COM asynchronous interfaces, part 9: Asynchronous release, assembling a solution

devblogs.microsoft.com/oldnewthing/20220224-00

February 24, 2022

Raymond Chen

Last time, we learned about the complex juggling required in order to accomplish a successful asynchronous release. Let's try to put them together.

One of the things we need to do is aggregate the call object so that we can learn when the call has completed. This tells us when it's safe to call `Finish_Release` and complete the client-side portion of the operation.

```
struct SyncForRelease :
    winrt::implements<SyncForRelease, ISynchronize>
{
  winrt::com_ptr<::IUnknown> m_inner;
  ::AsyncIUnknown* m_asyncUnknown;

  int32_t query_interface_tearoff(winrt::guid const& id, void** object)
    const noexcept override {
    if (m_inner) return m_inner.as(id, object);
    return E_NOINTERFACE;
  }

  auto Sync() noexcept { return m_inner.as<ISynchronize>(); }

  STDMETHODIMP Reset() { return Sync()->Reset(); }
  STDMETHODIMP Signal() {
    auto hr = Sync()->Signal();
    m_asyncUnknown->Finish_Release();
    m_inner.detach(); // don't Release it
    Release(); // I am dead to me
    return hr;
  }
  STDMETHODIMP Wait(DWORD flags, DWORD timeout) {
    return Sync()->Wait(flags, timeout);
  }
};
```

This is the object we're going to use to aggregate the call. This follow the pattern we had seen earlier for aggregating the call object in order to override the `ISynchronize` method, and doing out bonus work inside the `Signal` method.

Actually, if the `Signal` and `Wait` calls fail, we fail to clean up or fail to wait for the operation to complete, and we have nowhere to report the failure. We may as well just fail fast. Instead of trying to catch the exception coming from the `Sync()` method, I just mark it as `noexcept`, which terminates the process if the query fails.

The stuff we do in the `Signal` won't make sense until we understand how things are set up. So let's set them up:

```
void ReleaseAsynchronously(IUnknown* unk)
{
  winrt::com_ptr<::ICallFactory> factory;
  unk->QueryInterface(IID_PPV_ARGS(factory.put()));
  unk->Release();
  if (!factory) return;

  winrt::com_ptr<SyncForRelease> sync;
  try {
    sync = winrt::make_self<SyncForRelease>();
  } catch (std::bad_alloc const&) { }
  if (!sync) return;

  factory->CreateCall(
    __uuidof(::AsyncIUnknown), sync.get(),
    __uuidof(::IUnknown), sync->m_inner.put());
  factory = nullptr;
  if (!sync->m_inner) return;

  sync->m_inner.as(IID_PPV_ARGS(&sync->m_asyncUnknown));
  if (!sync->m_asyncUnknown) return;

  // Release + AddRef cancel out

  sync->m_asyncUnknown->Begin_Release();
}
```

This function guarantees that the incoming `IUnknown` is released, one way or another: If we can't release it asynchronously, then we'll release it synchronously. This makes things easier for the caller, who can treat it as a fire-and-forget type of function.

First, we query the `IUnknown` for `ICallFactory`, and then immediately release the `IUnknown`. If the object is local, then the query will fail, and the `Release` will be a synchronous one. We detect this failure and return: The object has been release synchronously, and we're done.

If the query for `ICallFactory` succeeds, then we have a proxy to a remote object. The release of the `IUnknown` won't destroy the proxy because the `ICallFactory` is still outstanding.

Next up, we create the `SyncForRelease` object, which we will use to aggregate the call so that we can be called back when the asynchronous method completes. We do it inside of a `try` block so we can handle the low-memory case and abandon the operation. The `return` will release the factory, which will be a synchronous release of the proxy. Sorry, we tried.

Assuming we have the `SyncForRelease` object, we ask the factory to create a call (saving it as the aggregated inner object), and then immediately release the factory. This is a repeat of the previous pattern: If the `CreateCall` fails, then the release of the factory is a synchronous release, and we just return immediately. Otherwise, we keep going.

We ask the aggregated inner object for `AsyncIUnknown` so we can call the `Begin_Release` and `Finish_Release` methods. Again, if this fails, we just return, and the destructors will release the proxy synchronously.

Now, the normal pattern for querying an inner object for an interface is to perform the `QueryInterface`, and then perform a counteracting `Release` on the outer object. So in theory, there should be a `Release()` call here.

But we have a trick up our sleeve.

The next step would normally be to call `AddRef()` on the `SyncForRelease` object so that it keeps itself alive while the call is in flight. This `AddRef()` cancels out the `Release()`, so the net result is that we don't have to do anything! We are basically repurposing the reference count created by the `QueryInterface` call.

Now that everything is set up, we perform the `Begin_Release()`, which sets the asynchronous call into motion.

And then we wait.

Eventually, the asynchronous call completes, and the `SyncForRelease::Signal` method is called. After asking the inner object to do the standard signaling work, we proceed with our custom response to the signal. We start by calling `Finish_Release`, which tells the call object that we have acknowledged the release of the object, and once `Finish_Release` returns, the object is truly released. The call to `Finish_Release` will not block because we forwarded the `Signal` call to the inner object, so the call is definitely complete.

When `Finish_Release` returns, the call object has been released, so we must throw away our references to it without calling `Release`. For our raw pointer, we can just abandon it. For our `m_inner` smart pointer, we use `detach()` to take ownership of the pointer. We

just throw the pointer away, because it has already been released by the call to `Finish_Release()`.

It took us a long time to get here, but we finally got it: A function for asynchronously releasing a COM pointer to a remote object.

Bonus chatter: Note in particular that our call to `Sync()->Signal()` was done with a temporary reference to the inner `ISynchronize`, so it got released when `Signal()` returned. If you do some tweaking of this method, make sure that you release the inner `ISynchronize` before calling `Finish_Release()`. Because `Finish_Release()` tears down the inner object, and all references to it become dead.

Raymond Chen

Follow