垃圾回收机制的原理与实现

黄梓淳

Nov 16, 2025

在计算机编程的早期阶段，内存管理完全依赖于开发者的手动操作。以 C 和 C++ 为例，程序员必须显式调用 `malloc` 和 `free` 或 `new` 和 `delete` 来分配和释放内存。这种手动方式虽然提供了极高的灵活性，但也带来了诸多挑战。例如，内存泄漏是指分配的内存未被及时释放，导致系统资源逐渐耗尽；悬空指针是访问已释放内存的指针，可能引发程序崩溃；双重释放则是多次释放同一块内存，会造成未定义行为。这些问题不仅调试困难，还严重影响了程序的稳定性和安全性。

垃圾回收机制的引入，正是为了应对这些挑战。它自动识别并回收程序中不再使用的内存对象，从而将开发者从繁琐的内存管理中解放出来。垃圾回收的核心价值在于提升开发效率、增强程序健壮性，并显著减少内存相关错误。本文将带领读者从基本概念出发，逐步深入主流垃圾回收算法的核心原理与实现细节，并探讨现代高级垃圾回收技术，帮助读者全面理解自动内存管理的内部机制。

1　基础篇：垃圾回收的「世界观」

垃圾回收的核心问题在于如何定义「垃圾」。简单来说，垃圾是程序中无法再被访问到的对象。为了准确识别这些对象，垃圾回收器依赖于可达性分析算法。该算法以一系列称为「GC Roots」的根对象作为起点，根据对象之间的引用关系遍历整个对象图。能够被遍历到的对象被视为存活，而其余对象则被判定为垃圾。GC Roots 通常包括虚拟机栈中局部变量引用的对象、方法区中静态属性引用的对象、方法区中常量引用的对象、本地方法栈中 JNI 引用的对象、虚拟机内部引用以及被同步锁持有的对象。与引用计数法相比，可达性分析能够有效解决循环引用问题，因为它从根对象出发，忽略无法到达的孤立环。

在垃圾回收过程中，一个关键挑战是如何在可达性分析时确保对象引用关系的稳定性。为此，虚拟机必须暂停应用程序的执行，即发生「Stop-The-World」事件。安全点是程序执行中的一些特定位置，如方法调用、循环跳转或异常跳转，在这些点上虚拟机的状态是确定的，可以安全地开始垃圾回收。安全区域则是一段代码片段，在该区域内引用关系不会发生变化，因此从任意点开始垃圾回收都是安全的。这些机制保证了垃圾回收的准确性和一致性。

2　经典 GC 算法原理与实现剖析

标记-清除算法是垃圾回收中最基础的算法。其过程分为两个阶段：首先通过可达性分析标记所有存活对象，然后遍历整个堆内存，回收未被标记的对象所占用的空间。这种算法实现简单，是许多后续算法的基础。然而，它的效率不稳定，堆内存越大，标记和清除过程就越慢，并且会产生内存碎片问题。碎片化是指回收后内存空间变得不连续，导致即使总空闲内存足够，也无法分配大对象。

标记-复制算法通过将可用内存分为两个相等的部分（例如 From Space 和 To Space）来工作。每次只使用其

中一部分，当该部分内存用尽时，将存活对象复制到另一部分，并清理已使用的整块内存。这种算法运行高效且没有内存碎片，但内存利用率只有 50%，并且当存活对象较多时，复制开销会显著增加。它特别适合处理「朝生夕死」的年轻代对象，因为这些对象存活率低，复制成本较小。

标记-整理算法在标记存活对象后，将所有对象向内存空间的一端移动，然后直接清理掉边界以外的内存。这种算法消除了内存碎片，并且内存利用率达到 100%，但移动存活对象并更新所有引用地址的开销较大，通常会导致更长的 Stop-The-World 时间。因此，它常用于存活对象较多的老年代，其中对象生命周期长，移动频率较低。

3 现代垃圾回收器的核心思想：分代收集理论

分代收集理论基于两个关键假说：弱分代假说指出绝大多数对象都是朝生夕死的；强分代假说则认为熬过越多次垃圾收集的对象就越难以消亡。基于这些假说，堆内存被划分为年轻代和老年代。年轻代进一步分为 Eden 区和两个 Survivor 区（From 和 To），新创建的对象首先分配在 Eden 区。老年代则存放从年轻代晋升而来的长时间存活对象。

分代收集过程包括 Minor GC 和 Major GC。Minor GC 在 Eden 区满时触发，使用标记-复制算法将 Eden 和 From Survivor 中存活的对象复制到 To Survivor。对象每存活一次，年龄就增加一，当年龄超过阈值（通常为 15）时，晋升到老年代。Major GC 或 Full GC 则在对整个堆进行回收时发生，通常由老年代满、空间分配担保失败或显式调用 `System.gc()` 触发。这个过程常使用标记-整理或更复杂的算法，Stop-The-World 时间较长，对应用程序响应速度影响显著。

4 前沿与实战：主流垃圾回收器探秘

在垃圾回收器的发展中，出现了以吞吐量优先和低延迟优先的两大流派。Parallel Scavenge 和 Parallel Old 是吞吐量优先的代表，作为 JDK8 的默认组合，它们通过多线程并行执行垃圾回收来达到可控制的吞吐量。吞吐量定义为用户代码运行时间与总时间（用户代码运行时间加 GC 时间）的比值，即 $\text{吞吐量} = \frac{\text{用户代码运行时间}}{\text{用户代码运行时间}+\text{GC 时间}}$。这种组合适合后台运算和批处理任务，其中高吞吐量比低延迟更重要。

CMS 是第一款并发收集器，以最短回收停顿时间为目标。其过程包括四个阶段：初始标记、并发标记、重新标记和并发清除。初始标记和重新标记需要 Stop-The-World，而并发标记和清除则与应用程序线程同时运行。CMS 的优点在于低停顿，但对 CPU 资源敏感，无法处理「浮动垃圾」，并且会产生内存碎片。G1 垃圾回收器则是一个里程碑式的创新，它将堆划分为多个大小相等的独立区域（Region），不再是物理分代，而是逻辑分代。G1 的核心思想是建立可预测的停顿时间模型，通过跟踪各个 Region 的回收价值（回收所需空间与回收所得空间的经验值），优先回收价值最大的 Region。其过程包括初始标记、并发标记、最终标记和筛选回收。

ZGC 和 Shenandoah 是下一代超低延迟垃圾回收器，目标是将 Stop-The-World 时间控制在 10 毫秒以内，无论堆内存多大。它们采用革命性技术如读屏障、染色指针和并发整理来实现这一目标。例如，ZGC 使用染色指针在指针中存储元数据，从而允许并发执行大部分回收操作，极大减少了停顿时间。

5 理解 GC 对编码的指导意义

垃圾回收机制对编程实践有重要指导意义。首先，对象分配应优先在年轻代的 Eden 区进行，大对象可能直接进入老年代以避免频繁复制。为了减少垃圾回收的压力和 Stop-The-World 时间，开发者应避免创建过多不必要

的对象。例如，在循环内创建对象或使用字符串拼接操作符可能导致大量临时对象。相反，使用 StringBuilder 可以更高效地处理字符串拼接。

以下是一个代码示例对比：

```
// 低效方式：每次循环创建新 String 对象
String result = "";
for (int i = 0; i < 1000; i++) {
   result += i; // 这会导致多次对象分配
}

// 高效方式：使用 StringBuilder 减少对象创建
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
   sb.append(i);
}
String result = sb.toString();
```

在低效方式中，每次循环迭代都会创建一个新的 String 对象，因为字符串是不可变的，这会导致大量临时对象产生，增加垃圾回收负担。而在高效方式中，StringBuilder 在内部维护一个可变的字符数组，仅在必要时扩展，从而显著减少对象分配次数。此外，开发者应谨慎使用全局集合类，及时清理无用的引用，并避免随意调用 System.gc()，因为它可能触发不必要的 Full GC。根据应用特性（如吞吐量优先或低延迟优先），合理选择和调优垃圾回收器也是优化性能的关键。

垃圾回收技术的发展是一个不断在吞吐量、延迟和内存开销之间寻求平衡的艺术。从手动管理到自动管理，从标记-清除到分代收集，再到 G1 和 ZGC，每一次进步都解决了前一代的局限性。未来，垃圾回收将向着更低延迟、更大堆内存和更智能化的方向发展。例如，硬件创新如 NVMe SSD 正在改变垃圾回收的设计思路，允许更高效的数据处理。总之，理解垃圾回收机制不仅有助于编写高效代码，还能为应对未来技术挑战奠定基础。

6 参考资料与延伸阅读

本文内容参考了《深入理解 Java 虚拟机》、Oracle 官方垃圾回收调优指南以及相关学术论文如 G1 和 ZGC 的原始论文。读者可进一步阅读这些资料以深入了解垃圾回收的细节和最新进展。