How do you get into a context via IContext-Callback::ContextCallback?

devblogs.microsoft.com/oldnewthing/20191128-00

November 28, 2019

Raymond Chen

For the past few articles, I've been talking about COM contexts and how you can use them to resolve the impasse that is created when the COM programming model requires you to keep your DLL loaded because there are still outstanding references to objects in it, but a competing programming model (say, a Windows NT service or an application's custom plug-in model) requires you to unload your DLL.

As I noted before, the method for doing this is the `IContextCallback:: Context-Callback` method. When you call this method, you provide a callback function: The `IContextCallback:: ContextCallback` switches to the target context and calls your callback function. When your callback function returns, its `HRESULT` return value is propagated out of the context back to the original caller.

Okay, now here is where I admit that the `IContextCallback:: ContextCallback` method is kind of weird. It's weird because it's really a backdoor into the very low-level COM infrastructure. This is the same infrastructure that COM itself uses to marshal method calls, but we're repurposing it to marshal a simple function call.

The parameters to `IContextCallback:: ContextCallback` are as follows:

- A callback function. This is the function that executes in the context.
- A pointer to a `ComCallData` structure. The only thing interesting in this structure is the `pUserDefined` , which allows you to pass a pointer's worth of data to the callback.
- An interface ID that represents the interface you are fake-marshaling.
- A zero-based index representing the method on the interface you are fake-marshaling.
- A reserved parameter that must be `nullptr` .

I say "fake-marshaling" because what we're doing is prank-calling the existing marshaling infrastructure.

> Two teenagers sit at a kitchen table. One is on the phone.
>
> “Um, hi, is this the COM marshaler? Great. Could you uh marshal a method call into a context for me?” (stifled giggle)
>
> (regains composure) “No, I’m totally serious.”
>
> “Who is this?” (looks at other prankster nervously) “This is um… your boss.”
>
> “The object is in this context I’m giving you. Do you see it? Okay, great. The interface is `IID_ ITotallyNotAJoke`.”
>
> (to other prankster) “I think they’re falling for it.”
>
> (to phone) “What’s the method index? Um…” (looks at other prankster, who shrugs) “Five? Yeah, five.”
>
> “Uh huh. Okay, great. Just marshal that method call for me, okay? I’ll stay on the line until you’re done. Thanks.”

What you’ve done is ask the COM infrastructure to marshal the specified method on the specified interface to an object living inside that context. COM thinks that your callback is going to invoke the target method, but instead, your callback is going to do something different entirely. The net effect is that you managed to get your callback to execute inside the context.

Since this is a prank call, you have to be careful not to raise any suspicions. The interface must not be `IUnknown`, because the COM marshaler treats `IUnknown` as a special case.[1] Similarly, the method index must not be less than 3, because the first three methods on every interface come from `IUnknown`.[2]

In practice, people tend to pick `IContextCallback` as the fake-marshaled interface and 5 as the fake-marshaled method index.

Now, it turns out that COM is not a total dupe when it comes to these fake-marshaled calls. Once they realized that the interface ID and method index were basically-garbage parameters, they decided to put them to use: You can request particular behavior by passing special sentinel values. Therefore, in practice, the values for the fake-marshaled interface and method index are as follows:

Behavior	Interface ID	Method index
Classic	`IID_ ContextCallback`	5

No activity lock	`IID_ IEnterActivityWithNoLock`	5
No ASTA reentrancy	`IID_ ICallbackWithNoReentrancyTo-ApplicationSTA`	5

For general purpose use (i.e., if you don't have any need for the other flavors), the recommended practice is to disable ASTA reentrancy, but you'll see the classic version in older code.

The callback is executed in the target context, switching threads if necessary, and the return value of the callback becomes the return value of the `ContextCallback` method.

Here's a handy wrapper function.

```
template<typename TLambda>
HRESULT InvokeInContext(IContextCallback* context, TLambda&& lambda)
{
  ComCallData data;
  data.pUserDefined = &lambda;
  return context->ContextCallback([](ComCallData* data) -> HRESULT {
    auto& lambda =
      *reinterpret_cast<TLambda*>(data->pUserDefined);
    return lambda();
  }, &data, IID_ICallbackWithNoReentrancyToApplicationSTA, 5, nullptr);
}
```

As a convenience, we can permit the lambda to return `void`, in which case we treat it as if it had returned `S_OK`.

```
template<typename TLambda>
HRESULT InvokeInContext(IContextCallback* context, TLambda&& lambda)
{
  ComCallData data;
  data.pUserDefined = &lambda;
  return context->ContextCallback([](ComCallData* data) -> HRESULT {
    auto& lambda =
      *reinterpret_cast<TLambda*>(data->pUserDefined);
    if constexpr (std::is_same_v<void, decltype(lambda())>) {
      lambda();
      return S_OK;
    } else {
      return lambda();
    }
  }, &data, IID_ICallbackWithNoReentrancyToApplicationSTA, 5, nullptr);
}
```

Creating an object inside a context and marshaling it out could be written something like this:

```
HRESULT CreateSomethingInContext(ISomething** something)
{
  *something = nullptr;
  Microsoft::WRL::AgileRef agileRef;
  HRESULT hr = InvokeInContext(context, [&]()
  {
    Microsoft::WRL::ComPtr<ISomething> something;
    HRESULT hr = MakeSomething(&something);
    if (SUCCEEDED(hr)) {
     hr = something.AsAgile(&agileRef);
    }
    return hr;
  });
  if (SUCCEEDED(hr)) {
    hr = agileRef.CopyTo(something);
  }
  return hr;
}
```

Inside the context, we create the `Something` and convert it to an agile reference, which we store in the `agileRef` variable, which is shared by reference with the code that runs outside the context. This is legal because agile references can be taken freely across contexts. When we're back outside the context, we convert the agile reference to an `ISomething`, which will cause a proxy to be created, and it's that proxy which we return to the caller.

Later, we can disconnect all the proxies from the context:

```
HJRSULT DisconnectAllProxiesFromContext()
{
  return InvokeInContext(context, []()
  {
    return CoDisconnectContext(INFINITE);
  });
}
```

Next time, we'll see how we can use existing contexts, rather than creating our own custom ones.

[1] COM has special knowledge of the methods of `IUnknown` because that's the interface that everything else is built out of. The `IUnknown` interface is the foot in the door that makes the rest of marshaling possible.

For example, when you call `IUnknown::AddRef` on a proxy, it doesn't marshal the `AddRef` call to the original object. it merely updates the reference count of the proxy. If you prank-called COM and said, "Yeah, can you get all ready to call `IUnknown::` `AddRef`, but at the last minute, instead of doing the `AddRef`, just call me back," it would say, "Well, I know that `IUnknown::` `AddRef` doesn't require any context switching at all, so I can optimize out the whole thing."

[2] There may be other interfaces that the COM marshaler gives special treatment, like `IInspectable` and `IWeakReference`, so you should avoid those too. If you stick to the values in the table, then you'll avoid the problems.

Raymond Chen

Follow