1. 事件函数

参考：[Unity3D中API常用方法和类详细讲解（事件函数、GameObject类、MonoBehaviour 类） (daimajiaoliu.com)](https://www.daimajiaoliu.com/daima/4edcd03af100410)

[unity常用API（一）\_牛客博客 (nowcoder.net)](https://blog.nowcoder.net/n/a908a90793db4f759bb2dbf401d4c253)

### 1.什么是事件函数

我们在unity创建的脚本都会继承MonoBehavior（当然可以删除继承），如图1-1是MonoBehavior的继承结构，

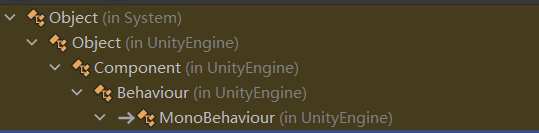


图 1-1 MonoBehavior继承结构

如图2-2我们可以查看事件函数的调用顺序

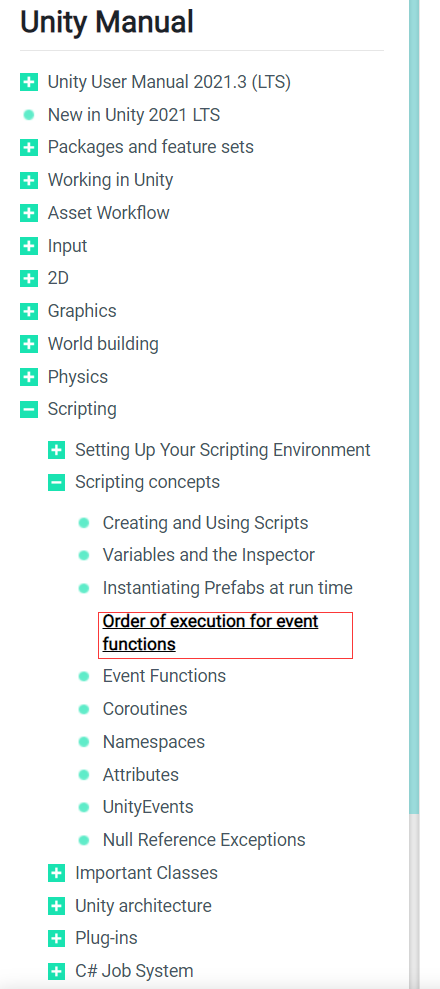


图 1-2

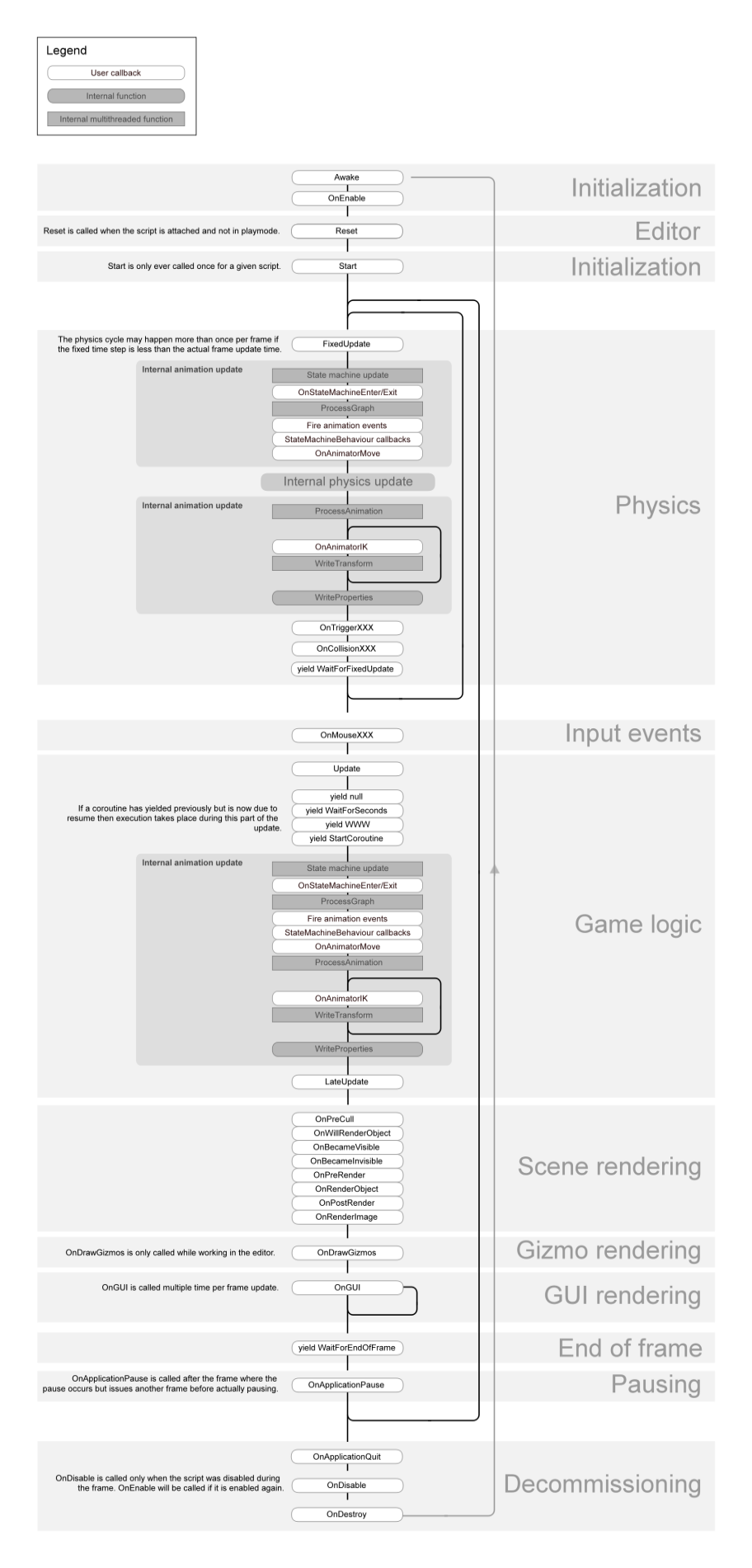


图 1-2 Event function调用顺序

### 第一次场景加载

#### Awake函数

始终在任何Start函数之前被调用，并在预制体实例化之后调用。如果预制体是inactive状态下则不会调用，直到变为active。

参考：[unity Awake、OnEnable、Start等的基本描述 - 简书 (jianshu.com)](https://www.jianshu.com/p/628e855842df)

#### OnEnable函数

只在object active下调用。该函数在对象启用之后被调用（This function is called just after the object is enabled.），会发生在Monobehavior被实例化，比如关卡加载和gameobject被实例化（携带的component脚本随着实例化）等情况下。

（This happens when a MonoBehaviour instance is created, such as when a level is loaded or a **GameObject** with the script component is instantiated.）

### Editor（编辑状态下？）

#### Reset是指Editor的reset，

当脚本第一次附加到一个对象上以及使用Reset命令时，调用Reset来初始化脚本的属性。如图1-4

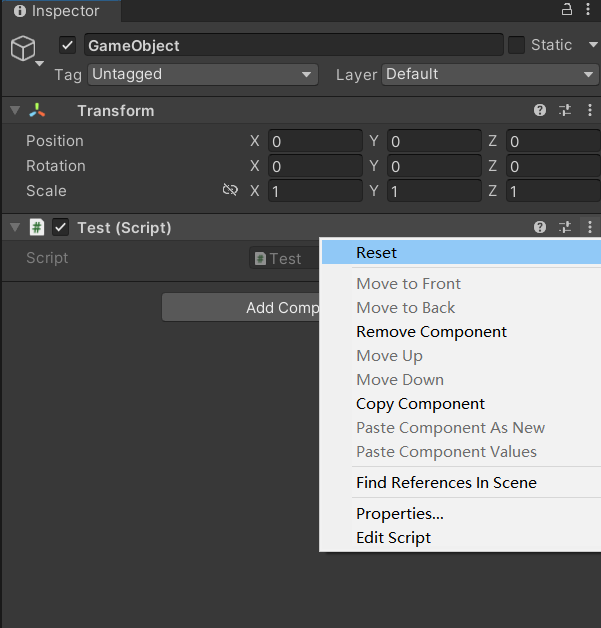


图 1-4 reset

经过测试，不运行项目的时候可以触发。运行项目不能触发Reset方法。

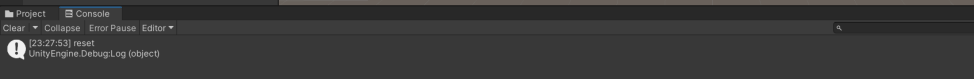


图 1-5 Reset打印

#### OnValidate

每次设置脚本的属性都会调用，包括反序列化对象时，这可能发生在不同的时间，例如编辑器打开新的场景或者在domain重新加载后。

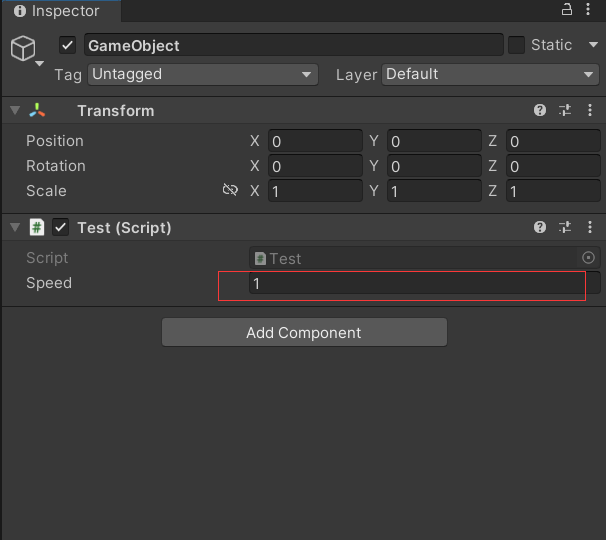


图1-6

（每次运行会触发，算是打开新的场景？实例化对象？）

### 在第一帧前

#### start函数

在第一帧前调用，不过只在脚本enable状态下有效

一般初始化的操作在这里做，如果需要更早，就需要在Awake中

### 在帧之间

#### 1．onApplicationPause

这在检测到暂停的帧结束时调用，在正常帧更新之间有效。调用 OnApplicationPause 后将发出一帧额外的帧，以允许游戏显示指示暂停状态的图形。（意思是允许在暂停状态下做一些画图操作？）

### Update

#### 1.FixedUpdate

固定时间执行，和帧率无关，在FixedUpdate之后会马上进行physics相关的计算（看图 1-2 Event function调用顺序），还有因为固定时间执行，所以在FixedUpdate中不需要乘**Time.deltaTime**。

#### 2.Update

每一帧调用一次，一般做什么更新就在这个函数中。

#### 3.LateUpdate

在更新完成后，每帧调用一次LateUpdate。当LateUpdate 开始时，在Update 中执行的任何计算都将完成。LateUpdate的常见用途是跟随第三人称相机。如果您让角色在Update中移动和转动，您可以在 LateUpdate 中执行所有摄像机移动和旋转计算。这将确保角色在相机跟踪其位置之前已完全移动。

通常情况下，你不应该依赖于不同游戏对象调用相同事件函数的顺序——除非顺序是明确记录或可设置的。(如果你需要更细粒度的玩家循环控制，你可以使用PlayerLoop API。)

不能为同一个MonoBehaviour子类的不同实例指定事件函数的调用顺序。例如，一个MonoBehaviour的更新函数可能会在另一个游戏对象上的同一个MonoBehaviour的更新函数之前或之后被调用——包括它自己的父或子游戏对象。

你可以指定一个MonoBehaviour子类的事件函数应该在另一个子类的事件函数之前调用(使用项目设置窗口的脚本执行顺序面板)。例如，如果你有两个脚本，EngineBehaviour和SteeringBehaviour，你可以设置脚本执行顺序，使enginebehaviors总是在steeringbehaviors之前更新。

也就是说同一个脚本在不同实例上，然后那个实例的同一event function先调用是不确定的，如果是不同脚本可以在project setting中设置调用顺序。

### Rendering

1. OnGUI

每帧调用多次以响应 GUI 事件。首先处理 Layout 和 Repaint 事件，然后是每个输入事件的 Layout 和键盘/鼠标事件。

### Coroutines

#### yield

在下一帧调用了所有 Update 函数后，协程将继续。

#### yield WaitForSeconds

在为帧调用了所有更新函数之后，在指定的时间延迟后继续。

#### yield WaitForFixedUpdate

在所有脚本上调用了所有 FixedUpdate 后继续。如果协程在 FixedUpdate 之前产生，那么它会在当前帧的 FixedUpdate 之后恢复。

#### 4 yield WWW

在 WWW 下载完成后继续。 yield StartCoroutine 链接协程，并等待 MyFunc 协程首先完成。

#### 5.yield StartCoroutine

链接协程，并等待 MyFunc 协程首先完成。

#### 6.停止协程

SetActive为false可以停止，改为true不会继续上一次的执行，enable=false则不能停止协程。

### 当对象被销毁时

#### 1.OnDestroy

这个函数在对象存在的最后一帧的所有帧更新之后被调用(对象可能会在object的响应中被Object.Destroy或在一个场景结束时)。

### 退出

#### 1.OnApplicationQuit:

这个函数在应用程序退出之前调用所有游戏对象。在编辑器中，当用户停止播放模式时调用。

#### 2．OnDisable:

当行为变为禁用或不活动时调用该函数。

# 第二章 Time

1. Time.captureDeltatime: 减慢应用程序的播放时间，以允许 Unity 在帧之间保存屏幕截图。其实就是可以设置每一帧的间隔时间为常量值。
2. Time.deltaTime:上一帧到当前帧的时间
3. Time.fixedDeltaTime: 执行物理和其他固定帧速率更新（如 MonoBehaviour 的 FixedUpdate）的间隔（以秒为单位）。
4. Time.[fixedTime](https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Time-fixedTime.html): 自上次 FixedUpdate 开始以来的时间（只读）。这是游戏开始后的秒数。相当于运行FixedUpdate以来的时间。
5. Time.frameCount:游戏开始到现在经过的帧数
6. Time. [maximumDeltaTime](https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Time-maximumDeltaTime.html): Time.deltaTime 在任何给定帧中的最大值。这是一个以秒为单位的时间，它限制了两帧之间 Time.time 的增加。根据图2-1，限制Time.time就相当于限制FixedTime的执行次数
7. Time.realTimeSinceStartUp:如果您想将 Time.timeScale 设置为零以暂停应用程序，但仍希望能够以某种方式测量时间，则使用 realtimeSinceStartup 很有用。在编辑器脚本中，您还可以使用 realtimeSinceStartup 在编辑器暂停时测量时间。
8. Time.smoothDeltaTime: 平滑的 Time.deltaTime（只读）。
9. Time.time: 这是自应用程序启动以来的时间（以秒为单位），Time.timeScale 缩放和 Time.maximumDeltaTime 调整。当从 MonoBehaviour.FixedUpdate 内部调用时，它返回 Time.fixedTime。 此值在唤醒消息期间未定义，并在所有这些消息完成后开始。如果编辑器暂停，此值不会更新。有关不受暂停影响的时间值，请参见 Time.realtimeSinceStartup。
10. Time.timeScale: 时间流逝的速度。
11. Time.unscaledTime：游戏运行以来的时间，不受timeScale影响
12. Time. timeSinceLevelLoad：该帧开始以来的时间（只读）。此为自加载上一个关卡以来的时间（以秒为单位）。

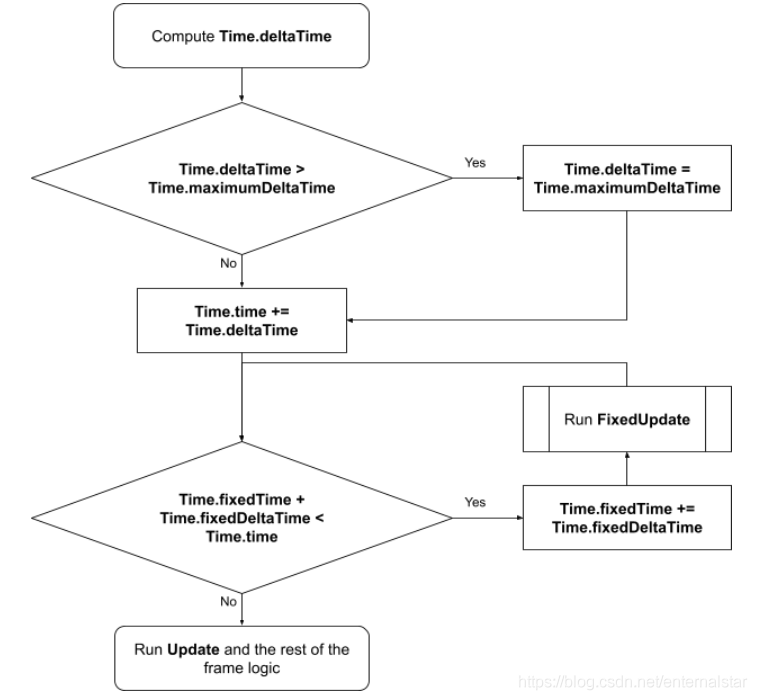


图2-1

【流程图说明】

* 1. 计算Time.deltaTime，其表示从上一帧开始到这一帧开始所用的时间，也即相邻两帧的时间间隔，单位为秒。其受Time.timeScale影响，默认值时1.0，可以通过修改其值来实现运动加速和减慢，设置为0可实现游戏暂停。
  2. 判断Time.deltaTime是否大于Time.maximumDeltaTime，其表示两帧间所允许的最大时间间隔。如果上一帧用时过长超过了这个值，那么帧间隔将被设置为这个值。这意味着上一帧可能并没有全部执行完，但下一帧就要开始执行了。而上一帧用时过长的原因最可能是物理帧更新用时长，maximumDeltaTime可以限制物理帧执行次数， maximumDeltaTime / fixedDeltaTime表示一个逻辑帧内物理帧的最大执行次数。该值一般设置为 1/10 秒到 1/3 秒之间。
  3. Time.time是通过Time.deltaTime累加得到的，其表示自游戏开始后所经历的时间，也表示当前帧在游戏开始后哪个时间开始执行的。Time.deltaTime是在Update中有效的，而Awake和Start在Update前执行，所以Time.time不包括游戏中所有Awake和Start执行的时间。
     1. 如果想知道自游戏开始后的真实物理时间，通过Time.realtimeSinceStartup获取，它不是一个常量，在同一帧内不同地方调用获取到的值是不一样的（deltaTime和time是一样的）。在大多数情况下，time可以代替realtimeSinceStartup。真实物理时间不受timeScale和编辑器暂停 影响，这个真实的物理时间从系统计时器获取，系统计时器受平台和硬件影响，有很小的可能出现这种情况：运行了多帧，但物理时间相同。
     2. 通过Time.frameCount可以获取当前帧是游戏开始以来的第几个帧，从0开始计数。
  4. Time.fixedTime类同Time.time，表示物理帧的间隔时间，用于FixedUpdate。Time.fixedDeltaTime是固定值，可以在Edit->ProjectSetting->time —>Fixedtimestep中设置。

1. timeScale影响deltatime，继而影响time，继而影响FixedUpdate的调用次数。而fixedDeltaTime是不受timeScale影响的，若改变了timeScale并想使得FixedUpdate调用次数不变，那么fixedDeltaTime就需要乘以timeScale。如果timeScale设置为0，则time不会增加，那么FixedUpdate就不会被调用。
2. time的增量即deltaTime和fixedDeltaTime会影响FixUpate的调用次数，每帧的deltaTime不确定，所以每帧FixedUpate的调用次数也不确定。
3. 执行Update，其在FixedUpate后执行。

参考：[(27条消息) Unity中的时间\_永恒星的博客-CSDN博客](https://blog.csdn.net/enternalstar/article/details/118242511)

# 第三章 GameObject类

#### 1.创建GameObject的三种方法

// 1第一种

GameObject go= new GameObject("cube");

// 2克隆一个已有的

GameObject.Instantiate(prefab);

// 3创建基本的物体

GameObject go =GameObject.CreatePrimitive(PrimitiveType.Cube);

#### 2.添加组件

// 添加刚体

go.AddComponent<Rigidbody>();

// 添加 脚本

go.AddComponent<API01Event>();  // API01Event 是脚本

#### activeInHierarchy：

定义 GameObject 在 Scene 中是否处于活动状态。

Debug.Log(go.activeInHierarchy); // True

go.SetActive(false);

Debug.Log(go.activeInHierarchy); // False

#### 5.Tag：此游戏对象的标签。

一般用来标识某个或者某些GameObject，比如MainCamera，Player，Enemy

#### 6.Layer

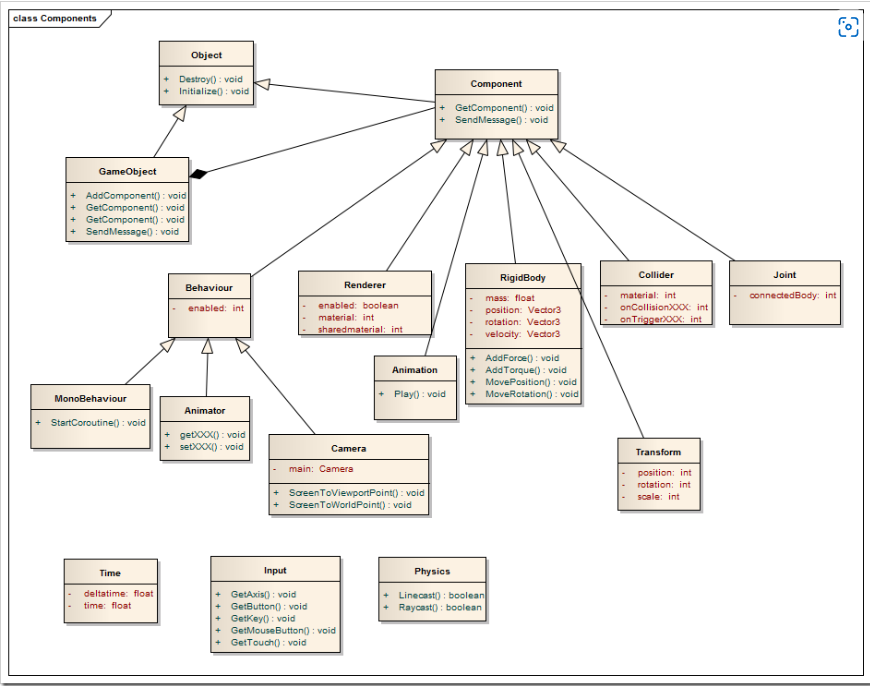
标记对象所在层，然后可以控制层的渲染。您可以使用图层从相机进行选择性渲染或忽略光线投射。 Unity 生成 32 层，用 0 到 31 的整数标记，并为自己的系统保留 0 到 5 层。您可以使用第 5 层及更高层。要添加或查看图层，请单击编辑器窗口右上角的布局按钮。

#### 7 name：对象的名称。

Debug.Log(go.name);

Debug.Log(go.GetComponent<Transform>().name)//获取组件的名字其实获取的是物体的名字

#### 8.scene：该 GameObject 所属的场景。



### Public function

**AddComponent : 将名为 className 的组件类添加到该游戏对象。**

go.AddComponent<Rigidbody>();

ComPareTag : 这个游戏对象是否带有 tag 标记？

SendMessage : 调用此游戏对象中的每个 MonoBehaviour 上名为 methodName 的方法。

**BroadcastMessage : 调用此游戏对象或其任何子项中的每个 MonoBehaviour 上名为 methodName 的方法。**

target.BroadcastMessage("Attack",null,SendMessageOptions.DontRequireReceiver);//如果有接受者则发送，如果没有不报错

void Attack() //当一个物体的脚本里拥有此方法则会被调用（子类也会调用）

{

    Debug.Log(this.gameObject + "正在攻击");

}

SendMessageUpwards : 在此游戏对象中的每个 MonoBehaviour 以及该行为的每个祖先上调用名为 methodName 的方法。

### Static function

CreatePrimitive : 创建一个具有原始网格渲染器和相应碰撞体的游戏对象.

Find:通过name查找gameobject，在awake或者是start中获取一次，不要频繁使用，具体看文档

FindGameObjectWithTag : 返回标记为 tag 的活动 GameObject 的列表。如果未找到 GameObject，则返回空数组。

FindWithTag：返回第一个带tag并且Active的GameObject，如果有多个，不保证每次返回同一个。

DontDestroyOnLoad：加载新场景时不销毁该Object

FindObjectsOfType：返回所有类型为 type 的已加载的激活对象的列表。

### Object.GetInstanceID

功能为：返回物体的实例id。 注意的是： 物体的实例id 需要确保始终是唯一的。

### MonoBehavior

继承层次：

* Object->Component->Behaviour->MonoBehaviour

MonoBehaviour是所有脚本的基类，使用javascript的话，每个脚本都会自动继承自MonoBehaviour，但使用C#或Boo就必须显式从MonoBehaviour继承。

注意： 在 Unity Editor上有一个用于禁用 MonoBehaviour 的勾选框。 取消勾选时将禁用以下某些（如果脚本中有的话）函数。如果脚本中不存在这些函数，则编辑器不会显示勾选框。这些函数包括：

* Start()
* Update()
* FixedUpdate()
* LateUpdate()
* OnGUI()
* OnDisable()
* OnEnable()

**Invoke : 在 time 秒后调用 methodName 方法。**

**InvokeRepeating : 在 time 秒后调用 methodName 方法，然后每 repeatRate 秒调用一次。**

**CancleInvoke : 取消该 MonoBehaviour 上的所有 Invoke 调用。**

# 第四章 coroutine

在使用协程的时候需要有以下几个关键词：

* + - 1. 函数返回值是IEnmerator
      2. 函数返回时要使用 yield return xxx
      3. 调用协程方法时 我们要使用StartCoroutine(xxx())

public GameObject cube;

private void Update()

{

if (Input.GetKeyDown(KeyCode.Space))

{

StartCoroutine(Fade());

}

}

IEnumerator Fade()

{

for (float i = 0; i <= 1; i += 0.1f)

{

// 第一种方式

//cube.GetComponent<MeshRenderer>().material.color = new Color(i,i,i,i);

// 第二种方式

Color color = cube.GetComponent<MeshRenderer>().material.color;

Color newColor = Color.Lerp(color, Color.black,0.1f); // 向黑色逐渐靠近

cube.GetComponent<MeshRenderer>().material.color = newColor;

yield return new WaitForSeconds(0.1f); // 暂停

}

}

**StopCoroutine : 停止在该行为上运行的第一个名为 methodName 的协同程序或存储在 routine 中的协同程序**

**三种方法：**

**coroutine = WaitAndPrint(3.0f);**

**StartCoroutine(coroutine);**

StopCoroutine(coroutine);

Coroutine b = StartCoroutine(coroutineB());

yield return new WaitForSeconds(2.0f);

Debug.Log("coroutineA() finished " + Time.time);

// B() was expected to run for 10 seconds

// but was shut down here after 3.0f

StopCoroutine(b);

# 第五章 鼠标函数函数

### 鼠标事件函数 OnMouseXX

**OnMouseDown : 当用户在 GUIElement 或 Collider 上按下鼠标按钮时，将调用 OnMouseDown。**

**OnMouseUp : 当用户松开鼠标按钮时，将调用 OnMouseUp。**

**OnMouseDrag : 当用户单击 GUIElement 或 Collider 并仍然按住鼠标时，将调用 OnMouseDrag。**

**OnMouseEnter : 当鼠标进入 GUIElement 或 Collider 时调用。**

**OnMouseExit : 当鼠标不再处于 GUIElement 或 Collider 上方时调用。**

**OnMouseOver : 当鼠标悬停在 GUIElement 或 Collider 上时，每帧调用一次。**

**OnMouseUpAsButton : 松开鼠标时，仅当鼠标在按下时所在的 GUIElement 或 Collider 上时，才调用 OnMouseUpAsButton。**

* 注意是否需要collider和isTrigger

# 第六章Mathf

### Mathf 工具类

#### Static Variables

1. Deg2Rad/Rad2Deg : 度到弧度换算常量（只读）/弧度到度换算常量（只读）。
2. PI : 有名的“3.14159265358979...”值（只读）。
3. Epsilon : 微小浮点值（只读）。
4. Infinity : 正无穷大的表示形式（只读）。
5. NegativeInfinity : 负无穷大的表示形式（只读）。

#### Static function

Clamp: 将给定值限制在给定的最小浮点数和最大浮点值之间。如果它在最小和最大范围内，则返回给定值。

DeltaAngle:计算以度为单位的两个给定角度之间的最短差。

Floor ： 返回小于或等于 f 的最大整数。

Ceil: 返回大于或等于 f 的最小整数。

Lerp:插值计算

PingPong ： 对值 t 进行 PingPong 操作，使它不会大于长度，并且不会小于 0。在 0 与 length 之间来回移动。

# 第七章Input 输入类

GetKey : 在用户按下 name 标识的键时返回 true。

GetKeyDown、GetKeyUp : 只会触发一次 按下 抬起

GetMouseButton ： 返回是否按下了给定的鼠标按钮。（0 左，1 右，2 中）

GetButton ： 当按住 buttonName 标识的虚拟按钮时，返回 true。

GetButtonDown GetButtonUp ： 在用户按下/抬起由 buttonName 标识的虚拟按钮的帧期间返回 true。

GetAxis ： 返回由 axisName 标识的虚拟轴的值。

anyKey ： 当前是否有任何键或鼠标按钮处于按下状态？（只读）

anyKeyDown : 在用户按任意键或鼠标按钮后的第一帧返回 true。（只读）

mousePosition : 鼠标当前的像素坐标位置。（只读）

# 第八章Vector相关

| **变量** | **解释** |
| --- | --- |
| **down** | 用于编写 Vector2(0, -1) 的简便方法。 |
| **left** | 用于编写 Vector2(-1, 0) 的简便方法。 |
| **one** | 用于编写 Vector2(1, 1) 的简便方法。 |
| **right** | 用于编写 Vector2(1, 0) 的简便方法。 |
| **up** | 用于编写 Vector2(0, 1) 的简便方法。 |
| **zero** | 用于编写 Vector2(0, 0) 的简便方法。 |
| **magnitude** | 返回该向量的长度。（只读） |
| **normalized** | 返回 magnitude 为 1 时的该向量。（只读） |
| **sqrMagnitude** | 返回该向量的平方长度。（只读） |
| **x** | 向量的 X 分量。 |
| **y** | 向量的 Y 分量。 |

void Start()

{

    Vector2 a = new Vector2(2, 2);

    Vector2 b = new Vector2(3, 4);

    Vector2 c = new Vector2(3, 0);

    print(Vector2.Angle(a, b))  // 8.130115 返回 from 与 to 之间的无符号角度（以度为单位）。

    print(Vector2.Angle(a, c)); // 45

    print(Vector2.ClampMagnitude(c, 2)); // 2.0 0.0  返回 vector 的副本，其大小被限制为 /maxLength/。

    print(Vector2.Distance(b, c)); //4  返回 a 与 b 之间的距离。

    print(Vector2.Lerp(a, b, 0.5f));//2.5 3     在向量 a 与 b 之间按 t 进行线性插值。

    print(Vector2.LerpUnclamped(a, b, 0.5f));//2.5 3   在向量 a 与 b 之间按 t 进行线性插值。

    print(Vector2.Lerp(a, b, 2f));//b 3,4

    print(Vector2.LerpUnclamped(a, b, 2f)); // 4.0 6.0

    print(Vector2.Max(a, b)); // 3.0 4.0  返回由两个向量的最大分量组成的向量。

    print(Vector2.Min(a, b)); //2.0 2.0

}

加减乘除 运算符重载 不再过多阐述  
二维和三维 大体相等 ，不再过多阐述

# 第九章Random

### Random 随机数

1. Random.Range : 返回介于 min [含] 与 max [含] 之间的随机浮点数（只读）。
2. InitState : 使用种子初始化随机数生成器状态。
3. ColorHSV : 通过 HSV 和 Alpha 范围生成随机颜色。
4. insideUnitCircle ： 返回半径为 1 的圆形内的随机点（只读）。
5. insideUnitSphere ： 返回半径为 1 的球体内的随机点（只读）。

# 第十章 Quaternion（四元数）

eularAngles：返回或设置旋转的欧拉角表示。

Euler:把欧拉角转换为四元数

LookRotation:会使物体旋转到目标方向（望向目标）

void Start()

{

    Vector3 dir = enemy.position - player.position;

    dir.y = 0; // 忽略高度差

    player.rotation = Quaternion.LookRotation(dir);

}

// 更加平滑

Quaternion target = Quaternion.LookRotation(dir);

player.rotation = Quaternion.Slerp(player.rotation, target, Time.deltaTime);

# 第十一章 Rigidbody、camera

### Rigidbody

1. postion : 刚体的位置。
2. MovePosition : 将刚体移动到 /position/。

|  |  |
| --- | --- |
| 1  2  3  4 | **void** Update()  {      playerRgd.MovePosition(playerRgd.position + Vector3.forward \* Time.deltaTime);  } |

1. rotation : 刚体的旋转。
2. MoveRotation : 将刚体旋转到 /rotation/。
3. AddForce : 向 Rigidbody 添加力。

|  |  |
| --- | --- |
| 1  2  3  4 | **void** Update()  {      playerRgd.AddForce(Vector3.forward\*force);  } |

在刚体上操作比在transform上性能要好

### Camera

1. main:直接获取mainCamera，tag必须为MainCamera

|  |  |
| --- | --- |
| 1 | Camera mainCamera = Camera.main |

1. ScreenPointToRay : 返回从摄像机通过屏幕点的光线。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | mainCamera =  Camera.main;  Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);  RaycastHit hit;  bool isColl = Physics.Raycast(ray,out hit);  **if** (isColl)  {    Debug.Log(hit.collider);  } |

### Application

1. dataPath: 数据路径
2. Streaming Assets: 资源文件(创建后不会被打包）
3. identifier: 包的名字
4. isFocused: 判断是否是焦点
5. isMobile： 是否在移动平台
6. isPlaying: 编辑模式下载运行会返回ture

|  |  |
| --- | --- |
| 1  2 | //在编辑器模式下退出游戏  UnityEditor.EditorApplication.isPlaying = **false**; |

1. Plaatform: 可以控制在某个平台下运行
2. identifier:标识名
3. companyName: 公司名
4. prodctName: 游戏名字
5. runlnBackGround: 是否在后台运行
6. unityVersion: 判断unity的版本
7. OpenURL: 可以打开页面

### sceneManager

1. LoadScene : 按照 Build Settings 中的名称或索引加载场景。

|  |  |
| --- | --- |
| 1 | SceneManager.LoadScene(sceneName); |

1. activeSceneChanged : 订阅此事件可在活动场景发生变化时收到通知。
2. sceneLoaded : 向此事件添加委托，以在加载场景时收到通知。
3. LoadSceneAsync : 在后台异步加载场景。
4. sceneCount : 当前加载的场景总数。
5. GetSceneAt : 获取 SceneManager 的已加载场景列表中索引处的场景。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | **public** **class** API17Scenes : MonoBehaviour  {      // Start is called before the first frame update  **void** Start()      {          print(SceneManager.sceneCount);          print(SceneManager.sceneCount);          print(SceneManager.sceneCountInBuildSettings);            print(SceneManager.GetActiveScene().name);          print(SceneManager.GetSceneAt(0).name);            SceneManager.activeSceneChanged += OnActiveSceneChanged;          SceneManager.sceneLoaded += OnSceneLoaded;      }  **void** OnActiveSceneChanged(Scene a,Scene b)      {          print(a.name);          print(b.name);      }  **void** OnSceneLoaded(Scene a, LoadSceneMode mode)      {          print(a.name + "" + mode);      }        // Update is called once per frame  **void** Update()      {    **if** (Input.GetKeyDown(KeyCode.Space))          {             // print(SceneManager.GetSceneByName("MenuScene").buildIndex );              SceneManager.LoadScene("MenuScene");          }      }  } |

### Physics

1. Raycast ： 向场景中的所有碰撞体投射一条射线，该射线起点为 /origin/，朝向 /direction/，长度为 /maxDistance/。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // 接受一个射线  Ray ray = **new** Ray(transform.position+transform.forward,transform.forward);  bool isCollider = Physics.Raycast(ray);  Debug.Log(isCollider);  **void** Update()  {      Ray ray = **new** Ray(transform.position+transform.forward,transform.forward);      //bool isCollider = Physics.Raycast(ray);      // 射线的距离      //bool isCollider = Physics.Raycast(ray, 1);      RaycastHit hit;      //bool isCollider = Physics.Raycast(ray, out hit);      //Debug.Log(hit.point);  // 获取捕获的点      //Debug.Log(hit.collider);   // 碰撞的游戏物体      //      // 碰撞 layer 为xx的层      // bool isCollider = Physics.Raycast(ray, Mathf.Infinity,LayerMask.GetMask("xx"));  }  //2D  Physics2D.Raycast(); |

### UGUI 事件监听

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | ```c#  **public** GameObject btnGameObject;    // 按钮  **public** GameObject sliderGameObject; // 滑动  **public** GameObject dropDownGameObject;    //下拉  **public** GameObject toggleGameObject;    // 勾选        // Use this for initialization  **void** Start () {          btnGameObject.GetComponent<Button>().onClick.AddListener(**this**.ButtonOnClick);          sliderGameObject.GetComponent<Slider>().onValueChanged.AddListener(**this**.OnSliderChanged);          dropDownGameObject.GetComponent<Dropdown>().onValueChanged.AddListener(**this**.OnDropDownChanged);          toggleGameObject.GetComponent<Toggle>().onValueChanged.AddListener(**this**.OnToggleChanged);      }    **void** ButtonOnClick()      {          Debug.Log("ButtonOnClick");      }  **void** OnSliderChanged(**float** value)      {          Debug.Log("SliderChanged:" + value);      }  **void** OnDropDownChanged(Int32 value)      {          Debug.Log("DropDownChanged:" + value);      }  **void** OnToggleChanged(bool value)      {          Debug.Log("ToggleChanged:" + value);      }  ```    ```c#      // 实现接口 UI,EventSystem       //interface  **public** **class** UIEventManager2 : MonoBehaviour//, IPointerDownHandler,IPointerClickHandler,IPointerUpHandler,IPointerEnterHandler,IPointerExitHandler          ,IBeginDragHandler,IDragHandler,IEndDragHandler,IDropHandler      {  **public** **void** OnBeginDrag(PointerEventData eventData)          {              Debug.Log("OnBeginDrag");          }    **public** **void** OnDrag(PointerEventData eventData)          {              Debug.Log("OnDrag");          }    **public** **void** OnDrop(PointerEventData eventData)          {              Debug.Log("OnDrop");          }    **public** **void** OnEndDrag(PointerEventData eventData)          {              Debug.Log("OnEndDrag");          }    **public** **void** OnPointerClick(PointerEventData eventData)          {              Debug.Log("OnPointerClick");          }    **public** **void** OnPointerDown(PointerEventData eventData)          {              Debug.Log("OnPointerDown");          }    **public** **void** OnPointerEnter(PointerEventData eventData)          {              Debug.Log("OnPointerEnter");          }    **public** **void** OnPointerExit(PointerEventData eventData)          {              Debug.Log("OnPointerExit");          }    **public** **void** OnPointerUp(PointerEventData eventData)          {              Debug.Log("OnPointerUp");          }      }    ``` |

### 其他

1. Camera  
   ScreenPointToRay : 返回从摄像机通过屏幕点的光线。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **private** Camera camera;  // Use this for initialization  **void** Start () {      camera = Camera.main;  }  // Update is called once per frame  **void** Update () {      Ray ray = camera.ScreenPointToRay(Input.mousePosition);      //Debug.DrawRay(ray.origin, ray.direction);      //Debug.DrawLine(ray.origin, ray.origin + ray.direction \* 100);      RaycastHit hit;      bool isCollider = Physics.Raycast(ray, out hit);      Debug.Log(hit.collider);  } |

1. CharacterController : 可使您轻松进行受碰撞约束的移动，同时不必处理刚体。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **public** **float** speed = 3;  **private** CharacterController cc;  // Use this for initialization  **void** Start () {      cc = GetComponent<CharacterController>();  }  // Update is called once per frame  **void** Update () {  **float** h = Input.GetAxis("Horizontal");  **float** v = Input.GetAxis("Vertical");      cc.SimpleMove(**new** Vector3(h, 0, v) \* speed);      //cc.Move(new Vector3(h, 0, v) \* speed \* Time.deltaTime);      Debug.Log(cc.isGrounded);  }  **private** **void** OnControllerColliderHit(ControllerColliderHit hit) // 检测碰撞  {      Debug.Log(hit.collider);  } |

1. Mesh : 材质

[复制代码](https://blog.nowcoder.net/n/88e3c1296e6d4ac497a58c120cab4724)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **public** Mesh mesh;  **private** Material mat;  // Use this for initialization  **void** Start () {      //GetComponent<MeshFilter>().sharedMesh = mesh;      //Debug.Log(GetComponent<MeshFilter>().mesh == mesh);      mat = GetComponent<MeshRenderer>().material;      // 获取材质  }  // Update is called once per frame  **void** Update () {      mat.color = Color.Lerp(mat.color, Color.red, Time.deltaTime); // 更新颜色  } |