The COM static store, part 5: Using COM weak references

devblogs.microsoft.com/oldnewthing/20210212-00

February 12, 2021

Raymond Chen

Last time, we looked at aggregating a bunch of things into a single object so you can access them all at once. That improves the efficiency of accessing objects in the COM static store, but you can do even better.

Use a COM weak reference.

The COM weak reference lets you access the object in the COM static store without having to actually dig through the COM static store to find it. It's basically a shortcut:

COM static store				
…	→	…		
…	→	…		
`"SharedState"`	→	SharedState	←	WeakReference

Instead of having to dig through the COM static store whenever we need to look for our `SharedState` object, we can use our weak reference to go straight to it.

```
// C++/WinRT
struct SharedState :
    winrt::implements<SharedState,
                      winrt::Windows::Foundation::IInspectable>
{
    int some_value = 0;
    winrt::com_ptr<IStream> stream;
    std::vector<winrt::com_ptr<IStorage>> storages;
};

winrt::weak_ptr<SharedState> weakSharedState;

winrt::com_ptr<SharedState>
GetSingletonSharedState()
{
    if (auto state = weakSharedState.get()) {
        return state;
    }

    auto value = winrt::make_self<SharedState>();
    static winrt::slim_mutex lock;
    winrt::slim_lock_guard const guard{ lock };
    if (auto state = weakSharedState.get()) {
        return state;
    }
    props.Insert(name, *value);
    weakSharedState = value;
    return value;
}
```

The weak reference shortcut avoids having to go rummage around in the static store every time we need it. It also adds protection against the case where somebody accidentally overwrites our entry in the static store with an unrelated object: Instead of taking the unrelated object and casting it to our `SharedState`, what happens is that our weak reference fails to resolve, and we go and make a new one.

But wait a second. Isn't this just jumping back into the fire? We're using the COM weak pointer object after COM has been torn down, which is the entire problem we were trying to fix! The important detail is that this is a COM weak pointer implemented in our DLL, that refers to an object in our own DLL. We don't have the problem of calling out to an already-unloaded DLL. The calls stay entirely within our DLL.

On the other hand, this approach still has its downsides. For one thing, the weak reference counts as a COM object. When COM calls your `DllCanUnloadNow` one last time just for funsies, so you can do some cleanup, your function will say, "Eh, don't need to clean up, because I still have active objects (namely, that weak reference)."

Another downside is that resolving a COM weak reference costs you a virtual method call, and releasing the resolved result is another virtual method call. This hampers inlining and other optimizations.

Fortunately, there’s another solution that avoids these problems. We’ll look at it next time.

Raymond Chen

Follow