UniTask中文文档

发表于 2022-07-16 | 更新于 2022-07-16 | 编程语言 CSharp
| 字数总计: 7.4k | 阅读时长: 31分钟 | 阅读量: 14690

UniTask

为Unity提供一个高性能，0GC的async/await异步方案。

- 基于值类型的 `UniTask<T>` 和自定义的 AsyncMethodBuilder 来实现0GC
- 使所有 Unity 的 AsyncOperations 和 Coroutines 可等待
- 基于 PlayerLoop 的任务(`UniTask.Yield` , `UniTask.Delay` , `UniTask.DelayFrame` , etc...) 可以替换所有协程操作
- 对MonoBehaviour 消息事件和 uGUI 事件进行 可等待/异步枚举 拓展
- 完全在 Unity 的 PlayerLoop 上运行，因此不使用Thread，并且同样能在 WebGL、wasm 等平台上运行。
- 带有 Channel 和 AsyncReactiveProperty的异步 LINQ，
- 提供一个 TaskTracker EditorWindow 以追踪所有UniTask分配来预防内存泄漏
- 与原生 Task/ValueTask/IValueTaskSource 高度兼容的行为

有关技术细节，请参阅博客文章：UniTask v2 — Unity 的0GC async/await 以及 异步LINQ 的使用
有关高级技巧，请参阅博客文章：通过异步装饰器模式扩展 UnityWebRequest — UniTask 的高级技术

入门

通过UniTask/releases页面中提供的UPM 包或资产包 (`UniTask.*.*.*.unitypackage`)安装。

```
// 使用UniTask所需的命名空间
using Cysharp.Threading.Tasks;

// 你可以返回一个形如 UniTask<T>(或 UniTask) 的类型，这种类型事为Unity定制的，作为替代原生Task<T>的轻量
// 为Unity集成的 0GC，快速调用，0消耗的 async/await 方案
async UniTask<string> DemoAsync()
{
    // 你可以等待一个Unity异步对象
```

```
    var asset = await Resources.LoadAsync<TextAsset>("foo");
    var txt = (await UnityWebRequest.Get("https://...").SendWebRequest()).downloadHandler.text;
    await SceneManager.LoadSceneAsync("scene2");

    // .WithCancellation 会启用取消功能，GetCancellationTokenOnDestroy 表示获取一个依赖对象生命周期的
    var asset2 = await Resources.LoadAsync<TextAsset>("bar").WithCancellation(this.GetCancellatio

    // .ToUniTask 可接收一个 progress 回调以及一些配置参数，Progress.Create是IProgress<T>的轻量级替代
    var asset3 = await Resources.LoadAsync<TextAsset>("baz").ToUniTask(Progress.Create<float>(x =

    // 等待一个基于帧的延时操作（就像一个协程一样）
    await UniTask.DelayFrame(100);

    // yield return new WaitForSeconds/WaitForSecondsRealtime 的替代方案
    await UniTask.Delay(TimeSpan.FromSeconds(10), ignoreTimeScale: false);

    // 可以等待任何 playerloop 的生命周期(PreUpdate, Update, LateUpdate, 等...)
    await UniTask.Yield(PlayerLoopTiming.PreLateUpdate);

    // yield return null 替代方案
    await UniTask.Yield();
    await UniTask.NextFrame();

    // WaitForEndOfFrame 替代方案 (需要 MonoBehaviour(CoroutineRunner))
    await UniTask.WaitForEndOfFrame(this); // this 是一个 MonoBehaviour

    // yield return new WaitForFixedUpdate 替代方案，(和 UniTask.Yield(PlayerLoopTiming.FixedUpdat
    await UniTask.WaitForFixedUpdate();

    // yield return WaitUntil 替代方案
    await UniTask.WaitUntil(() => isActive == false);

    // WaitUntil拓展，指定某个值改变时触发
    await UniTask.WaitUntilValueChanged(this, x => x.isActive);

    // 你可以直接 await 一个 IEnumerator 协程
    await FooCoroutineEnumerator();

    // 你可以直接 await 一个原生 task
    await Task.Run(() => 100);

    // 多线程示例，在此行代码后的内容都运行在一个线程池上
    await UniTask.SwitchToThreadPool();

    /* 工作在线程池上的代码 */

    // 转回主线程
    await UniTask.SwitchToMainThread();

    // 获取异步的 webrequest
    async UniTask<string> GetTextAsync(UnityWebRequest req)
    {
        var op = await req.SendWebRequest();
        return op.downloadHandler.text;
    }

    var task1 = GetTextAsync(UnityWebRequest.Get("http://google.com"));
    var task2 = GetTextAsync(UnityWebRequest.Get("http://bing.com"));
```

```
67      var task3 = GetTextAsync(UnityWebRequest.Get("http://yahoo.com"));
68
69      // 构造一个async-wait，并通过元组语义轻松获取所有结果
70      var (google, bing, yahoo) = await UniTask.WhenAll(task1, task2, task3);
71
72      // WhenAll简写形式
73      var (google2, bing2, yahoo2) = await (task1, task2, task3);
74
75      // 返回一个异步值，或者你也可以使用`UniTask`(无结果)，`UniTaskVoid`(协程，不可等待)
76      return (asset as TextAsset)?.text ?? throw new InvalidOperationException("Asset not found");
77  }
```

UniTask 和 AsyncOperation 基础知识

UniTask 功能依赖于 C# 7.0(task-like custom async method builder feature) 所以需要的 Unity 最低版本是 `Unity 2018.3` ，官方支持的最低版本是 `Unity 2018.4.13f1` .

为什么需要 UniTask（自定义task对象）？因为原生 Task 太重，与 Unity 线程（单线程）相性不好。UniTask 不使用线程和 SynchronizationContext/ExecutionContext，因为 Unity 的异步对象由 Unity 的引擎层自动调度。它实现了更快和更低的分配，并且与Unity完全兼容。

你可以在使用 `using Cysharp.Threading.Tasks;` 时对 `AsyncOperation` ， `ResourceRequest` ， `AssetBundleRequest` ， `AssetBundleCreateRequest` ， `UnityWebRequestAsyncOperation` ， `AsyncGPUReadbackRequest` ， `IEnumerator` 以及其他的异步操作进行 await

UniTask 提供了三种模式的扩展方法。

```
1   * await asyncOperation;
2   * .WithCancellation(CancellationToken);
3   * .ToUniTask(IProgress, PlayerLoopTiming, CancellationToken);
```

`WithCancellation` 是 `ToUniTask` 的简化版本，两者都返回 `UniTask` 。有关cancellation的详细信息，请参阅：取消和异常处理部分。

> 注意：await 会在 PlayerLoop 执行await对象的相应native生命周期方法时返回（如果条件满足的话），而 WithCancellation 和 ToUniTask 是从指定的 PlayerLoop 生命周期执行时返回。有关 PlayLoop生命周期 的详细信息，请参阅：PlayerLoop部分。

> 注意： AssetBundleRequest 有 `asset` 和 `allAssets` ，默认 await 返回 `asset` 。如果你想得到 `allAssets` ，你可以使用 `AwaitForAllAssets()` 方法。

`UniTask` 可以使用 `UniTask.WhenAll` 和 `UniTask.WhenAny` 等实用函数。它们就像 `Task.WhenAll` / `Task.WhenAny` 。但它们会返回内容，这很有用。它们会返回值元组，因此您可以传递多种类型并解构每个结果。

```
1   public async UniTaskVoid LoadManyAsync()
2   {
3       // 并行加载.
4       var (a, b, c) = await UniTask.WhenAll(
5           LoadAsSprite("foo"),
6           LoadAsSprite("bar"),
7           LoadAsSprite("baz"));
```

```
}

async UniTask<Sprite> LoadAsSprite(string path)
{
    var resource = await Resources.LoadAsync<Sprite>(path);
    return (resource as Sprite);
}
```

如果你想转换一个回调逻辑块，让它变成UniTask的话，可以使用 `UniTaskCompletionSource<T>`（`TaskCompletionSource<T>` 的轻量级魔改版）

```
public UniTask<int> WrapByUniTaskCompletionSource()
{
    var utcs = new UniTaskCompletionSource<int>();

    // 当操作完成时，调用 utcs.TrySetResult();
    // 当操作失败时，调用 utcs.TrySetException();
    // 当操作取消时，调用 utcs.TrySetCanceled();

    return utcs.Task; //本质上就是返回了一个UniTask<int>
}
```

您可以进行如下转换

- `Task -> UniTask`：使用 `AsUniTask`
- `UniTask -> UniTask<AsyncUnit>`：使用 `AsAsyncUnitUniTask`
- `UniTask<T> -> UniTask`：使用 `AsUniTask`，这两者的转换是无消耗的

如果你想将异步转换为协程，你可以使用 `.ToCoroutine()`，如果你只想允许使用协程系统，这很有用。

UniTask 不能await两次。这是与.NET Standard 2.1 中引入的ValueTask/IValueTaskSource相同的约束。

> 永远不应在 ValueTask 实例上执行以下操作：
>
> - 多次await实例。
> - 多次调用 AsTask。
> - 在操作尚未完成时调用 .Result 或 .GetAwaiter().GetResult()，多次调用也是不允许的。
> - 混用上述行为更是不被允许的。
>
> 如果您执行上述任何操作，则结果是未定义。

```
var task = UniTask.DelayFrame(10);
await task;
await task; // 寄了，抛出异常
```

如果实在需要多次await一个异步操作，可以使用 `UniTask.Lazy` 来支持多次调用。 `.Preserve()` 同样允许多次调用（由UniTask内部缓存的结果）。这种方法在函数范围内有多个调用时很有用。

同样的 `UniTaskCompletionSource` 可以在同一个地方被await多次，或者在很多不同的地方被await。

Cancellation and Exception handling

一些 UniTask 工厂方法有一个 `CancellationToken cancellationToken = default` 参数。Unity 的一些异步操作也有 `WithCancellation(CancellationToken)` 和 `ToUniTask(..., CancellationToken cancellation = default)` 拓展方法。

可以传递原生 `CancellationTokenSource` 给参数CancellationToken

```
var cts = new CancellationTokenSource();

cancelButton.onClick.AddListener(() =>
{
    cts.Cancel();
});

await UnityWebRequest.Get("http://google.co.jp").SendWebRequest().WithCancellation(cts.Token);

await UniTask.DelayFrame(1000, cancellationToken: cts.Token);
```

CancellationToken 可以由 `CancellationTokenSource` 或 MonoBehaviour 的 `GetCancellationTokenOnDestroy` 扩展方法创建。

```
// 这个CancellationTokenSource和this GameObject生命周期相同，当this GameObject Destroy的时候，就会执行
await UniTask.DelayFrame(1000, cancellationToken: this.GetCancellationTokenOnDestroy());
```

对于链式取消，所有异步方法都建议最后一个参数接受 `CancellationToken cancellationToken` ，并将 `CancellationToken` 从头传递到尾。

```
await FooAsync(this.GetCancellationTokenOnDestroy());

// ---

async UniTask FooAsync(CancellationToken cancellationToken)
{
    await BarAsync(cancellationToken);
}

async UniTask BarAsync(CancellationToken cancellationToken)
{
    await UniTask.Delay(TimeSpan.FromSeconds(3), cancellationToken);
}
```

`CancellationToken` 表示异步的生命周期。您可以使用自定义的生命周期，而不是默认的 CancellationTokenOnDestroy。

```
public class MyBehaviour : MonoBehaviour
{
    CancellationTokenSource disableCancellation = new CancellationTokenSource();
    CancellationTokenSource destroyCancellation = new CancellationTokenSource();

    private void OnEnable()
    {
        if (disableCancellation != null)
        {
            disableCancellation.Dispose();
```

```
        }
        disableCancellation = new CancellationTokenSource();
    }

    private void OnDisable()
    {
        disableCancellation.Cancel();
    }

    private void OnDestroy()
    {
        destroyCancellation.Cancel();
        destroyCancellation.Dispose();
    }
}
```

当检测到取消时，所有方法都会向上游抛出并传播 `OperationCanceledException` 。当异常（不限于 `OperationCanceledException` ）没有在异步方法中处理时，它将最终传播到 `UniTaskScheduler.UnobservedTaskException` 。接收到的未处理异常的默认行为是将日志写入异常。可以使用 `UniTaskScheduler.UnobservedExceptionWriteLogType` 更改日志级别。如果要使用自定义行为，请为 `UniTaskScheduler.UnobservedTaskException.` 设置一个委托

而 `OperationCanceledException` 是一个特殊的异常，会被 `UnobservedTaskException` .无视

如果要取消异步 UniTask 方法中的行为，请手动抛出 `OperationCanceledException` 。

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

如果您处理异常但想忽略（传播到全局cancellation处理的地方），请使用异常过滤器。

```
public async UniTask<int> BarAsync()
{
    try
    {
        var x = await FooAsync();
        return x * 2;
    }
    catch (Exception ex) when (!(ex is OperationCanceledException)) // when (ex is not OperationC
    {
        return -1;
    }
}
```

throws/catch `OperationCanceledException` 有点重，所以如果性能是一个问题，请使用 `UniTask.SuppressCancellationThrow` 以避免 OperationCanceledException 抛出。它将返回 `(bool IsCanceled, T Result)` 而不是抛出。

```
var (isCanceled, _) = await UniTask.DelayFrame(10, cancellationToken: cts.Token).SuppressCancellat
if (isCanceled)
{
```

```
    // ...
}
```

注意：仅当您在原方法直接调用SuppressCancellationThrow时才会抑制异常抛出。否则，返回值将被转换，且整个管道不会抑制throws。

超时处理

超时是取消的一种变体。您可以通过 `CancellationTokenSouce.CancelAfterSlim(TimeSpan)` 设置超时并将CancellationToken 传递给异步方法。

```
var cts = new CancellationTokenSource();
cts.CancelAfterSlim(TimeSpan.FromSeconds(5)); // 5sec timeout.

try
{
    await UnityWebRequest.Get("http://foo").SendWebRequest().WithCancellation(cts.Token);
}
catch (OperationCanceledException ex)
{
    if (ex.CancellationToken == cts.Token)
    {
        UnityEngine.Debug.Log("Timeout");
    }
}
```

> `CancellationTokenSouce.CancelAfter` 是一个原生的api。但是在 Unity 中你不应该使用它，因为它依赖于线程计时器。 `CancelAfterSlim` 是 UniTask 的扩展方法，它使用 PlayerLoop 代替。
>
> 如果您想将超时与其他cancellation一起使用，请使用 `CancellationTokenSource.CreateLinkedTokenSource` .

```
var cancelToken = new CancellationTokenSource();
cancelButton.onClick.AddListener(()=>
{
    cancelToken.Cancel(); // 点击按钮后取消
});

var timeoutToken = new CancellationTokenSource();
timeoutToken.CancelAfterSlim(TimeSpan.FromSeconds(5)); // 设置5s超时

try
{
    // 链接token
    var linkedTokenSource = CancellationTokenSource.CreateLinkedTokenSource(cancelToken.Token, ti

    await UnityWebRequest.Get("http://foo").SendWebRequest().WithCancellation(linkedTokenSource.T
}
catch (OperationCanceledException ex)
{
    if (timeoutToken.IsCancellationRequested)
    {
        UnityEngine.Debug.Log("Timeout.");
```

```
    }
    else if (cancelToken.IsCancellationRequested)
    {
        UnityEngine.Debug.Log("Cancel clicked.");
    }
}
```

为优化减少每个调用异步方法超时的 CancellationTokenSource 分配，您可以使用 UniTask 的 `TimeoutController`。

```
TimeoutController timeoutController = new TimeoutController(); // 复用timeoutController

async UniTask FooAsync()
{
    try
    {
        // 你可以通过 timeoutController.Timeout(TimeSpan) 传递到 cancellationToken.
        await UnityWebRequest.Get("http://foo").SendWebRequest()
            .WithCancellation(timeoutController.Timeout(TimeSpan.FromSeconds(5)));
        timeoutController.Reset(); // 当await完成后调用Reset（停止超时计时器，并准备下一次复用）
    }
    catch (OperationCanceledException ex)
    {
        if (timeoutController.IsTimeout())
        {
            UnityEngine.Debug.Log("timeout");
        }
    }
}
```

如果您想将超时与其他取消源一起使用，请使用 `new TimeoutController(CancellationToken)`。

```
TimeoutController timeoutController;
CancellationTokenSource clickCancelSource;

void Start()
{
    this.clickCancelSource = new CancellationTokenSource();
    this.timeoutController = new TimeoutController(clickCancelSource);
}
```

注意：UniTask 有 `.Timeout`，`.TimeoutWithoutException` 方法，但是，如果可能，不要使用这些，请通过 `CancellationToken`。由于 `.Timeout` 作用在task外部，无法停止超时任务。 `.Timeout` 表示超时时忽略结果。如果您将一个 `CancellationToken` 传递给该方法，它将从任务内部执行，因此可以停止正在运行的任务。

进度

一些Unity的异步操作具有 `ToUniTask(IProgress<float> progress = null, ...)` 扩展方法。

```
var progress = Progress.Create<float>(x => Debug.Log(x));

var request = await UnityWebRequest.Get("http://google.co.jp")
    .SendWebRequest()
    .ToUniTask(progress: progress);
```

您不应该使用原生的 `new System.Progress<T>`，因为它每次都会导致GC分配。改为使用 `Cysharp.Threading.Tasks.Progress`。这个 progress factory 有两个方法， `Create` 和 `CreateOnlyValueChanged`. `CreateOnlyValueChanged` 仅在进度值更新时调用。

为调用者实现 IProgress 接口会更好，因为这样可以没有 lambda 分配。

```
public class Foo : MonoBehaviour, IProgress<float>
{
    public void Report(float value)
    {
        UnityEngine.Debug.Log(value);
    }

    public async UniTaskVoid WebRequest()
    {
        var request = await UnityWebRequest.Get("http://google.co.jp")
            .SendWebRequest()
            .ToUniTask(progress: this);
    }
}
```

PlayerLoop

UniTask 在自定义PlayerLoop上运行。UniTask 的基于 playerloop 的方法（例如 `Delay`、 `DelayFrame`、 `asyncOperation.ToUniTask` 等）接受这个 `PlayerLoopTiming`。

```
public enum PlayerLoopTiming
{
    Initialization = 0,
    LastInitialization = 1,

    EarlyUpdate = 2,
    LastEarlyUpdate = 3,

    FixedUpdate = 4,
    LastFixedUpdate = 5,

    PreUpdate = 6,
    LastPreUpdate = 7,

    Update = 8,
    LastUpdate = 9,

    PreLateUpdate = 10,
    LastPreLateUpdate = 11,

    PostLateUpdate = 12,
    LastPostLateUpdate = 13

#if UNITY_2020_2_OR_NEWER
    TimeUpdate = 14,
    LastTimeUpdate = 15,
#endif
}
```

它表示何时运行，您可以检查PlayerLoopList.md到 Unity 的默认 playerloop 并注入 UniTask 的自定义循环。

`PlayerLoopTiming.Update` 与协程中的 `yield return null` 类似，但在 Update(Update 和 uGUI 事件(button.onClick, etc...)前被调用（在 `ScriptRunBehaviourUpdate` 时被调用），yield return null 在 `ScriptRunDelayedDynamicFrameRate` 时被调用。 `PlayerLoopTiming.FixedUpdate` 类似于 `WaitForFixedUpdate` 。

> `PlayerLoopTiming.LastPostLateUpdate` 不等同于协程的 `yield return new WaitForEndOfFrame()` . 协程的 WaitForEndOfFrame 似乎在 PlayerLoop 完成后运行。一些需要协程结束帧(`Texture2D.ReadPixels` , `ScreenCapture.CaptureScreenshotAsTexture` , `CommandBuffer` , 等) 的方法在 async/await 时无法正常工作。在这些情况下，请将 MonoBehaviour(coroutine runner) 传递给 `UniTask.WaitForEndOfFrame` . 例如， `await UniTask.WaitForEndOfFrame(this);` 是 `yield return new WaitForEndOfFrame()` 轻量级0GC的替代方案。

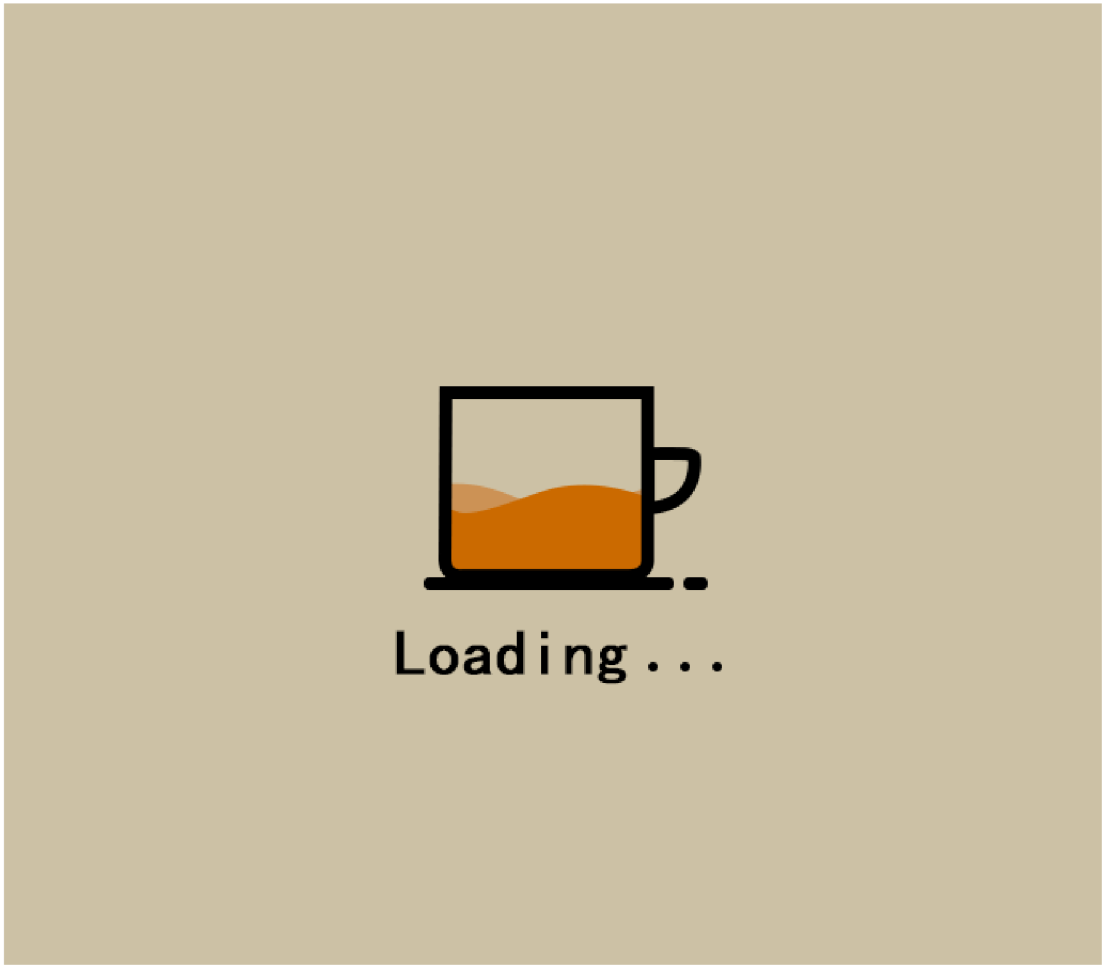
`yield return null` 和 `UniTask.Yield` 相似但不同。 `yield return null` 总是返回下一帧但 `UniTask.Yield` 返回下一个调用。也就是说， `UniTask.Yield(PlayerLoopTiming.Update)` 在 `PreUpdate` 上调用，它返回相同的帧。 `UniTask.NextFrame()` 保证返回下一帧，您可以认为它的行为与 `yield return null` 一致.

> UniTask.Yield(without CancellationToken) 是一种特殊类型，返回 `YieldAwaitable` 并在 YieldRunner 上运行。它是最轻量和最快的。

`AsyncOperation` 在原生生命周期返回。例如，await `SceneManager.LoadSceneAsync` 在 `EarlyUpdate.UpdatePreloading` 时返回，在此之后，加载的场景的 `Start` 方法调用自 `EarlyUpdate.ScriptRunDelayedStartupFrame` 。同样的， `await UnityWebRequest` 在 `EarlyUpdate.ExecuteMainThreadJobs` 时返回.

在 UniTask 中，await 直接使用原生生命周期， `WithCancellation` 和 `ToUniTask` 可以指定使用的原生生命周期。这通常不会有问题，但是 `LoadSceneAsync` 在等待之后，它会导致开始和继续的不同顺序。所以建议不要使用 `LoadSceneAsync.ToUniTask` 。

在堆栈跟踪中，您可以检查它在 playerloop 中的运行位置。



默认情况下，UniTask 的 PlayerLoop 初始化在 `[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]`.

在 BeforeSceneLoad 中调用方法的顺序是不确定的，所以如果你想在其他 BeforeSceneLoad 方法中使用 UniTask，你应该尝试在此之前初始化它。

```
// AfterAssembliesLoaded 表示将会在 BeforeSceneLoad之前调用
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterAssembliesLoaded)]
public static void InitUniTaskLoop()
{
    var loop = PlayerLoop.GetCurrentPlayerLoop();
    Cysharp.Threading.Tasks.PlayerLoopHelper.Initialize(ref loop);
}
```

如果您导入 Unity 的 `Entities` 包，则会将自定义playerloop重置为默认值 `BeforeSceneLoad` 并注入 ECS 的循环。当 Unity 在 UniTask 的 initialize 方法之后调用 ECS 的 inject 方法时，UniTask 将不再工作。

为了解决这个问题，您可以在 ECS 初始化后重新初始化 UniTask PlayerLoop。

```
// 获取ECS Loop.
var playerLoop = ScriptBehaviourUpdateOrder.CurrentPlayerLoop;

// 设置UniTask PlayerLoop
PlayerLoopHelper.Initialize(ref playerLoop);
```

您可以通过调用 `PlayerLoopHelper.IsInjectedUniTaskPlayerLoop()` 来诊断 UniTask 的PlayerLoop是否准备就绪。并且 `PlayerLoopHelper.DumpCurrentPlayerLoop` 还会将所有当前PlayerLoop记录到控制台。

```
void Start()
{
    UnityEngine.Debug.Log("UniTaskPlayerLoop ready? " + PlayerLoopHelper.IsInjectedUniTaskPlayerLo
    PlayerLoopHelper.DumpCurrentPlayerLoop();
}
```

您可以通过删除未使用的 PlayerLoopTiming 注入来稍微优化循环成本。您可以在初始化时调用 `PlayerLoopHelper.Initialize(InjectPlayerLoopTimings)` 。

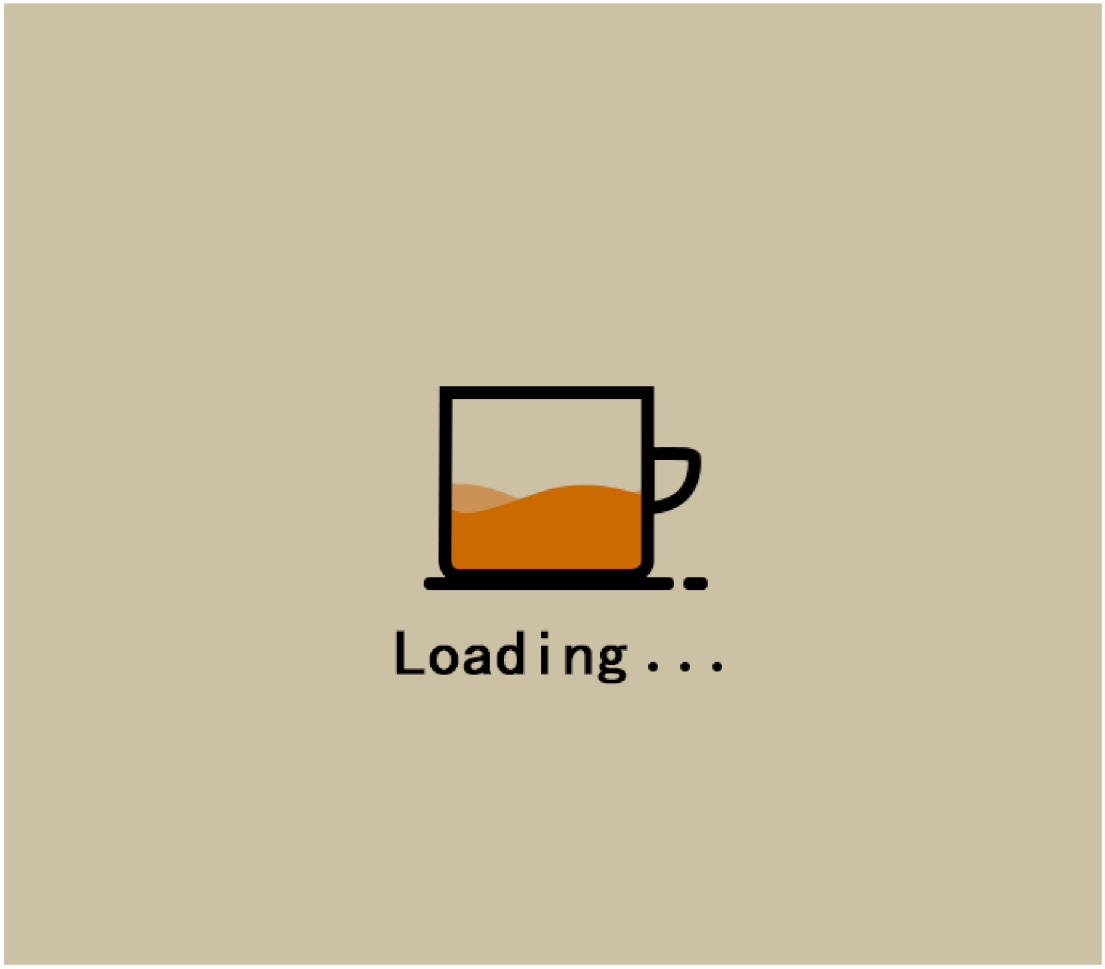
```
var loop = PlayerLoop.GetCurrentPlayerLoop();
PlayerLoopHelper.Initialize(ref loop, InjectPlayerLoopTimings.Minimum); // 最小化 is Update | Fixe
```

`InjectPlayerLoopTimings` 有三个预设， `All` ， `Standard` （除 LastPostLateUpdate 外）， `Minimum` （ `Update | FixedUpdate | LastPostLateUpdate` ）。默认为全部，您可以组合自定义注入时间，例如 `InjectPlayerLoopTimings.Update | InjectPlayerLoopTimings.FixedUpdate | InjectPlayerLoopTimings.PreLateUpdate` .

使用未注入 `PlayerLoopTiming` 的Microsoft.CodeAnalysis.BannedApiAnalyzers可能会出错。例如，您可以为 `InjectPlayerLoopTimings.Minimum` 设置 `BannedSymbols.txt`

```
F:Cysharp.Threading.Tasks.PlayerLoopTiming.Initialization; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastInitialization; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.EarlyUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastEarlyUpdate; Isn't injected this PlayerLoop in this project.d
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastFixedUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.PreUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastPreUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.PreLateUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastPreLateUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.PostLateUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.TimeUpdate; Isn't injected this PlayerLoop in this project.
F:Cysharp.Threading.Tasks.PlayerLoopTiming.LastTimeUpdate; Isn't injected this PlayerLoop in this project.
```

您可以将 `RS0030` 严重性配置为错误。



async void 与 async UniTaskVoid 对比

`async void` 是一个原生的 C# 任务系统，因此它不能在 UniTask 系统上运行。也最好不要使用它。 `async UniTaskVoid` 是 `async UniTask` 的轻量级版本，因为它没有等待完成并立即向 `UniTaskScheduler.UnobservedTaskException` 报告错误. 如果您不需要等待（即发即弃），那么使用 `UniTaskVoid` 会更好。不幸的是，要解除警告，您需要在尾部添加 `Forget()` .

```
public async UniTaskVoid FireAndForgetMethod()
{
    // do anything...
    await UniTask.Yield();
}

public void Caller()
{
    FireAndForgetMethod().Forget();
}
```

UniTask 也有 `Forget` 方法，类似 `UniTaskVoid` 且效果相同。但是如果你完全不需要使用 `await` ， `UniTaskVoid` 会更高效。

```
public async UniTask DoAsync()
{
    // do anything...
    await UniTask.Yield();
}

public void Caller()
{
```

```
9       DoAsync().Forget();
10  }
```

要使用注册到事件的异步 lambda，请不要使用 `async void`。相反，您可以使用 `UniTask.Action` 或 `UniTask.UnityAction`，两者都通过 `async UniTaskVoid` lambda 创建委托。

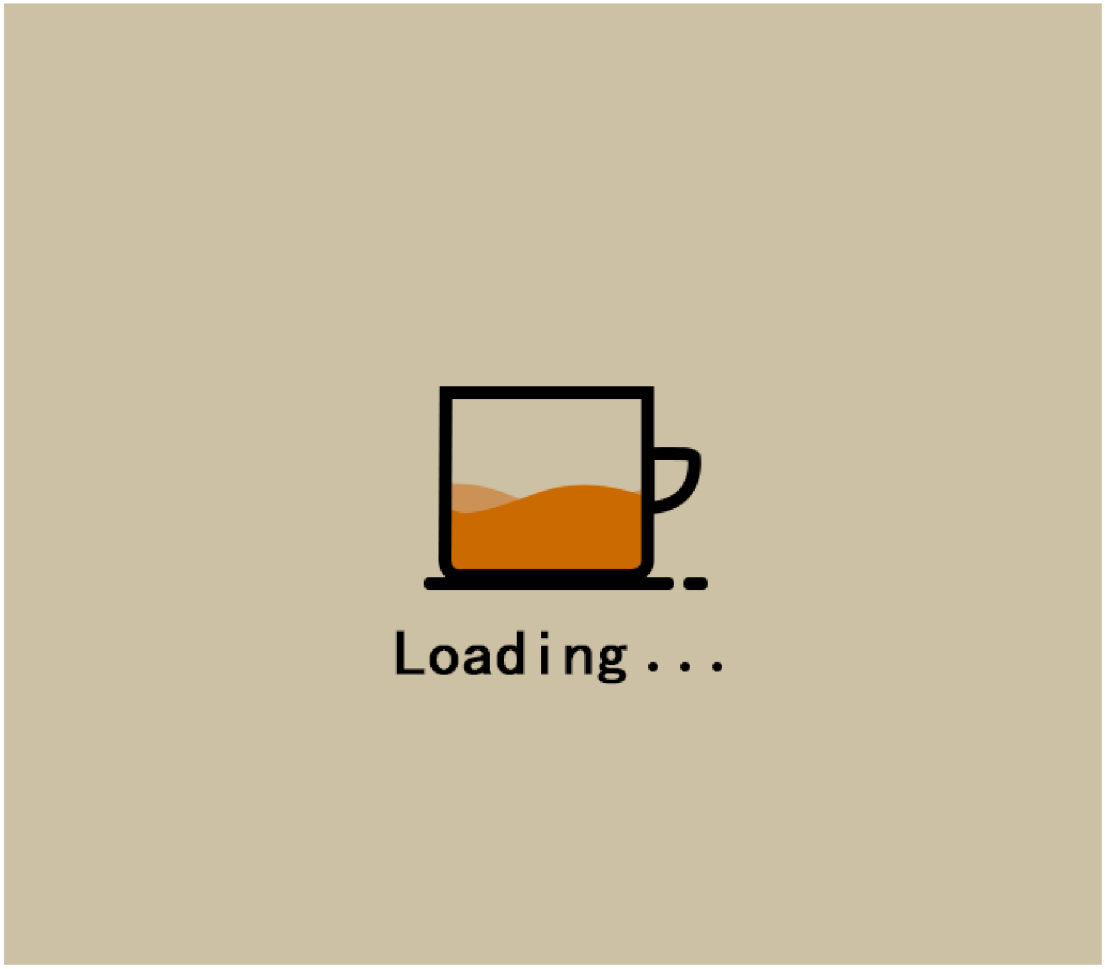
```
1   Action actEvent;
2   UnityAction unityEvent; // UGUI特供
3
4   // 这样是不好的: async void
5   actEvent += async () => { };
6   unityEvent += async () => { };
7
8   // 这样是可以的: 通过lamada创建Action
9   actEvent += UniTask.Action(async () => { await UniTask.Yield(); });
10  unityEvent += UniTask.UnityAction(async () => { await UniTask.Yield(); });
```

`UniTaskVoid` 也可以用在 MonoBehaviour 的 `Start` 方法中。

```
1   class Sample : MonoBehaviour
2   {
3       async UniTaskVoid Start()
4       {
5           // async init code.
6       }
7   }
```

UniTaskTracker

对于检查（泄露的）UniTasks 很有用。您可以在 `Window -> UniTask Tracker` 中打开跟踪器窗口。



- Enable AutoReload(Toggle) - 自动重新加载。
- Reload - 重新加载视图（重新扫描内存中UniTask实例，并刷新界面）。
- GC.Collect - 调用 GC.Collect。
- Enable Tracking(Toggle) - 开始跟踪异步/等待 UniTask。性能影响：低。
- Enable StackTrace(Toggle) - 在任务启动时捕获 StackTrace。性能影响：高。

UniTaskTracker 仅用于调试用途，因为启用跟踪和捕获堆栈跟踪很有用，但会对性能产生重大影响。推荐的用法是启用跟踪和堆栈跟踪以查找任务泄漏并在完成时禁用它们。

外部拓展

默认情况下，UniTask 支持 TextMeshPro（`BindTo(TMP_Text)` 和 `TMP_InputField`，并且TMP_InputField有同原生uGUI `InputField` 类似的事件扩展）、DOTween（`Tween` 作为等待）和Addressables（`AsyncOperationHandle``AsyncOperationHandle<T>` 作为等待）。

在单独的 asmdef 中定义，如 `UniTask.TextMeshPro`，`UniTask.DOTween`，`UniTask.Addressables`.

从包管理器导入包时，会自动启用 TextMeshPro 和 Addressables 支持。但是对于 DOTween 支持，需要 `com.demigiant.dotween` 从OpenUPM导入或定义 `UNITASK_DOTWEEN_SUPPORT` 以启用它。

```
// 动画序列
await transform.DOMoveX(2, 10);
await transform.DOMoveZ(5, 20);

// 并行，并传递cancellation用于取消
```

```
var ct = this.GetCancellationTokenOnDestroy();

await UniTask.WhenAll(
    transform.DOMoveX(10, 3).WithCancellation(ct),
    transform.DOScale(10, 3).WithCancellation(ct));
```

DOTween 支持的默认行为(`await` , `WithCancellation` , `ToUniTask`) await tween 被终止。它适用于 Complete(true/false) 和 Kill(true/false)。但是如果你想重用tweens (`SetAutoKill(false)`)，它就不能按预期工作。如果您想等待另一个时间点，Tween 中存在以下扩展方法， `AwaitForComplete` , `AwaitForPause` , `AwaitForPlay` , `AwaitForRewind` , `AwaitForStepComplete` 。

AsyncEnumerable 和 Async LINQ

Unity 2020.2 支持 C# 8.0，因此您可以使用 `await foreach` . 这是异步时代的新更新符号。

```
// Unity 2020.2, C# 8.0
await foreach (var _ in UniTaskAsyncEnumerable.EveryUpdate(token))
{
    Debug.Log("Update() " + Time.frameCount);
}
```

在 C# 7.3 环境中，您可以使用该 `ForEachAsync` 方法以几乎相同的方式工作。

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

UniTaskAsyncEnumerable 实现异步 LINQ，类似于 LINQ 的 `IEnumerable<T>` 或 Rx 的 `IObservable<T>` 。所有标准 LINQ 查询运算符都可以应用于异步流。例如，以下代码表示如何将 Where 过滤器应用于每两次单击运行一次的按钮单击异步流。

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

Fire and Forget 风格（例如，事件处理），你也可以使用 `Subscribe` .

```
okButton.OnClickAsAsyncEnumerable().Where((x, i) => i % 2 == 0).Subscribe(_ =>
{
});
```

Async LINQ 在 时启用 `using Cysharp.Threading.Tasks.Linq;` ，并且 `UniTaskAsyncEnumerable` 在 `UniTask.Linq` asmdef 中定义。

它更接近 UniRx（Reactive Extensions），但 UniTaskAsyncEnumerable 是pull-base的异步流，而 Rx 是基于push-base异步流。请注意，尽管相似，但特征不同，并且细节的行为也随之不同。

`UniTaskAsyncEnumerable` 是类似的入口点 `Enumerable` 。除了标准查询运算符之外，还有其他 Unity 生成器，例如 `EveryUpdate` 、 `Timer` 、 `TimerFrame` 、 `Interval` 、 `IntervalFrame` 和 `EveryValueChanged` 。并且还添加了额外的 UniTask 原始查询运算符，如 `Append` , `Prepend` , `DistinctUntilChanged` , `ToHashSet` , `Buffer` , `CombineLatest` , `Do` ,

`Never`, `ForEachAsync`, `Pairwise`, `Publish`, `Queue`, `Return`, `SkipUntil`, `TakeUntil`, `SkipUntilCanceled`, `TakeUntilCanceled`, `TakeLast`, `Subscribe`。

以 Func 作为参数的方法具有三个额外的重载，`***Await`, `***AwaitWithCancellation`。

```
Select(Func<T, TR> selector)
SelectAwait(Func<T, UniTask<TR>> selector)
SelectAwaitWithCancellation(Func<T, CancellationToken, UniTask<TR>> selector)
```

如果在 func 方法内部使用 `async`，请使用***Awaitor `***AwaitWithCancellation`。

如何创建异步迭代器：C# 8.0 支持异步迭代器（`async yield return`），但它只允许 `IAsyncEnumerable<T>` 并且当然需要 C# 8.0。UniTask 支持 `UniTaskAsyncEnumerable.Create` 创建自定义异步迭代器的方法。

```
// IAsyncEnumerable, C# 8.0 异步迭代器. ( 不要这样用，因为IAsyncEnumerable不被UniTask控制).
public async IAsyncEnumerable<int> MyEveryUpdate([EnumeratorCancellation]CancellationToken cancel
{
    var frameCount = 0;
    await UniTask.Yield();
    while (!token.IsCancellationRequested)
    {
        yield return frameCount++;
        await UniTask.Yield();
    }
}

// UniTaskAsyncEnumerable.Create 并用 `await writer.YieldAsync` 代替 `yield return`.
public IUniTaskAsyncEnumerable<int> MyEveryUpdate()
{
    // writer(IAsyncWriter<T>) has `YieldAsync(value)` method.
    return UniTaskAsyncEnumerable.Create<int>(async (writer, token) =>
    {
        var frameCount = 0;
        await UniTask.Yield();
        while (!token.IsCancellationRequested)
        {
            await writer.YieldAsync(frameCount++); // instead of `yield return`
            await UniTask.Yield();
        }
    });
}
```

可等待事件

所有 uGUI 组件都实现 `***AsAsyncEnumerable` 了异步事件流的转换。

```
async UniTask TripleClick()
{
    // 默认情况下，使用了button.GetCancellationTokenOnDestroy 来管理异步生命周期
    await button.OnClickAsync();
    await button.OnClickAsync();
    await button.OnClickAsync();
    Debug.Log("Three times clicked");
}

```

```
// 更高效的方法
async UniTask TripleClick()
{
    using (var handler = button.GetAsyncClickEventHandler())
    {
        await handler.OnClickAsync();
        await handler.OnClickAsync();
        await handler.OnClickAsync();
        Debug.Log("Three times clicked");
    }
}

// 使用异步LINQ
async UniTask TripleClick(CancellationToken token)
{
    await button.OnClickAsAsyncEnumerable().Take(3).Last();
    Debug.Log("Three times clicked");
}

// 使用异步LINQ
async UniTask TripleClick(CancellationToken token)
{
    await button.OnClickAsAsyncEnumerable().Take(3).ForEachAsync(_ =>
    {
        Debug.Log("Every clicked");
    });
    Debug.Log("Three times clicked, complete.");
}
```

所有 MonoBehaviour 消息事件都可以转换异步流 `AsyncTriggers` ，可以通过 `using Cysharp.Threading.Tasks.Triggers;` 进行启用，.AsyncTrigger 可以使用 UniTaskAsyncEnumerable 来创建，通过 `GetAsync***Trigger` 触发。

```
var trigger = this.GetOnCollisionEnterAsyncHandler();
await trigger.OnCollisionEnterAsync();
await trigger.OnCollisionEnterAsync();
await trigger.OnCollisionEnterAsync();

// every moves.
await this.GetAsyncMoveTrigger().ForEachAsync(axisEventData =>
{
});
```

`AsyncReactiveProperty` , `AsyncReadOnlyReactiveProperty` 是 UniTask 的 ReactiveProperty 版本。将异步流值绑定到 Unity 组件（Text/Selectable/TMP/Text） `BindTo` 的 `IUniTaskAsyncEnumerable<T>` 扩展方法。

```
var rp = new AsyncReactiveProperty<int>(99);

// AsyncReactiveProperty 本身是 IUniTaskAsyncEnumerable，可以通过LINQ进行查询
rp.ForEachAsync(x =>
{
    Debug.Log(x);
}, this.GetCancellationTokenOnDestroy()).Forget();

rp.Value = 10; // 推送10给所有订阅者
rp.Value = 11; // 推送11给所有订阅者
```

```
11
12  // WithoutCurrent 忽略初始值
13  // BindTo 绑定 stream value 到 unity 组件.
14  rp.WithoutCurrent().BindTo(this.textComponent);
15
16  await rp.WaitAsync(); // 一直等待，直到下一个值被设置
17
18  // 同样支持ToReadOnlyAsyncReactiveProperty
19  var rp2 = new AsyncReactiveProperty<int>(99);
20  var rorp = rp.CombineLatest(rp2, (x, y) => (x, y)).ToReadOnlyAsyncReactiveProperty(CancellationTo
```

在序列中的异步处理完成之前，pull-based异步流不会获取下一个值。这可能会从按钮等推送类型的事件中溢出数据。

```
1  // 在3s完成前，无法获取event
2  await button.OnClickAsAsyncEnumerable().ForEachAwaitAsync(async x =>
3  {
4      await UniTask.Delay(TimeSpan.FromSeconds(3));
5  });
```

它很有用（防止双击），但有时没用。

使用该 `Queue()` 方法还将在异步处理期间对事件进行排队。

```
1  // 异步处理中对message进行排队
2  await button.OnClickAsAsyncEnumerable().Queue().ForEachAwaitAsync(async x =>
3  {
4      await UniTask.Delay(TimeSpan.FromSeconds(3));
5  });
```

或使用 `Subscribe` , fire and forget 风格。

```
1  button.OnClickAsAsyncEnumerable().Subscribe(async x =>
2  {
3      await UniTask.Delay(TimeSpan.FromSeconds(3));
4  });
```

Channel

`Channel` 与System.Threading.Tasks.Channels相同，类似于 GoLang Channel。

目前只支持多生产者、单消费者无界渠道。它可以由 `Channel.CreateSingleConsumerUnbounded<T>()` .

对于 producer(`.Writer`)，用 `TryWrite` 推送值和 `TryComplete` 完成通道。对于 consumer(`.Reader`)，使用 `TryRead` 、 `WaitToReadAsync` 、 `ReadAsync` 和 `Completion` ， `ReadAllAsync` 来读取队列的消息。

`ReadAllAsync` 返回 `IUniTaskAsyncEnumerable<T>` 查询 LINQ 运算符。Reader 只允许单消费者，但使用 `.Publish()` 查询运算符来启用多播消息。例如，制作 pub/sub 实用程序。

```
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(); // Publish returns IConnectableUniTaskAsyncEnumer
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  }
```

单元测试

Unity 的 `[UnityTest]` 属性可以测试协程（IEnumerator）但不能测试异步。 `UniTask.ToCoroutine` 将 async/await 桥接到协程，以便您可以测试异步方法。

```
1   [UnityTest]
2   public IEnumerator DelayIgnore() => UniTask.ToCoroutine(async () =>
3   {
4       var time = Time.realtimeSinceStartup;
5
6       Time.timeScale = 0.5f;
7       try
8       {
9           await UniTask.Delay(TimeSpan.FromSeconds(3), ignoreTimeScale: true);
10
11          var elapsed = Time.realtimeSinceStartup - time;
12          Assert.AreEqual(3, (int)Math.Round(TimeSpan.FromSeconds(elapsed).TotalSeconds, MidpointRo
13      }
14      finally
15      {
16          Time.timeScale = 1.0f;
17      }
18  });
```

UniTask 自己的单元测试是使用 Unity Test Runner 和Cysharp/RuntimeUnitTestToolkit编写的，以与 CI 集成并检查 IL2CPP 是否正常工作。

线程池限制

大多数 UniTask 方法在单个线程 (PlayerLoop) 上运行，只有 `UniTask.Run` （`Task.Run` 等效）和 `UniTask.SwitchToThreadPool` 在线程池上运行。如果您使用线程池，它将无法与 WebGL 等平台兼容。

`UniTask.Run` 现在已弃用。你可以改用 `UniTask.RunOnThreadPool` 。并且还要考虑是否可以使用 `UniTask.Create` 或 `UniTask.Void` 。

IEnumerator.ToUniTask 限制

您可以将协程（IEnumerator）转换为 UniTask（或直接等待），但它有一些限制。

- 不支持 `WaitForEndOfFrame` ， `WaitForFixedUpdate` ， `Coroutine`
- Loop生命周期与 `StartCoroutine` 不一样，它使用指定 `PlayerLoopTiming` 的并且默认情况下，`PlayerLoopTiming.Update` 在 MonoBehaviour `Update` 和 `StartCoroutine` 的循环之前运行。

如果您想要从协程到异步的完全兼容转换，请使用 `IEnumerator.ToUniTask(MonoBehaviour coroutineRunner)` 重载。它在参数 MonoBehaviour 的实例上执行 StartCoroutine 并等待它在 UniTask 中完成。

关于UnityEditor

UniTask 可以像编辑器协程一样在 Unity 编辑器上运行。但是，有一些限制。

- UniTask.Delay 的 DelayType.DeltaTime、UnscaledDeltaTime 无法正常工作，因为它们无法在编辑器中获取 deltaTime。因此在 EditMode 上运行，会自动将 DelayType 更改为 `DelayType.Realtime` 等待正确的时间。
- 所有 PlayerLoopTiming 都在 `EditorApplication.update` 生命周期上运行。
- 带 `-quit` 的 `-batchmode` 带不起作用，因为 Unity `EditorApplication.update` 在单帧后不会运行并退出。相反，不要使用 `-quit` 并手动退出 `EditorApplication.Exit(0)` .

与原生Task API对比

UniTask 有许多原生的 Task-like API。此表显示了一一对应的 API 是什么。

使用原生类型。

.NET Type	UniTask Type
`IProgress<T>`	—
`CancellationToken`	—
`CancellationTokenSource`	—

使用 UniTask 类型.

.NET Type	UniTask Type
`Task` / `ValueTask`	`UniTask`
`Task<T>` / `ValueTask<T>`	`UniTask<T>`
`async void`	`async UniTaskVoid`
`+= async () => { }`	`UniTask.Void` , `UniTask.Action` , `UniTask.UnityAction`

.NET Type	UniTask Type
—	UniTaskCompletionSource
TaskCompletionSource<T>	UniTaskCompletionSource<T> / AutoResetUniTaskCompletionSource<T>
ManualResetValueTaskSourceCore<T>	UniTaskCompletionSourceCore<T>
IValueTaskSource	IUniTaskSource
IValueTaskSource<T>	IUniTaskSource<T>
ValueTask.IsCompleted	UniTask.Status.IsCompleted()
ValueTask<T>.IsCompleted	UniTask<T>.Status.IsCompleted()
new Progress<T>	Progress.Create<T>
CancellationToken.Register(UnsafeRegister)	CancellationToken.RegisterWithoutCaptureExecutionContext
CancellationTokenSource.CancelAfter	CancellationTokenSource.CancelAfterSlim
Channel.CreateUnbounded<T>(false){ SingleReader = true }	Channel.CreateSingleConsumerUnbounded<T>
IAsyncEnumerable<T>	IUniTaskAsyncEnumerable<T>
IAsyncEnumerator<T>	IUniTaskAsyncEnumerator<T>
IAsyncDisposable	IUniTaskAsyncDisposable
Task.Delay	UniTask.Delay
Task.Yield	UniTask.Yield
Task.Run	UniTask.RunOnThreadPool
Task.WhenAll	UniTask.WhenAll
Task.WhenAny	UniTask.WhenAny
Task.CompletedTask	UniTask.CompletedTask
Task.FromException	UniTask.FromException
Task.FromResult	UniTask.FromResult
Task.FromCanceled	UniTask.FromCanceled
Task.ContinueWith	UniTask.ContinueWith
TaskScheduler.UnobservedTaskException	UniTaskScheduler.UnobservedTaskException

池化配置

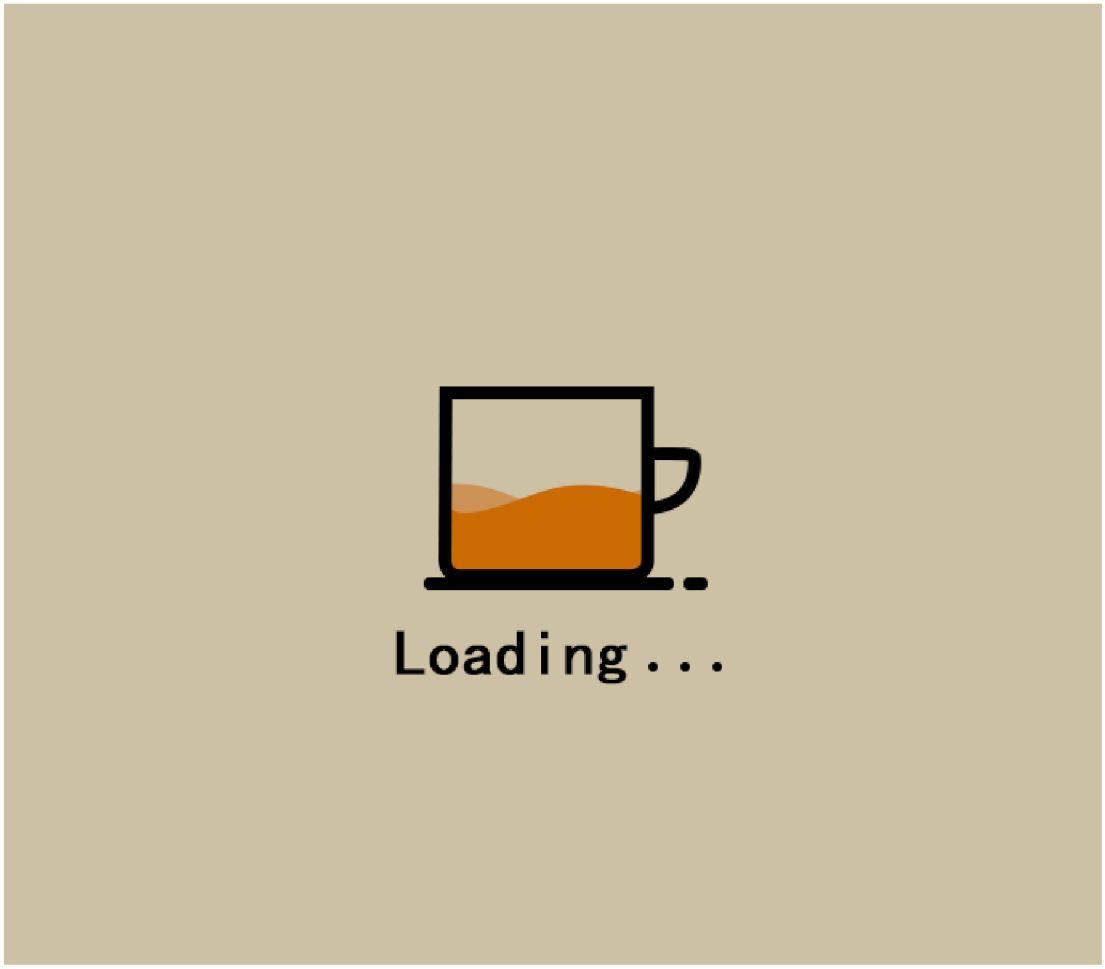
UniTask 积极缓存异步promise对象以实现零分配（有关技术细节，请参阅博客文章UniTask v2 — Unity 的零分配异步/等待，使用异步 LINQ）。默认情况下，它缓存所有promise，但您可以配置 `TaskPool.SetMaxPoolSize` 为您的值，该值表示每种类型的缓存大小。 `TaskPool.GetCacheSizeInfo` 返回池中当前缓存的对象。

```
foreach (var (type, size) in TaskPool.GetCacheSizeInfo())
{
    Debug.Log(type + ":" + size);
}
```

Profiler下的分配

在 UnityEditor 中，分析器显示编译器生成的 AsyncStateMachine 的分配，但它只发生在调试（开发）构建中。C# 编译器将 AsyncStateMachine 生成为 Debug 构建的类和 Release 构建的结构。

Unity 从 2020.1 开始支持代码优化选项（右，页脚）。



您可以将 C# 编译器优化更改为 release 以删除开发版本中的 AsyncStateMachine 分配。此优化选项也可以通过设置 `Compilation.CompilationPipeline-codeOptimization` 和 `Compilation.CodeOptimization` 。

UniTaskSynchronizationContext

Unity 的默认 SynchronizationContext(`UnitySynchronizationContext`) 在性能方面表现不佳。UniTask 绕过 `SynchronizationContext` (和 `ExecutionContext`) 因此它不使用它，但如果存在 `async Task` ，则仍然使用它。 `UniTaskSynchronizationContext` 是 `UnitySynchronizationContext` 性能更好的替代品。

```
public class SyncContextInjecter
{
```

```
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]
    public static void Inject()
    {
        SynchronizationContext.SetSynchronizationContext(new UniTaskSynchronizationContext());
    }
}
```

这是一个可选的选择，并不总是推荐； `UniTaskSynchronizationContext` 性能不如 `async UniTask` ，并且不是完整的 UniTask 替代品。它也不保证与 `UnitySynchronizationContext` 完全兼容

API References

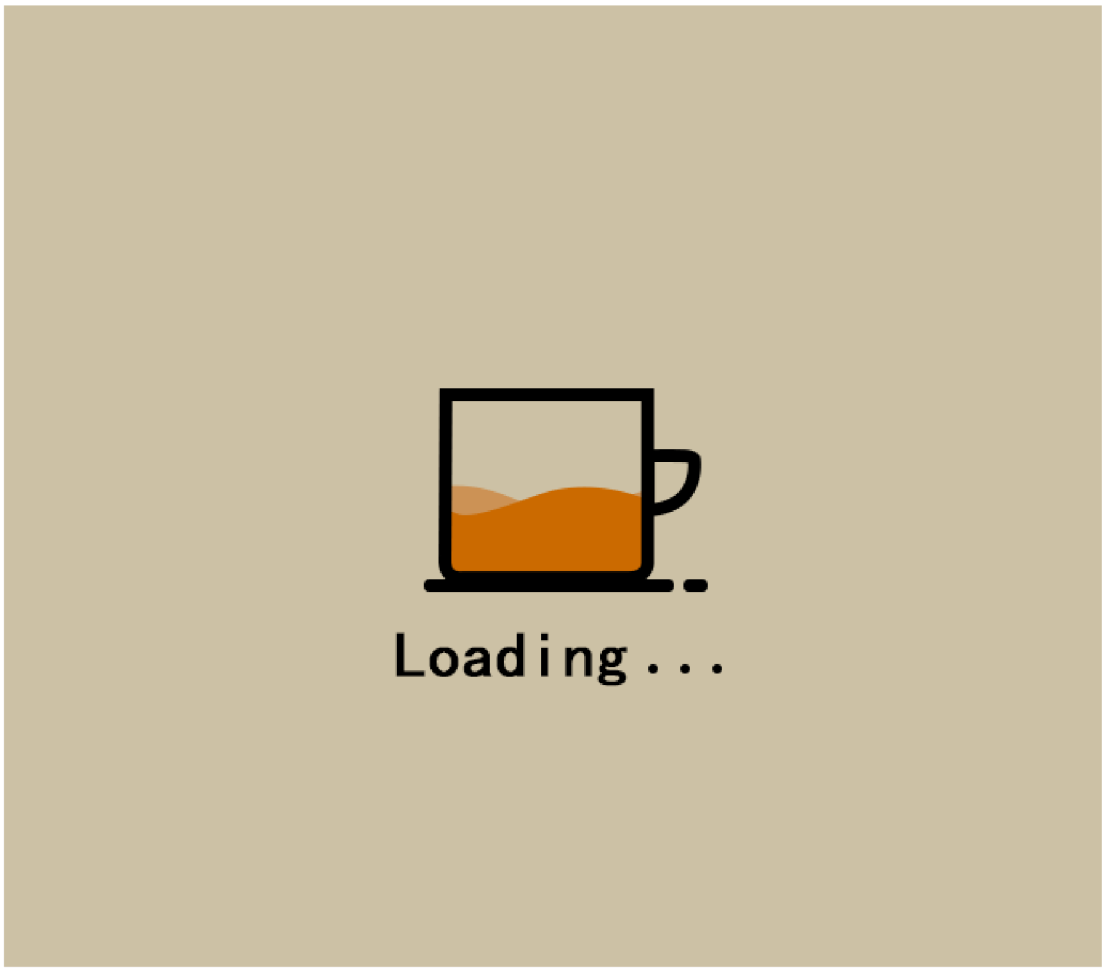
UniTask 的 API 参考由DocFX和Cysharp/DocfXTemplate托管在cysharp.github.io/UniTask上。

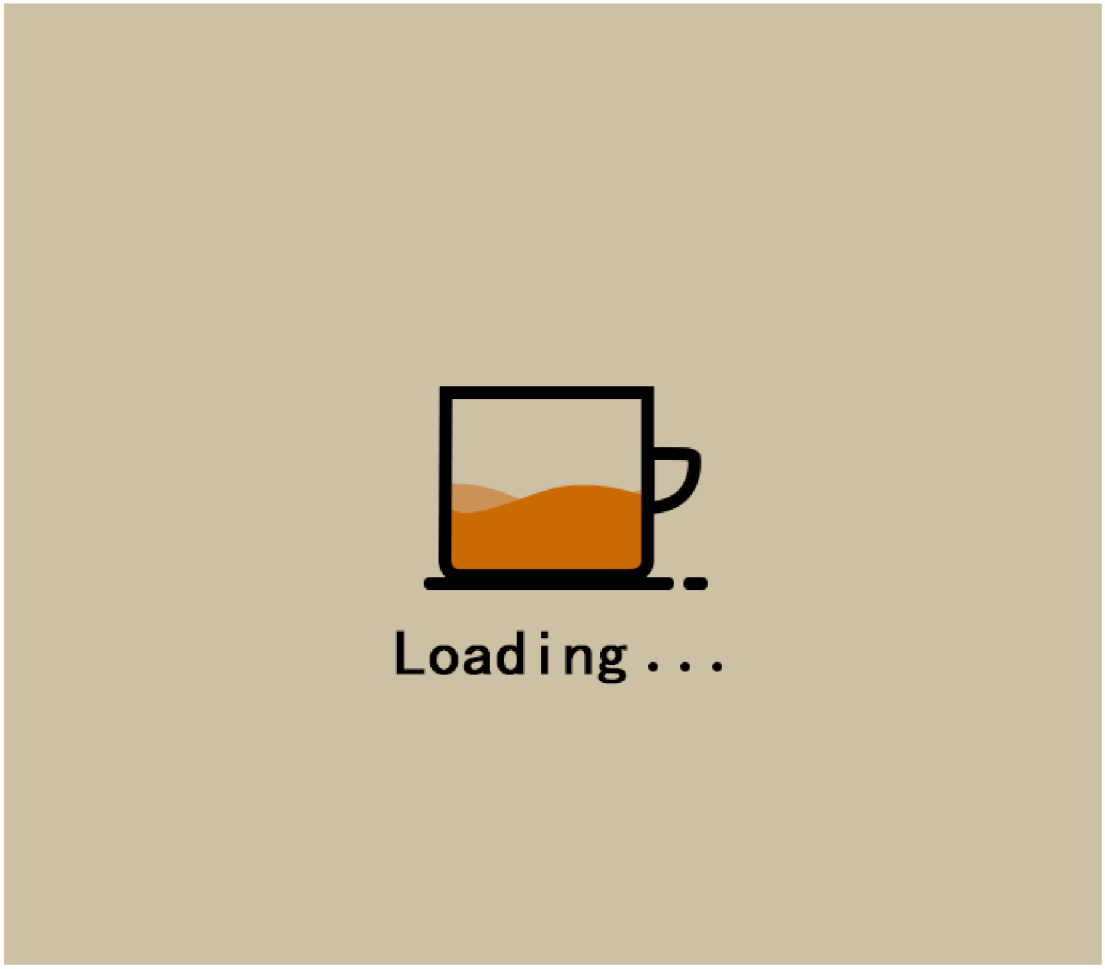
例如，UniTask 的工厂方法可以在UniTask#methods中看到。UniTaskAsyncEnumerable 的工厂/扩展方法可以在UniTaskAsyncEnumerable#methods中看到。

UPM Package

通过 git URL 安装

需要支持 git 包路径查询参数的 unity 版本（Unity >= 2019.3.4f1，Unity >= 2020.1a21）。您可以添加 `https://github.com/Cysharp/UniTask.git?path=src/UniTask/Assets/Plugins/UniTask` 到包管理器





或添加 `"com.cysharp.unitask": "https://github.com/Cysharp/UniTask.git?path=src/UniTask/Assets/Plugins/UniTask"` 到 `Packages/manifest.json`.

如果要设置目标版本，UniTask 使用 `*.*.*` 发布标签，因此您可以指定一个版本，如 `#2.1.0`. 例如 `https://github.com/Cysharp/UniTask.git?path=src/UniTask/Assets/Plugins/UniTask#2.1.0`.

通过 OpenUPM 安装

该软件包在openupm 注册表中可用。建议通过openupm-cli安装。

```
openupm add com.cysharp.unitask
```

.NET Core

对于 .NET Core，请使用 NuGet。

> PM> Install-Package UniTask

.NET Core 版本的 UniTask 是 Unity UniTask 的子集，移除了 PlayerLoop 依赖的方法。

它以比标准 Task/ValueTask 更高的性能运行，但在使用时应注意忽略 ExecutionContext/SynchronizationContext。 `AysncLocal` 也不起作用，因为它忽略了 ExecutionContext。

如果您在内部使用 UniTask，但将 ValueTask 作为外部 API 提供，您可以编写如下（受PooledAwait启发）代码。

```
public class ZeroAllocAsyncAwaitInDotNetCore
{
```

```
    public ValueTask<int> DoAsync(int x, int y)
    {
        return Core(this, x, y);

        static async UniTask<int> Core(ZeroAllocAsyncAwaitInDotNetCore self, int x, int y)
        {
            // do anything...
            await Task.Delay(TimeSpan.FromSeconds(x + y));
            await UniTask.Yield();

            return 10;
        }
    }
}

// UniTask 不会返回到原生 SynchronizationContext，但可以使用 `ReturnToCurrentSynchronizationContext
public ValueTask TestAsync()
{
    await using (UniTask.ReturnToCurrentSynchronizationContext())
    {
        await UniTask.SwitchToThreadPool();
        // do anything..
    }
}
```

.NET Core 版本允许用户在与Unity共享代码时（例如CysharpOnion），像使用接口一样使用UniTask。.NET Core 版本的 UniTask 可以提供丝滑的代码共享体验。

WhenAll 等实用方法作为 UniTask 的补充，由Cysharp/ValueTaskSupplement提供。

License

此仓库基于MIT协议