# 基本的垃圾回收（Garbage Collection）机制

李睿远

Oct 15, 2025

告别内存泄漏，亲手构建你的第一个微型垃圾收集器。

在软件开发中，内存管理一直是一个核心且复杂的问题。想象一下，你编写了一个 C 语言程序，动态分配了内存却忘记释放，就像打开水龙头后没有关闭一样，内存使用量会不断攀升，最终导致程序崩溃。例如，一段简单的 C 代码中，如果重复调用 `malloc` 而不调用 `free`，内存泄漏会逐渐累积，系统资源被耗尽。相比之下，像 Java 或 Python 这样的语言内置了垃圾回收机制，开发者无需手动管理内存，从而大大提升了开发效率。垃圾回收的核心价值在于自动化内存管理，它让程序员能专注于业务逻辑，而不是底层细节。同时，它还能避免悬挂指针、重复释放等经典内存错误，增强程序的健壮性。本文的目标是引导读者从理论出发，理解垃圾回收的基本概念和经典算法，并最终通过 Python 实现一个简化的标记-清除垃圾回收器，从而在实践中深化理解。

## 1　垃圾回收的基石：概念与术语

在垃圾回收领域，「垃圾」指的是程序中无法再被访问到的对象所占用的内存。判断垃圾的唯一标准是「可达性」，即一个对象是否还能通过引用链被访问到。根对象是垃圾收集器遍历的起点，常见的根对象类型包括全局变量、当前执行函数的栈上的变量以及寄存器中的引用。这些根对象构成了对象图的入口点。对象图是一个有向图，其中节点代表内存中的对象，边代表对象之间的引用关系。活动对象是指从根对象出发，通过引用链可以访问到的所有对象，而垃圾对象则是从任何根对象都无法到达的对象。理解这些基本概念是掌握垃圾回收机制的前提。

## 2　经典垃圾回收算法巡礼

引用计数法是一种直观的垃圾回收算法，其核心思想是为每个对象维护一个引用计数器。每当有新的引用指向该对象时，计数器加一；当引用消失时，计数器减一。一旦计数器归零，对象便立即被回收。这种方法的优点在于简单和实时性高，但存在致命的循环引用问题。例如，如果两个对象互相引用，它们的计数器永远不会归零，导致内存泄漏。此外，频繁更新计数器也会带来性能开销。

标记-清除法通过两个阶段来解决循环引用问题。在标记阶段，垃圾收集器从所有根对象开始，递归地遍历对象图，为所有可达的对象打上活动标记。在清除阶段，收集器遍历整个堆，回收未被标记的对象，并将它们的内存加入空闲链表以备后续分配。这种方法虽然能有效处理循环引用，但缺点包括「停止世界」现象，即标记和清除过程中应用程序需要暂停，以及内存碎片化问题，可能导致后续无法分配大对象。

标记-整理法在标记-清除的基础上增加了整理阶段，以解决内存碎片化。标记阶段与标记-清除法相同，随后收集器将所有活动对象向内存的一端移动，紧凑排列，并更新所有指向被移动对象的引用。这样消除了碎片，但移动对象和更新引用的开销较大。

复制算法将堆内存分为两个相等部分：From 空间和 To 空间。新对象只分配在 From 空间，当 From 空间满时，

程序暂停，所有活动对象被复制到 To 空间并紧凑排列，然后交换 From 和 To 空间的角色。这种方法的优点是分配速度快且无碎片，但内存利用率只有 50%。

分代收集法基于「弱代假说」，即大多数对象生命周期短，只有少数对象能长期存活。它将堆划分为不同代，如年轻代和老年代。新对象分配在年轻代，年轻代垃圾回收频繁且快速，通常使用复制算法。存活多次回收的对象晋升到老年代，老年代回收不频繁但耗时较长，常用标记-清除或标记-整理法。这种方法平衡了性能和内存使用。

## 3 动手实践：用 Python 实现一个微型标记-清除 GC

为了清晰展示垃圾回收原理，我们将用 Python 模拟一个虚拟的「堆」和「对象」，实现一个极简的标记-清除垃圾回收器。首先，我们设计一个「微型世界」，包括一个 `VM` 类来模拟虚拟机和一个 `Object` 类来代表对象。`VM` 类包含栈（模拟根集合）、堆（用字典存储对象）和下一个可用对象 ID。`Object` 类则包含对象 ID、标记位和引用列表。

在核心功能实现中，`VM.new_object` 方法用于在堆上分配新对象，它创建一个 `Object` 实例并分配唯一 ID。`VM.push` 和 `VM.pop` 方法分别用于将对象压入栈和弹出栈，模拟根集合的变化。`VM.mark` 方法实现标记阶段，从栈中的根对象开始，递归遍历所有可达对象，并将它们的标记位设为 True。`VM.sweep` 方法实现清除阶段，遍历堆中的所有对象，删除未标记的对象，并重置已标记对象的标记位。`VM.gc` 方法是垃圾回收的入口，依次调用 `mark` 和 `sweep`。

下面是一个简单的代码示例，展示如何创建和回收普通对象。首先，我们初始化虚拟机，创建对象 A 并将其压入栈，然后创建对象 B 并由 A 引用。接着，将 A 弹出栈，触发垃圾回收，观察 A 和 B 是否被正确回收。

```
class Object:
    def __init__(self, obj_id):
        self.id = obj_id
        self.marked = False
        self.refs = []

class VM:
    def __init__(self):
        self.stack = []
        self.heap = {}
        self.next_id = 0

    def new_object(self):
        obj_id = self.next_id
        self.next_id += 1
        obj = Object(obj_id)
        self.heap[obj_id] = obj
        return obj

    def push(self, obj):
```

```
        self.stack.append(obj)

    def pop(self):
        return self.stack.pop() if self.stack else None

    def mark(self):
        for obj in self.stack:
            self._mark_recursive(obj)

    def _mark_recursive(self, obj):
        if obj is None or obj.marked:
            return
        obj.marked = True
        for ref_id in obj.refs:
            ref_obj = self.heap.get(ref_id)
            self._mark_recursive(ref_obj)

    def sweep(self):
        to_remove = []
        for obj_id, obj in self.heap.items():
            if not obj.marked:
                to_remove.append(obj_id)
            else:
                obj.marked = False
        for obj_id in to_remove:
            del self.heap[obj_id]

    def gc(self):
        self.mark()
        self.sweep()
```

在这段代码中，_mark_recursive 方法递归地标记所有从根对象可达的对象，确保活动对象不被误删。sweep 方法则清理未标记对象，释放内存。通过这个实现，我们可以测试两种场景：普通对象的回收和循环引用的处理。在循环引用场景中，创建两个对象 C 和 D，让它们互相引用，但不放入栈中，触发垃圾回收后，它们会被正确回收，证明标记-清除法解决了循环引用问题。

随着计算需求的增长，垃圾回收技术不断演进。并行垃圾回收使用多个线程并行工作，但应用程序仍需暂停；而并发垃圾回收允许 GC 线程与应用程序线程同时运行，显著减少了暂停时间，这在现代高性能 GC 如 G1 和 ZGC 中广泛应用。三色标记抽象是一种优雅的建模方法，将对象分为白色（未访问）、灰色（已标记但子对象未检查）和黑色（完全标记），这有助于实现并发标记过程。

总之，垃圾回收的核心思想是自动化内存管理，基于可达性判断对象生命周期。不同算法在时间、空间和复杂度

上各有权衡，理解这些原理有助于在使用带 GC 的语言时编写高效、健壮的代码。通过本文的理论学习和实践，读者可以更深入地掌握内存管理的艺术。