个流程分为以下4个步骤：

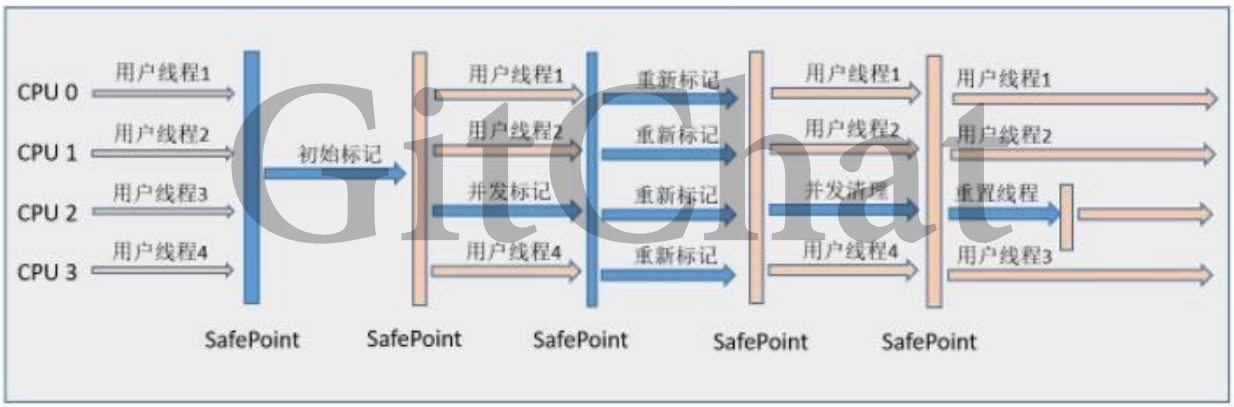
1. 初始标记（CMS initial mark）：仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要“Stop The World”。
2. 并发标记（CMS concurrent mark）：进行 GC Roots Tracing 的过程，在整个过程中耗时最长。
3. 重新标记（CMS remark）：为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。此阶段也需要“Stop The World”。
4. 并发清除（CMS concurrent sweep）。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

普通的串行标记算法与并行标记算法比较，如下图所示：



通过下图可以比较清楚地看到 CMS 收集器的运作步骤中并发和需要停顿的时间：



其优点为：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，因此 CMS 收集器也被称为并发低停顿收集器（Concurrent Low Pause Collector）。

其缺点有：

- 对 CPU 资源非常敏感。其实，面向并发设计的程序都对 CPU 资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但会因为占用了一部分线程（或者说 CPU 资源）而导致应用程序变慢，总吞吐量会降低。CMS 默认启动的回收线程数是（CPU数量+3）/4，也就是当 CPU 在4个以上时，并发回收时垃圾收集线程不少于25%的 CPU 资源，并且随着 CPU 数量的增加而下降。但是当 CPU 不足4个时（比如2个），CMS 对用户程序的影响就可能变得很大，如果本来 CPU 负载就比较大，还要分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无法接受。
- 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。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生。这一部分垃圾出现在标记过程之后，

CMS 无法再当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就被称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。

- 标记-清除算法导致的空间碎片。CMS 是一款基于“标记-清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象。

什么时候使用 CMS 垃圾收集器?

当你的应用程序需要有较短的应用程序暂停，而可以接受垃圾收集器与应用程序共享应用程序时，你可以选择 CMS 垃圾收集器。典型情况下，有很多长时间保持 live 状态的数据对象（一个较大的老年代）的应用程序，和运行在多处理上的应用程序，更适合使用 CMS 垃圾收集器。例如 Web 服务器。若应用程序需要有较短的暂停时间的话，也可以考虑 CMS 垃圾收集器。

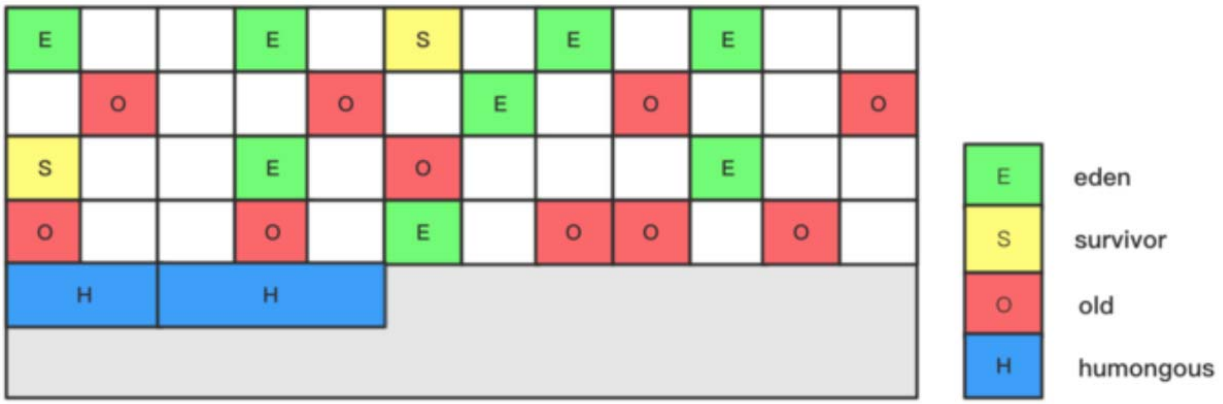
参数控制如下：

- `-XX:+UseConcMarkSweepGC`，使用 CMS 收集器；
- `-XX:+UseCMSCompactAtFullCollection`，Full GC 后，进行一次碎片整理，整理过程是独占的，会引起停顿时间变长。
- `-XX:+CMSFullGCsBeforeCompaction`，设置进行几次 Full GC 后，进行一次碎片整理。
- `-XX:ParallelCMSThreads`，设定 CMS 的线程数量（一般情况约等于可用 CPU 数量）。

G1 收集器

G1（Garbage-First）收集器是当今收集器技术发展最前沿的成果之一。它是一款面向服务端应用的垃圾收集器。

其特点为：在 G1 算法中，采用了另外一种完全不同的方式组织堆内存，堆内存被划分为多个大小相等的内存块（Region），每个 Region 是逻辑连续的一段内存，结构如下：



堆内存中一个 Region 的大小可以通过 `-XX:G1HeapRegionSize` 参数指定，大小区间只能是 1M、2M、4M、8M、16M 和 32M，总之是2的幂次方，如果 G1HeapRegionSize 为默认值，则在堆初始化时计算 Region 的实践大小。

G1中提供了三种模式垃圾回收模式：Young GC、Mixed GC 和 Full GC，在不同的条件下被触发。

Young GC

发生在年轻代的 GC 算法，一般对象（除了巨型对象）都是在 Eden Region 中分配内存，当所有 Eden Region 被耗尽无法申请内存时，就会触发一次 Young GC，这种触发机制和之前的 Young GC 差不多，执行完一次 Young GC，活跃对象会被拷贝到 Survivor Region 或者晋升到 Old Region 中，空闲的 Region 会被放入空闲列表中，等待下次被使用。

各参数的含义：

- `-XX:MaxGCPauseMillis`，设置 G1 收集过程目标时间，默认值200ms。
- `-XX:G1NewSizePercent`，新生代最小值，默认值5%。
- `-XX:G1MaxNewSizePercent`，新生代最大值，默认值60%。

Mixed GC

当越来越多的对象晋升到老年代 Old Region 时，为了避免堆内存被耗尽，虚拟机会触发一个混合的垃圾收集器，即 Mixed GC，该算法并不是一个 old gc，除了回收整个 Young Region，还会回收一部分的 Old Region，这里需要注意：是一部分老年代，而不是全部老年代，可以选择哪些 Old Region 进行收集，从而可以对垃圾回收的耗时时间进行控制。

那么 Mixed GC 什么时候被触发？

先回顾一下 CMS 的触发机制，如果添加了以下参数：

- `-XX:CMSInitiatingOccupancyFraction=80`
- `-XX:+UseCMSInitiatingOccupancyOnly`

当老年代的使用率达到80%时，就会触发一次 cms gc。相对的，Mixed GC 中也有一个阈值参数 `-XX:InitiatingHeapOccupancyPercent`，当老年代大小占整个堆大小百分比达到该阈值时，会触发一次 Mixed GC。

Mixed GC 的执行过程有点类似 CMS，主要分为以下几个步骤：

1. initial mark: 初始标记过程，整个过程 STW，标记了从 GC Root 可达的对象；
2. concurrent marking: 并发标记过程，整个过程 gc collector 线程与应用线程可以并行执行，标记出 GC Root 可达对象衍生出去的存活对象，并收集各个 Region 的存活对象信息；
3. remark: 最终标记过程，整个过程 STW，标记出那些在并发标记过程中遗漏的，或者内部引用发生变化的对象；

4. clean up: 垃圾清除过程，如果发现一个 Region 中没有存活对象，则把该 Region 加入到空闲列表中

Full GC

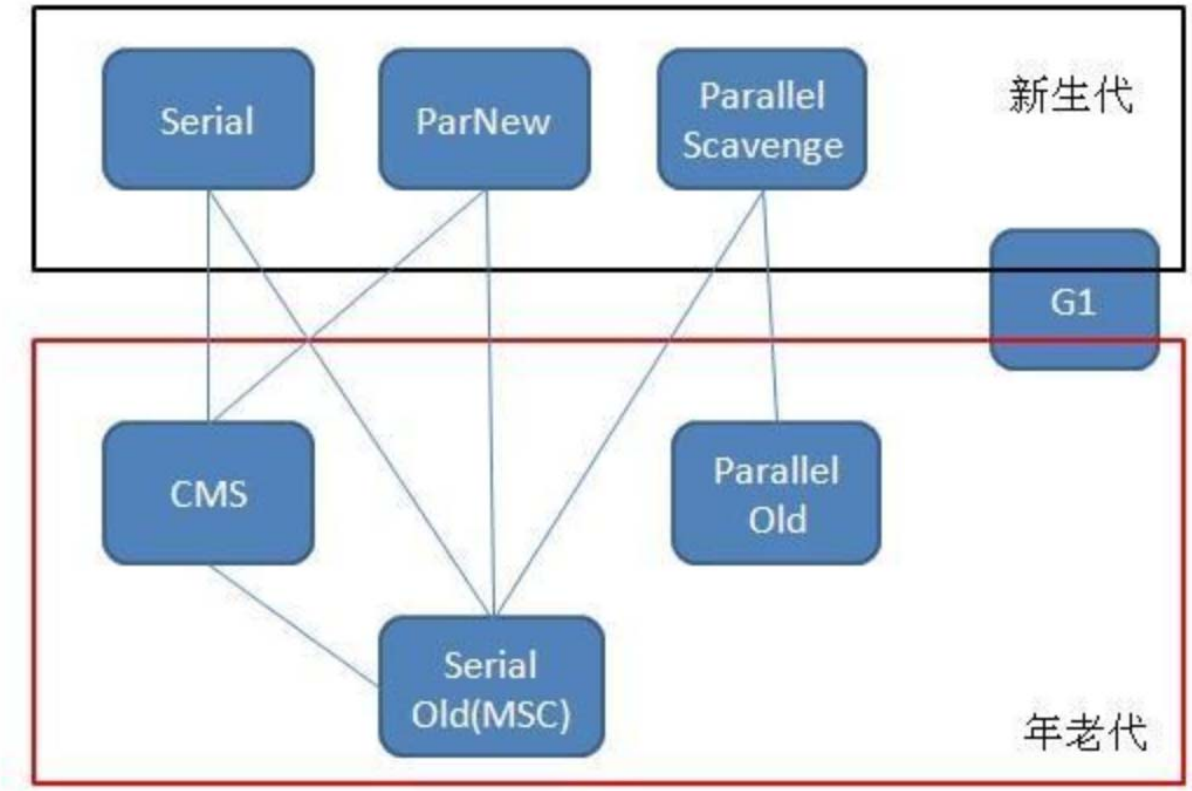
如果对象内存分配速度过快，Mixed GC 来不及回收，导致老年代被填满，就会触发一次 Full GC，G1 的 Full GC 算法就是单线程执行的 serial old gc，会导致异常长时间的暂停时间，需要进行不断的调优，尽可能的避免 Full GC。

> G1，我涉足的还不是特别多，就先把其两大特性简单介绍下。

五种垃圾搜集器的比较

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单 CPU 环境下的 Client 模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

我们用一张图来表示一下五种收集器的关系：



连线表示可以配合使用。

收集器预留两个问题咱们在讨论环节解答，也欢迎读者踊跃提问，我也会提前准备下。

1. 垃圾回收的触发时机是什么？
2. 对象四种引用方式是什么，在什么场景用到？

JVM 生产监控的指标有哪些？

关于 GC 的监控，我认为最重要的三点为：

1. 各个区的容量。
2. Full GC、Young GC 发生的次数。
3. 当前系统的内存比、CPU 使用率。

我们以 java/bin/jstat 为例看看相关的参数细节有哪些。

数据列	描述	支持的jstat 选项
S0C	Survivor0的当前容量	-gc -gccapacity -gcnew -gcnewcapacity
S1C	S1的当前容量	-gc -gccapacity -gcnew -gcnewcapacity
S0U	S0的使用量	-gc-gcnew
S1U	S1的使用量	-gc-gcnew

数据列	描述	支持的jstat 选项
EC	Eden区的当前容量	-gc -gccapacity -gcnew -gcnewcapacity
EU	Eden区的使用量	-gc -gcnew
OC	old区的当前容量	-gc -gccapacity -gcnew -gcnewcapacity
OU	old区的使用量	-gc-gcnew
PC	方法区的当前容量	-gc-gccapacity -gcold -gcoldcapacity -gcpermcapacity
PU	方法区的使用量	-gc -gcold
YGC	Young GC次数	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
YGCT	Young GC累积耗时	-gc -gcnew -gcutil -gccause
FGC	Full GC次数	-gc -gccapacity -gcnew -gcnewcapacity -gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
FGCT	Full GC累积耗时	-gc-gcold -gcoldcapacity -gcpermcapacity -gcutil -gccause
GCT	GC总的累积耗时	-gc -gcold -gcoldcapacity -gccapacity -gcpermcapacity -gcutil -gccause
NGCMN	新生代最小容量	-gccapacity -gcnewcapacity
NGCMX	新生代最大容量	-gccapacity -gcnewcapacity
NGC	新生代当前容量	-gccapacity -gcnewcapacity
OGCMN	老年代最小容量	-gccapacity -gcoldcapacity
OGCMX	老年代最大容量	-gccapacity -gcoldcapacity
OGC	老年代当前容量	-gccapacity -gcoldcapacity
PGCMN	方法区最小容量	-gccapacity -gcpermcapacity
PGCMX	方法区最大容量	-gccapacity -gcpermcapacity
PGC	方法区当前容量	-gccapacity -gcpermcapacity
PC	方法区的当前容量	-gccapacity -gcpermcapacity
PU	方法区使用量	-gccapacity -gcold
LGCC	上一次GC发生的原因	-gccause

数据列	描述	支持的jstat 选项
GCC	当前GC发生的原因	-gccause
TT	存活阀值，如果对象在新生代移动次数超过此阀值，则会被移到老年代	-gcnew
MTT	最大存活阀值，如果对象在新生代移动次数超过此阀值，则会被移到老年代	-gcnew
DSS	survivor区的理想容量	-gcnew

轻松应对 JVM 的面试和实际工作。

针对面试，建议读者保持两点：

- 思路清晰，说的是通的。建议读者可以根据作者整理的内容弄个简单的思路导图。
- 面要全，有一到两个点要细。概况起来了就三点，如下。
    - 不同视角的内存模型、
    - GC 对应的4种算法。
    - GC 对应的5种收集器。

针对于工作，希望大家了解以下几点：

1. 一定要知道 JDK 版本，JVM 默认参数有哪些？
2. 我可以设置哪些？哪些是必须设置的？
3. 可以直接参看很多容器，如 Tomcat、Jetty、Docker 的默认 JVM 参数。

如果大家有问题需要互相交流，可以加以下两个群。

- QQ交流群1：240619787
- QQ交流群2：559701472