Parallel Programming with .NET

All about Async/Await, System.Threading.Tasks, System.Collections.Concurrent, System.Linq, and more…

Visual Studio

Await, SynchronizationContext, and Console Apps: Part 3

★★★★★

February 2, 2012 by Stephen Toub - MSFT // 11 Comments

In Part 1 and Part 2 of this short series, I demonstrated how you can build a SynchronizationContext and use it run an async method such that all of the continuations in that method will run on serialized on the current thread. This can be helpful when executing async methods in a console app, or in a unit test framework that doesn't directly support async methods. However, the support I showed thus far targets async methods that return Task… what about async methods that return void?

C# and Visual Basic support two flavors of async methods: ones that return tasks (either Task or Task<T>) and ones that return void. The former use the returned Task to represent the completion of the async method. In the case of an "async void" method, however, there is no returned Task to represent the method's processing. Instead, "async void" methods interact with the current SynchronizationContext to alert the context to the async method's execution status. Before entering the body of the async method, if there is a current SynchronizationContext, it is retrieved and its OperationStarted method is called. And after the async method has completed, that same context has its OperationCompleted method called. Further, if an exception goes unhandled in the body of the async void method, the throwing of that exception is Post to the SynchronizationContext, so

that the exception escapes back to the context for it to handle as it pleases.

All of this means that if we want our AsyncPump to be able to handle “async void” methods in addition to “async Task” methods, we need to augment the type slightly. First, we need to augment our SingleThreadSynchronizationContext to react appropriate to calls to OperationStarted and OperationCompleted. These methods need to maintain a count of how many outstanding operations there are, such that when the count reaches 0, we call Complete, just as before we called Complete when the async method’s Task completed. We do this by adding three members to the custom context:

```
private int m_operationCount = 0;

public override void OperationStarted()

{

   Interlocked.Increment(ref m_operationCount);

}
public override void OperationCompleted()
{
    if (Interlocked.Decrement(ref m_opera-
tionCount) == 0)
        Complete();
}
```

Then we need to add a new AsyncPump.Run overload that works with Action (for “async void” methods) instead of with Func<Task> (for “async Task” methods). As a reminder, here’s the existing Run method from our AsyncPump class:

```
public static void Run(Func<Task> asyncMethod)

{

   var prevCtx = SynchronizationContext.Current;

   try

   {
```

```
    var syncCtx = new SingleThreadSynchronizationContext(false);
    SynchronizationContext.SetSynchronizationContext(syncCtx);
        var t = asyncMethod();
        t.ContinueWith(delegate { syncCtx.Complete(); }, TaskScheduler.Default);
        syncCtx.RunOnCurrentThread();
        t.GetAwaiter().GetResult();
    }
    finally
    {
        SynchronizationContext.SetSynchronizationContext(prevCtx);
    }
}
```

Most of this will remain the same for our new Action-based variant. In fact, for the new one, we primarily just need to delete all the code having to do with the returned task, since there isn't one, and all notion of completion is being handled by the OperationStarted and OperationCompleted methods we added to the context. We do surround the asyncMethod invocation with calls to OperationStarted and OperationCompleted, just in case the asyncMethod is actually just a void method and not an "async void" method, in which case we need to make sure the operation count is greater than 0 for the duration of the invocation in order to avoid races that could result if the delegate invoked other async void methods.

```
public static void Run(Action asyncMethod)
{
  var prevCtx = SynchronizationContext.Current;
  try
  {
```

```
    var syncCtx = new SingleThreadSynchronizationContext(true);
    SynchronizationContext.SetSynchronizationContext(syncCtx);
    syncCtx.OperationStarted();
    asyncMethod();
    syncCtx.OperationCompleted();
        syncCtx.RunOnCurrentThread();
    }
    finally
    {
        SynchronizationContext.SetSynchro-
nizationContext(prevCtx);
    }
}
```

That's it (note that I've added a parameter to SingleThreadSynchronizationContext's constructor, which allows me to specify whether operation count tracking should be performed: we want it for this new Run method, but not for the previously described ones). We're now able to use our AsyncPump to run "async void" methods synchronously with all continuations executed on the current thread, e.g.

```
static void Main()
{
  AsyncPump.Run((Action)FooAsync);
}
static async void FooAsync()
{
  Foo1Async();
  await Foo2Async();
```

```
    Foo3Async();
}
static async void Foo1Async()
{
    await Task.Delay(1000);
    Console.WriteLine(1);
}
static async Task Foo2Async()
{
    await Task.Delay(1000);
    Console.WriteLine(2);
}
static async void Foo3Async()
{
    await Task.Delay(1000);
    Console.WriteLine(3);
}
```

Happy async'ing.

AsyncPump.cs

Recent Posts

Tasks, Monads, and LINQ April 3, 2013

"Invoke the method with await"... ugh! March 13, 2013

Live Now on Developer Tools Blogs

Speed up R with Parallel Programming in the Cloud

Announcing the Bing Maps Fleet Tracker Solution

VS Subscriptions and linking your VSTS account to AzureAD

Tags

.NET 4 .NET 4.5 Announcement Article Summary Async C++ Cancellation Code Samples Coordination Data Structures Dataflow Debugging F# FAQ Feedback Requested Media Message Passing MSDN New Feature? Parallel Extensions ParallelExtensionsExtras Parallelism Blockers PLINQ Release Silverlight Talks Task Parallel Library Testing ThreadPool Tools Visual Studio Visual Studio 2010

Videos

Related Resources

Visual Studio Product Website

Visual Studio Developer Center

Archives

All of 2009 (70)
All of 2008 (54)
All of 2007 (16)

Tags .NET 4.5 Async Parallel Extensions

Join the conversation

Add Comment

Yann ROBIN

5 years ago

Hi,

I'm trying to use async on one thread for a ConsoleApp. And I went accross an issue with TransactionScope.

When I do an await in a TransactionScope then I get weird exception, it's like the context of the scope is not put back in place. I get nesting error or TransactionAbort error.

Is there a workaround ?

Stephen Toub - MSFT

5 years ago

@Yann ROBIN: Can you share a small repro of the problem?

Yann ROBIN

5 years ago

Here is a small case repro : http://pastebin.com/Eh1dxG4a

In the TransactionScope there is a ThreadStatic and of course ThreadStatic is not saved in the ExecutionContext.

I don't know the reason to use ThreadStatic instead of something that could be saved by the ExecutionContext, but this a huge limitation for people who wanna use TransactionScope and async (TransactionScope is often use when you do Sql and async is clearly useful when you do a SqlRequest. Having to choose between having a Transaction or an async method is a bummer)

@Yann ROBIN: I see, thanks for the code sample. I'm well aware of TransactionScope's usage of thread-local storage instead of something like CallContext... I was asking for a repro because you mentioned you were using a single thread for all of the processing, and the biggest problem I've seen with await as a result of TransactionScope's implementation has been suspending on one thread and resuming on another, thereby not having TransactionScope's context. The issue you're seeing is related, which is that multiple scopes are being interleaved onto the same thread in a manner inconsistent with its nesting rules (you can see this if you change your repro to immediately await the task returned from TestStuff, rather than making multiple concurrent invocations). Note that the SQL team is aware of the issue and has been considering options (it would, for example, be a breaking change to just change TransactionScope's existing implementation over to using CallContext instead of TLS). That said, you can still use asynchrony with transactions, even with concurrency... you just need to manage the transactions yourself, e.g. providing a similar wrapper to what TransactionScope provides around Transaction, but using a different mechanism to flow it.

andreas_huber69@live.com

5 years ago

@Stephen: It seems Run(Func<Task>) would not work correctly if the passed async method does itself call async void methods. So it would make sense to implement Run(Func<Task>) in much the same way as Run(Action).

Other points:

– It seems syncCtx.OperationCompleted(); should be called in a finally block.

– BlockingCollection<T> is disposable, so it seems SingleThreadSynchronizationContext should also implement IDisposable and have its dispose method called from Run.

Stephen Toub - MSFT

5 years ago

@andreas_huber69:

Thank you for the feedback.

Regarding the Func<Task> invoking void methods, I personally disagree. void asynchronous methods are fire-and-forget. If a Func<Task> is invoking such a method and itself not doing something to wait for that operation to complete, then it shouldn't be up to the caller to need to wait for an internal implementation detail of the Func<Task>... the caller has no knowledge of what it's waiting for and whether waiting for it is a good idea or not. Async voids are really intended as top-level entry points, which is why I have the Run method that allows for using an async void as such an entrypoint. I can see arguments for doing what you're suggesting, but I personally wouldn't want to take that approach.

Regarding putting OperatingCompleted in a finally block, you certainly could. I didn't here because, assuming an "async void" or "async Task" is used, no exceptions will escape: the compiler itself ensures that any exceptions within the async method don't propagate out of the synchronous call. But, of course, even if my intention was for someone to use this code with such compiler-generated methods, they might not be, so you could certainly add a finally block to code defensively.

Regarding BlockingCollection<T> and IDisposable, sure, you could do that. Not disposing in this case and this usage is not going to be a performance problem, but you're of course welcome to augment it if you feel more comfortable doing so.

Again, thanks for taking the time to share your thoughts!

andreas_huber69@live.com *5 years ago*

@Stephen: Thanks for your comment.

I think I have a legitimate use case for an async Task method to internally call an async void method. Suppose you need to implement a communication client class that opens a TCP connection to a remote host and then needs to raise an event whenever it has received a message from the remote host. In such a scenario it feels natural to me to first await ConnectAsync and then call an async void method that contains the read loop. Of course, you'd have to make sure that the read loop exits e.g. when you call Dispose on the client.

Now, I do agree that the read loop is an implementation detail and the caller should not be forced to wait for it to exit. Since the connection to the remote host is private to the client, there's no prob-

lem when the read loop exits at its own pace while the caller moves on to do something else.

However, I'm currently using AsyncPump.Run exclusively in unit tests and I've found that it's useful to wait for all async activity to complete before the test can complete, so that e.g. a bug in read loop exit could be uncovered.

Stephen Toub - MSFT

5 years ago

@andreas_huber69:

Of course you should feel free to do in your own applications what you like. I would just be concerned about such a design showing up in a library: the caller of a Task-returning library method can reasonably expect that all work initiated by that method will have completed by the time the Task completes, and if the Task kicks off some async work that it doesn't wait for, that's not the case. In your example, I would expect a different API to be exposed, one that accepts a CancellationToken to support breaking out to the read loop, and that returns a Task which represents the total sum of the request processing. (Related, trying to achieve this with a Dispose implementation could be problematic, as failing to Dispose would likely leak resources, and any finalizers involved might not trigger if the on-going asynchronous operation is keeping the relevant objects alive. Even if you were to work around that, though, I still don't like the idea of Task-returning operations allowing work to last past the returned Task's completion.)

AceHack

5 years ago

Is there something easy I could add to this to make my errors catch at the right place in the debugger without having to turn on first chance? Thanks.

korggy

4 years ago

@AceHack — I have the exact same question. Haven't found a solution for it yet.

@AceHack and @korggy, the exception behavior your're seeing is normal, more details: stackoverflow.com/.../1768303
