How come my custom exception message is lost when it is thrown from a IAsyncAction^?

devblogs.microsoft.com/oldnewthing/20231102-00

November 2, 2023

Raymond Chen

A customer implemented an `IAsyncAction^` using the Parallel Patterns Library (PPL). They had the action throw an exception with a custom message, but found that the custom message was lost when they tried to catch it:

```
IAsyncAction^ DoSomething()
{
    return concurrency::create_async([] {
        throw ref new Platform::Exception(E_FAIL, "Sorry!");
    });
}

// consumer
concurrency::create_task(DoSomething())
.then([](concurrency::task<void> precedingTask) {
    try {
        precedingTask.get();
    } catch (Platform::Exception^ ex) {
        printf("0x%08x %ls\n",
            ex->HResult,
            ex->Message->Data());
    }
});
```

This code successfully catches the exception, and the `HResult` is preserved, but the custom message is lost.

What's going on?

Recall that at the ABI layer, the only way to report an error from an `IAsyncAction` is through an `HRESULT`. You can retrieve it by reading the `ErrorCode` property, or you can experience it live by calling `GetResults()` method and receiving the error as the failure code.

Now, there is a side channel mechanism for providing additional information: The `RoOriginateError` function lets you attach a message to a failure, which is stored in the thread context, and some libraries like C++/WinRT sets and retrieves this context when it generates

and reconstructs a `hresult_error` object.

But let's see what PPL does when a C++/CX exception occurs:

```
    // ppltasks.h
    template<typename _Function>
    ref class _AsyncTaskGeneratorThunk sealed :
        _AsyncProgressBase<typename
_AsyncLambdaTypeTraits<_Function>::_AsyncAttributes,
        _AsyncLambdaTypeTraits<_Function>::_AsyncAttributes::_TakesProgress>
    {
        [[…]]

        virtual void _OnStart() override
        {
            _M_task = _AsyncAttributesTaskGenerator::
                _Generate_Task<_Function, _AsyncTaskGeneratorThunk<_Function>^,
                    _Attributes>(_M_func, this, _M_cts, _M_creationCallstack);
            _M_task.then([=](task<typename _Attributes::_ReturnType> _Antecedent) {
                try
                {
                    _Antecedent.get();
                }
                catch (task_canceled&)
                {
                    this->_TryTransitionToCancelled();
                }
                catch (::Platform::Exception^ _Ex)
                {
                    this->_TryTransitionToError(_Ex->HResult);
                }
                catch (...)
                {
                    this->_TryTransitionToError(E_FAIL);
                }
                this->_FireCompletion();
            });
        }
        [[…]]
    };
```

Observe that when a C++/CX exception is thrown from the lambda, the exception's `HResult` is passed to `_TryTransitionToError`, but all the other details are ignored. The PPL library doesn't call `RoOriginateError`, or `GetErrorInfo`, so it doesn't use the side channel for conveying additional failure information. All that survives is the `HRESULT`.

Now, if `DoSomething` is a function internal to your project, then you can change it to return a `concurrency::task<void>`. The PPL library preserves exceptions thrown from `tasks`, and rethrows them when you call `.get()`.

```
IAsyncAction^ DoSomething()
{
    return concurrency::create_task([] {
        throw ref new Platform::Exception(E_FAIL, "Sorry!");
    });
}

// consumer
DoSomething()
.then([](concurrency::task<void> precedingTask) {
    try {
        precedingTask.get();
    } catch (Platform::Exception^ ex) {
        printf("0x%08x %ls\n",
            ex->HResult,
            ex->Message->Data());
    }
});
```