How to wait for multiple C++ coroutines to complete before propagating failure, finding the awaiter

devblogs.microsoft.com/oldnewthing/20230707-00

July 7, 2023

Raymond Chen

Last time, we wrapped an object that was its own awaiter. In general, though, we may have to find the awaiter.

Fortunately, we already wrote a helper class that can find an awaiter. Let's bring it back, but with a little change to the `get_awaiter` function.

```
class awaiter_finder
{
    template<typename T>
    static void find_co_await_member(T&&, ...);
    template<typename T>
    static auto find_co_await_member(T&& value, int)
    -> std::decay_t<decltype(static_cast<T&&>(value).operator co_await())> {
        return static_cast<T&&>(value).operator co_await();
    }
    template<typename T>
    using member_awaiter = decltype(find_co_await_member(std::declval<T>(), 0));

    template<typename T>
    static void find_co_await_free(T&&, ...);
    template<typename T>
    static auto find_co_await_free(T&& value, int)
    -> std::decay_t<decltype(operator co_await(static_cast<T&&>(value)))> {
    return operator co_await(static_cast<T&&>(value));
    }
    template<typename T>
    using free_awaiter = decltype(find_co_await_free(std::declval<T>(), 0));

public:
    template<typename T>
    static decltype(auto) get_awaiter(T&& value)
    {
        if constexpr (!std::is_same_v<member_awaiter<T>, void>) {
            return find_co_await_member(static_cast<T&&>(value), 0);
        } else if constexpr (!std::is_same_v<free_awaiter<T>, void>) {
            return find_co_await_free(static_cast<T&&>(value), 0);
        } else {
            return std::forward<T>(value);
        }
    }

    template<typename T>
    using type = decltype(get_awaiter(std::declval<T>()));
};
```

In the case where the object is its own awaiter, we want `get_awaiter()` to return a reference to the object itself. This means that we need to change the return type of `get_awaiter()` to `decltype(auto)` so that it will infer a reference when we return `value`.

But that means that it will also infer a reference from `find_co_await_member` and `find_co_await_free`, so we have to `std::decay_t` those return types to remove any reference or cv-qualifiers.

We can use this helper to teach our `wrapped_awaitable` how to find the awaiter.

```
template<typename Inner>
struct wrapped_awaitable
{
    wrapped_awaitable(Inner& inner) :
    m_awaiter(awaiter_finder::get_awaiter(std::move(inner))) {}

    typename awaiter_finder::type<Inner> m_awaiter;
    std::exception_ptr m_eptr;

    bool await_ready() try
    { return m_awaiter.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_awaiter.await_suspend(handle));
        if constexpr (std::is_same_v<void, Ret>) {
            m_awaiter.await_suspend(handle);
            return std::noop_coroutine();
        } else if constexpr (std::is_same_v<bool, Ret>) {
            return m_awaiter.await_suspend(handle) ?
                static_cast<std::coroutine_handle<>>(
                    std::noop_coroutine()) :
                handle;
        } else {
            return m_awaiter.await_suspend(handle);
        }
    } catch (...) {
        m_eptr = std::current_exception();
        return handle;
    }

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

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);
}
```

Now that we have a `wrapped_awaitable` that completes with a `std::exception_ptr`, we have reduced the problem to the case way at the start of this series where we have a bunch of items, all of the same type, so we can put them in an `initializer_list` and use standard algorithms with them.

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    auto results = { co_await wrapped_awaitable(asyncs))... };
    auto it = std::find_if(results.begin(), results.end(),
        [](auto&& eptr) { return eptr; });
    if (it != results.end()) std::rethrow_exception(*it);
}
```

However, this misses an edge case: If you call `when_all_complete` with no parameters, the compiler cannot deduce `results` from what ends up being an empty set of braces. So we'll have to provide an explicit type for it.

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    std::initializer_list<std::exception_ptr>
        results = { co_await wrapped_awaitable(asyncs))... };
    auto it = std::find_if(results.begin(), results.end(),
        [](auto&& eptr) { return eptr; });
    if (it != results.end()) std::rethrow_exception(*it);
}
```

Okay, we finally did it: We wrote a `when_all_complete` which runs all of the awaitables to completion.[1] If any of the awaitables throws an exception, then the first such exception is reported.

We'll wrap up with some final thoughts next time.

[1] Now, if any of the awaitables throws in its awaiter because it is unable to await at all, then we won't wait for completion after all because the awaiter failed at its job. But we have no way of knowing whether any exception that is thrown from the awaiter means "I awaited successfully, and it completed in a failure state", or whether it means "I couldn't even await!" We just have to do the best we can with the information we have. At least we fulfilled our end of the deal: We did wait for the awaitable to run to completion. If it decided to report failure before finishing its work, that's the awaitable's problem.