Processing a ValueSet or PropertySet even in the face of possible mutation, part 1

devblogs.microsoft.com/oldnewthing/20220712-00

July 12, 2022

Raymond Chen

We've been looking at how you can non-intrusively monitor changes to a ValueSet or PropertySet. Typically this is so that you can take any changes made by the client and propagate them somewhere.

Let's say that you want to take the modified collection and save the whole thing to disk. How can you do this?

Well, your first attempt might be to do this:

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

Iterating over the wrapped collection and saving the results is a good start, but the iterator will throw `hresult_changed_state` if the collection changes during the iteration.

What we need to do is capture the collection, and when we succeed in capturing it all, we can save it. The copy can't mutate since we haven't given anybody else access to it, so iterating over it is safe from an `hresult_changed_state` exception.

```
void MyPropertySet::Save()
{
    auto copy = CapturePropertySet(m_propertySet);

    SomeKindOfDataBuffer buffer;
    for (auto [key, value] : copy) {
        buffer.AddKeyAndValue(key, value);
    }
    SaveToFile(buffer);
}
```

I'm assuming here that converting the property set to some kind of data buffer is a slow operation, which is why it's done as a separate pass over the captured data.

One way to capture the property set would be to transfer it into another property set:

```
auto CapturePropertySet(winrt::PropertySet const& propertySet)
{
    winrt::PropertySet copy;
    for (auto [key, value] : m_propertySet) {
        copy.Insert(key, value);
    }
    return copy;
}
```

Alternatively, since we really just want to capture the key/value pairs, we could just save the key/value pairs:

```
auto CapturePropertySet(winrt::PropertySet const& propertySet)
{
    return std::vector(begin(propertySet), end(propertySet));
}
```

Okay, so we ensured that the collection doesn't change while we're saving it, but what if it mutates while we're copying it? In that case, the `hresult_changed_state` exception occurs, and the `Save()` fails with an exception.

You probably don't want to propagate this exception back to the caller, because they have no idea that this is even happening. They added a property to the property set, and on another thread, they added another property to the same property set, and somehow the first thread gets a `hresult_changed_state` exception. What state changed? What did they do wrong?

They didn't do anything wrong. The problem is in your `Save` code.

Let's catch the exception and quietly abandon the `Save` operation. The idea here is that the `hresult_changed_state` exception occurs if another thread updated the property set after we started saving it. In that case, we should abandon our attempt to save the property set and let that other thread save it.

```
void MyPropertySet::Save()
{
    winrt::PropertySet copy{ nullptr };
    try {
        copy = CapturePropertySet(m_propertySet);
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        // The mutating thread will do its own Save.
        return;
    }

    SomeKindOfDataBuffer buffer;
    for (auto [key, value] : copy) {
        buffer.AddKeyAndValue(key, value);
    }
    SaveToFile(buffer);
}
```

As I mentioned earlier, I'm assuming here that we are converting the property set to a data buffer as a separate pass because it is slow. If the conversion is fast, you may as well do it while iterating:

```
void MyPropertySet::Save()
{
    SomeKindOfDataBuffer buffer;
    try {
        for (auto [key, value] : m_propertySet) {
            buffer.AddKeyAndValue(key, value);
        }
    } catch (winrt::hresult_changed_state const&) {
        // Abandon the operation.
        // The mutating thread will do its own Save.
        return;
    }
    SaveToFile(buffer);
}
```

There is still a problem here, though. Consider this sequence of events:

Thread 1	Thread 2
Insert	
Save	
Build the buffer	
	Insert
	Save

	Build the buffer
	SaveToFile
	Save returns
SaveToFile	
Save returns	

Both attempts to capture the data in the property set succeed because the property set did not change during the capture of the property set into the buffer. However, the second capture was able to race ahead of the first one, which means that the latest saved copy from Thread 2 gets overwritten by the stale copy in Thread 1.

One idea here is have the `Save` function make one last check before saving that what it saved is still the latest copy. To avoid the race between the final check and the `SaveToFile`, we will need a lock.

```
void MyPropertySet::Save()
{
    SomeKindOfDataBuffer buffer;
    try {
        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;
    }

}
```

After we build the results in the data buffer, we enter the lock and make one final check that the collection hasn't changed. The return value is not what we are interested in, since we know that it will return `false` if it returns at all, seeing as we iterated to the end of the collection in the preceding loop. What we are interested in is checking whether it will throw

an exception due to the collection having been mutated. Assigning to `std::ignore` is the same as throwing the value away, except it avoids a `[[nodiscard]]` warning and is arguably clearer that discarding the value is intentional.

Thread 1	Thread 2
Insert	
Save	
Build the buffer	
	Insert
	Save
	Build the buffer
enter lock	enter lock

If Thread 1 wins the race to enter the lock, then the final `HasCurrent()` check will throw `hresult_changed_state`, and the `SaveToFile` will not happen. Thread 2 then gets a chance to save, and its test of `HasCurrent()` will not throw, so it is the one that gets to perform the `SaveToFile`.

On the other hand, if Thread 2 wins the race to enter the lock, then Thread 2's `HasCurrent()` will not throw, so it will perform `SaveToFile`. And then Thread 1 gets the lock and checks `HasCurrent()`, which throws, so Thread 1 does not save its now-outdated data.

There is also a race condition where there is a redundant save:

Thread 1	Thread 2
Insert	
	Insert
Save	
	Save
Build the buffer	
	Build the buffer
enter lock	

HasCurrent does not throw	
SaveToFile	
exit lock	
	enter lock
	HasCurrent does not throw
	SaveToFile
	exit lock

Since Thread 1 got off to a late start, it started building the data buffer after Thread 2 already snuck in and changed the property set, so it unwittingly created an up-to-date copy. At least here the race condition is harmless, albeit perhaps inefficient.

The model here is that each `Save` operation tries to save as fast as it can, but bails out if it discovers that it is not the winner. This means that the Save method's running time is basically the time it takes to serialize and save the property set once.

Next time, we'll look at another solution to the concurrency problem which has its own separate advantages and disadvantages.

Raymond Chen

Follow