一、 脚本概览

这是一个关于 Unity 内部脚本如何工作的简单概览。

Unity 内部的脚本，是通过附加自定义脚本对象到游戏物体组成的。在脚本对象内部不同志的函数被特定的事件调用。最常用的列在下面：

Update：

这个函数在渲染一帧之前被调用，这里是大部分游戏行为代码被执行的地方，除了物理代码。

FixedUpdate：

这个函数在每个物理时间步被调用一次，这是处理基于物理游戏的地方。

在任何函数之外的代码：

在任何函数之外的代码在物体被加载的时候运行，这个可以用来初始化脚本状态。

注意：文档的这个部份假设你是用 Javascript，参考用 C#编写获取如何使用 C#和 Boo 编写脚本的信息。

你也能定义事件句柄，它们的名称都以 On 开始，（例如 OnCollisionEnter），为了查看完整的预定义事件的列表，参考 MonoBehaviour 文档。

概览：常用操作

大多数游戏物体的操作是通过游戏物体的 Transform 或 Rigidbody 来做的，在行为脚本内部它们可以分别通过 transform 和 rigidbody 访问，因此如果你想绕着 Y 轴每帧旋转 5 度，你可以如下写：

```
function Update（）{
transform.Rotate(0,5,0);
}
```

如果你想向前移动一个物体，你应该如下写：

```
function Update（）{
transform.Translate(0,0,2);
}
```

概览：跟踪时间

Time 类包含了一个非常重要的类变量，称为 deltaTime，这个变量包含从上一次调用 Update 或 FixedUpdate（根据你是在 Update 函数还是在 FixedUpdate 函数中）到现在的时间量。

所以对于上面的例子，修改它使这个物体以一个恒定的速度旋转而不依赖于帧率：

```
function Update（）{
transform.Rotate(0,5*Time.deltaTime,0);
}
```

移动物体：

```
function Update（）{
transform. Translate (0, ,0,2*Time.deltaTime);
}
```

如果你加或是减一个每帧改变的值，你应该将它与 Time.deltaTime 相乘。当你乘以 Time.deltaTime 时，你实际的表达：我想以 10 米/秒移动这个物体不是 10 米/帧。这不仅仅是因为你的游戏将独立于帧而运行，同时也是因为运动的单位容易理解。(米/秒)

另一个例子，如果你想随着时间增加光照的范围。下面的表达式，以 2 单位/秒改变半径。

```
function Update (){
light.range += 2.0 * Time.deltaTime;
}
```

当通过力处理刚体的时候，你通常不必用 Time.deltaTime，因为引擎已经为你考虑到了这一点。

概览：访问其他组件

组件被附加到游戏物体，附加 Renderer 到游戏物体使它在场景中渲染，附加一个 Camera 使它变为相机物体，所有的脚本都是组件，因为它们能被附加到游戏物体。

最常用的组件可以作为简单成员变量访问：

Component	可如下访问
Transform	transform
Rigidbody	rigidbody
Renderer	renderer
Camera	camera (only on camera objects)
Light	light (only on light objects)
Animation	animation
Collider	collider

…等等。

对于完整的预定义成员变量的列表。查看 Component，Behaviour 和 MonnoBehaviour 类文档。如果游戏物体没有你想取的相同类型的组件，上面的变量将被设置为 null。

任何附加到一个游戏物体的组件或脚本都可以通过 GetComponent 访问。

```
transform.Translate（0,3,0）;
//等同于
GetComponent(Transform).Translate(0, 1, 0);
```

注意 transfom 和 Transform 之间大小写的区别，前者是变量（小写），后者是类或脚本名称（大写）。大小写不同使你能够从类和脚本名中区分变量。

应用我们所学，你可以使用 GetComponent 找到任何附加在同一游戏物体上的脚本和组件，请注意要使用下面的例子能够工作，你需要有一个名为 OtherScript 的脚本，其中包含一个 DoSomething 函数。OtherScript 脚本必须与下面的脚本附加到相同的物体上。

```
//这个在同一游戏物体桑找到名为 OtherScript 的脚本
//并调用它上加的 DoSomething
function Update(){
otherScript = GetComponent(OtherScript);
otherScript.DoSomething();
}
```

概览：访问其它游戏物体

大多数高级的代码不仅需要操作一个物体，Unity 脚本接口有各种方法来找到并访问其他游戏物体和组件。在下面，我们假定有个一名为 OtherScript,js 的脚本附加到场景的游戏物体上。

```
var foo = 5;
function DoSomething ( param : String) {
print(param + " with foo: " + foo);
}
```

1.通过检视面板赋值引用

你可以通过检视面板赋值变量到任何物体

```
//变换拖动到 target 的物体
var target : Transform;
function Update ()
{
target.Translate(0, 1, 0);
}
```

你也可以在检视面板中公开到其他物体的引用，下面你可以拖动一个包含的游戏物体到检视面板中的 target 槽。

```
//设置在检视面板中赋值的 target 变量上的 foo，调用 DoSomething
var target : OtherScript;
function Update ()
{
//设置 target 物体的 foo 变量
target.foo = 2;
//  调用 target 上的 DoSomething
target.DoSomething("Hello");
}
```

2.通过物体层次定位

对于一个已经存在的物体，可以通过游戏物体的 Transform 组件来找到子和父物体；

```
//找到脚本所附加的
//游戏物体的子“Hand”
transform.Find("Hand").Translate(0, 1, 0);
```

一旦在层次视图中找到这个变换，你可以使用 GetComponent 来获取其他脚本，

```
//找到名为“Hand”的子
//在附加到它上面的 OtherScript 中，设置 foo 为 2；
transform.Find("Hand").Translate(0, 1, 0);
//找到名为“Hand”的子
//然后应用一个力到附加在 hand 上的刚体
transform.Find("Hand").GetComponent(OtherScript).DoSomething("Hello");
//  找到名为“Hand”的了
//  然后应用一个力到附加在 hand 上的刚体
transform.Find("Hand").rigidbody.AddForce(0, 10, 0);
```

你可以循环所有的子，

```
//变换的所有子向上移动 10 个单位
for (var child : Transform in transform)
{
child.Translate(0, 1, 0);
}
```

参考 Transform 类文档获取更多信息。

3.根据名称或标签定位.

你可以使用 GameObject.FindWithTag 和 GameObject.FindGameObjectsWithTag 搜索具有特定标签的游戏物体，使用 GameObject.Find 根据名称查找物体。

```
function Start ()
```

```
{
// 按照名称
var go = GameObject.Find("SomeGuy");
go.transform.Translate(0, 1, 0);
// 按照标签

var player = GameObject.FindWithTag("Player");
player.transform.Translate(0, 1, 0);
}
```

你可以在结果上使用 GetComponent，在找到的游戏物体上得到任何脚本或组件。

```
function Start ()
{
// 按名称
var go = GameObject.Find("SomeGuy");
go.GetComponent(OtherScript).DoSomething();

// 按标签
var player = GameObject.FindWithTag("Player");
player.GetComponent(OtherScript).DoSomething();
}
```

一些特殊的物体有快捷方式，如主相机使用 Camera.main。

4.作为参数传递

一些事件消息在事件包含详细信息。例如，触发器事件传递碰撞物体的 Collider 组件到处理函数。

OnTriggerStay 给我们一个到碰撞器的引用。从这个碰撞器我们可以获取附加到其上的刚体。

```
function OnTriggerStay( other : Collider ) {
// 如果另一个碰撞器也有一个刚体
// 应用一个力到它上面
if (other.rigidbody) {
other.rigidbody.AddForce(0, 2, 0);
}
}
```

或者我们可以通过碰撞器获取附加在同一个物体上的任何组件。

```
function OnTriggerStay( other : Collider ) {
// 如果另一个碰撞器附加了 OtherScript
// 调用它上面的 DoSomething
// 大多数时候碰撞器不会附加脚本
// 所以我们需要首先检查以避免 null 引用异常
if (other.GetComponent(OtherScript)) {
other.GetComponent(OtherScript).DoSomething();
}
}
```

注意通过上述例子中的 other 变量，你可以访问碰撞物体中的任何组件。

5.一种类型的所有脚本

使用 Object.FindObjectsOfType 找到所有具有相同类或脚本名称的物体，或者使用 Object.FindObjectOfType.找到这个类型的第一个物体。

```
function Start ()
{
// 找到场景中附加了 OtherScript 的任意一个游戏物体
var other : OtherScript = FindObjectOfType(OtherScript);
other.DoSomething();
}
```

概览：向量

Unity 使用 Vector3 类同一表示全体 3D 向量，3D 向量的不同组件可以通过想 x，y 和 z 成员变量访问。

```
var aPosition : Vector3;
aPosition.x = 1;
aPosition.y = 1;
aPosition.z = 1;
```

你也能够使用 Vector3 构造函数来同时初始化所有组件。

```
var aPosition = Vector3(1, 1, 1);
```

Vector3 也定义了一些常用的变量值。

```
var direction = Vector3.up; // 与 Vector3(0, 1, 0);相同
```

单个向量上的操作可以使用下面的方式访问：

```
someVector.Normalize();
```

使用多个向量的操作可以使用 Vector3 类的数；

```
theDistance = Vector3.Distance(oneVector, otherVector);
```

（注意你必须在函数名之前写 Vector3 来告诉 JavaScript 在哪里找到这个函数，这适用于所有类函数）

你也可以使用普通数学操作来操纵向量。

```
combined = vector1 + vector2;
```

查看 Vector3 类文档获取完整操纵和可用属性的列表。

概览：成员变量 & 全局变量变量

定义在任何函数之外的变量是一个成员变量。在 Unity 中这个变量可以通过检视面板来访问，任何保存在成员变量中的值也可以自动随工程保存。

```
var memberVariable = 0.0;
```

上面的变量将在检视面板中显示为名为"Member Variable"的数值属性。

如果你设置变量的类型为一个组件类型（例如 Transform, Rigidbody, Collider，任何脚本名称，等等）然后你可以在检视面板中通过拖动一个游戏物体来设置它们。

```
var enemy : Transform;
function Update()
{
if ( Vector3.Distance( enemy.position, transform.position ) < 10 );
print("I sense the enemy is near!");
}
}
```

你也可以创建私有成员变量。私有成员变量可以用来存储那些在该脚本之外不可见的

状态。私有成员变量不会被保存到磁盘并且在检视面板中不能编辑。当它被设置为调试模式时，它们在检视面板中可见。这允许你就像一个实时更新的调试器一样使用私有变量。

```
private var lastCollider : Collider;
function OnCollisionEnter( collisionInfo : Collision ) {
lastCollider = collisionInfo.other;
}
```

全局变量

你也可以使用 static 关键字创建全局变量

这创造了一个全局变量，名为 someGlobal

```
// 'TheScriptName.js'中的一个静态变量
static var someGlobal = 5;
// 你可以在脚本内部像普通变量一样访问它
print(someGlobal);
someGlobal = 1;
```

为了从另一个脚本访问它，你需要使用这个脚本的名称加上一个点和全局变量名。

```
print(TheScriptName.someGlobal);
TheScriptName.someGlobal = 10;
```

概览：实例化

实例化，复制一个物体。包含所有附加的脚本和整个层次。它以你期望的方式保持引用。到外部物体引用的克隆层次将保持完好，在克隆层次上到物体的引用映射到克隆物体。

实例化是难以置信的快和非常有用的。因为最大化地使用它是必要的。

例如， 这里是一个小的脚本，当附加到一个带有碰撞器的刚体上时将销毁它自己并实例化一个爆炸物体。

```
var explosion : Transform;
// 当碰撞发生时销毁我们自己
// 并生成给一个爆炸预设
function OnCollisionEnter (){
Destroy (gameObject);
var    theClonedExplosion : Transform;
theClonedExplosion = Instantiate(explosion, transform.position, transform.rotation);
}
```

实例化通常与预设一起使用

概览：Coroutines & Yield

在编写游戏代码的时候，常常需要处理一系列事件。这可能导致像下面的代码。

```
private var state = 0;
function Update()
{
if (state == 0) {
// 做步骤 0
state = 1;
return;
}
if (state == 1) {
// 做步骤 1
```

```
state = 2;
return;
}
// …
}
```

更方便的是使用 yield 语句。yield 语句是一个特殊类型的返回，这个确保在下次调用时该函数继续从该 yield 语句之后执行。

```
while(true) {
//  做步骤 0
yield; //等待一帧
//  做步骤 1
yield; //等待一帧
// ...
}
```

你也可以传递特定值给 yield 语句来延迟 Update 函数的执行，直到一个特定的事件发生。

```
//  做一些事情
yield WaitForSeconds(5.0); //等待 5 秒
//做更多事情…
```

可以叠加和连接 coroutines。

这个例子执行 Do，在调用之后立即继续。

```
Do ();
print ("This is printed immediately");
function Do ()
{
print("Do now");
yield WaitForSeconds (2);
print("Do 2 seconds later");
}
```

这个例子将执行 Do 并等待直到它完成，才继续执行自己。

```
//链接 coroutine
yield StartCoroutine("Do");
print("Also after 2 seconds");
print ("This is after the Do coroutine has finished execution");
function Do ()
{
print("Do now");
yield WaitForSeconds (2);
print("Do 2 seconds later");
}
```

任何事件处理句柄都可以是一个 coroutine

注意你不能在 Update 或 FixedUpdate 内使用 yield，但是你可以使用 StartCoroutine 来开始一个函数。

参 考 YieldInstruction, WaitForSeconds, WaitForFixedUpdate, Coroutine and

MonoBehaviour.StartCoroutine 获取更多使用 yield 的信息。

概览：用 C#编写脚本

除了语法，使用 C#或者 Boo 编写脚本还有一些不同。最需要注意的是：

1.从 MonoBehaviour 继承

所有的行为脚本必须从 MonoBehaviour 继承（直接或间接）。在 Javascript 中这自动完成，但是必须在 C#或 Boo 脚本中显示申明。如果你在 Unity 内部使用 Asset -> Create -> C Sharp/Boo Script 菜单创建脚本，创建模板已经包含了必需的定义。

```
public class NewBehaviourScript : MonoBehaviour {...} // C#
class NewBehaviourScript (MonoBehaviour): ... # Boo
```

2.使用 Awake 或 Start 函数来初始化

Javascript 中放置在函数之外的代码，在 C#或 Boo 中要放置在 Awake 或 Start 中。

Awake 和 Start 的不同是 Awake 在场景被加载时候运行，而 Start 在第一次调用 Update 或 FixedUpdate 函数之前被调用，所有 Awake 函数在任何 Start 函数调用之前被调用。

3.类名必须与文件名相同

Javascript 中，类名被隐式地设置为脚本的文件名（不包含文件扩展名）。在 c#和 Boo 中必须手工做。

4.在 C#中 Coroutines 有不同语法。

Coroutines 必有一个 IEnumerator 返回类型，并且 yield 使用 yield return… 而不是 yield…

```
using System.Collections;
using UnityEngine;
public class NewBehaviourScript : MonoBehaviour {
// C# coroutine
IEnumerator SomeCoroutine ()
{
// 等一帧
yield return 0;
//等两秒
yield return new WaitForSeconds (2);
}
}
```

5．不要使用命名空间

目前 Unity 还不支持将代码放置在一个命名空间中，这个需要将会出在未来的版本中。

6.只有序列化的成员变量会显示在检视面板中

私有和保护成员变量只在专家模式中显示，属性不被序列化或显示在检视面板中。

7．避免使用构造函数

不要在构造函数中初始化任何变量，使用 Awake 或 Start 实现这个目的。即使是在编辑模式中 Unity 也自动调用构造函数，这通常发生在一个脚本被编译之后，因为需要调用构造函数来取向一个脚本的默认值。构造函数不仅会在无法预料的时刻被调用，它也会为预设或未激活的游戏物体调用。

单件模式使用构造函数可能会导致严重的后果，带来类似随机 null 引用异常。

因此如果你想实现，如，一个单件模式，不要使用构造函数，而是使用 Awake。其实上，没有理由一定要在继续自 MononBehaviour 类的构造函数中写任何代码。

概览：最重要的类

Javascript 中可访问的全局函数或 C#中的基类

移动/旋转物体

动画系统

刚体

FPS 或第二人称角色控制器

概览：性能优化

1．使用静态类型

在使用 Javascript 时最重要的优化是使用静态类型而不是动态类型，Unity 使用一种叫做类型推理的技术来自自动转换 Javascript 为静态类型编码而不需要你做任何工作。

```
var foo=5；
```

在上面的例子里 foo 会自动被推断为一个整型值。因此，Unity 可能使用大量的编译时间来优化。而不使用耗时的动态名称变量查找等。这就是为什么 Unity 比其他在 JavaScript 的实现平均快 20 倍的原因之一。

唯一的问题是，有时并非一切都可以做类型推断。Unity 将会为这些变量重新使用动态类型。通过回到动态类型，编写 JavaScript 代码很简单。但是这也使得代码运行速度较慢。

让我们看一些例子：

```
function Start ()
{
var foo = GetComponent(MyScript);
foo.DoSomething();
}
```

这里 foo 将是动态类型，因此调用 DoSomething 函数将使用较长时间，因为 foo 的类型是未知的，它必须找出它是否支持 DoSomething 函数，如果支持，调用它。

```
function Start ()
{
var foo : MyScript = GetComponent(MyScript);
foo.DoSomething();
}
```

这里我们强制 foo 为指定类型，你将获得更好的性能。

2．使用#pragma strict

当然现在问题是，你通常没有意识到你在使用动态类型。#pragma strict 解决了这个！简单的在脚本顶部添加#pragma strict。然后，unity 将在脚本中禁用动态类型，强制使用静态类型，如果一个类型未知。Unity 将报告编译错误。那么在这种情况下 foo 将在编译时产生一个错误：

```
#pragma strict
function Start ()
{
var foo = GetComponent(MyScript);
foo.DoSomething();
}
```

3.缓存组件查找

另一个优化是组件缓存。不幸的是该优化需要一点编码，并且不一定是值得的，但是如

果你的脚本是真的用了很长时间了，你需要把最后一点表现出来，这是一个很好的优化。

当你访问一个组件通过 GetComponent 或访问变量，Unity 会通过游戏对象找到正确的组件。这一次可以很容易地通过缓存保存在一个私有变量里引用该组件。

简单地把这个：

```
function Update ()
{
transform.Translate(0, 0, 5);
}
```

变成：

```
private var myTransform : Transform;
function Awake ()
{
myTransform = transform;
}
function Update ()
{
myTransform.Translate(0, 0, 5);
}
```

后者的代码将运行快得多，因为 Unity 没有找到变换在每一帧游戏组件中的对象。这同样适用于脚本组件，在你使用 GetComponent 代替变换或者其它的东西。

4.使用内置数组

内置数组的速度非常快，所以请使用它们。

而整列或者数组类更容易使用，因为你可以很容易地添加元素，他们几乎没有相同的速度。内置数组有一个固定的尺寸，但大多数时候你事先知道了最大的大小在可以只填写了以后。关于内置数组最好的事情是，他们直接嵌入在一个结构紧凑的缓冲区的数据类型没有任何额外的类型信息或其他开销。因此，遍历是非常容易的，作为一切缓存在内存中的线性关系。

```
private var positions : Vector3[];
function Awake ()
{
positions = new Vector3[100];
for (var i=0;i<100;i++)
positions[i] = Vector3.zero;
}
```

5.如果你不需要就不要调用函数

最简单的和所有优化最好的是少工作量的执行。例如，当一个敌人很远最完美的时间就是敌人入睡时可以接受。直到玩家靠近时什么都没有做。这是种缓慢的处理方式的情况：

```
function Update ()
{
// 早期进行如果玩家实在是太遥远。
if (Vector3.Distance(transform.position, target.position) > 100)
return;
perform real work work...
}
```

这不是一个好主意，因为 Unity 必须调用更新功能，而你正在执行工作的每一个帧。一

个比较好的解决办法是禁用行为直到玩家靠近。有 3 种方法来做到这一点：

1.使用 OnBecameVisible 和 OnBecameInvisible。这些回调都是绑到渲染系统的。只要任何相机可以看到物体，OnBecameVisible 将被调用，当没有相机看到任何一个，OnBecameInvisible 将被调用。这种方法在很多情况下非常有用，但通常在 AI 中并不是特别有用，因为只要你把相机离开他们敌人将不

可用。

```
function OnBecameVisible () {
enabled = true;
}
function OnBecameInvisible ()
{
enabled = false;
}
```

2.使用触发器。一个简单的球形触发器会工作的非常好。一旦你离开这个影响球你将得到 OnTriggerEnter/Exit 调用。

```
function OnTriggerEnter (c : Collider)
{
if (c.CompareTag("Player"))
enabled = true;
}
function OnTriggerExit (c : Collider)
{
if (c.CompareTag("Player"))
enabled = false;
}
```

3.使用协同程序。Update 调用的问题是它们每帧中都发生。很可能会只需要每 5 秒检检查一次到玩家的距离。这应该会节省大量的处理周期。

概览：脚本编译（高级）

Unity 编译所有的脚本为.NET dll 文件，.dll 将在运行时编译执行。

这允许脚本以惊人的速度执行。这比传统的 javascript 快约 20 倍。比原始的 C++代码慢大约 50%。在保存的时候，Unity 将花费一点时间来编译所有脚本，如果 Unity 还在编译。你可以在 Unity 主窗口的右下角看到一个小的旋转进度图标。

脚本编译在 4 个步骤中执行：

1.所有在"Standard Assets", "Pro Standard Assets" 或 "Plugins"的脚本被首先编译。

在这些文件夹之内的脚本不能直接访问这些文件夹之外脚本。

不能直接引用或它的 变量，但是可以使用 GameObject.SentMessage 与它们通信。

2.所有在"Standard Assets/Editor", "Pro Standard Assets/Editor" 或 "Plugins/Editor"的脚本被首先编译。

如果你想使用 UnityEditor 命名空间你必须放置你的脚本在这些文件夹中，例如添加菜单项或自定义的向导，你都需要放置脚本到这些文件夹。

这些脚本可以访问前一组中的脚本。

3.然后所有在"Editor"中的脚本被编译。

如果你想使用 UnityEditor 命名空间你必须放置你的脚本在这些文件夹中。例如添加菜单单项或自定义的向导，你都需要放置脚本到这些文件夹。

这些脚本可以访问所有前面组中的脚本，然而它们不能访问后面组中的脚本。

这可能会是一个问题，当编写编辑器代码编辑那些在后面组中的脚本时。有两个解决方法：1、移动其他脚本到"Plugins"文件夹 2、利用 JavaScript 的动态类型，在 javascript 中你不需要知道类的类型。在使用 GetComponent 时你可以使用字符串而不是类型。你也可以使用 SendMessage，它使用一个字符串。

4.所有其他的脚本被最后编译

所有那些没有在上面文件夹中的脚本被最后编译。

所有在这里编译的脚本可以访问第一个组中的所有脚本（"Standard Assets"，"Pro Standard Assets" or "Plugins"）。这允许你让不同的脚本语言互操作。例如，如果你想创建一个 JavaScript。它使用一个 C#脚本;放置 C#脚本到"Standard Assets"文件夹并且 JavaScript 放置在"Standard Assets"文件夹之外。现在 JavaScript 可以直接引用 c#脚本。

放置在第一个组中的脚本，将需要较长的编译时间，因为当他们被编译后，第三组需要被重新编译。因此如果你想减少编译时间，移动那些不常改变 的到第一组。经常改变的到第四组。

二、 运行时类

AnimationCurve

类

动画曲线，在给定的时间添加关键帧并确定曲线。

变量

- var keys : Keyframe[]

描述：定义在动画曲线中的所有键。这让你从数组中清理，添加或移除键。

如果键没有按照时间顺序，它们会在赋值的时候自动排序。

- var length : int

描述：曲线中键的数量（只读）。

- var preWrapMode : WrapMode

描述：第一帧之前动画的行为。

- var this[index : int] : Keyframe

描述：取向索引为 index 的键（只读）。

构造函数

- static function AnimationCurve(params keys : Keyframe[]) : AnimationCurve

描述：从任意数量的关键帧创建一个动画曲线。

该函数从可变数量的 Keyframe 参数创建一个曲线，如果你想从一个关键帧数组中创建一个曲线，创建一个空的曲线并指定 keys 属性。

```
//一个慢退慢出的动画曲线（切线都是平的）。
var curve = new AnimationCurve(Keyframe(0, 0), Keyframe(1, 1);
function Update ()
{
transform.position.x = Time.time;
transform.position.y = curve.Evaluate(Time.time);
}
```

- static function AnimationCurve () : AnimationCurve

描述：创建一个空的动画曲线

函数

- function AddKey (time : float, value : float) : int

描述：添加一个新的键到曲线。

平滑切线将被自动为该键的计算，返回该键的索引，如果因为在同一时间上已经有另一个关键帧而不能添加键，将返回-1。

◆ **function AddKey (key : Keyframe) : int**

描述：添加一个新的键到曲线。

返回该键的索引，如果因为在同一时间上已经有另一个关键帧而不能添加键，将返回-1。

◆ **function Evaluate (time : float) : float**

描述：该动画曲线在 time 的值。

◆ **function MoveKey (index : int, key : Keyframe) : int**

描述：移除 index 处的关键帧并插入键。

如果一个关键帧已经存在于 key-time，老的关键帧位置时间 key[index].time/将被用来替换，这对于在一个曲线编辑器中拖动关键帧是一个理想的行为，移动它后返回关键帧的索引。

◆ **function RemoveKey (index : int) : void**

描述：移除一个键

◆ **function SmoothTangents (index : int, weight : float) : void**

描述：平滑位于 index 处的关键帧的进出切线。

权值为 0 时平均切线。

类方法

◆ **static function EaseInOut (timeStart : float, valueStart : float, timeEnd : float, valueEnd : float) : AnimationCurve**

描述：一个渐进渐出的曲线，开始于 timeStart,valueStart 并结束于 timeEnd, valueEnd.

◆ **static function Linear (timeStart : float, valueStart : float, timeEnd : float, valueEnd : float) : AnimationCurve**

描述：一个直线，开始于 timeStart,valueStart 并结束于 timeEnd, valueEnd.

AnimationEvent

类

AnimationEvent 类似于 SendMessage 让你调用一个脚本函数，这个脚本是动画播放的一部分。

变量

◆ **var animationState : AnimationState**

描述：引发这个事件的动画状态。

当这个方法在动画事件回调之外被调用用时返回 null。

◆ **var date：string**

描述：存储在动画剪辑中的字符串数据并将被发送到动画事件。

◆ **var functionName : string**

描述：被调用的函数的名称

这与调用 gameObject.SendMessage(animationEvent.functionName,animationEvent) 相同；

◆ **var messageOptions ：SendMessageOptions**

描述：如果选项被设置为 SendMessageOptions.RequireReceiver（缺省），当消息没有被任何组件接收时将打印一个错误消息。

◆ **var time:float**

描述：该事件被引发的时间。

构造函数

◆ **static function AnimationEvent () : AnimationEvent**

描述：创建一个新的动画事件

AnimationState

类

AnimationState 完全控制动画混合。

在大多数情况下 **Animation** 接口就足够了，并且更容易使用。如果你需要完全控制动画播放过程中的混合时，使用 **AnimationState**。

当动画播放时，**AnimationState** 允许你修改速度，权值。时间和层。也可以设置动画合成和 **wrapMode**

动画

变量

◆ **var blendMode : AnimationBlendMode**

描述：使用哪个混合模式？

// 设置 leanLeft 动画为附加混合

animation["leanLeft"].blendMode = AnimationBlendMode.Additive;

◆ **var clip ： AnimationClip**

描述：该动画状态播放的剪辑。

// 打印动画剪辑的帧频到控制台

print（animation[“walk”]clio.frameRate）;

◆ **var enabled : bool**

描述：启用/禁用动画

对于需要考虑任何影响的动画，权值需要设置成为一个大于零的值。如果动画被禁用，时间将暂停直到动画再次启用。

// 启用 walk 循环

animation["Walk"].enabled = true;

animation["Walk"].weight = 1.0;

◆ **var layer : int**

描述：动画的层。在计算混合权值时，位于较高层的动画将首先获得它们的权值。

只有较高层的动画没有使用完全全部权值时，较低层的动画才能接收混合权值。

// 放置 walk 和 run 动画在层 1

animation["Walk"].layer = 1;

animation["Run"].layer = 1;

◆ **var length : float**

描述：动画剪辑的长度，以秒计。

// 打印 Walk 动画的长度

print (animation["Walk"].length);

◆ **var name ： string**

描述：动画的名称。

◆ **var normalizedSpeed : float**

描述：归一化播放速度。

这最常用于混合两个动画时同步播放速度。在多数情况下使用 **animation.SyncLayer** 是更容易也更好

// 同步 run 和 walk 速度

animation["Run"].normalizedSpeed = animation["Walk"].speed;

◆ **var normalizedTime : float**

描述：动画的当前归一化时间。

1 为动画的末端。 **0.5** 为动画的中部。

// 快进到动画的中部

animation["Walk"].normalizedTime = 0.5;

◆ **var speed : float**

描述：动画的播放速度。**1** 为正常播放速度。

负的播放速度将回放动画。

// 向后走

animation["Walk"].speed = -1.0;

// 以双倍速度行走

animation["Walk"].speed = 2;

◆ **var time ： float**

描述：动画的当前时间

如果时间大于长度它将按照 **wrapMode** 回绕。该值可以大于动画的长度。看这种情况下播放模式将在采样前重映射时间。这个值从 **0** 到无穷。

// 回退 **walk** 动画

animation["Walk"].time = 0.0;

◆ **var weight : float**

描述：动画的权值

// 设置 **walk** 动画的混合权值为 **0.5**

animation["Walk"].weight = 0.5;

◆ **var wrapMode : WrapMode**

描述：动画的回绕模式

默认的 **wrapMode** 被初始化为在 **Animation** 组件中设置的回绕模式值。

// 设置 **walk** 动画回绕模式为循环

animation["Walk"].wrapMode = WrapMode.Loop;

函数

◆ **function AddMixingTransform (mix : Transform, recursive : bool = true) : void**

描述：添加应该被动画的变换。这允许你缩减需要创建的动画数量。

例如你可能有一个挥手的动画。你可能想在一个空闲角色或行走的角色上播放挥手动画。那么你需要为空闲和行走分别创建挥手动画。运用合成挥手动画，它将由肩膀完全控制。但是下半身不会受它的影响，继续播放空闲或行走动画。因此你只需要一个挥手动画。

如果 **recursive** 为真，所有 **mix** 变换的子也都将被动画。如果你不调用 **AddMixingTransform,**所有动画曲线将被使用。

// 使用路径添加混合

var shoulder : Transform;

animation["wave_hand"].AddMixingTransform(shoulder);

function Start ()

{

//使用路径添加混合变换

var mixTransform = transform.Find("root/upper_body/left_shoulder");

animation["wave_hand"].AddMixingTransform(mixTransform);

}

Application

类

访问应用程序的运行时数据。

这个类包含静态的方法来查找相关的信息并控制运行时数据。

类变量

◆ **static var absoluteURL : string**

描述：到 web 播放器数据文件夹的绝对路径（只读）。

Application.absoluteURL 和 **Application.srcValue** 允许你检测 **unityWeb** 数据文件是否被移动或链接接到其他位置。你也许想保护这两者来防止盗用数据文件的行为。

```
// 检测你的数据文件是否被移动到其他的服务器
// 或是被链接到其他地方
function Start ()
{
var isPirated = false;
if (Application.platform == RuntimePlatform.WindowsWebPlayer ||Application.platform == RuntimePlatform.OSXWebPlayer)
{
if (Application.srcValue != "game.unity3d")
isPirated = true;
if (String.Compare (Application.absoluteURL,http://www.website.com/Game/game.unity3d,true)!=0)
isPirated = true;
if (isPirated)
print("Pirated web player");
}
}
```

◆ **static var dataPath : string**

描述：包含游戏数据文件夹的路径（只读）。

这个值依赖于运行的平台：

Unity 编辑器： <工程文件夹的路径>/Assets

Mac 播放器： <到播发器应用的路径>/Contents

Win 播放器： < 包含可执行播发器的文件夹的路径>\Data

Dasboard 窗口： < dashboard widget bundle 的路径>

Web 播放器： 到播放器数据文件夹的绝对路径（没有实际的数据文件名称）

```
// 打印到数据文件夹的路径
Print（Application.dataPath）;
```

◆ **static var isEditor : bool**

描述：是在 **Unity** 编辑器内运行？（只读）

如果游戏从 **Unity** 编辑器中运行，返回真；如果从其他部署目标运行返回假。

```
if (Application.isEditor)
{
print("We are running this from inside of the editor!");
}
```

◆ **static var isLoadingLevel : bool**

描述：正在加载某些关卡？（只读）

LoadLevel 和 **LoadLevelAdditive** 不会立即发生 一个新的关卡在当前游戏帧之后被加载。如果关卡加载所请求的帧已经完成 **isLoadingLevel** 返回 **true**。

参见：**LoadLevel**，**LoadLevelAdditive**

◆ **static var isPlaying : bool**

描述：在任何类型的播放器中时返回真（只读）。

在 **Unity** 编辑器中，如果处于播放模式时返回真。

```
if (Application.isPlaying)
{
print("In player or playmode");
}
```

◆ **static var levelCount : int**

描述：可用的总关卡数（只读）。

```
// 加载一个随机的关卡
Application.LoadLevel (Random.Range(0, Application.levelCount-1));
```

◆ **static var loadedLevel : int**

描述：最后一个被加载的关卡的索引（只读）。

```
print (Application.loadedLevel);
```

◆ **static var loadedLevelName : string**

描述：最后一个被加载的关卡的名称（只读）。

```
print (Application.loadedLevelName);
```

◆ **static var platform : RuntimePlatform**

描述：返回游戏运行的平台（只读）。

如果你要做一些平台相关的操作使用这个属性。参见：**RuntimePlatform**

```
function Start ()
{
if (Application.platform == RuntimePlatform.WindowsPlayer)
print ("Do something special here!");
}
```

◆ **static var runInBackground : bool**

描述：应用程序在后太时是否应该被运行？

默认为假（当程序在后台时暂停）。

```
// 让游戏运行，即使是在后台
Application.runInBackground = true;
```

◆ **static var srcValue : string**

描述：相对于 **html** 文件的 **web** 播放器数据文件的路径（只读）。

这是被写到 **html** 文件中的路径，它是作为 **object** 的 **src** 参数和 **cmbed** 标签。因此如果它是绝对 **url**，**srcvalue** 将含有绝对路径。

Application.absoluteURL 和 **Application.srcValue** 允许你检测你的 **unityWeb** 数据文件是否被移动或链接到其他位置。你也许想保护这两者来阻止盗用数据文件的行为。

```
// 检测你的数据文件是否被移到其他的服务器
// 或是被链接到其他地方
function Start ()
```

```
{
Var isPirated = false;
if (Application.platform == RuntimePlatform.WindowsWebPlayer ||Application.platform == RuntimePlatform.OSXWebPlayer)
{
if (Application.srcValue != "game.unity3d")
isPirated = true;
if (String.Compare(Application.absoluteURL,"http://www.website.com/Game/game.unity3d",true)!= 0)
isPirated = true;
if (isPirated)
print("Pirated web player");
}
}
```

◆ static var streamedBytes : int

描述：我们从主 Unityweb 流中下载了多少字节（只读）。

在 web 播放器中这将返回到目前为止已经下载的压缩字节数。在独立模式或编辑器中这个总是返回零。

参见：GetStreamProgressForLevel 函数

◆ static var targetFrameRate : int

描述：命令游戏尝试以一个特定的帧率渲染。

设置 targetFrameRate 为-1（默认）使独立版游戏尽可能快的渲染，并且 web 播放器游戏以 50-60 帧/秒渲染，取决于平台。

注意设置 targetFrameRate 不会保证帧率，会因为平台的不同而波动，或者因为计算机太慢而不能取得这个帧率。

在编辑器中 targetFrameRate 被忽略。

◆ static var unityVersion : string

描述：用于播放内容的 Unity 运行时版本。

类方法

◆ static function CancelQuit () : void

描述：取消退出。这可以用来在退出游戏的时候显示一个退出画面。

这个函数只工作在播发器中，在 web 播放器或编辑器中不做任何事。

```
// 延迟 2 秒退出。
// 在这段时间内加载退出画面
var showSplashTimeout = 2.0;
private var allowQuitting = false;
function Awake () {
// 需要在多个关卡中使用的游戏物体
DontDestroyOnLoad (this);
}
function OnApplicationQuit () {
// 如果我们还没有加载到最后的退出画面
if (Application.loadedLevelName.ToLower()!= "finalsplash")
StartCoroutine("DelayedQuit");
```

```
// Don't allow the user to exit until we got permission in
if (!allowQuitting)
Application.CancelQuit();
}
function DelayedQuit ()
{
Application.LoadLevel("finalsplash");
// 等待 showSplashTimecout
yield WaitForSeconds(showSplashTimeout);
// 然后退出
allowQuitting = true;
Application.Quit();
}
```

◆ static function CanStreamedLevelBeLoaded(levelIndex : int) : bool

描述：可以加载流式关卡了吗？

参见：GetStreamProgressForLevel 函数。

◆ static function CanStreamedLevelBeLoaded(levelName : string) : bool

描述：可以加载流式关卡了吗？

参见：GetStreamProgressForLevel 函数。

◆ static function CaptureScreenshot(filename : string) : void

描述：截取屏幕为 PNG 文件放置在路径 filename。

如果文件已经存在，它将被覆盖。如果在 web 播放器或者 Dashboard 窗口中使用该函数，它将不做任何事情。

```
function OnMouseDown () {
Application.CaptureScreenshot("Screenshot.png");
}
```

◆ static function ExternalCall(functionName:string,params args:object[]):void

描述：调用一个包含中网页中的函数（只用于 Web Player）。

调用包含在网页中名为 functionNameJavaScript 函数，并传递给定的参数。支持原始的数据类型(string, int, float, char)和这些类型的数字。如何其他的对象被转化为字符串（使用 ToString 方法）并作为字符串传递。

传递的参数数量是可变的。

```
// 调用网页上的 MyFunction1 并不使用参数。
Application.ExternalCall ("MyFunction1");
//调用网页上的 MyFunction2 并使用字符串参数。
Application.ExternalCall ("MyFunction2", "Hello from Unity!");
//调用网页上的 MyFunction3 并使用几个不同类型的参数。
Application.ExternalCall ("MyFunction3", "one", 2, 3.0);
```

被调用的在 HTML 中的函数只需要使用标准的语法即可，例如：

```
<script language="JavaScript" type="text/javascript">
<!—
// 使用来自 Unity 的调用，这将接受
// "Hello from Unity!" 做为参数
```

function MyFunction2(arg)
{
alert(arg);
}
-->
</script>
See Also: Browser to Unity communication, Application.ExternalEval.

◆ static function ExternalEval (script : string) : void

描述：调用包含在网页中的片段脚本函数（只用于 Web Player）。
这将执行包含在网页中 JavaScript 片段 script
// 导航到前一个页面
Application.ExternalEval ("history.back()");
See Also: Browser to Unity communication, Application.ExternalCall.

◆ static function GetStreamProgressForLevel(levelIndex : int) : float

描述：下载了多少？
在 web 播放器中这将返回这个关卡的进度。
参见：CanStreamedLevelBeLoaded

◆ static function GetStreamProgressForLevel (levelName : string) : float

描述：下载了多少？[0.....1]
在 web 播放器中这将返回关卡的进度。
参见：CanStreamedLeverlBeLoaded 函数。

◆ static function LoadLevel(index : int) : void

描述：加载关卡。
这个函数按照索引加载关卡。在 Unity 中使用 File->Build Settings.....菜单可以看到所有关卡的索引列表。在你能过加载关卡之前你必须将它添加到游戏使用关卡列表中。在 Unity 中使用 File->Build Settings.....并添加你需要的关卡到关卡列表中。
//加载索引为 0 的关卡
Application . LoadLevel（0）;
当加载崭新的关卡时，所有已经加载的游戏物体都将被销毁。 如果你想让物体在被加载新关卡时不被销毁，使用 Object.DontDestroyOnLoad 。

◆ Static function LoadLevel(name : string) : void

描述：按照它的名称加载关卡。
在你能够加载关卡之前你必须将它添加到游戏使用的关卡列表中。在 Unity 中使用 File->Build Settings..... 并添加你需要的关卡到关卡列表中。关卡被加载所有激活物体上的 MonoBehaviour . OnLevelWasLoaded 都被调用。
// 加载名为“HighScore”的关卡。
Application . LoadLevel("HighScore");
当加载新的关卡时，所有已经加载的游戏物体都将被销毁。 如果你想让物体在加载新关卡时不被销毁，使用 Object. DontDestroyOnLoad。

◆ static function LoadLevelAdditive (index : int) : void

◆ static function LoadLevelAdditive (name : string) : void

描述：额外地加载一个关卡。
不同于 LoadLevel，LoadLeavelAdditive 不会销毁当前关卡中的物体。新关卡中的物体将被添加到当前关卡。这对于创建连续的虚拟世界时非常有用的，当你走过时更多的

内

荣被加载。

◆ static function OpenURL(url : string) : void

描述：在浏览器中打开 url 。

在编辑器或者独立播放器模式下，这将在缺省的浏览器中使用新页打开 url 。这将是浏览器位于前端。

但在网页中执行时，包含插件的页将被重定向到 url 。

```
Function Start ( ) {
Application . OpenURL ("http://unity3d.com");
}
```

◆ Static function Quit ( ) : void

描述：退出应用程序。在编辑器或者 web 播放器中退出被忽略。

```
//当用户点击 escape 时退出播放器
Function Update ( ){
If ( Input GetKey ( "escape" )){
Application . Quit ( ) ;
}
}
```

Array

类

数组允许你将多个对象存储在一个变量中。

Array 类只能用于 JavaScript 。更多关于 C#或 JavaScript 中 ArrayLists ，字典或哈希表的信息参考 MSDN 。

这是一个基本的例子，说明可以使用一个数组类做什么

```
    function Start( )
{
var arr = new Array ( ) ;
arr.Push ("Hello"); //添加一个元素
Print(arr[ 0]);        //打印第一个元素
arr length = 2 ;              //调整数组大小
arr [ 1] = "World"; //将“World”赋给第二个元素
for (var value : String in arr)        //遍历这个数组
{
Print ( value );
}
        }
```

Unity 中有两种类型的数组，内置数组和普通的 JavaScript 数组。

内置的数组（原始的.NET 数组），是非常快速和有效的但是它们不能被调整大小。

它们是静态类型的，这允许它们在检视面板中被编辑。这是如何使用内置数组的简单例子。

```
//在检视面板中公开一个浮点数组，你可以在那里编辑它
var value : float[ ];
Function Start ( )
{
```

```
//遍历数组
        for ( var value in values){
Print ( value );
}
//因为我们不能调整内置数组的大小
//我们必须重新创建一个数组来调整它的大小
value = new float[ 10 ];
value[ 1 ] = 5.0;//给第二个元素赋值
}
```

内置数组在性能相关的代码中非常有用的(使用 Unity 的 JavaScript 和内置数组可以非常容易使用 mesh interface 在一秒内处理两万个顶点。) 另一方面，普通的 JavaScript 数组可以调整大小，排序并可以做所有你期望的数组类的操作。JavaScript 数组不显示在检视面板中。你可以容易地在 JavaScript 数组和内置数组之间转换。

```
function Start ( )
{
var array = new Array ( Vector3(0,0,0),Vector3(0,0,1));
array .Push (Vector3 (0,0,2));
array .Push (Vector3 (0,0,3));
//拷贝 js 数组到内置数组
var builtinArray : Vector3[ ] = array . ToBuiltin ( Vector3 );
//将内置数组赋给 js 数组
var newarr = new Array ( builtinArray );
//newarr 与 array 包含相同的元素
print ( newarr );
}
```

注意按照 Unity 的命名规则下面所有函数均大写开头。为方便 JavaScript 用户 ， Unity 数组类也接受小写函数。

变量

◆ var length : int

描述：数组的长度属性，返回或设置数组中元素的数量。

```
function Start ( )
{
var arr = Array ( "Hello" , "World" ) ;
print (arr . length ) ;      //打印两个
arr . Length = 5 ;    //调整数组的大小为 5
}
```

函数

◆ function Add (value : object) : void

描述：添加 value 到数组末端。

```
var arr = new Array ("Hello");
arr.Add (" World ");
Print ( arr );          //打印"Hello ","World"
```

◆ function Clear ( ) : void

描述: 清空数组。 数组的长度将为零。

```
var hello = new Array ("Hello ","World ");
hello.Clear ( ) ;              //现在 hello 包含零个元素
```

◆ function Concat (array :Array , optionalArray0: Array, optionalArray1 : Array):Array

描述：连接两个或多个数组。这个方法不会改变已有的数字并返回连接后的数组拷贝

```
function Start ( ) {
   var arr = new Array ("Hello","World"):
   var arr2 =    new Array ("!");
   var joined = arr.Concat ( arr2 );        //现在 jointed 包含所有 3 个字符串
   Print ( joined );          //打印"Hello","World","!"
}
```

◆ function Join (seperator ： string) : String

描述：链接数组内容为一个字符串。元素将被 seperator 字符串分割,并返回数组的拷贝

```
function Start ( ){
      var arr = new Array ("Hello" , "World");
      print ( arr . join (" , "));//打印"Hello， World"
}
```

◆ function Pop ( ) : object

描述：移除数组最后一个元素并返回它。

```
      var arr = new Array ("Hello ","World");
      arr . Pop ( );
      print ( arr );//只打印"Hello"
```

◆ function Push (value : object) : int

描述： 添加 value 到数组末端。并返回新数组长度。

```
var arr = new Array ("Hello");
arr.Push ("World");
print ( arr );//打印"Hello","World"
```

◆ function RemoveAt (index : int) : void

描述：从数组中移除索引为 index 的元素。

```
var arr = new Array ("Hello" , " and good morning" , "World ");
arr.Remove ( 1 ) ;          //移除  "and good morning"
print ( arr );//打印  " Hello World "
```

◆ function Reverse () : Array

描述：颠倒数组中所有元素顺序。

```
var hello = new Array (" Hello " ," World ") ;
hello Reverse( ) ;
print (hello);//打印 World，Hello
```

◆ function Shift ( ) :object

描述：移除数组的第一个元素并返回它。

```
var arr = new Array ( " Hello " , " World ");
arr . Shift ( ) ;
print ( " World " ) ;        //现在 arr 只包含" World "
```

◆ function Sort( ) : Array

描述：排序所有数组元素

```
var hello = new Array ( " e " ," a " ," b ");
```

hello . Sort ( ) ;

print (hello) ;// 打印 a ， b ， c

◆ function Unshift (newElement : object , optionalElement : object) : int

描述： Unshift 添加一个或多个元素到数组的开始位置并返回新的数组长度。

var arr = new Array (" Hello "," World ");

arr . Unshift (" This "," is ");

print (arr) ;//打印 This，is，Hello，World

BitStream

类

BitStream 类表示序列化的变量，打包到一个流中。

数据可以被序列化，传输，然后远端使用这个类接受。参考 Network View component reference

获取关于网络同步的信息和 Network. OnSerializeNetworkView 函数获取更多信息。

变量

◆ var isReading : bool

描述：这个 BitStream 现在在被读吗？

参考 Network. OnSerializeNetworkView

◆ var isWriting : bool

描述：这个 BitStream 现在在被写吗？

参考 Network. OnSerializeNetworkView

函数

◆ function Serialize (ref value : bool) : void

◆ function Serialize (ref value : char) : void

◆ function Serialize (ref value : short) : void

◆ function Serialize (ref value : int) : void

◆ function Serialize (ref value : float , maxDelta : float = 0.00001F) : void

◆ function Serialize (ref value : Quaternion, maxDelta : float = 0.00001F) : void

◆ function Serialize (ref value : Vector3, maxDelta : float = 0.00001F) : void

◆ function Serialize (ref value : NetworkPlayer) : void

◆ function Serialize (ref viewID: NetworkViewID) : void

描述：BitStream 类可以序列化几个不同类型的变量。

包含：bool , char , short , int , float , Quaternion , Vector3 和 NetworkPlayer

注意 serialize (char)系列化一个字节，因此，它只能用于 0.....255 之间的字符。

BoneWeight

结构

网格上一个顶点的蒙皮骨骼权值

每个被蒙皮的点至多有四个骨头。所有权值的和应该为 1。权值和骨骼索引应该被以权值递减的顺序定义。如果一个顶点被少于四个骨骼影响，剩下的权值应该为 0。

参见：Mesh.boneWeights 变量。

变量

◆ var boneIndex0 : int

描述：第一个骨骼的索引。

参见：weight0 .

◆ var boneIndex1 : int

描述：第二个骨骼的索引。

参见：weight1.

◆ var boneIndex2 : int

描述：第三个骨骼的索引。

参见：weight2 .

◆ var boneIndex3 : int

描述：第四个骨骼的索引。

参见：weight3 .

◆ var weight0 : float

描述：第一个骨骼的蒙皮权值。

参见：boneIndex0.

◆ var weight1 : float

描述：第二个骨骼的蒙皮权值。

参见：boneIndex1.

◆ var weight2 : float

描述：第三个骨骼的蒙皮权值。

参见：boneIndex2.

◆ var weight3 : float

描述：第四个骨骼的蒙皮权值。

参见：boneIndex3.

Bounds

结构

代表一个轴对齐包围盒。

一个轴对齐包围盒，简称为 AABB，是与坐标轴对齐的 box 并且完全包围一些物体。因为这个 box 不会绕着轴旋转，所以它可以只用 center 和 extents 定义，或者用 min 和 max 点定义。

Bounds 被 Collider.bounds,Mesh.bounds,Renderer.bounds 使用。

变量

◆var center : Vector3

描述：包围盒子的中心

◆var extents : Vector3

描述：box 的宽度。这个总是 size 的一半

◆var max : Vector3

描述：box 的最大点。这个总是等于 center + extents。

◆var min : Vector3

描述：box 的最小点。这个总是等于 center - extents。

◆var size : Vector3

描述：box 的总大小。这个总是 extents 的二倍。

size.x 是宽度， size.y 是高度， size.z 是长度。

构造函数

◆static function Bounds (center : Vector3 , size : Vector3) : Bounds

描述：用给定的 center 和总 size 创建新的 Bounds。Bounds extents 将是给定 size 的一半。

var bounds = Bounds (Vector3.zero , Vector3(1,2,1));//在原点常见柱状包围盒

函数

◆function Contains (point : Vector3) : bool

描述：point 包含在这个包围盒中吗 ？

◆function Encapsulate (point : Vector3) : void

描述：增大 Bounds 以包含这个 point.

◆function Encapsulate (bounds : Bounds) : void

描述：增大 bounds 来封装另一个 bounds。

◆function Expand (amount : float) : void

描述：沿着每个面按照 amount 增加它的 size 来扩展这个 bounds。

◆function Expand (amount : Vector3) : void

描述：沿着每个面按照 amount 增加它的 size 来扩展这个 bounds。

◆function IntersectRay (ray : Ray) : bool

描述：ray 与这个包围盒相交吗？

◆function IntersectRay (ray : Ray , out distance : float) : bool

描述：ray 与这个包围盒相交吗？

当 IntersectRay 返回真，distance 将是到射线源点的距离。

◆function SetMinMax (min : Vector3, max : Vector3) : void

描述：设定边界为盒子的 min 和 max 值。

使用这个函数要比分别指定 min 和 max 更快。

◆function SqrDistance (point : Vector3) : float

描述： 点到这个包围盒的最小平方距离。

◆function ToString ( ) : string

描述：返回一个格式化好的字符串

collision

类

描述碰撞

Collision 信息被传递到 Collider . OnCollisionEnter , Collider . OnCollisionStay 和 Collider.OnCollisionExit 事件。参见： ContactPoint.

变量

◆var collider : Collider

描述：碰撞到的 Collider (只读).

为了得到所有被碰撞到的碰撞器的细节，你需要迭代接触点(contacts 属性)。

◆var contacts : ContactPoint [ ]

描述：接触点由物理引擎产生。

每个 contact 包含一个接触点，法线和两个发生碰撞的碰撞器（参考 ContactPoint）。在 OnCollisionStay 或者 OnCollisionEnter 内可以确保 contacts 有至少一个元素。

```
function OnCollisionStay ( collision : Collision ){
//检查碰到碰撞器是否有刚体
//然后使用一个力
for ( var contact : ContactPoint in collision . contacts ) {
print ( contact.thisCollider . name + "hit" + contact . otherCollider .name );
//可视化接触点
Debug.DrawRay ( contact . point , contact . normal, Color .white ) ;
}
```

```
}
//一枚手榴弹，在击中一个表面时初始化一个爆炸预设，然后销毁它
var explosionPrefab : Transform;
function OnCollisionEnter( collision : Collision ){
    //旋转这个物体使 y 轴面沿着表面法线的方向
    var contact = collision . contact [ 0 ];
    var rot = Quaternion . FromToRotation ( Vector3.up , contact . normal ) ;
    var pos = contact . point ;
    Instantiate ( explosionPrefab , pos , rot ) ;
    Destory ( gameObject );//销毁这个投射物
    }
```

◆var gameObject : GameObject

描述：/ gameObject / 是与之碰撞的物体（只读）

◆var relativeVelocity : Vector3

描述: 两个碰撞物体的相对线形速度（只读）。

```
//当以较大的速度碰到一个物体时播放声音
function OnCllisionEnter ( collision : Collision ) {
if ( collision . relativeVelocity . magnitude > 2 )
audio .Play ( );
}
```

◆var rigibody : Rigidbody

描述：碰撞到的 Rigidbody（只读），如果碰到的物体是一个没有附加刚体的碰撞器，返回 null

```
//让所有碰到的刚体向上飞
function OnCollisionStay ( collision : Collision ) {
//检查碰到的碰撞器是否有一个刚体 ，然后使用力
if ( collision . rigidbody ){
collision . rigidbody .AddForce ( Vector3 . up * 15 ) ;
}
}
```

◆var transform : Transform

描述：碰撞到的物体的 Transform（只读）。

如果碰到一个带有 Rigidbody 的碰撞器，transform 将是所有附加刚体的变换。如果碰到了一个没有刚体的碰撞器，transform 将是所有附加碰撞器的变换。

color

结构

表示 RGBA 颜色。

这个结构被用在整个 Unity 中传递颜色。每个颜色组件是一个 0 到 1 之间的浮点数。

组件(r ,g ,b)在 RGB 颜色空间内定义一个颜色。Alpha 组件（a）透明性 - alpha 为 0 是完全不透明，alpha 为 1 是完全透明。

变量

◆var a : float

描述：颜色的 Alpha 组件。

```
var color = Color . white ;
```

color . a = 0 ;

◆var b : float

描述：颜色的蓝色组件。

var color = Color .white;

color .b = 0 ;

◆var g : float

描述：颜色的绿色组件

var color = Color . white ;

color . g = 0 ;

◆var grayscale : float

描述：颜色的灰度值（只读）

var color = Color (3 , 4 ,6) ;

print (color . grayscale) ;

◆var r : float

描述：颜色的红色组件。

var color = Color . white ;

color . r = 0

◆var this [index : int] : float

描述：分别使用[0] ,[1] ,[2] ,[3]访问 r ， g ，b ，a 组件。

Color p ;

p [1] = 5 ;//与 p .g = 5 相同

构造函数

◆static function Color (r : float , g : float , b : float, a: float) : Color

描述：用给定的 r , g , b , a ,组件构建一个新的颜色。

var color = Color (0.2 , 0.3 , 0.4 , 0.5) ;

◆static function Color (r : float , g : float , b : float) : Color

描述：用给定的 r , g , b 组件构建一个新的颜色并设置 a 为 1

var color = Color (0.2 , 0.3 , 0.4) ;

函数

◆function ToString ( )：string

描述：返回格式化好的这个颜色的字符串。

print (Color .white) ;

类变量

◆static var black : Color

描述：黑色。 RGBA 为（ 0 , 0 , 0 , 1).

◆static var blue : Color

描述：蓝色。 RGBA 为（ 0 , 0 , 1 , 1).

◆static var clear : Color

描述：完全透明。 RGBA 为（ 0 ,0, 0, 0).

◆static var eyan : Color

描述：青色。 RGBA 为（ 0 , 1 , 1 , 1).

◆static var gray : Color

描述：灰色。RGBA 为 (5 , 5 , 5 , 1).

◆static var green : Color

描述：绿色。RGBA 为（0, 1 , 0, 1).

◆static var grey : Color

描述：英式拼法为 gray。RGBA 为（0.5 , 0.5 , 0.5 , 1).

◆static var magenta : Color

描述：紫红色。RGBA 为（1 , 0 , 1 , 1).

◆static var red : Color

描述：全红。RGBA 为（1 , 0 , 0, 1).

◆static var white : Color

描述：全白。RGBA 为（1 , 1 , 1 , 1).

◆static var yellow : Color

描述：黄色。RGBA 是怪异的（1 , 235/255 , 4/255 , 1)，但是这个颜色看起来非常好！

类方法

◆static function Lerp (a : Color , b : Color , t : float) : Color

描述：在颜色 a 和颜色 b 之间按照 t 插值。

/ t /被限定到 0 和 1 之间，当 t 为 0 时返回 a . 当 t 为 1 时返回 b

◆static operator * (a : Color , b : Color) : Color

描述：乘两个颜色，每个组件被分别乘。

◆static operator * (a : Color , b : float) : Color

描述:用浮点数 b 乘以颜色 a。每个组件被分别乘。

◆static operator * (a : float , b : Color) : Color

描述：用浮点数 b 乘以颜色 a。每个组件被分别乘。

◆static operator + (a : Color , b : Color) : Color

描述：加两个颜色，每个组件被分别加。

◆static operator - (a : Color , b : Color) : Color

描述：从颜色 a 中减去颜色 b。每个组件被分别减。

◆static operator / (a : Color , b : float) : Color

描述：乘用浮点数 b 除以 a。每个组件被分别除。

◆static implicit function Color (v : Vectro4) : Color

描述：Colors 可以被隐式转化为 Vector4，或由它转化而来。

◆static implicit function Vector4 (c : Color) : Vector4

描述：Colors 以被隐式的转化为 Vector4，或由它转化而来。

ContactPoint

结构

描述：碰撞发生的接触点。

接触点被存储在 collision 结构中，参见 collision ， collision . OnCollisionEnter, Collision . OnCollisionStay , Collision . OnCollisionExit.

变量

◆var normal : Vector3

描述：接触点的法线

◆var otherCollider : Collider

描述：碰撞中的另一个碰撞器

◆var point : Vector3

描述：接触点

◆var thisCollider : Collider

描述：碰撞中的第一个碰撞器

ControllerColliderHit

类

ControllerColliderHit 被 **CharacterController ． OnControllerColliderHit** 使用来给出详细的关于碰撞和如何处理它们的信息。

变量

◆var collider : Collider

描述：被控制器碰到的碰撞器。

◆var controller : CharacterController

描述：碰到该碰撞器的控制器。

◆var gameObject : GameObject

描述：被控制器碰到的游戏物体。

◆var moveDirection : Vector3

描述：从胶囊的中心到接触点的大致方向。

这可以用来找到一个合理的方向将力应用到接触的刚体。

◆var moveLength : float

描述：角色碰到这个碰撞器时已经行走了多远。

注意这可能不同于你传递到 **CharacterController . Move** 的。因为所有的移动都是被碰撞器制约的。

◆var normal : Vector3

描述：在世界空间中碰撞表面的法线。

◆var point : Vector3

描述：世界空间中的碰撞点。

◆var rigidbody : Rigidbody

描述：被控制器碰到的刚体。

如果没有接触一个刚体而是一个静态碰撞器时为 **null**。

◆var transform : Transform

描述：被控制器碰到的变换。

Debug

类

一个类，包含用于开发游戏时的调试方法。

类变量

◆static var isDebugBuild : bool

描述：在 **Build Settings....**对话框中，有一个被称为**"strip debug symbols"**的复选框。

如果它被选择 **isDebugBuild** 将为关。在编辑器中 **isDebugBuild** 总是返回真，建议在发布游戏的时候移除所有对 **Debug .Log** 的调用，这样你就能够容易的发布带有调试输出的测试版，而最终版没有调试输出。

```
//只有这是调试版时，记录调试信息
if ( Debug .isDebugBuild ) {
Debug . Log ( " Something bad happened ! " ) ;
}
```

类方法

◆static function Break ( ) : void

描述：暂停编辑器

Debug . Break ( ) ;

◆static function DrawLine (start:Vector3, end: Vector3, color : Color = Color . white) : void

描述：从 point 开始到 end 用颜色绘制一条线。

这个线将被绘制在编辑器的场景视图中。如果在游戏视图中启用了 gizmo 绘制，这个线也将被绘制在这里。

//从世界坐标的原点到点(1 , 0 , 0)绘制一条红色的线

function Update ( ) {

Debug . DrawLine (Vector3 . Zero , new Vector3 (1 , 0 , 0), Color . red) ;

}

◆static function DrawRay (start:Vector3, dir : Vector3, color : Color = Color . white) : void

描述：从 start 到 start+dir 用颜色绘制一条线。

//绘制一条 10 米长的线从 position，沿着变换的 z 轴向前。

function Update ( ) {

var forward = transform . TransformDirection (Vector3 . forward) * 10 ;

Debug . DrawRay (transform . position .Vector3 . forward * 10 , Color . green);

}

◆static function Log (message : object) : void

描述：记录 message 到 Unity 控制台。

Debug . Log ("Hello");

◆static function Log (message : object . context : Object) : void

描述：记录 message 到 Unity 控制台。

当你在控制台中选择消息的时候一个到上下文物体的链接将被绘制。这是非常有用的。如果你想知道那个物体发生了错误。

Debug . Log ("Hello" , gameObject) ;

◆static function LogError (message : object) : void

描述：Debug . Log 的一个变体，用来记录错误信息到控制台。

var memberVariable : Transform ;

if (memberVariable == null)

Debug . LogError (" memberVariable must be set to point to a Transform. ") ;

◆static function LogError (message : object , context : Object) : void

描述：Debug . Log 的一个变体，用来记录错误信息到控制台。

等你在控制台中选择消息的时候一个到上下文物体的链接将被绘制。这是非常有用的，如果你想知道那个发生了错误。

var memberVariable : Transform ;

if (memberVariable == null)

Debug . LogError ("memberVariable must be set to point to a Transform " , this) ;

◆static function LogWarning (message : object) : void

描述：Debug . Log 的一个变体，用来记录警告信息到控制台。

◆static function LogWarning (message : object , context : Object) : void

描述：Debug . Log 的一个变体，用来记录警告信息到控制台。

当你选择控制台中的一个消息时，一个到上下文物体的连接将被绘制。这是非常有用的， 如果你想知道那个物体发生了错误

Event

类

一个 UnityGUI 事件。

对应于用户的输入事件（按键，鼠标事件），或者是 UnityGUI 布局或渲染事件。

对于每个事件 OnGUI 在脚本中被调用；因此 OnGUI 在每帧中被潜在调用多次。

Event . current 对应于 OnGUI 调用“当前”事件。

参见：GUIScripting Guide

变量

◆var alt : bool

描述：Alt/Option 键被按住？（只读）

在 windows 下，如果 Alt 键被按下返回真。在 Mac 下，如果 Option 键被按下返回真。

◆var button : int

描述：哪个鼠标键被按下

0 表示左键，1 表示右键。2 表示中键。在 EventType . MouseDown ,EventType .MouseUp 事件中使用。

◆var capsLock : bool

描述：Caps Lock 处于打开状态？（只读）

如果 Caps Lock 为打开返回真

◆var character : char

描述：输入的字符

在 EventType . KeyDown 事件中使用，注意 EventType . KeyUp 事件不包含字符，只包含 Event . keyCode .

参见：Event . keyCode.

◆var command : bool

描述：Command/Windows 键被按住？（只读）

在 Windows 下，如果 Windows 键被按下返回真。在 Mac 下，如果 Command 键被按下返回真。

◆var control : bool

描述：Control 被按下？（只读）

如果 Control 被按下返回真。

◆var delta : Vector2

描述：与上次事件相比，鼠标的相对移动。

在 EventType .MouseMove，EventType .MouseDrag,，EventType .ScrollWheel 时间中使用。

参见：Event . mousePosition

◆var functionKey : bool

描述：当前按下的键是功能键？（只读）

如果当前按下的键是方向键，翻页键，退格键等等时返回真，如果这个键需要特殊处理才能用与文本编辑时，functionKey 为打开。

◆var isKey : bool

描述：这个事件是键盘事件？（只读）

◆var isMouse : bool

描述：这个事件是鼠标事件？（只读）

◆var keyCode : KeyCode

描述：用于键盘事件的原始键代码

在 EventType . KeyDown 和 EventType . KeyUp 事件中使用；返回匹配物理键盘的 KeyCode 值，使用这个来处理光标，功能键等等。

参见：Event . character 。

◆var mousePosition : Vector2

描述：鼠标位置

在 EventType . MouseMove 和 EventType . MouseDrag 事件中使用。

参见：Event . delta

◆var numeric : bool

描述：当前按下的数字的键？（只读）

使用这个表示区分主&数字键。

◆var shift : bool

描述：Shift 被按下？（只读）

如果 Shift 被按下返回真。

函数

◆function GetTypeForControl (controlID : int) : EventType

参数

controlID 查询的控件 ID。从 GUIUtilty . GetControlID ( ) 获取。参考 EventType 获取可能值的列表。

描述：为给定的控件 ID 获取一个过滤的事件类型。

这个函数可以用来实现鼠标锁和键盘焦点。

◆function Use ( ) : void

描述：使用这个事件。

当已经使用了一个事件时调用这个方法。事件类型将被设置为 EventType . Used。使其他 GUI 元素忽略它。

类变量

◆static var current : Event

描述：现在被处理的当前事件。

类方法

◆static function KeyboardEvent (key : string) : Event

描述：创建一个键盘事件。

这可用于检查某个键是否被按下。可能带有调整器。key 字符串是键的名称（与输入管理器中的相同），可以使用任意数量的调整器前缀：& = Alternate , ^ = Controler, % = Command , # = Shift 例如：&f12 = Alternate +F12., " ^[0] " = Control +keypad0

```
function OnGUI ( ) {
GUILayout . Lable ( " Press Enter To Start Game ") ;
  if ( Event . current . Equals ( Event . KeyboardEvent ("[enter]")))
      Application . LoadLevel ( 1 )
if(Event current Equals(Event KeybordEvent("return") ))
Print( "I said enter ,not return – try the keypad" );
}
```

GL

类

底层图像库。

使用这个类操作激活的变换矩阵，发送与 OpengGL 立即模式相同的渲染命令并做一些

其他的底层图像操作。注意，在所有情况下使用 Graphics.DrawMesh 比任何使用立即模式绘制更有效。

这个类只限于 Unity Pro.

类变量

◆ static var LINES: int

描述：用于 Begin 的模式:绘制直线。

参见：GL . Begin, GL . End.

◆ static var modelview：Matrix4x4

描述：当前模型视矩阵。

给这个变量赋值等同于 OpenGL 中的 glLoadMatrix(mat);在其他图形 API 中对应的功能被模拟。

改变模型视矩阵覆盖当前相机的视参数，因此最常用的是使用 GL.PushMatrix 和 GL.PopMatrix 来保存和恢复矩阵。

读取这个变量返回当前模型视矩阵。

◆ static var QUADS：int

描述：用于 Begin 的模式：绘制四边形

参见：GL.Begin,GL.End.

◆ static var TRIANGLE_STRIP：int

描述：用于 Begin 的模式：绘制三角面

参见：GL.Begin,GL.End.

◆ static var TRIANGLES：int

描述：用于 Begin 的模式：绘制三角形

参见：GL.Begin,GL.End.

类方法

◆ static function Begin(mode:int) : void

参数

mode 绘制的几何体：可以是 TRIANGLES,TRIANGLE_STRIP,QUADS 或 LINES.

描述：开始绘制 3D 几何体

这个对应 OpenGL 中的 glBegin，在其他图形 API 中相同的功能被模拟 ，在 GL.Begin 和 GL.End 之间，可以调用 GL .Veriex,GL.Color,GL.TexCoord 和其他立即模式绘制函数。

在绘制你自己的几何体时，你应该注意它们的裁剪。裁剪规则可能会因为不同的图形 API 而不同。在大多数情况下在 shader 中使用 Cull Off 命令是安全的。

参见：GL.End.

◆ static function Clear(clearDepth:bool,clearColor:bool,backgroudColor):void

参数

clearDepth 应该清除深度缓存？

clearColor 应该清除颜色缓存？

backgroudColor 颜色被清理为什么，只有当 clearColor 为 true 时使用.

描述：清除当前渲染缓存

这将清除屏幕或激活的 RenderTexture.

◆ static function Color(c : Color) : void

描述：设置当前顶点颜色

这个对应 OpenGL 中的 glColor4f(c. r,c.g,c.b,c.a):在其他图形中 API 中相同的功能被模拟，

为了使逐顶点颜色可以在不同的硬件上工作，你需要使用绑定了颜色通道的 shader。参考 BindChannels 文档.

这个函数只在 GL.Begin 和 GL.End 函数之间调用.

◆ static function End( ) : void

描述：结束绘制 3D 几何体

这个对应 OpenGL 中的 glEnd;在其他图形 API 中相同的功能被模拟.

参见：GL.Begin.

◆ static function LoadIdentity( ) : void

描述：加载单位矩阵到当前模型视矩阵。

这个函数覆盖当前相机的视参数，因此最常用的是使用 GL.PushMatrix 和 GL.PopMatrix 来保存和恢复矩阵。

◆ static function LoadOrtho( ) : void

描述：辅助函数用来设置一个正交透视变换

调用 LoadOrtho 知道，视锥从 (0,0,-1) 变化到(1,1,100).

◆ static function LoadPixelMatrix( ) : void

描述：设置一个用于像素修正渲染的矩阵。

这个设置模型视和投影矩阵，因此 X，Y 坐标直接映射到像素。(0,0)位于当前相机视口的左下角。Z 坐标从-1 到+100

这个函数覆盖当前相机的参数，因此最常用的是使用 GL.PushMatrix 和 GL.PopMatrix 来保存和恢复矩阵。

◆ static function LoadPixelMatrix(left:float,right:float,bottom:float,top:float):void

描述：设置一个矩阵的像素正确渲染。

这样设置投影矩阵点以便 X、Y 坐标图直接像素化。（0,0）在底部左侧当前摄像机的视角。Z 坐标是从-1 到+100。

这个函数覆盖了相机的参数，所以通常你要保存和恢复矩阵就使用 GLPushMatrix 和 GL.PopMatrix。

◆ static function LoadProjectionMatrix (mat : Matrix4x4) : void

描述：加载到当前任意矩阵投影矩阵。

这个函数重写当前摄像机的投影参数，所以通常你要保存和恢复投影矩阵就使用 GLPushMatrix 和 GL.PopMatrix。

◆ static function MultiTexCoord (unit : int, v : Vector3) : void

描述：设置当前纹理坐标(v.x,v.y,v.z)实际的纹理单元。

在 OpenGL 中 glMultiTexCoord 为特定的纹理单元如果多纹理是可用的。在其他图形的 API 中相同的功能进行了仿真。

这个函数只能被称为介于 GL.Begin 和 GL.End 功能之间。

◆ static function MultiTexCoord2 (unit : int, x : float, y : float) : void

描述：设置当前纹理坐标（x，y）的为实际纹理单元。

在 OpenGL 中 glMultiTexCoord 为特定的纹理单元如果多纹理是可用的。在其他图形的 API 中相同的功能进行了仿真。

这个函数只能被称为介于 GL.Begin 和 GL.End 功能之间。

◆ static function MultiTexCoord3(unit : int, x : float, y : float, z : float) :void

描述：设置当前纹理坐标（x，y，z）的为实际纹理单元。

在 OpenGL 中 glMultiTexCoord 为特定的纹理单元如果多纹理是可用的。在其他图形的 API 中相同的功能进行了仿真。

这个函数只能被称为介于 GL.Begin 和 GL.End 功能之间。

- **static function MultMatrix (mat : Matrix4x4) : void**

描述：复制当前的点矩阵和其中的一个说明。

相当于 glMultMatrix（垫在）OpenGL; 在其他图形 API 的相应功能是相仿的。

换点矩阵覆盖当前相机视图的参数，所以通常你要保存和恢复投影矩阵就使用 GLPushMatrix 和 GL.PopMatrix。

- **static function PopMatrix () : void**

描述：恢复了投影和点矩阵的矩阵堆栈的顶部。

换投影矩阵点覆盖当前相机视图的参数。这些矩阵可以用 GLPushMatrix 和 GL.PopMatrix 来保存和恢复。

参见：PushMatrix 函数。

- **static function PushMatrix () : void**

描述：节约双方投影矩阵对点和矩阵堆栈。

换投影矩阵点覆盖当前相机视图的参数。这些矩阵可以用 GLPushMatrix 和 GL.PopMatrix 来保存和恢复。

参见：PopMatrix 函数

- **static function SetRevertBackfacing(revertBackFaces : bool) : void**

描述：选择是否翻转隐面剔除，是（真）或者不是（假）

- **static function TextCoord (v : Vector3) : void**

描述：为所有纹理单元设置当前纹理坐标(v.x, v.y, v.z)

这个对应于 OpengGL 中用于所有纹理单元的 glMultiTexCoord 或者多纹理不可用时的 glTexCoord，在其他的图形 API 中仿真了相同的功能。

这个函数只在 GL.Begin 和 GL.End 函数之间调用。

- **static function TexCoord2(x : float, y : float) : void**

描述：为所有纹理单元设置当前纹理坐标(x, y)

这个对应于 OpengGL 中用于所有纹理单元的 glMultiTexCoord 或者多纹理不可用时的 glTexCoord，在其他的图形 API 中仿真了相同的功能。

这个函数只在 GL.Begin 和 GL.End 函数之间调用。

- **static function TexCoord3(x : float, y : float, z : float) : void**

描述：为所有纹理单元设置当前纹理坐标(x, y, z)

这个对应于 OpengGL 中用于所有纹理单元的 glMultiTexCoord 或者多纹理不可用时的 glTexCoord，在其他的图形 API 中仿真了相同的功能。

这个函数只在 GL.Begin 和 GL.End 函数之间调用。

- **static function Vertex(v : Vector3) : void**

描述：提交顶点

这个对应 OpenGL 中的 glVertex3f(v.x, v.y.v.z); 在其他图形 API 中相同的功能被模拟。

这个函数只在 GL.Begin 和 GL.End 函数之间调用。

- **static function Vertex3(x : float, y : float, z : float) : void**

描述：提交顶点

这个对应 OpenGL 中的 glVertex3f(x, y, z); 在其他图形 API 中相同的功能被模拟。

这个函数只在 GL.Begin 和 GL.End 函数之间调用。

- **static function Viewport(pixelRect : Rect) : void**

描述：设置渲染视口

所有的渲染都被限制在 pixelRect 之内。

GUIContent

类

GUI 元素的内容

这个与 GUIStyle 紧密相关，GUIContent 定义渲染什么而 GUIStyle 定义如何渲染。

参见：GUIStyle

变量

- var image : Texture

描述：包含图标的图像

- var text : string

描述：包含的文本

- var tooltip : string

描述：这个元素的提示

与这个内容相关的提示。读取 GUItooltip 来获取当前用户指向的 GUI 元素的提示。

构造函数

- static function GUIContent() : GUIContent

描述：用于所有形状和尺寸的 GUIContent 的构造函数

构建一个空的 GUIContent。

- static function GUIContent(text : string) : GUIContent

描述：构建一个只包含文本的 GUIContent 物体。

使用 GUI 是，你不需要为一个简单的文本字符创建 GUIContents – 这两行代码功能等效：

```
function OnGUI()
{
    GUI.Button(Rect(0, 0, 100, 20), “Click Me”);
    GUI.Button(Rect(0, 30, 100, 20), GUIContent(“Click Me”));
}
```

- static function GUIContent(image : Texture) : GUIContent

描述：构建一个只包含图片的 GUIContent 对象。

```
var icon : Texture;
function OnGUI()
{
    GUI.Button(Rect(0, 0, 100, 20), GUIContent(icon));
}
```

- static function GUIContent(text : string, image : Texture) : GUIContent

描述：构建一个包含 text 和图片的 GUIContent 对象

```
var icon : Texture;
function OnGUI()
{
    GUI.Button(Rect(0, 0, 100, 20), GUIContent(“Click me”, icon));
}
```

- static function GUIContent(text : string, tooltip : string) : GUIContent

描述：构建一个包含 text 的 GUIContent，当用户鼠标悬停在它上面的时候，全局 GUI.tooltip 被设置为 tooltip。

```
function OnGUI()
```

```
{
    GUI.Button(Rect(0, 0, 100, 20), GUIContent("Click me",  "This is a tooltip."));
//如果用户指向这个按钮，全局提示被设置
GUI.Label(Rect(0, 40, 100, 40), GUI.tooltip);
}
```

◆ static function GUIContent(image : Texture, tooltip : string) : GUIContent

描述：构建一个包含图片的 GUIContent，当用户鼠标悬停在它上面的时候，全局 GUItooltip 被设置为 tooltip。

◆ static function GUIContent(text : string, image : Texture, tooltip : string) : GUIContent

描述：构建一个包含 text 和 image 的 GUIContent,，当用户鼠标悬停在它上面的时候，全局 GUItooltip 被设置为 tooltip。

◆ static function GUIContent(src : GUIContent) : GUIContent

描述：从另一个 GUIContent 构建一个 GUIContent。

GUILayoutOption

类

内部类用来传递布局选项给 GUILayout 函数，不要直接使用这些，而是在 GUILayout 类的布局函数中构造它们。

参见：

GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

GUILayoutUtility

类

用于实现并扩展 GUILayout 类的工具函数。

使用这个类制作你自己的 GUI 布局代码

类方法

◆ static function BeginGroup(GroupName : string) : void

◆ static function BeginLayoutGroup(style : GUIStyle, options : GUILayoutOption[], LayoutType : System.Type) : GUILayoutGroup

描述：普通的辅助函数 – 当创建一个布局组的时候使用这个。它将确保所有的事情都正确的排列。

style : 组选项的风格

option : 使用的布局选项

LayoutType : 创建的布局组的类型

◆ static function EndGroup(groupName : string) : void

◆ static function GetAspectRect(aspect : float) : Rect

◆ static function GetAspectRect(aspect : floar, style : GUIStyle) : Rect

◆ static function GetAspectRect(aspect : float, params options : GUILayoutOption[]) : Rect

◆ static function GetAspectRect(aspect : float, style : GUIStyle, params options：GUILayoutOption[]) : Rect

参数

aspect 这个元素的宽高比（宽/高）

style 一个可选的风格。如果指定风格的 **padding** 将被添加到返回举行的尺寸并且这个风格的 **margin** 将被用于间距。

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **Style** 定义的设置。参见 **GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **Rect** – 控制的矩形

描述：用一个制定的宽高比获取一个矩形。

◆ **static function GetRect(content : GUIContent, style : GUIStyle, params options : GUILayoutOption[]) : Rect**

参数

content 让出空间所显示的内容

style 用于布局的 **GUIStyle**

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。参见： **GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILAyout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight**

返回 **Rect** – 一个足够大的矩形区域用来包含用/style/渲染时的/content/。

描述：获取一个以特定风格显示内容的矩形。

◆ **static function GetRect(width : float, height : float) : Rect**

◆ **static function GetRect(width : float, height : float, params options : GUILayoutOption[]) : Rect**

◆ **static function GetRect(width : float, height : float, style : GUIStyle, params options : GUILayoutOption[]) : Rect**

参数

width 你想要的区域的宽度

height 你想要的区域的高度

style 用了布局的可选 **GUIStyle**，如果指定，风格的 **padding** 将被添加到尺寸并且它的 **margin** 将被用于间距

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见： **GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **Rect** – 用于放置控件的矩形

描述：用一个固定的内容区域获取一个矩形

◆ **static function GetRect(minWidth : float, maxWidth : float, minHeight : float, maxHeight : float) : Rect**

◆ **static function GetRect(minWidth : float, maxWidth : float, minHeight : float, maxHeight : float, style : GUIStyle) : Rect**

◆ **static function GetRect(minWidth : float, maxWidth : float, minHeight : float, maxHeight : float, params options : GUILayerOption[]) : Rect**

◆ **static function GetRect(minWidth : float, maxWidth : float, minHeight : float,**

maxHeight : float, style : GUIStyle, paramas option : GUILayoutOption[]) : Rect

参数

minWidth 传回区域的最小宽度

maxWidth 传回区域的最大宽度

minHeight 传回区域的最小高度

maxHeight 传回区域的最大宽度

style 一个可选的风格。如果指定，风格的 **padding** 将被添加到尺寸并且它的 **margin** 将被用于间距

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **Rect** – 一个矩形区域在两个轴上的尺寸介于 **minWidth** 和 **maxWidth** 之间

描述：从布局系统获取一个可扩展的矩形，矩形的尺寸将在 **min** 和 **max** 值之间。

GUILayout

类

GUILayout 是用于 **UnityGUI** 自动布局的类。

参见：**GUI Layout tutoria**

类方法

- **static function BeginArea(screenRect : Rect) : void**
- **static function BeginArea(screenRect : rect, text : string) : void**
- **static function BeginArea(screenRect : Rect, image : Texture) : void**
- **static function BeginArea(screenRect : Rect, content : GUIContent) : void**
- **static function BeginArea(screenRect : Rect, style : GUIStyle) : void**
- **static function BeginArea(screenRect : Rect, text : string, style : GUIStyle) : void**
- **static function BeginArea(screenRect : Rect, image : Texture, style : GUIStyle) : void**
- **static function BeginArea(screenRect : Rect, content : GUIContent, style : GUIStyle) : void**

参数

text 可选的显示在该区域中的文本

image 可选的显示在该区域中的纹理

content 可选的在该区域顶部显示的文本，图片和提示

style 使用的风格。如果留空，空的 **GUIStyle(GUIStyle.none)**将被使用，给出一个透明的背景。参见：**EndArea**

描述：在屏幕的固定区域开始一个 **GUI** 空间的 **GUILayout** 块。

默认的，任何使用 **GUILayout** 制作的 **GUI** 空间都放置在屏幕的左上角。如果你想在任一区域放置一系列自动布局的控件，使用 **GUILayout.BeginArea** 定义一个新的区域以便自动布局系统可以使用它。

```
function OnGUI()
{
    GUILayout.BeginArea(Rect(200, 200, 100, 100));
    GUILayout.Button("Click me");
GUILayout.Button("Or me");
```

GUILayout.EndArea();

}

在混合 GUILayout 代码是这个函数是非常有用的。必须与 EndArea 调用匹配。BeginArea / EndArea 不能嵌套。

- static function BeginHorizontal(params options : GUILayoutOption[]): void
- static function BeginHorizontal(style : GUIStyle, params options : GUILayoutOption[]): void

参数

style 这个风格用于背景图片和填充值。如果留空，背景是透明的。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这个设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

描述：开始一个水平控件组。

所有在这个元素内部渲染的空间将被一个接一个的水平放置，改组必须调用 EndHorizontal 关闭。

- static function BeginScrollView(scrollPosition : Vector2, params options : GUILayoutOption[]) : Vector2
- static function BeginScrollView(scrollPosition : Vector2, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle, params options : GUILayoutOption[]) : Vector2
- static function BeginScrollView(scrollPosition : Vector2, alwaysShowHorizontal : bool, alwaysShowVertical : bool, params options : GUILayoutOption[]) : Vector2
- static function BeginScrollView(scrollPosition : Vector2, style : GUIStyle) : Vector2
- static function BeginScrollView(scrollPosition : Vector2, alwaysShowHorizontal : bool, alwaysShowVertical : bool, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle, params options : GUILayoutOption[]) : Vector2
- static function BeginScrollView(scrollPosition : Vector2, alwaysShowHorizontal : bool, alwaysShowVertical : bool, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle, background : GUIStyle, params options : GUILayoutOption[]) : Vector2

参数：

scrollPostion 用来显示的位置

alwaysShowHorizontal 可选的参数用来总是显示水平滚动条。如果为假或者留空，它将只在 ScrollView 中的内容比滚动视宽时显示。

alwaysShowVertical 可选的参数用来总是显示垂直滚动条。如果为假或者留空，它将只在 ScrollView 中的内容比滚动视高时显示。

horizontalScrollbar 用于水平滚动条的可选 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalScrollbar。

verticalScrollbar 用于垂直滚动条的可选 GUIStyle。如果不设置，将使用当前 GUISken 的 verticalScrollbar 风格。

返回 Vector2 – 修改过的 scrollPosition。回传这个变量，如下的例子：

描述：开始一个自动布局滚动视。

自动布局滚动视，将使用任何你放置在它们中的内容并正常显示出来。如果他不适合，

将显示滚动条，BeginScrollView 的调用总是与 EndScrollView 的调用匹配。

```
//这个变量对于空间来说就是这个滚动视查看子元素的位置。
var scrollPosition : Vector2;
//显示在滚动视中的字符串，下面两个按钮添加，清除这个字符串。
var longString = “This is a long-ish string”;
function OnGUI()
{
    //开始一个滚动视，所有矩形被自动计算。
    //它将使用任何可用的空间并确保内容排列正确
    //这是小的最后两参数来强制滚动条出现
    scrollPosition = GUILayout.BeginScrollView(scrollPosition, GUILayout.Width(100), GUILayout.Height(100));
    //添加一个简单的标签到滚动视内部。注意
    //滚动条如何与文字换行正确的工作。
    GUILayout.Height(longString);
    //添加一个按钮来消除字符串。这个是在滚动区域内部，因此它
    //也将被滚动。注意按钮如何变窄为垂直滚动条留出空间
    if(GUILayout.Button(“Clear”)
longString = “”;
    //结束我们上面开始的滚动条
    GUILayout.EndScrollView();
    //现在我们在滚动视外边添加一个按钮 – 这将显示在滚动区域的下边
    if(GUILayout.Button(“Add MoreText”))
{
    longString += “\nHere is another line.”;
}
```

- static function Box(params option : GUILayoutOption[]) : void
- static function Box(style : GUIStyle, params option : GUILayoutOption[]) : void

参数

style这个风格将用于背景图片和填充值，如果留空，背景是透明的。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

描述：开始一个垂直控件组。

所有在这个元素内部渲染的空间将被一个接一个地垂直放置。改组必须调用 EndVertical 关闭。

- static function Box(image : Texture, params option : GUILayoutOption[]) : void
- static function Box(text : string, params option : GUILayoutOption[]) : void
- static function Box(contend : GUIContent, params option : GUILayoutOption[]) : void
- static function Box(image : Texture, style : GUIStyle, params option : GUILayoutOption[]) : void
- static function Box(text : string, style : GUIStyle, params option : GUILayoutOption[]) :

void

◆ static function Box(contend : GUIContent, style : GUIStyle, params option : GUILayoutOption[]) : void

参数

text 显示在该 box 上的文本

image 显示在该 box 上的 Texture

content 用于这个 box 的文本，图形和提示

style使用的风格。如果不设置，将使用当前的 GUISkin 的 box 风格。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

描述：制作一个自动布局 box

这将制作一个实心 box，如果你想制作一个包含一些内容的 box，使用一个子组函数的风格参数(BeginHorizontal, BeginVertical, 等等)

◆ static function Button(image : Texture, params options : GUILayoutOption[]) : bool

◆ static function Button(text : string, params options : GUILayoutOption[]) : bool

◆ static function Button(content : GUIContent, params options : GUILayoutOption[]) : bool

◆ static function Button(image : Texture, style : GUIStyle, params options : GUILayoutOption[]) : bool

◆ static function Button(text : string, style : GUIStyle, params options : GUILayoutOption[]) : bool

◆ static function Button(content : GUIContent, style : GUIStyle, params options : GUILayoutOption[]) : bool

参数

text 显示在该按钮上的文本

image 显示在该按钮上的 Texture

content 用于这个按钮的文本，图形和提示。

style使用的风格。如果不设置，将使用当前的 GUISkin 的 button 风格。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回 bool – true 当用户单击按钮时。

描述：制作一个简单的按钮，用户点击它们的时候就会有事情发生。

◆ static functioin EndArea() : void

描述：关闭由 BeginArea 开始的 GUILayout 块

◆ static function EndHorizontal() : void

描述：关闭一个开始于 BeginHorizontal 的组

◆ static functioin EndScrollView() : void

描述：结束由 BeginScrollView 调用开始的滚动视。

◆ **static functioin EndVertical() : void**

描述：关闭一个开始于 **BeginVertical** 的组

◆ **static functioin ExpandHeight(expand : bool) : GUILayoutOption**

描述：传递给一个空间的选项来允许或不允许垂直扩展。

◆ **static functioin ExpandWidth(expand : bool) : GUILayoutOption**

描述：传递给一个空间的选项来允许或不允许水平扩展。

◆ **static functioin FlexibleSpace() : void**

描述：插入一个灵活的空格元素

灵活的空格用来填充一个布局中任何遗漏的空间。

◆ **static functioin Height(height : float) : GUILayoutOption**

描述：传递给空间的选项以便给它一个绝对高度。

◆ **static functioin HorizontalScrollbar(value : float, size : float, leftValue : float, rightValue : float, params options : GUILayoutOption[]) : float**

◆ **static functioin HorizontalScrollbar(value : float, size : float, leftValue : float, rightValue : float, style : GUIStyle, params options : GUILayoutOption[]) : float**

参数

value 在 **min** 和 **max** 之间的位置

size 能看见多大？

leftValue 滚动条左端的值

rightValue 滚动条右端的值

style用于滚动条背景的风格。如果不设置，将使用当前 **GUISkin** 的 **horizontalScrollbar**

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight**

返回 **floar** – 修改后的值。这可以通过拖动这个滚动条，或者单击两端的箭头来改变。

描述：一个水平滚动条。滚动条可以用来滚动文档。大多数情况下，你会使用滚动视代替。

找到额外的元素：

在滚动条两端的按钮将在当前皮肤中搜索“**leftButton**”和“**rightButton**”作为风格，滚动条的滑块（你拖动的东西）将搜索并使用名为“**thumb**”的风格。

//这将使用下面的风格名来决定该按钮的尺寸/位置

//MyScrollbarleftButton - 用于左侧按钮的风格位置

//MyScrollbarrightButton - 用于右侧按钮的风格位置

//MyScrollbarthumb - 用于滑块的风格名称

scrollPos = GUILayout.HorizontalScrollbar(scrollPos, 1, 0, 100, “MyScrollbar”);

◆ **static functioin HorizontalSlider(value : float, leftValue : float, rightValue : float, params options : GUILayoutOption[]) : float**

◆ **static functioin HorizontalSlider(value : float, leftValue : float, rightValue : float, slider : GUIStyle, thumb : GUIStyle, params options : GUILayoutOption[]) : float**

参数

value 滑块显示的值。这个决定可拖动滑块的位置。

leftValue 滑杆左端的值。

rightValue 滑杆右边的值。

slider 用于显示拖动区域的 **GUIStyle**。如果不设置，将使用当前 **GUISkin** 的 **horizontalSlider**。

thumb 用于显示拖动块的 **GUISkin**。如果不设置，将使用当前的 **GUISkin** 的 **horizontalSliderThumb**

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **float** – 被用户设置的值。

描述：一个用户可以拖动的滑杆。可以在 **min** 和 **max** 之间取一个值。

- **static function Label(image : Texture, params options : GUILayoutOption[]) : void**
- **static function Label(text : string, params options : GUILayoutOption[]) : void**
- **static function Label(content : GUIContent, params options : GUILayoutOption[]) : void**
- **static function Label(image : Texture, style : GUIStyle, params options : GUILayoutOption[]) : void**
- **static function Label(text : string, style : GUIStyle, params options : GUILayoutOption[]) : void**
- **static function Label(content: GUIContent, style : GUIStyle, params options : GUILayoutOption[]) : void**

参数

text 显示在该标签上的文本。

image 显示在标签上的 **Texture**。

content 用于这个标签的文本，图形和提示。

style使用的风格。如果不设置。将使用当前 **GUISkin** 的 **label**。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

描述：制作一个自动布局 **label**

标签没有用户交互。不会获取鼠标点击并总是以普通风格渲染。如果你想制作一个可视化响应用户输出的控件，使用一个 **Box** 控件。

- **static function MaxHeight(maxHeight : float) : GUILayoutOption**
  描述：传递给控件的选项来指定一个最大高度。
- **static function MaxWidth(maxWidth : float) : GUILayoutOption**
  描述：传递给控件的选项来指定一个最大宽度。
- **static function MinHeight(minHeight : float) : GUILayoutOption**
  描述：传递给控件的选项来指定一个最小高度。
- **static function MinWidth(minWidth : float) : GUILayoutOption**

描述：传递给控件的选项，来指定一个最小宽度

◆ **static function PasswordField(password : string, maskChar : char, params options : GUILayoutOption[]) : string**

◆ **static function PasswordField(password : string, maskChar : char, maxLength : int, params options : GUILayoutOption[]) : string**

◆ **static function PasswordField(password : string, maskChar : char, style : GUIStyle, params options : GUILayoutOption[]) : string**

◆ **static function PasswordField(password : string, maskChar : char, maxLength : int, style : GUIStyle, params options : GUILayoutOption[]) : string**

参数

password用于编辑的密码。这个函数返回值应该被赋回这个字符串。如下的例子。

maskChar 用来隐藏密码的字符。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style使用的风格。如果不设置，将使用当前 **GUISkin** 的 **textField** 风格。

返回 **string** – 编辑过的密码。

描述：制作一个用户可以输入密码的文本域。

```
var passwordToEdit = "My Password";
function OnGUI()
{// 制作一个密码与来调整 stringToEdit。
        passwordToEdit = GUILayout.PasswordField(passwordToEdit, "*", 25);
}
```

◆ **static function RepeatButton(image : Texture, params options : GUILayoutOption[]) : bool**

◆ **static function RepeatButton(text : string, params options : GUILayoutOption[]) : bool**

◆ **static function RepeatButton(content : GUIContent, params options : GUILayoutOption[]) : bool**

◆ **static function RepeatButton(image : Texture, style : GUIStyle, params options : GUILayoutOption[]) : bool**

◆ **static function RepeatButton(text : string, style : GUIStyle, params options : GUILayoutOption[]) : bool**

◆ **static function RepeatButton(content : GUIContent, style : GUIStyle, params options : GUILayoutOption[]) : bool**

参数

text 显示在该按钮上的文本。

image 显示在该按钮上的 **Texture**。

content 用于这个按钮的文本，图形和提示。

style 使用的风格，如果不设置，将使用当前 **GUISkin** 的 **button** 风格。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **bool - /true/**当用户按住按钮时

描述：制作一个重复按钮，只要用户按住鼠标这个按钮一直返回真。

◆ **static function SeletionGrid(selected : int, texts : string[], xCount : int, params options : GUILayoutOption[]) : int**

◆ **static function SeletionGrid(selected : int, images : Texture[], xCount : int, params options : GUILayoutOption[]) : int**

◆ **static function SeletionGrid(selected : int, contents : GUIContent[], xCount : int, params options : GUILayoutOption[]) : int**

◆ **static function SeletionGrid(selected : int, texts : string[], xCount : int, style : GUIStyle, params options : GUILayoutOption[]) : int**

◆ **static function SeletionGrid(selected : int, images : Texture[], xCount : int, style : GUIStyle, params options : GUILayoutOption[]) : int**

◆ **static function SeletionGrid(selected : int, contents : GUIContent[], xCount : int, style : GUIStyle, params options : GUILayoutOption[]) : int**

参数

selected 选择按钮的索引

texts 显示在按钮的字符串数组。

images 在按钮上纹理数组

contents 用于按钮的文本，图形和提示。

xCount 在水平方向多少个元素，元素将被缩放来适应，除非风格定义了一个 **fixedWidth**，空间高度将由元素的数量决定。

style使用的风格。如果不设置，将使用当前 **GUISkin** 的 **button** 风格。

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **int** – 选择按钮的索引

描述：制作一个选择网格

◆ **static function Space(pixel : float) : void**

描述：在当前布局组中插入一个空格

空格的方向依赖与使用这个命令时当前所在布局组。如果在垂直组，空格是垂直的。

```
function OnGUI()
{
    GUILayout.Button("I'm the first button");
    //在两个按钮间插入 20 像素
    GUILayout.Space(20);

}
    GUILayout.Button("I'm a bit futher down");
}
```

在水平组，**pixels** 将是水平的；

```
function OnGUI()
{
```

```
        //开始水平组以便显示水平空格
        GUILayout.BeginHorizontal("box");
        GUILayout.Button("I'm the first button");
        //在两个按钮间插入 20 像素
        GUILayout.Space(20);
        GUILayout.Button("I'm to the right");
        //结束上面开始的水平组
        GUILayout.EndHorizontal();
    }
```

◆ **static function TextArea(text : string, params options : GUILayoutOption[]) : string**

◆ **static function TextArea(text : string, maxLength : int, params options : GUILayoutOption[]) : string**

◆ **static function TextArea(text : string, style : GUIStyle, params options : GUILayoutOption[]) : string**

◆ **static function TextArea(text : string, maxLength : int, style : GUIStyle, params options : GUILayoutOption[]) : string**

参数

text 用于编辑的文本。这个函数返回值应该被赋回这个字符串。如下的例子。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style使用的风格。如果不设置，则使用当前 **GUISkin** 的 **textField** 风格。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 **string** – 编辑过的字符串。

描述：制作一个多行文本域。这里用户可以编辑这个字符串。

◆ **static function TextField(text : string, params options : GUILayoutOption[]) : string**

◆ **static function TextField (text : string, maxLength : int, params options : GUILayoutOption[]) : string**

◆ **static function TextField (text : string, style : GUIStyle, params options : GUILayoutOption[]) : string**

◆ **static function TextField (text : string, maxLength : int, style : GUIStyle, params options : GUILayoutOption[]) : string**

参数

text 用于编辑的文本。这个函数返回值应该被赋回这个字符串，如下的例子。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style使用的风格。如果不设置，将使用当前的 **GUISkin** 的 **textArea**

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.**

返回 string – 编辑过的字符串。

描述：制作一个单行文本域。这里用户可以编辑这个字符串。

```
var stringToEdit = "Hello, world";
function OnGUI()
{
        //制作一个文本域来调整 stringToEdit
        stringToEdit = GUILayout.TextField(stringToEdit, 25);
}
```

◆ static function Toogle(value : bool, image : Texture, params options : GUILayoutOption[]) : bool

◆ static function Toogle(value : bool, text : string, params options : GUILayoutOption[]) : bool

◆ static function Toogle(value : bool, content : GUIContent, params options : GUILayoutOption[]) : bool

◆ static function Toogle(value : bool, image : Texture, style : GUIStyle, params options : GUILayoutOption[]) : bool

◆ static function Toogle(value : bool, text : string, style : GUIStyle, params options : GUILayoutOption[]) : bool

◆ static function Toogle(value : bool, content : GUIContent, style : GUIStyle, params options : GUILayoutOption[]) : bool

参数

value 按钮是打开或关闭

text 显示在该按钮上的文本

image 显示在该按钮上的 Texture

content 用于这个按钮的文本，图形和提示

style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

返回 bool – 按钮的新值

描述：制作一个 on/off 开关按钮。

◆ static function Toolbar(selected: int, texts : string[], params options : GUILayoutOption[]) : int

◆ static function Toolbar(selected: int, images : Texture[], params options : GUILayoutOption[]) : int

◆ static function Toolbar(selected: int, contents: GUIContent[], params options : GUILayoutOption[]) : int

◆ static function Toolbar(selected: int, texts : string[], style : GUIStyle, params options : GUILayoutOption[]) : int

◆ static function Toolbar(selected: int, image : Texture[],style : GUIStyle, params

options : GUILayoutOption[]) : int

◆ static function Toolbar(selected: int, content : GUIContent[],style : GUIStyle, params options : GUILayoutOption[]) : int

参数

selected 选择按钮的索引

texts 显示在按钮上的字符串数组

images 在按钮上的纹理数组

contents 用于按钮的文本，图形和提示数组

style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

返回 int – 选择按钮的索引

描述：制作一个工具栏

◆ static function VerticalScrollbar(value: float, size : float, topValue : float, buttomValue : float, params options : GUILayoutOption[]) : float

◆ static function VerticalScrollbar(value: float, size : float, topValue : float, buttomValue : float, style : GUIStyle, params options : GUILayoutOption[]) : float

参数

value 在 min 和 max 之间的位置

size 能看见多大？

topValue 滚动条顶端的值

bottomValue 滚动条底端的值

style 用于滚动条背景的风格。如果不设置，将使用当前 GUISkin 的 horizontalScrollbar。

options 一个可选的布局选项的列表。它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

返回 float – 修改后的值。这可以通过拖动这个滚动条，或者单击两端的箭头来改变。

描述：制作一个垂直滚动条。滚动条可以用来滚动文档。大多数情况下，你会使用 scollView 代替。

找到额外的元素：

在滚动条两端的按钮将在当前皮肤中搜索“upbutton”和“downbutton”作为风格。滚动条的滑块（你拖动的东西）将搜索并使用名为 thumb 的风格。

//这将使用下面的风格名来决定该按钮的尺寸/位置

//MyVerticalScrollbarupbutton – 用于顶端按钮的风格名称

//MyVerticalScrollbardownbutton – 用于底端按钮的风格名称

//MyScrollbarthumb – 用于滑块的风格名称

scrollPos = GUILayout.HorizontalScrollbar(scrollPos, 1, 0, 100, “MyVerticalScrollbar”);

◆ static function VerticalSlider(value : floar, leftValue : float, rightValue : float, params

options : GUILayoutOption[]) : float

◆ static function VerticalSlider(value : floar, leftValue : float, rightValue : float, slider : GUIStyle, thumb : GUIStyle, params options : GUILayoutOption[]) : float

参数

value 滑杆显示的值。这个决定可拖动滑块的位置。

topValue 滑杆顶端的值。

downValue 滑杆底端的值。

slider 用于显示拖动区域的 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalSlider。

thumb 用于显示拖动块的 GUIStyle。如果不设置，将使用当前的 GUISkin 的 horizontalSliderThumb。

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

◆ static function Width(width : float) : GUILayoutOption

描述：传递给控件的选项以便给它一个绝对宽度

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, text : string) : Rect

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, image : Texture) : Rect

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, content : GUIContent) : Rect

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, text : string, style : GUIStyle) : Rect

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, image : Texture, style : GUIStyle) : Rect

◆ static function Window(id : int, screenRect : Rect, func : GUIWindowFunction, content : GUIContent, style : GUIStyle) : Rect

参数

id 用于每个窗口唯一的 ID。这是用于接口的 ID。

clientRect 屏幕上用于窗口的矩形区域。布局系统将试图使窗体在他内部。如果不能，它将调整矩形去适应它

func 在窗体内部创建 GUI 的函数。这个函数必须使用一个函数 – 当前创建 GUI 的窗体 id。

image 用于在标题栏上显示图片的 Texture。

content 用于这个窗口的文本，图形和提示。

style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth,

GUILayout.ExpandHeight.

返回 Rect – 窗口位于举行位置。这个可能与你传入的一个具有不同位置和尺寸。

描述：制作一个弹出窗口。它的内容是自动布局的。

窗口浮动在普通 GUI 控件之上。具有单击激活的特点并可以有选择的随意被用户拖动。不像其他的控件，你需要传递给他们一个独立的功能并放置在窗口中这儿是一个小例子来帮助你开始：

```
var windowRect = Rect(20, 20, 120, 50);

function OnGUI()
{
    //注册窗口，注意第三个参数
    windowRect = GUILayout.Window(0, windowsRect, DoMyWindow, “My Window”);
}

//制作窗口内容
function DoMyWindow(windowed : int)
{
    //这个按钮将调整以适应物体
    if(GUILayout.Button(“Hello World”))
        print(“Get a click”);
}
```

你传入的客户区域只是作为一个参考，为了对窗口使用额外的限制。闯入一些额外的布局选项。用在这里的一个将覆盖尺寸的计算。这是一个简单的例子。

```
var windowRect = Rect(20, 20, 120, 50);

function OnGUI()
{
    //注册窗口，这里我们指示布局系统必须使窗体为 100 像素宽。
    windowRect = GUILayout.Window(0, windowRect, DoMyWindow, “My Window”, GUILayout.width(100));
}

//制作窗体内容
function DoMyWindow(windowID : int)
{
    //这个按钮大小不能适应这个窗体
//通常，这个窗体将被扩展以便适应这个按钮。但是由于 GUILayout.Width 将只允//许窗体为 100 像素宽
if(GUILayout.Button(“Please click me a lot”))
    print(“Get a click”);

}
```

GUISettings

类

用于 GUI 如何表现的通用设置。

这些被所有在 GUISkin 中的元素共享。

变量

- var cursorColor : Color

  描述：文本域中光标的颜色。

- var cursorFlashSpeed : float

  描述：文本域指示的闪动速度。

这个是闪动/秒。如果设置为 0，闪动将被禁用。如果设置为-1，闪动速度将匹配系统的默认速度。

- var doubleClickSelectsWord : bool

  描述：文本域中双击是否选择单词。

- var selectionColor : Color

  描述：文本域中选择矩形的颜色。

- var tripleClickSelectsLine : bool

  描述：文本域中点击鼠标三次是否选择整行文本。

GUIStyleState

类

为 GUIStyle 物体使用的给定状态的一个特定值。

GUIStyle 包含用于显示 GUI 元素的所有值

变量

- var background : Texture2D

  描述：在这个状态中被 GUI 元素使用的背景图片。

- var textColor : Color

  描述：在这个状态中被 GUI 元素使用的文本颜色。

GUIStyle

类

GUI 元素的风格位置

这个类包含一个 GUI 元素如何被渲染的所有信息。它包含用于字体，图标位置，背景图标，和间距的信息。它不包含关于它包含什么的信息 – 仅仅定义用这个风格渲染的文本如何被显示。它不定义这个元素可以发生什么交互，但是定义了用于交互的显示设置。

一个 GUIStyle 的设置。这是模仿一个 CSS 样式。它包含下列事项的设置：

背景图片

这些被渲染在空间后面。不同的图像可以被指定为正常显示。当用户将鼠标放在元素上时的显示，当使用者按下选择 – 以及当元素被打开，如触发按钮，下面的这些都被称为风格的状态。参见：normal, hover, active, onNormal, onHover, onActive – 这些包含每个状态的背景图片，文本颜色属性。

文本渲染

这个风格都可以定义一个字体渲染，以及文本对齐，换行和剪裁的设置，它也定了该风格元素不同状态的文本颜色参考 ：font, alignment, wordWrap, normal, hover, active, onHover, onActive

图标位置

GUIStyle 可以渲染文本，图标或者两者兼而有之。GUIStyle 定义渲染时这两者间的相关位置（这可以强制它只显示他们其中之一）。参见：imagePosition

尺寸和间距选项

GUIStyle 包含填充，边缘和边界。这些不严格地对应类似命名的 CSS 属性。一个 GUIStyle 可以选择的定义一个固定的宽度和高度。参见：margin, padding, border, fixedWidth, fixedHieght

变量

- var active : GUIStyleState
  描述：空间被按下时的渲染设置。
- var alignment : TextAnchor
  描述：文本对齐
- var border : RectOffset
  描述：所有背景图片的边界。
  这个对应于 GUITexture 的边界设置。它只影响渲染的背景图像，不影响定位。
- var clipOffset : Vector2
  描述：用于该 GUIStyle 内容的剪裁偏移。
- var clipping : TextClipping
  描述：当渲染的内容相对于给定的区域太大时做什么
- var contentOffset : Vector2
  描述：用于该 GUIStyle 内容的像素偏移。
- var fixedHeight : float
  描述：如果非 0，任何用这个风格渲染的 GUI 元素将有这里指定的高度。
- var fixedWidth : float
  描述：如果非 0，任何用这个风格渲染的 GUI 元素将有这里指定的宽度。
- var focused : GUIStyleState
  描述：元素获得键盘焦点时的渲染设置。
- var font : Font
  描述：用于渲染的字体。如果 null，当前的 GUISkin 的默认字体将被使用。
- var hover : GUIStyleState
  描述：鼠标悬停在控件时的渲染设置。
- var imagePosition : ImagePosition
  描述：GUIContent 的图片和文本如何组合。
- var lineHeight : float
  描述：这个风格一行文本的高度。以像素为单位。（只读）
- var margin : RectOffset
  描述：以这种风格渲染的元素和任何其他 GUI 元素之间的边界。

这个只影响自动布局。（参见 : GUILayout）

- var name : string
  描述：GUIStyle 的名称。用来基于名称获取它们。
- var normal : GUIStyleState
  描述：组建正常显示是渲染设置。
- var onActive : GUIStyleState
  描述：元素被打开并具有键盘焦点时的渲染设置。
- var onFocused : GUIStyleState
  描述：元素被打开并被按下时的渲染设置。
- var onHover : GUIStyleState
  描述：控件被打开并且鼠标悬停在它上面时的渲染设置

- **var onNormal : GUIStyleState**

  描述：控件被打开时的渲染设置。

- **var overflow : RectOffset**

  描述：添加到背景图片的额外间距。

  这可用于如果你的图片要投射一个阴影，并且向扩展背景图片超出使用这个风格的制定 GUI 元素的矩形时。

- **var padding : RectOffset**

  描述：从 GUIStyle 边界到内容开始的间距

- **var stretchHeight : bool**

  描述：这个风格的 GUI 元素可以被垂直拉伸以便更好的布局么？

- **var stretchWidth : bool**

  描述：这个风格的 GUI 元素可以被水平拉伸以便更好的布局么？

- **var wordWrap : bool**

  描述：文本换行？

  这将导致任何包含的文本被换行以便适应控件的宽度。

构造函数

- **static function GUIStyle() : GUIStyle**

  描述：

- **static function GUIStyle(other：GUIStyle) : GUIStyle**

  描述：

函数

- **function CalcHeight(content : GUIContent, width : float) : float**

  描述：当渲染 content 并制定 width 时这个元素的高度。

- **function CalcMinMaxHeight (content : GUIContent, outminWidth : float, outmaxWidth : float) : void**

  描述：计算以这个风格渲染的 content 的最大最小宽度。

  被 GUILayout 使用来正确处理换行。

- **function CalcScreenSize(contentSize : Vector2) : Vector2**

  描述：计算用这个风格格式化的元素的尺寸，和一个给定的内容空格。

- **function CalcSize(content : GUIContent) : Vector2**

  描述：如果内容使用这个风格渲染，计算某些内容的尺寸。

  这个函数不考虑换行。要做到这点，你需要确定宽度，然后打调用 CalcHeight 分配计算出 wordwrapped 高度。

- **function Draw(position : Rect, isHover: bool, isActive : bool, on : bool, hasKeyboardFocus : bool) : void**

  描述：

- **function Draw(position : Rect, text : string, isHover: bool, isActive : bool, on : bool, hasKeyboardFocus : bool) : void**

  描述：用 GUIStyle 绘制一个文本字符串。

- **function Draw(position : Rect, image : Texture, isHover: bool, isActive : bool, on : bool, hasKeyboardFocus : bool) : void**

  描述：用 GUIStyle 绘制图片。如果图片太大，它将被缩小。

- **function Draw(position : Rect, content : GUIContent, isHover: bool, isActive : bool, on : bool, hasKeyboardFocus : bool) : void**

描述：用 GUIStyle 绘制文本和图片。如果图片太大，它将被缩小。

- **function Draw(position : Rect, content : GUIContent, controlID : int, on : bool = false) : void**

描述：GUI 代码中使用的主要绘制函数。

- **function DrawCursor(position : Rect, content : GUIContent, controlID : int, Character : int) : void**

描述：用选定的内容绘制这个 GUIStyle

- **function DrawWithTextSelection(position : Rect, content : GUIContent, controlID : int, firstSelectedCharacter : int, lastSelectedCharacter : int) : void**

描述：用选定的内容绘制这个 GUIStyle

- **function GetCursorPixelPosition(position : Rect, content : GUIContent, cursorStringIndex : int) : Vector2**

描述：获取给定字符串索引的像素位置。

- **function GetCursorStringIndex(postion : Rect, content : GUIContent, cursorPixelPosition : Vector2) : int**

描述：当用户在 cursorPixelPosition 处点击时获取光标位置（索引到内容文本）这部计算内容中任何图片。

类变量

- **static var none : GUIStyle**

描述：空 GUIStyle 的快捷方式。

这个风格不包含装饰而仅仅以缺省字体渲染所有东西。

```
function OnGUI()
{        //制作一个没有装饰的按钮
    GUI.Button("I'm very bare", GUIStyle.none);
}
```

类方法

- **static implicit function GUIStyle(str : string) : GUIStyle**

描述：从当前皮肤获取一个名为 GUI 风格。

GUIUtility

类

用于制作新 GUI 空间的工具类。

除非从头开始创建自己的 GUI 控件，否则你不需要使用这些函数。

类变量

- **static var hotControl : int**

描述：当前具有特点的控件 controlID。

热点空间是临时激活的一个控件。当用于在一个按钮上按下鼠标时，它变为热点。当一个控件具有热点时，其他控件不允许响应鼠标事件。一旦用户鼠标示范，这个控件设置 hotControl 为 0 以便表明其他控件可以响应用户输入。

- **static var keyboardControl : int**

描述：具有键盘焦点控件的 controlID。

类方法

- **static function GetControlID(focus : FocusType) : int**
- **static function GetControlID(hint : int, focus : FocusType) : int**

- **static function GetControlID(content : GUIContent, focus : FocusType) : int**
  描述：为一个控件获取唯一 ID。
- **static function GetControlID(focus : FocusType, position : Rect) : int**
- **static function GetControlID(hint : int, focus : FocusType, position : Rect) : int**
- **static function GetControlID(content : GUIContent, focus : FocusType, position : Rect) : int**

描述：为一个控件获取唯一 ID。

- **static function GetStateObject(t : type, controlID : int) : object**
  描述：从一个 controlID 获取一个状态对象。
  这将返回一个可重用的状态对象，它的 controlID 是唯一的。如果没有，一个新的将被创建并添加到 ControlID。
- **static function GUIToScreenPoint(guiPoint : Vector2) : Vector2**
  描述：将一个点从 GUI 位置转化为屏幕空间。
  参见：GUIUtility, ScreenToGUIPoint
- **static function MoveNextAndScroll(forward: bool) : void**
  描述：只允许从 OnGUI 内部调出 GUI 函数
- **static function QueryStateObject(t: Type, controlID : int) : object**
  描述：从一个 controlID 获取一个存在的状态物体。
  这个返回一个可回收的状态物体。这个物体具有唯一的 controlID。如果没有，这个函数将返回 null。
- **static function RotateAroundPivot(angle: float, pivotPoint : Vector2) : void**
  描述：使 GUI 围绕一个点旋转的辅助函数。
  修改 GUI.matrix 来绕着 pivotPoint 旋转所有的 GUI 元素 angel 度。
  参见：GUI.matrix, ScaleAroundPivot
- **static function ScaleAroundPivot(scale: Vector2, pivotPoint: Vector2) : void**
  描述：使 GUI 围绕一个点缩放的辅助函数。
  修改 GUI,.matrix 来绕着 pivotPoint 缩放所有的 GUI 元素 angel 度。
  参见：GUI.matrix, RotateAroundPivot
- **static function ScreenToGUIPoint(screenPoint : Vector2) : object**
  描述：将一个点从屏幕空间转化为 GUI 位置。
  用于反转计算 GUIToScreenPoint 的值。
  参见：GUIUtility.GUIToScreenPoint。

GUI

类

GUI 类是 Unity GUI 的手工定位接口
参见：GUI tutorial

类变量

- **static var backgroundColor : Color**
  描述：全局的修改由 GUI 渲染的所有元素的背景颜色。
  这个获取多个 color。
  参见：contentColor, color
- **static var changed : bool**
  描述：有任何控件的输入数据改变吗？
- **static var color : Color**

描述：全局的修改 GUI 颜色。

这将影响背景和文本颜色。

参见：backgroundColor, contentColor

- static var contentColor : Color

描述：修改由 GUI 渲染的所有文本的颜色。

这个获取多个 color。

参见：backgroundColor : Color

- static var depth : int

描述：当前执行的 GUI 行为的排序深度

当你有不同的脚本同时运行时。设置这个来决定顺序。

- static var enabled : bool

描述：GUI 启用了？

设置这个值为假将禁用所有 GUI 交互。所有的控件将以半透明方式绘制。并将不响应用户输入。

```
//这个值跟踪扩展的选项是否可以被打开。
var allOptions = true;
//两个扩展选项
var extended1 = true;
var extended2 = true;
function OnGUI()
{
    //制作一个开关控件以便允许用户编辑扩展的选项
    allOptions = GUI.Toogle(Rect(0, 0, 150, 20), allOptions, “Edit All Options”);

    //将它的值赋给 GUIEnabled – 如果上面的复选框被禁用
    //这些 GUI 元素将
    GUIenabled = allOptions

    //这两个控件只在上面的按钮为 On 时启用。
    extended1 = GUI.Toogle(Rect(20, 20, 130, 20), extended1, “Extended Option 1”);
    extended2 = GUI.Toogle(Rect(20, 40, 130, 30), extended2, “Extended Option 2”);

    //使用条件语句，以使 GUI 代码可以再次启用
    GUI.enabled = true;
    //制作一个 OK 按钮
    if(GUI.Button(Rect(0, 60, 150, 20), “OK”))
print(“user clicked ok”);
}
```

- static var matrix : Matrix4x4

描述：GUI 变换矩阵

- static var skin : GUISkin

描述：使用的全局皮肤

你可以在任何时候设置这个来改变 GUI 的外观。如果设置为 null，这个皮肤将使用默认的 Unity 皮肤。

◆ **static var tooltip : string**

描述：鼠标移动到空间上的提示信息（只读）。

创建 GUI 空间是。你可以给他传递一个提示。这可以通过改变内容参数来制作一个自定义 GUIContent 物体，而不是仅仅传递一个字符串。

但鼠标经过带有提示性的控件时，它设置全局的 GUI.tooltip 值为传入得知。在 OnGUI 代码的末端，你可以制作一个标签来显示 GUI.tooltip 的值。

```
function OnGUI()
{
    //制作一个按钮，它使用自定义 GUI.Content 参数来传递提示。
    GUI.Button(Rect(10, 10, 100, 20), GUIContent("Click me", "This is the tooltip"));

    //显示鼠标指向或具有键盘焦点的控件提示
    GUI.Button(Rect(10, 55, 100, 20), "No tooltip here");
}
```

你可以使用元素的次序来创建’层次化的’提示

```
function OnGUI()
{
    //这个 box 比随后的许多元素大，并且它有一个提示。
    GUI.Box(Rect(5, 35, 110, 75), GUIContent("Box", "this box has a tooltip"));

    这个按钮在 box 内部，但是没有提示，因此它不会覆盖这个 box 的提示。
    GUI.Button(Rect(10, 55, 100, 20), "No tooltip here");

    这个按钮在 box 内部，并且有一个提示，因此它会覆盖这个 box 的提示。
GUI.Button(Rect(10, 80, 100, 20), GUIContent("I have a tooltip", "This button overrides the box"));
//最后，显示来自鼠标指向或具有键盘焦点的提示
GUI.Label(Rect(10, 40, 100, 40), GUI.tooltip);
}
```

Tooltip 也能用来实现一个 OnMouseOver/OnMouseOut 消息系统：

```
var lastTooltip = "";
function OnGUI()
{
    GUILayout.Button(GUIContent("Play Game", "Button1"));
    GUILayout.Button(GUIContent("Quit", "Button2"));

    if(Event.current.type == EventType.repaint && GUI.tooltip! = lastTooltip)
{
        if(lastTooltip != "")
SendMessage(lastTooltip + "OnMouseOut", SendMessageOptions, DontRequireReceiver);
        if(GUI.tooltip != "")
SendMessage(GUI.tooltip + "OnMouseOut", SendMessageOptions, DontRequireReceiver);
        lastTool\tip = GUI.tooltip;
    }
```

}

类方法

- static function BeginGroup(position : Rect) : void
- static function BeginGroup(position : Rect, text : string) : void
- static function BeginGroup(position : Rect, image : Texture) : void
- static function BeginGroup(position : Rect, content : GUIContent) : void
- static function BeginGroup(position : Rect, style : GUIStyle) : void
- static function BeginGroup(position : Rect, text : string, style : GUIStyle) : void
- static function BeginGroup(position : Rect, image : Texture, style : GUIStyle) : void
- static function BeginGroup(position : Rect, content : GUIContent, style : GUIStyle) : void

参数

position 屏幕用于组的矩形区域。

text 显示在该组上的文本。

image 显示在该组上的 Texture。

content 用于这个组的文本，图形和提示。如果提供，任何鼠标点击被组捕获，并且如果没有设置，不会渲染背景，和传递鼠标点击。

style用于背景的风格。

描述：

开始组，必须与 EndGroup 调用匹配。

当你开始一个组时，用于 GUI 控件的坐标系统被设置为(0, 0)是组的左上角。所有控件被附加到组。组可以嵌套 - 如果使用，子被附加到他们的父。

当你在屏幕上移动一组 GUI 元素是这是非常有用的。一个普通的用法是设计你的菜单以适合一个特定的屏幕的尺寸。然后在更大的显示器上居中显示 GUI。

```
function OnGUI()
{
    //约束所有的绘图在屏幕中心 800*600 的区域
    GUI.BeginGroup(new Rect(Screen.width / 2 – 400, Screen.height / 2 – 300, 800, 600));
    //在由 BeginGroup 定义的新坐标空间中绘制一个 box
    //注意，现在（0，0）已经被移动了
    GUI.Box(new Rect(0, 0, 800, 600), “This box is new centered! – Here you would put your main menu”);
    //需要用一个 EndGroup 来匹配所有的 BeginGroup 调用。
    GUI.EndGroup();
}
```

参见：matrix, BeginScrollView

- static function BeginScrollView(position : Rect, scrollPostion: Vector2, viewRect : Rect) : Vector2
- static function BeginScrollView(position : Rect, scrollPostion: Vector2, viewRect : Rect, alwaysShowHorizontal : bool, alwaysShowVertical : bool) : Vector2
- static function BeginScrollView(position : Rect, scrollPostion: Vector2, viewRect : Rect, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle) : Vector2
- static function BeginScrollView(position : Rect, scrollPostion: Vector2, viewRect : Rect,

alwaysShowHorizontal : bool, alwaysShowVertical : bool, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle) : Vector2

参数

position 屏幕上用了 ScrollView 的矩形区域

scrollPosition 用来显示的位置。

viewRect 用在滚动视内部的矩形。

alwaysShowHorizontal 可选的参数用来总是显示水平滚动条。如果为假或不设置，它只在 clientRect 比 position 宽的时候显示

alwaysShowVertical 可选的参数用来总是显示垂直滚动条。如果为假或不设置，它只在 clientRect 比 position 长的时候显示。

horizontalScrollbar 用于水平滚动条的可选 GUIStyle，如果不设置，将使用当前 GUISkin 的 horizontalScrollbar.

verticalScrollbar 用于水平滚动条的可选 GUIStyle，如果不设置，将使用当前 GUISkin 的 verticalScrollbar.

返回 Vector2 – 修改过的 scrollPosition 回传这个变量。如下的例子。

描述：在你的 GUI 中开始滚动视。

ScrollViews 让你在屏幕上制作一个较小的区域。使用放置在 ScrollView 边上的滚动条来查看一个较大的区域。

```
//滚动视口的位置
var smallPosition = Vector2.zero;
function OnGUI()
{
    //一个绝对位置的例子。制作一个具有较大区域的滚动视。
    //并将它放置在一个小的矩形中。
    scrollPosition = GUI.BeginScrollView(Rect(10, 300, 100, 100), scrollPosition, Rect(0, 0, 220, 200));
    //制作四个按钮 – 每个角上有一个，由 BeginScrollView 最后一个参数定义的坐标系统。
GUI.Button(Rect(0, 0, 100, 20), “Top-left”);
GUI.Button(Rect(0, 0, 100, 20), “Top-right”);
GUI.Button(Rect(0, 0, 100, 20), “Bottom-left”);
GUI.Button(Rect(0, 0, 100, 20), “Bottom-right”);
//结束前面开始滚动视
GUI.EndScrollView();
}
```

- static function Box(position : Rect, text : string) : void
- static function Box(position : Rect, image : Texture) : void
- static function Box(position : Rect, content : GUIContent) : void
- static function Box(position : Rect, text : string, style : GUIStyle) : void
- static function Box(position : Rect, image : Texture, style : GUIStyle) : void
- static function Box(position : Rect, content : GUIContent, style : GUIStyle) : void

参数

position 屏幕上用于 box 的矩形区域

text 显示在该 box 上的文本

image 显示在该 box 上的 Texture
content 用于这个 box 的文本，图形和提示
style使用的风格。如果不设置，将使用当前 GUISkin 的 box 风格。
描述：制作一个图形 box。

- static function BringWindowToBack(windowID: int) : void

参数
windowID　　在 window 调用中创建窗体时使用的唯一标识。
描述：将特定的窗口放到浮动窗口的后面。

- static function BringWindowToFront(windowID: int) : void

参数
windowID　　在 Window 调用中创建窗体时使用的唯一标识。
描述：将特定的窗口放到浮动窗口的前面。

- static function Button(position : Rect, text : string) : bool
- static function Button(position : Rect, image : Texture) : bool
- static function Button(position : Rect, content : GUIContent) : bool
- static function Button(position : Rect, text : string, style : GUIStyle) : bool
- static function Button(position : Rect, image : Texture, style : GUIStyle) : bool
- static function Button(position : Rect, content : GUIContent, style : GUIStyle) : bool

参数
position 屏幕上用于按钮的矩形区域
text 显示在该按钮上的文本
image 显示在该按钮上的 Texture
content 用于这个按钮的文本，图形和提示
style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。
返回 bool - /true/当用户单击按钮时
描述：制作一个简单的按钮，用户点击它们的时候，就会有些事情发生。

- static function DragWindow(position : Rect) : void

参数
position 可以拖动的窗口部分。这个被附加到实际窗口。
描述：让窗口可拖动。
插入这个函数的在你的窗口代码中使窗口可拖动。

```
var windowRect = Rect(20, 20, 120, 50);

function OnGUI()
{
    //注册窗体
    windowRect = GUI.Window(0, windowRect, DoMyWindow, “My Window");
}
//制作窗体内容
function DoMyWindows(windowID : int)
{
    //制作一个非常长的矩形，高 20 像素。
    //这将使得窗口可以调整大小，通过顶部的标题栏 – 不管它有多宽。
    GUI.DragWindow(Rect(0, 0, 10000, 20));
```

}

- **static function DragWindow() : void**

如果你想将整个窗体背景作为一个可拖动区域，不使用参数并将 DragWindow 放置到整体函数的最后。

这意味着任何其他类型的控件将首先被处理并且拖动将只在没有其他控件获得焦点时使用。

- **static function DrawTexture(position : Rect, image : Texture, scaleMode : ScaleMode = stretchToFill, alphaBlend : bool = true, imageAspect : float = 0) : void**

参数

position 在屏幕上绘制一个内部包含有纹理的矩形。

image 需要被绘制的纹理。

scaleMode 定义了当矩形的长宽比和内部图像的长宽比不同时如何缩放图像。

alphaBlend 定义了 alpha 值是否参与图像的混合（默认为真）。如果为假，图像将会被绘制并显示。

imageAspect 源图像的的长宽比。如果是 0（默认值），则使用图像自身的长宽比。

描述：在一个矩形内部绘制一个纹理。

参见：GUI.color, GUI.contentColor

- **static function EndGroup() : void**

描述：结束一个组

参见：BeginGroup()。

- **static function EndScrollView : void**

描述：结束一个由 BeginScrollView 开始的滚动视。

- **static function FocusControl(name : string) : void**

描述：把键盘焦点转移到定义的控件。

参见：SetNextControlName, GetNameOfFocusedControl。

```
var username = "username";
function OnGUI ()
{       // 设置一个文本域中内部名称
        GUI.SetNextControlName ("MyTextField");
        // Make the actual text field.
        username = GUI.TextField (Rect (10,10,100,20), username);
        // If the user presses this button, keyboard focus will move.
        if (GUI.Button (Rect (10,40,80,20), "Move Focus"))
        GUI.FocusControl ("MyTextField");
}
```

- **static function FocusWindow(windowID : int) : void**

参数：

windowID 当调用 Window 而创建的窗口的标识符。

描述：使一个窗口被激活。

参见：GUI.UnfocusWindow

- **static function GetNameOfFocusedControl() : string**

描述：返回当前激活的控件的名字。控件的名字是由 SetNextControlName 函数创

建的。当有名字的控件被激活时，函数返回它的名字；否则的话返回一个空字符串。

```
var login = "username";
var login2 = "no action here";
function OnGUI ()
{
    GUI.SetNextControlName ("user");
    login = GUI.TextField (Rect (10,10,130,20), login);
    login2 = GUI.TextField (Rect (10,40,130,20), login2);
    if (Event.current.Equals (Event.KeyboardEvent ("return")) &&
GUI.GetNameOfFocusedControl () == "user")
        {
            Debug.Log ("Login");
         }
    if (GUI.Button (new Rect (150,10,50,20), "Login"))
        Debug.Log ("Login");
}
        参见：SetNextControlName, FocusControl
        login2 = GUI.TextField(new Rect(10, 40, 130, 20), login2);
if(Event.current.Equals(Event.KeyboardEvent.("return"))&&GUI.GetNameOfFocusedContr
ol() == "user")
    Debug.log("Login");
if(GUI.Button(new Rect(150, 10, 50, 20), "Login"))
    Debug.log("Login");
}
    参见：SetNextControlName
```

◆ static function HorizontalScrollbar(position : Rect, value : float, size : float, leftValue : float, rightValue : float) : bool

◆ static function HorizontalScrollbar(position : Rect, value : float, size : float, leftValue : float, rightValue : float, style : GUIStyle) : bool

参数

position 屏幕上用于滚动条的矩形区域

value 在 min 和 max 之间的位置

size 能看见多大？

leftValue 滚动条左边的值

rightValue 滚动条右边的值

style用于滚动条背景的风格。如果不设置，将使用当前 GUISkin 的 horizontalScrollbar。

返回 float – 修改后的值。这可以通过拖动这个滚动条，或者单击两端的箭头来改变。

描述：制作一个水平滚动条。滚动条可以用来滚动文档。大多数情况下，你会使用 scrollViews 代替。

找到额外的元素：

在滚动条两端的按钮将在当前皮肤中搜索”leftbutton”和”rightbutton”作为风格。滚动条的滑块（你拖动的东西）将搜索并使用名为“thumb”的风格。

//这将使用下面的风格名来决定该按钮的尺寸/位置

//MyScrollbarrightbutton – 用于左侧按钮的风格名称

//MyScrollbarleftbutton – 用于右侧按钮的风格名称

//MyScrollbarthumb – 用于滑块的风格名称

scrollPos = HoriztontalScrollbar(Rect(0, 0, 100, 20), scrollPos, 1, 0, 100, "My Scrollbar");

- static function HorizontalSlider(position : Rect, value : float, leftValue : float, rightValue : float) : float
- static function HorizontalSlider(position : Rect, value : float, leftValue : float, rightValue : float, slider : GUIStyle, thumb : GUIStyle) : float

参数

position 屏幕上用于滑杆的矩形区域

value 滑杆显示的值。这个决定可拖动的位置。

leftValue 滑杆左边的值

rightValue 滑杆右边的值

slider 用于显示拖动区域的 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalSlider。

thumb 用于显示拖动块的 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalSliderThumb。

返回 float – 被用户设置的值。

描述：一个用户可以拖动的滑杆。可以在 min 和 max 只见改变一个值。

- static function Label(position : Rect, text : string) : void
- static function Label(position : Rect, image : Texture) : void
- static function Label(position : Rect, content : GUIContent) : void
- static function Label(position : Rect, text : string, style : GUIStyle) : void
- static function Label(position : Rect, image : Texture, style : GUIStyle) : void
- static function Label(position : Rect, content : GUIContent, style : GUIStyle) : void

参数

position 屏幕上用于标签的矩形区域

text 显示在该标签上的文本

image 显示在该标签上的 Texture

content 用于这个标签的文本，图形和提示

style使用的风格。如果不设置，将使用当前 GUISkin 的 label

描述：在屏幕上制作一个文本或者纹理标签.

标签没有用户交互，不会获取鼠标点击并总是以普通风格渲染。如果你想制作一个可视化响应用户输入的控件，使用一个 Box 控件。

例如：绘制一个传统的 Hello world 字符串

```
function OnGUI
{
    GUI.Label(Rect(10, 10, 100, 20), "Hello world");
}
```

例如：在屏幕上绘制一个纹理。标签也用于显示纹理，而不仅是字符串。简单传递一个纹理。

```
var textureToDisplay : Texture2D;
```

```
function OnGUI()
{
    GUI.Label(Rect(10, 40, textureToDisplay.width, textureToDiplay.height), textureToDisplay);
}
```

- static function PasswordField(position : Rect, password : string, markChar : char) : string
- static function PasswordField(position : Rect, password : string, markChar : char, maxLength : int) : string
- static function PasswordField(position : Rect, password : string, markChar : char, style : GUIStyle) : string
- static function PasswordField(position : Rect, password : string, markChar : char, markChar : char, style : GUIStyle) : string

参数

position 屏幕上用于文本的矩形区域

password 用于编辑的密码。这个函数返回值应该被赋回这个字符串。如下的例子。

markChar 用来隐藏密码的字符。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style 使用的风格。不过不设置，将那个使用当前 GUISkin 的 textField 风格。

返回 string – 编辑过的密码

描述：制作一个用户可以输入密码的文本域。

```
var passwordToEdit = "My Password";

function OnGUI()
{
    //制作一个文本来调整 stringToEdit。
    passwordToEdit = GUI.PasswordField(Rect(10, 10, 200, 20), passwordToEdit, "*", 25);
}
```

- static function RepeatButton(position : Rect, text : string) : bool
- static function RepeatButton(position : Rect, image : Texture) : bool
- static function RepeatButton(position : Rect, content : GUIContent) : bool
- static function RepeatButton(position : Rect, text : string, style : GUIStyle) : bool
- static function RepeatButton(position : Rect, image : Texture, style : GUIStyle) : bool
- static function RepeatButton(position : Rect, content : GUIContent, style : GUIStyle) : bool

参数

position 屏幕上用于按钮的矩形区域

text 显示在该按钮上的文本

image 显示在该按钮上的 Texture

content 用于这个按钮的文本，图形和提示

style 使用的风格。不过不设置，将那个使用当前 GUISkin 的 button 风格。

返回 bool - /true/当用户单击按钮时。

描述：制作一个按钮。当用户按住它时一直是激活的。

◆ static function ScrollTo(position : Rect) : void

描述：滚动所有包含在 scrollview 中的数据以便 position 可见。

◆ static function SelectionGrid (position : Rect, selected : int, texts : string[], xCount : int) : int

◆ static function SelectionGrid (position : Rect, selected : int, images : Texture[], xCount : int) : int

◆ static function SelectionGrid (position : Rect, selected : int, content : GUIContent[], xCount : int) : int

◆ static function SelectionGrid (position : Rect, selected : int, texts : string[], xCount : int, style : GUIStyle) : int

◆ static function SelectionGrid (position : Rect, selected : int, images : Texture[], xCount : int, style : GUIStyle) : int

◆ static function SelectionGrid (position : Rect, selected : int, content : GUIContent[], xCount : int, style : GUIStyle) : int

参数

position 屏幕上用于网格的矩形区域。

selected 选择的网格按钮的索引

texts 显示在网格按钮上的字符串数组

images 显示在网格按钮上的纹理数组

contents 用于这个网格按钮的文本，图形和提示数组

xCount 在水平方向有多少个像素。空间将被缩放来适应，除非风格定义了一个 fixWidth。

style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。

返回 int – 选择按钮的索引。

描述：制作一个按钮网络。

◆ static function SetNextControlName(name : string) : void

描述：设置下一个控件的名称。

这是接下来的控件被注册。

◆ static function TextArea(position : Rect, text : string) : string

◆ static function TextArea(position : Rect, text : string, maxLength : int) : string

◆ static function TextArea(position : Rect, text : string, style : GUIStyle) : string

◆ static function TextArea(position : Rect, text : string, maxLength : int, style : GUIStyle) : string

参数

position 屏幕上用于文本的矩形区域

text 用于编辑的文本。这个函数返回值应该被赋回这个字符串。如下的例子。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style 使用的风格。如果不设置，将使用当前 GUISkin 的 textArea。

返回 string – 编辑过的字符串

描述：制作一个多行文本区域。这里用户可以编辑这个字符串。

```
var stringToEdit = “Hello World\nI’ve got 2 lines…”;
function OnGUI()
```

```
{
    //制作一个多行文本区域来调整 stringToEdit
    stringToEdit = GUI.TextArea(Rect(10, 10, 200, 100), stringToEdit, 200);
}
```

- static function TextField(position : Rect, text : string) : string
- static function TextField(position : Rect, text : string, maxLength : int) : string
- static function TextField(position : Rect, text : string, style : GUIStyle) : string
- static function TextField(position : Rect, text : string, maxLength : int, style : GUIStyle) : string

参数

position 屏幕上用于文本的矩形区域

text 用于编辑的文本。这个函数返回值应该被赋回这个字符串。如下的例子。

maxLength 字符串的最大长度。如果不设置，用户可以一直输入。

style 使用的风格。如果不设置，将使用当前 GUISkin 的 textField 风格。

返回 string – 编辑过的字符串

描述：制作一个单行文本域。这里用户可以编辑这个字符串。

```
var stringToEdit = "Hello World";

function OnGUI()
{
    //制作一个文本域来调整 stringToEdit
    stringToEdit = GUI.TextField(Rect(10, 10, 200, 20), stringToEdit, 25);
}
```

- static function Toggle(position : Rect, value : bool, text : string) : bool
- static function Toggle(position : Rect, value : bool, image : Texture) : bool
- static function Toggle(position : Rect, value : bool, content : GUIContent) : bool
- static function Toggle(position : Rect, value : bool, text : string, style : GUIStyle) : bool
- static function Toggle(position : Rect, value : bool, image : Texture, style : GUIStyle) : bool
- static function Toggle(position : Rect, value : bool, content : GUIContent, style : GUIStyle) : bool

参数

position 屏幕上用于按钮的矩形区域

value 这个按钮是打开的或关闭

text 显示在该按钮上的文本

image 显示在该按钮上的 Texture

content 用于这个按钮的文本，图形和提示

style使用的风格。如果不设置，将使用当前 GUISkin 的 toggle 风格。

返回 bool – 按钮的新值

描述：制作一个 on/off 开关按钮

- static function Toolbar(position : Rect, selected : int, texts : string[]) : int
- static function Toolbar(position : Rect, selected : int, images : Texture[]) : int
- static function Toolbar(position : Rect, selected : int, contents : GUIContent[]) : int
- static function Toolbar(position : Rect, selected : int, texts : string, style : GUIStyle[]) :

int

◆ static function Toolbar(position : Rect, selected : int, images : Texture, style : GUIStyle[]) : int

◆ static function Toolbar(position : Rect, selected : int, contents : GUIContent[], style : GUIStyle) : int

参数

position 屏幕上用于工具栏的矩形区域

selected 选择按钮的索引

texts 显示在该工具栏上的字符串数组

images 显示在工具栏按钮上的纹理数组

contents 用于这个工具栏的文本，图形和提示数组。

style使用的风格。如果不设置，将使用当前 GUISkin 的 button 风格。

返回 int – 选择按钮的索引

描述：制作一个工具栏

◆ static function UnfocusWindows() : void

描述：从所有窗体上移除焦点。

◆ static function VerticalScrollbar(position : Rect, value : float, size : float, topValue : float, buttonValue : float) : float

◆ static function VerticalScrollbar(position : Rect, value : float, size : float, topValue : float, buttonValue : float, style : GUIStyle) : float

参数

position 屏幕上用于滚动条的矩形区域。

value 在 min 和 max 之间的位置。

size 能看见多大？

topValue 滚动条顶端的值

bottomValue 滚动条底端的值

style使用的风格。如果不设置，将使用当前 GUISkin 的 horizontalScrollbar 风格。

返回 float – 修改后的值。这可以通过拖动这滚动条，或者单击两端的箭头来改变。

描述：制作一个垂直滚动条。滚动条可以用来滚动文档。大多数情况下，你会使用 scrollViews 代替。

找到额外的元素：

在滚动条两端的按钮将在当前皮肤中搜索“upbutton”和“downbutton”作为风格。滚动条的滑块（你拖动的东西）将搜索并使用名为“thumb”的风格。

//这将使用下面的风格名来决定该按钮的尺寸位置。

//MyVertScrollbarupbutton – 用于上端按钮的风格名称。

//MyVertScrollbardownbutton – 用于下端按钮的风格名称。

//MyVertScrollbarthumb – 用于滑块的风格名称。

scrollPos = HorizontalScrollbar(Rect(0, 0, 100, 20), scrollPos, 1, 0, 100, “MyVertScrollbar”);

◆ static function VerticalSlider(position : Rect, value : float, topValue : float,

buttonValue : float) : float

◆ static function VerticalSlider(position : Rect, value : float, topValue : float, buttonValue : float, slider : GUIStyle, thumb : GUIStyle) : float

参数

position 屏幕上用于滑杆的矩形区域。

value 滑杆显示的值。这个决定可移动滑块的位置。

topValue 滑杆顶端的值

bottomValue 滑杆底端的值

slider 用于显示拖动区域的 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalSlider。

thumb 用于显示土洞区域的 GUIStyle。如果不设置，将使用当前 GUISkin 的 horizontalSliderThumb。

返回 float – 被用户设置的值。

描述：一个用户可以拖动的垂直滑杆。可以在 min 和 max 之间改变一个值。

◆ static function Window(id : int, position : Rect, func : WindowFunction, text : string) : Rect

◆ static function Window(id : int, position : Rect, func : WindowFunction, image : Texture) : Rect

◆ static function Window(id : int, position : Rect, func : WindowFunction, content : GUIContent) : Rect

◆ static function Window(id : int, position : Rect, func : WindowFunction, text : string, style : GUISytle) : Rect

◆ static function Window(id : int, position : Rect, func : WindowFunction, image : Texture, style : GUIStyle) : Rect

◆ static function Window(id : int, clientRect : Rect, func : WindowFunction, title : GUIContent, style : GUIStyle) : Rect

参数

id 用于每个窗口的唯一 ID。这是用于接口的 ID。

clientRect 屏幕上用于组的矩形区域。

func 在窗体内部创建 GUI 的函数。这个函数必须使用一个函数 – 当前创建 GUI 的窗体 id

text 作为窗体标签的文本。

image 用于在标题栏上显示图片的 Texture

content 用于这个窗口的文本，图形和提示。

style 用于窗口的可选风格。如果不设置，将使用当前 GUISkin 的 window。

返回 Rect – 窗口位于的矩形位置

描述：制作一个弹出窗口

窗口浮动在普通 GUI 控件之上，具有单击激活的特点并可以有选择的随意被端用户拖动。不像其他的控件，你需要传递给他们一个独立的功能并放置在窗口中。注意：如果你使用 GUILayout 在窗口中放置你的组件，你应该使用 GUILayout.Window。这是一个小例子帮助你开始：

```
var windowRect = Rect(20, 20, 120, 50);
function OnGUI()
{
```

```
        //注册窗口。注意第三个参数。
        windowRect = GUI.Window(0, windowRect, DoMyWindow, “My Window”);
    }
    //制作窗口内容
    function DoMyWindow(windowID : int)
    {
        if(GUI.Button(Rect(10, 20, 100, 20), ”Hello World”))
print(“Get a click”);
    }
```

你可以使用相同的函数来创建多个窗口。需要确保每个窗口有一个自己的 ID。例如：

```
    var windowRect0 = Rect(20, 20, 120, 50);
    var windowRect1 = Rect(20, 100, 120, 50);
    function OnGUI()
        //注意窗口。我们创建了两个使用相同函数的窗体
        //注意他们的 ID 不同。
    windowRect0 = GUI.Window(0, windowRect0, DoMyWindow, “My Window”);
    windowRect1 = GUI.Window(1, windowRect1, DoMyWindow, “My Window”)
}
//制作窗口内容
function DoMyWindow(windowID : int)
{
    if(GUI.Button(Rect(10, 20, 100, 20), “Hello World”))
    print(“Get a click in window ” + windowID);
    //使窗口可以被拖动
    GUI.DragWindow(Rect(0, 0, 10000, 10000));
}
```

停止显示窗口，简单的在 DoGUI 函数内停止调用 GUI.Window。

```
//布尔变量以决定是否显示窗口
//从游戏 GUI，脚本，检视面板中或者其他地方改变这个决定窗口是否可见。
var doWindow() = true;
//制作窗口内容
function OnGUI()
{
    //制作一个开关变量来隐藏或显示窗口
    doWindow() = GUI.Toggle(Rect(10, 10, 100, 20), doWindow(), “Window 0”);
    //确保仅在 doWindow()为真时调用 GUI.Window
    if(doWindow())
        GUI.Window(0, Rect(110, 10, 200, 60), DoWindow(), “Basic Window”);
}
```

为了使窗口从自动 GUI 获取它的尺寸，使用 GUILayout.Window。

调用顺序

窗口需要从后向前绘制。在其他窗口顶部的窗口需要在其他窗口之后绘制。这就意味着你不能指望你的 DoWindow 函数以任何特定的顺序被调用。为了让这个能够工作，

当你的创建窗口时下面值被存储（使用 Window 函数），当 DoWindow 被调用时取回：GUI.skin, GUI.enabled, GUI.color, GUI.backgroundColor, GUI.contentColor, GUI.matrix

这就是说很容易像这样制作彩色窗口：

```
var windowRect0 = Rect(20, 20, 120, 50);
var windowRect1 = Rect(20, 100, 120, 50);
function OnGUI()
{
    //这里我们制作了 2 个窗口，在这个之前设置 GUI.color 的值。
    GUI.color = Color.red;
    windowRect0 = GUI.Window(0, windowRect0, DoMyWindow, “Red Window”);
    GUI.color = Color.green;
    windowRect1 = GUI.Window(1, windowRect1, DoMyWindow, “Green Window”);
}
//制作窗口内容
//GUI.color 的值被设置为窗口被创建之前的值。
function DoMyWindow(windowID : int)
{
    if(GUI.Button(Rect(10, 20, 100, 20), “Hello world!”))
        print(“Got a click in window with color ” + GUI.color);
    //使窗口可以被拖动
    GUI.DragWindow(Rect(0, 0, 10000, 10000));
}
```

提示：你可以使用 GUI.color 的 alpha 组件来淡入淡出窗口。

参见：DragWindow, BringWindowToFront, BringWindowToBack

GeometryUtility

类

用于普通集合功能的工具类

类方法

- static function CalculateFrusturnPlanes(camera : Camera) : Plane[]

  描述：计算视锥平面

  这个函数取给定的相机视锥并返回它的六个面。

  参见：Plane, GeometryUtility.TestPlanesAABB

- static function CalculateFrusturnPlanes(worldToProjection : Matrix4x4) : Plane[]

  描述：计算视锥平面

  这个函数返回由给定的视和投影矩阵定义的视锥的六个面。

  参见：Plane, GeometryUtility.TestPlanesAABB

- static function TestPlanesAABB(planes : Plane[], bounds : Bounds) : bool

  描述：如果包围盒在平面数组内部返回真。

  如果包围盒在平面内部或者与任何平面交互返回真。

  参见：GeometryUtility.CalculateFrusturnPlanes。

Gizmos

类

Gizmos 用于场景中给出一个可视化的调试或辅助设置。

所有的 Gizmos 绘制都必须在脚本的 OnDrawGizmos 或 OnDrawGizmosSelected 函数中完成。

OnDrawGizmos 在每一帧都被调用。所有在 OnDrawGizmos 内部渲染的 Gizmos 都是可见的。

OnDrawGizmosSelected 尽在脚本所附加的物体被选中时调用。

类变量

- static var color : Color

描述：设置下次绘制的 Gizmos 的颜色。

```
function OnDrawGizmosSelected()
{
    //在物体的前方绘制一个 5 米长的线
    Gizmos.color = Color.red;
    var direction = transform.TransformDirection(Vector3.forward) * 5;
    Gizmos.DrawRay(transform.position, direction);
}
```

- static var matrix : Matrix4x4

描述：设置用于渲染所有 gizmos 的矩阵。

类方法

- Static function DrawCube(center:Vector3,size:Vector3):void

描述:用 center 和 size 绘制一个立方体.

```
Function OnDrawGizmosSelected(){
        //在变换位置处绘制一个变透明的蓝色立方体
        Gizmos.color=Color(1,0,0,5);
        Gizmos.DrawCube(transform.position,Vector3(1,1,1));
}
```

- Static function DrawGUITexture(screenRect:Rect,texture:Texture,mat:Material=null):void

描述：在屏幕坐标下绘制一个纹理。用于 GUI 背景。

- Static function DrawGUITexture(screenRect:Rect,texture:Texture,leftBorder:int,rightBorder:int,topBorder:int,bottomBorder:int,mat:Material=null):void

描述:在屏幕坐标下绘制一个纹理。用于 GUI 背景。

- Static function Drawicon(center:Vector3,name:string):void

描述:在世界位置 center 处绘制一个图标.

这个图标被命名为 name 并放置在 Assets/Gizmos 文件夹或 Unity.app/Resoutces 文件夹.DrawIcon 允许你在游戏中快速选择重要的物体。

```
//在物体位置处绘制光源灯泡图标.
//因为我们在 OnDrawGizmos 函数内部调用它,在场景视图中
//这个图标总是可点选的.
function OnDrawGizmos(){
     Gizmos DrawIcon(transform.position,”Light Gizmo.tiff”);
}
```

- Static function DrawLine(from:Vector3,to:Vector3):void

描述:绘制一条线从 from 到 to.

```
Var Larget:Transform;
function OnDrawGizmosSelected(){
        if(target != null)
        {
                //从 transform 到 target 绘制一条蓝色的线
          Gizmos.color = Color.blue;
                Gizmos.DrawLine(transform.position,target.position);
}
}
```

◆ static function DrawRay(r:Ray):void

static function DrawRay(from:Vector3,direction:Vector3):void

描述:绘制一个射线从 from 开始到 from + direction.

◆ function OnDrawGizmosSelected(){

```
Gizmos.color = Color.red;
Direction = transform.TransformDirection(Vector3.forward)*5;
Gizmos.DrawRay(transform.positon,direction);
}
```

◆ Static function DrawSphere(center:Vector3,radius:flont):void

描述:用 center 和 randins 绘制一个球体.

```
Function OnDrawGizmosSelected(){
        //在变换位置处绘制一个黄色的球体
        Gizmos.color = Color.yellow;
        Gizmos.DrawSphere(transtorm.position,1);
}
```

◆ Static function DrawWireCube(center:Vector3, size: Vector3):void

描述:用 center 和 radius 绘制一个线框立方体.

```
Function OnDrawGizmosSelected(){
       //在变换位置处绘制一个黄色立方体
       Gizmos.color = Color.yellow;
       Gizmos.DrawWireCube (transtorm.position, Vector3(1,1,1));
}
```

◆ Static function DrawWireSphere(center:Vector3,radius:float):void

描述:用 center 和 radius 绘制一个线框球体.

```
Var explosionRadius = 5.0;
Function OnDrawGizmosSelected(){
       //选中的时候显示爆炸路劲
       Gizmos.color = Color.white;
       Gizmos.DrawSphere(transtorm.position,explpsionRadius);
}
```

Graphics

类

Unity 绘制函数的原始接口。

这个是高级快捷地进去 Unity 优化网格绘制的地方。只限于 Unity Pro.

类方法

◆ **static function DrawMesh(mesh:Mesh, position: Vector3, rotation: Quaternion, material: Material, layer: int, camera: Camera=null, submeshIndex: int, properties: MaterialPropertyBlock=null): void**

◆ **static function DrawMesh(mesh:Mesh,matrix:Matrix4x4,material:Material,layer:int, camera:Camera = null,submeshIndex:int,properties:MaterialPropertyBlock=null): void**

参数

mesh 用于绘制的 **Mesh.**

position 网格的位置。

rotation 网格的旋转。

matrix 网格的变换矩阵（由位置，旋转和变换 **x** 组合）

material 使用的 **Material.**

layer 使用的 **Layer.**

Camera 如果是 **null(缺省)**，该网格将在所有相机中被绘制，否则它将只会在给定的相机中渲染。

submeshIndex 那个子网格被渲染。这只是在网格使用了多个材质的时候使用。

Properties 在网格绘制前应用到才子的额外材质属性。参考 **MaterialPropertyBlock.**

描述:绘制一个网格

DrawMesh 在一帧中绘制一个网格。这个网格将受到光照的影响，可以投射接收阴影并被投射器影响。就像它是某个物体的一部分。他可以绘制于所有相机，或者只是特定的一些相机

在你想创建人景的网格，而又不想过多的创建和管理游戏物体的时候使用 **DrawMesh.** 注意，**DrawMesh** 不会立即绘制网格；它仅仅提交它用于渲染。网格将被作为普通渲染过程的一部分。如果想立即绘制一个网格。使用 **Graphics.DrawMeshNow**。

因为 **DrawMesh** 不会立即绘制网格，在调用这个函数之间修改材质属性并会使材质使用它们。如果你想绘制一系列相同材质的网格，但是稍微有些不同的属性（例如，改变每个网格的颜色），那么使用 **MaterialPropertyBlock** 参数。

参见：**MateralPropertyBlock.**

◆ **Static function DrawMeshNow(mesh:Mesh,position:Vector3,rotation:Quatemion):void**

描述：在给定的 **position**,用给定的 **rotation** 绘制一个 **mesh**。

这个函数将设置模型视矩阵并绘制网络。当前设置材质的 **pass** 可以被使用（参考 **Setpass**）

◆ **Static function DrawMeshNow(mesh:Mesh,position:Vector3,rotation:Quatemion,materalIndex:int):void**

描述：在给定的 **position**，用给定的 **rotation** 和一个 **materialIndex** 绘制一个 **mesh.**

◆ **Static function DrawMeshNow(mesh:Mesh,matrx:Matrix4x4):void**

描述:用给定的 **matrix** 绘制一个 **mesh.**

如果这个矩阵有一个负的缩放这个函数将不会正确渲染物体。

◆ **Static function DrawMeshNow(mesh:Mesh,matrx:Matrix4x4,materialIndex:int):void**

描述：用给定的 **matrix** 和 **materialIndex** 绘制一个 **mesh.**

如果这个矩阵有一个负的缩放这个函数将不会正确渲染物体。

◆ **Static function DrawTexture(screenRect:Rect,texture:Texture,mat:Material = null):void**

描述：在屏幕坐标下绘制一个纹理。

◆ **Static function DrawTexture(screenRect:Rect,texture:Texture,leftBorder:int,rughtBord:int,topBorder:int,bottomBorder:int,mat:Material=null):void**

描述：在屏幕坐标下绘制一个纹理。

◆ **Static function DrawTexture(screenRect:Rect,texture:Texture,sourceRect:Rect,leftBorder:int,rughtBord:int,topBorder:int,bottomBorder:int,mat:Material=null):void**

描述：在屏幕坐标下绘制一个纹理

◆ **static function DrawTexture(screenReet:Rect,texture:Texture,sourceRect:Rect,leftBordr:int.rightBrder:int.topBorder:int,bottomBorder:int,color:Bolor,mat:Material=null):void**

描述：在屏幕坐标下绘制一个纹理。

Hashtable

类

函数

◆ **function Add(key:Object, value:Object)：void**

描述：添加指定的键和值到哈希表。

◆ **function Clear():void**

描述：数量被设置为 **0**，并且从该集合中元素到其他对象的引用也将被释放，容量没有改变。**///**这个方法是 **O(n)**操作，这里 **n** 是元素数量。

◆ **function Contains(key:object):bool**

描述：决定该哈希表中是否包含指定的键。

◆ **function ContainsKey (key:object):bool**

描述：从哈希表中移除指定键的元素。

◆ **function Remove(key:object):void**

描述：从哈希表中移除指定键的元素。

HostData

类

这个是用来保存单个主机信息的数据结构。

从 **master** 服务器取回的主机列表，使用这个类表示单个主机。

变量

◆ **var comment:string**

描述：一个多用途的注释（可以保存数据）

◆ **var conneetedfPlaycrs:int**

描述：当前连接的玩家

◆ **var gameName:stting**

描述：游戏的名称（像 **John Doe's Game**）

◆ **var gameType:string**

描述：游戏类型（像 **MyUniqueGameType**）

◆ var ip:string[]

描述：服务器 IP 地址

◆ var passwordProtected:bool

描述：服务器需要密码吗？

◆ var playerLimit:int

描述：最大玩家限制

◆ var port:int

描述：服务器端口

◆ var nseNat:bool

描述：这个服务器需要 NAT 穿透吗？

input

类

到输入系统的接口。

使用这个类读取在 Input Manager 中设置的轴。

使用 Input.GetAxis 用下面的缺省轴读取轴心："Horizontal"和"Vertical"被映射到摇杆，A，W,S,D 和方向键，"Mouse X"和"Mouse Y"被映射到鼠标添量，"Fire1"，"Fine2"，"Fine3"映射到 Ctrl，Alt，Cmd 键和鼠标的三键或腰杆按钮。新的输入轴可以在 Input Manager 中添加。

如果你在为任何一种运动行为使用输入，那么使用 Inpur.GetAxis。它给你一种平滑的可配置的输入，这个输入可以被映射到键盘，摇杆或鼠标。

使用 Input.GetButton 用于像事件一样的行为，不要将它用于移动，Input.GetAxis 将使脚本代码更小更简单。

类变量

◆ static var anyKey:bool

描述：当前按住了任何键或鼠标按钮吗？（读取）

◆ static var anyKeyDown:bool

描述：用户按下任何键或鼠标按钮返回（Rend Only）。直到用户释放所用键/按钮并再次接下任何键/按钮时才返回真。

◆ static var input: String

描述：返回这一帧中键盘的输入（只读）。

只有 ASCII 可以包含在 inputString 中。

该字符串可以包含两个能被处理的特殊字符：字符“\b”代表退格。

*符"\n"表示回车。

```
//显示如何从键盘读取输入
//（例如，用户输入它的名字）。
//你需要附加这个脚本到一个 GUTText 物体。function Updarc(){
for (var c;char in Input inputString){
//退格，移除最后一个字符
if(c=="\b") {
iftguiText.Lcngth.text.Substring(0,guiText.text Length-1);
}
//结束
efse if (e=="\n"){
prinr("Uscr cntered his name:"+guiText,text);
```

```
}
//正常的文本输入.附加到尾部
elsc
}
guiText.text+=c;
}
}
}
```

◆ static var mousePosition:Vcctor3

描述:当前鼠标在像素坐标下的位置。

屏幕或窗体的左下为(0,0)，屏幕或窗体的左上为(Screen.width,Screen.height).

```
var particle:GamcObject,
function Update() {
if (lnput,GctButtonDown{"Firel")){
//从当前鼠标坐标处创建一个射线
var ray=Camera.main.ScreenPointToRay(Input mousePositicn);
if{Physics.Raycast(ray)) {
//如果碰到创建一个粒子
Instontiatc(particlc,transfonn.position,transfonn.rotation);
}
}
}
```

类方法

◆ static function CetAxis(axinName:string):float

描述：设置由 axinName 确实虚拟轴的值。

对于键盘和摇杆输入，这个值将在-1...1。

一个非常简单的行驶在 x-z 平面的汽车。

```
var spced=10.0;
var cotatinnSpeed=100.0;
function update()
}
//获取水平和垂直轴，
//默认滴它们被映射到方向键
//这个值的范围在-1 到 1
var transtation=loput.getaxis("vertical")*spccd;
var rotation=inpul getaxis ("horizontal"*rotarionspeed)
//使它以 10 米/秒速度移动而不是 10 米/帧
translation*=time delta time
//沿着物体的 Z 轴移动变换
transtorm.transtate(1010translation);
//绕着 Y 轴旋转
transform.rotate(0.rotation.0);
}
//指定一个鼠标查看
```

```
var horizontalspccd=2.0;
var verticelspeed=2.0
function update()
{
获取鼠标增量，这个没有在范围-1...1 之间
var h=horizontalspeed*inpul.getaxis("mouse X");
var v=verticalspeed*input.getaxis("mouse Y");
transform.rotate(v.h.0);
}
```

◆static function getaxisraw(axisname:string):float

描述：设置由 axisname 确定虚拟轴的值，并且没有使用平滑过滤，

对于键盘和摇杆输入，这个值将在-1...1 的范围内。因为输入没有被平滑，

总是-1，0 或 1.如果你想自己处理所有键盘输入的平滑，这个是很有用的。

```
function update(){
var speed=input.getaxisraw("horizoatal"*time deta time);
transform rotate(0.speed.0);
}
```

◆static function getbutton(button name:string):bool

描述：当时由 button name 确定的虚按钮被按住时返回真。

考虑自动开火-这个将在按钮被按住时一直返回真。

```
//如果每 0.5 秒实例化一个 projcctile,
//如果 firel 按钮（默认为 Ctrl）被按下
var projectile:game object;
var firerate=0.5;
private var nextfire=0.0;
function update()
{
fi(input getbutton("firel")&&time.time>nextfire){
nextfire=time.time+firerate;
clone=instantiate(peojecctile,teansform,position,transffrm,rotation);
}
}
```

只有使用这个实现事件行为例如，射击，使用 input.getaxis 用于任意类型的移动行为

◆static function getbuttondown(button name:steing):bool

描述：当由：button name 确定的虚拟按钮被按下时返回真。

知道用户释放并再次按下时返回真。

```
//当用户按下 firel 按钮时实黎化一个 projcctile
var projectile:gameobject;
function update(){
if(input getbution down("firel")){
elone=instantiate(projectile,transforrn,position.transforn.rotinon);
}
}
```

只使用这个实现事件行为例如，射击，使用 Input.GetAxis 用于任意类型的移动行为。

◆ static function GetButton Up(buttonName:string):bool

描述：当由 buttonName 确定的虚拟按钮被释放时返回真.

直到用户按下按钮并再次松开时返回真。

//当用户按下 Fire;按钮时实例化一个 projectile.

```
var projectile:GameObject;
function Update O;
if(lnput GetButtonUp("Firel")){
clcne=lnstantiate{projectile,transtorm.position,transform.rotation);
   }
}
```

只使用这个实现事件行为例如，射击。使用 lnput.GetAXis 用于任意类型的移动行为，

◆ static funcrion GetKey(name:string):bool

描述：当用户按住由 name 确定的键时返回真，考虑自动开火。

键标识列表参考输入管理器。在处理输入时建议使用 lnput.GetAxis 和 lnput.GetButton 因为它允许端用户定义这些键.

```
funcrion Update(){
if(lnput.GetKey("up")){
    print("up arrow key is held down");
}
if(lnput.GetKey("down"))
}
    print("down arrow key is held down");
   }
}
```

◆ static junction GetKey(key:keyGode):bool

描述：当用户按住由 key KeyCode 枚举参数确定的键时返回真.

```
funcrion Update(){
if(lnput.GetKey(KeyCode.UpArrow))}
    print("up arrow key is held down");
}
if(lnput.GetKey(KeyCode.DownArrow)){
    print("down arrow key is held down");
    }
}
```

◆ static function GetkeyDown(name:string):bool

描述：当用户开始按下由 name 确的键时返回真.

直到用户释放按钮并再次按下时返回真.

键标示列表参考 lnput Manager.在处理输入时建议使用 lnput.GetAxis 和 lnput.GetBUtton

```
function Update();
if(input GetkeyDown("space"))
    print("space key was pressed");
   }
}
static function GetKeyDown(Key:KeyCode):bool
```

描述：当用户按下由 Key KeyCode 枚举参数确定的键时返回真.

```
function Update();
  if{lnput GetKeyDown(KeyCode.Space))
print("space key was pressed");
}
}
```

◆ static function GetKeyUp(name:sting):bool

描述：当用户释放由 name 确定的键时返向真.

直到用户按下按钮并再次松开时返向真。

键标示列表参考 lntpot Manager.在处理输入时建议使用 lnput.GetAxis 和 lnput.GetButton

因为它允许端用户定义这些键.

```
function Update();{
if(lnput.GetKeyUp("space");
   }
}
```

◆ static function GetKeyUP(Key:KeyCode):bool

描述：当用户释放由 Key KeyCde 校举参数确定的键时返回真.

```
function Update(){
if(lnput,GetKeyUp(KeyCode.Space));
   print("space key was feleased");
   }
}
```

◆ statie function GetMouseButton(button:int):bool

描述：返回给定的鼠标按钮是否被按住。

/button/值为 0 表示左键，1 表示右键，2 表示中键.

◆ statie function GetMouseButtonDown(button:int)bool

描述：用户按下给定的鼠标按钮时返回真.

直到用户释放按钮并再次按下它时才返回真.button 值为 0 表示中键左键，1 表示右键，2 表示中键.

◆ statie function GetMouseButtonUp(button:int)bool

描述：用户释放给定的鼠标按钮时返回真.

直到用户释放按钮并再次释放它时才返回真.button 值为 0 表示中键左键，1 表示右键，2 表示中键.

◆ statie function ResetlnputAxes();Void

描述：重置所有输入。在 ReaetlnputAxes 之后所有轴将返回 0 并且所有按钮返回 0.

这可用于当重生玩家，并且当你不想任何来自键盘的输入还处于按下时。

```
lnpu.ReaetlnputAxes();
```

jointDrive

结构

关节如何沿着本地 X 轴移动

变量

◆ var maximumForce:float

描述：用于向指定的方向推动物体的力的量。仅在 Mode 包含 Velocity 时使用。

◆ **var mode:jointDriveMode**

描述：驱动是否尝试到这个位置和/或速度.

◆ **var positionDamper:float**

描述：位置弹簧的阻力强度，只用于 mode 包含位置的时候.

◆ **var positionSpring:float**

描述：朝着定义的方向推动的一个阻力强度。只用于 mode 包含位置的时候.

Jointlimits

结构

jointlimits 被 **Hingejoint** 使用来限制关节角度.

参见:**Hingejoint**

变量

◆ **var max:float**

描述：关节的上限。当关节角度或位置的上限。

关节将使用力来约束它。

//设置最小链接角度为 **20** 度

hingejoint.limits.max=40;

◆ **var maxBounce:float**

描述：当关节碰到关节的上限时关节的弹力。

//设置关节反弹时的上限

hingejoint.limits.maxBounce=1;

◆ **var min:float**

描述：关节的下限。当关节角度或位置的下限。

关节将使用力来约束它。

//设置最小链接角度为 **20** 度

hingejoint.limits.min=20;

◆ **var minBounce:float**

描述：当关节碰到关节的下限时关节的弹力。

//关节反弹时的下限

hingejoint.limits.minBounce=1;

jointMotor

结构

jointMotor 用来旋转一个关节

例如：**Hingejoint** 可以被告知以制定的速度和力旋转。关节然后用给定的最大力试图到达这个速度，参见：

变量

◆ **var force：float**

描述：动力将应用最大为 **force** 的力以便取得 **targetvelocity**

◆ **var freeSpin：bool**

描述：如果 **freeSpin** 被启用动力将只加速而不会减速

◆ **var targetvelocity：float**

描述：动力将应用最大为 **force** 的力以便取得 **targetvelocity**

Jointspring

结构

变量

◆ **var damper：float**

描述：用语阻尼弹簧的阻尼力。

◆ **var spring：float**

描述：用于达到目标位置的弹力

◆ **var targetposition：float**

描述：关节试图到达的目的位置

这里一个 **hingehoint** 目标位置是目标角度

Keyframe

结构

一个关键帧它可以插入动画的曲线

变量

◆ **var in tangent:float**

描述：在曲线上从下一个点到这个点时描述切线。

参加：**outtangent**

◆ **var outtangent:float**

描述：在曲线上从这个点到下一个点时描述切线。

参见：**intangent**

◆ **var time：float**

描述：关键帧时间

在一个 **2D** 图形中，你可以把这当做 **x** 值

参见：**value**

◆ **var value：float**

描述：该动画曲线在关键帧处的值

参见：**time**

构造函数

◆ **static function keyframe（time：float，value：float）：keyfranie**

描述：创建一个关键帧

◆ **static function keyframe（time：float，value：float,intangent:float,outangent:float):keyframe**

描述：创建一个关键帧

layermask

结构

layermask 允许你在检视面板中显示 **layermask** 弹出菜单

类似与 **camera，cullingmask。layermasks** 可以选择性地过滤物体，例如当投射射线，

//使用层蒙板投射一个射线

//它可以在检视面板中修改

```
var mask larermask=
function Update(){
if(Physics.Raycast(transform.position, transform.forward, 100, mask.value))
{
Debug.Log(“Hit something”;
}
}
```

变量

◆ var valuc：vin

描述：转化层蒙板的值为一个整形值

//使用层蒙板投射一个射线

//它可以在检视面板中修改

var mask：lnyermask=-1

fonction update（）{

if{physics.rayeast（transforn.position.transform.forwatd.100.mask.value))

{

Debug

}

}

类方法

◆ static function LayerToName（layer ：in）：string

描述：给定一个层的数字。返回内置的或在 tag Maneger 中定义的层名称。

◆ static function NameToLayer（layerName ：string）：int

描述：给定一个层的名字。返回内置的或在 tag Maneger 中定义的层索引。

◆ static implicit function LayerMask(intva1:int):layermask

描述：隐式转化一个整数位已个层蒙板

Lightmapdata

类

光照贴图数据

一个场景可以有多个光照贴图储存在这里，renderer 组建可以使用这些光照贴图，这就使得它能够在多个物体上使用相同的材质，而每个物体可以使用不同的光照贴图或同一个光照贴图的不同部分。

参见，linghrmapsettings 类 renderer lightmapindex 属性

变量

◆ var lightmap:texture2D

描述：光照贴图的纹理

参见：lightmspsettings 类 renderer lighunaplndex 属性

linghtmapsettings

类

储存这个场景的光照图

一个场景可以有多个光照贴图储存在这里，renderer 组件可以使用这些光照贴图，这就使得它能够在多个物体上使用相同的材质，而每个物体可以使用不同的光照贴图或同一个光照贴图的不同部分。

参见：lightmapdata 类 renderer.lightmaplndex 属性

类变量

◆ sartic var lightmaps:lightmapdata[]

描述：光照贴图数组

参见：lightmapdata 类 renderer.lightmaplndex 属性。

Masterserver

类，主服务器用来使服务器和客服端匹配。

你可以建立以个游戏主机或为你的游戏类型取回主机列表。这里的这个函数用来与主服务器通信，主服务器位于不同的服务器上，如果必要这个服务器可以被定制，主服务器

的概览和技术描述：参见 master server manual page

类变量

◆ static var dedicatedserver:bool

描述:申明这台机器为专用服务器。

如果作为一个服务器运行，连接数定义了玩家的数量，当在主服务器上注册的时候这个被报告，默认情况下主服务器假定这个实例不是专用服务器，因此玩家数会增加 1（占用服务器上的一个\“chent”\）如果不希望，这个变量可以设置为假，然后知有连接数被报告为玩家数。

```
function startserver{}{
nerwork lnitializeserver(32.25002):
masrerserver.dedicatedserver=true;
masterserver.registerhost("myuniquegametype","johndoes game","133tgame for all");
}
```

◆static var ip address:string

描述：主服务器的 IP 地址。

默认地这个是由 unity 技术运行的服务器。

```
function
masterserver ip address="127.0.0.1";
master server.port=10002;
}
```

◆static var port:int

描述：主服务器的链接端口。

默认地这个是由 unity 技术云顶的服务器，

```
masterserver.ipaddress="127.0.0.1";
master server.port=100021;
}
```

◆static var updaterate:int

描述：为主服务器主机信息更新设置最小更新速率。

通常，主机更新只在主机信息被改变的时候发生（如连接的玩家）。更新率定义了主机更新之间的最小时间量，缺省值为 60 秒（检查更新）因此，如果以个主机更新呗发送然后一些域在 10 秒后改变，然后更新可能会在 50 秒之后发送（在下一次检查到改变的时候）。如果这个被设置为 0，以后就没有更新被发送，只发送初始注册信息。

```
function startserver()
{
network lnitialigeserver(32.25002);
//在初始化注册之后没有主机信息更新
Masterserver.updaterate=0;
masterserver,registerhost("myuniquegametype","johndoes game","133tgame forall");
}
```

类方法

◆static function clearhostlist():void

描述：清楚由 masterserver.pollhostlist 接收到的主机列表。

如果你想更新列表并想确保你不使用较旧的数据时，使用这个，

```
function awakc()
```

```
{
//确保类表是空的并请求一个新的列表
masterserver clearhostlist();
masterserver.requesthostlist("myuniquegametype");
}
function lpdate()
{
//如果任何主机被接收，显示游戏名称，再次清理主机列表，}
if(masterserver.pollhostlist(),length!=0){
var hostdata;hostdata[]=masterserver.pollhostlist();
for(var i:int=0;i<hostdata.length;i++){
      debug,log("game name:"+hostdata[i]gamename);
}
masterserver,clearhostlist()
}
}
```

◆static function pollhostlist():hostdata[]

描述：使用 master server.requeshostlist 接收最新的主机列表

使用 masterserver.clearhostlist 清理当前主机列表。这样你可以确定返回的列表时最新的。

```
  function awaket(){
//确保列表时空的并请求一个新的列表
masterserver .clearhostlist();
masterserver.requesrhostlist("larustest"):
;
function update()
;
//如果任何主机被接收，显示游戏名称，再次清理主机列表；
if(masterserver.pollhostlist()length!=0){
varhostdata hostdata[]=masterserver.pollhostlist();
for(var i:int=0;i<hostdata.length;i++){
debug.log("game name"+hostdata[i].gamename);
}
masterserver.clearhostlist()
}
```

◆ static function registerhost(gametypename:string,gamename:string.comment:string=""):void

描述：在主服务器注册这个服务器。

如果主服务器地址信息尚未改变，缺省的主服务器将被使用。

```
function ongui()
{
if(guilayout.button("stnrt server")){
//如果没有公有 IP 地址，使用 NAT 穿透
network。usenat=！network.havepublicaddrss();
```

```
network.lnitializeserver(32.25002);
masterserver.registerhost("myuniquegametype","johndoes game","133tgame for all");
}
}
```

◆static function RequestHosList(gameTypeName:string):void

描述：从主服务器请求一个主机列表。

当它完成的时候，这个列表可以通过 MasterServer.PollHcstList 使用。

```
function Awakc(){
//确保列表时空的并请求一个新的列表
MastcrScrver.RequestHostList();
MastcrScrver.RequestHostList("LarusTcst");
}
function Update()
}
//如果任何主机被接收，显示游戏名称，再次清理主机列表。
if(MastcrScrvcr.PollHcstLisx().length!=0)}
var hostData:HostData[]=MasterSer.pollHostList();
for (var i:int=();i<hostData.length;i++){
Debug.Log("Game namc;"+hostData[i].gamceName);
}
MastcrScrvcr.ClearHostList();
}
}
```

static function UnregisterHost():void

描述：从主服务器注销这个服务器。

如果服务器没有被注销或已经被注销，不做任何事。

```
function OnGUt() {
if(GUILayout.Button("Disconnect")){
Nctwork,Disconnect();
NastcrServer.UnregisterHost();
}
}
```

MaterialPropertyBlock

类

应用的一块材质值。

MaterialPropcrtyBlock 被 Graphics.DrawMesh 使用。当你想用相同的材质（但有稍微有些不同的属性）绘制多个物体时，使用这个。例如，如果你想稍微改变每个网络的颜色时。

出于性能原因，属性块只能包含有限数量的属性值。两个 4x4 矩阵，六个向量颜色或 12 个浮点数可以存储在这个块中，存储空间时共享，所以存储一个矩阵留下两倍少的空间来

存储向量和浮点数。

Graphics。DrawMesh 拷贝传递的属性块，因此最有效的方法是使用它来创建一个快并为所有 DrawMesh 调用使用它。使用 Clear 来清除块的值，AddFloat，AddVector，AddColot，

AddMatrix 来添加值。

参见：Graphics，DrawMesh，Matcrial。

函数

- function AddColor(name：string，value:Colo):void
- function AddColor(nameID:int，value:Colo):void

描述：添加一个颜色材质属性。

出于性能原因，属性块只能包含有限量的属性值。两个 4x4 矩阵，六个向量/颜色或 12 个浮点数可以存储在这个块中。存储空间时共享，使用存储一个矩阵留下两个倍少的空间来存储向量和浮点数。当块的存储空间填满后，额外的 Add 调用将被忽略。

使用 nameID 的函数变量体更快，如果你重复添加相同名称的属性，使用 Shader.propertyToID 来获取不同的标示，并传递这个标示到 AddColor。

- function AddFloat(name：string，value:float):void
- function AddFloat(nameID:int，value:float):void

描述：添加一个浮点材质属性。

出于性能原因，属性块只能包含有限数量的属性值。两个 4x4 矩形，六个向量/颜色或存 12 个浮点数可以存储在这个块中，存储空间是共享，所以存储一个矩阵留下两倍少的空间来存储向量和浮点数。当块的存储空间填满后，额外的 Add 调用将被忽略。

使用 nameID 的函数变体更快，如果你重复添加相同名称的属性，使用 Shader.PropertyToID 来获取不同的标示，并传递这个标示到 AddFloat。

- function AddMatrix(name：string，value:Matrix4x4):void
- function AddMatrix(nameID:int，value:Matrix4x4):void

描述：添加一个矩阵材质属性。

处于性能原因，属性块只能包含有限数量的属性值。两个 4x4 矩阵，六个向量/颜色或 12 个浮点数可以存储在这个块中。存储空间时共享，所以存储一个矩阵留下两倍少的空间来存储向量和浮点数。当块的存储空间填满后，额外的 Add 调用将被忽略。

使用 nameID 的函数变体更快，如果你重复添加相同名称的属性，使用 Shader.PropertyToID 来获取不同的标示，并传递这个标示到 AddMatrix。

- function AddVector(name：string，value:Vector4):void
- function AddVector(nameID:int，value:Vector4):void

描述：添加一个向量材质属性。

出于性能原因，属性块只能包含有限数量的属性值。两个 4x4 矩阵，六个向量/颜色或 12 个浮点数可以存储在这个块中。存储空间是共享，所以存储一个矩阵留下两倍少的空间来存储向量和浮点数。当块的存储空间填满后，额外的 Add 调用将被忽略。

使用 nameID 的函数变体更快，如果你重复添加相同名称的属性，使用

Shader.ProrertyToLD 来获取不同的标示，并传递这个标示到 AddVector。

- function Clear():void

描述，清除材质属性值。

GrawMesh 拷贝传递的属性块，因此最有效的方式是使用它来创建一个块并为

所有 DrawMesh 调用使用它。使用 Clear 来清除块的值,ADDFloat,AddVector,AddColor,AddMatrix 来添加值.

Mathf

结构

常用数学函数的集合。

类变量

- **static var Deg2Rad:float**

描述：度到弧度的转化常量（只读）。

这个等于（PI*2)/360.

```
//转化 30 度为弧度
var deg=30.0;
var od=dcg*Mathf.Deg2Rad;
```

参见：Rad2Deg 常量。

- **static varEpsilon:float**

描述：一个小的浮点数值（只读）

比较小于它的值没有多人意见，因为浮点数是不精确的。

```
print(MathfEpsilon);
```

- **static var Infinity:float**

描述：表示正无穷（只读）.

- **static var NegativeIntinity:float**

描述：表示负无穷（只读）.

- **staic var PI:float**

描述：3.14159265358979...值（只读）.

- **static var Rad2deg:float**

描述：弧度到度的转化常量（只读）.

这个等于 360、（PI*2）.

转化 1 弧度为度

```
var rad=1.0;
var deg=rad*Mathf.Rad2Deg.
```

参见：Deg2Rad 常量

类方法

- **staric function Abs(f:float):float**

描述：返回 f 的绝对值。

```
//打印 10.5
print(Mathf.Abs(-10.5)).
```

- **staric function Abs(value:int):int**

描述：返回 value 的绝对值。

```
//打印 10
print(Mathf.Abs(-10));
```

- **staric function Acos(f:float):float**

描述:返回 f 的反余弦_一个弧度角它的余弦是 f.

```
print(Mathf.AcOS(0.5);
```

- **staric function Approximately(a:float,b:float):bool**

描述：比较两个浮点数值看看它们是否近似。

由于浮点数值不精确，不建议使用等操作来比较它们。例如，1.0==10.0/10.0 也许不会返回真。

```
if(Mathf.Approximately(1.0,10.0/10.0))
```

print("same");

◆ static function Asin(f:float):float

描述：返回 f 的反正弦_一个弧度角它的正弦是 f.

print(Matht.Asin(0.5));

◆ staric function Atan(f:float):float

描述：返回 f 的反切_一个弧度角它的正切是 f.

print(Mathf.Atan(0.5));

◆ staic function Atan2(y:float,x:float):float

描述：返回一个弧度角它的 Tan 为 y/x。

返回的值是一个角度，该角表示 x 轴和开始于零并终结在（x,y）处的向量之间的角度。

//通常使用 transform.lookAt.

//但这可以给你更多的对角度的控制

```
vartatget:Transform;
function Update()
{
var relativ=transfom.lnverseTeansformPoint(target.position);
var angle=Mathf.Atan2(reative,reative.z)*Mathf.Rad2Deg;
transform.Rotate(0,angle,0);
}
```

◆ static function Ceil(f:fioat):float

描述：返回大于等于 f 的最小整数.

◆ static function CeilTolnt(f:fioat):int

描述：返回大于等于 f 的最小整数.

◆ static function Clamp(value:float,min:float,max:float):float

描述：在一个最小的浮点数和最大的浮点数之间截剪。

//随着时间设置变换位置

//但是不会小于 1 或大于 3

```
function Update()
{
transtorm,position.x=Mathf.Clamp(lime.time.1.0,3.0);
}
```

◆ staric function Clamp(value:int,min:int,max:int):int

描述：在 min 和 max 之间裁剪值并返回裁剪后的值。

//在 1 到 3 之间裁剪 10，

//打印 3 到控制台

```
prnt(Mathf.Clamp(10,1,3))
```

◆ static function Clamp01(value;float):float

描述：在 0 和 1 之间裁剪值并返回裁剪后的值。

//随着时间设置变换位置

//但是不会小于 0 或大于 1

```
function Update(){
transform.position.x=Mathf.Clamp01(Time.time);
}
```

◆ staic function ClosestPowerOflwo(value:int):int

描述：返回最直接的 2 的幂次值。
例如。7 返回 8 或者 19 返回 16

◆ staric function Cos(f:float):float

描述：返回弧度 f 的余弦值.

```
print(Mathf.Cos(3));
```

◆ staic function EXP(power:float):float

描述：返回 e 的特定次幂。

```
print(Mathf.Cos(6));
```

◆ static function Floor(f:float):float

描述:返回小于 f 的最大整数.

◆ static function FloorToInt(f:float):int

描述：返回小于 f 的最大整数.

◆ static function InverseLerp(from:float,to:float,value:float):float

描述：在两个值之间计算 Lerp 参数.

```
varwalkSpeed=5.0;
varunSpeed=10.0;
var speed=8.0;
//参数现在是 3/5
vvar parameter=Mathf.laverseLerp(walkSpeed,runSpeed,speed);
```

◆ staic tunchon Lerp(a:float,b:float,t:float):float

描述：基本 t 在 a 到 b 之间插值。t 本裁剪到 0 到 1 之间。
当 t 为 0 时返回 from . 当 t 为 1 时返回 to .当 t=0.5 时返回 a 和 b 的平均。

```
var minimum = 10.0;
var maximum = 20.0;
//  在一秒内从 minimum 渐变到 maximum
Function Update()
{
transform.position.x = Mathf.lerp(minimum,maximum,Time.time);
}
```

◆ static function LerpAngle(a:float,b:float,t:float):float

描述：与 Lerp 相同，但是当它们可绕 360 度时确保插值正确，
变量 a 和 b 被认为是度。

```
//以 2 为底 6 的对数
//打印 2.584963
print(Mathf.Log(6,2));
```

◆ static function Log(f:float):float

描述：返回一个数以自然（以 e 为底）对数。

```
//10 的自然对数
//打印 4.60517
print(Mathf.Log(10));
```

◆ static function Log10(f:float):float

描述：返回一个数以 10 为底的对数。

```
//以 10 为底 100 的对数
//打印 2
```

print(Mathf.Log10(100));

◆ static function Max(a:float,b:float):float

描述：返回两个值中较大的一个。

//打印 2

print(Mathf.Max(1,2));

◆ static function Max(a:int,b:int):int

描述：返回两个值中较大的一个。

//打印 2

print(Mathf.Max(1,2));

◆ static function Min(a:float,b:float):float

描述：返回两个值中较小的一个。

//打印 1

print(Mathf.Min(1,2));

◆ static function Min(a:int,b:int):int

描述：返回两个值中较小的一个。

//打印 1

print(Mathf.Min(1,2));

◆ static function PingPong(t:float,length:float):float

描述：来回改变 t 值，t 值不会超过 length，也不会小于 0，只会在 length 和 0 之间循环。

```
function Update()
{
//让 x 位置在 0 到 3 之间循环
transform.position.x = Mahtf.PingPong(Time.time,3);
}
```

◆ static function Pow(f:float,p:float):float

描述：返回 f 的 p 次方。

print(Mahtf.Pow(6,1.8));

◆ static function Repeat(t:float,length:float):float

描述：使 t 值循环，不大于 length 不小于 0。它与操作模板类似，但可以使用浮点数。

```
function Update()
{
//让 x 位置在 0 到 3 之间循环
transform.position.x = Mahtf.Repeat(Time.time,3);
}
```

◆ static function Sign(f:float):float

描述：返回 f 的符号。

当 f 为正或为 0 则返回 1，为负返回-1。

◆ static function Sin(f:float):float

描述：返回以 f 为弧度的 sin 值。

print(Mahtf.Sin(3));

◆ static function SmoothDamp(current:float,target:float,ref currentVelocity:float,smoothTime:float,maxSpeed:float = Mahtf.Infinity,deltaTime:float =Time.deltaTime):float

描述：逐步的向期望值变化。

这个值就像被一个不会崩溃的弹簧防震器所影响。这个函数可以用来平滑任何类型的值，位置，颜色，标量。最常用于让一个跟随摄像机的速度变的平滑。

current 就是当前位置。**target** 是我们希望达到的位置。**currentVelocity** 是当前速度，这个值在你访问这个函数的时候会被随时修改。**smoothTime** 是要到达目标位置的近似时间，实际到达目标时要快一些。**maxSpeed** 可以让你随意的设定最大速度。**deltaTime** 是上次访问该函

数到现在的时间。缺省为 **Time.deltaTime**。

```
//平滑到目标高度
var target : Transform;
var smoothTime = 0.3;
private var yVelocity = 0.0;
function Update ()
{
var newPosition = Mathf.SmoothDamp(transform.position.y, target.position.y,yVelocity, smoothTime);
transform.position.y = newPosition;
}
```

◆ **Static function SmoothDampAngle(current: float, target: float, ref currentVelocity): float, smoothTime: float, maxSpeed: float=Mathf.Infinity, deltaTime: float=Time.deltaTime): float**

描述： 基于 **Game Programming Gems4** 章节 **1.10**

随着时间逐渐的改变一个角度为目的的角度。这个值被像弹簧阻尼一样的函数平滑。这个函数可以用来平滑任何一种值，位置，颜色，标量。最常见的是平滑一个跟随摄像机。

current 是当前位置。**target** 是我们试图到达的位置。**currentVelocity** 是当前速度，这个值在每次你调用这个函数的时候都被修改。**smoothTime** 是到达目的地近似时间，实际的时间将更短。**maxSpeed** 为允许的最大速度。**deltaTime** 为从上次调用该函数到现在的时间。缺省为 **Time.deltaTime**。

```
//一个简单的平滑跟随摄像机。
//跟随目标的朝向
var target : Transform;
var smooth = 0.3;
var distance = 5.0;
private var yVelocity = 0.0;
function Update ()
{
//从目前的 y 角度变换到目标 y 角度
var yAngle = Mathf.SmoothDampAngle(transform.eulerAngles.y,target.eulerAngles.y, yVelocity, smooth);
//target 的位置
var position = target.position;
//然后，新角度之后的距离便宜
position += Quaternion.Euler(0, angle, 0) * Vector3 (0, 0, -distance);
//应用位置
```

transform.position = position;

//看向目标

transform.LookAt(target);

}

◆ static function SmoothStep (from : float, to : float, t : float) : float

描述：在 min 与 max 中插值并在限定处渐入渐出

◆ static function Sqrt (f : float) : float

描述：返回 f 的平方根

print(Mathf.Sqrt(10));

◆ static function Tan (f : float) : float

描述：返回弧度 f 的正切值

print(Mathf.Tan(0.5));

Matrix4x4

一个标准的 4x4 变换矩阵。

一个变换矩阵可以执行任意的线形 3D 变换（例如，评议，旋转，缩放，切边等等）并且偷师变化使用齐次坐标。脚本中很少使用矩阵：最常用 Vector3，Quaternion，而且 Transform 类的功能更简单。单纯的矩阵用于特殊情况，如设置非标准相机投影。

参考任何图形学教程获取关于变换矩阵的深入揭示。

在 Unity 中， Matrix4x4 被 Transform， Camera， Material 和 GL 函数使用。

变量

◆ var inverse : Matrix4x4

描述：返回该矩阵的逆（只读）

如果用原始矩阵诚意逆矩阵结果为 identity 矩阵。

如果一些矩阵以一个特定的方式变换响亮，逆矩阵可以将他们变换回去。例如 worldToLoclaMatrix 和 localToWorldMatrix 是互逆的。

◆ var this[row : int,column : int]:float

描述：访问[row,column]处的元素。

row 和 column 必须在 0 到 3 之间，矩阵是一个 4x4 的数组，你可以通过使用这个函数访问单个的元素。

注意标准数学符号-row 是第一个索引。

◆ var this[index : int]:float

描述：按顺序索引存取元素（包括在 0..15）

矩阵是一个 4x4 的数组，所以它的总数为 16,。你可以使用一维索引来存取单个元素。index 是 row+column*4

◆ var transpose : Matrix4x4

描述：返回这个矩阵的转置（只读）。

转置矩阵是将原矩阵行列交换得到的（沿主对角线翻转）

函数

◆ function GetColumn(i : int):Vector4

描述：获取矩阵的一列。

第 i 列作为 Vector4 返回，i 必须在 0 到 3 之间。

参见：SetColumn

◆ function GetRow(i : int):Vector4

描述：返回矩阵的一行。

第 i 行作为 Vector4 返回，i 必须在 0 到 3 之间。

参见：SetRow

◆ function MultiplyPoint (v : Vector3):Vector3

描述：通过这个矩阵变换位置。

返回由任意矩阵变化得到的位置 v。如果这个矩阵是一个正规的 3D 变换矩阵，使用 MultiplyPoint3x4 比它更快。MultiplyPoint 是较慢的，但是能处理投影变换。

参见：MultiplyPoint，MultiplyVector。

◆ function MultiplyPoint3x4(v : Vector3):Vector3

描述：通过这个矩阵变换位置（快）。

返回由当前变换矩阵变换得到的位置 v。这个函数是 MultiplyPoint 的快速版，但是它只能处理常规的 3D 变化。Multiplypoint 是较慢的，但是能处理投影变换。

参见：MultiplyPoint，MultiplyVector。

◆ function MultiplyVector(v : Vector3):Vector3

描述：通过这个矩阵变换方向。

这个函数类似于 MultiplyPoint，但它是变换方向而不是位置。变换方向时，只考虑矩阵的旋转部分。

参见：MultiplyPoint，MultiplyPoint3x4。

◆ function SectColumn(i : int,v:vector4):void

描述：设置矩阵的一列。

使用这个来构建一个变换矩阵，这个矩阵使用 right，up 和 forward 向量。

```
//从变换构建一个矩阵
var matrix = Matrix4x4();
//从变换构建一个矩阵
function Start ()
{
matrix.SetColumn (0, transform.right);
matrix.SetColumn (1, transform.up);
matrix.SetColumn (2, transform.forward);
var p = transform.position;
matrix.SetColumn (3, Vector4 (p.x, p.y, p.z, 1));
}
```

//设置第 i 列为 v。i 必须在 0 到 3 之间。

参见：GetColumn

◆ function SetRow(i : int,v:Vector4):void

描述：设置矩阵的一行。

设置第 i 行为 v。i 必须在 0 到 3 之间。

参见：GetRow

◆ function SetTRS(pos : Vector3, q:Quaternon, s:Vector3):void

描述：设置这个矩阵为一个变换，旋转和缩放矩阵。

当前的矩阵本修改一遍具有位置 pos，旋转 q 和缩放 s。

◆ function ToString():string

描述：返回已格式化的该矩阵的字符串。

类变量

◆ static var identity : Matrix4x4

描述：返回单位矩阵（只读）。

这个矩阵在使用的时候不会影响任何东西。它的主对角线上全是 1，其他位置全是 0。

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

参见：zero 变量。

◆ **static var zero : Matrix4x4**

描述：返回所有元素都为零的矩阵（只读）。

0 0 0 0

0 0 0 0

0 0 0 0

0 0 0 0

参见：identity 变量。

类方法

◆ **static operator * (lhs : Matrix4x4,rhs : Matrix4x4) : Matrix4x4**

描述：两个矩阵相乘。

返回 lhs * rhs。

◆ **static operator * (lhs : Matrix4x4,v:Vector4):Vector4**

描述：由矩阵来变换一个 Vector4。

◆ **static function Ortho(left : float,right : float,bottom : float,top : float,zNear : float,zFar : float):Matirx4x4**

描述：创建一个正交投影矩阵。

返回的矩阵是视口 ledt 到 right，bottom 到 top 的区域，zNear 和 zFar 深度裁剪面板。

如果你想使用正交来做像素修正渲染，最好使用 GL.LoadPixelMatrix，因为它会为 Direct3D 渲染器运用适当的 half-texel 便宜。

参见：GL.LoadPixelMatrix，GL.LoadProjectionMatrix，GUI.matrix

◆ **static function Perspective(fov : float,aspect : float,zNear : float,zFar : float):Matrix4x4**

描述：创建一个透视投影矩阵。

fov 为透视矩阵的垂直视野，aspect 为宽高比，zNear 和 zFar 设置为深度裁剪面板。

参见：GL.LoadPixelMatrix，GL.LoadProjectionMatrix，GUI.matrix

◆ **static function Scale(v:Vector3):Matrix4x4**

描述：穿件一个缩放矩阵。

返回沿着坐标轴被响亮 v 缩放的矩阵，该矩阵看起来像这样：

v x 0 0 0

0 v y 0 0

0 0 v z 0

0 0 0 1

◆ **static function TRS(pos:Vector3,q:Quaternion,s:Vector3):Matrix4x4**

描述：创建一个变换，旋转和缩放矩阵。

返回的矩阵具有位置 pos，旋转 q 和缩放 s。

NetworkMessageInfo

这个数据结构包含一个刚收到的来自网络的消息。

它揭示了它从哪来，发送的时间和由什么网络视图发送的。

变量

- **var networkView : NetworkView**

描述：发送这个消息的 **NetworkView**

- **var sender : NetworkPlayer**

  描述：发送这个网络信息（拥有者）的玩家。

- **var TimeStamp : double**

描述：当消息被发送时的时间戳，以秒计。

时间邮戳可以用于实现插值或者连续刘宝的预测，时间戳被作为双精度数传递，以避免游戏运行长一段时间后溢出。内置的时间戳被设置为 **32** 位整数，以毫秒为精度以便节省带宽。时间戳现对于 **Network.time** 自动调整。因此 **Network.time-messageInfo.timeStamp** 是抱在传输时花费的时间。

```
var something : float;
var transitTime: double;
function OnSerializeNetworkView (stream : BitStream, info : NetworkMessageInfo) {
var horizontalInput : float = 0.0;
if (stream.isWriting) {
// 发送
horizontalInput = transform.position.x;stream.Serialize (horizontalInput);
} else {
// 接收
transitTime = Network.time - info.timestamp;
stream.Serialize (horizontalInput);
something = horizontalInput;
}
}
function OnGUI() {
GUILayout.Label("Last transmission time: "+ transitTime);
}
```

NetworkPlayer

NetworkPlayer 是一个数据结构，通过它你可以通过网络定位其他玩家。

例如，你可以直接发送一个消息给其他玩家。

变量

- **var externalIP : string**

描述：返回网络接口的外部 **IP** 地址。这个只能在建立了外接连接以后才能输入。

- **var externalPort : int**

描述：返回网络接口的外部接口。这个只能在建立了外接连接以后才能输入。

- **var ipAddress : string**

描述：该玩家的 **IP** 地址。

- **var port : int**

描述：该玩家的端口号。

构造函数

- **static function NetworkPlayer(ip : string , port : int) : NetworkPlayer**

描述：

函数

◆ function ToString() : string

描述：返回该网络玩家的指数。

类方法

◆ static operator != (lhs : NetworkPlayer,rhs : NetworkPlayer):bool

描述：如果两个 NetworkPlayers 不是同一个玩家返回真

◆ static operator == (lhs : NetworkPlayer,rhs : NetworkPlayer):bool

描述：如果两个 NetworkPlayers 是同一个玩家返回真

NetworkPlayer

NetworkPlayer 是一个数据结构，通过它你可以通过网络定位其他玩家。例如，你可以直接发送一个消息给其他玩家。

变量

◆ var externalIP : string

描述：返回网络接口的外部 IP 地址。这个只能在建立了外接连接以后才能输入。

◆ var externalPort : int

描述：返回网络接口的外部接口。这个只能在建立了外接连接以后才能输入。

◆ var ipAddress : string

描述：该玩家的 IP 地址。

◆ var port : int

描述：该玩家的端口号。

构造函数

◆ static function NetworkPlayer(ip : string , port : int) : NetworkPlayer

描述：

函数

◆ function ToString() : string

描述：返回该网络玩家的指数。

类方法

◆ static operator != (lhs : NetworkPlayer,rhs : NetworkPlayer):bool

描述：如果两个 NetworkPlayers 不是同一个玩家返回真

◆ static operator == (lhs : NetworkPlayer,rhs : NetworkPlayer):bool

描述：如果两个 NetworkPlayers 是同一个玩家返回真

NetworkViewID

在一个多玩家游戏中，NetworkViewID 是用于网络视实例的唯一标识符。这个是重要的，因为这个在所有客户端是唯一的数字，并且客户端自己可以产生这些数字，否则网络同步将被断开。

变量

◆ var isMine : bool

描述：如果是被我实例化的，返回真。

◆ var owner : NetworkPlayer

描述：拥有 NetworkView 的 NetworkPlayer。可以是服务器。

函数

◆ function ToString() : string

描述：返回 NetworkViewID 中的格式化字符串细节。

类方法

◆ static operator != (lhs : NetworkViewID,rhs : NetworkViewID):bool

描述：如果两个 NetworkViewIDs 不是同一个玩家返回真

◆ static operator == (lhs : NetworkViewID,rhs : NetworkViewID):bool

描述：如果两个 NetworkViewIDs 是同一个玩家返回真

Network

类

网络类是网络实现的核心并提供核心函数。

这个类定义了网络接口和所有网络参数。你可以使用它来设置一个服务器或链接到一个服务器并有一些列辅助函数来帮助你完成这些功能。获取更多关于编辑器中的信息请参考 Network Maner component reference。

消息传递

◆ function OnConnectedToServer() : void

描述：当成功链接到服务器上时，在客户端调用这个函数。

```
function OnConnectedToServer() {
Debug.Log("Connected to server");
}//发送本地玩家名称到服务器
```

◆ function OnDisconnectedFromServer(mode : NetworkDisconnection):void

描述：客户端从服务器上断开时在客户端上调用，但当连接被断开时在服务器上调用。

当链接丢失或被服务器断开时，在客户端调用这个函数。NetworkDisconnection 枚举将标示连接是否断开或是否连接丢失。连接成功断开时再服务器上调用这个函数（在 Network.Disconnect 之后）。

```
function OnDisconnectedFromServer(info : NetworkDisconnection) {
if (Network.isServer) {
Debug.Log("Local server connection disconnected");
}
else {
if (info == NetworkDisconnection.LostConnection)
Debug.Log("Lost connection to the server");
else
Debug.Log("Successfully diconnected from the server");
}
}
```

◆ function OnFailedToConnect(error : NetworkConnectionError):void

描述：当连接因为某些原因失败时，在客户端上调用该函数。

失败的原因作为 NetworkConnectionError 枚举传入。

```
function OnFailedToConnect(error: NetworkConnectionError){
Debug.Log("Could not connect to server: "+ error);
}
```

◆ function OnFailedToConnectToMasterServer(error : NetworkConnectionError):void

描述：当连接到主服务器出现问题时，在客户端或服务器端调用该函数。

错误原因作为 NetworkConnectionError 枚举传入。

```
function OnFailedToConnectToMasterServer(info: NetworkConnectionError){
Debug.Log("Could not connect to master server: "+ info);
```

}

◆ function OnNetworkInstantiate(info : NetworkMessageInfo):void

描述：当一个物体使用 NetworkInstantiate 进行网络初始化时在该物体上调用这个函数。这个对于禁用或启用一个已经初始化的物体组件来说是非常有用的，它们的行为取决于他们是本地还是远端。注意：在 NetworkMessageInfo 里的 networkView 属性不能在 OnNetworkInstantiate 里使用。

```
function OnNetworkInstantiate (info : NetworkMessageInfo) {
Debug.Log("New object instantiated by " + info.sender);
}
```

描述：当一个新的玩家成功连接时再服务器上调用这个函数。

```
private var playerCount: int = 0;
function OnPlayerConnected(player: NetworkPlayer) {
Debug.Log("Player " + playerCount++ + " connected from " + player.ipAddress + ":" + player.port);
}// 用玩家信息构建一个数据结构
```

◆ function OnPlayerDisconnected(player : NetworkPlayer) : void

描述：当玩家从服务器断开时再服务器上调用这个函数。

```
function OnPlayerDisconnected(player: NetworkPlayer) {
Debug.Log("Clean up after player " + player);
Network.RemoveRPCs(player);
Network.DestroyPlayerObjects(player);
}
```

◆ function OnSerializeNetworkView(stream : BitStream, info : NetworkMessageInfo) : void

描述：用来在一个被网络视架空的抄本中自定义变量同步。

它自动决定被序列化的变量是否应该发送或接收。这个依赖于谁拥有这个物体，例如，拥有者发送，而其他所有的接收。

```
//该物体的生命值信息
var currentHealth : int;
function OnSerializeNetworkView(stream : BitStream, info : NetworkMessageInfo){
var health : int = 0;
if (stream.isWriting){
health = currentHealth;
stream.Serialize(health);
}
else{
stream.Serialize(health);
currentHealth = health;
}
}
```

◆ function OnServerInitialized() : void

描述：当 Network . InitializeServer 被调用并完成时，在服务器上调用这个函数。

```
function OnServerInitialized() {
Debug.Log("Server initialized and ready");
```

}

类变量

- static var connections : NetworkPlayer[]

描述：所有连接上的玩家。在客户端中，该变量只包含服务器。

```
function OnGUI() {
if (GUILayout.Button ("Disconnect first player")) {
if (Network.connections.length > 0) {
Debug.Log("Disconnecting:
"+Network.connections[0].ipAddress+":"+Network.connections[0].port);
Network.CloseConnection(Network.connections[0], true);
}
}
}
```

- static var ConnectionTesterIP:string

描述：用在 Network.TestConnection 中的连接测试的 IP 地址。

```
function ResetIP() {
Network.connectionTesterIP = "127.0.0.1";
Network.connectionTesterPort = 10000;
}
```

- static var connectionTesterPort:int

描述：用在 Network.TestConnection 中的连接测试的 IP 端口。

```
function ResetIP() {
Network.connectionTesterIP = "127.0.0.1";
Network.connectionTesterPort = 10000;
}
```

- static var incomingPassword:string

描述：为这个服务器设置密码（对于进入的连接）。这个必须与客户端上 Network.Connect 中的相同，传递”” 表示没有密码（默认）。

```
function ConnectToServer () {
Network.Connect("127.0.0.1", 25000, "HolyMoly");
}
function LaunchServer () {
Network.incomingPassword = "HolyMoly";
Network.InitializeServer(32, 25000);
}
```

- static var isClinet:bool

描述：如果你的端类型是客户端则返回真。

```
function OnGUI() {
if (Network.isServer)
GUILayout.Label("Running as a server");
else if (Network.isClient)
GUILayout.Label("Running as a client");
}
```

- static var isMessageQueueRunning:bool

描述：启用或禁用网络消息处理。如果这个被禁用，没有 RPC 调用或网络视同步会替代。Network level loading 有如何使用这个函数的例子。

◆ static var isServer:bool

描述如果你的端类型是服务器端则返回真。

```
function OnGUI() {
if (Network.isServer)
GUILayout.Label("Running as a server");
else if (Network.isClient)
GUILayout.Label("Running as a client");
}
```

◆ static var maxConnections:int

描述：设置允许的连接/玩家的最大数量。设置 0，以为这没有心的连接可以被建立。但现有保持连接。设置为-1 表示最大连接数被设置为与当前开发的连接数相同。在这种情况下，如果一个玩家掉线，那么这个空位还是为他开放的。这个不能设置为高于 Network.InitializeServer 设置的连接数。

```
function StartGameNow() {
// 不允许更多玩家
Network.maxConnections = -1;
}
```

◆ static var minimumAllocatableViewIDs:int

描述：在 ViewIDc 池中获取或设置服务器分配给客户端 ViewID 的最小数。当玩家使用新的数字连接并被刷新时，ViewID 池被分配给每个玩家。服务器和客户端应该同步这个值。在服务器上设置的更高，将会发送比它们真正需要的更多视 ID 数到客户端。在客户端上设置更高，意味着它们需要更多视 ID。例如当池需要的 ID 数，服务器中并不包含足够的数量，则会在一行中使用两次。默认值为 100。如果一个游戏通过网络实例化大量新的物体，例如每秒超过 100 个的网络实例，那么这个值需要被设置的更高。

```
function Awake () {
// 使用更大的视 ID 池来分配
Network.minimumAllocatableViewIDs = 500;
}
```

◆ static var natFacilitatorIP:string

描述：NAT 穿透辅助的 IP 地址。通常这与服务器相同。

```
function ResetIP() {
Network.natFacilitatorIP = "127.0.0.1";
Network.natFacilitatorPort = 10001;
}
```

static var natFacilitatorPort:int

描述：NAT 穿透辅助的端口。

```
function ResetIP() {
Network.natFacilitatorIP = "127.0.0.1";
Network.natFacilitatorPort = 10001;
}
```

◆ static var peerType:NetworkPeerType

描述：端类型状态。例如，断开连接，连接，服务器或客户端。

```
function OnGUI() {
if (Network.peerType == NetworkPeerType.Disconnected)
GUILayout.Label("Not Connected");
else if (Network.peerType == NetworkPeerType.Connecting)
GUILayout.Label("Connecting");
else
GUILayout.Label("Network started");
}
```

◆ static var player:NetworkPlayer

描述：获取本地 NetworkPlayer 实例。

◆ static var ProxyIP:string

描述：代理服务器的 IP 地址。

```
function Awake(){
Network.proxyIP = "1.1.1.1";
Network.proxyPort = 1111;
}
```

◆ static var proxyPassword:string

描述：设置代理服务器密码。可以制作你自己的代理服务器。在这种情况下，你也许想用密码保护它。然后 Unity 玩家必须正确的设置这个值。

```
function Awake(){
//设置自定义代理服务器地址和密码
Network.proxyIP = "1.1.1.1";
Network.proxyPort = 1111;
Network.proxyPassword = "secret";
}
```

◆ static var proxyPort:int 描述：代理服务器的端口。

```
function Awake(){
Network.proxyIP = "1.1.1.1";
Network.proxyPort = 1111;
}
```

◆ static var sendRate:float 描述：用于所有网络视的默认网络更新发送速率。

```
function Awake () {
//增加默认的发送速率
Network.sendRate = 25;
}
```

◆ static var sendRate:float

描述：获取当前网络时间（秒）。这个可以用来比较 NetworkMessageInfo 中返回的时间。这个实例脚本需要附加到一个带有网络视的物体上，并使网络视监视这个脚本。它管理时间，发送这个物体的同步 X 位置消息。

```
var something : float;
var transitTime: double;
function OnSerializeNetworkView (stream : BitStream, info : NetworkMessageInfo) {
var horizontalInput : float = 0.0;
```

```
if (stream.isWriting) {//发送
horizontalInput = transform.position.x;
stream.Serialize (horizontalInput);
}
 else {//接收
transitTime = Network.time - info.timestamp;
stream.Serialize (horizontalInput);
something = horizontalInput;
}
}
function OnGUI() {
GUILayout.Label("Last transmission time: "+ transitTime);
}
```

◆ static var useNat:bool

描述：当连接（客户端）或接收连接（服务器）时，我们应该使用 NAT 穿透吗？如果这个在在服务器上设置，只有具有 NAT 穿透的客户端才能连接到它。但是如果服务器有一个 NAT 地址，这个需要打开以便连接。有些路由器不知道如何做 NAT 穿透。因此对于这些玩家唯一的方法就是修改路由器以便打开合适的转发端口（游戏端口）。参考 Network.TestConnection 和 Network.TestConnectionNAT 获取如何自动检测端用户的方法。

```
function OnGUI() {
if (GUILayout.Button ("Start Server"))
{
//如果没有共有 IP 地址，使用 NAT 穿透
Network.useNat = !Network.HavePublicAddress();
Network.InitializeServer(32, 25002);
MasterServer.RegisterHost("MyUniqueGameType", "JohnDoes game", "l33t game for all");
}
}
```

◆ static var useProxy:bool

描述：标示是否需要代理服务器支持，在这种情况下流量通过代理服务器被延迟。代理伺服器是一种与服务器和客户端连接性问题的解决方案。当机器有一个非 NAT 穿透能力的路由器，其连接的选择非常优先。一个游戏不能没有外部连接（只有客户端在本地网络中）。通过使用代理服务器，该机器可以具有完全的连接性，但是额外的代价是所有的流量都会被延迟。一个没有 NAT 穿透能力的客户端通过代理能够连接到任何服务器，只要代理服务器正确的设置。官方并不提供代理服务器为公众使用。所以你需要自己建立代理服务器。当然，用共有 IP 地址设置代理服务器并保证有大量可用带宽是明智的。当作为客户端运行时，只要启用 Network .useProxy 就可以。想往常昂使用 Network . Connect 连接到服务器。所有通过代理服务器的流量将被延迟。服务器的外部 IP 和内部 IP 还像往常一样工作。这样如果它们位于同一网络中，客户端可以直接连接到它而不需要代理。作为一个服务器运行时，OnServerInitialized(NetworkPlayer)返回一个 NetworkPlayer 结果表明游戏服务器中转的 IP 端口，代理服务器分配给游戏服务器的端口是什么。这个是其他客户端可以连接到的 IP 端口。当连接到服务器时，客户端不会将这个服务器与其他服务器区别对待。严格的

说，它们不需要知道这个服务器得到代理服务器的帮助。当使用主服务器时，你不能只依赖于它在使用代理服务器时为服务器注册的 IP 端口。服务器使用的代理服务器的 IP 地址和端口，可以防止在数据域的注释中来发送给主服务器。从主服务器接收主机信息的客户端可以去除注释域并查看它是否能够为那个主机使用另一个可选的 IP 端口。

重要：你不应该同时为连接到它的服务器和客户端启用代理支持，会发生意想不到的事情。

```
var imaserver: boolean;
var serverIP: String;
var serverPort: int;
var serverUsesNAT: boolean;
function Awake(){
//设置自定义代理服务器地址
Network.proxyIP = "1.1.1.1";
Network.proxyPort = 1111;
if (imaserver)
StartServerWithProxy();
else
ConnectToServerWithProxy();
}
function StartServerWithProxy(){
Network.useProxy = true;
Network.InitializeServer(2,25000);
}
function OnServerInitialized(player: NetworkPlayer){
if (Network.useProxy)
Debug.Log ("Successfully started server with proxy support. We are connectable through "+ player.ipAddress + ":" + player.port);
}
function OnFailedToConnect(msg: NetworkConnectionError){
if (Network.useProxy && imaserver){
Debug.LogError("Failed to connect to proxy server: " + msg);
}
}
function ConnectToServerWithProxy(){
Network.useProxy = true;
Network.useNat = serverUsesNAT;
Network.Connect(serverIP, serverPort);
}
function OnConnectedToServer(){
Debug.Log("Connected successfully to server");
}
```

类方法

- static function AllocateViewID():NetworkViewID

描述：查询下一个可用的网络视 ID 数并分配它（保留）。这个数字又可以被赋予一个实

例化物体的网络视。注意，为了使其可正常工作，必须有一个 NetworkView 附加到这个物体，这个物体必须有这个脚本并必须使这个脚本作为它的观察属性。必须有一个 Cube 预设，带有一个 NetworkView 它监视某些东西（例如该 Cube 的 Transform）。脚本中的 cubePrefab 变量必须设置为立方体预设。使用智能的 AllocateViewID 是最简单的方法。如果有超过一个 NetworkView 附加在初始化的 Cube 上着将变得更复杂。

```
var cubePrefab : Transform;
function OnGUI (){
if (GUILayout.Button("SpawnBox")){
var viewID = Network.AllocateViewID();
networkView.RPC("SpawnBox", RPCMode.AllBuffered, viewID, transform.position);
}
}
@RPC
function SpawnBox (viewID : NetworkViewID, location : Vector3) {
//实例化本地的 prefab
var clone : Transform;
clone = Instantiate(cubePrefab, location, Quaternion.identity);
var nView : NetworkView;
nView = clone.GetComponent(NetworkView);
nView.viewID = viewID;
}
```

◆ static function CloseConnection (target : NetworkPlayer, sendDisconnectionNotification : bool) : void

描述：关闭与其他系统的连接。/target/定义连接到的那个系统将被关闭，如果我们是客户端，连接到服务器的连接将会关闭。如果我们是服务器目标玩家，将会被踢出。sednDisconnectionNotification 启用或禁用通知将被发送到另一端。如果禁用连接被丢弃，如果没有断开连接通知被发送到远端，那么之后的连接将被丢弃。

```
function OnGUI() {
if (GUILayout.Button ("Disconnect from server")) {
if (Network.connections.length == 1) {
Debug.Log("Disconnecting:
"+Network.connections[0].ipAddress+":"+Network.connections[0].port);
Network.CloseConnection(Network.connections[0], true);
}
else if (Network.connections.length == 0)
Debug.Log("No one is connected");
else if (Network.connections.length > 1)
Debug.Log("Too many connections. Are we running a server?");
}
if (GUILayout.Button ("Disconnect first player")) {
if (Network.connections.length > 0) {
Debug.Log("Disconnecting:
"+Network.connections[0].ipAddress+":"+Network.connections[0].port);
Network.CloseConnection(Network.connections[0], true);
```

```
}
}
}
```

◆ static function Connect (IP:string, remotePort:int, password:string = ""):NetworkConnectionError

描述：连接到特定的主机（IP 或域名）和服务器端口。参数是主机的 IP 地址，点 IP 地址或一个域名。remotePort，指定连接到远端机器的端口。password，它是一个可选的用于服务器的密码。这个密码必须设置为与服务器的 Network.incomingPassword 相同。

```
function ConnectToServer () {
Network.Connect("127.0.0.1", 25000);
}
```

◆ static function Connect(IPs:string[], remotePort:int, password:string = ""):NetworkConnectionError

描述：该函数与 Network.Connect 类似，但是可以接受一个 IP 地址数组。当从一个主服务器的主机信息返回多个内部 IP 地址时，IP 数据结构可以被直接传入这个函数。它将实际的连接到相应 ping 的第一个 IP（可连接）。

◆ static function Destroy (viewID : NetworkViewID) : void

描述：跨网络销毁与该视 ID 相关的物体。本地的于远端的都会被销毁。

```
var timer : float;
function Awake () {
timer = Time.time;
}
//通过网络销毁拥有该脚本的物体
//其必须具备 NetworkView 属性
function Update() {
if (Time.time - timer >
2)
{
Network.Destroy(GetComponent(NetworkView).viewID);
}
}
function Update() {
if (Time.time - timer > 2){
Network.Destroy(GetComponent(NetworkView).viewID);
}
}
```

◆ static function Destroy (gameObject : GameObject) : void

描述：跨网络销毁该物体。本地的与远端的都会被销毁。

```
var timer : float;
function Awake () {
timer = Time.time;
}//通过网络销毁拥有该脚本的物体
function Update() {
if (Time.time - timer > 2){
```

```
Network.Destroy(gameObject);
}
}
```

◆ static function DestroyPlayerObjects (playerID : NetworkPlayer) : void

描述：基于视 ID 销毁所有属于这个玩家的所有物体。这个只能在服务器上调用。例如，清理一个已断开的玩家留下的网络物体。

```
function OnPlayerDisconnected(player: NetworkPlayer) {
Debug.Log("Clean up after player " + player);
Network.RemoveRPCs(player);
Network.DestroyPlayerObjects(player);
}
```

◆ static function Disconnect (timeout : int = 200) : void

描述：关闭所有开放的连接并关闭网络接口。timeout 参数表示网络接口在未收到信号的情况下，多长时间会关闭。网络状态，入安全和密码，也会被重置。

```
function OnGUI() {
if (GUILayout.Button ("Disconnect")) {
Network.Disconnect();
MasterServer.UnregisterHost();
}
}
```

◆ static function GetAveragePing (player : NetworkPlayer) : int

描述：到给定 player 的最后平均 ping 时间，以毫秒计。如果没有发现玩家，返回-1。Ping 会每隔几秒自动发出。

```
function OnGUI() {
var i: int;
GUILayout.Label("Player ping values");
for (i=0; i < Network.connections.length; i++) {
GUILayout.Label("Player " + Network.connections[i] + " - " +
Network.GetAveragePing(Network.connections[i]) + " ms");
}
}、
```

◆ static function GetLastPing (player : NetworkPlayer) : int

描述：到给定 player 的最后平均 ping 时间，以毫秒计。

如果没有发现玩家，返回-1。Ping 会每隔几秒自动发出。

```
function OnGUI() {
var i: int;
GUILayout.Label("Player ping values");
for (i=0; i < Network.connections.length; i++) {
GUILayout.Label("Player " + Network.connections[i] + " - " +
Network.GetLastPing(Network.connections[i]) + " ms");
}
}
```

◆ static function HavePublicAddress () : bool

描述：检查该机器是否有一个公共 IP 地址。检查所有接口来获取 Ipv4 公共地址。如发现返回真。

```
function OnGUI() {
if (GUILayout.Button ("Start Server")){
// 如果没有公共 IP 地址，使用 NAT 穿透
Network.useNat = !Network.HavePublicAddress();
Network.InitializeServer(32, 25002);
MasterServer.RegisterHost("MyUniqueGameType", "JohnDoes game", "l33t game for all");
}
}
```

◆ static function InitializeSecurity () : void

描述：初始化安全层。你需要再 Network.InitializeServer 调用之后在服务器上调用这个函数。不要再客户端调用该函数你的在线游戏达到一定知名度时就有人试图作弊。你讲需要再游戏层和网络层处理这个。如果你希望使用它们，Unity 可以通过提供安全连接处理网络层。

使用 AES 加密，阻止未授权读取并阻止重复攻击，添加 CRC 来检测数据篡改，使用随机的、加密的 SYNCookies 来组织未授权登录，使用 RSA 加密保护这个 AES 密钥

大多数游戏将使用安全连接。然后，它们会向每个数据包添加 15 自己并需要时间计算，所以你也许想限制使用此功能。

```
function Start (){
Network.InitializeSecurity();
Network.InitializeServer(32, 25000);
}
```

◆ static function InitializeServer (connections:int,listenPort:int) : NetworkConnectionError

描述：初始化安全层。connections 是允许进入的连接或玩家的数量。listenPort 是我们要监听的端口。

```
function LaunchServer () {
Network.incomingPassword = "HolyMoly";
Network.InitializeServer(32, 25000);
}
```

◆ static function Instantiate (prefab:Object, position:Vector3, rotation:Quaternion, group:int):Object

描述：网络实例化预设。给定的预设将在所有的客户端上初始化。同步被自动设置所以没有额外的工作需要做。位置、旋转和网络组数值作为给定的参数。这是一个 RPC 调用，因此

当 Network.RemoveRPCs 为这个组调用的使用，这个物体将被移除。注意在编辑器中必须设置 playerPrefab，你能在 Object.Instantiate 物体参考中获取更多实例化信息。

```
//当成功连接到服务器上时
//立即实例化新连接的玩家角色
var playerPrefab : Transform;
function OnConnectedToServer (){
```

Network.Instantiate(playerPrefab, transform.position, transform.rotation, 0);

}

◆ static function RemoveRPCs (playerID : NetworkPlayer) : void

描述：移除所有属于这个玩家 ID 的 RPC 函数。

function OnPlayerDisconnected(player: NetworkPlayer) {

Debug.Log("Clean up after player " + player);

Network.RemoveRPCs(player);

Network.DestroyPlayerObjects(player);

}

◆ static function RemoveRPCs (playerID : NetworkPlayer, group : int) : void

描述：移除所有属于这个玩家 ID 并给予给定组的所有 RPC 函数。

◆ static function RemoveRPCs (viewID : NetworkViewID) : void

描述：移除所有与这个视 ID 数相关的 RPC 函数调用。

◆ static function RemoveRPCsInGroup (group : int) : void

描述：移除所有属于给定组数值的 RPC 函数。

◆ static function SetLevelPrefix (prefix : int) : void

描述：设置关卡前缀，然后所有网络视 ID 都会使用该前缀。此处提供了一些保护，可以防止来自前一个关卡的旧的网络更新影响新的关卡。此处可以设置为任何数字并随着新关卡的加载而增加。这不会带来额外的网络负担，只会稍微减小网络视 ID 池。Network level loading 有如何使用该函数的例子。

◆ static function SetReceivingEnabled (player : NetworkPlayer, group : int, enabled : bool) : void

描述：启用或禁用特定组中来自特定玩家的信息接收。在你不希望任何网络消息进入的时候可以使用这个函数，然后在你准备好的时候启用。例如，可用于停止网络消息，知道关卡被加载。

//停止接收来自所有玩家（客户端）的组 0 的信息

for (var player : NetworkPlayer in Network.connections)

Network.SetReceivingEnabled(player, 0, false);

◆ static function SetSendingEnabled (group : int, enabled : bool) : void

描述：启用或禁用特定网络组上的信息传输和 RPC 调用。当你知道你不发送任何有用的信息到其他客户端时，可以设置这个。例如在你完全你家在关卡之前。Network level loading 中有例子。

static function SetSendingEnabled (player : NetworkPlayer, group : int, enabled : bool) : void

描述：基于目标玩家和网络组启用或禁用消息传说和 RPC 调用。当在客户端使用时，唯一可能的 NetworkPlayer 就是服务器。

◆ static function TestConnection (forceTest : bool = false) : ConnectionTesterStatus

描述：测试这个机器的网络连接。执行两种测试，这取决机器有公用 IP 还是只有一个私有 IP。公用 IP 测试主要用于服务器，不需要测试具有公用地址的客户端。为了公用 IP 测试成功，必须开启一个服务器实例。一个测试服务器将尝试连接到本地服务器的 IP 地址和端口，因此它被显示在服务器中位可连接状态。如果不是，那么防火墙是最有可能阻断服务端口的。服务器实例需要运行以便测试服务器能连接到它。另一个试验检测 NAT 穿透能力。服务器和客户端都可以进行，无需任何事先设定。如果用于服务器 NAT 测试失败，那么不设置端口转发是一个坏主意。本地 LAN 网络之外的客户端将不能连接。如果测试失败，

客户端就不能使用 NAT 连接到服务器，这些服务器将不会提供给用户作为主机。这个函数是异步的，并可能不会返回有效结果。因为这个测试需要一些时间来完成（1-2 秒）。测试完成后，测试的结果只在函数被再次调用时返回。这样，频繁访问该函数是安全的。如果需要其他的测试，入网络连接已改变，那么 forcTest 参数应该为真。该函数返回一个 ConnectionTesterStatus 枚举。

```
//是否应该在主机列表上隐藏 NAT 主机？
private var filterNATHosts = false;
private var doneProbingPublicIP = false;
function OnGUI (){
//开始/轮询连接测试
//在标签上显示结果并按照结果做出相应的反应
natCapable = Network.TestConnection();
if (natCapable == -2)
GUILayout.Label("Problem determining NAT capabilities");
else if (natCapable == -1)
GUILayout.Label("Undetermined NAT capabilities");
else if (natCapable == 0){
GUILayout.Label("Cannot do NAT punchthrough, " +"filtering NAT enabled hosts for client connections, " +"impossible to run a server.");
filterNATHosts = true;
Network.useNat = false;
}
else if (natCapable == 1){
if (doneProbingPublicIP)
GUILayout.Label("Non-connectable public IP address (port "+ serverPort +" blocked), NAT unchthrough can circumvent the firewall.");
else
GUILayout.Label("NAT punchthrough capable. " +"Enabling NAT punchthrough functionality.");
//一旦服务器开始 NAT 功能被启用
//客户端是否开启这个基于主机是否需要
Network.useNat = true;
}
else if (natCapable == 2){
GUILayout.Label("Directly connectable public IP address.");
Network.useNat = false;
}
else if (natCapable == 3){
GUILayout.Label("Non-connectble public IP address (port " + serverPort +" blocked), running a
server is impossible.");
Network.useNat = false;
if (!doneProbingPublicIP){
natCapable = Network.TestConnectionNAT();
```

doneProbingPublicIP = true;
}
}
else if (natCapable == 4){
GUILayout.Label("Public IP address but server not initialized, "+"it must be started to check server accessibility.");
Network.useNat = false;
}
if (GUILayout.Button ("Retest connection")){
Debug.Log("Redoing connection test");
doneProbingPublicIP = false;
natCapable = Network.TestConnection(true);
}
}

◆ static function TestConnectionNAT () : ConnectionTesterStatus

描述：测试 NAT 穿透的连接性。这个就像 Network.TestConnection，只不过 NAT 穿透是强制的，即使该机器有一个公用地址。请参考 Network.TestConnection。

Object

类

Unity 所涉及的所有物体的基类。任何从 Object 继承的公有变量将作为一个目标显示在监视面板中，允许你从 GUI 中设置。

变量

◆ var hideFlags : HideFlags 描述：该物体是否被隐藏，保存在场景中或被用户修改。

◆ var name : string

描述：对象的名称。组件与游戏物体和所有附加的组件共享相同名称。

//改变物体的名称为 Hello
name = "Hello";

函数

◆ function GetInstanceID () : int 描述：返回该物体的实例 id。一个物体的实例 ID 总是唯一。

描述：返回该物体的实例 id。

一个物体的实例 id 总是唯一的。

print(GetInstanceID());

类方法

◆ static function Destroy (obj : Object, t : float = 0.0F) : void

描述：移除一个游戏物体，组件或资源。物体 obj 将被小火或者 t 秒后被销毁。如果 obj 是一个 Component 它将被从 GameObject 中移除。如果 obj 是一个 GameObject 它将销毁这个 GameObject，以及它的组件和所子对象。实际的销毁总是推迟到下个 Update 来临时，但总在渲染前完成。

//销毁这个游戏物体
Destroy (gameObject);
//从物体上移除该脚本
Destroy (this);

```
//从游戏物体上移除刚体组件
Destroy (rigidbody);
//加载该游戏物体后 5 秒删除
Destroy (gameObject, 5);
//当玩家按下 Ctrl 时移除名为 FoooScript 的脚本
function Update () {
if (Input.GetButton ("Fire1") && GetComponent (FooScript))
Destroy (GetComponent (FooScript));
}
```

◆ **static function DestroyImmediate (obj : Object, allowDestroyingAssets : bool = false) : void**

描述：立即销毁物体。强烈建议使用 **Destroy** 代替它。该函数应该只在编写编辑器代码时使用，因为延迟的销毁将不会再编辑器模式调用。游戏代码中建议使用 **Destroy**。**Destroy** 总是延迟的（但是在同一帧执行）。小心使用该函数，因为它能永久的销毁资源。

◆ **static function DontDestroyOnLoad (target : Object) : void**

描述：加载新场景时确保物体 **target** 不被自动销毁。当加载一个新的关卡时，场景中的所有物体都会被销毁，然后心关卡中的物体将被加载。为了在关卡加载的时候保持物体在上面调用 **DontDestroyOnLoad**。如果物体是一个组件或游戏物体，那么它的整个变换层次将不会被销毁。

```
//保证该游戏物体及其变化子物体在载入新场景时不会被销毁。
function Awake () {
DontDestroyOnLoad (this);
}
```

◆ **static function FindObjectOfType (type : Type) : Object**

描述：返回第一个类型为 **Type** 的已激活加载的物体。参见 **Object.FindObjectsOfType** 。

◆ **static function FindObjectsOfType (type : Type) : Object[]**

描述：返回所有类型为 **Type** 的已激活加载的物体。

参见 **Object.FindObjectsOfType** 。

它将返回任何资源（网格、纹理、预设等）或已激活加载的物体。

```
//当点击该物体，它将禁用场景中所有铰链中的弹簧。
function OnMouseDown () {
hinges = FindObjectsOfType (HingeJoint);
for (var hinge : HingeJoint in hinges) {
hinge.useSpring = false;
}
}
c#版
public class Something : MonoBehaviour{
void OnMouseDown(){
HingeJoint[] hinges = FindObjectsOfType<HingeJoint>();
for (HingeJoint hinge in hinges) {
hinge.useSpring = false;
}
```

```
}
}
```

◆ **static function Instantiate (original : Object, position : Vector3, rotation : Quaternion) : Object**

描述：克隆 original 物体并返回该克隆。防御 position 兵设置旋转为 rotation，然后返回该克隆。本职上与 cmd-d 相同，并移动到给定位置。如果一个游戏物体、组件或脚本实例被传入，Instantiate 将克隆整个游戏物体层次，所有的子对象也被克隆。所有游戏物体被激活。参加：预设实例化的深入讨论。

```
//实例化预设的 10 个拷贝，间隔为 2 单位。
var prefab : Transform;
for (var i=0;i<10;i++) {
Instantiate (prefab, Vector3(i * 2.0, 0, 0), Quaternion.identity);
}
```

Instantiate 更多常用于实例化投射物、AI 敌人，粒子爆炸或 b 破损的物体。

```
//实例化一个刚体，然后设置速度。
var projectile : Rigidbody;
function Update () {
//按下 ctrl 时，发射一个物体
if (Input.GetButtonDown("Fire1")) {
//以该变化位置与旋转实例化投射物
var clone : Rigidbody;
clone = Instantiate(projectile, transform.position, transform.rotation);
//沿当前物体 Z 轴，给克隆体一个初始速度
clone.velocity = transform.TransformDirection (Vector3.forward * 10);
}
}
```

实例化也能直接克隆脚本实例。整个游戏物体层级将被克隆，并且克隆脚本的实例将被返回。

```
//初始化一个附加了 Missile 脚本的预设
var projectile : Missile;
function Update () {
//按下 ctrl 时，发射一个物体
if (Input.GetButtonDown("Fire1")) {
//以该变化位置与传转实例化投射物
var clone : Missile;
clone = Instantiate(projectile, transform.position, transform.rotation);
//设置火箭超市销毁为 5 秒
clone.timeoutDestructor = 5;
}
}
```

克隆一个物体之后可以使用 GetComponet 来设置附加到克隆物体上的特定组件的属性。

◆ **static function Instantiate (original : Object) : Object**

描述：克隆 original 物体并返回该克隆。这个函数保留克隆物体的位置与赋值命令相同

（cmd-d）。

```
//当任何刚体进入这个触发器时实例化预设。
//它保留预设的原始位置与旋转。
var prefab : Transform;
function OnTriggerEnter () {
Instantiate (prefab);
}
```

◆ static operator != (x : Object, y : Object) : bool

描述：比较两个物体是否不同。

```
var target : Transform;
function Update (){
//如果 target 不同于我们的变换。
if (target != transform){
print("Another object");
}
}
```

◆ static operator == (x : Object, y : Object) : bool

描述：比较两个物体是否相同。

```
var target : Collider;
function OnTriggerEnter (trigger : Collider){
if (trigger == target)
print("We hit the target trigger");
}
var target : Transform;
function Update (){
//该物体已被销毁。
if (target == null)
return;
}
```

◆ static implicit function bool (exists : Object) : bool

描述：这个物体是否存在？

if（rigidbody）

等同于

if（rigidbody != null）

AnimationClip

类，继承自 Object。

存贮基于动画的关键帧。

AnimationClip 被 Animation 使用来播放动画。

变量

◆ var frameRate : float

描述：关键帧被残阳的帧率。这个与用来制作动画/模型的动画程序相同。

```
//打印动画剪辑的帧率到控制台。
print(animation["walk"].clip.frameRate);
```

◆ var length : float

描述：动画播放的描述。

```
animation.Play(animation.clip);
//等待动画完成。
yield WaitForSeconds (animation.clip.length);
```

◆ var wrapMode : WrapMode

描述：在动画状态中设定默认的卷模式。

构造函数

◆ static function AnimationClip () : AnimationClip

描述：创建一个新的动画剪辑。

函数

◆ function AddEvent (evt : AnimationEvent) : void

描述：创建一个新的动画剪辑。这将添加这个时间直到退出播放模式或玩家退出。如果你想从编辑器添加一个固定的剪辑到 AnimationEvent，使用 UnityEditorAnimationUtility.SetAnimationEvents。

◆ function ClearCurves () : void

描述：从剪辑中清理所有曲线。

◆ function SetCurve (relativePath : string, type : Type, propertyName : string, curve : AnimationCurve) : void

参数

relativePath　　应用这个曲线的游戏物体的路径。relativePath 被格式化为一个路径名。入：“root/spine/leftArm”如果 relativePath 为空，表示该动画剪辑所附加的游戏物体。

type　　被进行动画处理的组件的类类型。

propertyName　　被动画处理的属性的名称或路径。

curve　　动画曲线。

描述：给动画指定一个特定的曲线属性。如果曲线为 null 该曲线将被移除。如果曲线为 null 该曲线将被移除。如果曲线属性已经存在，则会被替换。通常的名称是：“localPostion.x”，“localPostion.y”，“localPostion.z”，“localRotation.x”，“localRotation.y”，“localRotation.z”，“localRotation.w”，“localScale.x”，“localScale.y”，“localScale.z”。出于性能考虑 Transform 的位置、旋转和缩放只能被所谓一个动画属性。

```
//对 x 坐标的位置进行动画处理
function Start (){
//创建曲线
var curve = AnimationCurve.Linear(0, 1, 2, 3);
//用曲线创建剪辑
var clip = new AnimationClip();
clip.SetCurve("", Transform, "localPosition.x", curve);
//添加并播放剪辑
animation.AddClip(clip, "test");
animation.Play("test");
}
@script RequireComponent(Animation)
```

Material 属性可以使用 shader 到处的名称制作动画属性。通常的属性名称是：“_MainTex”，“_BumpMap”，“_LightMap”，“_Color”，“_SpecColor”，“_Emmission”。

Float 属性“PropertyName”

Vector4 属性“PropertyName.x”“PropertyName.x” “PropertyName.x”“PropertyName.x” Color 属性“PropertyName.r”“PropertyName.g”“PropertyName.b”“PropertyName.a”

UV 旋转属性“PropertyName.rotation”；UB 便宜和缩放“PropertyName.offset.x”“PropertyName.offset.y”“PropertyName.scale.x”“PropertyName.scale.y”

对于在同一个 Renderer 上的多个索引材质，你可以像这样加前缀“[1]._MainTex.offset.y”。

```
//对 alpha 值和主要材质地平线补偿进行动画处理
function Start () {
var clip = new AnimationClip ();
clip.SetCurve ("", typeof(Material), "_Color.a",AnimationCurve (Keyframe(0, 0, 0, 0), Keyframe(1, 1, 0, 0)));
clip.SetCurve ("", typeof(Material), "_MainTex.offset.x",AnimationCurve.Linear(0, 1, 2, 3));
animation.AddClip (clip, clip.name);
animation.Play(clip.name);
}
@script RequireComponent(Animation)
```

继承的成员

继承的变量

name 对象名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

AssetBundle

类，继承自 Object。AssetBundles 让你通过 WWW 类流式加载额外的资源并在运行时实例化它们。AssetBundles 通过 BuildPipeline.BuildAssetBundle 创建。参见：WWW.assetBundle ，Loading Resources at Runtime ，BuildPipeline.BuildPlayer

```
function Start () {
var www = new WWW ("http://myserver/myBundle.unity3d");
yield www;
//获取指定的主资源并实例化
Instantiate(www.assetBundle.mainAsset);
```

}

变量

◆ var mainAsset : Object

描述：竹资源在构建资源 boundle 时指定（只读）。该功能可以方便的找到 bundle 内的主资源。例如，你也许想将预设一个角色并包括所有纹理、材质、网格和动画文件。但是完全操纵角色的预设应该是你的 mainAsset 并且可以被容易的访问。

```
function Start () {
var www = new WWW ("http://myserver/myBundle.unity3d");
yield www;
//获取指定的主资源并实例化
Instantiate(www.assetBundle.mainAsset);
}
```

函数

◆ function Contains (name : string) : bool

描述：如果 AssetBundle 的名称中包含特定的对象则进行检索。如果包含则返回真。

◆ function Load (name : string) : Object

描述：从 bundle 中加载名为 name 的物体。

◆ function Load (name : string, type : Type) : Object

描述：从 bundle 中加载名为 name 的 type 类物体。

◆ function LoadAll (type : Type) : Object[ ]

描述：加载所有包含在资源 bundle 中且继承自 type 的物体。

◆ function LoadAll () : Object[ ]

描述：加载包含在资源 bundle 中的所有物体。

◆ function Unload (unloadAllLoadedObjects : bool) : void

描述：写在 bundle 中的所有资源。Unload 释放 bundle 中所有序列化数据。当 unloadAllLoaderObjects 为假，bundle 内的序列化数据将被写在，但是任何从这个 bundle 中实例化的物体都将完好。当然，你不能从这个 bundle 中加载更多物体。当 unloadAllLoaderObjects 为真，所有从该 bundle 中加载的物体也将被销毁。如果

场景中有游戏物体引用该资源，那么引用也会丢失。

继承的成员

继承的变量

name 对象名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

AudioClip

类，继承自 Object。音频数据的容器。一个 AudioClip 以压缩或未压缩的格式存储音频文件。AudioClips 被 AudioSources 参考或引用来播放声音。参见组件参考的 AudioClip.component。

变量

◆ var isReadyToPlay : bool

描述：有没流式音频剪辑准备播放？（只读）如果 AudioClip 是从网站上下载的，此变量用来判断下载到的数据是否足够不间断的播放。对于不是来自 web 的流的 AudioClips，该值总是真。

```
function Start (){
www=new WWW(url);
audio.clip=www.audioClip;
}
function Update (){
if(!audio.isPlaying && audio.clip.isReadyToPlay)
audio.Play();
}
```

◆ var length : float

描述：音频剪辑的长度，以秒计（只读）。

```
audio.Play();
//等待音频播放完成
yield.WaitForSeconds(audio.clip.length);
```

继承的成员

继承的变量

name　对象名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate　克隆 original 物体并返回这个克隆。

Destroy　移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator==　比较两个物体是否相同。

operator!=　比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

Component

类，继承自 Object。音频数据的容器。所有附加到游戏物体上的对象的基类。

变量

◆ var animation : Animation

描述：附加到这个 GameObject 的 Animation。（没有则为 null）

◆ var audio : AudioSource

描述：附加到这个 GameObject 的 AudioSource。（没有则为 null）

audio.Play()

◆ var camera : Camera

描述：附加到这个 GameObject 的 Camera。（没有则为 null）

◆ var collider : Collider

描述：附加到这个 GameObject 的 Collider。（没有则为 null）

collider.material.dynamicFriction =1

◆ var constantForce : ConstantForce

描述：附加到这个 GameObject 上的 ConstantForce（没有则为 null）。

◆ var gameObject : GameObject

描述：这个组件所附加的游戏物体。组件总是附着在游戏物体上。

print(gameObject.name);

◆ var guiText : GUIText

描述：附加到这个 GameObject 上的 GUIText（没有则为 null）。

guiText.text = "Hello World";

◆ var guiTexture : GUITexture

描述：附加到这个 GameObject 上的 GUITexture（只读）（没有则为 null）。

◆ var hingeJoint : HingeJoint

描述：附加到这个 GameObject 的 HingeJoint（没有则为 null）。

hingeJoint.motor.targetVelocity = 5;

◆ var hingeJoint : HingeJoint

描述：附加到这个 GameObject 的 HingeJoint（没有则为 null）。

◆ var light : Light

描述：附加到这个 GameObject 的 Light（没有则为 null）。

◆ var networkView : NetworkView

描述：附加到这个 GameObject 的 NetworkView（只读）（没有则为 null）。

networkView.RPC("MyFunction", RPCMode.All, "someValue");

◆ var particleEmitter : ParticleEmitter

描述：附加到这个 GameObject 的 ParticleEmitter（没有则为 null）。

particleEmitter.emit = true;

◆ var renderer : Renderer

描述：附加到这个 GameObject 的 Renderer。（没有则为 null）

◆ var rigidbodyr : Rigidbody

描述：附加到这个 GameObject 的 rigidbody。（没有则为 null）

◆ var tag : string

描述：附加到这个 GameObject 的 rigidbody。标签可以用来标识一个游戏物体。标签在使用前必须在标签管理器中定义。

◆ var transform : Transform

描述：附加到这个 GameObject 的 Transform。（没有则为 null）

transform.Translate(1, 1, 1);

函数

◆ function BroadcastMessage (methodName : string, parameter : object = null, options :

SendMessageOptions = SendMessageOptions.RequireReceiver) : void

描述：在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName 方法。通过使用零参数，结婚搜方法可以选择忽略 parameter。如果 options 被设置为 SednMessageOptions.RequireReceiver，那么如果这个消息没有被任何组件接受时，将打印一个错误信息。

```
//使用值 5 调用 ApplyDamage
BroadcastMessage ("ApplyDamage", 5.0);
//每个附加到该游戏物体及其所有子物体上含有 ApplyDamage 函数的脚本都会被调用
function ApplyDamage (damage : float) {
print (damage);
}
```

◆ function CompareTag (tag : string) : bool

描述：这个游戏物体有被标签为 tag 吗？

```
//立即销毁触发器，销毁任何进入到触发器的碰撞器，这些碰撞器被标记为 Player
function OnTriggerEnter (other : Collider) {
if (other.CompareTag ("Player")) {
Destroy (other.gameObject);
}
}
```

◆ function GetComponent (type : Type) : Component

描述：如果游戏物体上附加了这个组件，则返回一个 Type 类，如果没有则返回 null。

```
//等同于 Transform curTransform = transform
var curTransform : Transform = GetComponent (Transform);
//你可以像访问其他组件一样的访问脚本组件
function Start () {
var someScript : ExampleScript = GetComponent (ExampleScript);
someScript.DoSomething ();
}
for c#
public class Something : MonoBehaviour{
void Start(){
ExampleScript someScript = GetComponent<ExampleScript>();
someScript.DoSomething ();
}
}
```

◆ function GetComponent (type : string) : Component

描述：如果游戏物体上附加了这个组件，则返回一个 Type 类，如果没有则返回 null。处于性能原因，最好用 Type 调用 GetComponent 而不是字符串。不过有时你可能无法得到 Type。例如当是同从 Javascript 中访问 c#时。这时你可以简单的通过名称而不是类型访问该组件。

```
//为了访问附加在同一物体上的脚本中的公有变量与函数
script = GetComponent(ScriptName);
script.DoSomething ();
```

◆ function GetComponentInChildren (t : Type) : Component

描述：返回 type 类型组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。只有激活的最贱会被返回。

```
var script : ScriptName = GetComponentInChildren(ScriptName);
script.DoSomething ();
for c#
ScriptName script = GetComponentInChildren<ScriptName>();
script.DoSomething ();
```

◆ function GetComponents (type : type) : Component[]

描述：返回 GameObject 上所有 type 类型组件。

```
//关闭该游戏物体铰链上的所有弹簧
var hingeJoints = GetComponents (HingeJoint);
for (var joint : HingeJoint in hingeJoints) {
joint.useSpring = false;
}
for c#
HingeJoint[] hingeJoints = GetComponents<HingeJoint>();
for (HingeJoint joint in hingeJoints) {
joint.useSpring = false;
}
```

◆ function GetComponentsInChildren (t:Type , includeInactive:bool=false) : Component[]

描述：返回 GameObject 上或其子物体上所有 type 类型组件。

```
//关闭该游戏物体极其子物体上的铰链上的所有弹簧
var hingeJoints = GetComponentsInChildren (HingeJoint);
for (var joint : HingeJoint in hingeJoints) {
joint.useSpring = false;
}
for c#
HingeJoint[] hingeJoints = GetComponentsInChildren<HingeJoint>();
for (HingeJoint joint in hingeJoints) {
joint.useSpring = false;
}
```

继承的成员

继承的变量

name 对象名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

Behaviour

类，继承自 Component。Behaviours 是可以被启用或禁用的组件。参见 MonoBehaviour 和 Component。

变量

◆ var enabled : bool

描述：启用 Behaviours 被更新，禁用 Behaviours 不被更新。这将在 behaviour 的坚实面板中显示为一个小的复选框 GetComponent(PlayerScript)enabled = false;

继承的成员

继承的变量

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate　克隆 original 物体并返回这个克隆。

Destroy　移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType返回所有类型为 type 的激活物体。

FindObjectOfType　返回第一个类型为 type 的激活物体。

operator==　比较两个物体是否相同。

operator!=　比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

Animation

类，继承自 Behaviour，可枚举。动画组件用来播放动画。你可以给动画组件赋予一个动画剪辑，并从脚本中控制它的播放。Unity 中的动画系统是基于权值的，并且支持动画混合、附加动画、动画合成、层和所有动画播放方面的完全控制。

为了播放动画可使用 Animation.Play。

为了在动画间渐变可使用 Animation.CrossFade。

为了改变动画的层可使用 AnimationState.layer。

为了改变动画的包裹模式（循环、单次、往返）可使用 Animation.wrapMode 或 AnimationState.wrapMode。

AnimationState 可以被用来调整播放速度和直接控制混合语合成。

Animation 也支持枚举，因此你可以像这样循环所有的 AnimationStates。

```
//使该角色上所有动画半速播放
for (var state : AnimationState in animation) {
state.speed = 0.5;
}
```

变量

◆ var animateOnlyIfVisible : bool

描述：启用后，Unity 会在认为动画不可见的时候停止播放。该选项可以节省资源。

◆ var animatePhysics : bool

描述：启用后，动画奖在物理循环中执行。这只在与运动学刚体结合时有用。一个动画平台可以应用速度和摩擦到其顶部的刚体。为了利用这个，animatePhysics 必须被启用，并且动画物体必须为一个运动学刚体。

◆ var clip : AnimationClip

描述：默认动画。

```
animation.Play(animation.clip);
//等待动画完成
yield WaitForSeconds (animation.clip.length);
```

◆ var isPlaying : bool

描述：是否正在播放动画？

```
//在动画没有播放时播放这个动画。
function OnMouseEnter() {
if (!animation.isPlaying)animation.Play();
}
```

◆ **var playAutomatically : bool**

描述：默认的动画剪辑（Animation.clip）是否在开始时自动播放？

animation.playAutomatically = true;

◆ **var this[name : string] : AnimationState**

描述：返回名为 name 的动画状态。

//获取 walk 动画状态并设置它的速度

animation["walk"].speed = 2.0;

//获取 run 动画状态并设置它的速度

animation["run"].weight = 0.5;

◆ **var wrapMode : WrapMode**

描述：超出剪辑播放时间之外时如何处理？

WrapMode.Default:从剪辑中获取回绕模式（默认为 **Once**）。

WrapMode.Once:当时间到达末尾时停止动画。

WrapMode.Loop:当时间到达末尾时从头开始播放。

WrapMode.PingPong:在开始和结束之间来回播放。

WrapMode.ClampForever:播放动画。当它到达末端时，它将保持在最后一帧。

//让动画循环

animation.wrapMode = WrapMode.Loop;

函数

◆ **function AddClip (clip : AnimationClip, newName : string) : void**

描述：添加一个 **clip** 到 **Animation**，它的名称为 **newName**。

var walkClip : AnimationClip;

animation.AddClip(walkClip, "walk");

◆ **function AddClip (clip : AnimationClip, newName : string, firstFrame : int, lastFrame : int, addLoopFrame : bool = false) : void**

描述：添加 **clip** 只在 **firstFrame** 和 **lastFrame** 之间播放。这个新的剪辑也将被使用名称 **newName** 并添加到 **Animation**。**addLoopFrame**：是否要插入额外的帧一边匹配第一帧？如果制作循环动画就要打开这个。如果存在一个同名的剪辑，老的将被覆盖。

//分割默认的剪辑为 **shoot**，**walk** 和 **idle** 动画

animation.AddClip(animation.clip, "shoot", 0, 10);

//walk 和 **idle** 将在末尾添加额外的循环真

animation.AddClip(animation.clip, "walk", 11, 20, true);

animation.AddClip(animation.clip, "idle", 21, 30, true);

◆ **function Blend (animation : string, targetWeight : float = 1.0F, fadeLength : float = 0.3F) : void**

描述：在接下来的 **time** 内混合名为 **animation** 的动画得到 **targetWeight**。其他动画播放不会受影响。

◆ **function CrossFade (animation : string, fadeLength : float = 0.3F, mode : PlayMode = PlayMode.StopSameLayer) : void**

描述：淡入名为 **animation** 的动画，并在 **time** 内淡出其他动画。如果模式是 **PlayMode.StopSameLayer**，在同一层上淡入的动画将会被淡出。如果模式是 **PlayMode.StopAll**，当动画被淡入后，所有动画都会被淡出。如果动画被设置为循环，在播放后将停止并回放。

//淡入 **walk** 循环并淡出同一层上所有其他动画。

//0.2 秒内完成淡入

animation.CrossFade("Walk", 0.2);

//当玩家想要移动的时候，使角色动画在 Run 和 Idle 动画之间渐变。

function Update (){

if (Mathf.Abs(Input.GetAxis("Vertical")) > 0.1)animation.CrossFade("Run");

else

animation.CrossFade("Idle");

}

◆ function CrossFadeQueued (animation : string, fadeLength : float = 0.3F, queue : QueueMode = QueueMode.CompleteOthers, mode : PlayMode = PlayMode.StopSameLayer) : AnimationState

描述：在前一动画播放完后渐变到下一个动画。例如你可以播放一个特定的动画序列。动画在播放前复制自身，因此你可以再相同的动画间渐变，这可用来重叠两个相

同的动画。例如你可能有一个挥剑的动画，玩家快速挥动了 2 次，你可以回放这个动画并从开始播放它，但会跳帧。

下面是可用的 queue modes：

下面是可用的 queue modes：

如果 queue 为 QueueMode.CompleteOthers 这个动画纸在所有其他动画都停止播放时才开始。

如果 queue 为 QueueMode.PlayNow 这个动画将以一个复制的动画状态立即开始播放。

动画播放完成后它将自动清除它自己。在它播放完成后使用赋值的动画将导致一个异常。

function Update (){

if (Input.GetButtonDown("Fire1"))

animation.CrossFadeQueued("shoot", 0.3, QueueMode.PlayNow);

}

◆ function GetClipCount () : int

描述：获取当前 animation 的剪辑数。

◆ function IsPlaying (name : string) : bool

描述：名为 name 的动画是否在播放？

function OnMouseEnter() {

if (!animation.IsPlaying("mouseOverEffect"))

animation.Play("mouseOverEffect");

}

◆ function Play (mode : PlayMode = PlayMode.StopSameLayer) : bool

function Play (animation : string, mode : PlayMode = PlayMode.StopSameLayer) : bool

描述：立即播放该动画，没有任何混合。Play()将会开始播放名为 animation 的动画，或者播放默认动画，且会在没有混合的情况下突然播放。如果模式为 PlayMode.StopSameLayer 所有在同一层上的动画都将被停止。如果模式是 PlayMode.StepAll 所有当前播放的动画都将被停止。如果动画正在播放，其他动画将被停止，但该动画会回到开始位置。如果动画没有被设置为循环，播放完后将停止并回放。如果动画无法被播放（没有该剪辑或没有默认动画），Play()会返回假。

//播放默认动画

animation.Play ();

//播放 walk 动画，停止该层内所有其他动画

animation.Play ("walk");

//播放 walk 动画，停止其他所有动画

animation.Play ("walk", PlayMode.StopAll);

◆ function PlayQueued (animation : string, queue : QueueMode = QueueMode.CompleteOthers, mode : PlayMode = PlayMode.StopSameLayer) : AnimationState

描述：在前一动画播放完后渐变一个动画。例如你可以播放一个特定的动画序列。动画状态在播放前复制自身，因此你可以再相同的动画之间渐变。这可用来重叠两个相同的动画。例如你可能有一个挥剑的动画。玩家快速的砍了两次，你可以回放这个动画，并且从开始播放，只不过会跳帧。

下面是可用的 queue modes：

如果 queue 为 QueueMode.CompleteOthers 这个动画纸在所有其他动画都停止播放时才开始。

如果 queue 为 QueueMode.PlayNow 这个动画将以一个复制的动画状态立即开始播放。

动画播放完成后它将自动清除它自己。在它播放完成后使用赋值的动画将导致一个异常。

```
function Update (){
if (Input.GetButtonDown("Fire1"))
animation.PlayQueued("shoot", QueueMode.PlayNow);
}
```

◆ function RemoveClip (clip : AnimationClip) : void

描述：从动画列表中移除剪辑。这将移除该剪辑及所有棘突它的动画状态。

◆ function RemoveClip (clipName : string) : void

描述：从动画列表中移除剪辑。这将移除指定名称的动画状态。

◆ function Rewind (name : string) : void

描述：回退名为 name 的动画。

```
//回退 walk 动画到开始
animation.Rewind("walk");
```

◆ function Rewind () : void

描述：回退所有动画。

```
//回退所有动画到开始
animation.Rewind();
```

◆ function Sample () : void

描述：在当前状态采样动画。如果想要设置一些动画状态，并采样一次，可以用这个。

```
//设置一些状态
animation["MyClip"].time = 2.0;
animation["MyClip"].enabled = true;
//采样动画
animation.Sample();
animation["MyClip"].enabled = false;
```

◆ function Stop () : void

描述：停止所有由这个 Animation 开始的动画。停止一个动画并回退到开始。

```
//终止所有动画
animation.Stop();
```

◆ function Stop (name : string) : void

描述：停止名为 name 的动画。停止一个动画并回退到开始。

//停止 walk 动画

animation.Stop ("walk");

◆ function SyncLayer (layer : int) : void

描述：同步所有在该层的动画的播放速度。当混合两个循环动画时，它们通常有不同长度。例如一个 walk 循环要比 run 更长。当混合它们时，你需要确保行走于跑循环中脚的位置相同。也就是说，动画的播放速度必须被调整一边动画同步。SyncLayer 将给予它们的混合权值计算在该层上所有动画播放的平均诡异化速度，然后对该层上的动画使用播放速度。

//放置 walk 和 run 动画在同一层，并同步它们的速度

animation["walk"].layer = 1;

animation["run"].layer = 1;

animation.SyncLayer(1);

继承的成员

继承的变量

enable 启用则 Behaviour 被更新，不启用则不更新

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用

名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

AudioListener

类，继承自 Behaviour。标示在三维空间中的侦听器。这个类可实现麦克风一样的设备。它记录周围的声音，并通过玩家的扬声器播放。在场景中只能有一个侦听器。参见组件参考中 AudioSource,AudioListener,component。

变量

◆ var velocityUpdateMode : AudioVelocityUpdateMode

描述：可让你设置 Audio Listener 是否应该用固定或动态方式更新。如果你遇到了多普勒效应问题，请确保设置这个更新于 Audio Listener 的移动在同一循环内。如果它被附加到一个刚体，默认设置将自动设置该侦听器在固定的更新周期内更新，动态的以其他方式。

```
listener.velocityUpdateMode = AudioVelocityUpdateMode.Fixed;
```

类变量

◆ static var pause : bool

描述：音频的暂停状态。如果设置为真，该侦听器将不会产生声音。类似于设置音量为 0.0

```
AudioListener.pause = true;
```

◆ static var volume : float

描述：控制游戏的音量。

```
AudioListener.volume = 0.5;
```

继承的成员

继承的变量

enable 启用则 Behaviour 被更新，不启用则不更新

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody 附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

AudioSource

类，继承自 Behaviour。标示三位空间中的音频源。AudioSource 被附加到 GameObject 以便在三位环境中播放声音。单声道声音以 3D 播放。为了播放三位声音，也需要一个 AudioListener。音频侦听器通常附加在使用的相机上。立体声总是不以距离为基础衰减。你可以使用 Play，Pause，Stop 来播放音频剪辑。也可以再播放时使用 volume 属性调整音量，或者使用 time 来定位。多个声音可以使用 PlayOneShot 在一个 AudioSource 上播放。可以使用 PlayClipAtPoint 在三位空间中的一个静态位置播放剪辑。参见 AudioListener，AudioClip，AudioSource component。

变量

- var clip : AudioClip

描述：默认播放的 AudioClip。

```
var otherClip: AudioClip;
// 播放默认声音
audio.Play();
// 等待音频结束
yield WaitForSeconds (audio.clip.length);
// 指定其他音频并播放
audio.clip = otherClip;
audio.Play();
```

◆ var ignoreListenerVolume : bool

描述：这使音频源不考虑音频侦听器的音量。播放背景音乐时可启用这个。当播放背景音乐时，你希望音乐不受普通音量设置的影响，可以使用该变量。

```
audio.ignoreListenerVolume = true;
```

◆ var isPlaying : bool

描述：clip 现在生意否在播放？

```
//当音频组件停止播放时，播放 otherClip
var otherClip : AudioClip;

function Update (){
if (!audio.isPlaying) {
audio.clip = otherClip;
audio.Play();
}
}
```

◆ var loop : bool

描述：音频剪辑是否循环？如果你在一个正在播放的 AudioSource 上禁用循环，声音将在当前循环结束后停止。

```
// 停止声音循环
audio.loop = false;
```

◆ var maxVolume : float

描述：音频剪辑播放时的最大音量。不论你距离多近，声音不会比这个值更大。

```
audio.maxVolume = 0.5;
```

参见：minVolume, rolloffFactor.

◆ var minVolume : float

描述：音频剪辑播放时的最小音量。不论你距离多远，声音不会比这个还小。

```
audio.minVolume = 0.5;
```

参见：maxVolume, rolloffFactor.

◆ var pitch : float

描述：音频源的音调。

```
audio.pitch = 1.0;
```

◆ var playOnAwake : bool

描述：如果设置为真，音频源江在 awake 时自动播放。

```
if(! audio.playOnAwake) {
audio.Play();// 如果没有设置为 Awake 时播放，那么播放该剪辑
```

}

◆ var rolloffFactor : float

描述：设置声音衰减的速度。该值越大，侦听器必须更接近才能听到声音。

audio.rolloffFactor = 0.1;

参见：minVolume, maxVolume.

◆ var time : float

描述：以秒计算的播放位置。使用这个来读当前播放时间或寻找新的播放时间。

◆ var velocityUpdateMode : AudioVelocityUpdateMode

描述：Audio Source 是否应该以固定或动态的方式更新？如果你遇到了这个源的多普勒效应问题，请确保设置这个更新与 Audio Source 的移动

在同一循环内。如果它被附加到一个刚体，默认设置将自动设置该源在固定的更新周期内更新，以动态的以及其他方式。

◆ var volume : float

描述：音频源的音量。

audio.volume = 0.2;

函数

◆ function Pause () : void

描述：暂停播放 clip。

audio.Pause();

参见：Play，Stop 函数

◆ function Play () : void

描述：播放 clip。

audio.Play();

参见：Pause，Stop 函数

◆ function PlayOneShot (clip : AudioClip, volumeScale : float = 1.0F) : void

描述：播放一个 AudioClip。

//与其他物体碰撞时播放 impact 音频剪辑

var impact : AudioClip;

function OnCollisionEnter () {

audio.PlayOneShot(impact);

}

◆ function Stop () : void

描述：停止播放 clip。

audio.Stop();

参见：Play，Pause 函数

类方法

◆ static function PlayClipAtPoint (clip : AudioClip, position : Vector3, volume : float = 1.0F) : void

描述：在制定位置上播放剪辑。播放完成后自动消除音频源。正在播放的声音的音频源被返回。

//在制定位置播放 clip

var clip : AudioClip;

AudioSource.PlayClipAtPoint(clip, Vector3 (5, 1, 2));

如果想进一步控制播放，可以使用下面代码。

```
var theClip : AudioClip;
PlayAudioClip(theClip, transform.position, 1);
function PlayAudioClip (clip : AudioClip, position : Vector3, volume : float)
{
var go = new GameObject ("One shot audio");
go.transform.position = position;
var source : AudioSource = go.AddComponent (AudioSource);
source.clip = clip;
source.volume = volume;
source.Play ();
Destroy (go, clip.length);
return source;
}
Destroy (go, clip.length);
return source;
}
```

继承的成员

继承的变量

enable 启用则 Behaviour 被更新，不启用则不更新

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

Camera

类，继承自 Behaviour。Camera 是一个设备，玩家通过它来看世界。屏幕空间点用像素定义。屏幕左下为(0,0)；右上是(pixelWidth.pixelHeight)。z 的位置是以世界单位衡量到相机的距离。视口空间点是归一化的并且是相对于相机的。相机左下为(0,0)；右上是(1,1)。z 的位置是以世界单位衡量到相机的距离。世界空间点是以全局坐标定义的（例如 Transform.position）。

参见：camera component

变量

◆ var aspect : float

描述：长宽比（宽度除以高度）。默认的长宽比是从屏幕的长宽比计算得到的，即使相机没有被渲染到全屏。如果修改了相机的 aspect 比，这个值将保持到你调用 camera.ResetAspect()；这将充值猖狂比为屏幕的长宽比。

```
if(camera.aspect > 1.0)
print ("Screen is more wide than tall!");
else
print ("Screen is more tall than wide!");
```

参见：camera component，Screen 类。

◆ var backgroundColor : Color

描述：屏幕将被清理未这个颜色。只能在 clearFlags 被设置为 CameraClearFlags.SolidColor（或设置为 CameraClearFlags.Skybox 但没有设置天空盒）时使用。

```
//来回变化背景色
var color1 = Color.red;
var color2 = Color.blue;
var duration = 3.0;
//设置清楚标记为该颜色
camera.clearFlags = CameraClearFlags.SolidColor;
function Update () {
```

```
var t = Mathf.PingPong (Time.time, duration) / duration;
camera.backgroundColor = Color.Lerp (color1, color2, t);
}
```

参见：camera component,Camera.clearFlags 属性。

◆ var cameraToWorldMatrix : Matrix4x4

描述：从相机空间到世界空间的变换矩阵（只读）。使用次变量计算相机空间中的一个点在世界坐标中的什么位置。注意相机空间与 openGL 的约定相同。相机前味 z 轴负方向。这不同于 Unity 的约定，向前微 z 轴争相。

```
//以 distance 单位，沿着相机所看到的方向，在屏幕中绘制一个黄色的球，
var distance = -1.0;
function OnDrawGizmosSelected () {
var m = camera.cameraToWorldMatrix;
var p = m.MultiplyPoint (Vector3(0,0,distance));
Gizmos.color = Color.yellow;
Gizmos.DrawSphere (p, 0.2);
}
```

◆ var clearFlags : CameraClearFlags

描述：相机如何清楚背景。可以是 CameraClearFlags.Skybox，CameraClearFlags.SolidColor，CameraClearFlags.Depth，CameraClearFlags.Nothing。

```
//用背景色清除（忽略天空盒）
camera.clearFlags = CameraClearFlags.SolidColor;
```

◆ var cullingMask : int

描述：这个用来选择性的渲染部分场景。如果 GameObject 的 layerMask 与相机的 cullingMask 进行 AND 操作后为 0，则这个游戏物体对于该相机是不可见的。参考 Layers 获取更多信息。

```
//只渲染在第一层中的物体（默认层）
camera.cullingMask = 1 << 0;
```

参见：camera component

◆ var depth : float

描述：相机在渲染顺序上的深度。具有较低深度的相机将在较高深度的相机之前渲染。如果有多个相机并其中一些不需要覆盖整个屏幕，可以使用这个来控制相机的绘制次序。

```
//设置该相机在主相机之后渲染
camera.depth = Camera.main.depth + 1;
```

参见：camera component，Camera.rect 属性

◆ var farClipPlane : float

描述：远裁剪面的距离。

```
camera.farClipPlane = 100.0;
```

参见：camera component

◆ var fieldOfView : float

描述：相机的视野，以度为党委。这是垂直视野。水平 FOV 取决于视口的宽高比。当相机是正交时，fieldOfView 被忽略。（参考 orthographic）。

```
//设置附加到同一个游戏物体上的相机的 FOV 位 60
camera.fieldOfView = 60;
//设置主相机的视野为 80
```

Camera.main.fieldOfView = 80;

参见：camera component

◆ var nearClipPlane : float

描述：近才见面的距离。

camera.nearClipPlane = 0.1;

参见：camera component

◆ var orthographic : bool

描述：相机是正交的（true）还是透视的（false）？如果 orthographic 为 true，相机的视野由 orthographicSize 定义。如果 orthographic 为 false，相机的视野由 fieldOfView 定义。

//设置相机为正交

camera.orthographic = true;

//设置主相机为正交

Camera.main.orthographic = true;

参见：camera component

◆ var orthographicSize : float

描述：在正交模式下相机的一半尺寸。这个为视体垂直大小的一半。相机不是正交时，水平视大小拒绝于视口的长宽比。

//设置相机的正交尺寸为 5

camera.orthographic = true;

camera.orthographicSize = 5;

//设置主相机的正交尺寸为 5

Camera.main.orthographic = true;

Camera.main.orthographicSize = 5;

参见：camera component

◆ var pixelHeight : float

描述：相机的像素高度（只读）。

print ("Camera is " + camera.pixelHeight + " pixels high");

◆ var pixelRect : Rect

描述：相机被渲染到屏幕像素坐标中的位置。

function Update () {

var r = camera.pixelRect;

print ("Camera displays from " + r.xMin + " to " + r.xMax + " pixel");

}

◆ var pixelWidth : float

描述。相机的像素宽度（只读）。

print ("Camera is " + camera.pixelWidth + " pixels wide");

◆ var projectionMatrix : Matrix4x4

描述：设置自动以的投影矩阵。如果你改变该矩阵，相机的渲染将不再给予它的 fieldOfView 更新，直到调用到 ResetProjectionMatrix。只有当真正需要一个非标准的投影时，才使用自定义投影。该属性被 Unity 的水渲染使用来设置一个 oblique.projection 矩阵。使用自定义投影需要了解变换和投影矩阵。

//让相机以有节奏的方式晃动

var originalProjection : Matrix4x4;

originalProjection = camera.projectionMatrix;

```
function Update () {
var p = originalProjection;
//改变原始矩阵的某些值
p.m01 += Mathf.Sin (Time.time * 1.2) * 0.1;
p.m10 += Mathf.Sin (Time.time * 1.5) * 0.1;
camera.projectionMatrix = p;
}
```

//设置一个变异中心的投影，这里透视的消失点没有必要在屏幕的中心。left/right/top/bottom 定义近裁剪面大小。例如相机的近裁剪面偏移中心的距离，改变该值就可以看到相机视图的变化。

```
@script ExecuteInEditMode
var left = -0.2;
var right = 0.2;
var top = 0.2;
var bottom = -0.2;
function LateUpdate () {
var cam = camera;
var m = PerspectiveOffCenter(left, right, bottom, top,cam.nearClipPlane, cam.farClipPlane );
cam.projectionMatrix = m;
}
static function PerspectiveOffCenter(left : float, right : float,bottom : float, top : float,near : float, far : float ) : Matrix4x4{
var x = (2.0 * near) / (right - left);
var y = (2.0 * near) / (top - bottom);
var a = (right + left) / (right - left);
var b = (top + bottom) / (top - bottom);
var c = -(far + near) / (far - near);
var d = -(2.0 * far * near) / (far - near);
var e = -1.0;
var m : Matrix4x4;
m[0,0] = x; m[0,1] = 0; m[0,2] = a;
m[0,3] = 0;
m[1,0] = 0; m[1,1] = y; m[1,2] = b; m[1,3] = 0;
m[2,0] = 0;
m[2,1] = 0; m[2,2] = c; m[2,3] = d;
m[3,0] = 0; m[3,1] = 0; m[3,2] = e;
m[3,3] = 0;
return m;
}
```

◆ var rect : Rect

描述：相机被渲染到屏幕归一化坐标中的位置。rect 的范围从 0（左/下）到 1（右/上）。

```
//每次按下空格键时改变视口宽度
function Update () {
```

```
if (Input.GetButtonDown ("Jump")) {
//随机选择边缘
var margin = Random.Range (0.0, 0.3);
//设置矩形
camera.rect = Rect (margin, 0, 1 - margin * 2, 1);
}
}
```

◆ var targetTexture : RenderTexture

描述：目标渲染纹理（只限 UnityPro）。

◆ var velocity : Vector3

描述：获取世界空间中相机的速度（只读）这是相机在上一帧以秒为单位的运动。

```
function Update () {
print ("Camera moving at " + camera.velocity.magnitude + " m/s");
}
```

◆ var worldToCameraMatrix : Matrix4x4

描述：从世界到相机空间的变换矩阵。用这个计算物体的相机空间位置或提供自定义相机的位置。这个位置不是基于变化的。注意相机空间与 openGL 的约定相同：相机的前面为 Z 轴负方向。这不同于 Unity 的约定，向前微 Z 轴争相。如果你改变该矩阵，相机的渲染将不再基于它的 Transform 更新。知道调用 ResetWorldToCameraMatrix。

```
//从 offset 位置偏移相机的渲染
var offset = Vector3 (0,1,0);
function LateUpdate () {
//构建一个沿着 Z 轴偏移与镜像的矩阵。因为相机已经为 Z 轴镜像，并用于其余部分
var camoffset = Vector3 (-offset.x, -offset.y, offset.z);
var m = Matrix4x4.TRS (camoffset, Quaternion.identity, Vector3 (1,1,-1));
//重载 worldToCameraMatrix 为偏移镜像变换矩阵
camera.worldToCameraMatrix = m * transform.worldToLocalMatrix;
}
```

函数

◆ function CopyFrom (other : Camera) : void

描述：使该相机的设置于其他相机相同。这将从 er 相机拷贝到所有相机变量（视野，清楚标记，裁剪蒙版）。这也将使相机的变换与 other 相机相同，相机的层也与 other 相机相同。在做自定义渲染效果的时候，这可以用来设置一台具有与其他相机设置完全相同的相机。例如在使用 RenderWithShader 时。

◆ function Render : void

描述：手动渲染相机。这个将使用相机的清除标记，目标纹理和所有其他设置。相机将发送 OnPreCull，OnPreRender 和 OnPostRender 到任何附加的脚本上，并渲染任何最后的图像滤镜。这个用来精确的控制渲染次序，为了使用这个特性，创建一个相机并禁用它。然后在它上面调用 Render。参见：RenderWithShader。

◆ function RenderToCubemap (cubemap : Cubemap, faceMask : int = 63) : bool

描述：从这个相机渲染到一个立方贴图。这个是非常有用的。可以再编辑器中烘焙场景的静态立方贴图。参考下面的想到实例。相机的位置，清除标志和裁剪面距离将被使用来渲染到立方贴图表面。faceMask 是一个比特域，标示哪个立方贴图面应该被渲染。每个位对应于一个面。比特数是 CubemapFace 枚举的整型值。默认的所有六个立方贴图面都将

被渲染（默认值 63 最低的 6 位是打开的）。如果渲染失败，这个函数将返回 false。某些显卡不支持这个函数。参见：Cubemap assets，Reflective shaders。

//从给定的点渲染场景到以静态立方贴图放置这个脚本到工程的 Editor 文件夹中，然后用一个 Reflective shader 来使用立方贴图

```
class RenderCubemapWizard extends ScriptableWizard{
var renderFromPosition : Transform;
var cubemap : Cubemap;
function OnWizardUpdate () {
helpString = "Select transform to render from and cubemap to render into";
isValid = (renderFromPosition != null) && (cubemap != null);
}
function OnWizardCreate () {
//为渲染创建临时相机
var go = new GameObject( "CubemapCamera", Camera );//放置到物体上
go.transform.position = renderFromPosition.position;
go.transform.rotation = Quaternion.identity;
//渲染到立方贴图
go.camera.RenderToCubemap( cubemap );
//销毁临时相机
DestroyImmediate( go );
}
@MenuItem("GameObject/Render into Cubemap")
static function RenderCubemap () {
ScriptableWizard.DisplayWizard(
"Render cubemap", RenderCubemapWizard, "Render!");
}
}
```

◆ function RenderToCubemap (cubemap : RenderTexture, faceMask : int = 63) : bool

描述：从这个相机渲染到一个立方贴图。用于实时反射到立方贴图渲染纹理，也是非常耗时的，尤其是所有六个立方贴图面在每一帧中都被渲染。相机的位置，清楚标志和裁剪面距离将被使用来渲染到立方贴图表面。faceMask 是一

个比特域，标示哪个立方贴图面应该被渲染。每个位对应于一个面。比特数是 CubemapFace 枚举的整型值。默认的所有六个正方贴图面都将被渲染（默认值 63 的最低 6 位是打开的）。如果渲染失败该函数会返回 false。某些显卡不支持该函数。参加：RenderTexture.isCubemap, Reflective shaders。

//将该脚本附加到使用了 Reflective shader 的物体上实时反射立方贴图

```
script ExecuteInEditMode
var cubemapSize = 128;
var oneFacePerFrame = false;
private var cam : Camera;
private var rtex : RenderTexture;
function Start () {
//启动时渲染所有 6 个面
UpdateCubemap( 63 );
```

```
}
function LateUpdate () {
if (oneFacePerFrame) {
var faceToRender = Time.frameCount % 6;
var faceMask = 1 << faceToRender;
UpdateCubemap (faceMask);
}
else {
UpdateCubemap (63); //所有 6 个面
}
}
function UpdateCubemap (faceMask : int) {
if (!cam) {
var go = new GameObject ("CubemapCamera", Camera);
go.hideFlags = HideFlags.HideAndDontSave;
go.transform.position = transform.position;
go.transform.rotation = Quaternion.identity;
cam = go.camera;
cam.farClipPlane = 100; //不渲染较远的部分
cam.enabled = false;
}
if (!rtex) {
rtex = new RenderTexture (cubemapSize, cubemapSize, 16);
rtex.isPowerOfTwo = true;
rtex.isCubemap = true;
rtex.hideFlags = HideFlags.HideAndDontSave;
renderer.sharedMaterial.SetTexture ("_Cube", rtex);
}
cam.transform.position = transform.position;
cam.RenderToCubemap (rtex, faceMask);
}
function OnDisable () {
DestroyImmediate (cam);
DestroyImmediate (rtex);
}
```

◆ **function RenderWithShader (shader : Shader, replacementTag : string) : void**

描述：用 shader 替换渲染相机。参考 Rendering with Replaced Shaders 获取细节。此函数将渲染相机。这将使相机的的清除标记、目标纹理和所有其他设置。这个相机不会发送 OnPreCull, OnPreRender 或者 OnPostRender 到已附加的脚本上。

图像滤镜

也不会被渲染。该函数可以用于特效，例如渲染整个场景屏幕空间缓冲，热效果等。为了使用该特性，创建一个相机并禁用它。然后在它上面调用 RenderWithShader。参见：Rendering with Replaced Shaders, SetReplacementShader, Render。

◆ **function ResetAspect () : void**

描述：返回猖狂比为屏幕的长宽比。调用这个结束 aspect 的效果。

camera.ResetAspect();

◆ function ResetProjectionMatrix () : void

描述：让投影反映正常的相机参数。调用这个结束 projectionMatrix 的效果。

camera.ResetProjectionMatrix();

◆ function ResetReplacementShader () : void

描述：从相机上移除 shader 替换。调用这个结束 SetReplacementShader 效果。

◆ function ResetWorldToCameraMatrix () : void

描述：在场景中让渲染位置反映相机位置。调用这个结束 worldToCameraMatrix 的效果。

camera.ResetWorldToCameraMatrix();

◆ function ScreenPointToRay (position : Vector3) : Ray

描述：返回从相机出发，穿过屏幕点的一个射线。产生的射线是在世界空间中，从相机的近裁剪面开始并穿过屏幕 position(x,y)像素坐标（position.z 被忽略）。屏幕空间以像素定义。屏幕的左下为(0,0)；右上是(pixelWidth,pixelHeight)。

```
//在屏幕视图中绘制一条线，穿过一个到屏幕左下角 200 像素的点
function Update () {
var ray = camera.ScreenPointToRay (Vector3(200,200,0));
Debug.DrawRay (ray.origin, ray.direction * 10, Color.yellow);
}
```

◆ function ScreenToViewportPoint (position : Vector3) : Vector3

描述：从屏幕空间到视口空间变换 position。屏幕空间以像素定义。屏幕的左下为(0,0)；右上是(pixelWidth,pixelHeight)。z 的位置是以世界单位衡量的刀相机的距离。视口空间是归一化的并相对于相机的。相机的左下为(0,0)；右上是(1,1)。z 的位置是以世界单位衡量的刀相机的距离。

◆ function ScreenToWorldPoint (position : Vector3) : Vector3

描述：从屏幕空间到世界空间变换 position。屏幕空间以像素定义。屏幕的左下为(0,0)；右上是(pixelWidth,pixelHeight)。z 的位置是以世界单位衡量的到相机的距离。

```
//在所选相机的近裁剪面上绘制一个黄色的球，在离左下 100 像素的位置
function OnDrawGizmosSelected () {
var p = camera.ScreenToWorldPoint (Vector3 (100,100,camera.nearClipPlane));
Gizmos.color = Color.yellow;
Gizmos.DrawSphere (p, 0.1);
}
```

◆ function SetReplacementShader (shader : Shader, replacementTag : string) : void

描述：使相机用 shader 替换来渲染。参考 Rendering with Replaced Shaders。调用该函数后，相机将使用替换的 shader 来渲染它的视图。调用 ResetReplacementShader 来重置为普通渲染。参见：Rendering with Replaced Shaders, ResetReplacementShader, RenderWithShader。

◆ function ViewportPointToRay (position : Vector3) : Ray

描述：返回从相机出发穿过视点的一个射线。产生的射线是在世界空间中，从相机的近裁剪面开始并穿过视口 position(x,y)坐标（position.z 被忽略）。视口坐标是归一化的并相对于相机的。相机的左下为(0,0)；右上为(1,1)。

```
//打印相机直接看到的物体名称
function Update () {
```

```
//获取穿过屏幕中心的射线
var ray = camera.ViewportPointToRay (Vector3(0.5,0.5,0));
//投射
var hit : RaycastHit;
if (Physics.Raycast (ray, hit)) {
print ("I'm looking at " + hit.transform.name);
}
else {
print ("I'm looking at nothing!");
}
}
```

◆ function ViewportToScreenPoint (position : Vector3) : Vector3

描述：从视口空间到屏幕空间变换 position。视口空间是归一化的并相对于相机的。相机的左下为(0,0)；右上为(1,1)。z 的位置是以世界单位衡量的刀相机的距离。屏幕空间以像素定义。屏幕左下为(0,0；右上为(pixelWidth,pixelHeight)。z 的位置是以世界单位衡量的到相机的距离。

◆ function ViewportToWorldPoint (position : Vector3) : Vector3

描述：从视口空间到屏幕空间变换 position。视口空间是归一化的并相对于相机的。相机的左下为(0,0)；右上为(1,1)。z 的位置是以世界单位衡量的刀相机的距离。屏幕空间以像素定义。屏幕左下为(0,0；右上为(pixelWidth,pixelHeight)。z 的位置是以世界单位衡量的到相机的距离。

```
//在进裁剪面的右上角，针对在场景视图中选中的相机绘制一个黄色的球
function OnDrawGizmosSelected () {
var p = camera.ViewportToWorldPoint (Vector3 (1,1, camera.nearClipPlane));
Gizmos.color = Color.yellow;
Gizmos.DrawSphere (p, 0.1);
}
```

◆ function WorldToScreenPoint (position : Vector3) : Vector3

描述：从世界空间到屏幕空间变换 position。屏幕空间以像素定义。屏幕的左下为(0,0；右上为(pixelWidth,pixelHeight)。z 的位置是

以世界单位衡量的刀相机的距离。

```
var target : Transform;
function Update () {
var screenPos = camera.WorldToScreenPoint (target.position);
print ("target is " + screenPos.x + " pixels from the left");
}
```

◆ function WorldToViewportPoint (position : Vector3) : Vector3

描述：从世界空间到视口空间变换 position。视口空间是归一化的并相对于相机的。相机的左下为(0,0)；右上为(1,1)。z 的位置是以世界单位衡量的到相机的距离。

```
//找出 target 在屏幕的左边还是右边
var target : Transform;
function Update () {
var viewPos = camera.WorldToViewportPoint (target.position);
//视口坐标范围从 0 到 1
```

```
if( viewPos.x > 0.5 )
print ("target is on the right side!");
else
print ("target is on the left side!");
}
```

消息传递

◆ function OnPostRender () : void

描述：OnPostRender 在相机渲染场景之后调用。这个消息被发送到所有附加在相机上的脚本。

◆ function OnPreCull () : void

描述：OnPreCull 在相机开始裁剪场景之前调用。OnPreCull 仅仅在这个过程之间被调用。这个消息被发送到所有附加在相机上的脚本。如果你想改变相机的视觉参数（例如，fieldOfView 或者仅仅是变换），就在这里做这个。场景物体的可见性将给予相机的参数在 OnPreCull 之后确定。

◆ function OnPreRender () : void

描述：OnPreCull 在相机开始渲染场景之前调用。这个消息被发送到所有附加在相机上的脚本。注意如果你在这里改变了相机的视野参数（例如 fieldOfView），它们将只影响下一帧。用 OnPreCull 代替。

◆ function OnRenderImage (source : RenderTexture, destination : RenderTexture) : void

描述：OnRenderImage 在所有渲染完成后被调用，来渲染图片的后记处理效果（仅限 UnityPro）。该函数允许使用给予 shader 的过滤器来处理最后的图片。进入的图片是 source 渲染纹理。结果是 destination 渲染纹理。当多个图片过滤器附加在相机上时，它们序列化的处理图片，将第一个过滤器的目标作为下一个过滤器的源。

该消息被发送到所有附加在相机上的脚本。参见：UnityPro 中的 image effects。

◆ function OnRenderObject (queueIndex : int) : void

描述：该函数被用来渲染你自己的物体，使用 Graphics.DrawMeshNow 或者其他函数。queueIndex 指定用来渲染物体的 render queue。可以使用 RenderBeforeQueues 属性来指定你想绘制这个物体到哪个渲染队列。

◆ function OnWillRenderObject () : void

描述：如果物体可见，每个相机都会调用 OnWillRenderObject。该函数在裁剪过程中被调用。可用该函数创建具有依赖性的渲染纹理。只有在被渲染的物体可见时，才更新该渲染纹理。例如，水组件就使用了这个。Camera.current 将被设置为要渲染这个物体的相机。

类变量

◆ static var allCameras : Camera[]

描述：返回场景中所有启用的相机。

```
var count = Camera.allCameras.length;
print ("We've got " + count + " cameras");
```

◆ static var current : Camera

描述：当前用于渲染的相机，只用于低级的渲染控制（只读）。多数时候你会使用 Camera.main。只有在执行下面时间的时候使用该函数：MonoBehaviour.OnRenderImage, MonoBehaviour.OnPreRender, MonoBehaviour.OnPostRender。

◆ static var main : Camera

描述：第一个启用的被标记为“MainCamera”的相机（只读）。如果场景中没有这个相机返回 null。

继承的成员

继承的变量

enable 启用则 Behaviour 被更新，不启用则不更新

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody 附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

ConstantForce

类，继承自 Behaviour。一个不间断的力。这是一个小的物理工具类，用来将一个连续的力应用到一个物体上。Rigidbody.AddForce 只在一帧中应用力道 Rigidbody，因此你不得不持续调用这个函数。

领以方面 ConstantForce 将在每一帧中应用这个力，知道改变这个力或例举为一个新的值。参见 Rigidbody。

变量

◆ var force : Vector3

描述：这个力在每帧中应用到刚体。

```
//在世界坐标系中向上移动刚体。
constantForce.force = Vector3.up * 10;
```

◆ var relativeForce : Vector3

描述：力-相对于刚体坐标系统-在每帧中应用。

```
//向前移动刚体。
constantForce.relativeForce = Vector3.forward * 10;
```

◆ var relativeTorque : Vector3

描述：力矩-相对于刚体坐标系统-在每帧中应用。

```
//绕着其 X 轴旋转该物体。
constantForce.relativeTorque = Vector3.right * 2;
```

◆ var torque : Vector3

描述：这个力矩在每帧中应用到刚体。

```
//绕着世界的 Y 轴旋转该物体。
constantForce.torque = Vector3.up * 2;
```

继承的成员

继承的变量

enable　启用则 Behaviour 被更新，不启用则不更新

transform　附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera　附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation　附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio　附加到该 GameObject 的 Audio（没有返回 null）

guiText　附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture　附加到该 GameObject 的 GuiTexture（没有返回 null）

collider　附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint　附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter　附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject　该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag　该游戏物体的标签。

name　对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent　如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren　如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren　如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents　返回 GameObject 上所有 type 类的组件

CompareTag　该游戏物体被是否被标签为 tag？

SendMessageUpwards　在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate　克隆 original 物体并返回这个克隆。

Destroy　移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType　返回第一个类型为 type 的激活物体。

operator==　比较两个物体是否相同。

operator!=　比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

GUIElement

类，继承自 Behaviour。用于显示在 GUI 上的图片和文本字符串的基类。这个类处理所有 GUI 元素的基本功能。

函数

◆　function GetScreenRect (camera : Camera = null) : Rect

描述：在屏幕坐标返回 GUIElement 的包围矩阵。如果没有 camera 被指定，一个填充整个游戏窗口的相机将被使用。

```
function Start () {
var r = guiTexture.GetScreenRect ();
print ("This gui element is %d pixel wide.",r.width);
var r = guiText.GetScreenRect();
print ("This gui element is %d pixel wide.",r.width);
}
```

◆　function HitTest (screenPosition : Vector3, camera : Camera = null) : bool

描述：屏幕上的点在是否在元素内部？如果 screenPositon 包含在这个 GUIElement 内部时返回真，screenPosition 以屏幕坐标指定，例如由 Input.mousePosition 属性返回的值。如果没有 camera 被指定，一个填充整个游戏窗口的相机将被使用。注意，如果位置在元素内部，true 将被返回。即使游戏物体属于 Ignore Raycast 层（通常鼠标事件不会发送到 Ignore Raycast 物体）。参见 GUILayer.HitTest。

if (guiTexture.HitTest (Vector3 (360, 450, 0.)))
print ("This gui texture covers pixel 360, 450");
if (guiText.HitTest (Vector3(360, 450, 0.)))
print ("This gui texture covers pixel 360, 450");
继承的成员
继承的变量
enable 启用则 Behaviour 被更新，不启用则不更新
transform 附加到该 GameObject 的 Transform（没有返回 null）
rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）
camera 附加到该 GameObject 的 Camera（没有返回 null）
light 附加到该 GameObject 的 Light（没有返回 null）
animation 附加到该 GameObject 的 Animation（没有返回 null）
constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）
renderer 附加到该 GameObject 的 Renderer（没有返回 null）
audio 附加到该 GameObject 的 Audio（没有返回 null）
guiText 附加到该 GameObject 的 GuiText（没有返回 null）
networkView 附加到该 GameObject 的 NetworkView（没有返回 null）
guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）
collider 附加到该 GameObject 的 Collider（没有返回 null）
hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）
particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）
gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上
tag 该游戏物体的标签。
name 对象的名称
hideFlags 该物体是否被隐藏，保存在场景中或被用户修改
继承的函数
GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null
GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索
GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。
GetComponents 返回 GameObject 上所有 type 类的组件
CompareTag 该游戏物体被是否被标签为 tag？
SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法
SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法
BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName
GetInstanceID 返回该物体的实例 id
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体、组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

GUIText

类，继承自 GUIElement。显示在一个 GUI 中的文本字符。

变量

◆ var alignment : TextAlignment

描述：文本的对齐。

```
guiText.alignment = TextAlignment.Left;
```

◆ var anchor : TextAnchor

描述：文本的锚点。

```
guiText.anchor = TextAnchor.MiddleCenter;
```

◆ var font : Font

描述：用于文本的字体。

```
var font : Font;
guiText.font = font;
```

◆ var lineSpaceing : float

描述：行间距倍数。这个可以使被定义在字体中行间距增加。

```
//双倍行间距
guiText.lineSpacing = 2.0;
```

◆ var material : Material

描述：用于渲染的 Material。赋予一个新的材质来改变渲染处理。改变这个材质来改变渲染的字体。如果赋值 null 到 material，将使用内置基本字体。

```
//改变这个材质显示绿色文本。
guiText.material.color = Color.green;
```

参见：font variable

◆ var pixelOffset : Vector2

描述：文本的像素偏移。文本从它的原始位置偏移的量。

```
guiText.pixelOffset = Vector2 (10, 10);
```

◆ var tabSize : float

描述：这个标签增加的宽度。

```
guiText.tabSize = 4.0;
```

◆ var text : string

描述：需要显示的文本。

GUITexture

类，继承自 GUIElement。用于 2D GUI 的纹理图片。

变量

◆ var color : Color

描述：GUI 纹理的颜色。

```
//改变纹理的颜色为绿色
guiTexture.color = Color.green;
```

◆ var pixelInset : Rect

描述：pixelInset 用来调整尺寸和位置。为了使 GUI 纹理总是原始大小，设置 transform.localScale 为 Vector3.zero。

```
transform.position = Vector3.zero;
transform.localScale = Vector3.zero;
guiTexture.pixelInset = Rect (50, 50, 100, 100);
```

◆ var texture : Texture

描述：用于绘制的纹理。

```
//将 someTexture 赋值给 guiTexture。
var someTexture : Texture2D;
guiTexture.texture = someTexture;
```

继承的成员

继承的变量

enable 启用则 Behaviour 被更新，不启用则不更新

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate　克隆 original 物体并返回这个克隆。

Destroy　移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType返回所有类型为 type 的激活物体。

FindObjectOfType　返回第一个类型为 type 的激活物体。

operator==　比较两个物体是否相同。

operator!=　比较两个物体是否不同。

DontDestroyOnLoad　加载新场景时确保目标物体不被自动销毁。

GUILayer

类，继承自 Behaviour。

函数

◆ function HitTest (screenPosition : Vector3) : GUIElement

描述：在屏幕的制定位置获取 GUI 元素。返回屏幕上指定点的 GUIElement。如果 screenPosition 在某个 GUIElement 内部，那个元素将被返回。如果这个位置没有在任何 GUI 元素内部，返回 null。输入 Ignore Raycast 层的 GUI 元素将被忽略，就像它们不存在。screenPosition 在屏幕坐标系下，就像由 Input.mousePosition 属性返回的值。参见：GUIElement.HitTest，Input.mousePositon。

继承的成员

继承的变量

enable　启用则 Behaviour 被更新，不启用则不更新

transform　附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera　附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation　附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio　附加到该 GameObject 的 Audio（没有返回 null）

guiText　附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture　附加到该 GameObject 的 GuiTexture（没有返回 null）

collider　附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint　附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter　附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject　该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag　该游戏物体的标签。

name　对象的名称

hideFlags该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent　如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren　如果该组件位于 GameObject 或任何其子物体上，返回 type

类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 **GameObject** 或任何它的子物体上，返回 **type** 类组件。

GetComponents 返回 **GameObject** 上所有 **type** 类的组件

CompareTag 该游戏物体被是否被标签为 **tag**？

SendMessageUpwards 在该游戏物体的每个 **MonoBehaviour** 和该行为的父对象上调用名为 **methodName** 的方法

SendMessage 在该游戏物体的每个 **MonoBehaviour** 上调用 **methodName** 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 **MonoBehaviour** 上调用 **methodName**

GetInstanceID 返回该物体的实例 **id**

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代理。

FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

LensFlare

类，继承自 **Behaviour**。用于 **Lensflare** 组件的借口。这允许你在运行时改变镜头闪光的亮度和颜色。

变量

◆ **var brightness : float**

描述：闪光的强度。这个控制发光元素的尺寸和亮度。

参见：**Lens flare component, flare assets**。

◆ **var color : Color**

描述：闪光的颜色。这控制闪光元素的颜色（那些启用了 **use light color** 的）。

参见：**Lens flare component, flare assets**。

◆ **var flare : Flare**

描述：使用的 **flare asset**。

参见：**Lens flare component, flare assets**。

继承的成员

继承的变量

enable 启用则 **Behaviour** 被更新，不启用则不更新

transform 附加到该 **GameObject** 的 **Transform**（没有返回 **null**）

rigidbody 附加到该 **GameObject** 的 **Rigidbody**（没有返回 **null**）

camera 附加到该 **GameObject** 的 **Camera**（没有返回 **null**）

light 附加到该 **GameObject** 的 **Light**（没有返回 **null**）

animation 附加到该 **GameObject** 的 **Animation**（没有返回 **null**）

constantForce 附加到该 **GameObject** 的 **ConstantForce**（没有返回 **null**）

renderer 附加到该 **GameObject** 的 **Renderer**（没有返回 **null**）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetComponent 如果游戏附体上附加了一个，则返回 type 类组件，没有则返回 null

GetComponentInChildren 如果该组件位于 GameObject 或任何其子物体上，返回 type 类组件，使用深度优先搜索

GetComponentsInChildren 如果这些组件位于 GameObject 或任何它的子物体上，返回 type 类组件。

GetComponents 返回 GameObject 上所有 type 类的组件

CompareTag 该游戏物体被是否被标签为 tag？

SendMessageUpwards 在该游戏物体的每个 MonoBehaviour 和该行为的父对象上调用名为 methodName 的方法

SendMessage 在该游戏物体的每个 MonoBehaviour 上调用 methodName 方法

BroadcastMessage 在这个游戏物体或其任何子物体上的每个 MonoBehaviour 上调用 methodName

GetInstanceID 返回该物体的实例 id

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体、组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代理。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator!= 比较两个物体是否不同。

DontDestroyOnLoad 加载新场景时确保目标物体不被自动销毁。

Light

类，继承自 Behaviour。用于 light components 的脚本接口。使用这个来控制 Unity 光源的所有方面。这个属性完全与现实在监视面板中的值相同。通常光源都是在编辑器中创建，但是有时候需要从脚本中创建。

```
function Start () {
//制作一个游戏物体
var lightGameObject = new GameObject ("The Light");
//添加光源组件
lightGameObject.AddComponent (Light);
```

```
//设置颜色和位置
lightGameObject.light.color = Color.blue;
//在添加光照组件后，设置位置（或任何变换组件）
lightGameObject.transform.position = Vector3 (0, 5, 0);
}
```

变量

◆ var attenuate : bool

描述：光源是否随着距离衰减？对于 Directional 光源衰减总是关闭的。

```
//关闭光源衰减。
light.attenuate = false;
```

参见：Light component

◆ var color : Color

描述：光源的颜色。为了修改光源的强度你需要改变光源颜色的亮度。光源总是增加亮度，因此一个黑色的光源与没有光源相同。光源是否随着距离衰减？对于 Directional 光源衰减总是关闭的。

```
//在 2 秒内使颜色变黑。
function Update () {
light.color -= Color.white / 2.0 * Time.deltaTime;
}
//在 2 个颜色之间来回插值光源颜色。
var duration = 1.0;
var color0 = Color.red;
var color1 = Color.blue;
function Update () {
//设置光源颜色
var t = Mathf.PingPong (Time.time, duration) / duration;
light.color = Color.Lerp (color0, color1, t);
}
```

参见：Light component。

◆ var cookie : Texture

描述：被盖光源投影的 cookie 纹理。如果 cookie 是一个立方体贴图，光源将变为一个点光源。注意 cookie 只在像素光源下显示。

```
//在监视面板中公开一个纹理的引用
var newCookie : Texture2D;
//赋 cookie
light.cookie = newCookie;
```

参见：Light component

◆ var cullingMask : int

描述：这个用来选择性的照亮部分场景。如果 GameObject 的 layerMask 与光源的 cullingMask 进行 AND 操作后为 0.，那么这个游戏物体不能被这个光源照亮。参考 Layers 获取更多信息。

```
//只照亮第一层中的物体（默认层）
light.cullingMask = 1 << 0;
```

参见：Light component

◆ var flare : Flare

描述：为这个光源使用的 flare asset。

```
//在监视面板中公开一个闪光的引用
var newFlare : Flare;
//赋值闪光
light.flare = newFlare;
```

参见：Light component 和 flare asset。

◆ var intensity : float

描述：光源的强度被光源的颜色乘。该值可以介于 0~8 之间。允许你创建明亮的灯光。

```
//随时间改变光照强度
var duration = 1.0;
function Update() {
//余弦理论
var phi = Time.time / duration * 2 * Mathf.PI;
//获取余弦，并将范围从-1~1 变为 0~1
var amplitude = Mathf.Cos( phi ) * 0.5 + 0.5;
//设置光的颜色
light.intensity = amplitude;
}
```

◆ var range : float

描述：光源的范围。

即使光源关闭了 attenuate，它还是只影响在它范围内的物体。

```
//在原始范围与原始范围一般处变换光照范围
var duration = 3.0;
private var originalRange : float;
originalRange = light.range;
function Update() {
var amplitude = Mathf.PingPong( Time.time, duration );
//将 0..持续时间改为 0.5..1 范围
amplitude = amplitude / duration * 0.5 + 0.5;
//设置光照范围
light.range = originalRange * amplitude;
}
```

参见：Light component。

◆ var renderMode : LightRenderMode

描述：如何渲染该光源？此处可以是 LightRenderMode.Auto，LightRenderMode.ForceVertex 或 LightRenderMode.ForcePixel。像素光渲染比较慢但是看起来较好，尤其是对那些没有较高面数的几何体。一些效果（例如凹凸）只会在像素光照下显示。

```
//使光源只以点光照模式渲染
light.renderMode = LightRenderMode.ForceVertex;
```

参见：Light component

◆ var shadowConstantBias : float

描述：阴影偏移常量。

参见：shadows，shadowObjectSizeBias。

◆ var shadowSizeBias : float

描述：阴影偏移常量。

参见：shadows，shadowConstantBias。

◆ var shadows : LightShadows

描述：这个光源是否投射阴影？

```
//设置光源为投射硬阴影
light.shadows = LightShadows.Hard;
```

参见：LightShadows, shadowStrength property, Renderer.castShadows, Renderer.receiveShadows

◆ var shadowStrength : float

描述：光源阴影的强度？

```
//使光源的阴影非常弱
light.shadowStrength = 0.3;
```

参见：shadows property, Renderer.castShadows, Renderer.receiveShadows。

◆ var spotAngle : float

描述：光源的投射光角度。主要用于 Spot 光源。改变 Directional 光源的 cookie 尺寸。对 Point 光源没有影响。

```
//在”minAngle”与”maxAngle”之间随机改变投射角度
//每’interval’秒改变一次.
var interval=0.3;
var minAngle=10;
var maxAngle=90;
private var timeLeft:float;
teimleft=interval;
light.type=LightType.Spot;
function Update()
{
    timeLeft=Time.deltaTime;
if(timeLeft<0.0){
    //开始改变
timeLeft=interval;
light.spotAngle=Random Range(minAngle,maxAngle);
};
}
```

参见：Light component

var type：LightType

描述：光源的类型

可以是 LightType.Spot,LightType.Directional,LightType.Point.

```
//制作一个投射光源
Light.type=LightType.Spot;
```

参见：Light component

继承的成员

继承的变量

Enabled 启用 Behaviours 被更新，禁用 Behaviours 不被更新。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为 methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

MonoBehaviour

类，继承自 Behaviour

MonoBehaviour 是每个脚本所继承的基类。

每个使用 Javascript 的脚本自动地从 MonoBehaviour 继承。当使用 C#或 Boo 时你必须显式从 MonoBehaviour 继承。

参见：手册中的 chapter on scripting.

函数

◆function CancelInvoke():void

描述：取消所有在这个 MonoBehaviour 上调用

```
//在 2 秒时开始
//一个 projectile 所在每 0.3 秒时运行
var projectile:Rigidbody;
InvokeRepeating("LaunchProjectile",2,0.3);
//当用户按下 ctrl 按钮时，
//取消重复调用
Function Update()
{
    If(Input.GetButton("Fire1")){
        CancelInvoke();

    }
}
Function LaunchProjectile(){
    Instance=Instantiate(prtfab);
Instance.velocity=Random.insideUnitSphere*5;
}
```

◆function Cancellvoke(methodName:string):void

描述：撤销该行为中名为 methodName 的所有调用。

```
//在 2 秒时开始
//一个 projectile 将在每 0.3 秒时运行
var projectile :Rigidbady;
invokeRepeating("LaunchProjectile",2,0.3);
//当用户按下 crtl 按钮时
//取消重复调用
Function Update(){
    If(Input.GetButton(Fire1)){
        CancelInvode("LaunchProjectile");
    }
}
Function LanuchProjectile(){
    instance=Instntiate(prefab);
    instance.velocity=Random.insideUnitSphere*5;
}
```

◆function Invoke(methodName:string,time:float):void

描述：在 time 秒调用 methodName 方法。

```
//在两秒时发射一个投射物
var    Projectile:rigidbady;
Invoke("LaunchProjectile",2);
Function LaunchProjectile(){
    Instance=Instantiate(prefab);
    Instance.velocity=Random.insideUnitSphere*5;
}
```

◆function InvokeRepeating(methodName:string,time:float,repeatRate:float):void

描述：在 time 秒调用 methodName 方法。

第一次调用之后，每 repeatRate 秒重复调用这个函数

```
//在 2 秒时开始
//一个 Projectile 将在每 0.3 秒运行
var projectile:Rigidbody;
InvokeRepeating("LaunchProjectile",2,0.3);
Function LaunchProjectile(){
    instance=Instantiate(prefab);
    instance.velocity=Random.insideUnitSphere*5;
}
```

◆function IsInvoking(methodName:string)

描述：是否有任何对 methodName 的调用在等待？

◆function IsInvoking(methodName:string):bool

描述：是否有任何 MonoBehaviour 的调用在等待？

◆Function StartCoroutine(routine:IEnumerator):Coroutine

描述：开始一个 coroutine

一个 coroutine 的执行可以任何位置使用 yield 语句暂停，当 coroutine 继续的时候 yield 返回值，当行为需要跨越多个帧的时候，Coroutines 是非常好的，当 Coroutines 几乎没有性能负担。StartCoroutine 函数总是立即返回，然而你能够 yidle 结果，这个将等待直到 coroutine 执行完成。

当使用 javascript 时，没有必要使用 StartCorutine,编译器会为你做这件事。在写 C#代码的时候你必须调用 StartCoroutine

```
//在这个例子中我们显示如何并行调用
//一个 coroutine 并继续执行这个函数。
//0 秒后,打印" Starting0.0"
//0 秒后,打印" Before WaitAndPrint Finishes0.0"
//2 秒后，打印" WaitAndPrint 2.0"
print("Starting"+Time.time);
//WaitAndPrint 作为一个 coroutine 开始，并等待直到它完成
WaitAndPrint(2.0)
Print("Before WaitAndPrint Finishes"+Time.time);
function WaitAndPrint(waitTime:float){
    //暂停执行 waitTime 秒
    Yield WaitForSeconds(waitTime);
    Print("WaitAndPrint"+Time.time);
//WaitAndPrint 作为一个 coroutine 开始,并等待直到它完成
```

```
Yield WaitAndPrint(2.0);
Print("Done"+Time.time);
}
C#例子代码如下
//C#例子,在这个例子中我们显示如何并行调用
//一个 coroutine 并继续执行这个函数
void Start(){
//0 秒后，打印”Starting0.0”
//0 秒后，打印”Before WaitAndPrint Finishes0.0”
//2 秒后，print”waitAndPrint2.0”
Print("Starting"+Time.time);
StartCoroutine(WaitAndPrint(2.0F));
Print("Before WaitAndPrint Finishes"+Time.time);
}
IEnumerator WaitAndPrint(float waitTime){
// 暂停 waitTime 秒
Yield return new WaitForSeconds(waitTime);
Print("WaitAndPrint"+Time);
}
//c#例子
//在这个例子中我们展示如何调用一个 coroutine 并等待直到它完成
IEnumerator Start(){
//0 秒后，打印”strating0.0”
//2 秒后，prints”WaitAndPrint2.0”
//2 秒后，打印”Done 2.0”
Print("Starting"+Time.time);
Yield return StartCoutine(WaitAndPrint(2.0f);
Print("Done"+Time.time);
}
IEnumerator WaitAndPrint(float waitTime){
//暂停 waitTime 秒
    Yield return new WaitForSeconds(waitTime);
Print("WaitAndPrint"+Time.time);
}
◆Function StartCoroutine(methodName: string, value: object=null): Coroutine
```

描述：开始为 methodName 的 Coroutine。

在大多数情况下，你要用 StartCoroutine 的变体。然而，使用一个字符串方法名的 StartCoroutine 允许 StopCoroutine 使用一个特定的方法名称。字符串级本的特定时它有较高的运行时开销来开始 coroutine，并且你只能传递一个参数。

```
//在则合格例子中我们现实如何一个字符串名调用一个 coroutine 并停止它
Function Start(){
StartCoroutine("DoSomething",2.0);
Yield WaitForSeconds(1);
StopCoroutine（”DoSomething”）;
```

```
}
Function DoSomething(someParameter: float){
While(true)
{
Print(”DoSomething Loop”);
Yield
}
}
```

◆Function StopAllCoroutines(): void

描述：停止所有运行在这个行为上的 Coroutine。

```
//开始 coroutine
StartCoroutine(“DoSomething”);
//紧跟着取消 coroutine
Function DoSomething(){
While(true){
Yield;
}
}
StopAllCoroutines();
```

◆Function StopCoroutine(methodName: string): void

描述：停止所有运行在这个行为上的名为 methodName 的 Coroutine。

请注意只使用一个字符串方法名的 StartCoroutine 才能使用 StopCoroutine 停止。

```
//在这个例子中我们显示如何使用一个字符串名调用一个 coroutine 并停止它
Function Start(){
StartCoroutine(“DoSomething”,2.0);
Yield WaitForSeconds(1);
StopCoroutine（”DoSomething”）;
}
Function DoSomething(someParameter: float){
While(true)
{
Print(”DoSomething Loop”）;
Yield
}
}
```

重载函数

◆Function Awake(): void

描述：当脚本实例被加载时，Awake 被调用。

在游戏开始前，使用 Awake 来初始化任何变量或游戏状态。在脚本实例的生命期内 Awake 只被调用一次。Awake 在所有物体被初始化之后被调用，因此你可以安全地告诉其他物体或使用如 GameObject.FindWithTag 来查询它们。每个游戏物体的 Awake 以随机的顺序被调用。因此，你应该使用 Awake 来再脚本之间设置引用，并使用 Start 来传递消息。Awake 总是在任何 Start 函数之前调用。这允许你调整脚本的初始化顺序。Awake 不能作为一个 Coroutine

注意对于 C#哈 Boo 用户：使用 Awake 而不是构造函数来初始化，因为组件的序列化状态在构造的时候还没有确定。Awake 只被调用一次，就像构造函数。

```
Private var target: GameObject;
Function Awake(){
Target=GameObject.FindWithTag(‘”Player”)
}
```

Awake 不能作为一个 coroutine。

◆Function FixedUpdate (): void

描述：如果 MonoBehaviour 被启用，这个函数将以固定的帧率调用。

当处理 Rigidbody 是应该使用 FixedUpdate 而不是使用 Update。例如，当添加一个力到刚体时，你必须在 FixedUpdate 内以固定的帧率应用而不是在 Update 内。

```
//应用一个向上的力到刚体
Function FixedUpdate(){
Rigidbody.AddForce(Vector3.up);
}
```

力伟到从上一次调用 Update 的消逝时间，使用 Time.deltaTime，这个函数只在 Behaviour 被启用时调用。重载这个函数，以便给你的组件提供功能。

◆Function LateUpdate (): void

描述：如果该 Behaviour 被启用，LateUpdate 将在每帧中调用。

LateUpdate 在所有 Update 函数被调用后调用。这可用于调整脚本执行顺序。例如，一个跟随相机应该总是在 LateUpdate 中实现，因为他跟踪 Update 中移动的物体。

```
//向前以 1 米/秒的速度移动物体
Function LateUpdate (){
transfor.Translate(0,0,Time.deltaTime*1);
}
```

为了得到从最后一个调用 LateUpdate 的消逝时间，使用 Time.deltaTime。如果该 Behaviour 被启用，该函数将在每帧中调用。重载这个函数，以便给你的组件提供功能。

◆Function OnApplicationPause(pause: bool): void

描述：当玩家暂停时发送到所有游戏物体。

OnApplicationPause 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆Function OnApplicationQuit(): void

描述：在应用退出之前发送到所有游戏物体。

在编辑器中当用户停止播放模式时这个被调用。在网页播放器中当 web 被关闭时这个函数被调用。

◆Function OnBecameInvisible(): void

描述：OnBecameInvisible 函数在这个渲染上的脚本。OnBecameVisible 和 OnBecameInvisible 可以用于只需要在需要在物体可见时才进行的计算。

```
//当它不可见时禁用这个行为
Finction OnBecameInvisible(){
Enabled=false;
}
```

OnBecameInvisible 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。当在编辑器中运行时，场景试图相机也会导致这个函数被调用。

◆Function OnBecameVisible(): void

描述：OnBecameVisible 函数在这个渲染器对任何相机变得可见时被调用。

这个消息被发送到所有附加在渲染器上的脚本。OnBecameVisible 和 OnBecameInvisible 可以用于只需要在需要在物体可见时才进行的计算。

```
//当它不可见时禁用这个行为
Finction OnBecameVisible(){
Enabled=false;
}
```

OnBecameVisible 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。当在编辑器中运行时，场景试图相机也会导致这个函数被调用。

◆Function OnCollisionEnter(collisionInfo: Collision): void

描述：

当这个碰撞器/刚体开始接触另一个刚体/碰撞器时 OnCollisionEnter 被调用。

相对于 OnTriggerEnter，OnCollosionEnter 传递 Collision 类而不是 Collider，Collision 类包含接触点，碰撞速度等细细。如果在函数中不使用 CollisionInfo，省略 CollisionInfo 参数以避免不西药的计算。注意如果碰撞器附加了一个非动力学刚体，也只发送碰撞事件。

```
Function OnCollisionEnter(collision: Collision){
//调试绘制所有的接触点和法线
For(var contact: ContactPoint in collision.oontacts){
Debug.DrawRay(contact.point, contact.normal, Color.white);
}
//如果碰撞物体有较大的冲击就播放声音
If(collision.relativeVelocity.magnitude>2)
Audio.Play();
}
```

OnCollisionEnter 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆Function OnCollisionExit(collisionInfo: Collision): void

描述：当这个碰撞器/刚体开始接触另一个刚体/碰撞器时 OnCollisionEnter 被调用。

相对于 OnTriggerExit，OnCollosionExit 传递 Collision 类而不是 Collider. Collision 类包含接触点，碰撞速度等细细。如果在函数中不使用 CollisionInfo，省略 CollisionInfo 参数以避免不西药的计算。注意如果碰撞器附加了一个非动力学刚体，也只发送碰撞事件。

```
Function OnCollisionExit(collision: Collision){
Print("No longer in contact with"+collisionInfo.transform.name);
}
```

OnCollisionExit 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆function OnCollisionStay(collisionInfo:collision):void

描述：对于每个与刚体碰撞器相触碰的碰撞器刚体，OnCollisionStay 将在每一帧中被调用。

相对于 OnTriggerStay,OnCollisionStay 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞速度等细节。如果在函数中不使用 collisionInfo 省略 collisionInfo 参数以避免不要必要的计算。注意如果碰撞器附加了一个非动力学刚体，也只发送碰撞事件。

```
function OnCollisionStay(CollisionInfo:Collision){
    //调试绘制所有的接触点和法线
    for (var contact:ContactPoint in collision.contacts){
        Debug.DrawRay(contact.point,contact.normal,Color.white);
```

```
    }
}
```

OnCollisionStay 可以是一个 coroutine，也简单地在这个函数中使用 yield 语句

◆function OnConnectedToServer():void

描述：当成功链接到服务器时在客户端调用这个函数。

```
function OnConnectedToServer(){
Debug.Log("Connected to server");
//发送到本地玩家名称到服务器
}
```

◆function OnControllerColliderHit(Hit:controllerColliderHit):void

描述：在移动的时候，控制器碰到一个碰撞器时，OnControllerColliderHit 被调用。这可以用来在角色碰到物体时推开物体。

```
//这个脚本推开所有角色碰到物体时推开物体
Var pushPower=2.0;
Function OnControllerColliderHit(hit: OnControllerColliderHit){
    Var body:Rigidbody=hit.collider.attachedRigidboby;
    //无刚体
    If(body==null||body.isKinematic)
        return;
//不推开我们身后的物体
If(hit.moveDirection.y<-0.3)
    Return;
//从移动方向计算推的方向
//只推开物体到旁边而不是上下
Var pushDir=Vector3(hit.moveDirection.x,0,hit.moveDirection.z)
//如果知道角色移动有多快
//然后你就可以用它乘以推动速度
//使用推力
Body.velocity=pushDir*pushPower;
}
```

◆function OnDisable():void

描述：当这个行为禁用或不活动时这个函数被调用。

当物体被销毁的时候这个函数也会被调用并可以用于任何清理的代码。当脚本在编译结束后被加载时，OnDisable 将被调用，然后脚本加载完成后 OnEnable 被调用。

```
Function OnDisable(){
Print("scrip was remove");
}
```

OnDisabe 不能作为一个 coroutine.

◆function OnDisconnectedFromServer(mode:NetworkDisconnection):void

描述：当链接挂失或服务器断开时在客户端用这个函数。

```
Function OnDisconnectedFromServer(info.NetworkDisconnection){
    Debug.Log("Disconnected from server"+info);
}
```

◆function OnDrawGizmos():void

描述：如果你想绘制可被点的 Gizmos 时，实现 OnDrawGizmos,
这允许你在场景中快速选择重要的物体。

```
//在物体的位置上绘制光源灯泡
function OnDrawGizmos(){
    Gizmos.DrawIcon(transform.position,"light Gizmo.tiff");
}
```

◆function OnDrawGizmosSelected():void
描述：如果你想在物体被选择时绘制 gizmos，实现这个 OnDrawGizmosSelected。
Gizmos 只在物体被选择的时候绘制。Gizmos 不能点选。这可以设置更容易，例如一个爆炸脚本可以绘制一个球显示爆炸半径。

```
var explosionRadius=5.0;
function OnDrawGizmosSelected(){
//选中的时候显示爆炸半径
    Gizmos.color=Color.white;
    Gizmos.DrawSphere(transfom.position,explosinRadius);
}
```

◆function OnEnable():void
描述：当物体启用或激活时这个函数被调用

```
function OnEnbale(){
    print("script was enabled");
}
```

◆function OnFailedToconnect(error:NetworkConnectionError);void
描述：当链接因为某些原因失败时在客户端上调用这个函数。
失败的原因作为 NetworkConnectionError 枚举传入。

```
function OnFailedToconnect (info:NetworkConnectionError){
    Debug.Log("Could not connect to server"+error);
```

◆function OnFailedToConnectToMasterServer (error:NetworkConnectionError):void
描述：当链接到主服务器出现问题时在客户端或服务器调用这个函数。
错误的原因作为 NetworkConnectionError 枚举传入

```
function OnFailedToConnectToMasterServer(info:NetworkConnectionError){
    Debug.Log("Could not connect to master server"+info);
}
```

◆function OnGUI():void
描述：OnGUI 被调用来渲染并处理 GUI 事件。
如果 Monobehaviour 的启用属性被设置为假，OnGUI()将不会被调用。

```
Function OnGUI(){
    If(GUI>Button(Rect(10,10,150,100),"I am a button")){
        Print("You clicked the button")
    }
}
```

参考 GUI Scrtipting Guide 获取更多信息。
◆function ONJointBreak(breakForce:float):void
描述：当附加到相同游戏物体上的关节被断开时调用

当一个力大于这个关节的 breakForce 时，关节将被断开，当关节断开时，OnJointBreak 将被调用，应用到关节的断开力将被传入，OnJointBreak 之后这个关节将自动从游戏物体移除。参见：Joint.breakForce

◆function OnLevelWasLoaded(level:int):void

描述：这个函数在一个新的关卡被加载之后被调用

//level 是被加载的关卡的索引。使用菜单项 File>Build Settings 来查看索引引用的是那个场景，参见：Application.LoadLevel

```
//当关卡 13 被加载的关卡的索引”Woohoo”
Function OnLevelWasLoaded(level:int){
If(level==13){
    Print(“Woohoo”);
}
}
```

OnLevelWasLoaded 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆function OnMouseDown():void

描述：当用户在 GUIElement 或 Collider 上按下鼠标按钮时 OnMouseDown 被调用

这个事件被发送到所有附加在 Collider 或 GUIElement 的脚本上。

```
//加载名为“someLevel”的关卡
//来响应用户单击
Function OnMouseDown(){
    Application.LoadLevel(“SomeLevel”);
}
```

这个函数不会在属于 Ignore Raycast 的层上调用。

OnMouseDown 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。这个事件被发送到的有附加在 Collider 或 GUIElement 的脚本上。

◆function OnMouseDrag():void

描述：当用户在 GUIElement 或 Collider 上点击并按住鼠标时 OnMouseDrag 被调用。

当鼠标被按下时 OnMouseDrag 将在每帧中被调用。

```
//按住鼠标时使用材质颜色变暗
function OnMouseDrag(){
    renderer.material.color==Color.white*Time.deltaTime;
}
```

这个函数不会在属于 Ignore Ray 的层被调用 。

OnMouseDrag 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。这个事件被发送到所有附加在 collider 或 GUIElement 的脚本上。

◆Function OnMouseEnter():void

描述：当鼠标进入 GUIElement 或 collider 时，OnMouseEnter 被调用。

```
//附加这个脚本到网格
//使用当鼠标经过这个网格时使它变红
Function OnMouseEnter(){
    Renderer.material.color=Color.red;
}
```

这个函数不会在属于 Ignore Raycast 的层上调用。

OnMouseEnter 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。这个事件

被发送所有附加在 Collider 或 GUIElement 的脚本上。

◆function OnMouseExit():void

描述：当鼠标不再位于 GUIElement 或 Collider 上时，OnMouseExit 被调用。

OnMouseExit 与 OnMouseEnter 相反

```
//当鼠标在网格时
//渐变材质的红色组件为零
Function OnMouseExit(){
    Rnderer.material.color=Color.white;
}
```

这个函数不会在属于 Ignore Raycast 的层上调用。

OnMouseExit 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。这个事件被发送到所有附加在 Collider 或 GUIElement 上脚本上。

◆function OnMouseOver();void

描述：当鼠标在 GUIElement 或 Collider 上时，OnMouseOver 被调用。

```
//当鼠标在网格上时
//渐变材质的红色组件为零
function OnMouseOver(){
    renderer.material.color==0.1*Time.deltaTime;
}
```

这个函数不会在属于 Ignore Raycast 的层上调用。

OnMouseOver 可以是一个 coroutine，简单地在这个函数中使用 yield 语句，这个事件被发送到所有附加在 Collider 或 GUIElement 的脚本上。

◆function OnMouseUP():void

描述：当用户已经松开鼠标按钮时 OnMouseUp 调用。

在 GUIElement 或 Collider 上松开鼠标时 OnMouseUp 被调用。

```
//加载名为”someLevel”的关卡
//来响应用户单击
function OnMouseUp(){
    Application.LoadLevel(“SomeLevel”);
}
```

这个函数不会在属于 Ignore Raycast 的层上调用。

OnMouseUp 可以是一个 coroution,简单地在这个函数 yield 语句。这个事件被发送到所有附加在 Collider 或 GUIElement 的脚本上。

◆function OnNetworkInstantiate(info:NetworkMessageInfo):void

描述：当一个物体使用 Network.Instantiate 进行网络初始化在该物体上调用这个函数。

这个对于禁用或启用一个已经初始化的物体组件来说是非常有用的，它们的行为取决与它们是在本地还是在远端。

注意：在 NetworkMessageInfo 里的 networkView 属性不能在 OnNetworkInstantiate 里使用。

```
function OnNetworkInstantiate(info:NetworkMessageInfo){
    Debug.Log(“New object instantiated by”+info.sender);
}
```

◆function onParticleCollision(other:GameObject):void

描述：当一个粒子碰到一个碰撞器时 OnParticleCollision 被调用。

这个可以在游戏物体被粒子击中时应用伤害到它的上面，这个消息被发送到所有附加到

WorldParticleCollider 的脚本上和被击中的 Collider 上，这个消息只有当你在 WorldParticleCollider 检视面板中启用了 sendCollisionMessage 时才会被发送。

```
//应用力到所有被粒子击中的刚体上
function OnParticleCollison(other:GameObject){
    var body=other.rigidbody;
    if(body){
        var direction=other.transform.position+transform.position;
        direction=direction.normalized;
        body.AddForce(direction*5);
    }
}
```

OnParticleCollision 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆function OnPlayConnected(player:NetworkPlayer):void

描述：当一个新玩家成功连接在服务器上调用这个函数。

```
private var playerCount:int=int=0;
function OnPlayerConnect(player.NetworkPlayer){
Debug.log("Player"+playerCount+++"connected from"+player.ip.Address+":"+player.port);
//用玩家的信息构建一个数据结构
}
```

◆function OnPlayerDisconnected(player:NetworkPlayer):void

描述：当玩家从服务器断开时在服务器上调用这个函数。

```
function OnPlayerDisconnected (player:NetworkPlayer){
    Debug.Log("Clean up after player"+player);
    Network.RemoveRPcs(player);
    Network.DestroyPlayerObject(player);
}
```

◆function OnpostRender():void

描述：OnPostRender 在相机渲染场景之后调用。

只有脚本被附加到相机上时才会调用这个函数。OnPostRender 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

OnPostRender 在相机渲染完所有它的物体之后被调用。如果你想在所有相机和 GUI 被渲染之后做一些事情，使用 WaitForEndFrame coroutine

参见：OnPreRender,WaitForEndOfFrame

```
//当附加到机时，将消除
//相机纹理的 alpha 通道为纯白色
//如果你有一个渲染纹理并想将它显示在 GUI 上时可以使用这个函数
Parvate var mat:Material;
function OnPostRender(){
    //创建一个只渲染白色到 alpha 通道的着色器
    If(mat)
    {
    Mat=new Material("Shader Hidden SetAlpha"+)
```

```
"SubShader{"+"pass{"+"ZTest Always Cull off Zwrite Off"+"ColorMaskA"+"Color(1,1,1,1)"+"}"+"}"
+"}");
}
}
// 用上述着色器绘制一个全屏四边形
GL.PushMatrix();
GL.LoadOrtho();
for(var i=0;i<mat.pssCount;++i){
    mat.SetPass(i);
    GL.Begin(GL.QUADS);
    GL.Vertex3(0,0,0.1);
    GL.Vertex3(1,0,0.1);
    GL.Vertex3(1,1,0.1);
    GL.Vertex3(0,1,0.1);
    GL.End();
    }
    GL.PopMatrix();
}
```

◆function OnpreCull():void

描述：OnPreCull 在相机开始裁剪场景之前调用

裁剪决定那个物体对于相机来说是可见的。OnPreCull 仅仅在这个过程之间被调用。

只有脚本被附加到相机上时才会调用这个函数

如果你想改变相机的视觉参数（例如 fieldOfView 或者仅是变换），就在这里做这个。

场景物体的可见性将基于相机参数在 OnPerCull 之后确定。

function OnRender（）:void

描述：OnpreRender 在相机开始渲染场景之前调用。

只用脚本被附加到相机上时才会调用这个函数。

注意如是果你在这里改变了相机的视野参数（例如 fieldOfView），它们将影响下一帧。用 OnPreCull 代替。OnPreRender 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

参见：OnPostRender

function OnRenderImage(source:RenderTexture,destination:RenderTexture):void

描述：OnRenderImage 在所有渲染完成后被调用，来渲染图片的后期处理效果（限于 Unity Pro）

这允许你使用基于 shader 的过滤器来处理最后的图片。进入图片是 source 渲染纹理结果是 destination 渲染纹理。当有多个图片过滤附加在相机上时，它们序列化地处理图片，将第一个过滤器的目标作为下一个过滤器的源。

这个消息被发送到所有附加在相机上脚本。

参见：Unity Pro 中的 image effects

◆function OnRenderObject(queueindex:int):void

描述：OnRenderObject 被用来渲染你自己的物体，使用 Graphics.DarwMesh 或者其他函数。

queueIndex 指定用来渲染物体的 render queue。可以使用 RenderBeforeQueues 属性来

指定你想绘制这个物体到哪里渲染队列。

◆function OnserializeNetworkView(stream:Bitstream.info:NetworkMessageInfo):void

描述：用来在一个被网络视监控的脚本中自定义变量同步

它自动决定被序列化的变量是否应该发送或接收，查看下面的例子获取更好的描述：

```
//这个物体的生命信息
Int currentHealth;
function OnSerializeNetworkView(stream:BitStream,info:NetworkMessageInfo){
if(stream.isWriting){
        int health=currentHealth;
        stream.Serialize(health);
    }
    else{
        int health=0;
        stream.Serialize(health);
        currentHealth=health;
    }
}
```

◆function OnServerInitialized():void

描述：当 Network.InitializeServer 被调用并完成时，在服务器上调用这个函数。

```
function OnServerInitialized(){
    Debug.Log("Server initialize and ready");
}
```

◆function OnTriggerEnter(other:Collider):void

描述：当这个 Collider other 进入 trigger 时 OnTriggerEnter 被调用。

这个消息被发送到这个触发器碰撞器和接触到触发器的刚体（或者是碰撞器如果没有刚体）。注息如果碰撞器附加了一个刚体，也只发送触发器事件。

```
//销毁所有进入该触发器的物体
function OntriggerEnter(other:Collider){
    Destroy(other.gameObject);
}
```

OnTriggerEnter 可以是一个 coroutine 简单地在这个函数中使用 yield 语句。

◆function OnTriggerExit(other:Collider):void

描述：当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

这个消息被发送到触发和接触到这个触发器的碰撞器。注意如果碰撞附加了一个刚体，也只发送触发器事件。

```
//觥毁所有离开该触发器的物体
function OntriggerExit (other:Collider){
Destroy(other.gameObject);
}
```

OnTriggerExit 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆function OnTriggerStay(other:Collider):void

描述：对于每个 Collider other,当它触碰到 trigger 时，OnTriggerStay 会在每一帧中都被调用。

这个消息被发送到触发器和接触到这个触发器的碰撞器。注意如果碰撞器附加了一个

刚体，也只发送触发器事件。

```
//对所进入这个触发器的刚体使用一个向上的力
function OnTriggerStay(other:Collider){
    if(other.attachedRigidbody){
        other.attachedRigidbody.AddForce(Vector3.up*10);
    }
}
```

OnTriggerStay 可以是一个 coroutine，简单地在这个函数中使用 yield 语句。

◆function OnWillRenderObject():void

描述：如果物体可见，每个相机都会调用 OnWiiRenderObject。

这个函数在裁剪过程中被调用，在渲染所有被裁剪的物体之前被调用。可以用这个来创建具有依赖性的渲染纹理，只有在被渲染的物体可见时才更新这个渲染纹理。作为一个例子，水组件就使用这个。

Camera.current 将被设置为要渲染这个物体的相机

◆function Reset():void

描述：Reset 在用户点击检视面版的上下文菜单或者第一次添加该组件被调用。Reset 只在编辑模式下调用。Reset 最常用于在检视面板中给定一个好的默认值

```
//设置 target 为一个默认的值
//这可以用于跟随相机
var target:GameObject;
function Reset(){
    //如果 target 没有赋值，设置它
    if(!target){
        target=GameObject.FindWithTag(“play”);
    }
}
```

◆function Start():void

描述：Start 在所有 Update 方法被第一次调用前调用。

Start 在行为的生命期内只调用一次。Awake 和 Start 的不同在于 Start 只在脚本实例被启用时调用。这个允许你延迟所有初始化代码，知道真正需要它们的时候，Awake 总是在任何 Start 函数之前调用。这允许你调整脚本的初始化顺序。Start 不能作为一个 coroutine

Start 函数在所有脚本实例的 Awake 函数调用之后调用。

```
    //初始化 target 变量
    //target 是私有的并且不能在检视面板中编辑
parivate var target:GameObject;
function Start(){
    target=GameObject.FindWithTag(“Player”);
}
```

◆function Update():void

描述：如果 MonoBehaviour 被启用，Update 将在每帧中调用。

Update 是最常用的函数，来实现任何游戏行为。

```
//向前以 1 米的速度移动物体
function Update(){
    transform.Translate(0,0,Time.deltaTime*1);
```

}

为了得到从最后一次调用 Update 的消逝时间，使用 Time.deltaTime。如果该 Behaviour 被启用，该函数将在每帧中调用，重载这个函数，以便给你的组件提供功能。

继承的成员

继承的变量

endbled 启用 Behaviours 被更新，禁用 Behaviour 不被更新

transform 附加到这个 GameObject 的 Transform（如果没有 null）

rigidbody附加到这个 GameObject 的 Rigidbody（如果没有 null）

camera 附加到这个 GameObject 的 Camera（如果没有 null）

light 附加到这个 GameObject 的 Light（如果没有 null）

animation 附加到这个 GameObject 的 Animation（如果没有 null）

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有 null）

renderer 附加到这个 GameObject 的 Renderer（如果没有 null）

audio 附加到这个 GameObject 的 Audio（如果没有 null）

guiText 附加到这个 GameObject 的 GUIText（如果没有 null）

networkView 附加到这个 GameObject 的 NetworkView(只读)（如果没有 null）

guiTexture 附加到这个 GameObject 的 GUITexture（只读）（如果没有 null）

collider 附加到这个 GameObject 的 Collider（如果没有 null）

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有 null）

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有 null）

gameObject 这个组件附加的游戏物体。一个组件总是附加到一个游戏物体

tag 这个游戏物体的标签

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景或被用户修改

继承的函数

GetComponent 返回 type 类型组件，如果游戏物体上附加了一个如果没有返回

null。

GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子

物体上，使用深度优先搜索。

GetComponentsInChildren 返回所有 type 类型的组件，这些组件位于 GameObject 或任何它

的子物体上。

GetComponents 返回 GameObject 上所有 type 类型的组件。

CompareTag 这个游戏物体被标签为 tag？

SendmessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法

SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。

BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 ID。

继承的类函数

Operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
Operator== 比较两个物体是否相同。
Operator!= 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

Terrain
类，继承自 MonoBehaviour
渲染地形的 Terrain
变量
◆var basemapDistance:float
描述：超出 basemapDistance 的高度图，将使用预先计算的地解析度 basemap。
这个可以改善远处块的性能。Unity 混合任意数量的地形纹理，使用条图来渲染近处的高度图。

```
function Start(){
    Terrain.activeTerrain.basemapDistance=100;
}
```

◆var castShadows:bool
描述：地形应该投射阴影？
◆var detailObjectDistance:float
描述：在这个距离内细节将被显示。

```
function Start(){
    Terrain.activeTerrain.detailObjectDistance=40;
    }
```

◆var heightmapMaximumLOD:int
描述：让你实质降低用于渲染的高度图解析度
这个可用在低端的显卡上并不会显示最高的 LOD 地形。值为 0 表示总是显示最高细节。值为 1 表示三角形数量将减小到 1/4，高度图的解析度将是宽度和高度的一半。

```
function Start(){
    Terrain.activeTerrain.heightmapMaximumLOD=1;
}
```

◆var heightmapPixelError:float
描述：当切换 LOD 时大约有多少像素时地形将处于错误的情况。
较高的值减少多边形的绘制数量。

```
function Start(){
    Terrain.activeTerrain.heightmapPixelError=10;
}
```

◆var Lighting:Terrain.Lighting
描述：地形光照模式
参见：TerrainLighting 枚举，Terrain Lightmap,terrain Settings

◆var terrainData:TerrainData

描述：存储在高度图中的地形数据，地形纹理，细节网格和树。

◆var treebillboardDistance:float

描述：到相机的距离超过这个值，树只被作为公告板渲染。

降低这个值将改善性能，但是使过度看起来更差因为公告板和树的不同将更加明显。

```
Function Start(){
    Terrain.activeTerrain.treeBillboardDistance=100;
}
```

◆var treeCrossFadeLength:float

描述：树从公告变换到网格到网格的总距离增量。

降低这个值将使变换发生的越快。设置为 0 时，在从网格切换分告表示时将立即发生。

```
function Start(){
    Terrain.activeTerrain.treeCrossFadeLength=20;
}
```

◆var treeDistance:float

描述：渲染树的最大距离。

这个值越高，越远的树将看到，运行的也更慢。

参见：Terrain.treeBillboardDistance

```
function Start(){
    Terrain.activeTerrain.treeDistance=2000;
}
```

◆var treeMaximumFull.ODCount:int

描述：全 LOD 时渲染树的最大数量。

这个是一个简单的设置用来阻止太多的楼以过高的解析度和密度被绘制。因为如果 treeMaximumFullLodCount 过高，树将不会消失，你应该修改 treeBillboardDistance 为不

包含。

```
Function Start(){
    Terrain.activeTerrain.treeMaximumFullLODCount=200;
}
```

函数

◆function SampleHeight(worldPositon:Vector3):float

描述：在世界空间的给定位置处采样高度。

```
function LateUpdate(){
    transfom.positon.y=Terrain.activeTerrain.SampleHeght(transfom.position);
}
```

◆function SetNeighbors(left:Terrain,top:Terrain,right:Terrain,bottom:Terrain):void

描述：允许你在相邻地形间设置连接

这个确保 LOD 在相邻地形上相同。注意，在一个地形上调用这个函数是不够的，你需要在每个地形上设置邻居。

类变量

◆static var activeTerrain: Terrain

描述：激活的地形。这个是用来在场景中获取主地形的快捷方式。

类方法

◆Static functin CreateTerrainGameObject(assignTerrain: TerrainData): GameObject

描述：从 TerrainData 创建一个包含碰撞器的地形。

继承的成员

继承的变量

Enabled 启用 Behaviour 被更新，禁用 Behaviours 不被更新。

transform 附加到该 GameObject 的 Transform（没有返回 null）

rigidbody附加到该 GameObject 的 Rigidbody（没有返回 null）

camera 附加到该 GameObject 的 Camera（没有返回 null）

light 附加到该 GameObject 的 Light（没有返回 null）

animation 附加到该 GameObject 的 Animation（没有返回 null）

constantForce 附加到该 GameObject 的 ConstantForce（没有返回 null）

renderer 附加到该 GameObject 的 Renderer（没有返回 null）

audio 附加到该 GameObject 的 Audio（没有返回 null）

guiText 附加到该 GameObject 的 GuiText（没有返回 null）

networkView 附加到该 GameObject 的 NetworkView（没有返回 null）

guiTexture 附加到该 GameObject 的 GuiTexture（没有返回 null）

collider 附加到该 GameObject 的 Collider（没有返回 null）

hingeJoint 附加到该 GameObject 的 HingeJoint（没有返回 null）

particleEmitter 附加到该 GameObject 的 ParticleEmitter（没有返回 null）

gameObject 该组件所附加的游戏物体。组件总是会附加到游戏物体上

tag 该游戏物体的标签。

name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

Invok e 在 time 秒调用 methodName 方法。

Invok Repeating 在 time 秒调用 methodName 方法。

CancelInvoke 取消所有在这个 MonoBehaviour 上的调用

IsInvoking 是否有任何对 methodName 的调用在等待？

StartCoroutine 开始一个 coroutine

StopCoroutine 停止所有运行在这个行为上的名为 methodName 的 coroutine

StopAllCoroutines 停止所有运行在这个行为上的 coroutine

GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。

GetComponcetInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上。使用深度优先搜索

GetComponcetsInChildren 返回所有 type 类型组件，这些组件位于 Gameobject 或任何它的子物体上。

GetComponents 返回 GameObject 上所有 type 类型的组件。

CompareTag 这个游戏物体被标签为 tag?

SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。

SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。

BroadcastMessage 在这个游戏物体或其 2 任何子的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 ID。

继承的消息传递

Update 如果 MonoBehaviour 被启用，Update 将在每帧中调用。

LateUpdate 如果该 Behaviour 被禁用，LateUpdate 将在每帧中调用。

FixedUpdate 如果 MonoBehaviour 被启用，这个函数将在每帧中调用。

Awake 当脚本实例被加载时，Awake 被调用。

StartStart 在所有 Update 方法被第一次调用前调用。

Reset Reset 在用户点击检视面板的上下文菜单或第一次添加该组件时被调用。

OnMouseEnter 当鼠标进入 GUIElement 或 Collider 时，OnMouseEnter 被调用。

OnMouseOver 当鼠标在 GUIElement 或 Collider 时，OnMouseEnter 被调用。

OnMouseExit 当鼠标不再位于 GUIElement 或 Collider 上时，OnMouseExit 被调用。

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

OnMouseUp 当户已经松开鼠标按钮时 OnMouseUp 被调用。

OnMouseDrag 当用户在 EUIElement 或 Collider 上点击并按住鼠标时 OnMouseDrag 被调用。

OnTriggerEnter 当这个 Collider other 进入 trigger 时 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 进 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会在每一帧中都会被调用。

OnCollisionEnter 当这个碰撞器刚体开始触碰另一个刚体碰撞器

时 OnCollisionEnter 被调用。

OnCollisonExit 对于每个与碰撞器刚体停止触碰另一个刚体碰撞器时 OnCollisionExit 被调用。

OnCollisonStay 对于每个与刚体碰撞器相触的碰撞器刚体 OnCollisonStay 将在每一帧中被调用。

OnControllerColliderHit 在移动的时候，控制器碰到一个碰撞器时 OnControllerColliderHit

被调用。

OnJointBreak 当附加到相同游戏物体上的关节被断开时调用。

OnParticleCollision 当一个粒子碰到一个碰撞器时 OnParticleCollision 被调用。

OnBecameVisibleOnBecamevisible 函数在这个渲染器对任何相机变得可见时被调用。

OnBecameInvisibleOnBecamevisible 函数在这个渲染器对任何相机变得不可见时被调用。

OnLevelWasLoaded 这个函数在一个新的关卡被加载之后被调用。

OnEnable 当物体启用或激活时这个函数被调用。
OnDisable 当这个行为禁用或不活动时这个函数被调用。
OnPreCull OnPreCull 在相机开始裁剪场景之前调用。
OnPreRender OnPreRender 在相机开始渲染场景之后调用。
OnPostRenderOnPostRender 在相渲染场景之后调用。
OnRenderObjectOnRenderObject 被用来渲染你自己的物体，使用 Graphics.DrawMesh 或者其他函数。
OnWillRenderObject 如果物体可见，每个相机都会调用 OnWillRenderObject。
OnGUI OnGUI 被调用来渲染并处理 GUI 事件。
OnRenderImageOnRenderImage 在所有渲染完成后被调用，来渲染图片。
OnDrawGizmosSelected 如果你想在物体被选择时绘制 gizmos，实现这个 OnDrawGizmosSelected.
OnDrawGizmos 如果你想绘制可被点选的 gizmos 时，实现 OnDrawGizmos。
OnApplicationPause 当玩家暂停时发送到所有游戏物体。
OnApllicationQuit 在应用退出之前发送到所有游戏物体。
OnPlayerConnected 当一个新玩家成功连接时在服务器上调用这个函数。
OnServerInitialized 当 Network.InitializeServer 被调用并完成时，在服务上调用这个函数。
OnConnectedToServer 当成功链接到服务器上时在客户端调用这个函数。
OnPlayerDisconnected 当玩家从服务器断开时在服务器上调用这个函数。
OndisconnectedFormServer 当链接丢失或服务器断开时在客户端调用这个函数。
OnFailedToConnect 当链接因为某些原失败时在客户端上调用这个

函数
OnFailedToConnectToMasterServer 当链接到主服务器出现问题时在客户端或服务器端调用这个函数。
OnNetworkInstantiate 当一个物体使用 Network.Instantiate 进行网络初始化时在该物体上调用这个函数。
OnSerializeNetworkView 用来在一个被网络视监控的脚本中自定义变量同步。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
operator == 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
NetworkView
类，继承自 Behaviour
网络视是多用户游戏的绑定物。
使用这个你可以准确定义什么应该通过网络同步，和如何同步，游戏物体可以有 NetworkView 组件，该组件可以被定义为观察物体的其他组件，可以在 Network View manual

page 和 component reference page 获取更多信息。

变量

◆ var group : int

描述： 这个网络视所在的网格组。

所有的网格消息和 RPC 调用通过这个组。

```
Function Awake(){
//通过组 1 发送来自这个网格视的所有消息
Network View.group=1;
}
```

◆ var isMine : bool

描述：这个网格视是有这个物体控制吗？

```
Function OnNetworkInstantiate(info: NetworkMessageInfo){
If{networkView.isMine}
Debug.Log("New object instanted by me");
Else
Debug.Log("New object instantiated by"+info.sender);
}
```

◆ var obseved : Component

描述：网络视监控的组件。

```
//打印含有由数组中给出的 viewID 的对象的名称（Print the names of the objects which have the view IDs given in the array）
Function PrintNames(viewIDs: Array){
    For (var ID: NetworkViewID in IDs){
        Debug.log("Finding"+ID);
        Var view : NetworkView = networkView.Find(ID);
        Debug.log(view.observed.name);
}
}
```

◆ var owner : NetworkPlayer

描述： 拥有这个网络视的 NetworkPlayer

```
function OnNetworkInstantiate(info: NetworkMessageInfo){
    if(!networkView.isMine)
        Debug.log("New object instantiated by" + networkView.owner);
}
```

◆ var stateSynchrohization: NetworkStateSynchronization

描述：为这个网络视设置的 NetworkStateSynchronization 类型。

确保 NetworkView 在所有机器上使用相同可靠的方法是你的责任。在状态同步已经发生后不要在运行时改变状态的可靠性。

◆ var viewID: NetworkViewID

描述：这个网络视的 ViewID。

函数

◆ function RPC(name: string, mode: RPCMode, params args:object[]): void

描述：在所有连接端调用一个 RPC 函数。

调用的函数必须有@RPC 标志（[RPC]用于 C Sharp）。NetworkView 必须附加到 GameObject，在这个物体上 RPC 函数能够被调用。NetworkView 用于其他什么地方或者是仅仅用于 RPC 函数是没有关系的。如果他仅仅用于 RPC 函数，state synchronization 应该被关掉。Ovserved 属性设置为 none，在整个场景中 RPC 函数的名称应该是唯一的，如果不同脚本的两个 RPC 函数具有相同的名称，仅有一个会被调用。RPC 调用总是确保执行的顺序与他们调用的顺序相同。用 NetworkView.group 为 NetworkView 设置的通信用，被用于 RPC 调用。为了获取 RPC 自身的信息，可以添加一个 NetworkMessageInfo 参数到函数申明中，它将自动包含这个信息。这样的做的时候你不需要改变调用 RPC 函数的方式，可以参考 manual 的 RPC 部分以便获取更多关于 RPC 的信息。可用的 RPC 参数为 int，float，string，NetworkPlayer，NetworkViewID，Vector3 和 Quaternion。

```
var cubePrefab : Transform;
function OnGUI()
{
    if (GUILayout.Button("SpawnBox"))
    {
        var viewID = Netwok.AllocateViewID();

        networkView.RPC("SpawnBox",
                        RPCMode.AllBuffered,
                        viewID,
                        transform.position);
     }
}
@RPC
function SpawnBox (viewID : NetworkViewID, location : Vector3) {
// Instantate the prefab locally
var clone : Transform;
clone = Instantiate(cubePrefab, location, Quaternion.identity);
var nView : NetworkView;
nView = clone.GetComponent(NetworkView);
nView.viewID = viewID;
}
```

◆ function RPC(name: string, target: NetworkPlayer, params args: object[]): void

描述：在特定的玩家端调用 RPC

◆ function SetSeope(player: NetworkPlayer, relevancy: bool): bool

描述：相对于一个特定的网络玩家这是网络视的范围

这可以用来实现相关设置，设置它为真假取决于你是否想这个玩家从网络接收更新。

类方法

◆ static function Find(viewID: NetworkViewID): NetworkView

描述：基于 NetworkViewID 查找一个网络视。

```
// 打印物体的名称，这些物体具有数组中给定的视 ID (Print the names of the objects which have the view IDs given in the array)
function PrintNames(viewIDs : Array) {
for (var ID: NetworkViewID in IDs) {
```

```
Debug.Log("Finding "+ID);
var view : NetworkView = networkView.Find(ID);
Debug.Log(view.observed.name);
}
}
```

继承的成员

继承的变量

enabled 启用 Behaviours 被更新，禁用 Behaviours 不被更新。

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 GameObject 的 Light（如果没有为 null）。

animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）

guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）

collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏物体的标签。

name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。

GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。

GetComponentsInChildren 返回所有 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。

GetComponents 返回 GameObject 上所有 type 类型的组件。

CompareTag 这个游戏物体标签为 tag？

SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。

SendMessage 在这个游戏物体的每个 MonoBehaviour 上调用 methodName 方法。

BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
operator == 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
Projector
类，继承自 Behaviour。
脚本界面就是一个 Projector 组件。
这个 Projector 可用于场景工程中的任何材料——就像一个真实的世界。这类的属性值就是 Projector 的检查值。
它可以用来执行斑点或投射阴影。你也可以投射纹理动画或渲染纹理在电影场景的另一个部分。所有物体投影的视图在其观点和提供的材料中。
没有快捷方式在游戏对象或者组件来访问 Projector，所以你必须使用 GetComponent 来做：

```
function Start() {
// 获取 projector
Var proj.Projector=GetComponent(Projector);
//使用这个
Proj.nearClipPlane=0.5;
}
```

参见：projector.component;
变量
◆var aspectRatio:float
描述：投影的长宽比。
这个是投影的宽度除以高度。比为 1.0 使这个投影为正方形：比为 2.0 使得宽为高的 2 倍。

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.aspectRatio = 2.0;
}
```

参见：projector component.
◆var farClipPlane:float
描述：远裁剪面的距离。
投影器将不会影响任何运离这个距离的物体。

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.farClipPlane = 20.0;
}
```

参见：projector component
◆var fieldofView:float
描述：投影的视野，以度为单位。

这是垂直视野：水平 FOV 取决于 aspectRatio。当投射器是正交时 fieldOfView 被忽略（参考 orthographic）

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.fieldOfView = 80.0;
}
```

参见：project component

◆var ignoreLayers:int

描述：那个物体层将这个投射器忽略。

参见 layer mask

缺省为零，没有层被忽略，在 ignoreLayers 中设置的每个位将使这个层不会被投射器影响。

```
function Start() {
var proj : Projector = GetComponent (Projector);
//使投射器忽略默认的（0）层
proj.ignoreLayers = (1<<0);
}
```

参见：projector component,Layers.

◆var material:material

描述：这个材质将被投射到每个物体上

如果没有设置材质，投射器不会做任何事情，Standard Assets 中的 Blob.Shadow 文件夹包含一个投射器材质的例子

参见：projector component,Material 类。

◆var nearClipPlane:float

描述：近裁剪面的距离。

投影器将不会影响任何比这个距离近的物体。

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.nearClipPlane = 0.5;
}
```

参见：projector component;

◆var orthographic:bool

描述：投射是正交的（true）还是透视的（false）?

当正交为 true 时，投影被 orghographicSize 定义。

当正交为 false 时，投射被 fieldOfView 定义

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.orthographic = true;
}
```

参见：projector component

◆var orthographicSize:float

描述：在正交模式下投射的一半尺寸。

这个为投影体垂直大小的一半。水平投射的大小取决于 aspectRatio,投射器不是正交时，

orthographicSize 被忽略（参考 orthographic）

```
function Start() {
var proj : Projector = GetComponent (Projector);
proj.orthographic = true;
proj.orthographicSize = 2.0;
}
```

参见：projector component

继承的成员

继承的变量

enabled 启用 Behaviours 被更新，禁用 Behaviours 不被更新。

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 GameObject 的 Light（如果没有为 null）。

animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

Skybox

类，继承自 Behaviour

Skybox component 的脚本接口

天空盒只有 material 属性

参见：skybox component

变量

◆var material:Material

描述：该天空盒使用的材质。

参见：skybox component

继承的成员

继承的变量

Enabled 启用 Behaviours 被更新，禁用 Behaviours 不被更新。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

collider

类，继承自 Component

所有碰撞器的基类

参见：BoxCollider,SphereCollider,CapsuleCollider,MeshCollider,PhysicCollider,Rigidbody.

如果你打算经常移动一个碰撞器，建议再附加一个运动学刚体。

变量

◆var attachedRigidbody:Rigidbody

描述：该碰撞器所附加的刚体。

如果碰撞器附加到非刚体上返回 null。

碰撞器被自动链接到刚体，这个刚体附加在与碰撞器相同的游戏物体或者父游戏物体上。

```
//升起附加在碰撞器上的刚体
Collider.attachedRigidbody.AddForce(1,0,1);
```

◆var bounds:Bounds

描述：碰撞器在世界空间中的包围盒

◆var isTrigger:bool

描述：该碰撞器是一个触发器？

触发器不会与刚体碰撞，当刚体进入或离开这个触发器时，触发器将发送 OnTriggerEnter OnTriggerExit 和 OnTriggerStay.

```
//将附加的碰撞器转化为一个触发器
Collider.isTrigger=true;
```

◆var material:PhysicMaterial

描述：该碰撞器使用的材质。

如果材质被碰撞器共享，它将复制材质并将它赋给碰撞器。

```
//让碰撞器你冰一样
Collider.material.dynamicFriction=0;
collider.material.staticFricton=0;
```

◆var sharedMaterial:PhysicMaterial

描述：该碰撞器的共享物理材质。

修改这个材质将改变所有使用这个材质的碰撞器的表面属性。大多数情况下，你只要修改 Collider.material。

```
Var material:PhysicMaterial;
Collider.shareMaterial=material;
```

函数

◆function ClosestPointOnBounds(position:Vector3):Vector3

描述：到碰撞器包围上最近的点。

这可以用来计算受到爆炸伤害时伤害点数。

```
var hitPoints=100.0;
function ApplyHitPoint(explosionPos:Vector3,radius:float){
    //从爆炸位置到刚体表面的距离
    var closetPoint=collider.ClosePointOnBounds(explosionPos);
    var distance=Vector3f.Distance(closestPoint,explosionPos);
    //伤害点数随着到伤害点的距离而降低
    Var hitPoints=1.0-Math.Clamp0|(distance/radius);
    //这是我们要用的最终伤害点数。最大为 10
    hitPoints=10;
}
```

◆function Raycast(ray:Ray,outhitInfo:RaycastHIt,distance:float):bool

参数

ray　　射线的开始点和方向。

hitInfo　如果没有近回真，hitInfo 将包含更多关于碰撞器被碰到什么地方的信息（参考：

RaycastHit）.

Distance 射线的长度

返回：布尔值-当射线碰到任何碰撞器时为真，否则为假。

描述：投射一个 Ray,它忽略所有碰撞器除了这个。

同上使用 ray.origin 和 ray.direction 而不是 origin 和 disrection.

```
var ray=Camera.main.ScreenPointToRay(Input.mousePostion);
var hit:RaycastHit;
if(collider.Raycast(ray,hit,100){
    Debug.drawLine(ray.origin,hit.point);
}
```

消息传递

◆function OnCollisionEnter(collisionInfo:Collision):void

描述：当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时 OnCollisionEnter 被调用。

相对于 OnTriggerEnter,OnCollisionEnter 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞器速度等细节。如果在函数中不使用 CollisionInfo,省略 collisionInfo 参数以避免不必要的计算。注意如果碰撞器附加了一个非动力学刚体，也只发送碰撞事件。

```
◆Function OnCollisionEnter(collision:Collision){
//调试绘制所有的接触点和法线
for(var contact:ContacePoint in collision.contacts){
```

```
            Debug.DrawRay(contact.point,contact.normal,color.white);
        }
        //如果碰撞物体有较大的冲击就播放声音
        If(collision.relativeVelocity.magnitude.2)
            Audio.Play();
        }
        //一枚手榴弹
        //在击中一个表面时初始化一个爆炸预设
        //然后销毁它
        var exploionPrefab:Transform;
        function OnCollisionEnter(collision:Collision){
            //旋转这个物体使 y 轴面向沿着表面法线的方向
            var contact=collision.contact[0];
            var rot=Quaternion.FromToRotation(Vector3.up,contact.normal);
            var pos=contact.point;
            Instantiate(exploionPrefab,pos,rot);
            Destroy(gameObject);
        }
```

◆function OnCollisionExit(collisionInfo:Collision):void

描述：当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用。

相当于 OnTriggerExit,OnCollisionExit 传递 Collision 类而不是 Collider.Collision 类包含接触

点，碰撞速度等细节。如果在函数中不使用 collisionInfo，省略 collisionInfo 参数以避免不必要的计算，注意如果碰撞器附加了一个非动力学刚体，也只发关碰撞事件。

```
function OnCollisionExit(collisionInfo:Collision){
    print("No longer in contact with"+collisionInfo.transform.name);
}
```

◆function OnCollisionStay(collisionInfo:Collision):void

描述：对于每个与刚体/碰撞器相触碰的碰撞器/刚体，OnCollisionStay 将在每一帧中被调用。

相当于 OnTriggerStay,OnCollisionStay 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞速度等细节。如果在函数中不使用 collisionInfo，省略 collisionInfo 参数以避免

不必要的计算。注意如果碰撞器附加了一个非动力学刚体，也只发送碰撞事件。

```
    function OnCollisionStay(collisionInfo:Collision){
        //调试绘制所有的接触点和法线
        for(var contact:ContactPoint in collision.contacts){
Debug.DrawRay(contact.point,contact.normal,color.white);
        }
}
```

◆function OnTriggerEnter(other:Collider):void

描述：当这个 Collider other 进入 trigger 时 OnTriggerEnter 被调用。

这个消息被发送到这个触发器碰撞器和接触到触发器的刚体（或者是碰撞器如果没有刚体）。注意如果碰撞器附加了一个

刚体，也只发送触发器事件。

//销毁所有进入该触发器的物体

```
function OnTriggerEnter(other:Collider){
    Destroy(other.gameObject);
}
```

◆function OnTriggerExit(other:Collider):void

描述：当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

这个消息被发送到触发器和拉触到这个触发器的碰撞器。注意如果碰撞器附加了一个

刚体，也只发送触发器事件。

销毁所有离开该触发器的物体

```
function OnTriggerExit(other:Collider){
    Destroy(other.gameObject);
}
```

◆function OnTriggerStay(other:collider):void

描述：对于每个 Collider other,当它触碰到 trigger 时，OnTriggerStay 会在每一帧中都被调用。

这个消息被发送到触发器和接触到这个触发器的碰撞器。注意如果碰撞器附加了一个刚体，也只发送触发器事件。

//对所进入这个触发器的刚体使用一个向上的力

```
function OnTriggerStay(other:Collider){
    if(other.atachedRigidbody){
        other.attachedRigidbody.AddForce(Vector3.up*10);
    }
}
```

继承的成员

继承的变量

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 GameObject 的 Light（如果没有为 null）。

animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏的标签。

name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为 methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

BosCollider

类，继承自 Collider

一个盒状的简单碰撞器

参见：SphereCollider,copsuleCollider,PhysicMaterial,Rigidbody

变量

◆var center:vector3

描述：box 的中心，基于物体局部空间。

//重置中心到变换的位置

collider.center=Vector3.zero;

◆var size:Vector3

描述：box 的尺寸，基于物体局部空间

该 box 的尺寸将随着变换的缩放面缩放。

//使这个 box 碰撞器变长

collider.size=Vector3(10,1,1);

继承的成员

继承的变量

attachedRigidbody 该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial 该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSourec(如果没有为 null)。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 Ray，它忽略所有的碰撞器除了这个。

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会
在每一帧中都被调用。
OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时
OnCollisionEnter 被调用。
OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用
OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体 OnCollisionStay 将在每
一帧中被调用。
继承的类函数
Operator bool 这个物体存在吗？
Instatiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
Operator== 比较两个物体是否相同。
Operator!= 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
CapsuleCollider
类，继承自 Collider
一个胶囊状的简单碰撞器
Capsules 是术状的并在每端有一个半球。
参见：BoxCollider,SphereCollder，PhysicMaterial,Rigidbody
变量
◆var center:vector3
描述：胶囊的中心，基于物体的局部空间。
//重置中心到变换的位置
collider.center=Vector3.zero;
◆var direction:int
描述：胶囊的方向
0->胶囊的高度沿着 x 轴。1->胶囊的高度沿着 y 轴。2->胶囊的高度沿着 z 轴。
//使胶囊的高度沿着 x 轴
collider.direction=0;
◆var height:float
描述：胶囊的高度，基于物体的局部空间
该胶囊的高度将随着变换的缩放而缩放。注意 height 为包含两端两个半球的实际高度。
collider.height=5;
◆var radius:float
描述：球的半径，基于物体的局部空间。
该胶囊的半径随着变换的缩放而缩放。
collider.radius=1;
继承的成员

继承的变量

attachedRigidbody 该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial 该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

Rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

Camera 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

Light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

Animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce**（如果没有为 **null**）。

Renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSourec(**如果没有为 **null)**。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 **GameObject** 的 **NetworkView**（如果没有为 **null**）。

Collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。

hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。

particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 **Ray**，它忽略所有的碰撞器除了这个。

GetComponet 返回 **type** 类型的组件，如果游戏物体上附加一个，如果没有返回 **null**。

GetComponentInchildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 **type** 类型的组件，这些组件位于 **GameObject** 或者任何它的子物体上。

GetComponets 返回 **Gameobject** 所有 **type** 类型的组件。

CompareTag 这游戏物体被标签为 **tag?**

SendMessageUpwards 在这游戏物体的每个 **MonoBehaviour** 和该行为的组先上调用名为 **methodName** 方法。

SendMessage 在这游戏物体的每个 **MonoBehaviour** 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。

GetInstanceID 返回该物体的实例 **id**。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会

在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时

OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体 OnCollisionStay 将在每

一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

CharactController

类，继承自 Collider

CharacterController 允许你很容易地做受到碰撞制约的移动，而无需处理刚体。

CharacterController 不会受力的影响，并且只有在调用 Move 函数时才会移动，然后这个控制器将实现移动，但是会受到碰撞的制约。

参见：Character Controller component 和 Character animation examples

变量

◆var center:Vector3

描述：角色胶囊相对于变换的位置的中心。

```
//向上移动胶囊的中心
var Controller. CharacterController=GetComponent(CharacterController);
controller.center=Vector3(0,1,0);
```

◆var collisionFlags:CollisionFlags

描述：在最后一次调用 CharacterController.Move 时，胶囊的哪个部分与环境发生了碰撞。

```
function Update(){
    var controller:CharacterController=GetComponent(CharacterController);
    if((controller collisionFlags&collisionFlags.Above)!=0)
        print("touched the celling");
}
```

◆var detectCollisions:bool

描述：其他刚体或角色控制器应该与这个角色控制器碰撞吗（默认总是启用的）？

这个方法不会影响角色移动时的碰撞检测。它控制角色和其他物体之间的碰撞。例如，

一个箱子可以阻止控制器的运动，但是作为模拟的一部分这个箱子可以进入控制器。这可以用于临时禁用角色控制器。例如，你也许想装载角色到汽车上并禁用碰撞直到再次退出汽车。注意这只影响其他没有与碰撞的物体。**detectCollisions** 不是可序列化的。也就说，它不会显示要检视面板中并且当在场景中实例化或保存这个控制器时，它将不被保存。

```
GetComponent(CharacterController).detectCollisions=false;
```

◆**var height:float**

描述：这个胶囊的高度

```
//设置控制器的高度为 2.0
var controller:CharacterController=GetComponent(CharacterController);
controller.height=2.0;
```

◆**var isGrounded:bool**

描述：**CharacterController** 上一次移动的时候是否接触地面？

```
function Update(){
    var controller.CharacterController=GetComponent(CharacterController);
    if(controller.isGronded){
        print("We are grounded")
    }
}
```

◆**var radius:float**

描述：角色胶囊的半径

```
//这只控制器的半径为 0.3
var controller:CharacterController=GetComponent(CharacterController);
controller.radius=0.3;
```

◆**var slopLimit:float**

描述：角色控制器的斜度限制

```
//设置控制器的斜度限制为 45
var controller:CharacterController=GetComponent(CharacterController);
controller.slopLimit=45.0;
```

◆**var stepOffset:float**

描述：角色控制器的步高，以米为单位。

```
//这只控制器的步高为 2.0
var controller:CharacterController=GetComponent(CharacterController);
controller.stepOffset=2.0;
```

◆**var velocity:Vector3**

描述：角色的当前速度

这允许你知道角色实际的行走有多快，例如当它碰到墙时这个值为零向量

```
function Update(){
        var controller:CharacterController=GetComponent(CharacterController);
        var horizontalVelocity=controller.velocity;
        horizontalVelocity=0;
        //x-z 平面上的速率，忽略其他任何速率
```

```
        var horizontalSpeed= horizontalVelocity.y;
        //整体速率
        var overallSpeed=controller.velocity.magnitude;
}
```

函数

◆function Move(motion:Vector3):CollisionFlags

描述：一个更复杂的移动函数，可以使用绝对移动增量。

试图由 motion 来移动控制器，这个运动只受制于碰撞，它将沿着碰撞器滑动，collisionFlags 为移动期间所发生的碰撞总和。这个函数不使用任何重力。

```
//这个脚本基于方向键向前
//和两边移动角色控制器
//按下空格后它也会跳跃
//确保这个与角色控制器附加在同一个游戏物体上
var speed = 6.0;
var jumpSpeed = 8.0;
var gravity = 20.0;
private var moveDirection = Vector3.zero;
function FixedUpdate() {
var controller : CharacterController = GetComponent(CharacterController);
if (controller.isGrounded) {
//我们在地面上，因此重计算
//直接沿轴方向移动
moveDirection = Vector3(Input.GetAxis("Horizontal"), 0,
Input.GetAxis("Vertical"));
moveDirection = transform.TransformDirection(moveDirection);
moveDirection *= speed;
if (Input.GetButton ("Jump")) {
moveDirection.y = jumpSpeed;
}
}
// 使用重力
moveDirection.y -= gravity * Time.deltaTime;
// 移动控制器
controller.Move(moveDirection * Time.deltaTime);
}
```

◆function SimpleMove(speed:Vector3);bool

描述：以 speed 移动角色

沿着 y 轴的速度将忽略。速度单位 m/s，重力被自动应用。返回角色是否在地面上。

```
var speed = 3.0;
var rotateSpeed = 3.0;
function Update (){
    var controller : CharacterController = GetComponent(CharacterController);
    // Rotate around y - axis
```

```
    transform.Rotate(0, Input.GetAxis ("Horizontal") * rotateSpeed, 0);
    // Move forward / backward
    var forward = transform.TransformDirection(Vector3.forward);
    var curSpeed = speed * Input.GetAxis ("Vertical");
    controller.SimpleMove(forward * curSpeed);
}
    @script RequireComponent(CharacterController)
```

消息传递

◆function OnControllerColliderHit(hit:ControllerColliderHit):void

描述：在移动的时候，控制器碰到一个碰撞器时，OnControllerColliderHit 被调用。这可以用来在角色碰到物体时推开物体。

```
//这个脚本推开所有碰到的刚体
var pushPower = 2.0;
function OnControllerColliderHit (hit : ControllerColliderHit){
    var body : Rigidbody = hit.collider.attachedRigidbody;
    // 无刚体
    if (body == null || body.isKinematic)
        return;
    // 不推开我们身后的物体
    if (hit.moveDirection.y < -0.3)
        return;
    //从移动方向计算推的方向
    // 只推开物体到旁边而不是上下
    var pushDir = Vector3 (hit.moveDirection.x, 0, hit.moveDirection.z);
    //如果知道角色移动有多快
    //然后你就可以用它乘以推动速度
    //使用推力
    body.velocity = pushDir * pushPower;
}
```

继承的成员

继承的变量

attachedRigidbody　　该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial　　该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light　　附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSourec(如果没有为 null)。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 Ray，它忽略所有的碰撞器除了这个。

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会

在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时

OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体

OnCollisionStay 将在每一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

MeshCollider

类，继承自 Collider

网格碰撞器允许人在网格和几何体之间进行碰撞检测。

参见：BosCollider,CapsuleCollider,PhysicMaterial,Rigidbody

变量

◆var convex:bool

描述：为这个网格将使用一个凸碰撞器。

凸网格可以与其他凸碰撞器和非凸网格碰撞。因此凸网格碰撞器适用于刚体，如果你真的需要比几何碰撞器更多的详细的碰撞信息，可以使用这个。

◆var shareMesh:mesh

描述：用于碰撞检测的网格物体。

◆var smoothSphereCollision:bool

描述：为球形碰撞使用插值法线而不是平面多边形法线。

这可以让球体在平面上的滚动更加更滑。缺点是在从陡峭的角度滚落时，它的行为非常奇怪，显得有拉动某个方向的的球体。

继承的成员

继承的变量

attachedRigidbody 该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial 该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSourec(如果没有为 null)。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 Ray，它忽略所有的碰撞器除了这个。

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，

会在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时

OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体

OnCollisionStay 将在每一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

RaycastCollider

类，继承自 Collider

基于碰撞的一个线。

Raycast 碰撞器主要用于模型汽车，气垫船和船因为它们提供更精确的碰撞检测。然而，对于轮式交通工具，建议使用 WheelCollider。

一个 Raycast 碰撞器总是沿着本地 y 轴向下投射一个射线。Raycast 碰撞器在与设置为具有弹性的物理材质组全时是最有用的。

变量

◆var center:Vector3

描述：胶囊的中心，基于物体的局部空间。

//重置中心到变换的位置

collider.center=Vector3.zero;

◆var length:float

描述：本地空间的射线的长度，射线将发射

从 center 沿着变换的负 y 轴，length 将随着变换的缩放而缩放

collider.length=2;

继承的成员

继承的变量

attachedRigidbody　　该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial　　该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Light　　附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio　附加到这个 GameObject 的 AudioSourec(如果没有为 null)。

guiText　附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider　附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter　　附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag　　这个游戏的标签。

Name　对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast　投射一个 Ray，它忽略所有的碰撞器除了这个。

GetComponet　　返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren　返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren　返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag　这游戏物体被标签为 tag?

SendMessageUpwards　　在这游戏物体的每个 MonoBehaviour 和该行为的组先上调

用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为 methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时 OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体 OnCollisionStay 将在每一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

SphereCollider

类继承自 Collider

一个球形的几何碰撞器

参见：BoxCollider，CapsuleCollider，Physic Material，Rigidbody

变量

◆var center:Vector3

描述：球的中心，基于物体的局部空间。

//重置中心到变换的位置

Collider.center=Vector3.zero;

◆var radius:float

描述：球的半径，基于物体的局部空间。

球体随着变换的缩放而缩放

collider.radius=10;

继承的成员

继承的变量

attachedRigidbody 该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial 该碰撞器的共享物体材质。

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSourec(如果没有为 null)。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 Ray，它忽略所有的碰撞器除了这个。

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会

在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时

OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体
OnCollisionStay 将在每一帧中被调用。
继承的类函数
Operator bool 这个物体存在吗？
Instatiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
Operator== 比较两个物体是否相同。
Operator!= 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
TerrainCollider
类，继承自 Collider
基于高度图的碰撞器。
变量
◆var terrainData:TerrainData
描述：存储高度图的地形
继承的成员
继承的变量
attachedRigidbody 该碰撞器所附加的刚体。
isTrigger 该碰撞器是一个触发器？
material 该碰撞器使用的材质。
sharedMaterial 该碰撞器的共享物体材质。
bounds 碰撞器在世界空间中包围盘。
Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
Light 附加到这个 GameObject 的 Light（如果没有为 null）。
Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。
Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSourec(如果没有为 null)。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。
Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
Tag 这个游戏的标签。
Name 对象的名称。
hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？
继承的函数
ClosestPointOnBounds 附加碰撞器到包围盒最近的点。
Raycast 投射一个 Ray，它忽略所有的碰撞器除了这个。
GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为

methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会

在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时

OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体

OnCollisionStay 将在每一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

WheelCollider

类，继承自 Collider

用于车轮的特殊碰撞器。

车轮的碰撞器用来模拟车轮。它的功能类似与 RaycastCollider，但是还有车轮物理和基于轮胎摩擦力模拟的滑动。在大多数情况下 WheelCollider 是更精确的也更容易使用的。

车轮的碰撞检测是通过从 center 沿着局部 Y 轴向下投射一个射线进行的。车轮有一个 radius 并可以通过 suspensionDistance 向下扩展。

车轮使用 motoTorque,brakeTorque 和 steerAngle 属性控制。

车轮碰撞器使用不同于物理引擎的一个基于滑动的摩擦力模型来计算摩擦力。这允许更加真实的行为。而且使车轮忽略标准的 PhysiMaterial 设置。通过改变车轮所碰到的 rorwardFriction 和 sidewaysFriction 来模拟不同的路面材质。参见：GetGroundHit 和 WheelFrictionCurve。

变量

◆var brakeTorque:float

描述：制动的力矩。必须为正

//制动车轮

collider.brakeTorque=1000；

◆var center:vector3

描述：车轮的中心，基于物体的局部空间。

//重置中心到变换的位置

collider.center=Vector.zero;

◆var forwardFriction:WheelFrictionCurve

描述；在轮胎所指向上的摩擦力属性

◆var isGrounded:bool

描述：表示当前车轮是否与什么东西发生碰撞（只读）

◆var mass:float

描述：车轮的质量必须比 0 大

collider.mass=1;

◆var motorTorque:float

描述：车轮上的动力力矩。正负根据方向而定。

为了模拟制动，不要使用负的动力力矩，而使用 brakeTorque

//向前旋转的车轮

collider.motorTorque=10;

◆var radius:float

描述：本地空间的车轮的半径。

半径将随着变换的缩放而缩放。

collider.radius=5;

◆var rpm:float

描述：当前车轮轴的旋转速度，以旋转/秒（只读）

◆var sidewaysFriction:WheelfrictionCurve

描述：侧向的轮胎摩擦力属性。

◆var steerAngle:float

描述：转向的角度，总是绕着本地 y 轴。

//转向前

collider.steerAngle=0;

较高的速率使用小的转向角：一点角度就足够了。

◆var suspensionDistance:float

描述：在局部空间下，车轮悬挂的最大扩展距离。

悬挂总是沿着本地 y 轴向下扩展。悬挂过程将随着变换的缩放而缩放。

collider.susensionDistance:=0.1;

◆var suspensionSpring:JointSring

描述：车轮悬挂的参数。悬挂视图到达一个目标的位置

通过添加弹簧和阴尼力。

suspensionSpring.spring 力视图到达这个目标的位置。较大的值使得悬挂更快到达目标位置。

suspensionSpring.damer 力阻尼悬挂速度。较大的值使得悬挂更慢到达目标。

悬挂试图到达 **suspensionSpring.targetPositon.**它是悬挂沿着 **suspensionDistance** 的剩余长度。零值表示完全扩展。**1** 表示完全压缩，缺省的值为 **0**，它匹配于常规汽车的悬挂行为。

函数

◆**function GetGroundHit(out hit:WheelHit):bool**

描述：获取轮胎的地面碰撞数据。

如果车轮碰撞器与某些物体发生了碰撞，返回 **true** 并填充 **hit** 结构。如果车轮没有碰撞。返回 **false** 并保持 **hit** 结构不变。

报告的 **hit** 总是最接近的一个，因为轮胎摩擦力模型不会自动响应其他的 **physicMaterial**。

任何对不同地面材质的模拟必须基于这里所返回的碰撞器材质来手动调整 **forwordFriction** 和 **sidewayFriction** 完成。

继承的成员

继承的变量

attachedRigidbody 该碰撞器所附加的刚体。

isTrigger 该碰撞器是一个触发器？

material 该碰撞器使用的材质。

sharedMaterial 该碰撞器的共享物体材质。

bounds 碰撞器在世界空间中包围盘。

Transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

Light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

Animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce**（如果没有为 **null**）。

Renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSourec(**如果没有为 **null)**。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 **GameObject** 的 **NetworkView**（如果没有为 **null**）。

Collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。

hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。

particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

ClosestPointOnBounds 附加碰撞器到包围盒最近的点。

Raycast 投射一个 **Ray**，它忽略所有的碰撞器除了这个。

GetComponet 返回 **type** 类型的组件，如果游戏物体上附加一个，如果没有返回 **null**。

GetComponentInchildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调
用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为 methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnTriggerEnter 当这个 Collider other 进入 trigger 进 OnTriggerEnter 被调用。

OnTriggerExit 当这个 Collider other 停止触碰 trigger 时 OnTriggerExit 被调用。

OnTriggerStay 对于每个 Collider other，当它触碰到 trigger 时，OnTriggerStay 会
在每一帧中都被调用。

OnCollisionEnter 当这个碰撞器/刚体开始触碰另一个刚体/碰撞器时 OnCollisionEnter 被调用。

OnCollisionExit 当这个碰撞器/刚体停止触碰另一个刚体/碰撞器时 OnCollisionExit 被调用.

OnCollisionStay 对于每个与刚体/碰撞器相触碰的碰撞器/刚体

OnCollisionStay 将在每一帧中被调用。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

Joint

类，继承自 Comonent

Joint 是所有关节的基类。

参见：CharacterJoint，HingJoint,SpringJoint.

变量

◆var anchor:Vector3

描述：关节的移动被限制于绕着这个锚点的位置。

定义在本地空间中的位置。

hingeJoint.anchor=Vector3(2,0,0);

◆var axis:Vector3

描述：物体被限制于绕着这个轴的方向旋转。

定义在本地空间中的轴。

hingeJoint.axis=Vector3.up;

◆var breakForce:float

描述：需要断开关节的力

力可能来自与其他物体的碰撞，应用到刚体的力。**addTorque** 或来自其他关节。

//当一个大于 **10** 的力矩被应用时使用关节断开

hingeJoint.breakForce=10;

//使关节不会被断开

hingeJoint.breakForce=mathf.Infinity;

参见**:OnJointBreak**

◆var breakTorque:float

描述：需要断开关节的力矩

力矩可能来自与其他物体的碰撞，使用 **rigidbody.AddTorque** 或来自其他关节。

//当一个大于 **10** 的力矩被应用时使关节断开

hingeJoint.breakTorque=10;

//使关节不会被断开

hingeJoint.breakTorque=mathf.Infinity;

参见：**OnJointBreak**

◆var connectedBody:Rigidbody

描述：这个关节链接到的另一个刚体的引用。

如果不设置，这个关节将连接物体到世界。

//连接关节到世界而不是其他刚体

hingeJoint.connectedBody=Null;

连接关节到其他物体

var otherBody=Rigidbody;

hingeJoint.connectedBody=otherBody;

消息传递

◆function OnJointBreak(breakForce:float):void

描述：当附加到相同游戏物体上的关节被断开。当关节断开时，**OnJointBreak** 将被调用，应用到关节的断开力将被传入，**OnJointBreak** 之后这个关节自动从游戏物体移除。参见：

Joint.breakForce

继承的成员

继承的变量

Transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

Rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

Camera 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

Light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

Animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce**（如果没有为 **null**）。

Renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSouree**（如果没有为 **null**）。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 GameObject 的 NetworkView（只读）（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是否被隐藏。保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所有 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。
GetComponents 返回 GameObject 上所有 type 类型的组件。
CompareTag 这个游戏物体标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。
SendMessage 在这个游戏物体的每个 MonoBehaviour 上调用 methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectOfType 返回第一个类型为 type 的激活物体。
operator == 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
CharacterJoint
类，继承自 Joint
Joints 的属性主要用于碰撞效果。它们是一个扩展的球窝状 joint，允许你限制 joint 在每个轴上。
变量
◆var highTwistLimit:SoftJointLimit
描述：角色关节原始轴的上限。
这个限制是相对于两个刚体开始模拟时的角度。
◆var lowTwistLimit:SoftJointLimit

描述：角色关节原始轴上的下限。

这个限制是相对于两个刚体开始模拟时的角度。

◆var swing1Limit:SoftJointLimit

描述：在角色关节的原始轴上限制。

限制是对称的。因此，例如 30 将在此-30 到 30 之间限制是相对于两个刚体开始模拟时的角度。

◆var swing2Limit:SoftJointLimit

描述：在角色关节的原始轴上的限制。

限制是对称的。因此，例如 30 将在此-30 到 30 之间限制是相对于两个刚体开始模拟时的角度。

◆var swingAxis:vector3

描述：关节可以绕着旋转的第二轴

CharacterJoint.swing1Limit 是被允许的绕着这个轴旋转的限制。

继承的成员

继承的变量

Transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

Rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

Camera 附加到这个 GameObject 的 Camera（如果没有为 null）。

Light 附加到这个 GameObject 的 Light（如果没有为 null）。

Animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce（如果没有为 null）。

Renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（如果没有为 null）。

Collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

Tag 这个游戏的标签。

Name 对象的名称。

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponet 返回 type 类型的组件，如果游戏物体上附加一个，如果没有返回 null。

GetComponentInchildren 返回 type 类型的组件，这个组件位于 GameObject 或者任何它的子物体上，使用深度优先搜索。

GetComponentsInchildren 返回所有 type 类型的组件，这些组件位于 GameObject 或者任何它的子物体上。

GetComponets 返回 Gameobject 所有 type 类型的组件。

CompareTag 这游戏物体被标签为 tag?

SendMessageUpwards 在这游戏物体的每个 MonoBehaviour 和该行为的组先上调用名为 methodName 方法。

SendMessage 在这游戏物体的每个 MonoBehaviour 上调用名为 methodName 方法。

BoradcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数

Operator bool 这个物体存在吗？

Instatiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectOfType 返回第一个类型为 type 的激活物体。

Operator== 比较两个物体是否相同。

Operator!= 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

ConfigurableJoint

类，继承自 Joint

可配置关节是一种非常灵活的关节，它让你完全控制旋转和线性移动。

你也可以用它的构建所有其他类型的关节，但是它的设置也是非常复杂的。它让你在每个旋转轴线自由度上完全控制 motor.drive 和关节限制。

变量

◆var angularXDrive:jointDrive

描述：定义关节的旋转如何绕着局部 x 轴作用。仅在 Rotation Drive Mode 为 Swing& twist 时使用。

◆var angularXMotion：configurableJointMotion

描述：根据 Low 和 High Angular Xlimit 允许沿着 X 轴的旋转为 free,完全 locked 或者 limited.

◆var angularYLimit:SoftJointLimit

描述：基于原始旋转的增量定义的旋转约束边界。

◆var angularYMotion:ConfigurableJointMotion

描述：根据 Low 和 High AngularZLimit 允许沿着 Z 轴的旋转为 Free，完全 Locked 或者 Limited。

◆var angularYZDrive:JointDrive

描述:定义的旋转如何绕着局部 Y 和 Z 轴作用。仅在 Rotation Drive Mode 为 Swing&Twist 时使用。

◆var angularZLimit:SoftJointLimit

描述：基于到原始旋转的增量定义的旋转约束边界。

◆var angularZmotion:ConfigurableJointMotion

描述：根据 Low 和 High Angular Zlimit 允许沿着 Z 轴的族转为 Free，完全 Locked 或者 Limited.

◆var configuredInWorldSpace:bool

描述：如果启用，所有目标值将在世界空间中计算而不是物体的局空间。

◆var highAngularXLimit:SoftJointLimit

描述：基于到原始旋转的增量定义的最大旋转约束边界。

◆var linearLimit:SoftJointLimit

基于到关节原点的距离确定的移动约束边界定义。

◆var lowAngularXlimit:SoftJointLimit

基于到原始旋转的增量定义的最小旋转约束边界。

◆var projectionAngle:float

描述：到 Connected Body 的距离，在物体折回到一个可接受的位置之前必须超过这个距离。

◆var projectionDistance:float

描述：到 Connected Body 的距离，在物体折回到一个可接受的位置之前必须超过这个距离。

◆var projectionMode:JointProjectionMode

描述：该属性用来在物体偏离太多时候将它回到约束位置。

◆var rotationDriveMode:RotationDriveMode

描述：控制物体使用 X&YZ 或自身的 slerp Dirve 旋转

◆var slerpDrive:JointDrive

描述：定义关节的旋转如何绕着所有的局部轴作用。仅在 Rotation Drive Mode 为 slerp only 时使用。

◆var targetAngularVelocity:Vector3

描述：这是一个 Vector3。它定义了关节应该旋转的角速度。

◆var targetPosition:Vector3

描述：需要关节移动到的位置。

◆var targetVelocity:Vector3

描述：需要关节移动的速度。

◆var xDrive:JointDrive

描述：定义关节的移动如何沿着局部 X 轴作用。

◆var xMotion:JointDrive

描述：按照 Linear Limit 允许沿着 X 轴的移动为 Free，完全 Locked 或者 Limited。

◆var yDrive:JointDrive

描述：定义关节的移动如何沿着局部 Y 轴作用。

◆var yMotion:JointDrive

描述：按照 Linear Limit 允许沿着 Y 轴的移动为 Free，完全 Locked 或者 Limited。

◆var zDrive:JointDrive

描述：定义关节的移动如何沿着局部 Z 轴作用。

◆var zMotion:JointDrive

描述：按照 Linear Limit 允许沿着 Z 轴的移动为 Free，完全 Locked 或者 Limited。

继承的成员

继承的变量

connectedBody 这个关机链接到的另一个刚体的引用。

axis 物体被限制于绕着这个轴的方向旋转。

anchor 关节的移动被限制于绕着这个锚点的位置。

breakForce 需要断开关节的力。

breakTorque 需要断开关节的力矩。

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 GameObject 的 Light（如果没有为 null）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。
constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。
renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。
audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。
guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。
networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。
guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。
collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。
hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。
particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的消息传递
OnJointBreak 当附加到相同游戏物体上的关节被断开时调用。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
FixedJoint
类，继承自 **Joint**

FixedJoint 将两个刚体组合在一起，使他们在边界位置粘合。
参见： CharacterJoint, HingeJoint, SpringJoint.
继承的成员
继承的变量
ConnectedBody 这个关机链接到的另一个刚体的引用。
axis 物体被限制于绕着这个轴的方向旋转。
anchor 关节的移动被限制于绕着这个锚点的位置。
breakForce 需要断开关节的力。
breakTorque 需要断开关节的力矩。
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。
GetComponents 返回 GameObject 上所以 type 类型的组件。

CompareTag 这个游戏物体被标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。
SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。

继承的消息传递

OnJointBreak 当附加到相同游戏物体上的关节被断开时调用。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。

Destroy 移除一个游戏物体，组件或资源。

DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。

FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。

HingeJoint

类，继承自 **Joint**

HingeJoint 组合两个刚体，约束它们的移动就像用一个铰链链接他们一样。

这个类关节对于门市非常好的，但是也能被用于模型链，等等...

HingeJoint 有一个动力，这个能够用来使链接绕着关节的轴旋转。一个弹簧，它通过绕着铰链关节轴旋转达到一个目标角度。和一个限制，用来约束关节角度。

变量

◆ **var angle: folat**

描述：链接相对于静止位置的当前角度。（只读）

两个物体间的静止角度再开始模拟时总是零。

print(hingeJoint.angle);

◆ **var limits: JointLimits**

描述：铰链链接的限制。

这个链接将被限制，这样角度总是在 **limits.min** 和 **limits.max** 之间，链接的角度是相对于静止角度的度数。两个物体间的静止角度再开始模拟时总是零。

为门制作一个铰链限制；

hinge.Joint.limits.min=();

hinge.Joint.limits.minBounce=();

hinge.Joint.limits.max=9();

hinge.Joint.limits.maxBounce=();

修改限制自动地启用它。

◆ **var motor: JointMotor**

描述：动力将使用一个最大力来试图以角度/秒来到达目标速度。

动力试图以角度/秒达到 **motor.targetVelocity** 角速度. 如果 **motor.force** 足够大，动力将只能达到 **motor.targetVelocity**。如果关节旋转的比 **motor.targetVelocity** 快，动力将断开，负

motor.targetVelocity 将使得动力以相反的方向旋转。

motor.force 是动力能够运用的最大力矩。如果它是零动力将禁用。如果

motor.freeSpin 为假，动力将只在旋转比 **motor.targetVelocity** 快时断开。如果 **motor.freeSpin**

为真，动力将不断开。

//制作一个铰链动力以每秒 **90** 度旋转，和一个较大的力。

hingeJoint.motor.force = 100;

hingeJoint.motor.targetVelocity = 90;

hingeJoint.motor.freeSpin = false;

设置 **HingeJoint.useMotor** 为真，将在修改动力时自动地启动动力。

◆ **var spring: JointSpring**

描述：通过添加弹力和阻力，弹簧试图达到一个目标角度。

spring.spring 力视图到达这个目标角度，较大的值使得弹簧更快达到目标位置。

spring.damper 阻尼角速度，较大的值使得弹簧更慢到达目标。

弹簧到达 **spring.targetPosition** 时相对于静止的角度，两个物体间的静止角度在开始模拟时总是零。

//是弹簧尽量到达 **70** 度角；

//这可能是用来发射弹弓；

hingeJoint.spring.spring = 10;

hingeJoint.spring.damper = 3;

hingeJoint.spring.targetPosition = 70;

修正弹簧自动地启用它。

◆ **var useLimits: bool**

描述：启用关节的限制。

hingeJoint.useLimits = true;

◆ **var useMotor: bool**

描述：启动关节的动力。

hingeJoint.useMotor = true;

◆ **var useSpring: bool**

描述：启用关节的弹性。

hingeJoint.useMotor = true;

◆ **var velocity: float**

描述：关节的角速度，度/秒。

print(hingeJoint.velocity);

继承的成员

继承的变量

ConnectedBody 这个关机链接到的另一个刚体的引用。

axis 物体被限制于绕着这个轴的方向旋转。

anchor 关节的移动被限制于绕着这个锚点的位置。

breakForce 需要断开关节的力。

breakTorque 需要断开关节的力矩。

transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。

renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。
GetComponents 返回 GameObject 上所以 type 类型的组件。
CompareTag 这个游戏物体被标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。
SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。
继承的消息传递
OnJointBreak 当附加到相同游戏物体上的关节被断开时调用。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
SpringJoint
类，继承自 Joint
弹性关节连接 2 个刚体，弹力将自动应用以便保持物体在给定的距离内。
弹性试图位置他们的开始距离，因此如果你的关节开始的时候是分离得，那么这个关节将试图维持这个距离。minDistance 和 maxDistance 属性添加这个隐式距离的顶部。
变量
◆ var damper: float

描述：用于阻尼弹簧的阻尼力。

◆ **var maxDistance: float**

描述：两个物体之间相对于它们的初试距离的最大距离。

距离将在 **minDistance** 和 **maxDistance** 之间，该值是相对于场景第一次加载时重心之间的

距离。

◆ **var minDistance: float**

描述：两个物体之间相对于它们的初试距离的最小距离。

距离将在 **minDistance** 和 **maxDistance** 之间，该值是相对于场景第一次加载时重心之间的

距离。

◆ **var spring: folat**

描述：弹力用于保持两个物体在一起。

继承的成员

继承的变量

ConnectedBody 这个关机链接到的另一个刚体的引用。

axis 物体被限制于绕着这个轴的方向旋转。

anchor 关节的移动被限制于绕着这个锚点的位置。

breakForce 需要断开关节的力。

breakTorque 需要断开关节的力矩。

transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。

renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。

guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。

collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。

hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。

particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏物体的标签。

name 对象的名称。

hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如
果没有返回 **null**。

GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。

GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或

任何它的子物体上。
GetComponents 返回 GameObject 上所以 type 类型的组件。
CompareTag 这个游戏物体被标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。
SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。
继承的消息传递
OnJointBreak 当附加到相同游戏物体上的关节被断开时调用。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
MeshFilter
类，继承自 Component
一个类用来访问 mesh filter 的 Mesh.
使用这个作为一个程序的网格接口。参见：Mesh class.
变量
◆ var mesh: mesh
描述：返回赋给网格过滤器的实例化 Mesh。
如果没有赋予网格过滤器，一个新的网格将被创建并被导入。
如果赋予网格过滤器的网格被共享，它将被自动赋值并且实例化的网格将被返回。
通过使用 mesh 属性你能只修改单个物体。其他使用相同网格的物体不会被修改。

```
function Update (){
//获取实例化网格
var mesh: Mesh = GetComponent(MeshFilter).mesh;
//随即改变顶点
var vertices = mesh. vertices;
for(var p in vertices)
{
p.y ~= Random.Range(-0.3,0.3);
}
mesh.vertices = vertices;
mesh.RecalculateNormals();
}
```

参见： **Mesh class.**

◆ **var sharedMesh: Mesh**

描述：返回网格过滤器的共享网格。

建议只使用这个函数来读网格数据而不是写，因为你可能修改导入的资源并且使用这个

网格的所有物体都会被影响。

继承的成员

继承的变量

transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。

renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。

guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。

collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。

hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。

particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏物体的标签。

name 对象的名称。

hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。

GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。

GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。

GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。

CompareTag 这个游戏物体被标签为 **tag**？

SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。

SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。

BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。

GetInstanceID 返回该物体的实例 **id**。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
ParticleAnimator
类，继承自 **Component**
粒子动画器随着时间移动你粒子，使用它们将风、力和颜色循环应用到你的粒子系统。
这个类是 **particle animator** 组件的脚本接口。
变量
◆ **var autodestruct: bool**
描述：这个粒子动画器的 **GameObject** 会自动销毁？
当设置为 **true**，该 **GameObject** 将在所有粒子消失后被销毁。
◆ **var colorAnimation: Color[]**
描述：粒子的色彩随着生命循环。
当前，你不能直接修改这个数组的索引。相反，你需要取间整个数组，修改它，然后将
它赋回粒子动画器。

```
//如何通过脚本正确的改变 colorAnimation 颜色
//附加这个脚本到 GameObject 这个物体包含完整的粒子系统
function Start ()
{
var modifiedColors: Color[]=particleAnimatorcolorAnimation;
mofifiedColor[2]=Color.yellow;
particleAnimator.colorAnimation=modifiedColors;
}
```

◆ **var damping: float**
描述：每帧粒子速度减慢多少
值为 **1** 没有阻尼，值越小使它们越慢。
◆ **var doesAnimateColor: bool**
描述：粒子在它们的生命期中循环它们的颜色？
◆ **var force: Vector3**
描述：应用到粒子的力
◆ **var localRotationAxis: Vector3**
描述：粒子绕着旋转的本地空间轴。
◆ **var rndForce: Vector3**
描述：添加到粒子的随机力
用这个来让烟变得更加有活力。
◆ **var sizeGrow: float**
描述：粒子的尺寸如何随着生命期增加
◆ **var worldRotationAxis: Vector3**

描述：粒子绕着旋转的世界空间轴。
继承的成员
继承的变量
transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。
rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。
camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。
light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。
animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。
constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。
renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。
audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。
guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。
networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。
guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。
collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。
hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。
particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较两个物体是否不相同。

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。

ParticleEmitter

类，继承自 Component

粒子发射器的脚本接口

参见：Particle documentation.

变量

◆ var emit: bool

描述：粒子应该被自动发射？

```
//发射粒子三秒
particleEmitter.emit=true;
yield WaitForSeconds(3);
//然后停止
particleEmitter.emit=false;
```

◆ var emitterVelocityScale: float

描述：粒子继承的发射器的速度量。

◆ var localVelocity: Vector3

描述：粒子沿着物体的 X,Y,和 Z 轴的开始速度。

◆ var maxEmission: float

描述：每秒生成的最大粒子的数量

◆ var maxEnergy: float

描述：每个粒子的最大生命期，以秒计。

◆ var maxSize: float

描述：每个粒子生成时的最大尺寸。

◆ var minEmission: float

描述：每秒生成的最小粒子的数量

◆ var minEnergy: float

描述：每个粒子的最小生命期，以秒计。

◆ var minSize: float

描述：每个粒子生成时的最小尺寸。

◆ var particleCount：int

描述：粒子的当前数量（只读）。

◆ var particles：Particle[]

描述：返回所有粒子的一个拷贝和指定所有粒子的数组到当前粒子。

注意，修改了粒子数组后，你必须将它赋回 particleEmitter 才能看到改变。能力以零或小于零的粒子将在赋给粒子数组的时候被销毁。因此，当创造一个完整的新粒子数组时，你需

要显式地设置所有粒子的能量。

```
//附加这个脚本到一个已有的粒子系统上。
function LateUpdate(){
//提取粒子
var particles=particleEmitter particles
for(var i=(), i=particles length; i**){
```

```
//在正弦曲线上上下移动粒子
car yPosition = Mathf.Sin (Time.time) * Time.deltaTime.Particles[i]position *=Vector3 (0,
yPosition, 0)
//使粒子变红
particles[i]color=Color.red
//按照正弦曲线修改粒子大小
particles[i]size=Mathf.Sin(Time.time)*0.2;
}
//将他们拷量回粒子系统
particleEmitter.particles=particles
}
```

◆ var rndVelocity: Vector3

描述：一个沿着 X,Y,和 Z 的随机速度，它被添加到当前速度。

```
//主要沿着 x=z 轴扩展随机速度
particleEmitter.rndVelocity=Vector3(2,0.1,2);
```

◆ var useWorldSpace: bool

描述：如果启用，当发射器移动的时候粒子不会移动。如果为假，当你移动发射器时，粒子将跟随它移动。

```
particleEmitter.useWorldSpace=true;
```

◆ var worldVelocity: Vector3

描述：粒子在世界空间的开始速度，沿着 X,Y,和 Z.

函数

◆ function ClearParticles(): void

描述：从粒子系统中移除所有粒子。

```
particleEmitter.ClearParticles();
```

◆ function Emit(): void

描述：发射大量粒子

根据 minEmission 和 maxEmission 属性使发射器发射随机数量的粒子，

```
//在 min 和 max 直接发射随机数量的粒子
particleEmitter.Emit();
```

◆ function Emit(count: int): void

描述：立即发射 count 数量的粒子

```
//发射 10 个粒子
particleEmitter.Emit(10);
```

◆ function Emit(pos: Vector3, velocity: Vector3, size: float, energy: float, color: color): void

描述：以给定的参数发射一个粒子。

```
//在原点处发射一个粒子
//粒子的大小是 0.2 并且它将存活 2 秒
particleEmitter.Emit(Vector3.zero, Vector3 up, 0.2,2 Color yellow);
```

Renderer

类，继承来自 Component

所以渲染器的一般功能

一个渲染器使物体显示在屏幕上。对于任何游戏物体或组件，它的渲染器可通过一个

renderer 属性来访问。

renderer.enabled=false;//使这个物体不可见!

使用这个类来访问任何物体的渲染器，网格或粒子系统。渲染器可以被禁用使物体不可见（参见 enabled），并且可以通过它们访问并修改材质（参考 material）。

参见：用于 meshes，particles，lines 和 trails 的渲染组件。

变量

◆ var bounds: Bounds

描述：渲染器的包围边界（只读）

这个是在世界空间中完全包围物体的包围盒。

使用 bounds 是方便的，可以粗略地近似物体的位置和长宽高，例如，

renderer.bounds.center 通常是比 transform.position 更精确地"物体的中心"，尤其是当物体

是对称的时候。

参考 Mesh.bounds 属性，这个属性返回局部坐标中网格的边框.

```
//打印包围盒 x 轴最左边的点
print(renderer.bounds.min.x);
//打印包围盒 x 轴最右边的点
print(renderer.bounds.max.x);
//场景视图中绘制一个网格球体
//完全包围这个球体
function OnDrawGizmosSelected(){
//一个完全包围这个包裹盒的球体
var center=renderer.bounds.center;
var radius=renderer.bounds.extents.magnitude;
//绘制它
Gizmos.color=Color.white;
Gizmos.DrawWireSphere(center,radius);
}
```

参见：Bounds 类，Mesh.bounds 属性.

◆ var castShadows: bool

描述：这个物体投射阴影？

```
//使物体不投射阴影
renderer.castShadows=false;
```

参见：receiveShadows, Light.shadows.

◆ var enabled: bool

描述：如果启用使渲染的 3D 物体可见。

```
//使这个物体不可见
renderer.enabled=false;
//使这个物体可见
renderer.enabled=true;
//每秒切换一次物体的可见性
function Update(){
//当前秒是奇数还是偶数
var seconds: int=Time.time;
```

```
var oddeven=(seconds%2)==0;
//据此启用着色器
renderer.enabled=oddeven;
}
```

◆ var lightmapIndex: int

描述：应用到这个渲染器的光照图的索引。

这个索引表示在 LightmapSettings 类中的光照贴图.值-I 表示没有光照贴图被赋值，这个是默认的。这个索引不能大于 254。

一个场景可以有多个光照贴图储存在这里，Renderer 组件可以使用这些光照贴图中的一

个，这就使得它能够在多个物体上使用相同的材质，而这个物体可以使用不同的光照贴图或

同一个光照贴图的不同部分。

参见：LightmapSettings 类，lightmapTilingOffset 属性，ShaderLad properties.

◆ var lightmapTilingOffset: Vector4

描述：用于光照图的平铺和偏移。

一个场景可以有多个光照贴图存储在这里，Renderer 组件可以使用这些光照贴图中的一

个。这就使得它能够在多个物体上使用相同的材质，而每个物体可以使用不同的光照贴图或

同一个光照贴图的不同部分。

向量的 x 和 y 表示光照图缩放，z 和 w 表示光照图偏移.

参见：LightmapSettings 类，lightmapIndex 属性，ShaderLab properties.

◆ var material: Material

描述：这个物体的材质。

修改 material 将只为这个物体改变材质。

如果材质被任何其他渲染器使用，这将克隆这个共享材质并从现在开始使用它。

```
//设置主颜色为红色
renderer.material.color=Color.red;
//每 changeInterval 秒从定义在检视面板中的
//纹理数组中改变这个渲染器的材质
var materials: Material[];
var changeInterval=0.33;
function Update(){
if(materials.length==0)//如果没有材质返回
return;
//计算材质的索引
var index: int=Time.time/changeInterval;
//对于材质数取模，这样动画可以重复
index=index%materials.length;
//赋值它到渲染器
prenderer.sharedMaterial=material[index];
}
```

◆ var materials: Material[]

描述：这个物体的所有材质。

这是一个被渲染其实有的所有材质的数组。Unity 支持一个物体使用多个材质；在这种情况下 materials 包含所有的材质。如果超过一个材质 sharedMaterial 和 material 属性返回

第一个使用的材质。

修改任何在 materials 的材质将只改变那个物体的外观.

```
print("I'm using"+renderer.materials.Length+"material(s)");
```

◆ var receiveShadows: bool

描述：这个物体接收阴影？

```
//使物体不接受阴影
renderer.receiveShadows=false;
```

参见：castShadows

◆ var sharedMaterial: Material

描述：这个物体的共享材质。

修改 sharedMaterial 将改变所有使用这个材质物体的外观，并且改变的材质色绘制并且改变的材质设置也被保存在工程中.

不建议修改由 sharedMaterial 返回的材质。如果你想修改一个渲染器的材质，使用 material。

参见：material 属性.

◆ var sharedMaterial: Material[]

描述：这个物体的所有共享材质。

这是一个渲染其使用的所有材质的数组。Unity 支持一个物体使用多个材质；在这种情况下 sharedMaterials 包含所有的材质。如果有超过一个材质 sharedMaterial 和 material 属性返回第一个使用的材质。

修改任何 sharedMaterial 将改变所有使用这个材质物体的外观，并且改变的材质设置也被保存在工程中.

不建议修改由 sharedMaterials 返回的材质。如果你想修改一个渲染器的材质，使用 material。

```
print("I'm using"+renderer.sharedMaterials.Length+"material(s)");
```

参见：material.sharedMaterial 属性.

消息传递

◆ function OnBecameInvisible(): void

描述：OnBecameVisible 函数在这个物体对任何相机变得不可见时被调用.

这个消息被发送到所有附加在渲染器上的脚本。OnBecameVisible 和 OnBecameInvisible 可以用于只需要在物体可见时才进行的计算。

```
//当它不可见时禁用这个行为
function OnBecameVisible(){
enabled=false;
}
```

注意：当在编辑器中运行时，场景视图相机也会导致这个函数被调用。

参见：OnBecameVisible.

◆ function OnBecamevisible(): void

描述：OnBecameVisible 函数在这个物体对任何相机变得不可见时被调用.

这个消息被发送到所有附加在渲染器上的脚本。OnBecameVisible 和 OnBecameInvisible 可以用于只需要在物体可见时才进行的计算。

```
//当它不可见时禁用这个行为
function OnBecameVisible(){
enabled=true;
}
```

注意：当在编辑器中运行时，场景视图相机也会导致这个函数被调用。

参见：OnBecameInvisible.

继承的成员

继承的变量

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 GameObject 的 Light（如果没有为 null）。

animation 附加到这个 GameObject 的 Animation（如果没有为 null）。

constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。

renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。

audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。

guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。

networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。

guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。

collider 附加到这个 GameObject 的 Collider（如果没有为 null）。

hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。

particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏物体的标签。

name 对象的名称。

hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。

GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。

GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。

GetComponents 返回 GameObject 上所以 type 类型的组件。

CompareTag 这个游戏物体被标签为 tag？

SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。

SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。

BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

GetInstanceID 返回该物体的实例 id。

继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
LineRenderer
类，继承自 **Renderer**
LineRender 用于在 **3D** 空间中绘制浮动的线。
这个类是 **line renderer** 组件的脚本接口.
变量
◆ **var userWorldSpace: bool**
描述：如果启用，这个线定义在世界空间中。
函数
◆ **function SetColors(start: Color, end: Color): void**
描述：设置线开始和结束位置的颜色。
◆ **function SetPostition(index: int, position: Vector3): void**
描述：设置线上点的位置。
参见：**SetVertexCount** 函数.
◆ **function SetVertexCount(count: int): void**
描述：设置线段数
参见：**SetPosition** 函数.
◆ **function SetWidth(start: float, end: float): void**
描述：设置开始和结束位置的线宽。
继承的成员
继承的变量
enabled 如果启用使渲染的 **3D** 物体可见。
castShadows 这个物体投射阴影？
receiveShadows 这个物体接收阴影？
material 这个物体的材质。
sharedMaterial 这个物体的共享材质。
sharedMaterials 这个物体的所有共享材质。
materials 这个物体的所有材质。
bounds 渲染器的包围边界（只读）。
lightmapIndex 应用这个渲染器的光照图的索引。
lightmapTilingOffset 用于光照图的平铺和偏移。
transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。
rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。
camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。
light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。
constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。
renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。
audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。
guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。
networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。
guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。
collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。
hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。
particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的消息传递
OnBecameVisible OnBecameVisible 函数在这个物体对任何相机变得可见时被调用.
OnBecameInvisible OnBecameInvisible 函数在这个物体对任何相机变得可见时被调用.
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
MeshRenderer
类，继承自 Renderer
渲染自 MeshFilter 或 TextMesh 插入的网格。
继承的成员
继承的变量
enabled 如果启用使渲染的 3D 物体可见。
castShadows 这个物体投射阴影？
receiveShadows 这个物体接收阴影？
material 这个物体的材质。
sharedMaterial 这个物体的共享材质。
sharedMaterials 这个物体的所有共享材质。
materials 这个物体的所有材质。
bounds 渲染器的包围边界（只读）。
lightmapIndex 应用这个渲染器的光照图的索引。
lightmapTilingOffset 用于光照图的平铺和偏移。
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何

它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的消息传递
OnBecameVisible OnBecameVisible 函数在这个物体对任何相机变得可见时被调用.
OnBecameInvisible OnBecameInvisible 函数在这个物体对任何相机变得可见时被调用.
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
ParticleRenderer
类，继承自 **Renderer**
在屏幕上渲染粒子。
这个类是 **particle renderer** 组件的脚本接口.
变量
◆ **var camera VelocityScale: float**
描述：粒子被拉伸多少取决于 **camera** 的速度。
如果相机具有较大的速度，使用这个使粒子变得较大。
◆ **var lengthScale: float**
描述：粒子在它的运动方向上拉伸多少。
使用这个使粒子总是比较长。
◆ **var maxParticleSize: float**
描述：最大的粒子尺寸
粒子会给填充率带来严重负担。使用这个设置来确保靠近观察者时，它们不会占用过多的性能。
◆ **var particleRenderMode: ParticleRenderMode**
描述：粒子如何被绘制

◆ var uvAnimationCycles: float
描述：设置 UV 动画循环
◆ var uvAnimationXTile: int
描述：设置水平平铺数。
◆ var uvAnimationYTile: int
描述：设置垂直平铺数。
◆ var velocityScale: float
描述：粒子被拉伸多少取决于"它们移动的多快"。
使用这个使粒子随着它们的速度变长。
继承的成员
继承的变量
enabled 如果启用使渲染的 3D 物体可见。
castShadows 这个物体投射阴影？
receiveShadows 这个物体接收阴影？
material 这个物体的材质。
sharedMaterial 这个物体的共享材质。
sharedMaterials 这个物体的所有共享材质。
materials 这个物体的所有材质。
bounds 渲染器的包围边界（只读）。
lightmapIndex 应用这个渲染器的光照图的索引。
lightmapTilingOffset 用于光照图的平铺和偏移。
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。
GetComponents 返回 GameObject 上所以 type 类型的组件。
CompareTag 这个游戏物体被标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName 方法。
SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。
继承的消息传递
OnBecameVisible OnBecameVisible 函数在这个物体对任何相机变得可见时被调用.
OnBecameInvisible OnBecameInvisible 函数在这个物体对任何相机变得可见时被调用.
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
SkinnedMeshRenderer
类，继承自 Renderer
蒙皮网格过滤器
变量
◆ var bones: Transform[]
描述：用于蒙皮网格的骨骼。

```
function Start(){
gameObject.AddComponent(Animation);
gameObject.AddComponent(MeshFilter);
gameObject.AddComponent(MeshRenderer);
gameObject.AddComponent(SkinnedMeshFilter);
//构建基本网格
var mesh: Mesh=new Mesh();
mesh.vertices=[Vector3(-1,0,0), Vector3(1,0,0), Vector3(-1,5,0), Vector3(1,5,0)];
```

```
mesh.uv=[Vector2(0,0), Vector2(1,0), Vector2(0,0), Vector2(0,1), Vector2(1,1),];
mesh.triangles=[0,1,2,1,3,2];
mesh.RecalculateNormals();
//赋值网格到 mesh filter 和 renderer
GetComponent(MeshFilter).mesh=mesh;
renderer.material=new Material(Shader.Find(" Diffuse"));
//赋值骨骼权值到网格
//使用两个骨骼，一个用于上部的顶点，一个用于下部的顶点
var weights=new BoneWeight[4];
weights[0].boneIndex0=0;
weights[0].weight0=1;
weights[1].boneIndex0=0;
weights[1].weight0=1;
weights[2].boneIndex0=1;
weights[2].weight0=1;
weights[3].boneIndex0=1;
weights[3].weight0=1;
mesh.boneWeights=weights;
//创建骨骼变换并绑定姿势
//一个骨骼在顶部一个在底部
var bones = new Transform[2];
var bindPoses = new Matrix4x4[2];
bones[0] = new GameObject("Lower").transform;
bones[0].parent = transform;
//设置相对于父的位置
bones[0].localRotation = Quaternion.identity;
bones[0].localPosition = Vector3.zero;
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵是相对与根的
//这样我们就能够随意移动根物体了
bindPoses[0] = bones[0].worldToLocalMatrix * transform.localToWorldMatrix;
bones[1] = new GameObject("Upper").transform;
bones[1].parent = transform;
//设置相对于父的位置
bones[1].localRotation = Quaternion.identity;
bones[1].localPosition = Vector3(0,5,0);
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵是相对与根的
//这样我们就能够随意移动根物体了
bindPoses[1] = bones[1].worldToLocalMatrix * transform.localToWorldMatrix;
//赋值骨骼并绑定姿势
GetComponent(SkinnedMeshFilter).bones = bones;
GetComponent(SkinnedMeshFilter).bindPoses = bindPoses;
GetComponent(SkinnedMeshFilter).mesh = mesh;
```

//赋值一个简单的挥动动画到底部的骨骼
var curve = new AnimationCurve();
curve.keys = [new Keyframe(0,0,0,0), new Keyframe(1,3,0,0), new Keyframe(2,0.0,0,0)],
//使用曲线创建剪辑 e
var clip = new AniamtionClip();
clip.SetCurve("Lower", Transform, "m_LocalPosition.z", curve);
//添加并播放剪辑
animation.AddClip(clip, "test");
aniamtion.Play("test"),
}
@ script RequireComponent(Animation)

◆ var quality: SkinQuality

描述：影响单个顶点的最大骨骼数量
renderer.quality = SkinQuality.Bone2;

◆ var sharedMesh: Mesh

描述：用于蒙皮的网格

◆ var skinNormals: bool

描述：如果启用，几何体法线将随着骨骼动画更新。
参见：Skinned Mesh Renderer component.

◆ var updateWhenOffscreen: bool

描述：如果启用，蒙皮网格将在离屏的时候更新。如果禁用，也将禁用动画更新。
参见：Skinned Mesh Renderer component.
继承的成员
继承的变量
enabled 如果启用使渲染的 3D 物体可见。
castShadows 这个物体投射阴影？
receiveShadows 这个物体接收阴影？
material 这个物体的材质。
sharedMaterial 这个物体的共享材质。
sharedMaterials 这个物体的所有共享材质。
materials 这个物体的所有材质。
bounds 渲染器的包围边界（只读）。
lightmapIndex 应用这个渲染器的光照图的索引。
lightmapTilingOffset 用于光照图的平铺和偏移。
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有

为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或
描述：trail 在出生点处的宽度。
参见：endWidth 变量.
◆ var time: float
描述：尾迹多长时间会消失。
继承的成员
继承的变量
enabled 如果启用使渲染的 3D 物体可见。
castShadows 这个物体投射阴影？
receiveShadows 这个物体接收阴影？
material 这个物体的材质。
sharedMaterial 这个物体的共享材质。
sharedMaterials 这个物体的所有共享材质。
materials 这个物体的所有材质。
bounds 渲染器的包围边界（只读）。
lightmapIndex 应用这个渲染器的光照图的索引。
lightmapTilingOffset 用于光照图的平铺和偏移。
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。

guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。
collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。
hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。
particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的消息传递
OnBecameVisible OnBecameVisible 函数在这个物体对任何相机变得可见时被调用.
OnBecameInvisible OnBecameInvisible 函数在这个物体对任何相机变得可见时被调用.
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
Rigidbody
类，继承自 **Component**
通过物理模拟控制一个物体的位置。

Rigidbody 组件控制对象的位置 - 它使物体在重力影响下下落，并可计算物体如何响应碰撞。

当操作刚体参数的时候，你应该在 FixedUpdate 函数中使用它，物理模拟以离散的时间步执行。FixedUpdate 函数在每一步之前被立即调用。

需要注意的事是何时使用刚体：

1. 如果你的模拟看起来像慢动作并且不真实：

这是缩放的问题。如果你的游戏世界非常大，所以的东西将显示的非常慢，确保所有你的模型有审视世界的大小。例如，一个汽车应该有 4 米长，一个角色 2 米高。物体以相同的加速度下落，不论它大还是小，重或是轻。如果你的游戏时间有较大的缩放，物体将还是以相同的加速度下落。但是因为物体都比较大，所以物体的下落显得比较慢。

变量

◆ var angularDrag: float

描述：物体的角阻力。

角阻力可以用来减缓物体的旋转。阻力越大，旋转减缓的越快。

```
rigidbody.angularDrag=10;
```

◆ var angularVelocity: Vector3

描述：刚体的角速度向量

在大多数情况下，你不应该直接修改它，以为这会导致不真实的结果。

```
//根据旋转的速度改变材质
var fastWheelMaterial: Material;
var slowWheelMaterial: Material;
function Update(){
if(rigidbody.angularVelocity.magnitude<5){
renderer.sharedMaterial=slowWheelMaterial;
}
else
{
renderer.sharedMaterial=fastWheelMaterial;
}
}
```

◆ var centerOfMass: Vector3

描述：相对于变换，原点的重心。

如果你不从脚边中设置重心，它将从所有附加到刚体的碰撞器上自动计算，当模拟汽车时，设置重心是非常有用的，可以使它更加稳定。具有较低重心的汽车不太可能倾翻。

```
rigidbody.centerOfMass=Vector3(0,-2,0);
```

◆ var detectCollisions: bool

描述：碰撞检测应该启用？ （默认总是启用的）

禁用碰撞检测是有用的，如果有一个人偶，它被设置为运动学并且你想避免刚体上大量的碰撞检测计算，detectCollisions 是非序列化的，也就是说，它不会显示在检视面板中并且

当在场景中实例化或保存这个刚体时，它将不被保存。

///让动画控制这个刚体并忽略碰撞

```
rigidbody.isKinematic=true;
rigidbody.detectCollision=false;
```

///让刚体使用空，检测碰撞

```
rigidbody.isKinematic=false;
rigidbody.detectCollision=true;
```

◆ **var drag: float**

描述：物体的阻力

阻力可用来减缓物体的速度。阻力越大，旋转减缓的越快。

```
function OpenParachute(){
rigidbody.drag=20;
}
function Update(){
if(Input.GetButton("Space"))
{
OpenParachute();
}
}
```

◆ **var freezeRotation: bool**

描述：控制物理是个改变物体的旋转。

如果 **freezeRotation** 被启用，旋转不会被物体模拟修改。这对于创建第一人称射击时有用的，因为玩家需要使用鼠标完全控制旋转。

//冻结旋转

```
rigidbody.freezeRotation=true;
```

◆ **var intertiaTensor: Vector3**

描述：相对于重心的质量对角惯性张量。

惯性张量是被 **intertiaTensorRotation** 旋转的。如果你不从脚本中设置惯性张量，它将从所以附加到刚体的碰撞器上自动计算。

//长砖的惯性张量

```
rigidbody.intertiaTensor=Vector3(5,1,1);
```

◆ **var intertiaTensorRotation: Quaternion**

描述：惯性张量旋转。

如果你不从脚本中设置惯性张量旋转，它将从所有附加到刚体的碰撞器上自动计算。

//重置惯性张量为变换的坐标系统

```
rigidbody.intertiaTensorRotation=Quaternion,identity;
```

◆ **var interpolation: RigidbodyInterpolation**

描述：插值允许你以固定的帧率平滑物理运行效果。

默认，插值是关闭的。普通的刚体插值用于玩家角色。物理以离散的时间步运行，而显卡以可变的帧率渲染。这可能导致物体的抖动，因为物理和显卡不完全同步。这个效果

是细微的但是通常会在玩家角色上看到，尤其是如果相机跟随主角色。建议为主角色打开

插值，但是禁用其他物体上的插值。

//是刚体插值

rigidbody.interpolation=RigidbodyInterpolation.Interpolate;

◆ var isKinematic: bool

描述：控制物理是够影响这个刚体。

如果 isKinematic 启用，力，碰撞和关节将不会再影响这个刚体。刚体通过改变 transform.postion 由动画或脚本的完全控制。动力学刚体也会通过碰撞或关机影响其他刚体的运动。例如，可以使用关节链接一个普通的刚体到动力学刚体，现在这个刚体受到动力学刚体运动的约束。动力学刚体也被用于制作角色，这个角色通常是由动画驱动的，但是在某些事件中可以通过设置 isKinematic 为 false 来讲它快速转化为一个人偶。

//不让刚体受到物理的影响!

rigidbody.isKinematic=true;

◆ var mass: float

描述：刚体的质量

你应该保持质量接近 0.1 并且不要超过 10。大的质量会使物理模拟不稳定。

当碰撞时较大质量的物体推动较小质量的物体。考虑一个大卡车，装上一个小汽车。

一个常见的错误时重的物体比轻的物体下落的快。这是不对的，速度依赖于重力和阻力。

rigidbody.mass=0.5;

◆ var maxAngularVelocity: float

描述：刚体的最大角速度向量（默认 7）范围{0，infinity}

刚体的角速度最大为 maxAngularVelocity 以避免高速旋转物体的数值不稳定性。因为这也许会阻止企图快速旋转的物体，例如车轮，你可以使用逐刚体重载该值。

rigidbody.maxAngularVelocity=10;

◆ var position: Vector3

描述：刚体的位置

这个与设置 transfor.position 相同，然而 position 只在物理的最后一步被应用到变换。

如果你想连续移动一个刚体或运动学刚体，使用 MovePosition 和 MoveRotation。

```
function Start(){
rigidbody.position=Vectorr3.zero;
}
```

◆ var rotation: Quaternion

描述：刚体的旋转

这个与设置 transform.rotation 相同，然而 rotation 只在物理的最后被应用到变换。如果你想连续移动一个刚体或运动学刚体，使用 MovePosition 和 MoveRotation。

```
function Start(){
rigidbody.rotation=Quaternion.identity;
}
```

◆ var sleepAngularVelocity: float

描述：角速度，低于该值的物体将开始休眠。（默认 0.14）范围{0, infinity}

参考 Rigidbody Sleeping 获取更多信息。

◆ var sleepVelocity: float

描述：线行速度，低于该值的物体将开始休眠。（默认 0.14）范围{0, infinity}

参考 Rigidbody Sleeping 获取更多信息。

```
Rigidbody.sleepingVelocity=0.1;
```

◆ var solverIterationCount: int

描述：允许你覆盖每个刚体的求解迭代数。

solverIterationCount 决定关节和接触点如何精确地计算。如果出现链接的物体震荡和行为怪异，为 solver Iteration Count 设置一个较高的值将改善他们的稳定性。

```
rigidbody.solverIterationCount=10;
```

◆ var useConeFriction: bool

描述：用于该钢铁的立锥摩擦力

这确保所有接触包含的行为将使用锥摩擦力。这对于性能有负面影响。默认这个是关闭的，一个更快和更好的被称为金字塔摩擦的近似方法被使用。在大多数情况下建议保留

这个值为关闭。

◆ var useGravity: bool

描述：控制重力是否影响这个刚体

如果设置为假刚体的行为会像是在外层空间。

在所有进入这个碰撞器的所有刚体上禁用重力

```
function OnTriggerEnter(other: Collider)
{
if(other.attachedRigidbody)
{
other.attachedRigidbody.useGravity=false;
}
}
//启用时将这个碰撞器改变为一个触发器
collider.isTrigger=true;
```

◆ var velocity: Vector3

描述：刚体的速度向量

在大多数情况下，你不应该直接修改速度，因为这会导致不真实的结果。不要再每个物体步设置物体的速度，这将导致不真实的物理模拟。一个典型的例子是，当你在 FPS 中

使用跳的时候，改变速度，因为你想立即改变速度。

```
function FixedUpdate(){
if(Input.GetButtonDown("Jump")){
rigidbody.velocity.y=10;}
}
```

◆ var worldCenterOfMass: Vector3

描述：世界空间中刚体的质量重心（只读）。

函数

◆ function AddExplosionForce (explosionForce : float, explosionPosition : Vector3, explosionRadius : float, upwardsModifier : float = 0.0F, mode : ForceMode = ForceMode.Force): void

描述：应用一个力到刚体来模拟爆炸效果。爆炸力将随着到刚体的距离线形衰减。

这个功能也对人偶有很好的作用。如果 radius 为 0，将使用全部的力不论 position 距离刚体多远.upwardModifier 就像从物体下方使用这个力. 这个是非常有用的，以为爆炸

将向上抛这个物体而不是将它们推向一边，这个看起来非常的酷。值 2 将应用一个力在低

于物体 2 米处，然而不会改变实际的爆炸位置. explosionPositon 是爆炸力被应用的位置. explosionRadius 是爆炸的半径，超过 explosionRadius 距离的刚体将不会受到影响.

```
var radius=5.0;
var power=10.0;
function Start(){
//应用一个爆炸力到所有附加的刚体上
var explosionPos=transform.position;
var colliders: Collider[]=Physics.OverlapSphere(explosionPos, radius);
for (var hit in colliders){
if(!hit)
continue;
if(hit.rigidbody){
hit.rigidbody.AddExplosionForce(power.explosionPos, radius, 3.0);
}
}
}
```

◆ function AddForce(force: Vector3, mode: ForceMode=ForceMode.Force): void

描述：为刚体添加一个力。作为结果刚体将开始移动。

```
//在全局坐标空间中添加一个向上的力
function FixedUpdate(){
rigidbody.AddForce(Vector3.up*10);
}
```

◆ function AddForce(x: float, y: float, z: float, mode: ForceMode=ForceMode.Force): void

描述：为刚体添加一个力，作为结果刚体将开始移动。

```
//全局坐标空间中添加一个向上的力
function FixedUpdate(){
rigidbody.AddForce(0,10,0);
}
```

如果你想在多针帧中使用力，你应该在 FixedUpdate 中而不是 Update 使用使用它。

◆ function AddForceAtPosition(force: Vector3, position: Vector3, mode: ForceMode= ForceMode.Force): void

描述：在位置 position 处使用 force。这个将应用力矩和力到这个物体上。

对于真的的效果 position 应该近似地在刚体表面的范围内. 这个最常用于爆炸。当使用爆炸的时候最好应用力到多帧而不是一帧中。主要当 position 远离刚体的中心时，使用的力

矩阵将非常不真实。

```
function ApplyForce(body: Rigidbody){
```

direction=body.transform.position-transform.position;
body.AddForceAtPosition(direction.normalized, transform.position);
}

◆ function AddRelativeForce(force: Vector3, mode: ForceMode=ForceMode.Force): void

描述：相对于它的坐标系统添加一个力到刚体。
作为结果刚体将开始移动。
//沿着自身 z 轴向前移动刚体
function FixedUpadate(){
rigidbody.AddRelativeForce(Vector3.forward*10);
}

◆ function AddRelativeForce(x: float, y: float, z: float, mode: ForceMode=
ForceMode.Force): void

描述：相对于它的坐标系统添加一个力到刚体。
作为结果刚体将开始移动。
//沿着自身 z 轴向前移动刚体
function FixedUpadate(){
rigidbody.AddRelativeForce(0,0,10);
}
如果你想在多帧中使用力，你应该在 FixedUpdate 中而不是 Update 使用使用它。

◆ function AddRelativeTorque(torque: Vector3, mode: ForceMode=ForceMode.Force):
void

描述：相对于刚体自身的坐标系统，添加一个力矩到刚体。
刚体将绕着 torque 轴旋转。
//绕着全局 y 轴旋转刚体
function FixedUpadate(){
rigidbody.AddRelativeForce(Vector3.up*10);}

◆ function AddRelativeTorque(x: float, y: float, z: float, mode: ForceMode=
ForceMode.Force): void

描述：相对于刚体自身的坐标系统，添加一个力矩到刚体。
刚体将绕着 torque 轴旋转。
//绕着自身 y 轴旋转刚体
function FixedUpadate(){
rigidbody.AddRelativeForce(0,10,0);}
如果你想在多帧中使用力，你应该在 FixedUpdate 中而不是 Update 使用使用它。

◆ function AddTorque((torque: Vector3, mode: ForceMode=ForceMode.Force): void

描述：为刚体添加一个力矩。
刚体将绕着 torque 轴旋转。
//绕着全局 y 轴旋转刚体
function FixedUpadate(){
rigidbody.AddTorque(Vector3.up*10);
}
如果你想在多帧中使用力，你应该在 FixedUpdate 中而不是 Update 使用使用它。

◆ function AddTorque(x: float, y: float, z: float, mode: ForceMode=ForceMode.Force):
void

描述：为刚体添加一个力矩。
刚体将绕着 torque 轴旋转。
/绕着全局 y 轴旋转到刚体
function FixedUpadate(){
rigidbody.AddTorque(0,10,0);
}

◆ function ClosestPointOnBounds(position: Vector3): Vector3

描述：到碰撞器包围盒上最近点。
这可以用来计算受到爆炸伤害时的伤害点数。或计算作用到刚体表面上一个点的爆炸力。

```
var hitPoints=10.0;
function ApplyHitPoints(explostionPos: Vector3, radius: float){
//从爆炸位置到刚体表面的距离
var ClosestPoint=rigidbody.ClosestPointOnBounds(explosionPos);
var distance=Vector3f.Distance(closestPoint, explosionPos);
//伤害点数随着到伤害的距离而降低
var damage=1.0-Mathf.Cllamp01(distance/radius);
//这是我们要用的最终伤害点数。 10 at maximum
damage*=10;
//应用伤害
hitPoints=damage;
}
```

◆ function GetPointVelocity(worldPoint: Vector3): Vector3

描述：刚体在世界空间中 worldPoint 点处的速度.
GetPointVelocity 在计算速度的时候将考虑刚体的 angularVelocity。

```
//打印车轮的速度
point=transform.InverseTransformPoint(Vector3(0,-2,0));
var velocity=rigidbody.GetPointVelocity(point);
print(velocity.magnitude);
```

◆ function GetRelativeVelocity(relativePoint: Vector3): Vector3

描述：相对于刚体在 relativePoint 处的速度。
GetRelativePointVelocity 在计算速度的时候将考虑刚体的 angularVelocity。

```
//打印车轮的速度
var relativeVelocity=rigidbody.GetRelativePointVelocity(Vector3(0,-2,0));
print(relativeVelocity.magnitude);
```

◆ function IsSleeping(): bool

描述：刚体处于休眠？
参考 Rigidbody Sleeping 获取更多信息。

```
if(rigidbody.IsSleeping())
print("Sleeping");
```

◆ function MovePosition(position: Vector3): void

描述：移动刚体到 position.
对于运动学刚体，它基于刚体的运动应用摩擦力。这个让你模拟刚体位于移动平台之上的情况。如果你想其他的刚体与运动学刚体交互，你需要在 FixedUpdate 函数中移动

它。

```
var speed=Vector3(3,0,0);
function FixedUpdate()
{
rigidbody.MovePosition(rigidbody.position+speed*Time.deltaTime);
}
```

◆ function MoveRotation(rot: Quaternion): void

描述：旋转刚体到 rotation.

对于运动学刚体，它基于刚体的运动应用摩擦力。这个让你模拟刚体位于移动/旋转平台之上的情况。如果你想其他的刚体与运动学刚体交互，你需要在 FixedUpdate 函数中移动它。

```
var eulerAngleVelocity=Vector3(0,100,0);
function FixedUpdate()
{
var deltaRotation=Quaternion.Euler(eulerAngleVelocity*Time.deltaTime);
rigidbody.MoveRotation(rigidbody.rotation*deltaRotation);
}
```

◆ function SetDensity(density: float): void

描述：基于附加的刚体和固定的密度设置质量。

这个可以用来设置质量为随着碰撞器的尺寸而缩放。

```
rigidbody.SetDensity(1.5);
```

◆ function Sleep(): void

描述：强制刚体休眠至少一帧

一个常见的使用是从 Awake 中调用它以便使刚体在启用的时候休眠。参考 Rigidbody Sleeping 获取更多信息。

```
rigidbody.Sleeping()
```

◆ function WakeUp(): void

描述：强制刚体苏醒

```
rigidbody.WakeUp()
```

参考 Rigidbody Sleeping 获取更多信息。

消息传递

◆ function OnCollisionEnter(collisionInfo: Collision): void

描述：当这个碰撞器/刚体开始碰撞另一个刚体/碰撞器时 OnCollisionEnter 被调用。

相对于 OnTriggerEnter，OnCollisionEnter 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞速度等细细。如果在函数中不使用 collisionInfo，省略 collisionInfo 参数以避免不必要的计算。主要如果碰撞器附加了一个非动力刚体，也只发送碰撞事件。

```
function OnCollisionEnter(collision:Collision){
//调试绘制所以的接触点和法线
for(var contact:ContactPoint in collision.contacts){
Debug.DrawRay(contact.point, contact.normal, Color.white);
}
//如果碰撞物体有较大的冲劲就播放声音
```

```
if(collision.relativeVelocity.magnitude>2)
audio.Play();
}
//一枚手榴弹
//-在击中一个表面时初始化一个爆炸预设
//-然后销毁它
var explosionPrefab: Transform;
function OnCollisionEnter(collision: Collision){
//旋转这个物体使 y 轴面向沿着表面法线的方向
var contact-collision.contacts[0];
var rot=Quaternion.FormToRotation(Vector3.up, contact.mormal);
var pos=contact.point;
Instantiate(explosionPrefab, pos, rot);
//销毁这个发射物
Destroy(gameObject);
}
```

◆ function OnCollisionExit(collisionInfo: Collision): void

描述：当这个碰撞器/刚体停止碰撞另一个刚体/碰撞器时 OnCollisionExit 被调用。

相对于 OnTriggerExit，OnCollisionExit 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞速度等细细。如果在函数中不使用 collisionInfo，省略 collisionInfo 参数以避免

不必要的计算。主要如果碰撞器附加了一个非动力刚体，也只发送碰撞事件。

```
function OnCollisionExit(collision:Collision){
print("No longer in contact with "+collisionInfo.transform.name);
}
```

◆ function OnCollisionStay(collisionInfo: Collision): void

描述：对于每个与刚体/碰撞器相触碰的碰撞器/刚体，OnCollisionStay 将在每一帧中被调

用。

相对于 OnTriggerStay，OnCollisionStay 传递 Collision 类而不是 Collider.Collision 类包含接触点，碰撞速度等细细。如果在函数中不使用 collisionInfo，省略 collisionInfo 参数以避免

不必要的计算。主要如果碰撞器附加了一个非动力刚体，也只发送碰撞事件。

```
function OnCollisionStay(collision:Collision){
function OnCollisionEnter(collision:Collision){
//调试绘制所以的接触点和法线
for(var contact:ContactPoint in collision.contacts){
Debug.DrawRay(contact.point, contact.normal, Color.white);
}
```

继承的成员

继承的变量

transform 附加到这个 GameObject 的 Transform（如果没有为 null）。

rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。

camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。

light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。
animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。
constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。
renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。
audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。
guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。
networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。
guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。
collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。
hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。
particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
TextMesh
类，继承自 **Component**

text mesh component 的脚本接口
参见：text mesh component.
变量
◆ var font: Font
描述：使用的 Font.
参见：text mesh component.
//设置附加的文本网格的文本
var newFont: Font;
GetComponent(TextMesh).font=newFont;
◆ var font: string
描述：显示的文本.
参见：text mesh component.
//设置附加的文本网格的文本
GetComponent(TextMesh).text="Hello World";
继承的成员
继承的变量
transform 附加到这个 GameObject 的 Transform（如果没有为 null）。
rigidbody 附加到这个 GameObject 的 Rigidbody（如果没有为 null）。
camrea 附加到这个 GameObject 的 Camera（如果没有为 null）。
light 附加到这个 GameObject 的 Light（如果没有为 null）。
animation 附加到这个 GameObject 的 Animation（如果没有为 null）。
constantForce 附加到这个 GameObject 的 ConstantForce 如果没有为 null）。
renderer 附加到这个 GameObject 的 Renderer（如果没有为 null）。
audio 附加到这个 GameObject 的 AudioSource（如果没有为 null）。
guiText 附加到这个 GameObject 的 GUIText（如果没有为 null）。
networkView 附加到这个 GameObject 的 NetworkView（只读）。（如果没有为 null）。
guiTexture 附加到这个 GameObject 的 GUITexture（只读）。（如果没有为 null）。
collider 附加到这个 GameObject 的 Collider（如果没有为 null）。
hingeJoint 附加到这个 GameObject 的 HingeJoint（如果没有为 null）。
particleEmitter 附加到这个 GameObject 的 ParticleEmitter（如果没有为 null）。
gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。
tag 这个游戏物体的标签。
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetComponent 返回 type 类型的组件，如果游戏物体上附加了一个，如
果没有返回 null。
GetComponentInChildren 返回 type 类型的组件，这个组件位于 GameObject 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 type 类型的组件，这些组件位于 GameObject 或任何它的子物体上。
GetComponents 返回 GameObject 上所以 type 类型的组件。
CompareTag 这个游戏物体被标签为 tag？
SendMessageUpwards 在这个游戏物体的每个 MonoBehaviour 和该行为的祖先

上调用名为 methodName 方法。
SendMessage 在这个游戏物体上的每个 MonoBehaviour 上调用
methodName 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 MonoBehaviour 上
调用 methodName 方法。
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
Transform
类，继承自 Behaviour，可计数
物体的位置，旋转和缩放。
场景中的每个物体都有一个变换。这个用来存储并操作物体的位置，旋转和缩放，每个变换可以有一个父，它允许你层次地应用位置，旋转和缩放。这个是可以在层次面板中
看到的层次，它们也支持计数器，这样你可以循环所以使用的子：

```
//变量的所有子像是移动 10 个单位!
for(var child: Transform in transform){
child.position+=Vector3.up*10.0;
}
```

参见：The component reference, Physics 类.
变量
◆ var childCount: int
描述：变化的子的数量。

```
//打印子物体的数量
print(transform.childCount);
```

◆ var eulerAngles: Vector3
描述：旋转作为欧拉角度。
仅仅使用这个变量来读取和设置绝对值角度。不要递增它们，因为当该角度超过 360 度时它将失败。使用 Transform.Rotate 代替.

```
//打印绕着全局 x 轴的旋转角度
print(transform.eulerAngles.x);
//打印绕着全局 y 轴的旋转角度
print(transform.eulerAngles.y);
//打印绕着全局 z 轴的旋转角度
print(transform.eulerAngles.z);
//使用 eulerAngles 赋值绝对旋转
```

```
var yRotation=5.0;
function Update()
{
yRotation+=Input.GetAxis("Horizontal");
transform.eulerAngles=Vector3(10,yRoation,0);
}
```

不要分别设置 eulerAngles 轴的（例如 eulerAngles.x=10)以为这将导致偏移和不期望的旋转。当设置它们为新值的时候，同事全部设置它们，如上所示。Unity 将转化储存在 Transfor.rotation 中的角度。

◆ var forward: Vector3

描述：在世界空间中变换的蓝色轴。

```
//设置刚体的速度为
//沿着变换的蓝色轴
rigidbody.velocity=transform.forward*10;
//计算 target 变换和这个物体之间的角度
var angleBetween=0.0;
var target: Transform;
function Update()
{
var targetDir=target.position-transform.position
angleBetween=Vector3.Angle(transform.forward, targetDir);
}
```

◆ var localEulerAngles: Vector3

描述：相对于父变换旋转的欧拉角度。

仅仅使用这个变量来读取和设置绝对值角度。不要递增它们，因为当该角度超过 360 度时它将失败。使用 Transform.Rotate 代替.

```
//打印绕着父 x 轴的旋转
print(transform.localEulerAngles.x);
//打印绕着父 y 轴的旋转
print(transform.localEulerAngles.y);
//打印绕着父 z 轴的旋转
print(transform.localEulerAngles.z);
```

Unity 将自动转化存储在 Transform.localRotation 中的角度。

◆ var localPosition: Vector3

描述：该变换的位置相对于父变换。如果变换没有父，它与 Transform.position 相同。

```
//移动物体到与父物体相同的位置
Transform.localPosition=Vector3(0,0,0);
//获取相对于父位置的 y 组件
//并打印它到控制台
print(transform.localPosition.y);
```

注意，当计算世界位置时父变换的世界旋转和缩放将被应用到本地位置。这就是说 Transform.position 的 1 单位总是 1 单位。Transform.localPosition 的 1 单位将受到所有父缩放
的影响。

◆ **var localRotation: Quaternion**

描述：该变换相对于父变换旋转的旋转。

Unity 内部以四元组方式存储旋转。使用 **Transform.Rotate** 旋转物体. 使用 **Transform.localEulerAngles** 来修改旋转的欧拉角。

```
//设置旋转与父相同
Transform.localRotation=Quaternion.identity；
```

◆ **var localScale: Vector3**

描述：该变换相对于父的缩放。

```
Transform.localScale.x+=0.1;// 物体加宽 0.1
```

◆ **var localToWorldMatrix: Matrix4x4**

描述：从本地空间到世界空间的变换矩阵（只读）。

如果你对使用矩阵进行坐标变换不熟悉那么使用 **Transform.TransformPoint** 代替.

◆ **var lossyScale: Vector3**

描述：对象的全局缩放。

请注意，如果有一个父变换，该变化具有缩放并且它的子被随意缩放，那么该缩放是有

误差的。因此缩放不能正确地表示在一个 **3** 组件向量中，而是一个 **3x3** 矩阵。这样的表示

是非常麻烦的，而且 **lossyScale** 是一个方便的属性，它尽量匹配实际世界缩放。如果你的

物体部没有偏差，这个值将是完全正确的。如果包含偏差，差别也不回太大。

```
print(transform.lossyScale);
```

◆ **var parent: Transform**

描述：变换的父

改变父将修改相对父的位置，缩放和旋转但是世界空间的位置，旋转和缩放时相同的，

```
//通过使相机成为该物体的子
//使它跟随这个物体.
//获取相机的变换
var cameraTransform=Camera main.transform;
//使它成为当前物体的子
cameraTransform.parent=transform;
//放置在当前物体之后
cameraTransform.localPosition=Vector3.forward*5;
//使它指向这个物体
cameraTransform.LookAt(transform);
//从变换的父上断开
transform.parent=mull;
```

◆ **var position: Vector3**

描述：在世界空间中变换的位置.

```
//移动物体到(0,0,0)
transform.position=Vector3(0,0,0);
//打印位置的 x 组件到控制台
print(transform.position.x);
```

◆ **var right: Vector3**

描述：在世界空间中变换的红色轴.

```
//设置刚体的速度为
//沿着变换的绿色轴
rigidbody.velocity=transform.right*10;
```

◆ var root: Transform

描述：返回层次最顶端的变换。

（这个永远不会为 Null，如果这个 Transform 没有父它返回它自身.）

```
//两个碰撞的物体是否有不同的层次？
function OnCollisionEnter(collision){
if(collision.other.transform.root!=transform.root){
print("The colliding objects are not in the same hierachy");
}
}
```

◆ var rotation: Quaternion

描述：在世界空间中作为 Quaternion 存储的旋转.

Unity 内部以四元组方式存储旋转。使用 Transform.Rotation 旋转物体，使用 Transform.eulerAngles 来修改旋转的欧拉角

```
//重置世界旋转
Transform.Rotation=Quaternion.identity
//平滑地向一个 target 旋转倾斜
var smooth=2.0;
var tiltAngle=30.0;
function Update(){
var tiltAroundZ=Input.GetAxis("Horizontal")*tiltAngles;
var tiltAroundX=Input.GetAxis("Vertical")*tiltAngles;
var target=Quaternion.Euler(tiltAroundX,0,tiltAroundZ);
//向 target 旋转衰减
Transform.Rotation=Quaternion.Slerp(Transform.Rotation,target,Time.deltaTime*smooth
);;
}
```

◆ var up: Vector3

描述：在世界空间中变换的绿色轴。

```
//设置刚体的速度为
//沿着变化的绿色轴
rigidbody.velocity=transform.up*10;
```

◆ var worldToLocalMatrix: Matrix4x4

描述：从世界空间到本地空间的变化矩阵（只读）

如果你对使用矩阵进行坐标变换不熟悉那么使用 Transform.InverseTransformPoint 代替.

函数

◆ function DetachChildren(): void

描述：解除所以的子的父子关系。

如果你想销毁层次的根而不销毁它的子，可以使用这个。

```
transform.DetachChildren();
Destroy(gameObject);
```

参见：Transfrom.parent 来分开/改变单个变换的父.

◆ function find(name: string): Transform

描述：根据 name 查找子并返回它.

如果没有子具有名称 name 返回 null.如果 name 包含 V 字符它将像一个路径名一样穿越层次.

```
//旋转手指
function Update(){
aFinger=transform.find("LeftShoulder/Arm/Hand/Finger");aFinger.Rotate(Time.delta.Time*20,0,0);
}
```

◆ function InverseTransfromDirection(direction: Vector3): Vector3

描述：从世界空间到本地空间变换 direction。相对于 Transform.TransfromDirection. 这个操作不受变换的影响。

```
//变换世界朝向到本地空间；
relative=transform.InverseTransfromDirection(Vector3.forward);
```

◆ function InverseTransfromDirection(x: float, y: float, z: float): Vector3

描述：从世界空间变换方向 x，y，z 到本地空间。相对于 Transform.TransfromDirection. 这个操作不受变换的影响。

```
//变换世界朝向到本地空间；
relative=transform.InverseTransfromDirection(0,0,1);
```

◆ function InverseTransfromPoint(position: Vector3): Vector3

描述：从世界空间到本地空间变换 position。相对于 Transform.TransformPoint. 注意返回位置会受到缩放的影响。如果你在处理方向，使用 Transform.InverseTransfromDirection.

```
//计算相对于相机的变换位置
camera=Camera.main.transform;
cameraRelative=camera.InverseTransfromPoint(transform.position);
if(cameraRelative.z>0){
print("The object is in front of the camera");
}
else{
print("he object is behind of the camera");
}
```

◆ function InverseTransfromPoint(x: float, y: float, z: float): Vector3

描述：从世界空间变换位置到 x，y，z 到本地空间. 相对于 Transform.TransformPoint. 注意返回位置会受到缩放的影响。如果你在处理方向，使用 Transform.InverseTransfromDirection.

```
//相对于这个变换计算世界原点.
relativePoint=transform.InverseTransfromPoint(0,0,0,);
if(RelativePoint.z>0){
print("The world origin is in front of the object");
}
else{
print("he world origin is behind of the object");
```

}

◆ function IsChildOf(parent: Transform): bool

描述：这个变换时 parent 的一个子？

返回一个布尔值，表面改变换是否为给定变换的一个子。如果是为真，否则为假。

```
function OnEnterTrigger(col: Collider){
//在碰撞器和器子碰撞器之间忽略碰撞
//例如，当你有一个带有多个触发碰撞器的复杂角色时
if(col.transform.IsChildOf(transform))
return;
print("Don something here")
}
```

◆ function LookAt(target: Transform, worldUp: Vector3=Vector3.up): void

描述：旋转变换以前向向量指向/target/的当前位置。

然后旋转变换的向上向量为 worldUp 向量. 如果你留空 worldUp 参数，该函数将使用世界 y 轴。worldUp 是唯一一个建议向量，如果前向量与 worldUp 垂直，旋转的向上向量只与 worldUp 向量相同。

```
//这个完成的脚本可以附加到一个相机上使它
//连续地指向另一个物体.
//target 作为一个属性显示在检视面板中
//拖动其他物体到它上面使相机看向它
var target: Transform;
//每帧旋转相机以便使它一直看向目标
function Update(){
transform.LookAt(target);
}
```

◆ function LookAt(worldPosition: Vector3, worldUp: Vector3=Vector3.up): void

描述：旋转该变换以便前向向量指向 worldPosition。

然后旋转变换的向上向量为 worldUp 向量. 如果你留空 worldUp 参数，该函数将使用世界 y 轴。worldUp 是唯一一个建议向量，如果前向量与 worldUp 垂直，旋转的向上向量只与 worldUp 向量相同。

```
//指向位于世界坐标原点的物体
transform.LookAt(Vector3.zero);
```

◆ function Rotate(eulerAngles: Vector3, relativeTo: Space=Space.Self): void

描述：绕着 x 轴旋转 eulerAngles.x 度，绕着 y 轴旋转 eulerAngles.y 度并绕着 z 轴旋转 eulerAngles.z 度.

如果 relativeTo 留空或者设置为 Space.Self 该旋转将绕着变换的本地轴。（当在场景视图中选择该物体时，显示物体的 x，y 和 z 轴）如果 relativeTo 是 Space.World 旋转绕着世界的 x，y，z 轴.

```
function Update(){
//绕着物体的 X 轴以 1 度/秒的速度旋转物体.
```

transform.Rotate(Vector3.right*Time.deltaTime);
//...同时现对于世界坐标
//的 Y 轴以相同的速度旋转，
transform.Rotate(Vector3.up*Time.deltaTime, Space.World);
}

◆ function Rotate(xAngle : float, yAngles : float, zAngles : float, relativeTo: Space= Space.Self): void

描述：绕着 x 轴旋转 xAngle 度，绕着 y 轴旋转 yAngles 度并绕着 z 轴旋转 zAngles 度.
如果 relativeTo 留空或者设置为 Space.Self 该旋转将绕着变换的本地轴。（当在场景视图中选择该物体时，显示物体的 x，y 和 z 轴）如果 relativeTo 是 Space.World 旋转绕着世界的
x，y，z 轴.

function Update(){
//绕着物体的 X 轴以 1 度/秒的速度旋转物体.
transform.Rotate(Time.deltaTime,0,0);
//...同时现对于世界坐标
//的 Y 轴以相同的速度旋转，
transform.Rotate(0.deltaTime, 0, Space.World);
}

◆ function Rotate(axis : Vector3, angles : float, relativeTo: Space=Space.Self): void

描述：绕着 axis 轴旋转 angle 度.
如果 relativeTo 留空或者设置为 Space.Self 该 axis 参数将相对于变换的本地轴。（当在场景视图中选择该物体时，显示物体的 x，y 和 z 轴）如果 relativeTo 是 Space.World 该 axis 相
对于世界的 x，y，z 轴.

function Update(){
//绕着物体的 X 轴以 1 度/秒的速度旋转物体.
transform.Rotate(Vector3.right, Time.deltaTime);
//...同时现对于世界坐标
//的 Y 轴以相同的速度旋转，
transform.Rotate(Vector3.up, Time.deltaTime, Space.World);
}

◆ function RotateAround(point: Vector3, axis : Vector3, angle: float): void

描述：绕着 axis 旋转改变换，并通过世界左边的 point,旋转 angle 度.
这个将同时修改变换的位置和旋转。

function Update(){
//绕着世界原点以 20 度/秒旋转物体.
transform.RotateAround(Vector3.zero, Vector3.up. 20*Time.deltaTime);
}

◆ function TransformDirection(direction: Vector3): Vector3

描述：从本地空间到世界空间变换 direction。
这操作不会受到缩放或变换位置的影响。返回的向量与 direction 有相同的长度.

//计算相对于相机的轴
camera=Camera.main.transform;

cameraRelativeRight=camera.TransformDirection(Vector3.right);

//相对于相机的 x 轴应用一个力

rigidbody.AddForce(cameraRelativeRight*10);

◆ function TransformDirection(x: float, y: float, z: float): Vector3

描述：从本地空间变换方向 x，y，z 到世界空间。

这操作不会受到缩放或变换位置的影响。返回的向量与 direction 有相同的长度.

//计算相对于相机的轴

camera=Camera.main.transform;

cameraRelativeRight=camera.TransformDirection(1,0,0);

//

相对于相机的 x 轴应用一个力

rigidbody.AddForce(cameraRelativeRight*10);

◆ function TransformPoint(position: Vector3): Vector3

描述：从本地空间到世界空间变换 position。

注意返回位置会受到缩放的影响。如果你在处理方向，使用

Transform.TransformDirection.

//你需要在检视面板中赋值一个物体到这个变量

var someObject: GameObject;

//在当前物体的右侧实例化一个物体

thePosition=transform TransformPoint(Vector3.right*2);

Instantiate(someObject, thePosition);

◆ function TransformPoint(x: float, y: float, z: float): Vector3

描述：从本地空间变换位置 x，y，z 到世界空间.

注意返回位置会受到缩放的影响。如果你在处理方向，使用

Transform.TransformDirection.

//你需要在检视面板中赋值一个物体到这个变量

var someObject: GameObject;

//在当前物体的右侧实例化一个物体

thePosition=transform TransformPoint(2,0,0);

Instantiate(someObject, thePosition);

◆ function Translate(translation: Vector3, relativeTo: Space=Space.Self): void

描述：在该方向上移动变换 transform 距离.

如果 relativeTo 留空或者设置为 Space.Self 该运动将相对于变换的本地轴.（当在场景视图中选择该物体时，显示物体的 x，y 和 z 轴）如果 relativeTo 是 Space.World 运动将相对于世界的坐标系统.

function Update(){

//沿着物体的 z 轴向前以 1 单位/秒的速度移动物体

transform.Translate(Vector3.up*Time.deltaTime, Space.World);

}

◆ function Translate(x: float, y: float, z: float, relativeTo: Space=Space.Self): void

描述：沿着 x 轴移动 x，沿着 y 轴移动 y，沿着 z 轴移动.

如果 relativeTo 留空或者设置为 Space.Self 该运动将相对于变换的本地轴.（当在场景视图中选择该物体时，显示物体的 x，y 和 z 轴）如果 relativeTo 是 Space.World 运动将相

对于世

界的坐标系统.

function Update(){

//沿着物体的 z 轴向前以 1 单位/秒的速度移动物体

transform.Translate(0,0，Time.deltaTime);

//世界空间中向上以 1 单位/秒的速度移动物体

transform.Translate(0,Time.deltaTime,0,Space.World);

}

◆ **function Translate(translation: Vector3, relativeTo: Transform): void**

描述：在该方向上移动变换 **translation** 距离.

运动相对于/**relativeTo**/的本地坐标系统. 如果 **relativeTo** 是 **null** 运动将相对于世界的坐标系统.

function Update(){

//相对于相机以 1 单位/秒的速度向右移动物体.

transform.Translate(Vector3.right*Time.deltaTime, Camera.main.transform);

}

◆ **function Translate(x: float, y: float, z: float, relativeTo: transform): void**

描述：沿着 **x** 轴移动 **x**，沿着 **y** 轴移动 **y**，沿着 **z** 轴移动.

运动相对于/**relativeTo**/的本地坐标系统. 如果 **relativeTo** 是 **null** 运动将相对于世界的坐标系统. **function Update(){**

//相对于相机以 1 单位/秒的速度向右移动物体.

transform.Translate(Time.deltaTime, 0,0, Camera.main.transform);

}

继承的成员

继承的变量

transform 附加到这个 **GameObject** 的 **Transform**（如果没有为 **null**）。

rigidbody 附加到这个 **GameObject** 的 **Rigidbody**（如果没有为 **null**）。

camrea 附加到这个 **GameObject** 的 **Camera**（如果没有为 **null**）。

light 附加到这个 **GameObject** 的 **Light**（如果没有为 **null**）。

animation 附加到这个 **GameObject** 的 **Animation**（如果没有为 **null**）。

constantForce 附加到这个 **GameObject** 的 **ConstantForce** 如果没有为 **null**）。

renderer 附加到这个 **GameObject** 的 **Renderer**（如果没有为 **null**）。

audio 附加到这个 **GameObject** 的 **AudioSource**（如果没有为 **null**）。

guiText 附加到这个 **GameObject** 的 **GUIText**（如果没有为 **null**）。

networkView 附加到这个 **GameObject** 的 **NetworkView**（只读）。（如果没有为 **null**）。

guiTexture 附加到这个 **GameObject** 的 **GUITexture**（只读）。（如果没有为 **null**）。

collider 附加到这个 **GameObject** 的 **Collider**（如果没有为 **null**）。

hingeJoint 附加到这个 **GameObject** 的 **HingeJoint**（如果没有为 **null**）。

particleEmitter 附加到这个 **GameObject** 的 **ParticleEmitter**（如果没有为 **null**）。

gameObject 这个组件所附加的游戏物体。一个组件总是附加到一个游戏物体。

tag 这个游戏物体的标签。

name 对象的名称。

hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数
GetComponent 返回 **type** 类型的组件，如果游戏物体上附加了一个，如果没有返回 **null**。
GetComponentInChildren 返回 **type** 类型的组件，这个组件位于 **GameObject** 或任何它的子物体上，使用深度优先搜索。
GetComponentsInChildren 返回所以 **type** 类型的组件，这些组件位于 **GameObject** 或任何它的子物体上。
GetComponents 返回 **GameObject** 上所以 **type** 类型的组件。
CompareTag 这个游戏物体被标签为 **tag**？
SendMessageUpwards 在这个游戏物体的每个 **MonoBehaviour** 和该行为的祖先上调用名为 **methodName** 方法。
SendMessage 在这个游戏物体上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
BroadcastMessage 在这个游戏物体或其任何子上的每个 **MonoBehaviour** 上调用 **methodName** 方法。
GetInstanceID 返回该物体的实例 **id**。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 **original** 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 **obj**。强烈建议使用 **Destroy** 代替。
FindObjectsOfType 返回所有类型为 **type** 的激活物体。
FindObjectsOfType 返回第一个类型为 **type** 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 **target** 不被自动销毁。
Flare
类，继承自 **Object**
一个闪光资源。在组件参考中获取更多关于闪光的信息.
这个类没有属性。他需要在检视面板中设置。你可以应用闪光并在运行时将它赋给一个光源。

```
//在检视面板中公开一个闪光的引用
var newFlare: Flare;
//赋值闪光
light.flare-newFlare;
```

参见：**Flare assets**，**LensFlare** 类.
继承的成员
继承的变量
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 **id**。
继承的类函数
operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
Font
类，继承自 Object
用于字体资源的脚本接口，
可以使用这个类为 GUI 文本或文本网格动态切换字体。
参见：GUIText 和 TextMesh.
变量
◆ var material: Material
描述：这个材质用于字体的显示。
函数
◆ function HasCharacter(c: char): bool
描述：这个字体是否有特定字符？
继承的成员
继承的变量
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
GameObject
类，继承自 Object
Unity 场景中所有实体的基类。
参见：Component
变量
◆ var active: bool
描述：游戏物体是激活的？激活/不激活该游戏物体，
//不激活该游戏物体.
gameObject.active=false;

◆ **var animation: Animation**

描述：附加到这个游戏物体的？？？（只读）（如果没有为 null）

```
var other: GameObject;
other.animation.Play();
```

◆ **var audio: AudioSource**

描述：附加到这个游戏物体的？？？（只读）（如果没有为 null）

```
var other: GameObject;
other.audio.Play();
```

◆ **var camera: Camera**

描述：附加到这个游戏物体的相机（只读）（如果没有为 null）

```
var other: GameObject;
other.camera.fieldOfView=45;
```

◆ **var collider: Collider**

描述：附加到这个游戏物体的碰撞器（只读）（如果没有为 null）

```
var other: GameObject;
other.collider.material.dynamicFriction=1;
```

◆ **var constantForce: ConstantForce**

描述：附加到这个游戏物体的恒定力（只读）（如果没有为 null）

```
var other: GameObject;
other.constantForce.relativeForce=Vector3(0,0,1);
```

◆ **var guiText: GUIText**

描述：附加到这个游戏物体的 GUIText（只读）（如果没有为 null）

```
var other: GameObject;
other.guiText.text="HelloWorld";
```

◆ **var guiTexture: GUITexture**

描述：附加到这个游戏物体的 GUIText（只读）（如果没有为 null）

◆ **var hingeJoint: HingeJoint**

描述：附加到这个游戏物体的 HingeJoint（只读）（如果没有为 null）

```
var other: GameObject;
other.hingeJoint Spring.targetPosition=70;
```

◆ **var layer: int**

描述：游戏物体所在的层，一个层在[0...32]之间.

Layer 可以用来选择性的渲染或忽略投射.

```
//设置游戏物体到忽略投射物体的层上
gameObject.layer=2;
```

◆ **var light: Light**

描述：附加到这个游戏物体的光影（只读）（如果没有为 null）

```
var other: GameObject;
other.light.range=10;
```

◆ **var networkView: NetworkView**

描述：附加到这个游戏物体的网络视（只读）（如果没有为 null）

```
var other: GameObject;
other.networkView.RPC("MyFunction",RPCMode.All,"someValue");
```

◆ **var particleEmitter: ParticleEmitter**

描述：附加到这个游戏物体的粒子发射器（只读）（如果没有为 null）

```
var other: GameObject;
other.particleEmitter.emite=true;
```

◆ var renderer: Renderer

描述：附加到这个游戏物体的渲染器（只读）（如果没有为 null）

```
var other: GameObject;
other.renderer.material.color=Color.green;
```

◆ var rigidbody: Rigidbody

描述：附加到这个游戏物体的刚体（只读）（如果没有为 null）

```
var other: GameObject;
other.rigidbody.AddForce(1,1,1);
```

◆ var tag: string

描述：这个游戏物体的标签。

标签可以用来标识一个游戏物体。标签在使用前必须在标签管理器中定义。

```
gameObject.tag="Player";
```

◆ var transform: Transform

描述：附加到这物体的变换. （如果没有为 null）

```
var other: GameObject;
other.transform.Translate(1,1,1);
```

构造函数

◆ static function GameObject(name:string): GameObject

描述：创建一个新的游戏物体，命名为 name.

Transform 总是被添加到该游戏物体.

```
//创建一个名为"Player"的游戏物体
//并给他添加刚体和立方体碰撞器.
player=new GameObject("Player");
player.AddComponent("Rigidbody");
player.AddComponent("BoxCollider");
```

◆ static function GameObject(): GameObject

描述：创建一个新的游戏物体.

Transform 总是被添加到该游戏物体.

```
//创建一个没有名称的游戏物体
//并给他添加刚体和立方体碰撞器.
player=new GameObject();
player.AddComponent("Rigidbody");
player.AddComponent("BoxCollider");
```

◆ static function GameObject(name: string, params components: Type[]): GameObject

描述：创建一个游戏物体并附加特定的组件.

函数

◆ function AddComponent(className: string): Component

描述：添加一个名为 className 的组件类型到该游戏物体.

使用这个函数来改变物体的行为。你也可以传递脚本的类名来添加脚本到游戏物体。有些组件也需要另一些组件存在于同一个游戏物体上。这个函数会自动添加需要的组件，例如如果你添加一个 HingeJoint 也会自动添加一个 Rigidbody.

//添加名为 FoobarScript 的脚本到游戏物体
gameObject.AddComponent("FoobarScript");
//添加球形碰撞器到游戏物体
gameObject.AddComponent("FoobarCollider");

◆ function AddComponent(componentType: Type): Component

描述：添加一个名为 componentType 类型的类到该游戏物体.

gameObject.AddComponent("FoobarScript");

注意，没有 RemoveComponent()，来移除组件，使用 Object.Destroy.

◆ function BroadcastMessage(methodName: string, parameter: object=mull, option: SendMessageOption=SendMessageOptions.RequireReceiver): void

描述：在这个游戏物体或其任何子上的每个 MonoBehaviour 上调用 methodName 方法。

通过使用零参数，接收方法可以选择忽略 parameter。如果 options 被设置为 SendMessageOptions.RequireReceiver，那么如果这个消息没有被任何组件接收时将打印一个

错误消息。

///使用值 5 调用函数 ApplyDamage
gameObject.BroadcastMessage("ApplyDamage",5);
//所有附加到该游戏物体和其子物体上脚本中的
//ApplyDamage 函数都将调用
function ApplyDamage(damage){
print(damage)
}

◆ function CompareTag(tag: string): bool

描述：这个游戏物体被标签为 tag？

//立即死亡触发器
//销毁任何进入到触发器的碰撞器，这些碰撞器被标记为 Player.
function OnTriggerEnter(other: Collider){
if(other.gameObject.CompareTag("Player"))
{
Destroy(other.gameObject);
}
}

◆ function GetComponent(type: Type): bool

描述：如果游戏物体有 type 类型的组件就返回它，否则返回 null. 你可以使用这个函数

访问内置的组件或脚本.

GetComponent 是防卫其他组件的主要方法。对于 Javascript 脚本的类型总是脚本显示在工程视图中的名称。例如：

function Start()
{
var curTransform: Transform;
curTransform=gameObject.GetComponent(Transform);
//这等同于
curTransform=gameObject.transform;

```
}
function Update() {
//为访问附加在同一游戏物体上
//其他脚本内的公用变量和函数
//(ScriptName 为 Javascript 文件名)
var other: ScriptName=gameObject.GetComponent(ScriptName);
//调用该脚本中的 DoSomething 函数
other DoSomething();
//设置其他脚本实例中的另一个变量
other.someVariable=5;
}
```

◆ **function GetComponent(type: string): Component**

描述：返回名为 **type** 的组件，如果游戏物体上附加了一个就返回它，如果没有返回 **null.** 出于性能的原因最好用 **Type** 电影 **GetComponent** 而不是一个字符串。然而有时你可能无法得到类型，例如当试图从 **Javascript** 中访问 **c#**时。在那种情况下你可以简单的通过名

称而不是类型访问这个组件。例如：

```
function Update()
{
//为访问附加在同一游戏物体上
//其他脚本内的公用变量和函数.
//(ScriptName 为 Javascript 文件名）
var other=gameObject.GetComponent("ScriptName");
//调用该脚本中的 DoSomething 函数
other.DoSomething().
//设置其他脚本实例中的另一个变量
other.someVariable=5;
}
```

◆ **function GetComponentInChildren(type: Type): Component**

描述：返回 **type** 类型的组件，这个组件位于这个游戏物体或任何它的子物体上，使用深度优先搜索。

只有激活的组件被返回。

```
var script: ScriptName=gameObject.GetComponentInChildren(ScriptName);
script.DoSomething();
```

◆ **function GetComponents(type: Type): Component[]**

描述：返回该游戏物体上所有 **type** 类型的组件。

```
//在这个游戏物体和所有它的子物体上
//的 HingeJoints 上禁用弹簧
var hingeJoints=gameObject.GetComponents(HingeJoint);
for(var joint: HingeJoint in hingeJoints){
joint.useSpring=false;
}
```

◆ **function GetComponentsInChildren(type: Type, includeInactive: bool=false): Component[]**

描述：返回所有 type 类型的组件，这些组件位于该游戏物体或任何它的子物体上。只有激活的组件被返回。

```
//在这个游戏物体和所有它的子物体上
//的所有 HingeJoints 上禁用弹簧
var hingeJoints=gameObject.GetComponentsInChildren(HingeJoint);
for(var joint: HingeJoint in hingeJoints){
joint.useSpring=false;
}
```

◆ function SampleAnimation(animation: AnimationClip, time: float): void

描述：在一个特定的时间采样动画，用于任何动画目的。

出于性能考虑建议使用 Animation 接口，这将在给定的 time 采用 animation，任何被动化的组件属性都将被这个采样值替换，多数时候你会使用 Animation.Play. SampleAnimation

用于当你需要以无序方式或给予一些特殊的输入在帧之间跳跃时使用。参见：Aniamtion

```
//通过采样每一帧或动画剪辑
varclip.AniamtionClip
function Update()
{
gameObject.sampleAnimation(clip, clip.length-Time.time);
}
```

◆ function SendMessage(methodName: string, value: object=null, options: SendMessageOption=SendMessageOption.RequireReceiver): void

描述：在这个游戏物体上的每个 MonoBehaviour 上调用 methodName 方法。

通过使用零参数，接收方法可以选择忽略参数。如果 options 被设置为 SendMessageOptions.RequireReceiver，那么如果这个消息没有被任何组件接收时将打印一个

错误消息。

```
//使用值 5 调用函数 ApplyDamage
gameObject.SendMessage("ApplyDamage",5);
//所以附加到该游戏物体上的脚本中的
//AplIyDamage 函数都将调用
function.ApplyDamage(damage){
pring(damage);
}
```

◆ function SendMessageUpwards(methodName: string, value: object=null, options: SendMessageOption=SendMessageOption.RequireReceiver): void

描述：在这个游戏物体上的每个 MonoBehaviour 和该行为的祖先上调用名为 methodName

方法。

通过使用零参数，接收方法可以选择忽略参数。如果 options 被设置为 SendMessageOptions.RequireReceiver，那么如果这个消息没有被任何组件接收时将打印一个

错误消息。

```
//使用值 5 调用函数 ApplyDamage
```

```
gameObject.SendMessageUpwards("ApplyDamage",5);
//所以附加到该游戏物体上的脚本中的
//AplIyDamage 函数都将调用
function.ApplyDamage(damage){
pring(damage);
}
```

◆ function SetActiveRecursion(rotate: bool): void

描述：设置这个物体和所以子游戏物体的机会状态。

```
gameObject.SetActiveRecursion(true);
```

类方法

◆ static function CreatePrimitive(type: PrimitiveType): GameObject

描述：用几何的网格渲染器和适当的碰撞器创建一个游戏物体。

```
///在场景中创建一个平面，球体和立方体
function Start()
{
GameObject.CreatePrimitive(PrimitiveType.Plane);
var cube=GameObject.CreatePrimitive(PrimitiveType.Cube);
cube.transform.position=Vector3(0,0.5,0);
var sphere=GameObject.CreatePrimitive(PrimitiveType.Sphere);
Sphere.transform.position=Vector3(0,1.5,0);
var capsule=GameObject.CreatePrimitive(PrimitiveType.Capsule);
capsule.transform.position=Vector3(2,1,0);
var cylinder=GameObject.CreatePrimitive(PrimitiveType.Cylinder);
cylinder.transform.position=Vector3(-2,1,0);
```

◆ static function Find(name: string): GameObject

描述：依据 name 查找物体并返回它.

如果没有物体具有名称 name 返回 null. 如果 name 包含'/'字符它将像一个路径名一样穿

越层次，这个函数只返回激活的游戏物体。

出于性能考虑建议不要在每帧中都是有该函数，而是在开始时调用并在成员变量中缓存结果

或者用 GameObject.FindWithTag.

```
//这返回场景中名为 Hand 的游戏物体.
hand=GameObject.Find("Hand");
//这将返回名为 Hand 的游戏物体.
//在层次试图中 Hand 也许没有父！
hand=GameObject.Find("/Hand");
//这将返回名为 Hand 的游戏物体.
//它是 Arm>Monster 的子.
//在层次试图中 Monster 也许没有父！
hand=GameObject.Find("/Monster/Arm/Hand");
//这将返回名为 Hand 的游戏物体.
//它是 Arm>Monster 的子.
```

```
//Monster 有父.
hand=GameObject.Find("/Monster/Arm/Hand");
```

这个函数最常用与在加载时自动链接引用到其他物体，例如，在 MonoBehaviour.Awake 或 MonoBehaviour.Start 内部．处于性能考虑你不应该在每帧中调用这个函数，例如 MonoBehaviour.Update 内．一个通用的模式是在 MonoBehaviour.Start 内将一个游戏物体赋给一个变量．并在 MonoBehaviour.Update 中使用这个变量.

```
//在 Start 中找到 Hand 并在每帧中选择它
private var hand: GameObject;
function Start(){
hand=GameObject.Find("/Monster/Arm/Hand");
}
function Update(){
hand.transform.Rotate(0,100*Time.deltaTime,0);
}
```

◆ static function FindGameObjectsWithTag(tag: string): GameObject[]

描述：返回标记为 tag 的激活物体列表，如果没有发现返回 null.

标签在使用前必须在标签管理中定义。

```
//在所有标记为"Respawn"的物体位置处
//实例化 respawnPrefab

var respawnPrefab: GameObject;
var respawns=GameObject.FindGameObjectsWithTag("Respawn");
for(var respawn in respawns)
Instantiate(respawnPrefab, respawn.position, respawn.rotation);
//打印最接近的敌人的名称
print(FindClosestEnemy().name);
//找到最近的敌人的名称
function FindClosestEnemy(): GameObject {
//找到所以标记为 Enemy 的游戏物体
var gos: GameObject[]
gos=GameObject.FindGameObjectsWithTag("Enemy");
var closest: GameObject;
var distance=Mathf.Infinity;
var position=transform.position;
//遍历它们找到最近的一个
for(var go: GameObject in gos){
var diff=(go.transform.position-position);
var curDistance=diff.sqrMagnitude;
if(curDistance<distance){
closest=go;
distance=curDistance;
}
}
```

return closest;
}
◆ static function FindGWithTag(tag: string): GameObject
描述：返回标记为 tag 的一个激活游戏物体，如果没有发现返回 null.
标签在使用前必须在标签管理中定义。
//在标记为"Respawn"的物体位置处
//实例化一个 respawnPrefab
var respawnPrefab: GameObject;
var respawns=GameObject.FindWithTag("Respawn");
Instantiate(respawnPrefab, respawn.position, respawn.rotation);
继承的成员
继承的变量
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
这与使用带有"_MainTex"名称的 GetTextureOffset 或 SetTextureOffset 相同。
//基于时间滚动主纹理
var scrollSpeed=0.5;
function Update(){
var offset=Time.time*scrollspeed;
renderer.material.mainTextureOffset=Vector2(offset,0);
}
参见：SetTextureOffset.GetTextureOffset.
◆ var mainTextureScale: Vector2
描述：主材质的纹理缩放。
这与使用带有"_MainTex"名称的 GetTextureScale 或 SetTextureScale 相同。
function Update(){
var scalex=Mathf.Cus(Timetime)*0.5+1;
var scaleY=Mathf.Sin(Timetime)*0.5+1;
renderer.material.mainTextureScale=Vector2(scaleX,scaleY);
}
参见：SetTextureScale.GetTextureScale.
◆ var passCount: int

描述：在这个材质中有多少个 pass（只读）.

这个最常用在使用 GL 类之间绘制的代码中（只限于 Unity Pro）. 例如，Image Effects 使用

材质来实现屏幕后期处理. 对材质中的每一个 pass（参考 SetPass）它们激活并绘制一个全屏

四边形。

这里是一个全屏图形效果的例子，它反转颜色。添加这个脚本到相机并在播放模式中查看。

```
private var mat: Material;
function Start()
{
mat=new Material(
"Shader\"Hidden/Invert\"{"+
"SubShader{"+
"Pass{"+
"ZTestAlways Cull Off ZWrite Off"+
"SetTexture[_RenderTex]{combine one-texture}"+
"}"+
"{"+
"}"+
);
}
function OnRenderImage(source: RenderTexture, dest: RenderTexture){
RenderTexture.active=dest;
source.SetGlobalShaderProperty("_RenderTex");
GL.PushMatrix();
GL.LoadOrtho();
//对于材质中的每个 pass（这里只有一个）
for(var i=0; i<mat.passCount; ++i){
//激活 pass
mat.SetPass(i);
//绘制一个四边形
GL.Begin(GLQUADS);
GL.TEXCoord2(0,0); GL.Vertex3(0,0,0.1);
GL.TEXCoord2(1,0); GL.Vertex3(1,0,0.1);
GL.TEXCoord2(1,1); GL.Vertex3(1,1,0.1);
GL.TEXCoord2(0,1); GL.Vertex3(0,1,0.1);
GL.End();
}
GL.PopMatrix();
}
```

参见：SetPass 函数，GL 类，ShaderLab documentation.

◆ var renderQueue: int

描述：这个材质的渲染队列（只读）

默认地材质使用 shader 的 render queue，你可以使用这个变量重载该渲染队列。注意一

旦渲染队列在该材质上被设置，它将保持这个值，集市以后 shader 被改变为不同的一个值。

渲染队列值需要时正的才能正常工作。

参见：Shader.renderQueue, RenderQueue tag.

◆ var shader: Shader

描述：该材质使用的着色器。

```
//按下空格键时，
//在 Diffuse 和 Transparent/Diffuse 着色器之间切换
private var shader1=Shader.Find("Diffuse");
private var shader2=Shader.Find("Transparent/Diffuse");
function Update() {
if(Input.GetButtonDown("Jump")){
If(renderer.material.shader--shader1)
rendere.material.shader=shader2;
else
renderer.material.shader=shader1;
}
}
```

参见：Shader.Find 方法，Material, ShaderLab documentation.

构造函数

◆ static function Material(contents: string): Material

描述：从一个源 shader 字符串创建一个材质。

如果你有一个实现自定义特效的脚本，你需要使用着色器和材质实现所有的图像设置。

在你的脚本内使用这个函数创建一个自定义的着色器和材质。在创建材质后，使用 SetColor，

SetTexture, SetFloat, SetVector, SetMatrix 来设置着色器属性值。

```
//创建一个附加混合材质并用它来渲染
var color=Color.white;
function Start()
{
var shader Text=
"shader\"Alpha Additive\"{"+
Properties{_Color(\"Main Color\", Color)=(1,1,1,0)}"+
"SubShader {"+
"Tags {\"Queue\"=\"Transparent\"}"+
"Pass {"+
"Blend One One ZWrite Off ColorMask RGB"+
"Material {Diffuse[_Color]Ambient[_Color]}"+
"Lighting On"+
"SetTexture[_Dummy]{combine primary double, primary}"+
"}"+
"}"+
```

"}";
renderer.material=new Material(shaderText);
renderer.material.color=color;
}
参见：ShaderLab documentation.
函数
◆ function CopyPropertiesFormMaterial(mat: Material): void
描述：从其他材质拷贝属性到这个材质。
◆ function GetColor(propertyName: string): Color
描述：获取一个命名的颜色值。
数多 shader 使用超过一个颜色，使用 GetColor 来获取 propertyName 颜色。
Unity 内置着色器使用的普通颜色名称；
"_Color"为材质的主颜色。这也能够通过 color 属性访问。
"_SpecColor"为材质的反射颜色（在 specular/glossy/vertexlit 着色器中使用）。
"_Emission"为材质的散射颜色（用在 reflective 着色器中使用）。
print(renderder.material.GetColor("_SpecColor));
参见：color 属性，SetColor.
◆ function GetFloat(propertyName: string): float
描述：获取一个命名的浮点值。
参见：SetFloat, Materials, ShaderLab documentation.
◆ function GetMatrix(propertyName: string): Matrix4x4
描述：从该 shader 中获取命名矩阵的值。
这个最常用于自定义的 shader，其中需要额外的矩阵参数，矩阵参数不需要在材质检视面板中公开，但是能够在脚本中通过 SetMatrix 和 GetMatrix 来设置和查询。
参见：SetMatrix, Materials, ShaderLab documentation.
◆ function GetTag(taf: string, searchFallbacks: bool, defaultValue: string=""): string
描述：获取材质的 shader 标签值。
如果材质的 shader 没有定义标签，defaultValue 被返回。
如果 searchFallbacks 为 true 那么这个函数将在所有的子 shader 和所有后备中查找标签。
如果 searchFallbacks 为 false 只在当前查询的子 shader 中查找这个标签。
使用不搜索后备的 GetTag 可以检视现在使用的是哪个子 shader：添加一个自定义具有不同值的标签到每个子 shader，然后再运行时查询这个值。例如，Unity Pro 的水使用这个
函数来检测 shader 何时退化为没有反射，然后关闭反射相机。
◆ function GetTexture(propertyNmae: string): Texture
描述：获取一个命名纹理。
数多 shader 使用超过一个纹理。使用 GetTexture 来获取 propertyName 纹理。
Unity 内置着色器使用的普通纹理名称；
"_MainTex"为主散射纹理. 这也能够通过 mainTexture 属性访问。
"_BumpMap"为法线贴图。
"_LightMap"为光照贴图。

"_Cube"为发射立方体贴图。
function Start(){

var tex=renderer.material.GetTexture("_BumpMap");
if(tex)
print("My bumpmap is "+ tex.name);
else
print("I have no bumpmap!");
}
参见：mainTexture 属性，SetTexture.

◆ function GetTextureOffset(propertyName: string): Vector2
描述：获取纹理 propertyName 的位置偏移。
Unity 内置着色器使用的普通纹理名称；
"_MainTex"为主散射纹理. 这也能够通过 mainTextureOffset 属性访问。
"_BumpMap"为法线贴图。
"_LightMap"为光照贴图。
"_Cube"为发射立方体贴图。
参见：mainTextureOffset 属性，SetTextureOffset.
◆ function GetTextureScale(propertyName: string): Vector2
描述：获取纹理 propertyName 的位置缩放。
Unity 内置着色器使用的普通纹理名称；
"_MainTex"为主散射纹纹理. 这也能够通过 mainTextureOffset 属性访问。
"_BumpMap"为法线贴图。
"_LightMap"为光照贴图。
"_Cube"为发射立方体贴图。
参见：mainTextureScale 属性，SetTextureScale.
◆ function GetVector(propertyName: string): Vector4
描述：获取一个命名向量的值。
在 Unity shader 中四组件向量和颜色是相同的。GetVector does exactly the same as GetColor just the input data type is different(xyzw in the vector becomes rgba in the color).
See Also: GetColor, SetVector.
◆ function HasProperty(propertyName: string): bool
描述：检查材质的 shader 是否有给定名称的属性。
参见：mainTextureScale 属性，SetTextureScale.
◆ function Lerp(Start: Material, end: Material, t: float): void
描述：在两个材质间插值属性。
使一个材质的所有颜色和浮点值从 start 到 end 基于 t 来插值。
当 t 为 0，所有的值为 start。
当 t 为 1，所有的值为 end。
通常你想要插值的两个材质是相同的（使用相同的着色器和纹理）除了颜色和浮点值。
然后你使用 Lerp 来混合它们。
//混合两个材质
var material1: Material;
var material2: Material;
var duration=2.0;

```
function Start()
{
//首先使用第一个材质
renderer.material=material[];
}
function Update()
{
//随着时间来回变化材质
var lerp=Mathf.PingPong(Time.time, duration)/duration;
renderer.material.Lerp(material1, materail2, lerp);
}
```

参见：Material.

◆ function SetColor(propertyName: string, color: Color): void

描述：设置一个命名的颜色值。

数多 shader 使用超过一个颜色。使用 SetColor 来获取 propertyName 颜色.

Unity 内置着色器使用的普通颜色名称；

"_Color"为材质的主颜色. 这也能够通过 color 属性访问.

"_SpecColor"为材质的反射颜色（在 specular/glossy/vertexlit 着色器中使用）.

"_Emission"为材质的散射颜色（用在 vertexlit 着色器中）.

"_ReflectColor"为材质的反射颜色（用在 reflective 着色器中）.

```
function Start(){
//设置 Glossy 着色器这样可以使用反射颜色
renderer.material.shader=Shader.Find("Glossy");
//设置红色的高光
renderer.material.SetColor("_SpecColor", Color.red);
}
```

参见：color 属性，GetColor.

◆ function SetFloat(propertyName: string, value: float): void

描述：设置一个命名的浮点值。

```
function Start(){
//在这个材质上使用 Glossy 着色器
renderer.material.shader=Shader.Find("Glossy");
}
function Start(){
//动画 Shininess 值
var shininess=Mathf.PingPong(Time.time, 1.0);
renderer.material.SetFloat("_Shininess, shininess);
}
```

参见：GetFloat, Materials, ShderLab documentation.

◆ function SetMatrix(propertyName: string, matrix: Matrix4x4): void

描述：为一个 shader 设置一个命名矩阵。

这个最常用于自定义的 shader，其中需要额外的矩阵参数，矩阵参数不需要在材质检视面板中公开，但是能够在脚本中通过 SetMatrix 和 GetMatrix 来设置和查询。

```
var rotateSpeed=30;
```

```
var texture: Texture;
function Start(){
//用于一个着色器创建一个新的材质
//这个着色器旋转纹理
var m=new Material
 (
"Shader\"Rotation Texture\"{"+
"Properties{_Main Tex(\"Base",2D)=\"white"{}}"+
"SubShader{"+
"Pass{"+
"Material{Diffuse(1,1,0)Ambient(1,1,1,0)}"+
"Lighting On"+
"SetTexture[_MainTex]{"+
"matrix[_Rotation]"+
"combing texture*primary double.texture"+
"{"+
"}"+
"{"+
"}"
);
m.mainTexture=texture;
renderer.material=m;
}
function Update()
}
//为这个着色器构建一个旋转矩阵并设置它
var rot=Quaternion, Euler(0,0,Time.time*rotateSpeed);
var m=Matrix4x4.TRS(Vector3.zero,rot.Vector3(1,1,1));
renderer.material.SetMatrix(*_Rotation", m);
}
```

参见：GetMatrix，Materials，ShaderLab documentation.

◆ function SetPass(pass: int): bool

描述：为渲染激活给定的 pass.

传递从零开始最大到 passCount（但不包含）的索引。

这个最常用在使用 GL 类直接绘制的代码中（只能 Unity Pro）。例如，Image Effects 使用

材质来实现屏幕后期处理，对材质中的每一个 pass 它们激活并绘制一个全屏四边形。

如果 SetPass 返回假，你不应该渲染任何东西。

这里是一个全屏图像效果的例子，它反转颜色，添加这个脚本到相机并在播放模式中查看。

```
private var mat: Material;
function Start()
{
mat=new Material(
```

```
"Shader\"Hidden/Invert\"{"+
"SubShader{"+
"Pass{"+
"ZTest Always Cull Off ZWrite Off"+
"SetTexture[_RenderTex]{combine one-texture}"+
"}"+
"}"+
"}"
);
}
function OnRenderImage(source: RenderTexrure, dest: RenderTexture){
RenderTexture.active=dest;
source.SetGlobalShaderProperty("_RenderTex");
GL.PushMatrix();
GL.LoadOrtho();
//激活第一个 pass（这里我们知道它只有仅有的 pass）
mat.SetPass(0);
//绘制一个四边形
GL..Begin(GL.QUADS);
GL..TexCoord2(0,0);GL..Vertex3(0,0,0.1);

GL..TexCoord2(1,0);GL..Vertex3(1,0,0.1);
GL..TexCoord2(1,1);GL..Vertex3(1,1,0.1);
GL..TexCoord2(0,1);GL..Vertex3(0,1,0.1);
GL..End();

GL.PopMatrix();
}
```

参见：passCount 属性，GL 类，ShaderLab documentation.

◆ function SetTexture(propertyName: string, texture: Texture): void

描述：设置一个命名纹理.

数多 shader 使用超过一个纹理。使用 SetTexture 来改变 propertyName 纹理。

Unity 内置着色器使用的普通纹理名称：

"_MainTex"为主射散纹理，这也能够通过 mainTexture 属性访问.

"_BumapMap"为法线贴图.

"_LightMap"为光照贴图.

"_Cub"为放射立方体贴图.

```
//基于实际滚动主纹理
var scrollSpeed=0.5;
function Update(){
var offset=Time.time*scrollSpeed;
rendereer.material.SetTextureOffset("_MatrixTex", Vector2(offset,0));
}
```

参见：mainTextureOffset 属性，GetTextureOffset.

◆ function SetTextureScale(propertyName: string, scale: Vector2): void
描述：设置纹理 propertyName 的位置缩放.
Unity 内置着色器使用的普通纹理名称：
"_MainTex"为主射散纹理，这也能够通过 mainTexture 属性访问.
"_BumapMap"为法线贴图.
"_LightMap"为光照贴图.
"_Cub"为放射立方体贴图.
参见：mainTextureScale 属性，GetTextureScale.

```
function Update()
{
//以一个流行的放式动画主纹理缩放！
var scaleX=Mathf.Cos(Time.time)*0.5+1;
var scaleY=Mathf.Sin(Time.time)*0.5+1;
rendereer.material.SetTextureScale("_MainTex", Vector2(ScaleX,ScaleY));
}
```

◆ function SetVector(propertyName: string, Vector: Vector4): void
描述：设置一个命名的向量值.
在 Unity shader 中四组件向量和颜色是相同的。SetVector 与 SetColor 完全相同，仅仅是输入数据类型不同（向量的 xyzw 变为颜色的 rgba）.
参见：SetColor, GetVector.
继承的成员
继承的变量
name 对象的名称。
hideFlags 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，组件或资源。
DestroyImmediate 立即销毁物体 obj。强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较两个物体是否不相同。
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁。
Mesh
类，继承自 Object
一个类允许你从脚本中创建或修改网格。
网格包含顶点和多个三角形数组。参考 Procedural example project 获取石油网格接口的例子.
三角数组只是顶点索引数组；三个索引为一个三角形。
对于每个顶点可以有一个法线，两个纹理坐标，颜色和切线。这些都是可选的并可以去掉。所有的顶点信息被存储相同尺寸的不同数组中，因此如果的网格有 10 个顶点，

你

应该有大小为 10 的数组用于法线和其他属性。

可能有三件事，你可能想要使用可调整的网格。

1. 从头开始构建网格：应该总是按照如下的顺序：1）赋值 vertices2）赋值 triangles

```
function start(){
var mesh=new Mesh();
GetComponent(MeshFilter).mesh=mesh;
mesh.vertices=newVertices;
nesh.uv=newUV;
mesh.triangles=newTriangles;
}
```

2. 没帧修改顶点属性：1）获取顶点，2）修改它们，3）将它们赋值回网格.

```
function Update(){
var mesh: mesh=GetComponent(MeshFilter).mesh;
var vertices=newVertices;
var normals=mesh.normals;
for(var i=0,i<vertices.length,i++)
{
vertices[i]+=mormals{i}*Mathf.Sin(Time.time);
}
mesh.vertices=vertices;
}
```

3. 连续地改变网格三角形和顶点：1）调用 Clear 开始刷心，2）赋值顶点和其他属性，3）赋

值三角形索引.

在赋值新的顶点或三角形时调用 Clear 是重要的。Unity 总是检查提供的三角形索引，

它们是否没有超出顶点边界。电影 Clear 然后赋值顶点然后三角形，确保没你没有超出数据边

界。

```
function Update(){
var mesh: mesh=GetComponent(MeshFilter).mesh;
mesh.Clear();
mesh.vertices=newVertices;
nesh.uv=newUV;
mesh.triangles=newTriangles;
}
```

变量

◆ var bindposes: Matrix4x4[]

描述：绑定的姿势。每个索引的绑定姿势使用具有相同的索引的骨骼。

当骨骼在绑定姿势中时，绑定姿势是骨骼变换矩阵的逆。

```
function Start(){
gameObject.AddComponent(Animation);
gameObject.AddComponent(SkinnedMeshRenderer);
var renderer: SkinnedMeshRenderer=GetComponent(SkinnedMeshRenderer);
```

```
//构建基本网格
var mesh: Mesh-new Mesh();
mesh.vertices=[Vector3(-1,0,0), Vector3(1,0,0), Vector3(-1,5,0), Vector3(1,5,0)];
mesh.uv=[Vector2(0,0), Vector2(1,0), Vector2(0,1), Vector2(1,1)];
mesh.triangles=[0,1,2,1,3,2];
mesh.RecalculateNormals();
//赋网格到网格过滤器和渲染器
renderer.material=new Material(Shader.Find("Diffuse"));
//赋骨骼权值到网格
//使用两个骨骼. 一个用于上部的顶点，一个用于下部的顶点
var weights=new BoneWeight[4];
weights[0].boneIndex0=0;
weights[0].weight0=1;
weights[1].boneIndex0=0;
weights[1].weight0=1;
weights[2].boneIndex0=1;
weights[2].weight0=1;
weights[3].boneIndex0=1;
weights[3].weight0=1;
mesh.boneWeights=weights;
//创建骨骼变换并绑定姿势
//一个骨骼在顶部一个在底部
var bones=new Transform[2];
var bindPoses=new Matrx4x4[2];
bones[0]=new GameObject("Lower").transform;
bones[0]parent=transform;
//设置相对于父的位置
bones[0].localRotation=Quaternion.identity;
bones[0].localPosition=Vector3.zero;
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵市相对与根的
//这样我们就能够随意移动根物体了
bindPose[0]=bones[0].worldToLocalMatrix*transform.localToWorldMatrix;
bones[1]=new GameObject("Upper").transform;
bones[1]parent=transform;
//设置相对于父的位置
bones[1].localRotation=Quaternion.identity;
bones[1].localPosition=Vector3.(0,5,0);
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵市相对与根的
//这样我们就能够随意移动根物体了
bindPose[1]=bones[1].worldToLocalMatrix*transform.localToWorldMatrix;
.mesh.bindposes=bindPoses;
.//赋值骨骼并绑定姿势
```

```
.renderer.bones=bones;
.renderer.sharedMesh=mesh;
//赋值一个简单的挥动动画到底部的骨骼
var curve=new AnimationCurve();
curve.keys=[new Keyframe(0,0,0,0:,new Keyframe(1,3,0,0),new Keyframe(2,0.0,0,0,)];
//使用曲线创建剪辑
var clip=new AnimationClip();
clip.SetCurve("Lower", Transform,"m_LocalPosition.z", curve);
//添加并播放剪辑
animation.AddClip(clip, "test");
animation.Play("test"); }
```

◆ var boneWeights: BondWeight[]

描述：每个顶点的骨骼权重

数组的大小与 vertexCount 相同或为空。

每个顶点可以被至多 4 个不同骨骼影响。4 个骨骼的权值加和应该为 1，

```
function Start(){
gameObject.AddComponent(Animation);
gameObject.AddComponent(SkinnedMeshRenderer);
var renderer: SkinnedMeshRenderer=GetComponent(SkinnedMeshRenderer);
//构建基本网格
var mesh: Mesh-new Mesh();
mesh.vertices=[Vector3(-1,0,0), Vector3(1,0,0), Vector3(-1,5,0), Vector3(1,5,0)];
mesh.uv=[Vector2(0,0), Vector2(1,0), Vector2(0,1), Vector2(1,1)];
mesh.triangles=[0,1,2,1,3,2];
mesh.RecalculateNormals();
//赋网格到网格过滤器和渲染器
renderer.material=new Material(Shader.Find("Diffuse"));
//赋骨骼权值到网格
//使用两个骨骼. 一个用于上部的顶点，一个用于下部的顶点
var weights=new BoneWeight[4];
weights[0].boneIndex0=0;
weights[0].weight0=1;
weights[1].boneIndex0=0;
weights[1].weight0=1;
weights[2].boneIndex0=1;
weights[2].weight0=1;
weights[3].boneIndex0=1;
weights[3].weight0=1;
mesh.boneWeights=weights;
//创建骨骼变换并绑定姿势
//一个骨骼在顶部一个在底部
var bones=new Transform[2];
var bindPoses=new Matrx4x4[2];
bones[0]=new GameObject("Lower").transform;
```

```
bones[0]parent=transform;
//设置相对于父的位置
bones[0].localRotation=Quaternion.identity;
bones[0].localPosition=Vector3.zero;
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵市相对与根的
//这样我们就能够随意移动根物体了
bindPose[0]=bones[0].worldToLocalMatrix*transform.localToWorldMatrix;
bones[1]=new GameObject("Upper").transform;
bones[1]parent=transform;
//设置相对于父的位置
bones[1].localRotation=Quaternion.identity;
bones[1].localPosition=Vector3.(0,5,0);
//绑定姿势是骨骼的逆变换矩阵
//在这种情况下我们也要使这个矩阵市相对与根的
//这样我们就能够随意移动根物体了
bindPose[1]=bones[1].worldToLocalMatrix*transform.localToWorldMatrix;
mesh.bindposes=bindPoses;
//赋值骨骼并绑定姿势
renderer.bones=bones;
renderer.sharedMesh=mesh;
//赋值一个简单的挥动动画到底部的骨骼
var curve=new AnimationCurve();
curve.keys=[new Keyframe(0,0,0,0:,new Keyframe(1,3,0,0),new Keyframe(2,0.0,0,0,)];
//使用曲线创建剪辑
var clip=new AnimationClip();
clip.SetCurve("Lower", Transform,"m_LocalPosition.z", curve);
//添加并播放剪辑
animation.AddClip(clip, "test");
animation.Play("test");
}
```

◆ **var bounds: Bounds**

描述：网格的包围体。

这个是在网格的局部坐标空间中轴对齐的包围盒（不会受到变换的影响）参考世界空间中的 **Renderer.Bounds** 属性。

```
//产生一个屏幕 UV 坐标，这个与网格尺寸无关
//通过缩放包围盒尺寸的顶点
function Start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
var.vertices=mesh vertices;
var.uv=new Vector2[vertices.length];
var.bounds=mesh.bounds;

for (var i=0,i<uvs.length;i++){
```

```
uvs[i]=Vector2(vertices[i].x/bounds.size.x,vertices[i].z/bounds.size.x);
}
mesh.uv=uvs;
}
```

参见：Bounds 类，Renderer.Bounds 属性.

◆ var Colors: Color[]

描述：返回网格的顶点颜色。

如果没有顶点颜色可用，一个空的数组将被返回。

```
//设置 y=0 的顶点位红色，y=1 的订单为绿色.
//（注意大多数设置着色器不显示顶点颜色，你可以
//使用例如，一个粒子渲染器来查看顶点颜色）
function Start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
var.vertices=mesh vertices;
var.color=new Color[Vertices.Length];
for (var i=0,i<vertices.length;i++){
colors[i]=Color.Lerp(Colored, Color.green, vertices[i].y);
}
mesh.colors=colors;
}
```

◆ var normals: Vectors[]

描述：网格的法线。

如果网格布包含发行，一个空的数组将被返回。

```
//以 speed 每帧旋转法线
var speed=100.0;
function Update(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
var.normals=mesh.normals;
var rotation=Quaternion.AngleAxis(Time.deltaTime*speeed, Vector3.up);
for (var i=0,i<normals.length;i++){
normals[i]=rotation*normals[i];
}
mesh.normals=normals;
}
```

◆ var subMeshCount: int

描述：子网格数。每个材质有一个不同的三角形列表。

◆ var tangents: Vector4[]

描述：网格的切线。

切线主要用于 bumpmap shader 中。切线是一个单位长度向量，a 沿着网格表面指向水平

(U)纹理方向。Unity 中的切线是由 Vector4 表示的，x.y.z 组件定义向量，如果需要 w 用来翻

转副法线。

Unity 通过计算向量和切线之间的叉乘来计算表面的向量（副法线），并乘以

tangnent.w.因此 w 应该是 1 或-1。

如果你想在网格上使用凹凸贴图着色器，你需要自己计算切线。赋值 normals 或使用 RecalculateNormals 之后计算切线。

◆ var tirangles: int[]

描述：一个数组包含网格中所有的三角形。

这个数组时包含顶点数组索引的三角形列表。三角形数组的大小总是 3 的倍数。顶点可以通过简单地索引同一顶点来共享。如果网格包含多个子网格（材质），三角形列表将包含所有子网格的所有三角形。建议赋值顶点数组之后赋值一个三角形数组，以避免越界错误。

```
//构建一个网格，这个网格包含一个带有 uv 的三角形.
function start(){
gameObject.AddComponent("MeshFilter);
gameObject.AddComponent("MeshRenderer);
var.mesh: Mesh=GetComponent(MeshFilter).mesh;
mesh.Clear();
mesh.vertices=[Vector3(0,0,0), Vector3(0,1,0), Vector3(1,1,0)];
mesh.uv=[Vector2(0,0), Vector2(0,1), Vector2(1,1)];
mesh.triangles=[0,1,2];
}
```

◆ var uv: Vector2[]

描述：网格的基本纹理坐标。

```
//产生一个平面 uv 坐标
function start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
var vertices=mesh.vertices;
var uvs=new Vector2[vertices.lenghth];
fpr(var i=0;i<uvs.Length,i++){
uvs[i]=Vector2(vertices[i].x, vertices[i].z);
}
mesh.uv=uvs;
}
```

◆ var uv2: Vector2[]

描述：网格的第二个纹理坐标集，如果提供。

```
//为第二个 uv 集产生一个平面 uv 坐标
function start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
var vertices=mesh.vertices;
var uvs=new Vector2[vertices.lenghth];
fpr(var i=0;i<uvs.Length,i++){
uvs[i]=Vector2(vertices[i].x, vertices[i].z);
}
```

```
mesh.uv2=uvs;
}
```

◆ **var vertexCount: int**

描述：网格顶点数（只读）

```
function start(){
var mesh=GetComponent(MeshFilter).sharedMesh;
print(mesh.vertexCount);
}
```

◆ **var vertices: Vector3[]int**

描述：返回一个顶点位置的拷贝或赋值一个新的顶点位置数组。

网格的顶点数可，以通过赋值一个不同数量的顶点数组来改变。注意，如果你调整了顶点数组，那么所有其他顶点属性（法线，颜色，切线，UV）将被自动地调整大小。设置顶点时，如果没有顶点被赋值到这个网格那么 RecalculateBounds 将自动被调用。

```
function start(){
var mesh=GetComponent(MeshFilter).sharedMesh;
var vertics=mesh vertices;

fpr(var i=0;i<vertices.Length,i++)
{
vertices[i]+=Vector3.up*Time.deltaTime;
}
mesh.vertices=vertices;
mesh.RecalculateBounds();
}
```

构造函数

◆ **static function Mesh(): Mesh**

描述：创建一个空的网格

```
//创建一个新的网格并将它赋给网格过滤器
function Start(){
var mesh=new Mesh();
GetComponent(MeshFilter).mesh=mesh;
}
```

函数

◆ **function Clear(): void**

描述：清除所有的顶点数据和所有的三角形索引。

你应该在重建 triangles 数组之间调用这个函数。

◆ **function GetTriangles(submesh: int): int[]**

描述：返回子网格的三角形列表。

一个子网格仅仅是一个独立的三角形列表。当网格渲染器使用多个材质时，你应该确保有尽可能多的子网格作为材质。

◆ **function Optimize(): void**

描述：优化网格以便显示。

这个操作将花费一点时间但是会使几何体显示的更快。例如，它从三角形中产生三角形

带。如果你从头创建一个网格并且想在运行的时候取得更好的运行时性能而不是较高的加

载时间，你应该使用它。它三角带化你的模型为每个顶点缓存位置优化三角形，对于导

载时间，你应该使用它。它三角带化你的模型并为每个顶点缓存位置优化三角形。对于导

入的模型你不应该调用这个，因为导入流水线已经为你做了。

```
function Start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
mesh.Optimize();
}
```

◆ **function RecalculateBounds(): void**

描述：从顶点重新计算网格的包围提。

修改顶点之后你应该调用这个函数以确保包围体式正确的。赋值三角形将自动计算这个包围体。

```
function Start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh=mesh;
mesh.RecalculateBounds();
}
```

◆ **function RecalculateNormals(): void**

描述：从三角形和顶点重新计算网格的法线。

在调整顶点之后通常需要更新法线以便反映这个改变。法线是从所有共享的顶点来计算的。导入的网格有时并不共享所有顶点。例如在 **UV** 法线接缝处的点将被分割成两个点。因

而 **RecalculateNormals** 函数在 **uv** 接缝处将创建不平滑的法线，**RecalculateNormals** 不会自动

产生切线，因此 **bumpmap** 着色器在调用 **RecalculateNormals** 之后不会工作。然而你可以提

取你自己的切线。

```
function Start(){
var mesh: Mesh=GetComponent(MeshFilter).mesh;
mesh.RecalculateNormals();
}
```

◆ **function SetTriangles(triangles: int[], submesh: int): void**

描述：为子网格设置多边形列表

一个子网格仅仅是一个独立的三角形列表。当网格渲染器使用多个材质时，你应该确保

有尽可能多的子网格作为材质。建议赋值顶点数组之后赋值一个三角形数组，以避免越界

错误。

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 **id**。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 **obj**，强烈建议使用 **Destroy** 代替。

FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 **target** 不被自动销毁。

PhysicsMaterial

类，继承自 **Object**

载体材质描述：如何处理物体碰撞（摩擦，弹性）

参见：**Collider**

变量

◆ **var bounceCombine: PhysicsMaterialCombine**

描述：决定弹力是如何组合的。

传统上弹性属性依赖于两种相互接触的材质的组合。然而在游戏中这是不切实的。可以

使用组合模式来调整两个材质的弹性如何被组合。

collider.material.bounceCombine=FrictionCombineMode.Average;

◆ **var bouncyness: float**

描述：表面的弹力如何？**0** 值没有弹力。**1** 值没有能力损失的反弹。

collider.bouncyness=1;

◆ **var dynamicFriction: float**

描述：移动时候使用的摩擦力。这个值在 **0** 到 **1** 之间。

0 值就像冰，**1** 像橡胶。

collider.dynamicFriction=1;

◆ **var dynamicFriction2: float**

描述：如果有向摩擦力被启用，**dynamicFriction2** 将沿着 **FrictionDirection2** 使用。

collider.physicMaterial.dynamicFriction2=0;

◆ **var frictionCombine: PhysicMaterialCombine**

描述：决定摩擦力是如何组合的。

传统上摩擦力属性依赖于两种相互接触的材质的组合。然而在游戏中这是不切实的。

你可以使用组合模式来调整两个材质的摩擦力如何被组合。

collider.material.frictionCombine=physicMaterialCombine.Average;

◆ **var frictionDirection2: Vector3**

描述：有向性方向。如果这个矢量是非零，有向摩擦力被启用。

dynamicFriction2 和 **staticFriction2** 将沿着 **frictionDirection2** 被应用。有向性方向相对于碰撞器的局部坐标系统。

//使碰撞向前滑动而不是侧滑
collider.physicMaterial.frictionDirection2=Vector3.forward;
collider.physicMaterial.dynamicFriction2=0;
collider.physicMaterial.dynamicFriction=1;

◆ var staticFriction: float

描述：当一个物体静止在一个表面上时使用的摩擦力。通常是 0 到 1 之间的值。0 值就像冰，1 像橡胶。

collider.staticFriction=1;

◆ var staticFriction2: float

描述：如果有向摩擦力被启用，staticFriction2 将沿着 frictionDirection2 使用。

collider.physicMaterial.staticFriction2=0;

构造函数

◆ static function PhysicMaterial(): PhysicMaterial

描述：创建一个新的材质

通常只使用 collider.material 和直接修改附加的材质更加简单。

//创建一个新的材质并附加它
function Start(){
var material=new PhysicMaterial();
material.dynamicfriction=1;
collider.material=material;
}

◆ static function PhysicMaterial(name: string): PhysicMaterial

描述：创建一个新的材质，命名为 name.

//创建一个新的材质并附加它
function Start(){
var material=new PhysicMaterial("New Material");
material.dynamicfriction=1
collider.material=material;
}

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁
ScriptableObject
类，继承自 Object
如果你想创建一个不需要附加到游戏物体的对象，可以从这个类继承、
这对于那些只存储数据资源是组有用的。
消息传递
◆ function OnDisable(): void
描述：当可编辑物体超出范围时调用这个函数
当物体被销毁的时候这个函数也会被调用并可以用于任何清理的代码。当脚本在编译结束后被加载时，OnDisable 将被调用，然后脚本加载完成后 OnDisable 将被调用。

```
function OnDisable()
{
print(:script was removed");
}
```

OnDisable 不能作为一个 coroutine.
◆ function OnEnable(): void
描述：物体被加载时调用该函数

```
function OnEnable()
{
print(:script was enabled");
}
```

OnEnable 不能作为一个 coroutine.
类方法
◆ static function CreateInstance(className: string): ScriptableObject
描述：使用 className 创建一个可编程物体的实例.
继承的成员
继承的变量
name 对象的名称
hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，缓存或资源。
DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较连个物体是否不相同。
DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁
GUISkin
类，继承自 ScriptableObject
变量

◆ **var box: GUIStyle**

描述：用于 **GUI.Box** 控件的缺省风格

◆ **var button: GUIStyle**

描述：用于 **GUI.Button** 控件的缺省风格

◆ **var GUIStyle[i]:**

描述：

◆ **var font: Font**

描述：用于所有风格的缺省字体。

◆ **var horizontalScrollbar: GUIStyle**

描述：

◆ **var horizontalScrollbarLeftButton: GUIStyle**

描述：

◆ **var horizontalScrollbarRightButton: GUIStyle**

描述：

◆ **var horizontalScrollbarThumb: GUIStyle**

描述：

◆ **var horizontalSlider: GUIStyle**

描述：用于 **GUI.HorizontalSlider** 控件背景部分的缺省风格.

用于决定滑块可拖动区域尺寸的填充属性。

◆ **var horizontalSliderThumb: GUIStyle**

描述：用于 **GUI.HorizontalSlider** 控件中可拖动滑块的缺省风格.

用于决定滑块尺寸的填充属性。

◆ **var label: GUIStyle**

描述：用于 **GUI.Label** 控件的缺省风格.

◆ **var scrollView: GUIStyle**

描述：

◆ **var settings: GUISettings**

描述：使用这个皮肤的空间如何表现得通用设置。

◆ **var textArea: GUIStyle**

描述：用于 **GUI.TextArea** 控件的缺省风格。

◆ **var textField: GUIStyle**

描述：用于 **GUI.textField** 控件的缺省风格。

◆ **var toggle: GUIStyle**

描述：用于 **GUI.toggle** 控件的缺省风格。

◆ **var verticalScrollbar: GUIStyle**

描述：

◆ **var verticalScrollbarDownButton: GUIStyle**

描述：

◆ **var verticalScrollbarThumb: GUIStyle**

描述：

◆ **var verticalScrollbarUpbutton: GUIStyle**

描述：

◆ **var verticalSlider: GUIStyle**

描述：用于 **GUI.VerticalSlider** 控件背景部分的缺省风格.

用于决定滑块可拖动区域尺寸的填充属性。

◆ **var verticalSliderThumb: GUIStyle**

描述：用于 **GUI.VerticalSlider** 控件中可拖动滑块的缺省风格.

用于决定滑块尺寸的填充属性。

◆ **var window: GUIStyle**

描述：用于 **GUI.Windows** 控件的缺省风格。

函数

◆ **function FindStyle(styleName: string): GUIStyle**

描述：获取一个命名 **GUIStyle.**

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 **id**。

继承的消息传递

OnEnable 物体被加载时调用该函数

OnDisable 可用编程物体超出范围时调用这个函数

继承的类函数

CreateInstance 使用 **className** 创建一个可编程物体的实例。

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 **obj**，强烈建议使用 **Destroy** 代替。

FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 **target** 不被自动销毁。

Shader

类，继承自 **Object**

用于所以渲染的着色器脚本

大多数高级的渲染都是通过 **Material** 类控制的. **Shader** 类类最常用于检查一个着色器时否能够运行在用户的硬件上（**isSupported** 属性）并根据名称找到着色器（**Find** 方法）.

参见：**Material** 类，**Materials**，**ShaderLab documentation.**

变量

◆ **var isSupported: bool**

描述：这个着色器能够运行在端用户的显卡上？（只读）

如果这个着色器重的设置和任何 **fallback** 函数被支持，返回真。在实现特定的效果时，最常使用这个。例如，**Unity Pro** 中的 **image effects**，如果这个着色器不被支持那么 **Unity** 将

自动禁用它们。

```
//如果材质的着色器不被支持，禁用渲染器
if(!renderer material.shader.isSupported)
```

renderer.enabled=false;

参见：Material 类，ShaderLab documentation.

◆ var maximumLOD: int

描述：该 shader 的 LOD 等级

参见：Shader Level of Detail, Shder globalMaximumLOD.

◆ var renderQueue: int

描述：这个 shader 的渲染队列（只读）

参见：Material.renderQueue, RenderQueue tag.

类变量

◆ static var globalMaximumLOD: int

描述：所有 shader 的 LOD 等级.

参见：Shader Level of Detail, Shader.MaximumLOD.

类方法

◆ static function Find(name: string): Shader

描述：找到名为 name 的着色器。

Shader.Find 能够用来切换到另一个着色器，而不需要保持一个到该着色的引用。name 为材质着色器下拉框中的名称。通常的名称是："Diffuse", "Bumped Diffuse", "VertexLit", "Transparent/Diffuse"等等。

在构建时，只包含那些使用中的 shader 或位置在"Resources"文件夹中 shader。

```
//从脚本中改变 shader
function Start()
{
//切换到透明散射着色器
renderer.material.shader=Shader.Find("Transparent/Diffuse");
}
//从代码创建一个材质
function Start()
//使用透明散射着色器创建一个材质
var material=new Material(Shader.Find("Transparent//Diffuse");
material.color=Color.green;
//赋值这个材质到渲染器
renderer.material=material;
}
```

参见：Material 类。

◆ static function PropertyToID(name: string): int

描述：为一个着色器属性名获取唯一标识。

着色器属性表示被 MaterialPropertyBlock 函数使用。

在 Unity 中着色器属性的每个名称都（例如，_MainTex 或_Color）被赋予一个唯一的整型

数，在整个游戏中都不变。

参见：MaterialPropertyBlock.

◆ static function SetGlobalColor(propertyName: string, color: Color): void

描述：为所以着色器设置全局颜色属性。

如果一个着色器需要而材质没有定义它们将使用全局属性（例如，如果着色器不在

Properties 模块中公开它们）.

通常在你有一组定义的着色器并使用相同的"全局"颜色（例如，太阳的颜色）。然后你可以从脚本中设置全局属性，并不需要在所有的材质中设置相同的颜色。

参见：SetGlobalFloat，SetGlobalVector，SetGlobalTexture；Material 类，ShaderLab documentation.

◆ static function SetGlobalFloat(propertyName: string, value: float): void

描述：为所有着色器设置全局浮点数属性。

如果一个着色器需要而材质没有定义它们将使用全局属性（例如，如果着色器不在 Properties 模块中公开它们）.

通常在你有一组定义的着色器并使用相同的"全局"浮点 z 数（例如，自定义雾类型的密度）。然后你可以从脚本中设置全局属性，并不需要在所有的材质中设置相同的浮点数。

参见：SetGlobalColor，SetGlobalTexture；Material 类，ShaderLab documentation.

◆ static function SetGlobalMatrix(propertyName: string, mat: Matrix4x4): void

描述：为所有着色器设置全局矩阵属性。

如果一个着色器需要而材质没有定义它们将使用全局属性（例如，如果着色器不在 Properties 模块中公开它们）.

参见：SetGlobalColor，SetGlobalFloat；Material 类，ShaderLab documentation.

◆ static function SetGlobalTexture(propertyName: string, tex: Texture): void

描述：为所有的着色器设置全局纹理属性。

如果一个着色器需要而材质没有定义它们将使用全局属性（例如，如果着色器不在 Properties 模块中公开它们）.

通常在你有一组定义的着色器并使用相同的"全局"纹理（例如，自定义散射光照立方贴图）。然后你可以从脚本中设置全局属性，并不需要在所有的材质中设置相同的纹理。

参见：SetGlobalColor，SetGlobalFloat；Material 类，ShaderLab documentation.

◆ static function SetGlobalVector(propertyName: string, vec: Vector4): void

描述：为所有着色器设置全局向量属性。

如果一个着色器需要而材质没有定义它们将使用全局属性（例如，如果着色器不在 Properties 模块中公开它们）.

通常在你有一组定义的着色器并使用相同的"全局"向量（例如，风的方向）。然后你可以从脚本中设置全局属性，并不需要在所有的材质中设置相同的向量。

参见：SetGlobalFloat，SetGlobalColor，SetGlobalTexture；Material 类，ShaderLab documentation.

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较连个物体是否不相同。
DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。
TerrainData
类，继承自 Object
TerrainData 类存储高度图，细节网格位置，树实例，和地形纹理 alph 图，
Terrain 组件链接地形数据并渲染它。
变量
◆ var heightmapHeight: int
描述：采样的地形高度（只读）
◆ var heightmapWidth: int
描述：采样的地形宽度（只读）
◆ var size: Vector3
描述：地形在世界单位下的总大小
函数
◆ function GetHeights(xBase: int, yBase: int, width: int, height: int): float[,]
描述：获取高度图采样的一个数组。
◆ function GetInterpolatedNormal(x: float, y: float): Vector3
描述：在一个给定的位置获取插值法线。
/x/和 y 坐标被作为 0...1 之间正规化的坐标被指定。
◆ function SetHeights(xBase: int, yBase: int, height: float): float[,]): void
描述：设置高度图采样的一个数组。
继承的成员
继承的变量
name 对象的名称
hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，缓存或资源。
DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较连个物体是否不相同。
DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。
TextAsset
类，继承自 Object
文本文件资源。
你可以在你的工程中使用原始的.txt 文件作为资源，并通过这个类获取它们的内容。

变量
描述：文本资源的原始字节
//通过添加.txt 扩展名到文件来加载一个.jpg 或.png 文件
//并拖动它到 imageTextAsset
```
var image TextAsset: TextAsset
function Start(){
var tex=new Texture2D(4,4);
tex.LoadImage(imageTextAsset.bytes);
renderer.material.mainTexture=tex;
}
```
◆ var text: string
描述：.txt 文件的文本内容作为一个字符串。
```
var asset: TextAsset;
function Start()
{
print(asset.text);
}
```
继承的成员
继承的变量
name 对象的名称
hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？
继承的函数
GetInstanceID 返回该物体的实例 id。
继承的类函数
operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，缓存或资源。
DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator != 比较连个物体是否不相同。
DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。
Texture
类，继承自 Object
用于处理纹理的基类，包含的功能被 Texture2D 和 RenderTexture 类共用。
变量
◆ var anisoLevel: int
描述：纹理的各向异性过滤等级
反走样过滤使纹理从一个较小的视角看时具有较好的效果，但是会带来显卡性能上的开
值。通常你可以将它用与地面，地板或路面纹理以使它看起来更好。参见：texture assets.
```
renderer.material.mainTexture.anisoLevel=2;
```
◆ var filterMode: FilterMode

描述：纹理的过滤模式

renderer.material.mainTexture.filterMode=FilterMode.trilinear;

参见：FilterMode, texture assets.

◆ var height: int

描述：纹理的像素高度（只读）

//打印纹理尺寸到控制台

var texture: Texture;

function Start(){

print("Size is"+texture.width+"by"+texture.height);

}

◆ var mipMapBias: float

描述：纹理的 mipMap 偏移。

一个正的偏移使纹理显得非常模糊，而一个负的偏移使纹理变得更加清晰。注意使用大的负值会降低性能，因此不建议使用小于 0.5 的偏移。在大多数情况先，纹理的锐化可

以通过使用反走样过滤来实现。

参见：texture.anisoLevel, texture assets.

renderer.material.mainTexture.mipMaoBias=0.5;

◆ var width: int

描述：纹理的像素宽度（只读）

//打印纹理尺寸到控制台

var texture: Texture;

function Start(){

print("Size is"+texture.width+"by"+texture.height);

}

◆ var wrapMode: TextureWrapMode

描述：纹理的包裹模式（Repeat 或 Clamp）

使用 TextureWrapMode.Clamp 在边界上设置纹理剪裁以避免包裹的不真实，或者用 TextureWrapMode.Repeat 平铺纹理。参见：TextureWrapMode, texture assets.

renderer.material.mainTexture.WrapMode=TextureWrapMode.Clamp;

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。

Cubemap

类，继承自 Texture

处理立方贴图的类，用这个来创建或修改已有的 cube map assets.

变量

◆ var format: TextureFormat

描述：纹理中像素数据的格式（只读）

使用这个确定纹理的格式。

构造函数

◆ static function Cubemap(size: int. format: TextureFormat, mipmap: bool): Cubemap

描述：创建新的空立方贴图纹理；

在每个面，纹理将是 size 大小的并且有或没有 mipmap.

通常你会想在创建它之后设置纹理的颜色，使用 SetPixel 和 Apply 函数.

```
function Start(){
//创建一个新的纹理并将它复制给渲染器材质
var texture=new Cubemap(128, TextureFormat.ARGB32, false)
renderer.material.mainTexture=Texture;
}
```

参见：SetPixel，Apply 函数.

函数

◆ function Apply(updateMipmaps: bool=true): void

描述：应用所有面前的 SetPixel 改变.

如果 updateMipMaps 为 true.mip 等级也被重新计算.这是非常耗时的操作，因此你要在 Apply 调用之间改变尽可能多的像素。参见：SetPixel 函数.

◆ function GetPixel(face: CubemapFace, x: int, y: int): Color

描述：返回坐标（face, X, Y）处的像素颜色.

如果像素坐标超出边界（大于宽/高或小于 0），它将基于纹理的包裹模式来限制或重复。

该函数只工作再 ARGB32,RGB24 和 Alpha8 纹理格式上。对于其他格式，他总是返回不透的白色。

◆ function GetPixels(face: CubemapFace, miplevel: int): Color[]

描述：返回立方贴图一个面的像素颜色.

这个函数返回立方贴图面上整个 mip 等级的像素颜色数组。

返回的数组被设置在 2D 数组中，这里，像素被从左到右，从上到下放置（行序）数组的大小是所使用的 mip 等级的宽乘高。默认的 mip 等级是零（基本纹理）在这种情况下

大小仅为纹理的大小。一般地，mip 等级尺寸是 mipSize=max(1,width>>miplevel)高度类似。

该函数只工作在 ARGB32,RGB24 和 Alpha8 纹理格式上。对于其他格式，GetPixels 被忽略.

使用 GetPixels 比重复调用 GetPixel 更快，尤其是对于大纹理，此外 GetPixels 可以访问单独的 mipmap 等级.

参见：SetPixels，mipmapCount.

◆ function SetPixel(face: CubemapFace, x: int, y: int, color: Color): void

描述：在坐标（face,x,y）处设置像素颜色。

调用 Apply 来实际上载改变后的像素到显卡，上载是非常耗时的操作，因此你要在 Apply 调用之间改变尽可能多的像素。

该函数只工作再 ARGB32,RGB24 和 Alpha8 纹理格式上。对于其他格式 SetPixels 被忽略. 参见：Apply 函数.

◆ function SetPixels(color: Color[], face: CubemapFace, mipmapFace, miplevel: int): void

描述：设置立方贴图一个面的像素颜色。

这个函数取回并改变整个立方贴图面的像素颜色数组。调用 Apply 来实际上载改变后的像素到显卡.

colors 数组被放置在 2D 数组中，这里，像素被从左到右，从上到下放置（行序）数组的大小必须至少是所使用的 mip 等级的宽乘高。默认的 mip 等级是零（基本纹理）在这种

情况下大小仅为纹理的大小。一般地，mip 等级尺寸是 mipSize=max(1,width>>miplevel)高

度类似。

该函数只工作在 ARGB32,RGB24 和 Alpha8 纹理格式上。对于其他格式，GetPixels 被忽略.

参见：GetPixel, Apply, mipmapCount.

继承的成员

继承的变量

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。

MovieTexture

类，继承自 Texture

Movie Textures 是可以播放电影的纹理

它们可以用于过场动画电影序列，或者渲染电影到场景中。

变量

◆ var audioClip: AudioClip

描述：返回属于 MovieTexture 的 AudioClip.

注意这是一个特定的 AudioClip 它总是与电影同步播放音频。在编辑器重如果你将电影的 audioClip 附加到一个源上，它将在电影播放的时候自动开始播放，否则你必须收动开

始它，剪辑只能被附加到一个 AudioSource。

◆ var isPlaying: bool

描述：返回电影是否在播放

◆ var isReadyToPlay: bool

描述：如果电影是从网站上下载的，这个返回是够已经下载了足够的数据以便能够不同

版的播放它。

对于不是来自 web 的流的电影，这个值是返回真。

```
function Start(){
www=new WWW(url);
guiTexture.texture=www.movie;
}
function Update(){
if(!guiTexture.texture.isPlaying&&guiTexture.texture.isReadyToPlay)
guiTexture.texture.Play();
}
```

◆ var loop: bool

描述：这个为真时电影循环。

函数

◆ function Pause(): void

描述：暂停播放电影。

```
function Start(){
renderer.material.mainTexture.Pause();
}
```

◆ function Play(): void

描述：开始播放电影。

注意运行的 MovieTexture 将使用大量的 CPU 资源，并且它将持续运行直到它被手、动停止或加载一个新的关卡。参见：stop

```
function Start(){
renderer.material.mainTexture.Play();
}
```

继承的成员

继承的变量

width 纹理的像素宽度（只读）

height 纹理像素高度（只读）

filterMode 纹理的过滤模式

anisoLevel 纹理的各向异性过滤等级

wrapMode 纹理的包裹模式（Repeat 或 Clamp）

mipMapBias 纹理的 mipMap 便宜。

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

描述：渲染纹理的尺寸限制为 2 的幂次？

当创建图像后刷处理效果时，你应该总是设置这个为 false 因为这允许四面

ReaderTexture 大小问任意屏幕大小。

当为普通的材质和 shader 使用 RenderTexture 时，你应该总是设置这个为 true 因为这允

许像普通纹理一样使用一个 RenderTexture.

◆ var useMipMap: bool

描述：生成 mipmap 等级？

当设置为 true，渲染到这个纹理将创建并生成 mipmap 等级面膜人的渲染纹理没有 mipmap.

这个这毙用于 2 的幂次方尺寸的渲染纹理（参考 isPowerOfTwo).

◆ var width: int

描述：渲染纹理的像素宽度。

注意不像 Texture.height 属性，这个是可读写的，设置一个值来改变大小

构造函数

◆ static function ReaderTexture(width: int, height: int, depth: int): ReaderTexture

描述：创建一个新的 RenderTexture 对象。

渲染纹理使用 width x height 尺寸创建，深度缓存为 depth 位（深度可以是 0,16 或 24）

渲染纹理或设置为非 2 的幂次纹理并使用默认的 color format

注意创建一个 RenderTexture 不会立即创建硬件表示。实际的渲染纹理是第一次使用

是创建或当 Create 被手动调用时创建。因此在创建渲染纹理之后，你可以设置额外的变量，如

isPowerOfTwo, format, isCubemap 等等。

参见：isPowerOfTwo 变量, format 变量.

函数

◆ function Create(): bool

描述：实际创建 RenderTexture。

RenderTexture 构造函数实际上并没有创建硬件纹理：默认的纹理第一次创建时被设置为 active，调用 Create 来创建它。如果纹理已经被创建 Create 不做任何事。

参见：Release，isCreated 函数.

◆ function IsCreate(): bool

描述：渲染纹理产生了？

RenderTexture 构造函数实际上并没有创建硬件纹理：默认的纹理第一次创建时被设置为 active，如果用于渲染的的硬件资源被创建了，IsCreate 返回 ture.

参见：Create，Release 函数.

◆ function Release(): void

描述：释放 RenderTexture。

这个函数释放由这个渲染纹理使用的硬件资源，纹理本身并不被销毁，并在使用的时候

被自动再次创建。

参见：Create，IsCreate 函数.

◆ function SetBorderColor(color: Color): void

描述：为这个渲染纹理设置为边框颜色。

如果显卡支持"剪裁到边界"，那么任何超出 0...1UV 范围的纹理采样将返回边界颜色。

◆ function SetGlobalShaderProperty(propertyName: string): void

描述：赋值这个 RenderTexture 为一个名为 propertyName 的全局 shader 属性。

类变量

◆ **static var active: RenderTexture**

描述：激活的渲染纹理。

所有的渲染将进入激活的 **RenderTexture** 如果活动的 **RenderTexture** 未 **null** 所有的东西都

被渲染到主窗口.

当你一个 **RenderTexture** 变为激活，如果它还没有被创建，硬件渲染内容将被自动创建。

类方法

◆ **static function GetTemporary(width: int, height: int, depthBuffer: int, format: RenderTextureFormat=RenderTextureFormat.ARGB32): RenderTexture**

描述：分配一个临时的渲染纹理。

这个函数被优化，用于当你需呀一个快速 **RenderTexture** 来做一些临时计算时，一旦完成使用 **ReleaseTemporary** 释放它，这样，如果需要，另一个调用能够开始重用它。

◆ **static function ReleaseTemporary(temp: RenderTexture): void**

描述：释放一个由 **GetTemporary** 分配的临时纹理。

如果可能，之后调用 **GetTemporary** 将重用前面创建的 **RenderTexture**，如果没有请来临时 **RenderTexture**，几帧后它将被销毁。

继承的成员

继承的变量

width 纹理的像素宽度（只读）

height 纹理像素高度（只读）

filterMode 纹理的过滤模式

anisoLevel 纹理的各向异性过滤等级

wrapMode 纹理的包裹模式（**Repeat** 或 **Clamp**）

mipMapBias 纹理的 **mipMap** 便宜。

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 **id**。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 **original** 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 **obj**，强烈建议使用 **Destroy** 代替。

FindObjectsOfType 返回所有类型为 **type** 的激活物体。

FindObjectsOfType 返回第一个类型为 **type** 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 **target** 不被自动销毁。

Texture2D

类，继承自 **Texture**

用于处理纹理的类。使用这个来创建临时的纹理或修改已经存在的 **texture assets**

变量

◆ **var format: TextureFormat**

描述：纹理中像素数据的格式（只读）

使用这个确定纹理的格式。

◆ var mipmapCount: int

描述：在这个纹理中有多少 Mipmap 等级（只读）

返回值也包含基本等级，因此他总是 1 或更大。

如果你使用 GetPixels 或 SetPixels 来取回或修改不同的 mip 等级时，需要使用 mipmapCount. 例如，你可以改变一个纹理以便每个 mip 等级以不同的颜色修改. 然后再游

戏中你可以看到那个 mip 等级被实际使用了。

参见：GetPixels 函数，SetPixels 函数。

构造函数

◆ static function Texture2D(width: int, height: int): Texture2D

描述：创建新的空纹理；

纹理为 width 乘 height 大小，TextureFormat 为 ARGB32 带有 mipmap.

通常你会想到在创建它之后设置纹理的颜色，使用 SetPixel，SetPixels 和 Apply 函数。

```
function Start(){
//创建一个新的纹理并将它赋给渲染器材质
var texture=new Texture2D(128,128);
renderer.material.mainTexture=texture;
}
```

参见：SetPixel，SetPixels，Apply 函数.

◆ static function Texture2D(width: int, height: int, format: TextureFormat, mipmap: bool):

Texture2D

描述：创建新的空纹理；

纹理为 width 乘 height 大小，具有给定的 format 有或没有 miamap.

通常你会想到在创建它之后设置纹理的颜色，使用 SetPixel，SetPixels 和 Apply 函数。

创建不允许有压缩纹理格式的贴图。

```
function Start(){
//创建一个新的纹理并将它赋给渲染器材质
var texture=new Texture2D(128,128, TextureFofrmat.AGRB32, false);
renderer.material.mainTexture=texture;
}
```

参见：SetPixel，SetPixels，Apply 函数.

函数

◆ function Apply(updateMipmaps: bool=true): void

描述：实际地应用前面的 SetPixel 和 SetPixels 改变.

如果 updateMipmaps 为 true，mipmap 等级也被重新计算，使用基本等级作为源.通常你会想在所有的情况下使用 true，除非你已经使用 SetPixels 修改了 mip 等级。

这是非常耗时的操作，因此你要在 Apply 调用之间改变尽可能多的像素。

```
function Start(){
//创建一个新的纹理并赋值它到渲染器材质
var texture=new Texture2D(128,128);
renderer.material.mainTexture=texture;
```

```
//用 Sierpinski 分形模式填充！
for(y=0; y<texture.height; ++y) {
for(x=0; x<texture.width; ++x) {
var color=(x&y)? Color.white: Color.gray;
texture.SetPixel(x, y, color);
}
}
//应用所有的 SetPixel 调用
texture.Apply();
}
```

参见：SetPixel，SetPixels 函数.

◆ function Compress(highQuality: bool): void

描述：压缩纹理问 DXT 格式。

使用这个来压缩在运行时生成的纹理。压缩后的纹理使用较少的显存并可以更快地被渲染。

压缩之后，如果原始纹理没有 alpha 通道纹理将是 DXT1 格式，如果它有 alpha 通道纹理

将是 DXT5 格式.

传递 true 到 highQuality 将在压缩过程中抖动源纹理，这可以帮助提高压缩质量但是会有一些慢。

如果显卡不支持压缩或者纹理已经是压缩格式，那么 Compress 将不做任何事情。

参见：SetPixels 函数.

◆ function EncodeToPNG(): byte[]

描述：编码这个纹理为 PNG 格式。

返回的字节数组是 PNG"文件"。你可以将它们写在键盘上以便获取 PNG 文件，并通过网格发送它们。

该函数只工作在 ARGB32 和 RGB24 纹理格式上。对于 ARGB32 纹理编码的 PNG 数据将包含 alpha 通道。对于 RGB24 纹理不包含 alpha 通道。PNG 数据将不包含伽马矫正或颜

色配置信息。

```
//存储截屏为 PNG 文件。
import System.1();
//立即截屏
function Start(){
UploadPNG();
}
function UploadPNG(){
//只在渲染完成后读取屏幕缓存
yield WaitForEndOfFrame();
//创建一个屏幕大小的纹理，RGB24 格式
var width=Screen.width;
var height=Screen.height;
var tex=new Texture2D(width, height, TextureFormat.RGB24, false);
//读取屏幕内容到纹理
```

```
tex ReadPixels(Rect(0, 0, width, height), 0, 0);
tex.Apply();
//编码纹理为 PNG 文件
var bytes=tex.EncodeToPNG();
Destroy(tex);
//处于测试目的，也在工程文件夹中写一个文件
//File.WriteAllBytes(Application.dataPath+"/../SavedScreen.png", bytes);
//创建一个 Web 表单
var form=new WWWForm();
form.AddField("frameCount", Time.frameCount.ToString());
form.AddBinaryData("fileUpload",bytes);
//上传到一个 CGI 脚本
var w=WWW("http://localhost/cgi-bin/env.cgi?post",form);
yield w;
if(w.error!=null)
{
print(w.error);
}
else{
print("Finished Uploading Screenshot");
}
}
```

参见：ReadPixels, WaitForEndOfFrame, LoadImage.

◆ function GetPixel(x: int, y: int): Color

描述：返回坐标(x, y)处的像素颜色。

如果像素坐标超出边界（大于宽/高或小于 0），它将给予纹理的包裹模式来限制或重复。如果你正在从纹理中读一个大的像素块，使用 GetPixels 可能会更快，它将返回整个像素颜色块。

该函数只工作在 ARGB32, RGB24 和 Alpha8 纹理格式上。对于其他格式，他总是返回不透的白色。

```
//设置变换的 y 坐标为跟着高度图
var heightmap: Texture2D;
var size=Vector3(100, 10, 100);
function Update()
{
var x: int=transform.position.x/size.x*heightmap.width;
var z: int=transform.position.z/size.z*heightmap.height;
transform.position.y=heightmap.GetPixel(x,z).grayscale*size.y;
}
```

参见：GetPixels, SetPixel, GetPixelBilinear.

◆ function GetPixelBilinear(u: float, v: float): Color

描述：返回正规化坐标(u, y)处的过滤像素颜色。

坐标 u 和 v 从 0.0 到 1.0，就像网格上的 UV 坐标。如果坐标超出边界（大于 1 小于 0）. 它基于纹理的包裹模式来限制或重复。

返回被双线性过滤的像素颜色。

该函数只工作在 ARGB32, RGB24 和 Alpha8 纹理格式上。对于其他格式，他总是返回不透的白色。

参见：GetPixel.

◆ function GetPixels(miplevle: int): Color[]

描述：获取一块像素颜色。

这个函数返回纹理整个 mip 等级的像素颜色数组。

返回的数组被放置在 2D 数组中，这里，像素被从左到右，从上到下放置（行序）数组的大小是所使用的 mip 等级的宽乘高。默认的 mip 等级是零（基本纹理）在这种情况下

大小仅为纹理的大小。一般地，mip 等级尺寸是 mipWidth-max(1,width>>miplecvel)高度类

似。

该函数只工作在 ARGB32, RGB24 和 Alpha8 纹理格式上。对于其他格式 GetPixels 被忽略。

使用 GetPixels 比重复调用 GetPixel 更快，尤其是对于大纹理. 此外，GetPixels 可以访问单独的 mipmap 等级.

参见：GetPixels, mipmapCount.

◆ function GetPixels(x: int, y: int, blockWidth: int, blockHeight: int, miplevel: int): Color[]

描述：获取一块像素颜色。

这个函数是上面 GetPixels 函数的扩展；它不会返回整个 mip 等级而只是返回开始于 x，y

点 blockWidth 乘 blockHeight 的区域。该块必须适合使用的 mip 等级。返回的数组是 blockWidth*blockHeight 尺寸。

◆ function LoadImage(data: byte[]): bool

描述：从一个字节数组中加载一个图片。

这个函数从原始的字节数组中加载一个 JPG 和 PNG 图片。

```
//通过添加.txt 扩展名刀文件来加载一个.jpg 或.png 文件
//并拖动它到 imageTextAsset
var imageTextAsset: TextAsset;
function Start()
{
var tex=new Texture2D(4,4);
tex.LoadImage(imageTextAsset.bytes);
renderer.material.mainTexture=tex;
}
```

参见：EncodeToPNG 函数.

◆ function PackTextures(textures: Textures2D[]. padding: int, maximumAtlasSize: int= 2048): Rect[]

参数

textures 纹理数组被打包到集合汇总。

padding 打包纹理的像素间距。

maximumAtlasSize 调整纹理的最大尺寸。

返回 Rect[]-一个矩形数组，数组包含每个输入纹理的 UV 坐标集合，如果打包失败

为 mull。

描述：打包多个 textures 到一个纹理集中。

这个函数将使用纹理集替换当前纹理。尺寸，格式和纹理是否有 Mipmap 可以在打包后

改变。

生成的纹理集尽可能的大以便适合所有输入的纹理，但是在没有维度上不超过 maximumAtlasSize. 如果输入的纹理不能适合纹理集合的大小，它们将被缩小。

如果所有导入的纹理是 DXT1 压缩格式的，纹理集也有 DXT1 格式. 如果所有输入纹理被以 DXT1 或 DXT5 格式压缩，那么集合将是 DXT5 格式。如果任何输入的纹理是没有压

缩的，那么集合将是 ARGB32 来压缩格式。

如果输入的纹理没有 mipmap，那么集合也将不会有 mipmap。

◆ function ReadPixels(source: Rect, destX: int, destY: int, recalculateMipMaps: bool= true: void

描述：读取屏幕像素到保存的纹理数据中。

这将从当前激活的 ReaderTexture 或试图（由/source/指定）拷贝一个矩形像素区域到由

destX 和 destY 定义的位置上. 两个坐标都是用像素空间-(0,0)为左下角。

如果 recalculateMipMaps 被设置为真，纹理的 Mip 贴图也将被更新。如果 recalculateMipMaps 被设置为假，你必须调用 Apply 重计算它们.

该函数只工作在 ARGB32 和 RGB24 纹理格式上。

参见：EncodeToPNG.

◆ function Resize(width: int, height: int, format: TextureFormat, hasMipMap: bool): bool

描述：调整纹理大小。

改变纹理的尺寸为 width 乘 height，格式为 textureFormat 并有选择地创建 Mip 贴图. 调

整大小后，纹理像素将是未定义的。这个函数非常类似与纹理构造器，除了它工作在已存

在的纹理物体上。

调用 Aplly 来实际上载改变后的像素到显卡。

不允许调整压缩纹理格式的大小。

◆ function Resize(width: int, height: int): bool

描述：调整纹理大小。

改变纹理的大小为 width 乘 height。调整大小后，纹理像素将是未定义的。这个函数非常类似与纹理构造器，除了它工作在已存在的纹理物体上。

调用 Aplly 来实际上载改变后的像素到显卡。

不允许调整压缩纹理格式的大小。

◆ function SetPixel(x: int, y: int, color: Color): void

描述：在坐标（x，y）处设置像素颜色。

调用 Aplly 来实际上载改变后的像素到显卡。上载是非常耗时的操作，因此你要在 Apply 调用之间改变尽可能多的像素。

如果你需要在运行时频繁重计算一个纹理，生产一个像素颜色数组并用 SetPixels 一次设

置它们，这种方法要快一些。

该函数只工作在 ARGN32, RGB24 和 Alpha8 纹理格式上。对于其他格式 SetPixel 被忽略。

```
function Start(){
//创建一个新的纹理并赋值它到渲染器材质
var texture=new Texture2D(128,128);
renderer.material.mainTexture=texture;
//用 Sierpinski 分形模式填充！
for(y=0; y<texture.height; ++y) {
for(x=0; x<texture.width; ++x) {
var color=(x&y)? Color.white: Color.gray;
texture.SetPixel(x, y, color);
}
}
//应用所有的 SetPixel 调用
texture.Apply(); }
```

参见：SetPixels，GetPixel, Apply.

◆ function SetPixels(Colors: Color[], miplevel: int): void

描述：设置一块像素颜色。

这个函数取的并改变纹理整个 mip 等级的像素颜色数组调用 Apply 来实际上载改变后的像素到显卡.

Colors 数组被放置在 2D 数组中，这里，像素被从左到右，从上到下放置（行序）数组的大小必须至少是所使用的 mip 等级的宽乘高。默认的 mip 等级是零（基本纹理）在这种

情况下大小仅为纹理的大小。一般地，mip 等级尺寸是 mipWidth-max(1,width>>miplecvel)

高度类似。

该函数只工作在 ARGB32, RGB24 和 Alpha8 纹理格式上。对于其他格式 GetPixels 被忽略。

使用 GetPixels 比重复调用 GetPixel 更快，尤其是对于大纹理. 此外，GetPixels 可以访问

单独的 mipmap 等级.

参见：GetPixels, Apply, mipmapCount.

```
//这个脚本用不同的颜色修改纹理的 Mip 等级
//(第一个等级为红色，第二个为绿色，第三个为蓝色)，你可以使用这个来查看
//那个 Mip 被实际使用和如何使用.
function Start(){
//赋值原始的纹理并赋值给材质
var texture: Texture2D=Instantiate(Renderer.material.mainTexture);
renderer.material.mainTexture=texture;
//colors 用来修改前 3 个 mip 等级
var colors=new Colors[3];
colors[0]=Color.red;
colors[1]=Color.green;
colors[2]=Color.blue;
```

```
var mipCount=Mathf.Min(3, texture.mipmapCount);
//修改每个 Mip 等级
for(var mip=0; mip<mipCount; ++mip){
var cols=texture.GetPixels(mip);
for(var i=0; i<cols.Length; ++i){
cols[i]=ColorLerp(cols[i], colors[mip], 0.33);
}
texture.SetPixel(cols, mip);
}
//实际应用 SetPixels，不重新计算 mip 等级
texture.Apply(false);
}
```

◆ function SetPixels(x: int, y: int, blockWidth: int, blochHeight: int, Colors: Color[], miplevel: int): void

描述：设置一块像素颜色。

这个函数是上面 SetPixels 函数的扩展；它不会修改整个 mip 等级而只是修改开始于 x，y

点 blockWidth 乘 blockHeight 的区域。该 colors 数组必须是 blockWidth*blockHeight 大小，并

且可修改的块比例适应适应的 Mip 等级。

继承的成员

继承的变量

width 纹理的像素宽度（只读）

height 纹理像素高度（只读）

filterMode 纹理的过滤模式

anisoLevel 纹理的各向异性过滤等级

wrapMode 纹理的包裹模式（Repeat 或 Clamp）

mipMapBias 纹理的 mipMap 便宜。

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的函数

GetInstanceID 返回该物体的实例 id。

继承的类函数

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator != 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。

Particle

结构

参见：ParticleEmitter，Particles documentation.

变量

◆ var color: Color

描述：粒子的颜色。

颜色的 Alpha 通道用来淡出粒子。

参见：Particles documentation.

◆ var energy: float

描述：粒子的能量。

这是粒子生存的时间。当能量低于零时粒子将被销毁。

能量也被用于 UV 动画。

如果 energy 等于 ParticleEmitter.maxEnergy 第一个瓦片将被使用。

如果 energy 等于 0 最后一个瓦片将被使用。

参见：Particles documentation.

◆ var position: Vector3

描述：粒子的位置。

参见：Particles documentation.

◆ var size: float

描述：粒子的大小。

在世界空间中粒子的尺寸，以来计。

参见：Particles documentation.

◆ var velocity: Vector3

描述：粒子的速度。

如果有一个 particel animator 它将根据 velocity 移动粒子. 如果 Stretch Particles 被设置为

ParticleRenderMode.Stretch，这个速度也被 particle render 使用。

参见：Particles documentation.

Path

类

类方法

◆ static function Combine(path1: string, path2: string): string

描述：

◆ static function GetDirectoryName(path: string): string

描述：

◆ static function GetExtension(path: string): string

描述：

◆ static function GetFileName(path: string): string

描述：

◆ static function GetFileNameWithoutExtension(path: string): string

描述：

Physics

类

全局屋里属性和辅助方法。

类变量

◆ static var bounceThreshold: float

描述：两个碰撞物体的相对速度对于这个值时将不会反弹（默认为 2）。必须为正
这个值可以在 Edit->Project Settings->Physics 的检视面板中改变而不是通过脚本。

◆ static var gravity: Vector3

描述：应用到场景所以刚体的重力。
重力可以通过在单体刚体上使用 useGravity 属性关闭。

```
Physics.gravity=Vector3(0,-1,0,0);
```

◆ static var maxAngularVelocity: float

描述：允许用于任何刚体的缺省最大角速度（默认为 7）。必须为正
刚体的角速度最大为 maxAngularVelocity 以避免高速旋转物体的数值不稳定性。因为这也许会阻止企图快速旋转的物体，例如车轮，你可以使用 Rigidbody.maxAngularVelocity
逐刚体重载该值。
这个值可以在 Edit->Project Settings->Physics 的检视面板中改变而不是通过脚本。

```
Physics.maxAngularVelocity=10;
```

◆ static var minPenetrationForPenalty: float

描述：最小的接触渗透值，以便应用一个罚力（默认为 0.05）必须为正
这个值可以在 Edit->Project Settings->Physics 的检视面板中改变而不是通过脚本。

```
Physics.minPenetrationForPenalty=0.1;
```

◆ static var sleepAngularVelocity: float

描述：缺省的角速度。低于该值的物体将开始休眠（默认为 0.14）。必须为正
参考 Rigidbody Sleeping 获取更多信息。这个值可以使用 Rigidbody.sleepAngularVelocity 来逐刚体重载。
这个值可以在 Edit->Project Settings->Physics 的检视面板中改变而不是通过脚本。

```
Physics.sleepAngularVelocity=0.1;
```

◆ static var sleepVelocity: float

描述：缺省的线性速度，低于改值的物体将开始休眠（默认为 0.15）。必须为正。
参考 Rigidbody Sleeping 获取更多信息。则会更值可以使用 Rigidbody.sleepVelocity 来逐刚体重载。
这个值可以在 Edit->Project Setting->Physics 的检视面板中改变而不是通过脚本

```
Physics.sleepVelocity=0.1;
```

◆ static var solverIterationCount: int

描述：允许用于任何刚体的缺省迭代数（默认为 7）。必须为正。
solverIterationCount 聚顶关节和接触点如何精确地计算。如果出现链接的物体震荡和行为怪异，为 solver Iteration Count 设置一个较高的值将改善他们的稳定性（但是比较慢）。通
常值 7 可以在几乎所有的情况下工作的很好。
这个值可以在 Edit->Project Settings->Physics 的检视面板中改变而不是通过脚本.

```
Physics.solverIterationCount=10;
```

类方法

◆ static function CheckCapsule(start: Vector3, end: Vector3, radius: float, layermask: int= kDefaultRaycastLayers): bool

描述：如果有任何碰撞器接触到由世界坐标中的 start 和 end 构成的轴并具有 radius 半径的胶囊时返回真。

◆ static function CheckSphere(position: Vector3, radius: float, layermask: int=

kDefaultRaycastLayers): bool

描述：如果有任何碰撞器接触到由世界坐标中的 position 和 radius 定义的球体时返回真。

◆ static function IgnoreCollision(collider1: collider, collider2: collider, ignore: bool= true): void

描述：使碰撞检测系统忽略所有 collider1 和 collider2 之间的任何碰撞。

这是最有用的，如投射物不与发射他们的物体碰撞。

IgnoreCollision 有一些限制：1）它不是持久的. 这个以为着当保存场景时，忽略碰撞状态将不会存储在编辑器重。2）你只能将忽略碰撞应用到激活物体的碰撞器上. 当不激活碰撞器或附加的刚体时，IgnoreCollision 将丢失并且你必须再次调用 Physics.IgnoreCollision.

```
//实例化一个子弹并使它忽略与这个物体的碰撞
var bulletPrefab: Transform;
function Start()
{
var bullet=Instantiate(bulletPrefab);
Physics.IgnoreCollision(bullet.collider, collider);
}
```

◆ static function Linecase(start: Vector3, end: Vector3, layerMask: int= kDefaultRaycastLayers): bool

描述：如果有任何碰撞器与从 start 开始到 end 的线段相交时返回真.

```
var target: Transform;
function Update(){
if(!Physics.Linecast(transform.position, target.position}){
//做一些事情
}
}
```

当投射射线时，Layer mask 被用来选择性的忽略碰撞器.

◆ static function Linecase(start: Vector3, end: Vector3, out hitInfo: RaycastHit, layerMask: int=kDefaultRaycastLayers): bool

描述：如果有任何碰撞器与从 start 开始到 end 的线段相交时返回真.

如果返回真，hitInfo 将包含更多关于碰撞器被碰到什么地方的信息（参考：RaycastHit）.

当投射射线时，Layer mask 被用来选择性的忽略碰撞器.

◆ static function OverlapSphere(position: Vector3, radius: float, layerMask: int= kAllLayers): Collider[]

描述：返回触碰到的或在球体内的所有碰撞器的列表.

注意：当前这个值检查碰撞器的包围体耳不是实际的碰撞器。

◆ static function Raycast(origin: Vector3, direction: Vector3, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): bool

参数

origin 世界坐标中射线的开始点。

direction 射线的方向。
distance 射线的长度。
layerMask 当投射射线时，layer mask 被用来选择性的忽略碰撞器.
返回：布尔值-当射线碰到任何碰撞器时为真，否则为假.
描述：投射一个射线与场景中的所有碰撞器相交。

```
function Update(){
var hit: RaycastHit;
var fwd=transform.TransformDirection(Vector3.forward);
if(Physics.Raycast(transform.position, fwd, 10))
{
pring("There is something in front of the object!");
}
}
```

◆ static function Raycast(origin: Vector3, direction: Vector3, out hitInfo: RaycastHit, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): bool
参数
origin 世界坐标中射线的开始点。
direction 射线的方向。
distance 射线的长度。
hitInfo 如果返回真，hitInfo 将包含更多关于碰撞器被碰到什么地方的信息（参考: RaycaseHit）.
layerMask 当投射射线时，layer mask 被用来选择性的忽略碰撞器.
返回：布尔值-当射线碰到任何碰撞器时为真，否则为假.
描述：投射一个射线并碰撞场景中的所有碰撞器闭并返回碰撞的细节。

```
function Update(){
var hit: RaycastHit;
if(Physics.Raycast(transform.position, -Vector3 up, hit)){
distanceToGround=hit.distance;
}
}
//向上投射 100 米
function Update()
{
var hit: RaycastHit;
if(Physics.Raycast(transform.position, -Vector3 up, hit, 100.0)){
distanceToGround=hit.distance;
}
}
```

◆ static function Raycast(ray: Ray, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): bool
参数
ray 射线的开始点和方向。
distance 射线的长度。
layerMask 当投射射线时，layer mask 被用来选择性的忽略碰撞器.

返回：布尔值-当射线碰到任何碰撞器时为真，否则为假.
描述：同上使用 ray.origin 和 ray.direction 而不是 origin 和 direction.

```
var ray=Camera.main.ScreenPointToRay(Input.mousePosition);
if(Physics.Raycast(ray, 100)){
print("Hit something");
}
```

◆ static function Raycast(ray: Ray, out hitInfo: RaycastHit, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): bool

参数
ray 射线的开始点和方向。
distance 射线的长度。
hitInfo 如果返回真，hitInfo 将包含更多关于碰撞器被碰到什么地方的信息（参考：RaycastHit）
layerMask 当投射射线时，layer mask 被用来选择性的忽略碰撞器.
返回：布尔值-当射线碰到任何碰撞器时为真，否则为假.
描述：同上使用 ray.origin 和 ray.direction 而不是 origin 和 direction.

```
var ray=Camera.main.ScreenPointToRay(Input.mousePosition);
var hit: RaycastHit;
if(Physics.Raycast(ray, hit, 100)) {
Debug.DrawLine(ray.origin, hit.point);
}
```

◆ static function RaycastAll(ray: Ray, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): RaycastHit[]
◆ static function RaycastAll(origin: Vector3, direction: Vector3, distance: float=Mathf.Infinity, layerMask: int=kDefaultRaycastLayers): RaycastHit[]

描述：投射一个穿过场景射线并返回所有的碰撞的东西。

```
function Update(){
var hit: RaycastHit[];
hits=Physics.RaycastAll(transform.position,transform.forward, 100.0);
//改变所有碰到的碰撞器的材质
//使用一个透明 Shader
for(var i=0,i<hits.length,i++) {
var hit: RaycastHit=hits[i];
var renderer=hit collider.renderer;
if(renderer)
{
renderer.material.shader=Shader.Find("Transparent/Diffuse")
renderer.material.color.a=0.3;
}
}
}
```

Ping
类
变量

◆ var ip: string
描述：ping 的 IP 目标。
◆ var isDone: bool
描述：ping 函数已经完成？
◆ var time: int
描述：在 isDone 返回真厚，这个属性包含 ping 的时间结果。
构造函数
◆ static function Ping(address: string): Ping
描述：对提供的目标 IP 地址执行 ping 操作。
这个不执行主机名的 DNS 插值，所以它只接受的 IP 地址。
Plane
结构
表示一个平面。
平面式由一个法向量和原点到平面的距离定义的。
变量
◆ var distance: float
描述：从原点到平面的距离
构造函数
◆ staic function Plane(inNormal: Vector3, inPoint: Vector3): Plane
描述：创建一个平面
平面具有法线 inNormal 并通过 inPoint 点，inNormal 不需要被规范化
◆ staic function Plane(inNormal: Vector3,d: float): Plane
描述：创建一个平面
平面具有法线 inNormal 和距离 d，inNormal 不需要被规范化
◆ staic function Plane(a: Vector3, b: Vector3, e: Vector3): Plane
描述：创建一个平面
屏幕通过给定的三个点
函数
◆ function GetDistanceToPoint(inPt: Vector3): float
描述：从平面到点的距离
◆ function GetSide(inPt: Vector3): bool
描述：一个点是否位于平面的正侧
◆ function Raycast(rayt: Ray, out enter: float): bool
描述：一个射线与平面相交
这个函数设置 enter 为沿着射线的距离，这个距离是它与平面相交的位置。如果这个射线与平面平行，函数返回 false 并设置 enter 为零。
◆ function SameSide(inPt0: Vector3, inPt1: Vector3): bool
描述：两个点是否在平面的同侧
PlayerPrefsException
类，继承自 Exception
这个异常时由 PlayerPrefs 类在 web 模式时抛出的，表示偏好文件超出了分配的存储空间.
PlayerPrefs
类

在游戏会话中保持并访问玩家偏好设置。

在 Mac OS X 上 PlayerPrefs 存储在-/Library/PlayerPrefs 文件夹，名文 unity/[company name]\[product name].plist,这里 company 和 product 是在 Project Setting 中设置的，相同

的 plist 用于在编辑器中运行的工程和独立模式.

在 Windows 独立模式下，PlayerPrefs 被存储在注册表的 HKCU Software[company name]\[product name]键下，这里 company 和 product 是在 Project Setting 中设置的.

在 Web 模式，PlayerPrefs 存储在 Mac OS X 的二进制文件

-/Library/Preferences/Unity/WebPlayerPrefs 中和 Windows 的

%APPDATA%\Unity\WebPlayerPrefs 中，一个偏好设置文件对应一个 web 播放器 URL 并且

文件大小被限制为 1 兆。如果超出这个限制，SetInt，SetFloat 和 SetString 将不会存储值并相

处一个 PlayerPrefsException.

类方法

◆ static function DeleteAll(): void

描述：从设置文件中移除所有键和值，谨慎的使用它们。

◆ static function DeleteKey(key: string): void

描述：从设置文件中移除 key 和它对应的值.

◆ static function GetFloat(key: string, defaultValue: float=OF): float

描述：返回设置文件中 key 对应的值，如果存在.

如果不存在，它将返回 defaultValue.

```
print(PlayerPrefs.GetFlat("Player score"));
```

◆ static function GetInt(key: string, defaultValue: int): int

描述：返回设置文件中 key 对应的值，如果存在.

如果不存在，它将返回 defaultValue.

```
print(PlayerPrefs.GetInt("Player score"));
```

◆ static function GetString(key: string, defaultValue: string=**): string

描述：返回设置文件中 key 对应的值，如果存在.

如果不存在，它将返回 defaultValue.

```
print(PlayerPrefs.GetString("Player Name"));
```

◆ static function HasKey(key: string): bool

描述：在设置文件如果存在 key 则返回真.

◆ static function SetFloat(key: string, value: float): void

描述：设置由 key 确定的值.

```
print(PlayerPrefs.SetFloat("Player Score", 10.0));
```

◆ static function SetInt(key: string, value: int): void

描述：设置由 key 确定的值.

```
PlayerPrefs.SetInt("Player Score", 10);
```

◆ static function SetString(key: string, value: string): void

描述：设置由 key 确定的值.

```
PlayerPrefs.Setstring("Player Name", "Foobar");
```

QualitySettings

类

用于 Quality Settings 的脚本接口.

有六个质量等级可以选择；每个等级的细节都在工程的 Quality Settings 中设置，在运行

的时候，当前质量等级可以使用这个类来改变。

类变量

◆ static var currentLevel: QualityLevel

描述：当前图像质量等级。

QualitySettings.currentLevel=QualityLevel.Good;

参见：QualityLevel 枚举，Quality Settings.

◆ static var pixelLightCount: int

描述：影响一个物体的最大像素光源的质量。

如果有更多的光照亮该物体，最暗的一个将作为顶点光源被渲染。

如果你想更好的控制而不是使用质量设置等级，从脚本中使用这个。

//为每个物体使用最多一个像素光

QualitySettings.pixelCount=1;

参见：Quality Settings.

◆ static var softVegetation: bool

描述：使用一个双 pass 着色器，用于地形引擎中的植被。

如果启用，植被将有平滑的边缘，如果禁用所有植物将有硬边但是渲染会快两倍。

参见：Quality Settings.

类方法

◆ static function DecreaseLevel(): void

描述：减小当前质量等级。

QualitySettings.DecreaseLevel();

参见：IncreaseLevel, currentLevel, Quality Settings.

◆ static function IncreaseLevel(): void

描述：增加当前质量等级。

QualitySettings.IncreaseLevel();

参见：DecreaseLevel, currentLevel, Quality Settings.

Quaternion

结构

四元组用来表示旋转

它们是紧凑的，不会出现万向节锁能够很容易被插值。Unity 内如使用 Quaternion 表示所有旋转。

然而，它们基于复述的并不容易被理解。因此你没有必要访问或修改 Quaternion 组

件(x,y,z,w)；最常用的应该是事业已有的旋转（例如，来自 Transform）并使用它们来构造

新的旋转（例如，在两个旋转间平滑地插值）.99%的时间你会使用四元组函数（其他函数仅额

外使用）Quaternion.LookRotation, Quaternion.Angle, Quaternion.Euler, Quaternion.Slerp, Quaternion.FromToRotation, Quaternion.identity

变量

◆ var eulerAngles: Vector3

描述：返回表示旋转的欧拉角。

返回一个旋转，这个旋转绕着 x 轴旋转 euler.x 度，绕着 y 轴旋转 euler.y 度，绕着 euler.z 轴旋转 euler.x 度。

```
//创建一个旋转
var rotation=Quaternion.identity;
//赋值一个旋转为绕着 y 轴 30 度；
rotation.eulerAngles=Vector3(0,30,0);
//打印燃着 y 轴的选择
print(rotation.eulerAngles.y);
```

◆ var this[index: int]: float

描述：分别使用[0], [1],[2],[3]访问 x,y,z,w 组件。

```
Quaternion p;
p[3]=0.5;//与 p.w=0.5 相同
```

◆ var w: float

描述：Quaternion 的 W 组件，不要直接修改这个，除非你了解四元组内部

◆ var x: float

描述：Quaternion 的 X 组件，不要直接修改这个，除非你了解四元组内部

◆ var y: float

描述：Quaternion 的 Y 组件，不要直接修改这个，除非你了解四元组内部

◆ var z: float

描述：Quaternion 的 Z 组件，不要直接修改这个，除非你了解四元组内部

构造函数

◆ static function Quaternion(x: float, y: float, z: float, w: float): Quaternion

描述：用给定的 x,y,z,w 组件构造新的四元组。

函数

◆ function SetFromToRotation(fromDirection: Vector3, toDirection: Vector3): void

描述：创建一个从 fromDirection 到 toDirection 的旋转。

◆ function SetLookRotation(view: Vector3, up: Vector3=Vector3.up): void

描述：创建一个旋转，沿着 forward 看并且头沿着 upwards 向上

如果前向为零将记录一个错误。

◆ function ToAngleAxis(out angle: float, out axis: Vector3): void

描述：转化一个旋转为角度-轴表示。

```
//提取角度-来自变换选择的轴旋转
var angle=0.0;
var axis=Vector3.zero;
transform.rotation.ToAngleAxis(angle.axis);
```

◆ function ToString(): string

描述：返回格式化好的这个四元组的字符串。

```
print(Quaternion.identity):
```

类变量

◆ static var identity: Quaternion

描述：返回单位旋转（只读）．这个四元组对应与"无旋转"：这个物体完全与世界或父的

轴对齐。

```
transform.rotation=Quaternion.identity;
```

类方法

◆ **static function Angles(a: Quaternion, b: Quaternion): float**

描述：返回两个旋转 a 和 b 之间的角度

```
//计算这个变换和
//target 之间的角度.
var target: Transform;
function Update()
var angle=Quaternion.Angles(transform.rotation,target.rotation);
```

◆ **static function AnglesAxis(angle: float, axis: Vector3): Quaternion**

描述：创建一个旋转绕着 axis 轴旋转 angle 度。

```
//设置旋转绕着 y 轴旋转 30 度
transform.rotation=Quaternion.AnglesAxis(30, Vector3.up);
```

◆ **static function Dot(a: Quaternion, b: Quaternion): float**

描述：两个旋转之间的点乘。

```
print(Quaternion.Dot(transform.rotation, Quaternion.identity));
```

◆ **static function Euler(x: float, y: float, z: float): Quaternion**

描述：返回一个旋转，这个旋转绕着 x 轴旋转 x 度，绕着 y 轴旋转 y 度，绕着 z 轴旋转 z 度。

```
//绕着 y 轴旋转 30 度
var rotation=Quaternion.Euler(0,30,0);
```

◆ **static function Euler(euler: Vector3): Quaternion**

描述：返回一个旋转，这个旋转绕着 x 轴旋转 x 度，绕着 y 轴旋转 y 度，绕着 z 轴旋转 z 度。

```
//绕着 y 轴旋转 30 度
var rotation=Quaternion.Euler(Vector3(0,30,0));
```

◆ **static function FromToRotation(fromDirection: Vector3, toDirection: Vector3): Quaternion**

描述：创建一个从 fromDirection 到 toDirection 的旋转.

通常你使用这个来旋转一个变换，这样它的一个周，例如 y 轴-遵循世界空间的 toDirection 目标方向。

```
//设置旋转这样变换的 y 轴沿着表面的法线
transform rotation=Quaternion.FromToRotation(Vector3.up, surfaceNormal);
```

◆ **static function Inverse(rotation: Quaternion): Quaternion)**

描述：返回 rotation 的逆.

```
//设置这个变换为具有 target 的方向旋转
var target: Transform;
function Update()
{
transform.rotation=Quaternion.Inverse(target.rotation);
}
```

◆ **static function Lerp(a: Quaternion, b: Quaternion, t: float): Quaternion**

描述：从 from 到 to 基于 t 插值并规范化结果.

这个比 Slerp 快但是如果旋转较远看起来就较差。

```
//在 from 到 to 之间
```

```
//插值旋转.
//（from 和 to 不能
//与附加脚本的物体相同）
var from: Transform;
var toL Transform;
var speed=0.1;
function Update()
{
transform.rotation=Quaternion.Lerp(from.rotation, to.rotation, Time.time*speed);
}
```

◆ static function LookRotation(forward: Vector3, upwards: Vector3=Vector3.up): Quaternion

描述：创建一个旋转，沿着 forward 看并且沿着 upwards 向上

如果向前为零将记录一个错误。

```
//大多数时候你可以使用。
//transform.LookAt 来代替
var target: Transform
function Update()
{
var relativePos=target.position-transform.position;
var rotation=Quaternion.LookRotation(relativePos);
transform.rotation=rotation;
}
```

◆ static operator!=(lhs: Quaternion, rhs: Quaternion): bool

描述：两个四元组不对等？

这个函数测试两个四元组的点乘是否小于 1.0

注意，因为四元组最多可以表示两个全旋转（720 度），这个比较可以返回 true 即使产生旋转看起来一起。

◆ static operator*(lhs: Quaternion, rhs: Quaternion): Quaternion

描述：组合 lhs 和 rhs 旋转.

首先用 lhs 旋转一个点然后用 rhs 继续旋转，与使用组合旋转相同。注意旋转不可交换：lhs*rhs 不等于 rhs*lhs.

```
//应用
//extraRotation 当当前旋转
var extraRotation: Transform;
transform.rotation*=extraRotation.rotation;
```

◆ static operator*(rotation: Quaternion, point: Vector3): Vector3

描述：用 rotation 旋转点 point

```
//沿着 relativeDirection 移动物体
//通常你应该使用 transform.Move
var relativeDirection=Vector3.forward;
function Update() {
var absoluteDirection=transform.rotation*relativeDirection;
```

transform.position+=absoluteDirection*Time.deltaTime;
}

◆ static operator==(lhs: Quaternion, rhs: Quaternion): bool

描述：两个四元组相等？

这个函数测试两个四元组的点乘是否接近 1.0

注意，因为四元组可以表示最大两个全旋转（720 度），这个比较可以返回 false 即使产生旋转看起来一起。

◆ static function Slerp(from: Quaternion, to: Quaternion, t: float): Quaternion

描述：从 from 到 to 基于 t 的球形插值.

```
//在 from 和 to 之间
//插值旋转.
//(from 和 to 不能
//与附加脚本的物体相同)
var from: Transform;
var to: Transform;
var speed=0.1;
function Update() {
transform.rotation= Quaternion.Slerp(from.rotation, to.rotation, Time.time*speed);
}
```

Random

类

用于生成随机数据的类。

类变量

◆ static var insideUnitCircle: Vector2

描述：返回半径为 1 的圆内部的一个随机点（只读）。

```
//设置这个位置为一个半径为
//5，中心在零点的圆环内部的一个点.
var newPosition: Vector2=Random.insideUnitCircle*5;
transform.position.x=newPosition.x;
transform.position.y=newPosition.y;
```

◆ static var insideUnitSphere: Vector3

描述：返回半径为 1 的球体内部的一个随机点（只读）。

```
//设置这个位置为一个半径为 5
//中心在零点的球体内部的一个点.
transform.position=Random.insideUnitSphere*5;
```

◆ static var onUnitSphere: Vector3

描述：返回半径为 1 的球体表面上的一个随机点（只读）。

```
//在一个随机的方向上设置刚体速度为 10.
rigidbody.velocity=Random.onUnitSphere*10;
```

◆ static var rotation: Quaternion

描述：返回 s 随机旋转（只读）。

```
//在世界坐标原点用一个随机旋转实例化一个预设
var prefab: GameObject;
Instantiate(prefab, Vector3.zero., Random.rotation);
```

◆ static var seed: int

描述：设置随机种子生成器。

◆ static var value: float

描述：返回一个 0.0[包含]和 1.0[包含]之间的随机数（只读）.

0.0 和 1.0 会被这个函数返回。

//打印 0 到 1 之间的值

```
print(Random.value);
```

类方法

◆ static function Range(min: float, max: float): float

描述：返回一个随机的浮点数，在 min[包含]和 max[包含]之间（只读）.

//在 x-z 平面上的-10 和 10 之间的某个位置实例化预设

```
var prefab: GameObject;
function Start()
{
var position=Vector3(Random Range(-10, 10), 0, Random Range(-10, 10));
Instantiate(prefab.position, Quaternion.identity);
```

◆ static function Range(min: int, max: int): int

描述：返回一个随机的整数，在 min[包含]和 max[包含]之间（只读）.

如果 max 等于 min，min 将被返回，返回的值不会是 max 除非 min 等于 max

//从关卡到列表中加载一个随机的关卡

```
Application.LoadLevel(Random Range(0, Application.levelCount));
```

Ray

结果

表示射线。

射线时一个无穷线段，开始于 origin 并沿着 direction 方向。

变量

◆ var direction: Vector3

描述：射线的方向。

方向总是一个规范化的向量。如果你赋值一个非单位化长度，它将被规范化。

◆ var origin: Vector3

描述：射线的起点。

构造函数

◆ static function Ray(origin: Vector3, direction: Vector3): Ray

描述：创建一个射线从 origin 开始沿着 direction 方向。

//从变换位置开始沿着变换的 z 轴创建一个射线

```
var ray=new Ray(transform.position, transform.forward);
```

函数

◆ static GetPoint(distance: float): Vector3

描述：返回沿着射线距离为 distance 单位的点。

```
var r: ray;
pting(r.GetPoint(10));//沿着这个射线 10 个单位的点
```

◆ function ToString(): string

描述：返回格式化好的这个射线的字符串

RaycastHit

结构

用来从一个 raycase 获取信息的结构。

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

变量

◆ var barycentricCoordinate: Vector3

描述：所碰到的三角形的重心坐标。

这允许你沿着 3 个轴插值任何顶点数据。

```
//附加这个脚本到相机然后它将
//绘制一个从法线指出的调试直线
function Update(){
//如果碰到物体，继续
var hit: RaycastHit;
if(!Physics.Raycst(Camera.main.ScreenPointToRay(Input.mousePosition), hit))
return;
//以防万一，确保碰撞器也有一个渲染器
//材质和纹理
var meshCollider=hit.collider as MeshCollider;
if(meshCollider==null || meshCollider.sharedMesh=null)
return;
var mesh: Mesh=meshCollider.sharedMesh;
var normals=mesh.normals;
var triangles=mesh.triangles;
//取得所碰到三角形的本地法线
var n0=normals[triangles[hit.triangleIndex*3+0]];
var n1=normals[triangles[hit.triangleIndex*3+1]];
var n2=normals[triangles[hit.triangleIndex*3+1]];
//使用碰撞点的中心坐标来插值
var baryCenter=hit.barycenterCoordinate;
//使用中心坐标插值法线
var interpolatedNormal=n0*baryCenter.x+n1*baryCenter.y+n2*baryCenter.z;
//规范化插值法线
interpolatedNormal=interpolatedNormal.Normalized;
//Transform local space normals to world space
var hitTransform: Transform:=hit.collider.transform;
interpolatedNormal=hitTransform.TransformDirection(interpolatedNormal);
//用 Debug.DrawLine 显示
Debug.DrawRay(hit.poing, interpolatedNormal);
```

◆ var collider: Collider

描述：被碰到的 Collider.

如果没有碰到任何东西该属性为 null 否则为非 null 参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var distance: float

描述：从射线的原点到碰撞点的距离。

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var normal: Vector3

描述：射线所碰到的表面的法线。

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var point: Vector3

描述：在世界空间中，射线碰到碰撞器的碰撞点。

参见：Physics.Raycast, Physics.Linecast.

◆ var rigidbody: Rigidbody

描述：碰到的碰撞器的 Rigidbody. 如果该碰撞器没有附加刚体那么它为 null。

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var textureCoord: Vector2

描述：碰撞点的 UV 纹理坐标。

这个可用于 3D 纹理绘制或绘制弹痕。如果碰撞器时非网格碰撞器，零 Vector2 将被返回.

```
//附加这个脚本到相机，当用户点击时
//它将在 3D 中绘制黑色的像素. 确保你想绘制的网格附加有
//一个网格碰撞器.
function Update()
(
//只有当我们按下鼠标
if(!input.GetMouseButton(0))
return;
//只有碰到某些东西，继续
var hit: RaycastHit;
if(!Physics.Raycast(camera.ScreenPointToRay(Input.mousePosition), hit))
return;
/以防万一，还要确保碰撞器也有一个渲染器
//材质和纹理.我们也应该忽略几何体碰撞器.
var renderer: Renderer=hit.collider.renderer;
var meshCollider=hit.collider as MeshCollider;
if(renderer==null || renderer.sharedMaterial==null ||
renderer.sharedMaterial.mainTexture==null ||
meshCollider==null)
return;
//现在在我们碰到的物体上绘制一个像素
var tex: Texture2D=renderer.material.mainTexture;
var pixelUV=hit.textureCoord;
pixelUV.x*=tex.width;
pixelUV.y*=tex.height;
tex.SetPixel(pixelUV.x, pixelUV.y, Color.black);
tex.Apply();
}
```

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var textureCoord2: Vector2

描述：碰撞点的第二个 UV 纹理坐标。

这个可用于 3D 纹理绘制或绘制弹痕。如果碰撞器时非网格碰撞器，零 Vector2.zero 将被

返回.如果网格没有包含第二个 uv 集，主 uv 集将被返回。

```
//附加这个脚本到相机，当用户点击时
//它将在 3D 中绘制黑色的像素. 确保你想绘制的网格附加有
//一个网格碰撞器.
function Update()
(
//只有当我们按下鼠标
if(!input.GetMouseButton(0))
return;
//只有碰到某些东西，继续
var hit: RaycastHit;
if(!Physics.Raycast(camera.ScreenPointToRay(Input.mousePosition), hit))
return;
/以防万一，还要确保碰撞器也有一个渲染器
//材质和纹理.我们也应该忽略几何体碰撞器.
var renderer: Renderer=hit.collider.renderer;
var meshCollider=hit.collider as MeshCollider;
if(renderer==null || renderer.sharedMaterial==null || renderer.sharedMaterial.mainTexture==null ||
meshCollider==null)
return;
//现在在我们碰到的物体上绘制一个像素
var tex: Texture2D=renderer.material.mainTexture;
var pixelUV=hit.textureCoord2;
pixelUV.x*=tex.width;
pixelUV.y*=tex.height;
tex.SetPixel(pixelUV.x, pixelUV.y, Color.black);
tex.Apply();
}
```

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

◆ var triangleIndex: int

描述：碰到的三角形的索引.

如果被击中的碰撞器时一个 MeshCollider 三角形索引才可用.

```
//附加这个脚本到相机然后它将
//绘制一个调试直线三角形
//在我们放置鼠标的三角形上
function Update()
(
//如果我们击中了某物，继续
var hit: RaycastHit;
if(!Physics.Raycast(camera.ScreenPointToRay(Input.mousePosition), hit))
```

```
return;
/以防万一，确保碰撞器也有一个渲染器
//材质和纹理
var meshCollider=hit.collider as MeshCollider;
if(meshCollider==null ||meshCollider.sharedMesh==null)
return;
var mesh: Mesh=meshCollider.sharedMesh;
var vertices=mesh.vertices;
var triangles=mesh.triangles;
//取回碰到的本地坐标点
var p0=vertices[triangles[hit.triangleIndex*3+0]];
var p1=vertices[triangles[hit.triangleIndex*3+1]];
var p2=vertices[triangles[hit.triangleIndex*3+1]];
//交换本地空间点到世界空间
var hitTransform: transform=hit.collider.transform;
p0=hitTransform.TransformPoint(p0);
p1=hitTransform.TransformPoint(p1);
p2=hitTransform.TransformPoint(p2);
//用 Debug.DrawLine 显示
Debug.DrawLine(p0, p1);
Debug.DrawLine(p1, p2);
Debug.DrawLine(p2, p0);
}
```

参见：Physics.Raycast, Physics.Linecast, Physics.RaycastAll.

RectOffset

类

用于矩形，边界等的偏移。

被 GUIStyle 用在所有地反。

变量

◆ var bottom: int

描述：底部边缘的尺寸。

◆ var horizontal: int

描述：left+right 的快捷方式（只读）。

◆ var left: int

描述：左边边缘的尺寸。

◆ var right: int

描述：右边边缘的尺寸。

◆ var top: int

描述：顶部边缘的尺寸。

◆ var vertical: int

描述：top+bottom 的快捷方式（只读）

函数

◆ function Add(rect: Rect): Rect

描述：向一个 rect 中添加边界偏移。

◆ function Remove(rect: Rect): Rect
描述：从一个 rect 中移除边界偏移。
Rect
结构
由 x，y 位置和 width，height 定义的 2D 矩阵
Rect 结构主要用于 2D 操作；UnityGUI 系统广泛的使用它，也可用来在屏幕上定位相机
参见：GUIScripting.Guide, Camera.rect, Camera.pixelRect.
变量
◆ var height: float
描述：矩形的高。

```
var rect=Rect(0,0,10,10);
print(rect.height);
rect.height=20;
```

◆ var width: float
描述：矩形的宽。

```
var rect=Rect(0,0,10,10);
print(rect.width);
rect.width=20;
```

◆ var x: float
描述：矩形的左侧坐标。
◆ var xMax: float
描述：矩形的右侧坐标
改变这个值将保持矩形的左边（这样宽度也将改变）
◆ var xMin: float
描述：矩形的左侧坐标
改变这个值将保持矩形的右边（这样宽度也将改变）
◆ var y: float
描述：矩形的顶部坐标。

```
var rect=Rect(10,10,100,100);
print(rect.y);
rect.x=20;
```

◆ var yMax: float
描述：矩形的底部坐标。
改变这个将保持矩形的顶边（这样高度也将改变）
◆ var yMin: float
描述：矩形的顶部坐标。
改变这个将保持矩形的底边（这样高度也将改变）
构造函数
◆ static function Rect(left: float, top: float, width: float, height: float): Rect
描述：创建一个新的矩形

```
var rect=Rect(0,0,10,10);
```

函数
◆ function Contains(point: Vector2): bool
◆ function Contains(point: Vector3): bool

描述：如果 point 的 x 和 y 位于矩阵的内部

```
//当坐标？？？
function Update()
{
var rect=Rect(0,0,150,150);
if(rect Contains(Input.???Position))
print("Inside")
}
```

◆ function ToString(): string

描述：返回格式化好的 Rect 字符串、

```
var rect=Rect(0,0,10,10);
print(rect);
```

类方法

◆ static function MinMaxRect(left: float, top: float, right: float, bottom: float): Rect

描述：根据 min/max 坐标值创建一个矩形。

◆ static operator!=(lhs: Rect, rhs: Rect): bool

描述：如果矩形不相同返回真。

◆ static operator==(lhs: Rect, rhs: Rect): bool

描述：如果矩形相同返回真。

RenderSettings

类

Render Settings 包含用于场景中可视元素范围的缺省值，如雾和环境光。

参见：Render Settings manager

类变量

◆ static var ambientLight: Color

变量

◆ var height : int

描述: 解析度的像素高度。

◆ var width : int

描述: 解析度的像素宽度。

◆ var refreshRate : int

描述: 解析度的垂直刷新率 Hz。

Resources

类

Resources 类允许你按照它们的路径名找到并加载物体。

所有位于 Assets 文件夹下名“Resources”的文件夹中的资源,都可以 Resources.Load 函数访问,

在 Unity 中你通常不需要使用路径名来访问资源,相反你可以通过声明一个成员变量来公开到一个资源的应用,然后在检视面板中给它赋值。使用这个技巧的时候 Unity 可以在构建的时候自动计算哪个资源被使用。这从根本上最大限度地减少了实际用于游戏的资源的尺寸。当你放置资源在“Resources”文件夹中时这个不会这么做,因此所有在“Resources”文件夹中的资源都将被包含在游戏中。

另一个使用路径名的缺点是，缺乏可重用性，因为脚本对于使用的资源具有硬编码要求。另一方面使用公开在检视面板中的引用，是自文档化的，对于使用你脚本的用户来说

也是显而易见的。

然而，有些情况下按照名称而不是在检视面板中取回一个资源是更方便的。尤其是当在检视面板中赋值引用是不方便的时候。例如你或许想从脚本创建一个游戏物体，为程序生成的网格赋值一个纹理。

```
var go = new GameObject.CreatePrimitive(PrimitiveType.Plane);
go.renderer.marerial.mainTexture = Resources.Load("glass");
```

类方法

◆ static function Load(path : string) : Object

描述: 加载储存在 Resources 文件夹中的 path 处的资源。

如果在 path 处发现这个资源返回它,否则返回 null。Path 相对于 Resources 文件夹,扩展名必须被忽略。Resources 文件夹可以在 Assets 文件夹中的任何位置。

```
//附加一个名为"Assets/Resources/glass"的纹理到 Plane.
function Start() {
        var go = new GameObject.CreatePrimitive(PrimitiveType.Plane);
        go.renderer.material.mainTexture = Resources.Load("glass");
}
//实例化路径"Assets/Resources/enemy"处的一个预设
function Start () {
var instance : GameObject = Instantiate(Resources.Load("enemy"));
}
```

◆ static function Load (path : String, type : Type) : Object

描述: 加载储存在 Resources 文件夹中 path 处的资源。

如果在 path 处发现这个资源就返回它，否则返回 null。只有 type 类型的物体将被返回. Path 相对于 Resources 文件夹，扩展名必须被忽略。Resouces 文件夹可以在 Assets 文件夹中的任何位置。

```
//附加一个名为”glass”的纹理到 Plane
function Start () {
    var go = new GameObject.CreatePrimitive(PrimitiveType.Cube);
    go.renderer.material.main.Texture = Resources.Load(“glass”, Texture2D);
}
//实例化名为”Resources/enemy”的一个预设
function Start () {
    var instance : GameObject = Instantiate(Resources.Load(“enemy”, GameObject);
}
```

◆ static function LoadAll (path : String, type : Type) : Object[]

描述: 加载 Resources 文件夹中的 path 文件夹或者文件中的所有资源。

如果 path 是一个文件夹，文件中的所有资源都将被返回。如果 path 为一个文件，只有这个资源将被返回。只有 type 类型的物体将被返回。Path 相对于 Resources 文件夹。Resources 文件夹可以在 Assets 文件夹中的任何位置。

```
    //加载”Resources/Texture”文件夹中所有资源
    //然后从列表中选择随机的一个
    function Start () {
        var go = new GameObject.CreatePrimitive(PrimitiveType.Cube);
        var textures : Object[] = Resources.LoadAll(“Textures”, Texture2D);
```

```
    var texture : Texture2D = textures[Random.Range(0, textures.Length)];
    go.renderer.material.mainTexture = texture;
}
```

◆ **static function LoadAll (path : String) : Object[]**

描述: 加载 Resources 文件夹中的 path 文件夹或者文件中的所有资源。

如果 path 是一个文件夹，文件中的所有资源都将被返回。如果 path 为一个文件，只有这个资源将被返回。只有 type 类型的物体将被返回。Path 相对于 Resources 文件夹。Resources 文件夹可以在 Assets 文件夹中的任何位置。

```
//加载”Resources/Texture”文件夹中所有资源
//然后从列表中选择随机的一个
function Start () {
    var go = new GameObject.CreatePrimitive(PrimitiveType.Cube);
    var textures : Object[] = Resources.LoadAll(“Textures”);
    var texture : Texture2D = textures[Random.Range(0, textures.Length)];
    go.renderer.material.mainTexture = texture;
}
```

◆ **static function LoadAssetAtPath (assetPath : String, type : Type) : Object**

描述: 返回资源路径上的一个资源(只在编辑器中)。

在独立版或者 web 版中这个函数总是返回 null。这个可以用来快速访问一个在编辑器中使用的资源。

```
var prefab : GameObject;
function Start () {
prefab = Resources.LoadAssetAtPath(“Assets/Artwork/mymodel.fbx”, GameObject);
}
```

Screen

类

访问显示信息。

屏幕类用来获取支持的解析度列表，切换当前分辨率，隐藏或显示系统的鼠标指针。

类变量

◆ **static var currentResolution : Resolution**

描述: 当前屏幕的解析度。

如果播放器运行在窗口模式，这将返回当前左面的解析度。

```
print(Screen.currentResolution);
```

◆ **static var fullScreen : bool**

描述: 游戏是否运行在全屏模式下？

通过改变这个属性来切换全屏模式。

```
//切换全屏
Screen.fullScreen = !Screen.fullScreen;
```

参见: SetResolution。

◆ **static var height : int**

描述: 屏幕窗口的当前像素高度(只读)。

播放窗体的实际高度(在全屏模式下也是当前的分辨率高度)

◆ **static var lockCursor : bool**

描述: 光标是否被锁定？

光标将被自动隐藏、居中并且不会离开这个视点。

在 web 播放器中光标只在用户点击内容并且用户的鼠标没有离开内容视图的时候锁定。用户按下 escape 或切换到其他应用程序时光标将被自动锁定。当退出全屏模式时，光标锁定也将丢失。通过检查 lockCursor 状态，你可以查询光标当前是否被锁定。为了提供一个好的用户体验，建议只在用户按下按钮的时候锁定光标。你也应该光标是否解锁，例如暂停游戏或者打开游戏菜单。在 web 播放器和编辑器中，当你按下 escape 时光标将自动被解锁。在独立播放模式下你有完全的鼠标锁定控制权，因此它不会自动释放鼠标锁定，除非你切换应用程序。

```
//当鼠标被锁定时调用
function DidLockCursor () {
    Debug.Log("Locking cursor");
    //禁用按钮
    guiTexture.enabled = false;
}
//当光标被解锁时调用
//或被脚本调用 Screen.lockCursor = false;
Function DidUnlockCursor () {
    Debug.Log("Unlocking cursor");
    //再次显示按钮
    guiTexture.enabled = true;
}

function OnMouseDown () {
    //锁定光标
    Screen.lockCursor = true;
}
private var wasLocked = false;
function Update () {
    //在独立模式下我们需要提供自己的键来锁定光标
    If(Input.GetKeyDown("escape")
        Screen.lockCursor = false;
    //我们失去了鼠标锁定？
    //例如因为用户按下 escape 键或因为他切换到了其他应用程序
//或者因为一些脚本设置 Screen.lockCursor = false;
If(!Screen.lockCursor && wasLocked) {
    wasLocked = false;
    DidUnlockCursor();
}
//再次获得光标锁定？
else if(Screen.lockCursor && wasLocked) {
    wasLocked = true;
    DidLockCursor();
```

```
}
}
```

◆ **static var resolutions : Resolution[]**

描述: 该显示器支持的所有全屏解析度(只读)

返回的解析度按照宽度排序，较低的解析度位于前面。

```
var resolutions : Resolutions[] = Screen.resolutions;
//打印解析度
for(var res in resolutions) {
    print(res.width + "x" + res.height);
}
//切换到支持的最低全屏解析度
Screen.SetResolution(resolutions[0].width, resolutions[0].height, true);
```

参见: Resolution 结构, SetResolution。

◆ **static var showCursor : bool**

描述: 光标是否可见？

完全可以实现一个自定义的光标而不是系统的。要做到这一点，你要隐藏系统的，然后跟踪位置或移动并在需要的位置显示你自己的图片。

```
//隐藏光标
Screen.showCursor = false;
```

◆ **static var width : int**

描述: 屏幕窗口的当前像素宽度(只读)。

播放窗体的实际宽度(在全屏模式下也是当前的分辨率宽度)

类方法

◆ **static function SetResolution (width : int, height : int, fullscreen : bool, preferredRefreshRate : int) : void**

描述: 切换屏幕的解析度。

Width x height 解析度将被使用。如果没有匹配的解析度支持，最近的一个将被使用。

如果 preferredRefreshRate 为 0 (默认)，Unity 将切换到显示器支持的最高刷新率。

如果 preferredRefreshRate 不是 0，如果显示器支持 Unity 将使用它，否则将选择所支持的最高的一个。

在 web 播放器中你只需要在用户点击内容之后切换分辨率。推荐的方法是只在用户点击指定按钮的时候切换分辨率。

```
//切换成 640 x 480  全屏模式
Screen. SetResolution(640, 480, true);
//切换成 640 x 480  全屏模式  60hz
Screen. SetResolution(640, 480, true, 60);
//切换成 800 x 600  窗口模式
Screen. SetResolution(800, 600, false);
```

参考: resolution 属性。

SoftJointLimit

结构

这限制是由 CharacterJoint 定义的。

变量

- **var bouneyness : float**
  描述: 当关节到达这个限定时，它可以使其弹回。
  bouneyness 决定弹回的限制，范围{0, 1};

- **var damper : float**
  描述: 如果弹力大于 0，限制被减缓。
  这是弹簧的 z 阻力，范围{0, 无穷};

- **var limit : float**
  描述: 关节的位置/角度限制。

- **var spring : float**
  描述: 如果大于零，限制被减缓。弹簧将推回该关节。 {0, 无穷}

String

类

表示一系列 Unicode 字符文本。

变量

- **var Length : int**
  描述: 获取该实例中字符的数量。

- **var String : static**
  描述: 初始化一个新的实例化 String 类，它的值为一个新的 Unicode 字符数组。数组开始字符的位置，和一个长度。表示一个空的字符串，这个域是只读的。

SystemInfo

类

类变量

- **static var graphicsDeviceName : string**
  描述: 显示设备的名称(只读)。
  这个是用户显卡的名称，由显卡设备驱动报告。

```
//打印”ATi Raedon X1600 OpenGL Engine”在 MacBook Pro OS X 10.4.8 上
print(SystemInfo.graphicsDeviceName);
```

  参见: SystemInfo.graphicsDeviceVendor, SystemInfo.graphicsDeviceVersion.

- **static var graphicsDeviceVendor : string**
  描述: 显示设备的生产厂商(只读)。
  这个是用户显卡的生产厂商，由显卡设备驱动报告。

```
//打印”ATi Technologies Inc.” 在 MacBook Pro OS X 10.4.8 上
print(SystemInfo.graphicsDeviceVendor);
```

  参见: SystemInfo.graphicsDeviceName, SystemInfo.graphicsDeviceVersion.

- **static var graphicsDeviceVersion : string**
  描述: 该显示设备所支持的图形 API 版本(只读)。
  这个是用户显卡设备所支持的底层图形 API 版本。
  如果是 OpenGL API，返回字符串包含”OpenGL”然后是”major.minor”格式的版本号，然后在方括号中试完整的版本号。

如果是 Direct3D9 API，返回的字符串包含”Direct3D 9.0c”，然后再方括号中试驱动名称和版本号。

//打印”OpenGL 2.0[2.0 ATi-1.4.40]”在 MacBook Pro OS X 10.4.8 下。

//打印”Direct3D 9.0c[atiumdag.dll 7.14 10.471]”在 MacBook Pro/ Windows Vista 下。

print(SystemInfo.graphicsDeviceVersion);

参见: SystemInfo.graphicsDeviceName, SystemInfo.graphicsDeviceVendor.

- static var graphicsMemorySize : int

描述: 显存的大小(只读)。

这个是以兆(MB)计算的显存容量。

- static var supportsImageEffects : bool

描述: 是否支持图像效果?(只读)

如果显卡支持后期处理特效则返回 true。

参见: Image Effects。

- static var supportsRenderTexture : bool

描述: 是否支持渲染到纹理?(只读)

如果显卡支持渲染到纹理则返回 true。使用 SupportsRenderTextureFormat 来检查支持的特定渲染纹理格式。

参见: Render Texture assets，RenderTexture 类。

- static var supportsShadows : bool

描述: 是否支持内置阴影?(只读)

如果显卡支持内置阴影则返回 true。

参见: Shadows doucumentation。

类方法

- static function SupportsRenderTextureFormat (format : RenderTextureFormat) : bool

描述: 是否支持该渲染纹理格式?

如果显卡支持指定的 RenderTextureFormat 格式，返回 true。

参见: Render Texture assets，RenderTexture 类。

Time

类

从 Unity 中获取时间的接口。

类变量

- static var captureFramerate : int

描述: 如果 captureFramerate 被设置为大于 0 的值，时间将为(1.0/ captureFramerate)每帧而不考虑实际时间。如果想用一个固定的帧率捕获一个视频时可以使用这个。

```
Time. captureFramerate = 25;
```

- static var deltaTime : float

描述: 用于完成最后一帧的时间(只读)。

如果你加或减一个每帧改变的值，你应该将它与 deltaTime 相乘。当你乘以 Time.deltaTime 时，你实际表达：我想以 10 米/秒移动这个物体而不是 10 米/帧。

在 MonoBehaviour 的 FixedUpdate 内部调用时，返回固定帧率增量时间。

```
function Update () {
    var translation = Time.deltaTime * 10;
    transform.Translate(0, 0, translation);
}
```

◆ **static var fixedDeltaTime : float**

描述: 物理和其他固定帧率更新(如 MonoBehaviour 的 FixedUpdate)以这个间隔来执行。

建议使用 Time.deltaTime 代替，因为如果在 FixedUpdate 函数或 Update 函数内部它将自动返回正确的增量时间。

注意 fixedDeltaTime 间隔对应的游戏时间会受到 timeScale 的影响。

◆ **static var fixedTime : float**

描述: 最后一次 FixedUpdate 调用已经开始的时间(只读)。这个是以秒计的游戏开始的时间。

固定时间以定期的间隔来更新(相当于 fixedDeltaTime)直到到达了 time 属性。

◆ **static var frameCount : int**

描述: 已经传递的帧总数(只读)。

```
//确保 RecaloulateValue 在每帧中只执行一次某些操作
 static private var lastRecaloulation = -1;
 static function RecaloulateValue () {
     if(lastRecaloulation == Time.frameCount)
         return;
     PerformSomeCalculations();
 }
```

◆ **static var realTimeSinceStartup : float**

描述: 从游戏开始的实时时间(只读)。

在大多数情况下你应该使用 Time.time。

realTimeSinceStartup 返回从开始到现在的时间，不会受到 Time.timeScale 的影响。当玩家暂停的时候 realTimeSinceStartup 也会增加(在后台)。使用 realTimeSinceStartup 时很有用的，当你想通过设置 Time.timeScale 为 0 来暂停游戏，但还在某些时候使用时间。

注意 realTimeSinceStartup 返回的时间是来源于系统计时器,根据不同平台和硬件，也许会在一些连贯的帧上报告相同的时间。如果通过不同的时间来区分不同的事情，需要考虑这个(时间的差别可能变为零！)。

```
//打印从开始到现在的真实时间
 print(Time.realTimeSinceStartup);
//一个 FPS 计数器，在每个 updateInterval 间隔处计算帧/秒，这样显示就不会随意的改变。
var updateInterval = 0.5;
private var last      Interval : double;   //最后间隔结束时间
private var frames = 0;//超过当前间隔帧
 private fps : float; //当前 fps

 function Start () {
     lastInterval = Time. realTimeSinceStartup;
     frames = 0;
 }
 function OnGUI () {
//显示两位小数
 GUILayout.xxxx("" + fps.ToString("F2");
```

```
    }
function Update () {
    ++frames;
    var timeNow = Time.realtimeSinceStartup;
    if(timeNow > lastInterval + updateInterval) {
        fps = frames / (timeNow – lastInterval);
        frames = 0;
        lastInterval = timeNow;
    }
}
```

◆ **static var smoothDeltaTime : float**

描述: 一个平滑的 Time.deltaTime(只读)。

◆ **static var time : float**

描述: 该帧开始的时间(只读)。从游戏开始到现在的时间。

在 MonoBehaviour 的 FixedUpdate 内部调用时，返回 fixedTime 属性。

```
//如果 Fire1 按钮(默认为 Ctrl)被按下，每 0.5 秒实例化一个 projectile。
var projectile : GameObject;
var fireRate = 0.5;
private var nextFire = 0.0;
function Update () {
    if(Input.GetButton("Fire1") && Time.time > nextFire) {
        nextFire = Time.time + fireRate;
        clone = Instantiate(projectile, transform.position, transform.rotation);
    }
}
```

◆ **static var timeScale : float**

描述: 传递时间的缩放，可以用来减慢动作效果。

当 timeScale 为 1.0 的时候时间和真实时间一样快；当 timeScale 为 0.5 的时候时间为真实时间的一半速度；当 timeScale 被设置为 0 的时候那么游戏基本上处于暂停状态，如果所有函数都是与帧率无关的。如果降低 timeScale 建议同时降低 Time.fixedDeltaTime 相同的量。当 timeScale 被设置为 0 时 FixedUpdate 将不会被调用。

```
//当用户按下 Fire1 按钮时可以在 1 和 0.7 之间切换时间。
function Update () {
    if(Input.GetButton("Fire1")) {
        if(Time.timeScale == 1.0) {
Time.timeScale = 0.7;
        }
        else {
Time.timeScale = 1.0;
        }
    //根据 timeScale 调整固定增量时间
    Time.fixedDeltaTime = 0.02 * Time.TimeScale;
    }
}
```

◆ **static var timeSinceLevelLoad : float**

描述: 该帧开始的时间(只读)。最后一个关卡被加载到现在的时间。

```
//设置 GUI 文本为玩家完成该关卡的时间。
 function PlayerCompletedGame () {
    guiText.text = Time. timeSinceLevelLoad.ToString;
 }
```

Vector2

结构

表示 2D 向量和点。

这个在某些地方用来表示2D位置和向量(例如，在Mesh中的纹理坐标或者Material中的纹理偏移)。在大多数情况下使用一个 Vector3。

变量

◆ **var magnitude : float**

描述: 返回这个向量的长度(只读)。

向量的长度是(x*x+y*y)的平方根。

如果你只需要比较向量的长度，你可以使用 sqrMagnitude(计算长度的平方式比较快的)比较它们的平方。

参见: sqrMagnitude

◆ **var sqrMagnitude : flaot**

描述: 返回这个向量长度的平方(只读)。

计算长度的平方比 magnitude 更快。通常如果你只比较两个向量的长度你可以只比较它们的长度的平方。

参见: magnitude

◆ **var this[index : int] : float**

描述: 分别使用 0 或 1 来访问 x 和 y 组件。

```
Vector2 p;
 p[1] = 5; //与 p.y = 5 相同
```

◆ **var x : float**

描述: 该向量的 x 组件。

◆ **var y : float**

描述: 该向量的 y 组件。

构造函数

◆ **static function Vector2 (x : float, y : float) : Vector2**

描述: 用指定的 x,y 组件构造一个新的向量。

函数

◆ **function ToString () : String**

描述: 返回格式化好的 Vector 字符串。

类变量

◆ **static var right : Vector2**

描述: Vector2(1,0)的简写。

◆ **static var up : Vector2**

描述: Vector2(0,1)的简写。

◆ **static var zero : Vector2**

描述: Vector2(0,0)的简写。

类方法

- **static function Distance(a:Vector2, b:Vector2):float**
  描述: 返回 a 和 b 之间的距离。
  Vector2.Distance(a,b)和(a-b).magnitude 相同。
- **static function Dot(lhs:Vector2, rhs:Vector2):float**
  描述: 2 个向量的点乘。
  返回 lhs.rhs.

对于规范化的向量，如果它们指向相同的方向 Dot 返回 1；如果指向相反的方向返回-1；在另外的情况下返回一个数字（例如，如果向量垂直则 Dot 返回 0）。

- **static operator* (a:Vector2, d:float):Vector2**
  描述: 用一个数乘以一个向量。
  用 d 乘以 a 的每个组件。

```
// 使向量变为二倍长；打印(2.0, 4.0)
print (Vector2(1,2) * 2.0);
```

- **staic operator* (d:float, a:Vector2):Vector2**
  描述: 用一个数乘以一个向量。
  用 d 乘以 a 的每个组件。

```
// 使向量变为二倍长； 打印（2.0, 4.0）
print(2.0 * Vector2(1, 2));
```

- **static operator + (a:Vector2, b:Vector2):Vector2**
  描述: 两个向量相加。
  将对应的组件加在一起。

```
// 打印(3.0, 5.0)
print(Vector2(1,2) + Vector2(2,3));
```

- **static operator – (a:Vector2, b:Vector2):Vector2**
  描述: 两个向量相减。
  从 a 中减去 b 的每个对应组件。

```
// 打印(-2.0, 0.0)
print(Vector2(1,2)–Vector2(3,2));
```

- **static operator – (a:Vector2):Vector2**
  描述: 向量取反。
  结果中的每个组件都被取反。

```
// 打印(-1.0, -2.0)
print(-Vector2(1,2);
```

- **static operator/(a:Vector2, d:float):Vector2**
  描述: 用一个数除以一个向量。
  用 d 除以 a 的每个组件。

```
// 使向量缩短一倍；打印(0.0, 1.0)
print(Vector2(1,2)/2.0);
```

- **static operator==(lhs:Vector2, rhs:float):Vector2**
  描述: 如果向量相等返回 true。
  对于非常接近相等的向量，这个也返回 true。
- **static implicit function Vector2(v:Vector3):Vector2**
  描述: 把一个 Vector3 转化成 Vector2。

Vector3 可被隐式的转化为 Vector2(z 被丢弃)。

- static implicit function Vector3(v:Vector2):Vector3

  描述: 把一个 Vector2 转化成 Vector3。

  Vector2 可被隐式的转化为 Vector3(z 被默认设置成 0)。

- static function Scale(a:Vector2, b:Vector2):Vector2

  描述: 两个向量组件相乘

  结果中的每个组件都是 a 中的一个组件乘以 b 中相同的组件。

```
// 打印(2.0, 6.0)
print(Vector2.Scale(Vector2(1,2), Vector2(2,3)));
```

Vector3

结构

表示 3D 的向量和点。

这个结构被用在这个 Unity 中传递 3D 位置和方向，它也包含函数来做普通的向量操作。

变量

- var magnitude : float

  描述: 返回这个向量的长度(只读)。

  向量的长度是(x*x+y*y+z*z)的平方根。

  如果你只需要比较向量的长度，你可以使用 sqrMagnitude(计算长度的平方根是较快的)比较它们的长度平方。

  参见: sqeMagnitude

- var normalized : Vector3

  描述: 返回这个向量，并且 magnitude 为 1(只读)。

  规范化之后，一个向量保持相同的方向但是它的长度为 1.0。

  注意当前的向量不改变并返回一个新的规范化向量，如果你像规范化当前向量，使用 normalized 函数。

  如果这个向量太小以至于不能规范化，将返回一个零向量。

  参见: Normalize 函数

- var sqeMagnitude : float

  描述: 返回这个向量长度的平方(只读)。

  计算长度的平方比 magnitude 更快。通常如果你只比较两个向量的长度你可以只比较它们的平方。

```
// 检测当一个变化比 closeDistance 更接近
// 这个比使用 Vector3 magnitude 更快
var other : Transform;
var closeDistance = 5.0;
function Update() {
    if(other) {
        var sqeLen = (other.position – transform.position).sqrMagnitude;
        // 平方我们需要比较的距离
        if(sqrLen < closeDistance * closeDistance)
print(“The other transform is close to me!”);
    }
}
```

参见: magnitude

- var this[index : int] : float

描述: 分别使用[0],[1],[2]访问 x,y,z 组件。

```
Vector3 p;
p[1] = 5; //与 p.y = 5 相同
```

- var x : float

描述: 该向量的 x 组件。

- var y : float

描述: 该向量的 y 组件。

- var z : float

描述: 该向量的 z 组件。

构造函数

- static function Vector3(x : float, y : float, z : float) : Vector3

描述: 用给定的 x,y,z 组件构建一个新的向量。

- static function Vector3(x : float, y : float) : Vector3

描述: 用给定的 x,y 组件构建一个新的向量并设置 z 为 0。

函数

- function Normalize() : void

描述: 使该向量的 magnitude 为 1。

规范化之后，一个向量保持相同的方向但是它的长度为 1.0。

注意这个函数将改变当前向量。如果你想保持当前向量不改变，使用 normalized 变量。

如果这个向量太小以至于不能规范化，它将被设置为 0.

参见: normalized 变量

- function Scale(scale : Vector3) : void

描述: 用 scale 的组件乘以这个向量中的每个对应组件。

- static function Scale(a : Vecotr3, b : Vector3) : Vecotr3

描述: 两个向量的组件相乘。

结果中的每个组件是 a 中的一个组件乘以 b 中的相对应组件。

```
// 打印(2.0, 6.0, 12.0)
print(Vector3.Scale(Vector3(1,2,3), Vector3(2,3,4)));
```

- function ToString() : string

描述: 返回格式化好的 Vector3 字符串。

类变量

- static var forward : Vector3

描述: Vector3(0, 0, 1)的简写。

```
transform.position += Vector3.forward * Time.deltaTime;
```

- static var one : Vector3

描述: Vector3(1, 1, 1)的简写。

```
transform.position = Vector3.one;
```

- static var right : Vector3

描述: Vector3(1, 0, 0)的简写。

```
transform.position += Vector3.right * Time.deltaTime;
```

- static var up : Vector3

描述: Vector3(0, 1, 0)的简写。

```
transform.position += Vector3.up * Time.deltaTime;
```

◆ static var zero : Vector3

描述: Vector3(0, 0, 0)的简写。

```
transform.position += Vector3.zero;
```

类方法

◆ static function Angle(from : Vector3, to : Vector3) : float

描述: 返回 from 和 to 之间的角度。

```
// 如果这个变化的 z 轴看向 target
// 打印”close”
var target : Transform;
function Update() {
    var targetDir = target.position – transform.position;
    var forward = transform.forward;
    var angle = Vector3.Angle(targetDir, forward);
    if(angle < 5.0)
        print(“close”);
}
```

◆ static function Cross(lhs : Vector3, rhs : Vector3) : float

描述: 两个向量的叉乘。

返回 lhs x rhs。

◆ static function Distance(a : Vector3, b : Vector3) : float

描述: 返回 a 和 b 之间的距离。

Vector3.Distance(a, b)与(a – b).magnitude 相同。

```
var other : Transform;
if(other) {
    var dist = Vector3.Distance(other.position, transform.position);
    print(“Distance to other: ” + dist);
}
```

◆ static function Dot(lhs : Vector3, rhs : Vector3) : float

描述: 两个向量的点乘。

返回 lhs.rhs。

对于 normalized 的向量，如果它们指向相同的方向 Dot 返回 1；如果指向完全相反的方向则返回-1；在另外的情况下返回一个数字(例如，如果向量垂直则 Dot 返回 0)。

对于任意长度的向量 Dot 返回的值是相同的。当两个向量的角度增加的时候，这个值会变大。

```
// 检测是否有其他变换在这个物体之后
var other : Transform;
function Update() {
    if(other) {
        var forward = transform.Transform.Distance(Vector3.forward);
        var toOther = other.position;
        transform.position;
        if(Vector3.Dot(forward, toOther) < 0)
```

```
print("The other transform is behind me!");
    }
}
```

◆ **static function Lerp(from : Vector3, to : Vector3, t : float) : Vector3**

描述: 在两个向量之间找到线性插值。

从 from 到 to 基于 t 插值。

t 被裁剪到（0…1）之间。当 t 为 0 时返回 from，当 t 为 1 时返回 to。当 t = 0.5 时返回 from 和 to 的平均值。

```
// 在一秒内从 from 开始移动到 to 结束
var start : Transform;
var end : Transform;
function Update() {
    transform.position = Vector3.Lerp(start.position, end.position, Time.time);
}
// 像一个弹簧一样跟随 target 位置
var target : Transform;
var smooth = 5.0;
function Update() {
    transform.position = Vector3.Lerp(transform.position, target.position, Time.deltaTime * smooth);
}
```

◆ **static function Max(lhs : Vector3, rhs : Vector3) : Vector3**

描述: 返回由 lhs 和 rhs 中最大组件组成的向量。

```
var a = Vector3(1,2,3);
var b = Vector3(4,3,2);
print(Vector3.Max(a,b)); //打印(4.0, 3.0, 3.0);
```

参见: Min 函数

◆ **static function Min(lhs : Vector3, rhs : Vector3) : Vector3**

描述: 返回由 lhs 和 rhs 中最小组件组成的向量。

```
var a = Vector3(1,2,3);
var b = Vector3(4,3,2);
print(Vector3.Min(a,b)); //打印(1.0, 2.0, 2.0);
```

参见: Max 函数

◆ **static operator != (lhs : Vecotr3, rhs : Vecotr3) : bool**

描述: 如果向量不同返回 true。

非常接近的向量被认为是相等的。

```
var other : Transform;
if(other && transform.position != other.position) {
    print("I'm at the different place than the other transform!");
}
```

◆ **static operator * (a : Vector3, d : float) : Vector3**

描述: 用一个数乘以一个向量。

用 d 乘以 a 的每个组件。

```
print(Vector3(1,2,3) * 2.0); //使向量变为二倍长，打印(2.0, 4.0, 6.0)
```

◆ **static operator * (d : float , a : Vector3) : Vector3**

描述: 用一个数乘以一个向量。

用 d 乘以 a 的每个组件。

```
print(Vector3(1,2,3) * 2.0); //使向量变为二倍长，打印(2.0, 4.0, 6.0)
```

◆ **static operator + (a : Vector3, b : Vector3) : Vector3**

描述: 两个向量相加。

将对应的组件加在一起。

```
// 打印(5.0, 7.0, 9.0)
print(Vector3(1,2,3) + Vector3(4,5,6));
```

◆ **static operator - (a : Vector3, b : Vector3) : Vector3**

描述: 两个向量相减。

将 a 与 b 对应的组件相减。

```
// 打印(-5.0, -3.0, -1.0)
print(Vector3(1,2,3) - Vector3(6,5,4));
```

◆ **static operator - (a : Vector3) : Vector3**

描述: 向量取反。

将 a 中的每个组件都取反。

```
// 打印(-1.0, -2.0, -3.0)
print(Vector3(1,2,3));
```

◆ **static operator / (a : Vector3, d : float) : Vector3**

描述: 用一个数除以一个向量。

用 d 除以 a 的每个组件。

```
// 使向量缩短一倍，打印(0.5, 1.0, 1.5)
print(Vector3(1,2,3) / 2.0);
```

◆ **static operator == (lhs : Vector3, rhs : Vector3) : bool**

描述: 如果向量相等返回 true。

对于非常接近相等的向量，这个也返回真。

```
var other : Transform;
if(other && transform.position == other.position) {
    print("I'm at the same place as the other transform");
}
```

◆ **static function OrthoNormalize(ref normal : Vector3, ref tangent : Vector3) : void**

描述: 使两个向量规范化并互相正交。

规范化 normal；规范化 tangent 并确保它正交于 normal（就是说，它们之间的角度是 90 度）。

参见: Normalize 函数

◆ **static function OrthoNormalize(ref normal : Vector3, ref tangent : Vector3, ref binormal : Vector3) : void**

描述: 使两个向量规范化并互相正交。

规范化 normal；规范化 tangent 并确保它与 normal。规范化 binormal 并确保它与

normal 和 tangent 正交。

参见: Normalize 函数

- **static function Project(vector : Vector3 , onNormal : Vector3) : Vector3**

  描述: 投影一个向量到另一个向量。

  返回投影到 onNormal 上的 vector。如果 onNormal 接近零，返回零矩阵。

- **static function Reflect(inDirection : Vector3, inNormal : Vector3) : Vector3**

  描述: 沿着法线反射这个向量。

  返回的值 inDirection 从以 inNormal 为法线的表面反射。

```
var originalObject : Transform;
var reflectedObject : Transform;
function Update() {
    // 使反射物体与凡是物体沿着世界的 z 轴镜像反射
    reflectedObject.position = Vector3.Reflect(originalObject.position, Vector3.right);
}
```

- **static function RotateTowards(from : Vector3, to : Vector3, maxRadiansDelta : float, maxMagnitudeDelta : float) : Vector3**

  描述: 旋转一个向量 from 到 to。

  该向量将被在一个弧线上旋转而不是线形插值。本质上与 Vector3.Slerp 相同，但是这个函数将确保角速度和长度的改变不会超过 maxRadiansDelta 和 maxMagnitudeDelta。

- **function Scale(scale : Vector3) : void**

  描述: 用 sacle 的组件乘以这个向量中的每个对应组件。

- **static function Scale(a : Vector3, b : Vector3) : Vector3**

  描述: 两个向量组件相乘。

  结果中的每个组件是 a 中的一个组件乘以 b 中相对应的组件。

```
// 打印(2.0, 6.0, 12.0)
print(Vector3.Scale(Vector3(1,2,3), Vector3(2,3,4)));
```

- **static function Slerp(from : Vector3, to : Vector3, t : float) : Vector3**

  描述: 在两个向量之间球形插值。

  从 from 到 to 基于 t 插值，返回向量的 magnitude 将在 from 和 to 的长度之间插值。

  t 被裁剪到[0…1]之间。

```
// 在 sunrise 和 sunset 之间以弧线变换一个位置
var sunrise : Transform;
var sunset : Transform;
function Update() {
    // 弧线的中心
    var center = (sunrise.position – sunset.position) * 0.5;
    // 向下移动一点中心使该弧线垂直
    center.y -= 1;
    //相对于中心插值这个弧线
    var riseRelCenter = sunrise.positioni – center;
    var setRelCenter = sunset.position – center;
    transform.position = Vector3.Slerp(riseRelCenter, setRelCenter. Time.time);
    transform.position += center;
}
```

参见: Lerp 函数

Vector4

结构

表示四维向量。

这个结构被用在一些地方来表示四个组件的向量（例如，网格切线，shader 的参数）。在其他的一些地方使用 Vector5。

变量

- **var magnitude : float**

  描述: 返回这个向量的长度（只读）。

  向量的长度是(x*x+y*y+z*z+w*w)的平方根。

  如果你只需要比较向量的长度，你可以使用 sqrMagnitude（计算长度的平方是比较快的）比较它们的长度平方。

  参见: sqrMagnitude

- **var normalized : Vector4**

  描述: 返回这个向量，并且 magnitude 为 1（只读）。

  注意当前的向量不改变并返回一个新的规范化的向量。如果你像规范化当前向量，使用 Normalize 函数。

  如果这个向量太小以至于不能规范化，将返回一个零向量。

  参见: Normalize 函数。

- **var sqrMagnitude : float**

  描述: 返回这个向量的长度的平方（只读）。

  计算长度的平方比 magnitude 更快。

  参见: magnitude

- **var this[index : int] : float**

  描述: 分别使用[0],[1],[2],[3]访问 x,y,z,w 组件。

```
Vector4 p;
p[3] = 5;// 与 p.w = 5 相同
```

- **var w : float**

  描述: 该向量的 w 组件。

- **var x : float**

  描述: 该向量的 x 组件。

- **var y : float**

  描述: 该向量的 y 组件。

- **var z : float**

  描述: 该向量的 z 组件。

构造函数

- **static function Vector4 (x : float, y : float, z : float, w : float) : Vector4**

  描述: 用给定的 x,y,z,w 组件构建一个新的向量。

- **static function Vector4 (x : float, y : float, z : float) : Vector4**

  描述: 用给定的 x,y,z 组件构建一个新的向量并设置 w 为 0。

- **static function Vector4 (x : float, y : float) : Vector4**

  描述: 用给定的 x,y 组件构建一个新的向量并设置 z 和 w 为 0。

函数

- **function Normalize() : void**

  描述: 使该向量的 magnitude 值为 1。

  注意这个函数将改变当前向量。如果你想保持当前向量不改变，使用 normalized 变量。

  如果这个向量太小以至于不能规范化，它将被设置为 0。

  参见: normalized 变量

- **function Scale(scale : Vector4) : void**

  描述: 用 scale 的组件乘以这个向量中的每个对应组件。

- **static function Scale(a : Vector4, b : Vector4) : Vector4**

  描述: 两个向量的组件相乘。

  结果中的每个组件是 a 中的一个组件乘以 b 中相同的组件。

```
// 打印(2.0, 6.0, 12.0, 12.0)
print(Vector4.Scale(Vector4(1,2,3,4), Vector4(2,3,4,5)));
```

- **function ToString() : String**

  描述: 返回格式化好的 vector 字符串

类变量

- **static var one : Vector4**

  描述: Vector4(1,1,1,1)的简写。

- **static var zero : Vector4**

  描述: Vector4(0,0,0,0)的简写。

类方法

- **static function Distance(a : Vector4, b : Vector4) : float**

  描述: 返回 a 和 b 之间的距离。

  Vector4 Distance(a,b)和(a-b).magnitude 相同。

- **static function Dot(lhsVector4, rhsVector4) : float**

  描述: 两个向量的点乘。

  返回 lhs.rhs。

- **static function Lerp(from : Vector4, to : Vector4, t : float) : Vector4**

  描述: 在两个向量之间线形插值。

  从 from 到 to 基于 t 插值。

  t 被裁剪到[0…1]之间，当 t 为 0 时返回 from，当 t 为 1 时返回 to，当 t = 0.5 时返回 from 与 to 的平均值。

- **static operator != (lhs : Vector4, rhs : Vector4) : bool**

  描述: 如果向量不同返回真。

  非常接近的向量被认为是相等的。

- **static operator * (a : Vector4, d : float) : Vector4**

  描述: 用一个数乘以一个向量。

  用 d 乘以 a 的每个组件。

```
// 使向量变为二倍长，打印(2.0, 4.0, 6.0, 8.0)
print(Vector4(1,2,3,4) * 2.0);
```

- **static operator * (d : float, a : Vector4) : Vector4**

  描述: 用一个数乘以一个向量。

  用 d 乘以 a 的每个组件。

```
// 使向量变为二倍长，打印(2.0, 4.0, 6.0, 8.0)
```

```
print(2.0 * Vector4(1,2,3,4));
```

◆ **static operator + (a : Vector4, b : Vector4) : Vector4**

描述: 两个向量相加。

将对应的组件加在一起。

```
// 打印(5.0, 7.0, 9.0, 11.0)
print(Vector4(1,2,3,4) + Vector4(4,5,6,7));
```

◆ **static operator - (a : Vector4, b : Vector4) : Vector4**

描述: 两个向量相减。

从 a 中减去 b 中每个对应的组件。

```
// 打印(-5.0, -3.0, -1.0, 1.0)
print(Vector4(1,2,3,4) + Vector4(6,5,4,3));
```

◆ **static operator - (a : Vector4) : Vector4**

描述:向量取反。

a 中的每个组件都被取反。

```
// 打印(-4.0, -3.0, -2.0, -1.0)
print(-Vector4(4,3,2,1));
```

◆ **static operator / (a : Vector4, d : float) : Vector4**

描述: 用一个数除以一个向量。

从 d 除以 a 中每个组件。

```
// 使向量缩短一半，打印(0.5, 1.0, 1.5, 2.0)
print(Vector4(1,2,3,4) / 2.0);
```

◆ **static operator == (lhs : Vector4, rhs : Vector4) : bool**

描述: 如果向量相等返回 true。

对于非常接近相等的向量，这个也返回 true。

◆ **static implicit function Vector3(v : Vector4) : Vector3**

描述: 把一个 Vector4 转换为一个 Vector3。

Vector4 可以被隐式转换成 Vector3(w 被丢弃)。

```
function Start() {
    // shader 总是 Vector4。但是这个值被转化成一个 Vector3。
    var value : Vector3 = renderer.material.GetVector("_SomeVariable");
}
```

◆ **static implicit function Vector4(v : Vector3) : Vector4**

描述: 把一个 Vector3 转换为一个 Vector4。

Vector3 可以被隐式转换成 Vector4(w 被设置为 0)。

```
function Start() {
    // shader 总是 Vector4。这个值从一个 Vector3 转换为一个 Vector4。
    var value : Vector3 = Vector3.one;
    renderer.material.SetVector("_SomeVariable", value);
}
```

◆ **static function Project(a : Vector4, b : Vector4) : Vector4**

描述: 投影一个向量到另一个向量。

返回投影到 b 的 a。

◆ **static Scale(scale : Vector4) : void**

描述: 用 scale 的组件乘以这个向量中的每个对应组件。

◆ static function Scale(a : Vector4, b : Vector4) : Vector4

描述: 两个向量的组件相乘。

结果中每个组件都是 a 中的一个组件乘以 b 中的对应组件。

```
// 打印(2.0, 6.0, 12.0 , 20.0)
print(Vector4.Scale(Vector4(1,2,3,4), Vector4(2,3,4,5)));
```

WWWFrom

类

辅助类。用来生成表单数据并使用 WWW 类传递到 web 服务器。

```
// 获取一个截屏并上传到 CGI 脚本
// 该 CGI 脚本必须能处理表单上传
var screenshotURL = "http://www.my-site.com/cgi-bin/screenshot.pl";
// 截屏
function Start() {
    UploadPNG();
}
function UploadPNG () {
    yield WaitForEndOfFrame(); // 我们应该只在所有渲染完成后读取屏幕
    var width = Screen.width;
    var height = Screen.height;
    var tex = new Texture2D(width, height, TextureFormatRGB24, false);    // 创建屏幕大小的纹理，RGB24 格式
    //读取屏幕内存到纹理
    tex.ReadPixels(Rect(0, 0, width, height), 0, 0);
    tex.Apply();
    // 编码纹理为 PNG
    var bytes = tex.EncodeToPNG();
    Destroy(tex);
    // 创建一个 Web 表单
    var form = new WWWForm();
    form.AddField("frameCount", Time.frameCount.ToString());
    form.AddBinaryData("fileUpload", bytes, "screenshot.png", "image.png");
    // 上传到一个 CGI 脚本
    var w = WWW(screenShotURL, form);
    yield w;
    if(w.error != null) {
        print(w.error);
    } else {
        print("Finished Uploading Screenshot");
}
 }
```

这里是一个简单的 Perl 脚本用处理存贮在 SQL 数据库中的高分表

```
// 这个例子假设玩家已经输入了他的名称到一个 name 变量中并且 score 包含玩家的当前分数
var highscore_url = "http://www.my-site.com/highscore.pl";
```

```
function Start() {
    // 创建一个表单来发送高分数据到服务器
    var form = new WWWForm();
    // 假设 perl 脚本为不同的游戏管理高分
    form.AddField("game", "MyGameName");
    // 玩家提交的名称
    form.AddField("playerName", name);
    // 分数
    form.AddField("score", score);
    //创建一个下载对象
    var download = new WWW(highscore_url, form);
    //等待直到下载完成
    yield download;
    if(download.error) {
        print("Error downloading " + download.error);
    } else {
        // 显示高分
    }
}
```

这里是一个单间的 Perl 脚本用来处理存贮在 SQL 数据库中的高分表

```
#/usr/bin/perl
#SQL 数据库需要有一个称为 highscores 的表
# 看起来像这样
#CREATE TABLE highscores (
#game varchar(255) NOT NULL,
#player varchar(255) NOT NULL,
#score integer NOT NULL
# );
use strict;
use CGI;
use DB1;
# 读取表单数据
my $cgi = new CGI;
# 来自高分脚本的结果将是 in 纯文本格式
print $cgi -> header("text/plain");
my $game = $cgi -> param('game');
my $playeName = $cgi -> param('playerName');
my $score = $cgi -> param('score');
exit 0 unless $game; #这个参数被请求链接到一个数据库
my $dbh = DB1 -> connect('DB1.mysql.databasename', 'username', 'password') || die
"Could not connect to database: $DB1::errstr";
# 如果插入玩家分数 if
if($playerName && $score) {
    $dbh -> do("insert into highscores(game, player, score) values(?,?,?)", undef, $game,
```

$playerName, $score);

}

取回高分

my $sth = $dbh -> prepare("SELECT player, score FROM highscores WHERE game=? ORDER BY score desc LIMIT 10");

#dbh -> execute($game);

while(my $r = $sth -> fetchrow_arrayref) {

print join('.', @$r), "\n"

}

变量

◆ var data : byte[]

描述: （只读）在发送表单的时候原始数据作为 POST 请求被发送。

通常，你只需要直接将 WWWForm 对象传递给 WWW 构造函数，但是如果你像改变发送到 web 服务器的头，你将需要这个变量。

参见: headers 变量

var form = new WWWForm();

form.AddField("name", "value");

var headers = form.headers;

var rawData = form.data;

// 给请求添加一个自定义的头，在这里用一个简单的授权来访问密码保护的资源

header["Authorization"]="Basic" +System.Convert.ToBase64String(System.Text.Encoding.ASCII.GetBytes("username:password));

// 用自定义的头传递一个请求到 URL

var www= new WWW(url, rawData, headers);

yield www;

// 这里处理 WWW 请求结果…

◆ var headers : Hashtable

描述: （只读）为使用 WWW 类传递的表单返回一个正确的请求头。

这个域只包含一个头，/"Content-Type"/，它被设置为正确的 mine 类型。"application/x-www-form-urlencoded"用于普通的表单，"multipart/form-data"用于使用 AddBinaryData 添加数据的表单。

var form = new WWWForm();

form.AddField("name", "value");

var headers = form.headers;

var rawData = form.data;

// 给请求添加一个自定义的头，在这里用一个简单的授权来访问密码保护的资源

header["Authorization"]="Basic" +System.Convert.ToBase64String(System.Text.Encoding.ASCII.GetBytes("username:password));

// 用自定义的头传递一个请求到 URL

var www= new WWW(url, rawData, headers);

yield www;

// 这里处理 WWW 请求结果…

构造函数

◆ static function WWWForm() : WWWForm

描述: 创建一个空的 WWWForm 对象。

使用 AddField 和 AddBinaryData 方法向表单中插入数据。

参见: WWW 类

函数

◆ function AddBinaryData(fieldname : string, contents : byte[], filename : string = null, mimeType : string = null) : void

描述: 添加二进制数据到表单。

使用这个函数来上传文件和图片到 web 服务器，注意数据从字节数组中读取而不是从一个文件中读取。fileName 参数用来告诉服务器用什么文件名来保存上传的文件。

如果 mimeType 没有给出，并且数据的前 8 字节与 PNG 格式头相同，然后数据用”image/png”mimetype 发送，否则它将用”application/octet-stream”mimetype 发送。

◆ function AddField(filename : string, value : string, e : Encoding = System.Text.Encoding.UTF8) : void

描述: 添加一个简单的域到表单。

用给的字符串值添加域 fileName。

◆ function AddField(filedName : string, i : int) : void

描述: 添加一个简单的域到表单。

用给定的 X 形值添加域 filedName。一个简单的方法是调用 AddField(fieldname, i.ToString)。

WWW

类，集成自 IDisposable

简单地访问 web 页。

这个是一个小的工具模块可以用来取回 URL 的内容。

通过调用 WWW(url)在后台开始一个下载，它将返回一个 WWW 物体。

你可以检查 isDone 属性来查看下载是否完成，或者 yield 下载物体来自动等待，直到它被下载完成（不会影响游戏的其余部分）。

如果你像从 web 服务器上获取一些数据例如高分列表或者调用主页，可以使用这个，也有一些功能可以使用从 web 上下载的图片来创建一个纹理，或者下载或加载新的 web 播放器数据文件。

WWW 类可以用来发送 GET 和 POST 请求到服务器，WWW 类默认使用 GET，如果提供一个 postData 参数就使用 POST。

参见: WWWForm 为 postData 参数构建可用的表单数据。

```
// 从时代广场上的外部”Friday’s” web 摄像头获取最新的数据
var url = “http://images.earthcam.com/ec_metros/ourcams/fridays.jpg”;
function Start() {
    // 开始下载给定的 URL
    var www : WWW = new WWW(url);
    // 等待下载完成
    yield www;
    //赋值纹理
    renderer.material.mainTexture = www.texture;
}
```

变量

◆ var assetBundle : AssetBundle

描述: 从工程文件中下载一个可以包含任意类型资源的 AssetBundle。

```
function Start() {
    var www = new WWW("http://myserver/myBundle.unity3d");
    yield www;
    // 获取制定的主资源并实例化它
    Instantiate(www.assetBundle.mainAsset);
}
```

参见: AssetBundle 类。

◆ var audioClip : AudioClip

描述: 从下载的数据生成一个 AudioClip（只读）。

该数据必须是一个 Ogg Vorbis 格式的音频剪辑。

即使音频没有完全下载完成，这个也立即返回，允许你开始播放已经下载完成的部分。

```
var url : String;
function Start() {
    www = new WWW(url);
    audio.clip = www.audioClip;
}
function Update() {
    if(!audio.isPlaying && audio.clip.isReadyToPlay) {
        audio.Play();
    }
}
```

◆ var bytes : byte[]

描述: 将取回的 web 页内容作为一个字节数组返回（只读）。

如果物体还没有完成数据的下载，它将返回一个空字节数组。使用 isDone 或者 yield 来查看数据是否可用。

参见: data 属性

◆ var data : string

描述: 将取回的 web 页内容作为一个字符串返回（只读）。

如果物体还没有完成数据的下载，它将返回一个空字符串。使用 isDone 或者 yield 来查看数据是否可用。

这个函数期望网页内容是 UTF-8 或者 ASCII 字符集。对于其他字符或者二进制数据返回的字符串也许是不正确的。在这些情况下，使用 bytes 属性来获取原始字节数组。

参见: bytes 属性

◆ var error : string

描述: 如果下载的时候出现了一个错误，返回错误信息（只读）。

如果没有错误，error 将返回 null。

如果物体没有下载完成，它将被阻止直到下载完成。使用 isDone 或 yield 来查看数据是否可用。

```
// 用一个无效的 URL 获取一个纹理
var url = "invalid_url";
function Start() {
    // 开始下载给定的 URL
```

```
    var www : WWW = new WWW(url);
    // 等待下载完成
    yield www;
    // 打印错误的控制台
    if(www.error!=null) {
        Debug.Log(www.error);
    }
    //赋值纹理
    renderer.material.mainTexture = www.texture;
}
```

◆ var isDone : bool

描述: 下载是否完成（只读）。

如果你试图访问任何 isDone 为 false 的数据，程序将被阻止知道下载完成。

var movie : MovieTexture

描述: 从下载的数据生成一个 MovieTexture（只读）。

数据必须为一个 Ogg Theora 格式视频。

即使视频完全没有下载完成，这个也立即返回，允许你开始播放已经下载完成的部分。

```
var url = “http://www.unity3d.com/webplayers/Movie/sample.ogg”;
function Start() {
    // 开始下载
    var www = WWW(url);
    // 确保视频在开始播放前已经准备好
    var movieTexture = www.movie;
    while(!movieTexture.isReadyToPlay)
        yield;
    //初始化 GUI 纹理为 1:1 解析度并居中
    guiTexture.texture = movieTexture;
    transform.localScale = Vector3(0,0,0);
    transform.position = Vector3(0.5, 0.5, 0);
    guiTexture.pixelInset.xMin = -movieTexture.width/2;
    guiTexture.pixelInset.xMax = movieTexture. width /2;
    guiTexture.pixelInset.yMin = -movieTexture.height/2;
    guiTexture.pixelInset.yMax = movieTexture. height /2;
    // 赋值剪辑到音频源
    // 与音频同步播放
    audio.clip = movieTexture.audioClip;
    // 播放视频和音频
    movieTexture.Play();
    audio.Play();
}
// 确保我们有 GUI 纹理和音频源
@script RequireComponent(GUITexture)
@script RequireComponent(AudioSource)
```

◆ var oggVorbis : AudioClip

```
        描述: 加载 Ogg Vorbis 文件到音频剪辑。
        如果流没有被完全下载，将返回 null。使用 isDone 或者 yield 来查看数据是否可用。
        参见: AudioClip, AudioSource
    var path = “http://ua301106.us.archive.org/2/items/abird2005-02-10t02.ogg”;
    function Start() {
  var download = new WWW(path); // 开始下载
  yield download; // 等待下载完成
    var clip : AudioClip = download.oggVorbis; // 创建 ogg vorbis 文件
    // 播放它
    if(clip!=null) {
        audio.clip = clip;
        audio.Play();
    // 处理错误
  } else {
            Debug.Log(“Ogg vorbis download failed(Incorrect link?)”);
  }
  }
    @script RequireComponent(AudioSource)
    // 一个通用的流式音乐播放器
    // 成功地下载音乐，然后随机地播放它们
    var downloadPath : String[] =
[“http:/ia301106.us.archive.org/2/items/abird2005-02-10t02.ogg”];
    private var downloadedClips : AudioClip[];
    private var playedSongs = new Array();
    function Start () {
        downloadedClips = new AudioClip(downloadPath.Length);
        DownloadAll();
        PlaySongs();
    }
    function DownloadAll() {
        for(var i=0;i<downloadPath.length;i++) {
            var path = downloadPath[i];
            var download = new WWW(path);
            yield download;
            downloadedClips[i] = download.oggVorbis;
            if(downloadedClips[i] == null)
  Debug.Log(“Failed audio download” + path);
        }
    }
    function PickRandomSong() : AudioClip
    {
        var possibleSongs = Array();
        // 构建一个下载完成的音乐列表
        for(var i=0;i<downloadedClips.length;i++) {
```

```
            if(downloadedClips[i] != null)
possibleSongs.Add[i];
        }
        // 还没有音乐被下载
        if(possibleSongs.length == 0)
            return null;
        // 我们播放了所有音乐，现在从任何音乐中选择
        if(possibleSongs.length == playedSongs.length)
            playerSongs.Clear();
        // 从列表中已出已经播放完成的音乐
        for(i=0;i<playedSongs.length;i++)
            possibleSongs.Remove(playedSongs[i]);
        // 获取一个随即的音乐
        if(possibleSongs.length != 0) {
            var index : int = possibleSongs[Random.Range(0,possibleSongs.length)];
            playedSongs.Add(index);
            return downloadedClips[index];
        } else return null;
}
 function PlaySongs() {
        while(true) {
            var clip : AudioClip = PickRandomSongs();
            if(clip != null) {
audio.clip = clip;
audio.Play();
yield WaitFormSecond(clip.length);
            }
            yield;
        }
 }
@script RequireComponent(AudioSource)
```

◆ **var texture : Texture2D**

描述: 从下载的数据生成一个 **Texture2D**（只读）。

数据必须是 **JPG** 或者 **PNG** 图片格式。如果数据图像不可用，产生的纹理将是一个带问号的纹理。建议使用 **2** 的幂次大小的图片；任意大小的也可以工作但是加载的比较慢并会占用较多的内存。每个纹理属性的调用都会分配一个新的 **Texture2D**。如果你连续的下载纹理，必须使用 **LoadImageInfoTexture** 或销毁前面创建的纹理。

对于 **PNG** 文件，如果包含伽马修正，伽马修正将被应用到该纹理，显示的伽马修正假定为 **2.0**，如果文件没有包含伽马信息，将不会执行任何颜色修正。

如果物体还没有完成数据的下载，它将返回一个虚构的图片，使用 **isDone** 或者 **yield** 来查看数据是否可以用。

```
 // 从时代广场上的外部”Friday’s” web 摄像头获取最新的数据
 var url = “http://images.earthcam.com/ec_metros/ourcams/fridays.jpg”;
 function Start() {
```

```
// 开始下载给定的 URL
var www : WWW = new WWW(url);
// 等待下载完成
yield www;
// 赋值纹理
renderer.material.mainTexture = www. texture;
}
```

- **var uploadProgress : float**

  描述: 上传了多少（只读）。

  这是 0 和 1 之间的值: 0 表示没有发送任何数据，1 表示上传完成。

  uploadProgress 目前没有完全在 web 播放器中实现，如果在 web 播放器中使用它将在上传的时候返回 0.5，上传完成后返回 1.0。

  因为所有到服务器的数据发送在接收数据以前完成，所以当 progress 大于 0 时，uploadProgress 将总是 1.0。

- **var url : string**
- **static function WWW(url : string, postData : byte[], headers : Hashtable) : WWW**

  参数:

  url 下载的 URL

  postData 传递到该 url 的字节数组数据

  headers　　一个自定义头部的 hash 表，随着请求发送

  返回: WWW，一个新的 WWW 物体。当它被下载后，结果可以从返回的物体中间取回。

  描述: 用给定的 URL 创建一个 WWW 请求

  这个函数创建并发送一个 POST 请求，该请求带有包含在 postData 中的 post 数据和 header 哈希表提供的自定义请求头。流将自动开始下载。如果你需要以自定义的格式传递原始数据到服务器，或者如果你需要提供自定义的请求头。

  流创建之后你不得不等待直到它完成，然后你可以访问下载的数据。作为一个快捷的方法，流可以被 yield，这样你能够非常容易的告诉 Unity 等待下载完成。

函数

- **function Dispose () : void**

  描述:

- **function LoadImageIntoTexture(tex : Texture2D) : void**

  参数:

  tex　　一个已存在的纹理物体可以被这个图像数据覆盖。

  描述: 用来自下载数据的图像替换已存在的 Texture2D 内容。

  数据必须是 JPG 或 PNG 图片格式。如果数据图像不可用，产生的纹理将是一个带问号的纹理。建议使用 2 的幂次大小的图片；任意大小的也可以工作但是加载的比较慢并会占用较多的内存。

  对于 PNG 文件，如果包含伽马修正，伽马修正将被应用到该纹理。显示的伽马修正假定为 2.0。如果文件没有包含伽马信息，将不会执行任何颜色修正。

  如果数据没有下载完成，纹理将保持不变。使用 isDone 或 yield 来查看数据是否可用。

```
// 获取时代广场最新的外部”Friday’s”web 摄像头数据
var url = “http://images.earthcam.com/ec_metros/ourcams/fridays.jpg”;
```

```
function Start() {
    renderer.material.mainTexture = new Texture2D(512,512);
    while(true) {
        // 开始下载给定的 URL
        var www = new WWW(url);

        // 等待直到下载完成
        yield www;

        // 赋值下载的图片到物体的主纹理
        www.LoadImageIntoTexture(renderer.material.mainTexture);
    }
}
```

◆ **function LoadUnityWeb() : void**

描述: 加载新的 web 播放器数据文件。

加载的 unity3d 文件的第一个关卡将自动被加载，所有来自前一个.unity3d 文件的物体、脚本和静态变量将被卸载。你可以使用 PlayerPrefab 类在两个绘画间移动信息。

这个函数只能在 web 播放器中使用。

如果物体还没有完全下载，unity3d 文件将不会被加载。使用 isDone 或 yield 来查看数据是否可用。

```
function Start() {
    // 开始缓存数据文件
    var stream = new WWW("http://www.unity3d.com/webplayers/Lightning/lightning.unity3d");
    // Yield 直到流完成
    yield stream;
    // 加载它!
    stream.LoadUnityWeb();
}
// 下载一个.unity3d 文件并在 GUI 纹理中显示速度，你需要确保 GUI 纹理被设置为具有一个 pixelInset
function Update() {
    // 保存原始的 pixelInset 并修改它
    var originalPixelRect = guiTexture.pixelInset;
    // 通过缩放 GUI 纹理来更新进度栏，直到到达末端
    var stream = new WWW("http://www.unity3d.com/webplayers/Lightning/lightning.unity3d");
    while(!stream.isDone) {
        guiTexture.pixelInset.xMax = originalPixelRect.xMin + stream.progress * originalPixelRect.width;
        yield;
    }
    // 在加载前更新最后一次
    guiTexture.pixelInset.xMax = originalPixelRect.xMax;
```

stream.LoadUnityWeb();
}

类方法

◆ static function EscapeURL(s : string, e : Encoding = System.Text.Encoding.UTF8) : string

参数:

s 用来编码的字符串

e 编码使用的字符集

返回: string。 一个新的字符串，所有非法的字符都使用%xx 替换，这里 xx 是十六进制编码。

描述: 编码字符串为 URL 友好的格式。

用正确的 URL 编码替换 s 中的非法字符，在构建 web 页参数时使用它。

参见: WWW.UnEscapeURL

```
// 这将打印字符串”Testing%201%2C2%2C3”到控制台
print(WWW.EscapeURL(“Testing 1,2,3));
```

◆ static function UnEscapeURL(s : string, e : Encoding = System.Text.Encoding.UTF8) : string

参数:

s 一个编码过的字符串

e 编码使用的字符集

返回: string – 一个新的字符串，其中所有的%xx 都将使用对应的字符来替换。

描述: 从 URL 友好的格式解码一个字符串。

是 WWW.EscapeURL 的反响。

参见: WWW.EscapeURL

```
// 这将打印字符串”Testing 1,2,3”到控制台
print(WWW.UnEscapeURL(“Testing%201%2C2%2C3”));
```

WheelFrictionCurve

结构

WheelFrictionCurve 被 WheelCollider 使用来描述:轮胎的摩擦力属性。

该曲线使用轮胎的滑动作为输入并输出一个力。该曲线近似由两端曲线构成。第一段从(0,0)到(extremumSlip, extremumValue)到(asymptoteSlip, asymptoteValue)，这里曲线的切线再次为零。

车轮碰撞器使用不同于物理引擎的一个基于滑动的摩擦力模型来计算摩擦力。它分割整个摩擦力为“向前”组件（在滚动的方向上，并负责加速和制动）和“侧滑”组件（垂直于滚动方向，负责保持车辆的方向）。轮胎的摩擦力在这些方向上分别使用 WheelCollider.forwardFrictioni 和 WheelCollider.sidewaysFriction 描述:在两个方向上它首先决定轮胎滑动了多少（橡胶和路面之间速度的不同），然后这个滑动值用来找到在接触点上轮胎受到的力。

真是的轮胎属性是较慢的滑动将获得较高的力因为橡胶通过拉伸补偿滑动，然后当滑动变的较高时，这个力被减小因为轮胎开始滑动或旋转。因此轮胎摩擦力曲线具有上图所示的形状。

因为轮胎的摩擦力是分别计算的，地面的 PhysicMaterial 不会影响车轮。通过改变车轮所碰到的 forwardFriction 和 sidewayFriction 来模拟不同的路面材质。参见:WheelCollider.GetGroudHit, WheelHit。

参见: WheelCollider, WheelCollider.forwardFriction, WheelCollider.sidewayFriction

变量

- var asymptoteSlip : float
  描述: 滑动渐进线点（默认为 2）。
- var asymptoteValue : float
  描述: 渐进线滑动上的力（默认为 10000）。
- var extremumSlip : float
  描述: 滑动极值点（默认为 1）。
- var extremumValue : float
  描述: 滑动极值的力（默认为 20000）。
- var stiffness : float
  描述: extremumValue 和 asymptoteValue 的倍数（默认为 1）。
  改变摩擦系数，设置这个为零将完全禁用车轮的所有摩擦。
  通常修改 stiffness 来模拟各种地面材质（例如，玻璃具有较低的摩擦系数）。

参见: WheelCollider.GetGroudHit

```
// 当附加到 WheelCollider 时，基于地面的材质的静态摩擦力修改轮胎的摩擦力
function FixedUpdate() {
    var hit : WheelHit;
    var wheel : WheelCollider = GetComponent(WheelCollider);
    if(wheel.GetGroundHit(hit)) {
        wheel.forwardFriction.siffness = hit.collider.material.staticFriction;
        wheel.sidewayFriction.siffness = hit.collider.material.staticFriction;
    }
}
```

WheelHit

结构

车轮的接触信息，由 WheelCollider 得到。

用于 WheelCollider 的摩擦力是独立于物理系统计算的，使用基于轮胎摩擦力模型的滑动。这允许更加真实的行为，而且使车轮忽略标准的 PhysicMaterial 设置。

模拟不同地面材质的方法是查询 WheelCollider 获取它的碰撞信息（参考 WheelCollider.GetGroundHit）。通常你可以获取车轮碰到的其他 collider，并基于地面 material 修改车轮的 forwardFriction 和 sidewayFriction。

变量

- var collider : Collider
  描述: 车轮碰撞到的 Collider。

```
// 当附加到 WheelCollider 时，基于地面的材质的静态摩擦力修改轮胎的摩擦力
function FixedUpdate() {
    var hit : WheelHit;
    var wheel : WheelCollider = GetComponent(WheelCollider);
    if(wheel.GetGroundHit(hit)) {
        wheel.forwardFriction.siffness = hit.collider.material.staticFriction;
        wheel.sidewayFriction.siffness = hit.collider.material.staticFriction;
    }
}
```

- var force : float

描述: 应用到接触点的力的大小。

- var forwardDir : Vector3

  描述: 车轮指向的方向。

- var forwardSlip : float

  描述: 在滚动方向上的滑动，加速滑动为负，制动滑动为正。

```
// 当轮胎滑动时打印”Braking Slip!”
function FixedUpdate() {
    var hit : WheelHit;
    var wheel : WheelCollider = GetComponent(WheelCollider);
    if(wheel.GetGroundHit(hit)) {
        if(hit.forwardSlip > 0.5)
print(“braking slip!”);
    }
}
```

- var normal : Vector3

  描述: 接触点的法线。

- var point : Vector3

  描述: 轮胎和地面的接触点。

- var sidewaysDir : Vector3

  描述: 车轮的侧向。

- var sidewaysSlip : float

  描述: 在侧面方向上的滑动。

YieldInstruction

类

用于所有 yield 指令的基类。

参考 WaitForSeconds，WaitForFixedUpdate，Coroutine 和 MonoBehaviour.StartCoroutine 获取更多信息。

Coroutine

类，继承自 YieldInstruction

MonoBehaviour.StartCoroutine 返回一个 Coroutine。

一个 Coroutine 是一个函数，这个函数可以被暂停(yield)直到给定的 YieldInstruction 完成。

```
// 打印”Starting 0.0”
// 打印”WaitAndPrint 5.0”
// 打印”Done 5.0”
print(“Starting “ + Time.time);
// WaitAndPrint 作为一个 Coroutine 开始
yield WaitAndPrint();
print(“Done ” + Time.time);
function WaitAndPrint() {
    // 暂停执行 5 秒
    yield WaitForSeconds(5);
    print(“WaitAndPrint “ + Time.time);
}
```

WaitForEndOfFrame

类，继承自 YieldInstruction

等待直到所有的相机和 GUI 被渲染完成，并在该帧显示在屏幕上之前。

你可以用它来读取显示到纹理中，编码它为一个图片（参考 Texture2D.ReadPixels, Texture2D.EncodeToPNG）并发送它。

```
yield new WaitForEndOfFrame();
// 存储截屏为 PNG 文件
import System.IO;
// 立即截屏
function Start() {
    UploadPNG();
}
function UploadPNG() {
    // 在渲染完成后读取屏幕缓存
    yield WatiForEndOfFrame();
    // 创建一个屏幕大小的纹理，RGB24 格式
    var width = Screen.width;
    var height = Screen.height;
    var tex = new Texture2D(width, height, TextureFormat.RGB24, false);
    // 读取屏幕内容到纹理
    tex.ReadPixels(Rect(0, 0, width, height), 0 ,0);
    tex.Apply();
    // 编码纹理为 PNG 文件
    var bytes = tex.EncodeToPNG();
    Destroy(tex);
    // 出于测试目的，也在工程文件夹中写一个文件
    // File.WriteAllBytes(Application.dataPath – "/…/SavedScreen.png", bytes);
    // 创建一个 Web 表单
    var form = new WWWForm();
    form.AddField("frameCount", Time.frameCount.ToString());
    form.AddBinaryData("fileUpload", bytes);
    //上传到一个 CGI 脚本
    var w = WWW("http://localhost/cgi-bin/cnv.cgi?post", form);
    yield w;
    if(w.error != null) {
        print(w.error);
    }
    else {
        print(Finished Uploading Screenshot");
    }
    // 在游戏视图中显示 alpha 通道的内容，需要 UnityPro 因为这个脚本使用了 GI 类
    private var mat : Material;
    // 在该帧被完全渲染后，我们将绘制提取了 alpha 通道的一个全屏矩形
```

```
    function Start ()
 {
while(true) {
        yield WaitForEndOfFrame();
        if(!mat) {
mat = new Material("Shader \" Hidden/Alpha "(" + "SubShader { " + " Pass {" + "ZTest Always Call off ZWrite off" + "Blend DstAlpha Zero" + "Color(1,1,1,1)" + "}" + "}" + "}");
        }
        GL.PushMatrix();
        GL.LoadOrtho();
        for(var i=0;i<mat.passCount;++i) {
mat.SetPass(i);
GL.Begin(GLQUADS);
GL.Vector3(0, 0, 0.1);
GL.Vector3(1, 0, 0.1);
GL.Vector3(1, 1, 0.1);
GL.Vector3(0, 1, 0.1);
GL.End();
        }
        GL.PopMatrix();
    }
 }
```

WaitForFixedUpdate

类，继承自 YieldInstruction

等待直到下一个固定帧率更新函数。　　参见: FixedUpdate

在 coroutine 中 WaitFixedUpdate 只能用于 yield 语句。

```
 yield new WaitForFixedUpdate();
```

WaitForSeconds

类，继承自 YieldInstruction

在给定的秒数内暂停协同进程的执行。

在 coroutine 中 WaitForSeconds 只能用于 yield 语句。

```
 // 打印 0
 print(Time.time);
 // 等待 5 秒
 yield new WaitForSeconds(5);
 // 打印 5.0
 print(Time.time);
```

构造函数

◆ static function WaitForSeconds(seconds : float) : WaitForSeconds

描述: 创建一个 yield 指令来等待给定的秒数。

```
 // 打印 0
print(Time.time);
 // 等待 5 秒
```

```
yield new WaitForSeconds(5);
// 打印 5.0
print(Time.time);
```

属性

AddComponentMenu

类，从 Attribute 继承。

AddComponentMenu 属性允许你防止一个脚本到”Component”菜单的任何位置，而不仅是”Component->Scripts”菜单。

使用这个更好的组织 Component 菜单，这种方法可以在添加脚本时改善工作流程。重要提示：需要重启！

```
// JavaScript 的例子
@script
AddComponentMenu(“Transform/FollowTransform”)
class FollowTransform : MonoBehaviour {
}
// C#的例子
[AddComponentMenu(“Transform/FollowTransform”)]
class FollowTransform : MonoBehaviour {
}
```

构造函数

- static function AddComponentMenu(menuName : string) : AddComponentMenu

描述: 这个脚本将根据 menuName 纺织在组件菜单中。menuName 是组件的路径”Rendering/DoSomething”。如果 menuName 为”” 组件将从菜单中隐藏。如果 menuName 为”” 组件将从菜单中隐藏。

ContextMenu

类，从 Attribute 继承

ContextMenu 属性允许你添加命令到上下文菜单。

在这个附加脚本的检视面板中。当用户选择这个上下文菜单，该函数将被执行。

这最适合用来从脚本中自动设置场景数据。这个函数必须是非静态的。

```
// JavaScript 的例子
@ContextMenu(“Do Something”)
function DoSomething() {
    Debug.Log(“Perform operation”);
}
// C#的例子
class ContextTesting : MonoBehaviour {
    // 在脚本的检视面板中添加名为”Do Something”的上下文菜单
    [ContextMenu(“Do Something”)]
    void DoSomething() {
        Debug.Log(“Perform operation”);
    }
}
```

构造函数

◆ **static function ContextMenu(name : string) : ContextMenu**

描述: 添加这个函数到组件的上下文菜单中。

在这个附加脚本的检视面板中。当用户选择这个上下文菜单后，该函数将被执行。

这最适合用来从脚本中自动设置场景的数据。这个函数必须是非静态的。

```
// JavaScript 的例子
 @ContextMenu("Do Something")
 function DoSomething() {
     Debug.Log("Perform operation");
 }
 // C#的例子
 class ContextTesting : MonoBehaviour {
     // 在脚本的检视面板中添加名为”Do Something”的上下文菜单
     [ContextMenu("Do Something")]
     void DoSomething() {
         Debug.Log("Perform operation");
     }
 }
```

ExecuteInEditMode

类，从 **Attribute** 继承

让脚本在编辑模式执行。

默认的，脚本只在运行模式时执行。这可以让这个脚本在编辑模式时执行。

```
 @script
 ExecuteInEditMode();
 // 只是一个简单的脚本来查找目标变换
 var target : Transform;
 function Update() {
     if(target)
         transform.LookAt(target);
 }
```

HiddenInInspector

类，从 **Attribute** 继承

使一个变量不会出现在检视面板中但是能被序列化。

```
 // 使 p 不显示在检视面板中，但是能被序列化
 @HiddenInInspector
 var p = 5;
```

NonSerialized

类，从 **Attribute** 继承

NonSerialized 属性标记一个变量没有被序列化。

用这种方法你能保持一个公开变量，并且 **Unity** 不会序列化它或在检视面板中显示它。

```
// p 不会显示在检视面板中或被序列化
 // JavaScript 的例子
 @System.NonSerialized
 var p = 5;
```

```
// C#的例子
class Test {
    // p 不会显示在检视面板中或者被序列化
    [System.NonSerialized]
    public int p = 5;
}
```

RPC

类，从 Attribute 继承

RenderBeforeQueues

类，从 Attribute 继承

定义在哪个渲染队列中 OnRenderObject 将被调用的属性。

参见: Render.queues，MonoBehaviour.OnRenderObject

```
// 在渲染不透明和透明物体之前 OnRenderObject 被调用
@RenderBeforeQueues(1000, 2000)
function OnRenderObject(queue : int) {
    // 做自定义的渲染
}
```

构造函数

◆ static function RenderBeforeQueues(params args : int[]) : RenderBeforeQueues

描述: 定义在哪个渲染队列中 OnRenderObject 将被调用。

参见: Render.queues，MonoBehaviour.OnRenderObject

```
// 在渲染不透明和透明物体之前 OnRenderObject 被调用
@RenderBeforeQueues(1000, 2000)
function OnRenderObject(queue : int) {
    // 做自定义的渲染
}
```

RequireComponent

类，从 Attribute 继承

这个 RequireComponent 属性自动添加所需组件作为一个依赖。

当使用 RequireComponent 添加一个脚本，这个需要的组件将自动被添加到游戏物体上，这可以用来避免设置错误。例如，一个脚本也许需要一个刚体添加到同一个游戏物体上。使用 RequireComponent 这个将被自动完成，因此不会出现设置错误。

```
// C#例子，标记 PlayerScript 需要一个刚体
[RequireComponent(typeof(Rigidbody))]
class PlayerScript : MonoBehaviour {
    void FixedUpdate() {
        rigidbody.AddForce(Vector3.up);
    }
}
```

构造函数

◆ static function RequireComponent(requiredComponent : Type) : RequireComponent

描述: 请求添加一个组件。

◆ static function RequireComponent(requiredComponent : Type, requiredComponent2 : Type) : RequireComponent

描述: 请求添加两个组件。

◆ static function RequireComponent(requiredComponent : Type, requiredComponent2 : Type, requiredComponent3 : Type) : RequireComponent

描述: 请求添加三个组件。

Serializable

类，从 Attribute 继承

序列化属性允许你在检视面板中嵌入一个类的子属性。

你可以使用这个来在检视面板中显示一个变量，类似于 Vector3 显示在检视面板中。名称和一个三角形可以展开它的属性。你需要从 System.Object 派生一个类，并给它 Serializable 属性。在 JavaScript 中 Serializable 属性是隐式的并不是必须的。

```
class Test extends System.Object {
    var p = 5;
    var c = Color.white;
}
var test = Test();
// C#的例子:
[System.Serializable]
class Test {
    public int p = 5;
    public Color c= Color.white;
}
```

枚举

AnimationBlendMode

枚举

由 Animation.Play 函数使用。

值

◆ AnimationBlendMode.Additive

描述: 动画将被附加。

◆ AnimationBlendMode.Blend

描述: 动画将被混合

AudioVelocityUpdateMode

枚举

描述: 一个 AudioSource 或 AudioListener 何时被更新。

值

◆ AudioVelocityUpdateMode.Auto

描述: 如果源或侦听器附加在一个 Rigidbody 上就以固定更新循环更新它，否则使用动态的。

◆ AudioVelocityUpdateMode.Dynamic

描述: 以动态的更新循环更新源或者侦听器。

◆ AudioVelocityUpdateMode.Fixed

描述: 以固定的更新循环更新源或者侦听器。

CameraClearFlags

枚举

Camera.clearFlags 值用来决定在渲染一个 Camera 时清除什么。

参见: camera 组件

值

◆ CameraClearFlags.Depth

描述: 只清除深度缓存。

这将留下前一帧的颜色或者任何被显示的东西。

```
// 只清除深度缓存
camera.clearFlags = CameraClearFlags.Depth;
```

参见: Camera.clearFlags 属性，camera 组件

◆ CameraClearFlags.Nothing

描述: 不清除任何东西。

这将留下前一帧的颜色的深度缓存或者任何被显示的东西。

```
// 不清除任何东西
camera.clearFlags = CameraClearFlags.Nothing;
```

参见: Camera.clearFlags 属性，camera 组件

◆ CameraClearFlags.Skybox

描述: 用天空盒清除。

如果没有设置天空盒，相机将继续使用 backgroundColor 来清除。

```
// 用天空盒清除
camera.clearFlags = CameraClearFlags.Skybox;
```

参见: Camera.clearFlags 属性，camera 组件，Render 设置

◆ CameraClearFlags.SolidColor

描述: 用背景颜色清除

```
camera.clearFlags = CameraClearFlags.SolidColor;
```

参见: Camera.clearFlags 属性，camera 组件，Camera.backgroundColor 属性

CollisionFlags

枚举

CollisionFlags 是由 CharacterController.Move 返回的一个 bitmask。

它给你一个角色和其他任何物体碰撞的大概位置。

值

◆ CollisionFlags.Above

描述: CollisionFlags 是由 CharacterController.Move 返回的一个 bitmask。它给你一个角色和其他任何物体碰撞的大概位置。

```
function Update() {
    var controller : CharacterController = GetComponent(CharacterController);
    if(controller.collisionFlags == CollisionFlags.None)
        print("Free floating!");
    if(controller.collisionFlags & CollisionFlags.Sides)
        print("Touching sides!");
    if(controller.collisionFlags == CollisionFlags.Sides)
        print("Only touching sides, nothing else!");
    if(controller.collisionFlags & CollisionFlags.Above)
        print("Touching ceiling!");
    if(controller.collisionFlags == CollisionFlags.Above)
        print("Only touching ceiling, nothing else!");
```

```
        if(controller.collisionFlags & CollisionFlags.Below)
            print("Touching ground!");
        if(controller.collisionFlags == CollisionFlags.Below)
            print("Only touching ground, nothing else!");
    }
```

◆ CollisionFlags.Below

描述: CollisionFlags 是由 CharacterController.Move 返回的一个 bitmask。它给你一个角色和其他任何物体碰撞的大概位置。

```
    function Update() {
    var controller : CharacterController = GetComponent(CharacterController);
        if(controller.collisionFlags == CollisionFlags.None)
            print("Free floating!");
        if(controller.collisionFlags & CollisionFlags.Sides)
            print("Touching sides!");
        if(controller.collisionFlags == CollisionFlags.Sides)
            print("Only touching sides, nothing else!");
        if(controller.collisionFlags & CollisionFlags.Above)
            print("Touching ceiling!");
        if(controller.collisionFlags == CollisionFlags.Above)
            print("Only touching ceiling, nothing else!");
        if(controller.collisionFlags & CollisionFlags.Below)
            print("Touching ground!");
        if(controller.collisionFlags == CollisionFlags.Below)
            print("Only touching ground, nothing else!");
    }
```

◆ CollisionFlags.None

描述: CollisionFlags 是由 CharacterController.Move 返回的一个 bitmask。它给你一个角色和其他任何物体碰撞的大概位置。

```
    function Update() {
        var controller : CharacterController = GetComponent(CharacterController);
        if(controller.collisionFlags == CollisionFlags.None)
            print("Free floating!");
        if(controller.collisionFlags & CollisionFlags.Sides)
            print("Touching sides!");
        if(controller.collisionFlags == CollisionFlags.Sides)
            print("Only touching sides, nothing else!");
        if(controller.collisionFlags & CollisionFlags.Above)
            print("Touching ceiling!");
        if(controller.collisionFlags == CollisionFlags.Above)
            print("Only touching ceiling, nothing else!");
        if(controller.collisionFlags & CollisionFlags.Below)
            print("Touching ground!");
        if(controller.collisionFlags == CollisionFlags.Below)
            print("Only touching ground, nothing else!");
```

```
}
```

◆ **CollisionFlags.Sides**

描述: CollisionFlags 是由 CharacterController.Move 返回的一个 bitmask。它给你一个角色和其他任何物体碰撞的大概位置。

```
function Update() {
    var controller : CharacterController = GetComponent(CharacterController);
    if(controller.collisionFlags == CollisionFlags.None)
        print("Free floating!");
    if(controller.collisionFlags & CollisionFlags.Sides)
        print("Touching sides!");
if(controller.collisionFlags == CollisionFlags.Sides)
        print("Only touching sides, nothing else!");
    if(controller.collisionFlags & CollisionFlags.Above)
        print("Touching ceiling!");
    if(controller.collisionFlags == CollisionFlags.Above)
        print("Only touching ceiling, nothing else!");
    if(controller.collisionFlags & CollisionFlags.Below)
        print("Touching ground!");
    if(controller.collisionFlags == CollisionFlags.Below)
        print("Only touching ground, nothing else!");
}
```

ConfigurableJointMotion

枚举

沿着 6 个轴限制 ConfigurableJoint 的移动。

值

◆ **ConfigurableJointMotion.Free**

描述: 沿着这个轴的运动将是完全自由和完全无约束的。

◆ **ConfigurableJointMotion.Limited**

描述: 沿着这个轴的运动将被分别限制。

◆ **ConfigurableJointMotion.Locked**

描述: 沿着这个轴的运动将被锁定。

ConnectionTesterStatus

枚举

值

◆ **ConnectionTesterStatus.Error**

描述:

◆ **ConnectionTesterStatus.PrivateIPHasNATPunchThrough**

描述: 私有地址被检测到并且能做 NAT 穿透。

◆ **ConnectionTesterStatus. PrivateIPNoNATPunchThrough**

描述: 私有地址被检测到并且不能做 NAT 穿透。

◆ **ConnectionTesterStatus.PublicIPIsConnectable**

描述: 公有的 IP 地址被检测到并且游戏的侦听端口可以通过互联网访问。

◆ **ConnectionTesterStatus.PublicIPNoServerStarted**

描述: 公有的 IP 地址被检测到但是服务器没有被初始化并且没有侦听端口。

◆ **ConnectionTesterStatus.PublicIPPortBlocked**

描述: 公有的 IP 地址被检测到但是它的端口不能通过互联网连接。

◆ **ConnectionTesterStatus.Undetermined**

描述: 测试结果未知，还在进行中。

CubemapFace

枚举

Cubemap 面。

被 Cubemap.GetPixel 和 Cubemap.SetPixel。

值

PositiveX	右面（+x）
NegativeX	左面（-x）
PostiveY	上面（+y）
NegativeY	下面（-y）
PostiveZ	前面（+z）
NegativeZ	后面（-z）

EventType

枚举

UnityGUI 输入和处理事件的类型。

参见: Event.type， Event，GUI 脚本手册。

值

◆ **EventType.ContexClick**

描述: 用户使用右键单机（或者在 mac 上的 Control+单机）。

如果是窗口的应用，应该显示一个上下文菜单。在编辑器中只发送。

◆ **EventType.DragExited**

描述: 只限于编辑器，存在的拖放操作。

参见: DragAndDrop 类

◆ **EventType.DragPerform**

描述: 只限于编辑器，拖放操作执行。

参见: DragAndDrop 类

◆ **EventType.DragUpdated**

描述: 只限于编辑器，拖放操作更新。

参见: DragAndDrop 类

◆ **EventType.ExecuteCommand**

描述: 执行特殊的命令（例如，拷贝和粘贴）。

“Copy”,”Cut”,”Paste”,”Delete”,”FrameSelected”,”Duplicate”,”SelectAll”

◆ **EventType.Ignore**

描述: Event 应该被忽略。

这个事件被临时禁用并应该被忽略。

◆ **EventType.KeyDown**

描述: 一个键盘按键被按下。

使用 Event.character 查看什么被键入。使用 Event.keyCode 处理箭头，home/end 或其他任何功能键，或者找到哪个物理键被按下。这个事件根据端用户键盘的重复设置来重复发送。

注意按键可以来自不同的事件，一个是 Event.keyCode，另一个是 Event.chatacter，

根据键盘布局，多个 Event.keyCode 可以产生一个 Event.character 事件。

- **EventType.KeyUp**

  描述: 一个键盘按键被释放。

  使用 Event.keyCode 查看哪个物理按键被释放。注意根据系统和键盘布局的不同，Event.character 也许不包含任何字符。

- **EventType.Layout**

  描述: 一个布局事件。

  这个事件先于其他任何事件被发送。这是一个几回来执行任何初始化，它被用于自动布局系统。

- **EventType.MouseDown**

描述: 鼠标按键被按下。

当任何鼠标按键被按下的时候发送该事件 - 使用 Event.button 决定哪个按键被按下。

- **EventType.MouseDrag**

描述: 鼠标被拖动。

鼠标移动并且按键被按下 – 拖动鼠标。使用 Event.mousePosition 和 Event.delta 来决定鼠标移动。

- **EventType.MouseMove**

描述: 鼠标被移动。

鼠标移动，没有任何按键被按下。使用 Event.mousePosition 和 Event.delta 来确定鼠标移动。

- **EventType.MouseUp**

  描述: 鼠标按键被释放。

  当任何鼠标按键被释放时发送该事件。使用 Event.button 决定哪个按键被释放。

- **EventType.Repaint**

  描述: 一个重绘事件。每帧发送一个。

  首先处理所有的其他事件，然后这个重绘事件被发送。

- **EventType.ScrollWheel**

  描述: 滚轮被滚动。

  使用 Event.delta 决定 X 和 Y 的滚动量。

- **EventType.Used**

  描述: 已处理的事件。

  这个事件已经被其他的一些控件使用并应该被忽略。

- **EventType.ValidateCommand**

  描述: 验证特殊的命令（例如，拷贝和粘贴）。

  “Copy”,” Cut”,” Paste”,” Delete”,” FrameSelected”,” Duplicate”,” SelectAll”等等，只在编辑器中发送。

FilterMode

纹理的过滤模式。对应于 texture.inspector 中的设置。

参见: Texture.filterMode，texture.assets

值

- **FilterMode.Bilinear**

  描述: 双线性过滤 – 纹理被平均采样。

```
renderer.material.mainTexture.filterMode = FilterMode.Bilinear;
```

参见: Texture.filterMode，texture assets

- FilterMode.Point

描述: 点过滤 － 纹理像素变得近乎斑驳。

renderer.material.mainTexture.filterMode = FilterMode.Point;

参见: Texture.filterMode，texture assets

- FilterMode.Trilinear

描述: 三线性过滤 － 纹理被平均采样并在 mipmap 等级之间混合。

renderer.material.mainTexture.filterMode = FilterMode.Trilinear;

参见: Texture.filterMode，texture assets

FocusType

被 GUIUtility.GetControlID 使用来通知 UnityGUI 系统给定的空间能否获取键盘焦点。

值

- FocusType.Keyborad

描述: 这是一个何时的键盘控制。在所有平台上它都能有输入焦点。用于 TextField 和 TextArea 控件。

- FocusType.Native

描述: 这个空间可以在 Windows 下获取焦点，但是在 Mac 下不能。用于按钮，复选框和其他的“可按下”物体。

- FocusType.Passive

描述: 这个控件永远不能接收键盘焦点。

ForceMode

Rigidbody.AddForce 如何使用力的选项。

值

- ForceMode.Acceleration

描述: 添加一个牛顿力到这个刚体，忽略它的质量。

这个模式不依赖于刚体的质量。因此推和旋转的应用将不会受到刚体质量的影响，相对于 ForceMode.Force 这将以相同的设置移动每个刚体而忽略它们的质量差别。这个模式更像一个加速度而不是速度。在这个模式线，应用到物体的力参数的单位是距离/时间^2。

- ForceMode.Force

描述: 添加一个牛顿力到这个刚体，使用它的质量。

这个模式依赖于刚体的质量。因此，必须对较大的质量的物体应用更多的力来推动或旋转它。这模式更像一个加速度而不是速度。在这个模式线，应用到物体的力的参数的单位是质量*距离/时间^2。

- ForceMode.Impulse

描述: 用刚体的质量改变它的速度。

这个模式依赖于刚体的质量。因此，必须对较大质量的物体应用更多的力来推动或旋转

它。这模式更像一个速度而不是加速度。在这个模式线，应用到物体的力的参数是质量*距离/时间。

- ForceMode.VelocityChange

描述: 改变刚体的速度。忽略它的质量。

这个模式不依赖于刚体的质量。因此推和旋转的应用将不会受到刚体质量的影响。

这可用于控制不同尺寸的飞船而不考虑质量差别。在这个模式，应用到物体的力的参数的单位是距离/时间。

HideFlags

Bit 蒙板，可以控制对象销毁和在检视面板中的可视性。

值

- **HideFlag.DontSave**

描述: 这个物体将不会被保存到场景。当一个新的场景被加载时它将不会被销毁。

使用 **DestroyImmediate** 手工清理这个物体是你的责任，否则它将泄漏。

- **HideFlags.HideAndDontSave**

描述: 不显示在层次视图中并且不保存到场景的组合。

这个最常用于那些由脚本创建并纯粹在它控制之下的物体。

- **HideFlags.HideInHierarchy**

描述: 如果这个对象是储存在一个资源中，这个对象将不会显示在层次视图中并且不会显示在工程视图中。

- **HideFlags.HideInInspector**

描述: 不能在检视面板中查看。

- **HideFlags.NotEditable**

描述: 这个物体在检视面板中不可编辑。

ImagePosition

在 **GUIStyle** 中图片和文本如何被放置。

值

- **ImagePosition.ImageAbove**

描述: 图片在文本上面。

- **ImagePosition.ImageLeft**

描述: 图片在文本左侧。

- **ImagePosition.ImageOnly**

描述: 只有这个图片被显示。

- **ImagePosition.TextOnly**

描述: 只有这个文本被显示。

JointDriveMode

ConfigurableJoint 试图基于这个表示来达到这个位置/速度的目的。

值

- **JointDriveMode.None**

描述: 不用任何力来达到目标。

- **JointDriveMode.Position**

描述: 尽量达到特定的位置。

- **JointDriveMode.PositionAndVelocity**

描述: 尽量达到特定的位置和速度。

- **JointDriveMode.Velocity**

描述: 尽量达到特定的速度。

JointProjectionMode

该属性用来决定在物体偏离太多的时候如果它吸附到约束位置。

参见: **ConfigurableJoint**

值

◆ **JointProjectionMode.None**

描述: 不吸附。

◆ **JointProjectionMode.PositionAndRotation**

描述: 吸附到位置和旋转。

◆ **JointProjectionMode.PositionOnly**

描述: 只吸附到位置。

KeyCode

KeyCode 是由 **Event.keyCode** 返回的。这些直接映射到键盘上的物理键。

值

Backspace	退格键
Delete	**Delete** 键
Tab	**Tab** 键
Clear	**Clear** 键
Return	回车键
Pause	暂停键
Escape	**ESC** 键
Space	空格键
Keypad0	小键盘 **0**
Keypad1	小键盘 **1**
Keypad2	小键盘 **2**
Keypad3	小键盘 **3**
Keypad4	小键盘 **4**
Keypad5	小键盘 **5**
Keypad6	小键盘 **6**
Keypad7	小键盘 **7**
Keypad8	小键盘 **8**
Keypad9	小键盘 **9**
KeypadPeriod	小键盘“.”
KeypadDivide	小键盘“/”
KeypadMultiply	小键盘“*”
KeypadMinus	小键盘“-”
KeypadPlus	小键盘“+”
KeypadEnter	小键盘“Enter”
KeypadEquals	小键盘“=”
UpArrow	方向键上
DownArrow	方向键下
RightArrow	方向键右
LeftArrow	方向键左
Insert	**Insert** 键
Home	**Home** 键
End	**End** 键
PageUp	**PageUp** 键
PageDown	**PageDown** 键
F1	功能键 **F1**

F2 功能键 F2
F3 功能键 F3
F4 功能键 F4
F5 功能键 F5
F6 功能键 F6
F7 功能键 F7
F8 功能键 F8
F9 功能键 F9
F10 功能键 F10
F11 功能键 F11
F12 功能键 F12
F13 功能键 F13
F14 功能键 F14
F15 功能键 F15
Alpha0 按键 0
Alpha1 按键 1
Alpha2 按键 2
Alpha3 按键 3
Alpha4 按键 4
Alpha5 按键 5
Alpha6 按键 6
Alpha7 按键 7
Alpha8 按键 7
Alpha9 按键 9
Exclaim ‘!’键
DoubleQuote 双引号键
Hash Hash 键
Dollar ‘$’键
AmpersandAmpersand 键
Quote 单引号键
LeftParen左括号键
RightParen 右括号键
Asterisk ‘ * ’键
Plus ‘ + ’键
Comma ‘ , ’键
Minus ‘ - ’键
Period ‘ . ’键
Slash ‘ / ’键
Colon ‘ : ’键
Semicolon ‘ ; ’键
Less ‘ < ‘键
Equals ‘ = ‘键
Greater ‘ > ‘键
Question ‘ ? ’键

At ‘@’键
LeftBracket ‘ [‘键
Backslash ‘ \ ’键
RightBracket ‘] ’键
Caret ‘ ^ ’键
Underscore ‘ _ ’键
BackQuote ‘ ` ’键
A ‘a’键
B ‘b’键
C ‘c’键
D ‘d’键
E ‘e’键
F ‘f’键
G ‘g’键
H ‘h’键
I ‘i’键
J ‘j’键
K ‘k’键
L ‘l’键
M ‘m’键
N ‘n’键
O ‘o’键
P ‘p’键
Q ‘q’键
R ‘r’键
S ‘s’键
T ‘t’键
U ‘u’键
V ‘v’键
W ‘w’键
X ‘x’键
Y ‘y’键
Z ‘z’键
Numlock Numlock 键
Capslock 大小写锁定键
ScrollLock Scroll Lock 键
RightShift 右上档键
LeftShift 左上档键
RightControl 右 Ctrl 键
LeftControl 左 Ctrl 键
RightAlt 右 Alt 键
LeftAlt 左 Alt 键
LeftApple 左 Apple 键
LeftWindows 左 Windows 键

RightApple 右 Apple 键
RightWindows 右 Windows 键
AltGr Alt Gr 键
Help Help 键
Print Print 键
SysReq Sys Req 键
Break Break 键
Mouse0 鼠标左键
Mouse1 鼠标右键
Mouse2 鼠标中键
Mouse3 鼠标第 3 个按键
Mouse4 鼠标第 4 个按键
Mouse5 鼠标第 5 个按键
Mouse6 鼠标第 6 个按键
JoystickButton0 手柄按键 0
JoystickButton1 手柄按键 1
JoystickButton2 手柄按键 2
JoystickButton3 手柄按键 3
JoystickButton4 手柄按键 4
JoystickButton5 手柄按键 5
JoystickButton6 手柄按键 6
JoystickButton7 手柄按键 7
JoystickButton8 手柄按键 8
JoystickButton9 手柄按键 9
JoystickButton10 手柄按键 10
JoystickButton11 手柄按键 11
JoystickButton12 手柄按键 12
JoystickButton13 手柄按键 13
JoystickButton14 手柄按键 14
JoystickButton15 手柄按键 15
JoystickButton16 手柄按键 16
JoystickButton17 手柄按键 17
JoystickButton18 手柄按键 18
JoystickButton19 手柄按键 19
Joystick1Button0 第一个手柄按键 0
Joystick1Button1 第一个手柄按键 1
Joystick1Button2 第一个手柄按键 2
Joystick1Button3 第一个手柄按键 3
Joystick1Button4 第一个手柄按键 4
Joystick1Button5 第一个手柄按键 5
Joystick1Button6 第一个手柄按键 6
Joystick1Button7 第一个手柄按键 7
Joystick1Button8 第一个手柄按键 8
Joystick1Button9 第一个手柄按键 9

Joystick1Button10 第一个手柄按键 10
Joystick1Button11 第一个手柄按键 11
Joystick1Button12 第一个手柄按键 12
Joystick1Button13 第一个手柄按键 13
Joystick1Button14 第一个手柄按键 14
Joystick1Button15 第一个手柄按键 15
Joystick1Button16 第一个手柄按键 16
Joystick1Button17 第一个手柄按键 17
Joystick1Button18 第一个手柄按键 18
Joystick1Button19 第一个手柄按键 19
Joystick2Button0 第二个手柄按键 0
Joystick2Button1 第二个手柄按键 1
Joystick2Button2 第二个手柄按键 2
Joystick2Button3 第二个手柄按键 3
Joystick2Button4 第二个手柄按键 4
Joystick2Button5 第二个手柄按键 5
Joystick2Button6 第二个手柄按键 6
Joystick2Button7 第二个手柄按键 7
Joystick2Button8 第二个手柄按键 8
Joystick2Button9 第二个手柄按键 9
Joystick2Button10 第二个手柄按键 10
Joystick2Button11 第二个手柄按键 11
Joystick2Button12 第二个手柄按键 12
Joystick2Button13 第二个手柄按键 13
Joystick2Button14 第二个手柄按键 14
Joystick2Button15 第二个手柄按键 15
Joystick2Button16 第二个手柄按键 16
Joystick2Button17 第二个手柄按键 17
Joystick2Button18 第二个手柄按键 18
Joystick2Button19 第二个手柄按键 19
Joystick3Button0 第三个手柄按键 0
Joystick3Button1 第三个手柄按键 1
Joystick3Button2 第三个手柄按键 2
Joystick3Button3 第三个手柄按键 3
Joystick3Button4 第三个手柄按键 4
Joystick3Button5 第三个手柄按键 5
Joystick3Button6 第三个手柄按键 6
Joystick3Button7 第三个手柄按键 7
Joystick3Button8 第三个手柄按键 8
Joystick3Button9 第三个手柄按键 9
Joystick3Button10 第三个手柄按键 10
Joystick3Button11 第三个手柄按键 11
Joystick3Button12 第三个手柄按键 12
Joystick3Button13 第三个手柄按键 13

Joystick3Button14 第三个手柄按键 14

Joystick3Button15 第三个手柄按键 15

Joystick3Button16 第三个手柄按键 16

Joystick3Button17 第三个手柄按键 17

Joystick3Button18 第三个手柄按键 18

Joystick3Button19 第三个手柄按键 19

LightRenderMode

Light 如何被渲染。

参见: light 组件

值

◆ LightRenderMode.Auto

描述: 自动选择渲染模式。

选择是否渲染这个 Light 为像素光或者顶点光源（建议缺省）。

```
// 设置光源的渲染模式为自动
light.renderMode = LightRenderMode.Auto;
```

参见: light 组件

◆ LightRenderMode.ForcePixel

描述: 强制 Light 为像素光源。

只将这个用于真正重要的光源，例如一个玩家的手电筒。

```
// 强制光源为像素光源
light.renderMode = LightRenderMode.ForcePixel;
```

参见: light 组件

◆ LightRenderMode.ForceVertex

描述: 强制 Light 为顶点光源。

这个选项对于背景光或远处的光照是非常好的。

```
// 强制光源为顶点光源
light.renderMode = LightRenderMode.ForceVertex;
```

参见: light 组件

LightShadows

Light 的阴影投射选项。

参见: light 组件

值

◆ LightShadows.Hand

描述：投射“hard”阴影（没有阴影过滤）

```
//设置光源为投射硬阴影
Light.shadows=LightShadows.Hard;
```

参见：light component

◆LightShadows.None

描述：不投射阴影（默认）

```
//设置光源为不投射阴影
Light.shadows=lightshadows.None;
```

参见:light component

◆LightShadows.Soft

描述：投射 Soft 阴影（带有 4 倍 PCF 过滤）

//设置光源为投射 4 倍过滤的软阴影阴影
Light.shadows=lightshadows. Soft;
参见:light component

LightType

Light 的类型。
参见：Light.typelight component
值
◆LightType.Directional
描述：这个光源是一个直射光源。
参见：Light.type， light component
//制作一个直射光源
Light.type=LightType.Directional；
◆LightType.Point
描述：这个光源是一个点光源。
参见：Light.type，light component
//制作一个直射光源
Light.type=LightType.Point；
◆LightType.Spot
描述：这个光源是一个透射光源。
参见：Light.type， light component
//制作一个透射光源
Light.type=LightType.Spot；

MasterServerEvent

值

RegistrationFailedGameName	注册失败，因为给出的游戏名称为空。
RegistrationFailedGameType	注册失败，因为给出的游戏类型为空。
RegistrationFailedNoServer	注册失败，因为没有服务器在运行。
RegistrationSueceeded	注册到主服务器成功，接受到确认。
HostListReceived	从主服务器接受到一个主机列表

NetworkConnectionError
值

NoError	
RSAPublicKeyMismatch	我们提供的 RSA 公匙与我们所连接系统的不匹配。
InvalidPassword	服务器需要密码并且拒绝我们的链接，因为我们没有设置正确的密码
ConnecttionFailed	连接失败，可能因为内部连接性问题。
TooManyConnectedPlayers	服务器到达最大限度，不能连接
ConnectionBanned	我们被试图连接到的系统禁止了（可能是临时的）
AlreadyConnectedToAnotherServer	不能同时链接到两个服务器，在再次连接之前关闭这个连接
CreateSocketOrThreadFailure	试图初始化网络接口时出现内部错误，套接字可能已经被使用了

IncorrectParameters Connect 函数具有不正确的参数

EmptyConnectTarget 没有给出链接目标

InternalDirectConnectFailed 客户端不能内部链接到位于相同网络中的启用了 NAT 的服务

NATTargetNotConnected 我们试图连接到的 NAT 目标没有连接到辅助服务器

NATTargetConnectionLost 当试图链接到 NAT 目标时，连接丢失

值

◆ ParticleRenderMode.Billboard

描述：作为面向玩家的公告板渲染例子（默认）

◆ ParticleRenderMode.HorizentalBillboard

描述：作为公告板渲染粒子，总是沿着 Y 轴

◆ ParticleRenderMode.SortedBillboard

描述：从后向前排序并作为公告板渲染。

这个使用混合例子着色器看起来更好，但是因为排序边得较慢

◆ ParticleRenderMode.Streteh

描述：在运动方向拉伸粒子

◆ ParticleRenderMode.VerticalBillboard

描述：作为公告板渲染粒子，总是面向玩家，但是不沿着 X 轴旋转

PhysicMaterialCombine

描述：碰撞物体的物理材质如何被组合

参见：PhysicMaterial.InctionCombine，PhysicMaterial.boticeCombine

值

Average 平均两个碰撞材质的摩擦/弹力

Multiply 两个碰撞材质的摩擦/弹力相乘

Minimum 使用两个碰撞材质的摩擦/弹力中较小的一个

Maximum 使用两个碰撞材质的摩擦/弹力中较大的一个

PlayMode

由 Animation.Play 函数使用

值

StopSameLayer 将停止在同一层上开始的所有动画。当播放动画的时候这个是默认值

StopAll 停止所有由这个组件开始的动画

PrimitiveType

各种变量可以通过使用 GameObject.CreatePrimitive 函数创建

参见 GameObject.CreatePrimitive

值

◆PrimitiveType Capsule

描述：胶囊几何体

参见：GameObject.CreatePrimitive

//创建一个胶囊几何体

functionsStart（）{

var capsule=

GameObject.CreatePrimitive(PrimitiveType.Capsule);

}
◆PrimitiveType Cube
描述：立方体
参见：GameObject.CreatePrimitive
//创建一个立方体
functionsStart（）{
var cube=
GameObject.CreatePrimitive(PrimitiveType.Cube);
}
◆PrimitiveType Cyhnder
描述：圆柱体
参见：GameObject.CreatePrimitive
//创建一个圆柱体
functionsStart（）{
var cyhnder=
GameObject.CreatePrimitive(PrimitiveType.Cylinder);
}
◆PrimitiveType Plane
描述：平面几何体
参见：GameObject.CreatePrimitive
//创建一个平面几何体
functionsStart（）{
var plane=
GameObject.CreatePrimitive(PrimitiveType.Plane);
}
◆PrimitiveType Sphere
描述：球形
参见：GameObject.CreatePrimitive
//创建一个球形
functionsStart（）{
var sphere=
GameObject.CreatePrimitive(PrimitiveType.Sphere);
}
QualityLeve
图像质量等级
有六个质量等级可以这样，每个等级的细节都在工程的 Quality Sertings 中设置
质量等级可以在脚本中使用 QualitySertings 类来切换
参见：QualitySertings currentlevel QualityLeveSertings
值
◆QualityLeve.Beautiful
描述：“beautiful”质量等级
参见：QualitySertings currentlevel QualityLeveSertings
◆QualityLeve.Fantastic
描述：“fantastic”质量等级

参见：QualitySertings currentlevel QualityLeveSertings

◆QualityLeve.Fast

描述：“fast”质量等级

参见：QualitySertings currentlevel QualityLeveSertings

◆QualityLeve.Fastest

描述：“fastest”质量等级

参见：QualitySertings currentlevel QualityLeveSertings

◆QualityLeve.Good

描述：“good”质量等级

参见：QualitySertings currentlevel QualityLeveSertings

◆QualityLeve.Simple

描述：“simple”质量等级

参见：QualitySertings currentlevel QualityLeveSertings

QueueMode

由 Animation.Play 函数使用

值

CompleteOmers 所有其他动画通知播放后开始播放

PlayNow 立刻开始播放，如果你只是想开苏创建一个复制动画可以使用这个

RPCmode

用来选择谁将接收这个 RPC

值

Server 只发送到服务器

Others 发送给所有人除了服务器

OthersBuffered 发送给每一个，除了服务器，并添加到缓存

All 发送到每个人

AllButFered 发送到每个人并添加到缓存

RenderTextureFormat

RenderTexture 的格式

参见：RenderTexture.fonmat，RenderTexture 类

值

◆RenderTextureFormat ARGB32

描述：一个 32 位颜色的渲染纹理格式

参见：RenderTexture，RenderTexture 类

◆RenderTextureFormat.Depth

描述：渲染纹理格式的深度

深度格式用来渲染高精度“深度”值到一个渲染纹理，实际使用哪个格式依赖于平台，在 OpenGL 中，它是原始“深度组件”格式（通常是 24 或 16 位），在 Direct3D9 中它是 32 位浮点（R32F）格式，在编写使用或渲染到深度纹理的 shader 时，必须确保它们能够工作在 OpenGL 和 Direct3D 中，参考 depth textures documentation

注意不是所有的显卡支持深度纹理。使用 SystemInfo supportsDepthRenderTextures 来检查是否支持

参见：RenderTexture format，RenderTexture 类，SystemInfo.supportsDepthRenderTextures

RigidbodyInterpolation

Rigidbody 插值模式

对于那些被相机跟随的主角色或交通工具，建议使用插值。对于任何其他刚体建议不使用插值

参见：**RigidbodyInterpolation** 变量

值

◆**RigidbodyInterpolation.Extrapolate**

描述：外插值将基于当前速度预测刚体的位置

如果你有一个快速移动的物体，这个能够导致刚体在一帧中穿过碰撞器然后弹回

Rigidbody.interpolation=RigidbodyInterpolation.Extrapolate;

参见：**RigidbodyInterpolation** 变量

◆**RigidbodyInterpolation.Interpolate**

描述：插值总是有些之后但是比外插值更光滑

Rigidbody.interpolation=RigidbodyInterpolation. Interpolate;

参见：**RigidbodyInterpolation** 变量

◆**RigidbodyInterpolation.None**

描述：不插值

Rigidbody.interpolation=RigidbodyInterpolation.None;

参见：**RigidbodyInterpolation** 变量

RotationDriveMode

用它自己的 **XYZ** 后者 **SlerpDrive** 控制 **ConfigurableJoint** 的旋转

值

XYAndZ 使用 **XY&Z** 驱动

Slerp 使用 **Slerp** 驱动

RuntimePlatform

应用程序运行的平台。由 **Application platform** 返回

值

OSXEditor **OSX** 下 **Unity** 编辑器模式

OSXPlayer **OSX** 上的播放器

WindowsPlayer **Windows** 上的播放器

OSXWebPlayer **OSX** 下的 **web** 播放器

OSXDashboard **OSX** 下 **Dashborard** 窗口

WindowsWebOlayer **Windows** 上的 **web** 播放器

WindowsEditor **Windows** 下 **Unity** 编辑器模式

ScaleMode

绘制纹理的缩放模式

值

StretchToFill 缩放纹理以便完全填充传入 **GUI.DrawTexture** 的矩形

SealeAndCrop 缩放纹理，维持长宽比，这样它完全覆盖传递到 **GUIDrawTexture** 的 **position** 矩形，如果纹理被描绘到具有不同长宽比的矩形上时，图像被裁剪

ScaleToFit 缩放纹理，维持长宽比，这样它完全与传递到 **GUIDrawTexture** 的 **position** 矩形相匹配

SendMessageOptions

如何发送一个消息的选项

这个用在 **GameObject** 和 **Component** 的 **SendMessage** 和 **BroadcastMessage** 上

值

◆SendMessageOptions.DontRequireReceiver

描述：**SendMessage** 不需要一个接收者

◆SendMessageOptions.RequireReceiver

描述：**SendMessage** 需要一个接收者

如果没有找到接收者，将在控制台上打印以错误（默认）

SkinQuality

影响单个顶点的最大骨骼数量

参见：**SkinnedMeshRenderer.quality**

值

Auto	从当前 **QualitySetings**（默认）数中选择骨骼数量
Bone1	仅使用 **1** 骨骼变形一个顶点，（最重要的骨骼被使用）
Bone2	仅使用 **2** 骨骼变形一个顶点，（最重要的骨骼被使用）
Bone4	仅使用 **4** 骨骼变形一个顶点

Space

在那个坐标空间中操作

参见：**Transform**

值

World	相对于世界坐标系统应用一个变换
Self	相对于局部坐标系统应用一个变换

TerrainLighting

Terrain 光照模式

参见：**Treeainlighting**，**TerrainLightings**，**TerrainSettings**

值

◆TerrainLighting Lightmap

描述：使用光照图渲染地形

该地形只使用广州图，并且不会受到游戏中光源的影响

参见：**TerrainLighting**，**TerrainLightmaps**，**TerrainLightmapsSettings**

◆TerrainLighting Pixel

描述：近处使用光源渲染地形，远处使用高度图

在游戏中靠近观察者的地形用光照，远处使用高度图混合，光源是顶点光照或像素光照模式，并能够有阴影

光照距离是（**Terrain.treeBillboardDistance**，**QualitySettings.shadowDistance**，**Terrain.basemapDistance**）的最小值

参见：**TerrainLighting**，**TerrainLightmaps**，**TerrainLightmapsSettings**

◆TerrainLighting Vertex

描述：使用顶点光照渲染地形

地形将以顶点光照模式被照亮，不使用光照贴图，投射光源就像顶点光

参见：**TerrainLighting**，**TerrainLightmaps**，**TerrainLightmapsSettings**

TextAlignment

多行文本应该如何被对齐

这个是被 GUIText.alignment 属性使用

参见：GUI Text component

值

Left 文本行左对齐

Center 文本行居中对齐

Right 文本行右对齐

TextAnchor

文本的锚点被放置在什么位置

这个是被 GUIText.anchor 属性使用

参见：GUI Text component

值

UpperLeft 文本被锚点在左上角

UpperCenter 文本被锚点在上边，垂直居中

UpperRight 文本被锚点在右上角

MiddleLeft 文本被锚点在左边，垂直居中

MiddleCenter 文本在水平和垂直方向上居中

MiddleRight 文本被锚点在右边，垂直居中

LowerLeft 文本被锚点在左下角

LowerCenter 文本被锚点在下边，垂直居中

LowerRight 文本被锚点在右上角

TextClipping

GUI 系统处理过大文本的以适合所分配矩形的方式

值

OverDow 文本随意浮动在该元素之外

Clip 文本被裁剪以便放置在该元素之内

TextureFormat

Texture 的格式，从脚本创建纹理时使用这个。

参见：Texture2D Texture2D，textureFormat

值

◆TextureFormat Alpha8

描述：只有 alpha 的纹理格式

```
function Start（）{
//创建一个新的只有 alpha 的纹理并将它赋予
//该渲染器材质
var texture=new Texture2D(128,128,TextureFormat,Alpht8,false),
rederer.material.mainTexture=texture;
}
```

参见：Texture2D Texture2D，textureFormat

◆TextureFormat ARGB32

描述：只有 alpha 的彩色纹理格式

```
function Start（）{
//创建一个新的纹理并将它赋予给渲染器材质
var texture=new Texture2D(128,128,TextureFormat,ARGB32,false),
```

rederer.material.mainTexture=texture;
}
参见：Texture2D Texture2D，textureFormat
◆TextureFormat DXT1
描述：一个压缩的彩色纹理格式
参见：Texture2D Texture2D，textureFormat
◆TextureFormat DXT5
描述：带有 alpha 通道的彩色压缩纹理格式
参见：Texture2D Texture2D，textureFormat
◆TextureFormat RGB24
描述：一个彩色纹理格式
function Start（）{
//创建一个新的纹理并将它赋予给渲染器材质
var texture=new Texture2D(128,128,TextureFormat,RGB32,false),
rederer.material.mainTexture=texture;
}
参见：Texture2D Texture2D，textureFormat
TextureWrapMode
纹理的包裹模式，对应与 texture inspector 中的设置
你可以能够平铺纹理（重复）或者映射一个纹理到物体上（裁剪）

参见：Texture.wrapMode，texture assets
值
◆TextureFormat.Clamp
描述：裁剪纹理到边界上最后一个像素
在映射一个纹理到物体上并且你不想纹理平铺时，这个可以避免包裹的不真实，UV 坐标将被裁剪到返回 0…1.当 UV 大于 1 或小于 0，将使用边界上的最后一个像素
renderer.material.mainTexture.wrapMode=TextureWrapMode.Clamp;
参见：Texture.wrapMode,texture assets.
◆TextureWrapMode.Repear
描述：平铺纹理，创建一个重复效果
当 UV 超出 0…1 范围，整数部分将被忽略，这样就创建了一个重复效果。
renderer.material.mainTexture.wrapMode=TextureWrapMode.Repear;
参见：Texture.wrapMode,texture assets.
WrapMode
在没有时间帧定义的地方如何对待时间
值
◆WrapMode.ClampForever
描述：播放动画。当它到达末端时，它将保持在最后一帧但不会停止播放
这个可用于附加的动画，当它们到达最大时不应该停止播放。
◆WrapMode.Default
描述：从动画曲线中读取重复模式，可以被设置为 Loop 或 PingPong
◆WrapMode.Loop
描述：当时间到达动画剪辑的末端，时间将从开始继续。

动画将不会停止播放

◆WrapMode.Once

描述：当时间到达动画剪辑的末端，剪辑将自动停止播放

◆WrapMode.PingPong

描述：当时间到达动画剪辑的末端时，时间将在开始和接受之间来回播放

动画不会停止播放

三、 编辑器类

AnimationClipCurveData

类

一个 AnimationClipCurveData 对象包含所有在 AnimationClip 中表示一个特定曲线所需要的所有信息，这个曲线制作一个附加到游戏物体/动画骨骼上的组件/材质属性的动画。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

变量

◆ var curve:AnimationCurve

描述：实际的动画曲线

◆ var path:string

描述：背动画的游戏物体/骨骼的路径

◆ var propertyName：string

描述：被动画的属性的名称

◆ var target：Object

描述：被动画的组件/材质

◆ var type：Type

描述：被动画的组件/材质类型

AnimationUtility

类

用于修改动画剪辑的编辑器工具函数

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

类方法

◆ static funciton GetAllCurves（clip：AnimationClip，includeCurveData：boll=ture）：AmimationClipCurveData[]

描述：从一个特定的动画剪辑上取回所有曲线

如果 includeCurveData 为假，所有在返回结果中的动画曲线将为 null，当你只想获取曲线类型和名称的列表时使用这个。

◆ static funciton GetAnimatableProperties （ go ： GameObject ）：AnimationClipCurveData[]

描述：取回附加在改游戏物体上所有组件/材质上的所有可动画的属性

◆ static function GetEditorCurve（clip：AnimationClip，relativePath：string，type：Type，propertyName：string）：AnimationCurve

描述：Unity 自动在内部组合位置曲线，缩放曲线，选择曲线

因此变换曲线总是组合的并且它们的关键帧总是所有关键帧点的联合。在编辑器中动

画剪辑让你指定特定的不必组合的编辑器曲线，因此它让用户以更直观的方式编辑曲线。

◆ static function GetFloatValue（root：GameObject，relativePath：string，type：Type，propertyName：string，out data：float）：bool

描述：通过采样特定的游戏对象上的一个曲线值得到当前的浮点值

用来记录关键帧

◆static function SetEditorCurve（clip：AnimationClip，relativePath：string，type：Type，propertyName：string，curve：AnimationClip）:void

描述：Unity 自动在内部组合位置曲线，缩放曲线，选择曲线

因此变换曲线总是组合的并且它们的关键帧总是所有关键帧点的联合。在编辑器中动画剪辑让你指定特定的不必组合的编辑器曲线，因此它让用户以更直观的方式编辑曲线。

AssetDatabase

类

一个用来访问资源并在资源上执行操作的接口

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

类方法

◆static funciton AddObjectToAsset（objectToAdd：bject,assetPath:string）:vold

描述：添加 objectToAdd 到 path 上已有的资源中

请注意你应该只添加资源到 asset 资源，例如导入模型或纹理资源将丢失它们的数据

```
@MenuItem（”eObject/Create Matenal”）
satic function CreateMaterial(){
// 创建一个简单材质资源
var material = new Material (Shader.Find("Specular"));
AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
// 给它添加一个动画剪辑
var animationClip = new AnimationClip();
animationClip.name=“My Clip”;
AssetDatabase.AdObjectToAsset（animationClip，matcrial）;
// 添加一个物体后重导入这个资源
// 否则这个改变只会显示在保持工程的时候
AssetDatabase.ImportAsset（AssetDatabase.GetAssetPath（animationClip））;
// 打印已创建资源的路径
Debug.Log(AssetDatabase.GetAssetPath(material));
}
```

◆static funciton AddObjectToAsset（objectToAdd：Object,assetObject：Object）：void

描述：添加 objectToAdd 到由 assObject 标识的已有资源中

请注意你应该只添加资源到 asset 资源，例如导入模型或纹理资源将在重新导入或退出时丢失它们的数据

```
@MenuItem（”GameObject/ Create Matenal”）
satic function CreateMaterial(){
// 创建一个简单材质资源
var material = new Material (Shader.Find("Specular"));
AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
```

// 给它添加一个动画剪辑
var animationClip = new AnimationClip();
animationClip.name=“My Clip”;
AssetDatabase.AdObjectToAsset（animationClip，matcrial）;
// 添加一个物体后重导入这个资源
// 否则这个改变只会显示在保持工程的时候
AssetDatabase.ImportAsset（AssetDatabase.GetAssetPath（animationClip））;
// 打印已创建资源的路径
Debug.Log(AssetDatabase.GetAssetPath(material));
}

◆ static funciton AddPathToGUID(path：string)：string

描述：获得指定 path 上的资源 GUID

◆ static funciton Contains（obj：Object）：bool

◆ static funciton Contains（instanceID：int）：bool

描述：对象是一个资源

当一个对象是一个资源的时候（对应与 Assets 文件夹中的一个文件）返回真，否则返回假（例如在场景中的物体，或在运行时创建的物体）

◆ static funciton CopyAsset（path：string，newPath：string）：bool

描述：复制在 path 上的资源并将它存储在 newPaht

◆ static funciton CreateAsset（asset：Object，path：shring）：void

描述：在路径上创建一个新的资源，你必须确保路径使用一个被支持的扩展名（材质用“mat”立方贴图用“cubemap”皮肤用“GUISkin”动画用“anim”并且其他任意的资源用“asset”）

资源被创建后你可以使用 AssetDatabaseAddObjectToAsset 添加更多资源到文件中，如果资源已经在 path 上，它将被删除并创建新的资源

@MenuItem（”GameObject/ Create Matenal”）
satic function CreateMaterial(){
// 创建一个简单材质资源
var material = new Material (Shader.Find("Specular"));
AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
// 打印已创建资源的路径
Debug.Log(AssetDatabase.GetAssetPath(material));
}

◆ static funciton DeleteAsset（path：string）：bool

描述：删除路径上的资源

如果资源被成功删除返回真，如果它不存在或者不能被移除返回假

◆ static funciton GenerateUniqueAssetPath（path：string）：string

描述：为资源产生一个新的唯一路径

◆ static funciton GetAssetPath（assetObject：Object）：string

◆ static funciton GetAssetPath（instanceID：int）：string

描述：返回相对于工程文件夹的路径名，资源被存储在哪里

@MenuItem（”GameObject/ Create Matenal”）
satic function CreateMaterial(){
// 创建一个简单材质资源

var material = new Material (Shader.Find("Specular"));
AssetDatabase.CreateAsset(material, "Assets/MyMaterial.mat");
// 打印已创建资源的路径
Debug.Log(AssetDatabase.GetAssetPath(material));
}

◆ static funciton GetCachedIcon（path：string）：Texture

描述：在给定的资源路径中取回该资源的图标

◆ static funciton GUIDToAssetPath（guid：string）：string

描述：转换 GUID 到它当前的资源路径

◆ static funciton ImportAsset（path：string，options：ImportAssetOptions=ImportAssetOptions.Default）：void

描述：导入路径上的资源

◆ static funciton IsMainAsset（obj:Object）:bool

描述：为资源产生一个新的唯一路径

◆ static funciton IsMainAsset(instaneeID:int):bool

描述：在工程窗口中，该资源是一个主资源

例如,一个导入的模型有一个游戏物体作为它的根，还有一些网格和子游戏物体，这个情况下，根游戏物体是一个主资源

◆ static funciton LoadAllAssetsAtPath（assetPath：string）：boject

描述：返回 assetPath 上的所有资源物体的数值

某些资源文件也许包含多个物体（例如一个 Maya 文件可能包含多个网格和游戏物体）assetPath 是相对于工程文件夹的路径

◆ static funciton LoadAllAssetsAtPath（assetPath：string，type：Type）：Object

描述：返回给定路径上的资源，如果它继承自 type

/assetPath 是相对工程文件夹的路径

◆ static funciton LoadMainAssetAtPath（assetPath：string）：Object

描述：返回位于 assetPath 的主资源

/assetPath 是相对工程文件夹的路径

◆ static funciton MoveAsset（oldPath：string，newPath：string）：string

参数

oldPath　　这个资源的当前路径

newPath　　资源应该被移动到的路径

返回：string　如果资源被成功移动这个是空字符串，否则是一个错误字符串

描述：从一个文件夹移动一个资源到另一个文件夹

◆ static funciton MoveAssetToTrash（path：string）：bool

描述：移动路径上的资源到回收站

如果资源被成功移除返回真，否则为假

◆static funciton OpenAsset（instanceID：int，lineNumber：int=-1）:bool

◆static funciton OpenAsset（target:Object,lineNumber：int=-1）:bool

描述：在外部编辑器中打开 target 资源，图像处理程序或者建模工具，根据资源的类型

如果是一个文本文件，lineNumber 指定文本编辑器选择那一行

◆static funciton Refresh（options：ImportAssetOptions=ImportAssetOptions，Default）：void

描述：导入任何改变的资源

导入任何已经改变了内容的或者被从工程文件夹中添加/移除的资源

◆ static funciton RenameAsset（pathName：string，newName：string）“string

参数

pathName 该资源的当前路径

newName 该资源的新名称

返回：string 如果资源被成功重命名，是一个空的字符串，否则是一个错误字符串

描述：重命名一个资源文件

◆static funciton SaveAssets（）：void

描述：将未保存的改变写入磁盘

◆static funciton StartAssetEditing（）：void

描述：开始资源导入，这可以让你组织几个资源导入为一个更大的导入操作

◆static funciton StopAssetEditing（）：void

描述：停止资源导入，这可以让你组织几个资源导入为一个的导入操作

◆static funciton ValidateMoveAsset（oldPath：string，newPath：string）：string

参数

oldPath 该资源的当前路径

newName 这个资源应该被移动到的路径

返回：string 如果资源被成功重命名，是一个空的字符串，否则是一个错误字符串

描述：检查一个资源文件是否可以被移动到另一个文件夹（并不实际移动这个文件）

参加：AssetDatabase.MoveAsset

AssettImporter

类，继承自 Object

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

变量

◆var assetPath：string

描述：用于这个导入期的资源的路径名

类方法

◆staric function GetAtPath（path：string）：AssetImporter

描述：为 path 处的资源收回资源导入期

参加：MobelImporter，TextmreLmporter，AudioImporter

继承的成员

继承的变量

Name 对象的名称

hideFlags 该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstauceID 返回该物体的实例 ID

继承的类函数

Operatorbool 这个物体存在吗

Instantlate 克隆 original 物体并返回这个克隆

Destroy 移除一个游戏物体，组件或资源

DestryImnedinte 立即销毁物体 obj，强力建议使用 Destroy 代替

FindObjectsOFType 返回所有类型为 type 的激活物体

FindObjectOFType 返回第一个类型为 type 的激活物体

Operator== 比较两个物体是否相同

Operator！= 比较两个物体是否不相同

DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁

AssetPostprocessor

类

AssetPostprocessor 让你进入导入流水线并且在导入资源之前或之后运行脚本

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这样你可以在导入设置中重载缺省的值或者修改导入的数据，如纹理和网格

变量

◆var assetImporter：AssetImporter

描述：指向资源导入期

◆var assetPath：string

描述：被导入的资源的路径名

◆var preview：Texture2D

描述：指定一个自定义的纹理到这个变量以便产生导入资源的预览

函数

◆function GetPostprocessOrder（）：int

描述：重载导入期执行的顺序

通过重载 GetPostprocessOrder 你能够排列后期处理执行的顺序，优先级较小的将首先载入

◆function LogError（warning：string，context：Object=null）：vold

描述：记录一个导入错误到控制台

传递一个资源作为第二个参数来连接这个错误到编辑器中的资源

参见：DebugLogError

◆function LogWarning（warning：string，context：Object=null）：vold

描述：记录一个导入警告道控制台

传递一个资源作为第二个参数来连接这个警告到编辑器中的资源

参见：DebugLogError

消息传递

◆function OnAssignMaterialModer（renderer：Renderer）：Material

描述：取得源材质

返回的材质将被赋予给渲染器，如果返回 null，Unity 将使用它的缺省材质找到产生方法来分配材质。soureeMaterial 在模型导入并将被销毁前，OnAssignMaterial 之后直接从模型生成。

```
class MyMeshPostprocessor extends AssetPostprocessor {
    function OnAssignMaterialModel (material : Material, renderer : Renderer) : Material
    {
        var materialPath = "Assets/" + material.name + ".mat";
        // 查找在材质路径上是否有一个材质
        // 关闭这个以便总是产生新材质
```

```
            if (AssetDatabase.LoadAssetAtPath(materialPath))
                return AssetDatabase.LoadAssetAtPath(materialPath);
            // 使用 specular shader 创建一个新的资源
            // 其他默认值来自模型
            material.shader = Shader.Find("Specular");
            AssetDatabase.CreateAsset(material, "Assets/" + material.name + ".mat");
            return material;
        }
    }
```

◆ function OnPostprocessAllAssets （ importedAssets ： string[],deletedAssets:string[],movedAssets:string[],movedFromPath:string[]）:void

描述：OnPostprocessAllAssets 在一些资源被导入后调用（资源进度栏到达末端）

```
class MyAllPostprocessor extends AssetPostprocessor {
    static function OnPostprocessAllAssets (
        importedAssets : String[],
        deletedAssets : String[],
        movedAssets : String[],
        movedFromAssetPaths : String[])
    {
        for (var str in importedAssets) {
            Debug.Log("Reimported Asset: " + str);
        }
        for (var str in deletedAssets) {
            Debug.Log("Deleted Asset: " + str);
        }
        for (var i=0;i<movedAssets.Length;i++) {
            Debug.Log("Moved Asset: " + movedAssets[i] + " from: " + movedFromAssetPaths[i]);
        }
    }
}
```

◆ function OnPostprocessAudio （clip：AudioClip）：void

描述:

◆ function OnPostprocessModel （root：GameObject）：void

描述：在子类中重载这个函数以便在模型完全导入后获得通知

在一个预设被生成为游戏物体层次前，root 是导入模型的根物体

```
class MyModelPostprocessor extends AssetPostprocessor {
    function OnPostprocessModel (g : GameObject) {
        Apply(g.transform);
    }
    // 添加网格碰撞器到每个名称中包含 collider 的游戏物体上
    function Apply (transform : Transform)
    {
        if (transform.name.ToLower().Contains("collider"))
```

```
            {
                transform.gameObject.AddComponent(MeshCollider);
            }

            // 循环
            for (var child in transform)
                Apply(child);
        }
    }
```

◆ function OnPostprocessTexture （texture：Teture2D）：void

描述：在子类中重载这个函数以便在纹理被完全导入并被存储到磁盘上之前获取一个通知

```
    //后期处理所有防止文件夹
    // "invert color" 中的文件，反转他们的颜色
    class InvertColor extends AssetPostprocessor {
        // 使用这个初始化
        function OnPostprocessTexture (texture : Texture2D) {
            // 后期处理之灾文件夹
            // "invert color" 或它的子文件夹中的文件
            // var lowerCaseAssetPath = assetPath.ToLower();
            // if (lowerCaseAssetPath.IndexOf ("/invert color/") == -1)
              //return;
            for (var m=0;m<texture.mipmapCount;m++)
            {
                var c : Color[] = texture.GetPixels(m);
                for (var i=0;i<c.Length;i++)
                {
                    c[i].r = 1 - c[i].r;
                    c[i].g = 1 - c[i].g;
                    c[i].b = 1 - c[i].b;
                }
                texture.SetPixels(c, m);
            }
            // 不需要设置每个 mip map 等级图片的像素你也可以只修改最高 mip 等级的像素并使用 texture.Apply（TRUE）；来产生较低 mip 等级
        }
    }
```

◆ function OnPreprocessAudio （）：void

描述：

◆ function OnPreprocessModel（）：void

描述：在子类中重载这个函数以便在模型被导入前获得一个通知

这个可以让你为模型的导入设置默认值

```
    class MyMeshPostprocessor extends AssetPostprocessor {
        function OnPreprocessModel () {
```

```
        // 禁用材质生成，如果文件包含@号，表明他是动画
        if (assetPath.Contains("@")) {
            var modelImporter : ModelImporter = assetImporter;
            modelImporter.generateMaterials = 0;
        }
    }
}
```

◆ function OnPostprocessTexture（）：void

描述：在子类中重载这个函数以便在纹理导入器运行前获得一个通知，
这个可以让你导入设置为默认值

```
// Postprocesses all textures that are placed in a folder
// "invert color" to have their colors inverted.
class InvertColor extends AssetPostprocessor {
    // Use this for initialization
    function OnPostprocessTexture (texture : Texture2D) {
        // Only post process textures if they are in a folder
        // "invert color" or a sub folder of it.
//          var lowerCaseAssetPath = assetPath.ToLower();
//          if (lowerCaseAssetPath.IndexOf ("/invert color/") == -1)
//              return;
        for (var m=0;m<texture.mipmapCount;m++) {
            var c : Color[] = texture.GetPixels(m);
            for (var i=0;i<c.Length;i++)
            {
                c[i].r = 1 - c[i].r;
                c[i].g = 1 - c[i].g;
                c[i].b = 1 - c[i].b;
            }
            texture.SetPixels(c, m);
        }
        // Instead of setting pixels for each mip map levels, you can also
        // modify only the pixels in the highest mip level. And then simply use
        // texture.Apply(true); to generate lower mip levels.
    }
}
```

AudioImporter

类

继承自 AssetImporter

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这个类的设置于 Audio Import Settings 中相同

变量

◆var channels：AudioImporterChannels

描述：导入音频的声道数

参见：AudioImporterChannels.

◆var compressionBitrate：float

描述：Ogg Verbix 压缩比特率

这个值以比特率为对单位，例如：128000 应该是 128kbps

◆var decompressOnLoad：bool

描述：Ogg Verbix 音频应该在加载时解压

◆var format：AudioImporterFormat.

描述：导入音频的格式

参见：AudioImporterFormat.

继承的成员

继承的变量

assetPath	
name	对象的名称
hideFlags	该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID	返回该物体的实例 ID

继承的类函数

GetAtpath	为 path 处的资源取回资源导入器
Operatorbool	这个物体存在吗
Instannate	克隆 original 物体并返回这个克隆
Desiroy	移除一个游戏物体，组件或资源
DestroyImmediate	立即销毁物体 obj，强力建议使用 Destroy 代替
FindObjectsOFType	返回所有类型为 type 的激活物体
FindObjectOFType	返回第一个类型为 type 的激活物体
Operator==	比较两个物体是否相同
Operator！=	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 target 不被自动销毁

ModelImporter

类

继承自 AssetImporter

ModerlImporter 让你从脚本编辑器中修改 model 的导入设置

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这个类的设置与 Mesh Import Settings 中相同

变量

◆var addCollider：bool

描述：为导入的网格添加网格编辑器

◆var bakeIK：bool

描述：导入时烘焙 IK

◆var clipAnimations：ModelImporterClipAnimation[]

描述：风格动画得到的动画剪辑

参见 splitAnimation，ModelImporterClipAnimation
◆var generateAnimations：ModelImporterGenerateAnimations
描述：动画生成选项
参见 ModelImporterGenerateAnimations
◆var generateMaterials：ModelImporterGenerateMaterials
描述：材质生成选项
参见 ModelImporterGenerateMaterials
◆var globalScale：float
描述：用于导入的全局缩放因子
◆var normalSmoothingAngle：float
描述：平滑角度来计算发现
计算法线时，尖锐的边缘怎样被变成一个硬边
参见：recalculateNormals
◆var recalculateNormals：bool
描述：导入时是否重新计算法线
参见：normalSmoothingAngle
◆var reduceKeyframes：bool
描述：为动画执行关键帧缩减
◆var splitAnimations：bool
描述：导入时动画是否应嘎被分割为多个剪辑
参见：clipAnimations
◆var splitTangentsAcrossSeams：bool
描述：切线是否跨越 uv 接缝分割
◆var swapUVChannels：bool
描述：导入时切换主副 UV 通道
继承的成员
继承的变量
assetPath
name 对象的名称
hideFlags 该物体是否被隐藏，保存在场景中或被用户修改
继承的函数
GetInstanceID 返回该物体的实例 ID
继承的类函数
GetAtpath 为 path 处的资源取回资源导入器
operatorbool 这个物体存在吗
Instannate 克隆 original 物体并返回这个克隆
Desiroy 移除一个游戏物体，组件或资源
DestroyImmediate 立即销毁物体 obj，强力建议使用 Destroy 代替
FindObjectsOFType 返回所有类型为 type 的激活物体
FindObjectOFType 返回第一个类型为 type 的激活物体
Operator== 比较两个物体是否相同
Operator！= 比较两个物体是否不相同
DontDestroyOnLoad 加载新场景时确保物体 target 不被自动销毁
TextureImporter

类

继承自 AssetImporter

纹理导入器可以让你从编辑器脚本中修改 Texture2D 的导入设置

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这个类的设置与 Texture Import Settings.中相同

变量

◆var borderMipmap：bool

描述：保持产生 mipmap 时的相同纹理边界

◆var convertToNormalmap：bool

描述：转化高度图为法线贴图

◆var correetGamma：bool

描述：mipmap 应使用伽马校正生成

参见：mipmapEnabled

◆var fadeout：bool

描述：淡出 mip 等级为灰色

参见：mipmapEnabled

◆var generateCubemap：TextureImporterGeneraterCubemap

描述：立方贴图生成模式

参见：TextureImporterGeneraterGubemap

◆var grayscaleToAlpha：float

描述：从灰度生成 alpha 通道

◆var heightmapScal：int

描述：最大纹理尺寸

较大的纹理将在导入时被缩小

◆var mipmapEnabled：bool

描述：为这个纹理生成 Mipmap

◆var mipmapFadeDistanceEnd：float

描述：纹理完全淡出的 mip 等级

参见：mipmapEnabled，fadeout，mipmapFadeDistanceStart

◆var mipmapFadeDistanceStart：float

描述：纹理开始淡出的 mip 等级

参见：mipmapEnabled，fadeout，mipmapFadeDistanceStart

◆var mipmapFilter：TextureImporterMipFilter

描述：Mipmap 过滤模式

参见：TextureImporterNormalFilter，mipmapEnabled

◆var normalmapFilter： TextureImporterNormalFilter

描述：法线图过滤模式

参见：TextureImporterNormalFilter，convertTONormalmap

◆var npotSeale：TextureImporterNPOTScale

描述：非 2 的幂次尺寸纹理的缩放模式

参见：TextureImporterNPOTScale

◆var recommendedTextureFormat：TextureImporterFormat

描述：自动决定最好的纹理格式（只读）

参见：TextureImporterFormat

◆var textureFormat：TextureImporterFormat

描述：导入纹理的格式　　参见：TextureImporterFormat

继承的成员

继承的变量

assetPath	
name	对象的名称
hideFlags	该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID	返回该物体的实例 ID

继承的类函数

GetAtpath	为 path 处的资源取回资源导入器
operatorbool	这个物体存在吗
Instannate	克隆 original 物体并返回这个克隆
Desiroy	移除一个游戏物体，组件或资源
DestroyImmediate	立即销毁物体 obj，强力建议使用 Destroy 代替
FindObjectsOFType	返回所有类型为 type 的激活物体
FindObjectOFType	返回第一个类型为 type 的激活物体
Operator==	比较两个物体是否相同
Operator！=	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 target 不被自动销毁

AssetPostprocessor

类

AssetPostprocessor 让你进入导入流水线并在导入资源之前或之后运行脚本

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这样你可以在导入设置中重载缺省的值或则修改导入的数据，如贴图和网格

变量

◆var assetImporter：AssetImporter

描述：指向资源导入器

◆var assetPath：string

描述：被导入的资源的路径名

◆var preview：Texture2D

描述：指定一个自定义的纹理到这个变量以便产生导入资源的预览

函数

◆function GetPostprocessOrder（）：int

描述：重载导入器执行的顺序

通过重载 GetPostprocessOrder 你能够排列后期处理执行的顺序，优先级较小的将首先载入

◆function LogError（warning：string，context：Object=null）：vold

描述：记录一个导入错误到控制台

传递一个资源作为第二个参数来连接这个错误到编辑器中的资源

参见：DebugLogError

◆function LogWarning（warning：string，context：Object=null）：vold

描述：记录一个导入警告道控制台

传递一个资源作为第二个参数来连接这个警告到编辑器中的资源

参见：DebugLogError

消息传递

◆function OnAssignMaterialModer（renderer：Renderer）：Material

描述：取得源材质

返回的材质将被赋予给渲染器，如果返回 null，Unity 将使用它的缺省材质找到产生方法来分配材质。soureeMaterial 在模型导入并将被销毁前，OnAssignMaterial 之后直接从模型生成。

```
class MyMeshPostprocessor extends AssetPostprocessor {
    function OnAssignMaterialModel (material : Material, renderer : Renderer) : Material
    {
        var materialPath = "Assets/" + material.name + ".mat";

        // 查找在材质路径上是否有一个材质
        // 关闭这个以便总是产生新材质
        if (AssetDatabase.LoadAssetAtPath(materialPath))
            return AssetDatabase.LoadAssetAtPath(materialPath);

        // 使用 specular shader 创建一个新的资源
        // 其他默认值来自模型
        material.shader = Shader.Find("Specular");
        AssetDatabase.CreateAsset(material, "Assets/" + material.name + ".mat");
        return material;
    }
}
```

◆ function OnPostprocessAllAssets （ importedAssets ：string[],deletedAssets:string[],movedAssets:string[],movedFromPath:string[]）:void

描述：OnPostprocessAllAssets 在一些资源被导入后调用（资源进度栏到达末端）

```
class MyAllPostprocessor extends AssetPostprocessor {
    static function OnPostprocessAllAssets (
        importedAssets : String[],
        deletedAssets : String[],
        movedAssets : String[],
        movedFromAssetPaths : String[])
    {
        for (var str in importedAssets) {
            Debug.Log("Reimported Asset: " + str);
        }
        for (var str in deletedAssets) {
            Debug.Log("Deleted Asset: " + str);
        }
```

```
            for (var i=0;i<movedAssets.Length;i++) {
                Debug.Log("Moved Asset: " + movedAssets[i] + " from: " + movedFromAssetPaths[i]);
            }
        }
    }
```

◆ function OnPostprocessAudio （clip：AudioClip）：void

描述:

◆ function OnPostprocessGameObjectWithUserProperties (root : GameObject, propNames : string[], values : object[]) : void

描述：在导入文件时，为每个至少附加了一个用户属性的游戏物体调用

propNames 是一个 string[ ]，其中包含了所有找到的属性的名称，该值是一个 object[ ]，包含了所有实际的值，这个可以是 Vector4, bool, string, Color, float, int.类型

典型的运用是从存储在 3dmax 中的对象中读取“用户数据”基于什么用户数据被写入到对象，你可以决定以不同的方式来后处理游戏物体。下面的列子中，如果用户数据字符串包含“addboxcollider”添加一个简单的 BoxCollider 组件

```
lass MyPostprocessor extends AssetPostprocessor {
    function OnPostprocessGameObjectWithUserProperties (
        go : GameObject,
        propNames : String[],
        values : System.Object[]
        )
    {
        for (var i : int =0; i!= propNames.Length; i++)
        {
            var propName : String = propNames[i];
            var value : Object = values[i];
            Debug.Log("Propname: "+propName+" value: "+values[i]);
            if (value.GetType() == String)
            {
                var s : String = value;
                if (s.Contains("addboxcollider")) go.AddComponent(BoxCollider);
            }
            if (value.GetType() == Vector4)
 {
                var v : Vector4 = value;
                // do something useful.
            }
            if (value.GetType() == Color)
            {
                var c : Color = value;
                // do something useful.
            }
```

```
            if (value.GetType() == int)
            {
                var myInt : int = value;
                // do something useful.
            }
            if (value.GetType() == float)
            {
                var myFloat : float = value;
                // do something useful.
            }
        }
    }
}
```

◆ function OnPostprocessModel （root：GameObject）：void

描述：在子类中重载这个函数以便在模型完全导入后获得通知

在一个预设被生成为游戏物体层次前，root 是导入模型的根物体

```
class MyModelPostprocessor extends AssetPostprocessor {
    function OnPostprocessModel (g : GameObject) {
        Apply(g.transform);
    }
    // 添加网格碰撞器到每个名称中包含 collider 的游戏物体上
    function Apply (transform : Transform)
    {
        if (transform.name.ToLower().Contains("collider"))
        {
            transform.gameObject.AddComponent(MeshCollider);
        }
        // 循环
        for (var child in transform)
            Apply(child);
    }
}
```

◆ function OnPostprocessTexture （texture：Teture2D）：void

描述：在子类中重载这个函数以便在纹理被完全导入并被存储到磁盘上之前获取一个通知

```
//后期处理所有放置文件夹
// "invert color" 中的文件，反转他们的颜色
class InvertColor extends AssetPostprocessor {
    // 使用这个初始化
    function OnPostprocessTexture (texture : Texture2D) {
        // 后期处理只在文件夹
        // "invert color" 或它的子文件夹中的文件
  // var lowerCaseAssetPath = assetPath.ToLower();
        // if (lowerCaseAssetPath.IndexOf ("/invert color/") == -1)
```

```
            //return;
            for (var m=0;m<texture.mipmapCount;m++)
            {
                var c : Color[] = texture.GetPixels(m);
                for (var i=0;i<c.Length;i++)
                {
                    c[i].r = 1 - c[i].r;
                    c[i].g = 1 - c[i].g;
                    c[i].b = 1 - c[i].b;
                }
                texture.SetPixels(c, m);
            }
            // 不需要设置每个 mip map 等级图片的像素你也可以只修改最高 mip 等级的像素并使用 texture.Apply（TRUE）；来产生较低 mip 等级
        }
    }
```

◆ function OnPreprocessAudio （）：void

描述：

◆ function OnPreprocessModel（）：void

描述：在子类中重载这个函数以便在模型被导入前获得一个通知

这个可以让你为模型的导入设置默认值

```
class MyMeshPostprocessor extends AssetPostprocessor {
    function OnPreprocessModel () {
        // 禁用材质生成，如果文件包含@号，表明他是动画
        if (assetPath.Contains("@")) {
            var modelImporter : ModelImporter = assetImporter;
            modelImporter.generateMaterials = 0;
        }
    }
}
```

◆ function OnPostprocessTexture（）：void

描述：在子类中重载这个函数以便在纹理导入器运行前获得一个通知，

这个可以让你导入设置为默认值

```
class MyTexturePostprocessor extends AssetPostprocessor
{
    function OnPreprocessTexture () {
        // 自动转化任何文件名中带有 "_bumpmap"的纹理文件为法线贴图

        if (assetPath.Contains("_bumpmap")) {
            var textureImporter : TextureImporter = assetImporter;
            textureImporter.convertToNormalmap = true;
        }
    }
}
```

BuildPipeline

类

类方法

◆ **static function BuildAssetBundle (mainAsset : Object, assets; Object[].pathName : string,options ; BuildAssetBundleOptions = BuildAssetBundleOptions CollectDependencies**

BuildAssetBundleOptionsCompleteAssets) : bool

描述：构建资源 **bundle(Unity Pro only).**

创建一个包含 **assets** 集合的压缩 **Unity3D** 文件。**AssetBundles** 可以包含工程文件夹中的任何资源，这可以让你流式下载任何类型的资源数据，完整设置的预设，纹理，网格，动画，显示在工程窗口中的任何类型的资源。**mainAsset** 让你指定一个特定的物体，它可以方便地使用 **AssetBundle.mainAsset** 取回。这个压缩的资源 **bundle** 文件将被存储在 **pathName.options** 允许你自动地包含依赖性或者总是包含完整的资源而不仅仅是引用的对象。

参见：**AssetBundle** 类，**WWW.assetBundle.**

◆ **Static function BuildPlayer (levels : string[],locationPathName : string,target:BuildTarget,**

Options : BuildOptions) : string.

描述：构建播放器**(Unity Pro only).**

/levels/是被包含在构建中的场景，（如果 **levels** 为空，当前打开的场景将被构建）**locationPathName** 是应用程序将被保存的路径，**target** 是将被构建的 **BuildTarget, options** 是一些额外的选项，如纹理压缩，调试信息剥离。

◆ **Static function PopAssetDependencies():void**

描述：让你管理不同资源 **bundle** 和播放器构建之间的交叉引用和依赖性，

如果一个资源 **bundles** 依赖于另一个资源 **dundles**，确保依赖的资源 **bundles** 通过

WWW 类加载是你的责任。

当你放入资源依赖性时它将共享那个层上的所有资源，放入递归地继承前一个依赖关系，**PushAssetDependencies** 和 **PopAssetDependencies** 必须成对出现。

@\lenulten(“Assets/Auto Build Asset Bundles”)

Static function ExportResource(){

// 对所有下面的资源 bundles 文件启用交叉引用直到 // 我们调用 PopAssetDependencies

QuikPipcliPushAssetDcpenclencies();

Var options=

BuildAssetBundleOptions CollectDependencies]

BuildAssetBundleOptions CompleteAssets;

//所有后续资源共享这个资源 bundles 中的资源

//它是由你来确保共享的资源 bundle

//优先于其他资源加载

Buldpipeline.BuldAssertBundle(

AssetDatabase.LoadMainAssetAtPath(“assets/artwork/lerpzuv.tif”,

null,”Sharde.unity3d”,options);

//通过收入和弹出资源 bundle,这个文件

//将共享资源，但是后面的资源 bundle 将不会共享这个资源中的资源

BuildPipeline.PushAssetDependencies();

BuildPipeline.PushAssetBundle(
AssetDatabaseLoadMainAssetAtPath(“Asset Artwork/Lerpztbx’),
Null,”Lerpz.unity3d”,options);
BuildPipeline.PushAssetDependencies();
BuildPipeline.PushAssetDependencies();
BuildPipeline.PushAssetDependencies();
}

参见：PushAssetDependencies,BuidAssetBundle.

DragAndDrop

类

编辑器拖放操作。

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor，”。

类变量

◆ Static var activeControlID : int

描述：

◆ Static var objectReferences :Object[]

描述：被拖动的 objects 的引用。

没有返回 null,如果没有物体引用可用返回一个空的数组。

参见：paths,PrepareStartDrag,StartDrag,

◆ Static var paths :string[]

描述：被拖放的文件名

没有返回 null。如果没有路径可用返回一个空的数组。

参见： objectReferences, PrepareStartDrag,StartDrag,.

◆ Static var visualMode:DragAndDropVisualMode

描述：拖放的视觉标志

默认为 DragAndDropVisualMode Link

function OnGUI ()
{
var eventType = Event.current.type;
if (eventType == EventType.DragUpdated || eventType == EventType.DragPerform)
{
// Show a copy icon on the drag
DragAndDrop.visualMode = DragAndDropVisualMode.Copy;
if (eventType == EventType.DragPerform)
DragAndDrop.AcceptDrag();
Event.current.Use();
}
}

类方法

◆ Static function AcceptDrag() : void

描述：接收拖动操作。

◆ Static function GetGenericData(type : string) : object

描述：

◆ Static function PrepareStartDrag() : void

描述：清除拖放数据。

清楚所有存储在拖放物体中的数据并准备它，这样你可以为初始化一个拖动操作写入它。

参见：StartDrag,paths,objectReferences.

```
function OnGUI ()
{
if (Event.current.type == EventType.MouseDrag)
{
// 清除拖动数据
DragAndDrop.PrepareStartDrag ();
// 设置我们需要拖动什么
DragAndDrop.paths = { "/Users/joe/myPath.txt" };
// 开始拖动
DragAndDrop.StartDrag ("Dragging title");
// 确保我们之后没有人使用这个事件
Event.current.Use();
}
}
```

◆ Static function SetGenericData(type:string,data:object) : void

描述：

◆ Static function StartDrag(title:string): void

描述：开始拖动操作。

用当前拖动物体状态初始化一个拖动操作。使用 paths 和/或 objectReferences 来设置拖动状态。

参见：PrepareStartDrag,paths,objectReferences.

```
function OnGUI ()
{
if (Event.current.type == EventType.MouseDrag)
{
// 清除拖动数据
DragAndDrop.PrepareStartDrag ();
// 设置我们需要拖动什么
DragAndDrop.paths = { "/Users/joe/myPath.txt" };
// 开始拖动
DragAndDrop.StartDrag ("Dragging title");
// 确保我们之后没有人使用这个事件
Event.current.Use();
}
}
```

DrawGizmo

类，继承自 System Attribute

DrawGizmo 属性允许你为任何类型的 Compartent 提供一个 gizmo 渲染器。

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor”.

目前，z 你只能为引擎组件提供 gizmo 绘制，所有的 gizmo 绘制方法需要是静态的，

```
//如果这个光源没有被选择将调用 RenderLightGizmo 函数
//当拾取时 gizmo 被绘制
@DrawGizmo (GizmoType.NotSelected | GizmoType.Pickable)
static function RenderLightGizmo (light : Light, gizmoType : GizmoType)
{
var position = light.transform.position;
// 绘制光源图标
// 在内置的光源 Gizmo 上面一点)
Gizmos.DrawIcon (position + Vector3.up, "Light Gizmo.tiff");
// 我们选择了吗? Draw a solid sphere surrounding the light
if ((gizmoType & GizmoType.SelectedOrChild) != 0)
{
// 使用一个较为明亮的颜色表明这是一个活动对象.
if ((gizmoType & GizmoType.Active) != 0)
Gizmos.color = Color.red;
else
Gizmos.color = Color.red * 0.5;
Gizmos.DrawSphere (position, light.range);
}
}
/*
//如果它被选择或者它的子被选择绘制这个 gizmo.
// 这是最常用的渲染一个 gizmo 的方法
@DrawGizmo (GizmoType.SelectedOrChild)
//只有它是激活物体时绘制这个 gizmo.
@DrawGizmo (GizmoType.Active)]
*/
/// C# 例子
using UnityEditor;
using UnityEngine;
class GizmoTest
{
/// 如果这个光源没有被选择调用 RenderLightGizmo 函数.
/// 当拾取时 gizmo 被绘制.
[DrawGizmo (GizmoType.NotSelected | GizmoType.Pickable)]
static void RenderLightGizmo (Light light, GizmoType gizmoType)
{
Vector3 position = light.transform.position;
// 绘制光源图标
// (在内置的光源 Gizmo 上面一点)
```

```
Gizmos.DrawIcon (position + Vector3.up, "Light Gizmo.tiff");
// 我们选择了吗？围绕光源绘制一个球体
if ((gizmoType & GizmoType.SelectedOrChild) != 0)
{
//使用一个较为明亮的颜色表明这是一个活动对象.
if ((gizmoType & GizmoType.Active) != 0)
Gizmos.color = Color.red;

else
Gizmos.color = Color.red * 0.5F;
Gizmos.DrawSphere (position, light.range);
}
}
}
/*
//如果它被选择或者它的子被选择绘制这个 gizmo.
// 这是最常用的渲染一个 gizmo 的方法
 [DrawGizmo (GizmoType.SelectedOrChild)]
//只有它是激活物体时绘制这个 gizmo.
[DrawGizmo (GizmoType.Active)]
*/
```

构造函数

◆ Static function DrawGizmo（gizmo: GizmoType）: DrawGizmo

描述：定义何时 Gizmo 应该被调用以便绘制。

参见： GizmoType

◆ Static function DrawGizmo（gizmo: GizmoType,drawnGizmoType:Type）:DrawGizmo

描述：同上，drawnGizmoType 决定要绘制的 Gizmo 物体是什么类型的。

如果 drawnGizmoType 为 null,这个类型将由该函数的第一个参数决定。

Editor

类，继承自 ScriptableObject

一个基类，可以从它派生自定义的编辑器，使用这个给你的物体创建自定义的检视面板和编辑器。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor，”。

通过使用 CustomEditor 属性可以附加 Editor 到一个自定义的组件。

JavaScript 例子：

```
//这个例子为一个”MyPlayer”物体显示一个自定义的检视面板。
它有两个变量，speed 和 ammo.
@CustomEditor(MyPlayer)
class MyPlayerEditor extends Editor
{
function OnInspectorGUI()
{
```

```
EditorGUILayout.BeginHorizontal();
EditorGUILayout.PrefixLabel("Speed");
target.speed = EditorGUILayout.Slider(target.speed, 0, 100);
EditorGUILayout.EndHorizontal();
EditorGUILayout.BeginHorizontal();
EditorGUILayout.PrefixLabel("Ammo");
target.ammo = EditorGUILayout.IntField(target.ammo);
EditorGUILayout.EndHorizontal();
}
}
```

变量

- **Var target:Object**

描述：被检视的对象

消息传递

- **Function OnInspectorGUI():void**

描述：实现这个函数来制作一个自定义的检视面板。

- **Function OnSceneGUI() : void**

描述：在场景视图中让编辑器处理一个事件。

在 **OnSceneGUI** 中你可以做，例如网格编辑，地形绘制或高级的 **gizmos**，如果调用 **Eventcurrent,Use(),**该事件将被编辑处理并不会被场景视图使用。

继承的成员

继承的变量

name	对象的名称
hideFlags	该物体是否被隐藏，保存在场景中或被用户修改

继承的函数

GetInstanceID	返回该物体的实例 **ID**

继承的消息传递

OnEnable	物体被加载时调用该函数
OnDisable	当可编辑物体超出范围时调用这个函数

继承的类函数

CreateInstance	使用 **className** 创建一个可编程物体的实例。
operatorbool	这个物体存在吗
Instannate	克隆 **original** 物体并返回这个克隆
Destroy	移除一个游戏物体，组件或资源
DestroyImmediate	立即销毁物体 **obj**，强力建议使用 **Destroy** 代替
FindObjectsOFType	返回所有类型为 **type** 的激活物体
FindObjectOFType	返回第一个类型为 **type** 的激活物体
Operator==	比较两个物体是否相同
Operator！=	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 **target** 不被自动销毁。

EditorApplication

类

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中，编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加”**using**

UnityEditor，”。

类变量

◆ Static var applicationContentsPath:string

描述：Unity 编辑器内容文件的路径（只读）

这个内容文件夹包含用来构建播放器的一些内部文件。

◆ Static var applicationPath:string

描述：返回 Unity 编辑器的路径（只读）

参见：applicationContentsPath

◆ Static var currentScene:string

描述：用户当前打开的场景的路径（只读）。

◆ Static var isPaused : bool

描述：编辑器当前暂停？

可编辑地改变暂停状态，就像按下主工具栏上的 Pause 按钮。

参见：isPlaying.

◆ Static var isPlaying : bool

描述：编辑器当前处于播放模式？

isPaying 的设置会延迟直到到该帧中所有的脚本代码已经完成。

◆ Static var isPlayingOrWillChangePlaymode : bool

描述：编辑器当前处于播放模式，或者将切换到播放模式？（只读）

当编辑器在完成某些任务（例如，在脚本被重编译之后）后将切换到播放模式时，这个将返回真。

参见：isPlaying，isCompiling.

类方法

◆ Static function Beep():void

描述：播放系统哔哔声

```
//在鼠标按下事件中播放声音
Function OnGUI(){
   if (Event.current.type == EventType.MouseDrag)
         EditorApplication.Bceo();
}
```

◆ Static function Exit(returnValue:int):void

描述：退出 Unity 编辑器。

调用这个函数将立即退出，不会询问保存改变，因此你将丢失数据！

◆ Static function LockReloadAssemblies():void

描述：当不方便的时候阻止部件的加载。

例如，在拖动操作的时候，你也许想阻止部件的重加载以便在拖动过程中不会丢失状态。

每个 LockReloadAssemblies 必须与 UnLockReloadAssemblies 匹配，否则脚本将不会被卸载。Unity 自动在鼠标按下的时候阻止重加载。

参见：EditorApplication. UnLockReloadAssemblies

◆ Static function NewScene():void

描述：创建新的场景。

◆ Static function OpenScene(path:string):bool

描述：打开位于 Path 的场景。

当前打开的场景不会被保存，使用 SaveSceneIfUserWantsTo 来做这个。

◆ Static function OpenSceneAdditive(path:string):void

描述：打开位于 path 的附加场景。

◆ Static function SaveAssets ():void

描述：保存所有还没有写到磁盘的可序列化资源（例如，材质）

也确保资源数据库被写入。

◆ Static function SaveCurrentSceneIfUserWantsTo():bool

描述：询问用户是否要保存打开的场景。

你可能想在打开其他场景或创建一个新的场景时调用这个函数。返回真表示你想继续。返回为假表示用户要取消这个操作，因此不应该打开其他场景。

◆ Static function SaveScene(path:string):bool

描述：保存场景到 path.

◆ Static function Step():voidl

描述：执行单帧步进。

就像你按下了主打工具栏上的 Step 按钮。

◆ Static function UnLockReloadAssemblies ():voidl

描述：必须在 LockReloadAssemblies 之后调用，来重启用集合的加载。

参见：EditorApplication, LockReloadAssemblies

EditorGUILayout

类

EditorGUI 的自动布局版本

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor，”。

类方法

◆ Static function BeginHorizontal(params options:GUILayoutOption[]):Rect

◆ Static function BeginHorizontal(style:GUIStyle, params options:GUILayoutOption[]):Rect

参数

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置

参见：GUIlayout,Width,GUILayout,Height,GUILayout MinWidth,
GUILayout MaxWidth, GUILayout MinHeight, GUILayout MaxHeight
GUILayout ExpandWidth, GUILayout ExpandHeight,

描述：开始一个水平组并获取它的矩形。

这是 GUILayout BeginHorizontal 的扩展，用来制作组合控件。

```
//组合按钮
Rect r = EditorGUILayout.BeginHorizontal ("Button");
if (GUI.Button (r, GUIContent.none))
Debug.Log ("Go here");
GUILayout.Label ("I'm inside the button");
GUILayout.Label ("So am I");
EditorGUILayout.EndHorizontal ();
```

◆ Static function BeginScrollView (scrollPosition : Vector2, params options:GUILayoutOption[]):Vector2

◆ Static function BeginScrollView (scrollPosition : Vector2, alwaysShowHorizontal:bool,alwaysShowVertical:bool, params options:GUILayoutOption[]):Vector2

◆ Static function BeginScrollView (scrollPosition : Vector2, horizontalScrollbar::GUIStyle,verticalScrollbar:GUIStyle, params options:GUILayoutOption[]):Vector2

static function BeginScrollView (scrollPosition : Vector2, alwaysShowHorizontal : bool, alwaysShowVertical : bool, horizontalScrollbar : GUIStyle, verticalScrollbar : GUIStyle, params options : GUILayoutOption[]) : Vector2

参数

scrollPosition 用来显示的位置

alwaysShowHorizontal 可选的参数用来总是显示水平滚动条，如果为黑线留空，它就只在 ScrollView 中的内容比滚动提高时显示。

alwaysShowVertical 可选的参数用来总是显示垂直滚动条，如果为黑线留空，它就只在 ScrollView 中的内容比滚动提高时显示。

horizontalScrollbar 用于水平滚动条的可选 GUIStyle,如果不设置，将使用当前 GUISkin 的 horizontalScrollbar

verticalScrollbar 用于垂直滚动条的可选 GUIStyle，如果不设置，将使用当前 GUISkin 的 verticalScrollbar 风格。

返回：Vector2,修改过的 scrollPosition,回传这个变量，如下的例子。

描述：开始有个自动布局滚动条。

这个就像 GUILayout BeginScrollView 一样，但是更像应用程序且应该在编辑器中使用。

◆ static function BeginToggleGroup (label : string, toggle : bool) : bool

◆ static function BeginToggleGroup (label : GUIContent, toggle : bool) : bool

参数

label 显示在开关控件上的标签

toggle 开关组的启用状态

返回 bool – 用户选择的启用状态。

描述：开始一个带有开关的组，以便一次启用或禁用其中的所有控件。

参见：EndToggleGroup

```
// 设置
settingsOn = EditorGUILayout.BeginToggleGroup("Settings", settingsOn);
backgroundColor = EditorGUILayout.ColorField(backgroundColor);
soundOn = EditorGUILayout.Toggle(soundOn, "Turn on sound");
EditorGUILayout.EndToggleGroup();
```

◆ static function BeginVertical (params options : GUILayoutOption[ ]) : Rect

◆ static function BeginVertical (style : GUIStyle, params options : GUILayoutOption[ ]) :Rect

参数

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定程序的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight.

描述：开始一个垂直组并获取它的矩形。

这是 GUILayout.BeginVertical 的扩展，用来制作组合控件。

```
// 组合按钮
Rect r = EditorGUILayout.BeginVertical ("Button");
if (GUI.Button (r, GUIContent.none))
Debug.Log ("Go here");
GUILayout.Label ("I'm inside the button");
GUILayout.Label ("So am I");
EditorGUILayout.EndVertical ();
```

◆ static function ColorField (value : Color, params options : GUILayoutOption[ ]) : Color

◆ static function ColorField (label : string, value : Color, params options : GUILayoutOption[ ]) : Color

◆ static function ColorField (label : GUIContent, value : Color,
params options : GUILayoutOption[ ]) : Color

参数

Label 显示在该域前面的可选标签。

Value 用于编辑的颜色

Options 一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：Color- 用户选择的颜色

描述：制作一个用来选择 Color 的域

◆ static function EndHorizontal () : void

描述：关闭一个开始于 BeginHorizontal 的组

◆ static function EndScrollView () : void

描述：结束开始于 BeginScrollView 的滚动视。

◆ static function EndToggleGroup () : void

描述：关闭一个开始于 BeginToggleGroup

```
// Settings
settingsOn = EditorGUILayout.BeginToggleGroup("Settings", settingsOn);
backgroundColor = EditorGUILayout.ColorField(backgroundColor);
soundOn = EditorGUILayout.Toggle(soundOn, "Turn on sound");
EditorGUILayout.EndToggleGroup();
```

◆ static function EndVertical () : void

描述：关闭一个开始于 BeginVertical 的组

◆ static function EnumPopup (selected : System.Enum, params options :

GUILayoutOption[]) : System.Enum

◆ static function EnumPopup (selected : System.Enum, style : GUIStyle, params options : GUILayoutOption[ ]) : System.Enum

◆ static function EnumPopup (label : string, selected : System.Enum, params options : GUILayoutOption[ ]) : System.Enum

◆ static function EnumPopup (label : string, selected : System.Enum, style : GUIStyle, params options : GUILayoutOption[ ]) : System.Enum

◆ static function EnumPopup (label : GUIContent, selected : System.Enum, params options : GUILayoutOption[ ]) : System.Enum

◆ static function EnumPopup (label : GUIContent, selected : System.Enum, style : GUIStyle, params options : GUILayoutOption[ ]) : System.Enum

参数

label 该域前面可选的标签

selected 该域显示的选项

style 可选的 GUIStyle

options 一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 Sryle 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：floar-被用户选择的选项

描述：制作一个弹出式选择域

使用当前选择的值作为参数并返回用户选择的值。

◆ static function FloatField (value : float, params options : GUILayoutOption[]) : float

◆ static function FloatField (value : float, style : GUIStyle, params options : GUILayoutOption[]) : float

◆ static function FloatField (label : string, value : float, params options : GUILayoutOption[]) : float

◆ static function FloatField (label : string, value : float, style : GUIStyle, params options : GUILayoutOption[]) : float

◆ static function FloatField (label : GUIContent, value : float, params options : GUILayoutOption[]) : float

◆ static function FloatField (label : GUIContent, value : float, style : GUIStyle, params options : GUILayoutOption[]) : float

参数

Label 显示在浮点域前面的可选标签。

Value 用于编辑的值

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：float – 用户输入的值

描述：制作一个文本域以便输入浮点数。

◆ **static function Foldout (foldout : bool, content : string, style : GUIStyle = EditorStyles.foldout) : bool**

◆ **static function Foldout (foldout : bool, content : GUIContent, style : GUIStyle = EditorStyles.foldout) : bool**

参数

Foldout	显示折叠状态
Content	显示的标签
Style	可选的 **GUIStyle**

返回：**bool** – 用户选择的折叠状态，如果为真，应该渲染子对象。

描述：制作一个左侧带有折叠箭头的标签。

这可以用来创建一个树或文件夹结构，这里子对象只在夫展开的时候显示

◆ **static function InspectorTitlebar (foldout : bool, targetObj : Object) : bool**

参数

Foldout	折叠状态用这个箭头显示。
targetObj	使用该标题栏的对象（例如组件）

返回：**bool** – 用户选择的折叠状态

描述：制作一个检视窗口标题栏

该标题栏有一个折叠箭头，一个帮助图标和一个设置菜单，设置菜单依赖于所提供的对象的类型。

◆ **static function IntField (value : int, params options : GUILayoutOption[ ]) : int**

◆ **static function IntField (value : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int**

◆ **static function IntField (label : string, value : int, params options : GUILayoutOption[ ]) : int**

◆ **static function IntField (label : string, value : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int**

◆ **static function IntField (label : GUIContent, value : int, params options : GUILayoutOption[ ]) : int**

◆ **static function IntField (label : GUIContent, value : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int**

参数

Label	显示在该整型域前面的可选标签。
Value	用于编辑的值
Style	可选的 **GUIStyle**
Options	一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 **style** 定义的设置 参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight**

返回：**int** – 用户输入的值。

描述：制作一个文本域以便输入整数。

◆ static function IntPopup (selectedValue : int, displayedOptions : string[ ], optionValues : int[ ], params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (selectedValue : int, displayedOptions : string[ ], optionValues :

int[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (selectedValue : int, displayedOptions : GUIContent[ ], optionValues : int[ ], params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (selectedValue : int, displayedOptions : GUIContent[ ], optionValues : int[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (label : string, selectedValue : int, displayedOptions : string[ ], optionValues : int[ ], params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (label : string, selectedValue : int, displayedOptions : string[ ], optionValues : int[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (label : GUIContent, selectedValue : int, displayedOptions : GUIContent[ ], optionValues : int[ ], params options : GUILayoutOption[ ]) : int

◆ static function IntPopup (label : GUIContent, selectedValue : int, displayedOptions : GUIContent[ ], optionValues : int[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

参数

Label	该域前面可选的标签。
selectedValue	该域显示的选项的值
displayedOptions	一个带有显示 X 显示选项的数组，用户可以从中选择。
optionValues	一个为每个选项存储值得数组。
Style	可选的 GUIStyle
Options	一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：int – 被用户选择的选项的值

描述：制作一个整形弹出式选择域

使用当前选择的整数作为参数并返回用户选择的整数。

◆ static function IntSlider (value : int, leftValue : int, rightValue : int, params options : GUILayoutOption[ ]) : int

◆ static function IntSlider (label : string, value : int, leftValue : int, rightValue : int, params options : GUILayoutOption[ ]) : int

◆ static function IntSlider (label : GUIContent, value : int, leftValue : int, rightValue : int, params options : GUILayoutOption[ ]) : int

参数

Label	该滑杆前面可选的标签。
Value	滑杆显示的值。这个决定可拖动滑块的位置。
leftValue	滑杆左端的值
rightValue	滑杆右端的值
options	一个可选的布局选项的列表，它用来指定额外的布

局属性。任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：int – 被用户设置的值

描述：制作一个用户可以拖动到滑杆可以在 min 和 max 之前改变一个整数值。

◆ static function LabelField (label : string, label2 : string, params options : GUILayoutOption[ ]) : void

参数

Label	该域前面的标签
Label2	显示在右侧的标签
Options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

描述：制作一个标签域（用来显示只读信息）

◆ static function LayerField (layer : int, params options : GUILayoutOption[ ]) : int

◆ static function LayerField (layer : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function LayerField (label : string, layer : int, params options : GUILayoutOption[ ]) : int

◆ static function LayerField (label : string, layer : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function LayerField (label : GUIContent, layer : int, params options : GUILayoutOption[ ]) : int

◆ static function LayerField (label : GUIContent, layer : int, style : GUIStyle, params options : GUILayoutOption[ ]) : int

参数

Label	该域前面可选的标签
Layer	显示在该域中的层
Style	可选的 GUIStyle
Options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：int – 用户选择的层

描述：制作一个层选择域

◆ static function ObjectField (obj : Object, objType : System.Type, params options : GUILayoutOption[ ]) : Object

◆ static function ObjectField (label : string, obj : Object, objType : System.Type, params options : GUILayoutOption[ ]) : Object

◆ **static function ObjectField (label : GUIContent, obj : Object, objType : System.Type, params options : GUILayoutOption[ ]) : Object**

参数

Label	该域前面可选的标签
Obj	该域显示的对象
objType	对象的类型
options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 **style** 定义的设置。

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight**

返回：**Object** – 被用户设置的对象

描述：制作一个物体放置槽域

◆ **static function PasswordField (password : string, params options : GUILayoutOption[ ]) : string**

◆ **static function PasswordField (password : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string**

◆ **static function PasswordField (label : string, password : string, params options : GUILayoutOption[ ]) : string**

◆ **static function PasswordField (label : string, password : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string**

◆ **static function PasswordField (label : GUIContent, password : string, params options : GUILayoutOption[ ]) : string**

◆ **static function PasswordField (label : GUIContent, password : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string**

参数

Label	显示在该密码域前面的可选标签
Password	用于编辑的密码
Style	可选的 **GUIStyle**
Options	一个可选的布局选项的列表，它用来指定额外 **iad** 布局属性

任何在这里设置的值将覆盖由 **style** 定义的设置

参见：**GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight,**

GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：**string** – 用户输入的密码

描述：制作一个用户可以输入密码的文本域

这个就像 **GUILayoutPasswordField,**但是正确的响应所有选择，在编辑器中，可以有一个可选的标签在前面。

◆ **static function Popup (selectedIndex : int, displayedOptions : string[ ], params options : GUILayoutOption[ ]) : int**

◆ **static function Popup (selectedIndex : int, displayedOptions : string[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int**

◆ static function Popup (selectedIndex : int, displayedOptions : GUIContent[ ], params options : GUILayoutOption[ ]) : int

◆ static function Popup (selectedIndex : int, displayedOptions : GUIContent[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function Popup (label : string, selectedIndex : int, displayedOptions : string[ ], params options : GUILayoutOption[ ]) : int

◆ static function Popup (label : string, selectedIndex : int, displayedOptions : string[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆ static function Popup (label : GUIContent, selectedIndex : int, displayedOptions : GUIContent[], params options : GUILayoutOption[ ]) : int

◆ static function Popup (label : GUIContent, selectedIndex : int, displayedOptions : GUIContent[ ], style : GUIStyle, params options : GUILayoutOption[ ]) : int

◆

参数

Label 该域前面可选的标签

selectedIndex 该域显示的选项的索引

displayedOptions 显示在弹出菜单中的选项数组

style 可选的 GUIStyle

options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：int – 被用户选择的选项的索引

描述：制作一个通用的弹出式选择域

使用当前选择的索引作为参数并返回用户选择的索引

◆ static function PrefixLabel (label : string, followingStyle : GUIStyle = "Button") : void

◆ static function PrefixLabel (label : string, followingStyle : GUIStyle, labelStyle : GUIStyle) : void

◆ static function PrefixLabel (label : GUIContent, followingStyle : GUIStyle = "Button") :

void

◆ static function PrefixLabel (label : GUIContent, followingStyle : GUIStyle, labelStyle : GUIStyle) : void

参数

Label 显示在控件前面的标签

描述：在一些控件前面制作一个标签。

GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

◆ static function RectField (value : Rect, params options : GUILayoutOption[ ]) : Rect

◆ static function RectField (label : string, value : Rect, params options : GUILayoutOption[ ]) : Rect

◆ static function RectField (label : GUIContent, value : Rect, params options : GUILayoutOption[ ]) : Rect

参数

Label	显示在该域上的标签
Value	用于编辑的值
Options	一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：Rect – 用户输入的值

描述：制作一个 X，Y，W&H 域以便输入一个 Rect。

◆ static function Separator () : void

描述：在前一个控件和后面的控件之前制作一个小的分隔符。

◆ static function Slider (value : float, leftValue : float, rightValue : float, params options : GUILayoutOption[ ]) : float

◆ static function Slider (label : string, value : float, leftValue : float, rightValue : float, params options : GUILayoutOption[ ]) : float

◆ static function Slider (label : GUIContent, value : float, leftValue : float, rightValue : float, params options : GUILayoutOption[ ]) : float

参数

Label	该滑杆前面可选的标签
Value	滑杆显示的值，这个决定可拖动滑块的位置。
leftValue	滑杆左端的值
rightValue	滑杆右端的值
options	一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：float – 被用户设置的值

描述：一个用户可以拖动到滑杆，可以在 min 和 max 之前改变一个值。

◆ static function Space():void

描述：在前一个控件和后面的控件之间制作一个小的空格。

◆ static function TagField (tag : string, params options : GUILayoutOption[ ]) : string

◆ static function TagField (tag : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string

◆ static function TagField (label : string, tag : string, params options : GUILayoutOption[ ]) : string

◆ static function TagField (label : string, tag : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string

◆ static function TagField (label : GUIContent, tag : string, params options : GUILayoutOption[ ]) : string

◆ static function TagField (label : GUIContent, tag : string, style : GUIStyle, params

options : GUILayoutOption[ ]) : string

参数

Label 该域前面可选定标签

Tag 该域显示的标签

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定额外的布局属性。任何在这里设置的值将覆盖由 style 定义的设置。

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：string – 用户选择的标签

描述：制作一个标签选择域

- static function TextArea (text : string, params options : GUILayoutOption[ ]) : string
- static function TextArea (text : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string

参数

Text 用于编辑的文本

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：string – 用户输入的文本

描述：制作一个文本区域

这个就像 GUIlayout。TextArea 一样。但是正确地响应全选，拷贝，粘帖等。在编辑器中

- static function TextField (text : string, params options : GUILayoutOption[ ]) : string
- static function TextField (text : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string
- static function TextField (label : string, text : string, params options : GUILayoutOption[ ]) : string
- static function TextField (label : string, text : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string
- static function TextField (label : GUIContent, text : string, params options : GUILayoutOption[ ]) : string
- static function TextField (label : GUIContent, text : string, style : GUIStyle, params options : GUILayoutOption[ ]) : string

参数

Label 显示在该文本前面的可选标签

Text 用于编辑的文本

Style 可选的 GUIStyle

Options 一个可选的布局选项的列表，它用来指定额外的布局

属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：string – 用户输入的文本

描述：制作一个文本区域

这个就像 GUIlayout。TextField 一样。但是正确地响应全选，拷贝，粘帖等。在编辑器中,可以有一个可选的标签在前面

- static function Toggle (value : bool, params options : GUILayoutOption[ ]) : bool
- static function Toggle (label : string, value : bool, params options : GUILayoutOption[ ]) : bool
- static function Toggle (label : GUIContent, value : bool, params options : GUILayoutOption[ ]) : bool

参数

Label	该开关前面可选的标签
Value	这个开关的显示状态
Options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回： bool – 这个开关的显示状态

描述： 制作一个开关

- static function Vector2Field (label : string, value : Vector2, params options : GUILayoutOption[ ]) : Vector2

参数

Label	显示在该域上的标签
Value	用于编辑的值
Options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：Vector2 – 用户输入的值

描述：制作一个 X&Y 域以便输入一个 Vector2

- static function Vector3Field (label : string, value : Vector3, params options : GUILayoutOption[ ]) : Vector3

参数

Label	显示在该域上的标签
Value	用于编辑的值
Options	一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参 见 ： GUILayout.Width, GUILayout.Height, GUILayout.MinWidth,

GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：Vector3 – 用户输入的值

描述：制作一个 X&Y 域以便输入一个 Vector3

◆ static function Vector4Field (label : string, value : Vector4, params options : GUILayoutOption[ ]) : Vector4

参数

Label 显示在该域上的标签

Value 用于编辑的值

Options 一个可选的布局选项的列表，它用来指定额外的布局属性，任何在这里设置的值将覆盖由 style 定义的设置

参见：GUILayout.Width, GUILayout.Height, GUILayout.MinWidth, GUILayout.MaxWidth, GUILayout.MinHeight, GUILayout.MaxHeight, GUILayout.ExpandWidth, GUILayout.ExpandHeight

返回：Vector4 – 用户输入的值

描述：制作一个 X&Y 域以便输入一个 Vector4

EditorGUIUtility

类，继承自 GUIUility

用户 EditorGUI 的各种辅助函数

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

变量

◆statie var systemCopyBuffer：string

描述：系统拷贝缓存

使用这个使拷贝和粘贴为你自己的数据工作

◆statie var whiteTexture：Texture2D

描述：获取一个白色纹理

类方法

◆statie function AddCursorRect（position：Rect，mouse：MouseCursor）：void

参数

position 显示控件的矩形

mouse 使用鼠标光标

描述：添加一个自定义的鼠标光标到一个控件

```
function OnGUI()
{
    // 当鼠标悬停在这个矩形上时，显示“link”光标
    EditorGUIUtility.AddCursorRect (Rect(10,10,100,100), MouseCursor.Link);
}
```

◆static function DrawColorSwatch (position : Rect, color : Color) : void

参数

position 绘制样板颜色的矩形。Color 绘制颜色

描述：绘制一个颜色样本

◆static function FindTexture (name : string) : Texture2D

描述：从源文件名获取一个纹理

◆static function HasObjectThumbnail (objType : Type) : bool

描述：给定的类有对象有小物体

◆static function Load (path : string) : Object

描述：加载一个内置资源

这个函数将在 Assets/Editor Default Resources/ + path 中查找资源，如果没有，它将按照名称尝试内置的编辑器资源

var handleMaterial : Material = EditorGUIUtility.Load (Load("HandleMaterial.mat"));

◆static function LoadRequired (path : string) : Object

描述：加载位于路径上的内置资源

这个函数将在 Assets/Editor Default Resources/ + path 中查找任何资源。

var handleMaterial : Material = EditorGUIUtility.Load (Load("HandleMaterial"));

◆ static function LookLikeControls (labelWidth : float = 100, fieldWidth : float = 70) : void

◆

参数

labelWidth 用于前缀标签的宽带

fieldWidth 文本条目的宽度。参见：LookLikeInspector

描述：使所有 ref::EditorGUI 外观像普通控件

这将使 EditorGUI::ref::使用默认风格，看起来想控件（例如 e.g. EditorGUI.Popup 成为一个完全是弹出菜单）

◆ static function LookLikeInspector () : void

描述：使所有 ref::EditorGUI 外观像简化边框的视控件

这个将使 EditorGUI 使用的默认风格的外观就像它在检视面板中一样

参见：LookLikeControls

◆ static function ObjectContent (obj : Object, type : System.Type) : GUIContent

描述：返回一个带有名称和图标的 GUIContent 物体

如果物体为空，图标将根据类型选择

◆ static function PingObject (obj : Object) : void

描述：Ping 窗口中的一个对象，就像在检视面板中点击它

◆ static function QueueGameViewInputEvent (evt : Event) : void

描述：发送一个输入事件到游戏

◆ static function RenderGameViewCameras (cameraRect : Rect, statsRect : Rect, stats : bool, gizmos : bool) : void

cameraRect 用于渲染所有游戏相机的设备坐标

statsRect 统计应该显示的地方

stats 叠加显示统计数据

gizmos 也显示 gizmos

描述：渲染所有游戏内的相机

继承的成员

继承的类变量

hotControl 当前具有热点的控件 controlID

keyboardControl 具有键盘焦点控件的 controlID

继承的类函数

GetControlID 为一个控件获取唯一 ID

GetStateObject 从一个 controlID.获取一个状态

QueryStateObject 从一个 controlID.获取一个存在的状态物体

MoveNextAndScroll 只允许从 OnGUI 内部调用 GUI 函数

GUIToScreenPoint 将一个点从 GUI 位置转化为屏幕空间

ScreenToGUIPoint 将一个点从屏幕空间转化为 GUI 位置

RotateAroundPivot 使 GUI 围绕一个点选择的辅助函数

ScaleAroundPivot

EditorGUI

类

只用于编辑器 GUI 的类，这个类包含了附加到 UnityGUI 的通用 2D 元素

注意：这是一个编辑器类。为了使用它你必须防止脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

这些工作的非常像普通的 GUI 函数，在 EditorGUILayout 中也有相同实现

类变量

static var actionKey : bool

描述：平台相关的"action"调整键被按下（只读）

Mac OS X 为 Command, Windows 上为 Control

类方法

◆static function ColorField (position : Rect, value : Color) : Color

◆static function ColorField (position : Rect, label : string, value : Color) : Color

◆static function ColorField (position : Rect, label : GUIContent, value : Color) : Color

参数

position 屏幕上用于域的矩形区域

label 显示在该域前面的可选标签

value 用于编辑的颜色

返回：Color – 用户选择的颜色

描述：制作一个用来选择 Color 的域

◆static function DrawTextureAlpha (position : Rect, image : Texture, scaleMode : ScaleMode = ScaleMode.StretchToFill, imageAspect : float) : void

参数

position 屏幕上用来绘制纹理的矩形区域

image 显示的 Texture

scaleMode 当纹理的长宽比不适合绘制的长宽比时如何缩放这个图片

alphaBlend 是否用 alpha 混合显示图片（默认）如果为假，图片将绘制到屏幕

imageAspect 用于源图像的宽高比，如果为 0（默认），使用来自图片的宽高比

描述：在一个矩形中绘制纹理的 alpha 通道

参见: GUI.color, GUI.contentColor

◆static function EnumPopup (position : Rect, selected : System.Enum, style : GUIStyle = EditorStyles.popup) : System.Enum

◆static function EnumPopup (position : Rect, label : string, selected : System.Enum, style : GUIStyle = EditorStyles.popup) : System.Enum

◆static function EnumPopup (position : Rect, label : GUIContent, selected : System.Enum,

style : GUIStyle = EditorStyles.popup) : System.Enum

参数

position 屏幕上用来绘制纹理的矩形区域

label 该域前面可选的标签

selected 该域显示的选项

style 可选的 **GUIStyle.**

返回：**float-** 被用户选择的选项

描述：制作一个弹出式选择域

使用当前选择的值作为参数并返回用户选择的值

◆ **static function FloatField (position : Rect, value : float, style : GUIStyle = EditorStyles.numberField) : float**

◆**static function FloatField (position : Rect, label : string, value : float, style : GUIStyle = EditorStyles.numberField) : float**

◆**static function FloatField (position : Rect, label : GUIContent, value : float, style : GUIStyle = EditorStyles.numberField) : float**

参数

position 屏幕上用来浮点值的矩形区域

label 显示在浮点域前面的可选标签

value 该域显示的选项

style 可选的 **GUIStyle.**

返回：**float-** 被用户输入的值

描述：制作一个文本域以便输入浮点值

◆**static function Foldout (position : Rect, foldout : bool, content : string, style : GUIStyle = EditorStyles.foldout) : bool**

◆**static function Foldout (position : Rect, foldout : bool, content : GUIContent, style : GUIStyle = EditorStyles.foldout) : bool**

参数

position 屏幕上用于箭头和标签的矩形区域

foldout 显示折叠状态

content 显示的标签

style 可选的 **GUIStyle.**

返回：**bool-** 用户选择的折叠状态，如果为真，应该渲染子对象

描述：制作一个左侧带有折叠箭头的标签

这个可以用来创建一个树或则文件夹结构，这里子对象只在父展开的时候显示

◆ **static function InspectorTitlebar (position : Rect, foldout : bool, targetObj : Object) : bool**

参数

position 屏幕上用于标题栏的矩形区域

foldout 显示折叠状态

targetObj 使用该标题栏的对象

返回：**bool-** 用户选择的折叠状态

描述：制作一个检视窗口标题栏

该标题栏有一个折叠箭头，一个帮助图标和设置菜单，设置菜单以来于所提供对象类型

◆ static function IntField (position : Rect, value : int, style : GUIStyle = EditorStyles.numberField) : int

◆static function IntField (position : Rect, label : string, value : int, style : GUIStyle = EditorStyles.numberField) : int

◆static function IntField (position : Rect, label : GUIContent, value : int, style : GUIStyle = EditorStyles.numberField) : int

参数

position 屏幕上用于整型域的矩形区域

label 显示在该整型域前面的可选标签

value 用于编辑的值

style 可选的 GUIStyle.

返回：int- 用户输入的值

描述：制作一个文本域以便输入整数

◆static function IntPopup (position : Rect, selectedValue : int, displayedOptions : string[], optionValues : int[], style : GUIStyle = EditorStyles.popup) : int

◆static function IntPopup (position : Rect, selectedValue : int, displayedOptions : GUIContent[], optionValues : int[], style : GUIStyle = EditorStyles.popup) : int

◆ static function IntPopup (position : Rect, label : string, selectedValue : int, displayedOptions : string[], optionValues : int[], style : GUIStyle = EditorStyles.popup) : int

◆static function IntPopup (position : Rect, label : GUIContent, selectedValue : int, displayedOptions :

GUIContent[], optionValues : int[], style : GUIStyle = EditorStyles.popup) : int

参数

position 屏幕上用于域的矩形区域

label 可选标签

selectedValue 显示的选项的值

displayedOptions 一个带有显示 X 选项的数组，用户可以选择

optionValues 一个为每个选项存储值的数组

style 可选的 GUIStyle.

返回：int- 被用户选择的值

描述：制作一个整形弹出式选择域

只用当前选择的整数作为参数并返回用户选择的整数

◆static function IntSlider (position : Rect, value : int, leftValue : int, rightValue : int) : int

◆static function IntSlider (position : Rect, label : string, value : int, leftValue : int, rightValue : int) : int

◆static function IntSlider (position : Rect, label : GUIContent, value : int, leftValue : int, rightValue : int) : int

参数

position 屏幕上用于滑竿的矩形区域

label 该滑竿前面可选择的标签

value 滑竿显示的值

leftValue 滑竿左端值

rightValue 滑竿右端值

返回：int- 用户设置的值

描述：制作一个可以拖动的滑竿，可以在 min 和 max 之间设置参数

◆ static function LabelField (position : Rect, label : string, label2 : string) : void

参数

position 屏幕上用于标签域的矩形区域

label 前面的标签

label 右侧的标签

描述：制作一个标签域（用来显示只读信息）

◆ static function LayerField (position : Rect, layer : int, style : GUIStyle = EditorStyles.popup) : int

◆static function LayerField (position : Rect, label : string, layer : int, style : GUIStyle = EditorStyles.popup) : int

◆static function LayerField (position : Rect, label : GUIContent, layer : int, style : GUIStyle = EditorStyles.popup) : int

◆

staticfunctionLayerField(position:Rect,label:GUIContent,Layer:ini,styleGUIStyle=EditorStyles.popup):int

参数

Position 屏幕上用于域的矩形区域

Label 该域前面可选的标签

layer 显示在该域中的层

返回：int 用户选择的层

描述：制作一个层选择域

◆ static function ObjectField(position : Rect,obj : Object,objType : System Type) : Object

◆ Static function ObjectField (position : Rect, label : string, obj : Object, objType System.Type) : Object

◆ Static function ObjectField(position : Rect,label : GUIContent,obj : Object,objType : System.Type):Object

参数

Position 屏幕上用于域的矩形区域

Label 该区前面可选的标签

Obj 该域显示的对象

objType 对象的类型

返回：Object 被用户设置的对象

描述：制作一个物体放置的槽域

◆ Static function PasswrodField (position : Rect,password : string,style : GUIStyle= EditorStyles.textField) : string

◆ Static function PasswordField (position : Rect,label : string,password : string,style : GUIStyle=EditorStyles.textField) : string

◆ Siatic function PasswordField(position : Rect,label:GUIConient,password : string,style : GUIStyle = EditorStyle.textField) “ string

参数

Position 屏幕上用于密码域的矩形区域

Label 显示在该密码域前面的可选标签

Password 用于编辑的密码

Style 可选的 GUIStyle

返回：string- 用户输入的密码

描述：制作一个用户可以输入密码的文本域

这个就像 GUI.PasswordField，但是正确的响应所有选择，在编辑器中，可以有一个可选的标签在前面。

◆ static function Popup (position : Rect, selectedIndex : int, displayedOptions : string[], style : GUIStyle = EditorStyles.popup) : int

◆ static function Popup (position : Rect, selectedIndex : int, displayedOptions : GUIContent[],

◆ style : GUIStyle = EditorStyles.popup) : int

static function Popup (position : Rect, label : GUIContent, selectedIndex : int, displayedOptions : GUIContent[], style : GUIStyle = EditorStyles.popup) : int

参数

Position 屏幕上用于域的矩形区域

Label 该域前面可选的标签

selectedIndex 该域显示的选项的索引

displayedOptions 显示在弹出菜单中的选项数组

style 可选的 GUIStyle

返回：int-被用户选择的选项的索引

描述：制作一个哦他能够用的弹出式选择域

使用当前选择的索引作为参数并返回用户选择的索引

◆ Static function PrefixLabel(totalPosition:Rect,id:int,label:GUIContent):Rect

参数

totaIPosition 用于标签和控件的屏幕上的矩形

id 空间的唯一 ID

label 显示在控件前面的标签

返回：Rect 屏幕上的矩形，只用于控件自身

描述：在一些空间前面制作一个标签

static function RectField (position : Rect, value : Rect) : Rect

static function RectField (position : Rect, label : string, value : Rect) : Rect

static function RectField (position : Rect, label : GUIContent, value : Rect) : Rect

参数

position

屏幕上用于域的矩形区域

label 显示在该域上的可选标签

value 用于编辑的值

返回：Rect －用户输入的值描述：制作一个 X.Y W&H 域以便输入一个 Rect

◆ static function Slider (position : Rect, value : float, leftValue : float, rightValue : float) : float

◆ static function Slider (position : Rect, label : string, value : float, leftValue : float, rightValue : float) : float

◆ static function Slider (position : Rect, label : GUIContent, value : float, leftValue : float, rightValue : float) : float

参数

position

屏幕上用于滑竿的矩形区域

label 该滑竿前面可选的标签

value 滑竿显示的值。这个决定可拖动滑块的位置。.

leftValue 滑竿左端的值.

rightValue 滑竿右端的值

返回：**float-** 被用户设置的值

描述：一个用户可以拖动的滑竿，可以在 **min** 和 **max** 之间改变的一个值

◆ **static function TagField (position : Rect, tag : string, style : GUIStyle = EditorStyles.popup) : string**

◆ **static function TagField (position : Rect, label : string, tag : string, style : GUIStyle = EditorStyles.popup) : string**

◆ **static function TagField (position : Rect, label : GUIContent, tag : string, style : GUIStyle = EditorStyles.popup) : string**

参数

position

屏幕上用于域的矩形区域

label 该域前面可选的标签

tag 该域显示的标签

style可选的 **GUIStyle**

返回：**string-** 用户选择的标签

描述：制作一个标签选择域

◆ **static function TextArea (position : Rect, text : string, style : GUIStyle = EditorStyles.textField) : string**

参数

position

屏幕上用于文本的矩形区域

text 用于编辑的文本

style可选的 **GUIStyle.** 返回：**string-** 用户输入的文本

描述：制作一个文本域

这个就像 **GUITextField** 一样,但是正确的响应全选，拷贝，粘贴等，在编辑器中。

◆ **static function TextField (position : Rect, text : string, style : GUIStyle = EditorStyles.textField) : string**

◆ **static function TextField (position : Rect, label : string, text : string, style : GUIStyle = EditorStyles.textField) : string**

◆ **static function TextField (position : Rect, label : GUIContent, text : string, style : GUIStyle = EditorStyles.textField) : string**

参数

position 屏幕上用于开关的矩形区域.

label 显示在该文本域前面的可选标签

text 用于编辑的文本.

Style 可选的 **GUIStyle**

返回：**string-** 用户输入的文本

描述：制作一个文本域

这个就像 GUITextField,但是正确的响应所有选择，拷贝，粘贴等等，在编辑器中，可以有一个可选的标签在前面。

- static function Toggle (position : Rect, value : bool) : bool
- static function Toggle (position : Rect, label : string, value : bool) : bool
- static function Toggle (position : Rect, label : GUIContent, value : bool) : bool

参数

position 屏幕上用于开关的矩形区域.

label 该开关前面可选的标签

value 这个开关的显示状态

返回：bool – 这个开关的显示状态

描述：制作一个开关

static function Vector2Field (position : Rect, label : string, value : Vector2) : Vector2

参数

position 屏幕上用于域的矩形区域

label 显示在该域上的标签

value 用于编辑的值

返回：Vector2- 用户输入的值

描述：为 Vector2 制作一个 X,Y 域

- static function Vector3Field (position : Rect, label : string, value : Vector3) : Vector3

参数

position 屏幕上用于域的矩形区域

label 显示在该域上的标签

value 用于编辑的值

返回：Vector3 – 用户输入的值

描述：制作一个 X,Y Z 域以便输入一个 Vector3

- static function Vector4Field (position : Rect, label : string, value : Vector4) : Vector4

参数

position 屏幕上用于域的矩形区域

label 显示在该域上的标签

value 用于编辑的值

返回：Vector4 – 用户输入的值

描述：制作一个 X,Y Z&W 域以便输入一个 Vector4

EditorPrefs

类

在储存并访问 Unity 编辑器的首选项

注意：这是一个编辑器类，为了使用它你必须房子脚本到工程文件夹的 Assets/Editor

描述：设置由 Key 确定的值

描述：设置由 Key 确定的值

EditorStyles

类

用于 EditorGUI 控件的通用 GUIStyle 通过调用 EditorGUIUtility.LookLikeInspector 和 EditorGUIUtility.LookLikeControls.来设置

注意：这是一个编辑器类，为了使用它你必须设置脚本的工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加"using

UnityEditor;"

类变量

Label	用于所有 EditorGUI 前部标签的风格
textField	用于 EditorGUI.TextField 的风格
popup	用于 EditorGUI.Popup, EditorGUI.EnumPopup 的风格
structHeadingLabel	用于结构标题的风格（Vector3,Rect,等）
objectField	用于对象域标题的风格
objectFieldThunmb	用于对象域中选择按钮标题的风格
colorField	用于颜色域标题的风格
layerMaskField	用于层蒙板标题的风格
toggle	用于 EditorGUI.Toggle 的风格
foldout	用于 EditorGUI.Foldout 的风格
toggleGroup	用于 EditorGUILayout.BeginToggleGroup.的风格
standardFont	标准字体
boldFont	黑体字体

EditorUtility

类

编辑器工具函数

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editior 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor”

类方法

◆ Static function ClearProgressBar():void

描述：移除工具条

参见：DisplayPrograssBar 函数

◆ Static function CloneComponent(c:Component):Component

描述：复制一个组件

◆ Static function CollectDependencies(roots:Object[]):Object[]

描述：计算并返回所有 roots 依赖的资源

◆ Static function CreateEmptyPrefab(path:srting):Object

描述：在路劲上创建一个空的预设，如果在路径上已经有一个预设，他将被删除并用一个空的预设替换

返回该预设的一个引用

◆ Static functionCreateGameObjectWithHideFlags(name:string,flags:HideFlags,prarmscomponents:Type[]):GameObject

描述：创建一个游戏物体并附加特定的组件

◆ staticfunctionDisplayDialog(title:string,massage:srting,ok:string,cancel:srting=””):bool

描述：显示一个模式对话框

使用它来在编辑器中显示信息框

Ok 和 cancel 是显示在对话框按钮上的标签，如果 cancel 是空（默认），那么只有一个按钮被显示，如果 OK 按钮被按下 DisplayDialog 返回 true

◆

staticfunctionDisplayDialogComplex(title:srting,massage:string,ok:srting,cancel:string,alt:string):int

描述：显示一个带有三个按钮的模式对话框

使用它来在编辑器中显示信息框

与 DisplayDialog 类似，只是这个版本显示带有三个按钮的对话框，ok,cancel 和 alt 是显示在按钮上的标签，DisplayDDialogComplex 返回一个证书，0，1 和 2 对应的 ok,cancel 和 alt 按钮

◆ Static function DisplayPopupMenu(position:Rect,menuItemPath:string,command:MenuCommand):void

描述 显示一个弹出菜单

菜单显示在 pos 处，从一个由 menuItemPath 指定的子菜单生成，使用 MenuCommand 作为菜单上下文

```
var evt = Event.current;
var contextRect = Rect(10,10,100,100);
if(evt.type == EventType.ContextClick)
{
var mousePos = evt.mousePosition;
if (contextRect.Contains (mousePos)) {
EditorUtility.DisplayPopupMenu
(Rect (mousePos.x,mousePos.y,0,0), "Assets/", null);
evt.Use();
}
}
```

◆ static function DisplayProgressBar (title : string, info : string, progress : float) : void

描述：显示或更新一个进度条

窗体标题栏被设置为 title 信息被设置为 info.进度应该被设置为 0.0 到 1.0 之间的值，0 表示没有做任何事，1.0 意味着 100%完成

在编辑器脚本或向导中执行任何长的操作并想通知用户关于这个操作的进度时，这个是非常有用的！

参见：ClearProgressBar 函数

◆ static function ExtractOggFile (obj : Object, path : string) : bool

描述：在路径处找到一个资源

路径名必须包含文件扩展名，它不应该用”Assets”做前缀，这个只返回在工程试图中可见的资源

参见 GetAssetPath 函数

◆ static function FormatBytes (bytes : int) : string

描述：将字节数返回为一个文本

```
print (EditorUtility.FormatBytes(100)); // prints "100 bytes"
print (EditorUtility.FormatBytes(2048));
// prints "2.0 KB"
```

Static function GetAssetPath(asset: Objct):string\

描述：返回一个资源的路径名

与 EditonUityFindAssent 相反

参见：GetAssentPath 函数

◆ Function GetDsconnectedPrefabParent(source: Object):Object

描述：返回 source 最后链接到游戏物体父，如果没有返回 null

Static function GetObjectEnabled(target: Object):int

描述：物体是否启用（0 禁用 1 启用 -1 没有启用按钮）

◆ Static function GetPrefabParent(source:Object):Object

描述：返回 source 的游戏物体父，如果没有发现返回 null

◆ Static function GetPrefabType(targe:Object):Prelabtype

描述：

◆ Static function InstanceIDtoObject(instanceID:int):Object

描述：转化实例 ID 为对象的引用

如果对象没有从磁盘加载，加载它

◆ Static function InstantiatePrefab(target:Object):Object

描述：实例化给定预设

这个累死与 Instantiated 但它创建一个到预设的预设连接

◆ Static function IsPersistent(target:Object):bool

描述：决定一个对象是否存储在磁盘上，典型资源如：预设，纹理，音频剪辑，动画，材质

如果一个物体在场景中返回假，典型的如一个游戏物体成组，但是它也可以是一个从代码中创建的材质，而且该材质没有存储在资源中而是存储在场景中！

◆ Stiatc function OpenFilePanel(title:string directory:srring,exlension:string):string

描述，显示“打开文件”对话框并返回选中的路径名

参见 SaveFilePanel 函数

static function ReconnectToLastPrefab (go : GameObject) : bool

描述

◆

StaticfunctionReplacePrefab(go:GameObject,targetPrefab:Object,connecToPrefab:bool=false):GameObject

描述：用一个游戏物体 GO 替换 targetPetPrefab

◆ static function SaveFilePanel (title : string, directory : string, defaultName : string,

◆ extension : string) : string

描述：显示“保存文件”对话框并返回选中的路径名

参见：OpenFilePanel 函数

◆ static function SetDirty (target : Object) : void

描述：标记 target 物体为脏

Unity 内部使用脏标记来查看资源何时被改变，并被保存到硬盘上，例如，如果你改变一个预设的 MonoBehaviour 或 ScriptabieObject 变量，你必须告诉 Unity 这个值被改变了，Unity 内置组件内会在一个属性改变后调用 SetDirty，MonoBehaviour 或 ScriprableObject 不会自动调用，因此如果你想改变的值被保存，你需要调用 SetDirty

◆ static function SetObjectEnabled (target : Object, enabled : bool) : void

描述：设置物体的启用状态

◆ Static function SmartResetGameObjectToPrefabState(go:GameObject):bool

描述

◆ Static functionSmartResetToPrefabState(obj:Object):bool

描述

EditorWindow

类：继承自 ScriptableObject

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

创建你自己的自定义窗口，可以自由的浮动并描写，就像 Unity 自身的窗口一样

编辑器窗口典型地使用一个菜单项打开

JAVAScript 例子

```
class MyWindow extends EditorWindow {
var myString = "Hello
World";
var groupEnabled = false;
var myBool =
true;
var myFloat = 1.23;
// Add menu named "My Window" to the Window menu
@MenuItem ("Window/My
Window")
static function Init () {
// Get
existing open window or if none, make a new one:
var window : MyWindow
= EditorWindow.GetWindow
(MyWindow);
window.Show ();
function OnGUI () {
GUILayout.Label ("Base Settings", EditorStyles.boldLabel);
myString = EditorGUILayout.TextField
("Text Field", myString);
groupEnabled = EditorGUILayout.BeginToggleGroup
("Optional Settings", groupEnabled);
myBool = EditorGUILayout.Toggle
("Toggle", myBool);
myFloat = EditorGUILayout.Slider
("Slider", myFloat, -3, 3);
EditorGUILayout.EndToggleGroup
```

C#例子

```
using UnityEngine;
using UnityEditor;
public
class MyWindow : EditorWindow {
string myString = "Hello
World";
bool groupEnabled;
```

```
bool myBool = true;
float myFloat = 1.23f;
//
Add menu named "My Window" to the Window menu
[MenuItem ("Window/My
Window")]
static void Init () {
// Get existing
open window or if none, make a new one:
MyWindow window = (MyWindow)EditorWindow.GetWindow (typeof
(MyWindow));
window.Show ();
}
void
OnGUI () {
GUILayout.Label ("Base Settings", EditorStyles.boldLabel);
myString
= EditorGUILayout.TextField
("Text Field", myString);
groupEnabled = EditorGUILayout.BeginToggleGroup
("Optional Settings", groupEnabled);
myBool = EditorGUILayout.Toggle
("Toggle", myBool);
myFloat = EditorGUILayout.Slider
("Slider", myFloat, -3, 3);
EditorGUILayout.EndToggleGroup
();
}
}
```

变量

◆ **Var autoRepaintOnSceneChange:bool**

描述：当场景更新时窗口自动重绘吗？

◆ **Var position:Rect**

描述：屏幕上的像素位置

设置这个将更锁定的窗口解锁定

◆ **Var wantsMouseMove:bool**

描述：在这个编辑器窗口中的 GUI 需要 MouseMove 事件吗？

函数

◆ **Function Close():void**

描述：关闭编辑器窗口

这将销毁编辑器窗口

◆ **Function Focus()” void**

描述：移动键盘焦点到这个 EditorWindow

参见：**focusWindow**

Function Repaint():void

描述：使窗口重绘

- **Function SendEvent(e:Event):bool**

描述：发送一个事件到窗口

Var win:

EditorWindow

//发送一个粘贴事件到 EditorWindow，就像从 Edit 菜单中选择 Paste

- **Function Show (immediateDisplay:bool=false):void**

描述：显示编辑器窗口

Function Show Unity():void

描述：**EditorWindow** 显示为一个浮动工具窗口

工具窗口总是在普通窗口的前面，并且当用户切换到其他应用程序时被隐藏。

//MyWindow 是 EditorWindow 的签名

Varwindow:MyWindow=new MyWindow():

Window.ShowUtility():

消息传递

- **Function OnDestroy():void**

描述：当窗口被关闭时调用

- **Funciton OnGUI():void**

描述：在这里实现你自己的 **GUI**

- **Function OnFocus():void**

描述：当窗体获得键盘焦点时调用

- **Function OnHierarchyWindowChange():void**

描述：当场景层次改变时被调用

当 **transform.parent** 改变，**gameObject.name** 创建一个新游戏物体时，等等

- **Function OnInspectorUpdate():void**

描述：**OnInspectorUpdate** 以 **10** 帧每秒被调用，以便给检视面板更新的机会

- **Function OnLostFocus():void**

描述：当窗体失去键盘焦点时被调用

- **Function OnProjectWindowChange():void**

描述：当选择改变时被调用

- **Function Update():void**

描述：在所有可视的窗口 **z** 中每秒被调用 **100** 次

类变量

- **Static var focusedWindow:EditorWindow**

描述：哪个编辑器窗体当前具有焦点（只读）

如果没有窗体具有焦点/**focusedWindow**/可为 **null**

参见：**mostseOverWindow,Focus**

- **Static var mouseOverWindow:EditorWindow**

描述：那个编辑器窗体当前位于鼠标之下（只读）

如果没有窗体位于鼠标之下/**mouseOverWindow**/可为 **null**

参见：**focusedWindow**

类函数

- **Static function GetWindow(t:System,Type,utility:bool=false,title,string null):**

EditorWindow

t 窗体的类型，必须从 EditorWindow 派生

utility 设置这个为真以便创建一个浮动的窗体，如果为假将创建一个普通窗体

title 如果 GetWindow 创建一个新的窗体，它将使用这个标题，如果这个值为 null，使用类作为标题

描述：返回当前屏幕上第一个类型为 T 的 EditorWindow

◆

StaticfunctionGetWindowWithRect(t:System.Type,reet:Rect,utility:bool=falsei,title:string=null):EditorWindow

参数

T 窗体的类型，必须从 EditorWindow 派生

Rect 新创建的窗体将显示在屏幕上的位置

Utility 设置这个为真以便创建一个浮动的窗体，如果为假将创建一个普通窗体

Title 如果 GetWindow 创建一个新的窗体，它将使用这个标题，如果这个值为 null，使用类名作为标题

描述：返回当前屏幕上第一个类型为 t 的 EditorWindow

如果没有，在位置 rect 创建并显示新窗体然后返回它的实例

继承的成员

继承的变量

Name	对象的名称
hideFlags	该物体是否被隐蔽，保存在场景中或被用户修改

继承的函数

GetInstanceID	返回该物体的实例 id

继承的消息传递

OnFnable	物体被加载时调用该函数
OnDisable	当可编程物体超出范围时调用这个函数

继承的类函数

CreateInstance	使用 className 创建一个可编程物体的实例
Operator bool	这个物体存在吗
Instantiate	克隆 original 物体并返回这个克隆
Destroy	移除一个游戏物体，组件或资源
DestroyImmediate	立即销毁物体 obj.强烈建议使用 Destroy 代替
FindObjectsOftype	返回所有类型为 type 的激活物体
FindObjectOfType	返回第一个类型为 type 的激活物体
Operator	比较两个物体是否相同
Operator	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 target 不被自动销毁

FileUtil

类

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”usingUnityEditor”

类方法

◆ Static function CopyFileOrDirectory(from:string,to:string):void

描述：

Static function CopyFileOrDirectoryFollowSymlinks(from:string,to:string):void

描述：

- Static function DeleteFileOrDirectory(path:string):bool

描述

- Static function MoveFileOrDirectory(from:string,to:string):void

描述

HandleUtility

类

用于场景试图类型 3DGUI 的辅助函数

注意，这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”usingUnityEditor”

类变量

- Static var acccleration:float

描述：为拖动值获取一个标准的加速度（只读）

通常的加速度为 1.0 当 shift 被按下，它是 4.0 当 ALT 被按下，加速度是 0.25

参见：niceMouseDalta

- Static var handleMaterial:Matenal

描述：

- Static var nearestControl:int

描述

- Static var niceMouseDelta:float

描述：取一个好的鼠标增值用于拖动浮点数（只读）

这将自动检测用户对 X/Y 轴拖动的设置并返回一个相应的浮点数，这个也可以正确的处理调整键，该增值已经被乘

参见：

- Static var niceMouseDeltaZoom:float

描述：取一个好的鼠标增值用于缩放（只读）

这将自动检测用户对 X/Y 轴拖动的设置并返回一个相应的浮点数，这个也可以正确的处理调整键，该增值已经被乘

参见：

类方法

- static function AddControl(controlId:int,distance:float):void

描述：记录一个从手柄到这里的距离

所有的手柄在布局时，使用它们的 controIID 调用这个，然后使用 nearestControl 来检查它们是否得到 mouseDown

- Static function AddDefaultControl(controlId:int):void

描述：为一个默认的控件添加 ID，如果没有其他被选择，这个将被选中

-

StaticfunctionCalcLineTranslation(src:vector2,dest:vector2.srcPosition:Vector3.constraintDir:Vector3):float

参数

Sre 拖动的源点

Dest 拖动的目标点

srcPosition 在 sre 射线上拖动物体的 3D 位置

constraintDir 约束移动的 3D 方向

返回：float-沿着 constraintDir 移动的距离

Staic function ClosestPointToAre(center:Vector3,normal:Vector3,from:Vector3,angle:float,rablus:float):Vector3

描述：获取最接近的 3D 点

◆ Static function ClosetPointToDise(center:Vector3,normal:Vector3,radius:float):Vector3

描述：获取最接近的 3D 点

◆ Static function ClosestPointToPolyLine(params vertices:Vector3):Vector3

描述：获取最接近的 3D 点

◆ static function DistancePointLine (point : Vector3, lineStart : Vector3, lineEnd : Vector3) : float

描述：计算点和线之间的距离

参见：ProjectPointLine

◆ static function DistanceToArc (center : Vector3, normal : Vector3, from : Vector3, angle : float, radius : float) : float

描述：从鼠标指针到圆盘 3D 部分的距离

在计算鼠标指针到圆盘部分的屏幕空间距离，这个圆盘在世界空间的 position 的位置上，并有给定的 radius 和 normal 圆盘部分由 from 向量和 angle 角度定义

参见：DistanceToDisc, DistanceToCircle.

◆ static function DistanceToCircle (position : Vector3, radius : float) : float

描述：从鼠标指针到相机朝向的圆的像素距离

在计算鼠标指针到圆的屏幕空间距离，这个圆在世界空间的 position 的位置上，并有给定的 radius，该圆被假定是朝向相机的

使用当前相机定义距离

参见：DistanceToDise

◆ static function DistanceToDisc (center : Vector3, normal : Vector3, radius : float) : float

描述：从鼠标指针到一个 3D 圆盘的像素距离

在计算鼠标指针到圆盘（圆）的屏幕空间距离，这个圆盘（圆）在世界空间的 position 的位置上，并有给定的 radius 和 normal

使用当前相机定义距离

参见：DistancetoCircle

◆ static function DistanceToLine (p1 : Vector3, p2 : Vector3) : float

描述：从鼠标指针到一个直线的像素距离

计算从鼠标指针到一条直线的距离空间距离，该直线从 p1 到 p2 世界空间点，使用当前相机定义距离

◆ static function DistanceToPolyLine (params points : Vector3[]) : float

描述：从鼠标指针到一个折线的像素距离

计算从鼠标指针到穿过给定世界空间点的多段直线的屏幕空间距离

使用当前相机定义距离

参见：DistanceToLine

◆ **Static function FindPrefabRoot(source:GameObject):GameObject**

描述：辅助函数用来周到一个物体的预设根（用来精确的选择）

◆ **Static function GetHandleSize(position:Vector3):float**

描述：在给定的位置上获取操作器手柄的世界空间尺寸

使用当前相机计算何时的尺寸

◆ **Static function GUIPointToWorldRay(position:Vector2):Ray**

描述：转化 **2DGUI** 位置到一个世界空间射线

使用当前相机计算射线

参见：**WorldToGUIPoint**

◆ **Static function PickGameObject(position:Vector2):GameObject**

描述：

◆ **Static function PopCamera(camera:Camera):void**

描述：取回所有的相机设置

◆ **Static function ProjectPointLine(point:Vector3,lineStart:Vector3,lineEnd:Vector3):Vector3**

描述：投影 **point** 到一个直线

参见：**DistancePointLine**

◆ **Static function PushCamera(camera:Camera):void**

描述：

保存所有的相机设置

◆ **Static function RaySnap(ray:Ray):object**

返回：**object** 一个装箱的 **RaycastHit**，如果没有碰到它为 **null**

描述：当 **raysnapping** 时忽略的物体（典型的是当物体被手柄拖动的时候）

朝着屏幕投射 **ray**

◆ **Static function Repaint():void**

描述：重绘当前试图

◆ **Static function WorldPointToSizedReet(position:Vector3,content:GUIContent,style:GUIStyle):Rect**

参数

Position	使用的世界空间位置
Content	让出空间显示的内容
Style	使用的风格，该风格的对齐

描述：在 **3D** 空间中制定一个矩形来显示一个 **2DGUI** 元素

◆ **Static function WorldToGUIPoint(world:Vector3):Vector2**

描述：转化世界空间点到 **2DGUI** 位置

使用当前相机计算投影

参见：**GUIPointToWorldRay**

Handles

类

各种绘制物

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中，编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加”**using UnityEditor**

变量

◆ **Var currentCamera:Camera**

描述：用来决定 3D 处理结束的相机

类变量

◆ **Static var color:Color**

描述：handles 的颜色

◆ **Static var lighting:bool**

描述：在处理光照？

类方法

◆ **Static function BeginGUI():void**

描述：在 3DhandleGUI 中开始一个 2DGUI 块

在当前处理相机的顶部开始一个 2DGUI 块

参考：EndGUI

◆ **Static function BeginGUI(position:Rect):void**

描述：在 3DhandleGUI 中开始一个 2DGUI 块

用来制作一个输入 GUI

参考：EndGUI

◆ **Static function Button(position:Vector3,direction:Quaternion,size:float,pickSize:float,capFune:Handles,DrawCapFunction):bool**

描述：制作一个 3D 按钮

这个就像一个普通的 GUIButton，但是它有一个 3D 位置并通过一个处理函数绘制。

◆ **Static function ClearCamera(position:Rect,camera:Camera):void**

描述：

◆ **static function Disc (rotation : Quaternion, position : Vector3, axis : Vector3, size : float,cutoffPlane : bool, snap : float) : Quaternion**

参数

rotation 圆盘的旋转

position 圆盘的中心

axis 旋转的轴

size 在世界空间中圆盘的尺寸参考：HandleUtility.GetHandleSizecutoffPlane 如果为真，之后前半部分的圆被绘制/可拖动，当你用许多重叠的旋转轴时（就像在默认的旋转工具中一样），这个可用来避免混乱！

描述：制作一个可以用鼠标拖动的 3D 圆盘

◆ **static function DrawArrow (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制一个像移动工具使用箭头！

◆ **static function DrawCamera (position : Rect, camera : Camera, renderMode : int) : void**

参数

Position	在 GUI 坐标下绘制这个相机的区域
Camera	需要绘制的相机

描述：在矩形内绘制一个相机

这个函数也设置 Camera.current 为 camera，并设置它的 pixelrect

◆ **static function DrawCone (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制一个球体，传递这个到处理函数

◆ **static function DrawCube (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制以立方体，传递这个到处理函数

◆ **static function DrawCylinder (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制一个圆柱，传递这个到处理函数

◆ **static function DrawLine (p1 : Vector3, p2 : Vector3) : void**

描述：从 **p1** 到 **p2** 绘制线

◆ **static function DrawPolyLine (params points : Vector3[]) : void**

描述：绘制穿过 **points** 列表中所有点的线

◆ **static function DrawRectangle (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制一个朝向相机的矩形，传递这个到处理函数

◆ **static function DrawSolidArc (center : Vector3, normal : Vector3, from : Vector3, angle : float, radius : float) : void**

参数

center 圆的中心

normal 圆的法线

from 圆周上点的方向，相对于中心，区域开始的位置

angle 扇形的角度

radius 圆的半径

描述：在 **3D** 空间中绘制一个圆的区域（饼状）

◆ **static function DrawSolidDisc (center : Vector3, normal : Vector3, radius : float) : void**

参数

center 圆盘的中心

normal 圆盘的法线

radius 圆盘的半径

描述：在 **3D** 空间中绘制一个平的实体圆盘

◆ **static function DrawSphere (controlID : int, position : Vector3, rotation : Quaternion, size : float) : void**

描述：绘制一个球体，传递这个到处理函数

◆ **static function DrawWireArc (center : Vector3, normal : Vector3, from : Vector3, angle : float, radius :float) : void**

参数

center 圆的中心

normal 圆的法线

from 圆周上点的方向，相对于中心，圆弧开始的位置

angle 圆弧的角度

radius 圆的半径

描述：在 **3D** 空间绘制一个圆弧

◆ **static function DrawWireDisc (center : Vector3, normal : Vector3, radius : float) : void**

参数

center 圆盘的中心

normal 圆盘的法线

radius 圆盘的半径

描述：在 **3D** 空间中绘制一个平的圆盘轮廓

◆ **static function EndGUI () : void**

描述：结束 **2DGUI** 块并返回主 **3D** 句柄类型

需要，是它正确地从相机恢复

◆ **static function FreeMoveHandle (position : Vector3, rotation : Quaternion, size : float snap : Vector3, capFunc : DrawCapFunction) : Vector3**

参数

position 手柄的位置

rotation 手柄的旋转，这个可以用 **raysnapping** 改变

size 手柄的尺寸

capFunc 这个函数用来绘制手柄，例如 **Handles.DrawRectangle** 描述：制作一个未约束耳朵移动手柄

这个可以在所有方向上移动，在场景中按下 **CMD** 以便 **raysnap** 碰撞器

static function FreeRotateHandle (rotation : Quaternion, position : Vector3, size : float) : Quaternion

描述：

◆ **static function Label (position : Vector3, text : string) : void**

◆ **static function Label (position : Vector3, image : Texture) : void**

◆ **static function Label (position : Vector3, content : GUIContent) : void**

◆ **static function Label (position : Vector3, text : string, style : GUIStyle) : void**

◆ **static function Label (position : Vector3, content : GUIContent, style : GUIStyle) : void**

参数

position **3D** 空间中的位置就像从当前处理相机中看到的一样

text 显示在该标签上的文本

image 显示在标签上的纹理

content 用于这个标签的文本，图形和提示

style使用的风格，如果不设置，将使用当前的 **GUISkin** 的 **label**

描述：在 **3D** 空间中制作一个定位文本标签

标签没有用户交互，不会获取鼠标点击并总是以普通风格渲染，如果你想制作一个可视化响应用户输入的空间，使用一个 **BOX** 空间

◆ **static function PositionHandle (position : Vector3) : Vector3**

◆ **static function PositionHandle (position : Vector3, rotation : Quaternion) : Vector3**

参数

rotation 手柄的朝向，如果提供，它将决定位置手柄的朝向，这个能够通过 **snapping** 改变，因此它必须是一个引用

position 在 **3D** 空间中手柄的中心

返回：**Vector3** 修改过的旋转

描述：制作一个 **3D** 场景视位置句柄

就像内置的移动哦给你根据一样工作，如果你已经赋值了某些东西到 **Undo.SetSnapshotInfo**，它将完全可以 **Undo**，如果你已经赋了一个非 **Null** 值到

ignoreRaycastObjects，居中手柄将安全支持 raycast 定位

◆ static function RotationHandle (rotation : Quaternion, position : Vector3) : Quaternion

参数

rotation 手柄的朝向

position 在 3D 空间中手柄的中心

返回：Quaternion 修改后的旋转

描述：制作一个 3D 场景视旋转手柄

就像内置的旋转工具一样工作，如果你已经赋值了某些东西到 Undo.SetSnapshotInfob，它将完全可以 Undo

◆ static function ScaleHandle (scale : Vector3, position : Vector3, rotation : Quaternion, size : float) : Vector3

参数

scale 缩放调整

position 手柄的位置

rotation 手柄的旋转

返回：Vector3 新缩放向量

描述：制作一个 3D 场景视缩放手柄

就像内置的缩放工具一样工作

static function ScaleSlider (scale : float, position : Vector3, direction : Vector3, rotation : Quaternion, size : float, snap : float) : float

参数

scale 用户可以修改的值

position 手柄的位置

direction 手柄的方向

rotation 整个物体的旋转

size 手柄的尺寸

snap用户修改它后的新值

描述：制作一个方向缩放滑块

◆ static function ScaleValueHandle (value : float, position : Vector3, rotation : Quaternion, size : float, capFunc : DrawCapFunction, snap : float) : float

参数

value 用户可以修改的值

position 手柄的位置

rotation 手柄的旋转

size 手柄的尺寸

Snap 用户修改它后的新值

描述：制作一个可拖动浮点数的手柄

这个用来制作中心缩放手柄，用户可以点击并上下拖动一个浮点值

◆ static function SetCamera (camera : Camera) : void

◆ static function SetCamera (position : Rect, camera : Camera) : void

描述：设置当前相机，这样所有的手柄和 Gizmos 都和用它的设置来绘制

设置 Cameracurrent 为 camera 并设置它的 pixeRect 这个不绘制相机，只设置它为

激活，使用 DrawCamera.绘制它，它也为手柄工具函数设置 Event.current.mouseRay 和

Event.current.lastMouseRay

◆ static function Slider (position : Vector3, direction : Vector3) : Vector3

◆ static function Slider (position : Vector3, direction : Vector3, size : float, drawFunc :DrawCapFunction, snap : float) : Vector3

参数

position 当前点的位置

direction 滑动的方向

float手柄的 3D 尺寸 HandleUtility.GetHandleSize（位置）drawFunc 调用这个函数来实际绘制，默认地，它的 Handles.DrawArrowc 但是可以使用任何具有相同名称的函数

描述：制作一个 3D 滑块

这将在屏幕上绘制一个 3D 可拖动的手柄，这个手柄被约束为沿着 3D 空间中一个方向向量滑动

Help

用来访问 Unity 文档的辅助类

注意，这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加“usingUnityEditor”

类方法

◆ Static function BrowseURL.(url:string):void

描述：在默认浏览器中打开 url

◆ Static function HasHelpForObject(obj:Object):bool

描述：这个对象有帮助文件吗？

◆ Static function ShowHelpForObject(obj:Object):void

描述：为这个物体显示帮助文件

◆ Static function ShowHelpPage(page:string):void

描述：显示帮助页

Page 应该是帮助页的 URL，通常用 file://开头，如果 page 用 file:///unity/开始，然后它指向 Unity 半年关注。

```
//打开脚本参考
Help.ShowHelpPage ("file:///unity/ScriptReference/index.html");
```

参见：Help.ShowHelpForObject

MenuCommand

类

用来为一个 MenuItem 提取向下问，MenuCommand 对象被传递到自定义菜单项函数中，这个函数是使用 MenuItem 属性定义的

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加” usingUnityEditor

```
// 添加名为 "Do Something" 的上下文菜单
@MenuItem ("CONTEXT/Rigidbody/Do
Something")
static function DoSomething (command : MenuCommand) {
var body : Rigidbody = command.context;
body.mass = 5;
```

}

变量

◆ Var context:Object

描述：上下文是菜单命令的目标对象

通常调用上下文菜单时，上下文是当前选择的或鼠标之下的项目

◆ Var userData:int

描述：一个整数用于传递自定义信息到一个菜单项

构造函数

◆ static function MenuCommand (inContext : Object, inUserData : int) : MenuCommand

描述：创建一个新的 MenuCommand 对象

上下文和用户数据将被分别用 inContext 和 inUserdata 初始化

◆ static function MenuCommand (inContext : Object) : MenuCommand

描述：创建一个新的 MenuCommand 对象

上下文将被初始化为 inContext 用户数据将被设置为 0

MenuItem

类，继承自 System.Aunbute

MenuItem 属性允许你添加菜单项到主菜单和检视面板的上下文菜单

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor”

MenuItem 属性转化任何静态函数为一个菜单命名，只有静态函数可以使用 MenuItem 属性

为了创建一个热键你可以使用下面的特殊字符：%(cmd),#(shift),&(ait),^(control), (no key modifiers). 例如为了创建一个具有热键 cmd-alt-g 的菜单使用“GameObject/DoSomething%&g”为了创建一个具有热键 g 并不包含功能键的菜单使用”GameObject/DoSomething_g”

```
// JavaScript example:
// Add menu named "Do
Something" to the main menu
@MenuItem ("GameObject/Do Something")
static function DoSomething () {
Debug.Log ("Perform operation");
}
// Validate the menu item.
// The item will be disabled
@MenuItem ("GameObject/Do Something", true)
static function ValidateDoSomething () {
return Selection.activeTransform != null;
}
// Add menu named "Do Something" to
//
and give it a shortcut (ctrl-o on Windows, cmd-o on OS X).
```

```
@MenuItem ("GameObject/Do Something %o")static function DoSomething () {
Debug.Log ("Perform operation");
}
// 添加名为 "Do Something"的上下文菜单
@MenuItem ("CONTEXT/Rigidbody/Do Something")
static function DoSomething (command:MenuCommand) {
var body : Rigidbody = command.context;
body.mass =
5;
}
// C# 例子
using UnityEditor;
using UnityEngine;
class MenuTest : MonoBehaviour {
// 添加名为"Do Something" 的菜单到主菜单
[MenuItem ("GameObject/Do Something")]
static void DoSomething () {
Debug.Log ("Perform operation");
}
// Validate the menu item.
// The item will be disabled
if no transform is selected.
[MenuItem
("GameObject/Do Something", true)]
static bool
ValidateDoSomething () {
return Selection.activeTransform
!= null;
}
// Add menu named "Do Something" to
the main menu
//
and give it a shortcut (ctrl-o on Windows, cmd-o on OS X).
[MenuItem ("GameObject/Do Something %o")]
static void DoSomething ()
{
Debug.Log ("Perform operation");
}
// Add context menu named "Do Something" to rigid body's
context menu
[MenuItem ("CONTEXT/Rigidbody/Do Something")]
static void DoSomething
(MenuCommand command) {
Rigidbody body = (Rigidbody)command.context;
```

body.mass =

5;

}

}

构造函数

◆ static function MenuItem (itemName : string, isValidateFunction : bool, priority : int) : MenuItem

描述：创建一个菜单项，当这个菜单项被选中的时候调用跟随它的静态函数

itemName 是像一个路径名一样的表示，例如”GameObject/Do Something”如果 isValidateFunction 为真，这是一个验证函数并将在调用具有同名的菜单函数之前被调用 Priority 定义了菜单项显示在菜单栏中的顺序

◆ static function MenuItem (itemName : string, isValidateFunction : bool) : MenuItem

描述：创建一个菜单项，当这个菜单被选中的时候调用跟随它的静态函数

itemName 是像一个路径名一样的表示，例如”GameObject/Do Something”如果 isValidateFunction 为真，这是一个验证函数并将在调用具有同名的菜单函数之前被调用

◆ static function MenuItem (itemName : string) : MenuItem

描述：创建一个菜单项，当这个菜单项被选中的时候调用跟随它的静态函数

itemName 为像路径名一样表示的菜单项，例如”GameObject/Do Something”

ModelImporterClipAnimation

类

风格动画得到的动画剪辑

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”usingUnityEditor”

参见：ModelImporterClipAnimation

变量

◆ Var firstFrame:int

描述：剪辑的第一帧

◆ Var lastFrame:int

描述：剪辑的最后一帧

◆ Var loop:bool

描述：剪辑是一个循环动画？

◆ Var name:string

描述：剪辑名称

MonoScript

类，继承自 TextAsset

表示脚本资源

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”usingUnityEditor”

这个类表示存储在工程中的 C#，javaScript 和 Boo 文件

函数

function GetClass () : System.Type

描述：返回由这个脚本实现的类的 System.Type 对象

继承的成员

继承的变量

Text	**.txt** 文件的文本内容作为一个字符串
Bytes	文本资源的原始字节
Name	对象的名称
hideFlags	该物体是否隐蔽，保存在场景中或被用户修改

继承的函数

GetInstanceID	返回该物体的实例 **id**

继承的类函数

operator bool	这个物体存在吗
Instantiate	克隆 **original** 物体并返回这个克隆.
Destroy	移除一个游戏物体，组件或资源
DestroyImmediate	立即销毁物体 **obj**，强烈建议使用 **Destroy** 代替
FindObjectsOfType	返回所有类型为 **type** 的激活物件
FindObjectOfType	返回第一个类型为 **type** 的激活物体
operator ==	比较两个物体是否相同
operator !=	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 **target** 不被自动销毁

MovieImporter

类，继承自 **AssetImporter**

用于导入视频纹理的资源导入器

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中，编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加”**using UnityEditor**”

变量

Var duration:float

描述

可被导入的视频的秒数

Var quality:float

描述：导入视频时的质量设置：这是一个从 **0** 到 **1** 的浮点数

0 用来最大的压缩以便得到最小的下载尺寸，**1** 用于做好的质量，导致较大的文件，这个直线对应与视频导入期检视面板中滑竿的值，默认值为**.5**

继承的成员

继承的变量

assetPath	用于该导入器的资源的路径名（只读）
name	对象的名称
hideFlags	该物体是否被隐蔽，保存在场景中或被用户修改？

继承的函数

GetInstanceID	返回该物体的实例 **id**

继承的类函数

GetAtPath	为 **path** 处的资源取回资源导入器
operator bool	这个物体存在吗
Instantiate	克隆 **original** 物体并返回这个克隆
Destroy	移除一个游戏物体，组件或资源

DestroyImmediate	立即销毁物体 obj，强烈建议使用 Destroy 代替
FindObjectsOfType	返回所有类型为 type 的激活物体
FindObjectOfType	返回第一个类型为 type 的激活物体
operator ==	比较两个物体是否相同
operator !=	比较两个物体是否不相同
DontDestroyOnLoad	加载新场景时确保物体 target 不被自动销毁

ObjectNames

类

辅助类用来给对象构建可显示的名称

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”using UnityEditor”

类方法

static function GetClassName (obj : Object) : string

描述：

对象的类名称

参见：**ObjectNamesGetInspectorTitle**

◆ **static function GetDragAndDropTitle (obj : Object) : string**

描述：

◆ **static function GetInspectorTitle (obj : Object) : string**

描述：该对象检视面板的标题

如果这个对象是一个脚本，这将返回“scriptname(Script)”例如

参见：**ObjectNamesGetClassName,ObjectNamesNicifyVariableName**

static function NicifyVariableName (name : string) : string

描述：为一个变量制作一个可显示的名称

这个函数将在大写字母前插入一个空格并移除名称大写字幕前面可选的 m_,_或 k

```
// prints "My Variable"
print (ObjectNames.NicifyVariableName
("MyVariable"));
//
prints "The Other Variable"
print (ObjectNames.NicifyVariableName
("m_TheOtherVariable"));
// prints "Some Constant"
print (ObjectNames.NicifyVariableName
("kSomeConstant"));
```

static function SetNameSmart (obj : Object, name : string) : void

描述：设置对象的名称

如果对象是一个 Asset，重命名这个资源和文件名来匹配对象

ScriptableWizard

类，继承自 ScriptableObject

从这个类继承来创建一个编辑器向导

注意：这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加”

usingUnityEditor”

变量

◆ var errorString : string

描述

允许你设置错误的向导文本

参见：ScriptableWizard.OnWizardUpdate

◆ var helpString : string

描述：允许你设置向导的帮助文本

参见：ScriptableWizard.OnWizardUpdate

◆ var isValid : bool

描述：允许你启用或禁用向导创建按钮，这样用户不能点击它

参见：ScriptableWizard.OnWizardUpdate、

消息传递

◆ function OnDrawGizmos () : void

描述：如果没帧调用，该向导是可见的

使用这个绘制场景中绘制向导 Gizmos

参见：Gizmos class

◆ function OnWizardCreate () : void

描述：当用户点击 Create 按钮时调用。

参见：ScriptableWizard DisplayWizard

function OnWizardOtherButton(): viod

描述：

当用户点击其他按钮时，请你提供一个动作

参见：ScriptableWizard DisplayWizard

function OnWizardOtherButton(): viod

描述：

当向导被打开或当用户在向导中改变一些东西的时候调用这个

这个允许你设置 helpString errorString 并通过 Valid 来使用/禁用 Create 按钮

参见: ScriptableWizard DisplayWizard

```
Void OnWizardUpdate()  {
//设置帮助字符串
helpString
="Please set the color of the
light!";
//禁用向导创建按钮
isValid=false;
//并告诉用户为什么
errorString="You absolutely
must set the color of the light!";
}
```

类方法

static function DisplayWizard(title : string, klass : Type, createButtonName: string = "Create", otherButtonName : string = ""):ScrptableWizard

描述：

用指定的 title 创建向导

```
//C: 例子
using UnityEditor;
using UnityEngine;
class WizardCreateLight : ScriptableWizard {
public float range =
500;
public Color color = Color.red;
[MenuItem ("GameObject/Create Light Wizard:)]
static void CreateWizard
() {
ScriptableWizard.DisplayWizard("Creat Light",
typeof(WizardCreateLight),
"Create","Apply");
/如果你不想使用第二个按钮简单地留下它；
//ScriptableWizard.DispalyWizard("Create
Light",
typeof(WizardCreateLight));
}
viod
OnWizardCreate() {
GameObject
go.AddComponent("Light");
go.light.range =
range;
go.light.color = color;
}
void
OnWizardUpdate() {
helpstring = "Please set the color
of the light!";
}
//
```

当用户按下"Aplly"按钮 OnWizardOtherButton 被调用，

```
Void
OnWizardOtherButton ()
{
//简单被选中东西的颜色为红色
if (Selection.activeTransform
== null)
return;
if (Select.activeTransform.Light
== null)
return;
```

Selection.activeTransform.light.color
= Color.red;
}
}
继承的成员
name 对象的名称
hideFlags 该物体是否被隐藏，保存在场景中或被用户修改；
继承的函数
GetInstanceID 返回该物体的实例 ID
继承的消息传递
OnEnable 物体被卸载时调用该函数
OnDisable 当可编程物体超出范围时调用这个函数
继承的类函数
Createinstance 使用 className 创建一个可编程物体的实例
Operator bool 这个物体存在吗？
Instantiate 克隆 original 物体并返回这个克隆。
Destroy 移除一个游戏物体，缓存或资源。
DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。
FindObjectsOfType 返回所有类型为 type 的激活物体。
FindObjectsOfType 返回第一个类型为 type 的激活物体。
operator== 比较两个物体是否相同。
operator 1= 比较连个物体是否不相同。
DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。
Selection
类
在编辑中访问选择的对象
注意:这是一个编辑器类，为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中，编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加"using UnityEditor,"。
类变量
◆ Static var activeGameObject: GameObject
描述：返回激活的游戏物体。（这个显示在检视面板中）
这个也将返回可能是预设的游戏物体或者非可修改的物体，
◆ Static var activeInstanceID: int
描述：返回实际选择物体的 instanceID。包括预设，非可修改物体。
当使用场景的主要物体时，建议使用 Selection.activeTransform 代替。
◆ static var activeObject：Object
描述：返回实际选择的物体。包括预设，非可修改的物体。
当使用场景的主要物体时，建议使用 Selection.activeTransform 代替。
◆ static var activeTransform: Transform
描述：返回激活的变换。（这个显示在检视面板中）
这将不会返回预设或非可修改物体。
◆ static var gameObjects：GameObject[]
描述：返回实际选择的游戏物体。包含预设，非可修改物体。

当使用场景的主要物体时，建议使用 Selection.transforms 代替。

◆ static var instanceIDs：int[]

描述：

◆ static var objects：Object[]

描述：实际的未过滤选择物。

只有在场景中或层次中的对象被返回，而不是工程视图中的。你也可以赋值一个对象到

选择。

◆ static var transforms：Transform[]

描述：发那会顶层选择物，不包含预设。

当时使用场景物体时这是最常用的选择类型。

变量

var fontRenderMode: fontRenderMode

描述：

？？？

var fontSize: int

描述：

用手导入字符的字体尺寸

var fontTextureCase: FontTextureCase

描述：

使用这个来衡量那个字符应该被导入

继承的成员

继承的变量

assetPath 用于该导入器的资源的路径名（只读）

name 对象的名称

hideFlages 该物体是够被隐藏，保存在场景中或被用户修改？

继承的类函数

GetAtPath 为 path 处的资源取回协商导入器。

operator bool 这个物体存在吗？

Instantiate 克隆 original 物体并返回这个克隆。

Destroy 移除一个游戏物体，缓存或资源。

DestroyImmediate 立即销毁物体 obj，强烈建议使用 Destroy 代替。

FindObjectsOfType 返回所有类型为 type 的激活物体。

FindObjectsOfType 返回第一个类型为 type 的激活物体。

operator== 比较两个物体是否相同。

operator 1= 比较连个物体是否不相同。

DomDestroyOnLoad 卸载场景时确保物体 target 不被自动销毁。

Undo

类

让你在特定物体上注册一个撤销操作，你可能会在后面执行它。

注意：这是一个编辑器类。为了使用它你必须放置脚本到 1 程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始放置添加 "using UnityEditor,"。

类方法

◆ **static function ClearSnapshotTarget (): void**

描述：

◆ **static function ClearUndo (identifier: Object): void**

描述：

◆ **static function CreatSnapshot (): void**

描述：

◆ **static function PerformRedo (): void**

描述：执行一个重做操作。

这个与从 **Edit** 菜单中选择 **Redo** 的效果相同。

◆ **static function PerformUndo (): void**

描述：执行一个恢复操作。

这个与从 **Edit** 菜单中选择 **Undo** 的效果相同。

◆ **static function RedisterSceneUndo (name: string): void**

描述：通过保存整个场景来恢复。

这个是最简单，最稳定，但是是最慢的存储恢复操作的方式。

◆ **static function RegisterSnapshot (): void**

描述：应用由 **RegisterSnapshot** 制作的快照到撤销缓存。

◆ **static function RegisterUndo (o: Object, name: string): void**

描述：

◆ **static function RegisterUndo (identifier: Object, o: object[], name: string): void**

描述：

◆ **static function SetSnapshotTarget (objectsToUndo: Object[], name: string): void**

参数

name 需要重做的动作的名称。就像主菜单中的"Undo...",

objectToUndo 需要保存撤销信息的对象。默认地，这些是 **null** - 就是说没有撤销的信息被保护。

描述：设置通过 **GUI** 或 **Handles** 所做的修改，这样他们可被合适地撤销。

这个并不压入一个撤销（实际的操作的是那些知道何时需要这个操作的人），但是只通知

句柄调用什么操作在哪里调用并应用它。

◆ **static function SetSnapshotTarget （undoObject: Object, name: string): void**

描述：

枚举

AudioImporterChannels

用于 **AudioImporter** 的导入音频声道。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中。编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加 **"using UnityEditor,"**。

参见：**AudioImporter.Channels.**

值

◆ **AudioImporterChannels.Automatic**

描述：使用文件提供的声道

参见：AudioImporter.Channels.

◆ AudioImporterChannels.Mono

描述：作为单声道（一声道）导入。

参见：AudioImporter.Channels.

◆ AudioImporterChannels.Stereo

描述：作为立体声（双声道）导入。

参见：AudioImporter.Channels.

AudioImporterFormat

用于 AudioImporter 的导入音频格式。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：AudioImporter.format.

值

◆ AudioImporterFormat.Automatic

描述：选择格式自动化。

参见：AudioImporter.Format.

◆ AudioImporterFormat.OggVorbis

描述：Ogg Vorbis 音频。

参见：AudioImporter.Format.

◆ AudioImporterFormat.Uncompressed

描述：未压缩的原始音频。

参见：AudioImporter.Format.

BuildAssetBundleOptions

Asset Bundle 构建选项。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：BuildPipeline.BuildAssetBundle.

值

◆ BuildAssetBundleOptions.CollectDependencies

描述：包括所有依赖。

这个根据到任何资源的引用，游戏物体或组件并在构建中包含它们。

◆ BuildAssetBundleOptions.CompleteAssets

描述：强制包含所有资源。

例如如果你在传递一个网格到 BuildPipeline.BuildAssetBundle 函数并使用 CompleteAssets，

它将包含游戏物体和任何动画剪辑到同一个资源。

BuildOptions

构建选项。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加

"using

UnityEditor,"。

参见：BuildPipeline.BuildPlayer.

值

◆ BuildOptions.AudioRunPlayer

描述：运行构建后的播放器。

参见：BuildPipeline.BuildPlayer.

◆ BuildOptions.BuildAdditonalStreamedScenes

描述：用额外的流式场景构建一个 wed 播放器。

参见：BuildPipeline.BuildPlayer.

◆ BuildOptions.CopmressTextures

描述：构建时压缩纹理。

参见：BuildPipeline.BuildPlayer.

◆ BuildOptions.ShowBuildPlayer

描述：显示构建播放器。

参见：BuildPipeline.BuildPlayer.

◆ BuildOptions.StripDebugSymbols

描述：从独立版中移除调试信息。

这个用于 OS X 独立模式。

参见：BuildPipeline.BuildPlayer.

BuildTarget

目标构建平台

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using

UnityEditor,"。

参见：BuildPipeline.BuildPlayer.

值

◆ BuildTarget.DashboardWidget

描述：构建一个于 OS X Dashboard 窗口.

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.StandaloneOSXIntel

描述：构建一个 OS X 独立模式（只限于 Intel）.

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.StandaloneOSXPPC

描述：构建一个 OS X 独立模式（只限于 PowerPC）.

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.StandaloneOSXUniversal

描述：构建一个 OS X 独立模式

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.StandaloneWindows

描述：构建一个 Windows 独立运行版。

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.WebPlayer

描述：构建一个 Web 播放器。

参见：BuildPipeline.BuildPlayer.

◆ BuildTarget.WebPlayerStreamed

描述：构建一个流式 Web 播放器。

参见：BuildPipeline.BuildPlayer.

DragAndDropVisualMode

用于拖放操作的可视提示模式。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：DragAndDrop 类.

值

◆ DragAndDropVisualMode.Copy

描述：拷贝被拖动物体

参见：DragAndDrop 类.

◆ DragAndDropVisualMode.Generic

描述：通用拖动操作

参见：DragAndDrop 类.

◆ DragAndDropVisualMode.Link

描述：链接被拖动物体到目标

参见：DragAndDrop 类.

◆ DragAndDropVisualMode.Move

描述：移动杯拖动物体

参见：DragAndDrop 类.

◆ DragAndDropVisualMode.None

描述：没有标记（拖动不应进行）

参见：DragAndDrop 类.

FontRenderMode

用于 TrueTypeFontImporter 的渲染模式常量

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

值

◆ FontRenderMode.LightAntialiasing

◆ FontRenderMode.NoAntialiasing

◆ FontRenderMode.StrongAntialiasing

FontTextureCase

用于 TrueTypeFontImporter 的纹理实例常量

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

值

Unicode 导入一个 **Unicode** 字符集通常用于拉丁脚本

ASCII 导入基本的 **ASCII** 字符集

ASCIIUpperCase 只导入大写的 **ASCII** 字符集

ASCIILowerCase 只导入小写的 **ASCII** 字符集

GizmoType

决定在 **Unity** 编辑器中的一个 **gizmo** 如何被绘制或选择。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中。编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加 **"using UnityEditor,"**。

可以使用 **OR** 操作组合这些类型。参见：**DrawGizmo.**

值

GizmoType.Active

描述：如果它被激活（显示在检视面板中）绘制 **gizmo**

◆ **GizmoType.NotSelected**

描述：如果它没有被选择绘制 **gizmo**

◆ **GizmoType.Pickable**

描述：该 **gizmo** 可以在编辑器中点选。

◆ **GizmoType.Selected**

描述：如果它被选择绘制 **gizmo**

建议使用 **GizmoType.SelectedOrChild** 代替

◆ **GizmoType.SelectedOrChild**

描述：如果它或它的子被选择绘制 **Gizmo**

ImportAssetOptions

资源导入选项

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中。编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加 **"using UnityEditor,"**。

值

◆ **ImportAssetOptions.Default**

描述：导入缺省选项

◆ **ImportAssetOptions.ForceSynchronourImport**

描述：所以资源导入必须被同步完成。

默认地一些资源可以被同步导入(如，脚本可以在后台编译)。在某些情况下所以导入都需要同步；使用这个标识然后。例如，当导入一个脚本+预设时，脚本必须在预设序列化之前被完全编译，否则它可能获取旧的数据。

◆ **ImportAssetOptions.ForceUpdate**

描述：用户强制更新。它单击了 **Reimport**。

更新可能是由导入引起的，因为修改日期改变了。

◆ **ImportAssetOptions.ImportRecursive**

描述：导入包含在这个文件夹中的所有文件

◆ ImportAssetOptions.TryFastReimportFromMetaData

描述：通过从元数据中加载，允许快速重新导入资源。

这个被例如 ModelImporter 使用，它存储所有的东西在元数据中，这样可以在下载时跳过导入。

ModelImporterGenerateAnimations

用于 ModelImporter 的动画生成选项。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：ModelImporter.GenerateAnimations.

值

◆ ModelImporterGenerateAnimations.InNodes

描述：在动画物体上产生动画。

参见：ModelImporter.GenerateAnimations.

◆ ModelImporterGenerateAnimations.InOriginalRoots

描述：在动画包的根物体上生成动画。

参见：ModelImporter.GenerateAnimations.

◆ ModelImporterGenerateAnimations.InRoots

描述：在变化的根物体上创建动画。

参见：ModelImporter.GenerateAnimations.

◆ ModelImporterGenerateAnimations.None

描述：不生成动画。

参见：ModelImporter.GenerateAnimations.

ModelImporterGenerateMaterials

用于 ModelImporter 的材质生成选项。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：ModelImporter.GenerateMaterials.

值

◆ ModelImporterGenerateMaterials.None

描述：不生成材质。

参见：ModelImporter.GenerateMaterials.

◆ ModelImporterGenerateMaterials.PerSourceMaterial

描述：为这个在源资源中的材质生成一个材质。

参见：ModelImporter.GenerateMaterials.

◆ ModelImporterGenerateMaterials.PerTexture

描述：为每个使用的纹理生成一个材质。

参见：ModelImporter.GenerateMaterials.

MouseCursor

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor

中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

值

Arrow

Text

ResizeVertical

ResizeHorizontal

Link

SlideArrow

ResizeUpRight

ResizeUpLeft

PrefabType

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

值

None

Prefab

ModePrefab

PrefabInstance

ModelPrefabInstance

MissingPrefabInstance

DisconnectedPrefabInstance

DisconnectedModelPrefabInstance

ReplacePrefabOptions

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

值

Default

ConnectToPrefab

ReplaceNameBased

UseLastUploadPrefabRoof

SelectionMode

SelectionMode 可用于调整 Selection.GetTransforms 将返回的选择。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

默认任的变换选择模式是：SelectionMode.TopLevel | SelectionMode.ExcludePrefab | SelectionMode.Editable

值

◆ **SelectionMode.Assets**

描述：只返回那些资源位于 **Assets** 目录下的物体。

◆ **SelectionMode.Deep**

描述：返回选择和所有选择的子

◆ **SelectionMode.DeepAssets**

描述：如果选择包含一个文件夹，在文件层次中也包含该文件夹中的所有资源和文件夹。

◆ **SelectionMode.Editable**

描述：排除任何不可修改的物体。

这将过滤向导入的 **fbx** 文件生成的预设而不是用户创建的预设。

◆ **SelectionMode.ExcludePrefab**

描述：从选择中去掉所有预设

◆ **SelectionMode.TopLevel**

描述：只返回选择变换的最顶层物体。选择变换的子将被过滤掉。

◆ **SelectionMode.Unfiltered**

描述：返回所有选择

TextureImporterFormat

用于 **TextureImporter** 的导入纹理格式。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 **Assets/Editor** 中。编辑器类位于 **UnityEditor** 命名空间因此对于 **C#**脚本你需要在脚本开始位置添加 **"using UnityEditor,"**。

参见：**TextureImporter.textureFormat.**

值

◆ **TextureImporterFormat.Alpha8**

描述：**Alpha8** 位纹理格式

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.ARGB16**

描述：**RGBA16** 位纹理格式。

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.ARGB32**

描述：**RGBA32** 位纹理格式。

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.Automatic**

描述：选自格式化。

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.DXT1**

描述：**DXT1** 压缩纹理格式。

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.DXT5**

描述：**DXT5** 压缩纹理格式。

参见：**TextureImporter.textureFormat.**

◆ **TextureImporterFormat.RGB16**

描述：RGBA16 位纹理格式。

参见：TextureImporter.textureFormat.

◆ TextureImporterFormat.RGB24

描述：RGBA24 位纹理格式。

参见：TextureImporter.textureFormat.

TextureImporterGenerateCubemap

用于 TextureImporter 的立方贴图生成模式。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：TextureImporter.GenerateCubemap.

值

◆ TextureImporterGenerateCubemap.Cylindrical

描述：

◆ TextureImporterGenerateCubemap.NiceSpheremap

描述：

◆ TextureImporterGenerateCubemap.None

描述：

◆ TextureImporterGenerateCubemap.SimpleSpheremap

描述：

◆ TextureImporterGenerateCubemap.Spheremap

描述：

TextureImporterMipFilter

用于 TextureImporter 的 mipmap 过滤器。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：TextureImporter.mipmapFilter.

值

◆ TextureImporterMipFilter.BoxFilter

描述：Box mipmap 过滤器。

参见：TextureImporter.mipmapFilter.

◆ TextureImporterMipFilter.KaiserFilter

描述：Kaiser mipmap 过滤器。

参见：TextureImporter.mipmapFilter.

TextureImporterNPOTScale

在 TextureImporter 中用于非 2 的幂次纹理的缩放模式。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：TextureImporter.npotScale.

值

◆ TextureImporterNPOTScale.None

描述：保存非 2 的幂次纹理大小。

参见：TextureImporter.npotScale.

◆ TextureImporterNPOTScale.ToLarger

描述：缩放到较大的幂次大小。

参见：TextureImporter.npotScale.

◆ TextureImporterNPOTScale.ToNearest

描述：缩放到最近的 2 的幂次大小。

参见：TextureImporter.npotScale.

◆ TextureImporterNPOTScale.ToSmaller

描述：缩放到较小的幂次大小。

参见：TextureImporter.npotScale.

TextureImporterNormalFilter

用于 TextureImporter 的法线图过滤器。

注意：这是一个编辑器类。为了使用它你必须放置脚本到工程文件夹的 Assets/Editor 中。编辑器类位于 UnityEditor 命名空间因此对于 C#脚本你需要在脚本开始位置添加 "using UnityEditor,"。

参见：TextureImporter.normalFilter.

值

◆ TextureImporterNormalFilter.Sobel

描述：Sobel 的法线图过滤器。

参见：TextureImporter.normalFilter.

◆ TextureImporterNormalFilter.Standard

描述：标准的法线图过滤器。

参见：TextureImporter.normalFilter.