How to wait for multiple C++ coroutines to complete before propagating failure, wrapping the awaitable

devblogs.microsoft.com/oldnewthing/20230706-00

July 6, 2023

Raymond Chen

So far, we've been trying to intercept exceptions that propagate from the awaitable by wrapping it in a coroutine that catches exceptions. This seems like a reasonable approach, but we got bogged down dealing with the edge case where the wrapper coroutine itself throws a memory allocation exception while creating the coroutine frame.

But there's another way to intercept the exception that doesn't involve creating a new coroutine at all. Instead of wrapping the awaitable in a coroutine, we can wrap the awaitable inside another awaitable.

Consider the simple case of an awaitable that is its own awaiter. (No `operator co_await`.) You can wrap the awaitable inside another awaitable that catches the exception. We'll start with a wrapper that does nothing.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) : m_inner(inner) {}

    Inner& m_inner;

    bool await_ready()
    { return m_inner.await_ready(); }

    template<typename Handle>
    auto await_suspend(Handle handle) {
        return m_inner.await_suspend(handle);
    }

    auto await_resume()
    { return m_inner.await_resume(); }
};
```

This wrapper doesn't do anything. It just forwards all of the awaiter methods to the wrapped object.

But now that we have a framework, we can start adding features to it.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) : m_inner(inner) {}

    Inner& m_inner;

    bool await_ready()
    { return m_inner.await_ready(); }

    template<typename Handle>
    auto await_suspend(Handle handle) {
        return m_inner.await_suspend(handle);
    }

    std::exception_ptr await_resume() try {
        m_inner.await_resume();
        return nullptr;
    } catch (...) {
        return std::current_exception();
    }
};
```

We change `await_resume()` to return any exception that is thrown when inspecting the result of the `co_await`. If no exception occurs, then we return `nullptr`, indicating that no exception occurred.

That's the basic idea. The rest is filling in the gaps.

For example, what happens if the awaiter throws an exception from one of the other two awaiter methods?

Well, the `await_ready` happens before the coroutine suspends, so any exception that is thrown from there just goes straight to the containing coroutine's `unhandled_exception`. Since our goal is to capture and return the exception rather than allowing it to be thrown, we have to include a `try`/`catch` in our `await_ready` wrapper, and if an exception occurs, we need to bypass the `await_suspend` and go directly to `await_resume`. Furthermore, the `await_resume` should not call the wrapped awaiter; it should just produce the caught exception immediately.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) : m_inner(inner) {}

    Inner& m_inner;
    std::exception_ptr m_eptr;

    bool await_ready() try
    { return m_inner.await_ready(); }
    catch (...) {
        m_eptr = std::current_exception();
        return true;
    }

    template<typename Handle>
    auto await_suspend(Handle handle) {
        return m_inner.await_suspend(handle);
    }

    std::exception_ptr await_resume() try {
        if (m_eptr) return m_eptr;
        m_inner.await_resume();
        return nullptr;
    } catch (...) {
        return std::current_exception();
    }
};
```

An exception in `await_suspend` is the trickiest one. According to the language specification, if an exception is thrown by `await_suspend`, the coroutine resumes but does *not* call `await_resume`. Instead, the exception thrown by `await_suspend` is immediately rethrown.

This means that if we catch an exception from `await_suspend`, we should return it immediately from `await_resume` without calling `m_inner.await_resume()`.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) : m_inner(inner) {}

    Inner& m_inner;
    std::exception_ptr m_eptr;

    bool await_ready() try
    { return m_inner.await_ready(); }
    catch (...) {
        m_eptr = std::current_exception();
        return true;
    }

    template<typename Handle>
    auto await_suspend(Handle handle) try {
        return m_inner.await_suspend(handle);
    } catch (...) {
        m_eptr = std::current_exception();
        return /* something */;
    }

    std::exception_ptr await_resume() try {
        if (m_eptr) return m_eptr;
        m_inner.await_resume();
        return nullptr;
    } catch (...) {
        return std::current_exception();
    }
};
```

The next tricky part is the `return /* something */`. We want to return something that means “Don’t suspend this coroutine after all.” But it also has to be compatible with the `return` statement inside the `try` block.

Fortunately, symmetric transfer is a superset of the other types of return values from `await_suspend()`, so we can always return a coroutine handle and adapt the other return values accordingly.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) : m_inner(inner) {}

    Inner& m_inner;
    std::exception_ptr m_eptr;

    bool await_ready() try
    { return m_inner.await_ready(); }
    catch (...) {
        m_eptr = std::current_exception();
        return true;
    }

    template<typename Handle>
    std::coroutine_handle<>
    await_suspend(Handle handle) try {
        using Ret = decltype(m_inner.await_suspend(handle));
        if constexpr (std::is_same_v<void, Ret>) {
            m_inner.await_suspend(handle);
            return std::noop_coroutine();
        } else if constexpr (std::is_same_v<bool, Ret>) {
            return m_inner.await_suspend(handle) ?
                static_cast<std::coroutine_handle<>>(
                    std::noop_coroutine()) :
                handle;
        } else {
            return m_inner.await_suspend(handle);
        }
    } catch (...) {
        m_eptr = std::current_exception();
        return handle;
    }

    std::exception_ptr await_resume() try {
        if (m_eptr) return m_eptr;
        m_inner.await_resume();
        return nullptr;
    } catch (...) {
        return std::current_exception();
    }
};
```

All right, we've wrapped an awaiter so that awaiting the wrapper reports any exception that may have resulted from awaiting the original awaiter.

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    std::exception_ptr eptr;

    auto accumulate = [&](std::exception_ptr e) {
        if (eptr == nullptr) eptr = e;
    };

    (accumulate(co_await wrapped_awaitable(asyncs)), ...);

    if (eptr) std::rethrow_exception(eptr);
}
```

Next time, we'll deal with `operator co_await`.