What does it even mean to Close a Windows Runtime asynchronous operation or action?

devblogs.microsoft.com/oldnewthing/20240809-00

August 9, 2024

Raymond Chen

Windows Runtime asynchronous operations and actions support the `Close` method. What does that even mean?

There are at least five different implementations of Windows Runtime asynchronous operations and actions, and each one deals with the `Close` operation differently. Let's look at the different implementations and see if we can infer the principle that guides the `Close` method.

C++/WinRT's asynchronous operations and actions have a very simple implementation of `Close`: It does nothing! It doesn't even check whether you called it before the asynchronous operation or action has completed.

Okay, so we didn't learn much from that.

Next up is an internal implementation used by many Windows components, which builds on top of WRL's `AsyncBase`. The `Close` method discards the result of `IAsyncOperation` if the result is a reference type or a string. If you `Close` one of those guys, then the `GetResults()` may return an empty result instead of the actual result.

Okay, so one thing we learned is that once you call `Close`, you can't call `GetResults()` and get reliable results.

Furthermore, the implementation of `AsyncBase` throws an “illegal state change” exception if you try to `Close` the asynchronous operation or action before it has completed. And any method calls or property accesses after you `Close` throw an “illegal method call” exception. So those are some other rules we can remember.

Another implementation ships with Win2D. This is also built on top of WRL’s `AsyncBase`. The Win2D version supports only reference types, and the `Close` method frees the completion result. So its behavior is a subset of the Windows internal implementation. We didn’t learn anything new here.

A fourth implementation can be found in the Parallel Patterns Library (PPL). In this implementation, the `Close` method throws an “illegal state change” exception if the asynchronous action or operation has not yet completed. Assuming that the operation has completed, the `Close` method delegates to the virtual `_OnClose` method, but the default implementation of `_OnClose` does nothing. After the asynchronous operation or action has been closed, you cannot ask for its `Id`, `ErrorCode`, `Status`, or `Progress` handler, you cannot set the `Completed` or `Progress` handler, nor can you call `GetResults()`. (Though it lets you *read* the `Completed` and `Progress` properties.)

The fifth implementation is in the `WindowsRuntimeSystemExtensions` extension class. I don’t have the source code, but I was able to reverse-engineer it with the help of ILSpy. In this implementation, calling `Close` before the asynchronous operation or action has completed throws an exception. If you call it after the completion, then the completion results are freed, as well as the completion handler, the progress handler (if applicable), and any supporting data structures for those things. And after you close the asynchronous operation or action, all future method calls or member accesses throw an exception.

Okay, so combining all of these observations allows us to infer the rules for closing asynchronous operations and actions:

- You may not call `Close` until the operation or action has completed.
- If you call `Close`, then the operation is permitted (but not required) to release any resources associated with the operation or action.
- Once you have `Close`d the operation or action, no method calls or member accesses are permitted.

This seems be a reasonable set of rules, and I don’t see any real opportunities for it to become any stricter, so I think we’ve found it.