How to clone a Windows Runtime map in the face of possible concurrent modification, part 1

devblogs.microsoft.com/oldnewthing/20230719-00

July 19, 2023

Raymond Chen

So far, we've been looking at cloning a Windows Runtime vector in the face of possible concurrent modification. We can apply what we've learned to Windows Runtime maps.

The idea is basically the same: Use `GetMany` to capture the contents atomically. The wrinkle is that there is no `GetMany` method on a Windows Runtime map. We'll have to get the iterator and call the iterator's `GetMany` method. But going to the iterator means that we could get a `hresult_changed_state` exception if the map mutates between the time we obtain the iterator and the time we try to read from it.[1]

```
template<typename M>
auto clone_as_map(M const& m)
-> std::map<
        decltype(m.First().Current().Key()),
        decltype(m.First().Current().Value())>
{
    using KVP = decltype(m.First().Current());
    std::vector<KVP> pairs;
    using K = decltype(KVP().Key());
    using V = decltype(KVP().Value());
    uint32_t expected;
    uint32_t actual;
    do {
        expected = m.Size();
        pairs.resize(expected + 1);
        try {
            actual = m.First().GetMany(pairs);
        } catch (winrt::hresult_changed_state const&) {
            continue;
        }
    } while (actual > expected);
    pairs.erase(pairs.begin() + actual, pairs.end());
    std::map<K, V> map;
    for (auto&& pair : pairs) {
        map.emplace(pair.Key(), pair.Value());
    }
    return map;
}

template<typename M>
auto CloneMap(M const& m)
-> winrt::Windows::Foundation::
        Collections::IMap<
            decltype(m.First().Current().Key()),
            decltype(m.First().Current().Value())>
{
    return winrt::multi_threaded_map(
        clone_as_map(m));
}
```

Note that we don't need to use `winrt_empty_value<T>()` when resizing the `pairs` vector because the value type of the vector is `IKeyValuePair<K, V>`, which is an interface type and therefore has a default constructor that creates an empty smart pointer.

Note also that I provide an explicit trailing return type instead of allowing the compiler to infer it from the `return` statement. This allows the return type to participate in SFINAE. Unfortunately, representing this trailing return type is rather cumbersome because we have to write it out without the assistance of the `K` and `V` types defined in the function body.

The C++/WinRT library allows you to create `IMap` implementations that are based either on `std::map` or `std::unordered_map`. The version above always uses `std::map` with the same key and value as the original. But maybe we want to let you choose a `std::unordered_map` or tweak the key and value types.

We can use type deduction from the future to default to using a `std::map` that matches the inbound parameter, but let you pick a different collection if you like.

```
template<typename, typename = void>
struct has_reserve_member : std::false_type {};
template<typename T>
struct has_reserve_member<T,
    std::void_t<decltype(
        std::declval<T>().reserve(0))>>
    : std::true_type {};

template<typename R = void, typename M>
auto clone_as_map(M const& m)
-> using C = std::conditional_t<
    std::is_same_v<R, void>,
    std::map<K, V>,
    R>
{
    using KVP = decltype(m.First().Current());
    std::vector<KVP> pairs;
    using K = decltype(KVP().Key());
    using V = decltype(KVP().Value());
    uint32_t expected;
    uint32_t actual;
    do {
        expected = m.Size();
        pairs.resize(expected + 1);
        try {
            actual = m.First().GetMany(pairs);
        } catch (hresult_changed_state const&) {
            continue;
        }
    } while (actual > expected);
    pairs.resize(actual);
    using C = std::conditional_t<
        std::is_same_v<R, void>,
        std::map<K, V>,
        R>;
    C map;
    if constexpr (has_reserve_member<C>::value) {
        map.reserve(actual);
    }
    for (auto&& pair : pairs) {
        map.emplace(pair.Key(), pair.Value());
    }
    return map;
}

template<typename R = void, typename M>
auto CloneAsMap(M const& m)
-> winrt::Windows::Foundation::
        Collections::IMap<
            decltype(m.First().Current().Key()),
            decltype(m.First().Current().Value())>
{
```

```
    return winrt::multi_threaded_map(
        clone_as_map<R>(m));
}
```

We create a detector class to see whether the underlying type has a `reserve` method (which `unordered_map` has); if so, then we call it to allow the container to prepare for the incoming elements.

An example usage of this new feature would be

```
winrt::Windows::Foundation::Collections::
    IMap<winrt::hstring, int32_t> v = /* ... */;
auto clone = CloneAsMap<
    std::unordered_map<winrt::hstring, int64_t,
        case_insensitive_hash, case_insensitive_equal>>
    (v);
```

This takes the original `IMap` and clones it into a new `IMap` which uses `int64_t` as its value type rather than `int32_t` (said conversion occuring through integer promotion), and which uses a custom hash class and custom equality class which treat the string as a case-insensitive string. We changed the underlying container to an `unordered_map`, changed the value type, and chose non-default hash and equality classes.

Another thing we might do to this function is add a lambda that lets you convert the original key-value pair to an arbitrary `std::pair`, but arguably that's really a job for `std::transform`.

But maybe we're working too hard. We'll try another approach next time.

[1] Note, however, that the iterator is not *required* to throw a state changed exception. So we cannot rely on the exception to tell us that a concurrent mutation has occurred.