C++ coroutines: Associating multiple task types with the same promise type

devblogs.microsoft.com/oldnewthing/20210423-00

April 23, 2021

Raymond Chen

We created two very similiar promises for hot-start and cold-start coroutines. It turns out that we can unify them.

The association between a promise and a task is not one-to-one, but rather one-to-many: A single promise can back multiple tasks. But how can you do that? After all, the task associated with a promise is the thing returned by the `get_return_object` method.

Well, not exactly.

The rule is that the thing returned by `get_return_object` method is used to *initialize* the task. It doesn't have to be the task itself.

Therefore, you can associate multiple tasks with the promise if you arrange for `get_return_object` to return something that all of the tasks can initialize from.

The coroutine code generation goes like this:

- Call `get_return_object` to get the object that initializes the task.
- Perform these two operations in some unspecified order:
  - Begin the coroutine at `initial_suspend` and let it run until its first suspension point (determined at runtime).
  - Create a task[1] from the return value of `get_return_object`.
- Return the task.

In our case, we have two flavors of awaiters, one of which leaves the coroutine cold, and the other of which hot-starts the coroutine. The coroutine machinery itself can be left unaware of this detail and leave the mechanics to the task.

For Windows developers, two kinds of tasks that would be useful are one that awaits in a thread-unaware way (the version we have been writing so far), and another that awaits in a way that preserves the COM context.

But for today, I'll show how a single promise can be used for both cold-start and hot-start tasks. Go back to our hot-start coroutine promise and make these changes:

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

    std::atomic<void*> m_waiting{ cold_ptr };

    static constexpr void* cold_ptr = reinterpret_cast<void*>(3);
```

The coroutine now starts out cold. The warm-start task will auto-start it, whereas the cold-start task will leave it cold until it is awaited.

```
    auto get_return_object() noexcept
    {
        return as_promise();
    }
```

We alter the `get_return_object` method so that it returns a pointer to the promise, rather than the task constructed from it. This allows us to have multiple tasks that construct in different ways, and more importantly, have different awaiters.

```
    void start()
    {
        m_waiting.store(running_ptr, std::memory_order_relaxed);
        as_handle().resume();
    }
```

A new explicit `start()` method kicks off the coroutine. The hot-start task will call this immediately, whereas the cold-start task will wait until the task is `co_await`ed.

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

The coroutine now suspends at its initial suspend point instead of continuing to run. This makes the coroutine a cold-start coroutine by default.

```
    bool cold_client_await_ready()
    {
        return false;
    }

    auto cold_client_await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
        start();
        return m_waiting.exchange(handle.address(),
            std::memory_order_acq_rel) == running_ptr;
    }
```

These new functions are carried over from our previous conversion from hot-start to cold-start, but with different names so we can keep both versions.

Of course, we need to create an awaiter that uses these cold versions.

```
template<typename T>
struct cold_promise_awaiter
{
    promise_ptr<T> self;

    bool await_ready()
    {
        return self->cold_client_await_ready();
    }

    auto await_suspend(std::experimental::coroutine_handle<> handle)
    {
        return self->cold_client_await_suspend(handle);
    }

    T await_resume()
    {
        return self->client_await_resume();
    }
};
```

This is analogous to our `promise_awaiter`, except that it uses the cold versions of `await_ready` and `await_suspend`.

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

        struct cannot_await_lvalue_use_std_move {};
        cannot_await_lvalue_use_std_move operator co_await() & = delete;

    protected:
        promise_ptr<T> promise;
    };
}

namespace async_helpers
{
    template<typename T>
    struct simple_task : details::simple_task_base<T>
    {
        using base = details::simple_task_base<T>;
        simple_task() = default;
        simple_task(details::simple_promise<T>*
            initial) : base(initial)
            { this->promise->start(); }

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

        using base::operator co_await;

        auto operator co_await() &&
        {
            return details::promise_awaiter<T>
                { std::move(this->promise) };
        }
    };

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

We factor out the promise-management code and the “you’re holding it wrong” class into a common base class `simple_task_base`.

The `simple_task` used to have a single constructor that covered both construction from a promise and construction of an empty task. We split them up, so that we can `start()` the promise in the case where we are being constructed as a result of a call to `get_return_object`. This is what turns the cold-start coroutine into a hot-start coroutine.

We can also create a `cold_simple_task` that is the cold-start version.

```
namespace async_helpers
{
    template<typename T>
    struct cold_simple_task : details::simple_task_base<T>
    {
        using base = details::simple_task_base<T>;
        cold_simple_task(details::simple_promise<T>*
            initial = nullptr) : base(initial) { }

        void swap(cold_simple_task& other)
        {
            std::swap(this->promise, other.promise);
        }

        using base::operator co_await;

        auto operator co_await() &&
        {
            return details::cold_promise_awaiter<T>
                { std::move(this->promise) };
        }
    };

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

This is the same as our `simple_task` except that

- It doesn't `start()` the coroutine, leaving it cold.
- It uses `cold_promise_awaiter` instead of `promise_awaiter`.

Finally, we teach the compiler how to create a coroutine that returns a `cold_simple_task`:

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

There we have it, a single promise that supports multiple kinds of tasks. This is particularly handy when you have different kinds of tasks that differ only in how they await, since the awaiter isn't even part of the promise at all.

Next time, we'll look at how coroutines interact with the `noexcept` keyword.

[1] The language specification says merely that `get_return_object()` "is used to initialize" the return object, but doesn't say what kind of initialization is used. Is it *copy-initialization*, or is it *direct-initialization*? (It's almost certainly not *list-initialized*.) Copy initialization considers only conversions, but direct initialization also considers the constructors of the destination. Different compilers have interpreted the standard differently.

Raymond Chen

Follow