Inside C++/WinRT: Coroutine completions: Cancellation propagation

devblogs.microsoft.com/oldnewthing/20230203-00

February 3, 2023

Raymond Chen

If you enable C++/WinRT cancellation propagation, and somebody asks to cancel your coroutine while you are awaiting another coroutine, the propagation infrastructure will try to cancel the coroutine you are awaiting.

We add support for propagating cancellation into a Windows Runtime asynchronous operation, we have our awaiter derive from `winrt::enable_await_cancellation` and implementing a canceller.

```
template <typename Async>
struct await_adapter : enable_await_cancellation
{
    ⟦ ... ⟧

    void enable_cancellation(cancellable_promise* promise)
    {
        promise->set_canceller([](void* parameter)
        {
            cancel_asynchronously(
                reinterpret_cast<await_adapter*>(parameter)
                    ->async);
        }, this);
    }

    ⟦ ... ⟧

    static fire_and_forget cancel_asynchronously(Async async)
    {
        co_await winrt::resume_background();
        try
        {
            async.Cancel();
        }
        catch (hresult_error const&)
        {
        }
    }
    ⟦ ... ⟧
};
```

To support cancellation, we derive from `winrt::enable_await_cancellation` and implement the `enable_cancellation` method. This method is given a `cancellable_promise`, and its job is to call the `set_canceller` method to tell the promise how to cancel this awaitable. The information takes the form of a function pointer (usually a captureless lambda that converts to a function pointer) and a context pointer (usually `this`).

If the coroutine is an `IAsyncAction` or `IAsyncOperation`, then it will support a `Cancel()` method, and upon cancellation, the C++/WinRT infrastructure will call the canceller to cancel the thing being awaited.

If we are awaiting another Windows Runtime asynchronous operation, what our canceller does is hop to a background thread and then try to cancel the thing we are awaiting. This is a best-effort operation, so we ignore any errors. (For example, we could encounter a race condition where the operation completes just as we’re about to cancel it.)

Note that cancellers can be called multiple times, so if you’re writing your own custom canceller, make sure it is harmless to call a second time.

Supporting cancellation is a bit awkward because it is designed for efficiency: Cancellation is extremely rare, so we want to make preparing for cancellation cheap, and are okay with the actual work of cancellation being expensive.

Okay, that was a whirlwind tour of how C++/WinRT implements `co_await` for Windows Runtime asynchronous operations. It's not that complicated, but it looks intimidating when broken down into pieces.[1]

The overall goal of the "Inside C++/WinRT" series is to share the knowledge of how C++/WinRT works, so that more people can pitch in when somebody has a question.

[1] It's similar to explaining a game to someone. When you play the game, the rules are simple: "You hit the ball back and forth, and the loser is the person who fails to hit the ball." However, when you try to explain the game to someone else, it becomes surprisingly complicated: "Well, except you don't have to hit the ball under these conditions, and sometimes hits don't count, and sometimes you're allowed to hit twice, and, gosh, now that I'm explaining it in detail, it doesn't sound simple at all."