Why am I getting a null pointer crash when trying to call a method on my C++/WinRT object?

devblogs.microsoft.com/oldnewthing/20220901-00

September 1, 2022

Raymond Chen

A customer found that their program ran fine on their machine, but it crashed when run on a Windows Server 2019 system.

```
namespace winrt
{
    using namespace winrt::Windows::Web::Http;
}

winrt::HttpClient httpClient;

// Crashes on the next line
auto result = co_await httpClient.TryGetStringAsync(L"http://example.com/");
if (result.Succeeded()) { /* use the result */ }
```

An inspection of the crash dump shows that we crashed inside the C++/WinRT projection:

```
 check_hresult(WINRT_IMPL_SHIM(Windows::Web::Http::IHttpClient2)->
    TryGetStringAsync(*(void**)(&uri), &operation));
```

The `WINRT_IMPL_SHIM` is producing a null pointer, and therefore the attempt to call the `TryGetStringAsync` method crashes.

The `WINRT_IMPL_SHIM` internal macro converts the `this` object into the appropriate interface so we can call it. And it does this via the conversion operator:

```
template <typename D, typename I>
struct require_one : consume_t<D, I>
{
    operator I() const noexcept
    {
        return static_cast<D const*>(this)->template try_as<I>();
    }
};
```

Observe that the conversion operator uses `try_as`, which means that if the `QueryInterface` fails, then it returns `nullptr`.

And that's the problem.

The `TryGetStringAsync` method was added to the `HttpClient` object in Windows 10, version 1903 (10.0.18362.0), and Windows Server 2019 corresponds to Windows 10, Version 1809 (10.0.17763.0). The `QueryInterface` call fails because the `IHttpClient2` interface is not supported on Windows Server 2019.

What happened is that the customer was using the Windows Runtime metadata files corresponding to a more recent version of Windows than their target platform. You are allowed to do that, but it also means that you have to be careful when trying to do things on older systems. You have this same problem in classic Win32, where you have to avoid using new versions of structures or calling new functions when running on older systems.

There are two general approaches for detecting whether a Windows Runtime feature is supported.

The first is to ask the system whether the method exists, using the `ApiInformation` runtime class:

```
namespace winrt
{
    using namespace winrt::Windows::Web::Http;
    using namespace winrt::Windows::Foundation::Metadata;
}

winrt::HttpClient httpClient;

winrt::hresult error;
winrt::hstring stringResult;
if (winrt::ApiInformation::IsMethodPresent(
        winrt::name_of<winrt::HttpClient>,
        L"TryGetStringAsync", 1)) {
    auto result = co_await client.TryGetStringAsync(L"http://example.com/");
    if (result.Succeeded()) stringResult = result.Value();
    else error = result.ExtendedError();
} else {
    // Running on older system that doesn't support TryGetStringAsync
    try {
        stringResult = co_await httpClient.GetStringAsync(L"http://example.com");
    } catch (...) {
        error = winrt::to_hresult();
    }
}
if (!error) { /* do something with stringResult */ }
else { /* deal with the error */ }
```

The above version probes by the method name and arity. "Is there a method on `HttpClient` called `TryGetAsync` that takes one parameter?" The arity parameter is optional, and if you omit it, then you are checking if a method by that name exists regardless of arity. But since we know we're calling the 1-parameter version, we should be specific and look for the 1-parameter version.

Alternatively, you can look up in the documentation which interface supports `TryGetString-Async` and probe for the presence of the interface. A bonus static assertion tells the compiler to check our work.

```
static_assert(&winrt::IHttpClient2::TryGetStringAsync);
if (winrt::ApiInformation::IsTypePresent(
        winrt::name_of<winrt::IHttpClient2>)) {
```

Another alternative is to look up in the documentation which contract version introduced the method and probe for presence of that version of the contract.

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

if (winrt::ApiInformation::IsApiContractPresent(
        winrt::name_of<winrt::UniversalApiContract>(), 8)) {
```

If you have an older version of C++/WinRT, you can use the string literal `L"Windows.Foundation.UniversalApiContract"`.

The second pattern is to to probe for for the interface that supports the method you want to check.

```
winrt::HttpClient httpClient;

winrt::hresult error;
winrt::hstring stringResult;
static_assert(&winrt::IHttpClient2::TryGetStringAsync);
if (httpClient.try_as<winrt::IHttpClient2>()) {
    auto result = co_await client.TryGetStringAsync(L"http://example.com/");
    if (result.Succeeded()) stringResult = result.Value();
    else error = result.ExtendedError();
} else {
    // Running on older system that doesn't support TryGetStringAsync
    try {
        stringResult = co_await httpClient.GetStringAsync(L"http://example.com");
    } catch (...) {
        error = winrt::to_hresult();
    }
}
if (!error) { /* do something with stringResult */ }
else { /* deal with the error */ }
```

As an optimization, you can save the result of the query and use it to call the method. This avoids a second query inside the projection. Note that if we do it this way, we no longer need a `static_assert` to ask the compiler to check our work. It will check our work when we try to call `client2.TryGetStringAsync()`.

```
winrt::HttpClient httpClient;

winrt::hresult error;
winrt::hstring stringResult;
if (auto client2 = httpClient.try_as<winrt::IHttpClient2>()) {
    auto result = co_await client2.TryGetStringAsync(L"http://example.com/");
    if (result.Succeeded()) stringResult = result.Value();
    else error = result.ExtendedError();
} else {
    // Running on older system that doesn't support TryGetStringAsync
    try {
        stringResult = co_await httpClient.GetStringAsync(L"http://example.com");
    } catch (...) {
        error = winrt::to_hresult();
    }
}
if (!error) { /* do something with stringResult */ }
else { /* deal with the error */ }
```

The story is the same: If an interface is marked as `required` in the metadata, then C++/WinRT assumes that it will always be there. Because that's what "required" means. If you need to support running on a system that fails to satisfy the requirements, you'll have to detect the unmet requirements yourself.

Bonus chatter: Why does C++/WinRT work this way? Why doesn't `require_one` use `as()`, so that you get an `hresult_no_interface` exception instead of a hard crash?

Early versions of C++/WinRT indeed did that: They used `as()` and consequently threw an `hresult_no_interface` exception if the interface was not present. The behavior was changed some time before version 1.0.171013.2 to make the absence of a required interface a fatal error instead of a recoverable one.

As I dimly recall, the reason is that checking the result and throwing a C++ exception was bloating the code. Querying for non-default interfaces happens a lot in Windows Runtime code, and all of the checks and throws were adding significant cost to each method call, as well as generating a lot of exception-handling infrastructure code. A "required" interface should never be missing (assuming you matched your SDK to the operating system), so there was little incentive to check for something whose failure indicates that the operating system and SDK teams messed up badly.

In the less common case that you need to write version-adaptive code, you can add the appropriate checks yourself.

Raymond Chen

Follow