How do I produce a Windows Runtime asynchronous activity from C++/CX?

devblogs.microsoft.com/oldnewthing/20240703-00

July 3, 2024

Raymond Chen

You might be working in a code base written in C++/CX.

First, I'm sorry.

Second, maybe you need to produce an `IAsyncAction^` or one of its relatives. How do you do that?

You use the `Concurrency::create_async` method.

The `Concurrency::create_async` method studies its parameter and infers what kind of Windows Runtime asynchronous activity to produce based on the signature of the lambda.

Given a lambda whose function call operator has the signature `R(Params...)`, the `create_async` function returns the following Windows Runtime interface:

	Params…	

	`()`
	`(cancellation_token)`
		`(progress_reporter<P>)`
		`(progress_reporter<P>, cancellation_token)`

`R` = `void`
`R` = `task<void>`
	`IAsyncAction^`	`IAsyncActionWithProgress<P>^`

R = T R = task<T>	IAsyncOperation<T>^	IAsyncOperationWithProgress<T, P>^

Inside the lambda, you can use the `progress_reporter<P>` to produce progress reports, and you can use the optional `cancellation_token` to detect whether the asynchronous activity has been canceled.

Here's the simplest case: An `IAsyncAction^` with no cancellation.

```
task<Widget^> GetWidgetAsync(String^);
task<void> EnableWidgetAsync(Widget^, bool);

// Old school: Task chain
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=]()
        -> task<void> {
        return GetWidgetAsync(id).then(
        [=](Widget^ widget) {
            return EnableWidgetAsync(widget, enable);
        });
    });
}

// New hotness: co_await
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=]()
        -> task<void> {
        Widget^ widget = co_await GetWidgetAsync(id);
        co_await EnableWidgetAsync(widget, enable);
    });
}
```

With cancellation but no progress:

```
void ThrowIfCanceled(cancellation_token const& cancel)
{
    if (cancel.is_canceled()) cancel_current_task();
}

// Old school: Task chain
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](cancellation_token cancel)
        -> task<void> {
        return GetWidgetAsync(id).then(
        [=](Widget^ widget) { // explicitly: [enable, cancel]
            ThrowIfCanceled(cancel);
            return EnableWidgetAsync(widget, enable);
        });
    });
}

// New hotness: co_await
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](cancellation_token cancel)
        -> task<void> {
        Widget^ widget = co_await GetWidgetAsync(id);
        ThrowIfCanceled(cancel);
        co_await EnableWidgetAsync(widget, enable);
    });
}
```

With progress but no cancellation:

```
// Old school: Task chain
IAsyncActionWithProgress<int>^
    EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress)
        -> task<void> {
        progress.report(0);
        return GetWidgetAsync(id).then(
        [=](Widget^ widget) { // explicitly: [enable, progress]
            progress.report(1);
            return EnableWidgetAsync(widget, enable);
        });
    });
}

// New hotness: co_await
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress))
        -> task<void> {
        progress.report(0);
        Widget^ widget = co_await GetWidgetAsync(id);
        progress.report(1);
        co_await EnableWidgetAsync(widget, enable);
    });
}
```

And with both progress and cancellation:

```
// Old school: Task chain
IAsyncActionWithProgress<int>^
    EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress,
                            cancellation_token cancel)
        -> task<void> {
        progress.report(0);
        return GetWidgetAsync(id).then(
        [=](Widget^ widget) { // explicitly: [enable, progress, cancel]
            ThrowIfCanceled(cancel);
            progress.report(1);
            return EnableWidgetAsync(widget, enable);
        });
    });
}

// New hotness: co_await
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress,
                            cancellation_token cancel)
        -> task<void> {
        progress.report(0);
        Widget^ widget = co_await GetWidgetAsync(id);
        ThrowIfCanceled(cancel);
        progress.report(1);
        co_await EnableWidgetAsync(widget, enable);
    });
}
```

We can generalize into a single pattern:

```
IAsyncSomething^
    DoSomethingAsync(Arg1 arg1, Arg2, arg2, ...)
{
    return create_async([=](
        progress_reporter<P> progress[optional],
        cancellation_token cancel[optional])
        -> task<P> {
        ⟦ async stuff which may include...
            progress.report(value);[optional]
            ThrowIfCanceled(cancel);[optional]
        ⟧
    });
}
```

You can also register a callback function on the `cancellation_token` that will be invoked when the activity is canceled. I leave using the cancellation callback as an exercise.

There are also variants for `IAsyncOperation`. I'll show just one of them and let you figure out the others:

```
// Old school: Task chain
IAsyncOperationWithProgress<bool, int>^
    EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress,
                           cancellation_token cancel)
        -> task<bool> {
        progress.report(0);
        return GetWidgetAsync(id).then(
        [=](Widget^ widget) { // explicitly: [enable, progress, cancel]
            ThrowIfCanceled(cancel);
            progress.report(1);
            if (!widget) {
                return task_from_result(false); // widget not found
            }
            return EnableWidgetAsync(widget, enable).then(
                []() { return true; });
        });
    });
}

// New hotness: co_await
IAsyncAction^ EnableWidgetByIdAsync(String^ id, bool enable)
{
    return create_async([=](progress_reporter<int> progress,
                           cancellation_token cancel)
        -> task<bool> {
        progress.report(0);
        Widget^ widget = co_await GetWidgetAsync(id);
        ThrowIfCanceled(cancel);
        progress.report(1);
        if (!widget) {
            co_return false;
        }
        co_await EnableWidgetAsync(widget, enable);
        co_return true;
    });
}
```