On the whole idea of giving away a reference to yourself at destruction

devblogs.microsoft.com/oldnewthing/20240228-00

February 28, 2024

Raymond Chen

One of the responses to my discussion of how to give away a COM reference to yourself at destruction was roughly “The crazy convolutions required to accomplish this demonstrate how COM is a disaster.”

Well, some people may feel that COM is a disaster, but at least this specific piece of COM isn’t a disaster.

In a non-disastrous language, say, C++ with the standard library,[1] if somebody asked you, “How do I run some code when my `shared_ptr` reference count drops to 1?”, the answer is simple: You can’t do it! C++’s `shared_ptr` does not give you that level of access to the internal reference count. You can peek at the reference count by asking for the `use_count()`, but that is necessarily just an approximation due to multithreading. Even knowing that the use count is 1 doesn’t prove that you have the last `shared_ptr`. The reference count might go back up to 2 due to a race against a `weak_ptr::lock()`, and your attempt to detect when the reference count has dropped to 1 has failed.

In C# and Java, you are allowed to rescue an object in its finalizer, known as *object resurrection*. Java weak references expire before the object is submitted for finalization, so even if you rescue the object in its finalizer, it’s too late to preserve the weak references. C# weak references default to Java style, but you can ask for a “long” weak reference which retains its connection to the object even past finalization. Unfortunately, the object cannot control the types of weak references that people create, and if somebody creates a long weak reference, then they can access your object while it is finalizing! That is likely to result in unhappiness if the finalizer cleaned up external resources (that being the primary purpose of finalizers), since the object no longer has its external resources and consequently is unable to perform any useful operations.

Maybe the real problem is the design that required an object to hand out a reference to itself at destruction. That’s what forced us into the weird contortions. But at least it’s *possible* with COM. That’s more than can be said for other frameworks.

Bonus chatter: I perhaps did not note clearly enough that the intended design of the `Closed` event is that it is raised only the first time the last application reference is released. If the application rescues the object in its `Closed` event handler, and then subsequently releases the rescued object, the `Closed` event does not get raised again. The rules for the Windows Runtime is that the `IClosable.Close` method tells the object, "I have no further intention of using this object," and further operations[2] will fail with `RO_E_CLOSED`.

The "did I already raise the `Closed` event?" flag is atomic because knowing that the reference count has dropped to 1 does not prove that only one thread can be using the object: A weak reference might increment the reference count back up, and then down to 1 a second time. C++/WinRT and C++/WRL objects support weak references by default, so this is something to worry about. If your framework doesn't support weak references (or if you've disabled them), then the flag doesn't need to be atomic because you know that there are no competing threads.

[1] Some people feel that C++ itself is also a disaster.

[2] With the exception of event handler unregistration and calling the `IClosable.Close` itself.