Creating an already-completed asynchronous activity in C++/WinRT, part 5

devblogs.microsoft.com/oldnewthing/20240715-00

July 15, 2024

Raymond Chen

Last time, we created a generalized `MakeFailed` for creating an already-completed asynchronous activity in the failure state. We left with a note that debuggability still needs to be investigated. So let's investigate it.

When the Watson post-mortem debugger captures a crash dump, one of the tools you can use to understand the source of the exception is the stowed exception information. This is information captured at the point the exception is raised, which is helpful because the exception may get caught and re-raised, and you want to see the place the originally threw the exception, not the place the did the final uncaught re-raise.

There are two ways to throw an `hresult_error` object.

- One is to construct a `hresult_error` normally and throw it. This captures a stack trace at the point of construction.
- The other way is to use `winrt::throw_hresult`. This recaptures the track trace left behind by a previous error.

We are originating the exception, so we want to construct a `hresult_error` at the point we create the failed asynchronous activity, so that the post-mortem stack trace will show the stack that led up to our error. Now, in our case, the entire coroutine body runs synchronously since there are no suspending `co_await`s, so the difference doesn't really matter much, aside from a little extra stack clutter in the post-mortem debugger. But if we wanted something like a delayed 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 MakeDelayedFailed(Error error,
                        winrt::Windows::Foundation::TimeSpan delay)
{
    (void) co_await winrt::resume_after(delay);
    throw error;
}
```

we want the debugging stack to tell us that the exception came from whoever called `Make-DelayedFailed()`, rather than telling us that the exception came from the thread pool.

This means that we were correct to accept exception object as a by-value parameter. That way, it has already been constructed and therefore has already captured a stack trace.

There's another case we haven't dealt with, though: Propagating an exception into the failed action.

```
winrt::IAsyncAction SetNameAsync(winrt::hstring const& name)
{
    try {
        SetName(name);
        return MakeCompletedAsyncAction();
    } catch (...) {
        return MakeFailed<
            winrt::Windows::Foundation::IAsyncAction>
            (⟦???⟧);
    }
}
```

We want to create a failed `IAsyncAction` that contains whatever exception was thrown by `SetName`, but how do you pass the caught `...` as a variable? Not all C++/WinRT exceptions derive from `hresult_error`, so it's not enough to just catch `winrt::hresult_error`.

The C++ standard library has a `exception_ptr` class which represents an arbitrary exception. This is what `std::current_exception()` returns, and you can throw whaever that `exception_ptr` represents by calling `std::rethrow_exception()`. exception. So we can add an overload of `MakeFailedAsyncAction` that takes an `exception_ptr`:

```
template<typename Async>
Async MakeFailed(std::exception_ptr ptr)
{
    (void) co_await winrt::get_cancellation_token();
    std::rethrow_exception(ptr);
}
```

We can use this overload from a `catch(...)` clause:

```
winrt::IAsyncAction SetNameAsync(winrt::hstring const& name)
{
    try {
        SetName(name);
        return MakeCompletedAsyncAction();
    } catch (...) {
        return MakeFailed<
            winrt::Windows::Foundation::IAsyncAction>
            (std::current_exception());
    }
}
```

This has the advantage of preserving the stack trace from the exception thrown by `Set-Name()`, so that your post-mortem debugging gets a stack trace that leads to whatever point inside `SetName()` triggered the exception, rather than just getting a stack trace that points to our `catch` clause.

In fact, we can use the `std::exception_ptr` as our common currency for exceptions.

```
template<typename Async, typename Error>
Async MakeFailed(Error&& error)
{
    return MakeFailed<Async>(
        std::make_exception_ptr(
            std::forward<Error>(error)));
}
```

Another problem is that it’s annoying having to write out the template type parameter all the time:

```
winrt::Windows::Foundation::IAsyncAction
    SaveAsync()
{
    return MakeFailed<
        winrt::Windows::Foundation::IAsyncAction>
        (winrt::hresult_access_denied());
}
```

We’ll look at that next time.