

Deep Dive async/await in Unity with UniTask

About the Speaker

Yoshifumi Kawai / @neuecc

Cysharp, Inc. - Founder/CEO/CTO

After working as CTO for Grani, Inc., he worked with Cygames, Inc. to establish the company.

As a research and development company specializing in C#,

Cysharp, Inc. uses C# in server (.NET Core) / client (Unity) implementation.

Mainly developing MagicOnion – Unified Realtime/API Engine for .NET Core and Unity.

<https://github.com/Cysharp/MagicOnion>

He awarded Microsoft MVP for Developer Technologies(C#)

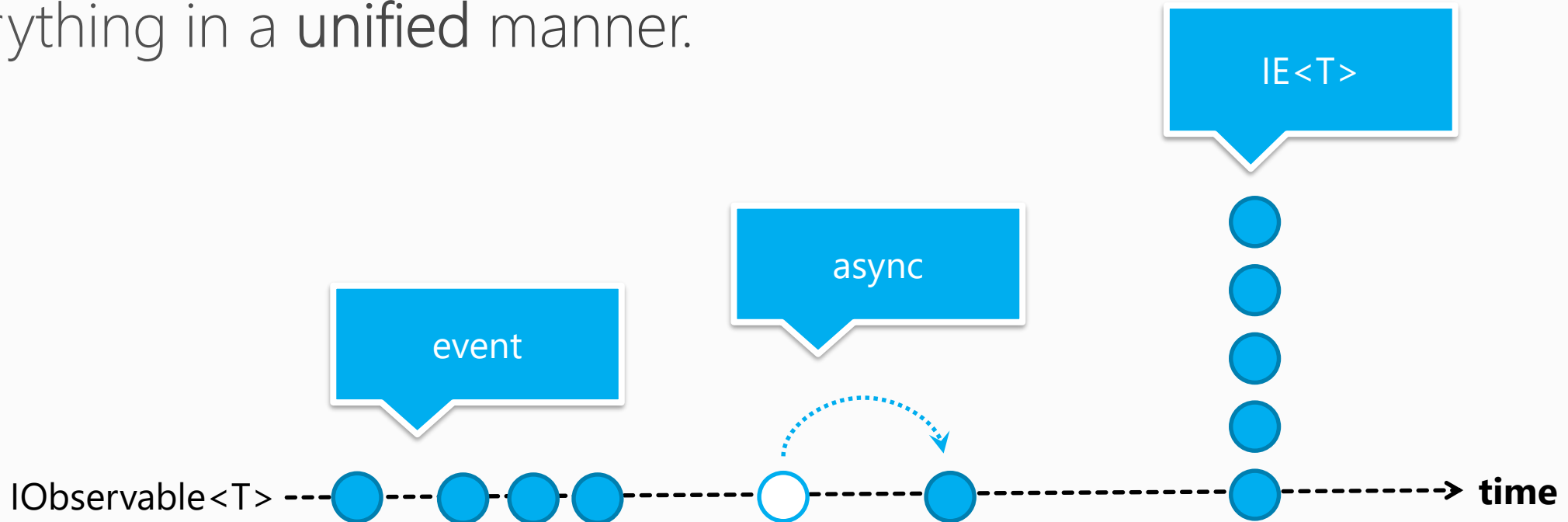
Developed a library of over 40 pieces of OSS (UniRx, MessagePack for C#, Utf8Json, etc...)

Rx vs Async/Await

LINQ to Asynchronous

Can Rx be used with asynchronous operations?

One of its strong points is, by putting time on an axis and making comparisons with collection operations, it's possible to handle everything in a **unified** manner.



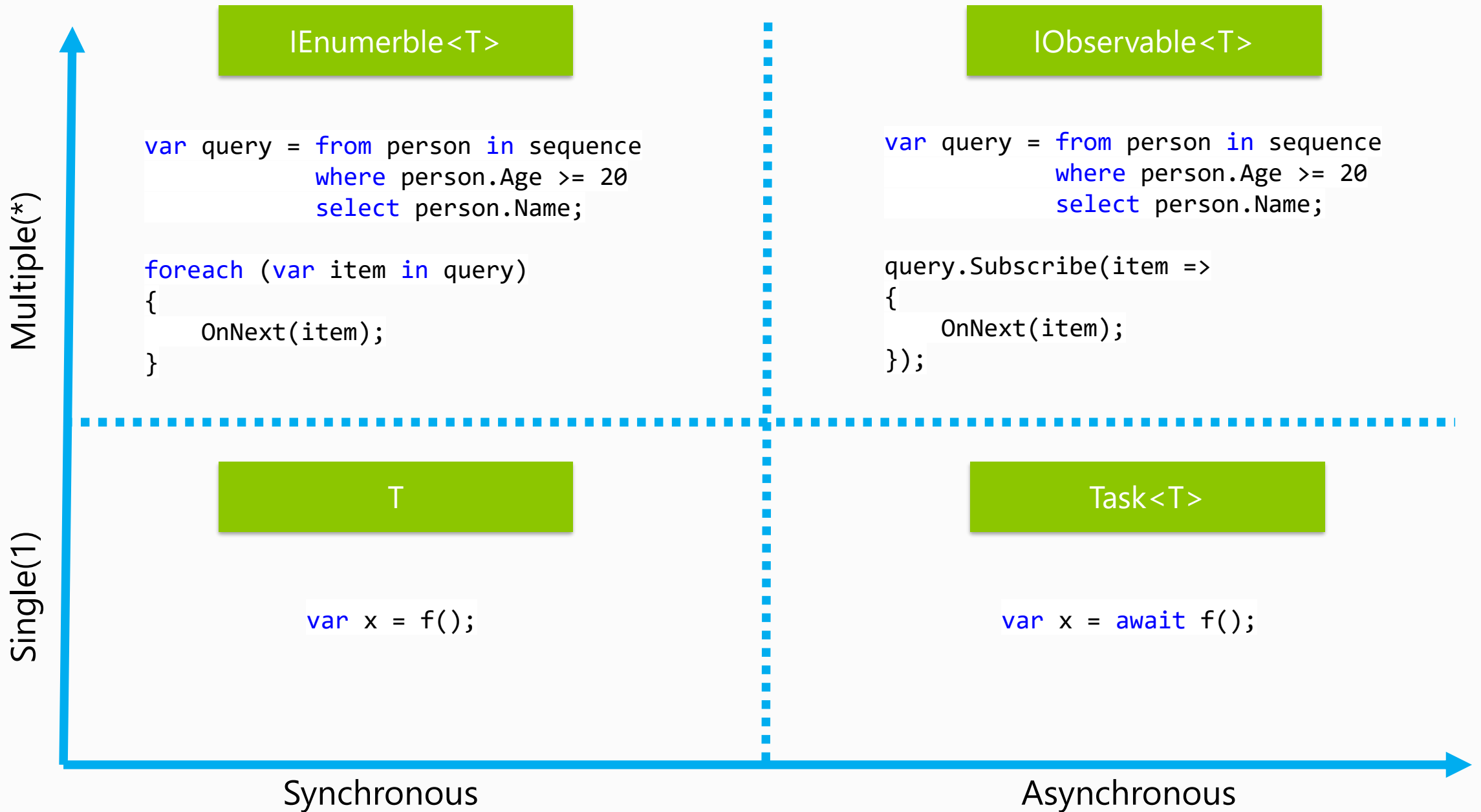
async/await

A structure which treats asynchronous as synchronous

With synchronous processing, which includes control flow, it is possible to write with almost no changes

```
static string GetSync(int page)
{
    try
    {
        var url = "http://...?page=" + page;
        var html = GetHttpStringSync(url);
        return html;
    }
    catch
    {
        return "Error";
    }
}
```

```
static async Task<string> GetAsync(int page)
{
    try
    {
        var url = "http://...?page=" + page;
        var html = await GetHttpStringAsync(url);
        return html;
    }
    catch
    {
        return "Error";
    }
}
```



Rx Code Smell

The ability to display everything with IObservable<T>

It is a strength and a weakness at the same time.

There's no way to tell if it's an event (length ∞) or asynchronous (length 1).

Although it's possible to apply the same process with the same operator and both can be combined without distinction,

since the way each will be handled upon application is completely different (resulting in bugs), it's better **not to handle it in a unified manner** (being able to make conversions when needed is sufficient).

That is why the forms Single (corresponding to Task<T>) and Completable (corresponding to Task) have been added to RxJava.

Rx Spaghetti Control Flow

It actually isn't suitable for complex control

While a complex control can be written simply, it can easily turn into a puzzle. As with things like Repeat and changing to a different flow with exception handling, complexity increases when the stream doesn't flow in a single direction.

Familiar control syntaxes such as "for" and "try-catch" can be used.

"Async/await" can be written in a way that is far more clear and easier to read.

The ability to write everything in "functional" is not necessarily good.

The ease of understanding found in procedural description should be re-examined.

Rx Spaghetti Control Flow

It actually isn't suitable for complex control

While a complex control can be written simply, it can easily turn into a puzzle. As with things like Repeat, it can lead to a different flow from exception handling. complexity is

Familiar control
"Async/await"
The ability to
The ease of

For cases in which it becomes necessary (control flow exceeds a single-round method execution procedure) to have field variables used for meeting when writing synchronously "while maintaining a single direction," such as in adding buffering or in merging flows in time, Rx can write things in an easy to understand manner. Since it is still true that "a complex control can be written in a simple, easy to understand manner," we need to use it better, taking care so it doesn't turn into a puzzle (Rx can also be seen as a powerful medicine).

Conclusion

Async/await for asynchronous, Rx for event

Even Erik Meijer (the original proposer of Rx) says so.

In recent years, async/await has been introduced in Python, Kotlin, JavaScript, Dart, Swift... and at the same time, the opinion of "Rx isn't needed" has arisen. And that is correct.

Asynchronous should stick with async/await. Survival is found in event stream usage (of which ReactiveProperty is also a part).



Erik Meijer
@headinthebox

フォロー中

[proandroiddev.com/forget-rxjava- ...](https://proandroiddev.com/forget-rxjava-...)

Absolutely, RX was designed to make *event streams* into first-class values.

For async programming, or anything that requires back pressure, Rx is not the right tool.

#tautology

🌐 ツイートを翻訳

Forget RxJava: Kotlin Coroutines are all you need

Working on Android? RxJava isn't the tool to reach for. It's a powerful library, but not made to manage async work. **Vladimir Ivanov** shows why, and then demonstrates [how Kotlin Coroutines can make your life better](#). This is part one of a two-part series.

3:05 - 2018年8月27日

What is `async/await`?

async/await: Multithread...

it is not!!!

async/await Is Not Multithread

Asynchronous is not multithread

This must be persistently said over and over again.

It is only correct that there are also times when it becomes multithread.

Coroutines aren't multithread, are they?

JavaScript isn't multithread, is it?

But task is multithread, right?

Yes and no.

Task is something that was originally multithread, and, since that is what is being recycled, there are often cases in which it is behaviorally multithread, making it easy for misunderstandings to form.

The History of Task and async/await

Created together with Parallel as something to easily handle multithreading.

Opposition to reference-type Task in asynchronous that ends in synchronous due to Task's heavy use and infectiousness. Creation of value-type ValueTask along with task-like language functions which make it possible to apply return values other than Task in async.

.NET 4.0 Task

C# 5.0 async/await
(.NET 4.5 Task)

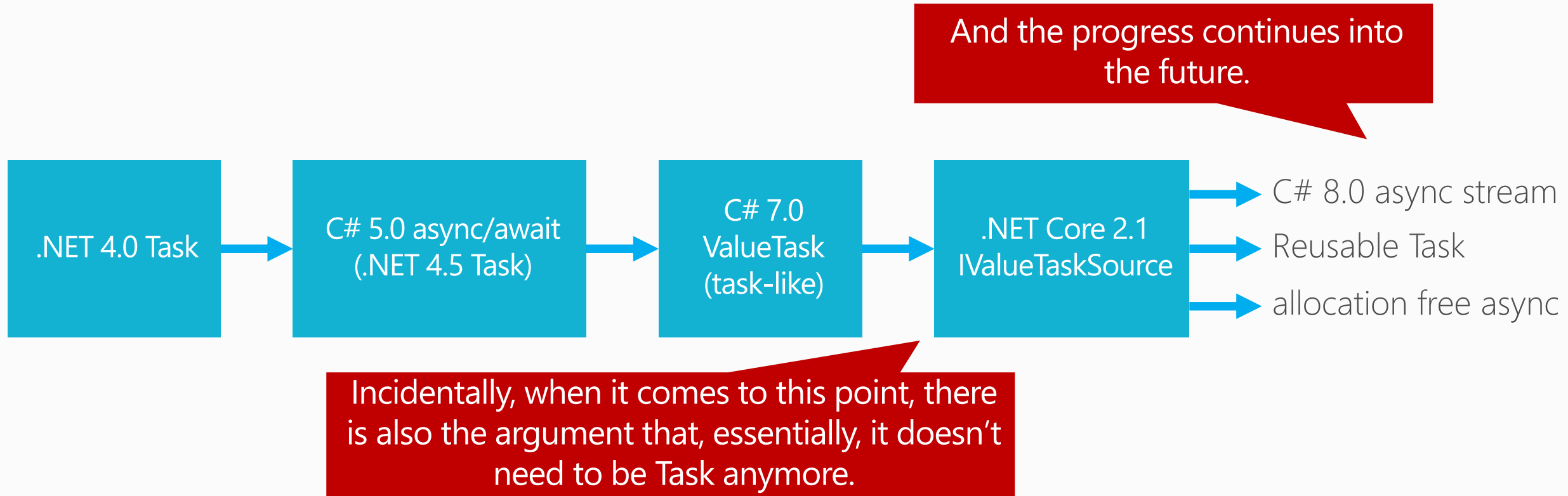
C# 7.0
ValueTask
(task-like)

.NET Core 2.1
IValueTaskSource

Creation of async/await.
Improvement of Task inner structure for async/await.
(Somewhat chaotic as it was retrofitted.)

Through addition of a mechanism based on ValueTask to bypass Task, the heavy-use heap maintenance of await is at a minimum.

The History of Task and async/await



Is Task Essential to async/Await in the First Place?

2012 for C#'s async/await.

It is a pioneer which has been implemented more than any other language.

Absolutely Not

Around when async/await was being implemented in C# 5.0(.NET 4.5), it was faster to recycle the already-existing structure (.NET 4.0 Task). There are things that aren't known until something is widely used.

There are also consequences and liabilities

In investigation of the asynchronous model of Midori (Microsoft's managed OS project using C# style language), in regard to Task, the performance side (in making something severe like an OS) was especially a let down.

<http://joeduffyblog.com/2015/11/19/asynchronous-everything/>

For C# as it is now, that's being repaid with the unique evolution of ValueTask.

Is Task Essential to async/await in the First Place?

Absolutely Not

Around when
it was faster
There are thi

There are

In investigati
managed OS project
performance side (i
a let down.

<http://joeduff.com/2015/11/19/asynchronous-everything/>

For C# as it is now, that's being repaid with the unique evolution of ValueTask.

UniTask is Task that has been optimized for Unity use through referencing ValueTask from NET Core 2.1. It is a project for bringing async/await into Unity in an ideal form, doing away with all the liability from the start.

How `async/await` Works

The true nature of async/await is CPS (Continuation Passing Style) conversion.

It cuts in as a continuation (Action continuation) after await.

```
async Task<string> SampleTextLoadAsync()  
{  
    Debug.Log("Before LoadAsync:" +Time.frameCount); // frame:1  
    var textAsset = await Resources.LoadAsync<TextAsset>("te") as TextAsset;  
    Debug.Log("After LoadAsync:" +Time.frameCount); // frame:2  
    return textAsset.text;  
}
```

When the right side of await (awaitable) has been completed, it calls for Action continuation (in other words, a callback)

Automatic Callback Creation Mechanism

Manual procedures -> automation

A callback chain is automatically created and run using `await`.

Since it also does things such as exception creation and optimization, there are also instances in which the efficiency is better than when using manual procedures.

Is `async/await` a coroutine?


The essential meaning is CPS conversion with implementation details as a state machine.

Since a coroutine is only chosen as an implementation for optimization, even if it enters into it, that's not what it means.

Optimization for Synchronization

Asynchronous is not asynchronous

Async transmits to a higher level (the method of calling async becomes async). As a result, there are often instances in which it may be async, but the content is synchronous.



For instance, a cache. When, in the lowest layer, there is an asynchronous method that starts out asynchronous and is returned synchronously from a cache after the second time, all behavior for the method that calls it, including upper layers, turns into a synchronous method.

Each time a continuation is called (call via a delegate) that way, there is delegate garbage production + a calling cost, so it isn't good.

An await-capable form requires a form (awaiter) that ultimately waits for the following method.

```
public class MyAwaiter<T> : INotifyCompletion
{
    bool IsCompleted { get; }

    T GetResult();

    void OnCompleted(Action continuation);
}
```

Has it already been completed (synchronous/asynchronous) or not?

Something to transmit return values / exceptions. (Even if there's a void, exceptions will be acquired by calling this.)

This is something for registering a continuation. The next part of await is called with (continuation.Invoke()).

```
// var result = await foo; ends up as follows.
if (awaiter.IsCompleted)
{
    // If there is an exception, it is thrown again with GetResult
    var result = awaiter.GetResult();
    // ...the beginning of await is executed.
}
else
{
    // Registration of a continuation (Actually, since it is optimized, the lambda expression isn't used every time).
    awaiter.OnCompleted(() =>
    {
        // If there is an exception, it is rethrown with GetResult.
        var result = awaiter.GetResult();
        // ...the beginning of await is executed.
    });
    return;
}
```

It is rather efficient since no excessive code is run when synchronous execution is possible.

Async Task implements **no** delay (at time of calling until await (to be exact, until await for IsCompleted = false)).

Because **synchronization is executed**. It may be complex in behavior, but the execution efficiency is very good.

Only when asynchronous is necessary is a continuation created, with execution through that.

Multiple Await Optimization

Not all continuations are created

```
public async Task FooBarBazAsync()  
{
```

```
    await Task.Yield();
```

State1

```
    Console.WriteLine("foo");  
    await Task.Yield();
```

State2

```
    Console.WriteLine("bar");  
    await Task.Yield();
```

State3

```
    Console.WriteLine("baz");  
}
```

↙ Action continuation = this.MoveNext

↙ Action continuation = this.MoveNext

↙ Action continuation = this.MoveNext

Multiple

Not all

The Async method creates an AsyncStateMachine and all continuations indicate the same method (this.MoveNext). MoveNext advances its own State by one.

This way, even if there are several await within the same method, the created continuation ends at one (as an optimization point for the user, there is also the technique of having things merged into one be more efficient than having an asynchronous method divide them).

```
public async Task FooBarBazAsync()  
{
```

```
    await Task.Yield();
```

State1

```
    Console.WriteLine("foo");  
    await Task.Yield();
```

State2

```
    Console.WriteLine("bar");  
    await Task.Yield();
```

State3

```
    Console.WriteLine("baz");  
}
```

← Action continuation = this.MoveNext

← Action continuation = this.MoveNext

← Action continuation = this.MoveNext

Why UniTask?

What is UniTask?

UniRx.Async's main class

<https://github.com/Cysharp/UniTask>

It has been possible to implement async return values other than Task since C# 7.0. It is Task (ValueTask equivalent) structured with the unique form compatible with async used in that.

In other words, a personalized asynchronous framework.

C# 7.0 can be used after **Unity 2018.3**.

Why is it needed?

By replacing everything, the liabilities of Task itself are completely ignored.

Since Unity itself is a special execution environment, it has the fastest implementation through specialization.

The Special Characteristics of Unity

Unity is (in general) single thread

C++ engine layer

Handling on the

(Coroutines, WWW)

When async/await

transferred to a thread pool

-> Delay, ContinueWith, Return, etc...

Async/await(Task) has a multithread -> single thread unification function, but, if it was single thread to begin with, wouldn't deleting that unification layer increase both performance and ease of handling?

More than this:

- Environments incompatible with multithread, such as WebGL, can also be used.
 - There is increased efficiency from not doing SynchronizationContext acquisition and returns.
- (+ since it's based on ValueTask, there won't be excessive allocation in a situation completed synchronously.)

Ignore XxxContext

XxxContext is the overhead of Task

Two varieties of capture, ExecutionContext and SynchronizationContext

```
▼ AsyncTaskMethodBuilder`1.Start()
  ▼ <Foo>d__3.MoveNext()
    ▼ AsyncTaskMethodBuilder`1.AwaitUnsafeOnCompleted()
      ► ResourceRequestAwaiter.UnsafeOnCompleted()
      ▼ AsyncMethodBuilderCore.GetCompletionAction()
        ▼ ExecutionContext.FastCapture()
          ► ExecutionContext.Capture()
            MoveNextRunner..ctor()
            ExecutionContext.get_IsPreAllocatedDefault()
            Debugger.NotifyOfCrossThreadDependency()
            GC.Alloc
          ► AsyncTaskMethodBuilder`1.get_Task()
          ► AsyncMethodBuilderCore.PostBoxInitialization()
            AsyncCausalityTracer.get_LoggingOn()
        ► Resources.LoadAsync()
        ► ResourceRequestAwaiter.get_IsCompleted()
        ► UnityAsyncExtensions.GetAwaiter()
      ▼ ExecutionContext.EstablishCopyOnWriteScope()
        ► ExecutionContext.EstablishCopyOnWriteScope()
          Thread.get_CurrentThread()
      ▼ ExecutionContextSwitcher.Undo()
        ► Thread.SetExecutionContext()
          ExecutionContext.OnAsyncLocalContextChanged()
        ► Thread.GetExecutionContextReader()
          Reader.DangerousGetRawExecutionContext()
          RuntimeHelpers.PrepareConstrainedRegions()
        AsyncTaskMethodBuilder`1.get_Task()
        AsyncTaskMethodBuilder`1.Create()
```

The stack trace for Task + async/await

Since Task + async/await performs such behavior as storing ExecutionContext (the data linked to each thread, also a SynchronizationContext holder) and operating while returning source thread data after await, when we look at the stack trace, we can see that there is a lot of Capture and such.

Ignore XxxContext

XxxContext is the overhead of Task

Two varieties of capture, ExecutionContext and SynchronizationContext

```
▼ AsyncTaskMethodBuilder`1.Start()  
  ▼ <Foo>d__3.MoveNext()  
    AsyncTaskMethodBuilder`1.AsyncMethodForOnCompleted()
```

The stack trace for UniTask + async/await

UniTask does not capture
ExecutionContext/SynchronizationContext! Due to that, it
results in an execution path (IsCompleted/OnCompleted ->
GetResult) nearly unchanged from the pseudocode
introduced in the explanation of the principle.

“The shorter the faster” is obvious!

```
▼ AsyncUniTaskMethodBuilder`1.Start()  
  ▼ <Bar>d__4.MoveNext()  
    AsyncUniTaskMethodBuilder`1.AwaitUnsafeOnCompleted()  
      ► ResourceRequestAwaiter.UnsafeOnCompleted()  
        GC.Alloc  
        Promise`1..ctor()  
        MoveNextRunner..ctor()  
      ► Resources.LoadAsync()  
      ► ResourceRequestAwaiter.get_IsCompleted()  
      ► UnityAsyncExtensions.GetAwaiter()  
    ► AsyncUniTaskMethodBuilder`1.get_Task()  
    <Bar>d__4..ctor()  
    AsyncUniTaskMethodBuilder`1.Create()  
    GC.Alloc
```

```
  ► Thread.GetExecutionContextReader()  
    Reader.DangerousGetRawExecutionContext()  
    RuntimeHelpers.PrepareConstrainedRegions()  
    AsyncTaskMethodBuilder`1.get_Task()  
    AsyncTaskMethodBuilder`1.Create()
```

Gameloop Based Async/Await

Utilities for coroutine replacement

UniTask.Delay

UniTask.WaitUntil

UniTask.WaitWhile

UniTask.WaitUntilValueChanged

UniTask.Run

UniTask.Yield

UniTask.SwitchToMainThread

UniTask.SwitchToThreadPool

await AsyncOperation

Since it runs based on Playeloop.Update, it's faster and lighter (there is no timer startup or SynchronizationContext return cost) than Task.Delay, which is thread based, and it also runs with things like WebGL.

Things such as WaitUntil/WaitWhile coroutines can be substituted.

Substitutions for things such as ObserveOnMainThread and ObserveOnThreadPool found in Rx.
In particular, Yield/SwitchToMainThread has high performance with zero allocation.

All of Unity's asynchronous objects, such as LoadAsync, become await capable. JobSystem is also capable of await.

vs Coroutine

```
IEnumerator FooCoroutine(Func<int> resultCallback, Func<Exception> errorCallback)
```

```
{
```

```
    int x = 0;
```

```
    Exception error = null;
```

```
    yield return Nanikamatu v => x = v, ex => error = ex);
```

```
    if (error == null)
```

```
    {
```

```
        resultCallback(x);
```

```
    }
```

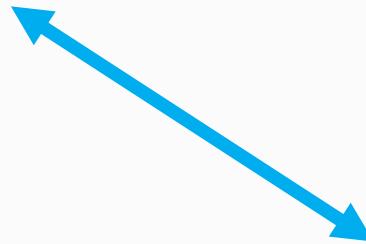
```
    else
```

```
    {
```

```
        errorCallback(error);
```

```
    }
```

```
}
```



```
UniTask<int> FooAsync()
```

```
{
```

```
    var x = await NanikasuruAsync();
```

```
    return x;
```

```
}
```


vs Coroutine

(When needed) there are allocations such as return value by delegate and exception callback (+ capture allocation if there is capture in lambda expression on the calling side).

```
IEnumerator FooCoroutine(Func<int> resultCallback, Func<Exception> exceptionCallback)
```

```
{  
    int x = 0;  
    Exception error = null;  
    yield return Nanikamatu(v => x = v, ex => error = ex);  
    if (error != null)
```

IEnumerator allocation as a return value and Coroutine allocation for StartCoroutine return values on the calling side.

Allocation through lambda expression capture when calling with multi-step.

(AsyncStateMachine itself is struct, but it becomes class only during Debug build (This is a C# compiler specification and is there for equilibrium with the debugger)).

```
else
```

```
{  
    exceptionCa  
}
```

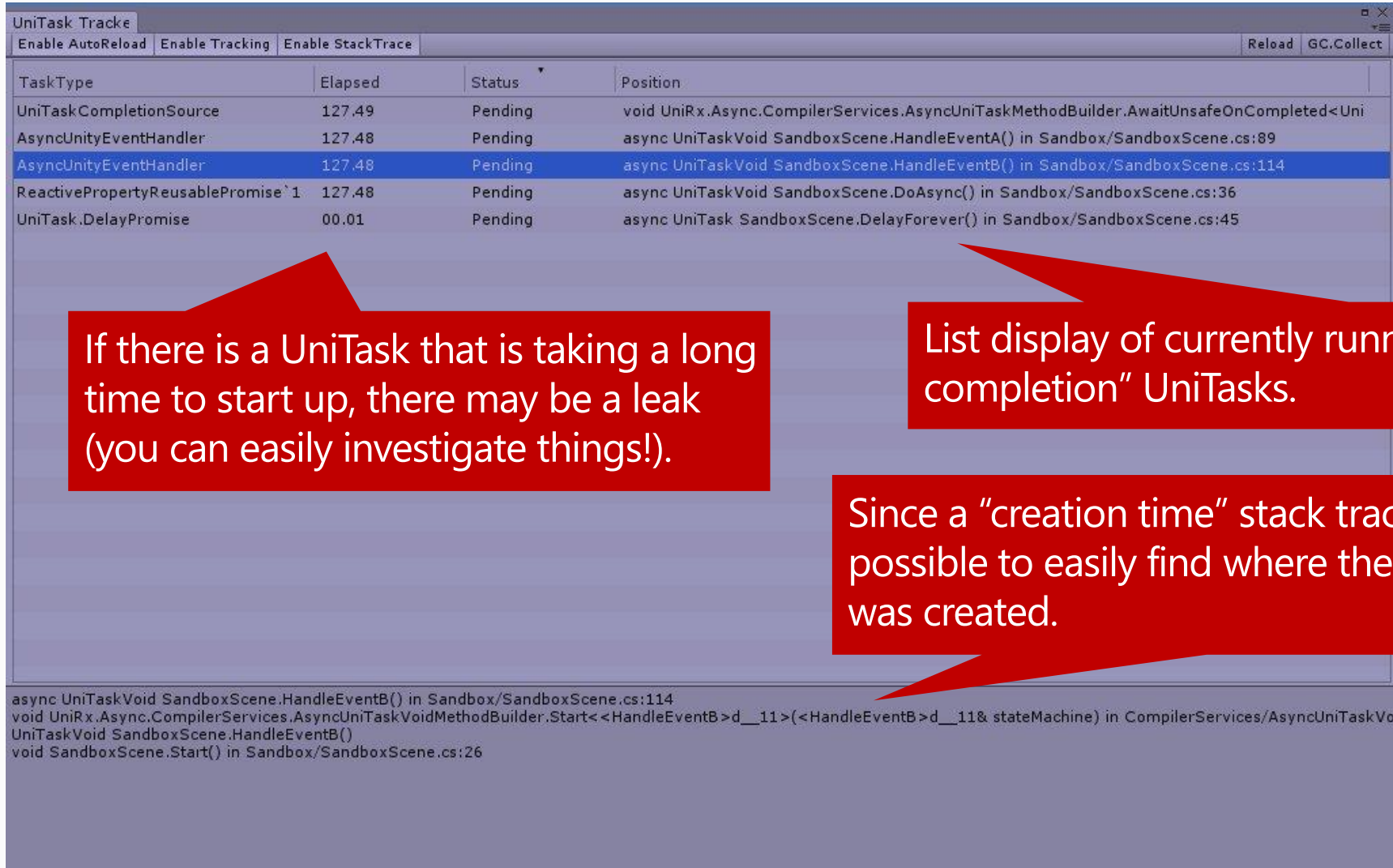
Allocation as a return value (Promise) (if the content happens to be synchronous, there is zero allocation due to struct).

```
UniTask<int> FooAsync()
```

```
{  
    var x = await NanikasuruAsync();  
    return x;  
}
```

Runner creation allocation in order to drive AsyncStateMachine + continuation delegate creation allocation.

UniTask Tracker



The screenshot shows the UniTask Tracker application window. At the top, there are tabs for 'Enable AutoReload', 'Enable Tracking', and 'Enable StackTrace'. On the right, there are buttons for 'Reload' and 'GC.Collect'. Below these is a table with four columns: 'TaskType', 'Elapsed', 'Status', and 'Position'. The table lists several tasks, with the third row highlighted in blue. Below the table, there is a text area showing the stack trace for the selected task.

TaskType	Elapsed	Status	Position
UniTaskCompletionSource	127.49	Pending	void UniRx.Async.CompilerServices.AsyncUniTaskMethodBuilder.AwaitUnsafeOnCompleted<Uni
AsyncUnityEventHandler	127.48	Pending	async UniTaskVoid SandboxScene.HandleEventA() in Sandbox/SandboxScene.cs:89
AsyncUnityEventHandler	127.48	Pending	async UniTaskVoid SandboxScene.HandleEventB() in Sandbox/SandboxScene.cs:114
ReactivePropertyReusablePromise`1	127.48	Pending	async UniTaskVoid SandboxScene.DoAsync() in Sandbox/SandboxScene.cs:36
UniTask.DelayPromise	00.01	Pending	async UniTaskVoid SandboxScene.DelayForever() in Sandbox/SandboxScene.cs:45

Stack Trace (from bottom to top):

```
void SandboxScene.Start() in Sandbox/SandboxScene.cs:26
UniTaskVoid SandboxScene.HandleEventB()
void UniRx.Async.CompilerServices.AsyncUniTaskVoidMethodBuilder.Start<<HandleEventB>d__11>(<HandleEventB>d__11& stateMachine) in CompilerServices/AsyncUniTaskVo
async UniTaskVoid SandboxScene.HandleEventB() in Sandbox/SandboxScene.cs:114
```

If there is a UniTask that is taking a long time to start up, there may be a leak (you can easily investigate things!).

List display of currently running "waiting for await completion" UniTasks.

Since a "creation time" stack trace is kept, it's possible to easily find where the (leaking) UniTask was created.

Conclusion

UniTask + async/await for performance

Since UniTask is specialized for Unity, it has far greater performance than Task.

No ExecutionContext, No SynchronizationContext

UniTask has less allocation than coroutine implementation.

It has greater performance in asynchronous sections than UniRx(Observable).

UniTask + async/await for ease of use

Since single thread is assumed, there are no multithread traps.

It offers an abundance of functions and has the ability to practically replace coroutines.

UniTask leaks can be easily avoided with UniTask Tracker.

There's also no problem in using it in combination with Task and Rx.

Conclusion

UniTask + async/await for performance

Since UniTask is specialized for Unity, it has far greater performance than Task.

No ExecutionContext, No SynchronizationContext

UniTask has less allocation than co

It has greater performance in asyn

Those who have used .NETでTask + async/await should definitely have fallen into these terrible things.

UniTask + async/await for ease of use

Since single thread is assumed, there are no multithread traps.

It offers an abundance of functions and has the ability to practically replace coroutines.

UniTask leaks can be easily avoided with UniTask Tracker.

There are many people who have had an "AddTo party" with Observable leaks in UniRx, haven't they?

The States of UniTask

```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

Four states exist in UniTask.

When created from UniTaskCompletionSource, it is possible to transition between each of TrySetResult/TrySetException/TrySetCanceled.

(These are fewer states than with Task, but it matches the latest ValueTaskSourceStatus. Task really does have a lot of unnecessary garbage.....)

```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

Now then, when does the Status change for a form made through compiler creation?

```
public async UniTask<int> FooAsync()
{
    await UniTask.Yield();
    return 10;
}
```

```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

The span from time of creation to status completion is Pending.

```
public async UniTask<int> FooAsync()
{
    [
        await UniTask.Yield();
    ]
    return 10;
}
```



```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

If the value has completed
return normally, it is
Succeeded.

```
public async UniTask<int> FooAsync()
{
    await UniTask.Yield();

    { return 10;
    }
```

```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

If an exception has been thrown, it is Faulted.

```
public async UniTask<int> FooAsync()
{
    await UniTask.Yield();
    { throw new System.Exception("Error"); }
}
```

```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

What should be done for it to be changed to Canceled?

```
public async UniTask<int> FooAsync()
{
    await UniTask.Yield();

    throw new System.Exception("Error");
}
```


```
public enum AwaiterStatus
{
    /// <summary>The operation has not yet completed.</summary>
    Pending = 0,
    /// <summary>The operation completed successfully.</summary>
    Succeeded = 1,
    /// <summary>The operation completed with an error.</summary>
    Faulted = 2,
    /// <summary>The operation completed due to cancellation.</summary>
    Canceled = 3
}
```

When
OperationCanceledException
is thrown, it becomes
Canceled.


```
public async UniTask<int> FooAsync()
{
    await UniTask.Yield();

    throw new OperationCanceledException();
}
```

```
public void Baz()  
{  
    BarAsync().Forget();  
}
```



```
public async Task<int> BarAsync()  
{  
    var x = await FooAsync();  
    return x * 2;  
}
```



```
public async Task<int> FooAsync()  
{  
    await Task.Yield();  
    throw new OperationCanceledException();  
}
```

In a case of Canceled/Faulted, it is transmitted to a higher level in the form of an exception thrown during await.

```
public void Baz()  
{  
    BarAsync().Forget();  
}
```



```
public async  
{  
    var x = ...  
    return x;  
}
```

Task transmission ends somewhere (generally OK as an image like Rx's Subscribe). Forget also has (Action<Exception> onError) and unprocessed exceptions are dealt with there. If unspecified, it is routed to a global unprocessed exception handler (UniTaskScheduler.UnobservedTaskException).

If the OperationCanceledException has been passed to the global unprocessed exception handler, it is ignored. In other words, the difference between Faulted and Canceled is whether it is ultimately ignored or reported.

```
await UniTask.Yield();  
throw new OperationCanceledException();
```

```
public async Task<int> BarAsync()
{
    try
    {
        var x = await FooAsync();
        return x * 2;
    }
    catch (Exception ex) when (!(ex is OperationCanceledException))
    {
        return -1;
    }
}
```

When an Exception is caught with catch, it is best to make it so the OperationCanceledException is clear through an exception filter.

```
public async Task<int> BarAsync()
{
    try
    {
        var x = await FooAsync();
        return x * 2;
    }
    catch (Exception ex) when (!(ex is OperationCanceledException))
    {
        // Since it seems to be an unrecoverable exception, do something along the lines of Open Dialog and Return to Title.
        DialogService.ShowReturnToTitleAsync().Forget(); // Handled like fire and forget.

        // Everything in the original call is tossed and terminated using a cancel chain.
        throw new OperationCanceledException();
    }
}
```

By rethrowing with `OperationCanceledException`, you can expect to display results similar to `Observable.Empty/Never`.

Cancellation of Async

Pain Point of Async/Await

Cancellation is troublesome

If only Rx could knock out IDisposable return values with one punch!

(Instead, there is no IDisposable allocation in async/await.)

Instead, as an argument (CancellationToken is passed around).

Having a CancellationToken at the end of the return value is the recommended rule for an asynchronous method.

```
public Task<int> FooAsync(int x, int y, CancellationTokentoken = default)
{
    var x = await BarAsync(x, y, cancellationTokentoken);
    return x;
}
```

Intently propagating that.

Who Threw OperationCanceledException?

Cancel = OperationCanceledException

There is no need for user code to check `cancellationToken.IsCancellationRequested`.

That's because the user code section is **synchronous**.

`OperationCanceledException` is thrown from an asynchronous source

= `asyncOperation.ConfigureAwait(token)`, `UniTask.Delay(token)`, etc...

At any rate, pass, that should be enough.

Since an asynchronous source should be processing it, reporting that much should be OK.

Oh, but that's so much trouble!!!

There's no way to hack it to get it to do it automatically (without dropping performance).


Who Created CancellationTokenSource?

MonoBehaviour/OnDestroy is convenient for Unity

```
public class FooBehaviour : MonoBehaviour
{
    CancellationTokenSource cts;

    void Start()
    {
        cts = new CancellationTokenSource();
    }

    void OnDestroy()
    {
        cts.Cancel();
        cts.Dispose();
    }
}
```



If Cancel is set in motion, hooked by Destroy, for the time being (if Token is passed around everywhere), there is absolutely no leak.

Exceptions?

Is there a high cost?

Yes!

If Cancel is a kind of exception and only goes off occasionally, it shouldn't be much of a problem, but when Cancel turns normal and becomes assumed, depending on the circumstances, it can turn into something pretty harsh.

For instance, what if an exception goes off with the cancel of 10,000 Cubes that are on screen at time of a scene transition linked to a MonoBehaviour scene...?

UniTask.SuppressCancellationThrow

In UniTask, there is SuppressCancellationThrow, which converts cancel into (bool isCanceled, T value).

However, be careful since only an asynchronous source can control an exception.

Async Event Handling

AsyncTrigger/AsyncUGUI

Events can be implemented with async/await

await button.OnClickAsync();

await gameObject.OnCollisionEnterAsync();

There are implementations for these in UniTask.

```
async UniTask TripleClick(Cancellation_token token)
{
    await button.OnClickAsync(token);
    await button.OnClickAsync(token);
    await button.OnClickAsync(token);
    Debug.Log("Three times clicked");
}
```

When using await for an event, it's best to pass Cancellation_token since there's a risk of infinite standby.

AsyncTrigger/AsyncUGUI

Events can be implemented with async/await

await button.OnClickAsync();

await gameObject.OnCollisionEnterAsync();

There are implementations for these in UniTask.

```
async UniTask TripleClick(Cancellation_token token)
{
    // Acquiring a Handler at the start is more efficient than doing OnClick/token passing each time.
    using (var handler = button.GetAsyncClickEventHandler(token))
    {
        await handler.OnClickAsync();
        await handler.OnClickAsync();
        await handler.OnClickAsync();
        Debug.Log("Three times clicked");
    }
}
```

Further, it's best on the performance side when you also mix in `SupressCancellationThrow`. (It's a little complicated, huh?)

When Should It Be Used?

Well, it may be a little unreasonable.

Rx is generally better for event handling.

The code becomes longer and items to consider on the performance side increase quite a bit.

However, when implementing complex flow, it's possible to write in a more clear and straightforward manner with "code awaiting a synchronous event" than by making do with an Rx operator.

It's not bad to have this kind of technique in your wallet, is it?

(In the same way, await is also possible with ReactiveProperty.)

Reusable Promise

Part of UniTask Can Be Reused

For the sake of performance

Reuse is possible with local variables (only for part of an asynchronous source).

```
async UniTask DelayFiveAsync1()
{
    for (int i = 0; i < 5; i++)
    {
        // Delay is created each time.
        await UniTask.Delay(i * 1000);
        Debug.Log(i);
    }
}
```

```
[[async UniTask DelayFiveAsync2()
{
    // Delay is recycled.
    var delay = UniTask.Delay(i * 1000);
    for (int i = 0; i < 5; i++)
    {
        await delay;
        Debug.Log(i);
    }
}
```

Since it's not possible to determine which parts can be reused by looking at it, check out the list in ReadMe

Conclusion

Ready to Go async/await

Don't be afraid!

Performance is not a problem (when UniTask is used).

Practice has already been established (when UniTask is used).

There's almost no damage from overdoing things (strongly speaking, asynchronous pollution).

It's at a level at which there's no reason not to go for it, so GO right now.

Recommended to use with UniRx.Async

async/await has been redesigned for Unity with performance and user-friendliness.

Everything used in Unity has been made capable of await.

Don't be afraid of normal Task not being used!

Let's traverse the leading edge, surpassing all language and the ordinary .NET!