**一、介绍**

协程Coroutine在Unity中一直扮演者重要的角色。可以实现简单的计时器、将耗时的操作拆分成几个步骤分散在每一帧去运行等等，用起来很是方便。  
但是，在使用的过程中有没有思考过协程是怎么实现的？为什么可以将一段代码分成几段在不同帧执行？  
本篇文章将从实现原理上更深入的理解协程，最后肯定也要实现我们自己的协程。  
关于协程的用法网上有很多介绍，不清楚的话可以看下[官方文档](https://docs.unity3d.com/ScriptReference/Coroutine.html)，这里不做赘述。

**二、迭代器**

在使用协程的时候，我们总是要声明一个返回值为**IEnumerator**的函数，并且函数中会包含**yield return xxx**或者**yield break**之类的语句。就像文档里写的这样

private IEnumerator WaitAndPrint(float waitTime)

{

yield return new WaitForSeconds(waitTime);

print("Coroutine ended: " + Time.time + " seconds");

}

想要理解IEnumerator和yield就不得不说一下迭代器。迭代器是C#中一个十分强大的功能，只要类继承了IEnumerable接口或者实现了GetEnumerator()方法就可以使用foreach去遍历类，遍历输出的结果是根据GetEnumerator()的返回值IEnumerator确定的，为了实现IEnumerator接口就不得不写一堆繁琐的代码，而yield关键字就是用来简化这一过程的。是不是很绕，理解这些内容需要花些时间。  
**不理解也没关系，目前只需要明白一件事，当在IEnumerator函数中使用yield return语句时，每使用一次，迭代器中的元素内容就会增加一个。就向往列表中添加元素一样，每Add一次元素内容就会多一个。**  
先来看看下面这段简单的代码

IEnumerator TestCoroutine()

{

Debug.log(1111)

yield return null; //返回内容为null

Debug.log(222)

yield return 1; //返回内容为1

Debug.log(333)

yield return "sss"; //返回内容为"sss"

yield break; //跳出，类似普通函数中的return语句

yield return 999; //由于break语句，该内容无法返回

}

void Start()

{

IEnumerator e = TestCoroutine();

while (e.MoveNext())

{

Debug.Log(e.Current); //依次输出枚举接口返回的值

}

}

/\* 枚举接口的定义

public interface IEnumerator

{

object Current

{

get;

}

bool MoveNext();

void Reset();

}\*/

/\*运行结果：

Null

1

sss

\*/

首先注意注释部分枚举接口的定义  
Current属性为只读属性，返回枚举序列中的当前位的内容  
MoveNext()把枚举器的位置前进到下一项，返回布尔值，新的位置若是有效的，返回true；否则返回false  
Reset()将位置重置为原始状态

**再看下Start函数中的代码，就是将yield return 语句中返回的值依次输出。**  
**第一次MoveNext()后，Current位置指向了yield return 返回的null，该位置是有效的（这里注意区分位置有效和结果有效，位置有效是指当前位置是否有返回值，即使返回值是null；而结果有效是指返回值的结果是否为null，显然此处返回结果是无意义的）所以MoveNext()返回值是true；  
第二次MoveNext()后，Current新位置指向了yield return 返回的1，该位置是有效的，MoveNext()返回true  
第三次MoveNext()后，Current新位置指向了yield return 返回的"sss"，该位置也是有效的，MoveNext()返回true  
第四次MoveNext()后，Current新位置指向了yield break，无返回值，即位置无效，MoveNext()返回false，至此循环结束**

最后输出的运行结果跟我们分析是一致的。关于C#是如何实现迭代器的功能，有兴趣的可以看下容器类源码中关于迭代器部分的实现就明白了

**三、原理**

先来回顾下Unity的协程具体有些功能：

1. 将协程代码中由yield return语句分割的部分分配到每一帧去执行。
2. yield return 后的值是等待类([WaitForSeconds](https://docs.unity3d.com/ScriptReference/WaitForSeconds.html)、[WaitForFixedUpdate](https://docs.unity3d.com/ScriptReference/WaitForFixedUpdate.html))时需要等待相应时间。
3. yield return 后的值还是协程([Coroutine](https://docs.unity3d.com/ScriptReference/Coroutine.html))时需要等待嵌套部分协程执行完毕才能执行接下来内容。

// case 1

IEnumerator Coroutine1()

{

//do something xxx //假如是第N帧执行该语句

yield return 1; //等一帧

//do something xxx //则第N+1帧执行该语句

Debug.log(“1”);

}

// case 2

IEnumerator Coroutine2()

{

//do something xxx //假如是第N秒执行该语句

yield return new WaitForSeconds(2f); //等两秒

//do something xxx //则第N+2秒执行该语句

}

// case 3

IEnumerator Coroutine3()

{

//do something xxx

yield return StartCoroutine(Coroutine1()); //等协程Coroutine1执行完

//do something xxx

Debug.log(“test”)；

}

好了，知道了IEnumerator函数和yield return语法之后，在看到上面几个协程的功能，是不是对如何实现协程有点头绪了？

**case1 : 分帧**

实现分帧执行之前，先将上述迭代器的代码简单修改下，看下输出结果

IEnumerator TestCoroutine()

{

Debug.Log("TestCoroutine 1");

yield return null;

Debug.Log("TestCoroutine 2");

yield return 1;

}

void Start()

{

IEnumerator e = TestCoroutine();

while (e.MoveNext())

{

Debug.Log(e.Current); //依次输出枚举接口返回的值

}

}

/\*运行结果

TestCoroutine 1

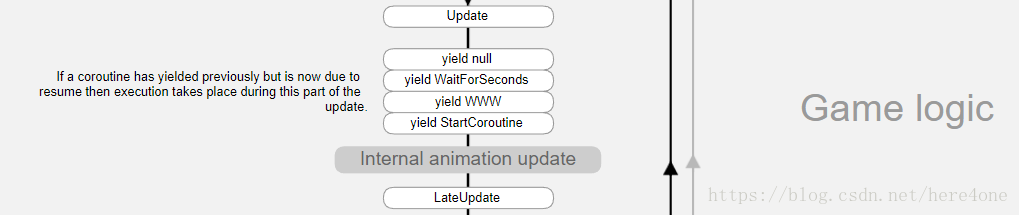
Null

TestCoroutine 2

1

\*/

前面有说过，每次MoveNext()后会返回yield return后的内容，那yield return之前的语句怎么办呢？  
**当然也执行啊，遇到yield return语句之前的内容都会在MoveNext()时执行的。**  
**到这里应该很清楚了，只要把MoveNext()移到每一帧去执行，不就实现分帧执行几段代码了么！**

既然要分配在每一帧去执行，那当然就是Update和LateUpdate了。**这里我个人喜欢将实现代码放在LateUpdate之中，为什么呢？因为Unity中协程的调用顺序是在Update之后，LateUpdate之前，所以这两个接口都不够准确**；但在LateUpdate中处理，至少能保证协程是在所有脚本的Update执行完毕之后再去执行。  
  
现在可以实现最简单的协程了

IEnumerator e = null;

void Start()

{

e = TestCoroutine();

}

void LateUpdate()

{

if (e != null)

{

if (!e.MoveNext())

{

e = null;

}

}

}

IEnumerator TestCoroutine()

{

Log("Test 1");

yield return null; //返回内容为null

Log("Test 2");

yield return 1; //返回内容为1

Log("Test 3");

yield return "sss"; //返回内容为"sss"

Log("Test 4");

yield break; //跳出，类似普通函数中的return语句

Log("Test 5");

yield return 999; //由于break语句，该内容无法返回

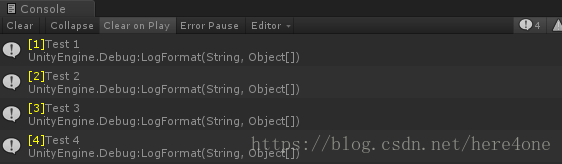
}

void Log(object msg)

{

Debug.LogFormat("<color=yellow>[{0}]</color>{1}", Time.frameCount, msg.ToString());

}

  
再来看看运行结果，黄色中括号括起来的数字表示当前在第几帧，很明显我们的协程完成了每一帧执行一段代码的功能。

**case2: 延时等待**

要是完全理解了case1的内容，相信你自己就能完成“延时等待”这一功能，其实就是加了个计时器的判断嘛！  
**既然要识别自己的等待类，那当然要获取Current值根据其类型去判定是否需要等待。假如Current值是需要等待类型，那就延时到倒计时结束；而Current值是非等待类型，那就不需要等待，直接MoveNext()执行后续的代码即可。**  
这里着重说下“延时到倒计时结束”。既然知道Current值是需要等待的类型，那此时肯定不能在执行MoveNext()了，否则等待就没用了；接下来当等待时间到了，就可以继续MoveNext()了。可以简单的加个标志位去做这一判断，同时驱动MoveNext()的执行。

private void OnGUI()

{

if (GUILayout.Button("Test")) //注意：这里是点击触发，没有放在start里，为什么？

{

enumerator = TestCoroutine();

}

}

void LateUpdate()

{

if (enumerator != null)

{

bool isNoNeedWait = true, isMoveOver = true;

var current = enumerator.Current;

if (current is MyWaitForSeconds)

{

MyWaitForSeconds waitable = current as MyWaitForSeconds;

isNoNeedWait = waitable.IsOver(Time.deltaTime);

}

if (isNoNeedWait)

{

isMoveOver = enumerator.MoveNext();

}

if (!isMoveOver)

{

enumerator = null;

}

}

}

IEnumerator TestCoroutine()

{

Log("Test 1");

yield return null; //返回内容为null

Log("Test 2");

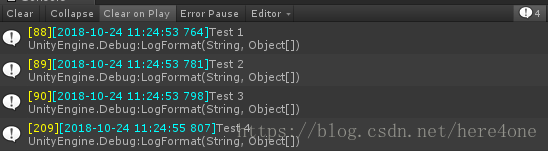
yield return 1; //返回内容为1

Log("Test 3");

yield return new MyWaitForSeconds(2f); //等待两秒

Log("Test 4");

}

  
运行结果里黄色表示当前帧，青色是当前时间，很明显等待了2秒（虽然有少许误差但总体不影响）。  
**上述代码中，把函数触发放在了Button点击中而不是Start函数中？  
这是因为我是用Time.deltaTime去做计时，假如放在了Start函数中，Time.deltaTime会受Awake这一帧执行时间影响，时间还不短（我测试时有0.1s左右），导致运行结果有很大误差，不到2秒就结束了，有兴趣的可以自己试一下~**

**case3: 协程嵌套等待**

协程嵌套等待也就是下面这种样子，在实际情况中使用的也不少。

IEnumerator Coroutine1()

{

//do something xxx

yield return null;

//do something xxx

yield return StartCoroutine(Coroutine2()); //等待Coroutine2执行完毕

//do something xxx

yield return 3;

}

IEnumerator Coroutine2()

{

//do something xxx

yield return null;

//do something xxx

yield return 1;

//do something xxx

yield return 2;

}

实现原理的话基本与延时等待完全一致，这里我就不贴例子代码了，最后会放出完整工程的。  
需要注意下协程嵌套时的执行顺序，先执行完内层嵌套代码再执行外层内容；即更新结束条件时要先更新内层协程（上例Coroutine2）在更新外层协程（上例Coroutine1）。

**四、总结**

前一节只是把每块内容的原理用例子代码实现了一下，实际使用中这样肯定不行，需要更通用的接口。  
我按照Unity的接口方式把上述这些功能用相同名称封装了一下，并做了一些测试样例与Unity原生接口运行结果作对比   
下图是最后一个测试样例的代码和运行结果，可以看出表现是完全一致的。

//Hi是命名空间

private void OnGUI()

{

GUILayout.BeginHorizontal();

if (GUILayout.Button("自己 嵌套的协程"))

{

Hi.CoroutineMgr.Instance.StartCoroutine(TestNesting());

}

GUILayout.Space(20);

if (GUILayout.Button("Unity 嵌套的协程"))

{

StartCoroutine(UnityNesting());

}

GUILayout.EndHorizontal();

}

IEnumerator TestNesting()

{

Log("Nesting 1");

yield return Hi.CoroutineMgr.Instance.StartCoroutine(TestNesting\_\_());

Log("Nesting 2");

}

IEnumerator TestNesting\_\_()

{

Log("Nesting\_\_ 1");

yield return Hi.CoroutineMgr.Instance.StartCoroutine(TestNormalCoroutine());

Log("Nesting\_\_ 2");

yield return Hi.CoroutineMgr.Instance.StartCoroutine(TestWaitFor());

Log("Nesting\_\_ 3");

}

IEnumerator UnityNesting()

{

LogWarn("UnityNesting 1");

yield return StartCoroutine(UnityTesting\_\_());

LogWarn("UnityNesting 2");

}

IEnumerator UnityTesting\_\_()

{

LogWarn("UnityTesting\_\_ 1");

yield return StartCoroutine(UnityNormalCoroutine());

LogWarn("UnityTesting\_\_ 2");

yield return StartCoroutine(UnityWaitFor());

LogWarn("UnityTesting\_\_ 3");

}

void Log(string message)

{

Debug.LogFormat("<color=yellow>[{0}]</color>-<color=cyan>[{1}]</color>{2}", Time.frameCount,

System.DateTime.Now.ToString("yyyy-MM-dd hh:mm:ss fff"), message);

}

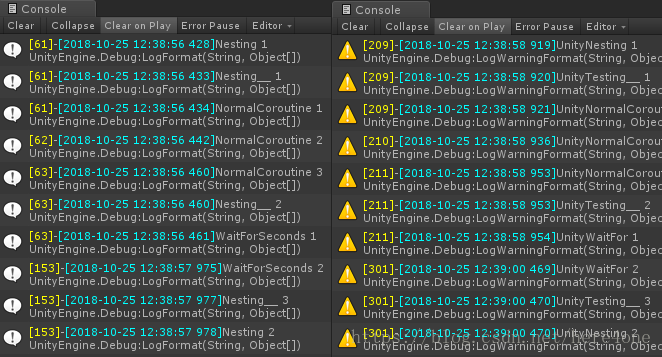
void LogWarn(string message)

{

Debug.LogWarningFormat("<color=yellow>[{0}]</color>-<color=cyan>[{1}]</color>{2}",

Time.frameCount, System.DateTime.Now.ToString("yyyy-MM-dd hh:mm:ss fff"), message);

}



# unity 协程原理与线程的区别

说到协程，我们首先回顾以下线程与进程这两个概念。在操作系统（os）级别，有进程（process）和线程（thread）两个我们看不到但又实际存在的“东西”，这两个东西都是用来模拟“并行”的，写操作系统的程序员通过用一定的策略给不同的进程和线程分配CPU计算资源，来让用户“以为”几个不同的事情在“同时”进行“。在单CPU上，是os代码强制把一个进程或者线程挂起，换成另外一个来计算，所以，实际上是串行的，只是“概念上的并行”。在现在的多核的cpu上，线程可能是“真正并行的”。

**1 线程、进程、协程的区别**

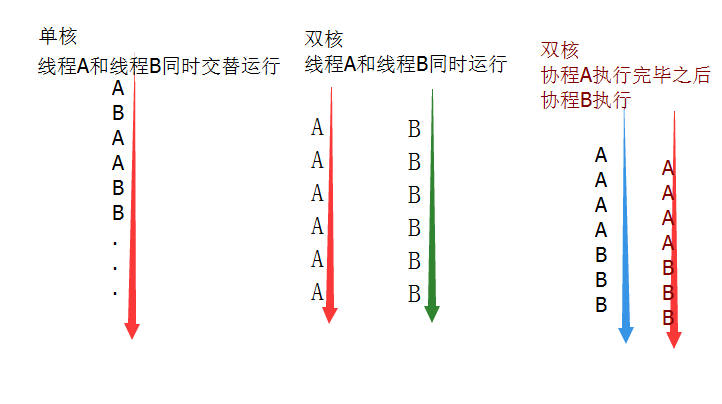
进程拥有自己独立的堆和栈，既不共享堆，亦不共享栈，进程由操作系统调度。

线程拥有自己独立的栈和共享的堆，共享堆，不共享栈，线程亦由操作系统调度(标准线程是的)。

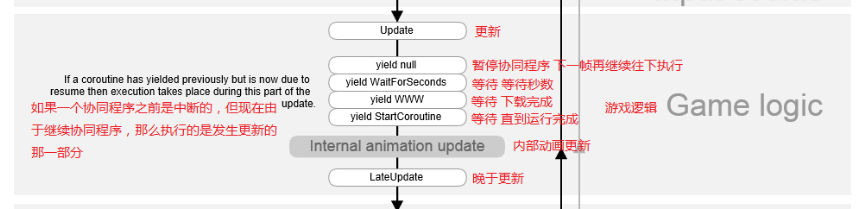
协程和线程一样共享堆，不共享栈，协程由程序员在协程的代码里显示调度。

一个应用程序一般对应一个进程，一个进程一般有一个主线程，还有若干个辅助线程，线程之间是平行运行的，在线程里面可以开启协程，让程序在特定的时间内运行。

协程和线程的区别是：协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。

打个比方吧，假设有一个操作系统，是单核的，系统上没有其他的程序需要运行，有两个线程 A 和 B ，A 和 B 在单独运行时都需要 10 秒来完成自己的任务，而且任务都是运算操作，A B 之间也没有竞争和共享数据的问题。现在 A B 两个线程并行，操作系统会不停的在 A B 两个线程之间切换，达到一种伪并行的效果，假设切换的频率是每秒一次，切换的成本是 0.1 秒(主要是栈切换)，总共需要 20 + 19 \* 0.1 = 21.9 秒。如果使用协程的方式，可以先运行协程 A ，A 结束的时候让位给协程 B ，只发生一次切换，总时间是 20 + 1 \* 0.1 = 20.1 秒。如果系统是双核的，而且线程是标准线程，那么 A B 两个线程就可以真并行，总时间只需要 10 秒，而协程的方案仍然需要 20.1 秒。   


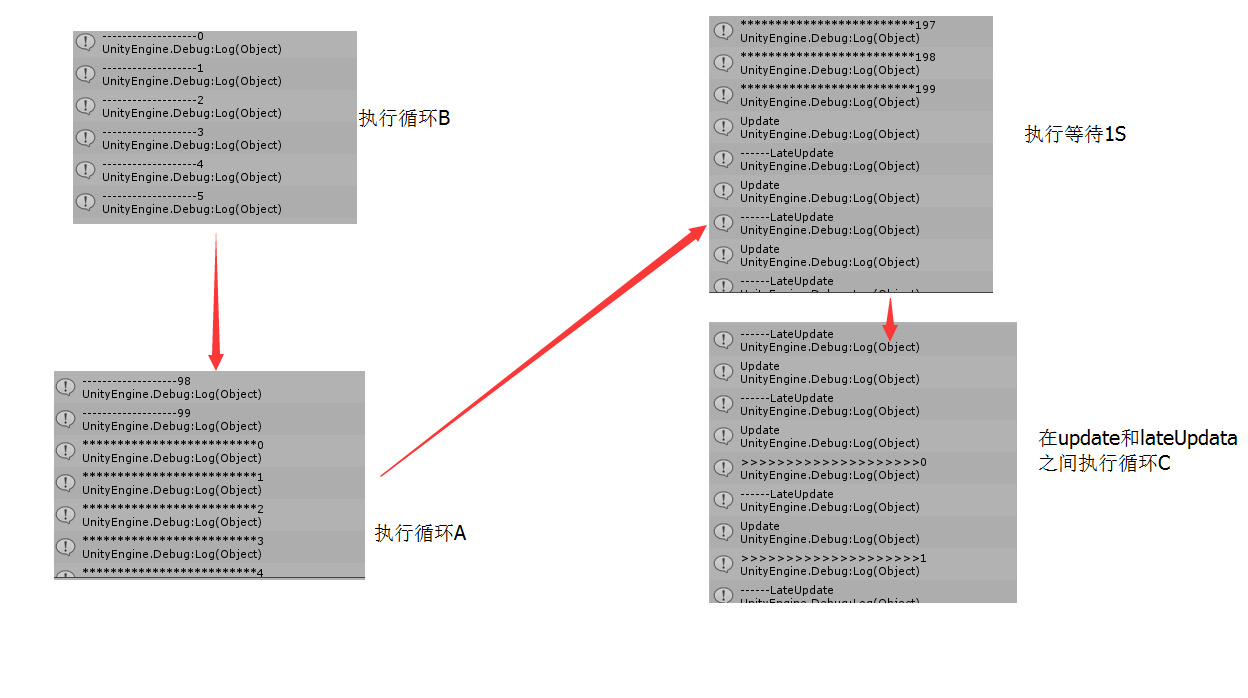
**Unity协程执行原理**

unity中协程执行过程中，通过yield return XXX，将程序挂起，去执行接下来的内容，注意协程不是线程，在为遇到yield return XXX语句之前，协程额方法和一般的方法是相同的，也就是程序在执行到yield return XXX语句之后，接着才会执行的是 StartCoroutine（）方法之后的程序，走的还是单线程模式，仅仅是将yield return XXX语句之后的内容暂时挂起，等到特定的时间才执行。   
那么挂起的程序什么时候才执行，这就要看monoBehavior的生命周期了。   
  
也就是协同程序主要是update（）方法之后，lateUpdate()方法之前调用的，接下来我们通过一个小例子去理解一下。

1. using UnityEngine;
2. using System.Collections;
3. using System.Threading;
4. public class test : MonoBehaviour
5. {
7. void Start()
8. {
9. StartCoroutine(tt());*//开启协程*
10. for (int i = 0; i < 200; i++) *//循环A*
11. {
12. Debug.Log("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" + i);
13. Thread.Sleep(10);
14. }
15. }

18. IEnumerator tt()
19. {
20. for (int i = 0; i < 100; i++) *//循环B*
21. {
22. Debug.Log("-------------------" + i);
23. }
25. yield return new WaitForSeconds(1); *//协程1*
27. for (int i = 0; i < 100; i++) *//循环C*
28. {
29. Debug.Log(">>>>>>>>>>>>>>>>>>>>" + i);
30. yield return null; *//协程1*
31. }
32. }
34. *// 更新数据*
35. void Update()
36. {
37. Debug.Log("Update");
38. }
40. *//晚于更新*
41. void LateUpdate()
42. {
43. Debug.Log("------LateUpdate");
44. }

47. }

程序的运行结果为：   
  
先执行循环B，然后执行循环A，然后执行update（）和lateUpdate（）的方法，等待1S之后，在updat（）和lateupda（）之间执行循环C的输出。