Java 垃圾回收机制详解

王思成

Dec 01, 2025

Java 垃圾回收机制是 Java 虚拟机（JVM）内存管理的最核心特性之一，它自动识别并回收不再使用的对象内存，避免了开发者手动管理内存的复杂性和错误风险。在现代 Java 应用中，尤其是在高并发、大规模分布式系统中，GC 的性能直接影响系统的吞吐量、延迟和稳定性。本文将从 JVM 内存结构入手，深入剖析垃圾判定机制、回收算法、分代策略、主流收集器实现，以及参数调优实战，帮助读者系统掌握 Java GC 的全貌。

为什么需要垃圾回收呢？在像 C/C++ 这样的语言中，开发者必须手动分配内存（如使用 malloc 或 new）和释放内存（如 free 或 delete），稍有不慎就会导致内存泄漏或悬垂指针问题。Java 通过 GC 自动处理这些事务，大大提高了开发效率和代码可靠性，但也引入了 GC 暂停（Stop-The-World，简称 STW）等性能开销。GC 的核心目标不仅是回收无用对象，还需在内存回收效率与应用暂停时间之间取得平衡，实现高吞吐量和低延迟的双重优化。

本文结构清晰，先介绍 JVM 内存基础，再探讨垃圾判定与算法，然后深入分代回收和收集器实现，最后聚焦调优实战和日志分析。适合有 Java 基础和 JVM 概述知识的开发者阅读，无论你是初学者想理解 GC 原理，还是架构师需要优化生产环境，都能从中获益。

1　2. JVM 内存结构基础

JVM 运行时数据区是理解 GC 的基石，它分为线程私有和线程共享区域。线程私有区域包括程序计数器（PC Register），用于记录当前线程执行的字节码指令地址；虚拟机栈，存储局部变量表、操作数栈等；本地方法栈，则服务于 Native 方法调用。这些区域随线程生命周期而生灭，不涉及 GC。

线程共享区域则包括方法区（JDK 8 前为永久代，之后演变为 Metaspace，用于存储类元数据、常量池等）和堆（Heap），后者是 GC 的主要战场。堆内存按对象生命周期分代设计，新生代（Young Generation）存放短生命周期对象，老年代（Old Generation）存放长生命周期对象，而 Metaspace 虽不直接回收对象，但其溢出会间接触发 Full GC。

堆内存的核心是新生代，由 Eden 区和两个 Survivor 区（S0 和 S1）组成。新对象优先分配在 Eden，当 Eden 满时触发 Minor GC，存活对象复制到 Survivor，经历多次 GC 后年龄达到阈值（如 15）则晋升老年代。这种分代设计基于「大多数对象朝生夕死」的经验假设，大幅提升了 GC 效率。

2　3. 垃圾判定机制

垃圾对象是指应用不再可达的对象，即没有有效引用指向它。Java 通过引用类型和可达性分析来判定垃圾。从 JDK 8 开始，引用分为强引用、软引用、弱引用和虚引用。强引用是最常见的，如 Object obj = new Object()，只要强引用存在，GC 绝不回收。软引用在内存不足时才回收，常用于图片缓存；弱引用在 GC 时无

条件回收，适合内存敏感的缓存场景；虚引用最弱，仅用于监控对象销毁，无法通过它访问对象，是 finalizer 的现代替代。

垃圾判定主要依赖可达性分析算法，从 GC Roots 出发遍历对象图。GC Roots 包括虚拟机栈中的局部变量、方法区中的静态变量、常量、JNI 句柄等。以一个简单示例说明：

```
public class GCRootsDemo {
   public static Object root = new Object(); // GC Root: 静态变量
   public void method() {
      Object local = new Object(); // GC Root: 局部变量
      root = local; // local 通过 root 可达
   }
}
```

在这里，root 是静态变量，故为 GC Root；method 中的 local 是栈帧局部变量，也为 GC Root。通过 root 引用，local 对象可达，不会回收。如果移除 root = local，则 local 在方法返回后不可达，成为垃圾。引用计数算法虽简单（每个对象计数器加减），但无法处理循环引用，故 HotSpot JVM 弃用它，转用可达性分析。

3 4. 垃圾回收算法详解

GC 过程分为标记（Mark）和清除/整理（Sweep/Compact）阶段。标记阶段从 GC Roots 遍历，标记存活对象。标记-清除算法最简单，先标记再清除未标记对象，但会产生内存碎片，导致分配大对象时失败。标记-整理则在清除后移动存活对象，消除碎片，但需要额外整理时间，STW 更长。标记-复制适用于对象存活率低的场景，将内存分为两块，复制存活对象到另一块后清空原块，简单高效但空间利用率仅 50%。

新生代常用标记-复制，老年代偏好标记-整理。并发时代引入三色标记算法：对象分为白（未标记）、灰（标记待扫描）和黑（已标记完成）。并发标记时，用户线程可能移动对象，导致「漏标」问题。为此，CMS 使用增量更新，G1 采用 Snapshot-At-The-Beginning（SATB）屏障，确保正确性。这些算法在 $\mathcal{O}(n)$ 时间复杂度内完成标记，其中 n 为对象数。

4 5. 分代回收策略与 GC 类型

分代回收基于两个假设：弱分代假设（大多数对象短命）和强分代假设（长命对象更长命）。新生代 GC 称为 Minor GC：Eden 满时，复制存活对象到 S0（S1 空闲时），交换 S0/S1 角色；对象年龄（GC 次数）累加，超阈值晋升老年代。动态年龄判定允许 Survivor 满时批量晋升同龄对象。

老年代 GC 为 Major 或 Full GC，触发于老年代满、空间分配担保失败或显式 System.gc()。Full GC 回收整个堆和方法区，STW 时间长，最该避免。Minor GC 频率高但暂停短（毫秒级），Major GC 中等频率，Full GC 罕见但代价大。

5 6. Java 垃圾收集器详解（HotSpot JVM）

HotSpot JVM 提供多种收集器，按并行/并发分代组合使用。Serial 是单线程收集器，适合客户端模式，新生代用标记-复制，老年代用标记-整理。ParNew 是其多线程版，常与 CMS 搭配。Parallel（PS）注重吞吐，并

行标记-复制新生代和标记-整理老年代。
CMS（Concurrent Mark Sweep）追求低延迟，并发标记和清除，仅初始标记和重新标记 STW，但易碎片化。G1（Garbage First）将堆分成 Region（2MB），优先回收垃圾最多的 Region，支持分代，低延迟是 JDK 9+ 默认。ZGC 和 Shenandoah 是超低延迟收集器，使用着色指针（Colored Pointers）实现并发整理，STW 亚毫秒级，适用于多 TB 大堆。
典型组合中，高吞吐场景选 Parallel + Parallel Old，低延迟用 G1 或 ParNew + CMS。以 G1 为例，其 RSet（Remembered Set）记录跨 Region 引用，Humongous 对象（>50% Region）单独处理，避免碎片。

6 7. GC 参数调优实战

调优从设置堆大小开始：-Xms 和 -Xmx 设为相同值避免动态调整，-Xmn 指定新生代大小。新生代与老年代比例用 -XX:NewRatio=2（1:2），Survivor 用 -XX:SurvivorRatio=8（Eden:Survivor=8:1）。G1 参数如 -XX:+UseG1GC -XX:MaxGCPauseMillis=200 控制目标暂停。
调优步骤：先用 jstat 监控 GC 频率，用 VisualVM 或 JFR 分析堆转储。常见问题如 Full GC 频繁，可增大老年代或调低晋升阈值。以电商高并发场景为例，初始 Full GC 占比 20%，调优后 -XX:NewRatio=1 -XX:MaxTenuringThreshold=10，Full GC 降至 1%，吞吐提升 30%。原则是遵循分代、避免 Full GC、监控先行。
考虑这段调优代码示例：

```
// JVM 参数：-Xms4g -Xmx4g -Xmn1g -XX:SurvivorRatio=8 -XX:+UseG1GC -XX:
    ↪ MaxGCPauseMillis=100
public class TuningDemo {
   public static void main(String[] args) {
      List<byte[]> list = new ArrayList<>();
      for (int i = 0; i < 100000; i++) {
         list.add(new byte[1024 * 100]); // 分配 100KB 对象，模拟短命对象
      }
      // 预期：频繁 Minor GC，大对象少量晋升
   }
}
```

这段代码分配大量小对象，Eden 快速满触发 Minor GC，G1 高效复制到 Survivor。参数确保新生代占 25%，暂停控制在 100ms 内。若无调优，Full GC 会因老年代压力而频发。

7 8. GC 日志分析

开启日志用 -Xlog:gc*:file=gc.log:time,uptime,level,tags。JDK 9+ 格式如 [0.123s][info][gc] GC(0) Pause Young (Normal)（allocation failure），解析显示时间、类型、原因。关键指标包括 GC 时间占比（理想 <5%）、吞吐量（>90%）和晋升率（高则增大新生代）。
用 GCViewer 加载日志，观察曲线：若 Full GC 峰值多，检查晋升失败；STW 陡峭，考虑 G1/ZGC。生产中集成 ELK 栈，实现告警。

8 9. 高级主题与未来趋势

内存泄漏常因集合未清空或线程池泄露，用 jmap -histo 排查。NUMA 优化和大页（HugePage）减少 TLB miss，提升 10-20% 性能。JDK 17+ 的 Epsilon 无 GC，适合短命任务；ZGC 生产就绪，云原生中容器感知 GC（如 -XX:+UseContainerSupport）自适应 CPU/Mem 限制，GraalVM Native Image 则 AOT 编译避开 GC。

GC 从判定到收集，核心是平衡回收与暂停。生产实践：预留 25% 堆空间，设置 Full GC 告警，用 JFR 定期诊断。推荐书籍《深入理解 Java 虚拟机》和《Java 性能权威指南》，工具 JMH 测试吞吐、JFR 火焰图分析。

常见疑问：G1 何时优于 CMS？答：堆 >4GB 或追求可预测延迟时。欢迎评论区分享你的 GC 调优经验！

9 附录

A. 示例代码：引用类型演示

```
public class ReferenceDemo {
    public static void main(String[] args) {
        // 软引用示例
        SoftReference<byte[]> softRef = new SoftReference<>(new byte[1024 * 1024 *
            ↪ 100]); // 100MB
        System.out.println(softRef.get() != null ? "软引用存活" : "已回收");
        // 施压内存，触发 OOM 前软引用回收
        List<byte[]> pressure = new ArrayList<>();
        for (int i = 0; i < 10; i++) pressure.add(new byte[1024 * 1024 * 100]);
    }
}
```

这段代码创建 100MB 软引用对象，循环分配更多大数组模拟 OOM 前压力。SoftReference 在内存紧张时 get() 返回 null，证明 GC 回收，避免崩溃。解读：软引用持有器不阻止回收，仅在空间不足时响应，get() 可能返回 null，需检查；适用于缓存，如浏览器图片池。

B. 参考文献

《深入理解 Java 虚拟机》（周志明）、Oracle JDK 文档、OpenJDK GC 源码。