Yo dawg, I hear you like COM apartments, so I put a COM apartment in your COM apartment so you can COM apartment while you COM apartment

devblogs.microsoft.com/oldnewthing/20191126-00

November 26, 2019

Raymond Chen

Last time, we learned about COM apartments, with the two main flavors the *single-threaded apartment* and the *multi-threaded apartment*. But it turns out you can also create "miniature apartments" inside your apartment. (Is this like Airbnb for COM or something?)

This "miniature apartment" is formally known as a COM *context* and goes by the name `CLSID_ ContextSwitcher`. You create one by calling

```
IContextCallback* context;
CoCreateInstance(CLSID_ContextSwitcher, nullptr,
    CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&context));
```

You can then enter the context by calling the `IContextCallback:: ContextCallback` method with a function you would like to execute inside that context. I'm going to postpone further discussion of the `IContextCallback:: ContextCallback` method to another article, because the `IContextCallback:: ContextCallback` method is kind of weird, and untangling it will take a while.

Back to contexts. Why would you want to create a custom context anyway?

The original audience for custom contexts was Windows NT services which expose COM objects to clients. Services also have to respond to shutdown requests. This puts them in a bit of a pickle: They are required to clean up and unload from the process when given a shutdown request,[1] but they also *cannot* unload (under penalty of access violation) until all clients have released their references to objects in the service.

The solution is for the service to create a private little COM context for its objects. It then enters the context and registers its object factory. When a client requests an object, the factory will be called to produce the desired object. Since the factory is inside a context, the resulting object will also be inside that same context. The client receives a proxy object that talks to the object living inside the context.

Context

object ← proxy ←Client

When the server is told to shut down, it enters the context one last time to revoke its factory and call `CoDisconnectContext`. The `CoDisconnectContext` function disconnects all outstanding proxies from the underlying objects, erasing the arrow from the proxy to the object:

Context

object proxy ←Client

The expectation is that disconnecting all the proxies will cause the reference counts of all the objects in the context to drop to zero, and everything will be destroyed. The service can destroy the context, and everything that had references to the service DLL is now gone, thus allowing the service DLL to unload itself from memory.

Meanwhile, the clients are left holding a broken proxy. Any attempt to access the underlying object from the proxy wll return the error `RPC_E_DISCONNECTED`.

Although Windows NT Services were the original audience for private contexts, they are not the only valid use for them. Next time, we'll look at another way they can become useful.

[1] This means that the shutdown "request" is more like a shutdown "demand".

Raymond Chen

Follow