**[说说C#的async和await](http://blog.csdn.net/tianmuxia/article/details/17675681)**

// call1 和 taskcall 都是异步执行

static void Main(string[] args)

{

Task.Factory.StartNew(call1, 399);

Console.WriteLine("Thread is starting...");

taskcall();

Console.WriteLine("Console is end!");

Console.ReadLine();

}

static async void taskcall()

{

int res = await show(100);

Console.WriteLine("good:{0}", res);

}

static async Task<int> show(int a)

{

for (int i = 0; i < 10; i++)

{

await Task.Delay(500);

Console.WriteLine("Show: {0} - {1}", i, a);

}

return a + 100;

}

static void call1(object ts)

{

for (int i = 0; i < 4; i++)

{

Thread.Sleep(1000);

Console.WriteLine("Call1: {0} - {1} - {2}", i, DateTime.Now, ts);

}

}

// taskcall 是同步执行

static async void taskcall()

{

Task<int> res = show(100); 可这里是同步执行。

Console.WriteLine("good:{0}", res.Result);

}

static async Task<int> show(int a)

{

for (int i = 0; i < 10; i++)

{

await Task.Delay(100); - 决定了是异步执行

Console.WriteLine("Show: {0} - {1}", i, a);

}

return a + 100;

}

//异步执行

static async void taskcall()

{

int res = await show(100); - 这才会异步执行

Console.WriteLine("good:{0}", res);

}

static void Main(string[] args)

{

Task.Factory.StartNew(call1, 399);

Console.WriteLine("Thread is starting...");

int res = await show(100); -- 出错

Console.WriteLine("good:{0}", res);

Console.WriteLine("Console is end!");

Console.ReadLine();

}

static async void taskcall()

{

}

static async Task<int> show(int a)

{

for (int i = 0; i < 10; i++)

{

await Task.Delay(500);

Console.WriteLine("Show: {0} - {1}", i, a);

}

return a + 100;

}

// 出错： await 必须是在 async 方法里使用

static void Main(string[] args)

{

Task.Factory.StartNew(call1, 399);

Console.WriteLine("Thread is starting...");

//taskcall();

int res = await show(100); -- 如此使用会出现编译错误

Console.WriteLine("good:{0}", res);

Console.WriteLine("Console is end!");

Console.ReadLine();

}

async 只能是 void, Task, Task<T> 三种方法

// HttpClient 这还是同步执行。

public string uploadBBStream(Stream stream)

{

string return\_string = string.Empty;

BBMessage bbMessage = BBStream.DecodeStream(stream);

/\* place your code here \*/

string res = callWCF("kdfkd").Result; 绝对的同步执行

/\* end of your code \*/

return return\_string;

}

public async Task<string> callWCF(string url)

{

HttpClient hp = new HttpClient();

HttpResponseMessage rep = await hp.PostAsync("dkdk", null);

string ret\_str = await rep.Content.ReadAsStringAsync();

return ret\_str;

}

看个例子：

**[csharp]** [view plaincopy](http://blog.csdn.net/tianmuxia/article/details/17675681)[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public class MyClass
2. {
3. public MyClass()
4. {
5. DisplayValue(); //这里不会阻塞
6. System.Diagnostics.Debug.WriteLine("MyClass() End.");
7. }
8. public Task<double> GetValueAsync(double num1, double num2)
9. {
10. return Task.Run(() =>
11. {
12. for (int i = 0; i < 1000000; i++)
13. {
14. num1 = num1 / num2;
15. }
16. return num1;
17. });
18. }
19. public async void DisplayValue()
20. {
21. double result = await GetValueAsync(1234.5, 1.01);//此处会开新线程处理GetValueAsync任务，然后方法马上返回
22. //这之后的所有代码都会被封装成委托，在GetValueAsync任务完成时调用
23. System.Diagnostics.Debug.WriteLine("Value is : " + result);
24. }
25. }

上面在MyClass的构造函数里调用了async关键字标记的异步方法DisplayValue()，DisplayValue()方法里执行了 一个await关键字标记的异步任务GetValueAsync()，这个异步任务必须是以Task或者Task<TResult>作为返回 值的，而我们也看到，异步任务执行完成时实际返回的类型是void或者TResult，DisplayValue()方法里 await GetValueAsync()之后的所有代码都会在异步任务完成时才会执行。

DisplayValue()方法实际执行的代码如下：

**[csharp]** [view plaincopy](http://blog.csdn.net/tianmuxia/article/details/17675681)[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public void DisplayValue()
2. {
3. System.Runtime.CompilerServices.TaskAwaiter<double> awaiter = GetValueAsync(1234.5, 1.01).GetAwaiter();
4. awaiter.OnCompleted(() =>
5. {
6. double result = awaiter.GetResult();
7. System.Diagnostics.Debug.WriteLine("Value is : " + result);
8. });
9. }

可以看到，async和await关键字只是把上面的代码变得更简单易懂而已。

程序的输出如下：

MyClass() End.

Value is : 2.47032822920623E-322

以下是我写的一个静态类，可以方便将一个普通Function执行异步调用：

**[csharp]** [view plaincopy](http://blog.csdn.net/tianmuxia/article/details/17675681)[在CODE上查看代码片](https://code.csdn.net/snippets/134105)

1. public static class TaskAsyncHelper
2. {
3. /// <summary>
4. /// 将一个方法function异步运行，在执行完毕时执行回调callback
5. /// </summary>
6. /// <param name="function">异步方法，该方法没有参数，返回类型必须是void</param>
7. /// <param name="callback">异步方法执行完毕时执行的回调方法，该方法没有参数，返回类型必须是void</param>
8. public static async void RunAsync(Action function, Action callback)
9. {
10. Func<System.Threading.Tasks.Task> taskFunc = () =>
11. {
12. return System.Threading.Tasks.Task.Run(() =>
13. {
14. function();
15. });
16. };
17. await taskFunc();
18. if (callback != null)
19. callback();
20. }
22. /// <summary>
23. /// 将一个方法function异步运行，在执行完毕时执行回调callback
24. /// </summary>
25. /// <typeparam name="TResult">异步方法的返回类型</typeparam>
26. /// <param name="function">异步方法，该方法没有参数，返回类型必须是TResult</param>
27. /// <param name="callback">异步方法执行完毕时执行的回调方法，该方法参数为TResult，返回类型必须是void</param>
28. public static async void RunAsync<TResult>(Func<TResult> function, Action<TResult> callback)
29. {
30. Func<System.Threading.Tasks.Task<TResult>> taskFunc = ()=>
31. {
32. return System.Threading.Tasks.Task.Run(()=>
33. {
34. return function();
35. });
36. };
37. TResult rlt = await taskFunc();
38. if(callback != null)
39. callback(rlt);
40. }
41. }

使用很简单，将方法名作为参数传进去就行了，最常用的是把很耗时的序列化函数传进去，以免阻塞UI进程，造成卡顿现象，影响用户体验。

**C#中的异步陷阱**

　　有时候，理解一种语言中的缺陷的最好方式是查看另一种语言如何防止这些缺陷发生。《Real-World Functional Programming》(注：该书已由清华大学出版社引进，中文名《C#与F#编程实践》)的作者Tomas Petricek讨论了异步C#代码中常见的7项错误，并说明F#如何降低这些缺陷出现的可能性。

　　以下是Tomas的文章《在C#和F#中使用异步：C#中的异步陷阱》所覆盖内容的简要介绍，而我们更鼓励读者阅读整篇文章。

　　Async没有异步运行：只有在第一个await语句之后出现的代码才会异步运行。

　　忽略结果：忘记await一个函数返回的任务，将会导致乱序执行。

　　Async void方法：不能await一个返回“async void”而不是“async Task”的异步函数，这会导致与上一条相同的问题。

　　Async void lambda函数：发生在当某个函数接收一个Action委托而不是Func<…，Task>委托的时候。此外也无法await一个async函数。

　 　嵌套任务：在语句“await Task.Factory.StartNew(async () => { await Task.Delay(1000); });”中，第一个和第二个await语句互相之间完全无关。这意味着第一个await将在第二个await之前完成，而语句中关联的1000ms延迟将 被忽略。

　　未按异步方式运行：使用Task.Wait()会将整个调用栈强制转为同步模式。

　　对那些不熟悉F#的开发者来说，在了解到F#中的async工作流并不是基于Task和Task类型的时候或许会感到惊讶。相反，F#使用自己的类型——Async。

[http://www.cr173.com/html/17820\_1.html](http://www.cr173.com/html/17820_1.html%20)

# .net 4.5如何使用Async和Await进行异步编程

## 出处：[西西整理](http://www.cr173.com) 作者：西西 日期：2012/10/29 14:00:10 [[大](javascript:doZoom(14)) [中](javascript:doZoom(12)) [小](javascript:doZoom(9))] 评论: 0 | [我要发表看法](http://www.cr173.com/html/17820_1.html#cms_show_ly)

通过使用异步编程，可避免出现性能瓶颈，并提高应用程序的整体响应。然而，技术编写异步应用程序的传统方法过于复杂，这使得异步程序难以编写，调试和维护。

[Visual Studio](http://www.cr173.com/k/vc/)2012引入了一个简单的开发方法，异步编程，我们可以充分利用.[net framework](http://www.cr173.com/k/netframework/) 4.5 和 Windows Runtime中对异步的支持。这项复杂的工作将会交由编译器来搞定，开发人员就像是在使用同步代码来编写应用程序的逻辑结构，但其结果是，得到了所有异 步编程的优点，但只要付出一点点工作。

本主题简要介绍何时以及如何使用异步编程，其中包括支持说明本主题的例子。由此下载VS2012.

异步提高响应能力

异 步性必不可少，因为现实中有很多潜在的，可阻塞应用程序响应的情况，如当你的应用程序访问网络，文件系统等等。比如访问Web资源，有时是缓慢的 或者是有延迟的。同步处理的话，如果有这样的一个阻塞产生，那么整个应用程序就必须等待。而在一个异步的过程中，应用程序可以先继续进行其他不依赖于网络 的工作，直到所有可能产生阻塞的任务完成后再处理这些任务。

下表显示了典型的可以通过异步编程提高响应的场景。列出的[.net framework](http://www.cr173.com/k/net/)4.5和Windows Runtime的API包含支持异步编程的方法。

|  |  |
| --- | --- |
| 应用领域 | 支持异步方法的API |
| 访问Web | HttpClient , SyndicationClient |
| 使用文件 | StorageFile, StreamWriter, StreamReader,XmlReader |
| 使用图像 | MediaCapture, BitmapEncoder, BitmapDecoder |
| WCF程序开发 | Synchronous and Asynchronous Operations |
| 使用sockets | Socket |

异步性已经被证明对所有通过一个线程访问UI，或是处理UI相关的活动的应用都特别的有价值。如果在同步的应用程序中任何一个处理过程被阻塞，那就意味着所有的东西都被阻塞了。你的应用程将会停止响应，更糟糕的是你可能会得出这样的结论，这只是等待并不是失败。

当你使用异步方法，应用程序将继续响应的UI。您可以调整大小或最小化窗口，例如，当你不想等下去的时候，你可以关闭该应用程序。

现在这种基于异步的方法在你设计异步操作时，就像一组可以选择的自动变速器，也就是说，你可以得到所有之前异步编程的好处，而不必像之前那样苦逼（太让人兴奋了）。

异步方法现在很容易编写

Async 和Await关键字是C#异步编程的核心。通过使用这两个关键字，你可以使用.NET Framework或Windows Runtime的资源创建一个异步方法如同你创建一个同步的方法一样容易。通过使用async和await定义的异步方法，这里被称为异步方法。

下面的例子显示了一个异步方法。代码中的几乎所有的东西你看起来应该非常熟悉。注释中描述了你为实现异步操作添加什么功能。

 1 // 在签名中三个要注意的事项:   
 2 //  - 该方法具有一个async修饰符.    
 3 //  - 返回类型为 Task or Task<t>. (参考 "返回类型" 一节.)  
 4 //    这里, 返回值是 Task<int> 因为返回的是一个整数类型.   
 5 //  - 这个方法的名称以 "Async" 结尾.  
 6 async Task<int> AccessTheWebAsync()  
 7 {   
 8     // 你需要添加System.Net.Http的引用来声明client  
 9     HttpClient client = new HttpClient();  
10   
11     // GetStringAsync 返回 Task<string>. 这意味着当Task结束等待之后   
12     // 你将得到一个string (urlContents).  
13     Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");  
14   
15     // 你可以做一些不依赖于 GetStringAsync 返回值的操作.  
16     DoIndependentWork();  
17   
18     // await 操作挂起了当前方法AccessTheWebAsync.   
19     //  - AccessTheWebAsync 直到getStringTask完成后才会继续.   
20     //  - 同时, 控制权将返回 AccessTheWebAsync 的调用者.   
21     //  - 控制权会在getStringTask完成后归还到AccessTheAsync.    
22     //  - await操作将取回getStringTask中返回的string结果.   
23     string urlContents = await getStringTask;  
24   
25     // return语句用来指定一个整数结果。  
26     // 调用AccessTheWebAsync将会收到一个返回值的长度.   
27     return urlContents.Length;  
28 }

如果AccessTheWebAsync没有什么不依赖于GetStringAsync的内容，也可以直接调用如下代码：

string urlContents = await client.GetStringAsync();

以下几个特点总结了一下前面的例子中的异步方法。

方法中包含了 **async** 修饰符。

一个async方法按照惯例以“Async”结尾。

返回类型是如下类型之一：

Task<TResult> 当你的方法有返回值时，TResult即返回值的类型

Task 当你的方法没有return语句，或者返回值并不参与任何形式的运算（包括赋值操作）。

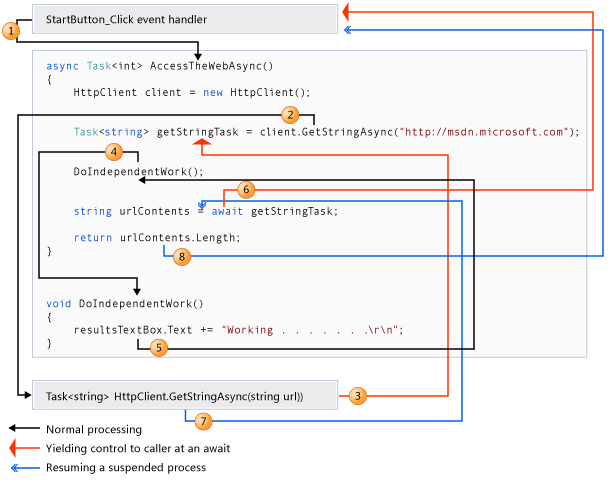
Void 当你编写一个异步事件处理时会用到

方法通常包括至少一个await的表达式，这意味着该方法在遇到await时不能继续执行，直到等待异步操作完成。在此期间，该方法将被暂停，并且控制权返回到该方法的调用者。本主题接下来的部分说明了悬挂点后会发生什么。

在 异步方法中，我们使用这些已经提供的关键字和类型来表达想要做什么的时候，编译器并没有闲着，他将处理包括跟踪在暂停方法中控制权返回到 await点后将会如何处理。一些常规流程，如循环和异常处理，在之前的异步代码中都比较难以处理。而现在都归结到了一个async方法中，你会感觉你在 写一个同步的代码，之前的那些困惑已经不复存在了。

在异步方法中发生了什么

了解异步编程最重要的是理解控制权是如何在方法之间转移的。下面的图标将会解释这个过程。



图中的数字对应于下面的步骤。

1. 一个事件的处理函数用await的方式调用了异步方法 AccessTheWebAsync。

2. AccessTheWebAsync 创建了一个 HttpClient 实例，并调用了异步方法 GetStringAsync 来以下载一个页面并将内容以string的形式返回.

3. 在GetStringAsync中将会碰到一些让进程挂起的事情。也许它必须等待一个网站下载完成或其他一些阻塞性的动作，为了避免阻塞资源，GetStringAsync把控制权移交给了它的调用者AccessTheWEbAsync。

GetStringAsync 返回Task<TResult>泛型，例子中的TResult是string类 型，AccessTheWebAsync将任务交给了getStringTask变量。这个任务代表了一个正在调用GetStringAsync的过程， 与一个承诺，当工作完成时，最终将产生string返回值。

4. 由于getStringTask并没有被（用await）等待着索取结果，so AccessTheWebAsync 可以继续其他不依赖于GetStringAsync最终返回结果的其他任务。这些任务由一个同步方法DoIndenpendentWork代表。

5.DoIndenpendentWork是一个同步方法，将会以同步的方式执行他的工作，并将返回值返回给调用者。

6. AccessTheWebAsync下一步是希望计算已经下载下来的字符长度，但是如果没有这个字符，这个希望也就破灭了。

因 此，AccessTheWebAsync 使用了await关键字挂起了自己的线程，并将控制权移交到了AccessTheWebAsync的调用者。AccessTheWebAsync将会返回 一个Task<int>给调用函数。这个任务承诺会在结束是返回给调用者一个int型的返回值。

**注意**

如 果GetStringAsync在AccessTheWebAsync 调用await之前就已经完成了，那么控制权将依然在AccessTheWebAsync中。这种挂起和等待的操作也是很消耗资源的，如果返回值在 await之前就已经得到了，AccessTheWebAsync没必要非得在用等待的方法去得到最终的结果。

在调用方法（这里是一个 event的处理函数）的内部，处理过程是叠加的，event的处理函数将会等待AccessTheWebAsync，而 AccessTheWebAsync在等待GetStringAsync，与此同时，调用函数依然可以执行不依赖于这些返回值的操作。

7. GetStringAsync计算并产生一个string结果。这个string结果可能不是按照你现在期望的方式直接返回给他的调用函数，相反，这个结 果保存在一个代表方法完成的任务中，getStringTask。await操作符将会从getStringTask中取回期望的结果。赋值语句将会把结 果交给urlContents。

8. 当AccessTheWebAsync获得了这个字符串结果，我们可以继续计算这个字符串的长度了。这样AccessTheWebAsync的工作也完成了，等待中的event处理函数也可以继续了。

如 果你刚刚接触异步编程，那应该花一分钟来思考一下同步行为和异步行为的不同。同步方法在工作完成之后返回（步骤5），但是异步方法返回一个 task值在他工作被暂停的时候（步骤3，6）.当异步方法完成了他的工作之后，task被标记为complete，工作的结果也保存在task之中。

异步的API方法

你 可能会想知道在哪里可以找到，如GetStringAsync，支持异步编程的API方法。.NET Framework 4.5包含许多使用await和async工作的成员方法。识别这些方法很简单，方法名都是以”Async”结尾的并且返回类型都是Task或者 Task<TResult>。例如，System.IO.Stream类包含的方法，如CopyToAsync，ReadAsync，及 WriteAsync的对应的同步方法是CopyTo，Read和Write。

线程

异步方法的目的是不阻塞操作。在async方法中, await任务在执行的过程中，并不会阻塞当前的线程，其余的方法可以继续执行，控制权将会移交到async方法的调用者。

async和await关键字并不会创建额外的线程，async方法不会去请求多线程操作。真正创建线程的操作是由Task.Run()实现的，一个async方法并不是在他自己的线程上执行的，只有当方法被激活时，才会使用当前线程的上下文和处理时间。

async方法要比BackgroundWorker更实用，而且使用起来更简单而且不用去过多的考虑竞态冲突神马的。async方法会将运行中的代码依据某些算法进行合理的拆分，并传递给线程池，这也是BackgroundWorker不能比的。

Async和Await

如果需要使用async或者await指定一个异步方法，我们需要注意一下两点：

用async标记的异步方应该使用await关键子来制定挂起点。await操作符会告诉编译器，这个async方放在完成之前，后面的代码无法继续执行，同时，控制权转移到async方法的调用者。

标记为async的方法，调用时应使用await。

一 个async方法里通常包含一个或多个的对应的await操作符，但如果没有await表达式也不会导致编译错误。但如果调用一个async方 法，却不使用await关键字来标记一个挂起点的话，程序将会忽略async关键字并以同步的方式执行。编译器会对类似的问题发出警告。

async和await都是上下文关键字：更多的细节可以参考：

Async (Visual Basic)

async (C# Reference)

Await Operator (Visual Basic)

await (C# Reference)

返回类型和参数

在.NET Framework编程中，一个async方法通常返回的类型是Task或者Task<TResult>。在异步方法中，await操作符作用于从另外一个异步方法返回的Task。

如果指定Task<TResult>为返回结果，那么这个方法必须包含return指定的TResult结果的语句。

如果使用Task作为返回值，那么这个方法应该不存在使用return语句返回结果的代码，或者返回的结果不参与任何运算（包括赋值操作）。

 1 // 明确指定 Task<tresult>  
 2 async Task<int> TaskOfTResult\_MethodAsync()  
 3 {  
 4     int hours;  
 5     // . . .  
 6     // return一个整数作为结果.  
 7     return hours;  
 8 }  
 9   
10 // 调用 TaskOfTResult\_MethodAsync  
11 Task<int> returnedTaskTResult = TaskOfTResult\_MethodAsync();  
12 int intResult = await returnedTaskTResult;  
13 // 或者使用一条语句  
14 int intResult = await TaskOfTResult\_MethodAsync();  
15   
16   
17 // 明确指定 Task  
18 async Task Task\_MethodAsync()  
19 {  
20     // . . .  
21     // 方法没有任何return语句.    
22 }  
23   
24 // 调用 Task\_MethodAsync  
25 Task returnedTask = Task\_MethodAsync();  
26 await returnedTask;  
27 // 或者使用一条语句  
28 await Task\_MethodAsync();

每一个返回的task都代表一个正在执行的工作，task包装的信息中包含了这个异步方法的执行时的状态，最终的结果，或者处理过程中抛出的异常。

如果返回值为void，这种类型主要使用于定义事件处理。异步事件通常被认为是一系列异步操作的开始。使用void返回类型不需要await，而且调用void异步方法的函数不会捕获方法抛出的异常。

另外，async方法不能使用ref或者out参数，但是可以调用含有这些参数的方法。

命名约定

按照约定，你应该在异步方法的名称后面追加“Async”用以标记此方法。但是在event，基类和接口中不需要遵守约定，就像本文例子中event处理函数Button1\_Click一样。

相关主题

一个完整的例子

 1 using System;  
 2 using System.Collections.Generic;  
 3 using System.Linq;  
 4 using System.Text;  
 5 using System.Threading.Tasks;  
 6 using System.Windows;  
 7 using System.Windows.Controls;  
 8 using System.Windows.Data;  
 9 using System.Windows.Documents;  
10 using System.Windows.Input;  
11 using System.Windows.Media;  
12 using System.Windows.Media.Imaging;  
13 using System.Windows.Navigation;  
14 using System.Windows.Shapes;  
15   
16    
17 using System.Net.Http;  
18   
19 namespace AsyncFirstExample  
20 {  
21     public partial class MainWindow : Window  
22     {  
23         // 将event处理函数用async标记，这样就可以在处理函数中使用await实现异步操作.   
24         private async void StartButton\_Click(object sender, RoutedEventArgs e)  
25         {  
26             // 调用和await分离的方式.   
27             //Task<int> getLengthTask = AccessTheWebAsync();   
28             //// 在这里做一些其他的工作.   
29             //int contentLength = await getLengthTask;   
30   
31             int contentLength = await AccessTheWebAsync();  
32   
33             resultsTextBox.Text +=  
34                 String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);  
35         }  
36   
37   
38            // 在签名中三个要注意的事项:   
39     //  - 该方法具有一个async修饰符.    
40     //  - 返回类型为 Task or Task<t>. (参考 "返回类型" 一节.)  
41     //    这里, 返回值是 Task<int> 因为返回的是一个整数类型.   
42     //  - 这个方法的名称以 "Async" 结尾.  
43     async Task<int> AccessTheWebAsync()  
44     {   
45             // 你需要添加System.Net.Http的引用来声明client  
46             HttpClient client = new HttpClient();  
47   
48             // GetStringAsync 返回 Task<string>. 这意味着当Task结束等待之后   
49             // 你将得到一个string (urlContents).  
50             Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");  
51   
52             // 你可以做一些不依赖于 GetStringAsync 返回值的操作.  
53             DoIndependentWork();  
54   
55             // await 操作挂起了当前方法AccessTheWebAsync.   
56             //  - AccessTheWebAsync 直到getStringTask完成后才会继续.   
57             //  - 同时, 控制权将返回 AccessTheWebAsync 的调用者.   
58             //  - 控制权会在getStringTask完成后归还到AccessTheAsync.    
59             //  - await操作将取回getStringTask中返回的string结果.   
60             string urlContents = await getStringTask;  
61   
62             // return语句用来指定一个整数结果。  
63             // 调用AccessTheWebAsync将会收到一个返回值的长度.   
64             return urlContents.Length;  
65     }  
66   
67   
68         void DoIndependentWork()  
69         {  
70             resultsTextBox.Text += "Working . . . . . . .\r\n";  
71         }  
72     }  
73 }  
74   
75 // 运行结果:   
76   
77 // Working . . . . . . .   
78   
79 // Length of the downloaded string: 41564.