C++ coroutines: What does it mean when I declare my coroutine as noexcept?

devblogs.microsoft.com/oldnewthing/20210426-00

April 26, 2021

Raymond Chen

Suppose you want a coroutine that terminates on unhandled exceptions, or equivalently (looking at it from the consumer side) a coroutine that never throws an exception when awaited. For regular functions, the way to say this is to put the `noexcept` exception specification on your function declaration:

```
int GetValue() noexcept
{
    return LoadValue();
}
```

If the `LoadValue()` function raises an exception, the exception propagation stops at the `noexcept` and turns into a `std::terminate`, which is a fatal error that terminates the application.

Looking at the contract from the other side, the `noexcept` specification tells the caller that no exceptions can escape the `GetValue()` function, so the caller can optimize accordingly. `GetValue()` will get you a value or die trying.

Okay, so what happens when you apply this to a coroutine?

```
simple_task<int> GetValueAsync() noexcept
{
    co_return LoadValue();
}
```

If an exception is thrown by the `LoadValue()` function, the exception is captured into the `simple_task` and is rethrown when the task is `co_await`ed.

Wait a second. I put the `noexcept` keyword on this function. Certainly that means that any unhandled exception in the function terminates the program, right?

Yes, that's what it means, but your coroutine isn't the function.

The function is the thing that returns a `simple_task`. And the `noexcept` says that the `GetValueAsync()` function can successfully return a `simple_task` without raising an exception.

Look at this from the caller's point of view: The caller sees only

```
simple_task<int> GetValueAsync() noexcept;
```

This is not a coroutine definition. This is just a function prototype. The caller doesn't know how GetValueAsync() is going to produce that `simple_task`. The implementation could be

```
simple_task<int> GetValueAsync() noexcept
{
    return simple_task<int>(constructor parameters);
}
```

Just the usual case of returning a constructed object. No coroutines involved at all.

If `GetValueAsync()` is implemented as a coroutine, then any unhandled exception is passed to the coroutine promise's `unhandled_exception` method, and it's up to the promise to decide what to do next.

So what can you do if you really want your coroutine to terminate on unhandled exception? We'll look at that next time.

Raymond Chen

Follow