Creating an already-completed asynchronous activity in C++/WinRT, part 8

devblogs.microsoft.com/oldnewthing/20240718-00

July 18, 2024

Raymond Chen

Last time, we created an already-completed `IAsyncAction` by implementing interface directly. Now we'll generalize this cover all four Windows Runtime asynchronous activities.

We start with a traits class that tells us the various pieces that go into each Windows Runtime asynchronous interface.

```
template<typename Async>
struct winrt_async_traits;

template<>
struct winrt_async_traits<
    winrt::Windows::Foundation::IAsyncAction>
{
    using Async = winrt::Windows::Foundation::IAsyncAction;
    static constexpr bool has_progress = false;
    using Progress = int32_t;
    using Result = std::monostate;
    using CompletedHandler =
        winrt::Windows::Foundation::AsyncActionCompletedHandler;
    using ProgressHandler = winrt::IInspectable;
};

template<typename P>
struct winrt_async_traits<
    winrt::Windows::Foundation::IAsyncActionWithProgress<P>>
{
    using Async = winrt::Windows::Foundation::IAsyncActionWithProgress<P>;
    static constexpr bool has_progress = true;
    using Progress = P;
    using Result = std::monostate;
    using CompletedHandler =
        winrt::Windows::Foundation::AsyncActionWithProgressCompletedHandler<P>;
    using ProgressHandler =
        winrt::Windows::Foundation::AsyncActionProgressHandler<P>;
};

template<typename T>
struct winrt_async_traits<
    winrt::Windows::Foundation::IAsyncOperation<T>>
{
    using Async = winrt::Windows::Foundation::IAsyncOperation<T>;
    static constexpr bool has_progress = false;
    using Progress = int32_t;
    using Result = T;
    using CompletedHandler =
        winrt::Windows::Foundation::AsyncOperationCompletedHandler<T>;
    using ProgressHandler = winrt::IInspectable;
};

template<typename T, typename P>
struct winrt_async_traits<
    winrt::Windows::Foundation::IAsyncOperationWithProgress<T, P>>
{
    using Async = winrt::Windows::Foundation::IAsyncOperationWithProgress<T, P>;
    static constexpr bool has_progress = true;
    using Progress = P;
    using Result = T;
    using CompletedHandler =
```

```
        winrt::Windows::Foundation::AsyncOperationWithProgressCompletedHandler<T, P>;
    using ProgressHandler =
        winrt::Windows::Foundation::AsyncOperationProgressHandler<T, P>;
};
```

I use `monostate` as placeholders for the cases where the interface doesn't have a result or a progress handler. I could have used `void`, but I'm picking an instantiatable type so I can use it as the type of member variables and parameters below. I could have tried to compress them out, but they aren't going to be large, and it allows for COMDAT folding of the various specializations since all the members lie at the same offsets.

```
template<typename D, typename Async>
struct completed_async_base
{
    using Traits = winrt_async_traits<Async>;
    using CompletedHandler = typename Traits::CompletedHandler;
    using ProgressHandler = typename Traits::ProgressHandler;

    auto outer() { return static_cast<D*>(this); }

    winrt::slim_mutex m_mutex;
    CompletedHandler m_completed;
    ProgressHandler m_progress;

    auto Id() { return 1; }
    // Status and Error code implemented by D
    auto Cancel() { }
    auto Close() { }

    auto Completed()
    {
        winrt::slim_lock_guard lock(m_mutex);
        return m_completed;
    }

    auto Completed(CompletedHandler const& handler)
    {
        {
            winrt::slim_lock_guard lock(m_mutex);
            if (m_completed) {
                throw winrt::hresult_illegal_delegate_assignment();
            }
            m_completed = handler;
        }
        handler(*outer(), outer()->Status());
    }

    auto Progress()
    {
        winrt::slim_lock_guard lock(m_mutex);
        return m_progress;
    }

    auto Progress(ProgressHandler const& handler)
    {
        winrt::slim_lock_guard lock(m_mutex);
        m_progress = handler;
    }
};
```

The `completed_async_base` takes care of most of the paperwork of being a Windows Runtime asynchronous interface. We defer the status reporting to our derived class because we're going to use this both for successful and failed asynchronous activities. Some of the

interfaces don't have a `Progress` property, in which case our implementations here will never be instantiated.

```
template<typename Async>
struct completed_async :
    winrt::implements<completed_async<Async>,
        Async,
        winrt::Windows::Foundation::IAsyncInfo>,
    completed_async_base<completed_async<Async>, Async>
{
    using Traits = typename completed_async::Traits;
    using Result = typename Traits::Result;

    completed_async() = default;
    completed_async(Result const& value) : m_result(value) {}

    Result m_result = winrt_empty_value();

    auto Status() { return winrt::Windows::Foundation::AsyncStatus::Completed; }
    auto ErrorCode() { return S_OK; }

    auto GetResults() {
        if constexpr (Traits::has_progress) {
            return m_result;
        } else {
            winrt::slim_lock_guard lock(m_mutex);
            return std::move(m_result);
        }
    }
};
```

For a successfully-completed asynchronous activity, we return a status of `Completed`, an error code of `S_OK`, and a result consisting of the value we were given at construction.

In the case of an `IAsyncAction` or `IAsyncActionWithProgress`, there is no `Result`, but we have a dummy `monostate` variable to save myself a bunch of annoying typing. I could have used `bool`, but I chose `monostate` because it is not a legal template argument to `IAsyncOperation`, so you can't accidentally synthesize a successful `IAsyncOperation` without also providing a value. Thanks to the magic of CRTP, the `monostate` returned by `GetResults()` is discarded by the projection.

It is legal to get the results of a `-WithProgress` interface multiple times, so we return a copy for progress interfaces, but move the result out if it is a one-shot.

But wait, moving the value out requires us to take a lock, so that we don't have the case that two threads both try to move the value out at the same time. We can avoid the mutex by always returning a copy.

```
    auto GetResults() {
        return m_result;
    }
```

Whether this is a net benefit depends on the relative cost between a mutex and a copy of the result type.

The other half of the story is the failed activity.

```
template<typename Async>
struct failed_async :
    winrt::implements<failed_async,
        Async,
        winrt::Windows::Foundation::IAsyncInfo>,
    completed_async_base<failed_async<Async>, Async>
{
    using Traits = typename completed_async::Traits;

    failed_async(std::exception_ptr const& exception) :
        m_exception(exception) {}

    std::exception_ptr m_exception;

    auto Status() { return AsyncStatus::Error; }
    auto ErrorCode() {
        try { GetResults(); }
        catch (...) { return winrt::to_hresult(); }
    }

    [[noreturn]] void GetResults() {
        std::rethrow_exception(m_exception);
    }
};
```

The failed variant is simpler. All it needs to remember is an `exception_ptr`, which it uses to extract the error code, and which it rethrows when the results are requested.

We can use our earlier trick of holding the information in a proxy object and producing the interface by conversion. Rename `completed_async` and `failed_async` to `completed_async_impl` and `failed_async_impl`, respectively, and define these proxy classes:

```
template<typename...Result>
struct completed_async
{
    static_assert(sizeof...(Result) <= 1);
    std::tuple<std::decay_t<Result>...> m_result;

    completed_async(Result&&... result) :
        m_result(std::forward<Result>(result)...) {}

    template<typename Async>
    Async as() {
        return std::make_from_tuple<
            completed_async_impl<Async>(
                std::move(m_result));
    }

    template<typename Async>
    operator Async() { return as<Async>(); }
};

template<typename Error>
struct failed_async
{
    std::exception_ptr m_exception;

    failed_async(std::exception_ptr const& exception) :
        m_exception(exception) {}

    template<typename Error>
    failed_async(Error&& error) :
        m_exception(
            std::make_exception_ptr(
                std::forward<Error>(Error))) {}

    template<typename Async>
    Async as() {
        return failed_async_impl<Async>(
            m_exception);
    }

    template<typename Async>
    operator Async() { return as<Async>(); }
};
```

Next time, we'll cheat a little bit.