**走进异步编程的世界 - 开始接触 async/await**

**序**

　　这是学习异步编程的入门篇。

　　涉及 C# 5.0 引入的 async/await，但在控制台输出示例时经常会采用 C# 6.0 的 $"" 来拼接字符串，相当于string.Format() 方法。

**目录**

* [What's 异步？](http://www.cnblogs.com/liqingwen/p/5831951.html#link1)
* [async/await 结构](http://www.cnblogs.com/liqingwen/p/5831951.html#link2)
* [What’s 异步方法？](http://www.cnblogs.com/liqingwen/p/5831951.html#link3)

**一、What's 异步？**

     启动程序时，系统会在内存中创建一个新的进程。进程是构成运行程序资源的集合。

     在进程内部，有称为线程的内核对象，它代表的是真正的执行程序。系统会在 Main 方法的第一行语句就开始线程的执行。

     线程：

     ①默认情况，一个进程只包含一个线程，从程序的开始到执行结束；

     ②线程可以派生自其它线程，所以一个进程可以包含不同状态的多个线程，来执行程序的不同部分；

     ③一个进程中的多个线程，将共享该进程的资源；

     ④系统为处理器执行所规划的单元是线程，而非进程。

     一般来说我们写的控制台程序都只使用了一个线程，从第一条语句按顺序执行到最后一条。但在很多的情况下，这种简单的模型会在性能或用户体验上不好。

     例如：服务器要同时处理来自多个客户端程序的请求，又要等待数据库和其它设备的响应，这将严重影响性能。程序不应该将时间浪费在响应上，而要在等待的同时执行其它任务！

     现在我们开始进入异步编程。在异步程序中，代码不需要按照编写时的顺序执行。这时我们需要用到 C# 5.0 引入的 async/await 来构建异步方法。

     我们先看一下不用异步的示例：

[复制代码](javascript:void(0);)

1 class Program

2 {

3 //创建计时器

4 private static readonly Stopwatch Watch = new Stopwatch();

5

6 private static void Main(string[] args)

7 {

8 //启动计时器

9 Watch.Start();

10

11 const string url1 = "http://www.cnblogs.com/";

12 const string url2 = "http://www.cnblogs.com/liqingwen/";

13

14 //两次调用 CountCharacters 方法（下载某网站内容，并统计字符的个数）

15 var result1 = CountCharacters(1, url1);

16 var result2 = CountCharacters(2, url2);

17

18 //三次调用 ExtraOperation 方法（主要是通过拼接字符串达到耗时操作）

19 for (var i = 0; i < 3; i++)

20 {

21 ExtraOperation(i + 1);

22 }

23

24 //控制台输出

25 Console.WriteLine($"{url1} 的字符个数：{result1}");

26 Console.WriteLine($"{url2} 的字符个数：{result2}");

27

28 Console.Read();

29 }

30

31 /// <summary>

32 /// 统计字符个数

33 /// </summary>

34 /// <param name="id"></param>

35 /// <param name="address"></param>

36 /// <returns></returns>

37 private static int CountCharacters(int id, string address)

38 {

39 var wc = new WebClient();

40 Console.WriteLine($"开始调用 id = {id}：{Watch.ElapsedMilliseconds} ms");

41

42 var result = wc.DownloadString(address);

43 Console.WriteLine($"调用完成 id = {id}：{Watch.ElapsedMilliseconds} ms");

44

45 return result.Length;

46 }

47

48 /// <summary>

49 /// 额外操作

50 /// </summary>

51 /// <param name="id"></param>

52 private static void ExtraOperation(int id)

53 {

54 //这里是通过拼接字符串进行一些相对耗时的操作

55 var s = "";

56

57 for (var i = 0; i < 6000; i++)

58 {

59 s += i;

60 }

61

62 Console.WriteLine($"id = {id} 的 ExtraOperation 方法完成：{Watch.ElapsedMilliseconds} ms");

63 }

64 }

[复制代码](javascript:void(0);)



     图1-1 运行的效果图，以毫秒（ms）为单位

　　【备注】一般来说，直接拼接字符串是一种比较耗性能的手段，如果对字符串拼接有性能要求的话应该使用 StringBuilder。

　　【注意】每次运行的结果可能不同。不管哪次调试，绝大部分时间都浪费前两次调用（CountCharacters 方法），即在等待网站的响应上。

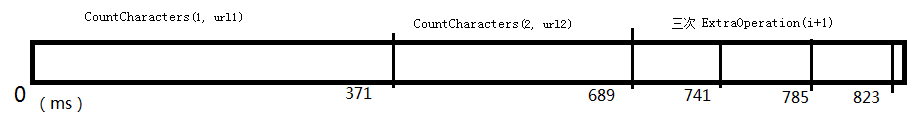


　　图1-2 根据执行结果所画的时间轴

     有人曾幻想着这样提高性能的方法：在调用 A 方法时，不等它执行完，直接执行 B 方法，然后等 A 方法执行完成再处理。

     C# 的 async/await 就可以允许我们这么弄。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

1 class Program

2 {

3 //创建计时器

4 private static readonly Stopwatch Watch = new Stopwatch();

5

6 private static void Main(string[] args)

7 {

8 //启动计时器

9 Watch.Start();

10

11 const string url1 = "http://www.cnblogs.com/";

12 const string url2 = "http://www.cnblogs.com/liqingwen/";

13

14 //两次调用 CountCharactersAsync 方法（异步下载某网站内容，并统计字符的个数）

15 Task<int> t1 = CountCharactersAsync(1, url1);

16 Task<int> t2 = CountCharactersAsync(2, url2);

17

18 //三次调用 ExtraOperation 方法（主要是通过拼接字符串达到耗时操作）

19 for (var i = 0; i < 3; i++)

20 {

21 ExtraOperation(i + 1);

22 }

23

24 //控制台输出

25 Console.WriteLine($"{url1} 的字符个数：{t1.Result}");

26 Console.WriteLine($"{url2} 的字符个数：{t2.Result}");

27

28 Console.Read();

29 }

30

31 /// <summary>

32 /// 统计字符个数

33 /// </summary>

34 /// <param name="id"></param>

35 /// <param name="address"></param>

36 /// <returns></returns>

37 private static async Task<int> CountCharactersAsync(int id, string address)

38 {

39 var wc = new WebClient();

40 Console.WriteLine($"开始调用 id = {id}：{Watch.ElapsedMilliseconds} ms");

41

42 var result = await wc.DownloadStringTaskAsync(address);

43 Console.WriteLine($"调用完成 id = {id}：{Watch.ElapsedMilliseconds} ms");

44

45 return result.Length;

46 }

47

48 /// <summary>

49 /// 额外操作

50 /// </summary>

51 /// <param name="id"></param>

52 private static void ExtraOperation(int id)

53 {

54 //这里是通过拼接字符串进行一些相对耗时的操作

55 var s = "";

56

57 for (var i = 0; i < 6000; i++)

58 {

59 s += i;

60 }

61

62 Console.WriteLine($"id = {id} 的 ExtraOperation 方法完成：{Watch.ElapsedMilliseconds} ms");

63 }

64 }

这是修改后的代码

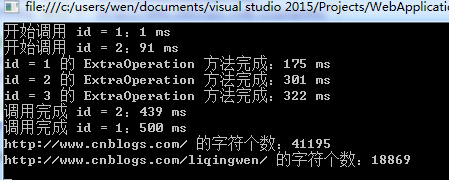


 图1-3 修改后的执行结果图

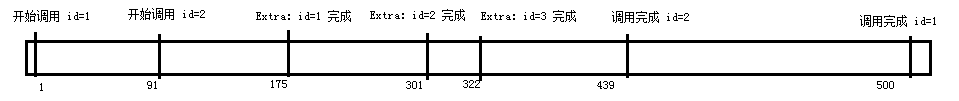


图1-4 根据加入异步后的执行结果画的时间轴。

　　我们观察时间轴发现，新版代码比旧版快了不少（由于网络波动的原因，很可能会出现耗时比之前长的情况）。这是由于 ExtraOperation 方法的数次调用是在 CountCharactersAsync 方法调用时等待响应的过程中进行的。所有的工作都是在主线程中完成的，没有创建新的线程。

　　【改动分析】只改了几个细节的地方，直接展开代码的话可能看不出来，改动如下：

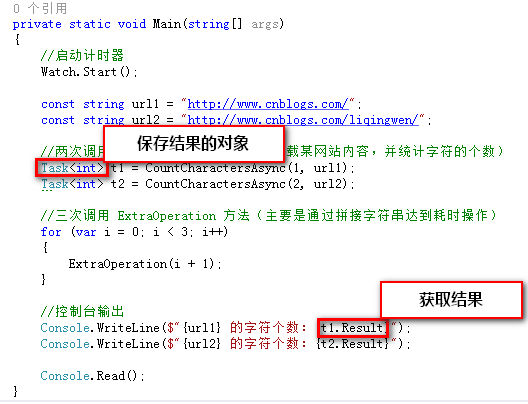


 图1-5

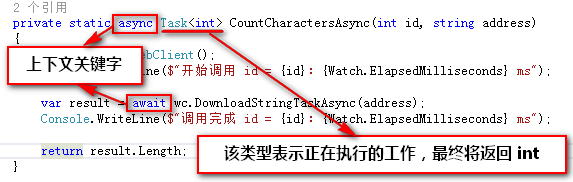


　　图1-6

　　①从 Main 方法执行到 CountCharactersAsync(1, url1) 方法时，该方法会立即返回，然后才会调用它内部的方法开始下载内容。该方法返回的是一个 Task<int> 类型的占位符对象，表示计划进行的工作。这个占位符最终会返回 int 类型的值。

　　②这样就可以不必等 CountCharactersAsync(1, url1) 方法执行完成就可以继续进行下一步操作。到执行 CountCharactersAsync(2, url2)  方法时，跟 ① 一样返回 Task<int> 对象。

　　③然后，Main 方法继续执行三次 ExtraOperation 方法，同时两次 CountCharactersAsync 方法依然在持续工作 。

　　④t1.Result 和 t2.Result 是指从 CountCharactersAsync 方法调用的 Task<int> 对象取结果，如果还没有结果的话，将阻塞，直有结果返回为止。

**二、async/await 结构**

     先解析一下专业名词：

     同步方法：一个程序调用某个方法，等到其执行完成之后才进行下一步操作。这也是默认的形式。

     异步方法：一个程序调用某个方法，在处理完成之前就返回该方法。通过 async/await 我们就可以实现这种类型的方法。

     async/await 结构可分成三部分：

     （1）调用方法：该方法调用异步方法，然后在异步方法执行其任务的时候继续执行；

     （2）异步方法：该方法异步执行工作，然后立刻返回到调用方法；

     （3）await 表达式：用于异步方法内部，指出需要异步执行的任务。一个异步方法可以包含多个 await 表达式（不存在 await 表达式的话 IDE 会发出警告）。

　　现在我们来分析一下示例。



　　图2-1

**三、What’s 异步方法**

     异步方法：在执行完成前立即返回调用方法，在调用方法继续执行的过程中完成任务。

     语法分析：

     （1）关键字：方法头使用 async 修饰。

     （2）要求：包含 N（N>0） 个 await 表达式（不存在 await 表达式的话 IDE 会发出警告），表示需要异步执行的任务。

     （3）返回类型：只能返回 3 种类型（void、Task 和 Task<T>）。Task 和 Task<T> 标识返回的对象会在将来完成工作，表示调用方法和异步方法可以继续执行。

     （4）参数：数量不限，但不能使用 out 和 ref 关键字。

     （5）命名约定：方法后缀名应以 Async 结尾。

     （6）其它：匿名方法和 Lambda 表达式也可以作为异步对象；async 是一个上下文关键字；关键字 async 必须在返回类型前。

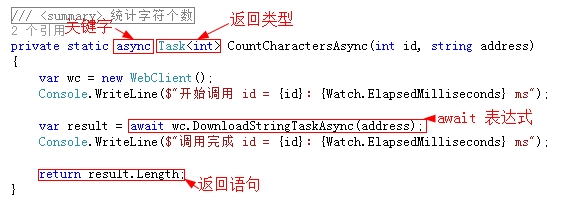


图3-1 异步方法的简单结构图

**小结**

　　1.解析了进程和线程的概念

　　2.异步的简单用法

　　3.async/await 结构体

　　4.异步方法语法结构

**走进异步编程的世界 - 剖析异步方法（上）**

**序**

　　这是上篇《[走进异步编程的世界 - 开始接触 async/await 异步编程](http://www.cnblogs.com/liqingwen/p/5831951.html)》（入门）的第二章内容，主要是与大家共同深入探讨下异步方法。

　　本文要求了解委托的使用。

**目录**

* [介绍异步方法](http://www.cnblogs.com/liqingwen/p/5844095.html#link0)
* [控制流](http://www.cnblogs.com/liqingwen/p/5844095.html#link1)
* [await 表达式](http://www.cnblogs.com/liqingwen/p/5844095.html#link2)
* [How 取消异步操作](http://www.cnblogs.com/liqingwen/p/5844095.html#link3)

**介绍异步方法**

     异步方法：在执行完成前立即返回调用方法，在调用方法继续执行的过程中完成任务。

     语法分析：

     （1）关键字：方法头使用 async 修饰。

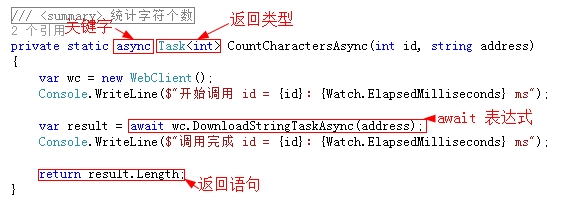
     （2）要求：包含 N（N>0） 个 await 表达式（不存在 await 表达式的话 IDE 会发出警告），表示需要异步执行的任务。【备注】感谢 [czcz1024](http://www.cnblogs.com/czcz1024/) 的修正与补充：没有的话，就和普通方法一样执行了。

     （3）返回类型：只能返回 3 种类型（void、Task 和 Task<T>）。Task 和 Task<T> 标识返回的对象会在将来完成工作，表示调用方法和异步方法可以继续执行。

     （4）参数：数量不限。但不能使用 out 和 ref 关键字。

     （5）命名约定：方法后缀名应以 Async 结尾。

     （6）其它：匿名方法和 Lambda 表达式也可以作为异步对象；async 是一个上下文关键字；关键字 async 必须在返回类型前。

图1 异步方法的简单结构图

　　关于 async 关键字：

　　①在返回类型之前包含 async 关键字

　　②它只是标识该方法包含一个或多个 await 表达式，即，它本身不创建异步操作。

　　③它是上下文关键字，即可作为变量名。

　　现在先来简单分析一下这三种返回值类型：void、Task 和 Task<T>

　　（1）Task<T>：调用方法要从调用中获取一个 T 类型的值，异步方法的返回类型就必须是Task<T>。调用方法从 Task 的 Result 属性获取的就是 T 类型的值。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 private static void Main(string[] args)

2 {

3 Task<int> t = Calculator.AddAsync(1, 2);

4

5 //一直在干活

6

7 Console.WriteLine($"result: {t.Result}");

8

9 Console.Read();

10 }

[复制代码](javascript:void(0);)

Program.cs

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Calculator

2 {

3 private static int Add(int n, int m)

4 {

5 return n + m;

6 }

7

8 public static async Task<int> AddAsync(int n, int m)

9 {

10 int val = await Task.Run(() => Add(n, m));

11

12 return val;

13 }

14 }

[复制代码](javascript:void(0);)

View Code

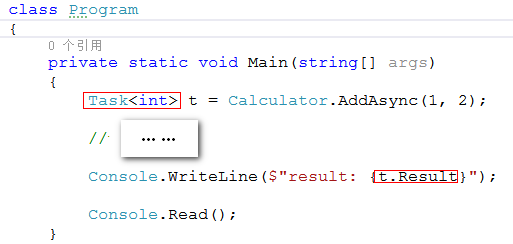


图2

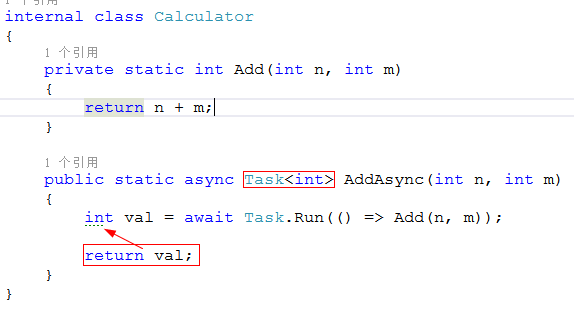


图3

　　（2）Task：调用方法不需要从异步方法中取返回值，但是希望检查异步方法的状态，那么可以选择可以返回 Task 类型的对象。不过，就算异步方法中包含 return 语句，也不会返回任何东西。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 private static void Main(string[] args)

2 {

3 Task t = Calculator.AddAsync(1, 2);

4

5 //一直在干活

6

7 t.Wait();

8 Console.WriteLine("AddAsync 方法执行完成");

9

10 Console.Read();

11 }

[复制代码](javascript:void(0);)

Program.cs

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Calculator

2 {

3 private static int Add(int n, int m)

4 {

5 return n + m;

6 }

7

8 public static async Task AddAsync(int n, int m)

9 {

10 int val = await Task.Run(() => Add(n, m));

11 Console.WriteLine($"Result: {val}");

12 }

13 }

[复制代码](javascript:void(0);)

View Code

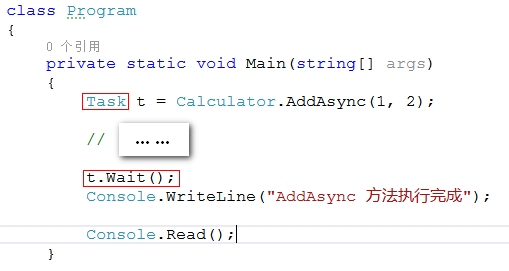


图4

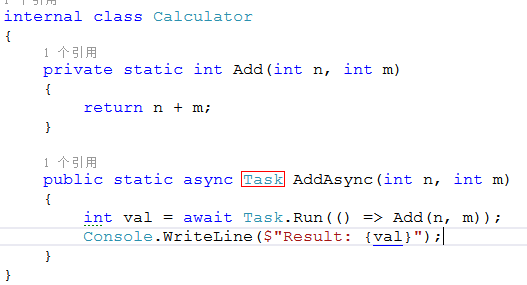


图5

　　（3）void：调用方法执行异步方法，但又不需要做进一步的交互。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 private static void Main(string[] args)

2 {

3 Calculator.AddAsync(1, 2);

4

5 //一直在干活

6

7 Thread.Sleep(1000); //挂起1秒钟

8 Console.WriteLine("AddAsync 方法执行完成");

9

10 Console.Read();

11 }

[复制代码](javascript:void(0);)

Program.cs

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Calculator

2 {

3 private static int Add(int n, int m)

4 {

5 return n + m;

6 }

7

8 public static async void AddAsync(int n, int m)

9 {

10 int val = await Task.Run(() => Add(n, m));

11 Console.WriteLine($"Result: {val}");

12 }

13 }

[复制代码](javascript:void(0);)

Calculator.cs

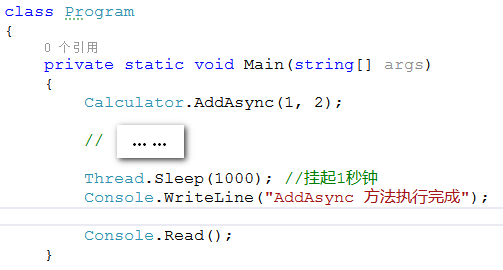


图6

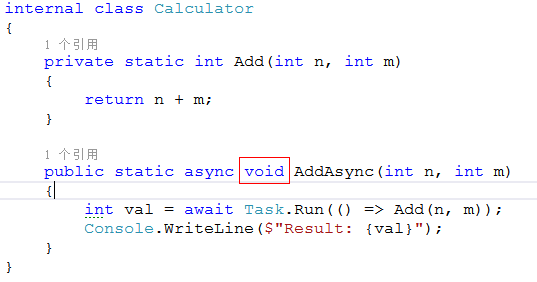


图7

**一、控制流**

     异步方法的结构可拆分成三个不同的区域：

     （1）表达式之前的部分：从方法头到第一个 await 表达式之间的所有代码。

     （2）await 表达式：将被异步执行的代码。

     （3）表达式之后的部分：await 表达式的后续部分。



　　图1-1

　　该异步方法执行流程：从await表达式之前的地方开始，同步执行到第一个 await，标识着第一部分执行结束，一般来说此时 await 工作还没完成。当await 任务完成后，该方法将继续同步执行后续部分。在执行的后续部分中，如果依然存在 await，就重复上述过程。

　　当到达 await 表达式时，线程将从异步方法返回到调用方法。如果异步方法的返回类型为 Task 或 Task<T>，会创建一个 Task 对象，标识需要异步完成的任务，然后将 Task 返回来调用方法。

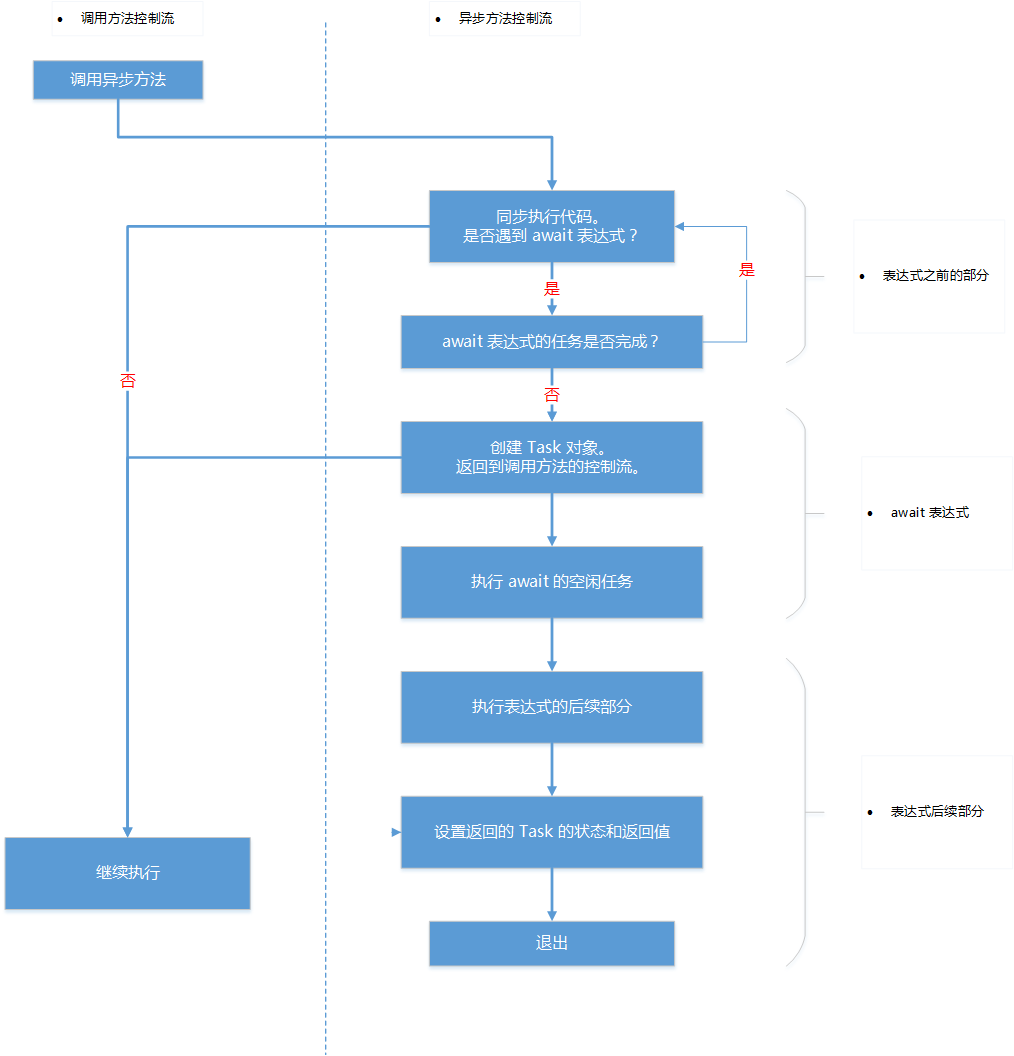


　　图1-2

　　异步方法的控制流：

　　①异步执行 await 表达式的空闲任务。

　　②await 表达式执行完成，继续执行后续部分。如再遇到 await 表达式，按相同情况进行处理。

　　③到达末尾或遇到 return 语句时，根据返回类型可以分三种情况：

　　　　a.void：退出控制流。

　　　　b.Task：设置 Task 的属性并退出。

　　　　c.Task<T>：设置 Task 的属性和返回值（Result 属性）并退出。

　　④同时，调用方法将继续执行，从异步方法获取 Task 对象。需要值的时候，会暂停等到 Task 对象的 Result 属性被赋值才会继续执行。

　　【难点】

　　①第一次遇到 await 所返回对象的类型。这个返回类型就是同步方法头的返回类型，跟 await 表达式的返回值没有关系。

　　②到达异步方法的末尾或遇到 return 语句，它并没有真正的返回一个值，而是退出了该方法。

**二、await 表达式**

　　await 表达式指定了一个异步执行的任务。默认情况，该任务在当前线程异步执行。

　　每一个任务就是一个 awaitable 类的实例。awaitable 类型指包含 GetAwaiter() 方法的类型。

　　实际上，你并不需要构建自己的 awaitable，一般只需要使用 Task 类，它就是 awaitable。

　　最简单的方式是在方法中使用 Task.Run() 来创建一个 Task。【注意】它是在不同的线程上执行方法。

　　让我们一起来看看示例。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 var t = Do.GetGuidAsync();

6 t.Wait();

7

8 Console.Read();

9 }

10

11

12 private class Do

13 {

14 /// <summary>

15 /// 获取 Guid

16 /// </summary>

17 /// <returns></returns>

18 private static Guid GetGuid() //与Func<Guid> 兼容

19 {

20 return Guid.NewGuid();

21 }

22

23 /// <summary>

24 /// 异步获取 Guid

25 /// </summary>

26 /// <returns></returns>

27 public static async Task GetGuidAsync()

28 {

29 var myFunc = new Func<Guid>(GetGuid);

30 var t1 = await Task.Run(myFunc);

31

32 var t2 = await Task.Run(new Func<Guid>(GetGuid));

33

34 var t3 = await Task.Run(() => GetGuid());

35

36 var t4 = await Task.Run(() => Guid.NewGuid());

37

38 Console.WriteLine($"t1: {t1}");

39 Console.WriteLine($"t2: {t2}");

40 Console.WriteLine($"t3: {t3}");

41 Console.WriteLine($"t4: {t4}");

42 }

43 }

44 }

[复制代码](javascript:void(0);)

View Code

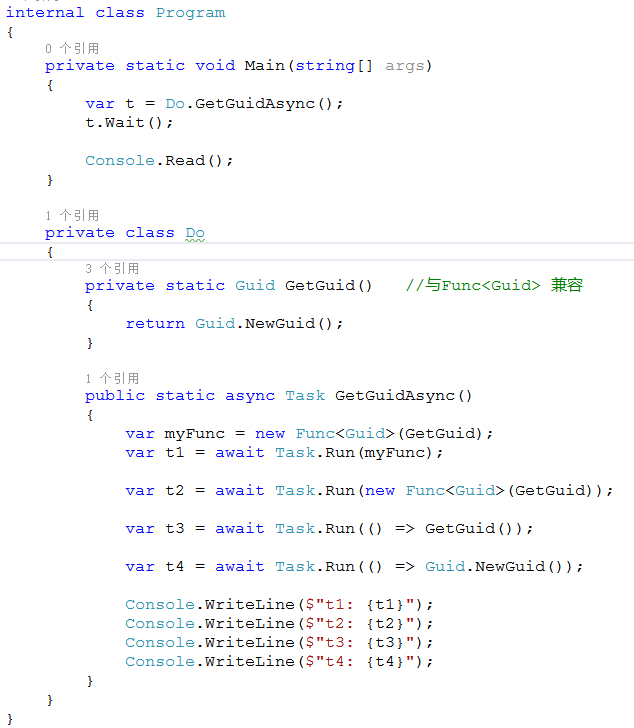


图2-1

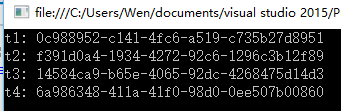


图2-2

 　　上面 4 个 Task.Run() 都是采用了 Task Run(Func<TReturn> func) 形式来直接或间接调用 Guid.NewGuid()。

　　Task.Run() 支持 4 中不同的委托类型所表示的方法：Action、Func<TResult>、Func<Task> 和 Func<Task<TResult>>

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 var t = Do.GetGuidAsync();

6 t.Wait();

7

8 Console.Read();

9 }

10

11 private class Do

12 {

13 public static async Task GetGuidAsync()

14 {

15 await Task.Run(() => { Console.WriteLine(Guid.NewGuid()); }); //Action

16

17 Console.WriteLine(await Task.Run(() => Guid.NewGuid())); //Func<TResult>

18

19 await Task.Run(() => Task.Run(() => { Console.WriteLine(Guid.NewGuid()); })); //Func<Task>

20

21 Console.WriteLine(await Task.Run(() => Task.Run(() => Guid.NewGuid()))); //Func<Task<TResult>>

22 }

23 }

24 }

[复制代码](javascript:void(0);)

View Code

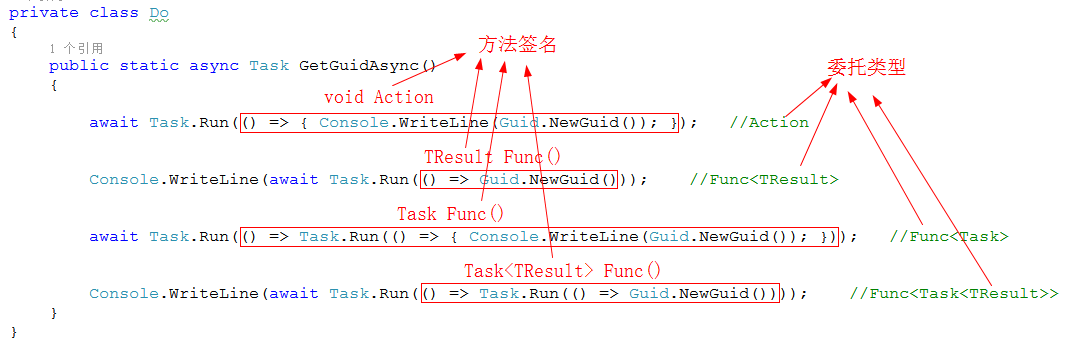


图2-3 Task.Run() 方法的重载

**三、How 取消异步操作**

 　　CancellationToken 和 CancellationTokenSource 这两个类允许你终止执行异步方法。

　　（1）CancellationToken 对象包含任务是否被取消的信息；如果该对象的属性 IsCancellationRequested 为 true，任务需停止操作并返回；该对象操作是不可逆的，且只能使用（修改）一次，即该对象内的 IsCancellationRequested 属性被设置后，就不能改动。

　　（2）CancellationTokenSource 可创建 CancellationToken 对象，调用 CancellationTokenSource 对象的 Cancel 方法，会使该对象的 CancellationToken 属性 IsCancellationRequested 设置为 true。

　　【注意】调用 CancellationTokenSource 对象的 Cancel 方法，并不会执行取消操作，而是会将该对象的 CancellationToken 属性 IsCancellationRequested 设置为 true。

　　示例

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 CancellationTokenSource source = new CancellationTokenSource();

6 CancellationToken token = source.Token;

7

8 var t = Do.ExecuteAsync(token);

9

10 //Thread.Sleep(3000); //挂起 3 秒

11 //source.Cancel(); //传达取消请求

12

13 t.Wait(token); //等待任务执行完成

14 Console.WriteLine($"{nameof(token.IsCancellationRequested)}: {token.IsCancellationRequested}");

15

16 Console.Read();

17 }

18

19

20 }

21

22 internal class Do

23 {

24 /// <summary>

25 /// 异步执行

26 /// </summary>

27 /// <param name="token"></param>

28 /// <returns></returns>

29 public static async Task ExecuteAsync(CancellationToken token)

30 {

31 if (token.IsCancellationRequested)

32 {

33 return;

34 }

35

36 await Task.Run(() => CircleOutput(token), token);

37 }

38

39 /// <summary>

40 /// 循环输出

41 /// </summary>

42 /// <param name="token"></param>

43 private static void CircleOutput(CancellationToken token)

44 {

45 Console.WriteLine($"{nameof(CircleOutput)} 方法开始调用：");

46

47 const int num = 5;

48 for (var i = 0; i < num; i++)

49 {

50 if (token.IsCancellationRequested) //监控 CancellationToken

51 {

52 return;

53 }

54

55 Console.WriteLine($"{i + 1}/{num} 完成");

56 Thread.Sleep(1000);

57 }

58 }

59 }

[复制代码](javascript:void(0);)

View Code

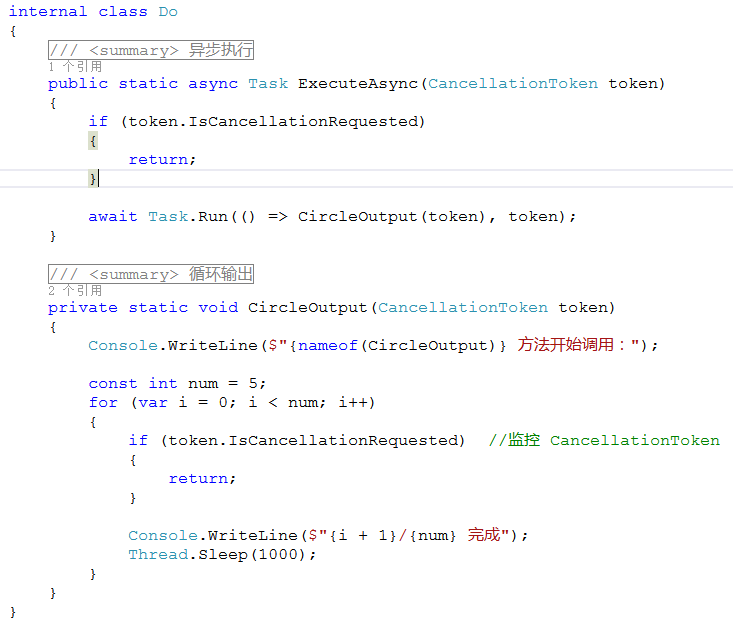


图3-1



图3-2 注释两行代码



图3-3：图3-1和图3-2的执行结果（注释两行代码）

　　上图是不调用 Cancel() 方法的结果图，不会取消任务的执行。

　　下图在 3 秒后调用 Cancel() 方法取消任务的执行：



图3-4：去掉注释

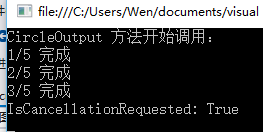
****

图3-5：图3-1和图3-4的执行结果（去掉注释）

**小结**

* 介绍异步方法的语法、三种不同的返回值类型（void、Task 和 Task<T>）和控制流程等。
* 简单常用的异步执行方式：Task.Run()。【注意】它是在不同的线程上执行方法。
* 如何取消异步操作。

**走进异步编程的世界 - 剖析异步方法（下）**

**序**

　　感谢大家的支持，这是昨天发布《[走进异步编程的世界 - 剖析异步方法（上）](http://www.cnblogs.com/liqingwen/p/5844095.html)》的补充篇。

**目录**

* [异常处理](http://www.cnblogs.com/liqingwen/p/5866241.html#link1)
* [在调用方法中同步等待任务](http://www.cnblogs.com/liqingwen/p/5866241.html#link2)
* [在异步方法中异步等待任务](http://www.cnblogs.com/liqingwen/p/5866241.html#link3)
* [Task.Delay() 暂停执行](http://www.cnblogs.com/liqingwen/p/5866241.html#link4)

**一、异常处理**

　　await 表达式也可以使用 try...catch...finally 结构。

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 var t = DoExceptionAsync();

6 t.Wait();

7

8 Console.WriteLine($"{nameof(t.Status)}: {t.Status}"); //任务状态

9 Console.WriteLine($"{nameof(t.IsCompleted)}: {t.IsCompleted}"); //任务完成状态标识

10 Console.WriteLine($"{nameof(t.IsFaulted)}: {t.IsFaulted}"); //任务是否有未处理的异常标识

11

12 Console.Read();

13 }

14

15 /// <summary>

16 /// 异常操作

17 /// </summary>

18 /// <returns></returns>

19 private static async Task DoExceptionAsync()

20 {

21 try

22 {

23 await Task.Run(() => { throw new Exception(); });

24 }

25 catch (Exception)

26 {

27 Console.WriteLine($"{nameof(DoExceptionAsync)} 出现异常！");

28 }

29 }

30 }

[复制代码](javascript:void(0);)

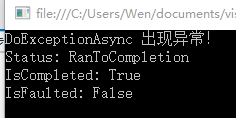


图1-1

　　【分析】await 表达式位于 try 块中，按普通的方式处理异常。但是，为什么图中的状态（Status）、是否完成标识（IsCompleted）和是否失败标识（IsFaulted）分别显示：运行完成（RanToCompletion） 、已完成（True） 和 未失败（False） 呢？因为：任务没有被取消，并且异常都已经处理完成！

**二、在调用方法中同步等待任务**

　　调用方法可能在某个时间点上需要等待某个特殊的 Task 对象完成，才执行后面的代码。此时，可以采用实例方法 Wait 。

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 var t = CountCharactersAsync("http://www.cnblogs.com/liqingwen/");

6

7 t.Wait(); //等待任务结束

8 Console.WriteLine($"Result is {t.Result}");

9

10 Console.Read();

11 }

12

13 /// <summary>

14 /// 统计字符数量

15 /// </summary>

16 /// <param name="address"></param>

17 /// <returns></returns>

18 private static async Task<int> CountCharactersAsync(string address)

19 {

20 var result = await Task.Run(() => new WebClient().DownloadStringTaskAsync(address));

21 return result.Length;

22 }

23 }

[复制代码](javascript:void(0);)

https://images2015.cnblogs.com/blog/711762/201609/711762-20160912230049180-840454239.png

图2-1

　　Wait() 适合用于单一 Task 对象，如果想操作一组对象，可采用 Task 的两个静态方法 WaitAll() 和 WaitAny() 。

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static int time = 0;

4 private static void Main(string[] args)

5 {

6 var t1 = GetRandomAsync(1);

7 var t2 = GetRandomAsync(2);

8

9 //IsCompleted 任务完成标识

10 Console.WriteLine($"t1.{nameof(t1.IsCompleted)}: {t1.IsCompleted}");

11 Console.WriteLine($"t2.{nameof(t2.IsCompleted)}: {t2.IsCompleted}");

12

13 Console.Read();

14 }

15

16 /// <summary>

17 /// 获取一个随机数

18 /// </summary>

19 /// <param name="id"></param>

20 /// <returns></returns>

21 private static async Task<int> GetRandomAsync(int id)

22 {

23 var num = await Task.Run(() =>

24 {

25 time++;

26 Thread.Sleep(time \* 100);

27 return new Random().Next();

28 });

29

30 Console.WriteLine($"{id} 已经调用完成");

31 return num;

32 }

33 }

[复制代码](javascript:void(0);)



图2-2 两个任务的 IsCompleted 属性都显示未完成

　　现在，在 Main() 方法中新增两行代码（6 和 7 两行），尝试调用 WaitAll() 方法。

[复制代码](javascript:void(0);)

1 private static void Main(string[] args)

2 {

3 var t1 = GetRandomAsync(1);

4 var t2 = GetRandomAsync(2);

5

6 Task<int>[] tasks = new Task<int>[] { t1, t2 };

7 Task.WaitAll(tasks); //等待任务全部完成，才继续执行

8

9 //IsCompleted 任务完成标识

10 Console.WriteLine($"t1.{nameof(t1.IsCompleted)}: {t1.IsCompleted}");

11 Console.WriteLine($"t2.{nameof(t2.IsCompleted)}: {t2.IsCompleted}");

12

13 Console.Read();

14 }

[复制代码](javascript:void(0);)

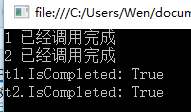


图2-3 两个任务的 IsCompleted 属性都显示 True

　　现在，再次将第 7 行改动一下，调用 WaitAny() 方法试试。

[复制代码](javascript:void(0);)

1 private static void Main(string[] args)

2 {

3 var t1 = GetRandomAsync(1);

4 var t2 = GetRandomAsync(2);

5

6 Task<int>[] tasks = new Task<int>[] { t1, t2 };

7 Task.WaitAny(tasks); //等待任一 Task 完成，才继续执行

8

9 //IsCompleted 任务完成标识

10 Console.WriteLine($"t1.{nameof(t1.IsCompleted)}: {t1.IsCompleted}");

11 Console.WriteLine($"t2.{nameof(t2.IsCompleted)}: {t2.IsCompleted}");

12

13 Console.Read();

14 }

[复制代码](javascript:void(0);)



图2-4 有一个任务的 IsCompleted 属性显示 True （完成） 就继续执行

**三、在异步方法中异步等待任务**

　　上节说的是如何使用 WaitAll() 和 WaitAny() 同步地等待 Task 完成。这次我们使用 Task.WhenAll() 和 Task.WhenAny()  在异步方法中异步等待任务。

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static int time = 0;

4

5 private static void Main(string[] args)

6 {

7 var t = GetRandomAsync();

8

9 Console.WriteLine($"t.{nameof(t.IsCompleted)}: {t.IsCompleted}");

10 Console.WriteLine($"Result: {t.Result}");

11

12 Console.Read();

13 }

14

15 /// <summary>

16 /// 获取一个随机数

17 /// </summary>

18 /// <param name="id"></param>

19 /// <returns></returns>

20 private static async Task<int> GetRandomAsync()

21 {

22 time++;

23 var t1 = Task.Run(() =>

24 {

25 Thread.Sleep(time \* 100);

26 return new Random().Next();

27 });

28

29 time++;

30 var t2 = Task.Run(() =>

31 {

32 Thread.Sleep(time \* 100);

33 return new Random().Next();

34 });

35

36 //异步等待集合内的 Task 都完成，才进行下一步操作

37 await Task.WhenAll(new List<Task<int>>() { t1, t2 });

38

39 Console.WriteLine($" t1.{nameof(t1.IsCompleted)}: {t1.IsCompleted}");

40 Console.WriteLine($" t2.{nameof(t2.IsCompleted)}: {t2.IsCompleted}");

41

42 return t1.Result + t2.Result;

43 }

44 }

[复制代码](javascript:void(0);)

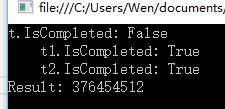


图3-1 调用 WhenAll()  方法

　　【注意】WhenAll() 异步等待集合内的 Task 都完成，不会占用主线程的时间。

 　　现在，我们把 GetRandomAsync() 方法内的 WhenAll() 方法替换成 WhenAny()，并且增大一下线程挂起时间，最终改动如下：

[复制代码](javascript:void(0);)

1 private static async Task<int> GetRandomAsync()

2 {

3 time++;

4 var t1 = Task.Run(() =>

5 {

6 Thread.Sleep(time \* 100);

7 return new Random().Next();

8 });

9

10 time++;

11 var t2 = Task.Run(() =>

12 {

13 Thread.Sleep(time \* 500); //这里由 100 改为 500，不然看不到效果

14 return new Random().Next();

15 });

16

17 //异步等待集合内的 Task 都完成，才进行下一步操作

18 //await Task.WhenAll(new List<Task<int>>() { t1, t2 });

19 await Task.WhenAny(new List<Task<int>>() { t1, t2 });

20

21 Console.WriteLine($" t1.{nameof(t1.IsCompleted)}: {t1.IsCompleted}");

22 Console.WriteLine($" t2.{nameof(t2.IsCompleted)}: {t2.IsCompleted}");

23

24 return t1.Result + t2.Result;

25 }

[复制代码](javascript:void(0);)

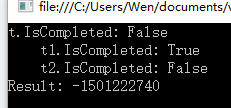


图3-2 调用 WhenAny() 方法

**四、Task.Delay() 暂停执行**

　　Task.Delay() 方法会创建一个 Task 对象，该对象将暂停其在线程中的处理，并在一定时间之后完成。和 Thread.Sleep 不同的是，它不会阻塞线程，意味着线程可以继续处理其它工作。

[复制代码](javascript:void(0);)

1 internal class Program

2 {

3 private static void Main(string[] args)

4 {

5 Console.WriteLine($"{nameof(Main)} - start.");

6 DoAsync();

7 Console.WriteLine($"{nameof(Main)} - end.");

8

9 Console.Read();

10 }

11

12 private static async void DoAsync()

13 {

14 Console.WriteLine($" {nameof(DoAsync)} - start.");

15

16 await Task.Delay(500);

17

18 Console.WriteLine($" {nameof(DoAsync)} - end.");

19 }

20 }

[复制代码](javascript:void(0);)

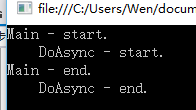


图4-1

**走进异步编程的世界 - 在 GUI 中执行异步操作**

**【博主】反骨仔　　【原文地址】**[**http://www.cnblogs.com/liqingwen/p/5877042.html**](http://www.cnblogs.com/liqingwen/p/5877042.html)

**序**

 　　这是继《[开始接触 async/await 异步编程](http://www.cnblogs.com/liqingwen/p/5831951.html)》、《[走进异步编程的世界 - 剖析异步方法](http://www.cnblogs.com/liqingwen/p/5844095.html)》后的第三篇。主要介绍在 WinForm 中如何执行异步操作。

**目录**

* [在 WinForm 中执行异步操作](http://www.cnblogs.com/liqingwen/p/5877042.html#link1)
* [在 WinForm 中使用异步 Lambda 表达式](http://www.cnblogs.com/liqingwen/p/5877042.html#link2)
* [一个完整的 WinForm 程序](http://www.cnblogs.com/liqingwen/p/5877042.html#link3)
* [另一种异步方式 - BackgroundWorker 类](http://www.cnblogs.com/liqingwen/p/5877042.html#link4)

**一、在 WinForm 程序中执行异步操作**

　　下面通过窗体示例演示以下操作-点击按钮后：

　　　　①将按钮禁用，并将标签内容改成：“Doing”（表示执行中）；

　　　　②线程挂起3秒（模拟耗时操作）；

　　　　③启用按钮，将标签内容改为：“Complete”（表示执行完成）。

[复制代码](javascript:void(0);)

1 public partial class Form1 : Form

2 {

3 public Form1()

4 {

5 InitializeComponent();

6 }

7

8 private void btnDo\_Click(object sender, EventArgs e)

9 {

10 btnDo.Enabled = false;

11 lblText.Text = @"Doing";

12

13 Thread.Sleep(3000);

14

15 btnDo.Enabled = true;

16 lblText.Text = @"Complete";

17 }

18 }

[复制代码](javascript:void(0);)

　　可是执行结果却是：

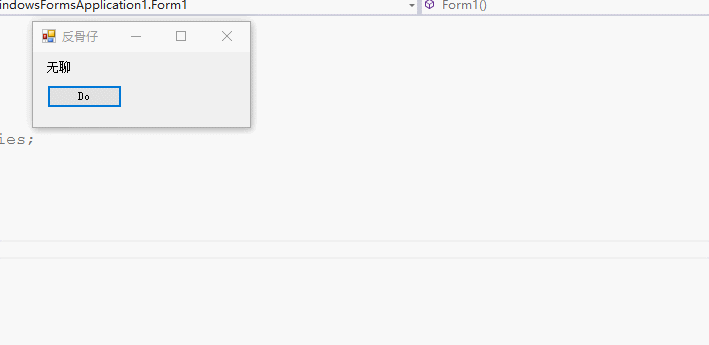


图1-1

　　【发现的问题】

　　　　①好像没有变成“Doing”？

　　　　②并且拖动窗口的时候卡住不动了？

　　　　③3秒后突然变到想拖动到的位置？

　　　　④同时文本变成“Complete”？

　　【分析】GUI 程序在设计中要求所有的显示变化都必须在主 GUI 线程中完成，如点击事件和移动窗体。Windows 程序时通过 消息来实现，消息放入消息泵管理的消息队列中。点击按钮时，按钮的Click消息放入消息队列。消息泵从队列中移除该消息，并开始处理点击事件的代码，即 btnDo\_Click 事件的代码。

　　btnDo\_Click 事件会将触发行为的消息放入队列，但在 btnDo\_Click 时间处理程序完全退出前（线程挂起 3 秒退出前），消息都无法执行。（3 秒后）接着所有行为都发生了，但速度太快肉眼无法分辨才没有发现标签改成“Doing”。

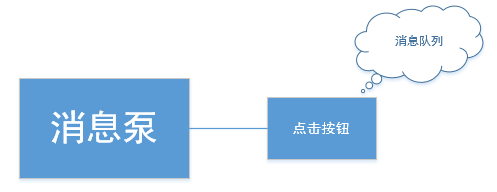


图1-2 点击事件

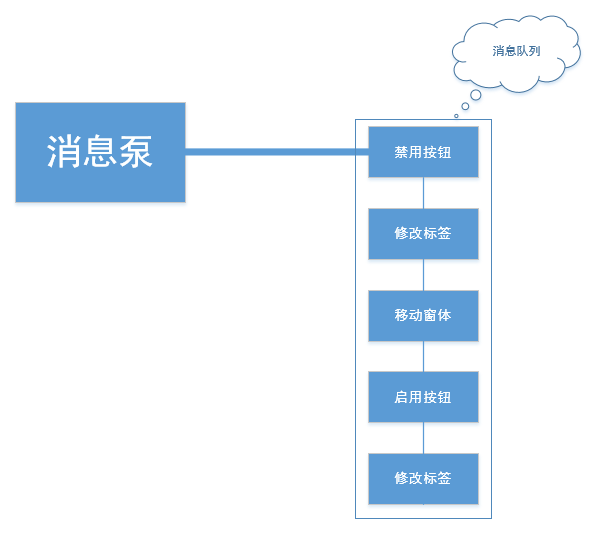


图1-3 点击事件具体执行过程

　　现在我们加入 async/await 特性。

[复制代码](javascript:void(0);)

1 public partial class Form1 : Form

2 {

3 public Form1()

4 {

5 InitializeComponent();

6 }

7

8 private async void btnDo\_Click(object sender, EventArgs e)

9 {

10 btnDo.Enabled = false;

11 lblText.Text = @"Doing";

12

13 await Task.Delay(3000);

14

15 btnDo.Enabled = true;

16 lblText.Text = @"Complete";

17 }

18 }

[复制代码](javascript:void(0);)

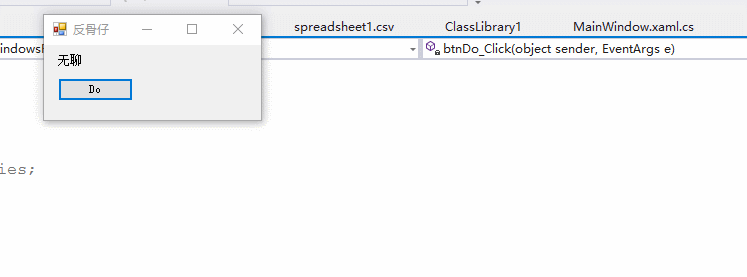


图1-4

　　现在，就是原先希望看到的效果。

　　【分析】btnDo\_Click 事件处理程序先将前两条消息压入队列，然后将自己从处理器移出，在3秒后（等待空闲任务完成后 Task.Delay ）再将自己压入队列。这样可以保持响应，并保证所有的消息可以在线程挂起的时间内被处理。

**1.1 Task.Yield**

　　Task.Yield 方法创建一个立刻返回的 awaitable。等待一个Yield可以让异步方法在执行后续部分的同时返回到调用方法。可以将其理解为 离开当前消息队列，回到队列末尾，让 CPU 有时间处理其它任务。

[复制代码](javascript:void(0);)

1 class Program

2 {

3 static void Main(string[] args)

4 {

5 const int num = 1000000;

6 var t = DoStuff.Yield1000(num);

7

8 Loop(num / 10);

9 Loop(num / 10);

10 Loop(num / 10);

11

12 Console.WriteLine($"Sum: {t.Result}");

13

14 Console.Read();

15 }

16

17 /// <summary>

18 /// 循环

19 /// </summary>

20 /// <param name="num"></param>

21 private static void Loop(int num)

22 {

23 for (var i = 0; i < num; i++) ;

24 }

25 }

26

27 internal static class DoStuff

28 {

29 public static async Task<int> Yield1000(int n)

30 {

31 var sum = 0;

32 for (int i = 0; i < n; i++)

33 {

34 sum += i;

35 if (i % 1000 == 0)

36 {

37 await Task.Yield(); //创建异步产生当前上下文的等待任务

38 }

39 }

40

41 return sum;

42 }

43 }

[复制代码](javascript:void(0);)

https://images2015.cnblogs.com/blog/711762/201609/711762-20160917180127695-561582939.png

 图1.1-1

　　上述代码每执行1000次循环就调用 Task.Yield 方法创建一个等待任务，让处理器有时间处理其它任务。该方法在 GUI 程序中是比较有用的。

**二、在 WinForm 中使用异步 Lambda 表达式**

　　将刚才的窗口程序的点击事件稍微改动一下。

[复制代码](javascript:void(0);)

1 public partial class Form1 : Form

2 {

3 public Form1()

4 {

5 InitializeComponent();

6

7 //async (sender, e) 异步表达式

8 btnDo.Click += async (sender, e) =>

9 {

10 Do(false, "Doing");

11

12 await Task.Delay(3000);

13

14 Do(true, "Finished");

15 };

16 }

17

18 private void Do(bool isEnable, string text)

19 {

20 btnDo.Enabled = isEnable;

21 lblText.Text = text;

22 }

23 }

[复制代码](javascript:void(0);)

　　还是原来的配方，还是熟悉的味道，还是原来哪个窗口，变的只是内涵。

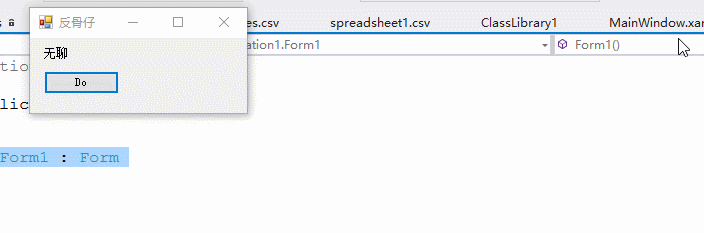


图2-1

**三、一个完整的 WinForm 程序**

　　现在在原来的基础上添加了进度条，以及取消按钮。

[复制代码](javascript:void(0);)

1 public partial class Form1 : Form

2 {

3 private CancellationTokenSource \_source;

4 private CancellationToken \_token;

5

6 public Form1()

7 {

8 InitializeComponent();

9 }

10

11 /// <summary>

12 /// Do 按钮事件

13 /// </summary>

14 /// <param name="sender"></param>

15 /// <param name="e"></param>

16 private async void btnDo\_Click(object sender, EventArgs e)

17 {

18 btnDo.Enabled = false;

19

20 \_source = new CancellationTokenSource();

21 \_token = \_source.Token;

22

23 var completedPercent = 0; //完成百分比

24 const int time = 10; //循环次数

25 const int timePercent = 100 / time; //进度条每次增加的进度值

26

27 for (var i = 0; i < time; i++)

28 {

29 if (\_token.IsCancellationRequested)

30 {

31 break;

32 }

33

34 try

35 {

36 await Task.Delay(500, \_token);

37 completedPercent = (i + 1) \* timePercent;

38 }

39 catch (Exception)

40 {

41 completedPercent = i \* timePercent;

42 }

43 finally

44 {

45 progressBar.Value = completedPercent;

46 }

47 }

48

49 var msg = \_token.IsCancellationRequested ? $"进度为：{completedPercent}% 已被取消！" : $"已经完成";

50

51 MessageBox.Show(msg, @"信息");

52

53 progressBar.Value = 0;

54 InitTool();

55 }

56

57 /// <summary>

58 /// 初始化窗体的工具控件

59 /// </summary>

60 private void InitTool()

61 {

62 progressBar.Value = 0;

63 btnDo.Enabled = true;

64 btnCancel.Enabled = true;

65 }

66

67 /// <summary>

68 /// 取消事件

69 /// </summary>

70 /// <param name="sender"></param>

71 /// <param name="e"></param>

72 private void btnCancel\_Click(object sender, EventArgs e)

73 {

74 if (btnDo.Enabled) return;

75

76 btnCancel.Enabled = false;

77 \_source.Cancel();

78 }

79 }

[复制代码](javascript:void(0);)

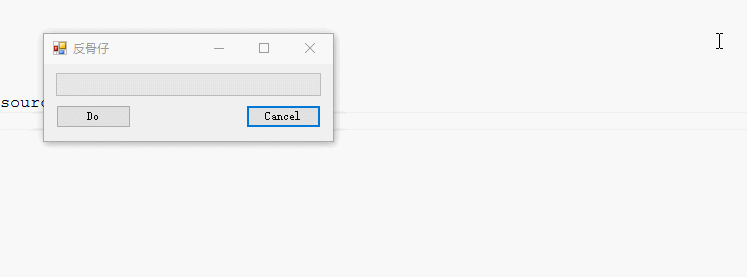


 图3-1

**四、另一种异步方式 - BackgroundWorker 类**

　　与 async/await 不同的是，你有时候可能需要一个额外的线程，在后台持续完成某项任务，并不时与主线程通信，这时就需要用到 BackgroundWorker 类。主要用于 GUI 程序。

　　书中的千言万语不及一个简单的示例。

[复制代码](javascript:void(0);)

1 public partial class Form2 : Form

2 {

3 private readonly BackgroundWorker \_worker = new BackgroundWorker();

4

5 public Form2()

6 {

7 InitializeComponent();

8

9 //设置 BackgroundWorker 属性

10 \_worker.WorkerReportsProgress = true; //能否报告进度更新

11 \_worker.WorkerSupportsCancellation = true; //是否支持异步取消

12

13 //连接 BackgroundWorker 对象的处理程序

14 \_worker.DoWork += \_worker\_DoWork; //开始执行后台操作时触发，即调用 BackgroundWorker.RunWorkerAsync 时触发

15 \_worker.ProgressChanged += \_worker\_ProgressChanged; //调用 BackgroundWorker.ReportProgress(System.Int32) 时触发

16 \_worker.RunWorkerCompleted += \_worker\_RunWorkerCompleted; //当后台操作已完成、被取消或引发异常时触发

17 }

18

19 /// <summary>

20 /// 当后台操作已完成、被取消或引发异常时发生

21 /// </summary>

22 /// <param name="sender"></param>

23 /// <param name="e"></param>

24 private void \_worker\_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

25 {

26 MessageBox.Show(e.Cancelled ? $@"进程已被取消：{progressBar.Value}%" : $@"进程执行完成：{progressBar.Value}%");

27 progressBar.Value = 0;

28 }

29

30 /// <summary>

31 /// 调用 BackgroundWorker.ReportProgress(System.Int32) 时发生

32 /// </summary>

33 /// <param name="sender"></param>

34 /// <param name="e"></param>

35 private void \_worker\_ProgressChanged(object sender, ProgressChangedEventArgs e)

36 {

37 progressBar.Value = e.ProgressPercentage; //异步任务的进度百分比

38 }

39

40 /// <summary>

41 /// 开始执行后台操作触发，即调用 BackgroundWorker.RunWorkerAsync 时发生

42 /// </summary>

43 /// <param name="sender"></param>

44 /// <param name="e"></param>

45 private static void \_worker\_DoWork(object sender, DoWorkEventArgs e)

46 {

47 var worker = sender as BackgroundWorker;

48 if (worker == null)

49 {

50 return;

51 }

52

53 for (var i = 0; i < 10; i++)

54 {

55 //判断程序是否已请求取消后台操作

56 if (worker.CancellationPending)

57 {

58 e.Cancel = true;

59 break;

60 }

61

62 worker.ReportProgress((i + 1) \* 10); //触发 BackgroundWorker.ProgressChanged 事件

63 Thread.Sleep(250); //线程挂起 250 毫秒

64 }

65 }

66

67 private void btnDo\_Click(object sender, EventArgs e)

68 {

69 //判断 BackgroundWorker 是否正在执行异步操作

70 if (!\_worker.IsBusy)

71 {

72 \_worker.RunWorkerAsync(); //开始执行后台操作

73 }

74 }

75

76 private void btnCancel\_Click(object sender, EventArgs e)

77 {

78 \_worker.CancelAsync(); //请求取消挂起的后台操作

79 }

80 }

[复制代码](javascript:void(0);)

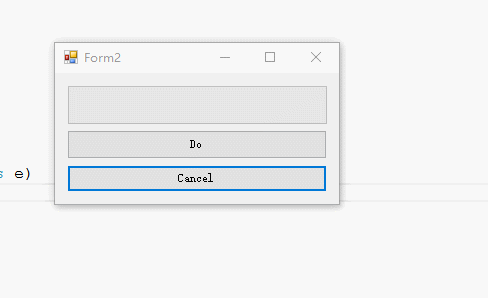


图4-1