Jvm 入门

前言

堆：比方说有一个 xxx.class 文件，里面有一些类的定义，这些类的定义需要放在某些地方，类的定义放在方法区，方法区也叫永久区，在程序运行中，我会不断的 new 对象，list，hashMap，这些对象存放了大量的数据，给它开辟了一块比较大的空间，主要是保存对象，这块空间称为堆。

栈：我的程序在运行过程中，本质上启动一个线程，就算没有 thread，也有 main 线程，线程在工作的过程中就是执行程序里面的各种方法，方法里面有一些局部对象，方法在运行过程中会创建一些对象（new）,这些对象都存放在堆里面，方法里面保存的是对象的引用（地址），这个引用是一个局部变量，这个栈里面除了 main 线程，可能还有其他线程，大家记住，每个线程内部的局部变量都是独立的，意思就是说我把局部变量定义相同的名称，也不受影响，每一个线程都会对应于一个栈空间。一个线程里面会调用很多方法，方法还会调用其他方法，这个栈里面有个概念：栈帧，每调用一个方法弹一个帧，有入栈出栈（压栈和弹栈）这种概念，调用另外一个方法会产生另外一个栈帧，又入栈出栈，有一个问题，如果这个方法的调用嵌套很深，那这个地方栈空间就会溢出，比如我们用递归，这样方法会被一直调用，这样也会导致栈空间的内存溢出。

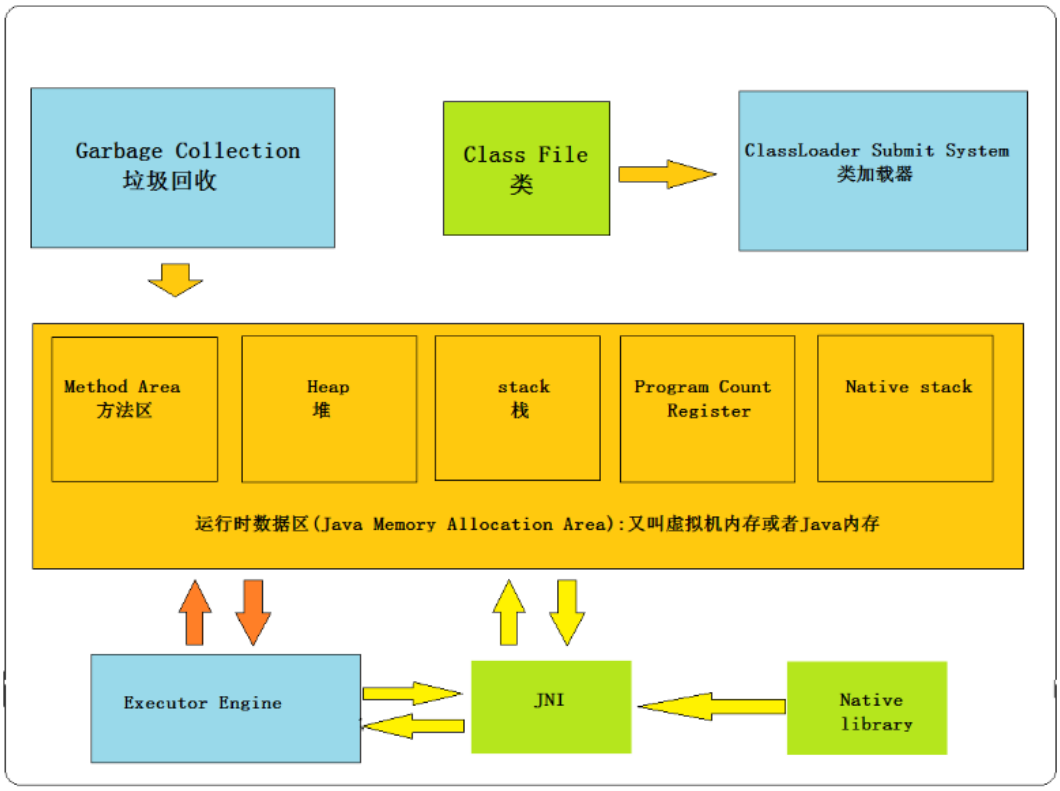
程序计数器：就是标记程序执行到哪一行了，如果没有这个机制，cpu 在下次执行的时候就不知道执行哪一行了，会乱了。

Native stack：本地方法，jvm 和本地平台是有调用的，植入在操作系统里面，比方在 windows 里面会调用 windows 里面的方法，在 Linux 里面会调用 Linux 里面的方法，IO，文件都需要调用本地的方法。

在内存之上,有一个垃圾回收机制，我们如果写 C 语言，开辟了一个空间，需要自己 free 掉，如果不这样，这个 C 程序会一直占用这个空间，这样很快内存就会爆掉。Java 里面，我们只管 new，不考虑释放，Java 不用考虑对象回收，JVM 里面自己有几种算法来管理这个事情。

1 JVM 内存模型

Java 虚拟机在执行 Java 程序过程中，会把它所管理的内存划分为若干个不同的数据区，这些区域有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机的进程的启动而存在，有的区域则依赖用户进程的启动和结束而建立与销毁，我们将这些区域统称为 Java 运行时的数据区域。



JVM 架构图

1.1 名词解释

1.1.1 方法区（Method Area）

方法区（Method Area）与堆（Java Heap）一样，是各个线程共享的内存区域，它用于存储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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是她却有一个别名叫做非堆（Non-Heap）。分析下 Java 虚拟机规范，之所以把方法区描述为堆的一个逻辑部分，应该觉得她们都是存储数据的角度出发的。一个存储对象数据（堆），一个存储静态信息(方法区)。

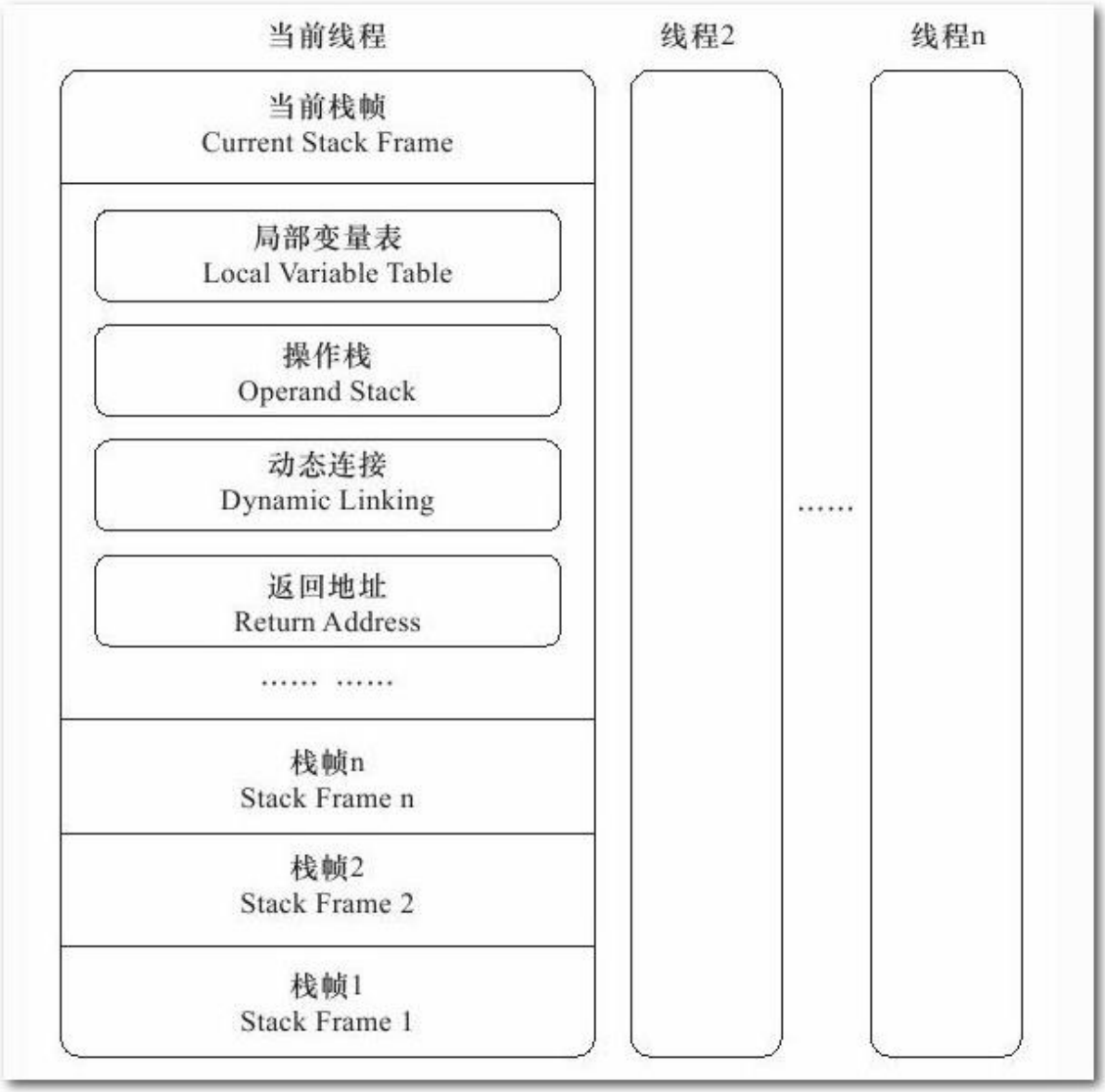
1.1.2 堆（Heap）

在 Java 虚拟机里面，堆是最大的一块，是随着 java 虚拟机的启动而创建的。在实际运用中，我们所创建的对象或者数组都是存放在这个区域，堆是一块共享区域，操作这块区域，如果考虑些线程安全的问题，需要加锁和同步。

与 Java Heap 相关的还有 Java 的垃圾回收机制（GC）,Java Heap 是垃圾回收器管理的主要区域。程序猿所熟悉的新生代、老生代、永久代的概念就是在堆里面，现在大多数的 GC 基本都采用了分代收集算法。Java Heap 可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。

1.1.3 栈（Stack）

相对于 Java Heap 来讲，Java Stack 是线程私有的，她的生命周期与线程相同。Java Stack 描述的是 Java 方法执行时的内存模型，每个方法执行时都会创建一个栈帧（Stack Frame）用语存储局部变量表、操作数栈、动态链接、方法出口等信息。从下图从可以看到，每个线程在执行一个方法时，都意味着有一个栈帧在当前线程对应的栈帧中入栈和出栈。



图中可以看到每一个栈帧中都有局部变量表。局部变量表存放了编译期间的各种基本数据类型，对象引用等信息。

1.1.4 本地方法栈（Native Stack）

本地方法栈（Native Stack）与 Java 虚拟机站（Java Stack）所发挥的作用非常相似，他们之间的区别在于虚拟机栈为虚拟机栈执行 java 方法（也就是字节码）服务，而本地方法栈则为使用到 Native 方法服务.

补充：什么是 Native Method

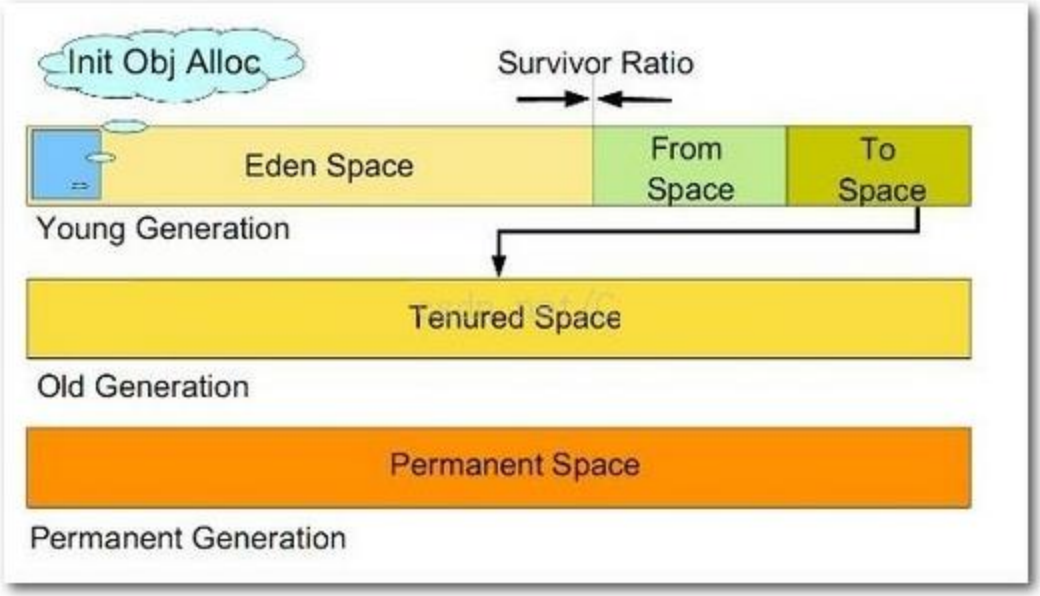
简单地讲，一个 Native Method 就是一个 java 调用非 java 代码的接口。一个 Native Method 是这样一个

java 的方法：该方法的实现由非 java 语言实现，比如 C。这个特征并非 java 所特有，很多其它的编程语言都有这一机制，比如在 C＋＋中，你可以用 extern "C"告知 C＋＋编译器去调用一个 C 的函数。

为什么我们将新生代、老生代、永久代三个概念一起说，那是因为 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 HotSpot 的垃圾收集器就能想管理 Java 堆一样管理这部分内存。简单点说就是 HotSpot 虚拟机中内存模型的分代，其中新生代和老生代在堆中，永久代使用方法区实现。根据官方发布的路线图信息，现在也有放弃永久代并逐步采用 Native Memory 来实现方法区的规划，在 JDK1.7 的 HotSpot 中，已经把原本放在永久代的字符串常量池移出。

，

2.GC 垃圾回收



上图解释：

新生代：产生的新的对象都需要放在新生代，首先是 Eden Space，对象进来后，过着无忧无虑的生活，然后过一段时间垃圾回收器来了，哪些人比较有价值，就留下来，没价值就杀了，有价值的是怎么留的呢？有价值的部分会被放到 survivor Ratio，如果被回收多次（比方 15 次）没有 kill 掉，就会被放到老年代。survivor Ratio 会被分为两块，一块是永远都不会放进去的，这是因为垃圾回收会考虑到算法的问题。

老年代：多次不回收的对象放到老年代，如果一个对象太大就会直接进入老年代。

永久区：就是我们前面说的方法区，里面定义了一些静态变量，常量，在之前是不回收的，后来也会存在永久区内存溢出的情况，这个时候就需要回收了。

新生代，老年代根据对象的存活年龄来划分的。

2.1 概念

Java 语言的一大特点就是可以进行自动垃圾回收处理，而无需开发人员过于关注系统资源，例如内存资源的释放情况。自动垃圾收集虽然大大减轻了开发人员的工作量，但是也

增加了软件系统的负担。

拥有垃圾收集器可以说是 Java 语言与 C++语言的一项显著区别。在 C++语言中，程序员必须小心谨慎地处理每一项内存分配，且内存使用完后必须手工释放曾经占用的内存空间。当内存释放不够完全时，即存在分配但永不释放的内存块，就会引起内存泄漏，严重时甚至导致程序瘫痪。

垃圾如果不回收，会导致内存泄漏，最终内存溢出。

2.2 垃圾回收的机制 判断有没有价值？

垃圾回收是怎么判断它是垃圾的呢？

主要有两种算法：对象 A 有没有被 B 应用，比方 A 被 B 引用，那么 A 就不是垃圾，如果引用就不算垃圾，那么如果 A 也引用 B，那么二者互相引用，是不是就回收不掉了呢？这个是早期的一种机制。

然后就是根节点搜索，就是看这个程序走走走，能走到哪些对象都不算垃圾，如果不能走到就算垃圾，至于怎么走，有一个映射表。

2.3 垃圾回收的算法 （算法[杀死]具体怎么回收）

那么具体怎么回收呢？有下面几种算法

2.3.1 标记-清除算法（Mark-Sweep）

1、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记的对象
2、在标记完成后统一回收所有被标记的对象

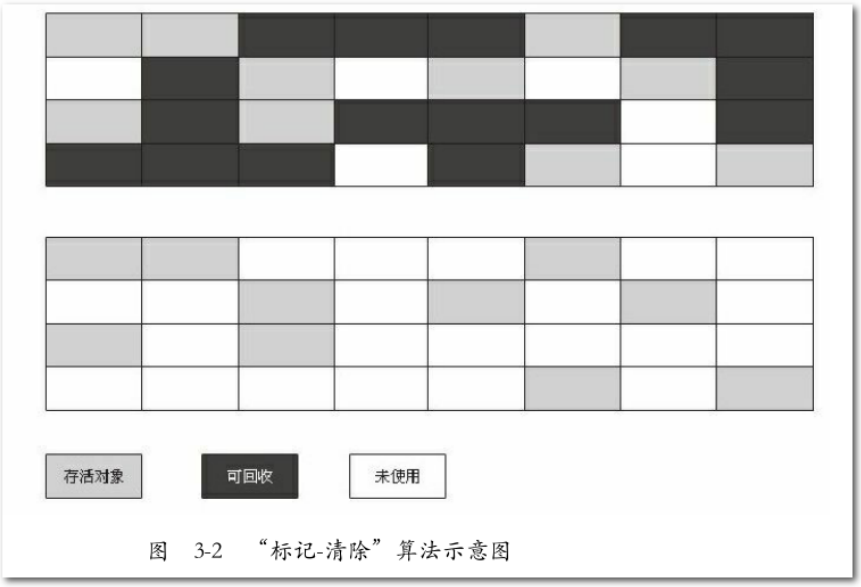


图 3-2 “标记-清除”算法示意图

缺点：一个是效率问题，标记和清除两个过程的效率都不高；
另一个是空间问题，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。
优点：效率高，很简单
缺点：一次回收这么多，回收的时候 jvm 很消耗资源，这个时候有可能会卡，其他程序可能没有资源

2.3.2 复制算法（Copying）

1、将可用内存按容量划分两块，每次只使用其中的一块。
2、当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

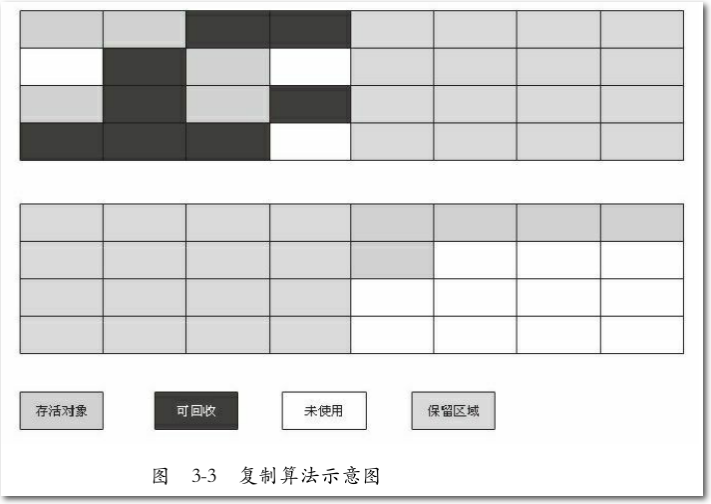


图 3-3 复制算法示意图

优点：这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。
缺点：复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低

2.3.3 整理算法（Mark-Compact） （一般最终采用方式）

1、标记
2、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

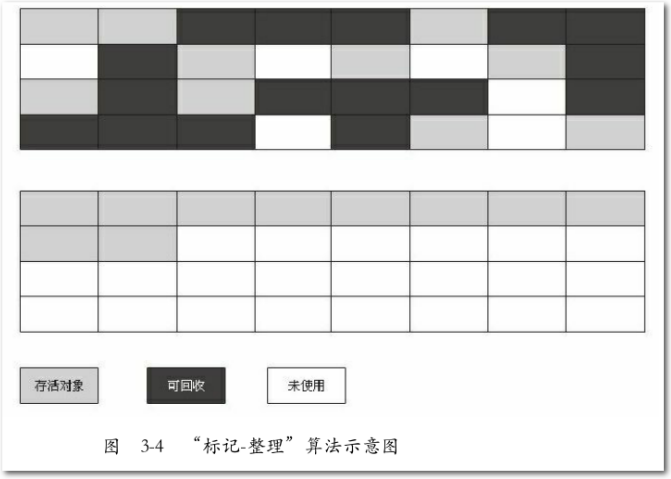


图 3-4 “标记-整理”算法示意图

2.3.4 分代收集策略（Generational Collection）

就是所谓的新生代，老年代，永久区，使用还是上面三种算法。

1、根据对象存活周期的不同将内存划分为几块。

2、一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

3、在新生代中，每次垃圾收集时都发现有大批对象死去，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成收集。

4、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。

2.4 垃圾回收策略 （使用收集器）

Serial 收集器：

1、是一个单线程的收集器，“Stop The World”

2、对于运行在 Client 模式下的虚拟机来说是一个很好的选择

4、简单而高效

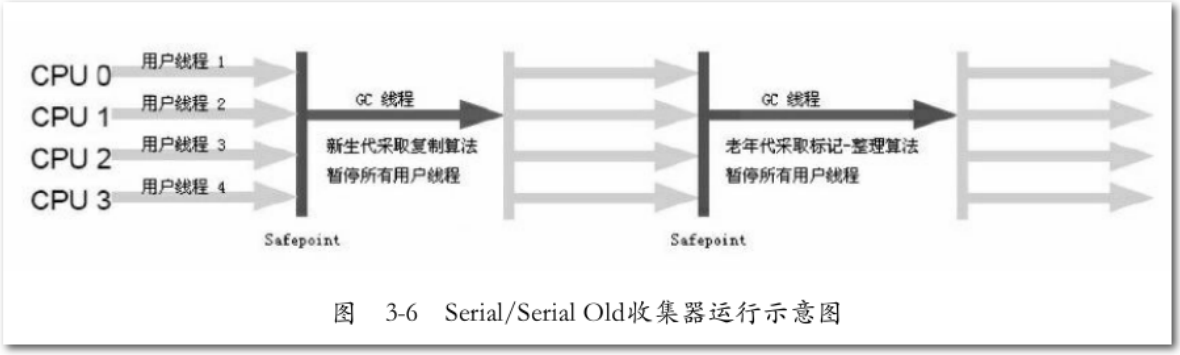


图 3-6 Serial/Serial Old收集器运行示意图

是一个单线程的操作，在垃圾回收的时候，只会执行垃圾回收这个操作，这样会让程序突然“卡一下”，这种方式，执行效率高，但是不适合一些有交互的场景。

Serial Old 收集器

1、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。

2、主要意义也是在于给 Client 模式下的虚拟机使用。

3、如果在 Server 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 收集器搭配使用[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 Failure 时使用。

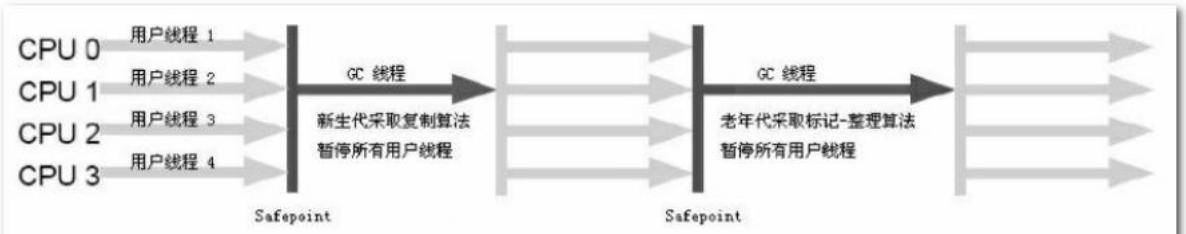


图 3-8 Serial/Serial Old收集器运行示意图

ParNew 收集器

1、Serial 收集器的多线程版本

2、单 CPU 不如 Serial

3、Server 模式下新生代首选,目前只有它能与 CMS 收集器配合工作

4、使用-XX：+UseConcMarkSweepGC 选项后的默认新生代收集器，也可以使用-XX：+UseParNewGC 选项来强制指定它。

5、-XX：ParallelGCThreads：限制垃圾收集的线程数。

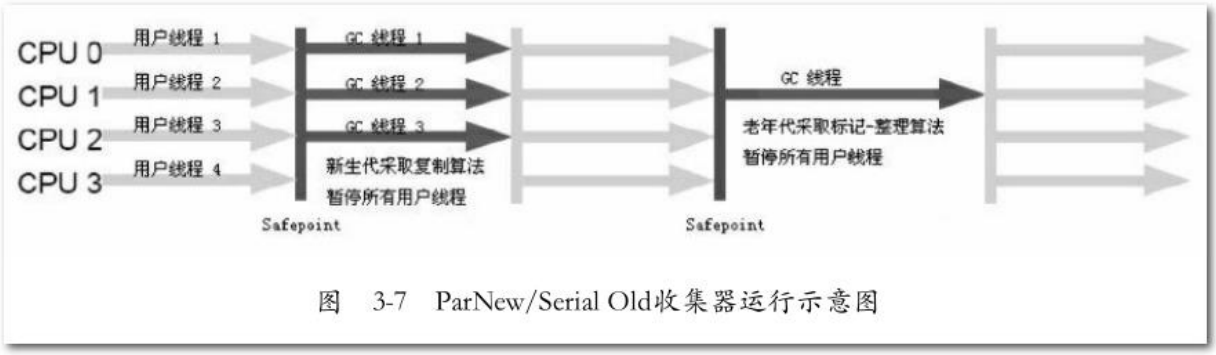


图 3-7 ParNew/Serial Old收集器运行示意图

Parallel Scavenge 收集器

1、吞吐量优先”收集器

2、新生代收集器，复制算法，并行的多线程收集器

3、目标是达到一个可控制的吞吐量（Throughput）。

4、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），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%。

5、两个参数用于精确控制吞吐量:

-XX：MaxGCPauseMillis 是控制最大垃圾收集停顿时间

-XX：GCTimeRatio 直接设置吞吐量大小

-XX：+UseAdaptiveSizePolicy:动态设置新生代大小、Eden 与 Survivor 区的比例、晋升老年代对象年龄

6、并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

7、并发（Concurrent）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户
程序在继续运行，而垃圾收集程序运行于另一个 CPU 上。

Parallel Old 收集器

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

2、在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

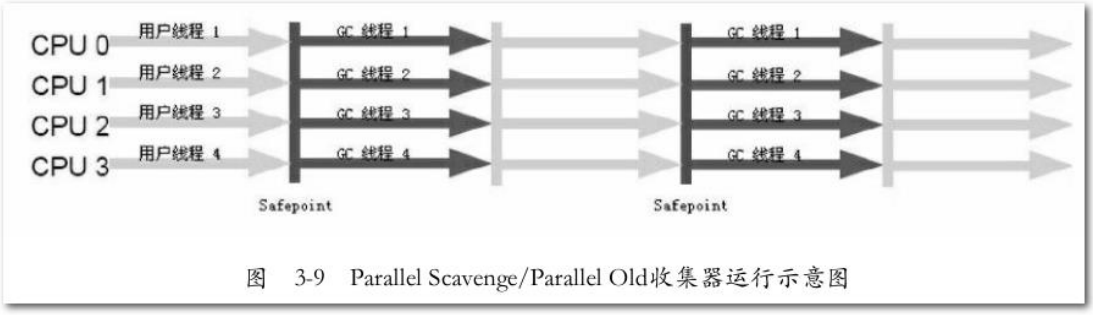


图 3-9 Parallel Scavenge/Parallel Old收集器运行示意图

CMS 收集器一款优秀的收集器

1、以获取最短回收停顿时间为目标的收集器。

2、非常符合互联网站或者 B/S 系统的服务端上，重视服务的响应速度，希望系统停顿时间最短的应用

3、基于“标记—清除”算法实现的

4、CMS 收集器的内存回收过程是与用户线程一起并发执行的

5、它的运作过程分为 4 个步骤，包括：

初始标记，“Stop The World”，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，并发标记阶段就是进行 GC RootsTracing 的过程

重新标记，Stop The World”，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，但远比并发标记的时间短

并发清除（CMS concurrent sweep）

6、优点：并发收集、低停顿

7、缺点：

对 CPU 资源非常敏感。

无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。

一款基于“标记—清除”算法实现的收集器

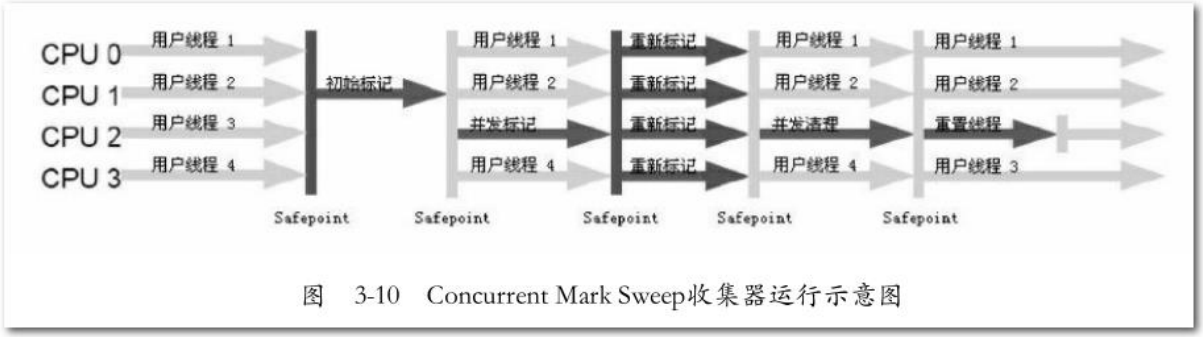


图 3-10 Concurrent Mark Sweep收集器运行示意图

G1（Garbage-First）收集器

1、当今收集器技术发展的最前沿成果之一

2、G1 是一款面向服务端应用的垃圾收集器。

3、优点：

并行与并发：充分利用多 CPU、多核环境下的硬件优势

分代收集：不需要其他收集器配合就能独立管理整个 GC 堆

空间整合：“标记—整理”算法实现的收集器，局部上基于“复制”算法不会产生内存空间碎片

可预测的停顿：能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒

4、G1 收集器的运作大致可划分为以下几个步骤：

初始标记：标记一下 GC Roots 能直接关联到的对象，需要停顿线程，但耗时很短

并发标记：是从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行

最终标记：修正在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标记记录

筛选回收：对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划

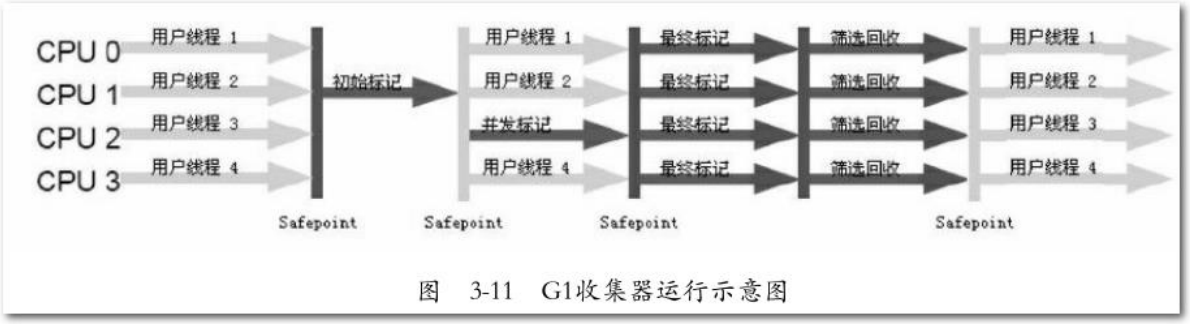


图 3-11 G1收集器运行示意图

2.5 小结

Gc 有多种策略，定期或者内存达到一个值后都会执行。

对象在创建后放在 Eden Space（伊甸园），当 Eden Space 满了的时候，gc 就把所有在 Eden Space 中的对象扫描一次，把所有有效的对象复制到第一个 Survivor Space，同时把无效的对象（垃圾）所占用的空间释放。当 Eden Space 再次变满了的时候，就启动移动程序把 Eden Space 中有效的对象复制到第二个 Survivor Space，同时，也将第一个 Survivor Space 中的有效对象复制到第二个 Survivor Space。如果填充到第二个 Survivor Space 中的有效对象被第一个 Survivor Space 或 Eden Space 中的对象引用，那么这些对象就是长期存在的，此时这些对象将被复制到 Permanent Generation。

若垃圾收集器依据这种小幅度的调整收集不能腾出足够的空间，就会运行 Full GC，此时 jvm gc 停止所有在堆中运行的线程并执行清除动作。

使用算法杀死，根据策略