计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(3):95−98 [doi: 10.15888/j.cnki.csa.006230]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于垃圾回收的三维引擎内存管理系统①

栾　添[1,2], 赵　奎[2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 沈阳计算技术研究所, 沈阳 110168)

摘　要: 本文面向三维引擎的上层内存管理, 提出了一种基于垃圾回收算法的管理系统, 辅助三维引擎使用者回收循环引用的内存. 该系统基于由库的形式提供, 并且不需要用户修改现有代码. 用户只需要使用系统提供的接口创建对象, 系统就能够在必要时自动回收内存. 测试表明, 本系统在可接受的开销下实现了循环引用内存的回收.

关键词: 垃圾回收; 三维引擎; 循环引用; 内存管理; 标记-清除

引用格式: 栾添,赵奎.基于垃圾回收的三维引擎内存管理系统.计算机系统应用,2018,27(3):95–98. http://www.c-s-a.org.cn/1003-3254/6230.html

3D Engine Memory Management System Based on Garbage Collection

LUAN Tian[1,2], ZHAO Kui[2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)

Abstract: The system, based on garbage-collection algorithm, orients to help the three-dimensional graphics engine to reclaim memory. The system is provided by library, without any other adapted source or compiler. With the objects constructed from interfaces of the system, the system would manage the memory automatically. The tests indicate that the system regain the circular reference memory with the affordable performance overhead.

Key words: garbage collection; three-dimensional (3D) graphics engine; circular reference; memory management; mark-and-sweep

当前, 三维引擎被广泛地应用在各个重要行业中. 为了满足三维引擎的性能要求, 开发语言一般为 C++. 作为非托管语言, C++中内存的释放需要由程序员介入. 三维引擎的对象引用关系非常复杂, 基于引用计数智能指针无法处理强循环引用的对象, 目前业界尚没有一个通用的、与标准库兼容的解决方案.

现存的解决方案存在一些问题, 比如虚幻引擎通过重新实现 C++运行库的方法来支持垃圾回收, 不兼容标准 C++; 而类似 Unity 逻辑脚本化的方式又存在与 C++跨语言交互困难和脚本语言本身的运行效率低下等问题.

本设计针对目前三维引擎回收复杂对象的困难, 使用标准 C++, 引入了垃圾回收算法辅助程序员释放循环引用的资源. 本系统在程序员无需修改现有代码和开发环境的情况下实现了循环引用对象的回收、空间不足自动回收内存、自动调用回收对象析构函数等功能.

1　技术概述

1.1　传统三维引擎的内存管理

三维引擎由众多模块构成. 内存管理系统作为三维引擎的支持系统, 管理内存的分配/释放. 如图, 传统三维引擎的内存管理大致分为两个部分: 面向性能的底层和回收内存的上层[1]. 底层对开发者透明, 旨在优化程序性能. 功能包括防止内存碎片化、加速内存分配速度等. 上层部分的功能是在合适的时机回收内存, 回收动作的执行和引擎运行时逻辑有关.

① 收稿时间: 2017-05-25; 修改时间: 2017-06-16; 采用时间: 2017-06-26; csa 在线出版时间: 2018-02-09

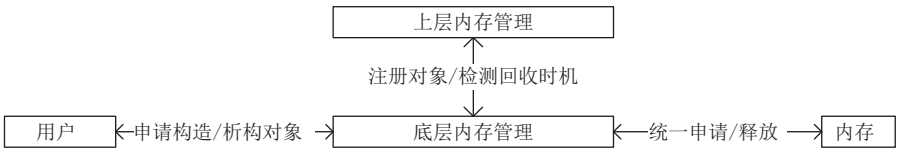

图 1 典型三维引擎内存管理结构

1.2 传统内存管理的不足

传统三维引擎的底层内存管理器主要面向内存的分配, 以加速分配/访问和防止碎片化为主[2]; 相对的, 上层的内存管理需要程序员手动管理, 对于比较复杂的引用关系无能为力. 业界的处理方式主要有两种: 顶层逻辑全部脚本化[2], 内存回收时机交由脚本解释器管理; 或者在 C++的基础上实现一个能被垃圾回收支持的基类, 所有的类都继承自该基类. 前者以 Unity 为代表, 缺点是跨语言的交互很容易出现性能瓶颈并且难以调试; 后者以虚幻引擎为代表, 既需要重写整个 C++标准库, 又很难混用托管和非托管的对象, 一定程度上损失了 C++语言的灵活性.

本文针对以上问题, 旨在和 C++标准库共存的前提下实现用户态的垃圾回收.

2 系统设计

2.1 需求分析

本文提出的内存管理系统旨在辅助程序员回收三维引擎运行中循环引用的对象, 考虑到三维引擎的复杂度和性能要求, 本系统需要有以下特性:

(1) 能和 C++标准内存模型共存的非侵入式设计

采用垃圾回收将造成一定的性能损失. 我们要保证这个系统只管理指定的对象而不干涉其他的部分. 在虚幻引擎中, 所有的类都继承自 UObject, 因此虚幻引擎重写了整个标准库. 本设计需要保证不影响程序的非托管部分.

(2) 保证和标准库的兼容性

对于 C++编写的三维引擎, 本管理系统需要保证兼容 C++的类型系统, 兼容性体现在两个方面: 迭代器和标准容器.

首先, 本系统提供的指针类型应该与 C++迭代器行为相仿, 由此来配合 C++标准库的交互. 其次, 托管对象需要和 C++标准容器兼容, 本系统需要考虑到标准库和标准容器使用的内存分配器对系统的潜在影响.

(3) 回收内存同时调用析构函数

作为 C++与 C 语言最重要的区别之一, 编译器保证了在对象离开作用域后自动调用的析构函数[3]. 此时, 对象持有的资源将被安全地释放, 以满足 RAII 原则. 本系统必须保证托管对象回收时能够正确地完成析构动作, 以保证 C++的基本语义. 比如, 引擎中的某个对象持有一个网络连接, 若对象析构, 析构函数中关闭链接的动作必须被执行.

2.2 整体架构设计

本系统架构如图 2 所示, 用户使用托管堆提供的接口构造托管对象, 调用接口的返回值为一个托管指针对象. 剩余部分的动作在托管对象的构造和托管指针的析构时分别激活.

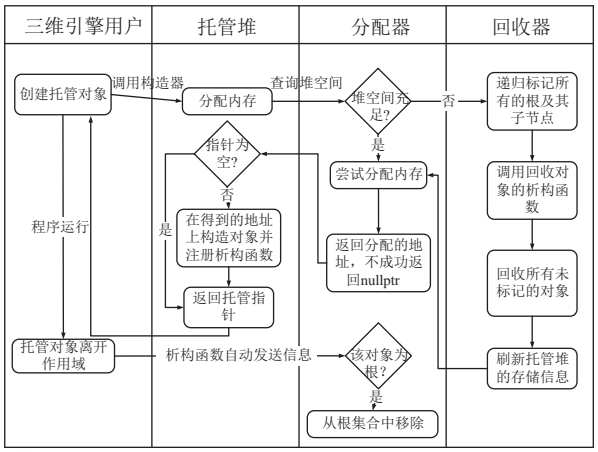

图 2 系统架构

本系统分为以下 4 部分.

(1) 托管堆: 用户通过调用托管堆提供的接口激活分配器分配内存. 托管堆持有存储所有托管对象的内存空间、提供构造托管对象的接口.

(2) 托管指针类: 托管堆返回给用户的操作对象都是托管指针. 这个类的对象保存所指向对象的真实地址, 类型签名, 对象占用内存大小等元信息. 需要和指针类型兼容 (重载指针运算的相关运算符). 根的集合由托管指针的构造/析构函数维护.

(3) 内存分配器: 由托管堆构造对象时激活, 尝试分配足够内存供用户使用, 返回托管指针, 空间不足则

激活回收器回收. 分配器在托管堆持有的内存中分配空间, 兼容 C++标准库的内存分配器 (allocator) 保证和标准库的交互.

(4) 回收器: 被分配器激活后负责递归搜索对象间引用关系, 执行回收动作时递归标记可达对象并回收其余内存.

各层组件交互的具体流程详见系统实现中伪代码部分的描述和注释.

3 系统实现

作为三维引擎内存管理系统中面向回收的部分, 本系统基于标准 C++11 实现, 以头文件的形式提供源码库. 本系统的使用者按照接口调用即可保证循环引用对象被安全回收.

面向系统设计中提到的三点需求: (1) 本系统通过库的方式实现提供服务, 保证了非侵入性的设计; (2) 在托管指针和内存分配器中通过定义相关 tag, 满足了迭代器和标准内存分配器的兼容性; (3) 构造托管对象时向管理系统注册析构函数, 满足了回收对象时调用析构函数的需求. 以下介绍各个组件的实现方式.

3.1 托管指针的实现

作为三维引擎的一部分, 托管指针需要胜任 C++指针的相关操作, 为了保证托管指针的灵活性以及类型安全, 本系统定义了一个泛型类 GcPtr<T>来指向各种指定类型的对象; 另一方面, 本系统实现了一个 GcPtrVoid 类型, 保证这个空类型能够作为任意托管指针的退化, 以此来模拟 C++中的 void*指针.

3.2 托管堆的实现

托管堆的负责分配内存和存储对象, 同时, 对象间的引用关系也由托管堆记录. 托管堆提供的操作是用户申请/释放托管对象的唯一方式. 托管堆工作的方式如下:

```
GcHeap: : make<T>(args…){
  p = allocate<T>();
  //调用分配器分配内存, 在得到的地址上就地调用构造//函数, 并将参数转发给 T 的构造函数
  if (p!= nullptr){
    construct<T>(p.get(),
      std: : forward<Args>(args)...);
  //注册析构函数以便回收时调用
  register(p->~T());
  }
  return p;
}
```

3.3 内存分配器的实现

本系统的内存分配器既可以和普通的三维引擎底层分配器配合以加速分配, 也可以单独使用.

内存分配器的工作以伪代码表示如下:

```
allocate<T>(){
  available = heap.available();//查询堆中现有空间
  //空间不足则调用回收器
  if (available < sizeof(T))
    collect();
  //为 p 分配一块足够的空间, 失败返回 nullptr
  auto p = allocate_from_existing_pages<T>(n);
  return p;
}
```

3.4 回收器的实现

为了解决三维引擎面向的上层逻辑中的复杂引用关系, 回收器的回收动作采用垃圾回收算法实现. 作为原型系统, 本系统目前采用“标记-清除”算法[4]作为原型实现. 回收器主要负责两个操作: (1) 从存活的根出发, 递归标记现有存活对象, 得到死亡的对象并回收空间. (2) 调用被回收对象的析构函数, 完成析构动作. 回收器以伪代码表示如下:

```
collcet(){
  //递归标记所有的根以及根的子节点
  for( p : roots)
    mark(p);
  //回收所有未标记的对象的内存, 并且调用对象的析构函数.
  for (obj : objects)
    if (unmarked(obj))
      destroy(obj);
  //刷新堆存储的信息, 比如剩余空间, 最大连续空间等
  flush_info();
}
```

3.5 核心算法描述

本设计的核心点为根的判定, 因为标记-清除算法没有给出根的普适判定方式. 为了解决这个问题, 在本设计中, 托管指针自身的内存地址在托管堆外 (程序运

行的栈和程序员手动申请的堆中) 被视为根结点.

此判定方式与标准容器结合会产生一个问题: 托管对象如果保存在未修改标准容器中 (如 vector) 会被视为一个根结点 (保存于 vector 申请的空间, 不在托管堆中). 循环引用表现为: 若容器没有析构, 容器中指针 (根结点) 指向的对象不会析构; 如果容器内有托管指针指向容器, 容器内的托管指针也不会析构. 为了解决这个问题, 对于可能存储托管指针的容器, 本系统重写了内存分配器 (此分配器将内存分配在托管堆中, 不会被误认为根结点), 分配器需要满足标准库中的内存分配器的特性 (Trait)[5], 来保证和 C++标准容器的兼容性.

4 实验与结果分析

本系统测试机配置如表 1.

表 1 测试机配置

设备名称	设备参数
CPU	Intel Xeon E3-1235@3.20 GHz
内存	16 G
操作系统	Windows 10 64bit
开发平台	Visual Studio 2017

本实验测试目的是对比托管指针相对 shared_ptr 的额外开销, 同时测试系统的可行性. 实验测试使用 C++11 标准库提供的 high_resolution_clock 计时, 对于每个测试, 运行 50 次取平均.

对于一个 int 对象, 构造新的托管指针和 shared_ptr 时间对比如表 2.

表 2 构造共享指针开销对比

指针对象个数(k)	运行时间(微秒)		运行时间比
	shared_ptr	GcPtr	
10	278	1341	4.82
20	571	2825	4.95
30	867	4626	5.34
40	1066	5811	5.45
50	1275	8541	6.70
60	1950	10 612	5.44
70	2336	22 115	9.47
80	2873	21 999	7.66
90	2572	22 406	8.71
100	2665	17 180	6.45

分别构造 shared_ptr 和 GcPtr 指向对象, 测试指针置空并调用回收动作运行的时间, 结果如表 3.

综上, 从根据运行结果中我们可以看出, 对于同一个对象, 构造/销毁共享指针的情况和 shared_ptr 的性能差距在 10 倍以内, 这大大优于脚本语言的性能 (数百到上千倍的开销)[6,7], 此时, 可以认为托管指针的开销是能够接受的. 此外, 构造循环引用的测试表明, 基于本系统构建的托管指针仍然能够回收无用对象, 测试运行结果证明了系统的可行性.

表 3 指针置空回收共享对象开销对比

指针对象个数(k)	运行时间(微秒)		运行时间比
	shaed_ptr	GcPtr	
10	73	79	1.08
20	141	324	2.30
30	240	334	1.39
10	73	79	1.08
20	141	324	2.30
30	240	334	1.39
40	282	576	2.04
50	354	704	1.99

5 结语

面向三维引擎的复杂对象引用关系, 本文讨论并实现了一个基于垃圾回收的内存管理系统, 在可接受的开销下解决了回收循环引用对象的问题. 本系统面向三维引擎开发, 但由于使用标准 C++实现, 并且对于被托管对象透明, 因此, 不限于三维引擎, 任何涉及到对象复杂引用关系的程序都可以使用.

实验结果表明, 本系统可以有效地回收循环引用的对象, 同时整个系统能够调用被回收对象的析构函数.

本系统目前实现仅采用了标记-清除算法作为原型实现, 下一步工作将考虑结合其他如复制算法、标记-压缩算法[4]等进一步改进该系统的性能.

参考文献

1 陈凯. 三维游戏引擎的设计与实现[硕士学位论文]. 杭州: 浙江大学, 2007.
2 Gregory J. Game Engine Architecture. Florida: CRC Press, 2009.
3 Stroustrup B. Programming: Principles and Practice Using C++. 2nd ed. Boston, Massachusetts: Addison-Wesley Professional, 2014.
4 张涛, 白瑞林, 邹骏宇. 基于生命期预测的分代式垃圾收集算法. 计算机工程, 2015, 41(7): 71–74, 81.
5 Heller T, Kaiser H, Diehl P. Closing the performance gap with modern C++. Taufer M, Mohr B, Kunkel J. High Performance Computing. Cham: Springer, 2016. 18–31.
6 李少华. 基于虚拟机的软件动态保护系统解释器的优化[硕士学位论文]. 西安: 西安电子科技大学, 2016.
7 吴作顺, 窦文华. 几个常用解释器的性能分析. 计算机工程与科学, 2002, 24(4): 83–84, 101.