The perils of async void

devblogs.microsoft.com/oldnewthing/20170721-00

July 21, 2017

Raymond Chen

We saw last time that `async void` is an odd beast, because it starts doing some work, and then returns as soon as it encounters an `await`, with the rest of the work taking place at some unknown point in the future.

Why would you possibly want to do that?

Usually it's because you have no choice. For example, you may be subscribing to an event, and the event delegate assumes a synchronous handler. You want to do asynchronous work in the handler, so you use `async void` so that your handler has the correct signature, but you can still `await` in the function.

The catch is that only the part of the function before the first `await` runs in the formal event handler. The rest runs after the formal event handler has returned. This is great if the event source doesn't have requirements about what must happen before the handler returns. For example, the `Button.Click` event lets you know that the user clicked the button, but it doesn't care when you finish processing. It's just a notification.

On the other hand, an event like `Suspending` assumes that when your event handler returns, it is okay to proceed with the suspend. But that may not be the case if your handler contains an `await`. The handler has not logically finished executing, but it did return from its handler, because the handler returned a `Task` which captures the continued execution of the function when the `await` completes.

Aha, but you can fix this by making the delegate return a `Task`, and the event source would `await` on the task before concluding that the handler is ready to proceed.

There are some problems with this plan, though.

One problem is that making the event delegate return a `Task` is that the handler might not need to do anything asynchronous, but you force it to return a task anyway. The natural expression of this results in a compiler warning:

```
// Warning CS1998: This async method lacks 'await'
// operators and will run synchronously.
async Task SuspendingHandler(object sender, SuspendingEventArgs e)
{
  // no await calls here
}
```

To work around this, you need to add `return Task.CompletedTask;` to the end of the function, so that it returns a task that has already completed.

A worse problem is that the return value from all but the last event handler is not used.

> If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

(If there is no event handler, then attempting to raise the event results in a null reference exception.)

So if there are multiple handlers, and each returns a `Task`, then only the last one counts.

Which doesn't seem all that useful.

The Windows Runtime developed a solution to this problem, known as the Deferral Pattern. The event arguments passed to the event handler includes a method called `GetDeferral()`. This method returns a "deferral object" whose purpose in life is to keep the event handler "logically alive". When you Complete the deferral object, then that tells the event source that the event handler has logically completed, and the event source can proceed.

If your handler doesn't perform any `await`s, then you don't need to worry about the deferral.

```
void SuspendingHandler(object sender, SuspendingEventArgs e)
{
  // no await calls here
}
```

If you do an `await`, you can take a deferral and complete it when you're done.

```
async void SuspendingHandler(object sender, SuspendingEventArgs e)
{
  var deferral = e.SuspendingOperation.GetDeferral();

  // Even though there is an await, the suspending handler
  // is logically still active because there is a deferral.
  await SomethingAsync();

  // Completing the deferral signals that the suspending
  // handler is logically complete.
  deferral.Complete();
}
```

The `Suspending` event is a bit strange for historical reasons.

Starting in Windows 10, there is a standard Deferral object which also supports `IDisposable`, so that you can use the `using` statement to complete the deferral automatically when control leaves the block. If the `Suspending` event were written today, you would be able to do this:

```
async void SuspendingHandler(object sender, SuspendingEventArgs e)
{
  using (e.GetDeferral()) {

    // Even though there is an await, the suspending handler
    // is logically still active because there is a deferral.
    await SomethingAsync();

 } // the deferral completes when code leaves the block
}
```

Alas, we don't yet have that time machine the Research division is working on, so the new `using`-based pattern works only for deferrals added in Windows 10. A `using`-friendly deferral will implement `IDisposable`. Fortunately, if you get it wrong and try to `using` a non-disposable deferral, the compiler will notice and report an error: "CS1674: type used in a using statement must be implicitly convertible to 'System.IDisposable'".

And that's the end of CLR We… no wait! CLR Week will continue into next week! What has the world come to!?

Raymond Chen

Follow