On awaiting a task with a timeout in C++/WinRT

devblogs.microsoft.com/oldnewthing/20220506-00

May 6, 2022

Raymond Chen

Last time, we studied ways of awaiting a task with a timeout in C#. Now we'll apply what we learned to C++/WinRT.

C++/WinRT already has a `when_any` function which completes as soon as any of the provided coroutines completes, so we can follow a similar pattern. An added wrinkle is that `winrt::resume_after` does not return an `IAsyncAction` so we'll have to adapt it. This isn't too much an extra wrinkle, since we ended up having to adapt `Task.Delay` in C# anyway, so it's work that gets done sooner or later. It's just that in C++/WinRT, it's done sooner.

```
co_await winrt::when_any(
    DoSomethingAsync(),
    [] -> IAsyncAction { co_await winrt::resume_after(1s); });
```

The C++/WinRT `when_any` doesn't tell you who the winner was. It just completes with the result of the task that completed first. This means that we don't know whether the `when_any` finished due to normal completion or timeout.

We could infer which one completed first by having the timeout set a flag.

```
auto timed_out = make_shared(false);

co_await winrt::when_any(
    DoSomethingAsync(),
    [](auto flag) -> IAsyncAction
        { co_await winrt::resume_after(1s); *flag = true; }(timed_out));
if (timed_out) { ... }
```

For `IAsyncOperation`, we need our timer to return some fallback value:

```
template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<std::is_same_v<Result, void>, int> = 0>
Async delayed_async_result(
    TimeSpan delay)
{
    co_await winrt::resume_after(delay);
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<!std::is_same_v<Result, void>, int> = 0>
Async delayed_async_result(
    TimeSpan delay,
    Result fallback = winrt_empty_value<Result>())
{
    co_await winrt::resume_after(delay);
    co_return fallback;
}
```

The `delayed_async_result` is cumbersome for multiple reasons.

First, there's the need to identify the result of the `Async` so we can generate a default fallback value if necessary. To do that, we infer it from the value returned by `GetResults()`. This covers `IAsyncAction`, `IAsyncActionWithProgress`, `IAsyncOperation`, and `IAsyncOperationWithProgress`.

If that result is `void`, then we need to remove the extra `Result` parameter.

If that result is not `void`, we need to come up with the default fallback value. This is tricky for the case of Windows Runtime classes, but we dealt with that a little while ago when we wrote the `winrt_empty_value` function.

Or maybe we want to raise a timeout exception if the operation times out:

```
template<typename Async>
Async delayed_timeout_exception(TimeSpan delay)
{
    co_await winrt::resume_after(delay);
    throw winrt::hresult_error(HRESULT_FROM_WIN32(ERROR_TIMEOUT));
}
```

We can use these in conjunction with `when_any` to await a Windows Runtime asynchronous activity with a timeout.

It could be an `IAsyncAction` or `IAsyncActionWithProgress`, in which case the `co_await` simple returns early.

```
auto somethingTask = DoSomethingAsync();
co_await winrt::when_any(
    somethingTask,
    delayed_async_result<decltype(somethingTask)>(1s));
```

Or it could be an `IAsyncOperation` or `IAsyncOperationWithProgress`, in which case the `co_await` produces the result or the fallback value:

```
auto somethingTask = GetSomethingAsync();
auto result = co_await winrt::when_any(
    somethingTask,
    delayed_async_result<decltype(somethingTask)>(1s));
```

Or you can ask for an exception to be raised if the operation takes too long:

```
auto somethingTask = GetSomethingAsync();
auto result = co_await winrt::when_any(
    somethingTask,
    delayed_timeout_exception<decltype(somethingTask)>(1s));
```

Having to pass a delayed result or delayed exception with a matching type as the thing you're waiting for calls for a helper function to save you the typing:

```
template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(Async async, TimeSpan delay)
{
    return co_await winrt::when_any(async,
        delayed_async_result<Async>(delay));
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<!std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(
    Async async,
    TimeSpan delay,
    Result fallback = winrt_empty_value<Result>())
{
    return co_await winrt::when_any(async,
        delayed_async_result<Async>(delay, fallback));
}

template<typename Async>
Async when_complete_or_timeout_exception(
    Async async,
    TimeSpan delay)
{
    return co_await winrt::when_any(async,
        delayed_timeout_exception<Async>(delay));
}
```

In the discussion of the C# version of these helpers, I noted that if the operation times out, it nevertheless continues to run. Windows Runtime asynchronous activities support the `Cancel()` method, so you can tell them to abandon whatever they were doing. We can add that feature to our helper function, so that all the incomplete activities are cancelled.

Note that we had been leaving our `delayed_...` tasks uncancelled, so the timers nevertheless continue to run and either complete with nobody listening, or raise an exception that nobody is listening to. If you're doing a lot of timeouts, these extra threadpool timers will eventually drain, but you may not want them to accumulate in the first place.

So let's cancel everything before we finish.

```
template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<std::is_same_v<Result, void>, int> = 0>
Async delayed_async_result(
    TimeSpan delay)
{
    (co_await winrt::get_cancellation_token()).enable_propagation();
    co_await winrt::resume_after(delay);
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<!std::is_same_v<Result, void>, int> = 0>
Async delayed_async_result(
    TimeSpan delay,
    Result fallback = winrt_empty_value<Result>())
{
    (co_await winrt::get_cancellation_token()).enable_propagation();
    co_await winrt::resume_after(delay);
    co_return fallback;
}

template<typename Async>
Async delayed_timeout_exception(TimeSpan delay)
{
    (co_await winrt::get_cancellation_token()).enable_propagation();
    co_await winrt::resume_after(delay);
    throw winrt::hresult_error(HRESULT_FROM_WIN32(ERROR_TIMEOUT));
}
```

To prevent timers from lingering, we enable cancellation propagation in our delayed result/exception coroutines. That way, when they are cancelled, they cancel the timer immediately rather than leaving the timer running, only for it to have nothing to do when it expires.

```
template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(Async async, TimeSpan delay)
{
    auto timeout = delayed_async_result<Async>(delay);

    auto cancel_async = wil::scope_exit([&] { async.Cancel(); });
    auto cancel_timeout = wil::scope_exit([&] { timeout.Cancel(); });

    return co_await winrt::when_any(async, timeout);
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<!std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(
    Async async,
    TimeSpan delay,
    Result fallback = winrt_empty_value<Result>())
{
    auto timeout = delayed_async_result<Async>(delay, fallback);

    auto cancel_async = wil::scope_exit([&] { async.Cancel(); });
    auto cancel_timeout = wil::scope_exit([&] { timeout.Cancel(); });

    return co_await winrt::when_any(async, timeout);
}

template<typename Async>
Async when_complete_or_timeout_exception(
    Async async,
    TimeSpan delay)
{
    auto timeout = delayed_timeout_exception<Async>(delay);

    auto cancel_async = wil::scope_exit([&] { async.Cancel(); });
    auto cancel_timeout = wil::scope_exit([&] { timeout.Cancel(); });

    return co_await winrt::when_any(async, timeout);
}
```

After accepting the `Async` and creating our matching timeout, we use an RAII type to ensure that both are cancelled when the coroutine completes, even if it completes with an exception. Cancelling an already-completed Windows Runtime asynchronous activity has no effect, so we don't have to keep track of which activity completed and which is being abandoned. We just cancel them all and let somebody else figure it out.

There's a lot of repetition in the version up above, so let's try to shorten it up a bit.

```
template<typename First, typename...Rest>
First when_any_cancel_others(First first, Rest...rest)
{
    auto cleanup = std::make_tuple(
        wil::scope_exit([&] { first.Cancel(); }),
        wil::scope_exit([&] { rest.Cancel(); })...);

    return co_await winrt::when_any(first, rest...);
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(Async async, TimeSpan delay)
{
    return when_any_cancel_others(async,
        [](TimeSpan delay) -> Async {
            (co_await winrt::get_cancellation_token()).enable_propagation();
            co_await winrt::resume_after(delay);
        }(delay));
}

template<
    typename Async,
    typename Result = decltype(std::declval<Async>().GetResults()),
    typename std::enable_if_t<!std::is_same_v<Result, void>, int> = 0>
Async when_complete_or_timeout(
    Async async,
    TimeSpan delay,
    Result fallback = winrt_empty_value<Result>())
{
    return when_any_cancel_others(async,
        [](TimeSpan delay, Result fallback) -> Async {
            (co_await winrt::get_cancellation_token()).enable_propagation();
            co_await winrt::resume_after(delay);
            co_return fallback;
        }(delay, std::move(fallback)));
}

template<typename Async>
Async when_complete_or_timeout_exception(
    Async async,
    TimeSpan delay)
{
    return when_any_cancel_others(async,
        [](TimeSpan delay) -> Async {
            (co_await winrt::get_cancellation_token()).enable_propagation();
            co_await winrt::resume_after(delay);
            throw winrt::hresult_error(HRESULT_FROM_WIN32(ERROR_TIMEOUT));
        }(delay));
}
```

Exercise: Why can't we simplify `when_any_cancel_others` to this?

```
template<typename...Args>
auto when_any_cancel_others(Args...args)
{
    auto cleanup = std::make_tuple(
        wil::scope_exit([&] { args.Cancel(); })...);

    return winrt::when_any(args...);
}
```

Exercise 2: Why not use perfect forwarding to avoid the extra `AddRef` and `Release`?

```
template<typename First, typename...Rest>
std::decay_t<First>
when_any_cancel_others(First&& first, Rest&...rest)
{
    auto cleanup = std::make_tuple(
        wil::scope_exit([&] { first.Cancel(); }),
        wil::scope_exit([&] { rest.Cancel(); })...);

    return co_await winrt::when_any(
        std::forward<First>(first),
        std::forward<Rest>(rest)...);
}
```

Raymond Chen

Follow