Источник: [C# async await Потоки ч.6 | www.nookery.ru](https://www.nookery.ru/thread6/) <https://www.nookery.ru/thread6/>

Вот мы и подобрались к завершающей части статей о потоках, сегодня мы рассмотрим работу async\await.

Ключевое слово async указывает компилятору, что метод, является асинхронным. await указывает компилятору, что в этой точке необходимо дождаться окончания асинхронной операции (при этом управление возвращается вызвавшему методу).

Оператор await заставляет метод, запускающий этот код, остановиться и дождаться завершения метода ShowAsync(), в итоге метод блокируется, пока пользователь не выберет какую-либо команду. В это время остальная программа продолжает отвечать на другие события. Как только метод ShowAsync() вернет управление, вызывавший его метод начнет работу с прерванной точки (хотя он может и дожидаться завершения возникших в промежутке событий). Метод, использующий оператор await, должен объявляться с модификатором async:

В async удобен тем что при возникновении исключительной ситуации, исключения выбрасывается в месте вызова асинхронной операции.

В этой теме мы рассмотрим как можно с помощью ключевых слов async\await использовать асинхронные методы, которые будут выполнятся асинхронно.

ВАЖНО! Стоит помнить что async\await в разных контекстах приложений используется по разному, к примеру в asp и wpf они имеют отличия в работе.

Простой пример работы с async await:

 {

            for (int i = 0; i < 80; i++)

            {

                Thread.Sleep(100);

                Console.ForegroundColor = ConsoleColor.Yellow;

                Console.Write("-");

            }

        }

        //помечаем ключевым словом async метод

        public async static void MethodAsync()

        {

//Эта часть кода будет работать в основном потоке

            //создаем задачу передаем в делегат метод Method

            Task t = new Task(Method);

            //запускаем задачу

            t.Start();

//а вот эта часть завершится в 2 потоке

            //ожидаем завершения задачи

            await t;

        }

        static void Main(string[] args)

        {

            MethodAsync();

            Console.WriteLine("Main завершился");

            Console.ReadKey();

        }

С первого взгляда кажется ничего сложного, и возникает больше вопросов чем ответов, если посмотреть код метода MethodAsync вообще не понятно что делает async и для чего нужен вообще await. Данный пример не несет полезной нагрузки и мы могли обойтись без этих ключевых слов, ничего бы не изменилось, однако он демонстрирует работу async await. Все дело в том что эти слова позволяют компилятору указать метод, для которого будет произведена автоматическая генерация кода. Это создано для упрощения в разработке и уменьшения кода программы, вам не придется строчить кучу однотипного кода каждый раз, это все будет сделано за вас!

А теперь посмотрите как наш пример, выглядит под рефлектором,  сгенерировалось куча строк кода.

using System;

using System.Diagnostics;

using System.Runtime.CompilerServices;

using System.Threading;

using System.Threading.Tasks;

namespace ConsoleApp50

{

internal class Prog

{

public static void Method()

{

for (int index = 0; index < 80; ++index)

{

Thread.Sleep(100);

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("-");

}

}

[AsyncStateMachine(typeof(Prog.d\_\_1))]

[DebuggerStepThrough]

public static void MethodAsync()

{

Prog.d\_\_1 stateMachine = new Prog.d\_\_1();

stateMachine.t\_\_builder = AsyncVoidMethodBuilder.Create();

stateMachine.E1\_\_state = -1;

stateMachine.t\_\_builder.Start < Prog.d\_\_1 > (ref stateMachine);

}

private static void Main(string[] args)

{

Prog.MethodAsync();

Console.WriteLine("Main завершился");

Console.ReadKey();

}

public Prog()

{

return;

}

[CompilerGenerated]

private sealed class d\_\_1 : IAsyncStateMachine

{

public int E1\_\_state;

public AsyncVoidMethodBuilder t\_\_builder;

private Task E5\_\_1;

private TaskAwaiter u\_\_1;

public d\_\_1()

{

return;

}

void IAsyncStateMachine.MoveNext()

{

int num1 = this.E1\_\_state;

try

{

TaskAwaiter awaiter;

int num2;

if (num1 != 0)

{

// ISSUE: method pointer

this.E5\_\_1 = new Task(new Action(Method));

this.E5\_\_1.Start();

awaiter = this.E5\_\_1.GetAwaiter();

if (!awaiter.IsCompleted)

{

this.E1\_\_state = num2 = 0;

this.u\_\_1 = awaiter;

Prog.d\_\_1 stateMachine = this;

this.t\_\_builder.AwaitUnsafeOnCompleted < TaskAwaiter, Prog.d\_\_1 > (ref awaiter, ref stateMachine);

return;

}

}

else

{

awaiter = this.u\_\_1;

this.u\_\_1 = new TaskAwaiter();

this.E1\_\_state = num2 = -1;

}

awaiter.GetResult();

}

catch (Exception ex)

{

this.E1\_\_state = -2;

this.t\_\_builder.SetException(ex);

return;

}

this.E1\_\_state = -2;

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

}

[DebuggerHidden]

void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)

{

}

}

}

}

Представленный выше код, я не стал исправлять, а скопировал его как он находился в рефлекторе. Давайте его разберем для большего понимания. Наши методы не поменялись, за исключением **MethodAsync()** и появился класс d\_\_1 : IAsyncStateMachine . Внутри которого имеются методы **MoveNext();** и **SetStateMachine();** Внутри класса так же имеются поля, основное из них это public AsyncVoidMethodBuilder t\_\_builder; который представляет конструктор для асинхронных методов, которые не возвращают никакое значение. Метод **stateMachine.t\_\_builder.Start < Prog.d\_\_1 > (ref stateMachine);** вызывает метод**MoveNext();** Где происходит запуск задачи, нового потока и вызова метода **Method();** Внутри метода **MoveNext()** так же имеется переменная которая ожидает завершения задачи, по окончанию работы метода, условия if (num1 != 0) становиться false и метод **MoveNext();** завершает свою работу. Для понимания работы я вам советую скопировать код в студию и под отладкой посмотреть его работу.

В следующем примере я покажу как можно передать аргументы в метод, а так же получить из него результат:

public static string Method(object o)

{

for (int i = 0; i < 40; i++)

{

Thread.Sleep(100);

Console.ForegroundColor = ConsoleColor.Yellow;

Console.Write("-");

}

return o+" nookery";

}

//помечаем ключевым словом async метод

public static async void MethodAsync()

{

//создаем задачу и запускаем ее, передав имя метода и аргумент.

Task<string> t = Task<string>.Factory.StartNew(Method,"Hello");

//ожидаем завершения задачи и выводим результат

Console.WriteLine(await t);

}

static void Main(string[] args)

{

MethodAsync();

Console.WriteLine("Main завершился");

Console.ReadKey();

}

Когда вызывается Task. Factory, происходит обращение к статическому свойс­тву класса Task, которое возвращает стандартный объект фабрики задач, т.е. TaskFactory. Назначение фабрики задач заключается в создании задач — в частнос­ти, трех видов задач:

* «обычных» задач (через метод StartNew);
* продолжений с множеством предшественников (через методы ContinueWhenAllи ContinueWhenAny);
* задач, которые являются оболочками для методов, следующих устаревшему шаб­лону АРМ

Следующий пример покажет как рассчитать сумму чисел в методе в отдельном потоке  и вернуть результат в основной поток:

public static int Method(int x, int y)

{

return x + y;

}

async static Task<int> Start(int x,int y)

{

int resualt= await Task.Run(() => Method(x,y));

return resualt;

}

static void Main(string[] args)

{

//Используем свойство Result который возвращает нам результат вычислений

Console.WriteLine("Сумма чисел: 3+4="+Start(3,4).Result);

Console.WriteLine("Main завершился");

Console.ReadKey();

}

Программисты считают, что модификатор async и оператор await вносят в код путаницу. Поэтому важно усвоить следующее. 

* Если метод объявлен с модификатором async , это еще не означает, что он выполняется в асинхронном режиме. Это означает, что метод может содержать инструкции, которые могут выполняться в асинхронном режиме.
* Оператор await показывает, что метод должен быть запущен в отдельной задаче и что вызывающий код приостанавливается, пока не будет завершен вызов метода. Поток, используемый вызывающим кодом, высвобождается и может быть использован повторно. Это важно в том случае, если это тот самый поток, который используется пользовательским интерфейсом, поскольку это позволяет сохранить его отзывчивость на действия пользователя.
* Оператор await не является функциональным аналогом принадлежащего задаче метода Wait , который всегда блокирует текущий поток и не допускает его повторного использования, пока задача не завершится. Изначально код, возобновляющий выполнение после оператора await , пытается получить исходный поток, который был использован для вызова асинхронного метода. Если этот поток занят, код будет блокирован. Чтобы указать, что выполнение кода может быть возобновлено в любом доступном потоке, и сократить шансы на его блокировку, можно воспользоваться методом ConfigureAwait(false) . Особую пользу это принесет веб-приложениями сервисам, которым может понадобиться обслуживать многие тысячи одновременно поступающих запросов.
* Неосмотрительное использование асинхронных методов, возвращающих результаты и запускаемых в потоке пользовательского интерфейса, может привести к возникновению взаимных блокировок и стать причиной зависания приложения.