How do I produce a Windows Runtime asynchronous activity from C#?

devblogs.microsoft.com/oldnewthing/20240704-00

July 4, 2024

Raymond Chen

Last time, we looked at how to produce a Windows Runtime asynchronous activity in C++/CX. Doing it in C# is similar: The method that converts a C# `Task` to a Windows Runtime asynchronous activity is `AsyncInfo.Run`, and just like PPL's `create_async`, it infers the resulting asynchronous activity from the signature of the object you pass to it.

Given a lambda that returns a type `R` and takes the parameters `Params...`, the `AsyncInfo.Run` method returns the following Windows Runtime interface:

	Params…	
	`(CancellationToken)`	`(CancellationToken, IProgress<P>)`
`R` = `Task`	`IAsyncAction`	`IAsyncActionWithProgress<P>`
`R` = `Task<T>`	`IAsyncOperation<T>`	`IAsyncOperationWithProgress<T, P>`

This is basically the same table that we had for `create_async`, but with three significant changes:

1. The lambda must accept at least a `CancellationToken`. (It was optional for `create_async`.)
2. The lambda must return a `Task` or `Task<T>`. (The `create_async` allowed you to return `void` or `T`.)

3. The lambda's cancellation token and progress reporter parameters are in opposite order!

As with `create_async`, the lambda can use the `IProgress<P&gt progress_reporter<P>` to produce progress reports, and the `CancellationToken` to detect whether the asynchronous activity has been canceled.

Here's the simplest case: An `IAsyncAction`.

```
Task<Widget> GetWidgetAsync(string id);
Task EnableWidgetAsync(Widget widget, bool enable);

IAsyncAction EnableWidgetByIdAsync(string id, bool enable)
{
    return AsyncInfo.Run(async (cancel) => {
        var widget = await GetWidgetAsync(id);
        cancel.ThrowIfCancellationRequested();
        await EnableWidgetAsync(widget, enable);
    });
}
```

And with progress:

```
IAsyncActionWithProgress<int> EnableWidgetByIdAsync(string id, bool enable)
{
    return AsyncInfo.Run(async (CancellationToken cancel,
                                IProgress<int> progress) => {
        progress.Report(0);
        var widget = await GetWidgetAsync(id);
        cancel.ThrowIfCancellationRequested();
        progress.Report(1);
        await EnableWidgetAsync(widget, enable);
    });
}
```

The compiler can't infer what the `P` is for the `IProgress<P>` parameter, so you have to specify it explicitly. And then you run into another rule that says that if you provide an explicit type for one lambda parameter, you must do so for all of them, so we're stuck typing `CancellationToken` again.

But wait, there's an easier way: Instead of helping the compiler infer `IProgress<P>`, we explicitly specialize the `Run` method.

```
IAsyncActionWithProgress<int> EnableWidgetByIdAsync(string id, bool enable)
{
    return AsyncInfo.Run<int>(async (cancel, progress) => {
        progress.Report(0);
        var widget = await GetWidgetAsync(id);
        cancel.ThrowIfCancellationRequested();
        progress.Report(1);
        await EnableWidgetAsync(widget, enable);
    });
}
```

May as well do the async operations, too.

```
IAsyncOperation<bool> EnableWidgetByIdAsync(string id, bool enable)
{
    return AsyncInfo.Run(async (cancel) => {
        var widget = await GetWidgetAsync(id);
        cancel.ThrowIfCancellationRequested();
        if (widget == null) return false;
        await EnableWidgetAsync(widget, enable);
        return true;
    });
}

IAsyncOperationWithProgress<bool, int> EnableWidgetByIdAsync(string id, bool enable)
{
    return AsyncInfo.Run<bool, int>(async (cancel, progress) => {
        progress.Report(0);
        var widget = await GetWidgetAsync(id);
        cancel.ThrowIfCancellationRequested();
        progress.Report(1);
        if (widget == null) return false;
        await EnableWidgetAsync(widget, enable);
        return true;
    });
}
```

Now, in the case where you don't support cancellation or progress, you have an alternative: For the case of `IAsyncAction`, you can create a `Task` and then call its `AsAsyncAction` method, and in the case of `IAsyncOperation<T>`, you call its its `AsAsyncOperation` method. These aren't true methods on `Task` but rather extension methods provided by the `System.WindowsRuntimeSystemExtensions` namespace, so you may need to add a `using System.WindowsRuntimeSystemExtensions;` to your file.

```
using System.WindowsRuntimeSystemExtensions;

IAsyncAction EnableWidgetByIdAsync(string id, bool enable)
{
    Func<Task> taskMaker = async () => {
        var widget = await GetWidgetAsync(id);
        await EnableWidgetAsync(widget, enable);
    };
    return taskMaker().AsAsyncAction();
}

IAsyncOperation<int> EnableWidgetByIdAsync(string id, bool enable)
{
    Func<Task<int>> taskMaker = async () => {
        var widget = await GetWidgetAsync(id);
        if (widget == null) return false;
        await EnableWidgetAsync(widget, enable);
        return true;
    };
    return taskMaker().AsAsyncOperation();
}
```

These conversions are more convenient to use if you just turn the lambda into its own named function:

```
// IAsyncAction version
async Task EnableWidgetByIdTaskAsync(string id, bool enable)
{
    var widget = await GetWidgetAsync(id);
    await EnableWidgetAsync(widget, enable);
}

IAsyncAction EnableWidgetByIdAsync(string id, bool enable)
{
    return EnableWidgetByIdTaskAsync(id, enable).AsAsyncAction();
}

// IAsyncOperation version
async Task<bool> EnableWidgetByIdTaskAsync(string id, bool enable)
{
    var widget = await GetWidgetAsync(id);
    if (widget == null) return false;
    await EnableWidgetAsync(widget, enable);
    return true;
}

IAsyncOperation<int> EnableWidgetByIdAsync(string id, bool enable)
{
    return EnableWidgetByIdTaskAsync(id, enable).AsAsyncOperation();
}
```