**POCO C++库学习和分析 -- 内存管理 (一)**

        对于内存的管理，Poco C++库中主要包含了引用计数，智能指针，内存池等几个部分。下面将分别对这几个部分进行介绍。首先回顾一下，对于内存的管理，出现过的几种技术。C时代的内存池，主要解决内存碎片，和内存的频繁获取和释放的开销问题。到了C++时代，内存池仍然存在，但是出现了面对对象分配的内存池，解决问题还是一样。C++中智能指针，如STL中的auto\_ptr，boost库中share\_ptr等。通过把堆上对象的委托给智能指针（智能指针本身可以看成是一个栈对象），并在智能指针内部实现引用计数，当引用计数为0时，删除堆对象，从而达到让编译器自动删除堆对象的目的，实现了堆对象的自动管理。Java和C#的垃圾收集，在语言层次分装，所有的对象都在堆上分配，然后交由语言本身管理，程序员无需关心对象内存的释放。

**1. 引用计数和智能指针概述：**

        对于C和C++来说，堆上内存的管理是交由程序员完成的，程序员如果在堆上分配了一块内存，就必须负责释放掉。如果不小心，就会造成内存泄露。因此所有C/C++程序员设计程序时，对指针和内存的管理都会如履薄冰，非常的小心。而Java和C#则不同，虽然所有对象都被放在堆上，但由于语言本身存在垃圾收集机制，程序员不再需要关心对象的释放。这或多或少的能够使程序员更多的把精力放在其业务编程上。

        讲到这里，就顺便扯开去，讲一些题外话。对于某些编程技术讨论时的一些看法。如同制造业一样，制造业存在很多种类，对制造业的划分方法当然也很多。在制造业中存在一个特殊的种类，装备制造业。也就是制造机器的制造业。对于程序员来说也是一样，绝大多数程序员都是面对业务进行编程的，而极少数程序员则是为了制造编程工具或者提供更方便的编程方法而编制程序。这个区别往往导致，不同程序员看问题的角度不同，结果当然也不同。我想很多时候，问题的答案都是不唯一的。接下去继续讨论Poco吧。

        通过引用计数和智能指针机制，C++也可以完成了某种意义上的垃圾收集的工作。程序员通过使用智能指针，同样不需要再关注堆对象的释放，当然胶水代码还是需要的。在Poco库中存在两种智能指针，AutoPtr和SharedPtr。

**1.1 引用计数(Reference Counting)**

        Poco中对于引用计数，是如此描述的。“Reference counting is a technique of storing the number of references, pointers, or handles to a resource such as an

object or block of memory. It is typically used as a means of deallocating objects which are no longer referenced. --Wikipedia”。

翻译过来即"引用计数是一项用来存储指向某个对象或者内存块的引用、指针或句柄的数量的计数。这项技术通常被用于对象不再存在任何引用关系时的释放 -- wikipedia"。

        对于引用计数有如下特点：

        1. 当一个对于对象的引用被销毁或者覆盖时，这个对象的引用计数的数量减少。

        2. 当一个对于对象引用被创建或者拷贝时，这个对象的引用计数数量增加。

        3. 初始化对象的引用，其引用对象的引用计数为1.

        4. 当对对象的引用计数为0时，对象被删除。

        5. 在多线程场景下，对引用计数的增加、减少和比较操作必须为原子操作.

**1.2 对象所有权(Object Ownership)**

 对象所有权有如下特点：

        1. 拥有动态对象所有权的某个所有者，当动态对象不在被需要时，必须负责删除动态对象。

        2. 如果动态对象的所有者删除动态对象失败，将导致程序内存泄露。

        3. 指向动态对象的其他物体，不能删除该动态对象。

        4. 动态对象的所有权是可传递的，但在任何给定时刻，动态对象只有一个所有者

**1.3 引用计数和对象所有权的关系**

        二者关系如下：

        1.  一个指针获取到引用计数对象的所有权时，不会增加引用计数的数目。分两种情况来讨论：

             a)  引用计数对象原先不存在所有者（换句话说，引用计数对象刚刚被创建）

             b)  引用计数对象原先存在所有者，由于原先的所有者放弃了对对象的所有权，所以新所有者获取对象所有权的动作并不会增加引用计数数目。

         2. 通常，第一个在对象创建后被对象赋值的指针拥有拥有权，其他的则没有

**2 AutoPtr**

**2.1 Poco中AutoPtr的例子**

         首先来看AutoPtr使用的一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. <span style="font-weight: normal;">#include "Poco/AutoPtr.h"
2. #include "Poco/RefCountedObject.h"
3. **class** A: **public** Poco::RefCountedObject
4. {
5. };
6. **int** main(**int** argc, **char**\*\* argv)
7. {
8. Poco::AutoPtr<A> p1(**new** A);
9. A\* pA = p1;
10. // Poco::AutoPtr<A> p2(pA); // BAD! p2 assumes sole ownership
11. Poco::AutoPtr<A> p2(pA, **true**); // Okay: p2 shares ownership with p1
12. Poco::AutoPtr<A> p3;
13. // p3 = pA; // BAD! p3 assumes sole ownership
14. p3.assign(pA, **true**); // Okay: p3 shares ownership with p1
15. **return** 0;
16. }</span>

         RefCountedObject是什么呢？其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. **class** Foundation\_API RefCountedObject
2. /// A base class for objects that employ
3. /// reference counting based garbage collection.
4. ///
5. /// Reference-counted objects inhibit construction
6. /// by copying and assignment.
7. {
8. **public**:
9. RefCountedObject();
10. /// Creates the RefCountedObject.
11. /// The initial reference count is one.
13. **void** duplicate() **const**;
14. /// Increments the object's reference count.
16. **void** release() **const**;
17. /// Decrements the object's reference count
18. /// and deletes the object if the count
19. /// reaches zero.
21. **int** referenceCount() **const**;
22. /// Returns the reference count.
24. **protected**:
25. **virtual** ~RefCountedObject();
26. /// Destroys the RefCountedObject.
28. **private**:
29. RefCountedObject(**const** RefCountedObject&);
30. RefCountedObject& operator = (**const** RefCountedObject&);
32. **mutable** AtomicCounter \_counter;
33. };

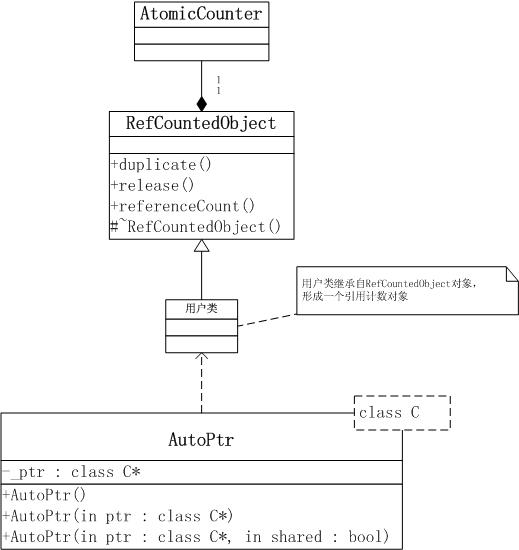
         RefCountedObject原来是一个引用计数对象，其中封装了原子计数类AtomicCounter。实现了两个接口，其中duplicate()用来增加引用计数数目，每次调用引用计数增加1；release()用来减少引用计数数目，每次调用引用计数减少1.

         事实上AutoPtr支持一切实现duplicate()和release()的引用计数对象。下面是另外一个例子：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. #include "Poco/AutoPtr.h"
2. **using** Poco::AutoPtr;
3. **class** RCO
4. {
5. **public**:
6. RCO(): \_rc(1)
7. {
8. }
10. **void** duplicate()
11. {
12. ++\_rc;                                                    // Warning: not thread safe!
13. }
15. **void** release()
16. {
17. **if** (--\_rc == 0) **delete** **this**;                             // Warning: not thread safe!
18. }
20. **private**:
21. **int** \_rc;
22. };
24. **int** main(**int** argc, **char**\*\* argv)
25. {
26. RCO\* pNew = **new** RCO;                                    // \_rc == 1
27. AutoPtr<RCO> p1(pNew);                                  // \_rc == 1
28. AutoPtr<RCO> p2(p1);                                    // \_rc == 2
29. AutoPtr<RCO> p3(pNew, **true**);                            // \_rc == 3
30. p2 = 0;                                                 // \_rc == 2
31. p3 = 0;                                                 // \_rc == 1
32. RCO\* pRCO = p1;                                         // \_rc == 1
33. p1 = 0;                                                 // \_rc == 0 -> deleted
35. // pRCO and pNew now invalid!
36. p1 = **new** RCO;                                           // \_rc == 1
37. **return** 0;
38. }
39. // \_rc == 0 -> deleted

**2.2 Poco中AutoPtr的类图**



         从类图中可以看出，Poco中AutoPtr类是和RefCountedObject是配套使用的，如果用户类继承自RefCountedObject，就可以由AutoPtr实现垃圾收集。

**2.3 Poco中AutoPtr的说明和注意事项**

         Poco::AutoPtr实现了一个引用计数对象的智能指针，能够实例化任何支持引用计数的类。符合下列要求的类可以被定义成为支持引用计数：

        1. 这个类必须存在引用计数，在对象被创建时，引用计数被初始化值为1

        2. 这个类必须支持duplicate()接口增加引用计数

        3. 这个类必须支持release()接口减少引用计数，并且在引用计数为0时，删除类对象。

        Poco::AutoPtr的操作符合1.2节中对于“对象所有权(Object Ownership)”的描述：

        1. 当AutoPtr<C>从原生指针C\*构造时，AutoPtr获取到对象C的所有权(对象的引用计数为1，保持不变)。

        2. 当使用赋值操作符“=”把原生指针C\*赋予AutoPtr<C>时，AutoPtr 获取到对象C的所有权（新对象的引用计数保持不变）。下面是AutoPtr关于这个实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. AutoPtr& assign(C\* ptr)
2. {
3. **if** (\_ptr != ptr)
4. {
5. **if** (\_ptr) \_ptr->release();
6. \_ptr = ptr;
7. }
8. **return** \***this**;
9. }
11. AutoPtr& operator = (C\* ptr)
12. {
13. **return** assign(ptr);
14. }

        3. 当AutoPtr<C>从另一AutoPtr<C>构造时，两个AutoPtr共享一个C的拥有权，引用计数增加

        4. 当使用赋值操作符“=”把AutoPtr<C>赋予另一个AutoPtr<C>时，两个AutoPtr 共享一个C的拥有权，引用计数增加

        Poco::AutoPtr的操作符与值语义：

        1. Poco::AutoPtr 支持关系表达式"=="," !=", "<", "<="," >"," >="

        2. 当Poco::AutoPtr 中的原生指针为空时，使用"\*"和"->"操作符，将抛出异常“NullPointerException”

        3. Poco::AutoPtr 支持所有的值语义函数(默认构造函数、拷贝构造函数、赋值操作符)，并且能够被各种容器所使用(如std::vector、std::map等)

        4. 使用AutoPtr::isNull()或者AutoPtr::operator ! ()可以测试其内部原生指针是否为空

        Poco::AutoPtr与转换函数：

        1. 和原生指针一样， Poco::AutoPtr 支持转换操作。其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. **template** <**class** Other>
2. AutoPtr<Other> cast() **const**
3. /// Casts the AutoPtr via a dynamic cast to the given type.
4. /// Returns an AutoPtr containing NULL if the cast fails.
5. /// Example: (assume class Sub: public Super)
6. ///    AutoPtr<Super> super(new Sub());
7. ///    AutoPtr<Sub> sub = super.cast<Sub>();
8. ///    poco\_assert (sub.get());
9. {
10. Other\* pOther = **dynamic\_cast**<Other\*>(\_ptr);
11. **return** AutoPtr<Other>(pOther, **true**);
12. }

        2. AutoPtr的cast总是安全的，因为其内部使用了dynamic\_cast，因此如果转换非法只会导致一个空指针。

        3. AutoPtr的赋值函数的兼容性通过模板构造函数和赋值操作符来支持。其定义如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. **template** <**class** Other>
2. AutoPtr(**const** AutoPtr<Other>& ptr): \_ptr(**const\_cast**<Other\*>(ptr.get()))
3. {
4. **if** (\_ptr) \_ptr->duplicate();
5. }

8. **template** <**class** Other>
9. AutoPtr& assign(**const** AutoPtr<Other>& ptr)
10. {
11. **if** (ptr.get() != \_ptr)
12. {
13. **if** (\_ptr) \_ptr->release();
14. \_ptr = **const\_cast**<Other\*>(ptr.get());
15. **if** (\_ptr) \_ptr->duplicate();
16. }
17. **return** \***this**;
18. }

21. **template** <**class** Other>
22. AutoPtr& operator = (**const** AutoPtr<Other>& ptr)
23. {
24. **return** assign<Other>(ptr);
25. }

       **注意事项和陷阱：**

        当使用赋值操作符“=”把一个AutoPtr赋给一个原生指针，然后再把这个原生指针赋予另个AutoPtr时要非常小心。这时候两个AutoPtr'都会声称拥有对象的所有权。这是非常坏的一件事情。必须明确的告知AutoPtr需要分享对象的所有权。使用下列两个函数可以解决问题：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. AutoPtr::AutoPtr(C\* pObject, **bool** shared);
2. AutoPtr& AutoPtr::assign(C\* pObject, **bool** shared);

        其中shared值必须为true。下面是一个样例：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8631621)

1. #include "Poco/AutoPtr.h"
2. #include "Poco/RefCountedObject.h"
3. **class** A: **public** Poco::RefCountedObject
4. {
5. };
7. **int** main(**int** argc, **char**\*\* argv)
8. {
9. Poco::AutoPtr<A> p1(**new** A);
10. A\* pA = p1;
11. // Poco::AutoPtr<A> p2(pA);             // BAD! p2 assumes sole ownership
12. Poco::AutoPtr<A> p2(pA, **true**);          // Okay: p2 shares ownership with p1
13. Poco::AutoPtr<A> p3;
14. // p3 = pA;                             // BAD! p3 assumes sole ownership
15. p3.assign(pA, **true**);                    // Okay: p3 shares ownership with p1
16. **return** 0;
17. }

**2.4 其他**

        很明显Poco中的AutoPtr和STL的auto\_ptr是不同,Stl中auto\_ptr某种意义上是一个Scope ptr,其实现并不依赖引用计数，它和boost库中的scoped\_ptr很相似。而Poco中的AutoPtr和boost库中的intrusive\_ptr是类似的，基本上可以看做是同一东西。

（版权所有，转载时请注明作者和出处  <http://blog.csdn.net/arau_sh/article/details/8631621>）