COM asynchronous interfaces, part 7: Being called directly when the operation completes

devblogs.microsoft.com/oldnewthing/20220222-00

February 22, 2022

Raymond Chen

Last time, we learned how we could wait on the internal event handle that is signaled when an asynchronous call completes. But that's still an indirect discovery of completion. You can register a threadpool wait on the handle, but when your wait callback runs, it's running on the threadpool, and if you were operating in a single-threaded apartment, you'll have to get control back into that apartment (say by using an `IContextCallback`).

But there's a way to tell the COM marshaling infrastructure to call you back directly, and it even respects your object's agility, so if you want to run code in the original apartment, you can do that by providing a non-agile object.

The way to ask for a direct callback is to *aggregate* the call object into your custom outer object. Effectively, you *become* the call object through the magic of aggregation, so all the things that normally are done to the call object are instead done to *you*.

The COM infrastructure uses the `ISynchronize` interface to communicate the state of the call to the call object. If you aggregate the call object, you can take over the responsibilities of `ISynchronize`.

The `ISynchronize` interface models a kernel event handle. The methods are called as follows:

- `ISynchronize::Reset`: COM calls this method when the asynchronous call starts. The idea is that it's resetting the kernel event, to indicate that the call has not completed.
- `ISynchronize::Signal`: COM calls this method when the asynchronous call completes. The idea is that it's setting the kernel event, to indicate that the call is now complete.
- `ISynchronize::Wait`: COM calls this method when the client calls the `Finish_` method, indicating that it wants to wait for the call to complete (if it hasn't completed already). When the `Wait` method returns, COM assumes that the call has completed and returns the answer that was saved in the call object.

You can substitute any other object that follows this same pattern. You don't even have to have a real kernel object. You just need something that can *pretend* to be a kernel object enough to satisfy the `ISynchronize` contract.

```
struct MySynchronize : winrt::implements<MySynchronize, ::ISynchronize>
{
  winrt::com_ptr<::IUnknown> m_inner;
  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;
  }

  wil::slim_event ready;

  STDMETHODIMP Reset() { ready.ResetEvent(); return S_OK; }
  STDMETHODIMP Signal() { ready.SetEvent();
    printf("Call completed!\n"); // do cool stuff here
    return S_OK; }
  STDMETHODIMP Wait(DWORD flags, DWORD timeout) {
    assert(is_mta()); // we won't be pumping messages
    assert(!(flags & COWAIT_ALERTABLE)); // we won't be waiting alertably
    return ready.wait(timeout) ? S_OK : RPC_S_CALLPENDING;
  }

  static bool is_mta() {
    APTTYPE type;
    APTTYPEQUALIFIER qualifier;
    THROW_IF_FAILED(CoGetApartmentType(&type, &qualifier));
    return type == APTTYPE_MTA;
  }
};
```

The `MySynchronize` class starts with one of the common aggregation outer object patterns: It has an inner object (`m_inner`), and we want to aggregate all the interfaces of the inner object. Therefore, our custom `query_interface_tearoff` method forwards *all* interface queries to the inner object.

After that comes our custom implementation of `ISynchronize` . Our version doesn't use a real kernel object. It uses the lightweight event-like object built out of `WaitOnAddress` as provided by the Windows Implementation Library.

One of the tricky parts here is the `Wait` method: Most of the flags relate to how the method should wait if running on an STA. We don't want to deal with any of that nonsense, so we just decide not to support them, nor do we support alertable waits.

Mind you, this decision not to support STA or alertable waits needs to be done in coordination with the clients of the call object. But if you yourself are the client, then you know whether you ever use it from an STA or with an alertable wait. (COM always calls with

`COWAIT_DEFAULT` from the thread that called the `Finish_` method.)

A simpler way is to delegate the `ISynchronize` methods back to the call object:

```
struct MySynchronize :
    winrt::implements<MySynchronize, ::ISynchronize, winrt::non_agile>
{
  winrt::com_ptr<::IUnknown> m_inner;
  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() { return m_inner.as<ISynchronize>(); }

  STDMETHODIMP Reset() { return Sync()->Reset(); }
  STDMETHODIMP Signal() {
    auto hr = return Sync()->Signal();
    printf("Call completed!\n"); // do cool stuff here
    return hr;
  }
  STDMETHODIMP Wait(DWORD flags, DWORD timeout) {
    return Sync()->Wait(flags, timeout);
  }
};
```

Let's take this out for a spin.

```
int main(int, char**)
{
  winrt::init_apartment(winrt::apartment_type::multi_threaded);

  auto pipe = CreateSlowPipeOnOtherThread();

  auto outer = winrt::make_self<MySynchronize>();
  auto factory = pipe.as<ICallFactory>();
  winrt::check_hresult(factory->CreateCall(
    __uuidof(::AsyncIPipeByte), winrt::get_unknown(*outer),
    __uuidof(::IUnknown), outer->m_inner.put()));
  auto call = outer.as<::AsyncIPipeByte>();

  printf("Beginning the Pull\n");
  winrt::check_hresult(call->Begin_Pull(100));

  printf("Doing something else for a while...\n");
  Sleep(100);

  printf("Getting the answer\n");
  BYTE buffer[100];
  ULONG actual;
  winrt::check_hresult(call->Finish_Pull(buffer, &actual));
  printf("Pulled %lu bytes\n", actual);

  return 0;
}
```

When the call completes, the `ISynchronize::Signal` method on the outer object is called, and we can take that opportunity to do some work. Our `MySynchronize` object is marked as non-agile, so this call is made in the same apartment in which it was created, which is convenient if the `Signal` method wants to access other objects with apartment affinity.

Note that we forward the call into the inner object first, before doing our work. That way, our work is done while the event is signaled. If we didn't do that, then if the work calls `Finish_` to get the results of the call that just completed, it will deadlock because the `Finish_` is going to wait for the call to be signaled as complete.

So there's a practical use for COM aggregation: It lets you become part of another object and respond to its methods.

Bonus chatter: I cheated a bit and used a throwing method when forwarding the `ISynchronize` methods. COM methods are not allowed to throw C++ exceptions (because C++ exceptions are not part of the ABI), so we need to convert them back to `HRESULT`s.

```
struct MySynchronize :
    winrt::implements<MySynchronize, ::ISynchronize, winrt::non_agile>
{
  winrt::com_ptr<::IUnknown> m_inner;
  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() { return m_inner.as<ISynchronize>(); }

  STDMETHODIMP Reset() try { return Sync()->Reset(); }
    catch (...) { return winrt::to_hresult(); }
  STDMETHODIMP Signal() try {
    auto hr = return Sync()->Signal();
    printf("Call completed!\n"); // do cool stuff here
    return hr;
  } catch (...) { return winrt::to_hresult(); }
  STDMETHODIMP Wait(DWORD flags, DWORD timeout) try {
    return Sync()->Wait(flags, timeout);
  } catch (...) { return winrt::to_hresult(); }
};
```

Bonus bonus chatter: Note how this differs from *containment*, which is the more usual pattern for combining objects. If the outer object *contained* a call object, then queries on the call object would be satisfied by the call object. The outer object never gets a chance to take over the `ISynchronize`.

Raymond Chen

Follow