**异步编程模型**

* 项目
* 2022/02/08

通过使用异步编程，你可以避免性能瓶颈并增强应用程序的总体响应能力。 但是，编写异步应用程序的传统技术可能比较复杂，使它们难以编写、调试和维护。

[C# 5](https://docs.microsoft.com/zh-cn/dotnet/csharp/whats-new/csharp-version-history#c-version-50) 引入了一种简便方法，即异步编程。此方法利用了 .NET Framework 4.5 及更高版本、.NET Core 和 Windows 运行时中的异步支持。 编译器可执行开发人员曾进行的高难度工作，且应用程序保留了一个类似于同步代码的逻辑结构。 因此，你只需做一小部分工作就可以获得异步编程的所有好处。

本主题概述了何时以及如何使用异步编程，并包括指向包含详细信息和示例的支持主题的链接。

**异步编程提升响应能力**

异步对可能会被屏蔽的活动（如 Web 访问）至关重要。 对 Web 资源的访问有时很慢或会延迟。 如果此类活动在同步过程中被屏蔽，整个应用必须等待。 在异步过程中，应用程序可继续执行不依赖 Web 资源的其他工作，直至潜在阻止任务完成。

下表显示了异步编程提高响应能力的典型区域。 列出的 .NET 和 Windows 运行时 API 包含支持异步编程的方法。

| **异步编程提升响应能力** | | |
| --- | --- | --- |
| **应用程序区域** | **包含异步方法的 .NET 类型** | **包含异步方法的 Windows 运行时类型** |
| Web 访问 | [HttpClient](https://docs.microsoft.com/zh-cn/dotnet/api/system.net.http.httpclient) | [Windows.Web.Http.HttpClient](https://docs.microsoft.com/zh-cn/uwp/api/windows.web.http.httpclient) [SyndicationClient](https://docs.microsoft.com/zh-cn/uwp/api/windows.web.syndication.syndicationclient) |
| 使用文件 | [JsonSerializer](https://docs.microsoft.com/zh-cn/dotnet/api/system.text.json.jsonserializer) [StreamReader](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.streamreader) [StreamWriter](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.streamwriter) [XmlReader](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.xmlreader) [XmlWriter](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.xmlwriter) | [StorageFile](https://docs.microsoft.com/zh-cn/uwp/api/windows.storage.storagefile) |
| 使用图像 |  | [MediaCapture](https://docs.microsoft.com/zh-cn/uwp/api/windows.media.capture.mediacapture) [BitmapEncoder](https://docs.microsoft.com/zh-cn/uwp/api/windows.graphics.imaging.bitmapencoder) [BitmapDecoder](https://docs.microsoft.com/zh-cn/uwp/api/windows.graphics.imaging.bitmapdecoder) |
| WCF 编程 | [同步和异步操作](https://docs.microsoft.com/zh-cn/dotnet/framework/wcf/synchronous-and-asynchronous-operations) |  |

由于所有与用户界面相关的活动通常共享一个线程，因此，异步对访问 UI 线程的应用程序来说尤为重要。 如果任何进程在同步应用程序中受阻，则所有进程都将受阻。 你的应用程序停止响应，因此，你可能在其等待过程中认为它已经失败。

使用异步方法时，应用程序将继续响应 UI。 例如，你可以调整窗口的大小或最小化窗口；如果你不希望等待应用程序结束，则可以将其关闭。

当设计异步操作时，该基于异步的方法将自动传输的等效对象添加到可从中选择的选项列表中。 开发人员只需要投入较少的工作量即可使你获取传统异步编程的所有优点。

**异步方法易于编写**

C# 中的 [Async](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/async) 和 [Await](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/await) 关键字是异步编程的核心。 通过这两个关键字，可以使用 .NET Framework、.NET Core 或 Windows 运行时中的资源，轻松创建异步方法（几乎与创建同步方法一样轻松）。 使用 async 关键字定义的异步方法简称为“异步方法”。

下面的示例演示了一种异步方法。 你应对代码中的几乎所有内容都很熟悉。

可从 [C# 中使用 Async 和 Await 的异步编程](https://docs.microsoft.com/zh-cn/samples/dotnet/samples/async-and-await-cs)中找到可供下载的完整 Windows Presentation Foundation (WPF) 示例。

C#复制

public async Task<int> GetUrlContentLengthAsync()

{

var client = new HttpClient();

Task<string> getStringTask =

client.GetStringAsync("https://docs.microsoft.com/dotnet");

DoIndependentWork();

string contents = await getStringTask;

return contents.Length;

}

void DoIndependentWork()

{

Console.WriteLine("Working...");

}

可以从前面的示例中了解几种做法。 从方法签名开始。 它包含 async 修饰符。 返回类型为 Task<int>（有关更多选项，请参阅“返回类型”部分）。 方法名称以 Async 结尾。 在方法的主体中，GetStringAsync 返回 Task<string>。 这意味着在 await 任务时，将获得 string (contents)。 在等待任务之前，可以通过 GetStringAsync 执行不依赖于 string 的工作。

密切注意 await 运算符。 它会暂停 GetUrlContentLengthAsync：

* 在 getStringTask 完成之前，GetUrlContentLengthAsync 无法继续。
* 同时，控件返回至 GetUrlContentLengthAsync 的调用方。
* 当 getStringTask 完成时，控件将在此处继续。
* 然后，await 运算符会从 getStringTask 检索 string 结果。

return 语句指定整数结果。 任何等待 GetUrlContentLengthAsync 的方法都会检索长度值。

如果 GetUrlContentLengthAsync 在调用 GetStringAsync 和等待其完成期间不能进行任何工作，则你可以通过在下面的单个语句中调用和等待来简化代码。

C#复制

string contents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");

以下特征总结了使上一个示例成为异步方法的原因：

* 方法签名包含 async 修饰符。
* 按照约定，异步方法的名称以“Async”后缀结尾。
* 返回类型为下列类型之一：
  + 如果你的方法有操作数为 TResult 类型的返回语句，则为 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1)。
  + 如果你的方法没有返回语句或具有没有操作数的返回语句，则为 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task)。
  + void：如果要编写异步事件处理程序。
  + 包含 GetAwaiter 方法的其他任何类型（自 C# 7.0 起）。

有关详细信息，请参阅[返回类型和参数](https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model#BKMK_ReturnTypesandParameters)部分。

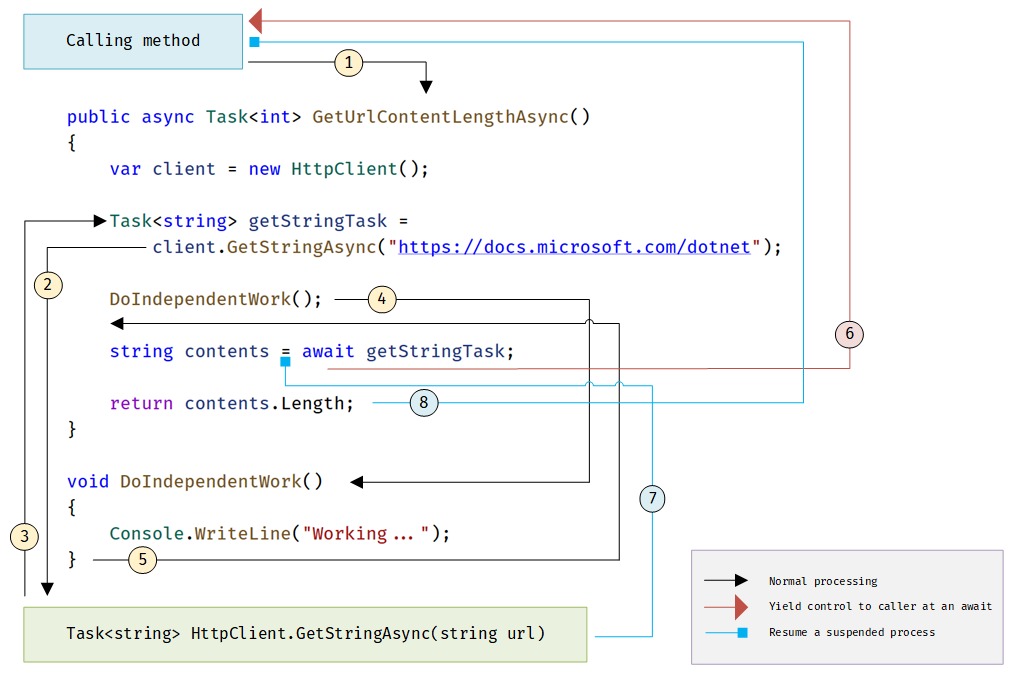
* 方法通常包含至少一个 await 表达式，该表达式标记一个点，在该点上，直到等待的异步操作完成方法才能继续。 同时，将方法挂起，并且控件返回到方法的调用方。 本主题的下一节将解释悬挂点发生的情况。

在异步方法中，可使用提供的关键字和类型来指示需要完成的操作，且编译器会完成其余操作，其中包括持续跟踪控件以挂起方法返回等待点时发生的情况。 一些常规流程（例如，循环和异常处理）在传统异步代码中处理起来可能很困难。 在异步方法中，元素的编写频率与同步解决方案相同且此问题得到解决。

若要详细了解旧版 .NET Framework 中的异步性，请参阅 [TPL 和传统 .NET Framework 异步编程](https://docs.microsoft.com/zh-cn/dotnet/standard/parallel-programming/tpl-and-traditional-async-programming)。

**异步方法的运行机制**

异步编程中最需弄清的是控制流是如何从方法移动到方法的。 下图可引导你完成此过程：

[](https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/concepts/async/media/task-asynchronous-programming-model/navigation-trace-async-program.png#lightbox)

关系图中的数字对应于以下步骤，在调用方法调用异步方法时启动。

1. 调用方法调用并等待 GetUrlContentLengthAsync 异步方法。
2. GetUrlContentLengthAsync 可创建 [HttpClient](https://docs.microsoft.com/zh-cn/dotnet/api/system.net.http.httpclient) 实例并调用 [GetStringAsync](https://docs.microsoft.com/zh-cn/dotnet/api/system.net.http.httpclient.getstringasync) 异步方法以下载网站内容作为字符串。
3. GetStringAsync 中发生了某种情况，该情况挂起了它的进程。 可能必须等待网站下载或一些其他阻止活动。 为避免阻止资源，GetStringAsync 会将控制权出让给其调用方 GetUrlContentLengthAsync。

GetStringAsync 返回 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1)，其中 TResult 为字符串，并且 GetUrlContentLengthAsync 将任务分配给 getStringTask 变量。 该任务表示调用 GetStringAsync 的正在进行的进程，其中承诺当工作完成时产生实际字符串值。

1. 由于尚未等待 getStringTask，因此，GetUrlContentLengthAsync 可以继续执行不依赖于 GetStringAsync 得出的最终结果的其他工作。 该任务由对同步方法 DoIndependentWork 的调用表示。
2. DoIndependentWork 是完成其工作并返回其调用方的同步方法。
3. GetUrlContentLengthAsync 已运行完毕，可以不受 getStringTask 的结果影响。 接下来，GetUrlContentLengthAsync 需要计算并返回已下载的字符串的长度，但该方法只有在获得字符串的情况下才能计算该值。

因此，GetUrlContentLengthAsync 使用一个 await 运算符来挂起其进度，并把控制权交给调用 GetUrlContentLengthAsync 的方法。 GetUrlContentLengthAsync 将 Task<int> 返回给调用方。 该任务表示对产生下载字符串长度的整数结果的一个承诺。

**备注**

如果 GetStringAsync（因此 getStringTask）在 GetUrlContentLengthAsync 等待前完成，则控制会保留在 GetUrlContentLengthAsync 中。 如果异步调用过程 getStringTask 已完成，并且 GetUrlContentLengthAsync 不必等待最终结果，则挂起然后返回到 GetUrlContentLengthAsync 将造成成本浪费。

在调用方法中，处理模式会继续。 在等待结果前，调用方可以开展不依赖于 GetUrlContentLengthAsync 结果的其他工作，否则就需等待片刻。 调用方法等待 GetUrlContentLengthAsync，而 GetUrlContentLengthAsync 等待 GetStringAsync。

1. GetStringAsync 完成并生成一个字符串结果。 字符串结果不是通过按你预期的方式调用 GetStringAsync 所返回的。 （记住，该方法已返回步骤 3 中的一个任务）。 相反，字符串结果存储在表示 getStringTask 方法完成的任务中。 await 运算符从 getStringTask 中检索结果。 赋值语句将检索到的结果赋给 contents。
2. 当 GetUrlContentLengthAsync 具有字符串结果时，该方法可以计算字符串长度。 然后，GetUrlContentLengthAsync 工作也将完成，并且等待事件处理程序可继续使用。 在此主题结尾处的完整示例中，可确认事件处理程序检索并打印长度结果的值。 如果你不熟悉异步编程，请花 1 分钟时间考虑同步行为和异步行为之间的差异。 当其工作完成时（第 5 步）会返回一个同步方法，但当其工作挂起时（第 3 步和第 6 步），异步方法会返回一个任务值。 在异步方法最终完成其工作时，任务会标记为已完成，而结果（如果有）将存储在任务中。

**API 异步方法**

你可能想知道从何处可以找到 GetStringAsync 等支持异步编程的方法。 .NET Framework 4.5 或更高版本以及 .NET Core 包含许多可与 async 和 await 结合使用的成员。 可以通过追加到成员名称的“Async”后缀和 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task) 或 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1) 的返回类型，识别这些成员。 例如，System.IO.Stream 类包含 [CopyToAsync](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.copytoasync)、[ReadAsync](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.readasync) 和 [WriteAsync](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.writeasync) 等方法，以及同步方法 [CopyTo](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.copyto)、[Read](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.read) 和 [Write](https://docs.microsoft.com/zh-cn/dotnet/api/system.io.stream.write)。

Windows 运行时也包含许多可以在 Windows 应用中与 async 和 await 结合使用的方法。 有关详细信息，请参阅[线程处理和异步编程](https://docs.microsoft.com/zh-cn/windows/uwp/threading-async/)进行 UWP 开发；如果使用的是旧版 Windows 运行时，还请参阅[异步编程（Windows 应用商店应用）](https://docs.microsoft.com/zh-cn/previous-versions/windows/apps/hh464924(v=win.10))和[快速入门：在 C# 或 Visual Basic 中调用异步 API](https://docs.microsoft.com/zh-cn/previous-versions/windows/apps/hh452713(v=win.10))。

**线程**

异步方法旨在成为非阻止操作。 异步方法中的 await 表达式在等待的任务正在运行时不会阻止当前线程。 相反，表达式在继续时注册方法的其余部分并将控件返回到异步方法的调用方。

async 和 await 关键字不会创建其他线程。 因为异步方法不会在其自身线程上运行，因此它不需要多线程。 只有当方法处于活动状态时，该方法将在当前同步上下文中运行并使用线程上的时间。 可以使用 [Task.Run](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task.run) 将占用大量 CPU 的工作移到后台线程，但是后台线程不会帮助正在等待结果的进程变为可用状态。

对于异步编程而言，该基于异步的方法优于几乎每个用例中的现有方法。 具体而言，此方法比 [BackgroundWorker](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.backgroundworker) 类更适用于 I/O 绑定操作，因为此代码更简单且无需防止争用条件。 结合 [Task.Run](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task.run) 方法使用时，异步编程比 [BackgroundWorker](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.backgroundworker) 更适用于 CPU 绑定操作，因为异步编程将运行代码的协调细节与 Task.Run 传输至线程池的工作区分开来。

**async 和 await**

如果使用 [async](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/async) 修饰符将某种方法指定为异步方法，即启用以下两种功能。

* 标记的异步方法可以使用 [await](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/await) 来指定暂停点。 await 运算符通知编译器异步方法：在等待的异步过程完成后才能继续通过该点。 同时，控制返回至异步方法的调用方。

异步方法在 await 表达式执行时暂停并不构成方法退出，只会导致 finally 代码块不运行。

* 标记的异步方法本身可以通过调用它的方法等待。

异步方法通常包含 await 运算符的一个或多个实例，但缺少 await 表达式也不会导致生成编译器错误。 如果异步方法未使用 await 运算符标记暂停点，则该方法会作为同步方法执行，即使有 async 修饰符，也不例外。 编译器将为此类方法发布一个警告。

async 和 await 都是上下文关键字。 有关更多信息和示例，请参见以下主题：

* [async](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/async)
* [await](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/await)

**返回类型和参数**

异步方法通常返回 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task) 或 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1)。 在异步方法中，await 运算符应用于通过调用另一个异步方法返回的任务。

如果方法包含指定 TResult 类型操作数的 [return](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/statements/jump-statements#the-return-statement) 语句，将 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1) 指定为返回类型。

如果方法不含任何 return 语句或包含不返回操作数的 return 语句，则将 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task) 用作返回类型。

自 C# 7.0 起，还可以指定任何其他返回类型，前提是类型包含 GetAwaiter 方法。 例如，[ValueTask<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.valuetask-1) 就是这种类型。 可用于 [System.Threading.Tasks.Extension](https://www.nuget.org/packages/System.Threading.Tasks.Extensions/) NuGet 包。

下面的示例演示如何声明并调用可返回 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1) 或 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task) 的方法：

C#复制

async Task<int> GetTaskOfTResultAsync()

{

int hours = 0;

await Task.Delay(0);

return hours;

}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();

int intResult = await returnedTaskTResult;

// Single line

// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()

{

await Task.Delay(0);

// No return statement needed

}

Task returnedTask = GetTaskAsync();

await returnedTask;

// Single line

await GetTaskAsync();

每个返回的任务表示正在进行的工作。 任务可封装有关异步进程状态的信息，如果未成功，则最后会封装来自进程的最终结果或进程引发的异常。

异步方法也可以具有 void 返回类型。 该返回类型主要用于定义需要 void 返回类型的事件处理程序。 异步事件处理程序通常用作异步程序的起始点。

无法等待具有 void 返回类型的异步方法，并且无效返回方法的调用方捕获不到异步方法引发的任何异常。

异步方法无法声明 [in](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/in-parameter-modifier)、[ref](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/ref) 或 [out](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/out-parameter-modifier) 参数，但可以调用包含此类参数的方法。 同样，异步方法无法通过引用返回值，但可以调用包含 ref 返回值的方法。

有关详细信息和示例，请参阅[异步返回类型 (C#)](https://docs.microsoft.com/zh-cn/dotnet/csharp/programming-guide/concepts/async/async-return-types)。 若要详细了解如何在异步方法中捕获异常，请参阅 [try-catch](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/keywords/try-catch)。

Windows 运行时编程中的异步 API 具有下列返回类型之一（类似于任务）：

* [IAsyncOperation<TResult>](https://docs.microsoft.com/zh-cn/uwp/api/windows.foundation.iasyncoperation-1)（对应于 [Task<TResult>](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task-1)）
* [IAsyncAction](https://docs.microsoft.com/zh-cn/uwp/api/windows.foundation.iasyncaction)（对应于 [Task](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.tasks.task)）
* [IAsyncActionWithProgress<TProgress>](https://docs.microsoft.com/zh-cn/uwp/api/windows.foundation.iasyncactionwithprogress-1)
* [IAsyncOperationWithProgress<TResult,TProgress>](https://docs.microsoft.com/zh-cn/uwp/api/windows.foundation.iasyncoperationwithprogress-2)

**命名约定**

按照约定，返回常规可等待类型的方法（例如 Task、Task<T>、ValueTask 和 ValueTask<T>）应具有以“Async”结束的名称。 启动异步操作但不返回可等待类型的方法不得具有以“Async”结尾的名称，但其开头可以为“Begin”、“Start”或其他表明此方法不返回或引发操作结果的动词。

如果某一约定中的事件、基类或接口协定建议其他名称，则可以忽略此约定。 例如，你不应重命名常用事件处理程序，例如 OnButtonClick。