Async-Async: Consequences for ordering of multiple calls in flight simultaneously

devblogs.microsoft.com/oldnewthing/20190502-00

May 2, 2019

Raymond Chen

The feature known as Async-Async makes asynchronous operations even more asynchronous by pretending that they started before they actually did. As I noted earlier, you might notice a difference if you have been breaking the rules and getting away with it. We saw last time what happens if you mutate a parameter that was passed to an asynchronous method. (Short answer: Don't do that until the asynchronous method completes.)

Another place you may notice a difference is if you race multiple calls against each other.

```
// Code in italics is wrong.

// Queue two animations in sequence.

// Queue the first animation.
var task1 = widget.QueueAnimationAsync(source1, curve1);

// Queue the second animation.
var task2 = widget.QueueAnimationAsync(source2, curve2);

// Wait for the tasks to complete.
await Task.WhenAll(task1, task2);
```

This code "knows" that the `Widget.QueueAnimationAsync` method chooses the point at which the animation will be added *before* it returns with an `IAsyncOperation`. It therefore "knows" that it can queue two items in order by starting the `QueueAnimationAsync` operations in order, and waiting for both of them to complete.

This code does not work when Async-Async is enabled because the two calls to `Widget.QueueAnimationAsync` are not required to start on the server in the same order that the client issued them. The fake `IAsyncOperation` issues a request to start the operation on the server, but does not wait for the server to acknowledge the start of the operation. If you start two operations, they race to the server, and the second one may reach the server first, in which case the operations will be started on the server in the opposite order.

Of course, proper client code should not have had this dependency on the order of asynchronous operations in the first place. After all, the server might decide not to choose the position of the animations until later in the asynchronous operation, and the second operation may have raced to the decision point ahead of the first operation. For example, the internal behavior of `QueueAnimationAsync` may have been

1. Create a new animation from the `source` and `curve` parameters.
2. Add that animation to the list of animations.

If you start two `QueueAnimationAsync` operations in parallel, you don't really know which one will reach step 2 first.

If the order of queueing is important, then you need to wait until the first animation is definitely queued before you queue the second one. You'll have to run the operations in series, rather than in parallel.

```
// Queue two animations in sequence.

// Queue the first animation.
await widget.QueueAnimationAsync(source1, curve1);

// Queue the second animation.
await widget.QueueAnimationAsync(source2, curve2);
```

In a sense, this is another case of "mutating a parameter passed to an asynchronous method": The parameter that is being mutated is the `widget` itself! Well, more specifically, the animation queue of the widget.

Next time, we'll look at another consequence of Async-Async that you may notice if you have been cheating the rules.

Raymond Chen

Follow