

UniTask v2 — Zero Allocation async/await for Unity, with Asynchronous LINQ



Yoshifumi Kawai · Follow

10 min read · Jun 12, 2020



Listen



Share

I've previously published [UniTask, a new async/await library for Unity](#), now I've rewritten all the code and released new one.

[GitHub — Cysharp/UniTask](#)

In UniTask v2, almost everything is zero-allocated due to a thorough rewrite of the code (Technical details to follow). In addition to significant performance improvements, the new asynchronous sequences and Asynchronous LINQ. Others, await extensions for external assets such as DOTween and Addressables Support has also been built in for added convenience.

Before I talk about v2, here's a recap. async/await is a feature that has been included since C# 5.0. Just like writing asynchronous code in synchronous code instead of asynchronous code handled by callback chains and coroutines. Both return values and exception handling will be handled naturally. When handling only callbacks, the complexity of the process results in multiple nests, and the inner exception is not propagated outward, making it difficult to process errors.

```
1 FooAsync(x =>
2 {
3     BarAsync(x, y =>
4     {
5         BazAsync(y, z =>
6         {
7             });
8         });
9     });
```

blog_unitask1.cs hosted with ❤ by GitHub

[view raw](#)

In Unity, the coroutine, which is achieved with a yield return (generator), allows you to use asynchronous processing can be used to flatten the nests to some extent, but return value and error cannot be handled due to the constraints, so used with delegate.

```

1  IEnumerator FooCoroutine(Func<int> resultCallback, Func<Exception> exceptionCallback)
2  {
3      int x = 0;
4      Exception error = null;
5      yield return DoAsync(v => x = v, ex => error = ex);
6
7      if (error == null)
8      {
9          resultCallback(x);
10     }
11     else
12     {
13         exceptionCallback(error);
14     }
15 }

```

blog_unitask2.cs hosted with ❤ by GitHub

[view raw](#)

It can be flattened nest slightly, but

- Callback process still remains
- Can't use try-catch-finally because of the yield syntax
- Allocation of the lambda and the coroutine itself
- Difficult to do cancellation process(coroutine only stop, not call dispose)
- Impossible to control multiple coroutines (serial/parallel processing)

In async/await

```

1  async UniTask<int> FooAsync(int x)
2  {
3      try
4      {
5          var y = await BarAsync(x);
6          var z = await BazAsync(y);
7          return x + y + z;
8      }
9      catch (Exception ex)
10     {
11     }
12     finally
13     {
14     }
15 }

```

blog_unitask3.cs hosted with ❤ by GitHub

[view raw](#)

with language-level support for asynchronous code that is almost identical to those of synchronous code.

However, the Unity framework itself doesn't have much async/await support. UniTask provides...

- await support for each Unity's AsyncOperations
- Unity's PlayerLoop-based switching process (Yield, Delay, DelayFrame, etc...) allows for all the functionality of a coroutine on UniTask
- await support for MonoBehaviour events and uGUI events

I've implemented a custom UniTask type based on struct with a dedicated AsyncMethodBuilder. Ignore .NET Task and without ExecutionContext/SynchronizationContext that is unnecessary for Unity, it achieves Unity-optimized performance.

```
1 // await Unity's AsynchronousObject directly
2 var asset = await Resources.LoadAsync<TextAsset>("foo");
3
4 // wait 100 frame(instead of Coroutine)
5 await UniTask.DelayFrame(100);
6
7 // Alternative of WaitForFixedUpdate
8 await UniTask.WaitForFixedUpdate();
```

blog_unitask4.cs hosted with ❤ by GitHub

[view raw](#)

with UniTask, Unity is now able to take full advantage of the power of async/await.

And things have changed in the two years since the first release. NET Core 3.1, and then .NET 5 will be released and the runtime was rewritten. And we're seeing C# 8.0 in Unity as well. So, while the above elements have been retained, I have completely revised the API and

- Zero-allocation of the entire async method to further improve performance
- Asynchronous LINQ(UniTaskAsyncEnumerable, Channel, AsyncReactiveProperty)
- Increased PlayerLoop timing (new LastPostLateUpdate will have the same effect as WaitForEndOfFrame)
- Support for external assets such as Addressables and DOTween

I have implemented performance improvements and in particular, zero-allocation is now able to reduce GC, even with heavy use of async/await Therefore, you can expect a significant performance improvement.

In addition, I have also adjusted the behavior to be similar to .NET Core ValueTask/IValueTaskSource(e.g. Delay is launched on call as well as Task, with the exception of twice await throw, etc.). This gives you the performance benefits of a UniTask, but the behavior The learning gap has been reduced by making it standards-aligned for.

With support for C# 8.0 starting in Unity 2020.2.0a12, asynchronous stream notation is now possible. So UniTask v2 supports asynchronous streams. Added UniTaskAsyncEnumerable.

```

1 // Unity 2020.2.0a12~, C# 8.0
2 await foreach (var _ in UniTaskAsyncEnumerable.EveryUpdate())
3 {
4     Debug.Log("Update() " + Time.frameCount);
5 }
6
7 // C# 7.3(Unity 2018.3~)
8 await UniTaskAsyncEnumerable.EveryUpdate().ForEachAsync(_ =>
9 {
10     Debug.Log("Update() " + Time.frameCount);
11 });

```

blog_unitask5.cs hosted with ❤ by GitHub

[view raw](#)

Even in Unity 2018, 2019 and 2020.1, where C# 8.0 is not supported In combination with the on-board asynchronous LINQ, almost identical processing is possible. Also, since it is LINQ, all standard LINQ query operators can be applied to asynchronous streams. For example, the following code runs once every two times on a button-click asynchronous stream Here's an example of the Where filter.

```

1 await okButton.OnClickAsyncEnumerable()
2     .Where((x, i) => i % 2 == 0)
3     .ForEachAsync(_ =>
4     {
5     });

```

blog_unitask6.cs hosted with ❤ by GitHub

[view raw](#)

In addition to button clicks, it provides a rich asynchronous stream factory that integrates with a number of Unity, so you can write all kinds of processes depending on your ingenuity.

Principle of AsyncStateMachine and Zero Allocation

While there are many new features, such as asynchronous stream support, the biggest feature of UniTask v2 is a significant performance improvement.

Compared to the standard Task implementation of async/await, UniTask v2 is much better than the standard Task implementation in terms of where allocation occurs and how it was deterred. Let's break down the structure, taking the following relatively simple asynchronous method as an example.

```

1  public class Loader
2  {
3      object cache;
4
5      public async Task<object> LoadAssetFromCacheAsync(string address)
6      {
7          if (cache == null)
8          {
9              cache = await LoadAssetAsync(address);
10         }
11         return cache;
12     }
13
14     Task<object> LoadAssetAsync(string address)
15     {
16         // do something...
17     }
18 }

```

blog_unitask7.cs hosted with ❤ by GitHub

[view raw](#)

If there is a cache that has been read, it is returned, otherwise it reads asynchronously. This await code is a built at compile-time to GetAwaiter -> IsCompleted/ GetResult/UnsafeOnCompleted.

```

1  public class Loader
2  {
3      public Task<object> LoadAssetFromCacheAsync(string address)
4      {
5          if (cache == null)
6          {
7              var awaiter = LoadAssetAsync(address).GetAwaiter();
8              if (awaiter.IsCompleted)
9              {
10                 cache = awaiter.GetResult();
11             }
12             else
13             {
14                 // register callback, where from moveNext and promise?
15                 awaiter.UnsafeOnCompleted(moveNext);
16                 return promise;
17             }
18         }
19         return Task.FromResult(cache);
20     }
21 }

```

blog_unitask8.cs hosted with ❤ by GitHub

[view raw](#)

It's a optimization that avoids the cost of creating/registering/calling a callback when a callback is not needed (e.g. this `LoadAssetFromCacheAsync` itself returns the value immediately if it's also cached).

The methods declared in `async` are converted by the compiler into a state machine. The `MoveNext` method registers this state machine with the `await` callback .

```

1  public class Loader
2  {
3      object cache;
4
5      public Task<object> LoadAssetFromCacheAsync(string address)
6      {
7          var stateMachine = new __LoadAssetFromCacheAsync
8          {
9              __this = this,
10             address = address,
11             builder = AsyncTaskMethodBuilder<object>.Create(),
12             state = -1
13         };
14
15         var builder = stateMachine.builder;
16         builder.Start(ref stateMachine);
17
18         return stateMachine.builder.Task;
19     }
20
21     // compiler generated async-statemachine
22     // Note: in debug build statemachine as class.
23     struct __LoadAssetFromCacheAsync : IAsyncStateMachine
24     {
25         // local variables to field.
26         public Loader __this;
27         public string address;
28
29         // internal state
30         public AsyncTaskMethodBuilder<object> builder;
31         public int state;
32
33         // internal local variables
34         TaskAwaiter<object> loadAssetAsyncAwaiter;
35
36         public void MoveNext()
37         {
38             try
39             {
40                 switch (state)
41                 {
42                     // initial(call from builder.Start)
43                     case -1:
44                         if (__this.cache != null)
45                         {
46                             goto RETURN;
47                         }
48                         else
49                         {
50                             // await LoadAssetAsync(address)
51                             loadAssetAsyncAwaiter = __this.LoadAssetAsync(address).GetAwaiter();
52                             if (loadAssetAsyncAwaiter.IsCompleted)
53                             {
54                                 goto case 0;
55                             }
56                             else
57                             {
58                                 state = 0;
59                                 builder.AwaitUnsafeOnCompleted(ref loadAssetAsyncAwaiter, ref this);
60                                 return; // when call MoveNext again, goto case 0;

```

```

60         return; // when call movenext again, goto case 0:
61     }
62 }
63 case 0:
64     __this.cache = loadAssetAsyncAwaiter.GetResult();
65     goto RETURN;
66 default:
67     break;
68 }
69 }
70 catch (Exception ex)
71 {
72     state = -2;
73     builder.SetException(ex);
74     return;
75 }
76
77 RETURN:
78 state = -2;
79 builder.SetResult(__this.cache);
80 }
81
82 public void SetStateMachine(IAsyncStateMachine stateMachine)
83 {
84     builder.SetStateMachine(stateMachine);
85 }
86 }
87 }

```

blog_unitask9.cs hosted with ❤ by GitHub

[view raw](#)

The StateMachine implementation is a bit longer. You can think of it as being split by a line and advancing a state. It's a little hard to read because it's set with the Builder described below. The awaiter calls GetResult immediately if IsCompleted is true. If false, register its own MoveNext to UnsafeOnCompleted. This MoveNext will be called again when the asynchronous process is completed and calls own GetResult.

The last character is the `AsyncTaskMethodBuilder`, which is not a compiler-generated class, corresponding builder class with a `Task`. The original source is a bit long, so I'll post the simplified code.

```

1  public struct AsyncTaskMethodBuilder<TResult>
2  {
3      MoveNextRunner runner;
4      Task<TResult> task;
5
6      public static AsyncTaskMethodBuilder<TResult> Create()
7      {
8          return default;
9      }
10
11     public void Start<TStateMachine>(ref TStateMachine stateMachine)
12         where TStateMachine : IAsyncStateMachine
13     {
14         // when start, call stateMachine's MoveNext directly.
15         stateMachine.MoveNext();
16     }
17
18     public Task<TResult> Task
19     {
20         get
21         {
22             if (task == null)
23             {
24                 // internal task creation(same as TaskCompletionSource but avoid tcs allocation)
25                 task = new Task<TResult>();
26             }
27
28             return task.Task;
29         }
30     }
31
32     public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine)
33         where TAwaiter : ICriticalNotifyCompletion
34         where TStateMachine : IAsyncStateMachine
35     {
36         // at first await, copy struct state machine to heap(boxed).
37         if (runner == null)
38         {
39             _ = Task; // create TaskCompletionSource
40
41             // create runner
42             runner = new MoveNextRunner((IAsyncStateMachine)stateMachine); // boxed.
43         }
44
45         // set cached moveNext delegate(as continuation).
46         awaiter.UnsafeOnCompleted(runner.CachedDelegate);
47     }
48
49     public void SetResult(TResult result)
50     {
51         if (task == null)
52         {
53             _ = Task; // create Task
54             task.TrySetResult(result); // same as TaskCompletionSource.TrySetResult.
55         }
56         else
57         {
58             task.TrySetResult(result);
59         }
60     }

```

```

60     }
61 }
62
63 public class MoveNextRunner
64 {
65     public Action CachedDelegate;
66
67     IAsyncStateMachine stateMachine;
68
69     public MoveNextRunner(IAsyncStateMachine stateMachine)
70     {
71         this.stateMachine = stateMachine;
72         this.CachedDelegate = Run; // Create cached delegate.
73     }
74
75     public void Run()
76     {
77         stateMachine.MoveNext();
78     }
79 }

```

blog_unitask10.cs hosted with ❤ by GitHub

[view raw](#)

The Builder does the initial call (Start), getting the return value Task, registering the callback (AwaitUnsafeOnCompleted) and setting the result (SetResult/SetException).

The await chain is similar to the callback chain, but if you write the callback chain by hand, the cannot avoid the occurrence of lambda's closure allocations, but async/await can be used with the compiler generates a single delegate to do all the work, which reduces the allocations . These mechanisms make writing in async/await more powerful than handwriting in It will be.

Now that we have all the pieces in place. async/await and Task is very good basically because it includes a detailed optimization that takes advantage of the compiler generation, but there are some problems.

In terms of memory allocation, the following four allocations occur in the worst case

- Task Allocation
- Boxing the AsyncStateMachine
- Allocation of the Runner encapsulating the AsyncStateMachine
- Allocation of delegate for MoveNext

If you declare the return value in Task, even if the value is in the immediate return state, you can always use Task's Allocation occurs. To deal with this problem, the .NET Standard 2.1 introduces the ValueTask type. However, if a callback is needed, the Task allocation will still be there, and the Boxing of the AsyncStateMachine are also present. Since this allocation requires the StateMachine to be placed on a heap.

UniTask solves these problems with a custom AsyncMethodBuilder since C# 7.0.

```

1 // modify Task<T> -> UniTask<T> only.
2 public async UniTask<object> LoadAssetFromCacheAsync(string address)
3 {
4     if (cache == null)
5     {
6         cache = await LoadAssetAsync(address);
7     }
8     return cache;
9 }
10
11 // Compiler generated code is same as standard Task.
12 public UniTask<object> LoadAssetFromCacheAsync(string address)
13 {
14     var stateMachine = new __LoadAssetFromCacheAsync
15     {
16         __this = this,
17         address = address,
18         builder = AsyncUniTaskMethodBuilder<object>.Create(),
19         state = -1
20     };
21
22     var builder = stateMachine.builder;
23     builder.Start(ref stateMachine);
24
25     return stateMachine.builder.Task;
26 }
27
28 // UniTask's AsyncMethodBuilder
29 public struct AsyncUniTaskMethodBuilder<T>
30 {
31     IStateMachineRunnerPromise<T> runnerPromise;
32     T result;
33
34     public UniTask<T> Task
35     {
36         get
37         {
38             // when registered callback
39             if (runnerPromise != null)
40             {
41                 return runnerPromise.Task;
42             }
43             else
44             {
45                 // sync complete, return struct wrapped result
46                 return UniTask.FromResult(result);
47             }
48         }
49     }
50
51     public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter awaiter, ref TStateMachine stateMachine)
52         where TAwaiter : ICriticalNotifyCompletion
53         where TStateMachine : IAsyncStateMachine
54     {
55         if (runnerPromise == null)
56         {
57             // get Promise/StateMachineRunner from object pool
58             AsyncUniTask<TStateMachine, T>.SetStateMachine(ref stateMachine, ref runnerPromise);
59         }
60     }

```

```

60
61     awaiter.UnsafeOnCompleted(runnerPromise.MoveNext);
62 }
63
64 public void SetResult(T result)
65 {
66     if (runnerPromise == null)
67     {
68         this.result = result;
69     }
70     else
71     {
72         // SetResult signal Task continuation, it will call task.GetResult and finally return to pool self.
73         runnerPromise.SetResult(result);
74
75         // AsyncUniTask<TStateMachine, T>.GetResult
76         /*
77         try
78         {
79             return core.GetResult(token);
80         }
81         finally
82         {
83             TryReturn();
84         }
85         */
86     }
87 }
88 }

```

blog_unitask11.cs hosted with ❤ by GitHub

[view raw](#)

UniTask(value type) removes allocation in the case of an immediate value return. Strongly typed Runner(which does not occur boxing) is integrated with the return value of Task(RunnerPromise). Further,

It is retrieved from the object pool. When the call completes(`GetResult`), it returns to the pool. This completely removes Task and state machine related allocations.

As a limitation, all `UniTask` objects cannot await twice, as they automatically return to the pool when the await completes.

This constraint is the same as [ValueTask/IValueTaskSource](#).

The following operations should never be performed on a `ValueTask` instance:

- * Awaiting the instance multiple times.
- * Calling `AsTask` multiple times.
- * Using `.Result` or `.GetAwaiter().GetResult()` when the operation hasn't yet completed, or using them multiple times.
- * Using more than one of these techniques to consume the instance.

If you do any of the above, the results are undefined.

Although there are some inconveniences, aggressive pooling is possible thanks to this restriction.

Note that the implementation of these zero-allocation `async/await` methods is similar as introduced in [Async ValueTask Pooling in .NET 5](#). `UniTask v2` is not waiting for a runtime update in the distant future, but it is available right now in Unity.

If you monitor by the profiler in the UnityEditor or Development Build, you'll see the allocation. This is because the `AsyncStateMachine` generated by the C# compiler is "class" in a debug build. In the release build, it will be a "struct" and there will be no allocation.

The pool size is unlimited by default, but you can use `TaskPool.SetMaxPoolSize` to set the maximum size and `TaskPool.GetCacheSizeInfo` allows you to retrieve the number of items currently in the cache. Unlike .NET Core, Unity has a higher impact on the GC, so you should actively pool but it may be better to adjust it for some applications.

Coroutine and PlayerLoop

One of the key features of `UniTask`, which differs from `Task`, is to not use `SynchronizationContext`(and `ExecutionContext`) at all.

In the case of Unity, it automatically returns to the main thread (by the `UnitySynchronizationContext`). This is convenient at first glance, but there is overhead and no needed in Unity. Because Unity's asynchronous processing(`AsyncOperation`) is run on Unity's engine layer(C++) and that is already returned to the main thread in the scripting layer(C#).

So I cut the `SynchronizationContext` to make it lighter.

One more thing, the `SynchronizationContext` has only one place to come back to, and that's In the case of Unity, there are many situations in which the call to the execution sequence is finely controlled. There is.

For example, coroutines can also use `WaitForEndOfFrame` often, with things like `WaitForFixedUpdate`, we need to adjust the points in the execution sequence.

So instead of a single `SynchronizationContext`, `UniTask` now allows you to manually specify points in the execution sequence to return to.

```
1 // same as yield return null
2 await UniTask.NextFrame();
3
4 // same as yield return new WaitForEndOfFrame
5 await UniTask.WaitForEndOfFrame()
6
7 // Run on ThreadPool under this switching
8 await UniTask.SwitchToThreadPool();
9
10 // delay and wait at PreUpdate(similar as WaitForSeconds but you can set timing)
11 await UniTask.Delay(TimeSpan.FromSeconds(1), delayTiming: PlayerLoopTiming.PreUpdate);
12
13 // Call Update per 30 frames
14 await UniTaskAsyncEnumerable.IntervalFrame(30, PlayerLoopTiming.Update).ForEachAsync(_ =>
15 {
16 });
```

blog_unitask12.cs hosted with ❤ by GitHub

[view raw](#)

Currently in Unity, the standard `PlayerLoop` mechanism allows all event functions to modify. Here's the list, with `UniTask` injected at the beginning and end of each, for a total of 14 locations to choose.

Initialization

UniTaskLoopRunnerYieldInitialization

UniTaskLoopRunnerInitialization

PlayerUpdateTime

DirectorSampleTime

AsyncUploadTimeSlicedUpdate

SynchronizeInputs

SynchronizeState

XREarlyUpdate

UniTaskLoopRunnerLastYieldInitialization

UniTaskLoopRunnerLastInitialization

EarlyUpdate

UniTaskLoopRunnerYieldEarlyUpdate

UniTaskLoopRunnerEarlyUpdate

PollPlayerConnection

ProfilerStartFrame

GpuTimestamp

AnalyticsCoreStatsUpdate

UnityWebRequestUpdate

ExecuteMainThreadJobs

ProcessMouseInWindow

ClearIntermediateRenderers

ClearLines

PresentBeforeUpdate

ResetFrameStatsAfterPresent

UpdateAsyncReadbackManager

UpdateStreamingManager

UpdateTextureStreamingManager

UpdatePreloading

RendererNotifyInvisible

PlayerCleanupCachedData

UpdateMainGameViewRect

UpdateCanvasRectTransform

XRUpdate

UpdateInputManager

ProcessRemoteInput

ScriptRunDelayedStartupFrame

UpdateKinect

DeliverPlatformEvents

TangoUpdate

DispatchEventQueueEvents

PhysicsResetInterpolatedTransformPosition

SpriteAtlasManagerUpdate

PerformanceAnalyticsUpdate

UniTaskLoopRunnerLastYieldEarlyUpdate

UniTaskLoopRunnerLastEarlyUpdate

FixedUpdate

UniTaskLoopRunnerYieldFixedUpdate

UniTaskLoopRunnerYieldFixedUpdate

UniTaskLoopRunnerFixedUpdate

ClearLines

NewInputFixedUpdate

DirectorFixedSampleTime

AudioFixedUpdate

ScriptRunBehaviourFixedUpdate

DirectorFixedUpdate

LegacyFixedAnimationUpdate

XRFixedUpdate

PhysicsFixedUpdate

Physics2DFixedUpdate

DirectorFixedUpdatePostPhysics

ScriptRunDelayedFixedFrameRate

UniTaskLoopRunnerLastYieldFixedUpdate

UniTaskLoopRunnerLastFixedUpdate

PreUpdate

UniTaskLoopRunnerYieldPreUpdate

UniTaskLoopRunnerPreUpdate

PhysicsUpdate

Physics2DUpdate

CheckTexFieldInput

IMGUISendQueuedEvents

NewInputUpdate

SendMouseEvents

AIUpdate

WindUpdate

UpdateVideo

UniTaskLoopRunnerLastYieldPreUpdate

UniTaskLoopRunnerLastPreUpdate

Update

UniTaskLoopRunnerYieldUpdate

UniTaskLoopRunnerUpdate

ScriptRunBehaviourUpdate

ScriptRunDelayedDynamicFrameRate

ScriptRunDelayedTasks

DirectorUpdate

UniTaskLoopRunnerLastYieldUpdate

UniTaskLoopRunnerLastUpdate

PreLateUpdate

UniTaskLoopRunnerYieldPreLateUpdate

UniTaskLoopRunnerPreLateUpdate

AIUpdatePostScript

DirectorUpdateAnimationBegin

LegacyAnimationUpdate

DirectorUpdateAnimationEnd

DirectorDeferredEvaluate

EndGraphicsJobsAfterScriptUpdate

ParticleSystemBeginUpdateAll
ConstraintManagerUpdate
ScriptRunBehaviourLateUpdate
UniTaskLoopRunnerLastYieldPreLateUpdate
UniTaskLoopRunnerLastPreLateUpdate

PostLateUpdate

UniTaskLoopRunnerYieldPostLateUpdate
UniTaskLoopRunnerPostLateUpdate
PlayerSendFrameStarted
DirectorLateUpdate
ScriptRunDelayedDynamicFrameRate
PhysicsSkinnedClothBeginUpdate
UpdateRectTransform
UpdateCanvasRectTransform
PlayerUpdateCanvases
UpdateAudio
VFXUpdate
ParticleSystemEndUpdateAll
EndGraphicsJobsAfterScriptLateUpdate
UpdateCustomRenderTextures
UpdateAllRenderers
EnlightenRuntimeUpdate
UpdateAllSkinnedMeshes
ProcessWebSendMessage
SortingGroupsUpdate
UpdateVideoTextures
UpdateVideo
DirectorRenderImage
PlayerEmitCanvasGeometry
PhysicsSkinnedClothFinishUpdate
FinishFrameRendering
BatchModeUpdate
PlayerSendFrameComplete
UpdateCaptureScreenshot
PresentAfterDraw
ClearImmediateRenderers
PlayerSendFramePostPresent
UpdateResolution
InputEndFrame
TriggerEndOfFrameCallbacks
GUIClearEvents
ShaderHandleErrors
ResetInputAxis
ThreadedLoadingDebug
ProfilerSynchronizeStats
MemoryFrameMaintenance
ExecuteGameCenterCallbacks
ProfilerEndFrame
UniTaskLoopRunnerLastYieldPostLateUpdate
UniTaskLoopRunnerLastPostLateUpdate

It's long, so let's just take out Update.

Update

UniTaskLoopRunnerYieldUpdate

UniTaskLoopRunnerUpdate

ScriptRunBehaviourUpdate

ScriptRunDelayedDynamicFrameRate

ScriptRunDelayedTasks

DirectorUpdate

UniTaskLoopRunnerLastYieldUpdate

UniTaskLoopRunnerLastUpdate

blog_unitask14.md hosted with ❤ by GitHub

[view raw](#)

The MonoBehaviour “Update” is available in ScriptRunBehaviourUpdate, coroutine (yield return null) is “ScriptRunDelayedDynamicFrameRate”, UnitySynchronizationContext is a It’s run by PlayerLoop in “ScriptRunDelayedTasks”.

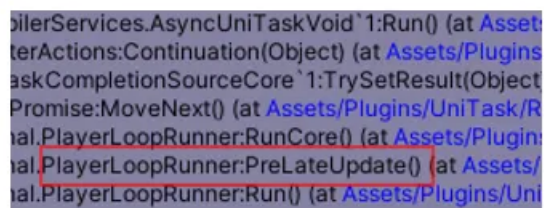
If you look at it this way, there’s nothing special about Unity’s coroutine.

PlayerLoop(ScriptRunDelayedDynamicFrameRate) drives IEnumerator, it only calls MoveNext on every frame. UniTask’s custom loop that not much different from a coroutine loop.

However, Unity’s coroutine is an old-fashioned mechanism, and due to the limitations of yield returns and high overhead, it’s not an ideal mechanism.

The use of UniTask as an alternative to coroutines that is not limited to asynchronous processing, it has no limitations and also performance is good. Therefore, I believe that replacing the coroutine with UniTask is a practical choice.

You can also find out which PlayerLoop the script is currently running in by checking the stack trace in the Debug.Log.



```
PlayerServices.AsyncUniTaskVoid`1:Run() (at Assets/Plugins/UniTask/PlayerServices/AsyncUniTaskVoid.cs:100)
Continuation:ContinueFrom() (at Assets/Plugins/UniTask/PlayerServices/Continuation.cs:100)
TaskCompletionSourceCore`1:TrySetResult(Object) (at Assets/Plugins/UniTask/PlayerServices/TaskCompletionSourceCore.cs:100)
Promise:MoveNext() (at Assets/Plugins/UniTask/PlayerServices/Promise.cs:100)
PlayerLoopRunner:RunCore() (at Assets/Plugins/UniTask/PlayerLoopRunner.cs:100)
PlayerLoopRunner:PreLateUpdate() (at Assets/Plugins/UniTask/PlayerLoopRunner.cs:100)
PlayerLoopRunner:Run() (at Assets/Plugins/UniTask/PlayerLoopRunner.cs:100)
```

If it’s running in UniTask’s PlayerLoop, the second position from the bottom PlayerLoop is displayed (in this case, PlayerLoopTiming. PreLateUpdate).

Asynchronous LINQ

With the introduction of C# 8.0 support in Unity 2020.2.0a12, asynchronous Stream is now possible. For example, the following statement replaces Update()!

```
1 // Unity 2020.2.0a12, C# 8.0
2 await foreach (var _ in UniTaskAsyncEnumerable.EveryUpdate(token))
3 {
4     Debug.Log("Update() " + Time.frameCount);
5 }
```

blog_unitask15.cs hosted with ❤ by GitHub

[view raw](#)

As you can see, C# 8.0 is too early, but in C# 7.3 environments, the ForEachAsync method is used in We'll use this one for practical purposes, since it can be moved in almost the same way.

```
1 // C# 7.3(Unity 2018.3~)
2 await UniTaskAsyncEnumerable.EveryUpdate(token).ForEachAsync(_ =>
3 {
4     Debug.Log("Update() " + Time.frameCount);
5 });
```

blog_unitask16.cs hosted with ❤ by GitHub

[view raw](#)

Also, UniTaskAsyncEnumerable has LINQ that same as the `IEnumerable`'s LINQ or Rx of `IObservable`.

Asynchronous LINQ has all standard LINQ query operators that can be applied to asynchronous streams. For example, the following code runs once every two times on a button-click asynchronous stream Where filtered.

```
1 await okButton.OnClickAsyncEnumerable().Where((x, i) => i % 2 == 0).ForEachAsync(_ =>
2 {
3 });
```

blog_unitask17.cs hosted with ❤ by GitHub

[view raw](#)

It is similar to UniRx (Reactive Extensions), but with Rx being a push type Asynchronous stream, where as UniTaskAsyncEnumerable is a Pull type asynchronous stream.

And we also have a Channel that same as .NET Core's System.Threading.Channels. It's like a Golang channel but arranged for async/await. Channel.Reader.ReadAllAsync returns IUniTaskAsyncEnumerable, so it can consume by foreach, or asynchronous LINQ.

For example, when combined with Publish, UniTask's proprietary operator, it can be converted to a Pub/Sub implementation.

```

1  public class AsyncMessageBroker<T> : IDisposable
2  {
3      Channel<T> channel;
4
5      IConnectableUniTaskAsyncEnumerable<T> multicastSource;
6      IDisposable connection;
7
8      public AsyncMessageBroker()
9      {
10         channel = Channel.CreateSingleConsumerUnbounded<T>();
11         multicastSource = channel.Reader.ReadAllAsync().Publish();
12         connection = multicastSource.Connect();
13     }
14
15     public void Publish(T value)
16     {
17         channel.Writer.TryWrite(value);
18     }
19
20     public IUniTaskAsyncEnumerable<T> Subscribe()
21     {
22         return multicastSource;
23     }
24
25     public void Dispose()
26     {
27         channel.Writer.TryComplete();
28         connection.Dispose();
29     }
30 }

```

blog_unitask18.cs hosted with ❤ by GitHub

[view raw](#)

Enhanced await support

In default, await support for AsyncOperation, ResourceRequest, AssetBundleRequest, AssetBundleCreateRequest, and UnityWebRequestAsyncOperation are all included in UniTask.

I've organized them into three patterns.

- await asyncOperation;
- asyncOperation.WithCancellation(CancellationToken);
- asyncOperation.ToUniTask(IProgress, PlayerLoopTiming, CancellationToken);

blog_unitask19.md hosted with ❤ by GitHub

[view raw](#)

In addition to await directly, you can also call the WithCancellation method to get the support of the cancellation. Also, the return value of this is UniTask, so it can be used with WhenAll for parallel processing.

ToUniTask is a more advanced option than WithCancellation. Progress callbacks, PlayerLoop to run, and It is now a method that can pass a CancellationToken.

In addition, support for DOTween and Addressable has been added as an external asset, and For example, the following implementations are possible in DOTween.

```
1 // sequential
2 await transform.DOMoveX(2, 10);
3 await transform.DOMoveZ(5, 20);
4
5 // parallel
6 var ct = this.GetCancellationTokenOnDestroy();
7
8 await UniTask.WhenAll(
9     transform.DOMoveX(10, 3).WithCancellation(ct),
10    transform.DOScale(10, 3).WithCancellation(ct));
```

blog_unitask20.cs hosted with ❤ by GitHub

[view raw](#)

And all UniTasks can be monitored for usage with the UniTaskTracker.

UniTask Tracker					⋮	□	✕
Enable AutoReload		Enable Tracking	Enable StackTrace			Reload	GC.Collect
TaskType	Elapse	Status	Position				
DOTweenAsyncExtensions.TweenConfiguredSource	01.39	Pending	async UniTaskVoid SandboxMain.Start() (at Assets//Scenes/				
DOTweenAsyncExtensions.TweenConfiguredSource	01.39	Pending	async UniTaskVoid SandboxMain.Start() (at Assets//Scenes/				
UniTask.WhenAllPromise	01.39	Pending	async UniTaskVoid SandboxMain.Start() (at Assets//Scenes/				
Select`2_Select<AsyncUnit, AsyncUnit>	01.38	Pending	async void Linq.ForEach+<ForEachAsync>d__0<AsyncUnit>				
EveryUpdate._EveryUpdate	01.38	Pending	IUniTaskAsyncEnumerator<AsyncUnit> Linq.EveryUpdate.Ge				
AsyncUniTask<ForEach.<ForEachAsync>d__0<AsyncU	01.38	Pending	async void Linq.ForEach+<ForEachAsync>d__0<AsyncUnit>				
UnityEventHandlerAsyncEnumerable.UnityEventHandle	01.38	Pending	UniTask Linq.ForEach.ForEachAsync<AsyncUnit>(IUniTaskA				
AsyncUniTask<ForEach.<ForEachAsync>d__0<AsyncL	01.38	Pending	async void Linq.ForEach+<ForEachAsync>d__0<AsyncUnit>				
</							

This makes it easier to prevent memory leaks.

New from UniTask v2, the .NET Core version of UniTask is now available in NuGet. It has a higher performance than the standard Task/ValueTask, but it is not as easy to use as ignoring the ExecutionContext/SynchronizationContext.

In order to ignore the ExecutionContext, the AysncLocal does not work either. If you want to use it, we recommend you understand the limitations and use it only as a pinpoint.

If you want to use UniTask internally and provide ValueTask as an external API, you can write it in the following way

```
1  public class ZeroAllocAsyncAwaitInDotNetCore
2  {
3      public ValueTask<int> DoAsync(int x, int y)
4      {
5          return Core(this, x, y);
6
7          static async UniTask<int> Core(ZeroAllocAsyncAwaitInDotNetCore self, int x, int y)
8          {
9              // do anything...
10             await Task.Delay(TimeSpan.FromSeconds(x + y));
11             await UniTask.Yield();
12
13             return 10;
14         }
15     }
16 }
```

blog_unitask21.cs hosted with ❤ by GitHub

[view raw](#)

.NET Core version was primarily intended to be available when sharing code with Unity (e.g. for [Cysharp/MagicOnion](#), realtime framework for Unity and .NET Core) as an interface.

You can use UniTask's WhenAll for ValueTask by the [Cysharp/ValueTaskSupplement](#). Please check it.

Conclusion

It's been two years since the release of the first UniTask, and while many games have adopted it, we've been able to there are still many people who don't understand or misunderstand about async/await.

UniTask v2 is the perfect solution for Unity, I hope that many people will now know the power of C#, and async/await.

Unity

Csharp

Linq

Async