

# Магические слова `async / await`

C# Professional



Проверить, идет ли запись

# Меня хорошо видно & слышно?



Ставим в чат “+”, если все хорошо  
“-”, если есть проблемы



## Михаил Дмитриев

Ведущий программист НИПК Электрон

- 5+ лет опыта промышленной разработки на платформе Net
- 20+ лет опыта работы в IT сфере
- В моей компании разрабатываю и поддерживаю приложения для работы с радиологическими комплексами

 <https://t.me/sf321>

# Правила вебинара



Активно  
участвуем



Off-topic обсуждаем  
в учебной группе в  
телеграмме



Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

# Карта курса



**СТАРТ**

Архитектура  
проекта и БД

Клиент-серверная  
архитектура и  
микросервисы

**Многопоточность и  
шаблоны  
проектирования**

C# Advanced

Процессы и  
подходы

Командный  
проект

**ФИНИШ**



# Маршрут вебинара

статусы Task

AWAIT - паттерн awaitable

ASYNC - асинхронная машина состояний

контекст синхронизации

примеры

рефлексия



# Цели вебинара

К концу занятия вы сможете

# Смысл

Для чего вам это уметь

1. Понимать что находится “под капотом”  
async/ await

2. Изучить понятие контекста  
синхронизации

3. Максимально эффективно  
использовать ресурсы системы, избегая  
блокировок потоков

писать и поддерживать корректный  
асинхронный код





# Тестирование



# Жизненный цикл Task

---

# Что хранит в себе Task

- Метод который нужно выполнить

```
internal object m_action;
```

- Статус

```
public TaskStatus status;
```

- Результат

```
internal TResult m_result;
```

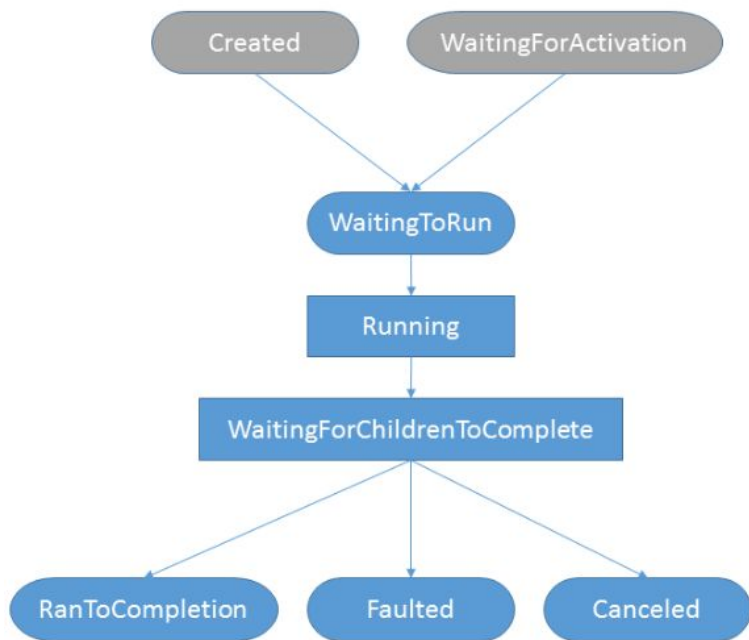
- Список исключений

```
internal volatile TaskExceptionHolder m_exceptionsHolder; // в свойствах ContingentProperties
```

<https://referencesource.microsoft.com/#mscorlib/system/threading/Tasks/Task.cs>

<https://referencesource.microsoft.com/#mscorlib/system/threading/Tasks/Future.cs>

# Статусы Task

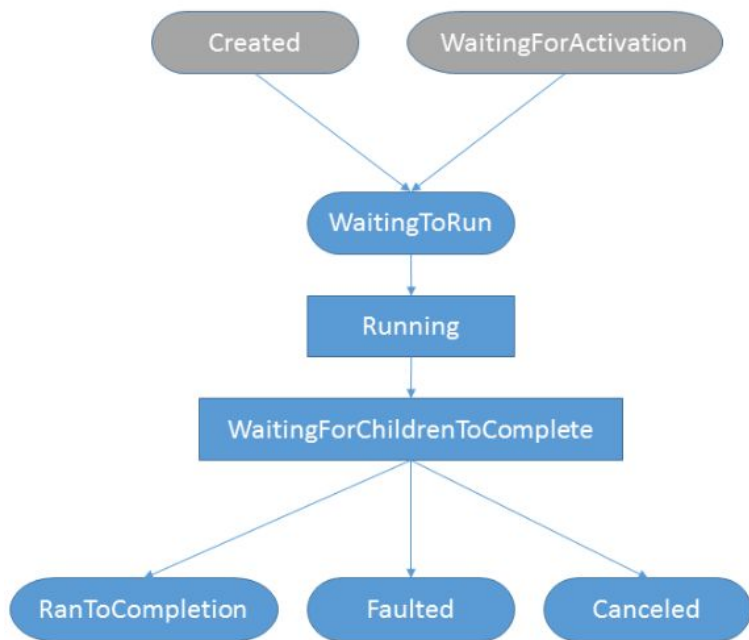


**Created** – создана, но не запланирована. Задача, созданная через конструктор. До вызова **Start** пребывает в этом статусе.

**WaitingForActivation** – ждет активации. В этом статусе создаются задачи при использовании метода **ContinueWith**

**WaitingToRun** – задаче уже назначен шедюлер, но она еще не запущена. Задачи, созданные методами **Run**, **TaskFactory.StartNew** начинают цикл с этого статуса.

# Статусы Task



**Running** – выполняется

**WaitingForChildrenToComplete** – ожидание окончания дочерних тасок. В этот статус таска попадает после своего завершения

**RanToCompletion** – успешно завершена

**Canceled** – отменена

**Faulted** – выполнение закончилось ошибкой

Status	IsCompleted	IsCanceled	IsFaulted
other	✗	✗	✗
RanToCompletion	✓	✗	✗
Canceled	✓	✓	✗
Faulted	✓	✗	✓

# ContinueWith

**Продолжение(continuation)** - это задача, созданная в состоянии **WaitingForActivation**. Она активируется автоматически по завершению предыдущей задачи или предыдущих задач.

```
var task = Task.Run(DoSomething);

task.ContinueWith(t : Task =>
{
    // Выполнится асинхронно в отдельном потоке
}); // Task

await task;
// Выполнится в оригинальном контексте
```

# Awaitable Pattern

---

# Awaitable

Чтобы быть **awaitable**, тип T должен иметь экземплярный метод **GetAwaiter()** без параметров (или должен быть соответствующий метод расширения).

Метод **GetAwaiter()** должен возвращать тип **awaiter**.

Тип **awaiter** должен:

- реализовывать интерфейс **INotifyCompletion**, имеющий метод **void OnCompleted(Action)**.
- иметь экземплярное свойство **bool IsCompleted**.
- иметь экземплярный метод **GetResult()**.

Типы реализующие паттерн **Awaitable**:

- **Task**
- **Task<TResult>**
- **ValueTask**
- **ValueTask<TResult>**
- *Custom types that implement the pattern*

**LIVE**

---



# TaskAwaiter

**IsCompleted** – делегируется taskе

```
public bool IsCompleted
{
    get { return m_task.IsCompleted; }
}
```

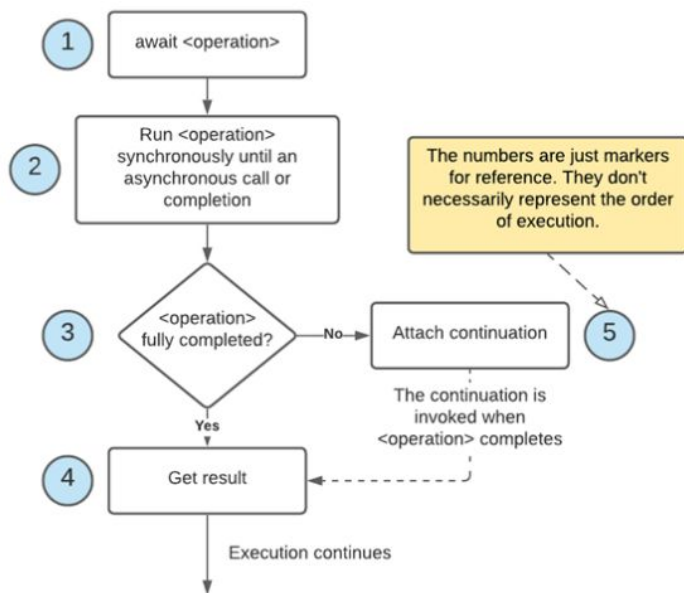
**GetResult** – если есть исключения, бросает. Если вызвано вручную, ждет синхронно

**OnCompleted(Action action)** – выполняет заданное действие после завершения ожидания

<https://referencesource.microsoft.com/#mscorlib/system/runtime/compiler/services/TaskAwaiter.cs.be57b6bc41e5c7e4>



# TaskAwaiter - схема работы



```
1. public async Task MyAsyncMethod(int x)
2. {
3.     if(x == 0)
4.         return;
5.
6.     await Task.Delay(1000);
7.
8.     // Do something after await
9. }
```

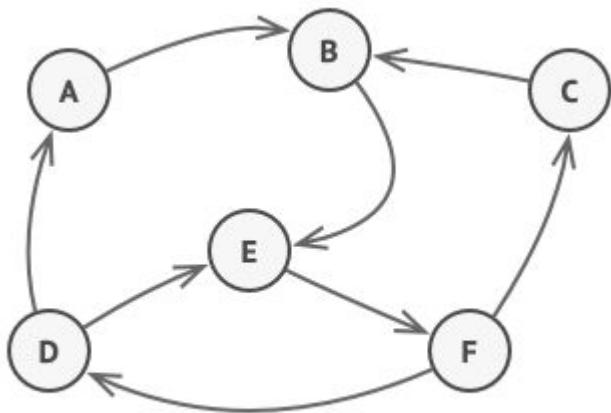
```
1. static async Task Main()
2. {
3.     await MyAsyncMethod(0);
4. }
```

```
1. static async Task Main()
2. {
3.     await MyAsyncMethod(1);
4. }
```

# Асинхронная машина состояний

---

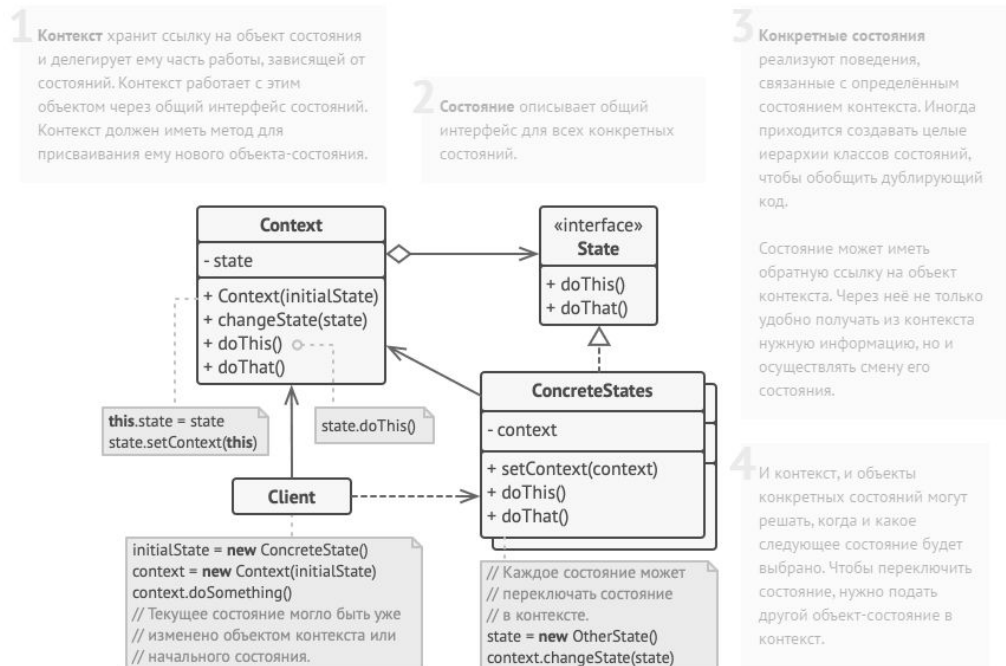
# Стейт машина - реализация паттерна состояние



*Конечный автомат.*

# Состояние (State)

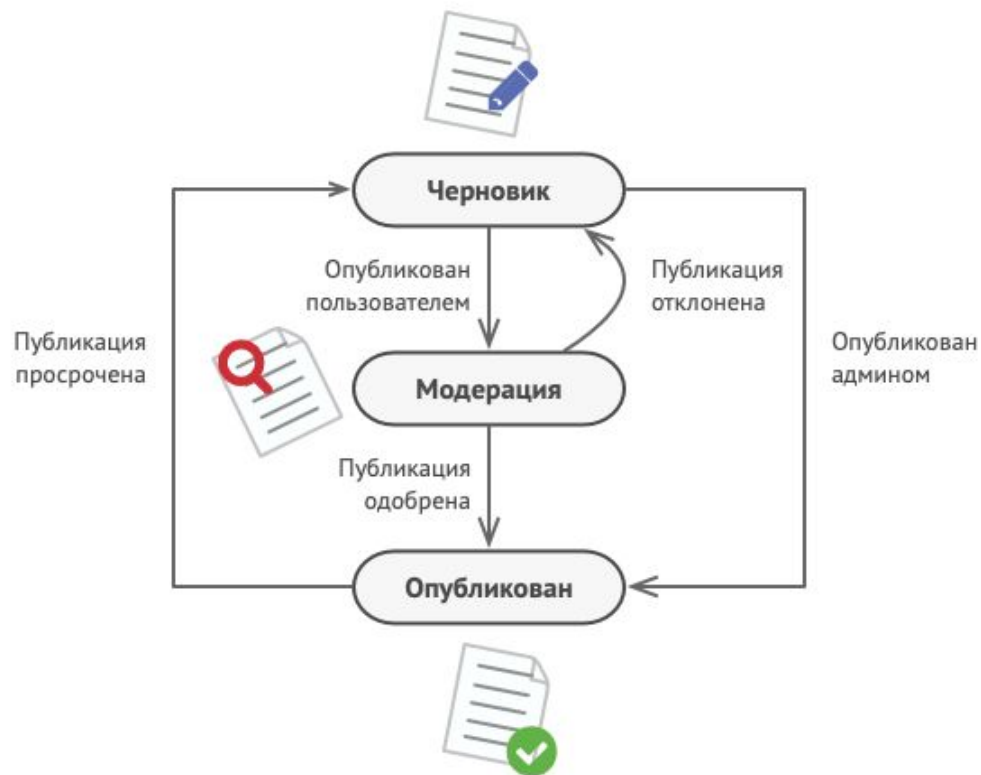
Паттерн State позволяет объектам менять поведение в зависимости от своего состояния. Извне создается впечатление, что изменился класс объекта.



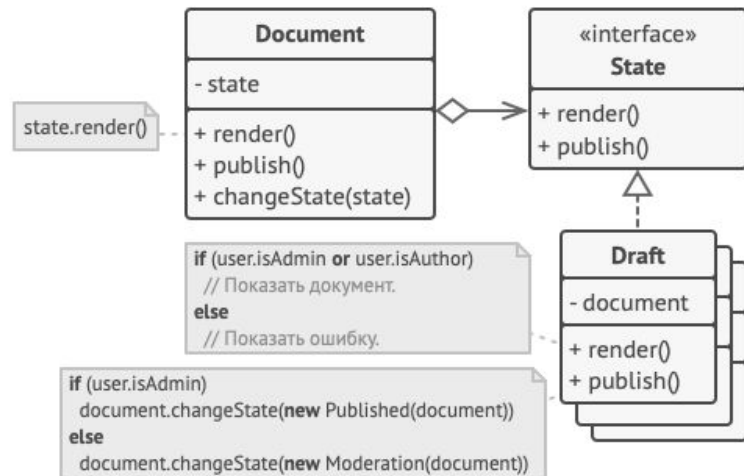
Условия применимости:

- поведение объекта зависит от его состояния и должно изменяться во время выполнения;
- когда в коде операций встречаются состояния из многих ветвей условные операторы, в которых выбор ветви зависит от состояния. Обычно в таком случае состояние представлено перечисляемыми константами. Часто одна и та же структура условного оператора повторяется в нескольких операциях. Паттерн состояние предлагает поместить каждую ветвь в отдельный класс. Это позволяет трактовать состояние объекта как самостоятельный объект, который может изменяться независимо от других.

# Состояние (State)



Возможные состояния документа и переходы между ними.



Документ делегирует работу своему активному объекту-состоянию.

# IAsyncStateMachine

```
/// <summary>
/// Represents state machines generated for asynchronous methods.
/// This type is intended for compiler use only.
/// </summary>
public interface IAsyncStateMachine
{
    /// <summary>Moves the state machine to its next state.</summary>
    void MoveNext();
    /// <summary>Configures the state machine with a heap-allocated replica.</summary>
    /// <param name="stateMachine">The heap-allocated replica.</param>
    void SetStateMachine(IAsyncStateMachine stateMachine);
}
```

<https://github.com/microsoft/referencesource/blob/master/mscorlib/system/runtime/compiler/services/IAsyncStateMachine.cs>

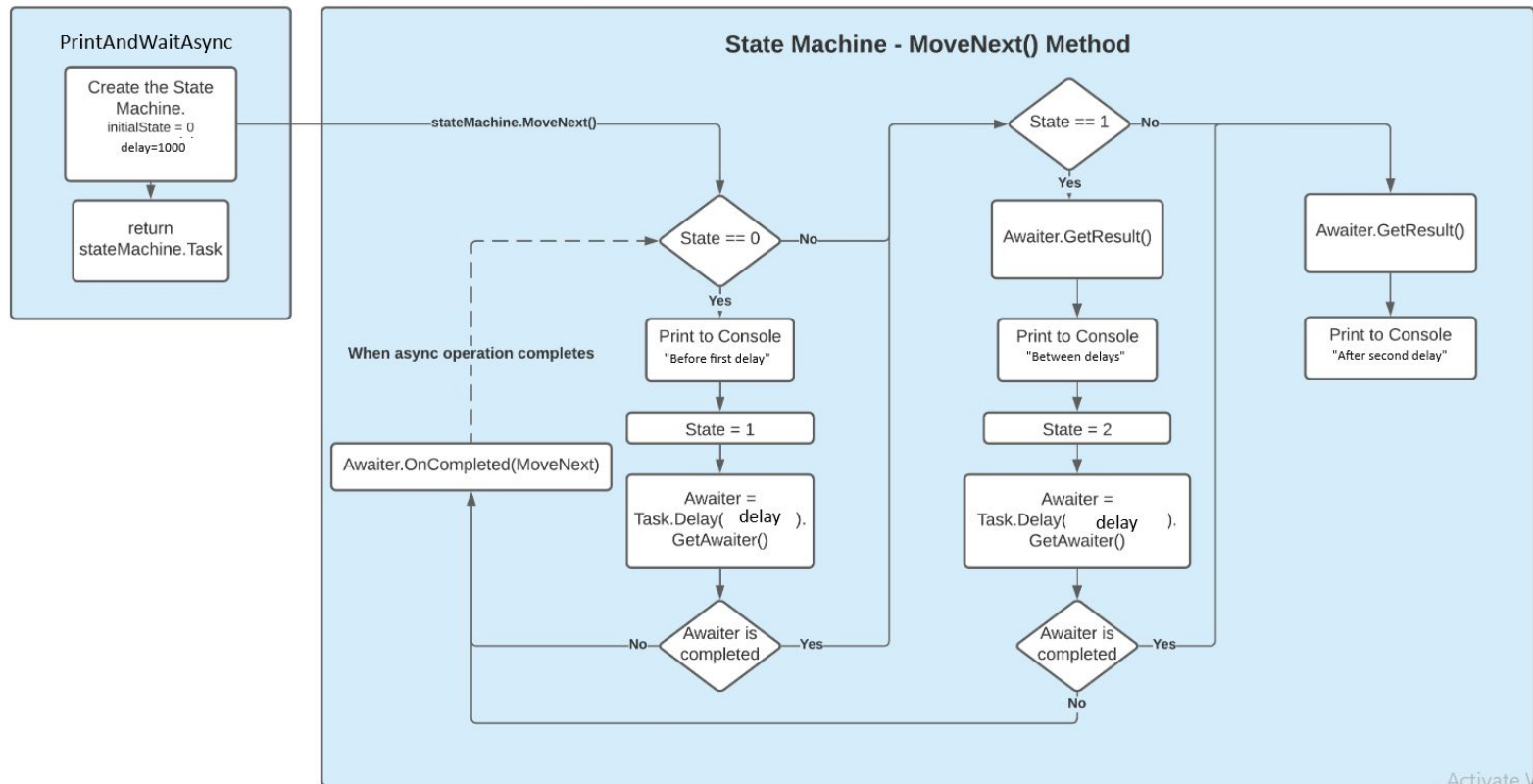


# Стейт машина - изменения состояний

```
public async Task PrintAndWaitAsync(TimeSpan delay, int arg2)
{
    Console.WriteLine("Before first delay");           State 0
    await Task.Delay(delay);
    Console.WriteLine("Between delays");               State 1
    await Task.Delay(delay);
    Console.WriteLine("After second delay");           State 2
}
```



# Стейт машина - изменение состояний



# Boxing-unboxing и SetStateMachine

Асинхронная стейтмашина – структура, но **после первого await она кладется в кучу**, чтобы не потерять данные во время ожидания.

Метод **SetStateMachine** применяется для того чтобы сослаться на сохраненную в кучу копию.

```
/// <summary>Configures the state machine with a heap-allocated replica.</summary>  
/// <param name="stateMachine">The heap-allocated replica.</param>  
void SetStateMachine(IAsyncStateMachine stateMachine);
```

# IAsyncStateMachine

Статусы асинхронной стейтмашины:

**-2** – финальный статус (или успешное выполнение, или исключение, или отмена задачи)

**-1** – обычное состояние, когда нет ожидания await

**0** – в первом await


**1** – во втором await (если есть)

**2** – в третьем await (если есть)

и т.д. (по количеству await в методе)

# Асинхронный метод “под капотом”

```
public async Task PrintAndWait(TimeSpan delay, int arg2)
{
    Console.WriteLine("Before first delay");
    await Task.Delay(delay);
    Console.WriteLine("Between delays");
    await Task.Delay(delay);
    Console.WriteLine("After second delay");
}
```



```
1 [AsyncStateMachine(typeof(PrintAndWaitStateMachine))]
2 [DebuggerStepThrough]
3 public Task PrintAndWait(TimeSpan delay, int arg2)
4 {
5     PrintAndWaitStateMachine stateMachine = new PrintAndWaitStateMachine()
6     {
7         Delay = delay,
8         Arg2 = arg2,
9         Builder = AsyncTaskMethodBuilder.Create(),
10        State = -1
11    };
12    stateMachine.Builder.Start(ref stateMachine);
13    return stateMachine.Builder.Task;
14 }
```

# Асинхронная стейтмашина

```
1 [CompilerGenerated]
2 class PrintAndWaitStateMachine : IAsyncStateMachine
3 {
4     public int State;
5     public AsyncTaskMethodBuilder Builder;
6     public TimeSpan delay;
7     public int arg2;
8
9     private TaskAwaiter _awaiter;
10
11     void IAsyncStateMachine.MoveNext()
12     {
13         int num = State;
14         try
15         {
16             здесь основной код MoveNext (удалено)
17         }
18         catch (Exception exception)
19         {
20             State = -2;
21             Builder.SetException(exception);
22             return;
23         }
24         State = -2;
25         Builder.SetResult();
26     }
27
28     void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)
29     {
30         this.Builder.SetStateMachine(stateMachine);
31     }
32 }
```

# StateMachineBuilder

*AsyncTaskMethodBuilder*

*AsyncTaskMethodBuilder<TResult>*

*AsyncVoidMethodBuilder*

**SetResult** - завершает задачу.

**SetException** – завершает задачу, сохраняет исключения в задачу

**AwaitOnCompleted (AwaitUnsafeOnCompleted)** – вызывает OnCompleted у TaskAwaiter, который вызывает SetContinuationForAwait у задачи

<https://github.com/microsoft/referencesource/blob/master/Microsoft.Bcl/System.Threading.Tasks.v1.5/System/Runtime/CompilerServices/AsyncTaskMethodBuilderOfTResult.cs>

<https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Runtime/CompilerServices/AsyncTaskMethodBuilderT.cs>



# OnCompleted vs UnsafeOnCompleted

**AwaitOnCompleted** передает **ExecutionContext**,

**AwaitUnsafeOnCompleted** не передает потому что он передается другим способом

```
public void OnCompleted(Action continuation)
{
    TaskAwaiter.OnCompletedInternal(m_task, continuation, m_continueOnCapturedContext, flowExecutionContext: true);
}
```

```
public void UnsafeOnCompleted(Action continuation)
{
    TaskAwaiter.OnCompletedInternal(m_task, continuation, m_continueOnCapturedContext, flowExecutionContext: false);
}
```

**UnsafeOnCompleted** предназначен для вызова только доверенной асинхронной инфраструктурой, такой как `AsyncTaskMethodBuilder`. `AsyncTaskMethodBuilder` гарантирует, что он всегда захватывает контекст выполнения. Вот почему он вызывает небезопасный метод, чтобы `TaskAwaiter` избежал его повторного захвата.



# Что навешивается на колбек

```
public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(
    ref TAwaiter awaiter, ref TStateMachine stateMachine)
    where TAwaiter : ICriticalNotifyCompletion
    where TStateMachine : IAsyncStateMachine
{
    try
    {
        var continuation = m_coreState.GetCompletionAction(ref this, ref stateMachine);
        Contract.Assert(continuation != null, "GetCompletionAction should always return a valid action.");
        awaiter.UnsafeOnCompleted(continuation);
    }
    catch (Exception e)
    {
        AsyncServices.ThrowAsync(e, targetContext: null);
    }
}
```

<https://github.com/microsoft/referencesource/blob/master/Microsoft.Bcl/System.Threading.Tasks.v1.5/System/Runtime/CompilerServices/AsyncTaskMethodBuilderOfTResult.cs>





# Что навешивается на колбек

```
internal void Run()
{
    Contract.Assert(m_stateMachine != null, "The state machine must have been set before calling Run.");

    if (m_context != null)
    {
        try
        {
            // Get the callback, lazily initializing it as necessary
            Action<object> callback = s_invokeMoveNext;
            if (callback == null) { s_invokeMoveNext = callback = InvokeMoveNext; }

            if (m_context == null)
            {
                callback(m_stateMachine);
            }
            else
            {
                // Use the context and callback to invoke m_stateMachine.MoveNext.
                ExecutionContextLightup.Instance.Run(m_context, callback, m_stateMachine);
            }
        }
        finally { if (m_context != null) m_context.Dispose(); }
    }
    else
    {
        m_stateMachine.MoveNext();
    }
}
```

<https://github.com/microsoft/referencesource/blob/master/Microsoft.Bcl/System.Threading.Tasks.v1.5/System/Runtime/CompilerServices/AsyncMethodBuilderCore.cs#L99>



# Основные методы используемые в стейт машине

Метод вызываемый MoveNext	Основная функция
GetResult	Бросить исключения если они записаны в таске
IsCompleted	Проверка, закончил ли выполнение метод данного эвейтера
SetException	Сохранение исключения в таску
SetResult	1.Перевод таски в финальный статус 2.Сохраняется результат (если он есть) 3.Выполнение методов ContinueWith (если они есть)
AwaitUnsafeOnCompleted	Запланировать вызов MoveNext после завершения метода эвейтера

# Метод MoveNext()

```
1 MoveNext()
2 {
3     int num = State;
4     try
5     {
6         TaskAwaiter awaiter;
7         TaskAwaiter awaiter2;
8         if (num != 0)
9         {
10             if (num == 1)
11             {
12                 awaiter = _awaiter;
13                 _awaiter = default(TaskAwaiter);
14                 num = (State = -1);
15                 goto IL_00ef;
16             }
17             Console.WriteLine("Before first delay");
18             awaiter2 = Task.Delay(delay).GetAwaiter();
19             if (!awaiter2.IsCompleted)
20             {
21                 num = (State = 0);
22                 _awaiter = awaiter2;
23                 PrintAndWaitStateMachine stateMachine = this;
24                 Builder.AwaitUnsafeOnCompleted(ref awaiter2, ref stateMachine);
25                 return;
26             }
27 }
```

```
28     else
29     {
30         awaiter2 = _awaiter;
31         _awaiter = default(TaskAwaiter);
32         num = (State = -1);
33     }
34     awaiter2.GetResult();
35     Console.WriteLine("Between delays");
36     awaiter = Task.Delay(delay).GetAwaiter();
37     if (!awaiter.IsCompleted)
38     {
39         num = (State = 1);
40         _awaiter = awaiter;
41         PrintAndWaitStateMachine stateMachine = this;
42         Builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);
43         return;
44     }
45     goto IL_00ef;
46     IL_00ef:
47     awaiter.GetResult();
48     Console.WriteLine("After second delay");
49 }
50 catch (Exception exception)
51 {
52     State = -2;
53     Builder.SetException(exception);
54     return;
55 }
56 State = -2;
57 Builder.SetResult();
58 }
```

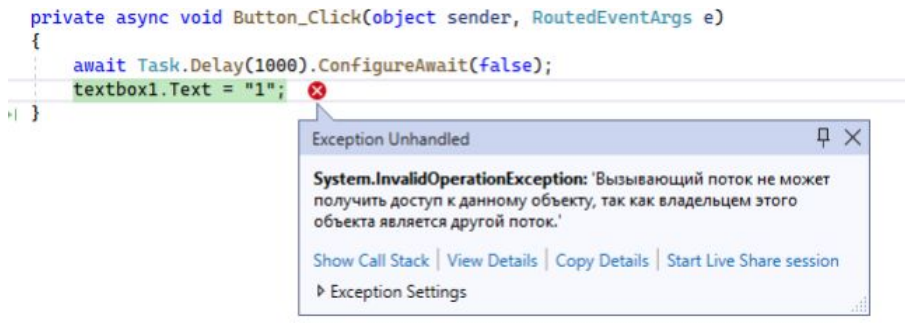
# Контекст синхронизации

---

# В каком потоке выполнится продолжение асинхронного метода после возвращения из await?

```
public async Task ExecuteAsync()
{
    Console.WriteLine("Start");
    await InternalAsync();
    Console.WriteLine("Finish");
}
```

Продолжение может быть выполнено либо в том же потоке, либо в другом



**LIVE**

---

# ConfigureAwait(false)

**ConfigureAwait(bool continueOnCapturedContext)**

**continueOnCapturedContext = true** – исполнять продолжение на исходном потоке

**continueOnCapturedContext = false** – не исполнять продолжение на исходном потоке

# SynchronizationContext

```
public async Task ExecuteAsync()
{
    Console.WriteLine("Start");
    await InternalAsync();
    Console.WriteLine("Finish");
}
```

За то, в каком потоке **может быть выполнено продолжение**, отвечает контекст синхронизации **SynchronizationContext**

Терминология:

“Выполнить продолжение в том же потоке, что его начало” = “Продолжить на захваченном контексте”,  
“Захватить контекст”

captureContext, continueOnCapturedContext





# SynchronizationContext

## **Преимущества возврата в исходный контекст:**

- В некоторых случаях это необходимо, например для WPF (пример выше)

## **Преимущества отсутствия возврата в исходный контекст:**

- Исключен deadlock
- Работает быстрее

<https://devblogs.microsoft.com/dotnet/configureawait-faq/>

# SynchronizationContext

По-разному используется в разных подплатформах .Net (WPF, winforms, Asp.Net (framework) есть, в Asp.Net Core, консольные приложения нет ).

```
public ActionResult Index()  
{
```

```
    Console.WriteLine(SynchronizationContext.Current);
```

> [AspNetSynchronizationContext]

```
[HttpGet(template: "{id}")]
```

```
2 1 <1> *
```

```
public async Task<IActionResult> Get(int id, CancellationToken cancel
```

```
{
```

```
    Console.WriteLine(SynchronizationContext.Current);
```

[SynchronizationContext] null

# SynchronizationContext

Перед await сохраняется, а при возобновлении после await загружается и колбек после возобновления выполняется уже в нем

```
// If the caller wants to continue on the current context/scheduler and there is one,
// fall back to using the state machine's delegate.
if (continueOnCapturedContext)
{
    SynchronizationContext? syncCtx = SynchronizationContext.Current;
    if (syncCtx != null && syncCtx.GetType() != typeof(SynchronizationContext))
    {
        var tc = new SynchronizationContextAwaitTaskContinuation(syncCtx, stateMachineBox.MoveNextAction);
        if (!AddTaskContinuation(tc, addBeforeOthers: false)){...}
        return;
    }
    else
    {
        TaskScheduler? scheduler = TaskScheduler.InternalCurrent;
        if (scheduler != null && scheduler != TaskScheduler.Default)
        {
            var tc = new TaskSchedulerAwaitTaskContinuation(scheduler, stateMachineBox.MoveNextAction);
            if (!AddTaskContinuation(tc, addBeforeOthers: false)){...}
            return;
        }
    }
}
```

# SynchronizationContext

Частью сущности продолжения асинхронного метода является контекст синхронизации, несущий информацию о потоке выполнения

```
/// <summary>Task continuation for awaiting with a current synchronization context.</summary>
internal sealed class SynchronizationContextAwaitTaskContinuation : AwaitTaskContinuation
{
    /// <summary>SendOrPostCallback delegate to invoke the action.</summary>
    private static readonly SendOrPostCallback s_postCallback = static state =>
    {
        Debug.Assert(state is Action);
        ((Action)state)();
    };

    /// <summary>Cached delegate for PostAction</summary>
    private static ContextCallback? s_postActionCallback;

    /// <summary>The context with which to run the action.</summary>
    private readonly SynchronizationContext m_syncContext;
```

# SynchronizationContext

Частью сущности продолжения асинхронного метода является контекст синхронизации, несущий информацию о потоке выполнения

```
/// <summary>Task continuation for awaiting with a current synchronization context.</summary>
internal sealed class SynchronizationContextAwaitTaskContinuation : AwaitTaskContinuation
{
    /// <summary>SendOrPostCallback delegate to invoke the action.</summary>
    private static readonly SendOrPostCallback s_postCallback = static state =>
    {
        Debug.Assert(state is Action);
        ((Action)state)();
    };

    /// <summary>Cached delegate for PostAction</summary>
    private static ContextCallback? s_postActionCallback;

    /// <summary>The context with which to run the action.</summary>
    private readonly SynchronizationContext m_syncContext;
```

# SynchronizationContext

Метод может возобновиться в потоке, отличном от того, где был начат, при выполнении одного из следующих условий:

- если объект Task сконфигурирован так, что при возобновлении исходный поток не используется (**ConfigureAwait(false)**), при условии существования текущего контекста синхронизации (это WPF или winforms приложение, использующее **.net framework** или **Asp.net framework** приложение, также это может быть кастомный контекст синхронизации).
- если в точке, где встретился оператор await, вообще не было текущего контекста синхронизации, как, например, в **консольном приложении**, **ASP.NET CORE**;
- если запомненный контекст SynchronizationContext инкапсулирует несколько потоков, например пул потоков;



# Контексты синхронизации

- Базовый  
<https://referencesource.microsoft.com/#mscorlib/system/threading/synchronizationcontext.cs>
- Winforms  
<https://referencesource.microsoft.com/#System.Windows.Forms/winforms/Managed/System/WinForms/WindowsFormsSynchronizationContext.cs,c7dfb662bbd6227d>
- WPF - Dispatcher  
<https://referencesource.microsoft.com/#WindowsBase/Base/System/Windows/Threading/DispatcherSynchronizationContext.cs,f640e296cad20594>
- WinRT  
<https://github.com/dotnet/runtime/blob/60d1224ddd68d8ac0320f439bb60ac1f0e9cdb27/src/libraries/System.Runtime.WindowsRuntime/src/System/Threading/WindowsRuntimeSynchronizationContext.cs>
- Default SynchronizationContext (консоль) – асинхронные операции берет на себя ThreadPool
- AspNetSynchronizationContext (ASP.Net) – основная задача – сохранить HttpContext.Current (контекст выполнения HTTP запросов: логины, хедеры, язык и т.д.)

**LIVE**

---



# Что делать с `ConfigureAwait(false)`



При разработке во фреймворке `ConfigureAwait(false)` использовать не нужно. Если же вы пишете код внутренней библиотеки, используйте `ConfigureAwait(false)`

# Типы возвращаемых значений async методов

- Task — для асинхронного метода, не возвращающего значение
- Task<TResult> — для асинхронного метода, возвращающего значение
- ValueTask
- ValueTask<TResult>
- void — для обработчика событий (event handler)
- IEnumerable<T>\* — для асинхронного метода, который возвращает асинхронный поток

## Comparison with `IEnumerable<T>` and `Task<IEnumerable<T>>`

Feature	<code>IEnumerable&lt;T&gt;</code>	<code>Task&lt;IEnumerable&lt;T&gt;&gt;</code>	<code>IAsyncEnumerable&lt;T&gt;</code>
Synchronous	Yes	No	No
Asynchronous	No	Partial	Yes
Memory Efficiency	Streams data	Whole collection in memory	Streams data
Cancellation Support	No	No	Yes

**LIVE**

---

# Список материалов для изучения

<https://blog.stephencleary.com/2014/04/a-tour-of-task-part-0-overview.html>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-8.0>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task-1?view=net-8.0>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask?view=net-8.0>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1?view=net-8.0>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1.getawaiter?view=net-8.0>  
<https://referencesource.microsoft.com/#mscorlib/system/runtime/compiler/services/TaskAwaiter.cs,be57b6bc41e5c7e4>  
<https://vkontech.com/exploring-the-async-await-state-machine-the-awaitable-pattern/>  
<https://refactoring.guru/ru/design-patterns/state>  
<https://learn.microsoft.com/ru-ru/shows/on-dotnet/diagnosing-thread-pool-exhaustion-issues-in-net-core-apps>  
<https://metanit.com/sharp/tutorial/12.4.php>  
<https://devblogs.microsoft.com/dotnet/configureawait-faq/>  
<https://blog.stephencleary.com/2015/04/a-tour-of-task-part-10-promise-tasks.html>  
<https://devblogs.microsoft.com/pfxteam/executioncontext-vs-synchronizationcontext/>  
<https://learn.microsoft.com/en-us/dotnet/api/system.threading.executioncontext?view=net-5.0>  
<https://learn.microsoft.com/ru-ru/dotnet/api/system.threading.asynclocal-1?view=net-8.0>  
<https://codeblog.jonskeet.uk/>

**Подведём  
итоги занятия -  
рефлексия**

---

# Вопросы



если есть вопросы



если вопросов нет

---

# Заполните, пожалуйста, опрос о занятии

---

Мы читаем все ваши сообщения  
и берем их в работу 🖋️❤️

**You are our  
OTUS heroes**

