## **基本介绍**

Async、Await是net4.x新增的异步编程方式，其目的是为了简化异步程序编写，和之前APM方式简单对比如下。

APM方式，BeginGetRequestStream需要传入回调函数，线程碰到BeginXXX时会以非阻塞形式继续执行下面逻辑，完成后回调先前传入的函数。

HttpWebRequest myReq =(HttpWebRequest)WebRequest.Create("http://cnblogs.com/");

myReq.BeginGetRequestStream();

//to do

Async方式，使用Async标记Async1为异步方法，用Await标记GetRequestStreamAsync表示方法内需要耗时的操作。主线程碰到await时会立即返回，继续以非阻塞形式执行主线程下面的逻辑。当await耗时操作完成时，继续执行Async1下面的逻辑

[IMG_256](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

static async void Async1()

{

HttpWebRequest myReq = (HttpWebRequest)WebRequest.Create("http://cnblogs.com/");

await myReq.GetRequestStreamAsync();

//to do

}

[IMG_257](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

上面是net类库实现的异步，如果要实现自己方法异步。  
APM方式：

public delegate int MyDelegate(int x);

MyDelegate mathDel = new MyDelegate((a) => { return 1; });

mathDel.BeginInvoke(1, (a) => { },null);

Async方式:

[IMG_258](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

static async void Async2()

{

await Task.Run(() => { Thread.Sleep(500); Console.WriteLine("bbb"); });

Console.WriteLine("ccc");

}

Async2();

Console.WriteLine("aaa");

[IMG_259](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

对比下来发现，async/await是非常简洁优美的，需要写的代码量更少，更符合人们编写习惯。   
**因为人的思维对线性步骤比较好理解的。**

APM异步回调的执行步骤是：A逻辑->假C回调逻辑->B逻辑->真C回调逻辑，这会在一定程度造成思维的混乱，当一个项目中出现大量的异步回调时，就会变的难以维护。  
Async、Await的加入让原先这种混乱的步骤，重新拨正了，执行步骤是：A逻辑->B逻辑->C逻辑。

## **基本原理剖析**

作为一个程序员的自我修养，刨根问底的好奇心是非常重要的。 Async刚出来时会让人有一头雾水的感觉，await怎么就直接返回了，微软怎么又出一套新的异步模型。那是因为习惯了之前的APM非线性方式导致的，现在重归线性步骤反而不好理解。 学习Async时候，可以利用已有的APM方式去理解，**以下代码纯属虚构**。  
比如把Async2方法想象APM方式的Async3方法：

[IMG_260](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

static async void Async3()

{

var task= await Task.Run(() => { Thread.Sleep(500); Console.WriteLine("bbb"); });

//注册task完成后回调

task.RegisterCompletedCallBack(() =>

{

Console.WriteLine("ccc");

});

}

[IMG_261](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

上面看其来就比较好理解些的，再把Async3方法想象Async4方法：

[IMG_262](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

static void Async4()

{

var thread = new Thread(() =>

{

Thread.Sleep(500);

Console.WriteLine("bbb");

});

//注册thread完成后回调

thread.RegisterCompletedCallBack(() =>

{

Console.WriteLine("ccc");

});

thread.Start();

}

[IMG_263](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

这样看起来就非常简单明了，连async都去掉了，变成之前熟悉的编程习惯。虽然代码纯属虚构，但基本思想是相通的，差别在于实现细节上面。

## **内部实现剖析**

作为一个程序员的自我修养，严谨更是不可少的态度。上面的基本思想虽然好理解了，但具体细节呢，编程是个来不得半点虚假的工作，那虚构的代码完全对不住看官们啊。

继续看Async2方法，反编译后的完整代码如下：

IMG_264

[IMG_265](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

internal class Program

{

// Methods

[AsyncStateMachine(typeof(<Async2>d\_\_2)), DebuggerStepThrough]

private static void Async2()

{

<Async2>d\_\_2 d\_\_;

d\_\_.<>t\_\_builder = AsyncVoidMethodBuilder.Create();

d\_\_.<>1\_\_state = -1;

d\_\_.<>t\_\_builder.Start<<Async2>d\_\_2>(ref d\_\_);

}

private static void Main(string[] args)

{

Async2();

Console.WriteLine("aaa");

Console.ReadLine();

}

// Nested Types [CompilerGenerated]

private struct <Async2>d\_\_2 : IAsyncStateMachine

{

// Fields

public int <>1\_\_state;

public AsyncVoidMethodBuilder <>t\_\_builder;

private object <>t\_\_stack;

private TaskAwaiter <>u\_\_$awaiter3;

// Methods

private void MoveNext()

{

try

{

TaskAwaiter awaiter;

bool flag = true;

switch (this.<>1\_\_state)

{

case -3:

goto Label\_00C5;

case 0:

break;

default:

if (Program.CS$<>9\_\_CachedAnonymousMethodDelegate1 == null)

{

Program.CS$<>9\_\_CachedAnonymousMethodDelegate1 = new Action(Program.<Async2>b\_\_0);

}

awaiter = Task.Run(Program.CS$<>9\_\_CachedAnonymousMethodDelegate1).GetAwaiter();

if (awaiter.IsCompleted)

{

goto Label\_0090;

}

this.<>1\_\_state = 0;

this.<>u\_\_$awaiter3 = awaiter;

this.<>t\_\_builder.AwaitUnsafeOnCompleted<TaskAwaiter, Program.<Async2>d\_\_2>(ref awaiter, ref this);

flag = false;

return;

}

awaiter = this.<>u\_\_$awaiter3;

this.<>u\_\_$awaiter3 = new TaskAwaiter();

this.<>1\_\_state = -1;

Label\_0090:

awaiter.GetResult();

awaiter = new TaskAwaiter();

Console.WriteLine("ccc");

}

catch (Exception exception)

{

this.<>1\_\_state = -2;

this.<>t\_\_builder.SetException(exception);

return;

}

Label\_00C5:

this.<>1\_\_state = -2;

this.<>t\_\_builder.SetResult();

}

[DebuggerHidden]

private void SetStateMachine(IAsyncStateMachine param0)

{

this.<>t\_\_builder.SetStateMachine(param0);

}

}

public delegate int MyDelegate(int x);

}

Collapse Methods

[IMG_266](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

发现async、await不见了，原来又是编译器级别提供的语法糖优化，所以说async不算是全新的异步模型。 可以理解为async更多的是线性执行步骤的一种回归，专门用来简化异步代码编写。  
从反编译后的代码看出编译器新生成一个继承IAsyncStateMachine 的状态机结构asyncd（代码中叫<Async2>d\_\_2，后面简写AsyncD)，**下面是基于反编译后的代码来分析的**。

IAsyncStateMachine最基本的状态机接口定义：

public interface IAsyncStateMachine

{

void MoveNext();

void SetStateMachine(IAsyncStateMachine stateMachine);

}

既然没有了async、await语法糖的阻碍，就可以把代码执行流程按线性顺序来理解，其整个执行步骤如下：

1. 主线程调用Async2()方法  
2. Async2()方法内初始化状态机状态为-1，启动AsyncD  
3. MoveNext方法内部开始执行，其task.run函数是把任务扔到线程池里，返回个可等待的任务句柄。MoveNext源码剖析：

//要执行任务的委托

Program.CS$<>9\_\_CachedAnonymousMethodDelegate1 = new Action(Program.<Async2>b\_\_0);

//开始使用task做异步，是net4.0基于任务task的编程方式。

awaiter =Task.Run(Program.CS$<>9\_\_CachedAnonymousMethodDelegate1).GetAwaiter();

//设置状态为0，以便再次MoveNext直接break，执行switch后面的逻辑，典型的状态机模式。

this.<>1\_\_state = 0;

//返回调用async2方法的线程，让其继续执行主线程后面的逻辑

this.<>t\_\_builder.AwaitUnsafeOnCompleted<TaskAwaiter, Program.<Async2>d\_\_2>(ref awaiter, ref this);return;

4. 这时就已经有2个线程在跑了，分别是主线程和Task.Run在跑的任务线程。

5. 执行主线程后面逻辑输出aaa，任务线程运行完成后输出bbb、在继续执行任务线程后面的业务逻辑输出ccc。

Label\_0090:

awaiter.GetResult();

awaiter = new TaskAwaiter();

Console.WriteLine("ccc");

这里可以理解为async把整个主线程同步逻辑，分拆成二块。 第一块是在主线程直接执行，第二块是在任务线程完成后执行， 二块中间是任务线程在跑，其源码中awaiter.GetResult()就是在等待任务线程完成后去执行第二块。   
从使用者角度来看执行步骤即为： 主线程A逻辑->异步任务线程B逻辑->主线程C逻辑。

[IMG_267](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

Test();

Console.WriteLine("A逻辑");

static async void Test()

{

await Task.Run(() => { Thread.Sleep(1000); Console.WriteLine("B逻辑"); });

Console.WriteLine("C逻辑");

}

[IMG_268](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

回过头来对比下基本原理剖析小节中的虚构方法Async4()，发现区别在于一个是完成后回调，一个是等待完成后再执行，这也是实现异步最基本的两大类方式。

## **重点注意的地方**

### **主线程A逻辑->异步任务线程B逻辑->主线程C逻辑。**

**注意：**这3个步骤是有可能会使用同一个线程的，也可能会使用2个，甚至3个线程。 可以用Thread.CurrentThread.ManagedThreadId测试下得知。

[IMG_269](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

Async7();

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

static async void Async7()

{

await Task.Run(() =>

{

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

});

Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

}

[IMG_270](http://www.cnblogs.com/mushroom/p/javascript:void(0);" \o "复制代码)

正由于此，才会有言论说Async不用开线程，也有说需要开线程的，从单一方面来讲都是对的，也都是错的。 上面源码是从简分析的，具体async内部会涉及到线程上下文切换，线程复用、调度等。 想深入的同学可以研究下ExecutionContextSwitcher、 SecurityContext.RestoreCurrentWI、ExecutionContext这几个东东。

其实具体的物理线程细节可以不用太关心，知道其【主线程A逻辑->异步任务线程B逻辑->主线程C逻辑】这个基本原理即可。 另外Async也会有线程开销的，所以要合理分业务场景去使用。

## **总结**

从逐渐剖析Async中发现，Net提供的异步方式基本上一脉相承的,如：  
1. net4.5的Async，抛去语法糖就是Net4.0的Task+状态机。   
2. net4.0的Task， 退化到3.5即是(Thread、ThreadPool)+实现的等待、取消等API操作。

本文以async为起点，简单剖析了其内部原理及实现，希望对大家有所帮助。