链滴

# 《深入理解 Java 虚拟机》读书笔记：垃圾收集器与内存分配策略

作者：jingqueyimu

原文链接：https://ld246.com/article/1575113857509

来源网站：链滴

# 正文

垃圾收集器关注的是 Java 堆和方法区，因为这部分内存的分配和回收是动态的。只有在程序处于运期间时才能知道会创建哪些对象，也才能知道需要多少内存。

虚拟机栈和本地方法栈则不需要过多考虑回收的问题，因为栈中每一个栈帧分配多少内存基本上是在结构确定下来时就已知的，因此这几个区域的内存分配和回收具有确定性。

# 一、对象已死吗

垃圾收集器在对堆进行回收前，第一件事就是要确定堆中对象哪些还“存活”着，哪些已“死去”（不可能再被任何途径使用的对象）。

## 1、 引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器值加 1；当引用失效时，计数器值减；任何时刻计数器为 0 的对象就是不可能再被使用的。

优点：实现简单，判定效率高。

缺点：很难解决对象之间相互循环引用的问题。

## 2、可达性分析算法

通过一系列被称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径称引用链，当一个对象到 GC Roots 没有任何引用链相连时，则此对象不可用。

Java 语言中，可作为 GC Roots 的对象：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用对象。
- 本地方法栈中 JNI（即 Native 方法）引用的对象。

可达性分析算法中不可达的对象，至少要经历两次标记过程，才会被回收。

1. 发现没有与 GC Roots 相连的引用链时，进行第一次标记。

2. 当对象覆盖了 finalize() 方法，并且没有被调用过时，将会被放入一个叫做 F-Queue 的队列中，后 GC 将对 F-Queue 中的对象进行第二次标记。如果在 finalize() 方法中，对象没有重新与引用链上一个对象建立关联，那么将会被回收。

## 3、四种引用

无论是引用计数算法，还是可达性分析算法，判断对象是否存活都与“引用”有关。Java 中有 4 种用，按强度由强至弱依次为：强引用、软引用、弱引用、虚引用。

- 强引用：类似“Object obj = new Object()”的引用。只要强引用还存在，对象就永远不会回收。
- 软引用：用来描述一些还有用但并非必需的对象。内存不足时，对象有可能被回收。可通过 SoftRef

rence 类实现软引用。

● 弱引用：用来描述非必需的对象，但强度比软引用弱。GC时，无论内存是否足够，对象都会被回收可通过 WeakReference 类来实现弱引用。

● 虚引用：也称幽灵引用或幻影引用，虚引用不会对对象的生存时间构成影响。虚引用的唯一作用就能在对象被回收时收到一个系统通知。可通过 PhantomReference 类实现虚引用。

## 4、回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。

**如何判定废弃常量：**

● 常量池中的常量（字面量、符号引用）没有在任何地方被引用。

**如何判定无用的类：**

● 该类的所有实例都已被回收。

● 加载该类的 ClassLoader 已被回收。

● 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方
。

# 二、垃圾收集算法

## 1、标记-清除算法

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，然后再统一回收所有被标记的
象。

该算法会产生大量不连续的内存碎片，因而在分配较大对象时，可能会由于无法找到足够的连续内存不得不提前触发一次 GC。

## 2、复制算法

将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当一块内存用完时，就将还存活的
象复制到另一块，然后再把已使用过的内存空间一次清理掉。

该算法的代价是始终会有一块内存被“浪费”掉。

由于新生代的对象 98% 是“朝生夕死”，因此并不需要按 1:1 的比例来划分内存空间。现在的商业拟机，是将内存划分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中
块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象复制到另一块 Survivor 上，最后清理
Eden 和使用过的 Survivor。

HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1。

**分配担保机制：**

当另一块 Survivor 没有足够空间来存放存活对象时，则需要其他内存（老年代）进行分配担保，将
象移入其他内存（老年代）。

## 3、标记-整理算法

首先标记出所有需要回收的对象，然后将所有存活对象向一端移动，最后直接清理掉端边界以外的内
。

## 4、分代收集算法

根据对象存活周期的不同，将 Java 堆划分为新生代和老年代，然后根据各个年代的特点采用最适当
收集算法。

● 新生代：采用复制算法。因为新生代每次 GC 都有大量对象死去，故只需付出少量存活对象的复制
本即可完成 GC。

● 老年代：采用“标记-清除”或“标记-整理”算法。因为老年代中对象存活率高，而且没有额外空
进行分配担保。

# 三、HotSpot 的算法实现

## 1、枚举根节点

可达性分析时，需要枚举 GC Roots 节点，以便标记出所有的不可用对象。

可作为 GC Roots 的节点主要在全局引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本
变量表）中。如果逐个检查里面的引用，会消耗很多时间。因此，目前主流的 Java 虚拟机使用准确式
GC 来完成 GC Roots 枚举。

**Stop The World（STW）：**

可达性分析期间，不可以出现对象引用关系还在不断变化的情况。因此 GC 时，必须停顿所有 Java
行线程，此时整个执行系统看起来就像被冻结某个时间点上。

**准确式 GC：**

虚拟机可以直接得知哪些地方存放着对象引用，因此 STW 时，不需要一个不漏地检查所有执行上下
和全局的引用位置。

**HotSpot 中准确式 GC 的实现：**

HotSpot 使用一组称为 OopMap 的数据结构来记录对象的引用位置。这样，GC 在扫描时就可以直
得知对象的引用位置信息。

类加载完成时，HotSpot 会把对象内什么偏移量上是什么类型的数据计算出来记录到 OopMap 中。J
T 编译过程中，也会在 OopMap 中记录下栈和寄存器中哪些位置是引用。

## 2、安全点

HotSpot 只在特定的位置上记录了 OopMap，这些位置称为安全点。

程序执行时，只有到达安全点才能停顿下来进行 GC。因为只有到达安全点，才能访问到 OopMap
录。

**如何在 GC 时让线程跑到最近的安全点再停顿下来：**

● 抢先式中断：GC 时，先中断所有线程，如果发现有线程中断的地方不在安全点上，就恢复线程，

它跑到安全点上。

● 主动式中断：GC 时，设置一个中断标志，各个线程执行时主动去轮询这个标志，发现中断标志为时就自己中断挂起。

## 3、安全区域

安全区域是指一段代码片段中，引用关系不会发生变化。在这个区域中的任意地方开始 GC 都是安全。可以把安全区域看做是被扩展了的安全点。

**为什么需要安全区域：**

当线程没有分配 CPU 时间时，将无法响应 JVM 的中断请求，跑到安全点中断挂起，JVM 也不太可等待线程重新被分配 CPU 时间。这种情况就需要安全区域来解决。

**安全区域的使用：**

1. 线程执行到安全区域的代码时，标识自己进入了安全区域。

2. JVM 发起 GC 时，不用管进入安全区域的线程。

3. 线程要离开安全区域时，必须检查系统是否完成了根节点枚举（或整个 GC 过程）。如果完成了，程就继续执行，否则必须等待，直到收到可以离开安全区域的信号。

# 四、垃圾收集器

## 1、Serial 收集器

● 最基本、历史最悠久的收集器。

● 单线程收集器：使用一个 CPU 或一条线程进行垃圾收集。

● 新生代收集器，是运行在 Client 模式下的虚拟机的默认新生代收集器。

● 简单而高效，单个 CPU 下，没有线程交互的开销。

## 2、ParNew 收集器

● Serial 收集器的多线程版本。

● 新生代收集器，是许多运行在 Server 模式下的虚拟机中首选的新生代收集器。

● 除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。

● 默认开启的收集线程数与 CPU 数量相同。

## 3、Parallel Scavenge 收集器

● 多线程收集器。

● 新生代收集器。

● 关注吞吐量，即 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值。高吞吐量可以高效利用 PU 时间，尽快完成程序的运算任务，适合于在后台运算而不需要太多交互的任务。

● 可开启自适应调节策略，把内存管理的调优任务交给虚拟机去完成。

**自适应调节策略：**

虚拟机根据当前系统的运行情况收集性能监控信息，动态调整虚拟机参数以提供最合适的停顿时间或大的吞吐量。

# 4、Serial Old 收集器

- Serial 收集器的老年代版本。
- 单线程收集器。
- 使用“标记-整理”算法。
- 给 Client 模式下的虚拟机使用。

# 5、Parallel Old 收集器

- Parallel Scavenge 收集器的老年代版本。
- 多线程收集器。
- 使用“标记-整理”算法。
- 在注重吞吐量以及 CPU 资源敏感的场合，可优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

# 6、CMS 收集器

- CMS：Concurrent Mark Sweep。
- 并发收集器：垃圾收集线程与用户线程（基本上）同时工作。
- 使用“标记-清除”算法。
- 关注点是如何缩短垃圾收集时用户线程的停顿时间。停顿时间短意味着响应速度快，因此它适合于要与用户交互的应用。

**CMS 运作过程：**

1. 初始标记：标记 GC Roots 能直接关联到的对象，需要 STW。
2. 并发标记：进行 GC Roots Tracing 的过程，即可达性分析。
3. 重新标记：修正并发标记期间引用关系发生变化的那一部分对象的标记记录，需要 STW。
4. 并发清除：清除垃圾对象。

**CMS 的缺点：**

- 对 CPU 资源非常敏感。并发阶段虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或说 CPU 资源）导致应用程序变慢，总吞吐量会降低。
- 无法处理浮动垃圾。并发清除阶段产生的垃圾称为“浮动垃圾”，这部分垃圾只能等下次 GC 再清。
- 会产生大量内存碎片。内存碎片过多时会提前触发 Full GC，CMS 收集器默认会在 Full GC 时开启存碎片的合并整理过程。

# 7、G1 收集器

- G1：Garbage-First。
- 是一款面向服务端应用的垃圾收集器。

**G1 特点：**

- 并行与并发
- 分代收集
- 空间整合：G1 从整体上看是基于“标记-整理”算法，从局部上（两个 Region 之间）看是基于复算法。因此，不会产生内存空间碎片。
- 可预测的停顿：G1 能通过建立可预测的停顿时间模型，让使用者明确指定在 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。

**Region：**

G1 将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留新生代和老年代的概念但新生代和老年代不再是物理隔离的，而是一部分 Region（不需要连续）的集合。

**可预测的时间停顿模型：**

G1 之所以能建立可预测的时间停顿模型，是因为它可以有计划地避免在整个 Java 堆中进行全区域的圾收集。

G1 跟踪各个 Region 的垃圾堆积的价值大小（回收所获得的空间大小及所需时间），在后台维护一优先列表，每次根据允许的收集时间，优先回收价值最大的 Region（Garbage-First 名称的由来）。

**G1 运作过程：**

1. 初始标记：标记 GC Roots 能直接关联的对象，并修改 TAMS（Next Top at Mark Start）的值，下一阶段用户程序并发运行时，能在正确可用的 Region 中创建新对象。需要 STW。
2. 并发标记：进行可达性分析。
3. 最终标记：修正并发标记期间引用关系发生变化的那一部分对象的标记记录。需要 STW。
4. 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 停顿时间制定回收划。

# 五、内存分配与回收策略

## 1、对象优先在 Eden 分配

- 大多数情况下，对象在新生代的 Eden 区中分配。
- 当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。

## 2、大对象直接进入老年代

- 大对象是指需要大量连续内存空间的 Java 对象。
- 经常出现大对象容易导致内存还有不少空间时，就提前触发 GC 以获取足够的连续空间来安置它们。
- 由于新生代采用复制算法收集内存，因此为了避免在 Eden 区及两个 Survivor 区之间发生大量的存复制，大对象将直接进入老年代。

## 3、 长期存活的对象进入老年代

● 虚拟机给每个对象定义了一个对象年龄计数器。

● 对象在 Eden 出生并经过一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将移入 Survivor 中，并且对象年龄设为 1。

● 对象在 Survivor 中每“熬过”一次 Minor GC，则年龄加 1，当对象年龄增加到一定程度（默认 15 岁），将会晋升到老年代。

## 4、动态对象年龄判定

● 为了更好地适应不同程序的内存状况，虚拟机并不要求对象必须达到某个年龄才能晋升老年代。

● 如果 Survivor 中相同年龄的对象大小总和，大于 Survivor 空间的一半，则大于等于该年龄的对象接进入老年代。

## 5、空间分配担保

● 当出现大量对象在 Minor GC 后仍然存活的情况，就需要老年代进行分配担保，让 Survivor 无法纳的对象直接进入老年代。