How can I co_await on a Windows Runtime async action or operation with a timeout?

devblogs.microsoft.com/oldnewthing/20220415-00

April 15, 2022

Raymond Chen

Say you want to `co_await` on an `IAsyncOperation` but also want to abandon the await if it takes too long.

```
auto widgetOperation = GetWidgetAsync();
if (winrt::wait_for(widgetOperation, 15s) == AsyncStatus::Started)
{
    // timed out
    widgetOperation.Cancel(); // abandon the operation
} else {
    // will throw if operation failed or was cancelled
    auto widget = widgetOperation.GetResults();
}
```

The downside of this approach is that it blocks the thread for 15 seconds. This is bad enough for a background thread, since you are holding a thread hostage for the duration, but it's really bad for a UI thread, since it makes your UI hang.

Instead, you can use `when_any` which creates a new `IAsyncXxx` that completes when any of its parameters completes. The first parameter is the `widgetOperation`, and the second is a coroutine that returns the same type of operation, but completes after a set timeout.

```
auto widgetOperation = GetWidgetAsync();
auto widgetTimeout = [] -> IAsyncOperation<Widget>
    {
        co_await winrt::resume_after(15s);
        co_return nullptr;
    }();
auto widget = co_await winrt::when_any(widgetOperation, widgetTimeout);

widgetOperation.Cancel();
widgetTimeout.Cancel();

if (!widget) {
    // timed out or GetWidgetAsync() returned nullptr
} else {
    // GetWidgetAsync() produced a widget
}
```

We create two `IAsyncOperation<Widget>`s. One is the operation we care about (`GetWidgetAsync()`) and the other is an operation that waits 15 seconds, and then completes with `nullptr`.

We then use `winrt::when_any` to await until anything completes. Once anything completes, we cancel everything. This has no effect on asynchronous operations that are completed, but it tells the still-incomplete operations to abandon their work if they can. Cancelling everything isn't technically necessary, but it does avoid doing extra work that is going to be ignored anyway.

After all the tidying is done, we see if we actually got something. If the `GetWidgetAsync()` is taking too long, then the winning operation will be the `widgetTimeout`, and it completes with `nullptr`. If the `GetWidgetAsync()` finishes in under 15 seconds, then it will be the winning operation, and the result will be the thing it produced. That thing it produced could have been `nullptr`, so you can't distinguish it from a timeout, but the idea here is that you picked the `co_return` at the end of the `widgetTimeout` lambda so that `nullptr` corresponds to what you would have wanted to do in the case of a timeout.

If you really need to distinguish between completing with `nullptr` and timing out, you could add an explicit flag.

```
auto timedOut = std::make_shared<bool>();
auto widgetOperation = GetWidgetAsync();
auto widgetTimeout = [](auto timedOut) -> IAsyncOperation<Widget>
    {
        co_await winrt::resume_after(15s);
        *timedOut = true;
        co_return nullptr;
    }(timedOut);
auto widget = co_await winrt::when_any(widgetOperation, widgetTimeout);

widgetOperation.Cancel();
widgetTimeout.Cancel();

if (*timedOut) {
    // timed out
} else {
    // GetWidgetAsync() produced something (possibly nullptr)
}
```

Or, perhaps more sneakily, you can check the status of the `widgetTimeout` operation.

```
auto widgetOperation = GetWidgetAsync();
auto widgetTimeout = [] -> IAsyncOperation<Widget>
    {
        co_await winrt::resume_after(15s);
        co_return nullptr;
    }();
auto widget = co_await winrt::when_any(widgetOperation, widgetTimeout);
auto timedOut = widgetTimeout.Status() == AsyncStatus::Completed;

widgetOperation.Cancel();
widgetTimeout.Cancel();

if (timedOut) {
    // timed out
} else {
    // GetWidgetAsync() produced something (possibly nullptr)
}
```

We'll expand upon this pattern in a few weeks, so stay tuned. (Or "tune out", if that's more to your liking.)

Raymond Chen

Follow