Microsoft Logo

Gray Pipe

search clear

Implementing the Task-based Asynchronous Pattern

打印 导出 (0)

Implementing the Task-based Asynchronous Pattern

要查看英语原文，请勾选“英语”复选框。也可将鼠标指针移到文本上，在弹出窗口中显示英语原文。	翻译 英语

Implementing the Task-based Asynchronous Pattern

.NET Framework (current version)

你可以使用以下三种方式来实现基于任务的异步模式 (TAP)：即手动使用 C# 和 Visual Studio 中的 Visual Basic 编译器，或将编译器和手动方法结合使用。 以下各节详细地讨论了每一种方法。 你可以使用 TAP 模式来实现计算密集型和 I/O 密集型异步操作；“工作负载”一节讨论了每种类型的操作。

生成 TAP 方法

使用编译器

在 Visual Studio 2012 和 .NET Framework 4.5 中，任何具有 **async** 关键字（Visual Basic 中为 **Async**）的方法都被看作是一种异步方法，并且 C# 和 Visual Basic 编译器会执行必要的转换，以通过 TAP 来异步实现该方法。 异步方法应返回 System.Threading.Tasks.Task 或 System.Threading.Tasks.Task<TResult> 对象。 如果是后者，函数的主体应返回 **TResult**，并且编译器确保此结果是通过生成的任务对象得到的。 同样，未在方法的主体中处理的任何异常都会被封送处理为输出任务并导致生成的任务结束以 TaskStatus.Faulted 状态结束。 此异常发生在 OperationCanceledException（或派生类型）未得到处理时，在这种情况下生成的任务以 TaskStatus.Canceled 状态结束。

手动生成 TAP 方法

你可以手动实现 TAP 模式以更好地控制实现。 编译器依赖从 System.Threading.Tasks 命名空间公开的公共外围应用和 System.Runtime.CompilerServices 命名空间中支持的类型。 如要自己实现 TAP，你需要创建一个 TaskCompletionSource<TResult> 对象、执行异步操作，并在操作完成时，调用 SetResult、SetException、SetCanceled 方法，或调用这些方法之一的**Try**版本。 手动实现 TAP 方法时，需在所表示的异步操作完成时完成生成的任务。 例如：

C#

VB

```
public static Task<int> ReadTask(this Stream stream, byte[] buffer, int offset, int
count, object state)
{
   var tcs = new TaskCompletionSource<int>();
   stream.BeginRead(buffer, offset, count, ar =>
   {
      try { tcs.SetResult(stream.EndRead(ar)); }
      catch (Exception exc) { tcs.SetException(exc); }
   }, state);
   return tcs.Task;
}
```

混合方法

你可能发现手动实现 TAP 模式、但将实现核心逻辑委托给编译器的这种方法很有用。 例如，当你想要验证编译器生成的异步方法之外的实参时，可能需要使用这种混合方法，以便异常可以转义到该方法的直接调用方而不是通过 System.Threading.Tasks.Task 对象被公开：

C#

VB

```
public Task<int> MethodAsync(string input)
{
   if (input == null) throw new ArgumentNullException("input");
   return MethodAsyncInternal(input);
}

private async Task<int> MethodAsyncInternal(string input)
{

  // code that uses await goes here

  return value;
}
```

这种委托有用的另一种情况是：你在实施快速路径优化并想返回缓存的任务时。

工作负载

你可将计算密集型和 I/O 密集型异步操作作为 TAP 方法实现。 但是，当 TAP 方法从库中公开地公开时，应仅向涉及 I/O 密集型操作的工作负载提供这种方法（它们也可能涉及计算，但不是应仅仅是计算）。 如果一种方法仅受计算限制，则它应只能作为同步实现公开；使用它的代码选择是否要将该同步方法的调用包装到任务中，以将工作卸载到另一线程或实现并行。

计算密集型任务

System.Threading.Tasks.Task 类非常适合表示计算密集型操作。 默认情况下，它利用 ThreadPool 类中的特殊支持来提供有效的执行，还对执行异步计算的时间、地点和方式提供重要控制。
你可以通过以下方式生成计算密集型任务：

- 在 .NET Framework 4 中，使用 TaskFactory.StartNew 方法，这种方法接受异步执行委托（通常是 Action<T> 或 Func<TResult>）。 如果你提供 Action<T> 委托，该方法会返回表示异步执行该委托的 System.Threading.Tasks.Task 对象。 如果你提供 Func<TResult> 委托，该方法会返回 System.Threading.Tasks.Task<TResult> 对象。 StartNew 方法的重载接受一个取消标记（CancellationToken）、任务创建选项（TaskCreationOptions）和一个任务计划程序（TaskScheduler），它们都对计划和任务执行提供细粒度控制。 以当前任务计划程序为目标的工厂实例可用作 Task 类中的静态属性（Factory）；例如:Task.Factory.StartNew(...).
- 在 .NET Framework 4.5 中，将静态 Task.Run 方法用作 TaskFactory.StartNew 的快捷方式。 你可以使用 Run 来轻松启动针对线程池的计算密集型任务。 在 .NET Framework 4.5 中，这是用于启动一项计算密集型任务的首选机制。 仅当您需要对该任务进行更细化的控制，才直接使用 **StartNew**。
- 想要分别生成并计划任务时，请使用**Task**类型或**Start**方法的构造函数。 公共方法必须仅返回已开始的任务。
- 使用 Task.ContinueWith 方法的重载。 此方法创建一项在另一任务完成时已排好计划的新任务。 某些 ContinueWith 重载接受一个取消标记、延续选项和一个任务计划程序，以更好地控制计划和执行延续任务。
- 使用 TaskFactory.ContinueWhenAll 和 TaskFactory.ContinueWhenAny 方法。 这些方法会在提供的全部任务或任意一组任务完成时创建已计划的新任务。 这些方法还提供了用于控制这些任务的计划和执行的重载。

在计算密集型任务中，如果系统在开始运行任务之前收到取消请求，则它可以防止执行已计划的任务。 同样，如果你提供一个取消标记（CancellationToken 对象），则可以将标记

传递给监视该标记的异步代码。 你也可以将此标记提供给先前提过的方法（如 **StartNew** 或 **Run**），以便**Task**运行时也能监视该标记。
例如，请考虑使用呈现图像的异步方法。 任务的主体可以轮询取消标记，如果在呈现过程中收到取消请求，代码可提前退出。 此外，如果呈现启动之前收到取消请求，你需要阻止呈现操作：
C#
VB

```
internal Task<Bitmap> RenderAsync(
            ImageData data, CancellationToken cancellationToken)
{
   return Task.Run(() =>
   {
      var bmp = new Bitmap(data.Width, data.Height);
      for(int y=0; y<data.Height; y++)
      {
         cancellationToken.ThrowIfCancellationRequested();
         for(int x=0; x<data.Width; x++)
         {
            // render pixel [x,y] into bmp
         }
      }
      return bmp;
   }, cancellationToken);
}
```

如果满足下列条件之一，则计算密集型任务以 Canceled 状态结束：

- 取消请求通过 CancellationToken 对象到达，该对象在任务转换到 Running 状态前，作为创建方法的实参（例如 **StartNew** 或 **Run**）提供。
- OperationCanceledException 异常在此类任务的主体内未得到处理，该异常包含传给该任务的同一 CancellationToken，并且该标记显示已请求取消操作。

如果另一个异常在任务的主体内未得到处理，则此任务以 Faulted 状态结束，并且任何等待该任务或访问其结果的尝试都将引发异常。

I/O 密集型任务

若要创建一个不应由线程直接支持其全部执行的任务，请使用 TaskCompletionSource<TResult> 类型。 此类型公开一个返回关联 Task<TResult> 实例的 Task 属性。 此任务的生命周期是由 TaskCompletionSource<TResult> 方法控制的，比如 SetResult、SetException、SetCanceled 以及它们的 **TrySet** 变形。
假设你想创建一个将在指定时间段后完成的任务。 例如，你可能想延迟用户界面中的活动。 System.Threading.Timer 类已提供在指定时间段后以异步方式调用委托的能力，并且你可以通过使用 TaskCompletionSource<TResult> 将 Task<TResult> 前端放在计时器上，例如：

C#
VB

```
public static Task<DateTimeOffset> Delay(int millisecondsTimeout)
{
   TaskCompletionSource<DateTimeOffset> tcs = null;
   Timer timer = null;

   timer = new Timer(delegate
   {
      timer.Dispose();
      tcs.TrySetResult(DateTimeOffset.UtcNow);
   }, null, Timeout.Infinite, Timeout.Infinite);

   tcs = new TaskCompletionSource<DateTimeOffset>(timer);
   timer.Change(millisecondsTimeout, Timeout.Infinite);
   return tcs.Task;
}
```

从 .NET Framework 4.5 开始，Task.Delay 方法正是为此而提供的，并且你可以在另一个异步方法内使用它。例如，若要实现异步轮询循环：

C#
VB

```
public static async Task Poll(Uri url, CancellationToken cancellationToken,
                              IProgress<bool> progress)
{
   while(true)
   {
      await Task.Delay(TimeSpan.FromSeconds(10), cancellationToken);
```

```
        bool success = false;
        try
        {
            await DownloadStringAsync(url);
            success = true;
        }
        catch { /* ignore errors */ }
        progress.Report(success);
    }
}
```

TaskCompletionSource<TResult> 类并没有对应的非泛型类。 然而 Task<TResult> 派生自 Task，因此你可以为仅返回任务的 I/O 密集型方法使用泛型 TaskCompletionSource<TResult> 对象。 为了做到这一点，你可以使用具有虚拟 **TResult**（Boolean 是一个很好的默认选项，但是如果你担心 Task 用户将其向下转换成 Task<TResult>，那么你可以转而使用私有 **TResult** 类型）。 例如，上一个示例中的 Delay 方法返回现有时间和所产生的偏移量（Task<DateTimeOffset>）。 如果结果值是不必要的，则可对该方法进行如下改写（注意对 TrySetResult 的返回类型的更改和实参的更改）：

C#
VB

```
public static Task<bool> Delay(int millisecondsTimeout)
{
    TaskCompletionSource<bool> tcs = null;
    Timer timer = null;

    timer = new Timer(delegate
    {
        timer.Dispose();
        tcs.TrySetResult(true);
    }, null, Timeout.Infinite, Timeout.Infinite);

    tcs = new TaskCompletionSource<bool>(timer);
    timer.Change(millisecondsTimeout, Timeout.Infinite);
    return tcs.Task;
}
```

计算密集型和 I/O 密集型混合任务

异步方法不只局限于计算密集型或 I/O 密集型操作，还可以是两者的结合。 事实上，多个异步操作通常组合成较大的混合操作。 例如，请考虑前面示例中的 RenderAsync 方法，该方法执行计算密集型操作以根据某些输入 imageData 呈现图像。 此 imageData 可能来自你异步访问的 Web 服务：

C#

VB

```
public async Task<Bitmap> DownloadDataAndRenderImageAsync(
    CancellationToken cancellationToken)
{
    var imageData = await DownloadImageDataAsync(cancellationToken);
    return await RenderAsync(imageData, cancellationToken);
}
```

此示例还演示了如何通过多个异步操作使单个取消标记线程化。 有关详细信息，请参阅 Consuming the Task-based Asynchronous Pattern中的“取消的用法”一节。

另请参阅

Task-based Asynchronous Pattern (TAP)

Consuming the Task-based Asynchronous Pattern

Interop with Other Asynchronous Patterns and Types

显示: 继承 保护

打印共享

本文内容

- Microsoft Azure
- 更多...

学习资源

- Microsoft 虚拟学院
- 第 9 频道
- MSDN 杂志

社区

- 论坛
- 博客
- Codeplex

支持

- 自助支持

计划

- BizSpark（针对新创企业）
- Microsoft Imagine (for students)

中国 (简体中文)

- 新闻稿
- 隐私& Cookie
- 使用条款
- 商标

logo

© 2017 Microsoft