C++ 和 Python3 内存安全性比较

魏思远

2024.12.13

摘要

内存安全是现代编程语言设计中的一个重要考虑因素。本文将比较 C++ 和 Python3 在内存安全方面的不同特性与机制。通过分析两种语言的内存管理方式、常见内存问题及其解决方案。本文认为 C++ 允许更精细的内存控制，但同时也引入了更多的内存管理风险；Python3 作为一种动态类型语言，提供了自动内存管理和强大的内存安全机制。

1 引言

内存安全 (Memory Safety) 指的是程序在运行时能够有效地管理内存，避免出现内存相关的错误和漏洞，从而确保程序的稳定性和安全性。内存安全是衡量程序设计语言的安全性的重要标准。Python3 和 C++ 是两种广泛使用的编程语言，它们在内存管理和安全性方面的设计理念不同。Python3 通过自动内存管理和动态类型系统来提高内存安全性，而 C++ 则允许开发者手动管理内存，提供了更大的灵活性，但也增加了内存错误的风险。

2 内存管理机制

Python3 采用了自动内存管理机制，最主要的机制是垃圾回收机制（Garbage Collection）。Python 的垃圾回收机制以引用计数（Reference Counting）为主，在此基础上，通过标记-清除（mark and sweep）解决容器对象可能会产生的循环引用问题，通过分代回收（generation collection）以空间换时间的方法来提高垃圾回收效率 [1,2]。C++ 主要依赖于程序员手动管理内存，允许使用指针，同时也设计了一些功能来提高自身的内存安全性，其中最具代表性的是智能指针。

2.1 Python3 的内存管理

Python3 的内存管理机制包括以下几个关键特征：

2.1.1 引用计数

引用计数器（Reference Counting）是 Python3 内存管理机制最核心的部分，它通过跟踪每个对象的引用数量来管理内存。具体来说，引用计数器会记录每个对象有多少个地方在使用它，当某个对象的引用计数为零时，表示该对象不再被使用，可以被安全地销毁，释放内存。引用计数机制的优点是简单直接，能够即时释放不再使用的对象，因此非常适用于短生命周期的小对象。通过引用计数，Python 能够有效地减少内存泄漏的风险。

```
1  a = []     # 创建一个列表对象，引用计数为 1
2  b = a      # 变量 b 引用了列表对象，引用计数增加为 2
3  del a      # 删除 a，引用计数减少为 1
4  del b      # 删除 b，引用计数为 0，内存被释放
```

图 1: 引用计数

然而，引用计数会遇到这样一种情况：如果两个对象互相引用对方，那么即使它们不再被程序中的任何其他部分引用，它们的引用计数也不会降为零，导致内存泄漏。为此，Python3 通过标记-清除算法来解决这一问题。

2.1.2 标记-清除算法

- **标记阶段**：垃圾回收器会遍历所有可达的对象（从根对象开始，如全局变量、栈中的变量等），并标记这些对象。
- **清除阶段**：遍历所有对象，清除未被标记的，释放其占用的内存。

2.1.3 分代收集

Python 的垃圾回收器将对象分为三代：**第一代（young generation）**，新创建的对象；**第二代（middle generation）**，存活了一段时间的对象；**第三代（old generation）**，存活时间较长的对象。新创建的对象在第一代，经

过多次垃圾回收后可能会被提升到更高的代。年轻一代的对象更频繁地被检查和回收，因为它们通常存活时间较短。[3]

2.1.4 触发机制

垃圾回收器会在以下情况下触发：

- 当第一代的对象数量达到一定阈值时。
- 当系统内存不足时。
- 手动触发（例如使用 `gc.collect()`，如图2）。
- 程序退出的时候。

```
1  import gc
2  class Node:
3      def __init__(self):
4          self.ref = None
5      '''创建循环引用'''
6      node1 = Node()
7      node2 = Node()
8      node1.ref = node2
9      node2.ref = node1
10     del node1
11     del node2
12     '''手动触发垃圾回收'''
13     gc.collect()
14     '''此时循环引用的对象会被回收'''
```

图 2: 循环垃圾回收

2.2 C++ 的内存管理

C++ 采用了手动内存管理的方式，程序员需要显式地分配和释放内存。C++ 的内存管理机制包括：

2.2.1 内存分配

C++ 的内存分配方式可以分为静态内存分配和动态内存分配。静态内存分配是在编译时进行的，内存的大小和生命周期在编译时就已经确定。通常，静态分配的内存用于全局变量、静态变量和常量。动态内存分配是在程序运行时进行的，使用 `new` 和 `delete` 运算符来分配和释放内存。这种

方式允许程序根据实际需要分配内存，但需要程序员手动管理内存的释放。3中展示了静态分配和动态分配在操作上的区别。

```
int globalVar; // 全局变量，静态分配
static int staticVar; // 静态变量
int* ptr = new int; // 动态分配一个int
delete ptr; // 释放内存
```

图 3: 静态内存分配与动态内存分配

2.2.2 智能指针

C++ 允许直接操作指针，使得程序能够直接访问和操作内存，从而提供了灵活性和高效性。但是，直接操作指针可能会带来内存安全方面的问题。C++11 引入了智能指针（如 `std::unique_ptr` 和 `std::shared_ptr`），提供了更安全的内存管理方式，减少了手动管理内存的复杂性 [4,5]。

3 常见内存问题

通过分析两者的内存管理机制，我们可以找到两者常见的内存问题。

3.1 C++ 中的内存问题

内存泄漏 内存泄漏是指程序在动态分配内存后，未能正确释放这些内存，导致内存被浪费和程序运行过程中占用的内存不断增加，最终可能导致系统内存耗尽，甚至程序崩溃。C++ 如果手动管理内存不当，非常容易出现内存泄漏风险。

悬空指针 悬空指针是指一个指针指向的内存位置已经被释放或不再有效，但指针本身仍然存在。使用悬空指针会导致未定义行为，可能引发程序崩溃、数据损坏或安全漏洞。

缓冲区溢出 缓冲区溢出是指程序试图向一个固定大小的缓冲区写入超过其容量的数据，从而导致数据覆盖了相邻的内存区域。由于缺乏边界检查，C++ 程序容易出现缓冲区溢出。

3.2 Python3 中的内存问题

循环引用 由于引用计数的机制并不完美，循环引用如果没有配备循环垃圾回收，仍然可能会导致内存泄漏。但是，我们可以通过使用 `weakref` 模块来创建弱引用，避免循环引用的发生。

性能开销 Python3 的垃圾回收机制可能导致性能波动，特别是在内存使用的高峰期。有时候我们会需要优化垃圾回收的策略来减少性能的影响。

4 结论

C++ 允许更精细的内存控制，适合对性能要求较高的系统开发，但手动内存管理带来的风险需要开发者谨慎处理。Python3 和 C++ 在内存安全方面各有优缺点。Python3 通过自动内存管理和动态类型系统提供了较高的内存安全性，而 C++ 则提供了更大的灵活性，适用于需要高性能、底层控制的场景，但也需要开发者特别关注内存管理的问题。

参考文献

[1] Python 3.8 documentation —Memory management `https://docs.python.org/3/cpython/memory.html`

[2] Python developer's guide - Memory Management `https://devguide.python.org/`

[3] Garbage Collection `https://docs.python.org/3/library/gc.html`

[4] C++ Primer Stanley B. Lippman, Josée Lajoie, Barbara E. Moo

[5] C++ Reference `https://en.cppreference.com/w/cpp/memory`