Turning anything into a fire-and-forget coroutine

devblogs.microsoft.com/oldnewthing/20190322-00

March 22, 2019

Raymond Chen

Last time, we wrote a helper function for converting an awaitable into a `winrt::fire_and_forget`, as well as another helper function that takes a lambda that *returns* an awaitable, and which invokes the lambmda as a `winrt::fire_and_forget`.

After I wrote the two functions, I wondered if I could unify them. Mostly because I wanted to use the same name `no_await` for both functions.

This took me down the horrible rabbit hole known as C++ template metaprogramming. I wanted two versions of the function, one that is used if the parameter is awaitable, and another that is used if the parameter is a functor. This led me to try using things like `std::enable_if` to detect which case I'm in, and that led to lots of frustration, especially because there's no easy way to detect if a type is awaitable. My closest approach was

```
template<typename T, typename Promise = std::void_t<>>
struct is_awaitable : std::false_type {};

template<typename T>
struct is_awaitable<T, std::void_t<typename
std::experimental::coroutine_traits<T>::promise_type>> : std::true_type {};

template<typename T>
inline constexpr bool is_awaitable_v = is_awaitable<T>::value;
```

which infers that a type is awaitable by sniffing whether it has an associated `promise_type`. This isn't foolproof, because some types like `winrt::fire_and_forget` have a `promise_type` that cannot be awaited.

My first realization was that I could flip the test. Instead of checking whether the argument is awaitable, I check whether it is invokable.

My second realization was that I didn't have to do fancy template metaprogramming at all. I could take advantage of the new `if constexpr` feature.

```
template<typename T>
fire_and_forget no_await(T t)
{
    if constexpr (std::is_invocable_v<T>)
    {
        co_await t();
    }
    else
    {
        co_await t;
    }
}
```

Now you can use `no_await` with awaitables or functors that return awaitables.

```
void Stuff()
{
  // Start this operation but don't wait for it to finish
  no_await(DoSomethingAsync());

  // Start this sequence of things and don't wait for
  // them to finish.
  no_await([=]() -> IAsyncAction
  {
    co_await Step1Async();
    // Step 2 doesn't start until Step 1 completes.
    co_await Step2Async();
  });
}
```

On the other hand, for the case of the lambda passed to `no_await`, you could just declare your lambda as returning a `winrt::fire_and_forget`, and then you wouldn't need `no_await`.

```
void Stuff()
{
  // Start this operation but don't wait for it to finish
  no_await(DoSomethingAsync());

  // Start this sequence of things and don't wait for
  // them to finish.
  invoke_async_lambda([=]() -> winrt::fire_and_forget
  {
    co_await Step1Async();
    // Step 2 doesn't start until Step 1 completes.
    co_await Step2Async();
  });
}
```

But I like the fact that the first example uniformly uses the name `no_await` to describe the concept of "I'm not going to wait for this thing to finish." And also I'm perhaps unduly attached to the cute name.

Raymond Chen

Follow