# What are the various usage patterns for manually-marshaled interfaces?

devblogs.microsoft.com/oldnewthing/20220614-00

June 14, 2022

Raymond Chen

COM organizes threads into *apartments*. All the thread in an apartment have access to the same objects, and if you want to grant access to another apartment, you have to do it by a mechanism known as *marshaling*.

The easiest way to marshal an object's interface is to ask somebody else to manage it for you: The `RoGetAgileReference` function creates a new object that represents an *agile reference* to the original object. In COM, the term *agile* means that it can be used in any apartment. You can then ask that agile reference to *resolve* a reference to the original interface, and it will give you an object which you can use from the apartment doing the resolving.

**Bonus reading**: When should I use delayed-marshaling when creating an agile reference?

A lower-level method is to use `CoMarshalInterThreadInterfaceInStream` to take an object interface and save it into a byte stream. That stream is a magic cookie that can be used to recover the original interface from any apartment. And since it's a stream of bytes, you have any number of ways of sharing those bytes with another thread: You could save them in a global variable, you could write them to a named pipe, or you could print them on a piece of paper, bury it in the ground, then come back and dig up the paper and OCR the digits. Whatever the mechanism, you can pass the bytes to `CoUnmarshalInterface` (or its own helper function `CoGetInterfaceAndReleaseStream`) and it will produce an object that you can use.

But we're going to look at what happens at an even lower level: The `CoMarshalInterface` function is the one that generates the stream. In addition to the obvious parameters (the stream to which to write the bytes, the interface being marshaled, and an interface pointer), there are two somewhat more mysterious parameters: The destination context and the marshal flags.

The destination context describes *where* you intend to unmarshal the object. Here are the destination contexts in order of distance from the source:

| Flag | Meaning | Group |
|---|---|---|
| `MSHCTX_CROSSCTX` | Another context in the same apartment. | Same-process |
| `MSHCTX_INPROC` | Another apartment in the same process. | |
| `MSHCTX_LOCAL` | Another process on the same computer which can share memory with the source. | Same-machine |
| `MSHCTX_NOSHAREDMEM` | Another process on the same computer which cannot share memory with the source. | |
| `MSHCTX DIFFERENT-MACHINE` | Another computer. | Cross-machine |

Marshaling across integrity levels would be a case where the source and destination processes cannot share memory.

Although there are five different marshaling contexts, most marshaling code cares only about which group they belong to: The same-process group ( `CROSSCTX` and `INPROC` ) the same-machine group ( `LOCAL` and `NOSHAREDMEM` ) and the cross-machine group ( `DIFFERENT-MACHINE` ).

Meanwhile, the marshal flags describe *how* you intend to unmarshal the object.

| Flag | Number of times to unmarshal | Strong or weak reference |
|---|---|---|
| `MSHLFLAGS_NORMAL` | Exactly once. | Strong |
| `MSHLFLAGS TABLESTRONG` | Any number of times (possibly zero). | Strong |
| `MSHLFLAGS_TABLEWEAK` | Any number of times (possibly zero). | Weak |

The first case is called “normal” because it is the most common case of marshaling: You have a single reference that you want to transfer to another apartment. We’ll see that knowing in advance that this is how you intend to marshal the object interface allows for some optimizations.

The last two cases are for where the object can be unmarshaled any number of times (possibly zero). They differ in whether the stream itself keeps the object alive.

The marshal flags control the usage pattern for managing the stream.

If you choose either of the “table” (reusable) marshaling flags, the sequence is

- Call `CoMarshalInterface` to write the bytes to the stream.
- Call `CoUnmarshalInterface` to produce an object from the stream. Repeat this as many times as you like, or skip it entirely.
- Call `CoReleaseMarshalData` to signal that you are not going to be unmarshaling from the stream any more.

The difference between the two “table” versions is whether the stream itself keeps the object alive. If you choose a weak reference, then it is your responsibility not to call `CoUnmarshal-Interface` once the object has been destroyed. (Typically, you accomplish this by ensuring that the stream’s lifetime is encompassed by the object’s lifetime.)

If you choose “normal” (one-time) marshaling, the sequence is

- Call `CoMarshalInterface` to write the bytes to the stream.
- Either
  - Call `CoUnmarshalInterface` to produce an object from the stream, or
  - Call `CoReleaseMarshalData` to abandon the operation.

In the case of “normal” marshaling, once you call `CoUnmarshalInterface` or `CoRelease-MarshalData`, the stream can no longer be used further. Conceptually, you can imagine that calling `CoUnmarshalInterface` “normal” marshaling is like “table strong” marshaling, with the added feature that the `CoUnmarshalInterface` implicitly performs a `CoReleaseMarshal-Data` when it’s done.

Next time, we’ll start looking at the mechanics of how COM marshaling is performed.

Raymond Chen