



# コルーチン

📁 コース	🔗 EX
🔧 ツール	🔗 Unity
📅 作成日時	9月4日
👤 作成者	👤 マーシー（岩崎正孟）
📊 いいね数	♡7
▼ さらに6件のプロパティ	

🌐 コメントを追加...

👍 いいね

👎 いいねを外す



## 目次

1. コルーチンとは？
2. コルーチンの使い方
3. コルーチンの本質
  - 3-1. 非同期処理とは
  - 3-2. コルーチンでの非同期実行のテスト
4. コルーチンの便利な使い方
  - 4-1. コルーチン内で別のコルーチンの実行終了を待機する
  - 4-2. コルーチン以外の用意されている非同期処理を待機する
  - 4-3. Unityの提供メソッドをコルーチン化する
5. コルーチンの応用的な使い方
  - 5-1. While文と組み合わせた条件待機処理
    - 5-1-1. 条件待機処理の簡略化
  - 5-2. コールバックを設定してコルーチン完了後の処理を登録する
6. コルーチンの止め方（キャンセル処理）
  - 6-1. yield breakを使ったキャンセル処理
  - 6-2. 破棄によるキャンセル処理
  - 6-3. StopCoroutineメソッドを使ったキャンセル処理
7. コルーチン以外の非同期処理

## 1. コルーチンとは？

Unityで標準提供されている**非同期処理**です。  
時間待機処理、と理解するとドツボ踏みます。

## 2. コルーチンの使い方

コルーチンの使い方は、基本的に以下の三点を守れば機能します。

1. 関数を作る際にいつも `void` とつけている部分を `IEnumerator` とする。
2. `yield return ~~~` みたいな処理を一回以上、コルーチン内に書く。(必須)
3. しかるべき場所で `StartCoroutine` メソッドを使ってコルーチンを起動する。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour { private void Start() { // コルーチンの実行 StartCoroutine(Marcy()); } // コルーチン private IEnumerator Marcy() { // ここに処理を書く // 3秒停止 yield return new WaitForSeconds(3); // ここに再開後の処理を書く } }
```

`StartCoroutine` の使い方に関しては何通りか記述方法があります。

正直、上記のサンプルコード内で示した書き方がオススメですが、テキストやインターネット上のブログでは2番目の方法をよく見かける気がします。

1. `StartCoroutine(Marcy());` ...関数呼び出しをそのまま記述する方法
2. `StartCoroutine("Marcy");` ...文字列で関数名を指定する方法
3. `StartCoroutine(nameof(Marcy));` ...nameof演算子を使う方法。2番目の方法の亜種。

また、いわゆる任意の時間待機を実装する書き方について、とりあえずテキストで出していた & 出したことがある書き方を全部載せておきます。

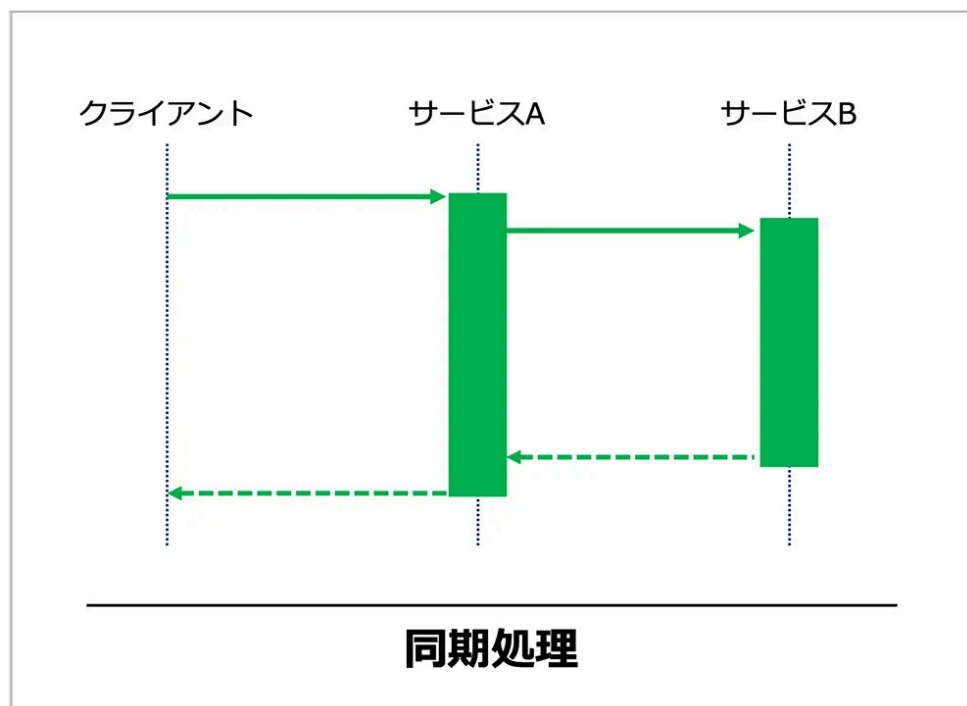
1. `yield return null;` ... 1 フレーム待機。
2. `yield return new WaitForSeconds(3);` ...括弧の中に秒数を指定して、その秒数分待機。
3. `yield return new WaitForEndOfFrame();` ...フレーム最後まで待機。

## 3. コルーチンの本質

### 3-1. 非同期処理とは

さて、コルーチンの基本事項はこれくらいにして、コルーチンの本質に触れていきましょう。  
何度でも言いますが、コルーチンは**非同期処理**です。

適当にGoogleで「非同期処理」と検索すると、AWSがなんかいい感じの画像を作っていました。  
とりあえず、いつもの関数、いわゆる「同期処理」がこんな感じの図になります。



クライアントがサービスAの処理を起動させる。

マーシー（岩崎正孟） 2023/09/04（編集済み）

主に画面描画に絡む処理とかで使う頻度が高い。  
EXテキストでは、ボクが作った ... 詳細

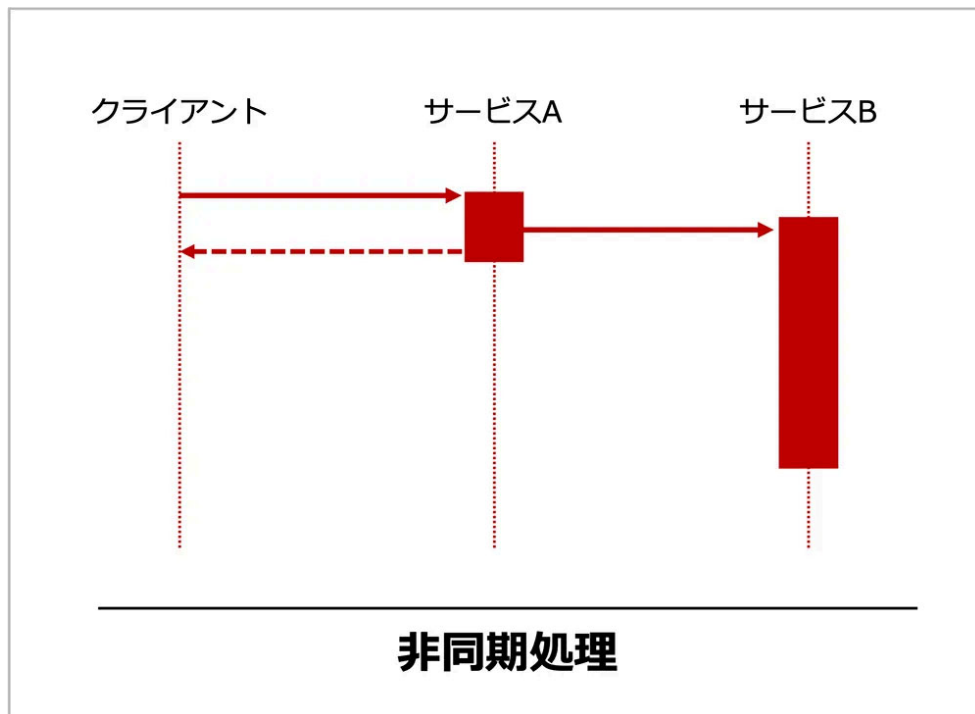
1件の返信を表示

マーシー（岩崎正孟） 2023/09/05

2番目は言わずもがな、1番目はこのノートでもサンプルコード出てくるから、3番目だけコメントしておいた。

サービスAの処理からサービスBの処理を起動させる。  
サービスBの処理が止まるまで待って、サービスAの処理が再開し、結果がクライアントに通知される。  
と、いう流れですね。

非同期処理なら、このような図になります。



クライアントがサービスAの処理を起動させる。  
サービスAの処理からサービスBの処理を起動させる。  
サービスBの処理が止まるのを待たず、サービスAの処理が並列で実行され、結果がクライアントに通知される。

この、処理が並列に進む構造が非同期処理の肝です。  
この一つ一つの処理を「スレッド」と呼ぶことがあります。  
マルチスレッド、とか言ったりします。

スレッド、という言葉はSlackでおなじみですね。  
times\_マーシーに同時にいくつものスレッドが立ち、それぞれ違う話を同時にする、というのはまさに非同期処理みたいなものです。


## 3-2. コルーチンでの非同期実行のテスト

では実際に、Unityのコルーチンで非同期処理を体感してみましょう。  
以下のサンプルコードをご覧ください。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour { private void Start() { // コルーチンの実行 StartCoroutine(Marcy()); // まどーのーゆーきー Debug.Log("まどーのゆき"); } // コルーチン private IEnumerator Marcy() { // ほたーるのっ Debug.Log("ほたるの"); // 2秒停止 yield return new WaitForSeconds(2); // ひーかーりっ Debug.Log("ひかり"); } }
```

こちら、Marcy関数が非コルーチンであれば、卒業式の曲が自然な形でログに出力されることになったでしょう。

しかし、コルーチン化してしまった上記の実装の場合、結果は以下のようになります。

 マーシー（岩崎正孟） 2023/09/05  
厳密な話をすると、実はコルーチンは純粋なマルチスレッドではない、という話があります。  
これはUnityの仕様で、未だに大半の標準メソッドがいわゆるスレッドセーフ対応されておらず、メインスレッ...



コルーチンが非同期で行われたため、曲の歌詞の出力順がずれてしまったわけです。

## 4. コルーチンの便利な使い方

コルーチンは単に、指定した時間だけ待機するというだけでなく、いろいろと便利な活用ができます。

非同期処理、という基本があれば、なるほどそんな記述法があるのか、となるくらいのTipsばかりなので、そこまで難しい実装法でもありません。

### 4-1. コルーチン内で別のコルーチンの実行終了を待機する

先ほどの蛍の雪コードを、実際の歌詞順にログ出力する修正を加える例として、以下のような書き方ができます。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour { private void Start() { // コルーチンの実行 StartCoroutine(StartMarcy()); } private IEnumerator StartMarcy() { // Marcy関数の実行終了を待機 yield return StartCoroutine(Marcy()); // まどーのゆきー Debug.Log("まどゆき"); } // コルーチン private IEnumerator Marcy() { // ほたるのつ Debug.Log("ほたる"); // 2秒停止 yield return new WaitForSeconds(2); // ひかりっ Debug.Log("ひかり"); } }
```

### 4-2. コルーチン以外の用意されている非同期処理を待機する

Unityには標準実装としても意外と非同期処理が準備されており、それもコルーチンで実行完了を待機することができます。

以下のサンプルコードは、Unityで標準提供されている `SceneManager.LoadSceneAsync` メソッドを非同期で実行することで、シーンのロード後にログ出力する実装を作成した例です。

```
using System.Collections; using UnityEngine; using UnityEngine.SceneManagement; public class Test : MonoBehaviour { IEnumerator SceneLoading() { Debug.Log("シーンロード開始"); // 非同期でシーンをロードする yield return SceneManager.LoadSceneAsync("GameScene", LoadSceneMode.Additive); Debug.Log("シーンロード完了"); } }
```

### 4-3. Unityの提供メソッドをコルーチン化する

Start関数など、Unityが用意した関数はコルーチン化できる場合があります。

Update関数はコルーチン化できないなど、何でもできるわけではないですが、この方法をうまく使えればコルーチンの呼び出しをこちらで書かなくてよくなるので便利です。

以下のサンプルコードは、4-1のサンプルコードを修正し、Start関数をコルーチン化することで、より読みやすいコードにできた実装例です。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour { //
Start関数をコルーチン化する private IEnumerator Start() { // Marcy関数の実行終了を待機 yield
return StartCoroutine(Marcy()); // まどーの一ゆーきー Debug.Log("まどゆき"); } // コルー
チン private IEnumerator Marcy() { // ほたーのつ Debug.Log("ほたる"); // 2秒停止 yield
return new WaitForSeconds(2); // ひーかーりつ Debug.Log("ひかり"); } }
```

## 5. コルーチンの応用的な使い方

では前節まででコルーチンの様々な使い方を理解したところで、より応用的なコルーチンの使い方もみておきましょう。

応用的、といっても非常に難しい実装を扱うわけではなく、むしろお子さんのゲームで使用する場面がありそうな実装をいくつかご紹介します。

### 5-1. While文と組み合わせた条件待機処理

何かの条件を満たすまで非同期で待機して、満たせば任意の処理を実行する、というプログラムは、While文による繰り返しと組み合わせると簡単に書けます。

以下のサンプルコードは、Wキーが押されたらプレイヤーを前方に動かす処理を非同期で起動し続けるように実装した例です。

このようなキー入力操作実装は、Update関数で実装するとどうしてもすべての要素を並列に行えず、一つのUpdate関数に条件分岐を直列に記述するほかないですが、非同期処理にすればそれらを並列で行えることになります。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour {
[SerializeField, Tooltip("キャラクターコントローラー")] private CharacterController
characterController = null; [SerializeField, Tooltip("速度")] private float moveSpeed =
10.0f; private void Start() { // WMove関数の実行 StartCoroutine(WMove()); } // Wキーでのキ
ー入力操作を制御する private IEnumerator WMove() { // Wキーが押されてなければ繰り返す while
(!Input.GetKey(KeyCode.W)) { // 1フレーム待機 yield return null; } // 前方に動かす
characterController.Move(transform.forward * moveSpeed * Time.deltaTime); // 次のフレーム
まで待機 yield return null; // 再びWキー入力がされていれば動き続ける
StartCoroutine(WMove()); } }
```

#### 5-1-1. 条件待機処理の簡略化

また、Unityでは独自に、コルーチンで何度もWhile文を書かなくて済むように `WaitUntil` ・

`WaitWhile` クラスの二つを提供しています。

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour {
[SerializeField, Tooltip("キャラクターコントローラー")] private CharacterController
characterController = null; [SerializeField, Tooltip("速度")] private float moveSpeed =
10.0f; private void Start() { // WMove関数の実行 StartCoroutine(WMove()); } // Wキーでのキ
ー入力操作を制御する private IEnumerator WMove() { // Wキーが押されてなければ繰り返す yield
return new WaitWhile(!Input.GetKey(KeyCode.W)); // 前方に動かす
characterController.Move(transform.forward * moveSpeed * Time.deltaTime); // 次のフレーム
まで待機 yield return null; // 再びWキー入力がされていれば動き続ける
StartCoroutine(WMove()); } }
```



マーシー（岩崎正孟） 2023/09/04

ちなみに、StartCoroutineを使わなくても同じ挙動をする。

```
yield return Marcy();
```



マーシー（岩崎正孟） 2023/09/05

SceneManager.LoadScene系メソッドの第二引数は、LoadSceneMode.SingleとLoadSceneMode.Additiveの二つが選択できます。  
いわゆる、「Sceneを重ねる」という話で、Unityのとても便利な機能な... 詳細

```
using System.Collections; using UnityEngine; public class Test : MonoBehaviour {
[SerializeField, Tooltip("キャラクターコントローラー")] private CharacterController
characterController = null; [SerializeField, Tooltip("速度")] private float moveSpeed =
10.0f; private void Start() { // WMove関数の実行 StartCoroutine(WMove()); } // Wキーでのキ
ー入力操作を制御する private IEnumerator WMove() { // Wキーが押されてなければ繰り返す yield
return new WaitUntil(Input.GetKey(KeyCode.W)); // 前方に動かす
characterController.Move(transform.forward * moveSpeed * Time.deltaTime); // 次のフレーム
まで待機 yield return null; // 再びWキー入力がされていれば動き続ける
StartCoroutine(WMove()); } }
```

## 5-2. コールバックを設定してコルーチン完了後の処理を登録する

後述しますが、コルーチンは返り値を伴う実装ができません。

ゆえに、どうしてもコルーチン完了後にその結果を返してほしいような場合や、あるいは単純にコルーチン完了後に行う処理をコルーチン外部で設定したい場合、デリゲートによるコールバックを設定する手法があります。

以下のサンプルコードは、UnityAction型の引数をコルーチン関数に設定することで、非同期による実行結果を外部で受け取る実装例です。

```
using System.Collections; using UnityEngine; using UnityEngine.Events; public class Test
: MonoBehaviour { // プレファブ private GameObject prefab = null; private void Start() {
// プログラムからプレファブをロードして取得する StartCoroutine(LoadAsync("bullet", obj =>
prefab = obj)); } // Resourcesフォルダ下に配置されたプレファブを動的にロードする private
IEnumerator LoadAsync(string filePath, UnityAction<GameObject> callback) { // 非同期ロー
ード開始 ResourceRequest resourceRequest = Resources.LoadAsync<GameObject>(filePath); //
ロードが終わるまで待機 while(!resourceRequest.isDone) { yield return null; } // ロード完
了、resourceRequest.assetからロードしたアセットを取得 GameObject asset =
resourceRequest.asset as GameObject; // ロードしたGameObject型のアセットを引数に設定してコー
ルバックを起動 callback.Invoke(asset); } }
```

## 6.コルーチンの止め方（キャンセル処理）

ここは少し発展的な話にもなりそうですが、コルーチンの止め方を説明します。

コルーチンのような非同期処理が、Update関数のような常時駆動の関数よりも取り回しが良いのは、ひとえに頑張ったら止められるから、というところがあります。

他言語の非同期処理と比較すると少々異質なコルーチンは、以下の3点のような止め方があります。

1. `yield break;` を使う
2. コルーチンを書いたスクリプトか、それがアタッチされているゲームオブジェクトをゲームシーン上で破壊する
3. StopCoroutineメソッドを使う

ちなみに上記、実装しやすい順に上記並べています。

特にStopCoroutineが...まあ癖が強いです。

### 6-1. yield break;を使ったキャンセル処理

まず一つ目の `yield break;` は、通常の関数の `return;` に相当するものです。

以下略、天空の城ラピュタの「バルス！」ですね（諸説あり）。

この方法は、コルーチン関数内部からコルーチンを停止させる方法になります。

以下、サンプルコードです。



マーシー（岩崎正孟） 2023/09/05

だからといって、別にキー入力操作実装でコルーチンを推奨するものではない。  
別にUpdate関数に実装していいと思う。  
あくまでこれは、理屈の話。



```
using UnityEngine; using System.Collections; public class Test : MonoBehaviour { private
IEnumerator Start() { Debug.Log("コルーチンが開始されました。"); // 1秒間待機 yield return
new WaitForSeconds(1.0f); Debug.Log("1秒経過しました。"); // コルーチンを終了 yield break;
Debug.Log("この行は実行されません。"); } }
```

## 6-2. 破棄によるキャンセル処理

二つ目の、「コルーチンを書いたスクリプトか、それがアタッチされているゲームオブジェクトをゲームシーン上で破棄する」というのは、コルーチンを外部からキャンセルする実装の一つです。

これはサンプルコードを書きようがないですね。

単純に破棄すればいいだけです。

なぜ、これでキャンセルできるのかについて説明しておきます。

コルーチンはUnityのスクリプトを作るとテンプレートで記載されている **MonoBehavior** にて実装されている機能で、**MonoBehavior** という基盤がなければ実行できません。

ゆえに、スクリプトを破棄されたり、ゲームオブジェクトを破壊されてしまうと強制的に実行停止してしまうわけです。



マーシー (岩崎正孟) 2023/09/27

サンプルコード見ていただくとわかりやすいけど、ボクは条件がif文のような感じで書けるWaitUntilが可読性良くておすすめ

## 6-3.StopCoroutineメソッドを使ったキャンセル処理

三つ目のStopCoroutineメソッドも、コルーチンを外部からキャンセルする方法なのですが...

こいつは結構、罠とかもある上に、自分の経験上でもプログラムのタイミングなどでこいつを上手く活用できなかったことが多々あります。

このメソッドを使う例はなかなか見たことがないですが、もし使われる方は、ぜひ下記のサイト記事はご確認ください。

ってことで、以下略！

【Unity】StopCoroutineメソッドに潜まれた罠 - はなちろのマイノ...

はじめに今回はStopCoroutineメソッドを使うにあたって注意しなければなら  
ないことについての記事になります！コルーチンを止める方法として用意され

 <https://www.hanachiru-blog.com/entry/2019/07/17/181223>

Unity  
【Unity】  
StopCoroutineメソッドに  
潜まれた罠  
はなちろのマイノート

## 7. コルーチン以外の非同期処理

最後に、Unityで使用できるコルーチン以外の非同期処理に触れておきます。

純粋C#でいうところだと、定番の非同期処理といえばTaskですが、これはUnityでも使用できます。

ただし、TaskはUnityエンジンのプログラム処理の流れから考えると最適化されておらず、使えはするもののパフォーマンスがかなりよろしくありません。

ただ、コルーチンはいろいろとデメリットがあります。

いくつかあげると、こんな感じ。

1. 値を返す実装ができない
2. ゲームオブジェクト破棄などによる外部キャンセル（5-2の項目を参照）などを使うとメモリパフォーマンスがよくない
3. **MonoBehavior** に依存していて、非 **MonoBehavior** のクラスでは基本的に使用できない

なので、Unity上級者になるとコルーチンを卒業し、サードパーティーライブラリなどを使ったりします。

Unityの非同期処理で代表的なサードパーティーといえば、UniTaskですね。

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

Provides an efficient allocation free async/await integration for Unity - C# -  
Cysharp/UniTask: Provides an efficient allocation free async/await integration for

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

Cysharp/UniTask

Provides an efficient allocation free async/await  
integration for Unity.



49 Contributors 10 Issues 45 Discussions 8k Stars 847 Forks

いわゆる `async/await` 対応は、実はUnity公式でも着実に進んでいます。

Unity2023からは `Awaitable` というUnity標準提供される非同期処理も使用できるようになるのですが...現状はまだ先の話ですね。

とりあえずここで言っておきたいのは、`Task` はUnityでのプログラミング現場（教育現場も含む）では使用してほしくないということです。

Unity界限でも避けられているものを敢えて授業で使わなくてもいいと思っています。





マーシー（岩崎正孟） 2023/09/05

つまりコルーチンは、MonoBehaviorを継承していないクラスでは使用できない。

無理やりにでもMonoBehaviorを取得しないと起動できない...。



マーシー (岩崎正孟) 2023/09/05

上のほうでコメントしましたが、Unityは基本的に、厳密にはメインスレッドですべてのプログラムが動いている、という弊害がここでも原因になっています。