

# U3D 笔记

郑华

2018 年 8 月 22 日



# 目录

<b>第一章 基础</b>	<b>9</b>
1.1 如何将脚本与具体对象绑定 . . . . .	9
1.2 序列化- [SerializedField] . . . . .	9
1.3 常用技巧 . . . . .	10
1.4 MonoBehaviour 生命周期、渲染管线 . . . . .	11
1.4.1 脚本渲染流程 . . . . .	11
1.4.2 核心方法 . . . . .	13
1.5 Unity 委托 . . . . .	13
1.6 Unity 协程 . . . . .	14
1.6.1 开启方式 . . . . .	14
1.6.2 终止方式 . . . . .	14
1.6.3 yield 方式 . . . . .	15
1.6.4 执行原理 . . . . .	15
<b>第二章 事件</b>	<b>17</b>
2.1 必然事件 . . . . .	17
2.2 碰撞事件 . . . . .	17
2.3 触发器事件 . . . . .	17
<b>第三章 实体-人物、物体、组件</b>	<b>19</b>

3.1	实体类	19
3.2	Prefabs -预设体	20
3.2.1	预设动态加载到场景	20
3.3	获取实体上的组件	22
3.4	物理作用实体类	22
<b>第四章</b>	<b>世界空间相关 3D 基础</b>	<b>25</b>
4.1	Transform 类	25
4.1.1	位置	25
4.1.2	旋转	25
4.1.3	缩放	25
4.1.4	平移	25
4.1.5	Transform.localPosition	25
4.1.6	注意	26
4.2	摄像机 -Camera	26
4.2.1	Clear Flags	26
4.2.2	Culling Mask -剔除遮罩	26
4.2.3	Projection -透视模式	26
4.2.4	Clipping Planes -裁剪模式	26
4.2.5	Viewport Rect	27
4.2.6	Depth -控制渲染顺序	27
4.2.7	Rendering Path -渲染路径	27
4.2.8	Target Texture -目标纹理	27
4.2.9	HDR -高动态光照渲染	27
4.3	3D 模型	28
4.3.1	Mesh	28

4.3.2	Texture	28
4.3.3	Material	28
4.3.4	骨骼动画	28
<b>第五章</b>	<b>键盘鼠标控制</b>	<b>29</b>
5.1	普通按键 -keyDown(KeyCode xx)	29
5.2	根据输入设备 -getAxis()	29
<b>第六章</b>	<b>时间</b>	<b>31</b>
6.1	Time 类	31
<b>第七章</b>	<b>数学</b>	<b>33</b>
7.1	Random 类	33
7.2	Mathf 类	33
<b>第八章</b>	<b>光照</b>	<b>35</b>
8.1	光照	35
8.2	烘焙	35
<b>第九章</b>	<b>寻路</b>	<b>37</b>
9.1	简介	37
9.2	流程	37
<b>第十章</b>	<b>UGUI</b>	<b>39</b>
10.1	Spirit	39
10.2	Canvas	39
10.2.1	Screen Space-Overlay -覆盖模式	40
10.2.2	Screen Space-Camera -摄像机模式	40
10.2.3	World Space -世界空间模式	41

10.2.4	使用总结	42
10.2.5	Canvas Scalar	42
10.2.6	Layer	42
10.3	RectTransform	42
10.3.1	Pivot(中心)	42
10.3.2	锚点- 自适应屏幕	42
10.3.3	sizeDelta	47
10.3.4	RectTransform.rect	47
10.3.5	示例	47
10.3.6	FramDebug	48
10.4	按钮	48
10.4.1	原始 Button	48
10.4.2	Image 等 -添加 button 组件	48
10.4.3	添加事件处理脚本	48
10.5	文本- Text	48
10.5.1	添加文字阴影 -shadow 组件	48
10.5.2	添加文子边框 -outline 组件	49
10.6	图片- ImageView	49
10.7	选中标记- Toggle	49
10.8	滚动区域、滚动条	49
10.9	其他工具条	49
10.10	布局- Layout	49
10.10.1	grid layout group	49
10.10.2	horizontal layout group	50
10.10.3	vertical layout group	50

<b>第十一章 物理</b>	<b>51</b>
11.1 流程	51
11.2 刚体	51
11.3 碰撞器	51
11.4 物理材质	51
11.5 触发器	51
11.6 射线	51
11.7 关节	51
<b>第十二章 动画</b>	<b>53</b>
12.1 编辑动画	53
12.2 动画系统配置	53
12.3 代码控制动画	53
<b>第十三章 粒子系统</b>	<b>55</b>
<b>第十四章 着色器渲染</b>	<b>57</b>
<b>第十五章 资源管理- AssetBundle</b>	<b>59</b>
15.1 工作流程	59
15.2 创建	59
15.2.1 BuildAssetBundle	59
15.2.2 BuildStreamedSceneAssetBundle	60
15.2.3 BuildAssetBundleExplicitAssetNames	61
15.3 使用	61
15.4 卸载	61
15.5 内存模型	61

15.6 其他 . . . . .	61
<b>第十六章 Editor 扩展</b>	<b>63</b>
16.1 流程 . . . . .	63
16.2 在编辑器上增加一个 MenuItem . . . . .	63
16.3 创建一个对话框 . . . . .	63
16.4 扩展 Inspector 面板 . . . . .	63
16.5 编辑器插件常用函数 . . . . .	63
16.5.1 资源导入回调函数 . . . . .	63
16.6 一些常用的 Inspector 属性设置 . . . . .	64
16.7 参考 . . . . .	64
<b>第十七章 跨平台发布 apk</b>	<b>65</b>
17.1 流程 . . . . .	65
17.2 Apk 安装常见错误 . . . . .	65
<b>第十八章 调试技巧</b>	<b>67</b>
18.1 以父类为基点 . . . . .	67



# 第一章 基础

入门参考: <https://unity3d.com/cn/learn/tutorials>

## 1.1 如何将脚本与具体对象绑定

1. 右键asset 文件夹, 创建 C# 脚本
2. 编写脚本
3. 将asset 中的脚本拖拽到 Hierarchy 视图中的MainCamera 中
4. 如果脚本是作用于场景中的某个物体, 则将该脚本拖拽到该物体上

## 1.2 序列化- [SerializeField]

通常情况下, GameObject 上挂的 MonoBehaviour 脚本中的私有变量不会显示在 *Inspector* 面板上, 即不会被序列化。

但如果指定了 `SerializedFiled` 特性, 就可以被序列化了。

```
public class Test : MonoBehaviour
{
    public string Name;
    [SerializeField]
    private int Hp;
}
```

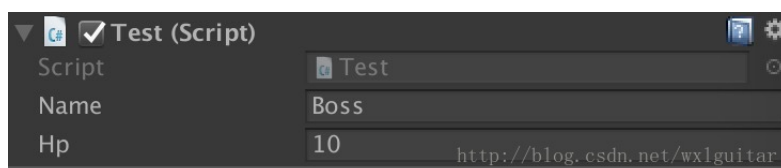


图 1.1: 序列化操作 -在 Inspector 上显示

## 1.3 常用技巧

- `ctrl + d` 复制
- `shift + 鼠标` 等比例缩放
- `shift + alt + 鼠标` 原地等比例缩放
- 在Unity 编辑器中输入汉字 需要借助其他文本拷贝粘贴
- `q`、`w`、`e`、`r`、`t` 在操作 UI 时尽量使用 `T`，以避免 `z` 轴发生的变化

## 1.4 MonoBehaviour 生命周期、渲染管线

### 1.4.1 脚本渲染流程

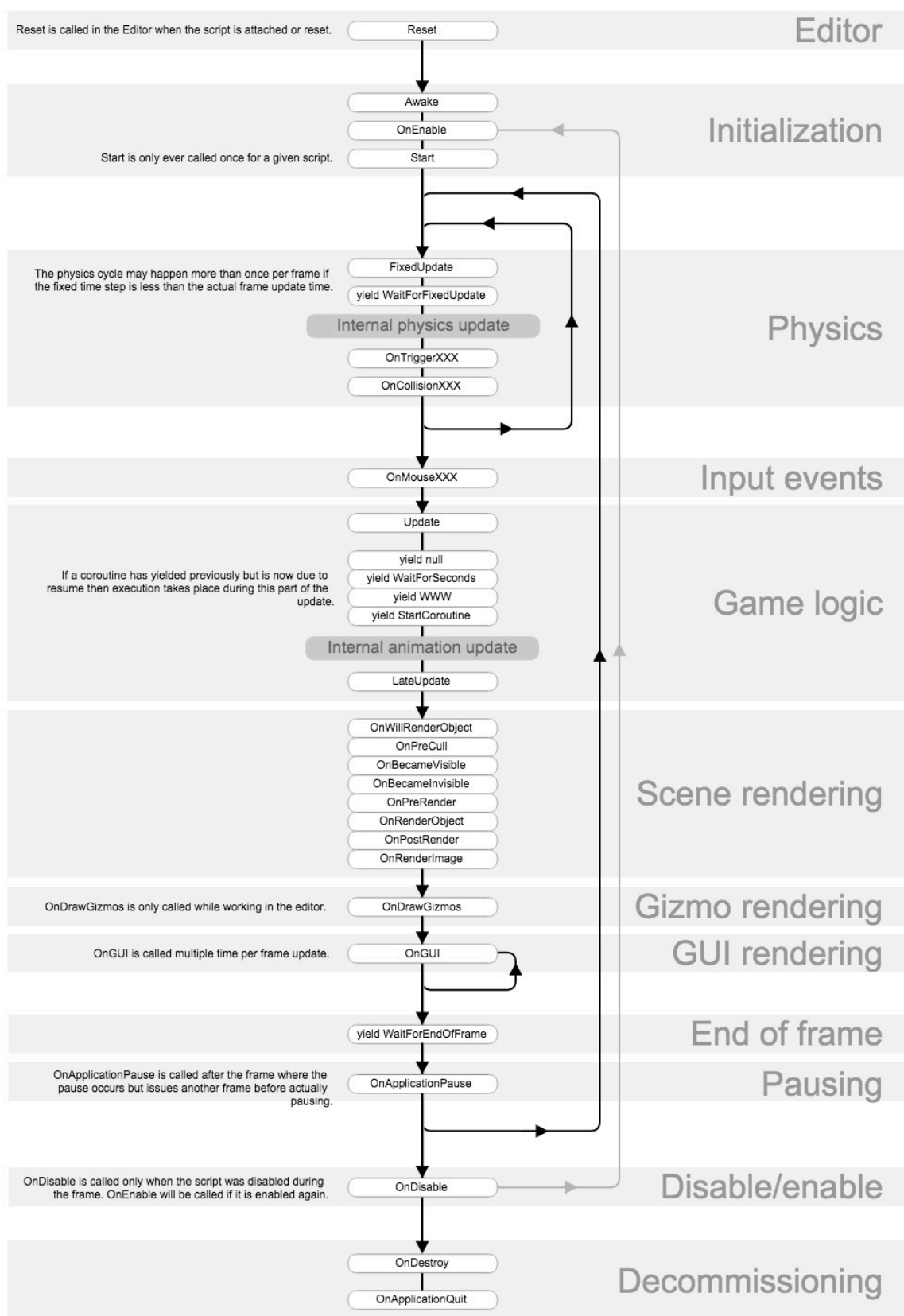


图 1.2: 脚本生命周期核心方法

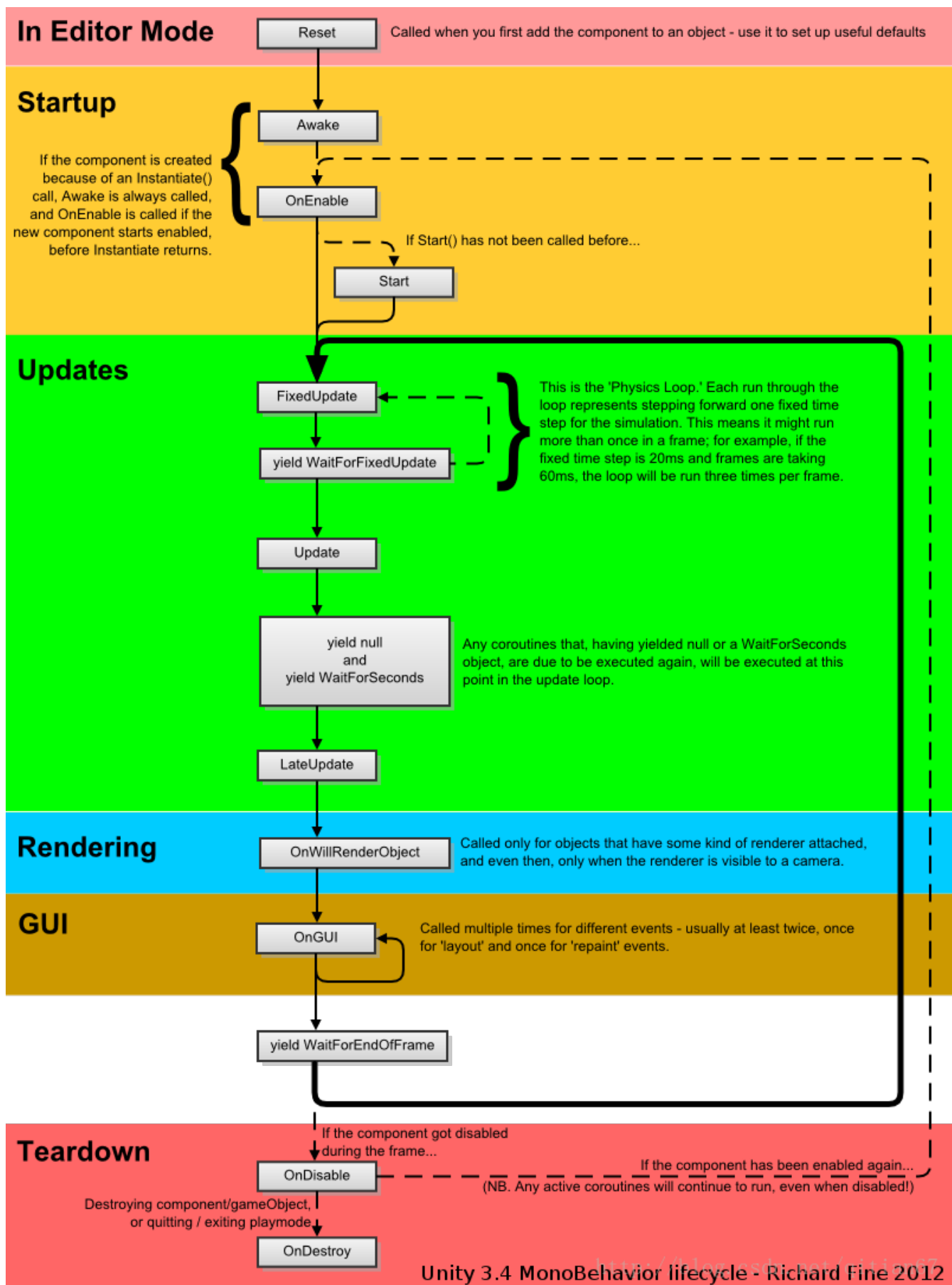


图 1.3: 简要核心方法

update: 当其所在的物体属于未激活的话 (`active`为`false`), 该物体上所有脚本中包含的协程代码都是不会被执行的。

## 1.4.2 核心方法

1. `Reset` :
2. `Awake` : 脚本唤醒函数, 当游戏对象被创建的时候, 游戏对象绑定的脚本会在该帧 (`Frame`) 内执行 `Awake()` 函数, 无论脚本是否处于激活 (`enable`) 状态。
3. `OnEnable` : 激活函数, 当脚本被激活时调用。
4. `Start` : 该函数在脚本被激活的时候执行, 位于 `Awake` 之后, 该函数同样也是在游戏对象被创建的帧里, 不同的是, 如果脚本处于不激活状态 (`MonoBehaviour.enable = false`), `start` 函数是不会执行的。
5. `FixedUpdate` :
6. `yield WaitForFixedUpdate` :
7. `OnTriggerXXX` :
8. `Update` : 只要处于激活状态的脚本, 都会在每一帧里调用 `Update()` 函数, 该函数也是最为常用的一个函数, 用来更新逻辑。
9. `LateUpdate` : 延迟更新函数
10. `OnWillRenderObject` :
11. `OnGUI` : 绘制界面函数。
12. `yield WaitForEndOfFrame` :
13. `OnDisable` :
14. `OnDestroy` : 在当前脚本销毁时调用该函数。

## 1.5 Unity 委托

定义 `public delegate void MyDelegate(int num);`

委托就是C# 封装的 C++ 的函数指针。

定义一个委托 `MyDelegate`, 如同定义一个类一样, 此时的委托没有经过实例化是无法使用

的，而他的实例化必须接收一个返回值和参数都与他等同的函数，此处的委托 `MyDelegate` 只能接收返回值为 `void`，参数为一个 `int` 的函数

**实例化委托** : `MyDelegate _MyDelegate=new MyDelegate(TestMod);`

以 `TestMod` 函数实例化一个 `MyDelegate` 类型的委托 `_MyDelegate`，此处 `TestMod` 函数的定义就应如下：

```
public void TestMod(int _num);
```

之后调用 `_MyDelegate(100)` 时就完全等同于调用 `TestMod(100)`

## 1.6 Unity 协程

### 1.6.1 开启方式

协程：协同程序，在主程序运行的同时，开启另外一段逻辑处理，来协同当前程序的执行。

**StartCoroutine(string MethodName)**

- 参数是方法名
- 形参方法可以有返回值

**StartCoroutine(IEnumerator method)**

- 参数是方法名 (`TestMethod()`)，方法中可以包含多个参数
- `IEnumerator` 类型的方法不能含有 `ref` 或者 `out` 类型的参数，但可以含有被传递的引用
- 必须有有返回值，且返回值类型为 `IEnumerator`，返回值使用 (`yield return + 表达式或者值`，或者 `yield break`) 语句

### 1.6.2 终止方式

**StopCoroutine(string MethodName)** 只能终止指定的协程

**StopAllCoroutine()** 终止所有协程

### 1.6.3 yield 方式

**yield return** 挂起，程序遇到yield 关键字时会被挂起，暂停执行，等待条件满足时从当前位置继续执行

- `yield return 0` or `yield return null`: 程序在下一帧中从当前位置继续执行
- `yield return 1,2,3,...`: 程序等待 1, 2, 3... 帧之后从当前位置继续执行
- `yield return new WaitForSeconds(n)`: 程序等待 n 秒后从当前位置继续执行
- `yield new WaitForEndOfFrame()`: 在所有的渲染以及 GUI 程序执行完成后从当前位置继续执行
- `yield new WaitForFixedUpdate()`: 所有脚本中的 `FixedUpdate()` 函数都被执行后从当前位置继续执行
- `yield return WWW()`: 等待一个网络请求完成后从当前位置继续执行
- `yield return StartCoroutine()`: 等待一个协程执行完成后从当前位置继续执行

**yield break** 如果使用yield break 语句，将会导致如果协程的执行条件不被满足，不会从当前的位置继续执行程序，而是直接从当前位置跳出函数体，回到函数的根部

相当于: `return;` + 暂停

### 1.6.4 执行原理

协程函数的返回值是 `IEnumerator`，它是一个迭代器，可以把它当成执行一个序列的某个节点的指针，它提供了两个重要的接口，分别是 `Current`(返回当前指向的元素) 和 `MoveNext()`(将指针向后移动一个单位，如果移动成功，则返回 `true`)

`yield` 关键词用来声明序列中的下一个值或者是一个无意义的值，如果使用 `yield return x`(`x` 是指一个具体的对象或者数值) 的话，那么 `MoveNext` 返回为 `true` 并且 `Current` 被赋值为 `x`，如果使用 `yield break` 使得 `MoveNext()` 返回为 `false`

如果 `MoveNext` 函数返回为 `true` 意味着协程的执行条件被满足，则能够从当前的位置继续往下执行。否则不能从当前位置继续往下执行。

委托 + 协程 <https://blog.csdn.net/q992817263/article/details/51514449>

- 实现延时
- 实现给定函数传参
- 实现特定功能

```
// 延时执行

// <param name="action">执行的委托</param>
// <param name="obj">委托的参数</param>
// <param name="delaySeconds">延时等待的秒数</param>
public IEnumerator DelayToInvokeDo(Action<GameObject> action, GameObject obj, float
    delaySeconds)
{
    yield return new WaitForSeconds(delaySeconds); // delaySeconds 后执行
    action(obj); // 特定功能
}
// 使用例子
StartCoroutine(
    DelayToInvokeDo(
        delegate(GameObject task) {
            task.SetActive(true);
            task.transform.position = Vector3.zero;
            task.transform.rotation = Quaternion.Euler(Vector3.zero);
            task.doSomethings();
        },
        /*传参*/GameObject.Find("task1"),
        1.5f)/*End 匿名委托*/
    );/*End 协程初始*/
```



## 第二章 事件

### 2.1 必然事件

继承自MonoBehaviour 类后，会自动按序提供以下方法：

- `Awake()`: 在加载场景时运行，用于在游戏开始前完成变量初始化、以及游戏状态之类的变量。
- `Start()`: 在第一次启动游戏时执行，用于游戏对象的初始化，在`Awake()` 函数之后。
- `Update()`: 是在每一帧运行时必须执行的函数，用于更新场景和状态。
- `FixedUpdate()`: 与`Update()` 函数相似，但是在固定的物理时间后间隔调用，用于物理状态的更新。
- `LateUpdate()`: 是在`Update()` 函数执行完成后再次被执行的，有点类似收尾的东西。

### 2.2 碰撞事件

U3D 的碰撞检测。具体分为三个部分进行实现，碰撞发生进入时、碰撞发生时和碰撞结束，理论上不能穿透

- `OnCollisionEnter(Collision collision)` 当碰撞物体间刚接触时调用此方法
- `OnCollisionStay(Collision collision)` 当发生碰撞并保持接触时调用此方法
- `OnCollisionExit(Collision collision)` 当不再有碰撞时，既从有到无时调用此函数

### 2.3 触发器事件

类似于红外线开关门，有个具体的范围，然后进入该范围时，执行某种动作，离开该范围时执行某种动作。类似于物体于一个透明的物体进行碰撞检测，理论上需要穿透，在 U3D 中通过

勾选 `Is Trigger` 来确定该物体是可以穿透的。

- `OnTriggerEnter()` 当其他碰撞体进入触发器时，执行该方法
- `OnTriggerStay()` 当其他碰撞体停留在该触发器中，执行该方法
- `OnTriggerExit()` 当碰撞体离开该触发器时，调用该方法

## 第三章 实体-人物、物体、组件

### 3.1 实体类

`GameObject` 类, 游戏基础对象, 用于填充世界。

**复制** `Instantiate(GameObject)` 或 `Instantiate(GameObject, position, rotation)`

- `GameObject` 指生成克隆的游戏对象, 也可以是 `Prefab` 的预制品
- `position` 克隆对象的初始位置, 类型为 `Vector3`
- `rotation` 克隆对象的初始角度, 类型为 `Quaternion`

**销毁** `Destroy(GameObject xx)`- 立即销毁 或 `Destroy(GameObject xx, Time time)`- 几秒后销毁

**可见否** 通过设置该参数调整该实体是否可以在游戏中显示, 具体设置方法为 `gameObject.SetActive(true)`

**游戏中获取**

1. 在整个场景中寻找名为 `xx` 的游戏对象, 并赋予 `obj` 变量

```
obj = GameObject.Find("xx");
```

2. 当需要获取某个 `gameObject` 下的组件时, 使用 `Transform.Find.GetComponent`

```
gameObjVar1.transform.Find("ImageItemIcon/TextMonthCardLeftDays").GetComponent<Text>().text = "MonthCardLeftDays";
```

3. 但是设置为激活状态则需要如下通过 `gameObject` 进行设置。

```
obj = gameObjVar1.transform.Find("xx/xx").gameObject.SetActive(true);
```

## 3.2 Prefabs -预设体

prefabs 基础: <https://www.cnblogs.com/yuyaonorthroad/p/6107320.html>

动态加载 Prefabs: <https://blog.csdn.net/linshuhe1/article/details/51355198>

在进行一些功能开发的时候,我们常常将一些能够复用的对象制作成prefab 的预设物体,然后将预设体存放到 Resources 目录之下,使用时再动态加载到场景中并进行实例化。例如:子弹、特效甚至音频等,都能制作成预设体。

**概念** 组件的集合体,预制物体可以实例化成游戏对象。

**作用** 可以重复的创建具有相同结构的游戏对象。

### 3.2.1 预设动态加载到场景

预设体资源加载 ->

假设预设体的位置为下图所示

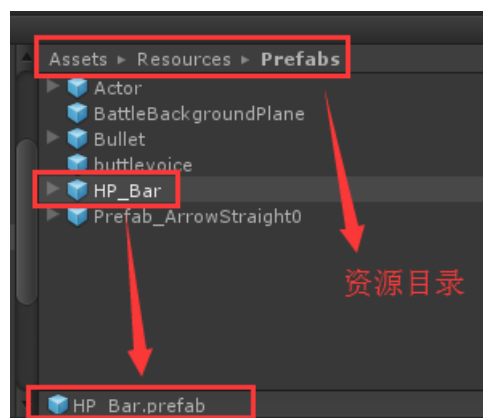


图 3.1: Prefab 资源位置

```
//加载预设体资源
```

```
GameObject hp_bar = (GameObject)Resources.Load("Prefabs/HP_Bar");
```

通过上述操作,实现从资源目录下载入HP\_Bar.prefab 预设体,用一个GameObject 对象来存放,此时该预设物体并未真正载入到场景中,因为还未进行实例化操作。

预设体实例化 ->

实例化使用的是MonoBehaviour.Instantiate 函数来完成的，其实质就是从预设体资源中克隆出一个对象，它具有与预设体完全相同的属性，并且被加载到当前场景中

完成以上代码之后，在当前场景中会出现一个实例化之后的对象，并且其父节点默认为当前的场景最外层，如下图所示。

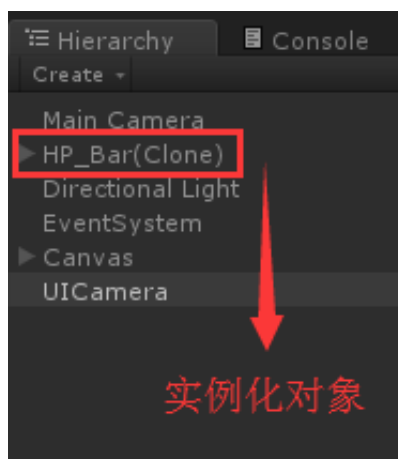


图 3.2: Prefab 实例后位置

### 实例化对象属性设置 ->

完成上述步骤之后，我们已经可以在场景中看到实例化之后的对象，但是通常情况下我们希望我们的对象之间层次感分明，而且这样也方便我们进行对象统一管理，而不是在 Hierarchy 中看到一大堆并排散乱对象，所以我们需要为对象设置名称以及父节点等属性。

-->Notice: 常见错误：对未初始化的hp\_bar 进行属性设置，设置之后的属性在实例化之后无法生效。这是因为我们最后在场景中显示的其实并非实例化前的资源对象，而是一个克隆对象，所以假如希望设置的属性在最后显示出来的对象中生效，我们需要对实例化之后的对象进行设置。

正确的设置代码如下，可以看到实例化对象已成功挂在到父节点 Canvas 上，在层次视图效果如下图所示：

```
GameObject hp_bar = (GameObject)Resources.Load("Prefabs/HP_Bar");

//搜索画布的方法！
GameObject mUICanvas = GameObject.Find("Canvas");
hp_bar = Instantiate(hp_bar);
hp_bar.transform.parent = mUICanvas.transform;
```

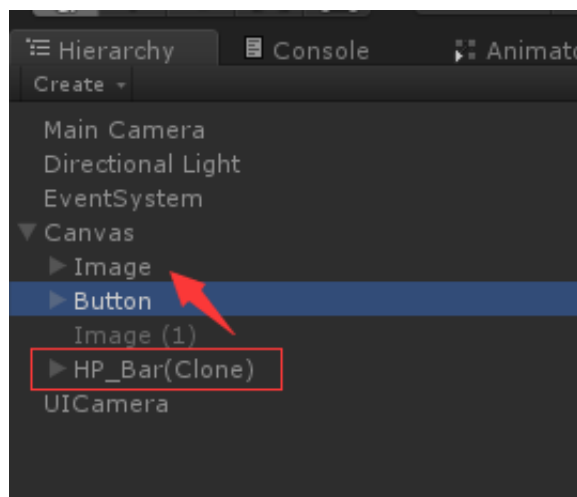


图 3.3: Prefab 对象设置父子关系

**简化写法** 上述实例步骤与属性设置代码可以简化为

```
GameObject hp_bar = (GameObject)Instantiate(Resources.Load("Prefabs/HP_Bar"));
GameObject mUICanvas = GameObject.Find("Canvas");
hp_bar.transform.parent = mUICanvas.transform;
```

**预制体添加脚本** 在预制体上不能直接添加脚本，首先需要将其拖入场景，然后再对其操作，这个时候可以添加脚本，添加组件等，在完成这些操作后，在 Inspector 选项中选中 Apply, 然后删除其在场景中的刚才拖过来的，即可。

### 3.3 获取实体上的组件

**调用方式** `GameObject.GetComponent<Type>().xx = xx;`

- `cube1.GetComponent<Rigidbody>().mass = 20;` //设置重量
- `cube1.GetComponent<BoxCollider>().isTrigger = true;` //开启 Trigger 穿透方式检测
- `cube2.GetComponent<Test>().enable = false;` //禁用 Test 脚本

### 3.4 物理作用实体类

**Rigidbody** 类，一种特殊的游戏对象，该类对象可以在物理系统的控制下来运动。

**AddForce()** 此方法调用时`rigidBody.AddForce(1, 0, 0);`，会施加给刚体一个瞬时力，在力的作用下，会产生一个加速度进行运动。

**AddTorque()** 给刚体添加一个扭矩。

**Sleep()** 使得刚体进入休眠状态，且至少休眠一帧。类似于暂停几帧的意思，这几帧不进行更新、理论位置也不进行更新。

**WakeUp()** 使得刚体从休眠状态唤醒。





## 第四章 世界空间相关 3D 基础

### 4.1 Transform 类

<https://blog.csdn.net/yangmeng13930719363/article/details/51460841>

#### 4.1.1 位置

```
transform.position = new Vector3(1, 0, 0);
```

#### 4.1.2 旋转

```
transform.Rotate(x, y, z);
```

```
transform.eulerAngles = new Vector3(x, y, z);
```

#### 4.1.3 缩放

```
transform.localScale(x, y, z); // 基准为 1、1、1，数为缩放因子。
```

#### 4.1.4 平移

```
transform.Translate(x, y, z);
```

#### 4.1.5 Transform.localPosition

`position` 是世界坐标中的位置，可以理解为绝对坐标

`localPosition` 是相对于父对象的位置，是相对坐标，既父级窗体为原点坐标

### 4.1.6 注意

在变化的过程中需要乘以 `Time.deltaTime` , 否则会出现大幅不连贯的画面。

## 4.2 摄像机 -Camera

### 4.2.1 Clear Flags

清除标记。决定屏幕的哪部分将被清除。一般用户使用对台摄像机来描绘不同游戏对象的情况，有 3 中模式选择：

- **Skybox:** 天空盒。默认模式。在屏幕中的空白部分将显示当前摄像机的天空盒。如果当前摄像机没有设置天空盒，会默认用 Background 色。
- **Solid Color:** 纯色。选择该模式屏幕上的空白部分将显示当前摄像机的 background 色。
- **Depth only:** 仅深度。该模式用于游戏对象不希望被裁剪的情况。
- **Dont Clear:** 不清除。该模式不清除任何颜色或深度缓存。其结果是，每一帧渲染的结果叠加在下一帧之上。一般与自定义的 shader 配合使用。

### 4.2.2 Culling Mask -剔除遮罩

剔除遮罩，选择所要显示的layer, 摄像机将看到勾选的层，忽略未被勾选的层。

### 4.2.3 Projection -透视模式

透视 摄像机模式，截锥体

正交 前后显示一样，不存在远小近大的样子。长方体

### 4.2.4 Clipping Planes -裁剪模式

剪裁平面。摄像机开始渲染与停止渲染之间的距离。

### 4.2.5 Viewport Rect

标准视图矩形。用四个数值来控制摄像机的视图将绘制在屏幕的位置和大小，使用的是屏幕坐标系，数值在 0 1 之间。坐标系原点在左下角。

### 4.2.6 Depth -控制渲染顺序

深度。用于控制摄像机的渲染顺序，较大值的摄像机将被渲染在较小值的摄像机之上。

### 4.2.7 Rendering Path -渲染路径

渲染路径。用于指定摄像机的渲染方法。

Use Player Settings: 使用Project Settings-->Player 中的设置。

Forward: 快速渲染。摄像机将所有游戏对象按每种材质一个通道的方式来渲染。

Deferred: 延迟光照 Legacy Vertex Lit: 顶点光照。摄像机将对所有的游戏对象座位顶点光照对象来渲染。

Legacy Deferred Lighting: 延迟光照。摄像机先对所有游戏对象进行一次无光照渲染，用屏幕空间大小的 Buffer 保存几何体的深度、法线已经高光强度，生成的 Buffer 将用于计算光照，同时生成一张新的光照信息 Buffer。最后所有的游戏对象会被再次渲染，渲染时叠加光照信息 Buffer 的内容。

### 4.2.8 Target Texture -目标纹理

用于将摄像机视图输出并渲染到一张贴图SSS。一般用于制作导航图或者画中画等效果。

### 4.2.9 HDR -高动态光照渲染

高动态光照渲染。用于启用摄像机的高动态范围渲染功能。

## 4.3 3D 模型

### 4.3.1 Mesh

### 4.3.2 Texture

### 4.3.3 Material

尽管是近似的灰色，也同样会因为材质的不同显示出不同的效果，入灰色 T 恤衫和灰色不锈钢，首先是他们对光照的反应不同（漫反射、平面），其次是表面的各种属性。

### 4.3.4 骨骼动画

## 第五章 键盘鼠标控制

### 5.1 普通按键 -keyDown(KeyCode xx)

#### 方式一

- 定义按键码: KeyCode keycode;
- 判断键是否被按下: if(Input.GetKeyDown(keycode)){}
- 在Inspirit -> Keycode 指定关联按键

#### 方式二

- 在Update 中更新添加如下代码
- if(Input.GetKeyDown(KeyCode.UpArrow))
- KeyCode.xx 包括了键盘所有的按键,常用的 AWSD 如下,鼠标同 (Input.GetMouseButtonDown(0) 0 左键, 1 右键)
  - if (Input.GetKeyDown(KeyCode.S)) 按下 S 键
  - if (Input.GetKey(KeyCode.S)) 按住 S 键
  - if (Input.GetKeyUp(KeyCode.W)) 抬起 S 键

### 5.2 根据输入设备 -getAxis()

参数分为两类:

#### 一、触屏类

1. Mouse X 鼠标沿屏幕 X 移动时触发 Mouse Y 鼠标沿屏幕 Y 移动时触发 Mouse ScrollWheel 鼠标滚轮滚动是触发

```
float mouseX = Input.GetAxis("Mouse_X");  
float mouseY = Input.GetAxis("Mouse_Y");  
  
transform.Rotate(Vector3.Up * mouseX * rotateSpeed); // 根据具体需求进行操作
```

## 二、键盘类

1. Vertical 键盘按上或下键时触发
2. Horizontal 键盘按左或右键时触发

```
float horizontal = Input.GetAxis("Horizontal");  
float vertical = Input.GetAxis("Vertical");  
  
Vector3 desPos = (transform.forward * vertical + transform.right * horizontal) *  
    Time.deltaTime * moveSpeed;  
  
_rigidBody.position += desPos;
```

返回值是一个数，正负代表方向

## 第六章 时间

### 6.1 Time 类

该类是 U3D 在游戏中获取时间信息的接口类。常用变量如下：

表 6.1: 时间变量对照表

变量名	意义
<code>time</code>	单位为秒
<code>deltaTime</code>	从上一帧到当前帧消耗的时间
<code>fixedTime</code>	最近 <code>FixedUpdate</code> 的时间，从游戏开始计算
<code>fixedDeltaTime</code>	物理引擎和 <code>FixedUpdate</code> 的更新时间间隔
<code>timeSceneLevelLoad</code>	从当前 Scene 开始到目前为止的时间
<code>realTimeSinceStartup</code>	程序已经运行的时间
<code>frameCount</code>	已经渲染的帧的总数





## 第七章 数学

### 7.1 Random 类

随机数类

### 7.2 Mathf 类

数学类



# 第八章 光照

## 8.1 光照

## 8.2 烘焙

**简介** 只有静态场景才能完成烘焙（Bake）操作，其目的是在游戏编译阶段完成光照和阴影计算，然后以贴图的形式保存在资源中，以这种手段避免在游戏运行中计算光照而带来的 CPU 和 GPU 损耗。

- **如果不烘焙：**游戏运行时，这些阴影和反光是由 CPU 和 GPU 计算出来的。
- **如果烘焙：**游戏运行时，直接加载在编译阶段完成的光照和阴影贴图，这样就不用再进行计算，节约资源。

**流程**



# 第九章 寻路

## 9.1 简介

NPC 完成自动寻路的功能。

## 9.2 流程

- 将静态场景调至 (Navigation Static)
- 烘焙
- 添加 Navigation Mesh Agent 寻路组件
- 在脚本中设置组件的目标地址，添加目标



# 第十章 UGUI

在脚本中使用时记得加上 `using UnityEngine.UI`

<https://blog.csdn.net/wangmeiqiang/article/category/6364468>

## 10.1 Spirit

在 UI 系统中，所有的图片的显示都必须通过 *Spirit*。

如果建立工程时选择的是 2D 工程，那么导入的所有图片会自动设置为 *Spirit*。

如果建立工程时选择的是 3D 工程，那么导入的所有图片需要手动的设置为 *Spirit.Inspector* -> *Texture* 最后点击 Apply 保存更改。

## 10.2 Canvas

Canvas 画布是承载所有 UI 元素的区域。Canvas 实际上是一个游戏对象上绑定了 Canvas 组件。

所有的 UI 元素都必须是 Canvas 的子对象。如果场景中没有画布，那么我们创建任何一个 UI 元素，都会自动创建画布，并且将新元素置于其下。

在 Canvas 的 Render Mode 中有三个选择：

1. Screen Space - Overlay 屏幕最上层，主要是 2D 效果。
2. Screen Space - Camera 绑定摄像机，可以实现 3D 效果。
3. World Space 世界空间，让 UI 变成场景中的一个物体。

### 10.2.1 Screen Space-Overlay -覆盖模式

Screen Space-Overlay（屏幕控件-覆盖模式）的画布会填满整个屏幕空间，并将画布下面的所有的 UI 元素置于屏幕的最上层，或者说画布的画面永远“覆盖”其他普通的 3D 画面，如果屏幕尺寸被改变，画布将自动改变尺寸来匹配屏幕

Screen Space-Overlay 模式的画布有 Pixel Perfect 和 Sort Layer 两个参数：

1. Pixel Perfect: 只有RenderMode 为 Screen 类型时才有的选项。使 UI 元素像素对应，效果就是边缘清晰不模糊。
2. Sort Layer: Sort Layer 是 UGUI 专用的设置，用来指示画布的深度。

### 10.2.2 Screen Space-Camera -摄像机模式

与 Screen Space-Overlay 模式类似，画布也是填满整个屏幕空间，如果屏幕尺寸改变，画布也会自动改变尺寸来匹配屏幕。

不同的是，在该模式下，画布会被放置到摄影机前方。在这种渲染模式下，画布看起来绘制在一个与摄影机固定距离的平面上。所有的 UI 元素都由该摄影机渲染，因此摄影机的设置会影响到 UI 画面。在此模式下，UI 元素是由perspective 也就是视角设定的，视角广度由Filed of View 设置。

这种模式可以用来实现在 UI 上显示 3D 模型的需求，比如很多 MMO 游戏中的查看人物装备的界面，可能屏幕的左侧有一个运动的 3D 人物，左侧是一些 UI 元素。通过设置 Screen Space-Camera 模式就可以实现上述的需求，效果如下图所示：

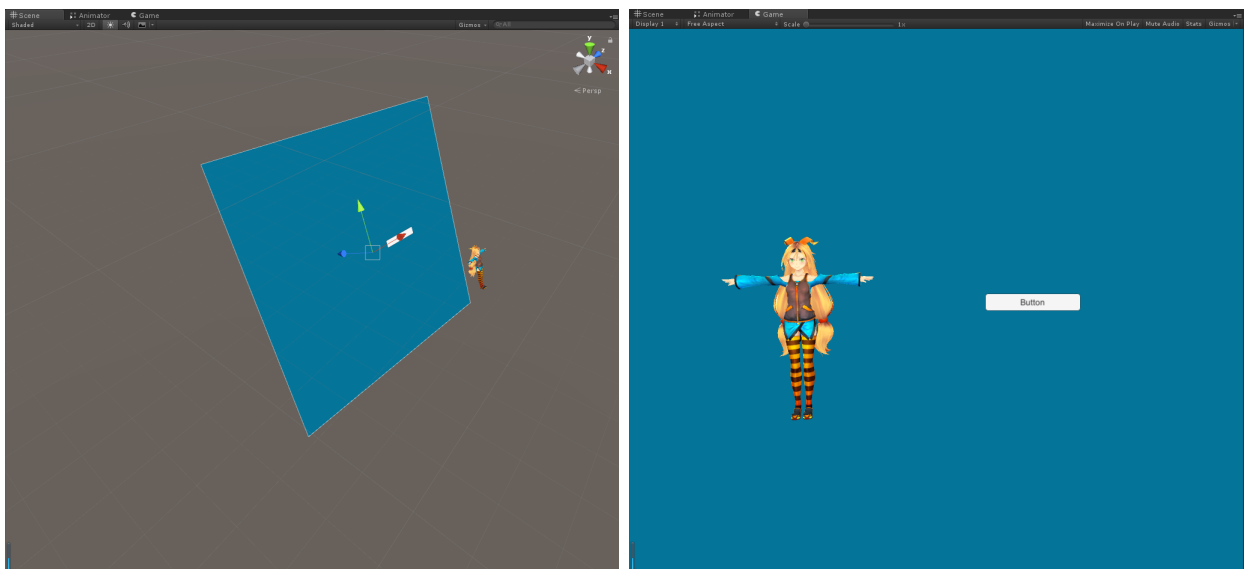


图 10.1: 摄像机模式-画布



它比 Screen Space-Overlay 模式的画布多了下面几个参数：

1. **Render Camera:** 渲染摄像机
2. **Plane Distance:** 画布距离摄像机的距离
3. **Sorting Layer:** Sorting Layer 是 UGUI 专用的设置，用来指示画布的深度。可以通过点击该栏的选项，在下拉菜单中点击“Add Sorting Layer”按钮进入标签和层的设置界面，或者点击导航菜单->edit->Project Settings->Tags and Layers 进入该页面。
4. **Order in Layer:** 在相同的 Sort Layer 下的画布显示先后顺序。数字越高，显示的优先级也就越高。

### 10.2.3 World Space -世界空间模式

World Space 即世界空间模式。在此模式下，画布被视为与场景中其他普通游戏对象性质相同的类似于一张面片（Plane）的游戏物体。

画布的尺寸可以通过 **RectTransform** 设置，所有的 UI 元素可能位于普通 3D 物体的前面或者后面显示。当 UI 为场景的一部分时，可以使用这个模式。

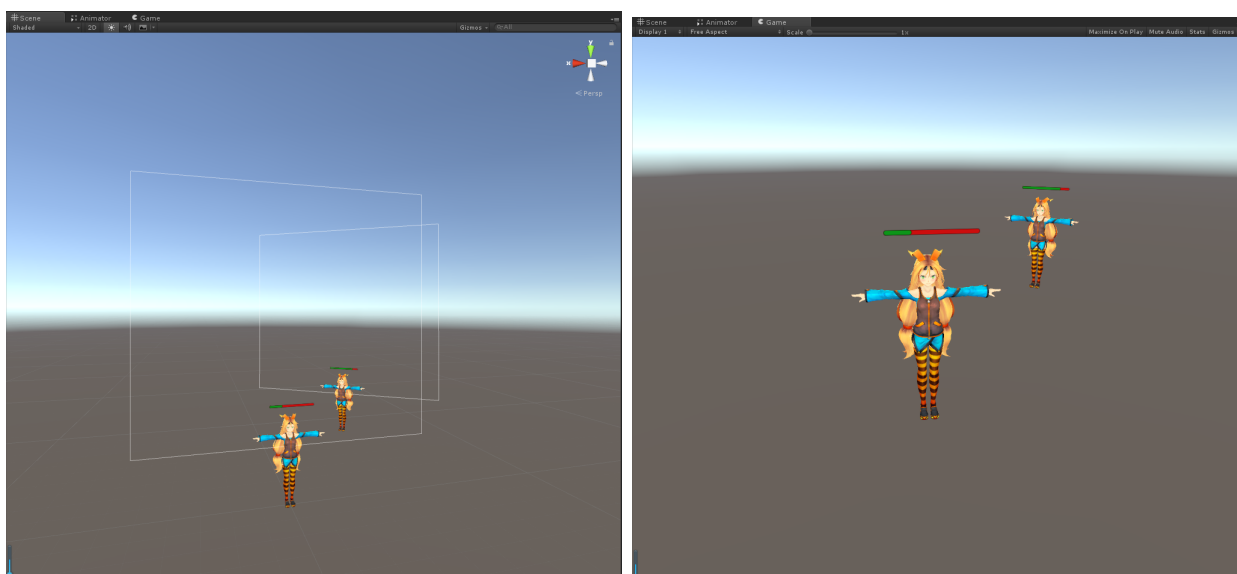


图 10.2: 世界空间模式- 画布

## 10.2.4 使用总结

表 10.1: 渲染模式使用场景说明

渲染模式	画布匹配屏幕?	摄像机?	像素对应	适应
覆盖-overlay 模式	是	不需要	可选	2D
摄像机-camera 模式	是	需要	可选	2D+3D
世界空间-world 模式	否	需要	不可选	3D

## 10.2.5 Canvas Scalar

<https://blog.csdn.net/qq168213001/article/details/49744899>

## 10.2.6 Layer

## 10.3 RectTransform

<https://blog.csdn.net/jk823394954/article/details/53861539>

<https://blog.csdn.net/rickshaozhiheng/article/details/51569073>

<https://blog.csdn.net/serenahaven/article/details/78826851>

核心看: [https://blog.csdn.net/Happy\\_zailing/article/details/78835482](https://blog.csdn.net/Happy_zailing/article/details/78835482)

<http://lib.csdn.net/article/unity3d/36875>

RectTransform 继承自 Transform, 又增加锚点、中心轴点等信息, 主要提供一个矩形的位置、尺寸、锚点和中心信息以及操作这些属性的方法, 同时提供多种基于父级 *RectTransform* 的缩放形式。

### 10.3.1 Pivot(中心)

Pivot 用来指示一个RectTransform (或者说是矩形) 的中 (重) 心点。

### 10.3.2 锚点- 自适应屏幕

<http://www.bubuko.com/infodetail-2384845.html>

锚点（四个）由两个Vector2 的向量确定，这两个向量确定两个点，归一化坐标分别是Min和Max，再由这两个点确定一个矩形，这个矩形的四个顶点就是锚点。

在Hierarchy 下新建一个 Image，查看其Inspector。

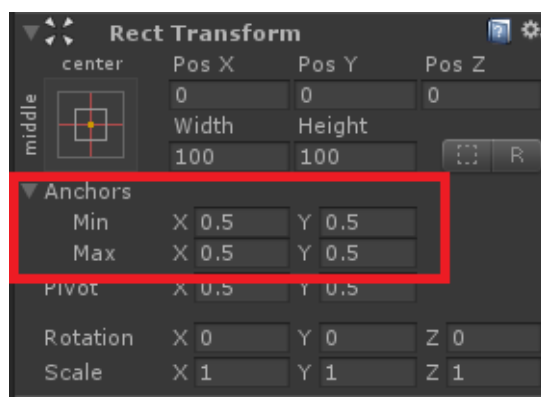


图 10.3: Anchor 属性

在 Min 的 x、y 值分别小于 Max 的 x、y 值时，Min 确定矩形左下角的归一化坐标，Max 确定矩形右上角的归一化坐标。

刚创建的 Image，其Anchor的默认值 为Min (0.5, 0.5) 和Max (0.5, 0.5)。也就是说，Min和Max 重合了，四个锚点合并成一点。锚点在 Scene 中的表示如下：

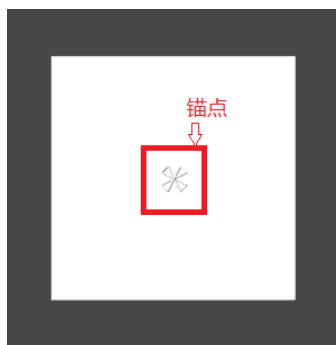


图 10.4: 锚点初始位置

将 Min 和 Max 的值分别改为 (0.4, 0.4) 和 (0.5, 0.5)。可以看见四个锚点已经分开了。

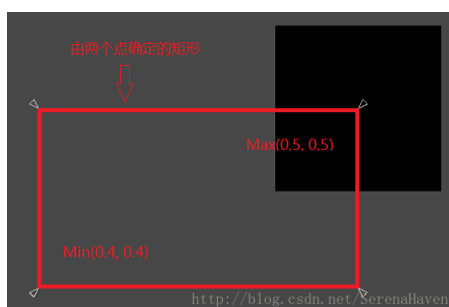


图 10.5: min Max 位置、确定矩形

**需要注意** 在不同的 Anchor 设置下，控制该 RectTransform 的变量是不同的。

比如设置成全部居中（默认）时，属性里包含熟悉的用来描述位置的PosX、PosY和PosZ，以及用来描述尺寸的Width和Height；

切换成全部拉伸时，属性就变成了Left、Top、Right、Bottom 和PosZ，前四个属性用来描述该 RectTransform 分别离父级各边的距离，PosZ 用来描述该 RectTransform 在世界空间的 Z 坐标

## 锚点类型

- 位置类型 左上角、中心等
- 拉伸类型 纵向拉伸适配、横向、整体

## 锚点在一块的时候

- Anchor 是打在父级窗体上的
- Anchor 的位置在父级窗体上的标记方式是按照百分比记录的，单位（百分比）
- Anchor 的Min(RectTransform.anchorMin) Max(RectTransform.anchorMax) 的信息保持一致
- 子物件的坐标系为纵向 Y, 横向 X, 并且以Anchor 为原点，自身坐标用中心轴点Pivot 表示
- 子物件的 Pivot 与 Anchor 位置始终保持不变，单位（像素）

## 锚点单向（横或者纵）分开的时候

- 分开的部分 (拉伸方向) 与父级窗体保持一致变化，单位（百分比）
- 与相对方向则绝对保持，单位（像素）

## 锚点双向分开的时候

- 双向都与父级窗体保持一致的变化，单位（百分比）
- 上-top、下-bottom、左、右边距绝对保持，单位（像素）

**anchorMax、anchorMin** `anchorMin.x` 表示锚点在x 轴的起始点位置，`anchorMax.x` 表示锚点在 x 轴的终点位置，取值0~1，表示**百分比值**，该值乘以父窗口的`width` 值就是实际锚点相对于父窗口 x 轴的位置。y 轴与 x 轴同理。

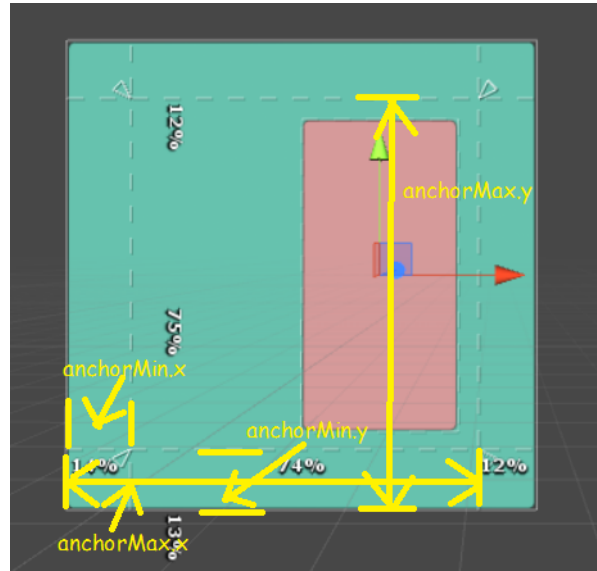


图 10.6: Anchor.Min 与 Anchor.Max

这个值确定了锚点相对于父窗口的位置，是**真正决定锚点位置的值**

## offsetMax 和 offsetMin 属性

**锚点分开时** 在锚点分开的状态下：锚点其实是四个钉子，分为左上，左下，右下及右上四个，每个空间在 UI 模型中都是一个矩形，也有左上，左下，右下及右上四个顶点，那么锚点的每个钉子可以关联一个点，即左上——左上；左下——左下；右下——右下；右上——右上。这样进行绑定。

`offsetMax` 是 `RectTransform` 右上角相对于右上 `Anchor` 的距离；

`offsetMin` 是 `RectTransform` 左下角相对于左下 `Anchor` 的距离。

`offset` 可以认为是以像素为单位。

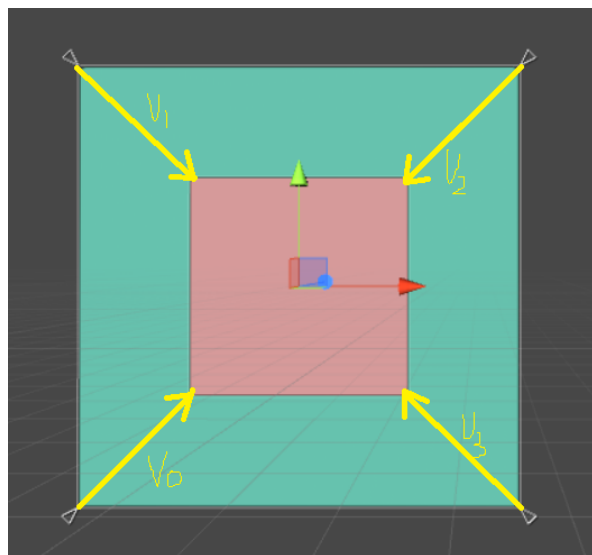


图 10.7: 锚点在一起时 Offset 求取向量示例

锚点在一处时 锚点 offset 计算如下:

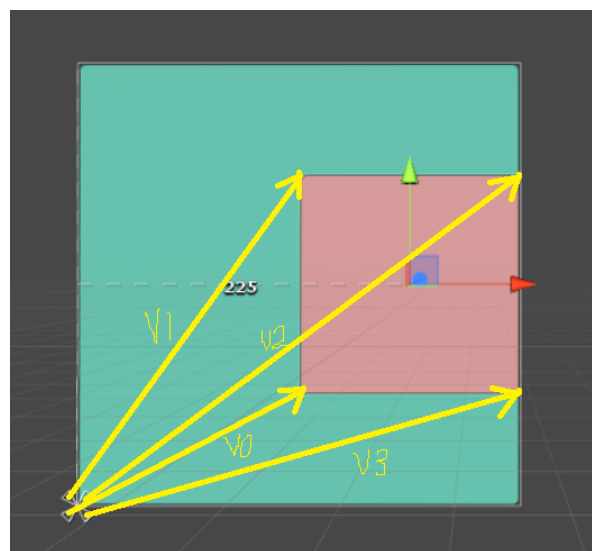


图 10.8: 锚点分开时 Offset 求取向量示例

**求取** 首先计算锚点的每个钉子到其对应的顶点矢量值，分别记作 $v_0$ ,  $v_1$ ,  $v_2$ ,  $v_3$ ，入上图。

然后比较四个向量的 $x$  值，将 $x$  的最大值赋给`offsetMax.x`，将 $x$  的最小值赋给`offsetMin.x`； $y$  的值同理。

## anchoredPosition

锚点在一处时 `anchorPosition` 就是从锚点到本物体的轴心（Pivot）的向量值。

锚点分开时

### 10.3.3 sizeDelta

sizeDelta 是  $\text{offsetMax} - \text{offsetMin}$  的结果。在锚点全部重合的情况下，它的值就是面板上的 (Width, Height)。

在锚点完全不重合的情况下，它是相对于父矩形的尺寸。

一个常见的错误是，当 RectTransform 的锚点并非全部重合时，使用 sizeDelta 作为这个 RectTransform 的尺寸。此时拿到的结果一般来说并非预期的结果。

### 10.3.4 RectTransform.rect

RectTransform.rect 的各值如图所示。

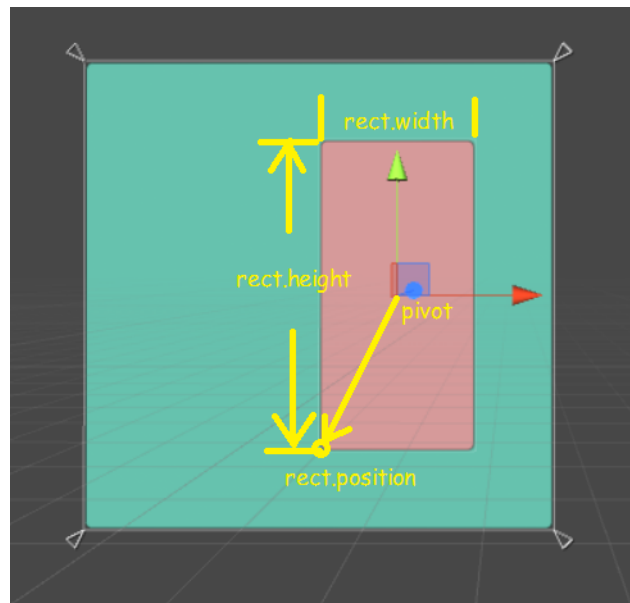


图 10.9: RectTransform rect 属性

### 10.3.5 示例

```
GameObject webText = new GameObject("webText");
webText.AddComponent<UnityEngine.UI.Text>();
webText.GetComponent<UnityEngine.UI.Text>().text = "";
webText.GetComponent<RectTransform>().anchorMin = new Vector2(0, 0);
webText.GetComponent<RectTransform>().anchorMax = new Vector2(1, 1);
webText.GetComponent<RectTransform>().sizeDelta = new Vector2(0, 0);
```

```
webText.GetComponent<RectTransform>().anchoredPosition = new Vector2(0, 0);  
webText.transform.localPosition = new Vector3(0,0,0);  
webText.transform.SetParent(webObj.transform, false);
```

### 10.3.6 FramDebug

查看渲染的先后顺序

windows->FrameDebug

## 10.4 按钮

### 10.4.1 原始 Button

### 10.4.2 Image 等 -添加 button 组件

- create -> UI -> Image
- Inspirit -> Add Component -> button

### 10.4.3 添加事件处理脚本

- 书写脚本并添加到 Button gameObject 上
- 如果是 Button 组件的话直接在 button 组件上添加，如果是 Image 则添加 button 组件后再添加
- 添加脚本对象到onClick() 部分：+ -> gameObject 拖进来 -> 选择脚本中的具体函数

## 10.5 文本- Text

### 10.5.1 添加文字阴影 -shadow 组件

addComponent -> shadow



## 10.5.2 添加文字边框 -outline 组件

`addComponent -> outline`

## 10.6 图片- ImageView

## 10.7 选中标记- Toggle

Toggle 基本

Toggle Group

选项栏设定 将 panel 拖入 toggle 中的value changed 部分

预设 确定默认打开哪个 panel，然后将其IsOn 勾选，其余取消勾选

## 10.8 滚动区域、滚动条

## 10.9 其他工具条

## 10.10 布局- Layout

- 具体页面下创建空物体 `GameObject`
- 其次在`GameObject` 下添加组件 -> `grid layout group`
- 最后在这个`GameObject` 下创建出各种 Image 组件，然后这些组件将会以`grid layout` 的布局进行自动调整

### 10.10.1 grid layout group

- 调整`cell size` 进行调整子物件的大小
- `cell size` 的改变只影响子组件的第一层，既最下面一层

**10.10.2 horizontal layout group**

**10.10.3 vertical layout group**

# 第十一章 物理

## 11.1 流程

- Rigidbody : 创建，以完成受力接收。
- Physical Material: 创建，以完成多种力的添加。
- Material : 拖入材质球。

## 11.2 刚体

## 11.3 碰撞器

## 11.4 物理材质

## 11.5 触发器

## 11.6 射线

## 11.7 关节



## 第十二章 动画

### 12.1 流程

- └ 在要动态显示的物体的父节点上创建 **Animation**
  - └ 编辑 **Animation**
    - └ 在脚本中获取该父亲节点上的 **Animator** 组件
    - └ 播放动画



## 第十三章 粒子系统





## 第十四章 着色器渲染



# 第十五章 资源管理- AssetBundle

## 15.1 工作流程

AssetBundle 的使用流程

- └ 1- 创建 AssetBundle
  - └ 2- 上传到 Server
    - └ 3- 游戏运行时根据需要下载 AssetBundle 文件
      - └ 4- 解析加载 Assets
        - └ 5- 使用完后释放内存等资源

## 15.2 创建

通过编译管线 BuildPipeline 来创建 AssetBundle 文件，总共有三种方法，具体如下所示。

### 15.2.1 BuildAssetBundle

该 API 将编辑器中的任意类型的 Assets 打包成一个 AssetBundle，适用于对单个大规模场景的细分。

- 名称: **BuildPipeline.BuildAssetBundle**
- 参数-1: **mainAsset** : Object
- 参数-2: **assets** : Object[]
- 参数-3: **pathName** : string
- 参数-4: **options** : BuildAssetBundleOptions
- 参数-5: **targetPlatform** : BuildTarget = BuildTarget.WebPlayer
- 返回值: **bool**

## BuildAssetBundleOptions

**CompleteAssets**

**CollectDependencies**

**DisableWriteTypeTree**

**DeterministicAssetBundle**

**UncompressedAssetBundle**

**AssetBundle 之间的依赖** 如果游戏中的某个资源被多个资源引用（例如游戏中的 Material），单独创建 AssetBundle 会使多个 AssetBundle 都包含被引用的资源（这里跟 flash 编译选项中的链接选项有些像），从而导致资源变大，这里可以通过指定 AssetBundle 之间的依赖关系来减少最终 AssetBundle 文件的大小（把 AssetBundle 解耦）。

具体方法是在创建 AssetBundle 之前调用 BuildPipeline.PushAssetDependencies 和 BuildPipeline.PopAssetDependencies 来创建 AssetBundle 之间的依赖关系，它的用法就是一个栈，后压入栈中的元素依赖栈内的元素。

### 15.2.2 BuildStreamedSceneAssetBundle

该 API 将一个或多个场景中的资源及其所有依赖以流加载的方式打包成 AssetBundle，一般适用于多单个或多个场景进行集中打包

- 名称: **BuildPipeline.BuildStreamedSceneAssetBundle**
- 参数-1: **level** : string[]
- 参数-2: **locationPath** : string
- 参数-3: **target** : BuildTarget
- 返回值: **String**

### 15.2.3 BuildAssetBundleExplicitAssetNames

该 API 功能与 `a` 相同，但创建的时候可以为每个 Object 指定一个自定义的名字。（一般不太常用）

- 名称: `BuildPipeline.BuildAssetBundleExplicitAssetNames`
- 参数-1: `assets : Object[]`
- 参数-2: `assetNames : string[]`
- 参数-3: `pathName : string`
- 参数-4: `options : BuildAssetBundleOptions`
- 参数-5: `targetPlatform : BuildTarget`
- 返回值: `bool`

## 15.3 使用

## 15.4 卸载

## 15.5 内存模型

## 15.6 其他



## 第十六章 Editor 扩展

Unity 编辑器扩展是扩展 Unity 菜单功能，也可以说是自定义 Unity 菜单，以此来便利我们能够更快捷地开发游戏。

### 16.1 流程

1. 在 Asset 文件夹下创建一个文件夹Editor，如果已经存在则忽略此步
2. 在该Editor 文件夹下，创建一个C# 脚本，无需继承于任何类
3. 引用UnityEditor 命名空间
4. 写静态方法既菜单的功能,类似于[SerializedFiled] 下的东西,与[MenuItem("Netease/xx")] 一一对应。
5. 给该方法上添加[MenuItem("")] 特性

### 16.2 在编辑器上增加一个 MenuItem

### 16.3 创建一个对话框

### 16.4 扩展 Inspector 面板

### 16.5 编辑器插件常用函数

#### 16.5.1 资源导入回调函数

当导入资源到 Unity 项目中的某个资源文件夹下时，当 Unity Editor 获得焦点后，会在加载完资源后，先为其创建.meta 文件，然后再触发该回调函数。

```

public class TestBundleNameAndTexture : UnityEditor.AssetPostprocessor
{
    static void OnPostprocessAllAssets( // 这个函数必须为静态的，其他可以不是！
        string[] importedAssets,
        string[] deletedAssets,
        string[] movedAssets,
        string[] movedFromAssetPaths)
    {
        foreach (var path in importedAssets)
        {
            DirectoryInfo dir = new DirectoryInfo(path);
            Debug.Log(dir.Parent.FullName);
            BundleNameCreator.Proc(dir.Parent.FullName);
        }
    }
}

```

## 16.6 一些常用的 Inspector 属性设置

## 16.7 参考

[https://blog.csdn.net/puppet\\_master/article/details/51012298](https://blog.csdn.net/puppet_master/article/details/51012298)

<http://blog.csdn.net/asd237241291/article/details/38235091>

[http://blog.sina.com.cn/s/blog\\_471132920101n8cr.html](http://blog.sina.com.cn/s/blog_471132920101n8cr.html)



# 第十七章 跨平台发布 apk

## 17.1 流程

- 安装 JavaSDK、Android Studio 并在 SDK manager 里添加对应的 API 包
- 在 unity 中的edit 选项下的preferences, 并选中External Tools 选项,配置JDK 和Android SDK 安装位置。
- 在 unity 中的File -> Build Settings 中, 添加需要添加的场景, 并选择对应的平台 (Android, IOS) 等
- 在 unity 中的Build Settings 中的Player Settings 设置以下几个重要内容。
  1. Company Name
  2. Product Name
  3. Default Icon :192×192
  4. Default Orientation
  5. Other Settings -> Identification : 修改为com.netease(Or Other).TestName(Or Other)

## 17.2 Apk 安装常见错误

[http://mumu.163.com/2017/03/30/25905\\_680657.html](http://mumu.163.com/2017/03/30/25905_680657.html)



## 第十八章 调试技巧

### 18.1 以父类为基点

在 Inspector 中查看是否存在父类脚本 [SerializedField] 的变量，这样方便对空间进行查找，并且添加新的控制