Parallel Programming with .NET

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

Await, SynchronizationContext, and Console Apps: Part 2

★★★★★

January 21, 2012 by Stephen Toub - MSFT // 2 Comments

0 in 0

Yesterday, I blogged about how you can implement a custom SynchronizationContext in order to pump the continuations used by async methods so that they may be processed on a single, dedicated thread. I also highlighted that this is basically what UI frameworks like Windows Forms and Windows Presentation Foundation do with their message pumps.

Now that we understand the mechanics of how these things work, it's worth pointing out that we can achieve the same basic semantics without writing our own custom SynchronizationContext. Instead, we can use one that already exists in the .NET Framework: DispatcherSynchronizationContext. Through its Dispatcher class and its PushFrame method, WPF provides the ability to (as described in MSDN) "enter an execute loop" that "processes pending work items." This is exactly what our custom SynchronizationContext was doing with its usage of BlockingCollection<T>, so we can just use WPF's support instead of developing our own. (You might ask then why I started by describing how to do it manually and writing my own to exemplify it. I do so because I think it's important to really understand how things work; I typically find that developers write better higher-level code if they have the right mental model for what's happening under the covers, allowing them to better reason about bugs, about performance, about reliability, and the like.)

Below you can see how few lines of code it takes to achieve this support. (To compile this code, you’ll need to reference WindowsBase.dll to bring in the relevant WPF types.)

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Threading;

public static class AsyncPump
{
    public static void Run(Func<Task> func)
    {
        if (func == null) throw new ArgumentNullException(“func”);

        var prevCtx = SynchronizationContext.Current;
        try
        {
            var syncCtx = new DispatcherSynchronizationContext();
            SynchronizationContext.SetSynchronizationContext(syncCtx);

            var t = func();
            if (t == null) throw new InvalidOperationException();

            var frame = new DispatcherFrame();
            t.ContinueWith(_ => { frame.Continue = false; },
                TaskScheduler.Default);
            Dispatcher.PushFrame(frame);

            t.GetAwaiter().GetResult();
        }
        finally
        {
            SynchronizationContext.SetSynchronizationContext(prevCtx);
        }
    }
}
```

In short, we instantiate a DispatcherSynchronizationContext and publish it to be Current on the current thread; this is what enables the awaits inside of the async method being processed to queue their continuations back to this thread and this thread's Dispatcher. Then we instantiate a DispatcherFrame, which represents an execute loop for WPF's message pump. We use a continuation to signal to that DispatcherFrame when it should exit its loop (i.e. when the Task completes). And then we start the frame's execute loop via the PushFrame method.

All in all, just a few lines of code to achieve some powerful and useful behavior.

Search MSDN with Bing

○ Search this blog ● Search all blogs

Recent Posts

New Task APIs in .NET 4.6 February 2, 2015

.NET memory allocation profiling and Tasks April 4, 2013

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

February 2015 (1)
All of 2015 (1)
All of 2013 (6)
All of 2012 (46)
All of 2011 (37)
All of 2010 (67)
All of 2009 (70)
All of 2008 (54)
All of 2007 (16)

Tags .NET 4 .NET 4.5 Async Parallel Extensions Task Parallel Library

Join the conversation

Add Comment

Adrian

4 years ago

Hi Stephen

Thank you for those two articles on this await/SynchronizationContext topic, I read them with pleasure. I am currently using your AsyncPump.Run(..) within unit tests to properly test the viewmodels of my wpf app.

I do have one question from the pratical perspective:

When an exception occurs within this wpf application under test, the Visual Studio 2012 debugger properly breaks at the correct line. However the call stack only reaches back a few calls (until some Task.Run(..) within my wpf code) – the call stack does not reach back until the line of the unit test. within the function() delegate. It seems like some sort of task correlation is missing.

Is there a way I can adapt AsyncPump.Run(..) to support Visual Studio in showing the correct stack trace back to my unit test?

Thanks for any hints

Adrian

Adrian

4 years ago

Followup:

I replaced t.GetAwaiter().GetResult(); by t.Await() and that did the trick.

As far as I understand, those are equal except regarding their exception behavior.