Windows Runtime asynchronous operations can fail in two different ways, so make sure you get them both

devblogs.microsoft.com/oldnewthing/20230119-00

January 19, 2023

Raymond Chen

CLR Tasks, PPL tasks, JavaScript Promises, and Windows Runtime asynchronous actions and operations can fail in two ways.

- They can throw an exception instead of returning the `Task`, `task`, `IAsyncAction`, or `IAsyncOperation`. "Synchronous failure."
- They can return a `Task`, `task`, `IAsyncAction`, or `IAsyncOperation` which completes with an exception. "Asynchronous failure."

Synchronous failures are raised at the point you call the method; you can think of them as "immediate failure". Asynchronous failure are raised at the point you check the result; you can think of them as "delayed failure".

Framework	Synchronous failure	Asynchronous failure
C#	`var task = o.DoSomethingAsync()`	`task.Result` `await task`
PPL	`auto task = o->DoSomethingAsync()`	`task.get()` `co_await task`
C++/WinRT	`auto op = o.DoSomethingAsync()`	`op.GetResults()` `co_await op`
JavaScript	`var p = o.DoSomethingAsync()`	`p.catch()` `await p`

A customer reported that they were getting exceptions from some code, which they couldn't understand because they thought they were handling exceptions.

```
// C++ with PPL
using namespace Concurrency;
using namespace Platform;

task<String^>
Widget::GetNameAsync()
{
    return m_doodad->GetNameAsync() // crash here
    .then([](task<String^> outerTask) {
        String^ name;

        try {
            name = outerTask.get();
        } catch (...) {
        }

        return name;
    }, task_continuation_context::use_arbitrary());
}
```

The code wraps the `outerTask.get()` inside a `try` block, so that should catch all the exceptions that come out of the `m_doodad->GetNameAsync()` task.

And that's true, it does catch all the exceptions that come out of the task.

But the exception that crashed didn't come out of the task!

The debugger pointed at the line that raised the exception: It was from the call to `GetNameAsync()` itself, before it even returned a task. The customer got so focused on the Concurrency Runtime that they forgot about the basic rules of C++: If you want to catch an exception, you have to do it inside a `try` block.

In order to catch that exception, the call to `GetNameAsync()` must itself be inside a `try` block.

```
task<String^>
Widget::GetNameAsync()
{
    task<String^> nameTask;
    try {
        nameTask = m_doodad->GetNameAsync();
    } catch (...) {
        return task_from_result<String^>(nullptr);
    }

    return nameTask.then([](task<String^> outerTask) {
        String^ name;

        try {
            name = outerTask.get();
        } catch (...) {
        }

        return name;
    }, task_continuation_context::use_arbitrary());
}
```

I separated the `return m_doodad->GetNameAsync().then()` into two steps:

```
    task<String^> nameTask = m_doodad->GetNameAsync();
    return nameTask.then(...);
```

The `try` statement inside the `then` lambda deals with exceptions that come out of the task. We just need another `try` to deal with the exceptions that occur while trying to produce the task:

```
    task<String^> nameTask;
    try {
        nameTask = m_doodad->GetNameAsync();
    } catch (...) {
        return task_from_result<String^>(nullptr);
    }
```

If an exception occurs, we catch it and return an already-completed task that produces an empty string. Otherwise, we hook up the continuation that deals with the task completion as before.

Once you see how the expression was taken apart, you can combine them again, putting the entire statement inside a giant `try` block, even though it's only the `->GetNameAsync()` that we're interested in. (Most languages with exceptions make it cumbersome to catch exceptions that come out of part of an expression, so most people just expand the scope of the `try` to include the entire statement.)

```
task<String^>
Widget::GetNameAsync()
{
    try {
        return m_doodad->GetNameAsync()
        .then([](task<String^> outerTask) {
            String^ name;

            try {
                name = outerTask.get();
            } catch (...) {
            }

            return name;
        }, task_continuation_context::use_arbitrary());
    } catch (...) {
        return task_from_result<String^>(nullptr);
    }
}
```

Note that if the customer had been using PPL with `co_await` support, the `try` block would naturally have enclosed both the production of the task as well as handling for its completion: The inability to wrap just part of an expression in a `try` block actually helps you write correct code this time:

```
task<String^>
Widget::GetNameAsync()
{
    try {
        co_return co_await m_doodad->GetNameAsync();
    } catch (...) {
        co_return nullptr;
    }
}
```

One catch with this rewrite is that `co_await` of a Concurrency Runtime `task` does not let you control the task continuation context. It always uses `get_current_winrt_context()` when awaiting tasks, and `CallbackContext::Same` when awaiting Windows Runtime asynchronous actions and operations.