Why is coroutine_handle::resume() potentially-throwing?

devblogs.microsoft.com/oldnewthing/20210505-00

May 5, 2021

Raymond Chen

In our explorations of making `co_await`able objects, we had largely been ignoring the possibility of the coroutine handle throwing an exception upon resume. But according to the language specification, the `resume` method (and its equivalent, the `operator()` overloaded function call operator) is potentially-throwing. Is this an oversight or an intentional decision?

Well, the `noop_coroutine`'s coroutine handle does mark its `resume()` method as `noexcept`, so it's not like the authors of the coroutine specification simply forgot about `noexcept`. They consciously put it on the resumption of a `noop_coroutine`, but omitted it from other coroutines.

What's more, if you look at libraries that operate on coroutines, all of them treat the `resume` method as if it were `noexcept`.

What's the deal?

Gor Nishanov explained it to me.

Allowing `resume` to throw was introduced in P0664R6 section 25, with this remark:

> This resolution allows generator implementations to define unhandled_exception as follows:
>
> ```
> void unhandled_exception() { throw; }
> ```
>
> With this implementation, if a user of the generator pulls the next value, and during computation of the next value an exception will occur in the user authored body it will be propagate back to the user and the coroutine will be put into a final suspend state and ready to be destroyed when generator destructors is run.

Yeah but what does that all mean?

The scenario here is the use of coroutines as generators.

If a generator encounters an exception, the normal mechanism would be for the exception to be captured in the coroutine's `unhandled_exception` method so that it can be re-thrown when the caller performs an `await_resume`. But if the generator is synchronous (performs no `co_await` operations), then it is more efficient to just let the exception propagate across the coroutine boundary directly to the caller.

The coroutine implementation (specifically, the promise) can indicate that it wants the exception to propagate by rethrowing the exception in `unhandled_exception`, rather than capturing it.

But if you're not in the case of a synchronous generator (and when dealing with coroutines as tasks, you won't be), then `resume` is indeed nonthrowing.

Bonus reading: Another reason for not marking `resume()` as `noexcept` is that `resume()` requires that the coroutine be suspended. The presence of a precondition means that, according to the Lakos Rule, the function should not be marked `noexcept`. This allows the implementation to choose to report the precondition violation in the form of an exception.

Raymond Chen

Follow