Inside C++/WinRT: Apartment switching: The basic idea

devblogs.microsoft.com/oldnewthing/20230124-00

January 24, 2023

Raymond Chen

One of the features of C++/WinRT is that if you `co_await` an IAsyncAction or `IAsyncOperation`, the C++/WinRT library returns to the original COM apartment before resuming the coroutine. This behavior is generally desirable because you expect that COM objects prior to performing a `co_await` are still usable after it returns.

This task is accomplished with the assistance of `IContextCallback`.

Here's the basic idea:[1]

```
inline int32_t __stdcall resume_apartment_callback(
    com_callback_args* args) noexcept
{
    coroutine_handle<>::from_address(args->data)();
    return 0;
};

void resume_apartment(
    com_ptr<IContextCallback> const& context,
    std::coroutine_handle<> handle)
{
    com_callback_args args{};
    args.data = handle.address();

    check_hresult(
        context->ContextCallback(resume_apartment_callback,
            &args,
            guid_of<ICallbackWithNoReentrancyToApplicationSTA>(),
            5, nullptr));
}
```

To resume a coroutine synchronously in a particular context, we use the `IContext-Callback::ContextCallback` method to ask COM to run a particular function in that desired context. We convert the coroutine handle to a pointer to use as our reference data, and in the callback, we convert the pointer back to a coroutine handle so we can invoke it, thereby resuming the coroutine.

We can use this to build the `apartment_context` object.

```
struct apartment_context
{
    apartment_context() = default;
    apartment_context(std::nullptr_t) : context(nullptr) { }

    operator bool() const noexcept { return context != nullptr; }
    bool operator!() const noexcept { return context == nullptr; }

    com_ptr<IContextCallback> context =
            capture<IContextCallback>(WINRT_IMPL_CoGetObjectContext);
};

struct apartment_awaiter
{
    apartment_context const& context;

    bool await_ready() const noexcept
    {
        return false;
    }

    void await_suspend(coroutine_handle<> handle)
    {
        apartment_context extend_lifetime = context;
        resume_apartment(context.context, handle);
    }

    void await_resume() const noexcept
    {
    }
};

apartment_awaiter operator co_await(apartment_context const& context)
{
    return { context };
}
```

To construct an `apartment_context`, we call `CoGetObjectContext` (through the C++/WinRT alias) to obtain an `IContextCallback`.

There is also a `nullptr` constructor if you want to declare an empty `apartment_context`. Empty contexts aren't usable, but they are useful: They let you declare a variable and initialize it with a proper context later.

To `co_await` an `apartment_context`, we construct an `apartment_awaiter` which remembers the context being awaited, and the `await_suspend` method uses it to call `resume_apartment()`.

We can now add COM context support to our oversimplified Windows Runtime awaiter.

```
template <typename Async>
struct await_adapter
{
    await_adapter(Async const& async) : async(async) { }

    Async const& async;

    bool await_ready() const noexcept
    {
        return false;
    }

    void await_suspend(std::experimental::coroutine_handle<> handle) const
    {
        auto extend_lifetime = async;
        async.Completed([
            handle,
            context = apartment_context()
        ](auto&& ...)
        {
            resume_apartment(context.context, handle);
        });
    }

    auto await_resume() const
    {
        return async.GetResults();
    }
};
```

We capture an `apartment_context` in the lambda and use `resume_apartment()` to resume the coroutine in that captured context.

This code is still flawed, though. We'll continue the discussion next time.

[1] The C++/WinRT library does not `#include <windows.h>`. All of the dependencies on Windows are wrapped inside parallel declarations within the C++/WinRT library. The `com_callback_args` structure, for example, is an ABI-equivalent version of the `ComCallData` structure.