Task.Run vs Task.Factory.StartNew

Stephen

October 24th, 2011

In .NET 4, Task.Factory.StartNew was the primary method for scheduling a new task. Many overloads provided for a highly configurable mechanism, enabling setting options, passing in arbitrary state, enabling cancellation, and even controlling scheduling behaviors. The flip side of all of this power is complexity. You need to know when to use which overload, what scheduler to provide, and the like. And "Task.Factory.StartNew" doesn't exactly roll off the tongue, at least not quickly enough for something that's used in such primary scenarios as easily offloading work to background processing threads.

So, in the .NET Framework 4.5 Developer Preview, we've introduced the new Task.Run method. This in no way obsoletes Task.Factory.StartNew, but rather should simply be thought of as a quick way to use Task.Factory.StartNew without needing to specify a bunch of parameters. It's a shortcut. In fact, Task.Run is actually implemented in terms of the same logic used for Task.Factory.StartNew, just passing in some default parameters. When you pass an Action to Task.Run:

```
Task.Run(someAction);
```

that's exactly equivalent to:

```
Task.Factory.StartNew(someAction,
    CancellationToken.None,
TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);
```

In this way, Task.Run can and should be used for the most common cases of simply offloading some work to be processed on the ThreadPool (what TaskScheduler.Default targets). That doesn't mean Task.Factory.StartNew will never again be used; far from it. Task.Factory.StartNew still has many important (albeit more advanced) uses. You get to control TaskCreationOptions for how the task behaves. You get to control the scheduler for where the task should be queued to and run. You get to use overloads that accept object state, which for performance-sensitive code paths can be used to avoid closures and the corresponding allocations. For the simple cases, though, Task.Run is your friend.

Task.Run provides eight overloads, to support all combinations of the following:

1. Task vs Task<TResult>
2. Cancelable vs non-cancelable
3. Synchronous vs asynchronous delegate

The first two bullets should be self-explanatory. For the first bullet, there are overloads that return Task (for operations that don't have a result) and there are overloads that return Task<TResult> (for operations that have a result of type TResult). There are also overloads that accept a CancellationToken, which enables the Task Parallel Library (TPL) to transition the task to a Canceled state if cancellation is requested prior to the task

beginning its execution.

The third bullet is more interesting, and is directly related to the async language support in C# and Visual Basic in Visual Studio 11. Let's consider Task.Factory.StartNew for a moment, as that will help to highlight what this distinction is. If I write the following call:

```
var t = Task.Factory.StartNew(() =>
{
    Task inner =Task.Factory.StartNew(() => {});
    return inner;
});
```

the type of 't' is going to be Task<Task>; the task's delegate is of type Func<TResult>, TResult in this case is a Task, and thus StartNew is returning a Task<Task>. Similarly, if I were to change that to be:

```
var t = Task.Factory.StartNew(() =>
{
    Task<int> inner = Task.Factory.StartNew(() => 42));
    return inner;
});
```

the type of 't' is now going to be Task<Task<int>>. The task's delegate is Func<TResult>, TResult is now Task<int>, and thus StartNew is returning Task<Task<int>>. Why is this relevant? Consider now what happens if I write the following:

```
var t = Task.Factory.StartNew(async delegate
{
    await Task.Delay(1000);
    return 42;
});
```

By using the async keyword here, the compiler is going to map this delegate to be a Func<Task<int>>: invoking the delegate will return the Task<int> to represent the eventual completion of this call. And since the delegate is Func<Task<int>>, TResult is Task<int>, and thus the type of 't' is going to be Task<Task<int>>, not Task<int>.

To handle these kinds of cases, in .NET 4 we introduced the Unwrap method. Unwrap has two overloads, which are both extensions methods, one on type Task<Task> and one on type Task<Task<TResult>>. We called this method Unwrap because it, in effect, "unwraps" the inner task that's returned as the result of the outer task. Calling Unwrap on a Task<Task> gives you back a new Task (which we often refer to as a proxy) which represents the eventual completion of the inner task. Similarly, calling Unwrap on a Task<Task<TResult>> gives you back a new Task<TResult> which represents the eventual completion of that inner task. (In both cases, if the outer task is Faulted or Canceled, there is no inner task, since there's no result from a task that doesn't run to completion, so the proxy task then represents the state of the outer task.) Going back to the prior example, if I wanted 't' to represent the return value of that inner task (in this case, the value 42), I could write:

```
var t = Task.Factory.StartNew(async delegate
{
    await Task.Delay(1000);
    return 42;
}).Unwrap();
```

The ‘t’ variable will now be of type Task<int>, representing the result of that asynchronous invocation.

Enter Task.Run. Because we expect it to be so common for folks to want to offload work to the ThreadPool, and for that work to use async/await, we decided to build this unwrapping functionality into Task.Run. That’s what’s referred to by the third bullet above. There are overloads of Task.Run that accept Action (for void-returning work), Func<TResult> (for TResult-returning work), Func<Task> (for void-returning async work), and Func<Task<TResult>> (for TResult-returning async work). Internally, then, Task.Run does the same kind of unwrapping that’s shown with Task.Factory.StartNew above. So, when I write:

```
var t = Task.Run(async delegate
{
    await Task.Delay(1000);
    return 42;
});
```

the type of ‘t’ is Task<int>, and the implementation of this overload of Task.Run is basically equivalent to:

```
var t = Task.Factory.StartNew(async delegate
{
    await Task.Delay(1000);
    return 42;
}, CancellationToken.None,
TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default).Unwrap();
```

As mentioned before, it’s a shortcut.

All of this then means that you can use Task.Run either with either regular lambdas/anonymous methods or with async lambdas/anonymous methods, and the right thing will just happen. If I wanted to offload this work to the ThreadPool and await its result, e.g.

```
int result = await Task.Run(async () =>
{
    await Task.Delay(1000);
    return 42;
});
```

the type of *result* will be int, just as you’d expect, and approximately one second after this work is invoked, the *result* variable be set to the value 42.

Interestingly, the new await keyword can almost be thought of as a language equivalent to the Unwrap method. So, if we return back to our Task.Factory.StartNew example, I could rewrite the last snippet above as follows using Unwrap:

```
int result = await Task.Factory.StartNew(async delegate
{
    await Task.Delay(1000);
    return 42;
}, CancellationToken.None,
TaskCreationOptions.DenyChildAttach,
TaskScheduler.Default).Unwrap();
```

or, instead of using Unwrap, I could use a second await:

```
int result = await await Task.Factory.StartNew(async delegate
{
    await Task.Delay(1000);
    return 42;
}, CancellationToken.None,
TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);
```

“await await” here is not a typo. Task.Factory.StartNew is returning a Task<Task<int>>. Await’ing that Task<Task<int>> returns a Task<int>, and awaiting that Task<int> returns an int. Fun, right?

Stephen Toub

Partner Software Engineer, .NET

Follow

Tagged .NET 4 .NET 4.5 Parallel Extensions Task Parallel Library

Read next

When at last you await

When you start using async methods heavily, you’ll likely see a particular pattern of composition pop up from time to time. Its structure is typically either ...

Stephen Toub - MSFT October 24, 2011

0 comment

Updated Async CTP

In April, we released the Async CTP Refresh, and since then we've seen fantastic adoption of the technology. We've also seen the technology landscape evolve. ...

Stephen Toub - MSFT November 1, 2011

0 comment

0 comments

Comments are closed. Login to edit/delete your existing comments

Relevant Links

corefx repository on GitHub

.NET

Microsoft Azure

Top Bloggers

Archive

Stay informed