深入理解 JVM

目录

前言

Java GC（Garbage Collection）垃圾回收机制，Java VM 中，存在自动内存管理和垃圾清理机制。GC 机制对 JVM（Java Virtual Machine）中的内存进行标记，并确定哪些内存需要回收，根据一定的回收策略，自动的回收内存，永不停息（Nerver Stop）的保证 JVM 中的内存空间，防止出现内存泄露和溢出问题。Java 中不能显式分配和注销内存。有些开发者把对象设置为 null 或者调用 System.gc()显式清理内存。设置为 null 至少没什么坏处，但是调用 System.gc()会一定程度上影响系统性能。Java 开发人员通常无须直接在程序代码中清理内存，而是由垃圾回收器自动寻找不必要的垃圾对象，并且清理掉它们。

Java GC 主要做三件事：

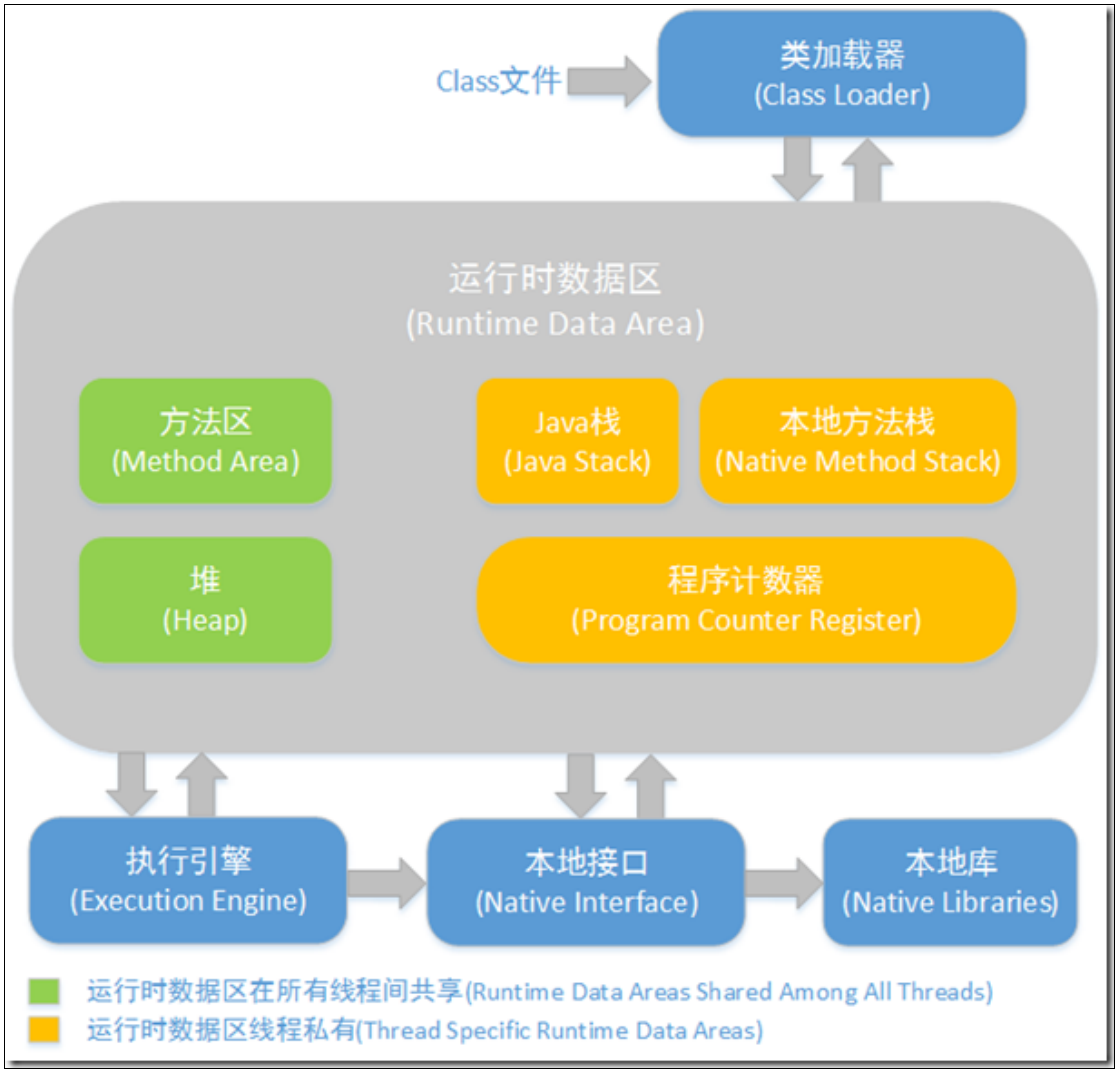
1、哪些内存需要 GC？

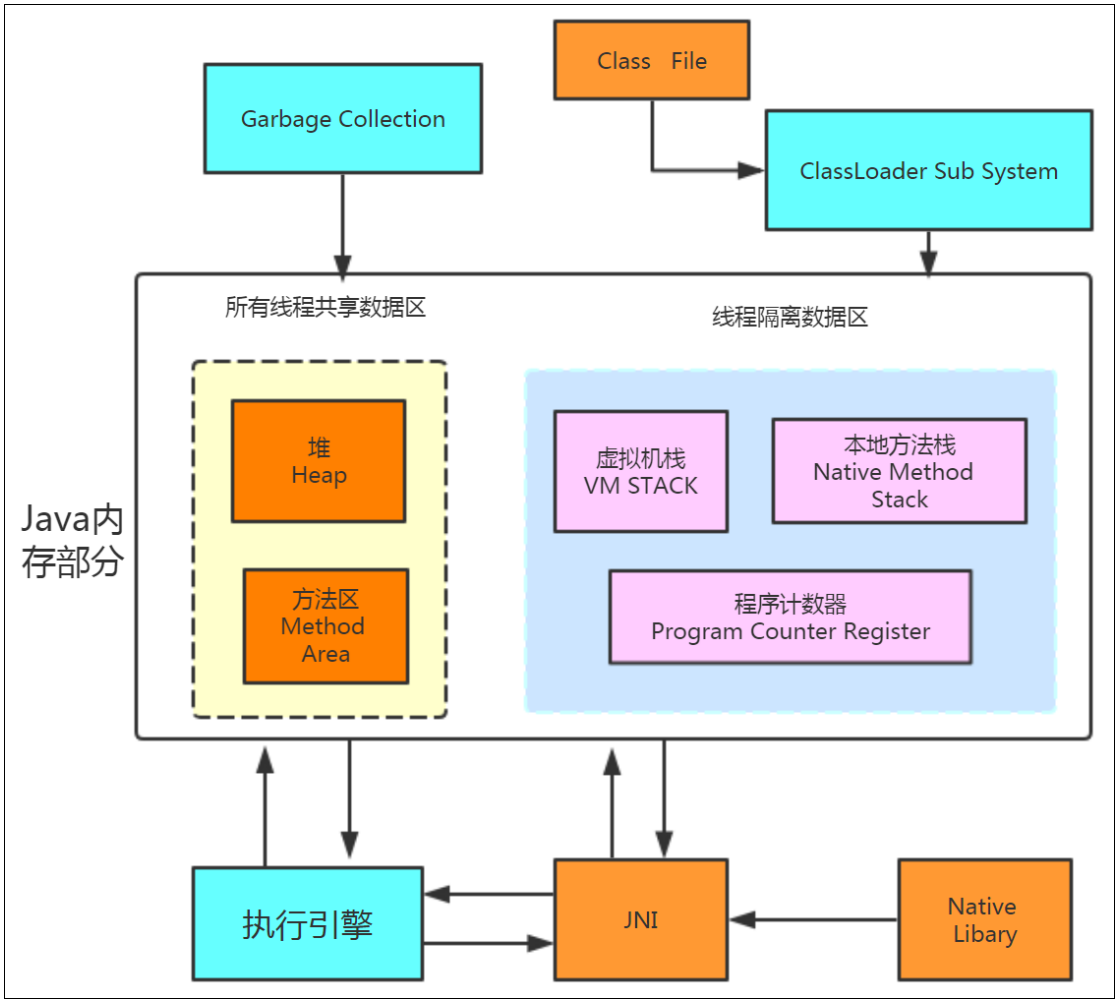
2、何时需要执行 GC？

3、以何策略执行 GC？

1、JVM 的组织结构

1.1 JVM 和系统调用之间的关系





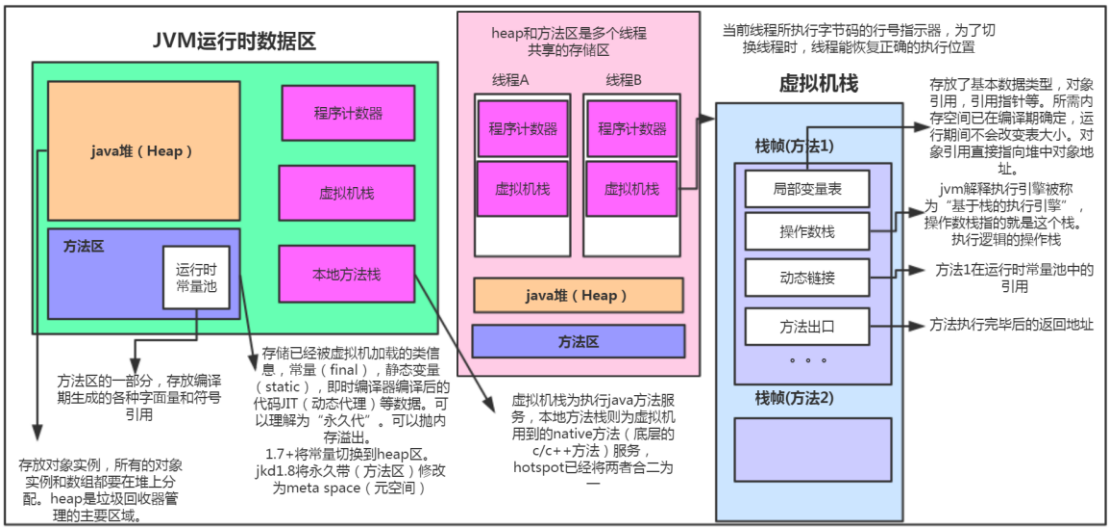
1、类加载器（ClassLoader）：在 JVM 启动时或者在类运行时将需要的 class 加载到 JVM 中

2、执行引擎：负责执行 class 文件中包含的字节码指令

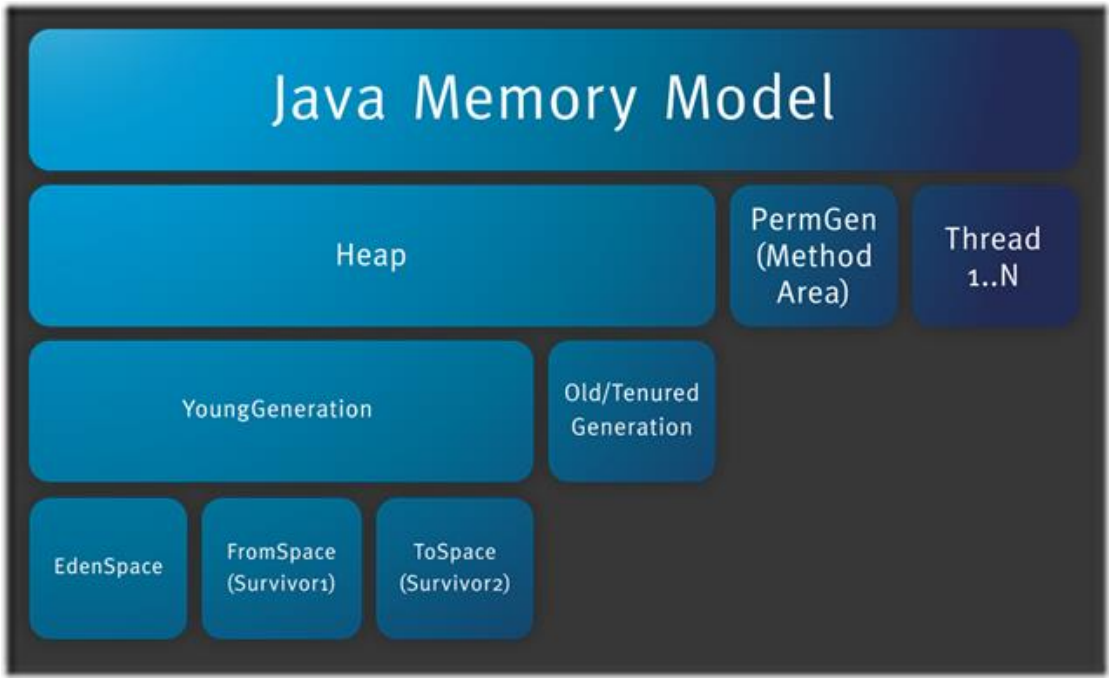
3、内存区（也叫运行时数据区）：是在 JVM 运行的时候操作所分配的内存区。运行时内存区主要可以划分为 5 个区域

4、本地方法接口：主要是调用 C 或 C++实现的本地方法及返回结果。

方法区和堆是所有线程共享的内存区域；
而 Java 栈、本地方法栈和程序员计数器是运行时线程私有的内存区域。



1.2、JVM 内存结构



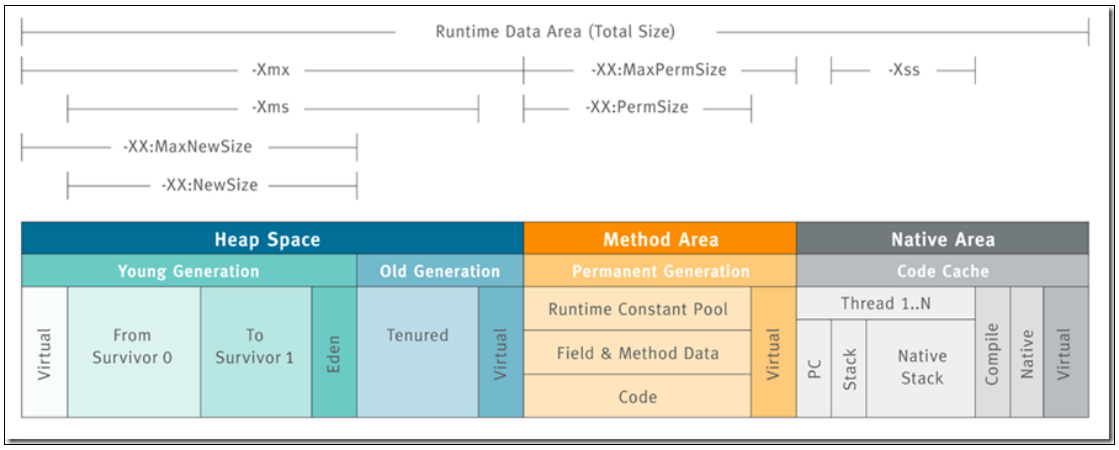
JVM 内存结构主要有三大块：**堆内存、方法区和栈**

堆内存是 JVM 中最大的一块由年轻代和老年代组成，而年轻代内存又被分成三部分，**Eden 空间、From Survivor 空间、To Survivor 空间**，默认情况下年轻代按照 8:1:1 的比例来分配

方法区存储类信息、常量、静态变量等数据，是线程共享的区域，为与 Java 堆区分，方法区还有一个别名 Non-Heap(非堆)

栈又分为 **Java 虚拟机栈**和**本地方法栈**和**程序计数器**，主要用于方法的执行

1.3、如何通过参数来控制各区域的内存大小



1.4、控制参数

-Xms 设置堆的最小空间大小
-Xmx 设置堆的最大空间大小
-XX:NewSize 设置新生代最小空间大小
-XX:MaxNewSize 设置新生代最大空间大小
-XX:PermSize 设置永久代最小空间大小
-XX:MaxPermSize 设置永久代最大空间大小
-Xss 设置每个线程的堆栈大小

没有直接设置老年代的参数，但是可以设置堆空间大小和新生代空间大小两个参数来间接控制：**老年代空间大小=堆空间大小-年轻代大空间大小**

2、JVM 各区域的作用

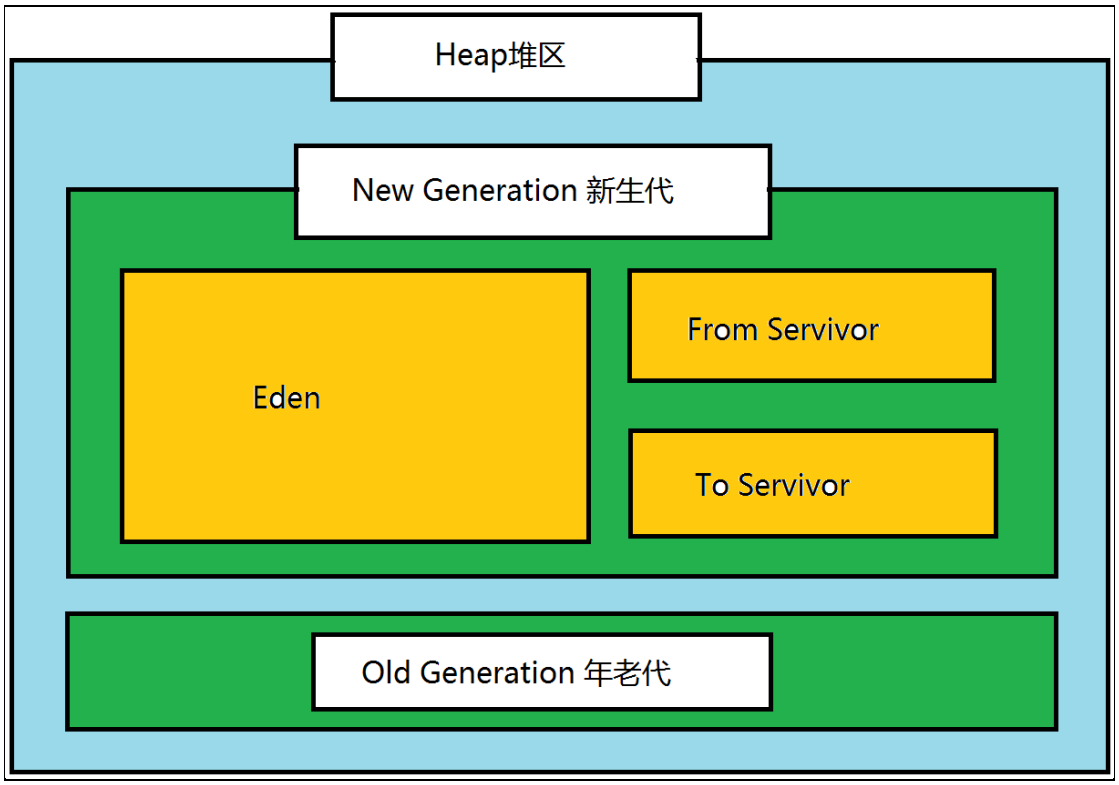
2.1、Java 堆（Heap）

对于大多数应用来说，Java 堆（Java Heap）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内存区域，在虚拟机启动时创建。此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都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以 Java 堆中还可以细分为：新生代和年老代；再细致一点话，新生代又分为 Eden 空间、From Survivor 空间、To Survivor 空间等。

根据 Java 虚拟机规范的规定，Java 堆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，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx 和-Xms 控制）。

如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError 异常



2.2、方法区（Method Area）

方法区（Method Area）与 Java 堆一样，是各个线程共享的内存区域，它用于**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**。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

> 对于习惯在 HotSpot 虚拟机上开发和部署程序的开发者来说，很多人愿意把方法区称为“永久代”（Permanent Generation），本质上两者并不等价，仅仅是因为 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。

----这段话是对方法区和永久代的区别解释

Java 虚拟机规范对这个区域的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个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，一般来说这个区域的回收“成绩”比较难以令人满意，尤其是类型的卸载，条件相当苛刻，但是这部分区域的回收确实是有必要的。

根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

2.3、程序计数器（Program Counter Register）

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的**行号指示器**。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），**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**。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“**线程私有**”的内存。

如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Natvie 方法，这个计数器值则为空（Undefined）。

此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

2.4、Java 虚拟机栈（JVM Stacks）

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：**每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。**

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和 returnAddress 类型（指向了一条字节码指令的地址）。

其中 64 位长度的 long 和 double 类型的数据会占用 2 个局部变量空间（Slot），其余的数据类型只占用 1 个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在 Java 虚拟机规范中，对这个区域规定了两种异常状况：
1、**如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常**；

2、如果虚拟机栈可以动态扩展（当前大部分的 Java 虚拟机都可动态扩展，只不过 Java 虚拟机规范中也允许固定长度的虚拟机栈），当**扩展时无法申请到足够的内存时会抛出 OutOfMemoryError 异常**。

2.5、本地方法栈（Native Method Stacks）

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是**Java 虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的 Native 方法服务**。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryError 异常。

3、JVM 垃圾回收

3.1、概述

垃圾收集 Garbage Collection 通常被称为“GC”，它诞生于 1960 年 MIT 的 Lisp 语言，经过半个多世纪，目前已经十分成熟了。

JVM 中，程序计数器、虚拟机栈、本地方法栈都是随线程而生随线程而灭，栈帧随着方法的进入和退出做入栈和出栈操作，实现了自动的内存清理，因此，我们的内存垃圾回收主要集中于 Java 堆和方法区中，在程序运行期间，这部分内存的分配和使用都是动态的。

3.2、垃圾收集器(Garbage Collector (GC)) 是什么？

GC 其实是一种自动的内存管理工具，其行为主要包括 2 步

1、在 Java 堆中，为新创建的对象分配空间
2、在 Java 堆中，回收没用的对象占用的空间

3.3、为什么需要 GC？

应用程序的回收目标是构建一个仅用来处理内存分配，而不执行任何真正的内存回收操作的 GC。即仅当可用的 Java 堆耗尽的时候，才进行顺序的 JVM 停顿操作。首先需要理解为什么需要 GC。随着应用程序所应对的业务越来越庞大、复杂，用户越来越多，没有 GC 就不能保证应用程序正常进行。而经常造成 STW（Stop The World）的 GC 又跟不上实际的需求，所以才会不断地尝试对 GC 进行优化。

3.4、为什么需要多种 GC?

首先，Java 平台被部署在各种各样的硬件资源上，其次，在 Java 平台上部署和运行着各种各样的应用，并且用户对不同的应用的性能指标(吞吐率和延迟)预期也不同，为了满足不同应用的对内存管理的不同需求，JVM 提供了多种 GC 以供选择

性能指标：
最大停顿时长：垃圾回收导致的应用停顿时间的最大值
吞吐率：垃圾回收停顿时长和应用运行总时长的比例

不同的 GC 能满足不同应用不同的性能需求，现有的 GC 包括：

1、**序列化 GC(serial garbage collector)**：适合占用内存少的应用
2、**并行 GC 或吞吐率 GC(Parallel or throughput Garbage Collector)**：适合占用内存较多，多 CPU，追求高吞吐率的应用
3、**并发 GC**：适合占用内存较多，多 CPU 的应用，对延迟有要求的应用

3.5、对象存活的判断

判断对象是否存活一般有两种方式：

1、引用计数：每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，缺点是无法解决对象相互循环引用的问题。

2、可达性分析（Reachability Analysis）：从 GC Roots 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时，则证明此对象是不可用的不可达对象。

在 Java 语言中，GC Roots 包括：

1、虚拟机栈中引用的对象
2、方法区中类静态属性实体引用的对象
3、方法区中常量引用的对象
4、本地方法栈中 JNI 引用的对象

由于循环引用的问题，一般采用跟踪（可达性分析）方法

3.6、并发和并行

这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下

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。

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。

3.7、Minor GC 和 Full GC

两者的概念和区别：

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的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。

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（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。Major GC 的速度一般会比 Minor GC 慢 10 倍以上。

3.6、垃圾回收算法

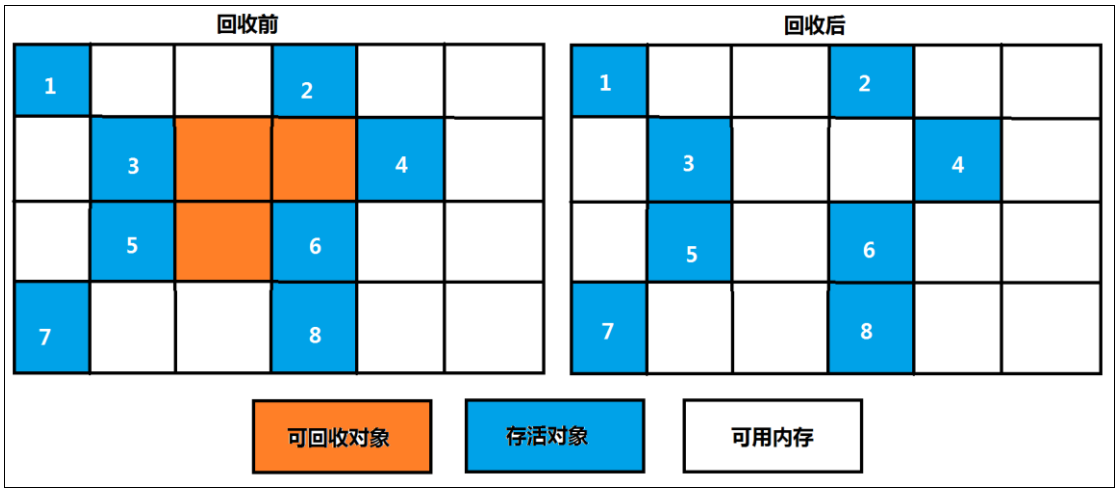
3.6.1、标记-清除算法

“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：

1、一个是效率问题，标记和清除过程的**效率都不高**；

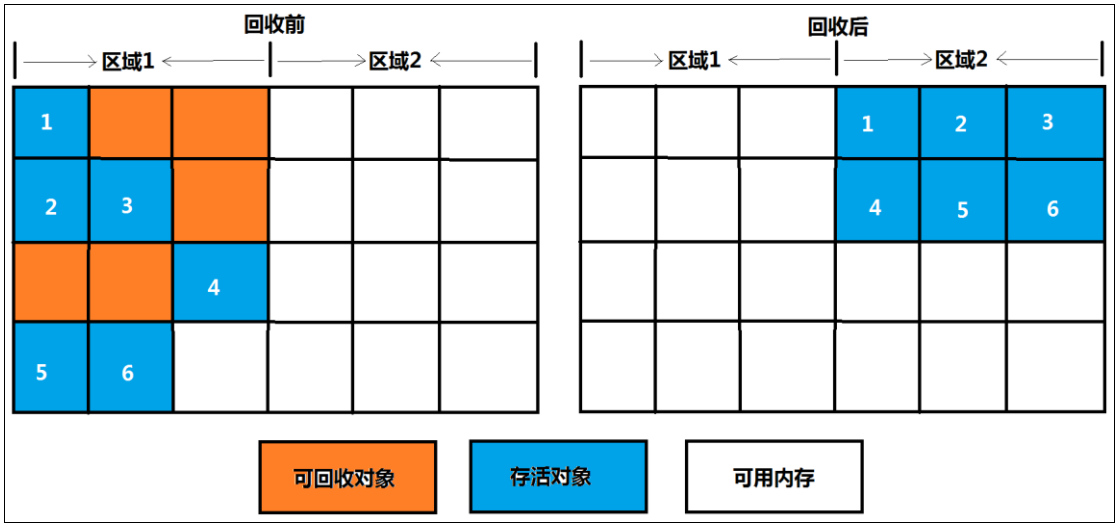
2、另外一个是空间问题，标记清除之后会产生大量不连续的**内存碎片**，空间碎片太多可能会导致，当程序在以后的运行过程中**需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作**。



3.6.2、复制算法

“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。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。

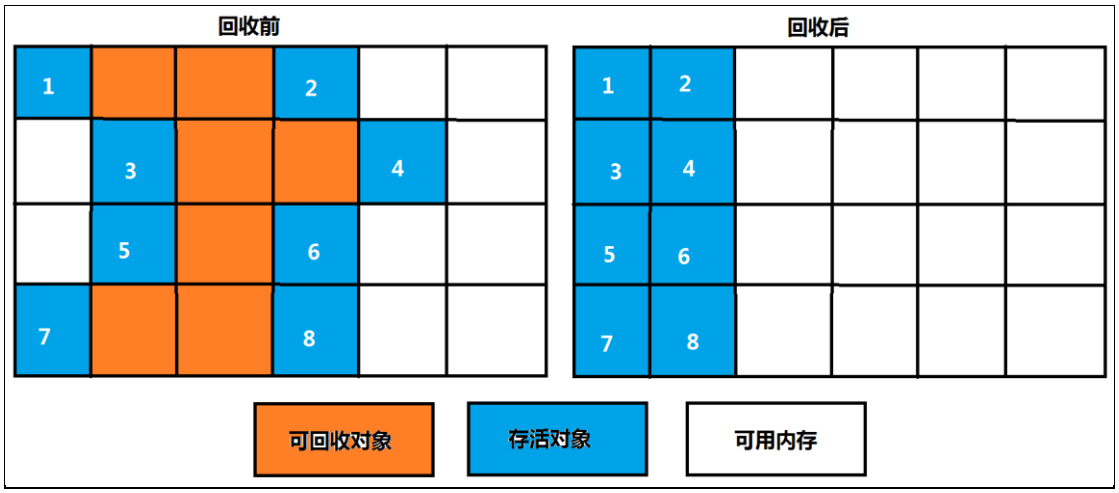
这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种**算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。**



3.6.3、标记-整理算法

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



3.6.4、分代收集算法

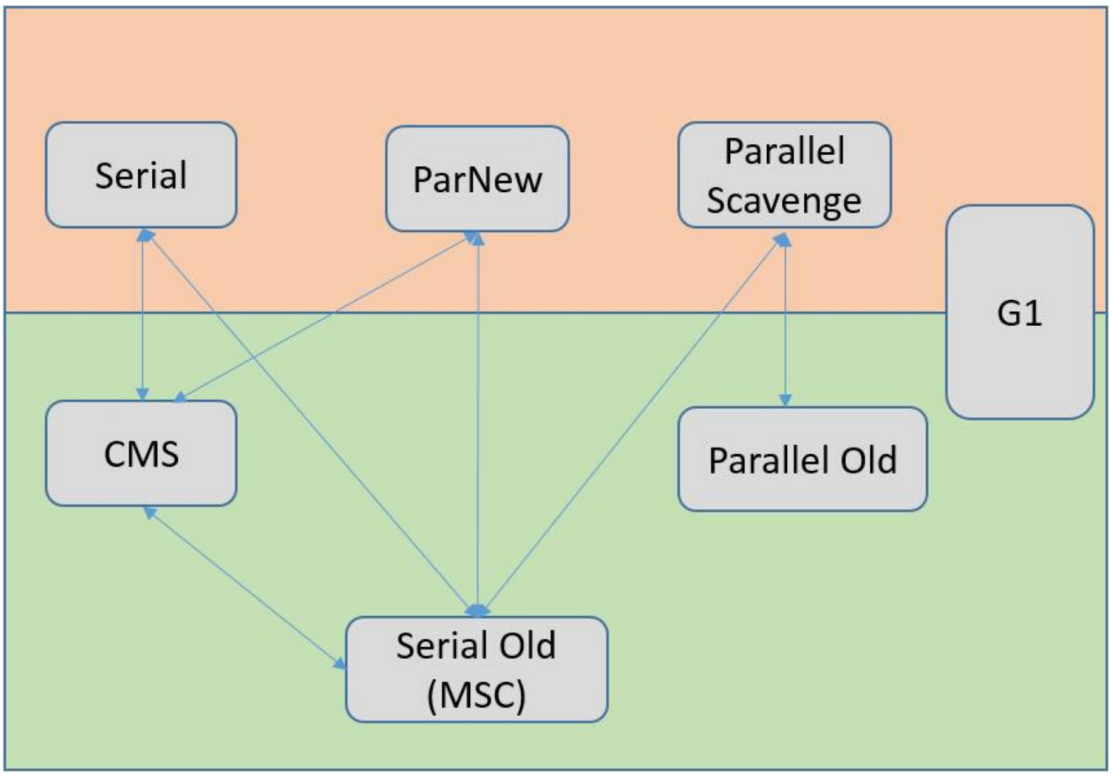
GC 分代的基本假设：绝大部分对象的生命周期都非常短暂，存活时间短。谷歌的测试表明：98%以上的对象都只是被使用一次。

“分代收集”（Generational Collection）算法，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。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。而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“标记-整理”算法来进行回收。

总结：
新生代：存活对象数量少，复制算法
年老代：对象存活率高，标记-清除 或者 标记-整理

3.7、垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现，不同厂商、不同版本的虚拟机实现差别很大，HotSpot 中包含的收集器如下：

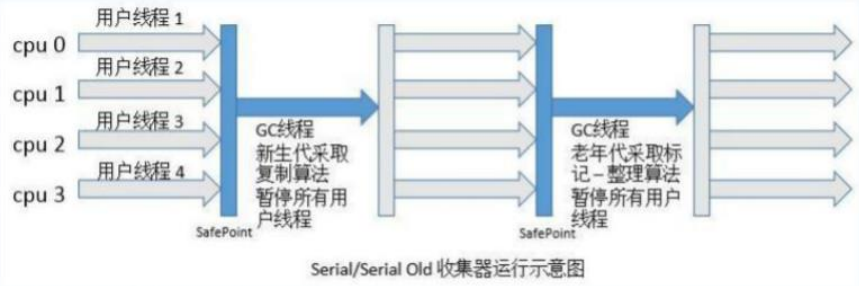


3.7.1、Serial 收集器

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会 **Stop The World**（服务暂停）

参数控制：-XX:+UseSerialGC　串行收集器

■ Serial（串行）收集器/Serial Old收集器



Serial/Serial Old 收集器运行示意图

1：是一个单线程的收集器，在垃圾收集时，会Stop-the-World
2：优点是简单，对于单cpu，由于没有多线程的交互开销，可能更高效，是默认的Client模式下的新生代收集器
3：使用-XX:+UseSerialGC来开启
会使用：Serial + Serial Old 的收集器组合
4：新生代使用复制算法，老年代使用标记-整理算法

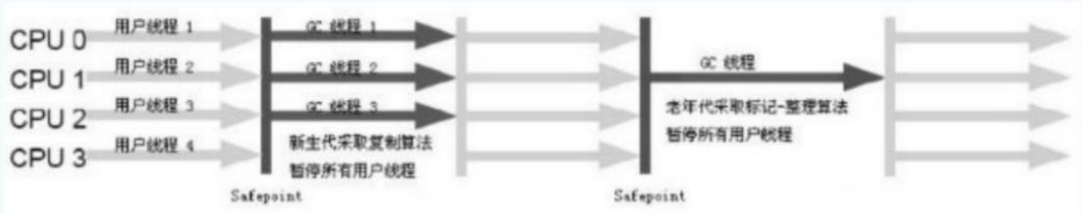
总结：单线程的 STW 模式

3.7.2、ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本。
新生代并行，老年代串行；新生代复制算法、老年代标记-整理

参数控制：
-XX:+UseParNewGC　　ParNew 收集器
-XX:ParallelGCThreads　　限制线程数量

■ ParNew（并行）收集器



1：使用多线程进行垃圾回收，在垃圾收集时，会Stop-the-World
2：在并发能力好的CPU环境里，它停顿的时间要比串行收集器短；但对于单cpu或并发能力较弱的CPU，由于多线程的交互开销，可能比串行回收器更差
3：是Server模式下首选的新生代收集器，且能和CMS收集器配合使用
4：使用-XX:+UseParNewGC来开启
会使用：ParNew + Serial Old的收集器组合
5：-XX:ParallelGCThreads：指定线程数，最好与CPU数量一致
6：新生代使用复制算法，老年代采用标记-整理算法

总结：Serial 收集器的多线程版本

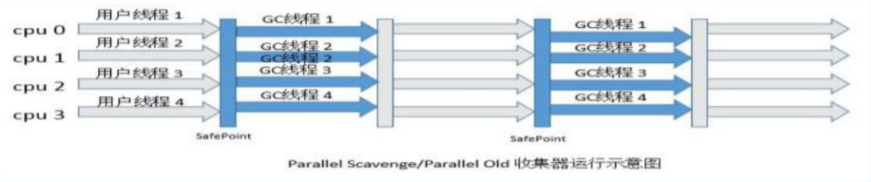
3.7.3、Parallel 收集器

Parallel Scavenge 收集器类似 ParNew 收集器，**Parallel 收集器更关注系统的吞吐量**。可以通过参数来打开自适应调节策略，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最大的吞吐量；也可以**通过参数控制 GC 的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-整理算法**

参数控制：-XX:+UseParallelGC　　使用 Parallel 收集器 + 老年代串行

■ 新生代Parallel Scavenge收集器/Parallel Old收集器

是一个应用于新生代的、使用复制算法的、并行的收集器，跟ParNew很类似，但更关注吞吐量（CPU吞吐量就是运行应用代码的时间/总运行时间，这种收集器能最高效率的利用CPU，适合运行后台应用）。



Parallel Scavenge/Parallel Old 收集器运行示意图

1：使用-XX:+UseParallelGC来开启新生代Parallel Scavenge收集器
2：使用-XX:+UseParallelOldGC来开启老年代使用Parallel Old收集器，使用Parallel Scavenge + Parallel Old的收集器组合
3：-XX:GCTimeRatio：指定运行应用代码的时间占总时间的比例，默认99，即1%的时间用来进行垃圾收集
4：-XX:MaxGCPauseMillis：设置GC的最大停顿时间
5：新生代使用复制算法，老年代使用标记-整理算法

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%。

总结：关注吞吐量的多线程 ParNew 进化版

3.7.4、CMS 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。目前很大一部分的 Java 应用都集中在互联网站或 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出 CMS 收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

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）

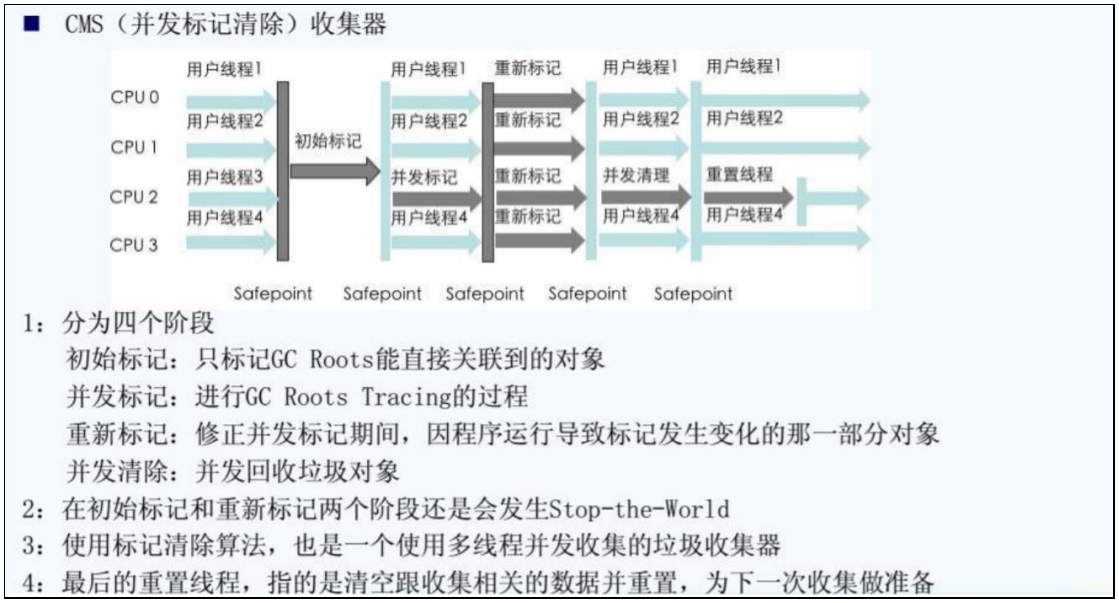
其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing 的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，**CMS 收集器的内存回收过程是与用户线程一起并发地执行**。老年代收集器（新生代使用 ParNew）

优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量

参数控制：

-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）



总结：追求最短回收停顿时间的业务线程和 GC 线程并行运行的垃圾收集器

3.7.5、G1 收集器

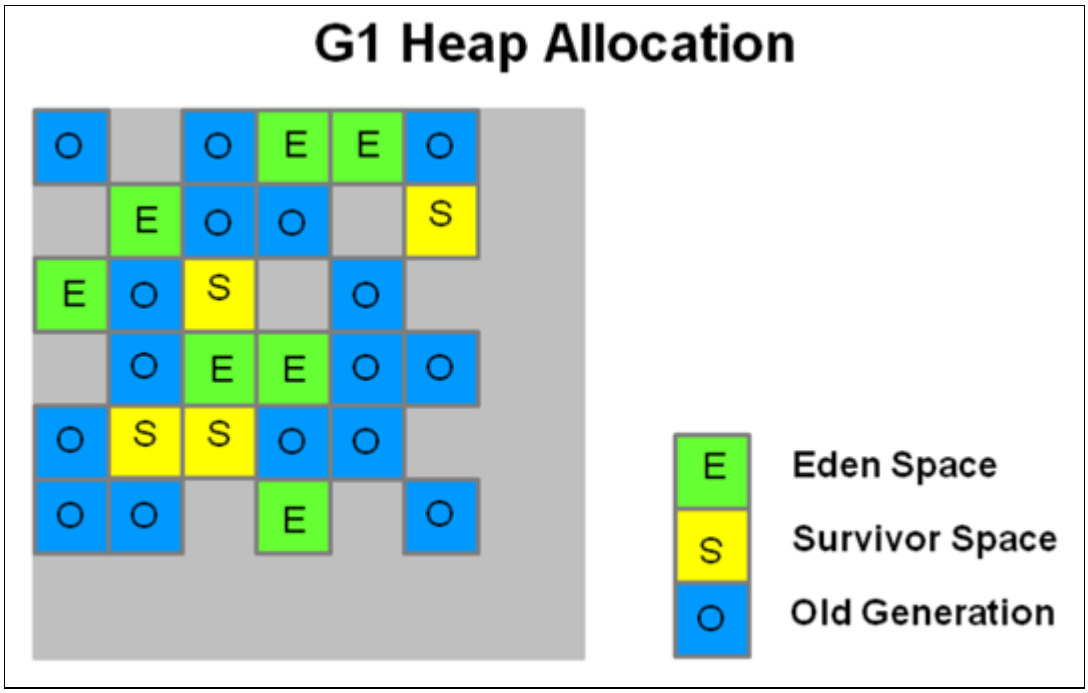
G1 是目前技术发展的最前沿成果之一，HotSpot 开发团队赋予它的使命是未来可以替换掉 JDK1.5 中发布的 CMS 收集器。与 CMS 收集器相比 G1 收集器有以下特点：

1、空间整合，**G1 收集器采用标记整理算法**，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次 GC。

2、可预测停顿，这是 G1 的另一大优势，降低停顿时间是 G1 和 CMS 的共同关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 N 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，这几乎已经是实时 Java（RTSJ）的垃圾收集器的特征了。

之前提到的垃圾收集器，收集的范围都是整个新生代或者老年代，而 G1 不再是这样。使用 **G1 收集器时，Java 堆的内存布局与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region 的集合**。

下面这张图是 G1 对 Heap 的划分：



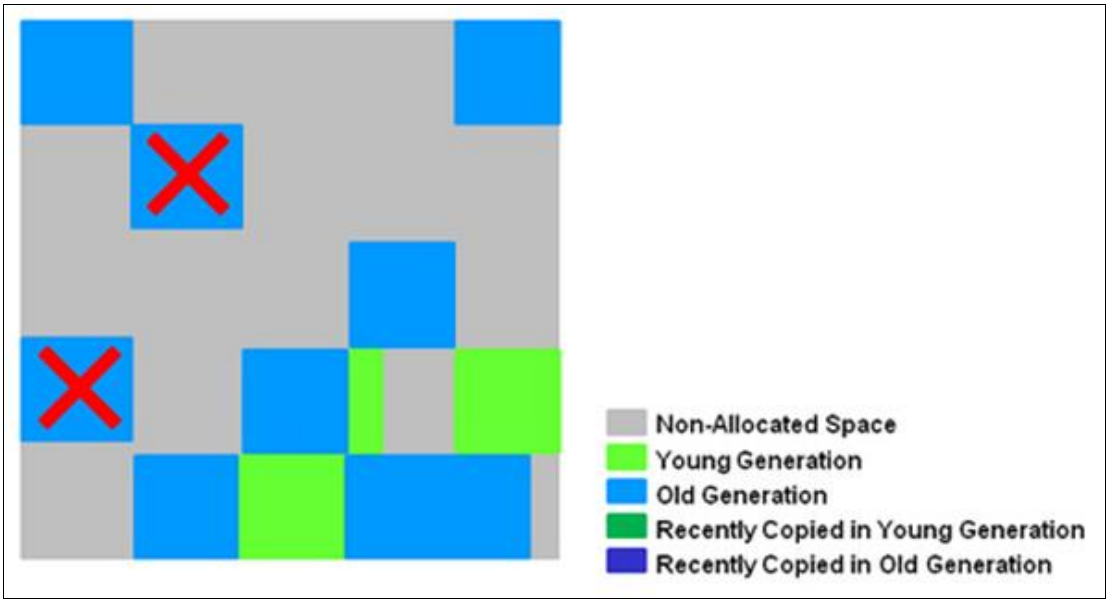
G1 的新生代收集跟 ParNew 类似，当新生代占用达到一定比例的时候，开始触发收集。和 CMS 类似，G1 收集器收集老年代对象会有短暂停顿。

收集步骤：

1、标记阶段，首先初始标记(Initial-Mark)，这个阶段是停顿的(Stop the World)，并且会触发一次普通 Minor GC。

2、Root Region Scanning，程序运行过程中会回收 survivor 区(存活到老年代)，这一过程必须在 young GC 之前完成。

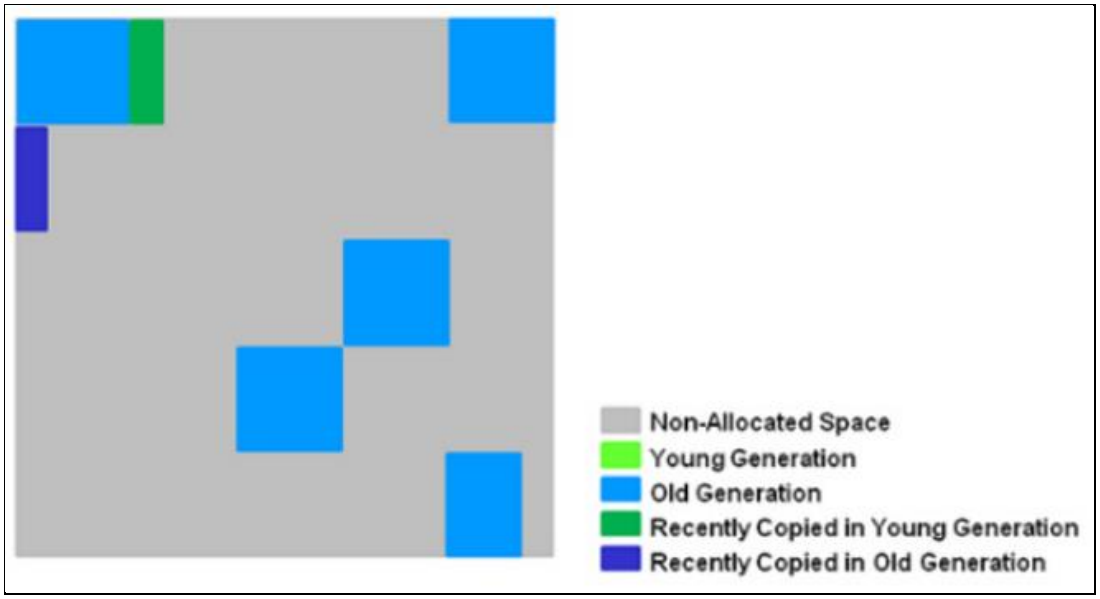
3、Concurrent Marking，在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被 young GC 中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那个这个区域会被立即回收(图中打 X)。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。



4、Remark，再标记，会有短暂停顿(STW)。再标记阶段是用来收集并发标记阶段产生新的垃圾(并发阶段和应用程序一同运行)；G1 中采用了比 CMS 更快的初始快照算法:snapshot-at-the-beginning (SATB)。

5、Copy/Clean up，多线程清除失活对象，会有 STW。G1 将回收区域的存活对象拷贝到新区域，清除 Remember Sets，并发清空回收区域并把它返回到空闲区域链表中。

6、复制/清除过程后。回收区域的活性对象已经被集中回收到深蓝色和深绿色区域。



G1 与 CMS 对比有一下不同：

1、分代：CMS 中，堆被分为 PermGen，YoungGen，OldGen；而 YoungGen 又分了两个 survivo 区域。在 G1 中，堆被平均分成几个区域(region)，在每个区域中，虽然也保留了新老代的概念，但是收集器是以整个区域为单位收集的。

2、算法：相对于 CMS 的“标记--清理”算法，G1 会使用“标记--整理”算法，保证不产生多余的碎片。收集阶段，G1 会将某个区域存活的对象拷贝的其他区域，然后将整个区域整个回收。

3、停顿时间可控：为了缩短停顿时间，G1 建立可预存停顿模型，这样在用户设置的停顿时间范围内，G1 会选择适当的区域进行收集，确保停顿时间不超过用户指定时间。

3.8、常用的收集器组合

	新生代 GC 策略	老年代 GC 策略	说明
组合 1	Serial	Serial Old	Serial 和 Serial Old 都是单线程进行 GC，特点就是 GC 时暂停所有应用线程。
组合 2	Serial	CMS + Serial Old	CMS（Concurrent Mark Sweep）是并发 GC，实现 GC 线程和应用线程并发工作，不需要暂停所有应用线程。另外，当 CMS 进行 GC 失败时，会自动使用 Serial Old 策略进行 GC。
组合 3	ParNew	CMS	使用-XX:+UseParNewGC 选项来开启。ParNew 是 Serial 的并行版本，可以指定 GC 线程数，默认 GC 线程数为 CPU 的数量。可以使用-XX:ParallelGCThreads 选项指定 GC 的线程数。 如果指定了选项-XX:+UseConcMarkSweepGC 选项，则新生代默认使用 ParNew GC 策略。

组合 4	ParNew	Serial Old	使用-XX:+UseParNewGC 选项来开启。新生代使用 ParNew GC 策略，年老代默认使用 Serial Old GC 策略。
组合 5	Parallel Scavenge	Serial Old	Parallel Scavenge 策略主要是关注一个可控的吞吐量：应用程序运行时间/(应用程序运行时间+ GC 时间)，可见这会使得 CPU 的利用率尽可能的高，适用于后台持久运行的应用程序，而不适用于交互较多的应用程序。
组合 6	Parallel Scavenge	Parallel Old	Parallel Old 是 Serial Old 的并行版本
组合 7	G1GC	G1GC	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC #开启 -XX:MaxGCPauseMillis =50 #暂停时间目标 -XX:GCPauseIntervalMillis =200 #暂停间隔目标 -XX:+G1YoungGenSize=512m #年轻代大小 -XX:SurvivorRatio=6 #幸存区比例

4、JVM 参数列表

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-Xmx3550m：最大堆内存为 3550M

-Xms3550m：初始堆内存为 3550m

此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g：设置年轻代大小为 2G

整个堆大小 = 年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss1m：设置每个线程的堆栈大小

JDK5.0 以后每个线程堆栈大小为 1M，在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

-XX:NewRatio=4：设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1:4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。

设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m：设置持久代大小为 16m。

-XX:MaxTenuringThreshold=15：设置垃圾最大年龄。

如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象 再年轻代的存活时间，增加在年轻代即被回收的概率。

收集器设置
-XX:+UseSerialGC 设置串行收集器
-XX:+UseParallelGC 设置并行收集器
-XX:+UseParalledlOldGC 设置并行年老代收集器
-XX:+UseConcMarkSweepGC 设置并发收集器

垃圾回收统计信息
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Xloggc:filename

并行收集器设置
-XX:ParallelGCThreads=n 设置并行收集器收集时使用的 CPU 数。并行收集线程数
-XX:MaxGCPauseMillis=n 设置并行收集最大暂停时间
-XX:GCTimeRatio=n 设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)

并发收集器设置
-XX:+CMSIncrementalMode 设置为增量模式。适用于单 CPU 情况
-XX:ParallelGCThreads=n 设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数
并行收集线程数

4.1、垃圾收集器参数总结

收集器设置：
-XX:+UseSerialGC 年轻串行（Serial），老年串行（Serial Old）
-XX:+UseParNewGC 年轻并行（ParNew），老年串行（Serial Old）
-XX:+UseConcMarkSweepGC 年轻并行（ParNew），老年串行（CMS），备份（Serial Old）
-XX:+UseParallelGC 年轻并行吞吐（Parallel Scavenge），老年串行（Serial Old）
-XX:+UseParalledlOldGC 年轻并行吞吐（Parallel Scavenge），老年并行吞吐（Parallel Old）
-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC 使用 G1 收集器
-XX:+UseConcMarkSweepGC 使用 CMS 收集器

收集器参数：
-XX:ParallelGCThreads=n 设置并行收集器收集时使用的 CPU 数。并行收集线程数
-XX:MaxGCPauseMillis=n 设置并行收集最大暂停时间
-XX:GCTimeRatio=n 设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)
-XX:+CMSIncrementalMode 设置为增量模式。适用于单 CPU 情况
-XX:ParallelGCThreads=n 设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数
并行收集线程数

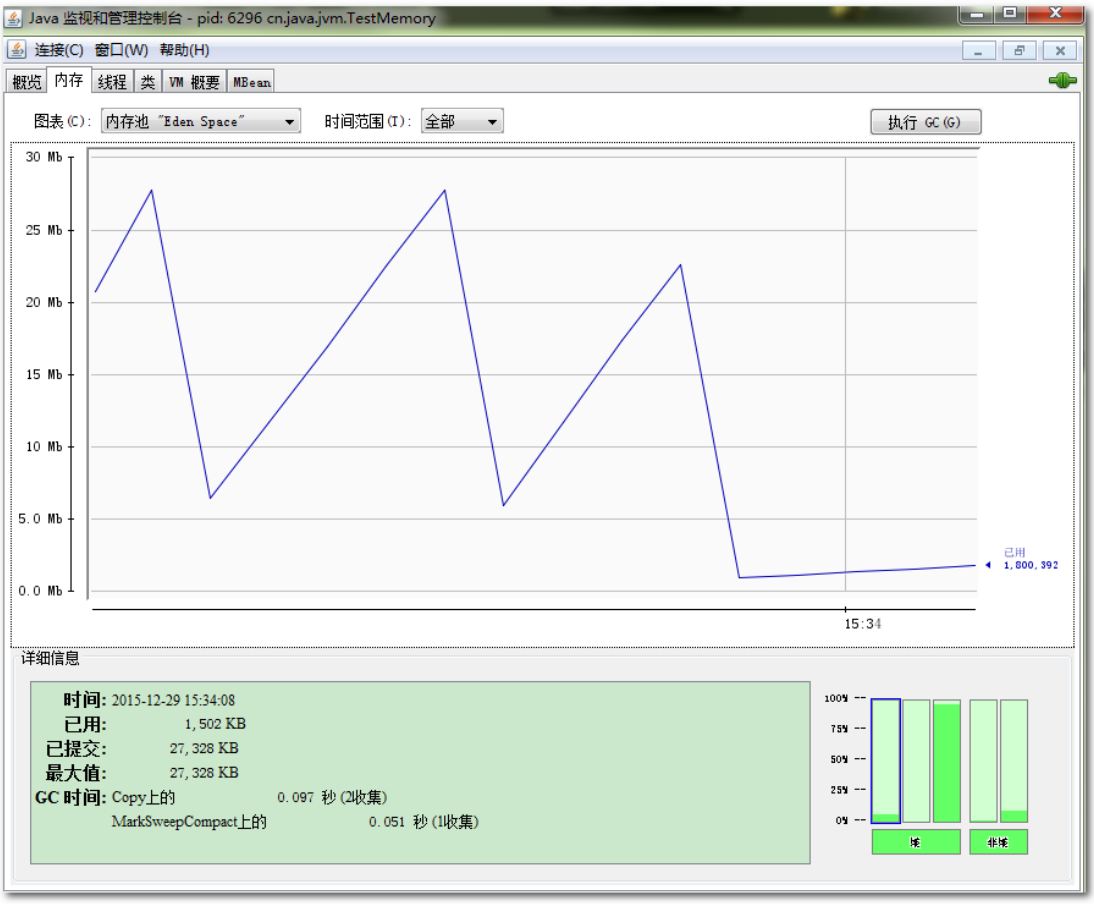
5、JVM 案例演示

5.1、内存

内存标签相当于可视化的 jstat 命令，用于监视收集器管理的虚拟机内存（Java 堆和永久代）的变化趋势。

我们通过下面的一段代码体验一下它的监视功能。运行时设置的虚拟机参数为：-Xms100m -Xmx100m -XX:+UseSerialGC，这段代码的作用是以 64kb/50ms 的速度往 Java 堆内存中填充数据。

```
package cn.java.jvm;

import java.util.ArrayList;

public class TestMemory {
    static class OOMObject {
        public byte[] placeholder = new byte[64 * 1024];
    }

    public static void fillHeap(int num) throws Exception {
        ArrayList<OOMObject> list = new ArrayList<OOMObject>();
        for (int i = 0; i < num; i++) {
            Thread.sleep(50);
            list.add(new OOMObject());
        }
        System.gc();
    }

    public static void main(String[] args) throws Exception {
        fillHeap(1000);
        Thread.sleep(500000);
    }
}
```



从图中可以看出，运行轨迹成曲线增长，循环 1000 次后，虽然整个新生代 Eden 和 Survivor 区都基本上被清空了，但是老年代仍然保持峰值状态，这说明，填充的数据在 GC 后仍然存活，因为 list 的作用域没有结束。如果把 System.gc()；移到 fillHeap(1000);后，就可以全部回收掉。

5.2、线程

线程相当于可视化了 jstack 命令，遇到线程停顿时，可以使用这个也签进行监控分析。线程长时间停顿的主要原因有：等待外部资源（数据库连接等），死循环、锁等待。下面的代码将演示这几种情况：

```
package cn.java.jvm;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class TestThread {
    /**
     * 死循环演示
     */
    public static void createBusyThread() {
        Thread thread = new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                System.out.println("createBusyThread");
                while (true)
                    ;
            }
        }, "testBusyThread");
        thread.start();
    }

    /**
     * 线程锁等待
     */
    public static void createLockThread(final Object lock) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("createLockThread");
                synchronized (lock) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

            }
        }, "testLockThread");
        thread.start();
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        br.readLine();
        createBusyThread();
        br.readLine();
        Object object = new Object();
        createLockThread(object);
    }
}
```

main 线程：追踪到需要键盘录入

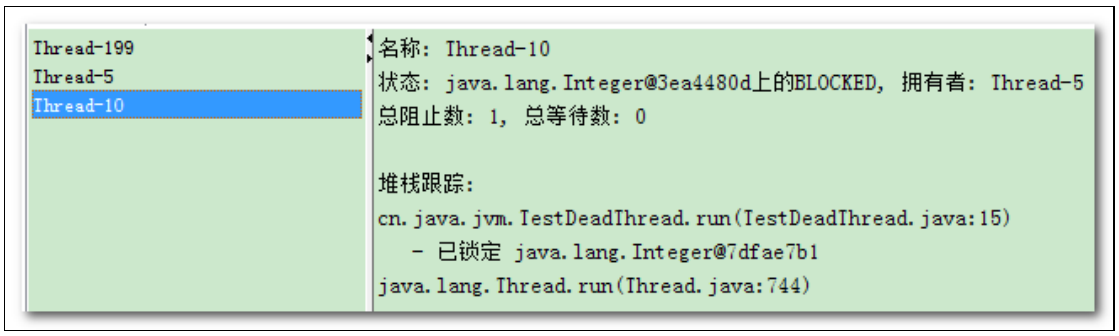
testBusyThread 线程：线程阻塞在 18 行的 while（true），直到线程切换，很耗性能

testLockThread 线程：出于 waitting 状态，等待 notify

死锁：

```
package cn.java.jvm;

public class TestDeadThread implements Runnable {
    int a, b;

    public TestDeadThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        System.out.println("createDeadThread");
        synchronized (Integer.valueOf(a)) {
            synchronized (Integer.valueOf(b)) {
                System.out.println(a + b);
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new Thread(new TestDeadThread(1, 2)).start();
            new Thread(new TestDeadThread(2, 1)).start();
        }
    }
}
```

点击检查死锁，会出现死锁的详情



thread-5 的锁被 thread-10 持有，相反亦是，造成死锁。