C++ coroutines: Making it impossible to co_await a task twice

devblogs.microsoft.com/oldnewthing/20210414-00

April 14, 2021

Raymond Chen

One design limitation of the coroutine implementation we've been developing is that it supports only one `co_await` client. We enforce this with a runtime assertion, but what if the problem occurs in the release build?

If the two `co_await` clients are concurrent, then the second one overwrites the `m_waiting` member that was set by the first. When the coroutine completes, only the second one is woken, and the first one remains suspended forever.

If the two `co_await` clients are sequential, so that the second call occurs after the coroutine has completed, then the second call bypasses the suspend and goes straight to `await_return`, where it receives the contents of a moved-from variable. Depending on what kind of object that variable represents, it could mean that the second caller gets a copy (if the object doesn't have a move copy constructor), or it could mean that the second caller gets an empty object (if the moved-from object is left empty), or it could mean that the second caller gets some sort of garbage (unlikely for move copy constructor, but technically legal).

Both of these errors are hard to diagnose. In the first case, one of the tasks just stops and makes no further progress. In the second case, one of the tasks gets unreliable results.

So let's make it easier to diagnose.

We made the mistake of making the task copyable. It wraps a reference-counted pointer, and copying increases the reference count. But really, when some code has a task and it passes the task to another component, it needs to coordinate ownership of the task with the other component so that only one `co_await` is ultimately peformed.

In C++, ownership is typically represented by a move-only object like a `unique_ptr`. If you want to transfer ownership, you `std::move` the object to the recipient.

So let's make our task a move-object object. This makes it impossible to have two references to the same promise, which would otherwise tempt you to `co_await` twice. What's more, let's make the `co_await` operation a destructive operation by having it consumes the move-

only object, leaving it empty. Once you await the task, the task object becomes null, and a subsequent `co_await` on the same task object will crash immediately.

First, let's fix our definition of `promise_ptr`. As a nice side-effect, it involves deleting a lot of code because our custom `promise_ptr` disappears.

```
struct promise_deleter
{
    void operator()(simple_promise_base<T>* promise) const noexcept
    {
        promise->decrement_ref();
    }
};

template<typename T>
using promise_ptr = std::unique_ptr<simple_promise_base<T>, promise_deleter<T>>;
```

Our `promise_ptr` is now just a `unique_ptr` with a custom deleter which calls `decrement_ref`.

Making `promise_ptr` a move-only object causes the `simple_task` to become a move-only object since contains a `promise_ptr` as a member.

And then we make the `co_await` operator require an rvalue reference, so that it consumes the promise rather than merely referencing it.

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

    auto operator co_await &&
    {
        ...
    }
    ...
};
```

Suffixing the function declaration with `&&` means that it applies only to rvalue references.

```
simple_task<Result> task = SomeFunctionReturningSimpleTask();
DoSomethingElseInTheMeantime();
co_await task; // does not compile
```

The error message is the somewhat baffling

```
error C3312: no callable 'await_resume' function found for type 'simple_task<Result>'
```

And that's if you're lucky. If you are performing the `co_await` from a coroutine provided by some other library, then the error message will depend on the library (for reasons we will learn later). For example, if you are doing this from a C++/WinRT `IAsyncAction` or `IAsyncOperation`, you get

```
error C2672: 'get_awaiter': no matching overloaded function found
```

The compiler is trying to figure out how to `co_await` a `simple_task` lvalue, and it can't find anything.

We once again enter the weird world of compiler error message metaprogramming.

I came up with this:

```
template<typename T>
struct simple_task
{
    ...
    struct cannot_await_lvalue_use_std_move {};
    cannot_await_lvalue_use_std_move operator co_await() & = delete;
    ...
};
```

If we provide no way to await an lvalue, then the compiler will report an error based on where in the evaluation process it finally got stuck. So let's make it get stuck at a predictable place, with a name we get to control.

```
error C3312: no callable 'await_resume' function found for type
'simple_task<Result>::cannot_await_lvalue_use_std_move'
```

The C++/WinRT custom awaiter error message remains, however. We can hack around this by fooling C++/WinRT into thinking that we are awaitable, and then get the compiler to generate the error message that contains our custom error message disguised as a class name.

```
    struct cannot_await_lvalue_use_std_move { void await_ready() {} };
```

The error message is now

```
error C2039: 'await_resume': is not a member of 'simple_task<Result>::
cannot_await_lvalue_use_std_move'
```

That's a little better.

Okay, so I left a bunch of `...` inside the body of `operator co_await &&`. We need to move the `promise_ptr` into the awaiter, and that means having to do some restructuring of the promise's awaiter code. Nothing essential has changed; we just need to appease the compiler.

To avoid a circular reference between the `simple_promise` and the `promise_ptr`, I'll pull the awaiter out into a separate class and have it forward its methods back into the promise. The order of declaration is

- `simple_promise_base`
- `promise_ptr`
- `promise_awaiter`

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

    // auto get_awaiter()
    // { ... }

    ...
};

struct promise_deleter
{
    void operator()(simple_promise_base<T>* promise) const noexcept
    {
        promise->decrement_ref();
    }
};

template<typename T>
struct promise_awaiter
{
    promise_ptr<T> self;

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

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

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

We take the `get_awaiter` anonymous awaiter and give it a name: `promise_awaiter`. This `promise_awaiter` is a separate class (avoiding the circular reference), and it retains its `self` in the form of a `promise_ptr`. This causes the promise to be released when the awaiter destructs at the end of the `co_await`.

Now we can fill in those missing dots.

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

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

We can now write

```
simple_task<Result> task = SomeFunctionReturningSimpleTask();
DoSomethingElseInTheMeantime();
co_await std::move(task); // explicit move
```

The explicit `std::move` makes it clear that you are giving the task to `co_await`, and that the task is no longer usable after that point. Furthermore, if you try to `co_await` it, you will take a null pointer exception since the task is now empty. We used to have a mysterious bug where `co_await` sometimes seemed to hang, or sometimes produced incorrect results. Now we have an immediate crash, which is much easier to diagnose.

Next time, we'll get rid of the mutex that protects the `coroutine_handle<>` which records the continuation.

Raymond Chen

Follow