Crash course in async and await

devblogs.microsoft.com/oldnewthing/20170720-00

July 20, 2017

Raymond Chen

I'm going to assume that you know how the `async` and `await` keywords work. If you need a refresher, you can read Eric Lippert's extensive exposition of the subject. Here's the short version. **People who know how `async` and `await` work can go take a nap.**

When you write a function that is marked `async`, then the function is broken up into a series of mini-functions at each `await` call. The code executes synchronously up until the first `await`, at which point the rest of the code is scheduled for resumption when the awaited thing produces a result. Optionally, a task is returned so that the caller can schedule its own continuation when the async function executes its `return` statement.

For example, let's take this function:

```
async Task<int> GetTotalAsync()
{
  int total = GetStartValue();
  int increment = await GetIncrementAsync();
  return total + increment;
}
```

This is shorthand for the following, with error checking has been elided for expository simplicity.

```
Task<int> GetTotalAsync()
{
  int total = GetStartValue();
  return GetIncrementAsync().ContinueWith((incrementTask) => {
    int increment = incrementTask.Result;
    return total + increment;
  });
}
```

(Actually, that's not really what happens; here are the gory details.)

The point is that the function executes normally until it encounters the first `await`, at which point it schedules itself as a continuation of the thing being awaited, and returns a new task that represents the continuation. When the thing being awaited completes, execution

resumes with the continuation. That continuation might do some work, and then perform another `await`, which once again schedules itself as a continuation of the thing being awaited. Eventually, the original function runs to completion, at which point the chain of tasks terminates with a result, namely the thing that the original function returned.

Note that when dealing with `async` functions, you have to distinguish with what the function *returns* and what it *produces* as a *result* when it *completes*. The *return value* is the thing that is returned synchronously by the function, typically a task of some sort. When execution reaches the end of the task chain, the task is said to have *completed*. The thing that comes out the end is called the *result*.

In other words, there are two ways to call an `async` function.

```
var task = SomethingAsync();
var result = await SomethingAsync();
```

If you call it without `await` then you get the raw task back. If you call it with `await`, then when the task completes, you get the result.

People who know how `async` and `await` work can start waking up now. You still know the stuff coming up next, but at least you'll be primed for the discussion to come after.

There are three ways of writing an `async` function:

- `async Task<T> SomethingAsync() { ... return t; }`
- `async Task SomethingAsync() { ... }`
- `async void SomethingAsync() { ... }`

In all the cases, the function is transformed into a chain of tasks. The difference is what the function returns.

In the first case, the function returns a task that eventually produces the `t`.

In the second case, the function returns a task which has no product, but you can still `await` on it to know when it has run to completion.

The third case is the nasty one. The third case is like the second case, except that *you don't even get the task back*. You have no way of knowing when the function's task has completed.

The `async void` case is a "fire and forget": You start the task chain, but you don't care about when it's finished. When the function returns, all you know is that everything up to the first `await` has executed. Everything after the first `await` will run at some unspecified point in the future that you have no access to.

Now that I've set up the story, we'll dig into the consequences next time.

Raymond Chen

Follow