Processing a ValueSet or PropertySet even in the face of possible mutation, part 3

devblogs.microsoft.com/oldnewthing/20220714-00

July 14, 2022

Raymond Chen

We've looked at a few different ways of one way of saving the contents of a ValueSet or PropertySet while remaining resilient to concurrency modification. A customer tried to implement the first pattern, where each mutation is followed by a processing pass that bails out if it notices a subsequent modification.

A customer tried to implement this pattern, but with a wrinkle: The collection was not self-saving; rather, the collection was part of a larger object that had an explicit `Save` method. They factored the code this way:

```
// Interface declaration
namespace Contoso
{
    runtimeclass Widget
    {
        Windows.Foundation.Collections.PropertySet ExtendedProperties { get; };
        void Save();
    }
}

// C++/WinRT code-behind
struct Widget : WidgetT<Widget>
{
    winrt::PropertySet ExtendedProperties() { return m_propertySet; }
    void Save();

    winrt::PropertySet m_propertySet;
};
```

The `Save` method followed the "abandon on failure" pattern:

```
void Widget::Save()
{
    try {
        SomeKindOfDataBuffer buffer;
        auto it = m_propertySet.First();
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        auto guard = m_lock.lock();

        // verify that the collection is still unchanged before saving
        std::ignore = it.HasCurrent();

        SaveToFile(buffer);
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        // The mutating thread will do its own Save.
        return;
    }
}
```

(Observe that this is a direct copy/pasta from our first pattern.)

Was this the correct implementation?

No.

In the two patterns discussed so far, the object was auto-saving. Therefore, if there was a conflicting modification, we know that the modifying thread will also perform its own `Save()`, and we could therefore just abandon the work, knowing that the other thread will assume responsibility for saving. (If that other thread subsequently fails due to a conflict modification, then the responsibility transfers to whoever made the conflicting modification. Eventually, the last modification will make it all the way to the end and save the collection for real.)

But the `Widget` object is not auto-saving. If the `Save` fails due to a concurrent modification, the conflicting thread is not going to "take over" the `Save` operation since there is no `Save` happening on the conflicting thread.

For this pattern, you have to decide what your object's policy is if a conflicting modification is made during a `Save`.

One option you might choose is that `Save()` silently fails if a concurrent modification occurs, under the expectation that the mutating thread will eventually perform its own `Save()` to bring things back in sync. Though you might want to change it so that the `Save()` method reports whether the save was interrupted.

```
bool Widget::TrySave()
{
    try {
        SomeKindOfDataBuffer buffer;
        auto it = m_propertySet.First();
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        auto guard = m_lock.lock();

        // verify that the collection is still unchanged before saving
        std::ignore = it.HasCurrent();

        SaveToFile(buffer);
        return true;
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        // The mutating thread willshould do its own Save.
        return false;
    }
}
```

This model assumes that everybody will want to (eventually) save their changes. Consider this guy:

```
    Widget widget;

    // Temporarily mark it as busy.
    widget.ExtendedProperties().Insert(L"IsBusy", winrt::box_value(true));

    DoSomething(widget);

    // Okay, not busy any more.
    widget.ExtendedProperties().Remove(L"IsBusy");
```

This caller has no intention of saving the changes. But their temporary modification of the collection may have prevent somebody else from saving. Then again, if the timing were different, their temporary modifications would have been saved by mistake! You get to decide if this is a pattern you want to follow.

Another option is to say make it forbidden to modify the `Widget` 's properties while it is being saved. In that case, you would just remove all the concurrent modification protection from the `Save` method and let the "changed state" exception propagate to the caller:

```
void Widget::Save()
{
    // try {
        SomeKindOfDataBuffer buffer;
        auto it = m_propertySet.First();
        if (it.HasCurrent()) {
            do {
                auto current = it.Current();
                buffer.AddKeyAndValue(current.Key(), current.Value());
            } while (it.MoveNext());
        }

        // auto guard = m_lock.lock();

        // verify that the collection is still unchanged before saving
        // std::ignore = it.HasCurrent();

        SaveToFile(buffer);
    // } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        // The mutating thread will do its own Save.
        // return;
    // }
}
```

The only reason for manually iterating was so we could make a final `HasCurrent()` check at the end, but since that got deleted, we can return to the simple version:

```
void Widget::Save()
{
    SomeKindOfDataBuffer buffer;
    for (auto [name, value] : m_propertySet)
        buffer.AddKeyAndValue(name, value);
    }

    SaveToFile(buffer);
}
```

If a concurrent modification occurs, then the `Save()` method fails with a "changed state" exception, which tells the caller, "You broke the rules and modified the collection while it was saving. Shame on you!"

Note that there is a tiny window where the concurrent modification is not detected, if it happens after we build the buffer and before we save it. That's not a problem, because as far as the caller can tell, the mutation could very well have occurred at the `ret` instruction at

the end of the `SaveToFile` function. (The caller has no insight into the moment in time after the buffer is built and the buffer being saved.)

Yet another pattern is to say that if you mutate the collection during a `Save`, then you can't predict whether it will save the pre-mutation version or post-mutation version, but will always save *something*. In that case, we need to back out and retry if a concurrent mutation is encountered.

```
void Widget::Save()
{
    while (true) {
        try {
            SomeKindOfDataBuffer buffer;
            for (auto [name, value] : m_propertySet)
                buffer.AddKeyAndValue(name, value);
            }

            SaveToFile(buffer);

            return;

        } catch (winrt::hresult_changed_state const&) {
            // Abandon the operation and try again.
        }
    }
}
```

The customer also had another scenario where they needed to process a PropertySet in the fact of concurrent mutation. We'll look at that next time.

Raymond Chen

Follow