How do I get the effect of C#'s async void in a C++ coroutine? Part 2: Keeping track of the lifetimes

devblogs.microsoft.com/oldnewthing/20190117-00

January 17, 2019

Raymond Chen

Last time, we looked at how to write a function that formally returns `void` that nevertheless performs `co_await` operations. The function acts like a fire-and-forget, where the remainder of the task runs asynchronously after the first `co_await` that needs to suspend.

To recap: We saw that the obvious single-function solution fails because of lifetime issues:

```
// Code in italics is wrong.
void MyEventHandler(int a, int b)
{
  [=]() -> Concurrency::task<void>
  {
    GetReady(a);
    co_await GetSetAsync(b);
    Go(a, b);
  }();
}
```

The problem is that the lambda is destroyed when the function returns, and then when the task tries to access it, it ends up trying to use a detroyed object, and that doesn't end well.

Functions that are converted to coroutines capture the function parameters and their local variables in a frame. The frame consists of the function's formal parameters and local variables, so we need to transfer the captured things into the frame:

```
void MyClass::MyEventHandler(int a, int b)
{
  [](std::shared_ptr<MyClass> self, int a, int b)
   -> Concurrency::task<void>
  {
   self->GetReady(a);
   co_await self->GetSetAsync(b);
   self->Go(a, b);
  }(this->shared_from_this(), a, b);
}
```

However, this forces us to repeat `self` all the time. What we want is a way to get `this` to refer to the class whose method we are in, so we won't have to do this `self` nonsense. But we also want a captureless lambda, so we won't have the problem of a task trying to access a destroyed lambda.

This is a contradiction. If you have a captureless lambda, then it cannot capture `this`. The problem looks hopeless.

Until you realize that there's nobody forcing you to do it all in just one lambda.

So let's use two lambdas. One of them captures `this` so it doesn't have to carry `self` around. And then we put that lambda into the frame of the second lambda and invoke it. Then we invoke the second lambda to get the party started.

```
void MyClass::MyEventHandler(int a, int b)
{
  auto lambda1 =
   [=, lifetime = this->shared_from_this()]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  };

  auto lambda2 = [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  };

  lambda2(lambda1);
}
```

The first lambda is the one we wish we could use. It captures anything it wants, and can `co_await` for whatever it likes. To ensure that `this` doesn't disappear out from under it, we use the trick of capturing an explicit lifetime object into the lambda to keep the parent object alive for the duration of the lambda.

The second lambda is the one that captures nothing, and is therefore suitable for running as a fire-and-forget coroutine. It accepts the first lambda as its formal parameter, and since formal parameters are part of the frame, this keeps that copy alive for the duration of the task. We then invoke that first lambda with a `co_await` so that the second lambda's task will not complete until the first lambda's task is finished. This keeps the first lambda alive for the duration of the task.

Finally, we invoke the second lambda, with the first lambda as its parameter, to set things into motion.

The rest of the work is fine-tuning.

We pass the first lambda by value, but that is a bit wasteful because the first lambda isn't used any more once it is passed to the second lambda. Let's move it rather than copying. Moving also allows the first lambda to capture move-only objects, like RAII types such as `std::unique_ptr`.

```
void MyClass::MyEventHandler(int a, int b)
{
  auto lambda1 =
   [=, lifetime = this->shared_from_this()]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  };

  auto lambda2 = [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  };

  lambda2(std::move(lambda1));
}
```

But we can get the same effect as an explicit move by simply passing the second lambda inline, which creates an rvalue reference.

```
void MyClass::MyEventHandler(int a, int b)
{
  auto lambda2 = [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  };

  lambda2([=, lifetime = this->shared_from_this()]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

And then you can go all the way and make `lambda2` an immediately-invoked lambda.

```
void MyClass::MyEventHandler(int a, int b)
{
  [](auto lambda1)
   -> Concurrency::task<void>
  {
   co_await lambda1();
  }([=, lifetime = this->shared_from_this()]()
   -> Concurrency::task<void>
  {
   GetReady(a);
   co_await GetSetAsync(b);
   Go(a, b);
  });
}
```

Next time, we'll see what we can do to reduce the boilerplate needed to carry out this pattern.

Raymond Chen

Follow