# U3D 笔记

郑华

2019 年 11 月 28 日

# 目录

# 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 hiearch 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- [SerializedField]

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;
}
```

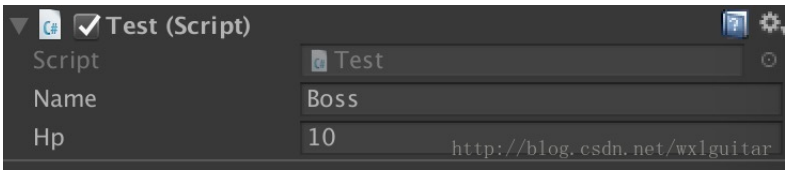


图 1.1: 序列化操作 -在 Inspector 上显示

## 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

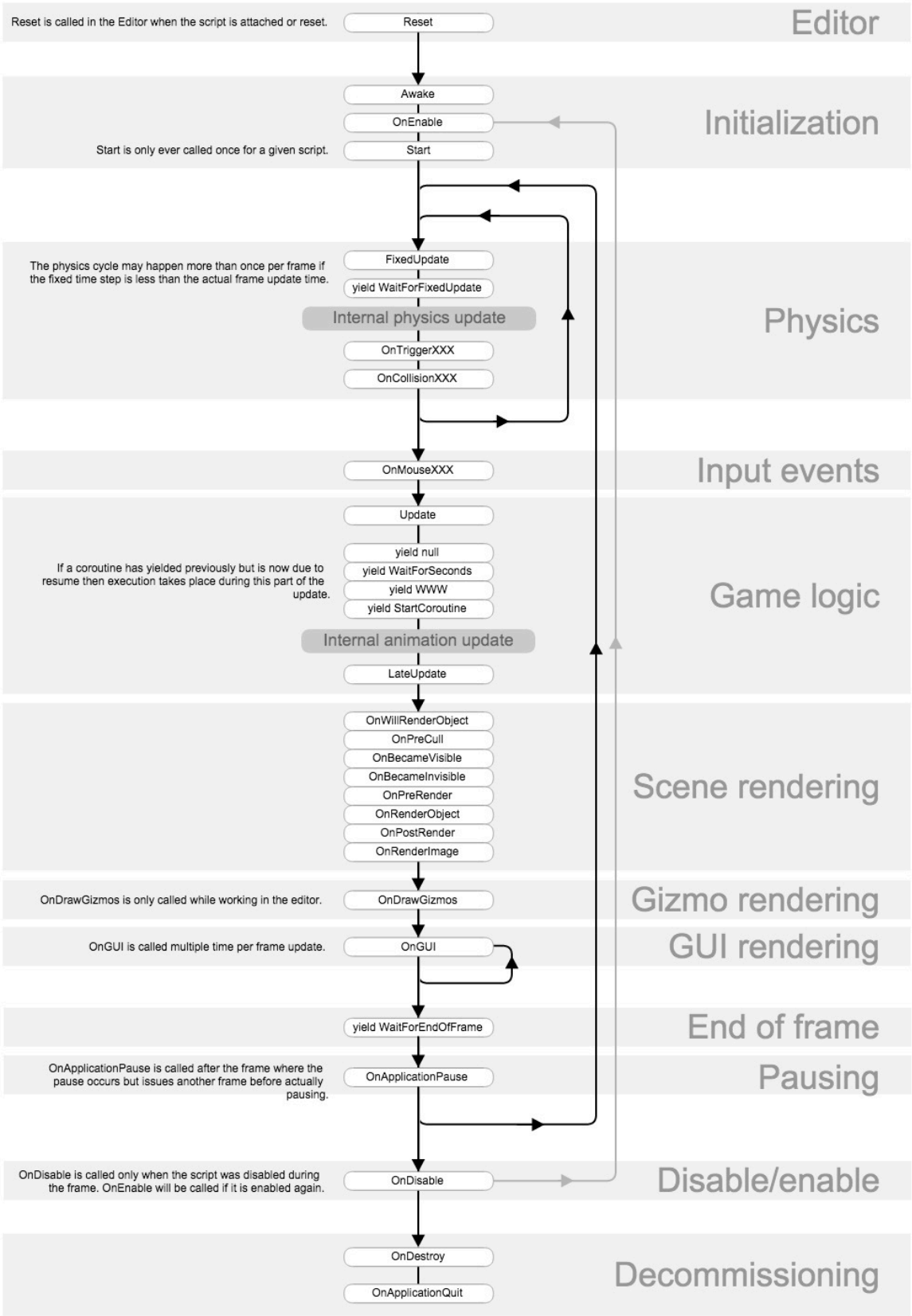


图 1.2: 脚本生命周期核心方法

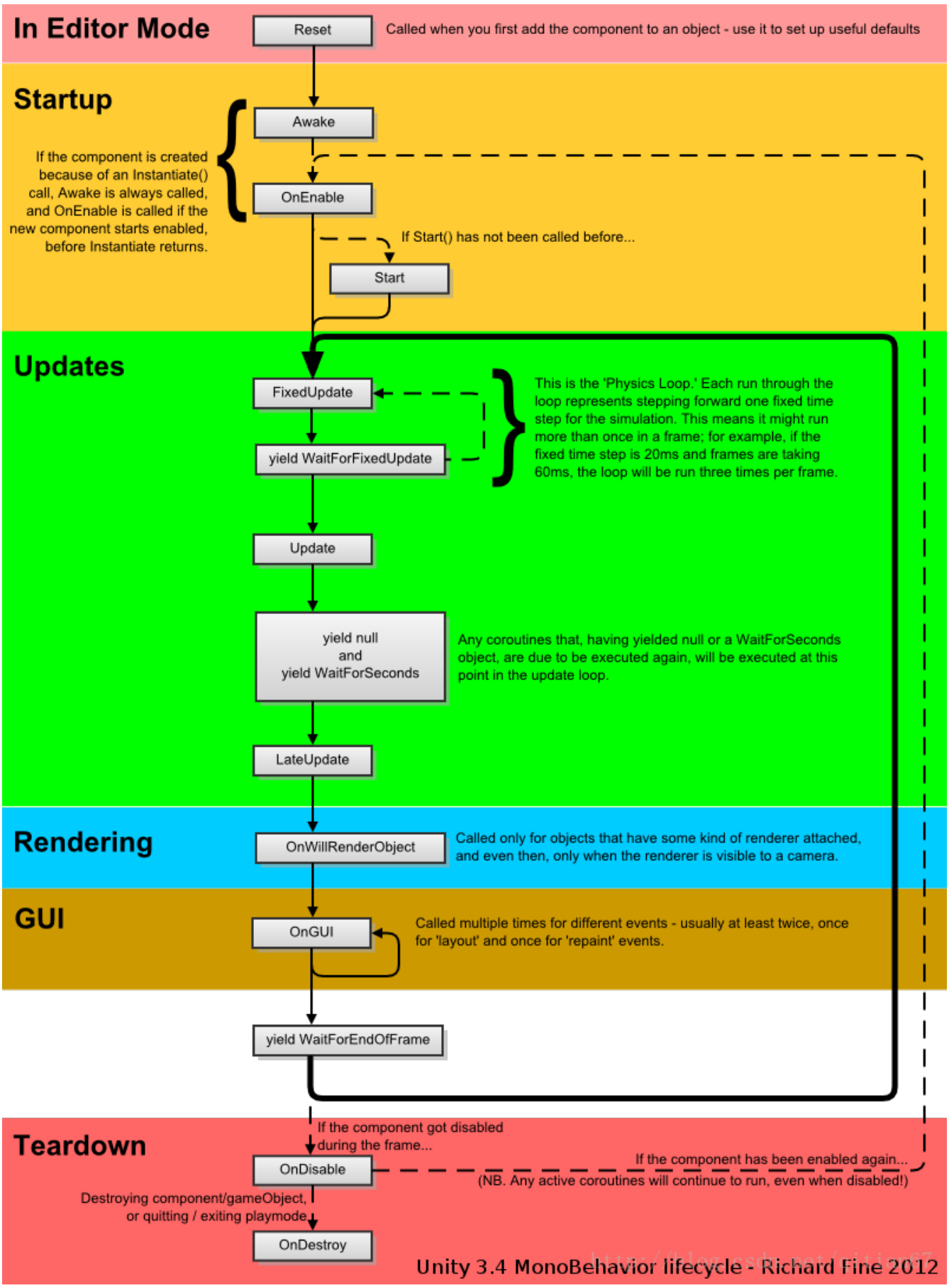
In Editor Mode
Reset
Called when you first add the component to an object - use it to set up useful defaults
Startup
Awake
If the component is created because of an Instantiate() call, Awake is always called, and OnEnable is called if the new component starts enabled, before Instantiate returns.
OnEnable
If Start() has not been called before...
Start
Updates
FixedUpdate
This is the 'Physics Loop.' Each run through the loop represents stepping forward one fixed time step for the simulation. This means it might run more than once in a frame; for example, if the fixed time step is 20ms and frames are taking 60ms, the loop will be run three times per frame.
yield WaitForFixedUpdate
Update
yield null
and
yield WaitForSeconds
Any coroutines that, having yielded null or a WaitForSeconds object, are due to be executed again, will be executed at this point in the update loop.
LateUpdate
Rendering
OnWillRenderObject
Called only for objects that have some kind of renderer attached, and even then, only when the renderer is visible to a camera.
GUI
OnGUI
Called multiple times for different events - usually at least twice, once for 'layout' and once for 'repaint' events.
yield WaitForEndOfFrame
Teardown
If the component got disabled during the frame...
OnDisable
If the component has been enabled again...
(NB. Any active coroutines will continue to run, even when disabled!)
Destroying component/gameObject, or quitting / exiting playmode
OnDestroy
Unity 3.4 MonoBehavior lifecycle - Richard Fine 2012

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。

**如何让已经存在的 GameObject 在 LoadScene() 后不被卸载掉**

```
void Awake()
{
    DontDestroyOnLoad(transform.gameObject);
}
```

## 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

https://blog.csdn.net/coffeecato/article/details/52153485

原理剖析：https://blog.csdn.net/qiudesuo/article/details/84551180

https://blog.csdn.net/tom_221x/article/details/78546037

之前一定得理解 IEnumerable, IEnumerator. 参考原理剖析第一部分。

### 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。

**将一段程序在多帧运行。**

**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**
- `IEnumrator` 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`IEnumrator`, 返回值使用（*yield retuen* + 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

Unity 的协程实在**每帧结束之后**，处理gameObject 中的协程，去检测yield的条件是否满足.

当一个协程启动时，本质创建迭代器对象，调用MoveNext() 方法，执行到 yield 暂时挂起退出，待满足条件后再次调用MoveNext() 执行后续代码，直至遇到下一个 yield 为止，如此循环至函数结束。

**那么 yield return 可以这样理解：**yield return 是“*停止执行方法，并且在下一帧从这里重新开始*”

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。

**总结** Unity 的协程实在**每帧结束之后**,处理gameObject 中的协程, 这个时候执行 MoveNext() 判断是否能够从当前的位置继续往下执行。

其实，这个MoveNext() 的作用就是判断epoll 中的等待事件有没有完成，可不可以执行下面的代码，如果可以执行，则执行到下一个需要等待的地方。

函数内有多少个 yield return 在对应的 MoveNext() 就会返回多少次 true （不包含嵌套）。另外非常重要的一点的是：同一个函数内的其他代码（不是 yield return 语句）会被移到 MoveNext 中去，也就是说，每次 MoveNext 都会顺带执行同一个函数中 yield return 之前的代码。

对于 Unity 引擎的 YieldInstruction 实现，其实就可以看着一个函数体，这个函数体每帧会实现去 check MoveNext 是否返回 false 。

```
yield retrun new WaitForSeconds(2f);

//上面这行代码的伪代码实现:

private float elapsedTime;
private float time;
private void MoveNext()
```

```
{
        elapesedTime += Time.deltaTime;

        if(time <= elapsedTime)
                return false;
        else return true;
}
```

**委托 + 协程** https://blog.csdn.net/qq992817263/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，自动会**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　实体之间的关系

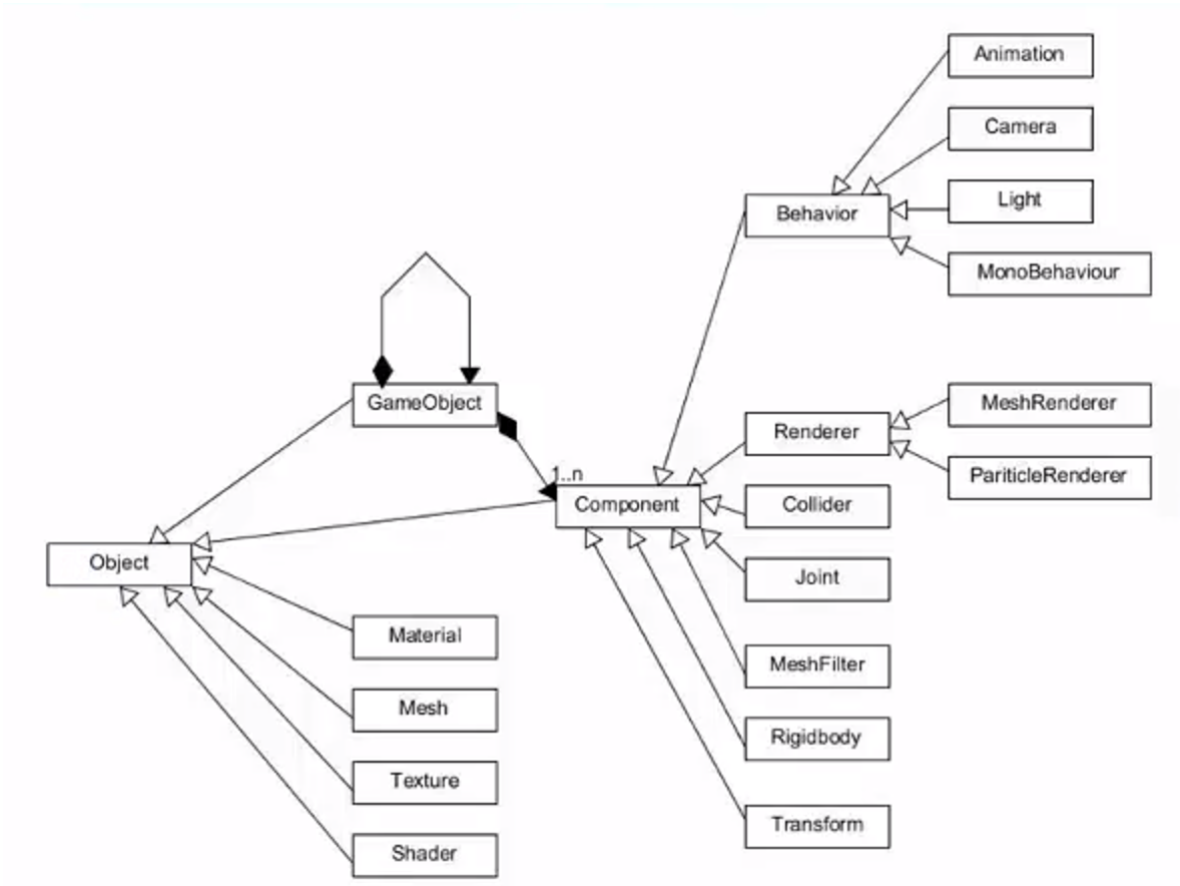


图 3.1: Unity 类之间的关系

## 3.2　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(tru

**游戏中获取或查找** 参考：https://blog.csdn.net/u010145745/article/details/39160141

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>
   ```

3. 返回一个用 tag 做标识的活动的游戏物体的列表，如果没有找到则为 null:

   ```
   static GameObject[] FindGameObjectsWithTag (string tag);
   ```

4. 遍历场景中所有物体, 获取泛型 T 类型物体,FindObjectsOfType(typeof(Type)) 返回Type 类型的所有激活的加载的物体列表，它将返回任何资源（网格，纹理，预设，...）**或未激活的物体**

   ```
   object[] gameObjects = GameObject.FindSceneObjectsOfType(typeof(Transform));
   ```

## 3.3 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。

### 3.3.1　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

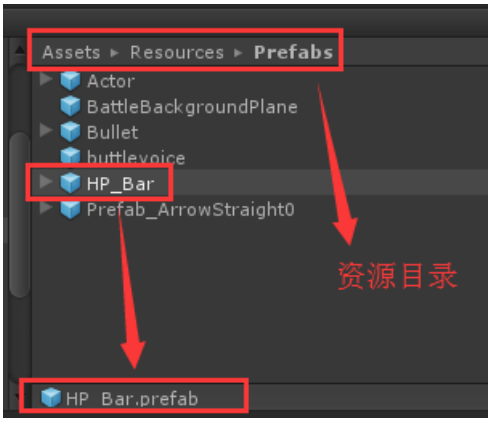


图 3.2: Prefab 资源位置

```
//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场的场景最外层，如下图所示。


图 3.3: 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;
```

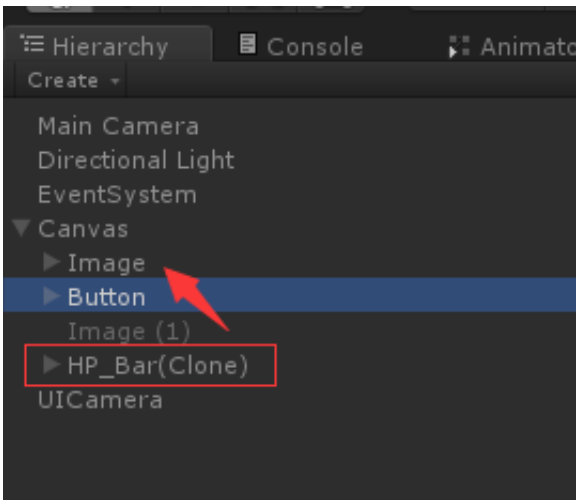


图 3.4: Prefab 对象设置父子关系

**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。

## 3.4 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**

## 3.5 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.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
- transform.Position：是**世界坐标中**的位置，可以理解为绝对坐标

### 4.1.2　旋转

- transform.localEulerAngles 以自身坐标系为参考，而不是世界坐标系

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

- transform.rotation 以自身坐标系为参考，而不是世界坐标系

```
Quaternion rotation= Quaternion.Euler(new Vector3(x, y,z));
Transform.rotation=rotation;
```

- transform.Rotate(x,y,z)

```
// 以自身坐标系为参考，而不是世界坐标系，分别以x度y度z度绕X轴、Y轴、Z轴匀速旋转
transform.Rotate(x,y,z);
// 以自身坐标系为参考
transform.Rotate(轴, Space.Self);
// 以世界坐标系为参考
transform.Rotate(轴, Space.World);
```

### 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 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 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将按每种材质一个通道的方式来渲染。

`Defferred`: **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。

### 4.2.10 提高性问题

- 移动摄像机的动作放在哪个系统函数中, 为什么 (LateUpdate)

## 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

## 三、继承 IPointerDrag 等

# 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 |
| --- | --- |
| time | 单位为秒 |
| `deltaTime` | 从上一帧到当前帧消耗的时间 |
| fixedTime | 最近 FixedUpdate 的时间，从游戏开始计算 |
| `fixedDeltaTime` | 物理引擎和 FixedUpdate 的更新时间间隔 |
| timeSceneLevelLoad | 从当前 Scene 开始到目前为止的时间 |
| realTimeSinceStartup | 程序已经运行的时间 |
| `frameCount` | 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

## 7.3　坐标系

左右手坐标系：http://www.cnblogs.com/mythou/p/3327046.html

## 7.4　向量计算

## 7.5　矩阵计算

# 第八章　光照

## 8.1　光照

## 8.2　烘培

**简介**　只有静态场景才能完成烘培（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。

- **如果不烘培：**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 -> Text`
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：

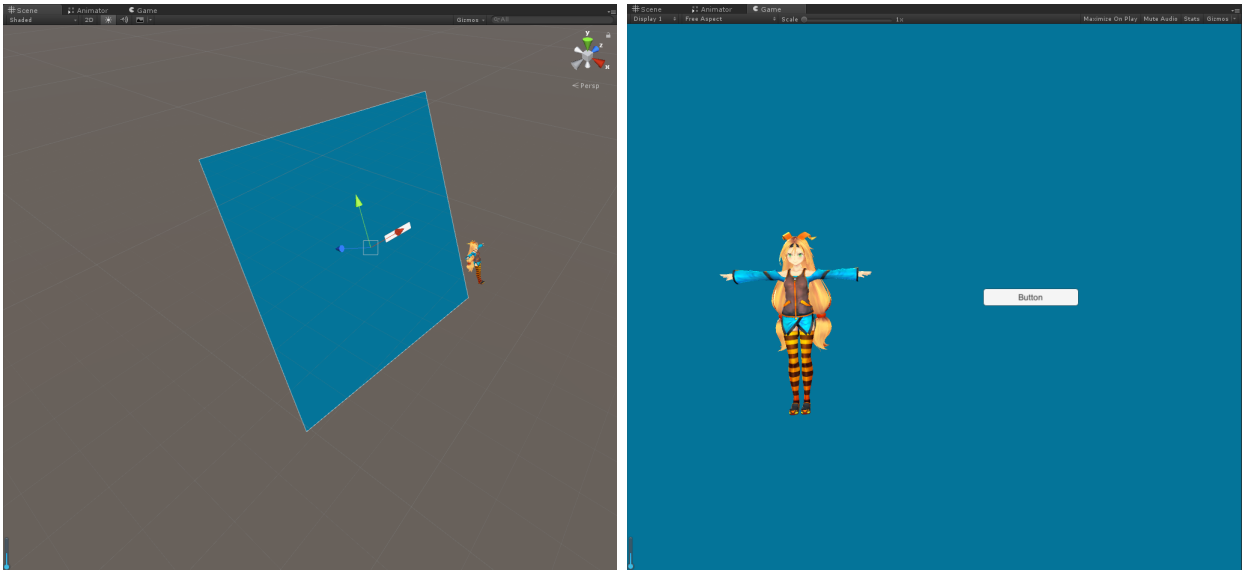


图 10.1: 摄像机模式-画布

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。

depth 参考世界空间相关一章，camera Depth.

### 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**。

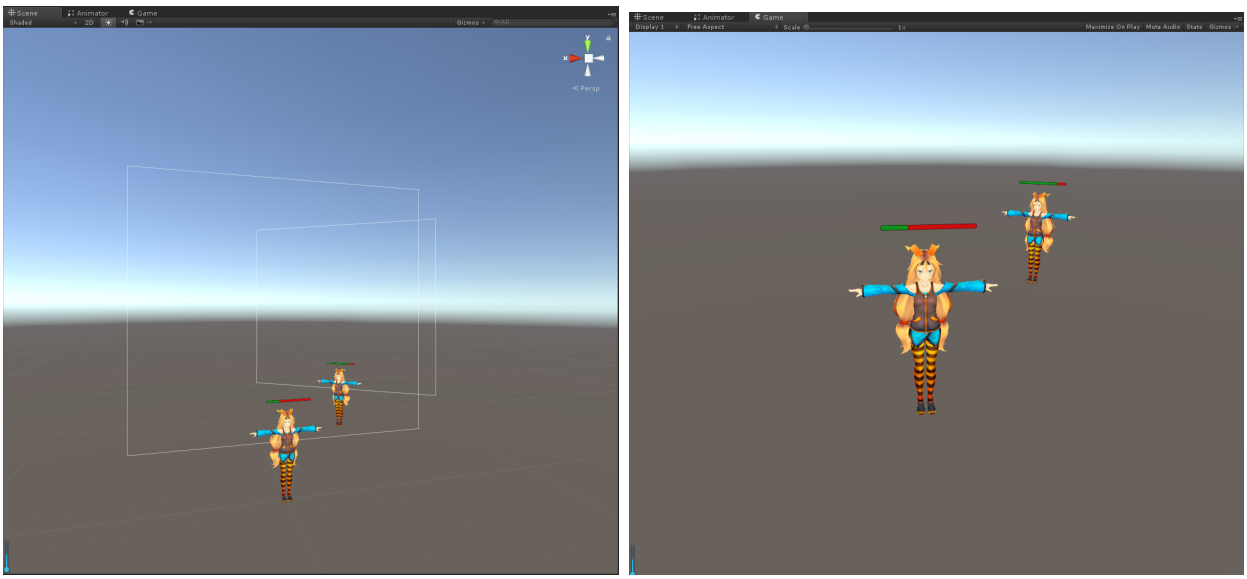

图 10.2: 世界空间模式- 画布

### 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

**"The Canvas Scaler component is used for controlling the overall scale and pixel density of UI elements in the Canvas. This scaling affects everything under the Canvas, including font sizes and image borders."**

用于画布上 UI 元素的整体缩放比例和像素密度。该缩放比例会影响画布中的所有东西，包括字体大小和图像边界。

为了适应不同的分辨率，我们可能会**允许适当的 UI 整体性的缩放，外加一些尽可能少的布局微调**。这样就能达到一个比较理想的效果。Unity 中 Canvas Scale 就是负责该功能的组件。

Canvas Scaler 的 ui scale mode 有三种值（constan pixel size、scale with screen size 和 constant physical size）

表 10.2: 简介

| UI 缩放模式 | 决定画布中的 UI 元素如何缩放 |
|---|---|
| Constant Pixel Size | 不管屏幕尺寸如何变化，UI 元素保持相同的像素大小 |
| Scale With Screen Size | 屏幕越大，UI 元素也越大 |
| Constant Physical Size | 不论屏幕大小和分辨率如何变化，UI 元素保持相同的物理尺寸 |

- **Constant Pixel Size** 这个模式一般用于需要整体缩放画布的对象（通过 Scale Factor）

  1. Scale Factor: 画布的缩放比例。默认况下为 1，表示正常大小。
  2. Reference Pixels Per Unit: 每单位代表的像素量

- **Constant Physical Size** 和前面这个类似也可以缩放 (通过 RectTransform)
- **Scale With Screen Size** 这个模式比较常用是我们开发自适应一般用的模式

  1. Reference Resolution(参考分辨率) ：参照这个 UI 布局所依据的分辨率，如果屏幕分

辨率更大，那么 UI 会变大，如果屏幕分辨率更小，那么 UI 会变小。

2. Screen Match Mode(屏幕匹配模式) ：Match Width or Height、Expand、Shrink

3. Reference Pixels Per Unit 每单位的参考像素

https://blog.csdn.net/gz_huangzl/article/details/52484611

表 10.3: Scale With Screen Size

| 属性 | 功能 |
| --- | --- |
| 参考分辨率 | 再设计 UI 布局时设置的分辨率，如果屏幕分辨率更大，UI 元素也会放大，反之亦然 |
| 屏幕匹配模式 | 当前分辨率下的宽高比不适合参考分辨率时使用的一种用于缩放画布的模式 |
| 匹配宽度或高度 | 用宽度或者高度又或者两者之间做参考来缩放画布 |
| 扩展 | 水平或垂直扩展画布，所以画布的尺寸不会比参考分辨率小 |
| 收缩 | 水平或垂直裁剪画布，所以画布的尺寸不会比参考分辨率大 |
| 匹配 | 决定缩放时，使用宽度/高度/两者之间用作参考 |
| 参考的每单位像素 | 如果一个 sprite 有“每单位像素数”这个设置，那么 spirit 中的一个像素会覆盖 UI 中的一个单元 |

### 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

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。

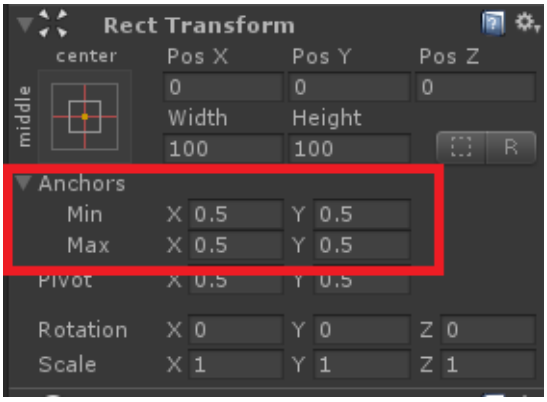


图 10.3: Anchor 属性

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：

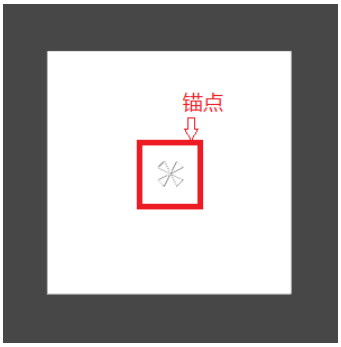


图 10.4: 锚点初始位置

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。

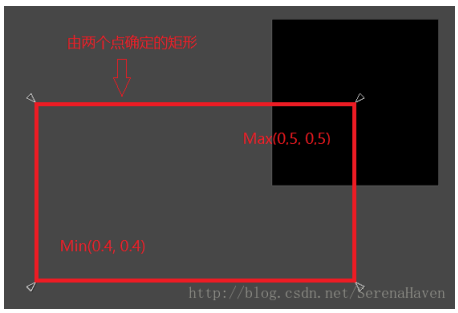


图 10.5: min Max 位置、确定矩形

**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。

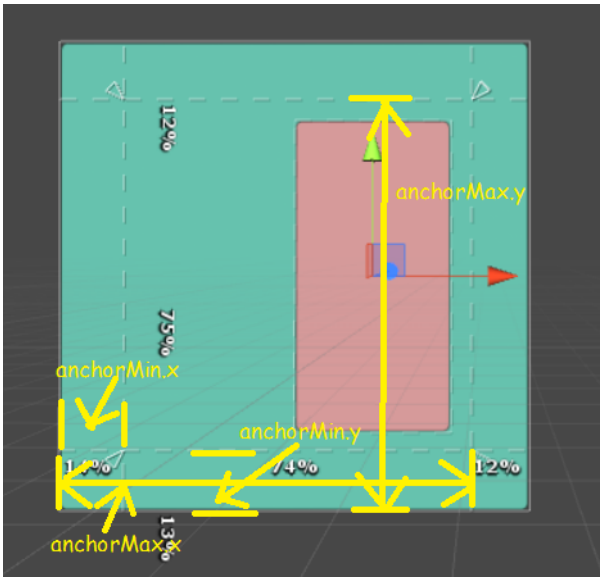


图 10.6: Anchor.Min 与 Anchor.Max

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。

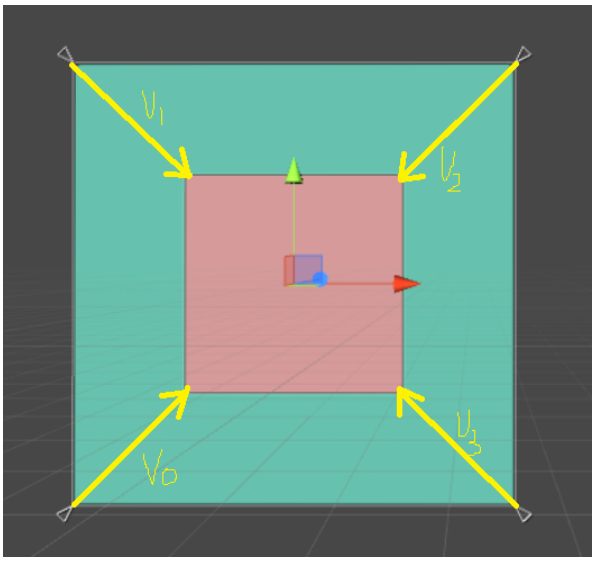


图 10.7: 锚点在一起时 Offset 求取向量示例

**锚点在一处时** 锚点 offset 计算如下：

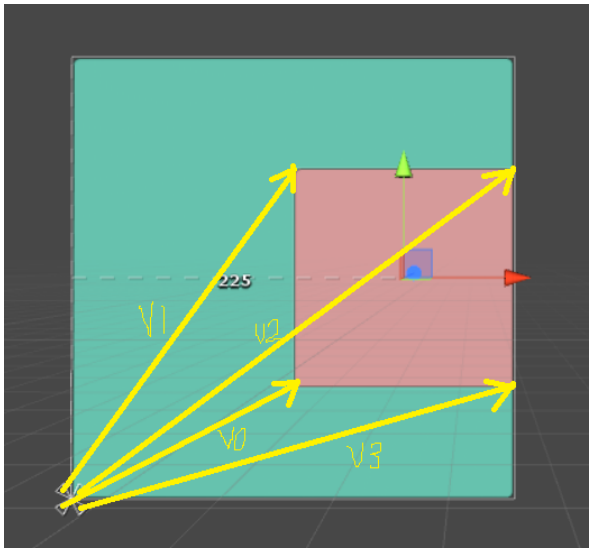


图 10.8: 锚点分开时 Offset 求取向量示例

**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`v0`，`v1`，`v2`，`v3`，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`offsetMax-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。

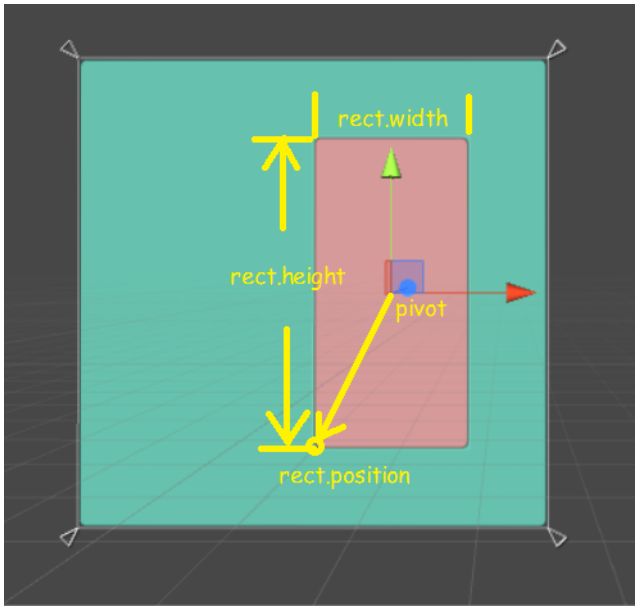


图 10.9: RectTransform rect 属性

### 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 RayCast Target-点击事件的获取原理

参考文献：https://blog.csdn.net/liujunjie612/article/details/55097789

UGUI 会遍历屏幕中所有 RaycastTarget 是 true 的 UI，接着就会发射线，并且排序找到玩家最先触发的那个 UI，在抛出事件给逻辑层去响应。

团队多人在开发游戏界面，很多时候都是复制黏贴，比如上一个图片是**需要响应** Raycast-Target，然后 ctrl+d 以后复制出来的也就带了这个属性，很可能新复制出来的图片是**不需要响应的**，开发人员**又没有取消勾选掉**，这就出问题了。

所以 RaycastTarget 如果被勾选的过多的话，效率必然会低。

### 10.4.2 原始 Button

### 10.4.3 Image 等 -添加 button 组件

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

### 10.4.4 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 添加文子边框 -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

## 10.11 提高性问题

- 为什么 UI 摄像机和场景摄像机能协同工作
- 怎么防止 UI 控件被点穿（如何过滤掉点击事件）
- UI 功能模块之间相互通信有什么好看法。（或者问成 Broadcast 和 sendMes 的看法）
- 屏幕适配有什么好主意

# 第十一章　物理

## 11.1　具体相关类

### 11.1.1　Rigidbody

刚体，套上这个组件的**物体会受到物理作用里的影响** (包括重力，推力，作用力等一切物理力)，如果没有这个组件物体也会静止不动.

**但必须有碰撞检测组件才能保证物体不被穿过**。值得注意的一点是，Is Kinematic 被勾选上表示物体不接受物理力的作用，只能通过 transform 的设置改变位置。

### 11.1.2　Collider

碰撞检测器，就是**检测一切范围内的触碰事件**，注意也就是说 rigidbody 的物理碰撞需要 collider 来检测才能发生，也就是物体的运动。

在**所有 Collider 上**有一个`Is Trigger` 的`bool` 型参数。当发生碰撞反应的时候，**会先检查此属性**。

`Is Trigger` 好比是一个物理功能的开关，**是要**“物理功能”**还是要**“OnTrigger 脚本”。

- 碰撞器有碰撞的效果，`IsTrigger=false`，可以调用`OnCollisionEnter/Stay/Exit` 函数
- 触发器没有碰撞效果，`IsTrigger=true`，可以调用`OnTriggerEnter/Stay/Exit` 函数

## 11.2　碰撞器、触发器

在 Unity 中参与碰撞的物体分 2 大块：1. 发起碰撞的物体。2. 接收碰撞的物体。

**发起碰撞的物体**　Rigodbody , CharacterController(其中的 RigidBody)

**接收碰撞的物体**　所有的 Collider

工作的原理为：发生碰撞的物体中必须要有“**发起碰撞**”的物体。否则，碰撞不响应。

判断碰撞就是需要计算力，无论是阻力也好，动力也好，**如果此时物体有刚体组件，那么物体就会在力的作用下运动**。如果这个物体没有刚体，那么碰撞产生的力就没有任何意义了，那计算碰撞也就没有任何意义了。

**所以，如果两个碰撞体都没有刚体组件，那么这两个物体即使相互发生了碰撞，那么也不会有碰撞事件的**。

**物体发生碰撞的必要条件**　两个物体**都必须**带有碰撞器(Collider),**其中一个还必须**带有Rigidbody。

**CharacterController 和 Rigidbody 的区别**

- CharacterController 自带胶囊碰撞器，里面包含有刚体的属性
- Rigidbody 就是刚体，使物体带有刚体的特征

**碰撞过程**

1. OnCollisionEnter
2. OnCollisionStay
3. OnCollisionExit

**施加力的方式**　都在 rigidbody 系列函数中。

- rigidbody.AddForce()
- rigidbody.AddForceAtPosition()

## 11.3 物理材质

## 11.4 射线

**射线 Ray**　`Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);`

可以从摄像机发出一条射线到 MousePosition。

**射线检测 Physics.Raycast()**

- static bool Raycast(Ray ray, RaycastHit hitInfo, float distance, int layerMask);
- Raycast(Ray ray, float distance, int layerMask);
- Raycast(Vector3 origin, Vector3 direction, float distance, int layerMask);
- Raycast(Vector3 origin, Vector3 direction, RaycastHit , float distance ， int layerMask );

`Ray ray` 是射线;`RaycastHit hitInfo` 是碰撞信息;`float distance` 是碰撞距离;`int layerMask` 是碰撞的层.

**RaycastHit** 是输出参数，所以需要先声明一个作为参数提供。其具体包括如下信息：

- `point` 射线碰到碰撞器的碰撞点，在世界空间中
- `normal` 射线所碰到的表面的法线
- `distance` 从光线的原点到碰撞点的距离
- `collider` 碰撞到的碰撞器
- `transform` 碰到的物体的位置信息

```
Physics.Raycast(ray, out hit, 100f, targetMask)
```

一条**射线，射线的范围**是 100 米，只和 **target 层**发生碰撞，碰撞后得到碰撞体的信息，并返回一个布尔值。

## 11.5 关节

# 第十二章　动画

## 12.1　游戏动画有哪几种，以及其原理

### 12.1.1　关节动画

把角色分成若干独立部分，一个部分对应一个网格模型，部分的动画连接成一个整体的动画，角色比较灵活，Quake2 中使用这种动画。

### 12.1.2　单一网格模型动画

由一个完整的网格模型构成，在动画序列的关键帧里记录各个顶点的原位置及其改变量，然后插值运算实现动画效果，角色动画较真实。

### 12.1.3　骨骼动画

广泛应用的动画方式，集成了以上两个方式的优点，骨骼按角色特点组成一定的层次结构，有关节相连，可做相对运动，皮肤作为单一网格蒙在骨骼之外，决定角色的外观，皮肤网格每一个顶点都会受到骨骼的影响，从而实现完美的动画。

## 12.2　Avatar

创建 Avatar
└ 配置 Avatar
  └ 肌肉设定
    └ 动画重定位
      └ 反向动力学

https://www.cnblogs.com/rainmj/p/5451847.html

## 12.3 Animation

- 在要动态显示的物体的父节点上创建 **Animation**-(在 Animation 窗口创建、在 Parameter 下添加需要改变状态的组件，在关键帧修改其状态)
  - 编辑 Animation
    - 在脚本中获取该父亲节点上的 Animator 组件
      - 播放动画

## 12.4 Animator

### 打开动画状态机界面

1. 右键点击“Assets”栏空白处，同样可以打开菜单栏进行创建：Create->Animater Controller
2. 打开“Animator”视窗的方法是：点击菜单“Window->Animator”或者双击任意“Animator Controller”资源文件
3. 点击“Animator”视窗左上角的“Parameters”视窗右上角的“+”按钮，可以创建参数。

### 属性编辑

1. 右键点击其他状态，选择“Set As Default”改变默认状态（橙色）
2. “Speed”表示动画的播放熟读倍率（1 为正常倍率）
3. “Motion”指定状态引用的动画剪辑
4. “Foot IK”勾选后会减少或消除动画中的脚滑现象
5. “Mirror”可以将动画左右对调
6. “Transition”
   - Transactions，默认是空的，我们创建一个 Transaction，可以让我们从一个动画平滑至另一个动画
   - 创建“Transaction”的方法是右键点击起始状态选择“Make Transaction”。然后会有一个箭头附着在你的指针位置，点击另一个状态，完成创建
   - 然后选中“Transition”线段，在“Inspection”中进行编辑
   - 每个“Transition”都有“Solo”和“Mute”选项，适用于特定区域状态机的调试工具，

发布游戏时，不要勾选任意一个

- **触发条件添加“Condition”**
  - 在**左侧 Parameters** 中添加条件 (int、float、bool、Trigger)，比如 bool Run
  - 然后在 **Transition** 的 condition 下添加该变量
  - 最后在游戏脚本中通过 animtor.SetBool(”Run”, false); 来修改 condition 结果。

## 12.5 iTween 动画用法

# 第十三章　粒子系统

# 第十四章　资源管理

## 14.1　资源

Unity 必须通过导入将所支持的资源序列化，生成 AssetComponents 后，才能被 Unity 使用。

资源 (Asset) 是硬盘中的文件，存储在 Unity 工程的 Assets 文件夹内。有些资源的数据格式是 Unity 原声支持的，有些资源则需要转换为源生的数据格式后才能被使用。

资源与对象时一对多的关系。

想使用 Unity 不支持未经导入的资源，只能使用 IO Stream 或者 WWW 方法进行。

### 14.1.1　GUID 与 fileID

Unity 会为每个导入到 Assets 目录中的资源创建一个`meta`文件，文件中记录了`GUID`，**GUID 用来记录资源之间的引用关系**。还有`local ID`（本地 ID），**用于标识资源内部的资源**。

资源间 **的依赖关系**通过 `GUID` 来确定；

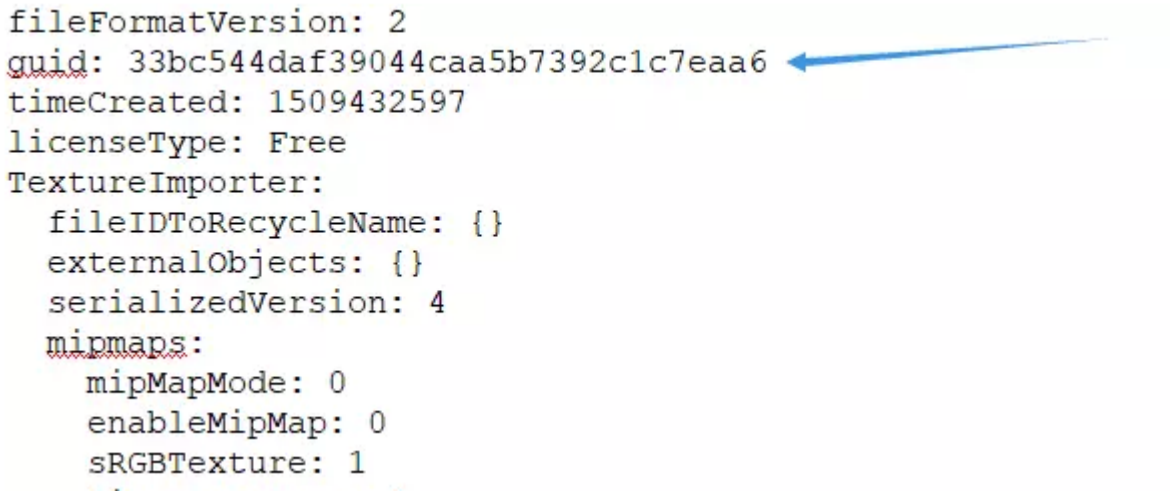


图 14.1: GUID 演示

如果一个资源引用了另一个外部资源，比如一个 Prefab 引用了其他脚本、纹理或 Prefab 等，则一定会标明引用资源文件的 File GUID

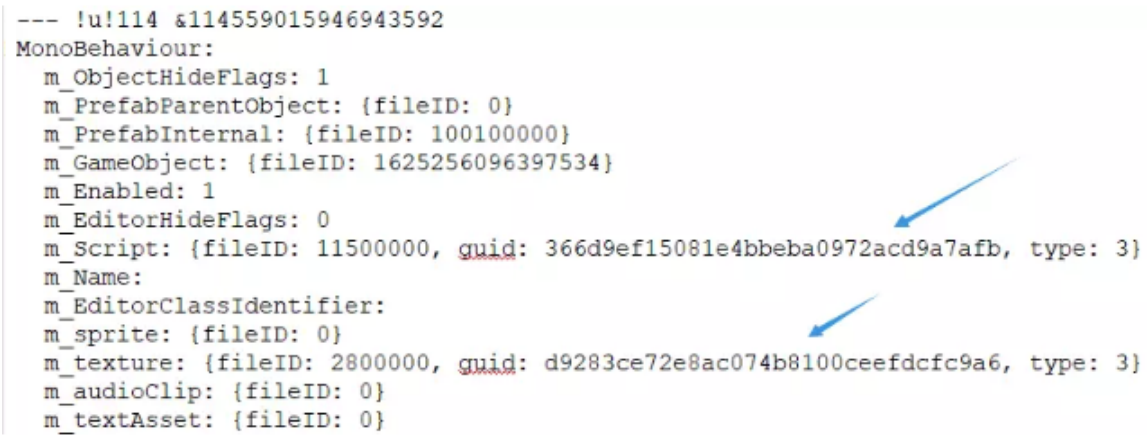


图 14.2: 引用演示

**资源内部 的依赖关系**使用 `local ID` 来确定。*如果说 File GUID 表示为文件和文件之间的关系，那么* **Local ID 表示的就是文件内部各对象之间的关系**，打开一个`*.Prefab` 文件可以很清晰的看到：

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!1001 &100100000
Prefab:
  m_ObjectHideFlags: 1
  serializedVersion: 2
  m_Modification:
    m_TransformParent: {fileID: 0}
    m_Modifications: []
    m_RemovedComponents: []
  m_ParentPrefab: {fileID: 0}
  m_RootGameObject: {fileID: 1239968257537650}
  m_IsPrefabParent: 1
--- !u!1 &1239968257537650
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 100100000}
  serializedVersion: 5
  m_Component:
  - component: {fileID: 4080989228340808}
  m_Layer: 0
  m_Name: Bullet_01
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!1 &1892309111936820
GameObject:
  m_ObjectHideFlags: 0
  m_PrefabParentObject: {fileID: 0}
  m_PrefabInternal: {fileID: 100100000}
  serializedVersion: 5
  m_Component:
  - component: {fileID: 4051113502400670}
  - component: {fileID: 336367055954951l8}
  - component: {fileID: 236575043354805 30}
  m_Layer: 0
  m_Name: Renderer
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &4051113502400670
Transform:
```

图 14.3: LocalId 演示

**一个对象通常是由一个或多个对象构成**，每个记录在 & 符号后面的数字都是一个`Local ID`，每一个`Local ID` 也表示这它将来也会被实例化成一个对象。也就是说，当一个 prefab 文件要实例化成一个 GameObject 时，它会自动尝试获取其内部`Local ID` 所指的那个对象。如果这个所指的对象当前还没有被实例化出来，那么 Unity 会自动实例化这个对象，如此递归，直到所有涉及的对象都被实例化。

### 14.1.2 InstanceID

Unity 为了在运行时，提升资源管理的效率，会在内部维护一个缓存表，负责**将文件的 GUID 与 fileID 转换成为整数数值**，这个数值在本次会话中是唯一的，**称作实例 ID(InstanceID)**。

Unity 通过 **Instance ID，来获取或判断一个对象是否已经被加载完毕**。

每当 Unity 读入一个 **File GUID 和 LocalID** 时，就会自动将其转换成一个简单好记的数字牌，*因为通过 File GUID 和 Local ID 定位资源的效率并没有直接解引用一个地址那么快。*

如果发现这个牌上并没有挂着一把钥匙，表示当前这个这个资源还在磁盘中，尚不在内存里(没有加载)；相反，如果这个牌子上有一把钥匙，表示这个资源已经被加载完毕，你可以快速的找到并使用它。

Unity 会在项目启动后，创建并一直维护一张**“映射表”**，这张映射表记录的就是 **File GUID、Local ID** 以及由它们转换而成的 **Instance ID** 之间的关系，**这样下次在请求资源时就可以快速的通过查看钥匙牌来获取资源了**

*程序启动时，实例 ID 缓存与所有工程内建的对象 (例如在场景中被引用)，以及 Resource 文件夹下的所有对象，都会被一起初始化。如果在运行时导入了新的资源，或从 AssetBundle 中载入了新的对象，缓存会被更新，并为这些对象添加相应条目。实例 ID 仅在失效时才会被从缓存中移除，当提供了指定文件 GUID 和 fileID 的 AssetBundle 被卸载时会产生移除操作。*

*卸载 AssetBundle 会使实例 ID 失效，实例 ID 与其文件 GUID 和 fileID 之间的映射会被删除以便节省内存。重新载入 AssetBundle 后，载入的每个对象都会获得新的实例 ID。*

Unity 维护的一套将 GUID 和 LocalID 解析为数据源地址的机制，这套机制中的信息，来自于：

- **场景加载时**，Unity 收集了与**该场景关联的**资源信息
- **项目启动时**，Unity 收集了所有 **Resources 文件夹下的**资源信息
- **读取 AssetBundle 时**，Unity 获取了 **AssetBundle 文件的**头部信息 (Header)

### 14.1.3 资源的生命周期

Object 从内存中加载或卸载的时间点是定义好的。Object 有两种加载方式：**自动加载**与**外部加载**。当对象的实例 ID 与对象本身解引用，对象当前未被加载到内存中，而且可以定位到对象的源数据，此时对象会被自动加载。对象也可以外部加载，通过在脚本中创建对象或者调用资源加载 API 来载入对象（例如：AssetBundle.LoadAsset）

对象加载后，Unity 会尝试修复任何可能存在的引用关系，通过将每个引用文件的 GUID 与 FileID 转化成实例 ID 的方式。一旦对象的实例 ID 被解引用且满足以下两个标准时，对象会被强制加载：

实例 ID 引用了一个没有被加载的对象。

实例 ID 在缓存中存在对应的有效 GUID 和本地 ID。

如果文件 GUID 和本地 ID 没有实例 ID，或一个已卸载对象的实例 ID 引用了非法的文件 GUID 和本地 ID，则引用本身会被保留，但实例对象不会被加载。在 Unity 编辑器中表现为空引用，在运行的应用中，或场景视图里，空对象会以多种方式表示，取决于丢失对象的类型：网格会变得不可见，纹理呈现为紫红色等等。

### 14.1.4 MonoScripts

一个 MonoScripts 含有三个字符串：*程序库名称*，*类名称*，*命名空间*。

构建工程时，Unity 会收集 **Assets 文件夹**中**独立的脚本文件**并编译他们，**组成一个 Mono 程序库**。Unity 会将 Assets 目录中的语言分开编译，**Assets/Plugins 目录**中的脚本**同理**。*Plugin 子目录之外的 C# 脚本会放在 Assembly-CSharp.dll* 中。而 Plugin 及其子目录中的脚本则放置在 *Assembly-CSharp-firstpass.dll* 中。

这些程序库会被 MonoScripts 所引用，并在程序第一次启动时被加载。

## 14.2 资源文件夹

### 14.2.1 Assets 文件夹

为 Unity 编辑器下的资源文件夹，Unity 项目编辑时的所有资源都将置入此文件夹内。在编辑器下，可以使用以下方法获得资源对象：

```
AssetDatabase.LoadAssetAtPath("Assets/x.txt");
```

注意：**此方法只能在编辑器下使用**，当项目打包后，在游戏内无法运作。参数为包含 Assets 内的文件全路径，并且需要文件后缀。

**Assets 下的资源除特殊文件夹内，或者在会打入包内的场景中引用的资源，其余资源不会被打入包中。**

### 14.2.2 Resources 文件夹

**资源载入 ->**

Assets 下的特殊文件夹，**此文件夹内的资源将会在项目打包时，全部打入包内**，并能通过以下方法获得对象：

```
Resources.Load("fileName");
```

注意：**函数内的参数为相对于 Resource 目录下的文件路径与名称**，不包含后缀。Assets 目录下可以拥有任意路径及数量的 Resources 文件夹，在运行时,Resources 下的文件路径将被合并。

例：`Assets/Resources/test.txt` 与 `Assets/TestFloder/Resources/test.png` 在使用 *Resource.Load("test")* 载入时，将被视为同一资源，只会返回第一个符合名称的对象。如果使用 *Resource.Load(“test”)* 将返回 *text.txt*；

如果在`Resources` 下有**相同路径及名称的资源**，使用以上方法只能获得第一个符合查找条件的对象，使用以下方法能**获得到所有符合条件的对象**：

```
Object[] assets = Resources.LoadAll("fileName");
TextAsset[] assets = Resources.LoadAll("fileName");
```

**相关机制 ->**

在工程进行打包后，Resource 文件夹中的资源将进行加密与压缩，**打包后的程序内将不存在 Resource 文件夹，故无法通过路径访问以及更新资源**。

在程序启动时会为 Resource 下的所有对象进行初始化，构建实例 ID。随着 Resource 内资源的数量增加，此过程耗时的增加是非线性的。故会出现程序启动时间过长的问题，请密切留意 Resource 内的资源数量。

**资源卸载 ->**

所有实例化后的`GameObject` 可以通过`Destroy` 函数销毁。请留意`Object` 与`GameObject` 之间的区别与联系.

Object 可以通过Resources 中的相关Api 进行卸载

```
Resources.UnloadAsset(Object);//卸载对应Object
Resources.UnloadUnusedAssets();//卸载所有没有被引用以及实例化的Object
```

注意以下情况：

```
Object obj = Resources.Load("MyPrefab");
GameObject instance = Instantiate(obj) as GameObjct;
......
Destroy(instance);
Resources.UnloadUnusedAssets();
```

此时UnloadUnusedAssets 将不会生效，因为obj 依然引用了MyPrefab，需要将obj = null，才可生效。

### 14.2.3 StreamingAssets 文件夹

**简介 ->**

StreamingAssets 文件夹为**流媒体文件夹**，此文件夹内的资源将**不会经过压缩与加密**，**原封不动的打包进游戏包内**。在游戏安装时，StreamAssets 文件件内的资源将根据平台，移动到对应的文件夹内。StreamingAssets 文件夹在 Android 与 IOS 平台上为**只读文件夹**.

你可以使用以下函数获得不同平台下的 StreamingAssets 文件夹路径：

```
Application.streamingAssetsPath
```

请参考以下各平台下 StreamingAssets 文件夹的等价路径，Application.dataPath 为程序安装路径。Android 平台下的路径比较特殊，请留意此路径的前缀，在一些资源读取的方法中是不必要的（*AssetBundle.LoadFromFile*，下详）

- Application.dataPath+"/StreamingAssets"//Windows OR MacOS
- Application.dataPath+"/Raw" //IOS
- "jar:file://"+Application.dataPath+"!/assets/" //Android

**文件读取 ->**

StreamingAssets 文件夹下的文件在游戏中**只能通过**IO Stream 或者WWW 的方式**读取**（AssetBundle 除外）.

**IO Stream 方式 ->**

```
using
(
    FileStream stream =
    File.Open(Application.streamingAssetsPath+"fileName", FileMode.Open)
)
{
    //处理方法
}
```

**WWW 方式 ->**

```
using
(
    WWW www =
    new WWW(Application.streamingAssetsPath+"fileName")
)
{
    yield return www;
    www.text;
    www.texture;
}
```

**AssetBundle 特有读取方式 ->**

```
string assetbundlePath =
#if UNITY_ANDROID
    Application.dataPath+"!/assets";
#else
    Application.streamingAssetsPath;
#endif
AssetBundle.LoadFromFile(assetbundlePath+"/name.unity3d");
```

**PersistentDataPath ->**

`Application.persistentDataPath`

Unity 指定的一个可读写的外部文件夹，该路径因平台及系统配置不同而不同。可以用来保存数据及文件。该目录下的资源不会在打包时被打入包中，也不会自动被 Unity 导入及转换。该文件夹只能通过 IO Stream 以及 WWW 的方式进行资源加载。

## 14.3 WWW 载入资源

### 简介 ->

WWW 是一个Unity 封装 的**网络下载模块**，支持 **Http** 以及 **ftp** 两种 URL 协议，并会尝试将资源转换成 *Unity* 能使用的 *AssetsComponents*（如果资源是 *Unity* 不支持的格式，则只能取出 *byte[]*）。WWW 加载是**异步方法**。

```
byte[] bytes = WWW.bytes;
string text = WWW.text;
Texture2D texture = WWW.texture;
MovieTexture movie = WWW.movie;
AssetBundle assetbundle = WWW.assetBundle;
AudioClip audioClip = WWW.audioClip;
```

### 相关机制 ->

每次 new WWW 时，Unity 都会启用一个线程去进行下载。通过此方式读取或者下载资源，会在内存中生成 WebStream，WebStream 为下载文件转换后的内容，占用内存较大。使用 WWW.Dispose 将终止仍在加载过程中的进程，并释放掉内存中的 WebStream。

如果 WWW 不及时释放，将占用大量的内存，推荐搭配 using 方式使用，以下两种方式等价。

#### 通过 new WWW、Try ：

```
// 方式一
WWW www = new WWW(Application.streamingAssetsPath+"fileName");
try
{
    yield return www;
    www.text;
    www.texture;
}
finally
{
    www.Dispose();
}
```

#### 通过 Using ：

```
// 方式二
```

```
using(WWW www =
    new WWW(Application.streamingAssetsPath+"fileName"))
{
    yield return www;
    www.text;
    www.texture;
}
```

如果载入的为 Assetbundle 且**进行过**压缩，则**还会在**内存中**占用一份** AssetBundle 解压用的缓冲区 *Deompresion Buffer*，**AssetBundle** 压缩格式的不同会影响此区域的大小。

**LoadFromCacheOrDownload ->**

```
    int version = 1;
    WWW.LoadFromCacheOrDownload(PathURL+"/fileName",version);
```

使用此方式加载，将**先从硬盘**上的存储区域**查找是否有对应的资源，再验证本地** Version 与传入值之间的关系，如果传入的 Version **大于**本地，则从传入的 *URL* 地址下载资源，并缓存到硬盘，替换掉现有资源，如果传入 Version **小于等于**本地，则直接从本地读取资源；**如果本地没有存储资源，则下载资源**。

此方法的存储路径无法设定以及访问。使用此方法载入资源，不会在内存中生成 WebStream（其实已经将 WebStream 保存在本地），如果硬盘空间不够进行存储，将自动使用 new WWW 方法加载，并在内存中生成 WebStream。在本地存储中，使用 fileName 作为标识符，所以更换 URL 地址而不更改文件名，将不会造成缓存资源的变更。保存的路径无法更改，也没有接口去获取此路径

## 14.4 AssetBundle

https://blog.csdn.net/qq_35361471/article/details/82854560

### 14.4.1 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

**图例** AssetBundle 资源组织形式：

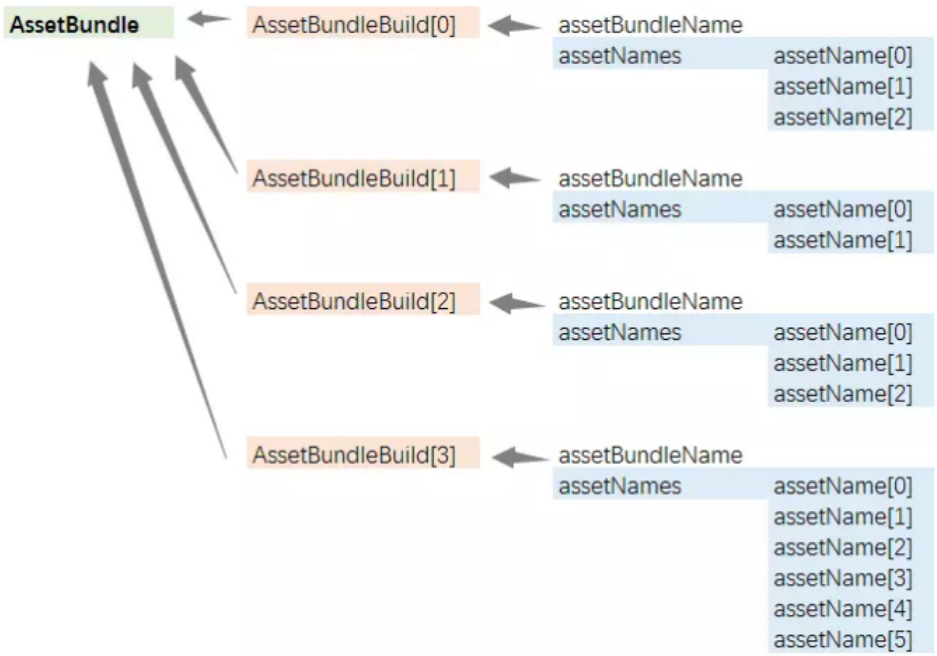


图 14.4: AssetBundle 组织形式

### 14.4.2 关键路径说明

**Resources -只读**

- Resources 文件夹下的资源无论使用与否都会被打包
- 资源会被压缩，转化成二进制
- 打包后文件夹下的资源只读
- 无法动态更改，无法做热更新
- 使用`Resources.Load` 加载

**StreamingAssets -只读**

- 流数据的缓存目录
- 文件夹下的资源无论使用与否都会被打包
- 资源不会被压缩和加密
- 打包后文件夹下的资源只读，主要存放二进制文件

- 无法做热更新
- WWW 类加载（一般用 CreateFromFile ，若资源是 AssetBundle，依据其打包方式看是否是压缩的来决定）
- 相对路径，具体路径依赖于实际平台

**Application.dataPath -只读**

- 游戏的数据文件夹的路径（例如在 Editor 中的 Assets）
- 无法做热更新
- 很少用到

**Application.persistentDataPath -读写**

- 持久化数据存储目录的路径（沙盒目录，打包之前不存在）
- 文件夹下的资源无论使用与否都会被打包
- 运行时有效，可读写
- 无内容限制，从 StreamingAsset 中读取二进制文件或从 AssetBundle 读取文件来写入 PersistentDataPath 中
- 适合热更新

### 14.4.3 热更新原理

- 将 **StreamingAsset** 目录下的文件，全部复制到目标平台（ios/Android）的数据目录中（第一次运行游戏时）
  - 启动游戏前刷新所有文件夹，客户端文件与服务上的文件对比 **MD5**，记录不同的文件，下载对应的文件替换
    - 从数据目录加载 AB（使用协程不使用线程）
      - 实例化后卸载内存中的 AB

https://www.cnblogs.com/fuyunzzy/p/6527120.html

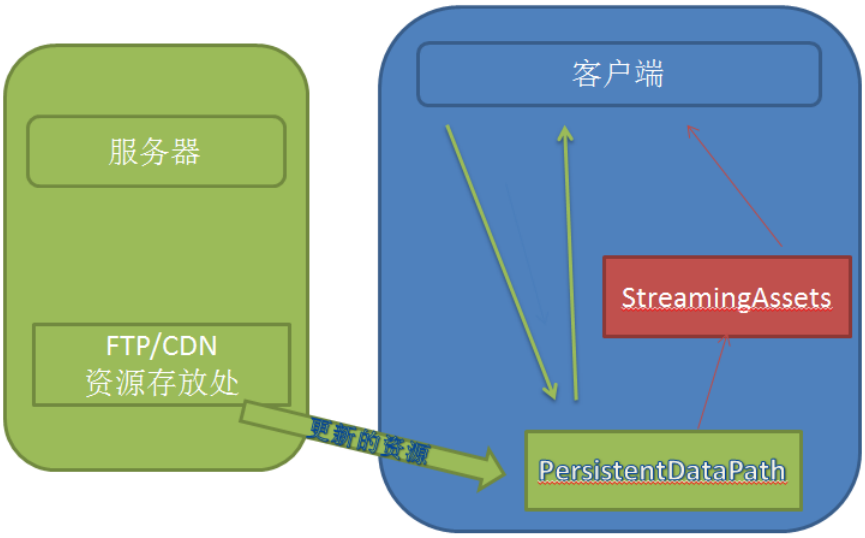


图 14.5: 热更目录关系

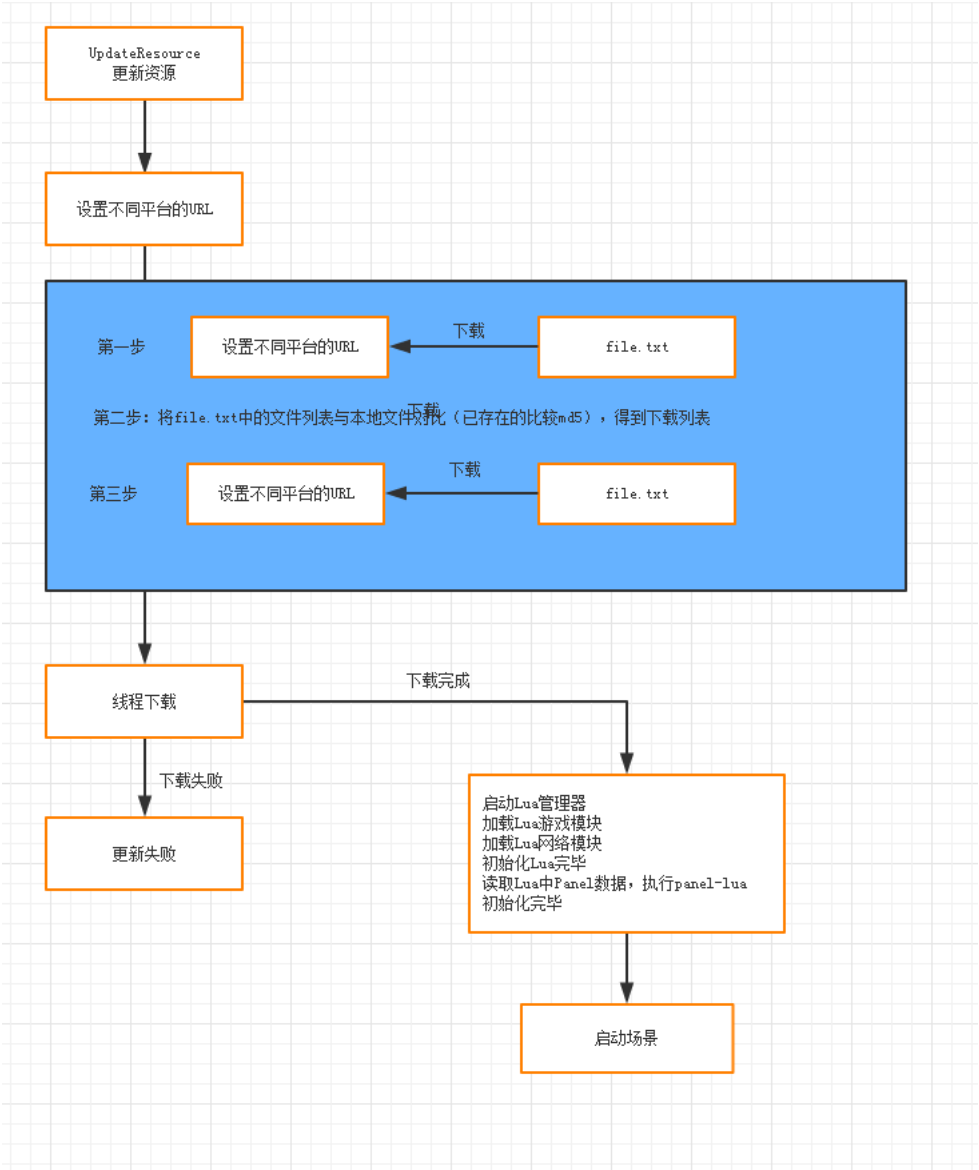


图 14.6: 热更流程

**Notice** 加载（和下载）AB 的思路

- 建立一个队列，队列中有加载 AB 的请求时，去加载，加载完成后移出队列，加载下一个直到完成
- Unity5.0 之后的 AB 资源是分开的，加载时难以判断依赖关系，而且 AB 资源在内存中只能有一个，所以**使用单线程加载**

### 14.4.4 AB 包之间的关系

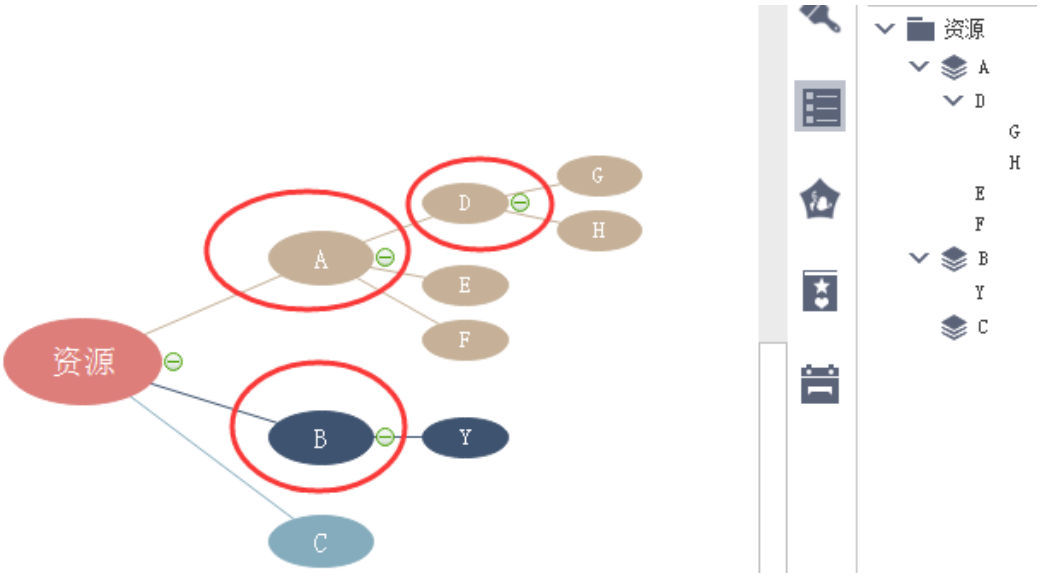


图 14.7: AB 包依赖关系说明

画红圈的就是有依赖关系的，在 Unity 中，如何知道有哪些资源有依赖关系呐。

一般，我们每次 Build 后会**自动生成 AndroidManifest.xml**，这里面就详细叙述了资源之间的依赖关系。

#### 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。

**加载依赖关系**

```
AssetBundle _manAB = AssetBundle.LoadFromFile(_manpath);
//固定格式
AssetBundleManifest manifest = _manAB.LoadAsset<AssetBundleManifest>("
    AssetBundleManifest");
string[] dependencies = manifest.GetAllDependencies(ABName);
```

### 14.4.5 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。

**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**。

*每一次调用 BuildPipleLine.BuildAssetBundles* 时，**将会生成一批 AssetBundle 文件**，具体数量根据传递 AssetBundleBuild 数组决定，**每一个 AssetBundleBuild 对象将对应一个 *AssetBundle* 及一个同名 *+.manifest* 后缀文件**。其中 AssetBundle 文件的后缀用户自行设置，比如".unity3d"，".ab" 等等；而.manifest 文件是给人看的，**里面有这个 AssetBundle 的基本信息以及非常关键的资源列表**。

除了 AssetBundleBuild 数组所定的 AssetBundle 外，还将额外在 *output* 路径下生成的一对与 *output* 文件夹同名的文件及一个同名*.manifest* 后缀文件。**这个同名文件可厉害了，它记录了这批次 AssetBundle 之间的相互依赖关系**。当然*.manifest* 文件还是给人看的，我们可以用它分析资源间的依赖关系，但是在项目实际运行时，*Unity* 并不会关心它。

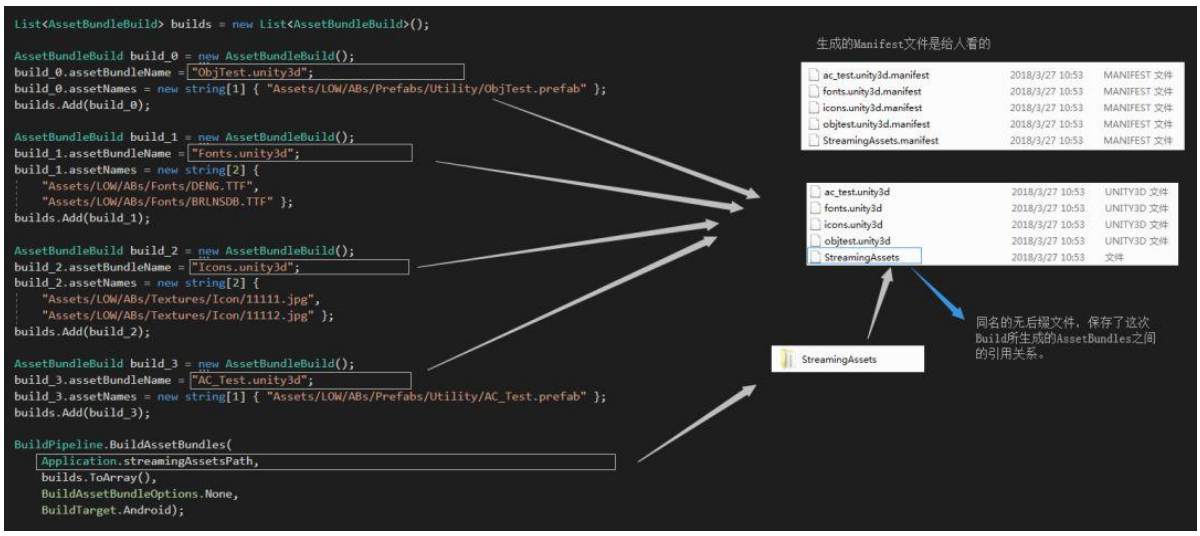


图 14.8: BuildAssetBundles 示例

**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**

**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）

### 14.4.6 下载方式

通过对比 persistencePath 下的 AB 下的 version 文件，决定需要下载什么。

```
luascripts.ab,162822234013420231022341933436167877156,9KB
otherres.ab,13515120917389401171151792311681056611613l120,3700KB
animation/emoticonanimation/animbengkui.ab,13617368129261369424524913623561471911353 9,1KB
animation/emoticonanimation/animbishi.ab,16422111320016167126146236244280166986 5,1KB
animation/emoticonanimation/animbizui.ab,24713122716711720623824642311871506221192,1KB
animation/emoticonanimation/animdaku.ab,13176595425121418512231941561781831581172 18,1KB
animation/emoticonanimation/animdamuzhi.ab,68591721832339311135792522441851221372139,1KB
animation/emoticonanimation/animdeyi.ab,19824411117436211201122149252751216650422 08,1KB
animation/emoticonanimation/animhaixiu.ab,23923926831659231811517018370168223229 38,1KB
animation/emoticonanimation/animhuaji.ab,14013971181139155149893812610410211612124421 6,1KB
animation/emoticonanimation/animjingya.ab,17749189138209233141355211761899516714122 7,1KB
animation/emoticonanimation/animpenshui.ab,15627452022518431882236577172851828521 1,1KB
animation/emoticonanimation/animsikao.ab,18848180254128266718093615311719018820110 2,1KB
animation/emoticonanimation/animtaoxin.ab,38255104120233106196452242544210672432314 6,1KB
animation/emoticonanimation/animwoniu.ab,20718891901232174149732382181222381071 57,1KB
animation/emoticonanimation/animxiao.ab,98115126249787658913618712194681379 0,1KB
animation/propanimation/animprop_caidao1.ab,24710922621550458022016537107173165150229234,1KB
animation/propanimation/animprop_caidao2.ab,13013618627186945324319823916417878229111 60,1KB
animation/propanimation/animprop_dushen1.ab,26188218811823317770165521994181149115 95,1KB
```

图 14.9: AbDownload 方式

### 14.4.7 AssetBundle 压缩类型

- 不压缩
- LZMA
- LZ4

### 14.4.8 LZMA 与 LZ4 压缩对比

假如你是一个 Prefab, 妹纸是一个资源, 妹纸在一个闺蜜圈. 你想约妹纸出来吃饭

LZ4: 直接约妹纸出来就可以了, **块读取**

LZMA: 首先要通知整个闺蜜圈, 然后再看圈内人跟其他圈子会有什么关系, 把相关人也一起通知 (闺蜜丈夫等) 然后闺蜜丈夫再找相关关系人, 直到全部都批准了, 才能和妹纸吃饭.(**依赖读取**),Android.manifest 包含了 AB 资源的信息 (名称, 依赖等),可以使用 AssetDatabase.GetDependencies(pat 找出资源引用关系

LZMA 会比 LZ4 读取成本高不少, 特别是在 Assetbundle 包引用混乱的情况下

### 14.4.9 选择策略

### 14.4.10 加载方式

根据 AssetBundle 文件所在的位置 (本地、远端)，AssetBundle 有不同的加载方式，一般可以分为以下 4 种。

- WWW
- LoadFromFile 或 LoadFromFileAsync
- WebRequest
- LoadFromMemory

**AssetBundle 的加载**可以理解为：**Bundle 加载**和 **Asset 加载**两部分。因为 AssetBundle 文件可以从功能上分为两大块：

1. Header 部分：记录文件标记、压缩信息、文件列表

2. Data 部分：记录资源实际内容

当使用 *AssetBundle.LoadFromFile* 或 *LoadFromFileAsync* 时，在 pc 平台及移动平台上，unity **仅会为我们读取 AssetBundle 的 header 部分，并不会将 bundle 的 data 部分整个读入内存**。

**当调用上一步生成的 AssetBundle 对象读取具体资源时**（*LoadAsset, LoadAssetAsync, LoadAllAssets*），Unity 会参考已经缓存的文件列表，**找到目标资源在 data 部分的位置并读入到内存中**。

如果一个资源引用到了其他资源，则必须要先读入被引用资源的 AssetBundle 文件，否则就会发生引用 Miss。

为了避免上面 Miss 的情况，在加载资源时，首先需要将该资源的依赖项全部加载完毕，不过仅需加载依赖资源的 AssetBundle 文件。也就是说，我们只要将该依赖 AssetBundle 的 Header 部分加载（AssetBundle.LoadFromFile 或 LoadFromFileAsync）就可以，这样在真正读取 Asset 时，Unity 会自动处理好真实依赖的 Asset，我们不用操心。

AssetBundle 的依赖关系如何读取呢？加载上面提到的那个很厉害的文件就可以了。

```
// 用与读取AssetBundle 一样的方式读取AssetBundleManifest
var bundle = AssetBundle.LoadFromFile(bundleManifestPath);
if(bundle)
{
    // 读取AssetBundleManifest 资源
    AssetBundleManifest abManifest = bundle.LoadAsset<AssetBundleManifest>("
        AssetBundleManifest");
    // 获取生命的全部AssetBundle, 对应一次BuildPipeline.BuildAssetBundles
    string[] assets = abMainifest.GetAllAssetBundles();
    // 遍历所有AB 资源
    foreach(var item in assets)
    {
        // 获取AB资源的直接依赖 A->B->C 则返回B
        string[] dependencies = abManifest.GetDirectDependencies(item);
        // 获取AB 资源的所有依赖资源 A->B->C 则返回BC
        // string[] dependencies = abManifest.GetAllDependencies(item);
    }
}
```

### 14.4.11 加载优先级

**首先**加载的是 **常驻公共资源**，

**再**加载经常**反复资源**,

表 14.1: AssetBundle 加载方式对比

| 方式 | 不压缩 | LZ4 | LZMA |
|---|---|---|---|
| WWW 加载 | | | |
| LoadFromCacheOrDownload 加载 | | | |
| LoadFromMemory (异步) 加载 | | | |
| LoadFromFile(异步) 加载 | | | |
| WebRequest 加载 | | | |

当然如果进入战斗（某个**特定模块**），就加载**非常驻公共资源**。

**其他资源是当使用时加载**。

### 14.4.12 缓存

一般常驻资源，最好缓存 **把 AB 存为需要的 GameObject**，例如子弹之类。

### 14.4.13 使用

当`AssetBundle` 被成功加载后，调用该`Assebbundle` 对象的`LoadAsset`、`LoadAllAssets` 或对应的异步版本即可加载资源，也就是实例化对象。**如果这个对象已经被加载过，Unity 并不会重复加载**，还记得之前所说的映射表么，被加载过的资源就好比挂上了数字牌的钥匙，直接对地址解引用即可。

### 14.4.14 释放资源-卸载

常驻资源肯定是不能 unload(false) 的，这边释放资源，释放的是其他资源。

对于 unload(false), 还是 unload(true). unload(false) 适合反复使用资源，一次 Load 之后，再也不需要 Load，**且与其他资源没有" 被依赖关系"**。

- `unload(false)`, 释放的是自身。
- `unload(true)`, 释放的是自身以及自身所有的子节点。因此，unload（true），慎用。

### 14.4.15 内存模型

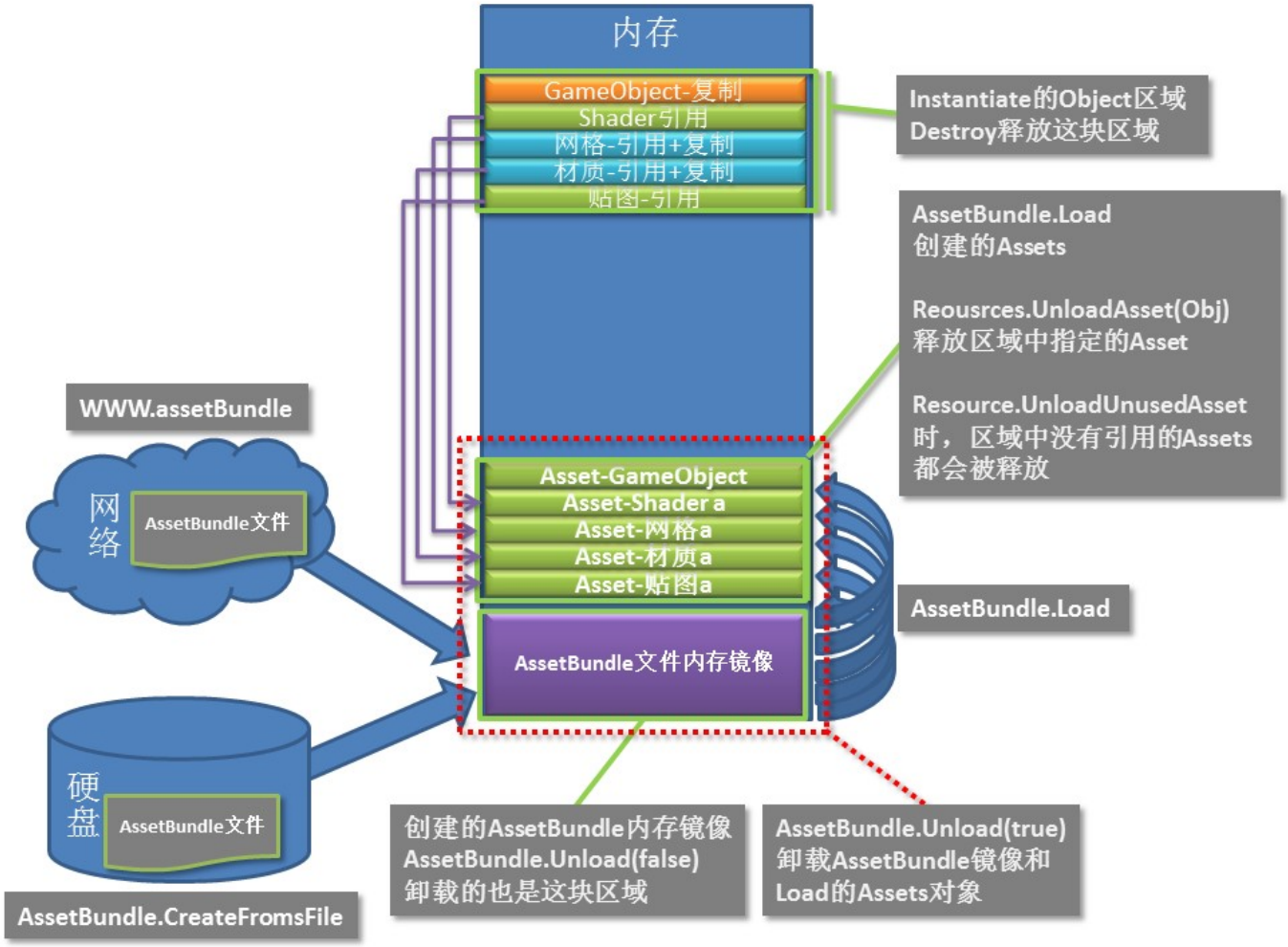


图 14.10: AssetBundle 内存模型

## 14.5 资源打包

Unity 以下资源会被打进包里。

- Assets 下所有脚本
- Assets 下所有被引用到的所有文件
- Resource 下的所有文件
- Plugins 下的对应平台下的所有文件
- StreamingAssets 下的所有文件
- AB 包打成大包的时候会打进包体

### 14.5.1 降低包体准则

☑ 打包 Android 可以选择 sdk 版本不用太高

☐ 在 StreamingAssets 删除不用的东西

☐ 打包时查看 log 纪录，由此判断需要减少的文件类型

☐ 优化，压缩网格和动画，减少文件大小

☐ 优化，压缩图片，减少图片大小

☐ 剔除教程：https://blog.csdn.net/yxriyin/article/category/2879081

- 分析打包的日志文件删除无用文件
- 重制图集
- 压缩 png
- 剔除重复资源

针对于目前这种困境，不少的服务商推出了分包技术，而传统的分包是将资源切割，分段下载，进入游戏前进行二次下载；处理游戏包体瘦身时，往往采用删除代码，精简资源甚至作资源取舍的方式；在资源加载时，玩家必须要中断游戏，并在等待中进行缓慢的资源加载。

官方解释对我们帮助甚少。所以，我们还需要一样利器，www.LoadFromCacheOrDownload。我把包分成三段：

1. **首包** (里面包含了最最必要的资源)。
2. **首次进入包加载** (加载游戏运行必要的资源)。
3. **游戏运行中资源加载** (按每个游戏不同各自定义，以场景和单位个体为主要，在画面进入时加载资源，加载结束后再运行并显示)。

### 14.5.2 针对删除无用资源的方案

https://www.cnblogs.com/alongu3d/p/3203292.html

https://github.com/handcircus/Unity-Resource-Checker

## 14.6 参考


https://blog.csdn.net/swj524152416/article/details/54022282

https://blog.csdn.net/foxyfred/article/details/73995577

https://www.jianshu.com/p/2a7c4a48aaee

http://www.360doc.com/content/17/1130/11/50169787_708566648.shtml

https://docs.unity3d.com/Manual/ReducingFilesize.html

# 第十五章　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。

## 15.1　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`[SerilizedFiled]` 下的东西, 与`[Menuitem("Netease/xx")]` 一一对应。
5. **给该方法上添加**`[Menuitem("")]` 特性

## 15.2　在编辑器上增加一个 MenuItem

## 15.3　创建一个对话框

## 15.4　扩展 Inspector 面板

## 15.5　编辑器插件常用函数

### 15.5.1　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);
            BundelNameCreater.Proc(dir.Parent.FullName);
        }
    }

}
```

## 15.6 一些常用的 Inspector 属性设置

## 15.7 参考


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

# 第十六章　跨平台发布 apk

## 16.1　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)

## 16.2　Apk 安装常见错误

http://mumu.163.com/2017/03/30/25905_680657.html

# 第十七章　AI

Most game AI that currently exists is hand coded, consisting of decision-trees with sometimes up to thousands of rules. All of which must be maintained by hand, and thoroughly tested. In contrast, ML relies on algorithms which can make sense of raw data, without the need of an expert to define how to interpret that data.

This automated learning can be applied specifically to game agent behavior a.k.a. NPCs. We can use Reinforcement Learning (RL) to train agents to estimate the value of taking actions within an environment.

## 17.1　行为树

## 17.2　Machine Learing

https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/?_ga=2.160895102.2116653917.1574651205-1790393125.1531538815

### 17.2.1　BanditDungeon

https://github.com/Unity-Technologies/BanditDungeon

https://blogs.unity3d.com/2017/06/26/unity-ai-themed-blog-entries/

在最简单的情况下，有一个包含两个箱子的房间。打开箱子会产生钻石（好东西）或鬼（坏东西）。多次打开相同的箱子会根据产生钻石的一些潜在概率产生不同的钻石和幻影序列。例如，概率为 0.5 的胸部表示将产生 50-50 的钻石和鬼影的混合，而概率为 0.9 的胸部则表示将产生十分之一的钻石（约十分之一）。注意，每个箱子都有其自己的真实概率，该主体（在这种情况下，是决定打开哪个箱子的实体）不知道的。代理人每次选择箱子时，在发现钻石的情况下要么获得正面奖励，要么在发现鬼影的情况下获得负面奖励。代理商的目标是在许多试验中最大化其总奖

励-在每次试验中，代理商都可以选择任何箱子。

如果探员知道每个箱子的真实潜在概率，那么它的任务就很简单，它要做的就是反复选择产生钻石可能性最高的箱子。但是，在没有此信息的情况下，最好的办法是在估计概率（称为探索）与选择具有最高估计概率的胸部（称为利用）之间进行明智的权衡。仅进行探索的特工将浪费所有估计每个箱子概率而不最大化其自身奖励的试验，而进行有限探索的特工将基于不准确的概率估算浪费大部分试验。这里的关键是如何有效地平衡勘探与开发。

包含两个箱子的单个房间的最简单方案可以扩展为包括多个箱子的多个房间。在此演示中，您将可以在无状态的强盗（一个房间）或上下文的强盗（三个房间）之间进行选择。对于这两种情况中的任何一种，除了下面讨论的其他一些设置之外，您还可以选择每个房间的箱子数（2 至 5）。

**最后收敛到一种策略获取到最大的 Rewards.**

### 17.2.2 Q-GridWorld

https://github.com/Unity-Technologies/Q-GridWorld

https://blogs.unity3d.com/2017/08/22/unity-ai-reinforcement-learning-with-q-learning

在最简单的情况下，我们有一个 5x5 的网格世界，其中有一个特工（蓝色方块），一个目标（绿色方块）和障碍物（红色方块）。对于演示的每次运行，都会随机选择代理商，目标和障碍物的位置（但在同一演示运行中保持一致）。在这种网格世界环境中，代理的目标是学习一种策略，以有效地从其起始位置导航到目标位置，同时避免障碍。它通过学习针对每个状态所采取的最佳措施来实现这一目标（通常称为强化学习中的策略）。这里的动作是移动的方向（北，南，东和西），而这里的状态是其在网格世界中的位置。从本质上讲，它学习从起始位置到目标位置的最短无障碍路径。

此处实施的 Q 学习算法通过为每个操作状态对维护一个数值来学习策略，该数值表示在处于特定状态时采取特定操作的程度。当代理探索网格世界时，每个操作状态对的此数字值都会增量更新。直观上讲，它执行几次试验，其中试验是一系列在障碍物或目标位置结束的动作。然后，对于它在整个试验中执行的每个动作状态对，如果是肯定试验（在目标位置结束），则将其值增加；如果是否定试验（与障碍物相撞），则将其值减小。代理还鼓励其发现每一条最短路径时所采取的每一个步骤都会得到少量的负面奖励。

与我们之前的多臂匪徒演示类似，这里需要进行勘探与开发的权衡。当代理程序正在进行试验时，它会在随机选择操作和在给定状态下遵循当前对最佳操作的猜测之间进行混合。这种权衡是由一个从 1 开始的  参数控制的（鼓励进行全面探索），并且在整个试验过程中将其逐渐减小到 0.1（限制探索）。因此，在运行演示时，您会注意到，随着演示的进行和 epsilon 值的下降，代

理的动作变得越来越可预测，**收敛到从其起始位置到目标位置的最佳路径**。

# 第十八章　Unity 优化经验 @ 实况

## 18.1　加载时间

- IO
- 网络请求
- GC
- 加密解密
- 协程等待
- 网格重建
- Mono 初始化

## 18.2　内存分类

- 资源内存：Texture、Sound、Material
- 引擎模块：Web Stream(WWW)、SerializedFile(AB)
- 托管堆：Class Array Container

### 18.2.1　内存问题

- 内存泄漏
- 纹理格式错误
- Unload
- 字符串操作

- Instantiate
- Linq
- 堆内存碎片化

### 18.2.2 内存工具

- Unity Profile
- Memory Profile
- IOS Instruments

## 18.3 资源缩减

- 清除无用资源，重复资源
- 确定资源压缩格式
- 确认贴图是否开启了 mipmap
- 确认贴图是否合理的使用了九宫格
- 确认大贴图是否是缩小显示了
- 在保证视觉效果不受太大损失的情况下，尝试减少贴图尺寸

# 第十九章　GC-Garbage Collection

介绍性:https://blog.csdn.net/znybn1/article/details/76464896 优化减少 GC:https://blog.csdn.net/chenxuezhi123/article/details/45249935

## 19.1

## 19.2

## 19.3

# 第二十章　调试技巧

## 20.1　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**

## 20.2　Android 与 Unity 互相调用

参考:`https://www.cnblogs.com/Colored-Mr/p/5677209.html`

### 20.2.1　Unity 调用 Android 非静态方法

```
// Unity 代码
AndroidJavaClass jc = new AndroidJavaClass ("com.unity3d.player.UnityPlayer");
AndroidJavaObject jo = jc.GetStatic<AndroidJavaObject> ("currentActivity");
jo.Call ("login","");

// Android 代码
public void login( String str ) {
  // 写上自己的操作
}
```

当我自己实际测试的时候发现，Android 这边的`login()` 不一定要写在 *com.unity3d.palyer* 包名下的 UnityPalyer 类下。

你只需要把`login()` 写在你自己定义包名下的 *UnityPlayerActivity.java* 中就可以了。当然了，该类肯定是继承 Activity 的。

你可以把鼠标放在 unity 代码的 Call 上查看方法可以填写的参数，你会发现方法可以填写的参数可以是`params object[]`。也就是可以传递多个参数，以数组的形式传递给`Android`。

### 20.2.2 Unity 调用 Android 静态方法

```
// Unity 代码
public void PKBtnClick() {
    this.test("test1", "test2", "test3", 1, true);
}

public void test( params object[] args ){
    AndroidJavaClass jc = new AndroidJavaClass ("com.Indra.Dark.UnityPlayerActivity"
        );
    jc.CallStatic ("login", args);
}

// Android 代码
public static void login( String str1, String str2, String str3, int a, boolean
    isShow ) {
    if( isShow ){
        Log.e("test", str1 + "==" + str2 + "==" + str3 + "==" + a );
    }
}
```

unity 代码中"com.Indra.Dark.UnityPlayerActivity" 为真实的包名. 类名,即 com.Indra.Dark 是**包名**，UnityPlayerActivity 为**类名**

### 20.2.3 Android 调用 Unity

```
UnityPlayer.UnitySendMessage("btnTest", "showLog", "a#b#c");
```

UnitySendMessage 的第一个参数是 unity 控件的名字，第二个参数是方法名，第三个参数是要传递的参数。而且只能传递一个参数。不过你可以把你要传递的参数做成一个字符串传递过去，unity 那边做分割字符串就行了。例如上面就是把 a,b,c 用 # 连接起来。unity 那边用 # 作为分隔符去分割就 OK 了。

那么问题来了，脚本挂在的 unity 控件名字不好找怎么办。其实还有个办法轻松搞定，那就是在脚本的 Start() 方法中指定 name 为你方法传递的控件名字就 O K 了。

如我上面的方法中要找的控件是 btnTest，则：

```
void Start () {
    this.name = "btnTest";
}

void showLog（string str）{
```

```
        Debug.Log ("str:" + str );
    }
```

# 第二十一章　书籍推荐

GPU Gems:https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch01.html

Real Time Rendering

Uniyt Shader 入门精要

## 21.1　Unity AB 包入门参考

- 各目录关系https://www.cnblogs.com/u3ddjw/p/6691932.html
- AB 加载 https://www.cnblogs.com/u3ddjw/p/9120430.html
- AB 资源关系 https://www.cnblogs.com/u3ddjw/p/9120430.html