+<http://www.jianshu.com/p/1d93ece664e2>

<http://blog.csdn.net/llj1985/article/details/51505111>

**Unity3D中脚本的执行顺序和编译顺序**

<http://www.cnblogs.com/champ/p/execorder.html>

<http://blog.csdn.net/qitian67/article/details/18516503>

**简书**

<http://www.jianshu.com/p/1d93ece664e2>

注意：如果我们在运行中动态添加脚本时，也会立即执行到Start方法

**Awake函数**

当脚本实例被加载时会调用Awake函数，Awake函数在所有的游戏对象被初始化完毕之后才会被调用；在脚本实例的整个生命周期中，Awake函数仅调用一次。总结如下:

1. 如果游戏对象（即gameObject）的初始状态为关闭状态，那么运行程序，Awake函数不会执行；反之，如果游戏对象的初始状态为开启状态，那么Awake函数会执行。并且，值得注意的一点是，Awake函数的执行与否与脚本实例的状态（启用或禁用）并没有关系，而是与脚本实例所绑定的游戏对象的状态有关。
2. 在不切换场景的前提下，原本处于关闭状态的游戏对象，当它被首次开启时，Awake函数会执行，并且是只在首次开启时才会执行；而已经开启（激活）的游戏对象（Awake函数已经执行过一次），将它关闭后再次开启，Awake函数不会再次执行。看来这是对应了手册中说明的在脚本实例的整个生命周期中仅执行一次的情况。
3. 如果重新加载场景，那么场景内Awake函数的执行情况重新遵循上述两点。

**实际上在切换场景的时候会发现awake会重新调用**

**Start函数**

当Update函数第一次被调用前会调用Start函数；Start函数只在脚本实例被启用时才会执行；Start函数总是在Awake函数之后执行。在此基础之上，自己的测试总结如下 （没有特殊说明，均为在不切换场景的前提下）：

1. 如果游戏对象被关闭，那么Start函数不会执行。想一想这是理所当然的，游戏对象关闭了，脚本实例不会被加载，Awake函数不会执行，Start函数不会执行，Update函数等也不会执行。

（2）如果游戏对象开启了，对象上绑定的脚本实例被禁用了，那么Start函数不会执行。这是Start函数的特点，只有在脚本实例被启用时它才会执行，这个说明中有提到。注意，这与Awake函数是有区别的。

（3）如果游戏对象开启了，对象上绑定的脚本实例也开启了，那么Start函数会执行。并且Start函数只会在脚本实例首次被开启时才会执行。如果是已经开启过的脚本实例被关闭后再次开启，那么Start函数不会再次执行。

（4）如果重新加载场景，那么场景内Start函数的执行情况重新遵循上述三点。

**实际上在切换场景的时候会发现start会重新调用**

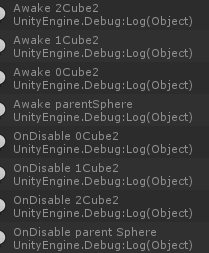
**如果将当前脚本禁止了，会发现update也不会再调用了**

除此之外，还有对Awake函数和Start函数都比较重要的一点，那就是当游戏对象之间存在父子关系时（不论层级的多少）。这时，父游戏对象的状态（开启或关闭）完全决定了子游戏对象上的脚本函数的执行情况。总的来说就是，只有在父游戏对象被开启（active=true）的状态下，程序才会考虑是否调用子游戏对象上的脚本函数（Awake与Start等）。自上而下，以此类推。（已经验证），但是如果只是设置了脚本的enabled，则只会影响到当前绑定的GameObject

但是如果是通过代码设置父对象为active=false，实际上还是会执行子物体的Awake的



在父物体绑定脚本的Awake中添加gameObject.SetActive(false);子物体的Awake优先于父物体的Awake执行了



**但是在编辑器中将父物体设置active=false，就不会有任何输出了**

**记住：子物体的Awake、Start等等是有可能优先于父物体的Awake、Start执行的**

void Awake()

{

Debug.Log("Awake");

// this.enabled = false;

gameObject.SetActive(false);

}

void Start()

{

Debug.Log("Start");

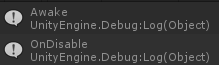
}

void OnDisable()

{

Debug.Log("OnDisable");

}



上面的代码中如果将GameObject.SetActive(false)修改为this.enabled=false，结果一样

**OnDisable**

一个GameObject上同时添加三个脚本

public class DisableTest : MonoBehaviour

{

void Awake()

{

Debug.Log("Awake 0");

}

void Start()

{

Debug.Log("Start 0");

}

void OnDisable()

{

Debug.Log("OnDisable 0");

}

}

public class DisableTest1 : MonoBehaviour {

void Awake()

{

Debug.Log("Awake 1");

//this.enabled = false;

//gameObject.SetActive(false);

}

void Start()

{

Debug.Log("Start 1");

}

void OnDisable()

{

Debug.Log("OnDisable 1");

}

}

public class DisableTest2 : MonoBehaviour {

void Awake()

{

Debug.Log("Awake 2");

}

void Start()

{

Debug.Log("Start 2");

}

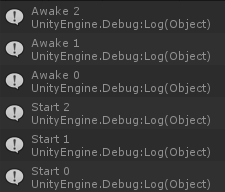
void OnDisable()

{

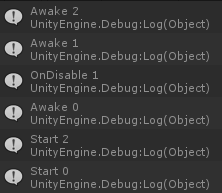
Debug.Log("OnDisable 2");

}

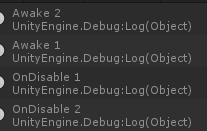
}



如果开启this.enabled = false;



如果开启gameObject.SetActive(false);



从上面的结果也可以看出SetActive影响所有脚本的OnDisable（前提是它可以执行），而enabled仅仅影响当前脚本

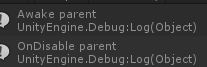
实际上我们可以发现，调用

this.enabled = false;  
this.enabled = true;  
//gameObject.SetActive(false);  
//gameObject.SetActive(true);

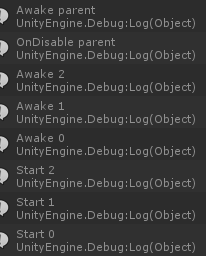
上面两种方式都可以调用OnEnable与Disable，但是如果状态没有变化，不会调动OnEnable与OnDisable

如果有父子关系，cube的父节点cube1

如果使用gameObject.SetActive(false);



如果使用this.enabled = false;



**Script03.cs**

public class Script03 : MonoBehaviour {

bool hasFixedUpdate = false;

bool hasUpdate = false;

void Reset()

{

Debug.Log("Script03 Reset");

}

void Awake()

{

Debug.Log("Script03 Awake");

}

void Start () {

Debug.Log("Script03 Start");

}

public void FixedUpdate()

{

if (!hasFixedUpdate)

{

Debug.Log("Script03 FixedUpdate");

hasFixedUpdate = true;

gameObject.AddComponent<Test0>();

}

}

public void Update () {

if (!hasUpdate)

{

Debug.Log("Script03 Update");

hasUpdate = true;

}

}

}

**Test0.cs**

public class Test0 : MonoBehaviour {

bool hasFixedUpdate = false;

bool hasUpdate = false;

void Awake()

{

Debug.Log("Test0 Awake");

}

void OnEnable()

{

Debug.Log("Test0 OnEnable");

}

void Start () {

Debug.Log("Test0 Start");

}

void FixedUpdate()

{

if (!hasFixedUpdate)

{

Debug.Log("Test0 FixedUpdate");

hasFixedUpdate = true;

}

}

void Update () {

if (!hasUpdate)

{

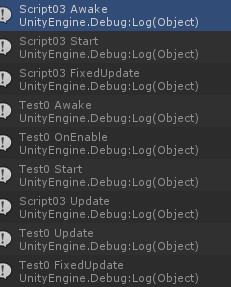
Debug.Log("Test0 Update");

hasUpdate = true;

}

}

}



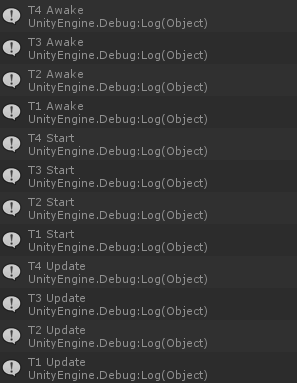
**父子，兄弟，多个脚本执行的先后顺序**

通常是先执行子节点的脚本，再执行父节点的脚本，当然我们也可以改变这种顺序

Edit -> Project Settings -> Script Execution Order

**执行顺序**

实际上顺序是不固定的，比如下面的T1与T4脚本是挂在同一个GameObject上：



注意：Awake和OnEnable是一个例外，这两者总是成对出现，

a.Awake()

a.OnEnable()

b.Awake()

b.OnEnable()

c.Awake()

c.OnEnable()

同时注意：OnDisable与OnDestroy不具备顺序特点

前面的可能是abc，而这两个可能变为bac（已验证）

例子：

两个物体，四个脚本：Sphere(a.cs, b.cs) Plane(c.cs, d.cs)

结果：

运行时

d.Awake

d.OnEnable

c.Awake

c.OnEnable

b.Awake

b.OnEnable

a.Awake

a.OnEnable

d.Start

c.Start

b.Start

a.Start

d.FixedUpdate

c.FixedUpdate

b.FixedUpdate

a.FixedUpdate

d.Update

c.Update

b.Update

a.Update

d.LateUpdate

c.LateUpdate

b.LateUpdate

a.LateUpdate

d.OnGUI

c.OnGUI

b.OnGUI

a.OnGUI

关闭时：

a.OnDisable

b.OnDisable

a.OnDestroy

b.OnDestroy

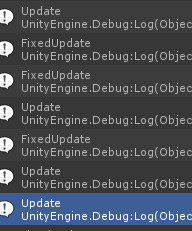
c.OnDisable

d.OnDisable

c.OnDestroy

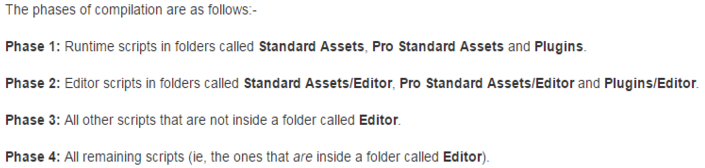
d.OnDestroy

关于Update与FixedUpdate

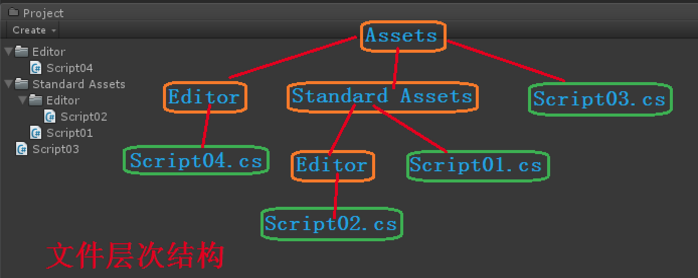


**编译顺序**

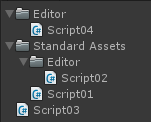
关于脚本的编译顺序官方的说法也有点模糊

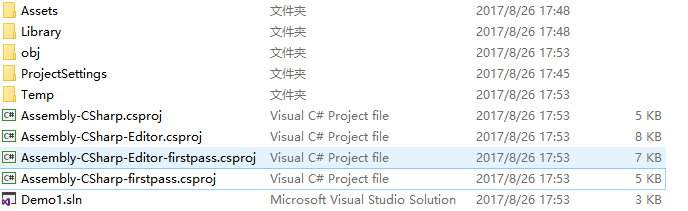


由于脚本的编译顺序会涉及到特殊文件夹，比如上面提到的Plugins。Editor还有Standard Assets等标准的资源文件夹，所以脚本的放置位置就非常重要了，下面用一个例子来说明不同文件夹中的脚本编译顺序：



实际上，如果细心的话会发现，如果你的项目中建立如上图所示的文件夹层次结构时，编译项目之后会在项目文件夹中申城一些文件名中包含Editor。Firstpass这些字样的项目文件。比如按照上图的文件结构，我们打开项目文件夹来看一下产生的项目是什么样的？





**1 首先从脚本语言来看，Unity3d支持三种脚本语言，都会被编译成CLI的DLL**

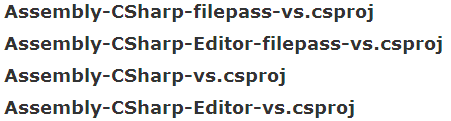
如果项目中包含C#脚本，那么unity3d会产生以Assembly-CSharp为前缀的工程，名字中包含“vs”是产生给Vistual Studio使用的，不包含“vs”是产生给MonoDevelop使用的。



如果项目中这三种脚本都存在，那么Unity将会生成3种前缀类型的工程

**2 对于每一种脚本语言，根据脚本放置的位置（其实也部分根据脚本的作用，比如编辑器扩展脚本，就必须放在Editor文件夹下），Unity会生成4种后缀的功能，其中firstpass表示先编译，Editor表示放在Editor文件夹下的脚本**

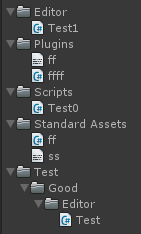
在上面的示例中，我们得到了两套项目工程文件：分别被Visual Studio和MonoDevelop使用（后缀不包含vs），为了简单起见，我们只分析vs项目，得到的文件表如下：

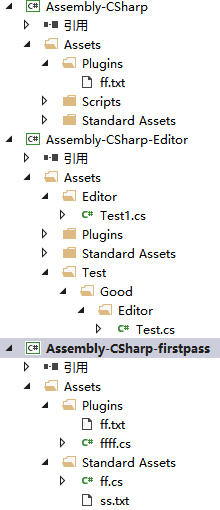


根据官方的解释，它们的编译顺序如下：

1. 所有在Standard Assets、Pro StandardAssets或者Plugins文件夹中的脚本会产生一个Assembly-CSharp-firstpass-vs.csproj文件，并且先编译；
2. 所有在Standard Assets/Editor、Pro StandardAssets/Editor或者Plugins/Editor文件夹中的脚本产生Assembly-CSharp-Editor-firstpass-vs.csproj工程文件，接着编译；
3. 所有在Assets/Editor外面的，并且不再（1）、（2）中的脚本文件（一般这些脚本就是我们自己写的非编辑器扩展脚本）会产生Assembly-CSharp-vs.csjroy工程文件，被编译；
4. 所有在Assets/Editor中的脚本产生一个Assembly-CSharp-Editor-vs.csproj工程文件，被编译。

注意：它会将所有在Assets下包含除了（2）之外的Editor全部包含进去





之所有按照这样建立功能并按此顺序编译，也是因为dll间存在的依赖关系所决定的。

