Unity 内置协程

本篇教程主要讲解了如何使用 Unity 中的协程。

介绍

每个 Unity 脚本都有两个重要的功能：开始和更新。 前者的作用是当一个对象被创建后，在每一帧对后者进行调用。设计规定，下一帧只有更新结束才能开始。这样会出现一个设计局限：更新的持续时间不太容易超过一帧。

坦白而言，每个你能想到的自定义行为都能用开始和更新进行实现。然而，基于多帧的事件发生要难实现一些（例如动画，对白，等待，...）。这是因为其设定无法写入一个持续的流中，必须分段，分布在很多帧中。这往往让代码难写，维护也很困难。

如果能在短暂的单帧中不受任何约束那就非常完美了。如果你是程序员，那么肯定知道线程的概念。线程是并行执行的代码段，使用线程需要谨慎。这是因为当多线程不加限制地共享一个变量会出现问题。Unity 的设计并不建议使用线程。然而，Unity 提供了折中的方案：协程。协程持续超过一帧的时间。此

外，协程可以在任意情况下中断和恢复执行。

协程是常规的 C#函数，返回一个 IEnumerator。为了执行协程（并不同于以往的函数），必须使用 StartCoroutine 方法(UnityDoc)。例如：

[C#] 纯文本查看 复制代码

```
void  Start ()
{
        // Execute A as a coroutine
        StartCoroutine( A() );
}

IEnumerator A ()
{
        ...
}
```

将 A 当做协程执行。StartCoroutine 方法立即终止，同时产生新的协程并行执行。

同步等待

如果你之前用过协程，那么应该已经遇到过 WaitForSeconds (UnityDoc)。像继承 YieldInstruction 的其它类，它允许协程短暂的暂停。当用 yield 进行连接时， WaitForSeconds 提供了一种方式去延迟剩余代码的执行。

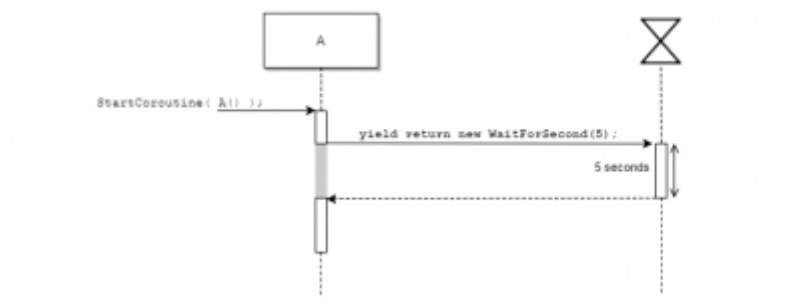
下面的代码展示了如何使用协程：

[C#] 纯文本查看 复制代码

```
IEnumerator A()
{
        ...
```

```
        yield return new WaitForSeconds(10f);

        ...
}
```



UML 的序列图如上(Wikipedia)，验证了 WaitForSeconds 的作用。当调用协程（即调用 A），它暂停执行，直到消耗一定的时间。这个等待称为同步，因为协程等待另一个操作的完成。

异步协程

Unity 还允许在现有协程中开启一个新的协程。最简单的方法就是使用 StartCoroutine。这么调用的话，新生的协程会和以前的协程共存。它们不发生直接交互，最重要的是它们不会相互等待。与之前的同步等待相比，这种情况是异步的，两个协程不要试图保持同步。

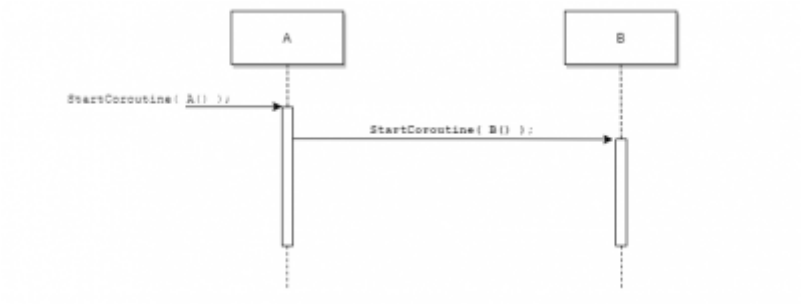
[C#] 纯文本查看 复制代码

```
IEnumerator A()
{
        ...

        // Starts B as a coroutine, and continue the execution
```

```
        StartCoroutine( B() );

        ...
}
```



需要注意的是，在这个例子中 B 是一个完全独立的协同程序。终止不会影响 B，反之亦然

同步协程

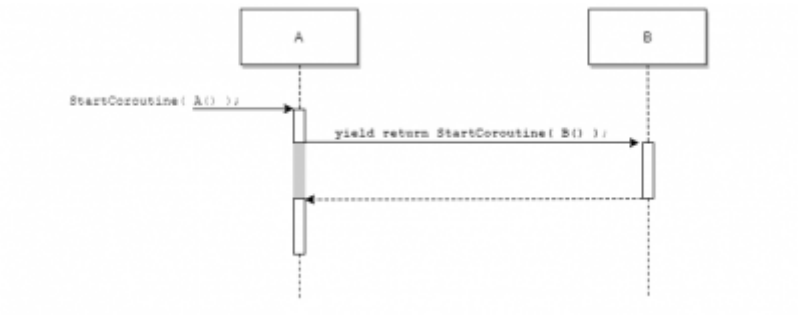
可以执行嵌套的协程并等待其实行完毕。最简单的办法就是使用 yield 返回。

[C#] 纯文本查看 复制代码

```
IEnumerator A()
{
        ...

        // Waits for B to terminate
        yield  return  StartCoroutine( B() );

        ...
}
```



值得注意的是，由于执行 B 期间暂停了 A，这种特殊情况下不需要启动另一个协程。有人可能会像下面这样试图优化协程：

[C#] 纯文本查看 复制代码

```
IEnumerator A()
{
        ...

        // Executes B as part of A
        B();

        ...
}
```

B 的执行和普通函数有一样的效果，不同的是 B 是在单帧内执行的。相反，通过使用 StartCoroutine，A 已经暂定的同时下一帧可以开始进行。

引入这个例子是为了介绍更加复杂的例子，同步协程。

并行协程

当协程通过 StartCoroutine 启动时，返回了一个特殊的对象。 这可以用来查询协程的状态，随时等待其结束。

下面的例子中，协程 B 是异步执行的。父类 A 可以继续执行直到 B 需要的时

候。如果有必要，它可以为了同步等待让步于 B 的引用。

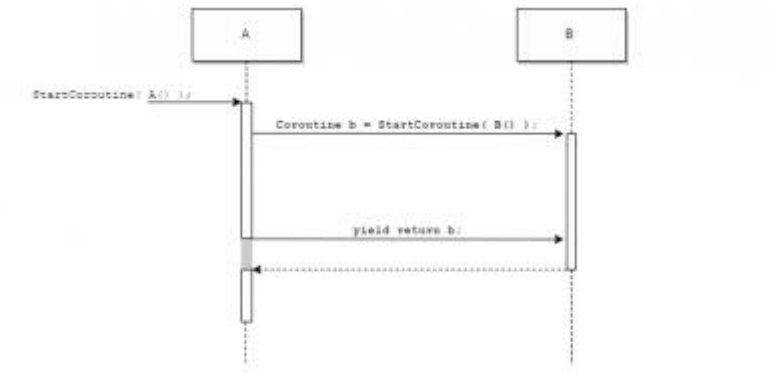
```
IEnumerator A()
{
        ...

        // Starts B as a coroutine and continues the execution
        Coroutine b = StartCoroutine( B() );

        ...

        // Waits for B to terminate
        yield  return  b;

        ...
}
```



如果你想要开始几个并行协程的话这会非常有用，所有代码在同一刻运行：

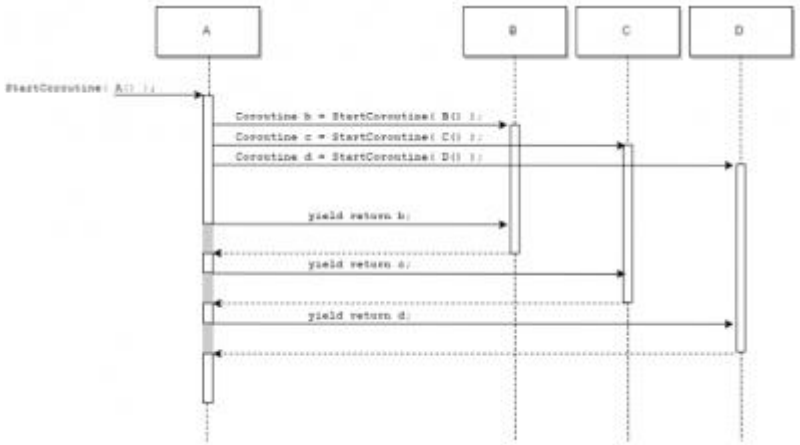
```
IEnumerator A()
{
        ...
        // Starts B, C, and D as coroutines and continues the execution
```

```
        Coroutine b = StartCoroutine( B() );
        Coroutine c = StartCoroutine( C() );
        Coroutine d = StartCoroutine( D() );

        ...

        // Waits for B, C and D to terminate
        yield return b;
        yield return c;
        yield return d;

        ...
}
```



这种新模式允许任意数量的并行计算，当这些并行计算终止时恢复执行。

总结

这篇文章展示了通过有效利用协程的几种不同的模式来完成你的游戏。本系列接下来的文章将集中在如何扩展协程来支持自定义的等待和事件。