Creating a non-agile delegate in C++/WinRT, part 4: Waiting synchronously from a background thread

devblogs.microsoft.com/oldnewthing/20200409-00

April 9, 2020

Raymond Chen

Warning to those who stumbled onto this page: Don't use the code on this page without reading all the way to the end.

This week, we assembled a function `resume_synchronous` that synchronously resumes execution in another apartment. The use case for this is a a delegate running on a background thread that needs to run code synchronously on a UI thread.

You can get the same effect using things that come built into C++/WinRT: You can call the `get()` method on a Windows Runtime asynchronous operation, and C++/WinRT will block the calling apartment until the asynchronous operation is complete.

```
winrt::IAsyncAction DoActualWorkAsync(
    CoreDispatcher dispatcher, DeviceInformation info)
{
  co_await winrt::resume_foreground(dispatcher);
  viewModel.Append(winrt::make<DeviceItem>(info));
}

deviceWatcher.Added(
    [=](auto&& sender, auto&& info)
    {
        DoActualWorkAsync(Dispatcher(), info).get();
    });
```

The idea here is that we start by calling `DoActualWorkAsync`, which does its work asynchronously. But instead of `co_await`ing the result, we `get()` it: The `get()` method waits synchronously for the operation to complete.

Waiting synchronously for the asynchronous operation to complete is advisable only from background threads. Performing a synchronous wait from a UI thread will naturally make your program unresponsive for the duration of the wait. But it's worse than that: The asynchronous operation may itself want to use the UI thread, but it won't able to since you blocked it. The result is a deadlock on your UI thread, and that makes everybody sad.

Now, splitting out the asynchronous part into a separate function is a bit of an annoyance, since everything you want to use on the UI thread needs to be passed in as a parameter. Maybe we can do better.

```
template<typename TLambda>
winrt::IAsyncAction DispatchAsync(
    CoreDispatcher dispatcher, TLambda&& lambda)
{
  co_await winrt::resume_foreground(dispatcher);
  lambda();
}

deviceWatcher.Added(
    [=](auto&& sender, auto&& info)
    {
        DispatchAsync(Dispatcher(), [&]
        {
            viewModel.Append(winrt::make<DeviceItem>(info));
        }).get();
    });
```

This version accepts a lambda and runs it after switching to the dispatcher's UI thread.

But this code looks all wrong. We're taking objects captured by reference and using them across a suspending `co_await` boundary! Isn't this a recipe for disaster, using a reference after the referent may have been destroyed?

Yes, this is dangerous in general, but it works in this specific case because the outer `IAsync-Action` is not `co_await`ed; it is passed to `get()`, which performs a synchronous wait. Therefore, all the parameters will remain valid for the lifetime of the coroutine, because the destruction of the parameters doesn't happen until the end of the "full expression" that ends in `get()`.

Once we realize that, we can take things a step further:

```
deviceWatcher.Added(
    [=](auto&& sender, auto&& info)
    {
        [&]() -> winrt::IAsyncAction
        {
            co_await winrt::resume_foreground(Dispatcher());
            viewModel.Append(winrt::make<DeviceItem>(info));
        }().get();
    });
```

The lambda won't be destructed until the end of the full expression, which happens after `get()` returns, which means that all the reference captures will remain valid for the lifetime of the lambda.

I guess you could factor this:

```
template<typename TLambda>
void RunSyncOnDispatcher(
    CoreDispatcher const& dispatcher,
    TLambda&& lambda)
{
  [&]() -> winrt::IAsyncAction
  {
    co_await winrt::resume_foreground(dispatcher);
    lambda();
  }().get();
}

deviceWatcher.Added(
    [=](auto&& sender, auto&& info)
    {
        RunSyncOnDispatcher(Dispatcher(), [&]()
        {
            viewModel.Append(winrt::make<DeviceItem>(info));
        });
    });
```

It does create the risk that somebody will pass a lambda that is itself a coroutine. Since `RunSyncOnDispatcher` does not `co_await` the result of the lambda, the synchronous execution lasts only up until the lambda reaches its first suspending `co_await`, which makes the rest of the lambda coroutine execute without the protection of the `get()`.

There's a sneaky bug in this code, however. We'll look at it next time.

Raymond Chen

Follow