What is COM marshaling and how do I use it?

devblogs.microsoft.com/oldnewthing/20151020-00

October 20, 2015

Raymond Chen

COM has this concept called “marshaling”, with one L. Basically, marshaling is the mechanism by which an object that is accessible to one apartment can be made accessible to another apartment.

Incomplete backgrounder on apartments: In COM, an apartment is a collection of threads that are treated as equivalent from a COM standpoint. The details of apartments aren’t important for today’s discussion, so let’s assume that all the apartments in question are single-threaded apartments (STA). Single-threaded apartments, as you might guess from their name, are apartments that consist of a single thread. In this world, the concepts of thread and apartment line up one-to-one, which makes discussion easier.

Incomplete backgrounder on threading models: Each COM object declares how it deals with threads. The most common cases are *apartment model objects*,[1] which can be used only on the thread that they were created on, and *free-threaded objects*, which can be used from any thread. Free-threaded objects are easier to use, but apartment model objects are much, much easier to write, since you don’t have any of that troublesome multi-threading to deal with.

Okay, back to marshaling.

Since the rules for apartment model objects is that they can be accessed only from the thread on which they were created, you need to do some extra work if you want to access them from another thread: You need to hire a lackey. COM calls this lackey a “proxy”. When you invoke a method on the proxy object, the call is routed back to the originating apartment, the method executes on its original apartment, and then the results are routed back to original caller. (And if any of the parameters to the method are themselves objects, then COM needs to create proxies for those objects, too!) Marshaling is a mechanism for creating proxies.[2]

What if I need to marshal an object to another thread (same process or different process), and I already have access to an object in the destination thread?

The easy way to do this is to take advantage of the parenthetical remark up above: If you already have an interface pointer to an object in another apartment, you can define a marshalable interface that accepts an interface pointer. For example, if you need to pass a widget to another object, you probably already have a method on that other object called `IMumble::ColorizeWidget` that takes a widget and some other parameters. Just call the method from your originating apartment. The implementation of `IMumble::Colorize-Widget` will receive a pointer to the proxy, and it can use the proxy to access the original widget. It can event retain that pointer (with an appropriate `AddRef`, of course) to communicate with the original widget even after the method returns.

Of course, this creates a chicken-and-egg problem: In order to get my object to another apartment, I need access to an object on that apartment. In other words, once you have one object operating across apartments, you can use it to get more objects to operate across apartments. But how do you marshal the *first* object?

What if I need to marshal an object from one thread to another thread in the same process, and I don't have a friendly object on the other thread yet?

You can use the `RoGetAgileReference` function. You give it an interface accessible to the current apartment, and the `RoGetAgileReference` function returns you an `IAgile-Reference` interface pointer that figures out how to create proxies so you don't have to. What makes the agile reference special is that it is free-threaded: You can share the pointer freely among apartments.[3] To gain access to the original interface pointer, call `IAgile-Reference::Resolve`. Note, of course, that the pointer that comes out of `IAgile-Reference::Resolve` is valid only in the apartment in which you called `IAgile-Reference::Resolve`.

In the case of passing an object to a newly-created thread, you would do something like this:

- Calling thread calls `RoGetAgileReference` and gets an agile reference.
- Calling thread creates the background thread and passes it the agile reference pointer, transferring to the background thread the obligation to release the agile reference. (If the calling thread could not create the background thread, then it needs to release the agile reference itself since it was unable to transfer the obligation.)
- The background thread calls `CoInitialize[Ex]`.
- The background thread takes the agile reference and calls `IAgile-Reference::Resolve` to recover the original object.
- The background thread releases the agile reference, since it doesn't need it any more.

What if I need to marshal an object from one process to another?

Again, the easy way is to pass the object as a parameter to a method on an object you already have in the destination process.

But what if those mechanisms aren't available to me?

The `RoGetAgileReference` function was introduced in Windows 8.1, so it may not be available to you. And maybe you don't have an object in the destination process that you can use as a foothold. We'll look at the mechanism that operates under the covers (and answer some more questions) over the next couple of days.

[1] This unfortunate reuse of the word *apartment* has been the source of significant confusion. A better name would have been something like *fixed-threaded objects*.

[2] Of course, this entire discussion assumes that the interface in question is marshalable. Otherwise, you get the mysterious `E_NOINTERFACE`.

[3] Note that I didn't say that you can share the pointer freely among *threads*. The threads you share the pointer with must still belong to apartments. For single-threaded apartments, that means calling `CoInitialize` or `CoInitializeEx` with the `COINIT_APARTMENT-THREADED` flag. For multi-threaded apartments, this means calling `CoInitializeEx` with the `COINIT_MULTITHREADED` flag somewhere in the process.[4]

[4] Preferably, you called `CoInitializeEx` with the `COINIT_MULTITHREADED` flag on the thread itself. It is technically legal to call it from some other thread, but then you're using the implicit MTA, and that has its own problems.

Raymond Chen

Follow