Cancelling a Windows Runtime asynchronous operation, part 2: C++/CX with PPL, explicit continuation style

devblogs.microsoft.com/oldnewthing/20200702-00

July 2, 2020

Raymond Chen

We began our investigation of Windows Runtime cancellation with how task cancellation is projected in C#. But how about C++/CX with PPL and explicit continuations?

Okay, let's do this.

```
auto picker = ref new FileOpenPicker();
picker->FileTypeFilter.Append(L".txt");

cancellation_token_source cts;
auto do_cancel = std::make_shared<call<bool>>([cts](bool) { cts.cancel(); });
auto delayed_cancel = std::make_shared<timer<bool>>(3000U, false, do_cancel.get());
delayed_cancel->start();

create_task(picker->PickSingleFileAsync()).
    then([do_cancel, delayed_cancel](task<StorageFile^> precedingTask)
    {
        StorageFile^ file;
        try {
            file = precedingTask.get();
        } catch (task_canceled const&) {
            file = nullptr;
        }

        if (file != nullptr) {
            DoSomething(file);
        }
    });
```

Setting up the timer to cancel the task is quite annoying. Both `call` objects and `timer` objects are non-copyable, but we need to keep both of the objects alive for the duration of the asynchronous operation, so we need to copy them into the lambda so that they will not be destructed prematurely. But then you run into that whole "non-copyable" business.

Your next thought would be to initialize the objects directly into the lambda:

```
    then([do_cancel = call<bool>(...),
          delayed_cancel = timer<bool>(...)]
         (task<StorageFile^> precedingTask)
```

But that too doesn't work because the lambda is copied around internally by PPL, so we once again run into the "non-copyable" problem.

We address the problem by putting both the `call` and the `timer` in a `shared_ptr`. The `shared_ptr` is copyable, and when the last one destructs, `call` and `timer` are destroyed.

Okay, that was a long and annoying aside.

When the underlying Windows Runtime asynchronous operation completes, PPL propagates the status into the task. You can see this happen in `ppltasks.h`. (I've simplified the code a bit for expository purposes.)

```
_AsyncOp->Completed = ref new AsyncOperationCompletedHandler<_ReturnType>(
              [_OuterTask](auto^ _Operation, AsyncStatus _Status) mutable
{
    if (_Status == AsyncStatus::Canceled)
    {
        _OuterTask->_Cancel(true);
    }
    else if (_Status == AsyncStatus::Error)
    {
        _OuterTask->_CancelWithException(
            std::make_exception_ptr(::ReCreateException(_Operation-
>ErrorCode.Value)));
    }
    else
    {
        _ASSERTE(_Status == AsyncStatus::Completed);

         try
        {
            _OuterTask->_FinalizeAndRunContinuations(_Operation->GetResults());
        }
        catch (...)
        {
            // unknown exceptions thrown from GetResult
            _OuterTask->_CancelWithException(std::current_exception());
        }
}
```

When the operation completes, PPL looks at the status code. If the status code says that the operation was canceled, then it cancels the wrapper task. If it says that the operation encountered an error, then it synthesizes an exception object from the error code and puts it in the wrapper task. Otherwise, the operation succeeded, so we get the results from the operation (`_Operation->GetResults()`) and set that as the result of the wrapper task. (There's an extra wrinkle: If `GetResults` itself throws an exception, then the wrapper task is set into an error state with that exception.)

Okay, so that's how the cancellation gets *into* the wrapper task. How does it come out?

PPL throws a `task_canceled` object when you try to get the results of a canceled task. This is documented under `task.get()`, and you can see it happen in `ppltask.h` :

```
_ReturnType get() const
{
    if (!_M_Impl)
    {
        details::_DefaultTaskHelper::_NoCallOnDefaultTask_ErrorImpl();
    }

    if (_M_Impl->_Wait() == canceled)
    {
        _THROW(task_canceled{});
    }

    return _M_Impl->_GetResult();
}
```

Next time, we'll look at PPL with coroutines.

Raymond Chen

Follow