基本的垃圾回收（Garbage Collection）机制

王思成

Oct 29, 2025

从手动管理到自动化 —— 亲手打造一个微型内存管理器

在编程领域，内存管理始终是一个核心挑战。以 C 和 C++ 为例，开发者必须手动管理堆内存，使用诸如 `malloc` 和 `free` 或 `new` 和 `delete` 等函数。然而，这种手动方式容易导致内存泄漏、悬空指针和双重释放等问题。随着软件系统复杂度的提升，这些错误的维护成本急剧增加，往往成为程序崩溃和安全漏洞的根源。垃圾回收机制应运而生，它自动追踪和释放不再使用的内存，从而提升开发效率、减少内存相关错误并增强程序健壮性。垃圾回收已成为许多现代高级语言如 Java、C#、Go、Python 和 JavaScript 的运行时核心。本文旨在引导读者从理论到实践，深入理解垃圾回收的核心原理，并最终使用 C 语言实现一个简单的标记-清除垃圾回收器，完成一次从手动管理到自动化的探索之旅。

1　垃圾回收的基础概念

垃圾回收的核心在于准确识别哪些内存是存活的，哪些是垃圾。首先，我们需要定义根对象。根对象是垃圾回收器遍历的起点，通常包括全局变量、栈上的局部变量和寄存器中的变量。所有能被根对象直接或间接访问到的对象都被视为存活对象，其余则被标记为垃圾。这个概念称为可达性，它通过引用链形成一幅存活对象图。例如，如果对象 A 引用对象 B，而对象 B 引用对象 C，且根对象能访问 A，那么 A、B、C 都是可达的，因此存活。垃圾回收的步骤可以分解为分配、追踪和回收。分配阶段处理程序的内存请求；追踪阶段从根对象开始遍历所有存活对象；回收阶段则释放垃圾对象占用的内存，以便后续复用。这一过程确保了内存资源的有效利用。

2　经典的垃圾回收算法

垃圾回收算法有多种实现方式，其中三种经典方法是引用计数法、标记-清除法和复制算法。引用计数法为每个对象维护一个引用计数器，记录指向它的指针数量。当计数器为零时，对象被立即回收。这种方法的优点是实时性高，垃圾一经产生即刻回收，但缺点是无法处理循环引用，且计数器更新开销大。标记-清除法分为两个阶段：标记阶段从根对象开始遍历对象图，为所有可达对象打上标记；清除阶段遍历整个堆，回收未标记的对象。它能正确处理循环引用，但可能导致内存碎片，并在执行时暂停整个程序。复制算法将堆分为两个空间，只在一个空间分配内存，GC 时将存活对象复制到另一个空间，然后交换角色。这解决了内存碎片问题，分配速度快，但内存利用率低，复制存活对象开销较大。总体而言，引用计数法适用于简单场景，标记-清除法平衡了复杂性和功能，而复制算法在特定条件下效率高。这些算法各有优劣，实际应用中常根据需求选择或组合使用。

3 动手实现：一个简单的标记-清除 GC 器

我们将使用 C 语言实现一个基本的标记-清除垃圾回收器，目标是为一个简单的对象系统提供 GC 支持。首先，设计核心数据结构。定义一个 Object 结构体，包含类型、标记位以及指向其他对象的指针。例如：

```
typedef struct Object {
    int type; // 对象类型，例如用 0 表示整数，1 表示配对
    int marked; // 标记位，用于 GC 标记阶段
    struct Object* left; // 左子对象，模拟引用关系
    struct Object* right; // 右子对象，模拟引用关系
    struct Object* next; // 下一个对象，用于维护全局链表
} Object;
```

在这个结构中，type 字段标识对象类型，marked 用于标记阶段记录对象是否存活，left 和 right 模拟对象间的引用关系，next 用于将所有对象链接成一个链表，便于遍历。接下来，我们模拟一个虚拟机来管理 GC 状态。定义一个 VM 结构体：

```
typedef struct {
    Object* stack[STACK_SIZE]; // 栈数组，模拟根集合
    int stack_size; // 当前栈大小
    Object* first_object; // 指向第一个对象的指针，用于遍历所有对象
    size_t num_objects; // 已分配对象数量
    size_t max_objects; // GC 触发阈值，当 num_objects 达到此值时触发 GC
} VM;
```

stack 数组存储根对象，例如全局变量或局部变量；stack_size 记录栈中元素数量；first_object 指向对象链表的头部；num_objects 和 max_objects 用于控制 GC 触发条件。现在，实现核心函数。首先是分配函数 gc_alloc：

```
Object* gc_alloc(VM* vm, int type) {
    if (vm->num_objects >= vm->max_objects) {
        gc(vm); // 如果达到 GC 阈值，则触发垃圾回收
    }
    Object* obj = malloc(sizeof(Object));
    obj->type = type;
    obj->marked = 0;
    obj->left = NULL;
    obj->right = NULL;
    // 将新对象添加到链表头部
    obj->next = vm->first_object;
    vm->first_object = obj;
    vm->num_objects++;
```

```
    return obj;
}
```

这个函数在分配新对象前检查是否达到 GC 阈值，如果是则调用 GC 函数。它使用 malloc 分配内存，初始化对象字段，并将对象添加到全局链表中，同时更新对象计数。标记函数 mark 采用递归方式遍历对象图：

```
void mark(Object* object) {
    if (object == NULL || object->marked) return;
    object->marked = 1;
    mark(object->left); // 递归标记左子对象
    mark(object->right); // 递归标记右子对象
}
```

它检查对象是否为空或已标记，否则设置标记位并递归标记其子对象，确保所有可达对象都被标记。mark_all 函数从虚拟机的根集合开始标记：

```
void mark_all(VM* vm) {
    for (int i = 0; i < vm->stack_size; i++) {
        mark(vm->stack[i]);
    }
}
```

这个函数遍历栈中的所有根对象，对每个根对象调用 mark 函数，从而覆盖整个存活对象图。清除函数 sweep 负责回收未标记对象：

```
void sweep(VM* vm) {
    Object** obj = &vm->first_object;
    while (*obj) {
        if (!(*obj)->marked) {
            Object* unreached = *obj;
            *obj = unreached->next;
            free(unreached);
            vm->num_objects--;
        } else {
            (*obj)->marked = 0; // 重置标记位以备下次 GC
            obj = &(*obj)->next;
        }
    }
}
```

它遍历对象链表，如果对象未标记，则释放其内存并更新链表；否则重置标记位。GC 主函数 gc 组合这些步骤：

```
void gc(VM* vm) {
    mark_all(vm);
```

```
    sweep(vm);
}
```

这个函数简单地调用 mark_all 和 sweep，完成整个垃圾回收过程。为了测试，我们可以编写代码创建对象、构建引用关系（包括循环引用），然后强制触发 GC 并验证回收效果。例如，创建两个对象相互引用，然后移除根引用，触发 GC 后检查它们是否被正确回收。

4 现代 GC 的进阶与优化

在实际应用中，垃圾回收器远比我们实现的简单版本复杂。分代收集是一种常见优化，基于对象生命周期观察：绝大多数对象都是「朝生夕死」的。因此，堆被划分为年轻代和老年代。新对象在年轻代创建，经历多次 GC 后仍存活的对象会晋升到老年代。年轻代使用复制算法进行频繁 GC，而老年代使用标记-清除或其他算法，减少 GC 频率。这提升了整体效率，例如在年轻代中，复制算法的快速分配和碎片避免特性得到充分利用。增量式与并发式 GC 则旨在缩短 Stop-The-World 暂停时间。增量式 GC 将 GC 工作分解为多个小步骤，与用户程序交替执行；并发式 GC 在后台线程运行，与用户程序线程同时执行，但需处理复杂的并发问题，如读写屏障以确保数据一致性。这些优化使得垃圾回收在现代系统中更加高效和透明。

垃圾回收自动化了内存管理，显著提升了软件可靠性和开发效率。我们实现的标记-清除 GC 器简单有效，能处理循环引用，但存在内存碎片和执行暂停等局限性。读者可以在此基础上扩展，例如实现复制算法以优化碎片问题，或添加多线程支持以探索并发 GC。在实际语言运行时中，垃圾回收是一个极其复杂的子系统，通常是多种算法的结合，需要根据应用场景精心调优。通过本次实践，我们希望读者不仅理解了垃圾回收的核心原理，还能激发进一步探索的兴趣，亲手打造更高效的内存管理器。