

异步编程 - 学习笔记（一）

序

这篇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]
- [控制流]
- [await 表达式]
- [How 取消异步操作]

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 [zcz1024](#) 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`：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`。调用方法从 `Task` 的 `Result` 属性获取的就是 `T` 类型的值。

```

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    }

```

```

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    }

```

`class Program`

```

{
    0 个引用
    private static void Main(string[] args)
    {
        Task<int> t = Calculator.AddAsync(1, 2);

        // ... ..

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

        Console.Read();
    }
}

```

图2

```

1 1 个引用
internal class Calculator
{
    1 个引用
    private static int Add(int n, int m)
    {
        return n + m;
    }

    1 个引用
    public static async Task<int> AddAsync(int n, int m)
    {
        int val = await Task.Run(() => Add(n, m));
        return val;
    }
}

```

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。

```

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     }

```

```

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     }

```

```

class Program
{
    0 个引用
    private static void Main(string[] args)
    {
        Task t = Calculator.AddAsync(1, 2);

        // ... ..

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

        Console.Read();
    }
}

```

图4

```

1 1 个引用
internal class Calculator
{
    1 个引用
    private static int Add(int n, int m)
    {
        return n + m;
    }

    1 个引用
    public static async Task AddAsync(int n, int m)
    {
        int val = await Task.Run(() => Add(n, m));
        Console.WriteLine($"Result: {val}");
    }
}

```

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。

```

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    }

```

```

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 }

```

```

class Program
{
    0 个引用
    private static void Main(string[] args)
    {
        Calculator.AddAsync(1, 2);

        // ... ..

        Thread.Sleep(1000); //挂起1秒钟
        Console.WriteLine("AddAsync 方法执行完成");
    }

    Console.Read();
}

```

图6

```

internal class Calculator
{
    1 个引用
    private static int Add(int n, int m)
    {
        return n + m;
    }

    1 个引用
    public static async void AddAsync(int n, int m)
    {
        int val = await Task.Run(() => Add(n, m));
        Console.WriteLine($"Result: {val}");
    }
}

```

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。

```
internal class Download
{
    /// <summary>
    /// 统计字符个数
    /// </summary>
    /// <param name="id"></param>
    /// <param name="address"></param>
    /// <returns></returns>
    0 个引用
    public async Task<int> CountCharacters(int id, string address)
    {
        var wc = new WebClient();
        Console.WriteLine($"id: {id}, {address} 开始:");

        var result =
        Console.WriteLine($"id: {id}, {address} 结束.");

        return result.Length;
    }
}
```

The diagram illustrates the structure of an asynchronous method. It is divided into three regions by red boxes and arrows:

- await 表达式**: Points to the `await wc.DownloadStringTaskAsync(address);` line.
- 表达式之前的部分**: Points to the code block containing `var wc = new WebClient(); Console.WriteLine($"id: {id}, {address} 开始:");`.
- 表达式之后的部分**: Points to the code block containing `var result = Console.WriteLine($"id: {id}, {address} 结束."); return result.Length;`.

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，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。

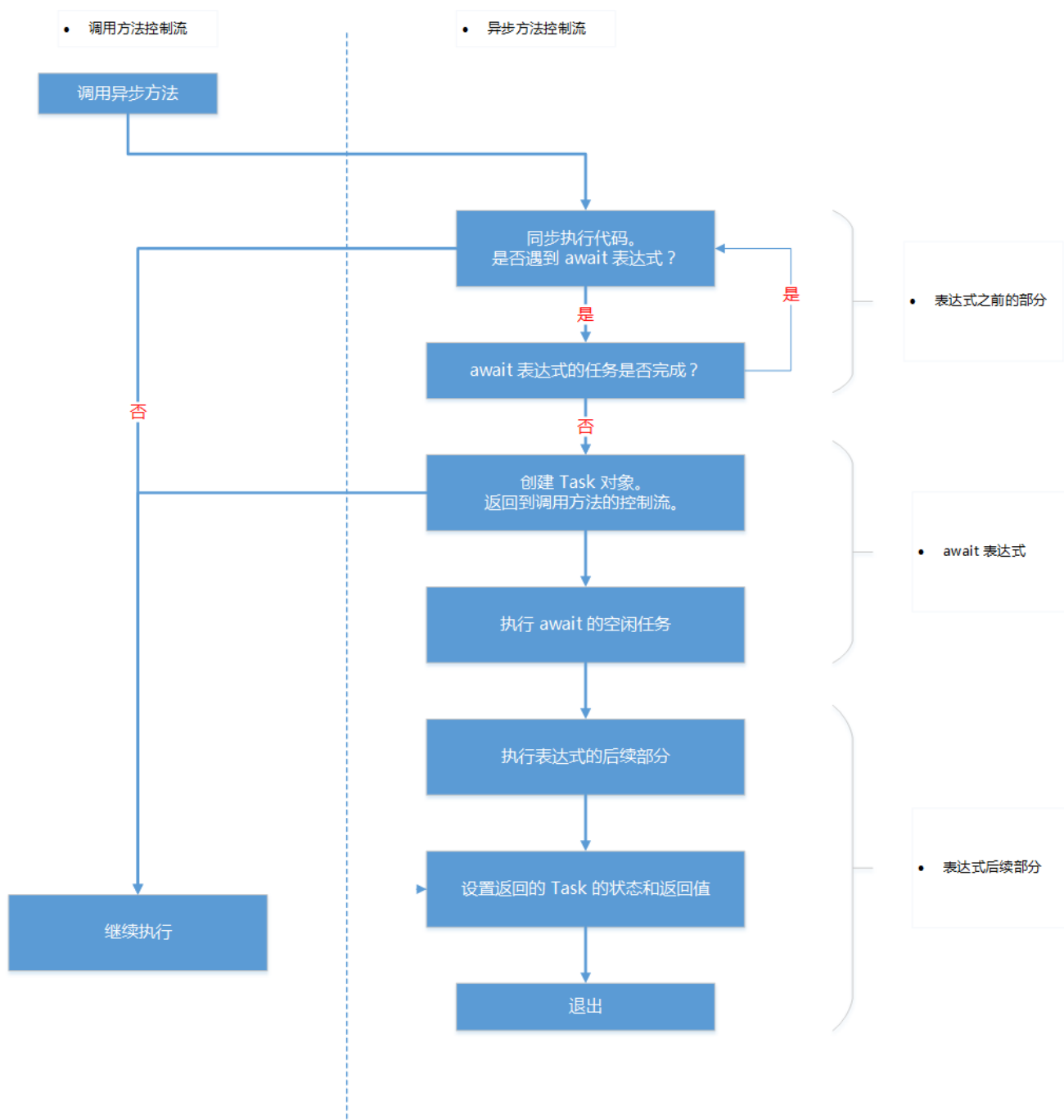


图1-2

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：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。

```
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 }
```

```

internal class Program
{
    0 个引用
    private static void Main(string[] args)
    {
        var t = Do.GetGuidAsync();
        t.Wait();

        Console.Read();
    }

    1 个引用
    private class Do
    {
        3 个引用
        private static Guid GetGuid()    //与Func<Guid> 兼容
        {
            return Guid.NewGuid();
        }

        1 个引用
        public static async Task GetGuidAsync()
        {
            var myFunc = new Func<Guid>(GetGuid);
            var t1 = await Task.Run(myFunc);

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

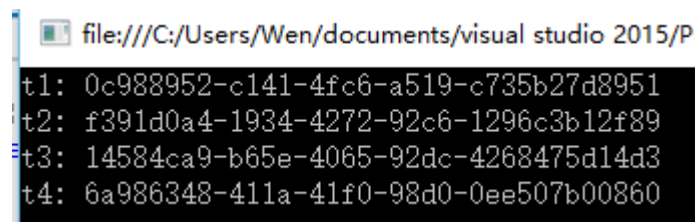
            var t3 = await Task.Run(() => GetGuid());

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

            Console.WriteLine($"t1: {t1}");
            Console.WriteLine($"t2: {t2}");
            Console.WriteLine($"t3: {t3}");
            Console.WriteLine($"t4: {t4}");
        }
    }
}

```

图2-1



```

file:///C:/Users/Wen/documents/visual studio 2015/P
t1: 0c988952-c141-4fc6-a519-c735b27d8951
t2: f391d0a4-1934-4272-92c6-1296c3b12f89
t3: 14584ca9-b65e-4065-92dc-4268475d14d3
t4: 6a986348-411a-41f0-98d0-0ee507b00860

```

图2-2

上面 4 个 Task.Run() 都是采用了 Task Run(Func 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、Func 和 Func<Task>

```

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()); });
16           //Action
17
18               Console.WriteLine(await Task.Run(() => Guid.NewGuid()));
19           //Func<TResult>
20
21               await Task.Run(() => Task.Run(() => {
22               Console.WriteLine(Guid.NewGuid()); })); //Func<Task>
23
24               Console.WriteLine(await Task.Run(() => Task.Run(() =>
25               Guid.NewGuid()))); //Func<Task<TResult>>
26           }
27       }
28   }

```

```

private class Do
{
    1 个引用
    public static async Task GetGuidAsync()
    {
        await Task.Run(() => { Console.WriteLine(Guid.NewGuid()); }); //Action
        Console.WriteLine(await Task.Run(() => Guid.NewGuid())); //Func<TResult>
        await Task.Run(() => Task.Run(() => { Console.WriteLine(Guid.NewGuid()); })); //Func<Task>
        Console.WriteLine(await Task.Run(() => Task.Run(() => Guid.NewGuid()))); //Func<Task<TResult>>
    }
}

```

Diagram annotations:

- 方法签名 (Method Signature) points to the lambda expressions in the Task.Run calls.
- 委托类型 (Delegate Type) points to the Task, Func, and Task<TResult> types.
- void Action points to the first Task.Run call.
- TResult Func() points to the second Task.Run call.
- Task Func() points to the third Task.Run call.
- Task<TResult> Func() points to the fourth Task.Run call.

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

```
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)}:
15            {token.IsCancellationRequested}");
16
17            Console.Read();
18        }
19    }
20
21    internal class Do
22    {
23        /// <summary>
24        /// 异步执行
25        /// </summary>
26        /// <param name="token"></param>
27        /// <returns></returns>
28        public static async Task ExecuteAsync(CancellationToken token)
29        {
30            if (token.IsCancellationRequested)
31            {
32                return;
33            }
34
35            await Task.Run(() => CircleOutput(token), token);
36        }
37
38        /// <summary>
39        /// 循环输出
40        /// </summary>
41        /// <param name="token"></param>
42        private static void CircleOutput(CancellationToken token)
43        {
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 }

```

```

internal class Do
{
    /// <summary> 异步执行
    1 个引用
    public static async Task ExecuteAsync(CancellationToken token)
    {
        if (token.IsCancellationRequested)
        {
            return;
        }

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

    /// <summary> 循环输出
    2 个引用
    private static void CircleOutput(CancellationToken token)
    {
        Console.WriteLine($"{nameof(CircleOutput)} 方法开始调用: ");

        const int num = 5;
        for (var i = 0; i < num; i++)
        {
            if (token.IsCancellationRequested) //监控 CancellationToken
            {
                return;
            }

            Console.WriteLine($"{i + 1}/{num} 完成");
            Thread.Sleep(1000);
        }
    }
}

```

图3-1

```

0 个引用
private static void Main(string[] args)
{
    CancellationTokenSource source = new CancellationTokenSource();
    CancellationToken token = source.Token;

    var t = Do.ExecuteAsync(token);

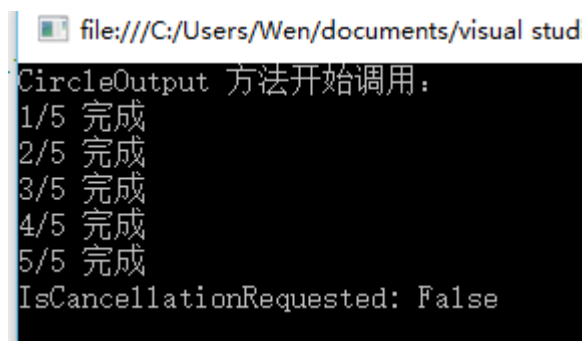
    //Thread.Sleep(3000);    //挂起 3 秒
    //source.Cancel();        //传达取消请求    注释

    t.Wait(token);    //等待任务执行完成
    Console.WriteLine($"{nameof(token.IsCancellationRequested)}: {token.IsCancellationRequested}");

    Console.Read();
}

```

图3-2 注释两行代码



```

file:///C:/Users/Wen/documents/visual stud
CircleOutput 方法开始调用:
1/5 完成
2/5 完成
3/5 完成
4/5 完成
5/5 完成
IsCancellationRequested: False

```

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：

```

private static void Main(string[] args)
{
    CancellationTokenSource source = new CancellationTokenSource();
    CancellationToken token = source.Token;

    var t = Do.ExecuteAsync(token);

    Thread.Sleep(3000);    //挂起 3 秒
    source.Cancel();        //传达取消请求    去掉注释

    t.Wait(token);    //等待任务执行完成
    Console.WriteLine($"{nameof(token.IsCancellationRequested)}: {token.IsCancellationRequested}");

    Console.Read();
}

```

图3-4: 去掉注释



```

file:///C:/Users/Wen/documents/visual
CircleOutput 方法开始调用:
1/5 完成
2/5 完成
3/5 完成
IsCancellationRequested: True

```

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）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。