C++/WinRT doesn't let your coroutine cheat death, but it does get to say good-bye

devblogs.microsoft.com/oldnewthing/20200724-00

July 24, 2020

Raymond Chen

I noted in a footnote that it's important to use RAII types for all of your cleanup in a coroutine because cancellation will cause the coroutine to abruptly stop executing at the next suspension point. C++/WinRT doesn't let your coroutine cheat death, but it does get to say good-bye.

C++/WinRT reports cancellation to the coroutine by throwing an `hresult_canceled` exception in the context of the coroutine itself. This takes advantage of the existing coroutine machinery to stow exceptions in the asynchronous operation so they can be observed by the code that is awaiting the result.

But it also means that you can see the exception fly past as you proceed inexorably to your death.

```
IAsyncOperation<int> DoSomethingAsync()
{
    try {
        /* do stuff */
        co_return 42;
    } catch (hresult_canceled const&) {
        printf("Pardonnez-moi, monsieur, je ne l'ai pas fait exprès.");
        throw;
    }
}
```

You can catch the `hresult_canceled` exception and perhaps do some final cleanup. When you're done, you rethrow the exception to allow normal cleanup of the coroutine to proceed.

Of course, if your intention is to perform actual cleanup (rather than merely making a statement for posterity),[1] then you'll need to catch *all* exceptions, not just cancellation.

```
IAsyncOperation<int> DoSomethingAsync()
{
    ClaimWidget();
    try {
        /* do stuff */
        co_return 42;
    } catch (...) {
        DisclaimWidget();
        throw;
    }
}
```

At this point, you may as well just use an RAII type.

```
struct ScopedWidgetClaim
{
    ScopedWidgetClaim() { ClaimWidget(); }
    ~ScopedWidgetClaim() { DisclaimWidget(); }

    // non-copyable, non-assignable
    ScopedWidgetClaim(ScopedWidgetClaim const&) = delete;
    void operator=(ScopedWidgetClaim const&) = delete;
};

IAsyncOperation<int> DoSomethingAsync()
{
    ScopedWidgetClaim claim;
    /* do stuff */
    co_return 42;
}
```

Maybe you think you can cheat death by swallowing the exception instead of rethrowing it?

```
IAsyncOperation<int> DoSomethingAsync()
{
    try {
        /* do stuff */
        co_return 42;
    } catch (hresult_canceled const&) {
        co_return 0;
    }
}
```

What happens here depends on where the cancellation exception was generated. If it was the result of a dependent coroutine failing with cancellation, then you successfully caught the failed dependent coroutine and can recover:

```
IAsyncOperation<int> GetWidgetCountAsync()
{
    try {
        auto widgets = co_await GetAllWidgetsAsync();
        co_return widgets.Count();
    } catch (hresult_canceled const&) {
        co_return 0;
    }
}
```

If the `GetAllWidgetsAsync` operation was cancelled (say, because it prompted the user for permission, and the user hit *Cancel*), then the `co_await GetAllWidgetsAsync()` will throw the cancellation exception, and you caught it and dealt with the problem by returning a widget count of zero.

On the other hand, if the cancellation exception was generated because somebody cancelled *you*, then your coroutine has already been put into the cancelled state. It's too late to `co_return` a value; the coroutine has already been cancelled. You can still do the `co_return`, but the value you return won't be the result of the coroutine.

Bonus chatter: C++/WinRT does not provide a way to rescue a coroutine from cancellation. You might think of adding a feature like *co_await uncancel_current_coroutine()* so that a coroutine could treat cancellation as a way to abandon an operation and generate partial results, but C++/WinRT doesn't do that. Once it's cancelled, it's cancelled.

In theory, it could be possible to add "uncancellation", but it would come at the expense of responsiveness: Right now, when an `IAsyncXxx` is cancelled, the operation immediately transitions to the cancelled state. The completion handler is called immediately, allowing the code that is awaiting the result of the coroutine to proceed without delay. If operations could be uncancelled, then the transition to the cancelled state would have to be delayed until the coroutine itself decided whether to accept the cancellation or to uncancel it.

C++/WinRT opted not to slow down the common case in order to add a feature that would rarely be used.

[1] The technical term for "making a statement for posterity" is *logging*.

Raymond Chen

Follow