垃圾回收（Garbage Collection）

杨岢瑞

Aug 13, 2026

1　为什么要关心垃圾回收

在现代应用程序运行过程中，内存管理始终是性能优化的核心环节之一。开发者经常遇到线上环境因堆内存耗尽而引发的 OOM 错误，或者因垃圾回收停顿时间过长导致请求响应时间出现明显波动。对于延迟敏感的在线交易系统，GC 停顿可能直接影响用户体验，而对于吞吐量优先的批处理任务，GC 策略的选择则更侧重于整体执行效率的提升。本文面向具备 C、C++、Java 或 Go 等语言基础的读者，系统梳理垃圾回收的理论模型与工业级实现。

2　垃圾回收的三要素

垃圾回收器的核心任务可以概括为三个相互关联的环节：识别无用对象、执行回收策略以及管理内存碎片。识别无用对象主要依赖可达性分析或引用计数两种方法。可达性分析从一组根对象出发，沿着引用链遍历所有可到达的对象，未被标记的对象即被判定为垃圾。引用计数则为每个对象维护一个计数器，当计数器归零时立即回收。回收策略决定了如何组织空闲内存与存活对象。标记-清除算法首先遍历堆空间标记存活对象，随后一次性释放未标记区域。标记-压缩算法在标记阶段之后，将存活对象向堆的一端移动，消除内存碎片。复制收集则将堆划分为两个半区，每次仅使用其中一个，回收时将存活对象复制到另一个半区。

内存碎片整理需要与分配器紧密协作。传统 malloc/free 由程序员显式管理，GC 则在运行时自动完成分配与回收的协调。分配器通常采用空闲链表或位图方式记录可分配区域，而 GC 则需要在回收过程中更新这些元数据结构。

3　标记-清除算法

标记-清除是最基础的追踪式垃圾回收算法。算法分为标记和清除两个阶段。标记阶段从根集合出发，递归遍历所有可达对象并打上标记。清除阶段则线性扫描整个堆，将未标记的对象所占空间加入空闲链表。

三色标记是标记阶段的抽象模型。白色表示未访问对象，灰色表示已访问但子对象尚未处理，黑色表示所有子对象均已处理。写屏障用于维护三色不变性，即黑色对象不能直接引用白色对象。当赋值操作发生时，写屏障会将目标对象标记为灰色，确保其后续被正确处理。

4 标记-压缩算法

标记-压缩算法在标记阶段之后增加了一个整理阶段。整理阶段将所有存活对象向堆的一端移动，空闲空间自然形成连续区域。移动过程中需要更新所有指向被移动对象的引用，这通常通过一个转发指针或全局引用表实现。该算法解决了标记-清除产生的内存碎片问题，但移动对象需要额外的 CPU 周期。对于大对象或引用关系复杂的对象图，移动成本可能抵消碎片整理带来的收益。现代垃圾回收器通常结合分代假设，仅对老年代使用标记-压缩以平衡开销。

5 复制收集算法

复制收集将堆空间划分为大小相等的两个半区，分别称为 From 空间和 To 空间。程序运行时仅使用 From 空间进行对象分配。当 From 空间耗尽时，垃圾回收器将所有存活对象复制到 To 空间，随后交换两个半区的角色。复制过程采用广度优先或深度优先遍历，依次将可达对象复制到 To 空间的新位置。复制完成后，From 空间中的所有对象均被视为垃圾，整个空间可一次性释放。该算法天然避免了内存碎片，但要求堆空间至少翻倍，且每次 GC 都需要复制所有存活对象。

6 分代收集

分代收集基于弱分代假说，即大多数对象在分配后很快死亡，只有少数对象能存活较长时间。堆被划分为年轻代和老年代，年轻代采用复制收集快速回收短生命周期对象，老年代则使用标记-压缩或标记-清除处理长生命周期对象。

对象在年轻代经历多次 GC 后仍存活，会被晋升到老年代。晋升阈值通常由对象年龄或存活次数决定。跨代引用通过记忆集或卡表记录，避免每次 GC 都扫描整个老年代。

7 增量与并发标记

增量标记将标记阶段拆分为多个小步，与应用线程交替执行。并发标记则允许标记线程与应用线程同时运行。这两种方式都面临三色不变性的挑战，需要通过写屏障或读屏障确保标记的正确性。

SATB（Snapshot At The Beginning）在标记开始时记录对象图快照，后续赋值操作产生的引用变化不会影响本次标记结果。增量更新则在赋值时将新引用对象标记为灰色，确保其在后续标记中被处理。

8 引用计数及其变种

引用计数通过为每个对象维护引用数量实现即时回收。当引用计数归零时，对象立即释放，无需等待全局 GC 周期。循环引用是引用计数无法处理的场景，需要结合可达性分析或弱引用机制解决。

惰性释放延迟实际内存释放操作，将待释放对象放入一个队列，由后台线程批量处理。Rust 采用编译时所有权检查，在编译阶段确定对象生命周期，完全避免运行时垃圾回收开销。

9 HotSpot JVM 的 GC 实现

CMS 垃圾回收器针对老年代设计，采用并发标记和增量更新策略。标记阶段与应用线程并发执行，仅在初始标记和重新标记阶段需要短暂停顿。重新标记阶段采用并行标记加速，减少停顿时间。

G1 将堆划分为多个大小相等的 Region，每个 Region 可独立进行垃圾回收。SATB 写屏障记录跨 Region 引用变化，混合收集阶段同时处理年轻代和老年代 Region。预测停顿模型根据历史数据估算每次收集的 Region 数量，满足用户指定的停顿时间目标。

ZGC 使用彩色指针在指针中嵌入标记信息，通过加载屏障在读操作时触发对象重定位。重定位与应用线程并发执行，实现了亚毫秒级的停顿时间。Shenandoah 采用 Brooks 指针实现并发整理，同样追求极低的延迟指标。

10 Go 运行时的 GC 策略

Go 运行时采用并发标记和非移动式回收策略。GC 触发条件包括堆内存增长达到阈值以及定期两分钟强制回收。P 级并发标记允许每个 P 同时执行标记任务，辅助 GC 机制让应用线程在分配对象时协助完成标记工作。

逃逸分析在编译阶段确定对象是否需要分配在堆上，栈上分配的对象无需 GC 管理。GOGC 参数控制堆增长比例，pacer 算法动态调整标记速度与分配速度的匹配关系。debug.gctrace 环境变量可输出详细的 GC 统计信息。

11 不同语言运行时的 GC 差异

Java HotSpot 提供多种 GC 选择器，开发者可根据应用特征选择 CMS、G1 或 ZGC。Go 运行时 GC 设计简洁，强调低延迟和简单调优。NET CLR 采用与 Java 相似的分代模型，但具体实现细节存在差异。

V8 引擎为 JavaScript 设计了并行与增量 GC 策略，适应浏览器环境的实时性要求。CPython 使用引用计数结合循环检测，内存管理相对简单但存在全局解释器锁限制。移动端 ART 针对电池续航优化 GC 策略，Hermes 则为 React Native 提供专门的 GC 实现。

12 性能度量与调优

关键性能指标包括吞吐率、平均与最大停顿时间、对象分配速率以及堆内存使用曲线。吞吐率衡量应用线程实际执行时间占比，停顿时间直接影响用户体验。分配速率过高可能导致 GC 频率上升，堆内存曲线反映对象生命周期分布特征。

VisualVM 和 JFR 提供图形化 GC 日志分析，async-profiler 可采样 GC 相关系统调用。Go 工具链中的 pprof 和 trace 支持堆分析和 GC 事件追踪。典型问题包括过早晋升导致老年代碎片、内存泄漏引发堆持续增长，以及分配风暴造成 GC 频繁触发。

13 前沿研究方向

C4 收集器通过压缩写屏障实现全并发压缩，LISP2 算法探索极致延迟场景下的对象移动优化。分代 ZGC 在保持低延迟的同时引入分代假设，进一步提升吞吐表现。Epsilon GC 作为实验性收集器完全跳过垃圾回收，适用于已知堆内存上限的场景。

Rust 通过所有权系统在编译期完成内存管理，避免运行时 GC 开销，但要求开发者适应新的编程范式。硬件层面，CXL 内存池和内存压缩加速技术为 GC 提供了新的优化空间。Serverless 环境对亚毫秒级 GC 提出了更高要求，推动了增量式和部分堆收集等新技术发展。

14 决策与持续优化

选择 GC 策略需要综合考虑延迟要求、吞吐目标、内存上限以及 JDK 版本兼容性。持续观测堆内存曲线和 GC 日志，结合自动化压测流水线验证调优效果，是维持系统稳定运行的有效手段。深入理解 GC 原理有助于开发者编写更友好的代码，减少不必要的内存分配和对象生命周期延长。