Using contexts to return to a COM apartment later

devblogs.microsoft.com/oldnewthing/20191129-00

November 29, 2019

Raymond Chen

We've been looking at COM contexts lately, and so far all of these COM contexts were custom contexts created for the purpose of being able to bulk-disconnect all objects in them. But there are also the COM contexts that COM creates automatically for you, and those are also interesting.

Each apartment has a COM context object, and you can access it by calling the `CoGet-ObjectContext` function. You can then use the `IContextCallback:: ContextCallback` to get back to that context.

In other words, you can capture the current context and return to it any time you like. This can be used if you want to get the effect of marshaling, but when the thing you want to marshal isn't a COM object.

```
void StartSave(std::function<void(bool)> saveComplete)
{
  start_save().on_completed(
    [saveComplete = std::move(saveComplete)](bool result)
  {
    saveComplete(result);
  });
}
```

This version of `StartSave` starts the *save* operation, and when the save is complete, it calls the `std::function` with the result. The callback could happen on any thread, but the `std::function` may have captured objects that have COM apartment affinity, like references to other COM apartment-affine objects.

We can update the `StartSave` function so that the `saveComplete` is invoked in the same apartment that initiated the `StartSave` operation.

```
auto CaptureCurrentApartmentContext()
{
  winrt::com_ptr<IContextCallback> context;
  check_hresult(CoGetObjectContext(IID_PPV_ARGS(context.put())));
  return context;
}

void StartSave(std::function<void(bool)> saveComplete)
{
  start_save().on_completed(
    [saveComplete = std::move(saveComplete),
     context = CaptureCurrentApartmentContext()](bool result)
  {
    InvokeInContext(context.Get(), [&]()
    {
      saveComplete(result);
    });
  });
}
```

This trick is useful if you have an object that was created on a UI thread and must be destructed on that same UI thread, but you also capture a strong reference to the object so it can be used by background threads. If the background thread is the one that releases the last strong reference, the object will be destructed on the background thread. To fix that, you can make the destructor run on the UI thread.

```
// Error checking elided for expository purposes.
class MyThing
{
  Microsoft::WRL::ComPtr<IContextCallback> m_context;
  MyThing()
  {
    CoGetObjectContect(IID_PPV_ARGS(&m_context));
  }

  ...
  ULONG Release()
  {
      LONG refCount = InterlockedDecrement(&m_refCount);
      if (refCount == 0) {
        // Normally, we would do a "delete this",
        // but we will go through the ContextCallback to ensure
        // that the deletion happens on the correct thread.
        InvokeInContext(m_context.Get(), [this]()
        {
          delete this;
        });
      }
      return refCount;
  }
};
```

More generally, you may have a C++ object that has UI thread affinity, but you want to kick off some background work, and when the background work is complete, it wants to switch back to the UI thread to finish the work. You can capture the `IContextCallback` on the UI thread, and then use the `IContextCallback` to get back to the UI thread when you're ready.

Another case where you would want to return to an earlier context is in the case of a coroutine. By default, in C#, `await` operations resume execution in the same context that performed the `await`. In C++, you can accomplish this by capturing the `IContext-Callback` at the point of the `co_await` and then resume execution inside that same context. This is how C++/WinRT makes `co_await` on `IAsyncAction` and `IAsync-Operation` objects resume execution in the same thread context.

Capturing the current `IContextCallback` gives you a way to "go back home again": You can use it to get back to the thread context at some future point.

Raymond Chen

Follow