C++ coroutines: Making the promise itself be the shared state, the outline

devblogs.microsoft.com/oldnewthing/20210405-00

April 5, 2021

Raymond Chen

Last time, we got the idea of putting the result holder state directly inside the coroutine state. This time, we'll set to work on the implementation.

A restriction we are placing on our `simple_task` is that it can be `co_await`ed only once. This enables the return of a move-only object, and avoid potentially-expensive copy operations. It also discourages some inefficient usage patterns, which we'll discuss later.

I'll present the code without some of the annoying bits, and then we'll spend the next few days filling it in. The code is conceptually simple, but there's a lot of paperwork. Placeholders are marked with ⟦brackets⟧.

```
namespace async_helpers
{
    template<typename T> struct simple_task;
}
namespace async_helpers::details
{
    template<typename T> struct simple_promise;

    ⟦simple_promise_result_holder definition⟧

    template<typename T>
    struct simple_promise_base
    {
        std::atomic<uint32_t> m_refcount{ 2 };
        std::mutex m_mutex;
        std::experimental::coroutine_handle<> m_waiting{ nullptr };
        simple_promise_result_holder<T> m_holder;

        using Promise = simple_promise<T>;
        auto as_promise() noexcept
        {
            return static_cast<Promise*>(this);
        }

        ⟦simple_promise_base reference count methods⟧

        auto get_return_object() noexcept
        {
            return simple_task<T>(as_promise());
        }

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

        template<typename...Args>
        void set_value(Args&&... args)
        {
            m_holder.set_value(std::forward<Args>(args)...);
        }

        void unhandled_exception() noexcept
        {
            m_holder.unhandled_exception();
        }

        auto final_suspend() noexcept
        {
            ⟦return an awaiter that decrements the reference count
             and resumes any awaiting coroutine⟧
        }
```

```
        〖awaiter support methods〗

        auto get_awaiter() noexcept
        {
            〖return an awaiter that waits for the coroutine
             to complete〗
        }
    };

    template<typename T>
    struct simple_promise : simple_promise_base<T>
    {
        〖implement return_value〗
    };

    template<>
    struct simple_promise<void> : simple_promise_base<void>
    {
        〖implement return_void〗
    };

    // promise_ptr<T> is a reference-counted
    // pointer to a simple_promise<T>
    〖implement promise_ptr〗
}

namespace async_helpers
{
    template<typename T>
    struct simple_task
    {
        details::promise_ptr<T> promise;
        simple_task(details::simple_promise<T>*
            initial = nullptr) : promise(initial) {}

        void swap(simple_task& other)
        {
            std::swap(promise, other.promise);
        }

        auto operator co_await() const
        {
            return promise->get_awaiter();
        }
    };

    template<typename T>
    void swap(simple_task<T>& left, simple_task<T>& right)
    {
        left.swap(right);
    }
```

```
}

template <typename T, typename... Args>
struct std::experimental::coroutine_traits<
    async_helpers::simple_task<T>, Args...>
{
    using promise_type =
        async_helpers::details::simple_promise<T>;
};
```

I put it all out there at one go just to highlight the overall shape. But let's go through it more slowly.

```
    template<typename T>
    struct simple_promise_base
    {
        std::atomic<uint32_t> m_refcount{ 2 };
```

The initial reference count of the promise is two: One reference is held by the coroutine itself because the coroutine keeps its promise alive until it completes. The other reference is held by the `simple_task` that is the return value of the coroutine function.

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

We need a mutex to protect the `m_waiting` variable so it can be updated atomically with respect to state changes. And of course we have the object that holds the result of the coroutine (successful completion result or an exception).

```
        using Promise = simple_promise<T>;
        auto as_promise() noexcept
        {
            return static_cast<Promise*>(this);
        }
```

The `simple_promise_base` is a CRTP-like type whose derived type is always a `simple_promise<T>`. We create a type alias `Promise` to refer to that full `simple_promise` type and a helper function to produce a pointer to that type.

```
        ⟦simple_promise_base reference count methods⟧
```

Managing the reference counts is a major hassle, so I'll defer that discussion as well. Neither the result holder nor the reference count is particularly complicated, but they're rather wordy, and there are some subtle parts that deserve closer discussion.

```
        auto get_return_object() noexcept
        {
            return simple_task<T>(as_promise());
        }
```

This produces the `simple_task` that is the formal return value of the coroutine function. The caller is expected to `co_await` this `simple_task` to get the result of the coroutine function.

```
        std::experimental::suspend_never initial_suspend() noexcept
        {
            return {};
        }
```

As I noted, this is a hot-start coroutine, so there is nothing to do at the initial suspension.

```
        template<typename...Args>
        void set_value(Args&&... args)
        {
            m_holder.set_value(std::forward<Args>(args)...);
        }

        void unhandled_exception() noexcept
        {
            m_holder.unhandled_exception();
        }
```

These are the methods which store the coroutine result in the result holder. Don't be scared by the variadic template parameter list for `set_value`. The actual parameter list to `set_value` will be either empty (for `void`) or a single parameter (for non-`void`). We forward the results into the holder, or if the coroutine function throws an exception, then we capture it as an exception. We'll look at these more closely when we study the result holder.

```
        auto final_suspend() noexcept
        {
            ⟦return an awaiter that decrements the reference count
             and resumes any awaiting coroutine⟧
        }
```

One of our earlier improvements was to delay resuming any awaiting coroutines until we reach the `final_suspend`. The additional wrinkle here is that when the coroutine reaches its final suspension point, we decrement the reference count on the promise, which might or might not trigger destruction of the coroutine state. We'll discuss this some more later.

```
        ⟦awaiter support methods⟧

        auto get_awaiter() noexcept
        {
            ⟦return an awaiter that waits for the coroutine
             to complete⟧
        }
    };
```

The `get_awaiter` method produces an awaiter that waits for the coroutine to complete and returns the result (either in the form of a value or an exception). We've basically seen this before in our `result_holder`, but the wrinkles are slightly different due to our ability to process move-only types. We'll see more about this later.

```
    template<typename T>
    struct simple_promise : simple_promise_base<T>
    {
        ⟦implement return_value⟧
    };

    template<>
    struct simple_promise<void> : simple_promise_base<void>
    {
        ⟦implement return_void⟧
    };
```

As I noted earlier, it is not legal for a promise to have both `return_value` and `return_void`, so we have to split them into separate classes. We'll look at the implementation later, because there are some annoyances here.

```
    // promise_ptr<T> is a reference-counted
    // pointer to a simple_promise<T>
    ⟦implement promise_ptr⟧
}
```

The `promise_ptr` is a reference-counted pointer to our `simple_promise`. This class is basically all-annoying with nothing of interest inside it. I'll defer its implementation to later.

```
namespace async_helpers
{
    template<typename T>
    struct simple_task
    {
        simple_task(details::simple_promise<T>*
            initial = nullptr) : promise(initial) {}

        void swap(simple_task& other)
        {
            std::swap(promise, other.promise);
        }

        auto operator co_await() const
        {
            return promise->get_awaiter();
        }
    private:
        details::promise_ptr<T> promise;
    };

    template<typename T>
    void swap(simple_task<T>& left, simple_task<T>& right)
    {
        left.swap(right);
    }
}
```

The `simple_task` itself is very simple. It wraps a reference-counted pointer to the `simple_promise` and forwards its `co_await` operator to the promise's awaiter. When the `simple_task` destructs, the reference in the `promise_ptr` is released, and that takes us one step closer to the end of the coroutine state. (Of course, if you copy the `simple_task`, then the reference count goes up, and you end up extending the lifetime further.)

Half of the code is just there to support ADL swap!

```
template <typename T, typename... Args>
struct std::experimental::coroutine_traits<
    async_helpers::simple_task<T>, Args...>
{
    using promise_type =
        async_helpers::details::simple_promise<T>;
};
```

Finally, we tell the coroutine infrastructure that if somebody writes

```
async_helpers::simple_task<T> MyCoroutine()
{
    ...
    co_return ...;
}
```

then it should use the `simple_promise` to assist with the implementation of the coroutine.

Okay, so that's it! There's a lot of paperwork, but the basic idea is that the promise is where all the action is. The coroutine and the task each have a reference to the promise, and that's how the coroutine and the task communicate with each other.

Oh wait, I have a lot of code to fill in. We'll start that next time.

Raymond Chen

Follow