How to get your C++/WinRT asynchronous operations to respond more quickly to cancellation, part 1

devblogs.microsoft.com/oldnewthing/20200722-00

July 22, 2020

Raymond Chen

C++/WinRT provides an implementation for Windows Runtime asynchronous actions and operations, and they even support cancellation, even if your code doesn't realize it.

Whenever[1] your coroutine performs a `co_await`, the C++/WinRT library checks whether the coroutine has already been cancelled.[2] If so, then it abandons the coroutine and goes to the `Canceled` state.

```
IAsyncAction ProcessAllWidgetsAsync()
{
    auto widgets = co_await GetAllWidgetsAsync();
    for (auto&& widget : widgets) {
        ProcessWidget(widget);
    }
    co_await ReportStatusAsync(WidgetsProcessed);
}
```

This function gathers all the widgets and then processes them one by one. But say there are thousands of widgets, and you try to `Cancel` the operation:

```
IAsyncAction DoOperationAsync()
{
    // Remember the operation so we can cancel it.
    operation = ProcessAllWidgetsAsync();

    co_await operation;
}

void CancelOperation()
{
    operation.Cancel();
}
```

When the `Cancel` is called, the C++/WinRT library remembers that the coroutine has been cancelled and looks for a chance to stop the coroutine. But right now, the coroutine is busy running the loop inside `ProcessAllWidgets`, and the C++/WinRT library doesn't get

control until the `co_await` when it comes time to report the status. Once that happens, the coroutine stops executing and reports its cancellation.[3]

That could be hours from now.

You can hasten the cancellation process in your coroutine by polling for cancellation.

```
IAsyncAction ProcessAllWidgetsAsync()
{
    auto cancellation = co_await get_cancellation_token();

    auto widgets = co_await GetAllWidgetsAsync();
    for (auto&& widget : widgets) {
        if (cancellation()) co_return;
        ProcessWidget(widget);
    }
    co_await ReportStatusAsync(WidgetsProcessed);
}
```

The `co_await get_cancellation_token()` produces a cancellation token for the current coroutine.[4]

Before processing each widget, we check if we have been cancelled. If so, then we just give up immediately. The `co_return` is another point where the C++/WinRT library regains control, and that also processes the pending cancellation.

But wait, what if the caller tries to cancel the operation while the `GetAllWidgetsAsync` is in progress? Control is now inside that other asynchronous operation, and it could take a very long time to get all of the widgets. Next time, we'll look at how to propagate the cancellation into dependent coroutines.

[1] There are a few exceptions to this rule, but it's true enough.

[2] I spell *cancelled* with two L's.

[3] This highlights the importance of using RAII types for all of your cleanup. If the coroutine stops executing due to cancellation, then its automatic objects are destructed according to the usual rules of C++, and that's where your abnormal cleanup happens. We'll talk more about this soon.

[4] The `co_await get_cancellation_token()` is one of the exceptions to the rule that `co_await` always checks for cancellation. In this case, `co_await get_cancellation_token()` doesn't actually "await" anything. Rather, it's a backdoor into the C++/WinRT library. We'll learn more about how these backdoors work when we look at how to implement your own coroutines in C++20, at some unspecified point in the future.

Raymond Chen

Follow