COM asynchronous interfaces, part 1: The basic pattern

devblogs.microsoft.com/oldnewthing/20220214-44

February 14, 2022

Raymond Chen

You can mark your marshaled COM interfaces as supporting asynchronous calls, which unlocks a brand new calling pattern. When you attach the `async_uuid` attribute to your interface, COM generates a parallel interface with the same name as the synchronous interface, but with the `Async` prefix. For example, the asynchronous partner of `IMumble` is `AsyncIMumble`. (This is an exception to the general rule that COM interface begin with the letter "I".)

For each method on the synchronous interface, COM creates *two* methods on the asynchronous interface: `Begin_MethodName` and `Finish_MethodName`. The parameters to the `Begin_MethodName` method are all of the `[in]` parameters of the original method, and the parameters to the `Finish_MethodName` method are all of the `[out]` parameters. A parameter that is annotated as `[in, out]` shows up in *both* methods.

The general pattern for making an asynchronous call is

- Query the proxy for `ICallFactory`.
- Call `ICallFactory::CreateCall` with the asynchronous interface you want to call. This produces a *call object*.
- Call the `Begin_MethodName` method on the call object.
- Go do something else for a while, if you like.
- Optionally, check in on the progress of the asynchronous operation by calling `ISynchronize::Wait` on the call object.
- Call the `Finish_MethodName` method to wait for the call to complete and get the results.

Today, we'll kick the tires a bit. In future articles, we'll try out some variations, finishing with a practical application of asynchronous calls that you can use even if your interface isn't marked as asynchronous.

Today's smart pointer library is (rolls dice)[1] C++/WinRT.

```
#include <windows.h>
#include <stdio.h> // Horrors! Mixing C and C++!

struct SlowPipe :
    winrt::implements<SlowPipe, ::IPipeByte, winrt::non_agile>
{
  // exit the STA thread when we destruct
  ~SlowPipe() {  PostQuitMessage(0); }

  STDMETHODIMP Pull(BYTE* buffer, ULONG size, ULONG* written)
  {
    printf("Pulling %lu bytes...\n", size);
    ULONG index;
    for (index = 0; index < size / 2; index++) {
      Sleep(100);
      buffer[index] = 42;
      printf("Pulled byte %lu of %lu\n", index, size);
    }
    *written = index;
    printf("Finished pulling %lu% of %lu bytes\n", index, size);
    return S_OK;
  }

  STDMETHODIMP Push(BYTE* buffer, ULONG size)
  {
    printf("Pushing %lu bytes...\n", size);
    ULONG index;
    for (index = 0; index < size; index++) {
      Sleep(100);
      printf("Pushed byte %08x\n", buffer[index]);
    }
    printf("Finished pushing %lu bytes\n", size);
    return S_OK;
  }
};
```

Our `SlowPipe` object is a pipe that is slow, taking 100ms for each byte. C++/WinRT objects are agile by default, but we mark ours as `winrt::not_agile` to override this, thereby forcing the interface to be marshaled through a proxy. Just for fun, our `Pull` method pulls only half of the bytes requested.

Let’s create a thread that hosts our `SlowPipe` object.

```
struct CreateSlowPipeInfo
{
  winrt::agile_ref<::IPipeByte> agile;
  bool ready = false;
};


DWORD CALLBACK ThreadProc(void* p)
{
  winrt::init_apartment(winrt::apartment_type::single_threaded);

  auto& info = *reinterpret_cast<CreateSlowPipeInfo>(p);
  info.agile = winrt::make<SlowPipe>();
  info.ready = true;
  WakeByAddressSingle(&info.ready);

  MSG msg;
  while (GetMessage(&msg, nullptr, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessageW(&msg);
  }

  winrt::uninit_apartment();
  return 0;
}
```

The thread is given a pointer to a `CreateSlowPipeInfo` structure. We initialize COM in single-threaded mode on the thread, create a `SlowPipe` object, and store an agile reference to that object in the thread creator-provided structure. We then let the thread creator know that the agile reference is ready. And then we process messages until we get a quit message, which will happen when the `SlowPipe` is destructed.

So let’s write the code that creates the thread and gets the `SlowPipe` from that thread.

```
winrt::com_ptr<::IPipeByte> CreateSlowPipeOnOtherThread()
{
    CreateSlowPipeInfo info;
    auto nope = info.ready;

    DWORD id;
    winrt::handle(winrt::check_pointer(
        CreateThread(0, 0, ThreadProc, &info, 0, &id)));

    while (!info.ready) {
      WaitOnAddress(&info.ready, &nope, sizeof(nope), INFINITE);
    }
    return info.agile.get();
}
```

We create the thread with the `CreateSlowPipeInfo` and wait for it to signal that it's ready, at which point we convert the agile reference to a `com_ptr<IPipeByte>` so we can use it locally.

Okay, now on to the main event:

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

  auto pipe = CreateSlowPipeOnOtherThread();

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

  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;
}
```

Once we create the pipe on another thread and marshal it back to the main thread, we make an asynchronous call to the `Pull` method.

First step: Get the factory, which we do by querying the proxy for `ICallFactory`.

Second step: Create a new call. The parameters to `CreateCall` take a little time to get used to.

- First is the asynchronous interface you want to call.
- Second is the controlling unknown. You usually pass `nullptr` here, but we'll see later how you can replace this to get special effects.
- Third and fourth are the usual pair of an interface ID and an output pointer. Somewhat cumbersome here is that the final parameter is typed as `IUnknown**` rather than the traditional `void**`, which means you can't use the usual `IID_PPV_ARGS` pattern.

The call object itself supports the following interfaces:

- `IUnknown` because all COM objects support `IUnknown`.
- `AsyncIMumble`, the asynchronous interface you are calling.
- `ISynchronize`, which we'll learn about later.
- `ISynchronizeHandle`, which we'll learn about later.
- `ICancelMethodCalls`, which we'll learn about later.
- `IClientSecurity`, which you can use to customize the security of the marshaled call. I'm not going to go into this part.

You definitely are going to need the `AsyncIMumble`, seeing as that's how you're going to make the asynchronous call in the first place. The other interfaces might or might not be useful, depending on the scenario.

We call the `Begin_Pull` method with the input parameters to initiate the pull operation. This call goes out to the helper thread, but since we are calling asynchronously, the call to `Begin_Pull` returns immediately, while the call gets delivered to the other thread for execution.

We pretend to do some other work for a while, and then come back and call `Finish_Pull` method to get the answer. This is a blocking call that waits for the operation to complete, and then propagates the unmarshaled output parameters and `HRESULT`.

That's the basics of COM asynchronous calls. Next time, we'll start getting a little fancier.

[1] The dice are loaded.

Raymond Chen

Follow