Creating an already-completed asynchronous activity in C++/WinRT, part 2

devblogs.microsoft.com/oldnewthing/20240710-00

July 10, 2024

Raymond Chen

Last time, we tried to create an already-completed asynchronous activity in C++/WinRT. We were able to create a coroutine that represented a successful already-completed operation:

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    co_return 42;
}
```

But the analogous function for creating a failed already-completed operation didn't work because its lack of any `co_await` or `co_return` statement means that it wasn't a coroutine at all!

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    throw winrt::hresult_access_denied();
}
```

To make the function a coroutine, we need to put a `co_await` or `co_return` in the body somewhere. We have a few options.

One is to put the `co_await` after the `throw`, so it is physically present in the function body (thereby making it a coroutine), but is unreachable. A safe thing to await is `std::suspend_never()`, which is a built-in awaitable that never awaits.

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    throw winrt::hresult_access_denied();
    co_await std::suspend_never();
}
```

Perhaps a more reasonable thing is to actually try to `co_return` something.

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    throw winrt::hresult_access_denied();
    co_return 0;
}
```

Both of these work, and it appears that the major compilers (gcc, clang, msvc) all optimize out the code that follows a `throw`. And as of this writing, they also do not raise a dead code diagnostic, though that's the thing that worries me: It's possible that a future version of the compiler will decide to produce dead code diagnostics, and then this code will cause problems when used in code bases that treat warnings as errors.

We can put a `co_await` in front of the `throw`, but again, if we put it in a dead code block, we risk a diagnostic:

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    if (false) co_return 0;
    throw winrt::hresult_access_denied();
}
```

So maybe the thing to do is actually `co_await` something, but await something that does nothing. That's where we can use the built-in `suspend_never`.

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    co_await std::suspend_never{};
    throw winrt::hresult_access_denied();
}
```

The major compilers recognize this pattern, and it's not dead code, so we don't risk an unreachable code diagnostic.

But wait, it's still a problem, thanks to our pal `await_transform`. The C++/WinRT implementation of `await_transform` for Windows Runtime `IAsyncXxx` interfaces wraps all awaitables inside an awaiter that check whether the `IAsyncXxx` has been cancelled and throws an `hresult_canceled` exception if so.

Now, we know that the coroutine is never cancelled before it completes, but the compilers' escape analysis can't see that, so in practice, they will include that extra check. The `co_await std::suspend_never{}` cannot be optimized out entirely.

At this point, you have to go looking for a rabbit to pull out of your hat. And in this case, the rabbit is `winrt::cancellation_token`.

The `winrt::cancellation_token` is a sentinel object that is not generally awaitable, but C++/WinRT's implementation of `IAsyncXxx` recognizes it as as a special awaitable in its `await_transform` and (here's where the rabbit comes from) the custom awaiter returns an object (which is just a wrapper around a pointer to the promise) without any other fanfare.

The trick, therefore, is to await the cancellation token and discard the resulting object.

```
winrt::Windows::Foundation::IAsyncOperation<int>
    ComputeAsync()
{
    (void)co_await winrt::get_cancellation_token();

    throw winrt::hresult_access_denied();
}
```

We explicitly discard the result of the `co_await`, just in case some future version of C++/WinRT adds the `[[nodiscard]]` attribute.

Now that we know how to code up this specific case, we'll work on generalizing it next time.