

# ValueTask. Что, зачем и почему

Егор Гришечко

Senior Developer, Insolar

@egor\_grishechko

[github.com/egorikas](https://github.com/egorikas)

# Мы с вами поговорим про:

- Какая проблема есть с Task
- Почему это проблема
- Вспомним механизм работы async
- Рассмотрим task-like типы
- Рассмотрим ValueTask
- Рассмотрим IValueTaskSource



# ~~ValueTask. Что, зачем и почему~~

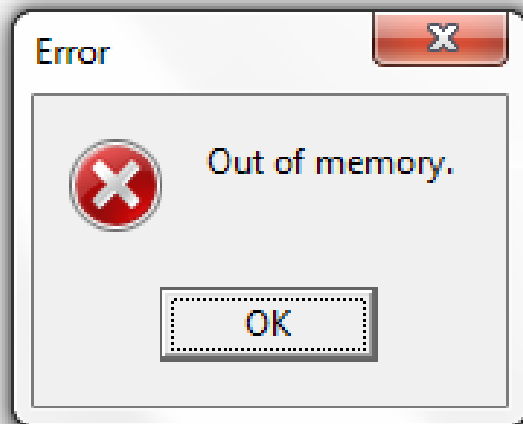
Егор Гришечко

Senior Developer, Insolar

@egor\_grishechko

[github.com/egorikas](https://github.com/egorikas)

# ОСТОРОЖНО



# ЗЛЫЕ

# АЛЛОКАЦИИ



**Ben Adams**

@ben\_a\_adams

Читаю



You don't need to worry about garbage collection; if you don't allocate!

🌐 Перевести твит



16:33 - 8 авг. 2017 г.

# C Task что-то не так?



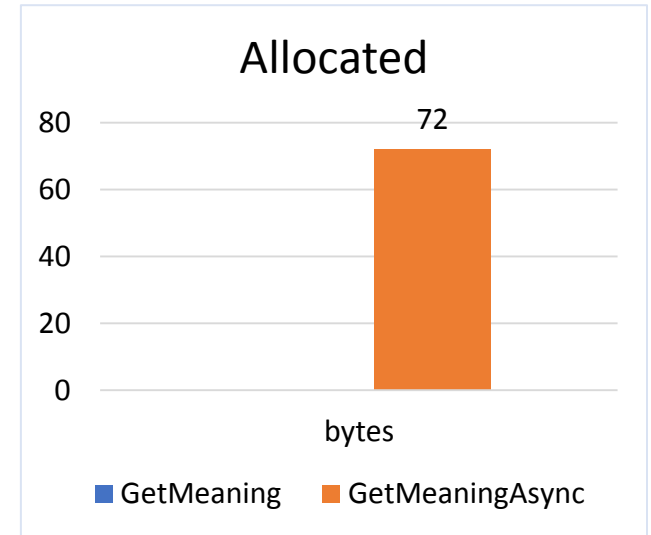
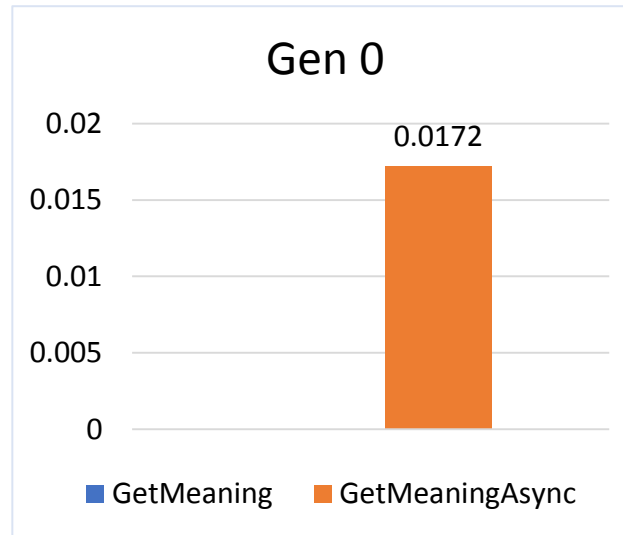
```
public class TheMeaningOfLife
{
    public int GetMeaning()
    {
        return 42;
    }

    public Task<int> GetMeaningAsync()
    {
        return Task.FromResult(42);
    }
}
```

# Побенчмаркаем?



GetMeaning  $\approx$  0.0259 ns



**GARBAGE COLLECTION**



# C Task что-то не так?



```
public class TheMeaningOfLife
{
    public int GetMeaning()
    {
        return 42;
    }

    public Task<int> GetMeaningAsync()
    {
        return Task.FromResult(42);
    }
}
```



# Hot path



Inside a program, the code paths that are executed most often belong to the **hot path** (with the remaining code paths being on the **cold path**). One can look at this at multiple levels: instruction, language statements, features. Over the years, multiple studies have shown how to optimize software for the hot path execution.

- **DO** avoid allocations in compiler hot paths:
  - Avoid LINQ.
  - Avoid using `foreach` over collections that do not have a `struct` enumerator.
  - Consider using an object pool. There are many usages of object pools in the compiler to see an example.

# C Task что-то не так?



```
public class BenchmarkImpl
{
    [Params(100, 1000, 100000)]
    public int Repeats { get; set; }

    [Benchmark]
    public void GetMeaning()
    {
        for(var i = 0; i < Repeats; i++)
            _meaning.GetMeaning();
    }

    [Benchmark]
    public void GetMeaningAsync()
    {
        for(var i = 0; i < Repeats; i++)
            _meaning.GetMeaningAsync().GetAwaiter().GetResult();
    }
}
```

# Использовали лишнюю память?

Метод	Повторения	Время	Gen 0	Allocated
GetMeaning	100	36.79 ns	-	0 B
GetMeaningAsync	100	688.94 ns	1.7157	7.2 Kb

Метод	Повторения	Время	Gen 0	Allocated
GetMeaning	100000	28,620.59 ns	-	0 B
GetMeaningAsync	100000	636,732.86 ns	1715.8203	7.2 Mb



# Использовали лишнюю память?



# Проблема? Проблема.

```
public interface IPriceProvider
{
    Task<decimal> GetByName(string name);
}

public class NetworkPizzaPriceProvider : IPriceProvider
{
    public Task<decimal> GetByName(string name)
    {
        return HttpClient.GetPriceFromNetwork(name);
    }
}

public class MemoryPizzaPriceProvider : IPriceProvider
{
    public Task<decimal> GetByName(string name)
    {
        return Task.FromResult<decimal>(42);
    }
}
```

# Проблема? Проблема.

```
public class PizzaPriceProvider
{
    public Dictionary<string, decimal> _cache = new Dictionary<string, decimal>();

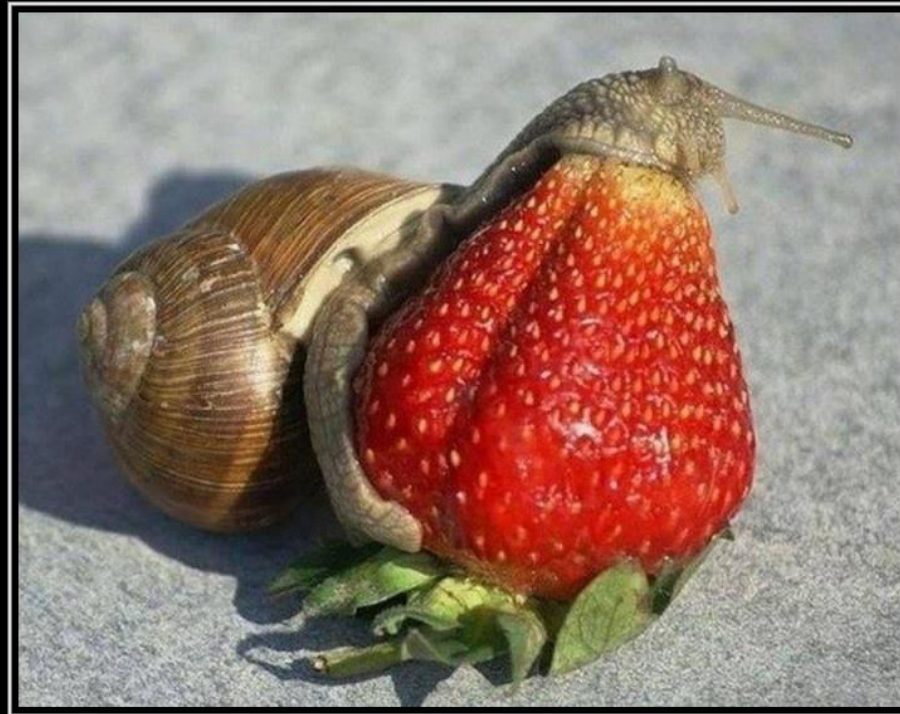
    private Task<decimal> GetPrice(string name) => HttpClient.GetPriceFromNetwork(name);

    public async Task<decimal> GetPizzaPriceAsync(string name)
    {
        if (_cache.TryGetValue(name, out var price))
        {
            // Возвращаем кэшированные цены
            return Task.FromResult(price);
        }

        return await GetPrice(name);
    }
}
```



А можно ли не выделять?

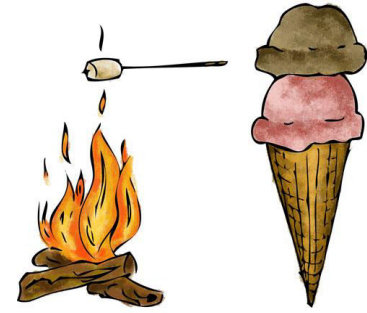


Если нельзя, но очень хочется  
то, можно

RusDemotivator.Ru



1. Не стоит выделять память в hot-path



2. Память может тратиться в неожиданных местах



3. Иногда она тратится там, где мы этого не хотим



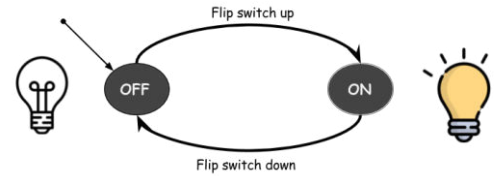
# Простой пример

```
public class TheMeaningOfLife
{
    public async Task<int> GetMeaning()
    {
        await Task.Delay(1);
        return 42;
    }
}
```

1. Task



2. StateMachine



3. AsyncTaskMethodBuilder



4. TaskAwaiter



5. MoveNextRunner

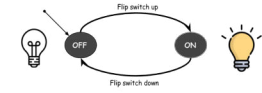


# Простой пример

```
public class TheMeaningOfLife
{
    public async Task<int> GetMeaning()
    {
        await Task.Delay(1);
        return 42;
    }
}
```



# GetMeaning метод

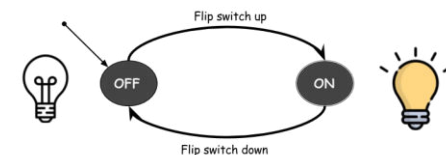


```
public class TheMeaningOfLife
{
    [AsyncStateMachine(typeof(GetMeaningStMch))]
    public Task<int> GetMeaning()
    {
        GetBookPrice stateMachine = default(GetMeaningStMch);
        stateMachine._this = this;
        stateMachine._builder = AsyncTaskMethodBuilder<int>.Create();
        stateMachine._state = -1;
        AsyncTaskMethodBuilder<decimal> _builder = stateMachine._builder;
        _builder.Start(ref stateMachine);
        return stateMachine._builder.Task;
    }
}
```

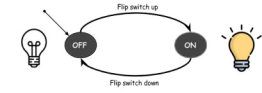
# StateMachine



```
[StructLayout(LayoutKind.Auto)]  
[CompilerGenerated]  
public struct GetMeaningStMch : IAsyncStateMachine  
{  
    public int _state;  
    public AsyncTaskMethodBuilder<int> _builder;  
    private TaskAwaiter _1;  
  
    private void MoveNext(){}  
    void IAsyncStateMachine.MoveNext(){}  
    private void SetStateMachine(IAsyncStateMachine stateMachine){}  
    void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine){}  
}
```



# GetMeaning метод



```
public class TheMeaningOfLife
{
    [AsyncStateMachine(typeof(GetMeaningStMch))]
    public Task<int> GetMeaning()
    {
        GetBookPrice stateMachine = default(GetMeaningStMch);
        stateMachine._this = this;
        stateMachine._builder = AsyncTaskMethodBuilder<int>.Create();
        stateMachine._state = -1;
        AsyncTaskMethodBuilder<decimal> _builder = stateMachine._builder;
        _builder.Start(ref stateMachine);
        return stateMachine._builder.Task;
    }
}
```

# AsyncTaskMethodBuilder ≈ TaskCompletionSource

```
public struct AsyncTaskMethodBuilder
{
    public Task Task { get; }

    public void AwaitOnCompleted(ref INotifyCompletion awaiter, ref IAsyncStateMachine stateMachine)
    public void AwaitUnsafeOnCompleted(
        ref ICriticalNotifyCompletion awaiter, ref IAsyncStateMachine stateMachine)

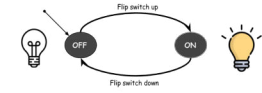
    public static AsyncTaskMethodBuilder Create()
    public void SetException(Exception exception)
    public void SetResult()
    public void SetStateMachine(IAsyncStateMachine stateMachine)

    public void Start(ref IAsyncStateMachine stateMachine)
}
```





# GetMeaning метод



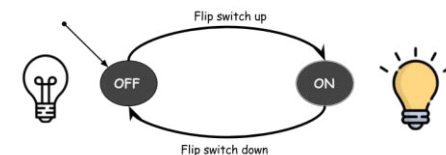
```
public class TheMeaningOfLife
{
    [AsyncStateMachine(typeof(GetMeaningStMch))]
    public Task<int> GetMeaning()
    {
        GetBookPrice stateMachine = default(GetMeaningStMch);
        stateMachine._this = this;
        stateMachine._builder = AsyncTaskMethodBuilder<int>.Create();
        stateMachine._state = -1;
        AsyncTaskMethodBuilder<decimal> _builder = stateMachine._builder;
        _builder.Start(ref stateMachine);
        return stateMachine._builder.Task;
    }
}
```

```

private void MoveNext()
{
    TaskAwaiter awaiter;
    if (state == "Первый запуск")
    {
        awaiter = Task.Delay(1).GetAwaiter();
        if (!awaiter.IsCompleted)
        {
            _1 = awaiter;
            _builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
            return;
        }
    }
    else
    {
        awaiter = _1;
        _1 = default(TaskAwaiter);
    }
    awaiter.GetResult();
    result = 42;

    _builder.SetResult(result);
}

```



Why would you want a custom awaiter?

11

You can see the compiler's interpretation of `await` [here](#). Essentially:

```
var temp = e.GetAsync();
if (!temp.IsCompleted)
{
    SAVE_STATE()
    temp.OnCompleted(&cont);
    return;

cont:
    RESTORE_STATE()
}
var i = temp.GetResult();
```

**Edit from comments:** `OnCompleted` should schedule its argument as a continuation of the asynchronous operation.

share edit flag

edited Sep 30 '12 at 13:21

answered Sep 30 '12 at 13:00



Stephen Cleary

269k ● 45 ● 452 ● 571

# Awaiter



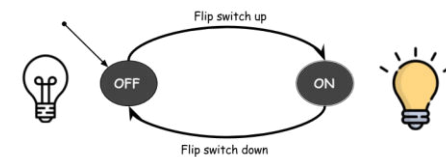
```
public struct HopToThreadPoolAwaitable : INotifyCompletion
{
    public HopToThreadPoolAwaitable GetAwaiter() => this;
    public bool IsCompleted => false;

    public void OnCompleted(Action continuation) => Task.Run(continuation);
    public void GetResult() { }
}

class Program
{
    static async Task Main(string[] args)
    {
        Console.WriteLine(Thread.CurrentThread.ManagedThreadId); // 1
        await default(HopToThreadPoolAwaitable);
        Console.WriteLine(Thread.CurrentThread.ManagedThreadId); // 3
    }
}
```

```
private void MoveNext()
{
    TaskAwaiter awaiter;
    if (state == "Первый запуск")
    {
        awaiter = Task.Delay(1).GetAwaiter();
        if (!awaiter.IsCompleted)
        {
            _1 = awaiter;
            _builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
            return;
        }
    }
    else
    {
        awaiter = _1;
        _1 = default(TaskAwaiter);
    }
    awaiter.GetResult();
    result = 42;

    _builder.SetResult(result);
}
```



# MoveNextRunner



```
internal sealed class MoveNextRunner
{
    private readonly ExecutionContext m_context;
    internal IAsyncStateMachine m_stateMachine;

    internal void Run()

    /// <summary>Invokes the MoveNext method on the supplied IAsyncStateMachine.</summary>
    /// <param name="stateMachine">The IAsyncStateMachine machine instance.</param>
    private static void InvokeMoveNext(object stateMachine)
}
```

Где ValueTask?



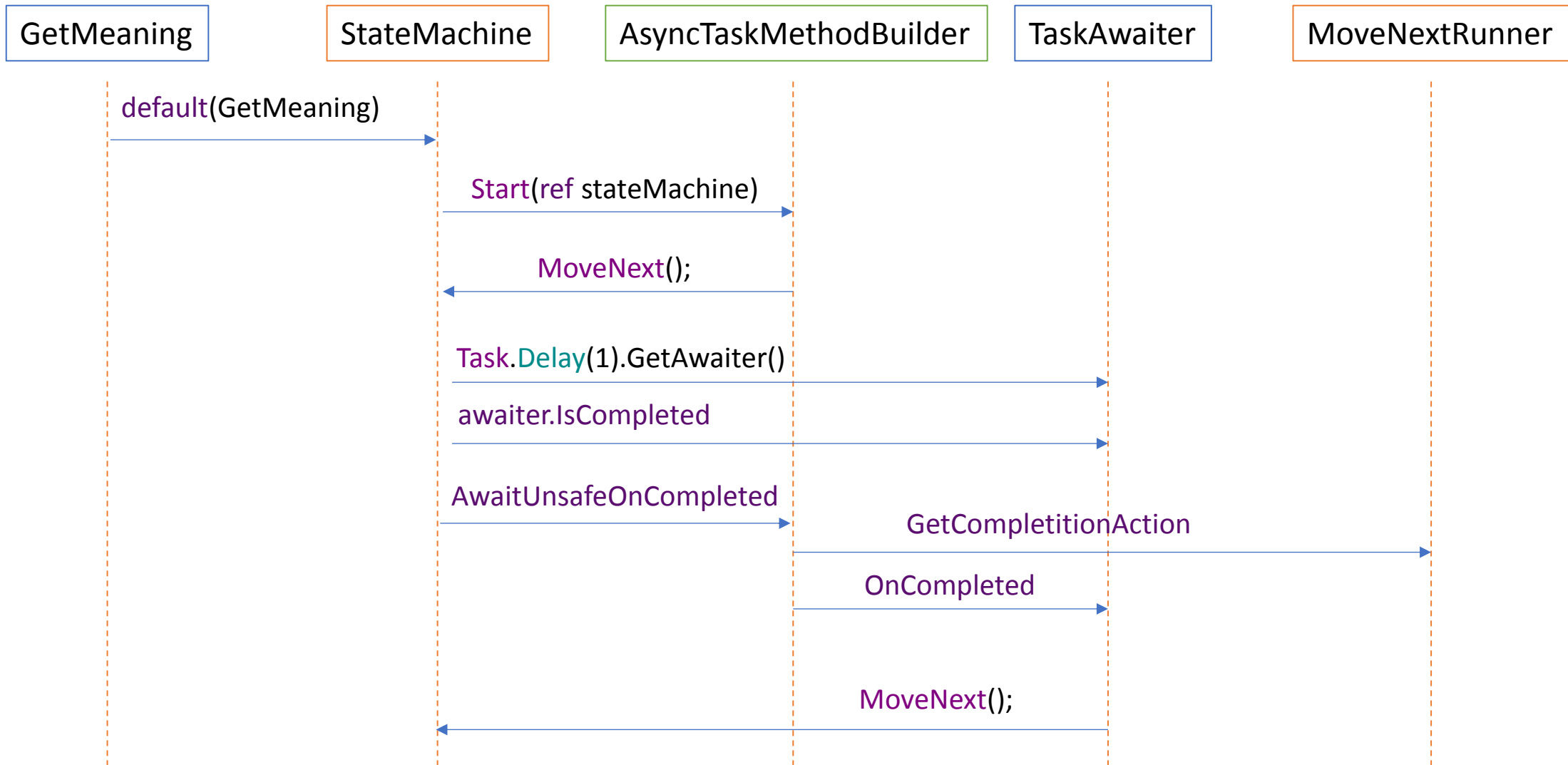


# Простой пример

```
public class TheMeaningOfLife
{
    public async Task<int> GetMeaning()
    {
        await Task.Delay(1);
        return 42;
    }
}
```



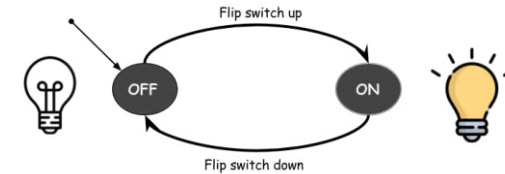
# Схема взаимодействия



1. **Task** – контейнер для задачи



2. **StateMachine** – промежуточный слой содержащий логику метода



3. **AsyncTaskMethodBuilder** – контроллер хода работы асинхронного метода



4. **TaskAwaiter** – принимает коллбэк и запускает его



5. **MoveNextRunner** – оборачивает коллбэк в мета-информацию



# Task-like типы

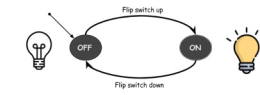
## Async Return Types (C#)

📅 05/29/2017 • ⌚ 7 minutes to read • Contributors  all

Async methods can have the following return types:

- [Task<TResult>](#), for an async method that returns a value.
- [Task](#), for an async method that performs an operation but returns no value.
- `void`, for an event handler.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The object returned by the `GetAwaiter` method must implement the [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.

# GetMeaning метод



```
public class TheMeaningOfLife
{
    [AsyncStateMachine(typeof(GetMeaning))]
    public Task<int> GetMeaning()
    {
        GetBookPrice stateMachine = default(GetMeaning);
        stateMachine._this = this;
        stateMachine._builder = AsyncTaskMethodBuilder<int>.Create();
        stateMachine._state = -1;
        AsyncTaskMethodBuilder<decimal> _builder = stateMachine._builder;
        _builder.Start(ref stateMachine);
        return stateMachine._builder.Task;
    }
}
```

# AsyncTaskMethodBuilder ≈ TaskCompletionSource

```
public struct AsyncTaskMethodBuilder
{
    public Task Task { get; }

    public void AwaitOnCompleted(ref INotifyCompletion awaiter, ref IAsyncStateMachine stateMachine)
    public void AwaitUnsafeOnCompleted(
        ref ICriticalNotifyCompletion awaiter, ref IAsyncStateMachine stateMachine)

    public static AsyncTaskMethodBuilder Create()
    public void SetException(Exception exception)
    public void SetResult()
    public void SetStateMachine(IAsyncStateMachine stateMachine)

    public void Start(ref IAsyncStateMachine stateMachine)
}
```




# Builder и его экземпляры

1. [AsyncVoidMethodBuilder](#)
2. [AsyncTaskMethodBuilder](#)
3. [AsyncTaskMethodBuilder<TResult>](#)



# Task-like типы

## Async Return Types (C#)

📅 05/29/2017 • ⌚ 7 minutes to read • Contributors  all

Async methods can have the following return types:

- [Task<TResult>](#), for an async method that returns a value.
- [Task](#), for an async method that performs an operation but returns no value.
- `void`, for an event handler.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The object returned by the `GetAwaiter` method must implement the [System.Runtime.CompilerServices.ICriticalNotifyCompletion](#) interface.





**Ben Adams**

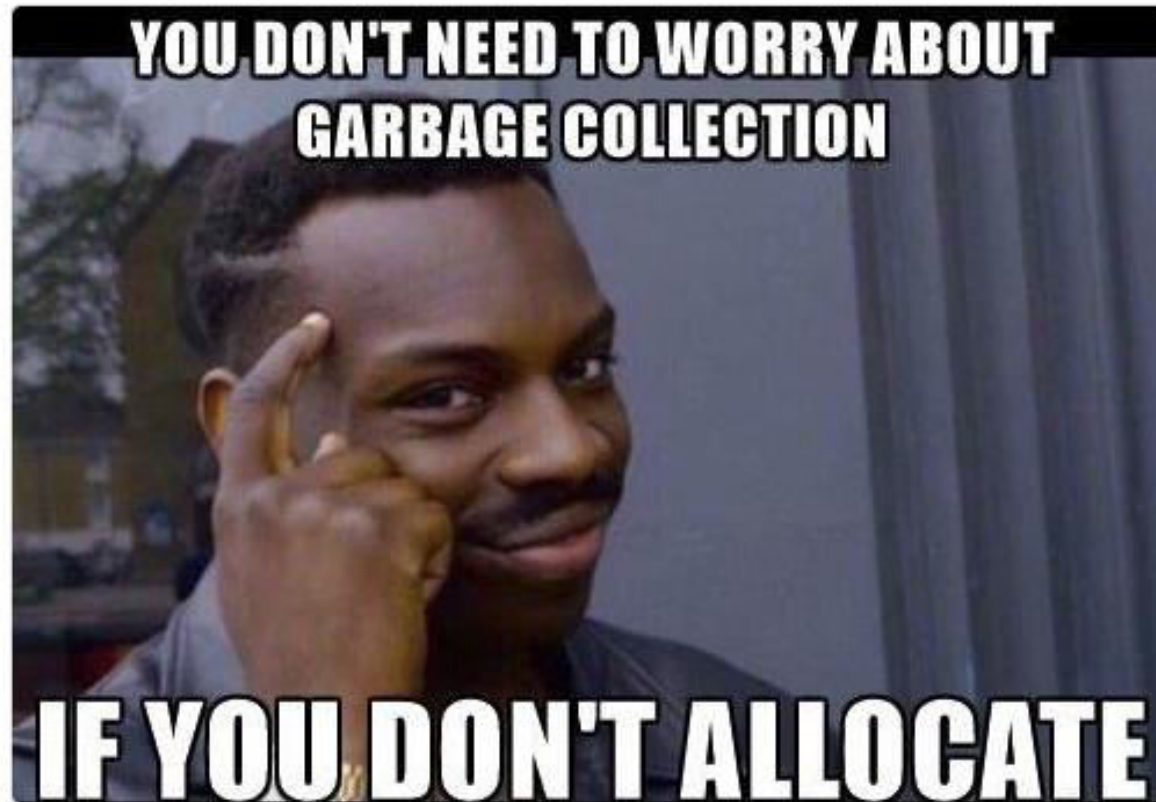
@ben\_a\_adams

Читаю



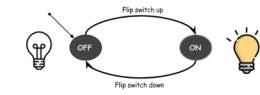
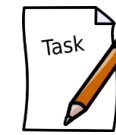
You don't need to worry about garbage collection; if you don't allocate!

🌐 Перевести твит



16:33 - 8 авг. 2017 г.

# GetMeaning метод



```
public class TheMeaningOfLife
{
    [AsyncStateMachine(typeof(GetMeaning))]
    public Task<int> GetMeaning()
    {
        GetBookPrice stateMachine = default(GetMeaning);
        stateMachine._this = this;
        stateMachine._builder = AsyncTaskMethodBuilder<int>.Create();
        stateMachine._state = -1;
        AsyncTaskMethodBuilder<decimal> _builder = stateMachine._builder;
        _builder.Start(ref stateMachine);
        return stateMachine._builder.Task;
    }
}
```

Custom TaskBuilder  
=== task-like



# Проблема? Проблема.

```
public class PizzaPriceProvider
{
    public Dictionary<string, decimal> _cache = new Dictionary<string, decimal>();

    private Task<decimal> GetPrice(string name) => HttpClient.GetPriceFromNetwork(name);

    public async Task<decimal> GetPizzaPriceAsync(string name)
    {
        if (_cache.TryGetValue(name, out var price))
        {
            // Возвращаем кэшированные цены
            return Task.FromResult(price);
        }

        return await GetPrice(name);
    }
}
```

# ValueTask

```
[AsyncMethodBuilder(typeof (AsyncValueTaskMethodBuilder<>))]
public readonly struct ValueTask<TResult>
{
    public bool IsCompleted { get; }

    public TResult Result { get; }

    public Task<TResult> AsTask();

    public ConfiguredValueTaskAwaitable<TResult> ConfigureAwait(bool continueOnCapturedContext);

    public ValueTaskAwaiter<TResult> GetAwaiter();
}
```

# PizzaPriceProvider + ValueTask = ❤️👉

```
public class PizzaPriceProvider
{
    public Dictionary<string, decimal> _cache = new Dictionary<string, decimal>();

    private Task<decimal> GetPrice(string name) => ...

    public ValueTask<decimal> GetPizzaPriceValueTask(string name)
    {
        if (_cache.TryGetValue(name, out var price))
        {
            // Возвращаем кэшированные цены
            return new ValueTask<decimal>(price);
        }

        return new ValueTask<decimal>(GetPrice(name));
    }
}
```

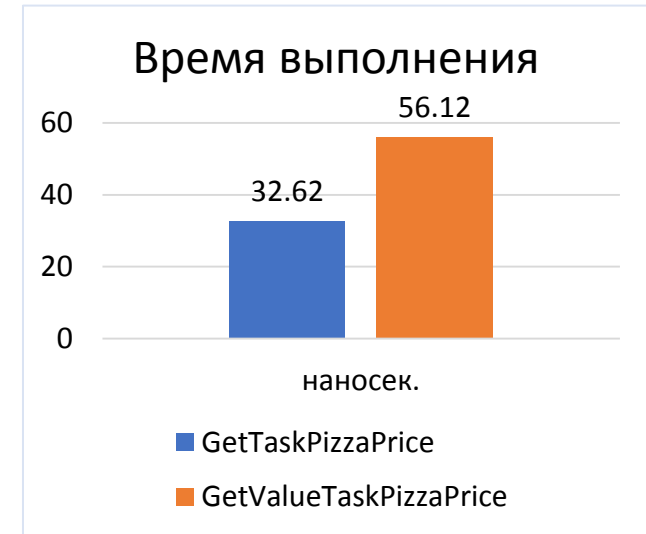
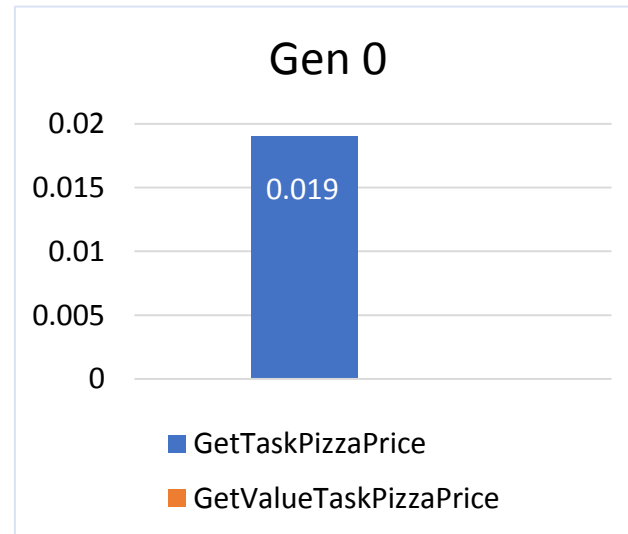
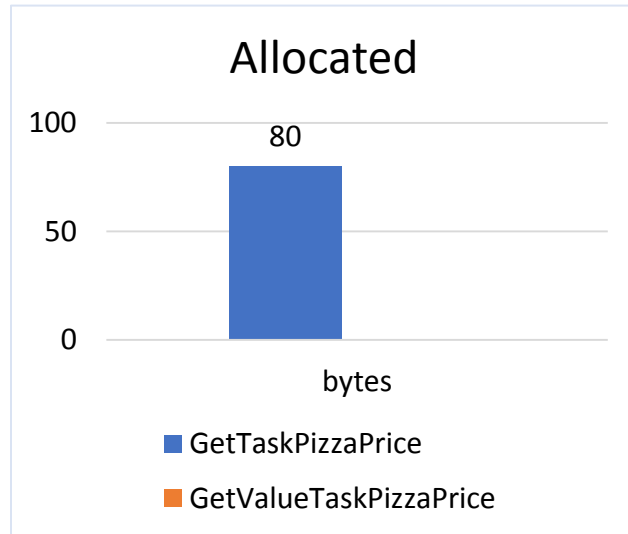
1. **async** – не только Task, Task<T> и void

2. **task-like** – это типы с кастомным taskbuilder

3. **ValueTask** – реализация task-like типа без выделения памяти в heap



# PizzaPriceProvider + ValueTask = ❤️👉

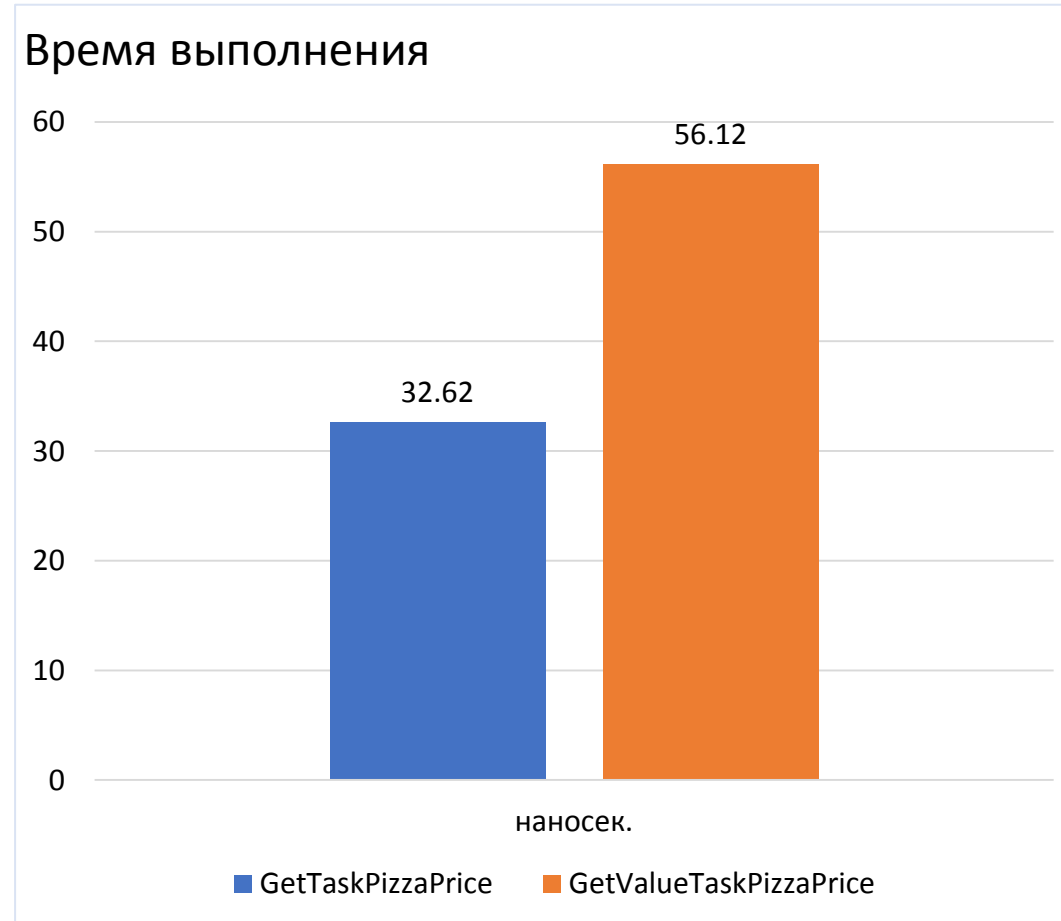


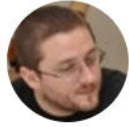
## GARBAGE COLLECTION





# Почему медленнее?





**Marc Gravell**

@marcgravell

Follow



Very confused by some `ValueTask<T>` vs `Task<T>` performance metrics - is it me?  
[gist.github.com/mgravell/c570e ...](https://gist.github.com/mgravell/c570e...) /cc  
[@ben\\_a\\_adams](#) [@jonskeet](#) [@davidfowl](#)



**async; ValueTask vs Task**

async; ValueTask vs Task. GitHub Gist: instantly share code, notes, and snippets.

[gist.github.com](https://gist.github.com)

11:34 AM - 16 Apr 2017

6 Retweets 26 Likes



7




6



26





**NDC { Oslo }**

**3**

## C# 7: Performance Summary

- Value Types have better performance characteristics:
  - Data Locality
  - No GC
- Value Tuples offer clean coding and great performance
- Safe references can make your code faster than unsafe!
  - Use them only when needed to keep your code clean

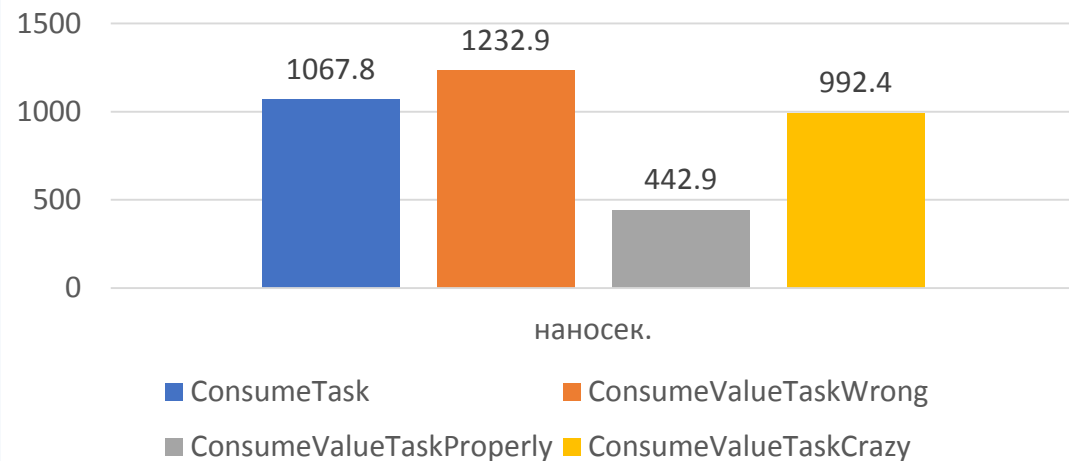
19

<https://www.youtube.com/watch?v=CSPSvBeqJ9c>

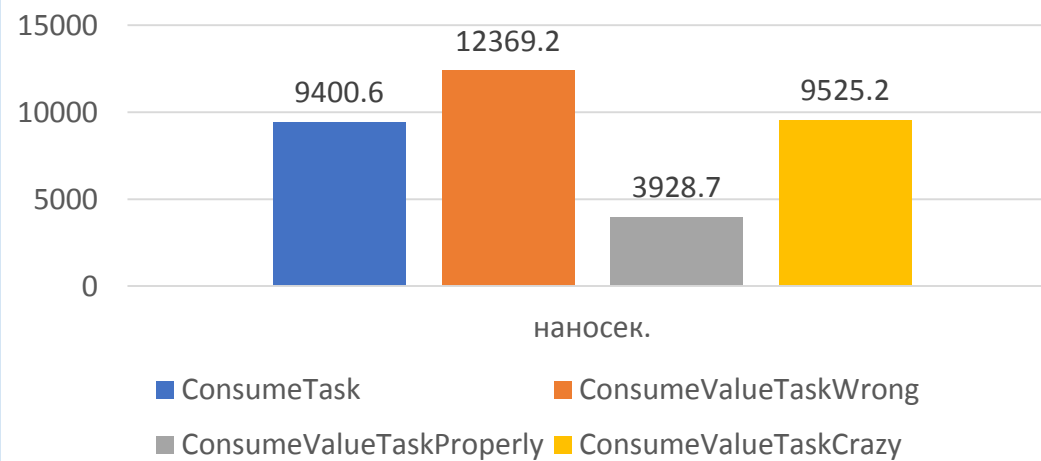
<https://github.com/adamsitnik/StateOfTheDotNetPerformance>



### Время выполнения – 100 ит.



### Время выполнения – 1000 ит.



# Why would one use Task<T> over ValueTask<T> in C#?

As such, the default choice for any asynchronous method should be to return a `Task` or `Task<TResult>`. Only if performance analysis proves it worthwhile should a `ValueTask<TResult>` be used instead of `Task<TResult>`.

share edit flag

answered Mar 24 '17 at 15:44



**Stephen Cleary**

266k ● 45 ● 445 ● 558

Can someone explain when ValueTask would fail to do the job?

The purpose of the thing is improved performance. It doesn't do the job if it doesn't *measurably* and *significantly* improve performance. There is no guarantee that it will.

share edit flag

edited Mar 24 '17 at 18:41

answered Mar 24 '17 at 16:25



**Eric Lippert**

519k ● 142 ● 1029 ●

1896

# Add ValueTask to corefx



stephentoub commented on 30 Nov 2015

Member



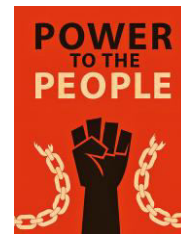
Are there plans to let `async` be used with this?

No concrete plans, but it's being discussed (cc: @jaredpar, @MadsTorgersen). There is value in doing so. However, using a `ValueTask<TResult>` instead of a `Task<TResult>` isn't a pure win. There is overhead associated with it (for example, it's a field bigger, which means you're not only passing back an extra field from a method, if you're awaiting one of these it's likely going to increase the size of an async caller's state machine object by the size of a field), and that's pure overhead without benefit in the case where the operation does complete asynchronously. That means developers really need to

```
private async ValueTask<Something> BarAsynccore(string arg) { ... }
```

There obviously is. I'm simply highlighting it to point out that `ValueTask<TResult>` should only be used after careful consideration, and with such careful consideration, you're likely also considering other optimizations that might lead you away from this. And of course there's cost involved in developing such a feature to the language, in developers needing to learn about it, etc.

1. Теперь можно **влиять** на `async/await`



2. Для этого мы можем создать свои **AsyncЧтоУгодноBuilder**



3. **ValueTask** – пример такого **AsyncЧтоУгодноBuilder**



4. Помогает убрать лишние **аллокации** в hot paths



5. **ValueTask** необходимо **применять** с умом и после измерений





# А что, если?



```
public class WhatIf
{
    public ValueTask<int> DoSomething()
    {
        if(можетБытьАсинхронным)
            return new ValueTask<int>(42);

        // Хочу, чтобы здесь не было аллокаций
        return new ValueTask<int>(Task.FromResult(42));
    }
}
```



# IValueTaskSource



```
public readonly struct ValueTask<TResult>
{
    public ValueTask(IValueTaskSource<TResult> source, short token);

    public ValueTask(Task<TResult> task);

    public ValueTask(TResult result);
}
```



# IValueTaskSource



```
public interface IValueTaskSource<out TResult>
{
    TResult GetResult(short token);

    ValueTaskSourceStatus GetStatus(short token);

    void OnCompleted(
        Action<object> continuation,
        object state,
        short token,
        ValueTaskSourceOnCompletedFlags flags);
}
```

# А что, если?



```
public class WhatIfV2
{
    public ValueTask<int> DoSomething()
    {
        if(можетБытьАсинхронным)
            return new ValueTask<int>(42);

        // Используем пул
        IValueTaskSource<int> vts = ...;
        return new ValueTask<int>(vts);
    }
}
```

# IValueTaskSource

Most developers should never have a need to see this interface: methods simply hand back a `ValueTask<TResult>` that may have been constructed to wrap an instance of this interface, and the consumer is none-the-wiser. The interface is primarily there so that developers of performance-focused APIs are able to avoid allocation.



# Как это работает?

Most developers **should never need to implement these interfaces**. They're also **not** particularly **easy** to implement. If you decide you need to, there are several implementations internal to .NET Core 2.1 that can serve as a reference, e.g.



```

/// <summary>A SocketAsyncEventArgs that can be awaited to get the result of an operation.</summary>
internal sealed class AwaitableSocketAsyncEventArgs : SocketAsyncEventArgs, IValueTaskSource, IValueTaskSource<int>
{
    internal static readonly AwaitableSocketAsyncEventArgs Reserved = new AwaitableSocketAsyncEventArgs() { _continuation = null };
    /// <summary>Sentinel object used to indicate that the operation has completed prior to OnCompleted being called.</summary>
    private static readonly Action<object> s_completedSentinel = new Action<object>(state => throw new Exception(nameof(s_completedSentinel)));
    /// <summary>Sentinel object used to indicate that the instance is available for use.</summary>
    private static readonly Action<object> s_availableSentinel = new Action<object>(state => throw new Exception(nameof(s_availableSentinel)));
    /// <summary>
    /// <see cref="s_availableSentinel"/> if the object is available for use, after GetResult has been called on a previous use.
    /// null if the operation has not completed.
    /// <see cref="s_completedSentinel"/> if it has completed.
    /// Another delegate if OnCompleted was called before the operation could complete, in which case it's the delegate to invoke
    /// when the operation does complete.
    /// </summary>
    private Action<object> _continuation = s_availableSentinel;
    private ExecutionContext _executionContext;
    private object _scheduler;
    /// <summary>Current token value given to a ValueTask and then verified against the value it passes back to us.</summary>
    /// <remarks>
    /// This is not meant to be a completely reliable mechanism, doesn't require additional synchronization, etc.
    /// It's purely a best effort attempt to catch misuse, including awaiting for a value task twice and after
    /// it's already being reused by someone else.
    /// </remarks>
    private short _token;
}

```



```

/// <summary>The representation of an asynchronous operation that has a result value.</summary>
/// <typeparam name="TResult">Specifies the type of the result. May be <see cref="VoidResult"/>.</typeparam>
internal partial class AsyncOperation<TResult> : AsyncOperation, IValueTaskSource, IValueTaskSource<TResult>
{
    /// <summary>Registration with a provided cancellation token.</summary>
    private readonly CancellationTokenRegistration _registration;
    /// <summary>>true if this object is pooled and reused; otherwise, false.</summary>
    /// <remarks>
    /// If the operation is cancelable, then it can't be pooled. And if it's poolable, there must never be race conditions
    /// which is the main reason poolable objects can't be cancelable, as then cancellation could fire, the object could
    /// and then we may end up trying to complete an object that's used by someone else.
    /// </remarks>
    private readonly bool _pooled;
    /// <summary>Whether continuations should be forced to run asynchronously.</summary>
    private readonly bool _runContinuationsAsynchronously;
}

```

```
/// <summary>
/// Default <see cref="PipeWriter"/> and <see cref="PipeReader"/> implementation.
/// </summary>
public sealed partial class Pipe
{
    private sealed class DefaultPipeReader : PipeReader, IValueTaskSource<ReadResult>
    {
        private readonly Pipe _pipe;

        public DefaultPipeReader(Pipe pipe)
        {
            _pipe = pipe;
        }

        public override bool TryRead(out ReadResult result) => _pipe.TryRead(out result);
    }
}
```



# Как начать этим всем пользоваться?

	<b>.NET Core</b>	<b>.NET FX</b>
Task-like types	C# 7.0	C# 7.0
ValueTask	1.0	<a href="#"><u>System.Threading.Tasks.Extensions</u></a>
IValueTaskSource	2.1	<a href="#"><u>System.IO.Pipelines</u></a>

# ССЫЛКИ

1. <https://blogs.msdn.microsoft.com/dotnet/2018/11/07/understanding-the-whys-whats-and-whens-of-valuetask/> - кратко и про все
2. <https://blog.scooletz.com/2018/05/14/task-async-await-valuetask-ivaluetasksource-and-how-to-keep-your-sanity-in-modern-net-world/> - еще раз кратко и обо всем
3. <http://blog.i3arnon.com/2015/11/30/valuetask/> - одно из первых упоминаний
4. <http://tooslowexception.com/implementing-custom-ivaluetasksource-async-without-allocations/> - IValueTaskSource
5. <https://blogs.msdn.microsoft.com/setepia/2017/11/30/dissecting-the-async-methods-in-c/> - классная серия про async
6. <https://github.com/adamsitnik/StateOfTheDotNetPerformance> - StateOfTheDotNetPerformance
7. <https://github.com/dotnet/roslyn/blob/master/docs/features/task-types.md> - дока из Roslyn
8. <https://github.com/dotnet/corefx/issues/4708#issuecomment-160658188> – issue в corefx
9. <https://stackoverflow.com/questions/43000520/why-would-one-use-taskt-over-valuetaskt-in-c> - Why would one use Task<T> over ValueTask<T> in C#?



**Ben Adams**

@ben\_a\_adams

Читаю



You don't need to worry about garbage collection; if you don't allocate!

🌐 Перевести твит



16:33 - 8 авг. 2017 г.