Do any Windows Runtime projections cache nondefault Windows Runtime interfaces?

devblogs.microsoft.com/oldnewthing/20201029-00

October 29, 2020

Raymond Chen

Some time ago, I discussed how Windows Runtime language projections call methods on nondefault interfaces. Do any projections cache nondefault interfaces?

The C++/CX and C++/WinRT projections represent Windows Runtime objects as a reference-counted pointer to the object's default interface. As a result, there's nowhere to cache the nondefault interfaces, since the only storage available is the pointer itself.[1]

The C# and JavaScript projections wrap the Windows Runtime inside native C# and JavaScript objects, and in that case, there's plenty of room to do things like cache nondefault interfaces in that native object.

If C++/CX or C++/WinRT wanted to break the rule that their projections are just wrappers around a single pointer, they could have made the projection be its own object. One option would be to have it passed by value:

```
class Thing
{
private:
  ComPtr<ABI::IThing> thing;
  ComPtr<ABI::IThing2> mutable thing2;

public:
  // Method on default interface IThing
  void Method1() const
  {
    ThrowIfFailed(thing->Method1());
  }

  // Method on nondefault interface IThing2
  void Method2() const
  {
    // Ignoring thread-safety for expository simplicity
    if (!thing2) {
      ThrowIfFailed(thing->
        QueryInterface(IID_PPV_ARGS(&thing2)));
    }
    ThrowIfFailed(thing2->Method2());
  }

  bool operator==(Thing const& other) const noexcept
  {
    return thing == other.thing;
  }
};
```

The downside of this implementation is that the object is now larger than a pointer, so it gets more expensive to pass around. The projection would become very large for a class like `UIElement` with seventeen interfaces. If you wanted to copy the cache when the object is copied, that's potentially sixteen extra `AddRef` calls when the parameter is passed. And regardless, you have up to sixteen extra `Release` calls when the parameter is destroyed.

Another option is to make the projection similar to the C# and JavaScript projections and have the projected object be a reference to a hidden wrapper.

```
class ThingImpl
{
private:
  ComPtr<ABI::IThing> thing;
  ComPtr<ABI::IThing2> mutable thing2;

public:
  // Method on default interface IThing
  void Method1() const
  {
    ThrowIfFailed(thing->Method1());
  }

  // Method on nondefault interface IThing2
  void Method2() const
  {
    // Ignoring thread-safety for expository simplicity
    if (!thing2) {
      ThrowIfFailed(thing->
        QueryInterface(IID_PPV_ARGS(&thing2)));
    }
    ThrowIfFailed(thing2->Method2());
  }
};

class Thing
{
private:
  std::shared_ptr<ThingImpl> impl;

  IThing* get_raw_pointer() const
  {
    return impl ? impl.get()->thing : nullptr;
  }

public:
  void Method1() const { return impl->Method1(); }
  void Method2() const { return impl->Method2(); }

  bool operator==(Thing const& other) const noexcept
  {
    return get_raw_pointer() == other.get_raw_pointer());
  }
};
```

In this case, the projected object is just a `std::shared_ptr` to a shared cache. This copies relatively quickly, since it's just bumping a reference count on the control block. The downside is that it costs an extra allocation each time a new wrapper is created.[2] (Copying a wrapper just copies the inner `shared_ptr`, but creating a new wrapper requires the creation of a new `shared_ptr`.)

C++/CX and C++/WinRT chose to make the projected object be a direct pointer to the Windows Runtime object's default interface. It's smaller, avoids extra allocations, and makes converting between projected and ABI types simpler. The cost is that members of nondefault interfaces are more expensive.

[1] Even if the cached interface were derived from the default interface (which doesn't happen in the Windows Runtime, but work with me here), you couldn't "upgrade" the pointer to the derived interface because you would have no way of knowing later whether that pointer is a boring default interface or an upgraded cached interface.

[2] C# (and I'm guessing probably JavaScript) will reuse a wrapper when it needs to wrap a Windows Runtime object and finds that a wrapper already exists. The C++ projections could have done this, noting that the lookup table would have to be per-module.

Raymond Chen

Follow