计算机学院 2019 级

高级语言程序设计 II 成果报告书

题目：动态内存的“垃圾回收管理”

课程号：304024030

课序号：02

教师名：游洪跃

学　号：2018141424166

姓　名：王月

目录

一、项目的目的与要求

1.1 背景介绍

C++程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高程序的效率，但是整体来说堆内存的管理是麻烦的，C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。在智能指针出现以前，通常使用 new 和 delete 来管理动态分配的内存，但是这种方式存在如下问题：

1) 忘记 delete 内存：导致内存泄露，并且这种错误很难发现（可以借用辅助工具来检测，比如：Windows 平台下的 Visual Leak Detector 和 Linux 平台下的 Valgrind 工具集中的 Memcheck）
2) 使用已经释放的对象：如果能够记得在释放掉内存后将指针置空并在下次使用前判空，尚可避免这种错误
3) 重复释放同一块内存：如果有两个指针指向相同的动态分配对象，很容易发生这种错误异常发生导致内存泄露：若在 new 和 delete 之间发生异常，则会导致内存泄漏

1.2 设计需求

设计一个类的“垃圾回收管理”，具有在程序运行结束时能自动释放用 new 所分配的还未被 delete 所释的动态内存，避免“内存泄漏”的出现。

二、工具/准备工作

2.1 回顾相关内容

垃圾（Garbage）就是程序需要回收的对象，如果一个对象不在被直接或间接地引用，那么这个对象就成为了垃圾，它占用的内存需要及时地释放，否则就会引起内存泄露。有些语言需要程序员来手动释放内存（回收垃圾），有些语言有垃圾回收机制（GC）。

C++程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高 了程序的效率，但是整体来说堆内存的管理是麻烦的，C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易 造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

- **引用计数**

引用计数是指，针对每一个对象，保存一个对该对象的引用计数，该对象的引用增加，则相应的引用计数增加。如果对象的引用计数为零，则回收该对象。

优点：引用计数最大的优点就是容易实现，C++程序员应该都实现过类似的机制。二是成本小，基本上引用计数为 0 的时候垃圾会被立即回收，而其他方法难以预测对象的生命周期，垃圾存在的时间都会比这个方法高。另，这种垃圾回收方式产生的中断时间最短。

缺点：最著名的缺点就是如果对象中存在循环引用，就无法被回收。例如，下面三个对象互相引用，但是不存在从根（Root）指向的引用，所以已经是垃圾了。但是引用计数不为 0.

还有一个缺点就是，引用计数不适合在并行中使用，多个线程同时操作引用计数，会引起数值不一样的问题从而导致内存错误。所以引用计数必须采用独占方式，如果引用操作频繁，那么加锁等并发控制机制的开销是相当大的。

- **shared_ptr**

利用引用计数->每有一个指针指向相同的一片内存时，引用计数+1，每当一个指针取消指向一片内存时，引用计数-1，减为 0 时释放内存。

- **week_ptr**

弱指针 ->辅助 shared_ptr 解决循环引用的问题

weak_ptr 是为了配合 shared_ptr 而引入的一种智能指针，因为它不具有普通指针的行为，没有重载 operator*和->,它的最大作用在于协助 shared_ptr 工作，像旁观者那样观测资源的使用情况。weak_ptr 可以从一个 shared_ptr 或者另一个 weak_ptr 对象构造，获得资源的观测权。但 weak_ptr 没有共享资源，它的构造不会引起指针引用计数的增加。使用 weak_ptr 的成员函数 use_count()可以观测资源的引用计数，另一个成员函数 expired()的功能等价于 use_count()==0,但更快，表示被观测的资源 (也就是 shared_ptr 的管理的资源)已经不复存在。weak_ptr 可以使用一个非常重要的成员函数 lock()从被观测的 shared_ptr 获得一个可用的 shared_ptr 对象， 从而操作资源。但当

expired()==true 的时候，lock()函数将返回一个存储空指针的 shared_ptr。

- **unique_ptr**

“唯一”拥有其所指对象，同一时刻只能有一个 unique_ptr 指向给定对象（禁止拷贝、赋值），可以释放所有权，转移所有权。

2.2 参考内容

[1] 深入理解计算机系统 [美]Randal E.Bryant / David O'Hallaron 机械工业出版社

[2] [Boehm, 1992] A proposal for garbage collector safe C Compilation.

[3] 代码的未来［日］松本行弘 人民邮电出版社

[4] [Boehm, 2002] Bounding space usage of conservative garbage collectors.

[5] [Hertz, 2005] Quantifying the performance of garbage collection vs. explicit memory management.

[6] [Wilson, 1992] Uniprocessor Garbage Collection Techniques

[7] C++标准提案 N2670 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2670.htm

[8] [Boehm, 2000] Fast Multiprocessor Memory Allocation and Garbage Collection.

[9] [Berger, 2000] Hoard: A Scalable Memory Allocator for Multithreaded Applications.

[10] 关于 C++ 0x 里垃圾收集器的讲座 http://blog.csdn.net/g9yuayon/article/details/1702694

[11] [Boehm, 1988] Garbage collection in an uncooperative environment.

[12] 《C++ Primer 中文版 第五版》

[13] [Boehm, 1993] Space Efficient Conservative Garbage Collection.

[14] [Henderson, 2002] Accurate garbage collection in an uncooperative environment.

[15] [Boehm, 1996] Simple garbage collector safety.

[16] [Hertz, 2005] Garbage Collection Without Paging

[17] [Ellis, 1993] Safe, efficient garbage collection for C++

[18] [Boehm, 2000] Reducing garbage collector cache misses.

2.3 硬件设施与 C++集成开发环境软件

- MinGW
- Visual Studio Code

三、分析

3.1 分析项目的实现方法

关于背景介绍之中的三点，通过程序员良好的代码习惯是有机会规避掉的，但是异常发生导致的内存泄露却无能无力，而使用智能指针的重要意义得以体现。另外智能指针可以极大的解放程序员管理动态内存，因为有些时候对象的析构时机比较难把握，尤其是我们将对象指针作为函数返回值时。而有了智能指针，我们就只需要关注内存的申请，内存的释放则由程序自动完成。

智能指针(smart pointer)的通用实现技术是使用引用计数(reference count)。智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象的指针指向同一对象。每次创建类的新对象时，初始化指针就将引用计数置为 1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，析构函数减少引用计数（如果引用计数减至 0，则删除基础对象）。

引用计数方式的原理是：在每个对象中保存该对象的引用计数，当引用发生增减时对计数进行更新。引用计数的增减，一般发生在变量复制，对象内容更新，函数结束（局部变量不在被引用），等时间点。当一个对象的引用计数为 0 时，则说明它将来不会再被引用，因此可以释放相应的内存空间。

3.2 并写出类声明

- **Sp_counter.h**

```
1.  #pragma once
2.  #include<iostream>
3.  #include<mutex>
4.  using namespace std;
5.
6.  /* 引用计数基类 */
7.  class Sp_counter {
8.  private:
9.      size_t* _count;
10.     std::mutex mt;
11. public:
12.     Sp_counter();
13.     virtual ~Sp_counter();
14.     Sp_counter& operator=(Sp_counter& spc);
15.     Sp_counter& GetCounter();
16.     size_t Get_Reference();
17.     virtual void Increase();
```

```
18.     virtual void Decrease();
19. };
```

- **smart_pointer.h**

```
1.  #pragma once
2.  #include<iostream>
3.  #include<mutex>
4.  #include<iostream>
5.  #include<mutex>
6.  #include "Sp_counter.h"
7.  using namespace std;
8.
9.  template<typename T>
10. class smart_pointer : public Sp_counter {
11. private:
12.     T* _ptr;
13. public:
14.     smart_pointer(T* ptr = NULL);
15.     ~smart_pointer();
16.     smart_pointer(smart_pointer<T>&);
17.     smart_pointer<T>& operator=(smart_pointer<T>&);
18.     T& operator*();
19.     T* operator->(void);
20.     size_t use_count();
21.
22. };
```

3.3 核心算法实现代码

```
1.  /*********************************************************
2.      核心算法的实现
3.   *********************************************************/
4.
5.  /* 得到引用计数值 */
6.  template<typename T>
7.  inline size_t smart_pointer<T>::use_count()
8.  {
9.      return this->Get_Reference();
10. }
11.
12. /* 拷贝构造 */
13. template<typename T>
```

```
14. inline smart_pointer<T>::smart_pointer(smart_pointer<T>& sp)
15. {
16.     cout << "子类拷贝构造" << endl;
17. 
18.     /* 防止自己对自己的拷贝 */
19.     if (this != &sp) {
20.         this->_ptr = sp._ptr;
21.         this->GetCounter() = sp.GetCounter();
22.         this->Increase();
23.     }
24. 
25. }
26. /* 赋值构造 */
27. template<typename T>
28. inline smart_pointer<T>& smart_pointer<T>::operator=(smart_pointer<T>& sp)
29. {
30. 
31.     /* 防止自己对自己的赋值以及指向相同内存单元的赋值 */
32.     if (this != &sp) {
33. 
34.         cout << "赋值构造" << endl;
35. 
36.         /* 如果不是构造一个新智能指针并且两个只能指针不是指向同一内存单元 */
37.         /* =左边引用计数减1,=右边引用计数加1 */
38.         if (this->_ptr && this->_ptr != sp._ptr) {
39.             this->Decrease();
40. 
41.             /* 引用计数为0时 */
42.             if (!this->Get_Reference()) {
43.                 cout << "引用计数为0,主动调用析构" << endl;
44.                 this->~smart_pointer();
45.                 //this->~Sp_counter();
46.                 cout << "调用完毕" << endl;
47.             }
48.         }
49. 
50.         this->_ptr = sp._ptr;
51.         this->GetCounter() = sp.GetCounter();
52.         this->Increase();
53.     }
54.     return *this;
55. }
56. 
```

```
57. /* 重载解引用*运算符 */
58. template<typename T>
59. inline T& smart_pointer<T>::operator*()
60. {
61.     return *(this->_ptr);
62. }
63. 
64. /* 重载解引用->运算符 */
65. template<typename T>
66. inline T* smart_pointer<T>::operator->(void)
67. {
68.     return this->_ptr;
69. }
70. 
71. /* 重载解引用=运算符 */
72. Sp_counter& Sp_counter :: operator = (Sp_counter& spc) {
73.     cout << "父类重载=" << endl;
74.     cout << "[释放 counter 内存]" << endl;
75.     delete _count;
76.     this->_count = spc._count;
77.     return *this;
78. }
79. 
80. /* 得到引用计数值 */
81. Sp_counter& Sp_counter::GetCounter() {
82.     return *this;
83. }
84. 
85. /* 增加引用计数值 */
86. void Sp_counter::Increase() {
87.     mt.lock();
88.     (*_count)++;
89.     //cout<<"_count++:"<<*_count<<endl;
90.     mt.unlock();
91. }
92. 
93. /* 减少引用计数值 */
94. void Sp_counter::Decrease() {
95.     mt.lock();
96.     (*_count)--;
97.     //cout<<"_count--:"<<*_count<<endl;
98.     mt.unlock();
99. }
```

四、完整程序代码

4.1 Sp_counter.cpp

```
/*********************************************************
    计数基类的实现
 *********************************************************/

#include<iostream>
#include<mutex>
#include "Sp_counter.h"
using namespace std;

/* 引用计数基类 */
Sp_counter::Sp_counter() {
    cout << "父类构造,分配 counter 内存" << endl;
    _count = new size_t(0);
}

Sp_counter :: ~Sp_counter() {
    if (_count && !(*_count)) {
        cout << "父类析构" << endl;
        cout << "[释放 counter 内存]" << endl;
        delete _count;
        _count = NULL;
    }
}

Sp_counter& Sp_counter :: operator = (Sp_counter& spc) {
    cout << "父类重载=" << endl;
    cout << "[释放 counter 内存]" << endl;
    delete _count;
    this->_count = spc._count;
    return *this;
}

Sp_counter& Sp_counter::GetCounter() {
    return *this;
}

size_t Sp_counter::Get_Reference() {
    return *_count;
```

```
}

void Sp_counter::Increase() {
    mt.lock();
    (*_count)++;
    //cout<<"_count++:"<<*_count<<endl;
    mt.unlock();
}

void Sp_counter::Decrease() {
    mt.lock();
    (*_count)--;
    //cout<<"_count--:"<<*_count<<endl;
    mt.unlock();
}
```

4.2 smart_pointer.cpp

```
#pragma once
#include<iostream>
#include<mutex>
#include<iostream>
#include<mutex>
#include "Sp_counter.h"
using namespace std;

template<typename T>
class smart_pointer : public Sp_counter {
private:
    T* _ptr;
public:
    smart_pointer(T* ptr = NULL);
    ~smart_pointer();
    smart_pointer(smart_pointer<T>&);
    smart_pointer<T>& operator=(smart_pointer<T>&);
    T& operator*();
    T* operator->(void);
    size_t use_count();

};
/*********************************************************
    智能指针类的实现
 *********************************************************/
```

```
26.
27. /* 子类参构造函数&带参数构造函数 */
28. template<typename T>
29. inline smart_pointer<T>::smart_pointer(T* ptr)
30. {
31.     if (ptr) {
32.         cout << "子类默认构造" << endl;
33.         _ptr = ptr;
34.         this->Increase();
35.     }
36. }
37.
38. /* 子类析构函数 */
39. template<typename T>
40. smart_pointer<T>::~smart_pointer()
41. {
42.     /* 指针非空才析构 */
43.     if (this->_ptr) {
44.         cout << "子类析构,计数减 1" << endl;
45.         if (this->Get_Reference())
46.             this->Decrease();
47.         if (!(this->Get_Reference())) {
48.             cout << "(((子类析构,主内存被释放)))" << endl;
49.             delete _ptr;
50.             _ptr = NULL;
51.         }
52.     }
53. }
54.
55. /* 得到引用计数值 */
56. template<typename T>
57. inline size_t smart_pointer<T>::use_count()
58. {
59.     return this->Get_Reference();
60. }
61.
62. /* 拷贝构造 */
63. template<typename T>
64. inline smart_pointer<T>::smart_pointer(smart_pointer<T>& sp)
65. {
66.     cout << "子类拷贝构造" << endl;
67.
68.     /* 防止自己对自己的拷贝 */
```

```
69.     if (this != &sp) {
70.         this->_ptr = sp._ptr;
71.         this->GetCounter() = sp.GetCounter();
72.         this->Increase();
73.     }
74.
75. }
76. /* 赋值构造 */
77. template<typename T>
78. inline smart_pointer<T>& smart_pointer<T>::operator=(smart_pointer<T>& sp)
79. {
80.
81.     /* 防止自己对自己的赋值以及指向相同内存单元的赋值 */
82.     if (this != &sp) {
83.
84.         cout << "赋值构造" << endl;
85.
86.         /* 如果不是构造一个新智能指针并且两个只能指针不是指向同一内存单元 */
87.         /* =左边引用计数减1,=右边引用计数加1 */
88.         if (this->_ptr && this->_ptr != sp._ptr) {
89.             this->Decrease();
90.
91.             /* 引用计数为0时 */
92.             if (!this->Get_Reference()) {
93.                 cout << "引用计数为0,主动调用析构" << endl;
94.                 this->~smart_pointer();
95.                 //this->~Sp_counter();
96.                 cout << "调用完毕" << endl;
97.             }
98.         }
99.
100.         this->_ptr = sp._ptr;
101.         this->GetCounter() = sp.GetCounter();
102.         this->Increase();
103.     }
104.     return *this;
105. }
106.
107. /* 重载解引用*运算符 */
108. template<typename T>
109. inline T& smart_pointer<T>::operator*()
110. {
111.     return *(this->_ptr);
```

```
112. }
113. template<typename T>
114. inline T* smart_pointer<T>::operator->(void)
115. {
116.     return this->_ptr;
117. }
```

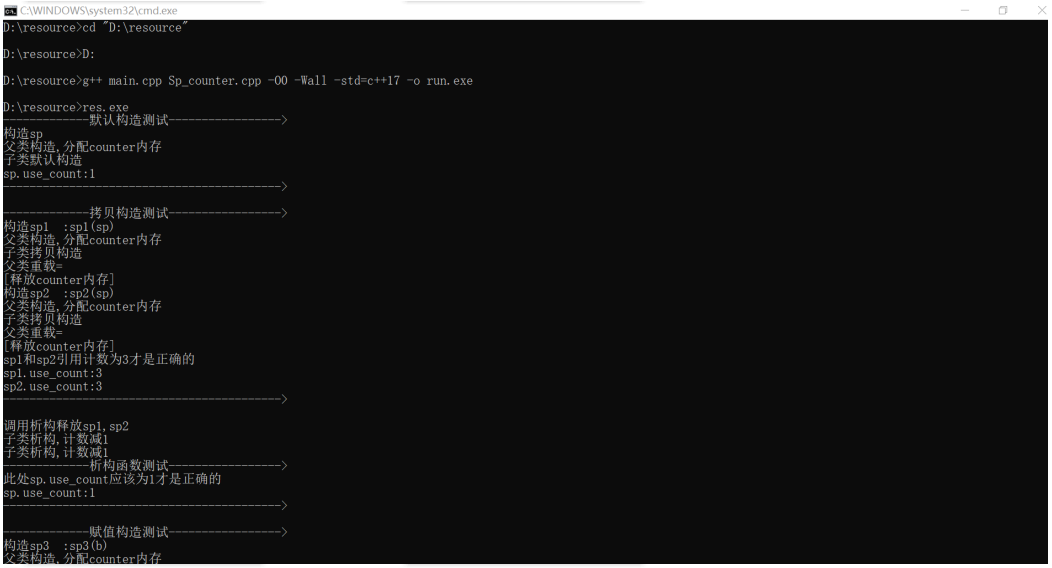
4.3 main.cpp

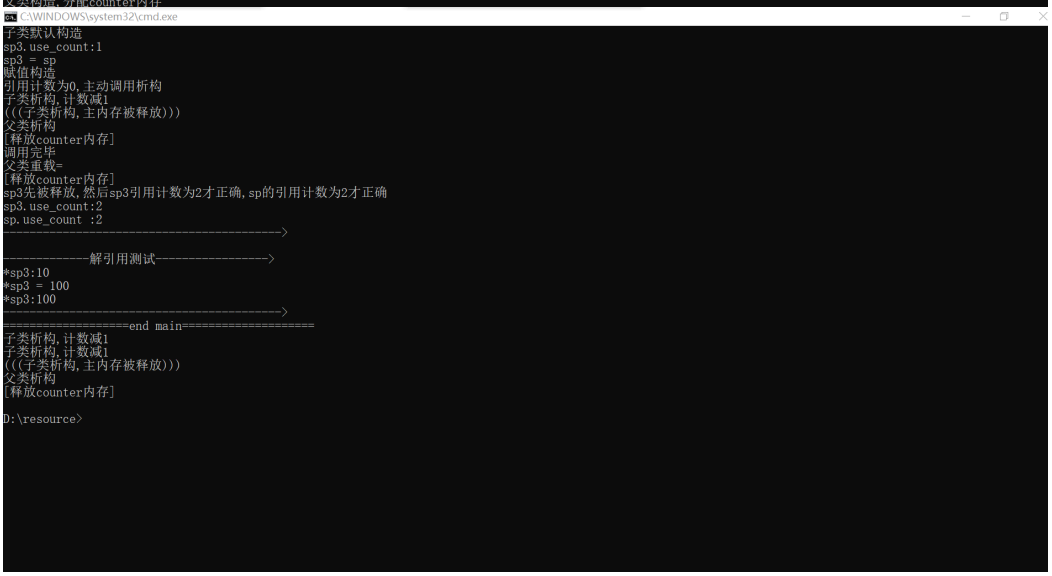
```
1.  /*************************************************************************
2.      测试
3.   *************************************************************************/
4.
5.  #include<iostream>
6.  #include<mutex>
7.  #include<string>
8.  #include"smart_pointer.hpp"
9.  #include"Sp_counter.h"
10. using namespace std;
11.
12. int main() {
13.     int* a = new int(10);
14.     int* b = new int(20);
15.     cout << "-------------默认构造测试----------------->" << endl;
16.     cout << "构造 sp" << endl;
17.     smart_pointer<int> sp(a);
18.     cout << "sp.use_count:" << sp.use_count() << endl;
19.     cout << "----------------------------------------->" << endl << endl;
20.
21.     {
22.         cout << "-------------拷贝构造测试----------------->" << endl;
23.         cout << "构造 sp1  :sp1(sp)" << endl;
24.         smart_pointer<int> sp1(sp);
25.         cout << "构造 sp2  :sp2(sp)" << endl;
26.         smart_pointer<int> sp2(sp1);
27.         cout << "sp1 和 sp2 引用计数为 3 才是正确的" << endl;
28.         cout << "sp1.use_count:" << sp1.use_count() << endl;
29.         cout << "sp2.use_count:" << sp2.use_count() << endl;
30.         cout << "----------------------------------------->" << endl << endl;
31.         cout << "调用析构释放 sp1,sp2" << endl;
32.     }
33.     cout << "-------------析构函数测试----------------->" << endl;
34.     cout << "此处 sp.use_count 应该为 1 才是正确的" << endl;
```

```
35.     cout << "sp.use_count:" << sp.use_count() << endl;
36.     cout << "---------------------------------------->" << endl << endl;
37.
38.     cout << "-------------赋值构造测试----------------->" << endl;
39.     cout << "构造 sp3  :sp3(b)" << endl;
40.     smart_pointer<int> sp3(b);
41.     cout << "sp3.use_count:" << sp3.use_count() << endl;
42.     cout << "sp3 = sp" << endl;
43.     sp3 = sp;
44.     cout << "sp3 先被释放,然后 sp3 引用计数为 2 才正确,sp 的引用计数为 2 才正确" << endl;
45.     cout << "sp3.use_count:" << sp3.use_count() << endl;
46.     cout << "sp.use_count :" << sp.use_count() << endl;
47.     cout << "---------------------------------------->" << endl << endl;
48.
49.     cout << "-------------解引用测试----------------->" << endl;
50.     cout << "*sp3:" << *sp3 << endl;
51.     cout << "*sp3 = 100" << endl;
52.     *sp3 = 100;
53.     cout << "*sp3:" << *sp3 << endl;
54.     cout << "---------------------------------------->" << endl;
55.
56.     // cout<<"sp3.use_count:"<<sp3.use_count()<<endl;
57.     //cout<<"sp.use_count:"<<sp.use_count()<<endl;
58.
59.     cout << "====================end main====================" << endl;
60.     return 0;
61. }
```

五、测试与结论

- cmd 中，cd 指令进入路径
- 输入 D: 就进入 D 盘的路径了
- -O0 不开启优化
- -Wall 显示所有 warning
- -std=c++17，即使用 c++17 标准
- -o run.exe 输出文件是 run.exe
- g++编译指令
- main.cpp，Sp_counter.cpp 编译文件
- gcc 会自动给链接





六、项目总结

6.1 项目具体实现功能

- 获取当前的引用指数
- 获取当前引用的内存地址
- 当引用指数减为 0 时释放内存
- 当一个指针取消指向一片内存时，引用计数-1
- 当一个指针指向相同的一片内存时，引用计数+1

6.2 项目中创新功能或实现思想

- 引用计数最大的优点就是相比标记清除和复制收集方式更容易实现；
- 当对象不再被引用的瞬间就会被释放，即引用计数为 0 的时候垃圾会被立即回收，而其他方法难以预测对象的生命周期，垃圾存在的时间都会比这个方法高；
- 由于释放操作是针对个别执行的，因此和其他算法相比，由 GC 而产生的中断时间就比较短。

6.3 项目功能不足之处

- 计数器值的增减处理繁重，严重依赖计数器。必须在引用发生增减时对引用计数做出正确的增减，而如果漏掉或者更改了引用计数就会引发很难找到的内存错误；
- 计数器需要占用很多位。假如我们用的是 32 位机器，那么就有可能要让 2 的 32 次方个对象同时引用一个对象，计数器有 32 位大小；
- 实现烦琐复杂。对于每次对象的变更都需要进行精确的维护，否则会造成内存无法会回收，即内存溢出；
- 无法释放循环引用的对象。如果对象中存在循环引用，就无法被回收；
- 引用计数不适合并行处理。如果多个线程同时对引用计数进行增减的话，引用计数的值就可能会产生不一致的问题（结果就会导致内存错误），为了避免这样的事情发生，对引用计数的操作必须采用独占的方式来进行。如果引用计数操作频繁发生，每次使用都要使用加锁等并发操作其开销也不可小觑。