COM asynchronous interfaces, part 8: Asynchronous release, the problems

devblogs.microsoft.com/oldnewthing/20220223-00

February 23, 2022

Raymond Chen

It is usually the case that when you release a COM object, you don't particularly care what happens to the object afterward. There are some cases where you do care, usually when there is some cleanup activity associated with the final release that you are relying upon. But usually, you are declaring lack of interest in the future activities of the object when you release.

By default, the `Release` method is synchronous. If the object is remote, then the final[1] release request goes out to the server, and you sit and wait until the server replies that yes, it has definitely released your object. I've run into cases where my code has hung because I'm cleaning up some object that has a reference to a remote object, and the release of the remote object is hung because the server has stopped responding. I really don't care about knowing when the release is completed and would be happy to let the release occur asynchronously. Can you release asynchronously?

Yes, you can release asynchronously. It roughly follows the same pattern we've been using so far, with `IUnknown` as the interface being run asynchronously. However, `AsyncIUnknown` requires extra care because you're messing with the underlying object lifetimes. I'll guide you through the treacherous waters.

Let's make our `SlowPipe` even slower by adding a delay to its destructor.

```
struct SlowPipe :
    winrt::implements<SlowPipe, ::IPipeByte, winrt::non_agile>
{
  // exit the STA thread when we destruct
  ~SlowPipe() {
    Sleep(2000);
    printf("Finally destroyed\n");
    PostQuitMessage(0);
  }

  ...
};
```

We can avoid this slow `Release` by making the release asynchronous. Our initial impulse is to follow the general pattern for asynchronous calls:

```
// Don't use this code. See text.
int main(int, char**)
{
  winrt::init_apartment(winrt::apartment_type::multi_threaded);

  {
    auto pipe = CreateSlowPipeOnOtherThread();

    winrt::com_ptr<::AsyncIUnknown> call;
    auto factory = pipe.as<ICallFactory>();
    winrt::check_hresult(factory->CreateCall(
      __uuidof(::AsyncIUnknown), nullptr,
      __uuidof(::AsyncIUnknown),
      reinterpret_cast<::IUnknown**>(call.put())));

    winrt::check_hresult(call->Begin_Release());

    // force all objects to destruct, to prove we're done
  }
  printf("Getting on with our life.\n");

  Sleep(5000); // just so we can see the release complete

  return 0;
}
```

We are following the general asynchronous pattern: Get the call factory, create an asynchronous call for `IUnknown`, begin the `Release`, and then throw it all away, to indicate that you are not interested in the result.

Unfortunately, this doesn't work.

The first mistake is failing to keep track of all the outstanding references. At the time we call `Begin_Release`, there are three outstanding references to the object: One in `pipe`, another in `factory`, and a third in `call`. That `Begin_Release` is not going to be the final release, so it's just going to decrement the local reference count in the proxy, and nothing will go out over the wire to the remote object. And then when we get around to cleaning up and releasing `pipe`, that's the one that releases the final reference in the proxy, and that's the one that triggers a call to the remote object.

That call is a synchronous call.

So we need to make sure that the only remaining reference to the remote object is in the call object. We can do that by releasing the `pipe` and `factory` references early, prior to their natural destruction.

The next problem is that we performed a `Begin_Release` to initiate an asynchronous `Release` operation, and then the `call` object destructs, which performs its own `Release`. We're performing a double-release of that last reference: One release is asynchronous (explicit call to `Begin_Release`) and the other is synchronous (implicit `Release` at destruction).

Okay, so instead of allowing `call` to destruct naturally, we need to perform a `detach` operation to remove control of the call from the `call` variable. In this case, we're telling the `call` variable, "Don't worry, I'll take care of it." And we took care of it by asking for the release to happen asynchronously.

But things are still not quite right.

You see, the usual pattern of throwing away a call doesn't work for `IUnknown::Release`: Under the usual pattern, the call object normally discovers whether you plan on calling `Finish_` by observing that you released the object. But when we use the async pattern for `IUnknown::Release`, we just throw away the call object *without even calling* `Release`. This leaves the call object in a pickle: "Should I remain valid so the caller can call `Finish_Release`? Or should I just clean up right away?"

The call object for `AsyncIUnknown` plays it safe and assumes you want to call `Finish_Release`. But that means you now *must* call `Finish_Release`.

But wait, calling `Finish_Release` means that we block until the `Release` completes. That brings us back full circle: Our attempt at an asynchronous `Release` resulted in a synchronous wait!

The solution here is to aggregate the call so we can be notified via `ISynchronize::Signal` that the call has completed. At that point, we call `Finish_Release` to complete the call.

Putting this all together will require us to apply a lot of what we've learned about COM aggregation. We'll set to work next time.

[1] Only the final release request goes over the wire. Non-final releases merely decrement the reference count of the local proxy.

Raymond Chen

Follow