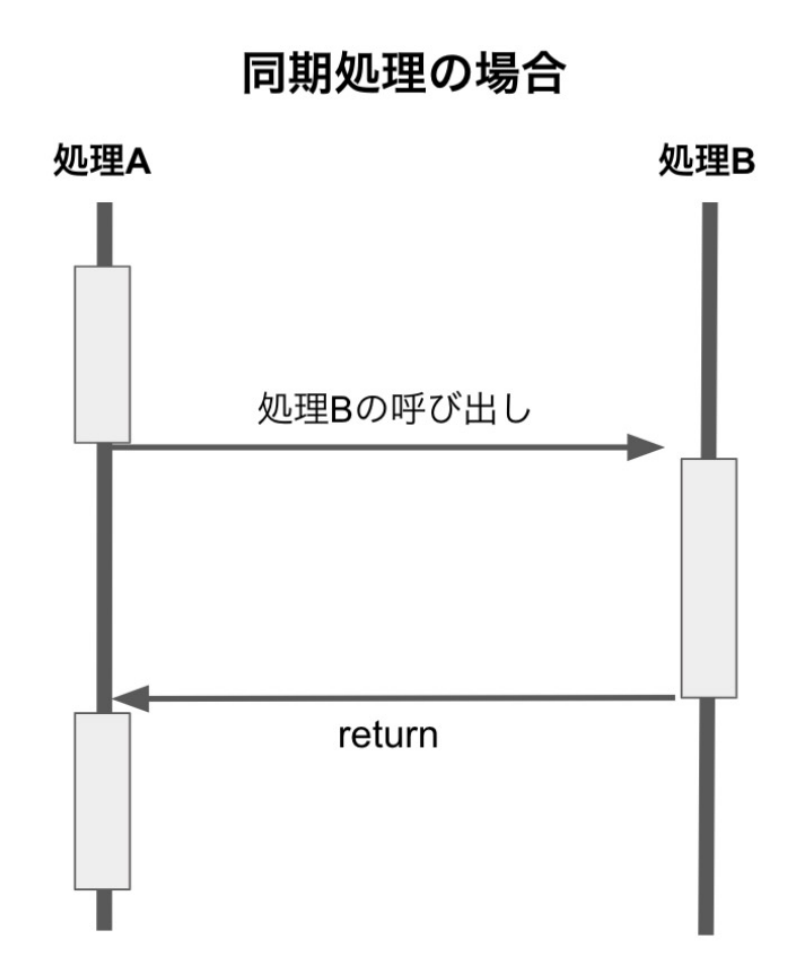
UniTask(async/await)

目次

１．同期処理と非同期処理とは

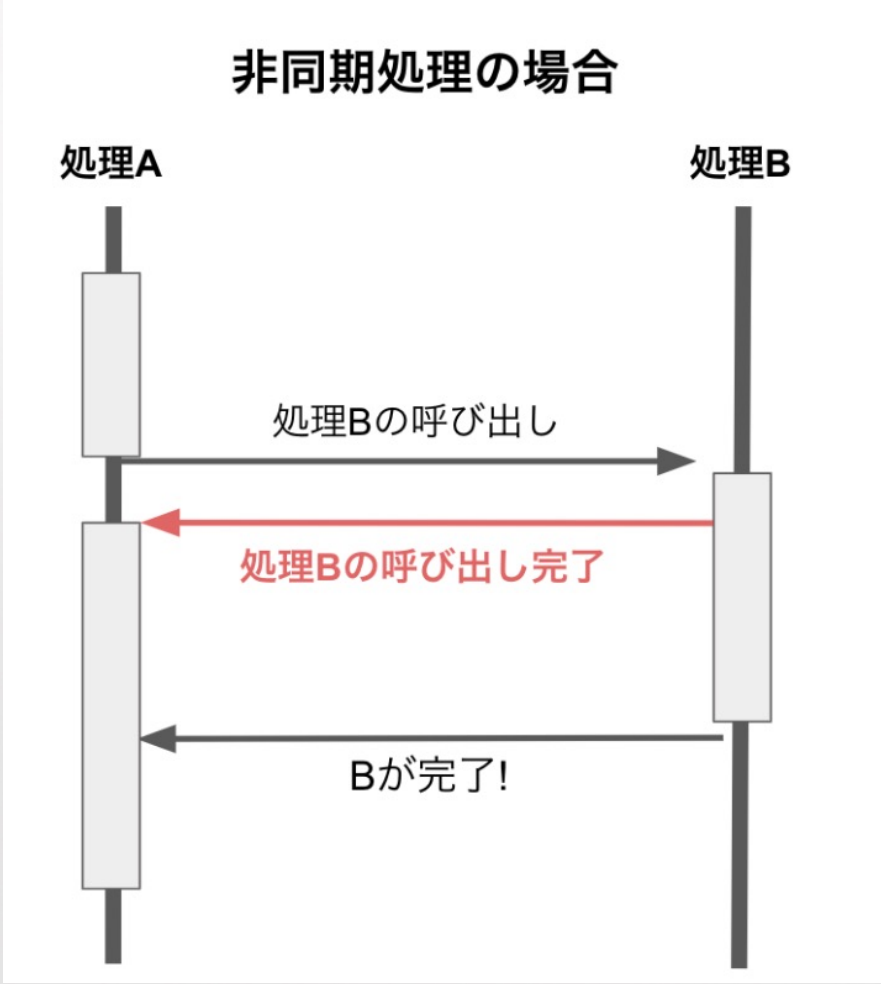
同期処理：複数の処理がある場合、一つずつ順番に処理していくもの。



処理Aの途中で処理Bが入った場合、
処理Aは停止し、処理Bを実行。
処理Bが完了しだい、処理Aの続き
から実行する。
(通常の関数呼び出しのイメージ)

１．同期処理と非同期処理とは

非同期処理：複数の処理がある場合、完了を待たずに実行していくもの



処理Aの途中で処理Bが入った場合、
処理Aは停止せず、処理Bを実行。

２．UniTask(async/await)とは

Unityでは元々コルーチンというUnity独自の非同期処理を実装する方法がある。
ただしその後、C#7でもっと使い勝手が良いasync/awaitという機能が実装された。
(ちなみにUnityは2018.3以降であればC#7が使用できる)

ただし、C#7のasync/awaitの機能はそのままUnityで使うと色々とクリアしないといけない問題があった。。。

２．UniTask(async/await)とは

＜マルチスレッド問題＞

C#7本来のasync/awaitはTaskというマルチスレッド用の処理にて実行されているが、Unity自体がマルチスレッドでの処理は向いていない設計となっていた。

(出来ないことは無いが、メインスレッドのみでしか扱えないメソッドがたくさんあり、メインスレッドではないスレッドでメソッド使うとフリーズするなどなど。。。。)

マルチスレッド・・・CPUの複数のコアを使い、プログラムの処理を分散して同時に処理を行うプログラミング手法。

2．UniTask(async/await)とは

そこで、async/awaitをゲームエンジン向けにに改良したものがCySharp社が制作しているUniTaskというライブラリであり、一般的にUnityの非同期処理で使われているライブラリである。
本家もこんな感じで言ってる。。

Cygames Engineers' Blog

UniTask – Unityでasync/awaitを最高のパフォーマンスで実現するライブラリ

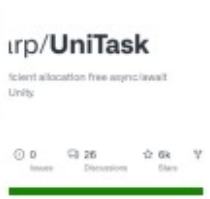
2．UniTask(async/await)とは

ただし、Unity公式でもawaitableという機能を実装しており、いずれは
コルーチンが廃止されasync/awaitが実装される予定。
その場合、UniTaskは使われなくなると思われる。。。

ついにUnity2023.1よりUnity公式版 `UniTask` が出ました。(結構語弊がありそうだか...)

GitHub - Cysharp/UniTask: Provides an efficient allocation free async/await integration f...

Provides an efficient allocation free async/await integration for Unity. - GitHub - Cysharp/UniTask: Provides an efficient allocation free async/await integration for Unity.



github.com 1 user

github.com

ただ現段階では `UniTask` と同等・もしくはそれ以上な機能を持っているわけではなく、軽く触った限りはまだまだAPIが足りず発展途上かなといった感じです。

しかし段々と使いやすくなるのは間違いないでしょうし、外部ライブラリをなるべく使いたくない方にとって魅力的なことも確かです。今後利用されるケースも増えるかと思いますので、今のうちから触っておくのも悪くないかと思います。

２．UniTask(async/await)とは

UniTask自体は外部のライブラリであるため、まずはプロジェクト作成後に
ライブラリをダウンロードする必要がある。

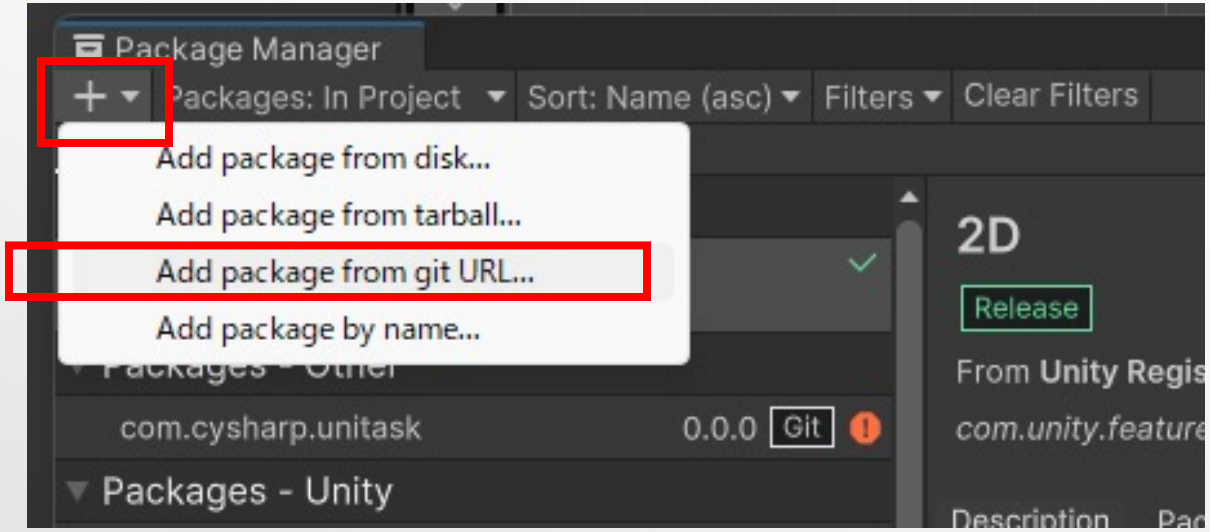
外部ライブラリは、PackageManagerからGit経由で取得したり、
あらかじめダウンロードしているUniTaskを読み込むことで実装できます。
Gitをインストールしている人はPackageManagerから、
インストールしていない人はGitHubからダウンロードするやり方でやっていきます。

2．UniTask(async/await)とは

＜Gitをインストールしている人＞

Window⇒PackageManagerから左上の＋ボタン⇒

Add package from git URL…をクリック。



https://github.com/Cysharp/UniTask.git?path=src/UniTask/Assets/Plugins/UniTask

を入力し、Addボタンを押す。エラーが出ていなければOK。

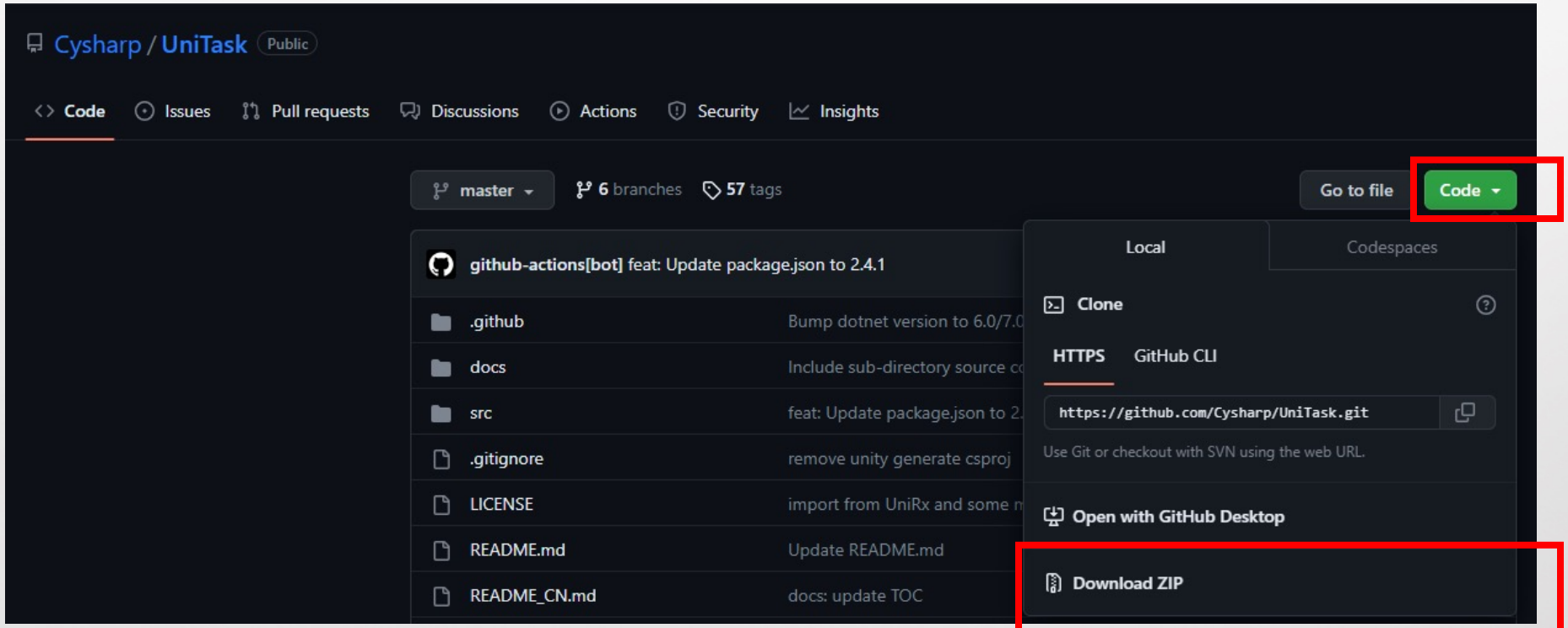
2．UniTask(async/await)とは

＜Gitをインストールしていない人＞

以下のGitHubにアクセス。

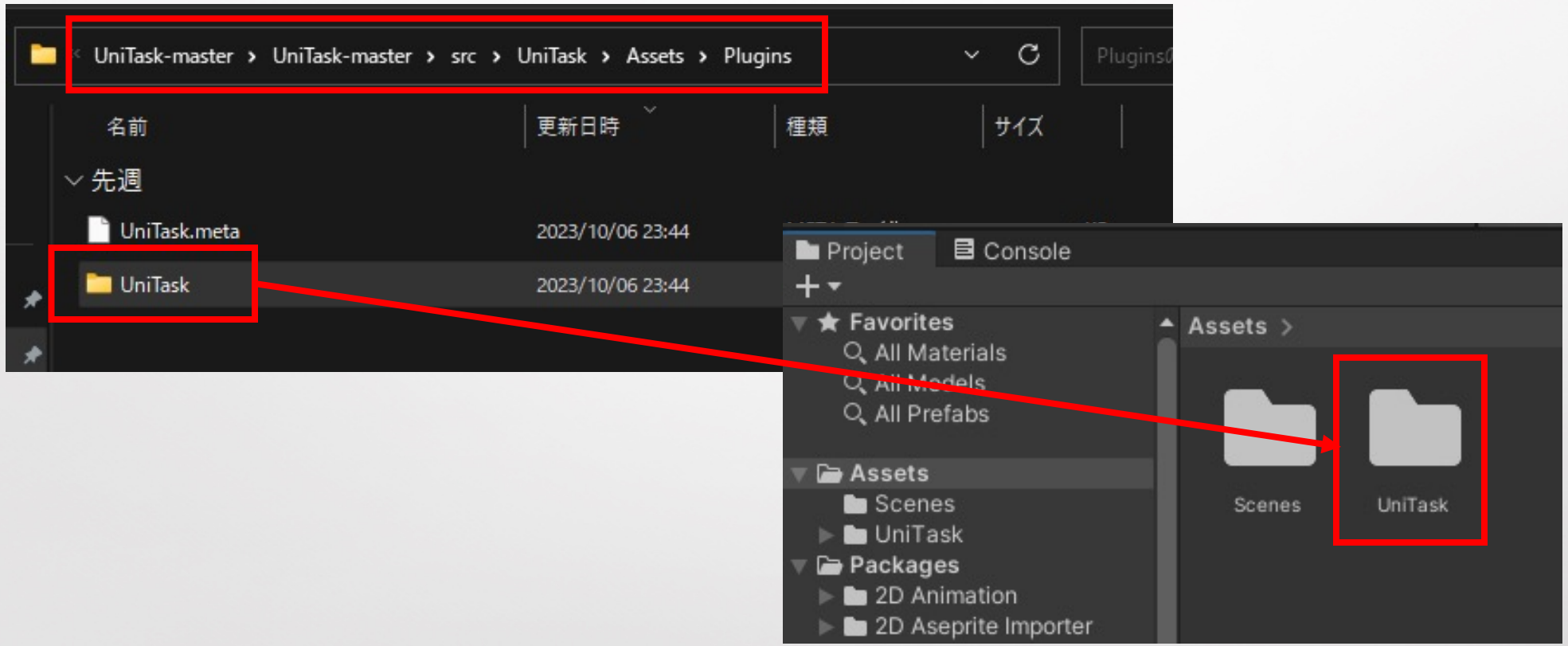
https://github.com/Cysharp/UniTask/tree/master

Code⇒Download ZIPにてダウンロード。



2．UniTask(async/await)とは

ダウンロード完了後、
UniTask-master¥src¥UniTask¥Assets¥Plugins
内のフォルダを、UnityのAssetsフォルダにドラッグする。



３．基本的な使い方

実際にUniTaskを使う前に、まずはみなさんの今知っている情報をもとに、ゲーム開始してから5秒後にDebug.Logに何か出力する処理を記載してみてください！

・コルーチン
・Time.deltaTime
・Invoke

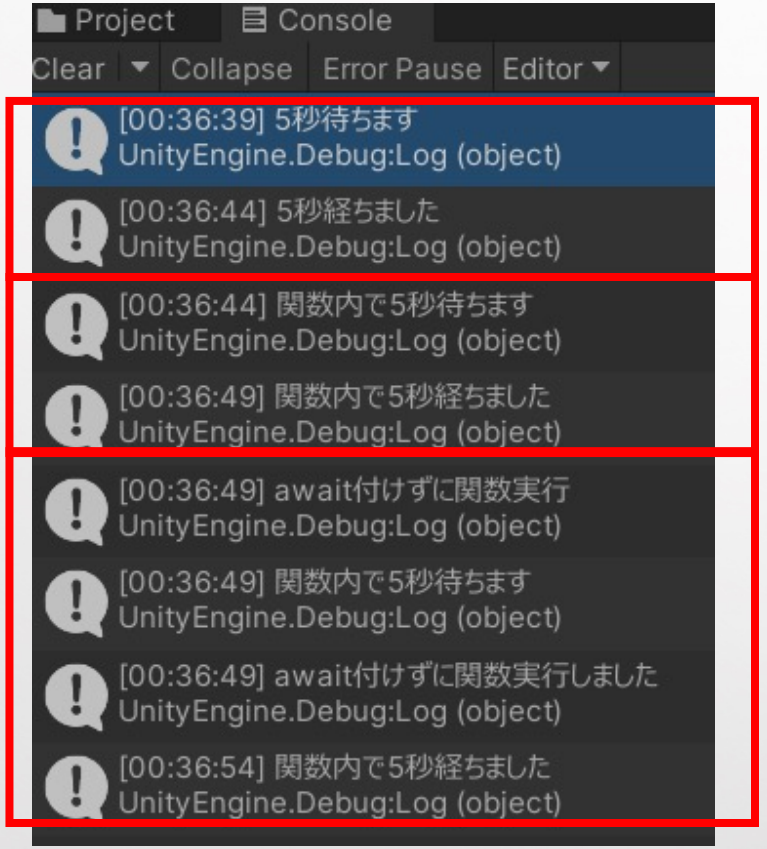
などなど、色々とあると思います。。。

３．基本的な使い方

では、実際にUniTaskを使って色々とやってみましょう！

作成するスクリプト：No1.cs

アタッチしてDebug.Logがどのような順序で出るか確認してみてください！



３．基本的な使い方

UniTaskを使うと、コード内にて完結に記載することが出来ます！
また、同期も非同期の処理も似たような形で記載することができ、コードが分かりやすくなったと思います。⇦これが重要！！

＜お約束＞
async/awaitを使用する際は、関数の頭にasycnの文字を付与する。

```
async void Start()
```

＜処理を待つ場合(同期処理)＞
関数呼び出し前にawaitを付与する。

```
await WaitMethod();
```

３．基本的な使い方

ちなみにn秒の時間を待つ、という処理は以下の1行のみで実行できるようになります。

```
await UniTask.Delay(TimeSpan.FromSeconds(5));
```

＜処理を待たない場合(非同期処理)＞

awaitをつけずに実行

```
WaitMethod();
```

このように、awaitをつけていれば同期処理、つけていなければ非同期処理と、とても分かりやすくなります。

３．基本的な使い方

＜関数内で実行したい場合＞

UniTaskを戻り値として指定すること。

```
private async UniTask WaitMethod()
```

※次でやりますが、コルーチンと違ってこのUniTaskを使えば、ジェネリック型で
戻り値を返すことが出来ます！

３．基本的な使い方

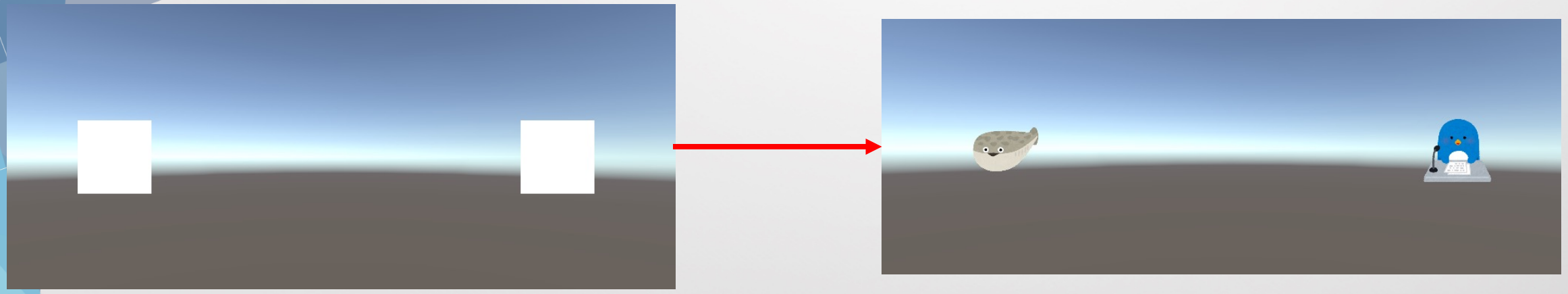
＜まとめ＞

・n秒待つ、という処理はUniTask使えば1行で書けてしまう。
・処理を待ちたいとき(同期処理)はawaitをつけよう！
・awaitをする場合は、その関数自体にasyncをつけること！
・処理を待たないとき(非同期処理)はawaitをつけない！
・関数内で使用する場合はUniTaskを戻り値に設定すること！

４．応用１

UniTaskはネットワーク系(サーバーとの通信)と、とても相性が良い。
(ネットワーク通信においては待つ、ということが頻繁にあるため)

ここではその一例でネットワーク越しに画像をダウンロードして表示する、
ということをUniTaskを使ってやっていこうと思います！
UniTaskでダウンロードが終わるまで待ち、画像を表示するという流れに
なります。



４．応用１

まずは、以下のプログラムを作成してください。

　作成するスクリプト：No2.cs

Unity上でアタッチし、設定値は以下でお願いします。

imageURL1：

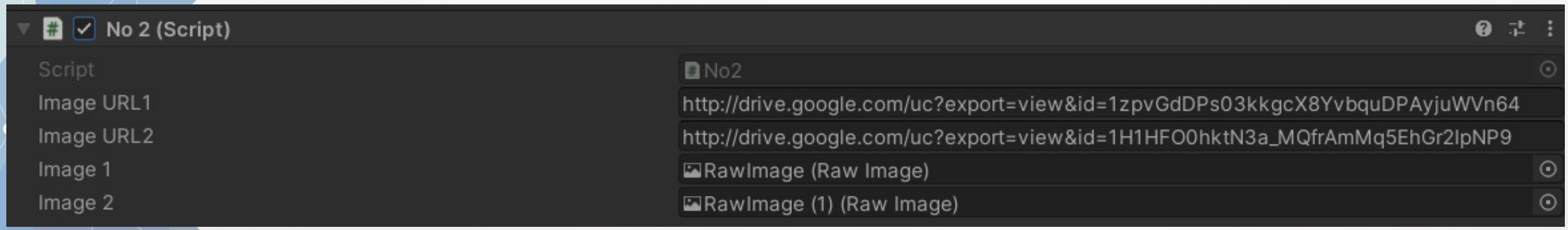
http://drive.google.com/uc?export=view&id=1zpvGdDPs03kkgcX8YvbquDPAyjuWVn64

imageURL2：

http://drive.google.com/uc?export=view&id=1H1HFO0hktN3a_MQfrAmMq5EhGr2IpNP9

４．応用１

UI⇒RawImageを2つ作成し、image1とimage2に割り当ててください。
最終的に以下のようになっていればOK！



出来たら、ゲーム実行し少し時間が経ったら画像が表示されればＯＫです！

４．応用１

UniTaskのawait(同期通信)を使うことで、通信処理を待つことが出来る。

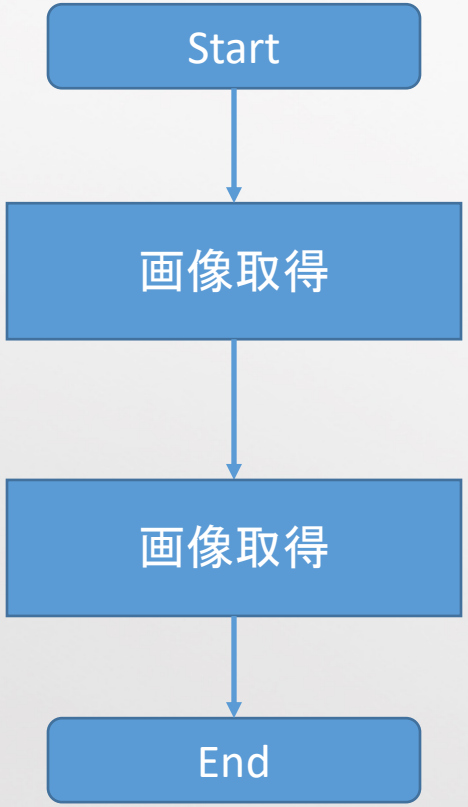
```
await www1.SendWebRequest();
```

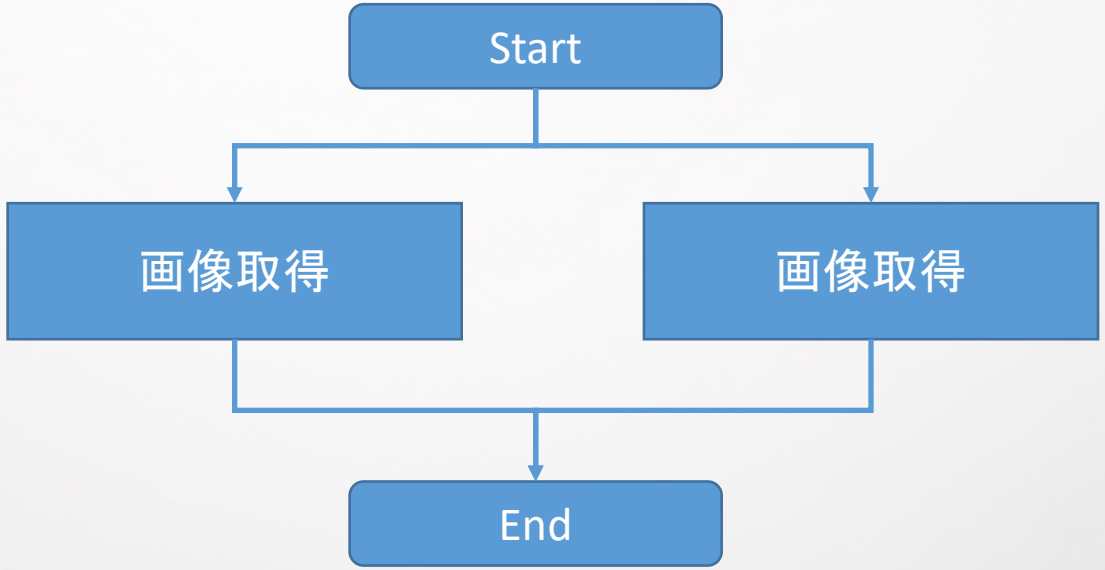
（ちなみに使わなかった場合は、request1.isDoneというデータを定期的に確認し、Trueになっていたら**OK**という判定をしなくてはいけないです)

これはUniTaskがUnityの持っているSendWebRequestという機能をAwait出来るようにしてくれているから使えるものです。

４．応用１

UniTaskについて、並列実行させる機能が実装されているので使用します！





４．応用１

まずは、実際に処理を書いてみようと思うので、新しく作成してみてください！

作成するスクリプト：No2_1.cs

並列実行の他、戻り値も使っているので色々確認しながら作成してもらえると良いかと思います！

＜並列実行＞

```
(tex1, tex2) = await UniTask.WhenAll(DownloadTexture(imageURL1), DownloadTexture(imageURL2));
```

UniTask.WhenAllを使用すると、その中に記載した処理すべてが並列実行させ、すべてのタスクが完了するまでawait(待つ) という処理を作成することが出来ます！

４．応用１

＜戻り値の指定＞

ここがコルーチンとは違い、UniTaskを使う大きなメリットになるところです！！

UniTaskではジェネリック型で戻り値を返すことが出来ます！

書き方は以下のようになります。

```
private async UniTask<Texture2D> DownloadTexture(string url)

    return DownloadHandlerTexture.GetContent(www);
```

UniTaskの後に＜戻り値の型＞を記載し、後はreturnで返すだけです！

今回は画像を返すのでTexture2D型を返していますが、サーバーから

受け取った文字を返したければUniTask<String>のようになります。

４．応用１

＜まとめ＞

・戻り値がある関数を作りたいならUniTask<型名>での指定！

・並列実行をうまく使うことで処理時間を短縮させることが出来る！

・本来は難しい通信処理も、UniTaskを使うことで通信が終わるまで待つ、
　ということが簡単に実現できる！

５．応用２

ここでは、Unityの機能を待つ方法についてやっていきます！
基本や応用１に比べると使用頻度は低いかと思いますが、もしかたらこの方法で解決出来る問題があるかもしれないので、覚えておいて損はないかと思います！

やることは以下の内容です！

・当たり判定が発生するまで待つ
・ボタンが押されるまで待つ

５．応用２

＜当たり判定が発生するまで待つ＞

これもまずはスクリプトを作ってみてください！

　作成するスクリプト：No3.cs

作成後、コライダーをつけたSphereを２つを配置して実際に動作するか確認してみてください。

5．応用２

```
async void Start()
{
    //当たり判定のawait用のメソッド取得
    var collision_enter_trigger = this.GetAsyncCollisionEnterTrigger();
    var collision_exit_trigger = this.GetAsyncCollisionExitTrigger();

    //当たり、または外れるまで待つ処理
    Debug.Log("当たるまで待つ");
    await collision_enter_trigger.OnCollisionEnterAsync();
    Debug.Log("当たった");
    Debug.Log("次は外れるまで待つ");
    await collision_exit_trigger.OnCollisionExitAsync();
    Debug.Log("外れた");
}
```

GetAsyncCollisionXXXTrigger();で当たり判定が発生したタイミングを通知するトリガーを受け取り、それを使って、awaitで待つことが出来る。

５．応用２

＜ボタンが押されるまで待つ＞

これもまずはスクリプトを作ってみてください！

　作成するスクリプト：No3_1.cs

作成後、UIのボタンを配置して実際に動作するか確認してみてください。

```
Debug.Log("ボタンが押されるまで待つ");
var event_handler = btnStart.GetAsyncClickEventHandler();
await event_handler.OnClickAsync();
Debug.Log("ボタンが押された");
```

クリックイベントのハンドラーを取得し、OnClickAsyncで押されるまで待つ処理を作成することが出来る。

５．応用２

＜まとめ＞

・ボタンが押されたときや当たり判定が発生したなど、Unity特有のイベントに対しても待つことができます！

・今回はごく一部しか紹介していませんが、気になった人は調べてみてください！

・UniRxという別機能を用いることで、ここら辺の処理を直感的に分かりやすく書くことが出来るので、興味ある人は調べてみましょう！
(UniRxもとても便利なのですが学習コストが高いため、結構、賛否両論があったりします。。。)

６．UniTaskのキャンセルについて

UniTaskもそうですが、C#のTaskも含めて、キャンセルという概念が分かっていない状態でUniTaskを使うとフリーズなど、結構大きな不具合を出してしまう危険性があります。

ここでは以下の内容について確認していきます。

・タスクをキャンセルしない場合、どのようなリスクがあるのか

・タスクのキャンセルの仕方

正直、async/awaitにおいてこのTaskのキャンセルということをしないといけないというのが最大のデメリットになってしまいます。。。

６．UniTaskのキャンセルについて

＜タスクをキャンセルしない場合、どのようなリスクがあるのか＞

とりあえず危険性を確認するため、スクリプト作成してアタッチしてください！

　作成するスクリプト：No4.cs

このコードにはどんな危険性があるか、、どうでしょうか？

大きく2つあるかと思います！

(そもそも無限ループの時点で怪しいけど)

６．UniTaskのキャンセルについて

```
private async UniTask LongTask()
{
    while (true)
    {
        //自身の名前を出力
        Debug.Log(gameObject.name);

        //1秒待つ
        await UniTask.Delay(TimeSpan.FromSeconds(1));
    }
}
```

UniTaskはシーンが終わっても動作し続けるため、ゲーム開始時にタスク実行し、ゲーム終了⇒再度タイトル画面に戻ってきたら、再度タスクが実行され、合計2つのタスクが実行されてしまます。
(終わらないタスクが2つ実行される危険な状態なる)

６．UniTaskのキャンセルについて

試しに、No3.csのStartに以下のようなDestroyの記述をしたら
どうなるでしょうか？

```
async void Start()
{
    //5病後にGameObjectを破棄する
    Destroy(gameObject,5);

    //終わらないタスクを待つ
    await LongTask();
    Debug.Log("ここに来ることはない");
}
```

６．UniTaskのキャンセルについて

GameObjectは破棄されるが、タスク自体は止まらないため、
gameObject.nameを参照した時点で、Nullのエラーが返ってきているかと思います。

⇓

GameObject/コンポーネントが破棄されてもTask自体は破棄されないため、動き続けてしまう。
(ちなみにコルーチンだとGameObjectが破棄されたら止まります！！)

というか今のままだと、そもそもタスクを止めることが出来ない。
(ソシャゲとかで大量にダウンロードしている最中にキャンセルさせることが出来ない)

６．UniTaskのキャンセルについて

＜タスクを止める方法＞

では、実際にタスクを止めれないかというと、止める方法があります。

CT(CancellationToken)という機能を使い、タスクを止めていきます。

まずはスクリプトを作成しましょう！

　作成するスクリプト：No4_1.cs

ここら辺からちょっと複雑になってきます。。。

６．UniTaskのキャンセルについて

＜CancellationTokenとCancellationTokenSource＞

CancellationToken：

タスクのキャンセルが要求されたか知ることが出来る

CancellationTokenSource：

タスクをキャンセルするための要求を出すことが出来る

CancellationTokenSouceを発行

⇓

CancellationTokenで検知、例外処理を発生させる

⇓

try～catchで受け取る

6．UniTaskのキャンセルについて

コード上での処理は以下のような流れになります！

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.A))
    {
        //キャンセル要求
        cs.Cancel();
    }
}
```

```
private async UniTask LongTask(CancellationToken ct)
{
    while (true)
    {
        //もしキャンセル要求があれば例外を発生させる
        ct.ThrowIfCancellationRequested();
```

```
try
{
    //終わらないタスクを待つ
    await LongTask(cs.Token);
    Debug.Log("ここに来ることはない");
}
catch (OperationCanceledException e)
{
    Debug.Log("キャンセルされました");
}
```

６．UniTaskのキャンセルについて

C#において、Taskを止める方法は以下の２つ。これは仕様として定められている。

・returnで終わらす

・例外処理を発生させる

＜まとめ＞

・永遠に動くタスクはバグの原因になりがちですよ！

・CancellationTokenSourceでキャンセルを発行できます！

・CancellationTokenでキャンセルを受け取って例外出せます！

問題

async/awaitを用いて、外部のテキストファイルを非同期に読み込み、
UI上のTextMeshProコンポーネントに表示してください。

＜やること＞
画面上にはTextMeshProのテキストとボタンのみ表示すること。
読み込む前は「Loading…」、読み込み後はテキストの内容を表示すること。
ネットワークなどのエラーの場合は「Error」と表示すること。

問題

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.Networking;
using Cysharp.Threading.Tasks;

public class Q1 : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI textResult;

    public async void LoadTextAsync() {
        // UIを更新（MainThread）
        textResult.text = "Loading...";

        // URLを指定
        string url = "https://yasucom.net/vantan/2025/zissen/10_Test.txt";
        UnityWebRequest request = UnityWebRequest.Get(url);

        // 非同期処理を実行（UnityWebRequestのSendWebRequestはTaskを返す）
        await request.SendWebRequest();

        // 成功時処理
        if (request.result == UnityWebRequest.Result.Success) {
            string text = System.Text.Encoding.UTF8.GetString(request.downloadHandler.data);
            textResult.text = text;
        } else
        {
            textResult.text = "Error";
        }
    }
}
```