C++ coroutines: The mental model for coroutine promises

devblogs.microsoft.com/oldnewthing/20210329-00

March 29, 2021

Raymond Chen

My earlier series on getting started with awaitable objects looked at how you produce awaitable objects, but up until now we haven't been looking at the other side: How can you be an object that can await other objects?

Lewis Baker covered this topic some time ago, but I'm going to take a stab at it as well. Because maybe if there are enough articles about it, one of them will actually make sense to you.

The idea behind language coroutines is that the coroutine function is transformed into a state machine, formally known as a *coroutine state*. The coroutine is represented to callers by some sort of object, and depending on how fancy you get, that object may provide functionality such as basic things like `co_await`ing the return value to retrieve the final result of the coroutine function, or fancier things like interacting with the coroutine-in-progress, say, by cancelling it or retrieving progress information.

The coroutine state is represented in memory by an aggregate of three things:

Compiler bookkeeping
Promise object
Coroutine "stack frame"

The compiler bookkeeping is where the compiler puts its state machine, so that it knows where to resume a suspended coroutine. It also needs to keep track of how far the code in the coroutine has reached, so that it knows which variables need to be destructed if the coroutine state is destroyed before the coroutine reaches its natural end.

The coroutine stack frame contains all the stuff that used to go into the traditional stack frame. Inbound parameters, local variables, that sort of thing.

Of course, this is all compiler implementation detail, but it's still handy to think of a coroutine state of consisting of these three pieces, even if it's not literally how things are organized. For example, the compiler might keep its live-object information in the space it reserved for the stack frame, particularly if the stack frame is where it kept the live-object information in non-coroutine functions. And if a local variable's lifetime does not extend across a suspension point, then it could be stored on the regular stack instead of inside the coroutine state's stack frame.

Anyway, the important thing is that there is this bonus "promise" object hiding inside the coroutine state. This promise object is something you provide as the implementor of the coroutine, and its main job is to mediate communication between the coroutine and the outside world.

For concreteness, let's say that we have something like this:

```
task DoSomethingAsync()
{
    co_await something();
    co_return 42;
}
```

To start things off, the compiler allocates a coroutine state, constructs the promise object inside the coroutine state, let's call it `p`, and then calls `p.get_return_object()` to ask the promise to produce the object that is returned to the caller, which matches the nominal return value of the coroutine. In the above example, `p.get_return_object()` should return a `task`.

The promise typically arranges so that it and the return object work together to establish the communication channel between the coroutine and its caller.

When the coroutine performs a `co_return`, the compiler calls `p.return_value()`, passing the value that the coroutine returned. In the above example, the compiler will call `p.return_value(42)`. The promise will typically put this value somewhere that the return object can access, so that the caller of the coroutine can obtain the result.

Here's a diagram of how the pieces fit together. If a caller does `t = DoSomethingAsync()`, what we have is this:

Coroutine state	`p.get_return_object()`	**Caller**
bookkeeping		
promise	→→→→→→	`t`
stack frame		

The promise produced a `task`, which the caller now possesses. Most of the time, the `task` object will be something the caller can `co_await`, so that the caller can retrieve the result produced by the coroutine.

Eventually, the coroutine finishes with a `co_return`, and the coroutine state forwards the value through to the original return object `t`, and that in turn causes the caller's `co_await t` to complete with that result.

Coroutine function	`p.return_value(42);`	Coroutine state		Caller
		bookkeeping		
`co_return 42;`	→→→	promise	→→→	`t`
		stack frame		

The compiler calls `p.return_value(42)` to pass the result from the coroutine to the promise, and it's up to the promise to figure out how to propagate that value to the task.

Next time, we'll take these diagrams and fill them in with actual code.

Raymond Chen

Follow