Creating a task completion source for a C++ coroutine: Failing to produce a result

devblogs.microsoft.com/oldnewthing/20210326-00

March 26, 2021

Raymond Chen

So far, we've been working on building a `result_holder` that can hold any type of result. But what about errors?

Because maybe you have code that's waiting for a result, and the code that's supposed to produce the result realizes that it messed up and wants to say, "Sorry, no result today."

We can do that by storing a `std::exception_ptr` as a possible result. This means that our result is no longer merely an optional value, but rather it's a variant, since it could be empty, or it could have a value, or it could have an exception. It also means that `ready` needs to become a discriminant.

```
template<typename T>
struct result_holder_state :
    async_helpers::awaitable_state<result_holder_state<T>>
{
    enum class result_kind {
        unset,
        value,
        exception,
    };

    std::atomic<result_kind> kind{ result_kind::unset };

    struct wrapper
    {
        T value;
    };

    union variant
    {
        variant() {}
        ~variant() {}

        wrapper wrap;
        std::exception_ptr ex;
    } result;

    result_holder_state() {}
    result_holder_state(result_holder_state const&) = delete;
    void operator=(result_holder_state const&) = delete;

    ~result_holder_state()
    {
        switch (kind.load(std::memory_order_relaxed)) {
        case result_kind::value:
            result.wrap.~wrapper();
            break;
        case result_kind::exception:
            result.ex.~exception_ptr();
            break;
        }
    }

    using typename result_holder_state::extra_await_data;
    using typename result_holder_state::node_list;

    bool fast_claim(extra_await_data const&) noexcept
    {
        return kind.load(std::memory_order_acquire)
            != result_kind::unset;
    }

    bool claim(extra_await_data const&) noexcept
    {
        return kind.load(std::memory_order_relaxed)
            != result_kind::unset;
    }

    void set_result(node_list& list, T v)
    {
        if (kind.load(std::memory_order_relaxed)
            == result_kind::unset) {
            new (std::addressof(result.wrap))
                wrapper{ std::forward<T>(v) };
            kind.store(result_kind::value,
                std::memory_order_release);
            this->resume_all(list);
        }
    }

    void set_exception(
        node_list& list, std::exception_ptr ex)
    {
        if (kind.load(std::memory_order_relaxed)
            == result_kind::unset) {
            new (std::addressof(result.ex))
                std::exception_ptr{ std::move(ex) };
            kind.store(result_kind::exception,
                std::memory_order_release);
            this->resume_all(list);
        }
    }

    T get_result()
    {
        switch (kind.load(std::memory_order_relaxed)) {
        case result_kind::value:
            return result.wrap.value;
        case result_kind::exception:
            std::rethrow_exception(result.ex);
        }
        std::terminate(); // shouldn't get here
    }
};
```

There isn't much exciting going on here. It's just changing the things that need to be changed: Instead of a simple `bool` tracking what is in the `result`, we use a discriminant which starts out `unset`. Cleaning up our variant requires us to call the appropriate destructor for the contents of the `result`, and setting the result requires us to update the discriminant.

Setting an exception is the same as setting a value, except that we put the result in to the `ex` member instead of the `wrap` wrapper.

When it comes time to fetch the result, we check what we have. If we have a value, we return it. If we have an exception, we rethrow it. (Otherwise, something went wrong and we terminate.)

The `result_holder` itself is basically the same, just with an extra method for storing the exception.

```
template<typename T>
struct result_holder
    : async_helpers::awaitable_sync_object<
        result_holder_state<T>>
{
    using typename result_holder::state;

    void set_result(T result) const noexcept
    {
        this->action_impl(&state::set_result,
            std::move(result));
    }

    void set_exception(std::exception_ptr ex) const noexcept
    {
        this->action_impl(&state::set_exception,
            std::move(ex));
    }
};
```

The Windows Runtime `IAsyncOperation` can be awaited only once, but you can use this `result_holder` to make it possible to await multiple times.

```
template<typename T>
result_holder<T> MakeMultiAwaitable(IAsyncOperation<T> async)
{
    result_holder<T> holder;
    async.Completed([holder, async](auto, auto status) {
        try {
            switch (status) {
            case AsyncStatus::Completed:
                holder.set_result(async.GetResults());
                break;
            case AsyncStatus::Canceled:
                throw hresult_canceled();
            case AsyncStatus::Error:
                throw_hresult(async.ErrorCode());
            }
        } catch (...) {
            holder.set_exception(std::current_exception());
        }
    });
    return holder;
}
```

You would use it something like this:

```
class MyClass
{
    result_holder<Widget> widget_result;

    MyClass()
    {
        // Kick off initialization but don't wait for it.
        widget_result = MakeMultiAwaitable(InitializeAsync());
    }

    IAsyncOperation<Widget> InitializeAsync();

    IAsyncAction DoSomethingAsync()
    {
        // Wait for the widget that InitializeAsync produced.
        // rethrow any exception that occurred during initialization.
        auto widget = co_await widget_result;
        ... do something interesting ...
    }
};
```

More generally, you would do something like this:

```
void CalculateResult(result_holder<Widget>& holder)
{
    try
    {
        /* do a bunch of calculations */
        widget = /* the answer */;
        holder.set_result(widget);
    } catch (...) {
        holder.set_exception(std::current_exception());
    }
}
```

You know what this looks like?

A coroutine!

```
result_holder<Widget> CalculateResult()
{
    /* do a bunch of calculations */
    widget = /* the answer */;
    co_return widget;

    /* exception is captured into the result_holder */
}
```

So I guess it’s time to learn how to create our own coroutines. The dive into the deep end begins next time.

Raymond Chen

Follow