How can I provide a Windows Runtime ValueSet or PropertySet while non-intrusively monitoring changes to it?

devblogs.microsoft.com/oldnewthing/20220711-00

July 11, 2022

Raymond Chen

We saw last time that the system-provided implementations of ValueSet and PropertySet support IObservableMap, but adding an observer is an intrusive operation because the collection is temporarily made read-only while the handlers are called. If your component is secretly monitoring changes (say because it wants to propagate any changes to a backing store), then the presence of the handler could coincide with a client write operation, causing the client to receive an unexpected error which will probably result in an application fatal exit.

One solution is to switch to the C++/WinRT multi-threaded collection. This allows write operations to occur concurrently with change notifications.

We start with a plain fake `PropertySet` that doesn't do anything interesting yet.

```
namespace winrt
{
    using namespace winrt::Windows::Foundation;
    using namespace winrt::Windows::Foundation::Collections;
}

struct MyPropertySet : winrt::implements<MyPropertySet,
    winrt::IPropertySet,
    winrt::IMap<winrt::hstring, winrt::IInspectable>,
    winrt::IIterable<winrt::IKeyValuePair<winrt::hstring, winrt::IInspectable>>,
    winrt::IObservableMap<winrt::hstring, winrt::IInspectable>>
{
    // save some typing
    using MapChangedEventHandler =
        winrt::MapChangedEventHandler<winrt::hstring, winrt::IInspectable>;

    winrt::hstring GetRuntimeClassName()
    {
        return winrt::hstring(winrt::name_of<winrt::PropertySet>());
    }

    auto Lookup(winrt::hstring const& key) { return m_propertySet.Lookup(key); }
    auto Size()                            { return m_propertySet.Size(); }
    auto HasKey(winrt::hstring const& key) { return m_propertySet.HasKey(key); }
    auto GetView()                         { return m_propertySet.GetView(); }
    auto Insert(winrt::hstring const& key, winrt::IInspectable const& value)
                                           { return m_propertySet.Insert(key, value);
}
    auto Remove(winrt::hstring const& key) { return m_propertySet.Remove(key); }
    auto Clear()                           { return m_propertySet.Clear(); }
    auto First()                           { return m_propertySet.First(); }
    auto MapChanged(MapChangedEventHandler const& handler)
                                           { return
m_propertySet.MapChanged(handler); }
    auto MapChanged(winrt::event_token const& token)
                                           { return m_propertySet.MapChanged(token);
}

    winrt::IObservableMap<winrt::hstring, winrt::IInspectable> m_propertySet =
        winrt::multi_threaded_observable_map<winrt::hstring, winrt::IInspectable>();
};
```

All of this work is necessary only because the `multi_threaded_observable_map` does not implement `IPropertySet` in the special case where the key/value pair is `hstring, IInspectable`. If the multi-threaded observable map had implemented `IPropertySet` in that case, then we could just use it directly.

But since we're here, we can make it fancy by listening to the `MapChanged` event on the inner object. And the C++/WinRT implementation of multi-threaded observable maps allows other threads to mutate the collection while the `MapChanged` event is in progress. This lets

us respond to the change without interfering with anything the app is doing.

```
struct MyPropertySet : winrt::implements<MyPropertySet,
    winrt::IPropertySet,
    winrt::IMap<winrt::hstring, winrt::IInspectable>,
    winrt::IIterable<winrt::IKeyValuePair<winrt::hstring, winrt::IInspectable>>,
    winrt::IObservableMap<winrt::hstring, winrt::IInspectable>>
{
    // save some typing
    using MapChangedEventHandler =
        winrt::MapChangedEventHandler<winrt::hstring, winrt::IInspectable>;

    winrt::hstring GetRuntimeClassName()
    {
        return winrt::hstring(winrt::name_of<winrt::PropertySet>());
    }

    auto Lookup(winrt::hstring const& key) { return m_propertySet.Lookup(key); }
    auto Size()                            { return m_propertySet.Size(); }
    auto HasKey(winrt::hstring const& key) { return m_propertySet.HasKey(key); }
    auto GetView()                         { return m_propertySet.GetView(); }
    auto Insert(winrt::hstring const& key, winrt::IInspectable const& value)
                                           { return m_propertySet.Insert(key, value);
}
    auto Remove(winrt::hstring const& key) { return m_propertySet.Remove(key); }
    auto Clear()                           { return m_propertySet.Clear(); }
    auto First()                           { return m_propertySet.First(); }
    auto MapChanged(MapChangedEventHandler const& handler)
                                           { return
m_propertySet.MapChanged(handler); }
    auto MapChanged(winrt::event_token const& token)
                                           { return m_propertySet.MapChanged(token);
}

    void OnMapChanged(winrt::IObservableMap<hstring, IInspectable> const&,
        winrt::IMapChangedEventArgs<hstring> const&)
    {
        ⟦ do stuff ⟧
    }

    MyPropertySet(MyPropertySet const&) = delete;
    void operator=(MyPropertySet const&) = delete;

    ~MyPropertySet() { m_propertySet.MapChanged(m_changedToken); }

    winrt::IObservableMap<winrt::hstring, winrt::IInspectable> m_propertySet =
        winrt::multi_threaded_observable_map<winrt::hstring, winrt::IInspectable>();

    winrt::event_token m_changedToken =
        m_propertySet.MapChanged({ get_weak(), &MyPropertySet::OnMapChanged });
};
```

There's a small problem here: If the client takes the `MPropertySet` and registers their own `MapChanged` event handler, then it's unspecified which handler runs first. Maybe we want to make sure our handler runs first (or last). To do that, we can wrap the event, too.

```
struct MyPropertySet : winrt::implements<MyPropertySet,
    winrt::IPropertySet,
    winrt::IMap<winrt::hstring, winrt::IInspectable>,
    winrt::IIterable<winrt::IKeyValuePair<winrt::hstring, winrt::IInspectable>>,
    winrt::IObservableMap<winrt::hstring, winrt::IInspectable>>
{
    // save some typing0
    using MapChangedEventHandler =
        winrt::MapChangedEventHandler<winrt::hstring, winrt::IInspectable>;

    winrt::hstring GetRuntimeClassName()
    {
        return winrt::hstring(winrt::name_of<winrt::PropertySet>());
    }

    auto Lookup(winrt::hstring const& key) { return m_propertySet.Lookup(key); }
    auto Size()                            { return m_propertySet.Size(); }
    auto HasKey(winrt::hstring const& key) { return m_propertySet.HasKey(key); }
    auto GetView()                         { return m_propertySet.GetView(); }
    auto Insert(winrt::hstring const& key, winrt::IInspectable const& value)
                                           { return m_propertySet.Insert(key, value);
}
    auto Remove(winrt::hstring const& key) { return m_propertySet.Remove(key); }
    auto Clear()                           { return m_propertySet.Clear(); }
    auto First()                           { return m_propertySet.First(); }
    auto MapChanged(MapChangedEventHandler const& handler)
                                           { return m_mapChanged.add(handler); }
    auto MapChanged(winrt::event_token const& token)
                                           { return m_mapChanged.remove(token); }

    void OnMapChanged(winrt::IObservableMap<hstring, IInspectable> const&,
        winrt::IMapChangedEventArgs<hstring> const& args)
    {
        ⟦ do stuff before notifying clients ⟧
        m_mapChanged(*this, args);
        ⟦ do stuff after notifying clients ⟧
    }

    MyPropertySet(MyPropertySet const&) = delete;
    void operator=(MyPropertySet const&) = delete;

    ~MyPropertySet() { m_propertySet.MapChanged(m_changedToken); }

    winrt::event<MapChangedEventHandler&;gt m_mapChanged;

    winrt::IObservableMap<winrt::hstring, winrt::IInspectable> m_propertySet =
        winrt::multi_threaded_observable_map<winrt::hstring, winrt::IInspectable>();

    winrt::event_token m_changedToken =
        m_propertySet.MapChanged({ get_weak(), &MyPropertySet::OnMapChanged });
};
```

Instead of letting clients register directly on the inner observable map, we have them register against our own event. That way, there is only one event handler on the inner observable map, namely our own `OnMapChanged`. In that method, we can choose whether our work happens before or after notifying the clients.

Things get complicated if we want to wrap a `ValueSet`: Although it has the same interface as `PropertySet`, the `ValueSet` imposes an additional requirement that the `IInspectable` be a serializable type. Adding this enforcement to our reimplementation of `ValueSet` is going to be annoying, but it turns out that we can sidestep the problem entirely: Now that we have our own custom event, we may as well use it for everything.

```
struct MyPropertySet : winrt::implements<MyPropertySet,
    winrt::IPropertySet,
    winrt::IMap<winrt::hstring, winrt::IInspectable>,
    winrt::IIterable<winrt::IKeyValuePair<winrt::hstring, winrt::IInspectable>>,
    winrt::IObservableMap<winrt::hstring, winrt::IInspectable>>
{
    // save some typing
    using MapChangedEventHandler = winrt::MapChangedEventHandler<winrt::hstring,
winrt::IInspectable>;

    winrt::hstring GetRuntimeClassName()
    {
        return winrt::hstring(winrt::name_of<winrt::PropertySet>());
    }

    auto Lookup(winrt::hstring const& key) { return m_propertySet.Lookup(key); }
    auto Size()                            { return m_propertySet.Size(); }
    auto HasKey(winrt::hstring const& key) { return m_propertySet.HasKey(key); }
    auto GetView()                         { return m_propertySet.GetView(); }
    auto Insert(winrt::hstring const& key, winrt::IInspectable const& value) {
        auto result = return m_propertySet.Insert(key, value);
        ReportChange(CollectionChange::ItemInserted, key);
        return result;
    }
    auto Remove(winrt::hstring const& key) {
        m_propertySet.Remove(key);
        ReportChange(CollectionChange::ItemRemoved, key);
    }
    auto Clear() {
        m_propertySet.Clear();
        ReportChange(CollectionChange::Reset);
    }
    auto First()                           { return m_propertySet.First(); }
    auto MapChanged(MapChangedEventHandler const& handler)
                                           { return m_mapChanged.add(handler); }
    auto MapChanged(winrt::event_token const& token)
                                           { return m_mapChanged.remove(token); }

    MyPropertySet(MyPropertySet const&) = delete;
    void operator=(MyPropertySet const&) = delete;

    ~MyPropertySet() { m_propertySet.MapChanged(m_changedToken); }

    struct Args : winrt::implements<Args, IMapChangedEventArgs<winrt::hstring>>
    {
        Args(winrt::CollectionChange change, winrt::hstring const& key) :
m_change(change), m_key(key) {}
        auto CollectionChange() { return m_change; }
        auto Key() { return m_key; }
        winrt::CollectionChange m_change;
        winrt::hstring m_key;
    };
```

```
    void ReportChange(winrt::CollectionChange change, winrt::hstring key = {})
    {
        ⟦ do stuff before notifying clients ⟧
        m_mapChanged(*this, winrt::make<Args>(change, key));
        ⟦ do stuff after notifying clients ⟧
    }

    winrt::event<MapChangedEventHandler&;gt m_mapChanged;

    winrt::PropertySet m_propertySet;

    // winrt::event_token m_changedToken =
    //  m_propertySet.MapChanged({ get_weak(), &MyPropertySet::OnMapChanged });
};
```

We don't need to register for the `MapChanged` event to detect a change to the collection, because we are controlling access to the collection ourselves and therefore know when the mutation has occurred. We can then perform our preprocessing, raise the event for clients, and then do our post-processing.

And using this pattern solves the `ValueSet` problem: Since the inner collection really is a `ValueSet`, we can let the `ValueSet` do its own validation.

So far, we've just been hand-waving over the "do stuff" parts. Next time, we'll look into the dangers lurking there.

Raymond Chen

Follow