What are the rules for CoMarshalInterThreadInterfaceInStream and CoGetInterfaceAndReleaseStream?

devblogs.microsoft.com/oldnewthing/20151021-00

October 21, 2015

Raymond Chen

Last time, we looked at the recommended ways of marshaling objects between apartments or between processes, but what if those mechanisms are not available to you? (Or if you're simply curious about the lower-level constructions that make the recommended ways possible.)

How do I share an object in one apartment with another apartment in the same process? (Assuming that the `RoGetAgileReference` function is not available.)

This is the most common case. You have an interface pointer that is valid in one apartment, and you want to use that interface pointer in another apartment. The pattern for this is

- On the originating apartment, call `CoMarshalInterThreadInterfaceInStream`. (What a mouthful.) This takes the object and generates a bunch of bookkeeping that allows the object reference to be used on a second apartment. The mental model for this is that you took the object, did an `AddRef`, and *stored the reference count in a stream*. The stream that it returns is safe for multi-threaded use.
- Transmit the stream to the second apartment by whatever means you wish. Since the source and destination apartments are in the same process, you can just copy the `IStream*` pointer across.
- The second apartment calls `CoGetInterfaceAndReleaseStream`. This reconstitutes the object from the stream in a form that can be used by the second apartment, transferring the reference count from the stream to the reconstituted object, and releases the stream. (This last clause is such a nasty gotcha, I'm going to devote a special day to it.)
- The second apartment happily accesses the object.

Behind the scenes, different things happens depending on the nature of the object you marshaled.

If the object being marshaled is apartment-threaded, then it expects that all its methods are called only on the original apartment. In this case, unmarshaling creates a proxy object.

(Obvious special case: If the apartment-threaded object is unmarshaled in the same apartment as it was marshaled, then no proxy is needed. But if you're going do that, why bother with marshaling in the first place?)

If the object being marshaled is free-threaded, then it is safe to call from any thread. In that case, unmarshaling the object just gives you a direct pointer to the original object.

Objects can provide custom marshaling behavior. For example, if an object is immutable, it may choose to marshal by value rather than by reference. In that case, unmarshaling creates a clone of the original.

What does the second apartment do when it is finished with the object?

The second apartment calls `Release()` just like a reference to any other COM object. If the second apartment got a proxy, this releases the proxy, and the proxy will notify the original object that there is one fewer outstanding reference.

What if I change my mind after calling `CoMarshalInterThreadInterfaceInStream` and decide that I don't want to share the object with a second apartment after all?

In that case, you call `CoReleaseMarshalData` from the originating apartment. This tells COM, "Hi, um, yeah, sorry about that." COM undoes its bookkeeping and releases the original object, thereby restoring the reference count.

What happens when all of the threads associated with the originating apartment release their last references to the object, while the second apartment still has an outstanding reference via a proxy?

The object is not yet destroyed because there is still an outstanding reference from the second apartment. The object gets destroyed only when all outstanding references are released.

What happens if the threads associated with the originating apartment exit while the second apartment still has an outstanding reference to the object via a proxy?

When the original apartment uninitializes COM, any outstanding proxies are disconnected from the underlying object. (When the proxies disconnect, they release their references, which causes the underlying object to be destroyed because there are no longer any references to it.) If somebody tries to use the proxy to talk to the object, the call fails with `RPC_E_SERVER_DIED_DNE`. The "DNE" stands for "did not execute". (If somebody was talking to the proxy *while it was being destroyed*, then they will get `RPC_E_SERVER_DIED`. The missing DNE means that the call may have started executing before the proxy was disconnected.)

What if I want to unmarshal more than once?

Oh, hey, look at the time. We'll pick up this question next time.

Bonus chatter: The need for marshaling comes from two principles:

1. Don't force objects to support multi-threading.

If you declare your object as apartment model, then you can assume that all accesses to your object will occur from a single thread. This simplifies most objects tremendously, since multi-threading is *hard*. And it seems rather mean to force an object to support multi-threading if the person writing it has no intention of using it from multiple threads.

You can also look at this as a compatibility constraint: Windows 3.1 didn't support multi-threading, so all COM objects back in Windows 3.1 were implicitly single-threaded. The apartment threading model allows this code to be ported in a relatively straightforward manner.

2. Let objects have direct pointers to each other whenever possible.

If two objects are in the same apartment and they want to talk to each other, each gets a pointer to the other with no proxy object in between. People writing high-performance code insist on this principle. "I don't want COM getting in my way and slowing down my business logic."

Marshaling reconciles the above two rules.

An alternative design would be that creating an object always created a free-threaded proxy, and all communication with the object would be done through the proxy. When a method is invoked on that proxy, it either would perform a direct call to the underlying object if the method is being invoked on the same thread that the object expects, or it would marshal the call to the correct thread if not. This would allow COM objects to be transmitted freely within a process without the need for explicit marshaling. However, it also violates rule 2.

When C++ added support for threading in C++11, it went a different way: All objects must support being created on any thread, any method can be called from any thread, and some methods are thread-safe, and some aren't, and it is the caller's responsibility to ensure that a previous not-thread-safe method call has completed before it makes a new not-thread-safe method call. This creates a burden on the implementor compared to the apartment model because the object needs to support being called from any thread, and it is responsible for its own marshaling if it needs to access resources that have hard thread affinity. It also creates a burden on the caller compared to using marshaling because the caller needs to keep in mind which methods are thread-safe and which aren't, and it needs to make sure that all threads which are sharing an object work together to ensure that only one not-thread-safe method call is active at a time.[2]

On the other hand, it means that there is no marshaling for in-process objects.

(C++ doesn't try to address cross-process access to objects, not does it particularly care about making C++ objects consumable from other languages, and it's okay with forcing all consumers of an object to recompile if the object's implementation changes.)

[1] Profound confusion has resulted from the fact that one of the object threading models is called *apartment* model, thereby overloading the word *apartment*. I don't know what they were thinking. It means you can have apartment model objects that aren't compatible with your apartment. (Apartment model objects are compatible with single-threaded apartments, but not multi-threaded apartments.)

[2] You also have to figure out how to solve problems like this:

- Object A creates an object X.
- Object A shares object X with object B.
- Object A registers a callback with object X.
- Object B calls into object X from thread 1.
- While object X is processing the request, object A on thread 2 decide that it also wants to call into object X.
- Object A must wait until object X has completed the request from object B.
- The request from object B requires object X to invoke the callback registered by object A.
- But object X cannot invoke the callback because object A is being used by thread 2 right now.

How do you avoid a deadlock?

Raymond Chen

Follow