并行计算

项目2015/07/09

SynchronizationContext 综述

Stephen Cleary

多线程编程相当困难，而且要进行多线程编程需要了解无数概念和工具。为此，Microsoft .NET Framework 提供了 SynchronizationContext 类。很遗憾，很多开发人员甚至不知道这个有用的工具。

无论是什么平台（ASP.NET、Windows 窗体、Windows Presentation Foundation (WPF)、Silverlight 或其他），所有 .NET 程序都包含 SynchronizationContext 概念，并且所有多线程编程人员都可以通过理解和应用它获益。

SynchronizationContext 的必要性

多线程程序在 .NET Framework 出现之前就存在了。这些程序通常需要一个线程将一个工作单元传递给另一个线程。Windows 程序围绕消息循环进行，因此很多编程人员使用这一内置队列传递工作单元。每个要以这种方式使用 Windows 消息队列的多线程程序都必须定义自己的自定义 Windows 消息以及处理约定。

当 .NET Framework 首次发布时，这一通用模式是标准化模式。那时，.NET 唯一支持的 GUI 应用程序类型是 Windows 窗体。不过，框架设计人员期待其他模型，他们开发出了一种通用的解决方案。ISynchronizeInvoke 诞生了。

ISynchronizeInvoke 的原理是，一个“源”线程可以将一个委托列入“目标”线程队列，选择等待该委托完成。ISynchronizeInvoke 还提供了一个属性来确定当前代码是否已在目标线程上运行；这样，就不必使委托继续排队。Windows 窗体提供了唯一的 ISynchronizeInvoke 实现，并且开发了一种模式来设计异步组件，这样皆大欢喜。

.NET Framework 2.0 版包含很多重大改动。其中一项重要改进是在 ASP.NET 体系结构中引入了异步页面。在 .NET Framework 2.0 之前的版本中，每个 ASP.NET 请求都需要一个线程，直到该请求完成。这会造成线程利用率低下，因为创建网页通常依赖于数据库查询和 Web 服务调用，并且处理请求的线程必须等待，直到所有这些操作结束。使用异步页面，处理请求的线程可以开始每个操作，然后返回到 ASP.NET 线程池；当操作结束时，ASP.NET 线程池的另一个线程可以完成该请求。

但是，ISynchronizeInvoke 不太适合 ASP.NET 异步页面体系结构。使用 ISynchronizeInvoke 模式开发的异步组件在 ASP.NET 页面内无法正常工作，因为 ASP.NET 异步页面不与单个线程关

联。无须将工作排入原来的线程队列，异步页面只需对未完成的操作进行*计数*以确定页面请求何时可以完成。经过精心设计，SynchronizationContext 取代了 ISynchronizeInvoke。

SynchronizationContext 的概念

ISynchronizeInvoke 满足了两点需求：确定是否必须同步，使工作单元从一个线程列队等候另一个线程。设计 SynchronizationContext 是为了替代 ISynchronizeInvoke，但完成设计后，它就不仅仅是一个替代品了。

一方面，SynchronizationContext 提供了一种方式，可以使工作单元列队列入上下文。请注意，工作单元是列入*上下文*，而不是某个特定线程。这一区别非常重要，因为很多 SynchronizationContext 实现都不是基于单个特定线程的。SynchronizationContext 不包含用来确定是否必须同步的机制，因为这是不可能的。

SynchronizationContext 的另一方面是每个线程都有"当前"上下文。线程上下文不一定唯一；其上下文实例可以与多个其他线程共享。线程可以更改其当前上下文，但这样的情况非常少见。

SynchronizationContext 的第三个方面是它保持未完成操作的计数。这样，就可以使用 ASP.NET 异步页面和需要此类计数的任何其他主机。大多数情况下，捕获到当前 SynchronizationContext 时，计数递增；捕获到的 SynchronizationContext 用于将完成通知列队到上下文中时，计数递减。

SynchronizationContext 还有其他一些方面，但这些对大多数编程人员来说并不那么重要。**图 1** 中列出了一些最为重要的方面。

图 1 **SynchronizationContext API 的各方面**

C#

```
// The important aspects of the SynchronizationContext APIclass
SynchronizationContext

{

  // Dispatch work to the context.
void Post(..); // (asynchronously)

  void Send(..); // (synchronously)

  // Keep track of the number of asynchronous operations.
void OperationStarted();

  void OperationCompleted();

  // Each thread has a current context.
// If "Current" is null, then the thread's current context is
```

```
  // "new SynchronizationContext()", by convention.
static SynchronizationContext Current { get; }

  static void SetSynchronizationContext(SynchronizationContext);
}
```

SynchronizationContext 的实现

SynchronizationContext 的实际“上下文”并没有明确的定义。不同的框架和主机可以自行定义自己的上下文。通过了解这些不同的实现及其限制，可以清楚了解 SynchronizationContext 概念可以和不可以实现的功能。我将简单讨论部分实现。

WindowsFormsSynchronizationContext（*System.Windows.Forms.dll：System.Windows.Forms*）Windows 窗体应用程序会创建并安装一个 WindowsFormsSynchronizationContext 作为创建 UI 控件的任意线程的当前上下文。这一 SynchronizationContext 使用 UI 控件的 ISynchronizeInvoke 方法，该方法将委托传递给基础 Win32 消息循环。WindowsFormsSynchronizationContext 的上下文是一个单独的 UI 线程。

在 WindowsFormsSynchronizationContext 列队的所有委托一次一个地执行；它们通过一个特定 UI 线程执行以便列队。当前实现为每个 UI 线程创建一个 WindowsFormsSynchronizationContext。

DispatcherSynchronizationContext（*WindowsBase.dll：System.Windows.Threading*）WPF 和 Silverlight 应用程序使用 DispatcherSynchronizationContext，这样，委托按“常规”优先级在 UI 线程的调度程序中列队。当一个线程通过调用 Dispatcher.Run 开始其调度程序时，这一 SynchronizationContext 作为当前上下文安装。DispatcherSynchronizationContext 的上下文是一个单独的 UI 线程。

在 DispatcherSynchronizationContext 列队的所有委托一次一个地执行；它们通过一个特定 UI 线程执行以便列队。当前实现为每个顶层窗口创建一个 DispatcherSynchronizationContext，即使它们都使用相同的基础调度程序也是如此。

默认 (ThreadPool) SynchronizationContext（*mscorlib.dll：System.Threading*）默认 SynchronizationContext 是默认构造的 SynchronizationContext 对象。根据惯例，如果一个线程的当前 SynchronizationContext 为 null，那么它隐式具有一个默认 SynchronizationContext。

默认 SynchronizationContext 将其异步委托列队到 ThreadPool，但在调用线程上直接执行其同步委托。因此，其上下文包含所有 ThreadPool 线程以及调用 Send 的任何线程。此上下文“借用”调用 Send 的线程，将它们放入其上下文，直至委托完成。从这种意义上讲，默认上下文可以包含进程中的*所有*线程。

默认 SynchronizationContext 应用于 ThreadPool 线程，除非代码由 ASP.NET 承载。默认 SynchronizationContext 还隐式应用于显式子线程（Thread 类的实例），除非子线程设置自己

的 SynchronizationContext。因此，UI 应用程序通常有两个同步上下文：包含 UI 线程的 UI SynchronizationContext 和包含 ThreadPool 线程的默认 SynchronizationContext。

很多基于事件的异步组件使用默认 SynchronizationContext 无法正常工作。一个 BackgroundWorker 启动另一个 BackgroundWorker 这样的 UI 应用程序就是一个糟糕的示例。每个 BackgroundWorker 都捕获并使用调用 RunWorkerAsync 的线程的 SynchronizationContext，之后在该上下文中执行其 RunWorkerCompleted 事件。在只有一个 BackgroundWorker 的情况下，这通常是基于 UI 的 SynchronizationContext，因此 RunWorkerCompleted 在 RunWorkerAsync 捕获的 UI 上下文中执行（请参见**图 2**）。

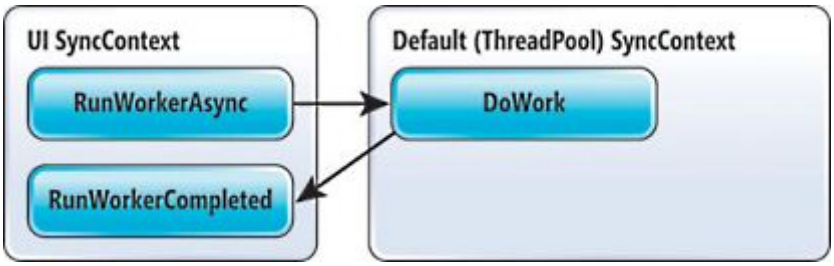

图 2 **UI 上下文中只有一个 BackgroundWorker**

但是，如果 BackgroundWorker 从其 DoWork 处理程序启动另一个 BackgroundWorker，那么嵌套的 BackgroundWorker 不会捕获 UI SynchronizationContext。DoWork 由 ThreadPool 线程使用默认 SynchronizationContext 执行。在这种情况下，嵌套的 RunWorkerAsync 将捕获默认 SynchronizationContext，因此它将对一个 ThreadPool 线程而不是 UI 线程执行其 RunWorkerCompleted（请参见**图 3**）。

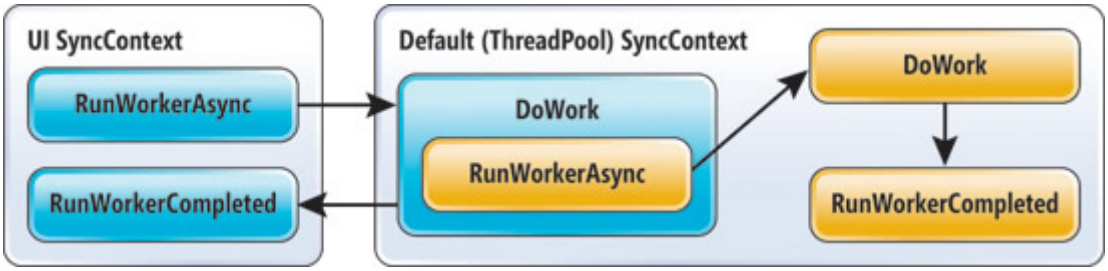

图 3 **UI 上下文中的嵌套 BackgroundWorker**

默认情况下，控制台应用程序和 Windows 服务中的所有线程都只有默认 SynchronizationContext。这会导致一些基于事件的异步组件失败。要解决这个问题，可以创建一个显式子线程，然后将 SynchronizationContext 安装在该线程上，这样就可以为这些组件提供上下文。本文不介绍如何实现 SynchronizationContext，不过，Nito.Async 库的 ActionThread 类 (nitoasync.codeplex.com) 可用作通用 SynchronizationContext 实现。

AspNetSynchronizationContext (*System.Web.dll:System.Web [internal class]*) ASP.NET SynchronizationContext 在线程池线程执行页面代码时安装在上面。当委托列对到捕获的 AspNetSynchronizationContext 中时，它恢复原始页面的标识和区域，然后直接执行委托。即使委托是通过调用 Post“异步”列队的，也会直接调用委托。

从概念上讲，AspNetSynchronizationContext 的上下文非常复杂。在异步页面的生存期中，该上下文从来自 ASP.NET 线程池的一个线程开始。异步请求开始后，该上下文不包含任何线程。异步请求结束时，执行其完成例程的线程池线程进入该上下文。这些可能是启动请求的线程，但更可能是操作完成时处于空闲状态的任何线程。

如果同一应用程序的多项操作同时完成，AspNetSynchronizationContext 确保一次只执行其中一项。它们可以在任意线程上执行，但该线程将具有原始页面的标识和区域。

一个常见的示例是在异步网页中使用 WebClient。DownloadDataAsync 将捕获当前 SynchronizationContext，之后在该上下文中执行其 DownloadDataCompleted 事件。当页面开始执行时，ASP.NET 会分配它的一个线程执行该页面中的代码。该页面可能调用 DownloadDataAsync，然后返回；ASP.NET 对未完成的异步操作进行计数，以便了解页面是否完成。当 WebClient 对象下载所请求的数据后，它将接收到一个线程池线程的通知。此线程将在捕获的上下文中引发 DownloadDataCompleted。该上下文将保持在相同的线程中，但会确保事件处理程序使用正确的标识和区域运行。

有关 SynchronizationContext 实现的注意事项

SynchronizationContext 提供了一种途径，可以在很多不同框架中编写组件。BackgroundWorker 和 WebClient 就是两个在 Windows 窗体、WPF、Silverlight、控制台和 ASP.NET 应用程序中同样应用自如的示例。但是，在设计这类可重用组件时，必须注意几点。

一般而言，SynchronizationContext 实现无法进行相等性比较。也就是说，ISynchronizeInvoke.InvokeRequired 没有等效项。不过，这不是多大的缺点；代码更为清晰，并且更容易验证它是否始终在已知上下文中执行，而不是试图处理多个上下文。

不是所有 SynchronizationContext 实现都可以保证委托执行顺序或委托同步顺序。基于 UI 的 SynchronizationContext 实现确实满足这些条件，但 ASP.NET SynchronizationContext 只提供同步。默认 SynchronizationContext 不保证执行顺序或同步顺序。

SynchronizationContext 实例和线程之间没有 1:1 的对应关系。WindowsFormsSynchronizationContext 确实 1:1 映射到一个线程（只要不调用 SynchronizationContext.CreateCopy），但任何其他实现都不是这样。一般而言，最好不要假设任何上下文实例将在任何指定线程上运行。

最后，SynchronizationContext.Post 方法不一定是异步的。大多数实现异步实现此方法，但 AspNetSynchronizationContext 是一个明显的例外。这会导致无法预料的重入问题。**图 4** 总结了这些不同的实现。

图 4 **SynchronizationContext 实现摘要**

展开表

	使用特定线程执行委托	独占（一次执行一个委托）	有序（委托按队列顺序执行）	Send 可以直接调用委托	Post 可以直接调用委托
Windows 窗体	能	能	能	如果从 UI 线程调用	從不
WPF/Silverlight	能	能	能	如果从 UI 线程调用	從不
默认	不能	不能	不能	Always	從不
ASP.NET	不能	能	不能	Always	Always

AsyncOperationManager 和 AsyncOperation

.NET Framework 中的 AsyncOperationManager 和 AsyncOperation 类是 SynchronizationContext 抽象的轻型包装。AsyncOperationManager 在第一次创建 AsyncOperation 时捕获当前 SynchronizationContext，如果当前 SynchronizationContext 为 null，则使用默认 SynchronizationContext。AsyncOperation 将委托异步发布到捕获的 SynchronizationContext。

大多数基于事件的异步组件都在其实现中使用 AsyncOperationManager 和 AsyncOperation。这些对于具有明确完成点的异步操作（即异步操作从一个点开始，以另一个点的事件结束）非常有效。其他异步通知可能没有明确的完成点；它们可能是一种订阅类型，在一个点开始，然后无限期持续。对于这些类型的操作，可以直接捕获和使用 SynchronizationContext。

新组件不应使用基于事件的异步模式。Visual Studio 异步社区技术预览 (CTP) 包含一篇描述基于任务的异步模式的文档，在这种模式下，组件返回 Task 和 Task<TResult> 对象，而不是通过 SynchronizationContext 引发事件。基于任务的 API 是 .NET 中异步编程的发展方向。

SynchronizationContext 的库支持示例

像 BackgroundWorker 和 WebClient 这样的简单组件是隐式自带的，隐藏了 SynchronizationContext 捕获和使用。很多库以更可见的方式使用 SynchronizationContext。通过使用 SynchronizationContext 公开 API，库不仅获得了框架独立性，而且为高级最终用户提供了一个可扩展点。

除了下面讨论的库，当前 SynchronizationContext 也被视为 ExecutionContext 的一部分。任何捕获线程的 ExecutionContext 的系统都会捕获当前 SynchronizationContext。当恢复 ExecutionContext 时，通常也会恢复 SynchronizationContext。

Windows Communication Foundation (WCF)：UseSynchronizationContext WCF 有两个用于配置服务器和客户端行为的特性：ServiceBehaviorAttribute 和 CallbackBehaviorAttribute。这

两个特性都有一个 Boolean 属性：UseSynchronizationContext。此特性的默认值为 true，这表示在创建通信通道时捕获当前 SynchronizationContext，这一捕获的 SynchronizationContext 用于使约定方法列队。

通常，这一行为正是我们所需要的：服务器使用默认 SynchronizationContext，客户端回调使用相应的 UI SynchronizationContext。在需要重入时，这会导致问题，如客户端调用的服务器方法调用一个客户端回调。在这类情况下，将 UseSynchronizationContext 设置为 false 可以禁止 WCF 自动使用 SynchronizationContext。

上面只是简单介绍了 WCF 如何使用 SynchronizationContext。有关详细信息，请参阅 *MSDN 杂志* 2007 年 11 月刊中的文章“WCF 中的同步上下文”(msdn.microsoft.com/magazine/cc163321)。

Windows Workflow Foundation (WF)：WorkflowInstance.SynchronizationContext WF 主机最初使用 WorkflowSchedulerService 和派生类型控制如何在线程上安排工作流活动。部分 .NET Framework 4 升级在 WorkflowInstance 类及其派生类 derived WorkflowApplication 上包含 SynchronizationContext 属性。

如果承载进程创建自己的 WorkflowInstance，则可以直接设置 SynchronizationContext。WorkflowInvoker.InvokeAsync 也使用 SynchronizationContext，它捕获当前 SynchronizationContext 并将其传递给其内部 WorkflowApplication。然后该 SynchronizationContext 用于发布工作流完成事件以及工作流活动。

任务并行库 (TPL)：TaskScheduler.FromCurrentSynchronizationContext 和 CancellationToken.Register TPL 使用任务对象作为其工作单元并通过 TaskScheduler 执行。默认 TaskScheduler 的作用类似于默认 SynchronizationContext，将任务在 ThreadPool 中列队。TPL 队列还提供了另一个 TaskScheduler，将任务在 SynchronizationContext 中列队。UI 更新的进度报告可以在一个嵌套任务中完成，如**图 5** 所示。

图 5 **UI 更新的进度报告**

C#

```
private void button1_Click(object sender, EventArgs e)
{
  // This TaskScheduler captures SynchronizationContext.Current.
TaskScheduler taskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
  // Start a new task (this uses the default TaskScheduler,
  // so it will run on a ThreadPool thread).
Task.Factory.StartNew(() =>
  {
    // We are running on a ThreadPool thread here.
; // Do some work.
// Report progress to the UI.
Task reportProgressTask = Task.Factory.StartNew(() =>
      {
        // We are running on the UI thread here.
```

```
; // Update the UI with our progress.
},
      CancellationToken.None,
      TaskCreationOptions.None,
      taskScheduler);
    reportProgressTask.Wait();

    ; // Do more work.
});
}
```

在 .NET Framework 4 中，CancellationToken 类可用于任意类型的取消操作。 为了与现有取消操作形式集成，该类允许注册委托以在请求取消时调用。 注册委托后，就可以传递 SynchronizationContext 了。 如果出现取消请求，CancellationToken 将该委托列入 SynchronizationContext 队列而不是直接执行它。

Microsoft 被动扩展 (Rx)：ObserveOn、 SubscribeOn 和 SynchronizationContextScheduler Rx 是一个库，它将事件视为数据流。 ObserveOn 运算符通过一个 SynchronizationContext 将事件列队，SubscribeOn 运算符通过一个 SynchronizationContext 将对这些事件的*订阅*列队。 ObserveOn 通常用于使用传入事件更新 UI，SubscribeOn 用于从 UI 对象使用事件。

Rx 还有它自己的工作单元列队方法：IScheduler 接口。 Rx 包含 SynchronizationContextScheduler，这是一个列入 SynchronizationContext 的 IScheduler 实现。

Visual Studio Async CTP：await、ConfigureAwait、 SwitchTo 和 EventProgress<T> Visual Studio 对异步代码转换的支持是在 2010 年 Microsoft 专业开发人员大会上发布的。 默认情况下，当前 SynchronizationContext 在一个等待点捕获，此 SynchronizationContext 用于在等待后继续（更确切地说，*仅当它不为 null 时*，才捕获当前 SynchronizationContext，如果为 null，则捕获当前 TaskScheduler）：

C#

```
private async void button1_Click(object sender, EventArgs e)
{
  // SynchronizationContext.Current is implicitly captured by await.
var data = await webClient.DownloadStringTaskAsync(uri);

  // At this point, the captured SynchronizationContext was used to resume
  // execution, so we can freely update UI objects.
}
```

ConfigureAwait 提供了一种途径避免捕获 SynchronizationContext 捕获；为 flowContext 参数传递 false 会阻止使用 SynchronizationContext 在等待后继续执行。 SynchronizationContext 实例还有一种扩展方法 SwitchTo；使用该方法，任何异步方法都可以通过调用 SwitchTo 并等待结果，更改为不同的 SynchronizationContext。

异步 CTP 引入了报告异步操作进展的通用模式：IProgress<T> 接口及其实现 EventProgress<T>。 该类在构造时捕获当前 SynchronizationContext 并在此上下文中引发其 ProgressChanged 事件。

除了这一支持外，返回 void 的异步方法还在异步操作开始时递增计数，在异步操作结束后递减计数。 这一行为使返回 void 的异步方法类似于顶层异步操作。

限制和功能

了解 SynchronizationContext 对任何编程人员来说都是有益的。 现有跨框架组件使用它同步其事件。 库可以将它公开以获得更高的灵活性。 技术精湛的编程人员了解 SynchronizationContext 限制和功能后，可以更好地编写和利用这些类。

Stephen Cleary *自第一次听到多线程这个概念，就对它具有浓厚的兴趣。他为很多主要客户完成了很多关键业务多任务处理系统，这些客户包括 Syracuse News、R. R. Donnelley 和 BlueScope Steel。他经常在 .NET 用户组、BarCamp 和北密歇根他家附近的 .NET 日活动上发言，主题通常与多线程有关。他的编程博客位于 nitoprograms.com 。*

*衷心感谢以下技术专家对本文的审阅：**Eric Eilebrecht***