How can I give away a COM reference just before my object destructs?

devblogs.microsoft.com/oldnewthing/20240125-00

January 25, 2024

Raymond Chen

Last time, we noted that most implementations of the `IMemoryBufferReference.Closed` event are broken because they hand out COM references to themselves after C++ has already committed to destructing the object, so calling `AddRef()` on the COM reference doesn't actually extend the object's lifetime.

So how do you raise this `Closed` event when the last reference is released, if releasing the last reference also commits you to destruction?

More generally, how can I execute some code when the last reference is released, if that code might try to extend the object's lifetime?

One solution is to keep track of two different reference counts, one for "client-visible reference counts" and one for "internal reference counts", and then raise the `Closed` event when the client-visible reference count goes to zero but before the internal reference count also goes to zero. (You would destruct the object when *both* reference counts go to zero.)

But I'm going to do things a little differently.

My trick is to have the object call `AddRef()` on itself at construction, so that the object reference count is 2 when it is given to the caller. Since one of those reference counts is artificial, we relize that when the reference count drops to 1, that means that all "client-visible" references have been released. The only thing keeping the object alive is the artificial reference.

We can now perform that cleanup operation, passing a COM reference to a proper object which is not yet committed to destruction. After the call returns, we release the artificial reference. If the cleanup operation did not attempt to extend the lifetime of the COM reference, then then our release of the artificial reference will drop the reference count to zero, and the object destructs immediately. But if it used `AddRef()` to extend the lifetime, then our release will not drop the reference count to zero, and the object will remain alive. It's being cleaned up, but still alive.

Of course, we have to be careful that when the reference count drops to 1 a *second* time, we don't get confused and try to clean up a second time and (worse) release the nonexistent artifical reference.

Here's how you could implement this pattern in various COM frameworks.

We start with WRL, which has no real surprises.

```
// C++/WRL

struct ObjectWithSpecialCleanup :
    RuntimeClass<IWhatever>
{
    ObjectWithSpecialCleanup()
    {
        // WRL does not support aggregation,
        // so don't need to assert "not aggregating"
        InternalAddRef();
    }

    STDMETHOD_(ULONG, Release)() override
    {
        auto count = RuntimeClass::Release();

        if (count == 1) {
            if (!m_cleanupStarted.exchange(true, std::memory_order_relaxed)) {
                DoCleanup(this); /* Might AddRef this object */
                count = RuntimeClass::Release();
            }
        }

        return count;
    }

    ⟦ ... ⟧

    std::atomic<bool> m_cleanupStarted;
};
```

Next comes C++/WinRT:

```
// C++/WinRT

struct ObjectWithSpecialCleanup :
    implements<ObjectWithSpecialCleanup,
               IWhatever>
{
    static_assert(!outer(), "Must not be composable");

    ObjectWithSpecialCleanup()
    {
        NonDelegatingAddRef();
    }

    decltype(std::declval<implements>().Release())
    __stdcall Release() noexcept override
    {
        auto count = NonDelegatingRelease();

        if (count == 1) {
            if (!m_cleanupStarted.exchange(true, std::memory_order_relaxed)) {
                DoCleanup(this); /* Might AddRef this object */
                count = NonDelegatingRelease();
            }
        }

        return count;
    }

    [[ ... ]]

    std::atomic<bool> m_cleanupStarted;
};
```

In C++/WinRT, we use the trick we learned a little while ago to ensure that the C++/WinRT implementation class is not aggregated.

The biggest pain point is figuring out the correct return value for the overridden `Release` method. The return type depends on whether classic COM interop has been enabled in C++/WinRT. If so, then the return value of `Release()` needs to be `ULONG`, to match the classic COM `Release()` method. But if not, then the return value needs to be `uint32_t`, to match the C++/WinRT version. We solve the problem by simply looking at the method we are trying to override and saying that we return whatever that guy returns.

Next up is manual implementation.

```
// Manually implemented
struct MyPage : IPage
{
    MyPage()
        // Bonus refcount is released when count drops to 1
        : m_refCount(2) { }

    STDMETHOD_(ULONG, Release)() override
    {
        auto count = InterlockedDecrement(&m_refCount);

        if (count == 1) {
            if (!m_cleanupStarted.exchange(true, std::memory_order_relaxed)) {
                DoCleanup(this); /* Might AddRef this object */
                count = InterlockedDecrement(&m_refCount);
            }
        }

        return count;
    }

    LONG m_refCount;
    std::atomic<bool> m_cleanupStarted;
    ⟦ ... ⟧
};
```

The sneaky part here is that instead of incrementing the reference count in the body of the constructor, we just combine the extra reference count into the initial value.

The final example is ATL.

```
// ATL
class ObjectWithSpecialCleanup :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<ObjectWithSpecialCleanup>,
    public IWhatever
{
public:
    // Prevent this class from being aggregated
    static constexpr void* m_pOuterUnknown = nullptr;

    BEGIN_COM_MAP(Widget)
        COM_INTERFACE_ENTRY(IAgileObject)
    END_COM_MAP()

    // NOTE! Requires creation via two-phase initialization
    HRESULT InitializeComponent() noexcept
    {
        InternalAddRef();
        m_cleanupNeeded.store(true, std::memory_order_relaxed);
        return S_OK;
    }

    ULONG InternalRelease()
    {
        auto count = CComObjectRootEx::InternalRelease();

        if (count == 1) {
            if (m_cleanupNeeded.exchange(false, std::memory_order_relaxed)) {
                DoCleanup(this); /* Might AddRef this object */
                count = CComObjectRootEx::InternalRelease();
            }
        }

        return count;
    }

    std::atomic<bool> m_cleanupNeeded;
    ⟦ ... ⟧
};
```

ATL is the outlier.

For ATL, we need to use two-phase initialization (which we learned how to do a little while ago) because we are not allowed to fiddle with the reference count in the constructor or in the optional `FinalConstruct()` method. We mark the object as needing cleanup only if we make it to the second phase. The sense of the flag is reversed for ATL because it starts out not needing cleanup, and gains the cleanup requirement later. The other libraries don't have this "nascent state" where the object is constructed but not yet ready.[1]

The memory order for the atomic accesses to the cleanup flag is relaxed because we are counting on the object to protect its own internal state from multithreaded access when necessary.

[1] ATL uses the policy that objects are created with a reference count of zero, rather than one. This may have made sense at the time, but it creates complications when you write code that runs during this dangerous “zero-refcount” period.