C++ coroutines: Cold-start coroutines

devblogs.microsoft.com/oldnewthing/20210421-00

April 21, 2021

Raymond Chen

So far, our coroutine promise has implemented a so-called *hot-start* coroutine, which is one that begins running as soon as it is created. Another model for coroutines is the so-called *cold-start* coroutine, which is one that doesn't start running until it is awaited.

C# and JavaScript use the hot-start model: Creating the coroutine runs the coroutine body synchronously to its first suspension point, and only after the coroutine suspends for the first time is the coroutine returned to the caller. The usual usage pattern is to create the coroutine, and then go do other stuff while the coroutine is running, on the assumption that the synchronous portion of the coroutine is brief, and the expensive portion runs asynchronously. The hot-start model makes it easy to start multiple coroutines in parallel, and await the combined result.

Python uses cold-start coroutines: The coroutine doesn't start running until you await it. With cold-start coroutines, you need other machinery if you want to do work in parallel with the await, although that machinery could be made relative simply, like Python's `create_task` that runs a coroutine in an event loop. Cold-start coroutines have simpler bookkeeping since the *running* and *awaiting* states are identical, which makes a lot of state transitions impossible.

You could also create a hybrid model where the coroutine is cold-start, but can be manually started. Mind you, doing so reintroduces the state transitions you thought you had simplified away.

The C++ language doesn't take a position on whether coroutines are hot-start or cold-start, or some hybrid of the two. It just provides the underlying infrastructure, and it's up to you to decide what you want to build on top of it.

If we define the initial state as *cold*, then our valid state transitions are as follows:

- cold → running → completed → abandoned: This is the common case where the task is awaited and then runs to completion.
- cold → abandoned: This is the case where the coroutine is abandoned without ever starting.

From To	running	completed	abandoned
cold	Resume coroutine		Destroy promise
running		Resume awaiter	
completed			Destroy promise

The nice thing about cold-start coroutines is that there are very few transitions, and none of them are contended. Furthermore, the state is completely implied by the actions of the task, so we don't even need to keep track of it explicitly.

Here's a sketch of the changes we can make to convert our hot-start coroutine promise to cold-start. Don't incorporate these yet, for reasons we'll see next time.

```
template<typename T>
struct simple_promise_base
{
    ...

    std::experimental::coroutine_handle<> m_waiting{ nullptr };
    simple_promise_result_holder<T> m_holder;

    ...

    void abandon()
    {
        destroy();
    }

    std::experimental::suspend_always initial_suspend() noexcept
    {
        return {};
    }

    auto final_suspend() noexcept
    {
        struct awaiter : std::experimental::suspend_always
        {
            simple_promise_base& self;
            void await_suspend(
                std::experimental::coroutine_handle<>)
                const noexcept
            {
                self.m_waiting();
            }
        };
        return awaiter{ {}, *this };
    }

    bool client_await_ready()
    {
        return false;
    }

    auto client_await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
        m_waiting = handle;
        as_handle().resume();
    }

    ...
};
```

What makes this a cold-start coroutine is the fact that the `initial_suspend` is a `suspend_always` rather than a `suspend_never`. This means that the coroutine body doesn't start until the coroutine is explicitly `resume()` d.

The other state transitions are significantly simplified. Destroying the coroutine doesn't need to check whether the coroutine is running, because it happens either before the coroutine even starts, or after it has completed, never when the coroutine is runinng. Completing the coroutine can always resume the `m_waiting` coroutine, since the awaiter registers completion before resuming, so the resumption handle is known to be valid by this point.

The other wrinkle about cold-start coroutines is that the awaiter is responsible for starting it, which we do by calling `resume()`.

You may have noticed an inefficiency here: If the coroutine completes synchronously,[1] then we end up calling into the coroutine's `resume()`, and the completion calls back into the awaiter's `resume()`. This accumulates stack frames, which is a problem for a coroutine that awaits other synchronously-completing coroutines in a loop, since each time through the loop uses another level of stack.

We'll address this problem next time, and it will require us to bring back some of the code we deleted, which is why I warned you not to incorporate it yet.

[1] "But why bother making it a coroutine if it completes synchronously?" The operation might complete synchronously under certain conditions, but asynchronously under other conditions. For example, the operation might complete asynchronously if a helper object needs to be started, but synchronously if the helper object is already up and running.

Raymond Chen

Follow