Creating an already-completed asynchronous activity in C++/WinRT, part 4

devblogs.microsoft.com/oldnewthing/20240712-00

July 12, 2024

Raymond Chen

Last time, we created a generalized `MakeCompleted` for creating an already-completed asynchronous activity in C++/WinRT. Today, we'll try to do the same for `MakeFailed`.

In one sense, `MakeFailed` is easier because the body is the same regardless of which of the four types of asynchronous activities you're making.

```
winrt::Windows::Foundation::IAsyncAction
MakeFailedAsyncAction(winrt::hresult_error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}

template<typename Progress>
winrt::Windows::Foundation::IAsyncActionWithProgress<Progress>
MakeFailedAsyncActionWithProgress(winrt::hresult_error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}

template<typename Result, typename Progress>
winrt::Windows::Foundation::IAsyncOperation<Result>
MakeFailedAsyncOperation(winrt::hresult_error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}

template<typename Result, typename Progress>
winrt::Windows::Foundation::IAsyncOperationWithProgress<Result, Progress>
MakeFailedAsyncOperationWithProgress(winrt::hresult_error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}

// Sample usage
winrt::Windows::Foundation::IAsyncOperation<int>
GetSizeAsync()
{
    return MakeFailedAsyncOperation<int>(
        winrt::hresult_not_implemented());
}
```

This easily generalizes:

```
template<typename Async>
Async MakeFailed(winrt::hresult_error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}
```

Unfortunately, it’s also wrong.

Since we receive the exception in the form of a `hresult_error` by value, anybody who passes a derived exception like `hresult_access_denied` will be subjected to slicing. You might think to avoid slicing by taking the parameter by reference, but that runs into two problems. First, you are carrying a reference across a `co_await`, which may cause static analysis tools to get upset at you. Worse is that receiving the parameter as a reference doesn't resolve the slicing because the `throw` throws only the `hresult_error` portion of the exception. Fortunately, this second problem is a non-issue in this specific case because it is captured and reported at the ABI to the coroutine consumer as an `HRESULT` failure code from the `GetResults()` method, and the consumer will realize that the `HRESULT` is, say, `E_INVALIDARG` and reconstruct a `hresult_invalid_argument`.¹

The real problem is that not all C++/WinRT exceptions derive from `hresult_error`. What if somebody wants to make a `IAsyncAction` that failed with `std::bad_alloc`?

So we'll accept anything that C++/WinRT supports, which is anything derived from `std::exception` or `winrt::hresult_error`.

```
template<typename Async, typename Error,
    typename = std::enable_if_t<
        std::is_base_of_v<std::exception, Error> ||
        std::is_base_of_v<winrt::hresult_error, Error>>>
Async MakeFailed(Error error)
{
    (void) co_await winrt::get_cancellation_token();
    throw error;
}
```

When dealing with coroutines or exceptions (and especially in our case here, which is coroutines *and* exceptions), you need to think about debuggability. Exceptions are nonlocal transfer, so it can be difficult to debug where an exception originated. And coroutines break up function execution into chunks, and most of the chunks run after the original call stack is lost, so that makes it even more frustrating. We'll look into this next time.

¹ Related reading: What happened to the custom exception description I threw from a C++/WinRT `IAsyncAction`? How come my custom exception message is lost when it is thrown from a `IAsyncAction^`?