Using C++/WinRT's final_release to control which thread destructs your object

devblogs.microsoft.com/oldnewthing/20220722-00

July 22, 2022

Raymond Chen

It is often that case that an object is intended to be used only from a single thread, particularly if it is tied to the user interface somehow, since user interface objects are generally single-threaded. On the other hand, it may have to subscribe to events that are raised from background threads.

```
namespace winrt
{
    using namespace Windows::System::Power;
}

struct MyViewModel : MyViewModelT<MyViewModel>
{
    ...

    winrt::fire_and_forget OnEnergySaverStatusChanged(
        winrt::IInspectable const&, winrt::IInspectable const&)
    {
        auto lifetime = get_strong();
        co_await winrt::resume_foreground(Dispatcher());
        m_isEnergySaverOn = (winrt::PowerManager::EnergySaverStatus()
                             == winrt::EnergySaverStatus::On);
        RaiseProperyChangeEvent(L"IsEnergySaverOn");
    }

    winrt::EnergySaverStatusChanged_revoker m_energySaverStatusChangedToken =
        winrt::PowerManager::EnergySaverStatusChanged(
            { get_weak(), &MyViewModel::OnEnergySaverStatusChanged });
};
```

In the above example, we have a view model that tracks the system Energy Saver status. The change notification for this is raised on a background thread, so we need to switch to the foreground thread before doing our calculations and raising the property-change event.

This pattern is common for many types of notifications: Receive the notification on a background thread and immediately switch to the UI thread to process it. Doing all the work on the UI thread avoids race conditions.

But there's a small problem here: There's a race condition if the foreground thread completes its work before the background thread releases its temporary strong reference.

To make the weak and strong references more explicit, let me rewrite the code this way:

```
struct MyViewModel : MyViewModelT<MyViewModel>
{
    ...

    winrt::EnergySaverStatusChanged_revoker m_energySaverStatusChangedToken =
        winrt::PowerManager::EnergySaverStatusChanged(
            [weak = get_weak()](auto&&, auto&&)
            -> winrt::fire_and_forget {
                if (auto strong = weak.get()) {
                    strong->Dispatcher().RunAsync([strong] {
                        m_isEnergySaverOn = (winrt::PowerManager::EnergySaverStatus()
                                             == winrt::EnergySaverStatus::On);
                        RaiseProperyChangeEvent(L"IsEnergySaverOn");
                    });
                }
            });
};
```

Now we can follow the reference counts. Initially the view model's reference count is 1 because the view has a reference to it.

UI thread	Background thread	Reference count
	EnergySaverStatusChanged	1
	get strong reference	2
	capture strong reference into lambda	3
View tears down		
View model released		2
	RunAsync	
update `m_isEnergySaverOn`		
raise property change event		

destruct strong reference inside lambda		1
	destruct strong reference	0

In this unfortunate sequence of events, the last reference is released on the background thread, and the view model therefore runs its destructor on the background thread, which is bad news because the view model really is a single-threaded object, and its destructor is going to assume that it is running on the UI thread. The reference on the background thread exists for the sole purpose of getting control back to the UI thread, but it ends up being the one holding the bag when everything cleans up.

The fix here is to use `final_release` to get control back to the UI thread for the purpose of destructing there.

```
struct MyViewModel : MyViewModelT<MyViewModel>
{
    ...

    winrt::fire_and_forget final_release(std::unique_ptr<MyViewModel> self)
    {
        co_await winrt::resume_foreground(self->Dispatcher());
        // destruct the object on the UI thread
    }

    ...
};
```

If your implementation class has a method called `final_release`, it will be called with a `std::unique_ptr` holding the object that is about to be destructed. This gives you a chance to do something just before destruction, and a common reason for doing this is to move the `unique_ptr` to the UI thread, so that when it destructs, the underlying object destructs on the UI thread.

Raymond Chen

Follow