When do CoreDispatcher.RunAsync and ThreadPool.RunAsync actions complete?

devblogs.microsoft.com/oldnewthing/20190326-00

March 26, 2019

Raymond Chen

The `CoreDispatcher::RunAsync` and `ThreadPool::RunAsync` methods take a delegate and schedule it to be invoked on the dispatcher thread or on a thread pool thread. These methods return an `IAsyncAction`, but when does that action complete?

When dealing with asynchronous methods, there are two ways of talking about the result.

First, there's the return value of the asynchronous method, which at the ABI level is an `IAsyncAction` or `IAsyncOperation`. Depending on the language projection, this is exposed to the programmer as a language-specific object.

Projection	`IAsyncAction`	`IAsyncOperation<T>`	Notes
C++/WinRT	`IAsyncAction`	`IAsyncOperation<T>`	
C++/CX	`IAsyncAction^ task<void>`	`IAsyncOperation<T>^ task<T>`	Projected as `IAsyncXxx` usually wrapped into `task` / `Task`.
C#	`IAsyncAction Task`	`IAsyncOperation<T> Task<T>`	
JavaScript	`Promise`	`Promise`	

The second result is the thing that you receive when the asynchronous operation completes.

Projection	`IAsyncAction`	`IAsyncOperation<T>`
C++/WinRT	`void`	`T`
C++/CX	`void`	`T`
C++/CX + PPL	`void`	`T`
C#	`void`	`T`

JavaScript	`undefined`	`T`

And there's also a third thing to worry about, which is *when* you receive that completion result.

Let's answer the three questions for the `CoreDispatcher::RunAsync` and `ThreadPool::RunAsync` methods.

First, they *return* an `IAsyncAction`. The methods schedule the delegate to be invoked later, and then return an `IAsyncAction` representing the pending operation.

Second, they *complete* with `void`. There is no additional information reported when the operation completes.

Third (and most interesting) is that they complete when the delegate *returns*.

Not when the delegate *completes*.

This means that when you pass a delegate that itself represents an asynchronous operation, the `IAsyncAction` returned by `RunAsync` completes once your delegate returns its own async operation. The dispatcher or thread pool doesn't even see that async operation; it's eaten by your language projection. All that the dispatcher or thread pool knows is that it invoked the delegate, and the delegate returned `void`, so we must be done.

The C++/WinRT, and JavaScript projections permit your delegate to return someting, even though the formal function signature returns `void`. The projection just throws your return value away, and the caller gets nothing. The C# language lets you make a function formally return `void`, even though it secretly continues running asynchronously. The syntax for this is `async void`, and I've discussed the perils of `async void` in the past.

This means that if you await the result of a `RunAsync`, the await will complete when your delegate either returns or performs its own await operation, whichever comes first.

```
// C++/WinRT

co_await Dispatcher().RunAsync(CoreDispatcherPriority::Normal,
    [lifetime = get_strong()]() -> fire_and_forget
    {
        co_await SomethingAsync();
        co_await SomethingElseAsync();
        Finished();
    });

// C++/CX

create_task(Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
    ref new DispatchedHandler([this]()
    {
        create_task(SomethingAsync()).then([this]() {
            return create_task(SomethingElseAsync());
        }).then([this]() {
            Finished();
        });
    }))).then([this]()
    {
        BackOnMainThread();
    });


// C++/CX + co_await

co_await Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
    ref new DispatchedHandler([this]()
    {
        []() -> task<void>
        {
            co_await SomethingAsync();
            co_await SomethingElseAsync();
            Finished();
        }();
    }));
BackOnMainThread();

// C#

await Dispatcher.RunAsync(CoreDispatcherPriority::Normal, async () =>
    {
        await SomethingAsync();
        await SomethingElseAsync();
        Finished();
    });
BackOnMainThread();

// JavaScript (pretend)[1]
```

```
await dispatcher.runAsync(CoreDispatcherPriority.normal, async () =>
    {
        await somethingAsync();
        await somethingElseAsync();
        finished();
    });
backOnMainThread();
```

When does the `await` / `co_await` complete and the `BackOnMainThread` run?

Answer: When `SomethingAsync` returns its `IAsyncAction`, that action gets wrapped inside a coroutine, and execution suspends, returning control to the dispatcher or thread pool. At this point, the delegate has returned, and the `RunAsync` declares its action to have completed. The object representing the coroutine (the `IAsyncAction`, `task`, `Task`, or `Promise`) is simply discarded.

In C++/WinRT and JavaScript, the discarding is done by the projection. In C++/CX, the discarding is explicit in the code: Observe that we create a task but do not `return` it. In C#, the discarding is done by the language itself because an `async` lambda can be implicitly converted to a non-async `void` lambda (by treating it as if were `async void`).

Another way of looking at this analysis is that the lambda returns when it encounters its first `await` / `co_await` or `return`. This in turn causes the `RunAsync` to complete its own `IAsyncAction`.

If we write things out explicitly, the sequence of operations might be more clear:

```
// C#
async () =>
{
    await SomethingAsync();
    await SomethingElseAsync();
    Finished();
}
```

This gets transformed by the compiler into

```
class Lambda
{
    async void Invoke()
    {
        await SomethingAsync();
        await SomethingElseAsync();
        Finished();
    }
}
```

which gets further transformed into

```
class Lambda
{
    void Invoke()
    {
        Task task1 = SomethingAsync();
        task1.ContinueWith(_ => {
            Task task2 = SomethingElseAsync();
            task2.ContinueWith(_ => {
                Finished();
            });
        });
    }
}
```

Once `SomethingAsync` returns its `Task`, the lambda attaches a continuation to it, so that it can resume execution when the task completes. At that point, the outer lambda has finished its work, and the `Invoke` method returns. This returns control back to the delegate or thread pool, which declares that the `RunAsync` has completed. And the completion of `RunAsync` means that `BackOnMainThread` starts to run.

This behavior is usually not what you want. You want to wait until the lambda has *completed*, not just returned. We'll look at one possible solution next time.

[1] JavaScript is a single-threaded language, so you can't actually do this, but I included it for completeness to demonstrate what *would* happen if it were possible.

Raymond Chen

Follow