Creating a non-agile delegate in C++/WinRT, part 2: The synchronous coroutine

devblogs.microsoft.com/oldnewthing/20200407-00

April 7, 2020

Raymond Chen

Last time, we saw that you could use an `ICallbackContext` to run code synchronously in another apartment from your delegate, which is important if the code that is calling your delegate is relying on the timing of your return.

We can also express this in the form of a coroutine that operates synchronously.

If we make the `await_suspend` invoke the handle synchronously, then the continuation of the coroutine runs synchronously with the code that called `co_await`.

```
auto resume_synchronous(ICallbackContext* context)
{
  struct awaiter : std::experimental::suspend_always
  {
    ICallbackContext* context;
    bool await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      InvokeInContext(context, handle);
      return true;
    }
  };
  return awaiter{ context };
}
```

This simplifies the delegate by letting you use `co_await` to do the dirty work.

```
deviceWatcher.Added(
    [=, context = CaptureCurrentApartmentContext()]
    (auto&& sender, auto&& info) -> winrt::fire_and_forget
    {
        co_await resume_synchronous(context.Get());
        viewModel.Append(winrt::make<DeviceItem>(info));
    });
```

Even though there is a `co_await`, execution continues synchronously because `await_suspend` runs the continuous synchronously.

Whether `co_await` resumes synchronously or not[1] is determined by the awaiter. If you `co_await` something whose awaiter resumes asynchronously, then the `co_await` will resume asynchronously.

```
deviceWatcher.Added(
    [=, context = CaptureCurrentApartmentContext()]
    (auto&& sender, auto&& info) -> winrt::fire_and_forget
    {
        auto original_context = CaptureCurrentApartmentContext();
        co_await resume_synchronous(context.Get());
        viewModel.Append(make<DeviceItem>(info));
        co_await resume_synchronous(original_context.Get());
        more_stuff();
        auto result = co_await GetMoreDataAsync();
        process_result(result);
    });
```

In the above example, the first two `co_await`s are synchronous, but the third one (`co_await GetMoreDataAsync()`) is presumably asynchronous. This means that the delegate will return at the point of the third `co_await`, and reference parameters (`sender` and `info`) are probably not going to be valid when the coroutine resumes.

[1] Or at all. The built-in awaiter `suspend_always` suspends and never wakes up.

Raymond Chen

Follow