

Unity的基本信息

各个面板

脚本

预制体

坐标系

快捷键

常用几个API

transform状态

属性

方法

Debug控制台输出

方法

GameObject游戏物体

属性

方法

递归查找子物体

Vector3方向、坐标、向量

Input输入控制

键盘Key

鼠标Mouse

GetAxis获取轴

Random

Resources加载游戏资源

Time时间类

Mathf数学类

PlayerPrefs存储数据

场景切换

关于Application的一些API

生命周期

生命周期函数

基础组件操作

Light灯光组件

组件获取与添加

组件的获取

组件添加

GameObject和Transform的关系

二进制标识

Audio Source声音组件

属性

方法

Audio Clip

OnGUI老版UI系统

GUI类

碰撞

Rigidbody刚体和Collider碰撞体

参数

发生碰撞的必要条件

检测碰撞OnCollision

发生触发的必要条件

触发器OnTrigger

触发器和碰撞器的区别和联系

缓存池

单个缓存池

缓存池存多个物体

3D数学

三角函数

基础公式

在Unity中的单位换算

Unity中的三角函数方法

向量与矢量

向量的加减

向量的点乘

Unity中的点乘公式

向量的叉乘

Unity中的叉乘公式

Vector3数学运算

四元素

四元素常用API

让摄像机随着鼠标移动

发射圆形子弹

协程

延迟函数

协程

开启关闭协程

协程的返回内容

异步加载

异步加载场景

异步加载场景实例

高级组件操作

Camera摄像机

摄像机静态属性

Camera类常用方法

摄像机组件属性

组件Line Renderer

LineRenderer类常用方法

组件的重要属性

Rigidbody刚体组件

力Force

Rigidbody类

根据鼠标的位置决定子弹方向

射线检测Physics

Ray射线类

Physics射线检测类

Physics球形搜索

图片

Texture属性

SpriteRenderer 2D图片渲染组件

效应器

修改代码模板

遇到问题解答

游戏的单例模式

为什么子弹会跟着特效乱跑

为什么播放了音效特效就出不来这个函数了

动画系统

老版的动画系统Animation

动画上的几个参数

帧事件

Animation组件

Animation组件的API

动画模型

动画模型上的参数

Avatar替身系统

新版动画系统Animator

- Animator**面板
 - Animator**组件
 - Animator**类中常用API
 - AnimatorStateInfo**类

- 动画融合树

- 1D**动画融合树
 - 2D**动画融合树
 - 融合树的嵌套**

- 动画遮罩AvatarMask

- Layers**动画层齿轮参数

- IK动画逆向动画

- Animator**类中设置IK的方法

寻路系统

- 烘焙路径

- Navigation面板

- Bake**面板

- Areas**面板

- Object**面板

- Agents** 代理模板

- off Mesh Link组件

- Nav Mesh Agent组件

- Nav Mesh Obstacle**组件

文件夹

- Unity的特殊文件夹

- 文件夹的操作

UGUI

- UI基础

- 常见的UI类型**

- 常见分辨率**

- Canvas

- Canvas**组件

- Canvas Scaler**组件

- UI基础组件

- Rect Transform**位置组件

- Text**文本组件

- Image**图片组件

- Raw Image**组件

- 图集制作**

- Mask**遮罩组件

- UI复合组件与交互

- Button**按钮组件

- 异形按钮**代码设置

- Button**添加事件代码向

- Toggle**单选控件

- InputField**输入框组件

- Slider**滑块组件

- Scrollbar**滚动组件

- Scroll View**矩形滚动组件

- Content Size Fitter

- 排列组件

- Grid Layout Group表格布局组件

- Vertical Layout Group纵向排序组件

- Horizontal Layout Group横向排序组件

- Dropdown**下拉列表

- DoTween插件

- 类拓展**

- DoTween**

- DoTween Animation图形参数

常用方法

UI控件的自定义交互

Button的自定义交互

Button长按效果的实现

拖拽效果的实现

自定义交互的组件EventTrigger

游戏开发的数据处理

JSON基础

JSON基础格式

JSON支持的数据结构

JSON的符号含义

JSON在游戏中的应用

C#使用JSON数据

操作使用的JSON文件内容

将JSON文件在Unity中读取出来

JSON工具LitJson

Unity中关于JSON的类：JsonUtility

XML

文件的读写

写

读

FileMode的几个类型

File静态类方法

Encoding编码格式

关于Close关闭流

老师给的配置文件使用方法

代码使用

网络传输

网络参考模型

网络通信

Unity网络请求类WWW

WWW下载文本

WWW下载文件

URL域名结构

HTTP请求类型（GET和POST的区别）

编码

常见状态号

TCP长链接

Socket套接字

TCP编程方法

线程

链接（三次握手）

断开（四次挥手）

发送

接收

lock关键字

数据包处理

字节序

数据包定制

数据包的生命周期

心跳包

MVC分层开发思想

MVC是什么

MVC开发步骤

AssetBundle包

AssetBundle资源管理

AB包和Resource的区别

AssetBundle的定义

AB包的创建与使用

AB包的使用流程

AB包的创建

插件AssetBundle Browser

Build下的参数信息

代码创建

AB包的使用的四种方式

本地加载AB包内部数据

服务端加载AB包

AB包的依赖关系

AB包的卸载内存的释放

AB包使用的一些问题

Lua

语言的执行方式

Lua文本工具安装和解析器安装

Lua语言特点

Lua语法

Lua变量的作用域

Lua函数与方法

Lua保留字

Lua数据类型

Lua字符串的操作

Lua运算符

Lua的表[数组]

Lua的分支判断循环控制语句

Lua的与或非

Lua迭代器

Lua函数

Lua多返回值与多变量赋值

Lua表Table的常见操作

Lua中的模块module

Lua模块(文件)的使用

Lua中C包

Lua元表

Lua元表中可设置元方法

Lua协程

Lua中的文件操作

Lua的错误处理/调试/垃圾回收

Lua面向对象

Lua作业

热更新

热更新

常见的Unity热更新插件

xLua的使用

自定义加载器

使用AB包加载Lua脚本

单例模式的解析器LuaEnv

xLua拓展学习

xLua与C#代码的相互调用

Lua调用C#静态结构

Lua调用C#非静态结构

Lua调用C#继承的方法

Lua调用C#拓展的方法

Lua调用C#结构体的方法

Lua使用C#枚举

Lua使用C#泛型

Lua的多返回值

Lua使用C#委托

C#与Lua代码的互相调用

C#使用Lua的变量

C#使用Lua的函数

C#使用Lua的表

矩阵

矩阵的定义

向量

矩阵

行矩阵

列矩阵

矩阵的转置

矩阵的乘法

矩阵乘标量

矩阵乘矩阵

Unity向量乘以矩阵

特殊矩阵

对角矩阵

数量矩阵

单位矩阵

逆矩阵

矩阵的行列式

代数余子式

标准伴随矩阵

逆矩阵

算出一个逆矩阵

矩阵变换

变换类型

线性变换

非线性变换

仿射变换

齐次矩阵构成

平移矩阵

缩放矩阵

旋转矩阵

复合变换

坐标空间

模型空间

世界空间

观察（摄像机）空间

裁剪空间

屏幕空间

Unity着色器中常见矩阵

Shader着色器

CPU编程和GPU编程的区别

GPU渲染流程

Shader着色器

Unity的Shader类型

创建一个Shader

固定管线Shader

基础CPU与GPU数据传递

着色器常用变量类型

顶点/片元着色器

Cg语言

Cg支持的7种基本数据类型

Cg向量数据类型

Cg矩阵数据类型

Cg数组

Cg结构体

- Cg中的数据类型转换
- Cg中的关系符
- Cg向量bool逻辑运算
- Cg数学操作符
- Cg位移操作符
- Cg的Swizzle操作符
- Cg条件运算符
- Cg控制流语句
- Cg关键字
- Cg输入数据流Uniform
- Cg输入输出修饰符
- Cg输入语义和输出语义的区别
- 常用输入语义
- 输出语义
- 顶点着色程序输出语义
- 片段着色程序输出语义
- 语义绑定方法
- Cg函数
- Cg标准函数库

Cg程序

基础顶点着色器

基础片元着色器

- 表面体着色器

- 去色效果实现

Shader光照

- 标准光照的构成结构

- 逐顶点光照和逐像素光照

- 漫反射光照模型

- 高光反射光照模型

- Phong着色

- 纹理采样

- 法线贴图

- 裁剪命令

- 渲染和渲染顺序

深度测试

渲染类型

深度写入

丢弃片元

- 实现透明图

- 用噪声图实现消融效果

渲染队列

颜色混合

- 光的混合方式

- 颜色混合

真机打包

- IOS打包

打包方法

- Android打包

环境配置

打包步骤

- 打包参数的设置

宏

- 真机调试

- 反编译和打包破解方法

- AB包管理

Unity的基本信息

各个面板

- **Scene** 场景面板
 - Y轴正方向相对屏幕向上
 - X轴正方向相对屏幕向右
 - Z轴正方向相对屏幕向右
 - **Game** 游戏视窗面板
 - 分辨率
 - **stats** : 场景中的各信息
 - **Project** 工程面板
 - **Inspector** 组件、属性面板
 - **Hierarchy** 层级面板
 - **Console** 控制台面板
 - **Clear on Play** : 运行之前清除之前的信息
 - **Error Pause** : 遇到错误信息就暂停游戏运行
-

脚本

- 脚本文件名必须要和类名一致
 - **Start()** : 游戏运行时该方法只会执行一次
 - **Update()** : 游戏运行时每一帧都会执行一次
 - 写之前清除自己写的代码是需要多次执行还是只需要执行一次
 - 组件必须继承 **MonoBehavior** 类才能挂载
 - 继承 **Monobehavior** 的类可以当成游戏物体组件挂载到游戏物体上
 - 继承 **MonoBehavior** 的类挂载到游戏物体上，游戏物体是激活状态¹ 就会执行代码
-

预制体

- 预制体的 **Prefab** 的三个按钮功能
 - **Select** 找到预制体本体
 - **Revert** 对预制体修改后可以重置这个修改过的预制体
 - **Apply** 对预制体的修改应用到相对的其他预制体身上
 - 取消预制体关系：菜单栏--> **GameObject** --> **Break Prefab Instance**
-

坐标系

- 世界坐标：Unity本身的世界坐标
 - 本地坐标：相对父物体的坐标，若无父对象则世界坐标就是父对象的坐标
-

快捷键

- `Ctrl+Shift+F` : 快速调整摄像机视角按当前你的游戏视窗视角
 - `Ctrl+D` : 快速复制物体在本目录
 - `Ctrl` : 按住 `Ctrl` 在场景中可以按一个单位位移
-

常用几个API

transform状态

属性

- `position` : 字段赋值是 `Vector3` 类型的, 针对的是世界坐标系的位置
- `localPosition` : 字段可赋值, 针对的是本地坐标系的位置
- `parent` : 父物体字段, 可以赋值就可以设置父物体
- `chileCount` : 子物体的个数
- `losalScale` : 相对本地坐标进行缩放
- `lossyScale` : 相对世界坐标进行缩放, 但是只能得到不能更改
- `forward` : 本地坐标的z轴, 且值是相对于世界坐标的方向值

还有其他方向的值, 可以F12查看其他方向

方法

- `Translate(Vector3)` : 做位置增量, 传进的是向量²
 - 重载函数 `Translate(Vector3, transform)` : 除了 `Vector3` 类型的, 还可以传进一个物体的 `transform`, 然后就会以这个物体的坐标系做位置增量
 - `RotateAround(Vector3, Vector3, float)` : 围绕着某个点选择, 传进的是一个点的坐标, 一个方向和转的速度值
 - `Rotate(Vector3, speed)` : 使物体朝某个方向旋转 `speed` 角度
 - `Rotate(Vector3(0, 1, 0), 10)` : 向y轴方向旋转10度
 - `Rotate(Vector3)` : 使物体在原始角度上朝这个方向旋转多少角度, 传进的是向量
 - `Rotate(Vector3(0, 10, 0))` : 朝y轴旋转10度
 - `Rotate(Vector3, Space.Self/world)` : 使物体基于自己/世界的坐标系旋转多少角度
 - `SetParent(Transform)` : 通过方法设置父物体, 传进的是父物体的 `transform`³ 信息
 - `GetGhild(int)` : 获取子物体, 类似列表的索引
 - `LookAt(Vector3)` : 物体的z轴始终指向传进游戏物体的坐标, 传进的是这个点的坐标
 - `Find(string)` : 传进游戏物体的名字 (字符串), 找到一个子物体 (只能是子物体, 子物体的子物体就无法找到), 即使游戏物体隐藏也可以找到该物体
-

Debug控制台输出

方法

- `Debug.Log()` 输出
- `Debug.LogError()` 错误输出
- `Debug.LogWarning()` 警告输出
- `Debug.DrawLine(Vector3,Vector3,color)` : 在 Unity 中画线, 传一个起始点一个终点, 就可以在场景中画出这条线来

GameObject 游戏物体

属性

- `name` : 当前游戏物体的名字, 可以修改

方法

- `Instantiate<T>(T)` : 克隆, 可以克隆游戏物体, 设置泛型参数类型为 `GameObject` 即可
- `Find(String)` : 传进游戏物体的名字, 整个场景中找一个游戏物体, 返回该游戏物体, 但是游戏物体如果是隐藏状态就无法找到, 且查找整个场景的物体消耗运行效率
- `FindGameObjectWithTag(string)` : 传进游戏物体的标签, 整个场景中找一个游戏物体, 且非激活状态无法找到
- `SetActive(bool)` : 传进一个布尔值可以控制一个游戏物体的激活状态, `true` 为激活, `false` 为非激活
- `Destroy(GameObject)` : 销毁一个游戏物体
 - 重载函数 `Destroy(GameObject, Time)` : 延迟多少时间销毁这个物体

递归查找子物体

```
//第一种方法
GameObject SetChild(GameObject parent,string childName)
{
    if(parent.name==childName)
        return parent;
    if(parent.transform.childCount<1)
        return null;
    GameObject go=null;
    for (int i = 0; i < parent.transform.childCount; i++)
    {
        go=SetChild(parent.transform.GetChild(i).gameObject,childName);
        if(go!=null)
            break;
    }
    return go;
}

//第二种方法
GameObject SetChild2(GameObject parent,string childName)
{
    GameObject go=new GameObject();
    if(parent.transform.childCount<1)
        return null;
    if(parent.transform.Find(childName)!=null)
        return parent.transform.Find(childName).gameObject;
    for (int i = 0; i < parent.transform.childCount; i++)
```

```
{
    go=SetChild2(parent.transform.GetChild(i).gameObject,childName);
    if(go!=null)
        break;
}
return go;
}
```

Vector3方向、坐标、向量

- `Vector3` 类型可以作为方向、坐标、向量²来使用
- `forward` : 本地坐标的z轴, (0,0,1)
- `back` : 本地坐标的z轴反方向, (0,0,-1)

还有其他方向的值可以VS中查看F12

Input输入控制

键盘Key

- `GetKeyDown(KeyCode)` : 按键按下瞬间执行一次, 返回布尔值, 传入的是按键键值
 - `Input.GetKeyDown(KeyCode.A)` : 判断是否输入了A键
- `GetKey(KeyCode)` : 按键按下期间一直执行, 不放键盘就会一直执行
- `GetKeyUp(KeyCode)` : 按键弹起瞬间执行一次

鼠标Mouse

- `GetMouseButton(int)` : 检测鼠标的键位, 返回的布尔值, 按下鼠标的时候一直执行
 - `int=0` : 鼠标左键
 - `int=1` : 鼠标右键
 - `int=2` : 鼠标中键
- `GetMouseButtonDow(int)` : 检测鼠标的键位, 按下瞬间执行一次
- `GetMouseUp(int)` : 检测鼠标键位, 键位弹起瞬间执行一次

GetAxis获取轴

- `Input` 设置热键: 菜单栏 `Edit --> Project Settings --> Input`
 - 可以设置对应轴的名称, 和按键
- `GetAxis(string)` : 传入的是 `Input` 设置中轴的名称, 返回的是一个 `float` 类型的值, 根据这个 `float` 值, 可以搭配 `Translate`、`Rotate` 进行左右旋转前后移动
 - “Horizontal” : 对应水平方向, 对应键盘 `A`、`D` 键, 对应游戏手柄的左右
 - “Vertical” : 对应竖直方向, 对应键盘 `W`、`S` 键, 对应游戏手柄的上下
 - “Mouse X” : 对应鼠标的左滑右滑

Random

- `Random.Range` : 随机数方法，和控制台中的 `Random` 用法类似

Resources加载游戏资源

- 从硬盘中将资源加载到游戏中
- 需要建一个**Resources**文件夹
- `Resources.Load<T>(String)` : 传入一个**Resources**文件夹下游戏物体的路径，返回的就是这个实例化后的物体
 - 例如找一个**Resources**下**Prefab**文件夹的 `GameObject` 类型的 `Bullet` 预设体
 - `Resources.Load<GameObject>("Prefab/Bullet");`
 - 加载这个文件夹下的所有文件，且按命名顺序
 - `Resources.LoadAll<GameObject>("Prefab/Bullet/");`

Time时间类

- `time` : 游戏开始运行时这个时间就从0开始计算，单位秒
- `deltaTime` : 上一帧到这一帧结束的单位时间，其实就是每帧所用的时间
- `fixedDeltaTime` : 固定的时间，且该时间是每帧的运行时间
- `timeScale` : 对时间进行缩放，可以赋值，赋值的就是要放大缩小的倍数数值，大于1时间加快，小于1时间变慢，等于0时间暂停
- `unScaleTime` : 不接受任何时间的暂停，即使游戏暂停运行它也在继续计算时间
- `frameCount` : 当前游戏跑了多少帧

Mathf数学类

- `Clamp(float, float, float)` : 限制大小，第一个参数只能在第二个和第三个之间，当小于第二个参数的时候等于它，当大于第三个参数的时候就等于它，可以用来设置血量大小等等
- `Lerp(float a, float b, float c)` : 插值运算，第一个参数是起始值，第二个是终点值，第三个是系数，计算公式--> `a+(b-a)*c`

PlayerPrefs存储数据

- `SetFloat(string, float)` : 类似字典，存储一个==标识--值内容==，将内容存在本地数据，随后可以在其他的脚本中也可以读取到这个数据，即使重新运行游戏还是会读取到这个数据
 - `SetInt()` : 存整型
 - `SetString()` : 存字符型
- `GetFloat(string)` : 通过键去寻找这个值内容，获得到这个值
 - `GetInt()`
 - `GetString()`
 - 重载函数 `GetFloat(string, int)` : `int`是一个默认值。当没有这个东西的时候，就返回的是默认值
- `DeleteAll()` : 删除所有数据
- `DeleteKey(string)` : 删除对应标识的数据

- 可以做一个排行榜

场景切换

- `SceneManager.LoadScene("场景名")`：需要引入命名空间 `UnityEngine.SceneManagement`

关于Application的一些API

- `Application.targetFrameRate = int`：限制游戏的帧率
- `Application.Quit()`：退出游戏

生命周期

生命周期函数

按执行顺序排序

- `Awake()`：只有在游戏物体唤醒的时候只执行一次，即被创建的时候执行一次，该函数不一定是最开始执行的，当这个绑定的物体再被创建的时候就会又执行一次
- `OnEnable()`：游戏物体每激活⁴一次就会执行一次
- `Start()`：游戏运行时调用一次
- `FixedUpdate`：固定步长更新，即一定时间内执行一次
 - 一般物理相关更新写在这里
 - 设置步长在菜单栏 `Edit` —> `Project Settings` —> `Time`
- `Update()`：游戏运行时每一帧执行一次
- `LateUpdate()`：每一帧都执行，但是是在 `Update` 后执行
 - 一般写摄像机跟随
- `OnDisable()`：游戏物体每失活⁵一次就会执行一次
- `OnDestroy()`：销毁之前调用的函数，销毁物体之前会调用一次 `OnDisable`

基础组件操作

Light灯光组件

- `Type`：灯光类型
 - `Directional` 直射光(类似太阳光)
 - `Point` 点光源(类似灯泡)
 - `Spot` 聚光灯
 - `Area` 区域光，常是烘焙使用
- `Color`：灯光颜色
- `Mode`：灯光烘焙模式
 - `Realtime` 实时光

- `Mixed` 混合光
 - `Baked` 烘焙光
 - `Intensity` : 光照强度
 - `Indirect Multiplier` : 光照乘数
 - 间接光从另一个游戏物体反射到另一个游戏物体的光线, 小于1时每次反射会变暗
 - `ShadowType` : 阴影类型
 - `Soft Shadows` 平滑阴影
 - `HardShadow` 硬阴影
 - `NoShadows` 没有阴影
 - `BakeShadowAngle` : 人为增加阴影平滑度, 让阴影更光滑
 - `Strength` : 阴影强度
 - `Resolution` : 阴影质量, 质量越高, 消耗越大
 - `Bias` : 阴影推移距离
 - `Normal Bias` : 光阴表面推移距离
 - `NearPlane` : (在点光源和聚光灯下起作用) 影子的近裁剪面
 - `Cookie` : 投射阴影的指定纹理
 - `cookiesize` 纹理大小
 - `Draw Halo` : 光晕开启
 - `Flare` : 光耀 (只有点光源和聚光灯下起作用) 赋值光耀文件
 - `Culling Mask` : 裁剪层 (二进制标识)
-

组件获取与添加

组件的获取

- 组件控制的步骤
 - 1.找到游戏物体
 - 2.获取到组件控制权
 - 3.控制组件

例如获得灯光的light组件, 并控制它

- 找到游戏物体
 - `go = GameObject.Find("Directional Light")`
- 获取组件控制权
 - 方法 `GetComponent<>()`
 - `object = go.GetComponent<Light>()`
- 控制组件
 - 此时就可以用 `object` 去点出该组件上的各个变量
 - `object.type = LightType.Point` 将灯光状态改成点光源

组件添加

- `AddComponent()` : 添加一个组件

GameObject和Transform的关系

- `GameObject` 必须有 `transform` 组件，所以 `GameObject` 内置了属性包含了自己的 `transform`
 - 所以所有游戏物体都可以点出 `transform` 属性
- `transform` 不能独立于游戏物体而存在，且任何组件都不能独立于游戏物体存在
 - 所以所有组件都可以点出 `gameObject`

二进制标识

- 以 `Light` 组件的 `Culling Mask` 为例，每个选项都代表一个二进制数值，当在代码选择这些数值的时候就会选择对应的选项，游戏中的应用常有buff的叠加等
 - 例如红BUFF的二进制标识是 `00000001`，蓝BUFF的二进制标识是 `00000010`，当同时获得这两个BUFF的时候，就会获得数值 `00000011`，以此来判断角色身上有多少个状态，这样就不需要用枚举来定义状态，只需要 `int` 类型的数值就可以来存储这个状态

Audio Source声音组件

属性

- `Audio.Clip`：音效片段
- `PlayOnAwake`：一开始就播放
- `Loop`：循环与否
- `Volume`：音量大小
- `Mute`: `true` 开启静音, `false` 关闭静音

方法

- `Play()`：开启声音播放

Audio Clip

- 声音片段，赋值给声音组件就可以播放对应片段

OnGUI老版UI系统

- Unity最原始的一版UI系统
- 优点是比较轻量级，缺点是操作不够方便
- 可以显示文字、图片、按钮等元素
- 全部是写在方法 `OnGUI` 中

GUI类

- `Label()`：显示一个文字框，有以下常用重载函数
 - `Label(Rect, string)`：第一个传进的是一个 `Rect` 参数，`Rect` 代表一个矩形框，可以设置位置，大小，第二个传的是显示的文字
 - `Rect(Vector2, Vector2)`：第一个设置的是在场景上的坐标，第二个是大小
 - `Rect(float, float, float, float)`：设置位置x,y，大小l,w

- `Label(Rect,string,GUIStyle)` : `GUIStyle` 可以设置字的字体颜色大小等风格
 - `Button()` : 显示一个按钮, 有以下常用重载函数
 - `Button(Rect,string)` : 传进一个矩形框的位置大小, 传进按钮上的文字
 - `Button(Rect,string,GUIStyle)` : 可以设置风格
 - `Button(Rect,Texture)` : 可以设置按钮的图片
 - `Button(Rect,GUIContent)` : `GUIContent` 可以设置图片, 文字加图片, 文字加提醒信息, 文字加图片加提醒信息
 - `HorizontalScrollbar (Rect position, float value, float size, float leftvalue, float rightvalue)` : 左右的滚动条, 需要的参数滚动条的位置、最小和最大之间的位置、滑块尺寸、滚动条左端的值、滚动条右端的值, 还可以传一个 `style`, 增加滚动条背景, 这个滚动条类似浏览器下方的那种滑动条
 - `HorizontalSlider` : 这个滚动条传的参数与上述一样, 少了个尺寸, 但这个样式就类似调节音量的
 - `GrawTexture()` : 画出这个图片
-

碰撞

Rigidbody刚体和Collider碰撞体

参数

- `Mass` : 质量
- `Drag` : 摩擦力
- `Angular Drag` : 转弯阻力
- `Use Gravity` : 是否使用重力
- `Is Kinematic` : 运动学刚体, 当两个刚体发生碰撞, 它不会发生位移
- `Collision Detection` : 检测精度
 - `Discrete` : 离散性监测
 - `Continuous` : 连续性监测
 - `Continuous Dynamic` : 连续动态监测
- `Constraints` :
 - `Freeze Position` : 锁轴位移
 - `Freeze Rotation` : 锁轴旋转
- `Interpolate` : 如果刚体抖动, `none` : 没有插值运算
- `Interpolate` : 根据上一帧位置做插值运算
- `Extrapolate` : 根据上一帧运动预测下一帧的插值

发生碰撞的必要条件

- 发生碰撞的两个游戏物体都必须要有碰撞体 `Collider`, 有一方游戏物体有刚体 `Rigidbody` 就可以发生碰撞了 (一般把刚体绑定绑在运动的游戏物体上)
- 两个游戏物体的碰撞器 `Is Trigger` 是关闭的

检测碰撞OnCollision

- `ONCollisionEnter(Collision)`：两个游戏物体==碰撞瞬间==调用这个函数，要传进的是对方的碰撞体
- `ONCollisionStay(Collision)`：两个游戏物体==碰撞期间==调用这个函数，要传进的是对方的碰撞体
- `ONCollisionExit(Collision)`：两个游戏物体碰撞==弹开瞬间==调用这个函数，要传进的是对方的碰撞体
- 即使当前脚本组件失活，也会执行这三个方法

发生触发的必要条件

- 发生碰撞出发的两个游戏物体都必须要有碰撞体（`Collider`），有一方游戏物体有刚体（`Rigidbody`）就可以发生碰撞触发了（一般是把刚体绑定在运动的物体上）
- 两个游戏物体有一个游戏物体的碰撞器和 `Is Trigger` 是开启的

触发器OnTrigger

- `ONTriggerEnter(Collider)`：两个游戏物体==触发瞬间==调用这个函数
- `ONTriggerStay(Collider)`：两个游戏物体==触发期间==调用这个函数
- `ONTriggerExit(Collider)`：两个游戏物体==触发后==调用这个函数

触发器和碰撞器的区别和联系

相同点：

- 两个发生碰撞的游戏物体必须都有 `Collider`，至少有一个游戏物体带刚体组件
- 代码组件失活了同样还会检测
- 检测碰撞代码无论写在两个发生碰撞的游戏物体的哪一方都会检测到

不同点

- 碰撞器不能开启 `collider` 的 `Is Trigger` 选项，触发器必须要有一方开启 `Is Trigger` 选项
- 触发函数的参数不同

缓存池

单个缓存池

1. 先在池子里看有没有游戏对象
2. 如果有就从池子拿出来用，没有就 `Instantiate` 出游戏对象
3. 游戏对象用完之后不进行销毁，将游戏对象回收到池子里，重复利用

使用发射子弹作为例子

```
//在玩家类中
{
    public GameObject bulletPool; //子弹缓存池

    void update()
    {
        if (Input.GetMouseButtonDown(0))
        {

```

```

GameObject go = null;
//1.需要使用子弹的时候现在缓存池查询是否有子弹，如果有直接用
//将物体设置成池子的子物体，然后进行重复使用就行
if (bulletPool.transform.childCount > 0)
{
    //池中有备用子弹
    go = bulletPool.transform.GetChild(0).gameObject;
    //将子物体从父物体拿出来，防止再次被调用
    go.transform.parent = null;
    //物体被放入缓存池时是失活的，所以拿出来的时候要激活
    go.SetActive(true);
}
else //2.若是池子里无子弹，就实例化出子弹
{
    go = Instantiate(bullet);
}
//将子弹回收，放入缓存池，这一步需要在子弹类中操作

//要给物体（子弹）设置位置和方向
go.transform.position = startPos.transform.position;
go.transform.rotation = startPos.transform.rotation;
}
}
}
//子弹类中
{
    private float timer_BulletPool; //设置一个计时器，规划子弹放入池子的时间
    private GameObject bulletPool; //设置一个标签用来找这个池子

    void update()
    {
        //设置一个计时器
        timer_BulletPool += Time.deltaTime;
        //当时间大于五秒时，将子弹放入缓存池中，重复利用
        if (timer_BulletPool > 5)
        {
            if (bulletPool == null) //如果没有找到缓存池就找一下缓存池
            {
                bulletPool = GameObject.FindGameObjectWithTag("BulletPool");
            }
            //将子弹物体作为缓存池的子物体
            transform.parent = bulletPool.transform;
            //将放入池中的物体失活
            gameObject.SetActive(false);
            //计时器设置为0
            timer_BulletPool = 0;
        }

        //让子弹飞
        transform.Translate(Vector3.forward * speed * Time.deltaTime,
        Space.Self);

        //就不需要在去销毁子弹物体，少让系统调用垃圾回收GC，可以提高效率
        //Destroy(this.gameObject, 5);
    }
}
}

```

- 如何在缓存池中放入不同的游戏物体
 - 设置一个缓存池管理类，在设置一个字典，将每个游戏物体放入对应的标签中，需要什么的时候就取这个标签字典下的游戏物体

缓存池存多个物体

```
/// <summary>
/// 缓存池 单例模式
/// </summary>
public class ObjectPool
{
    private Dictionary<string, List<GameObject>> pool = new Dictionary<string,
List<GameObject>>();
    private static ObjectPool instance;

    public static ObjectPool GetInstance()
    {
        if (instance == null)
            instance = new ObjectPool();
        return instance;
    }

    /// <summary>
    /// 指定缓存池名 得到其中的一个对象
    /// </summary>
    /// <param name="name">缓存池名</param>
    /// <returns></returns>
    public GameObject Pop(string name)
    {
        GameObject obj = null;
        //找下字典当中与没有这个线性表
        if (pool.ContainsKey(name))
        {
            //如果有这个线性表就查询表中有没有游戏物体
            if (pool[name].Count > 0)
            {
                //如果有游戏物体就将其取出来
                obj = pool[name][0];
                obj.SetActive(true);
                //取出来之后要把表中这个数据删除
                pool[name].RemoveAt(0);
                return obj;
            }
        }
        else
        {
            //如果没有这个线性表就构建这个线性表
            pool.Add(name, new List<GameObject>());
        }
        //如果没有这个游戏物体就将其实例化出来
        obj = GameObject.Instantiate<GameObject>(Resources.Load<GameObject>
(name));
        return obj;
    }
}
```

```

    /// <summary>
    /// 将使用完的对象 压入对象中
    /// </summary>
    /// <param name="name">缓存池名</param>
    /// <param name="obj">游戏物体</param>
    public void Push(string name, GameObject obj)
    {
        //在对应的线性表中添加这个游戏物体
        pool[name].Add(obj);
        obj.SetActive(false);
    }

    /// <summary>
    /// 清空缓存池
    /// </summary>
    public void Clear()
    {
        pool.Clear();
    }
}

```

- ==过场景时要清空缓存池==

3D数学

三角函数

基础公式

- 勾股定理： $a^2 + b^2 = c^2$
- 弧度与角度的换算：弧度 `rad`，边长 `l`，边长 `r`

$$\text{弧度} = \frac{\text{边长}}{\text{半径}} : rad = \frac{l}{r}$$

- $1 \text{ 弧度} = \frac{180}{\pi} \text{ 度} ; 1 \text{ rad} = \frac{180}{\pi} \text{ deg}$
- $1 \cdot \pi = 180 \text{ deg}$
- $1 \text{ deg} = \frac{1}{360} \text{ rad}$

在Unity中的单位换算

- `Mathf.Rad2Deg`：弧度转角度
- `Mathf.Deg2Rad`：角度转弧度

Unity中的三角函数方法

- `Mathf.Sin(float)`：正弦函数，传入一个弧度 `rad` 就可以计算正弦值，若是想传角度就可以使用 `Mathf` 中的单位换算进行换算
 - `float sinValue = Mathf.Sin(30 * Mathf.Deg2Rad);`：计算 $\sin 30^\circ$ 的值

- `Mathf.Cos(float)`：余弦函数，传入一个弧度 `rad` 就可以计算余弦值，若是想传角度就可以使用 `Mathf` 中的单位换算进行换算
 - `float cosValue = Mathf.Cos(30 * Mathf.Deg2Rad);`：计算 $\cos 30^\circ$ 的值
- `Mathf.Acos(float)`：反余弦三角函数，传入一个弧度值就可以得出这个夹角，返回的是弧度值，可以换算为角度值
 - `float includedAngle = Mathf.Acos(dotValue) * Mathf.Rad2Deg;`：计算这个夹角并转成度数

向量与矢量

- 向量没有位置
- 向量只有大小有方向的
- 单位向量就是向量除与模长： $\frac{\vec{a}}{|\vec{a}|}$

向量的加减

- 相对应坐标的加减
- 平行四边形法则、三角形法则

向量的点乘

- 模长：就是向量的长度
- 当有一个向量 \vec{a} ，和一个向量 \vec{b} 相点乘，即是 \vec{a} 的模长乘与 \vec{b} 在 \vec{a} 上面的投影， \vec{b} 在 \vec{a} 上面的投影即是 \vec{b} 的模长乘两个向量的夹角余弦值

$$\vec{a} \cdot \vec{b} = |\vec{a}| \times |\vec{b}| \times \cos \langle a, b \rangle$$

- 一般 $|\vec{a}|$ 和 $|\vec{b}|$ 都使用单位向量表示，即最后得到的其实就是两个向量夹角余弦值

$$\vec{a} \cdot \vec{b} = \cos \langle a, b \rangle$$

Unity中的点乘公式

- `Vector3.Dot(Vector3, Vector3)`：传入两个向量，就可以得出这两个向量的点乘值，即这两个向量的余弦值，一般要传入两个标准化的向量，即单位向量
- `Vector3.normalized`：向量标准化，即转化为一个单位向量
- ==通过点乘可以判断物体是不是在自己的角度范围内==

```
public GameObject cube1; // 自己这个物体
public GameObject cube2; // 敌方物体
{
    // 计算物体离自己的距离
    float distance = Vector3.Distance(cube1.transform.position,
    cube2.transform.position);
    // 计算物体与自己之间的向量与自己正方向向量的点乘值，计算点乘值的时候传入的必须是标准化的向量
    float dotValue = Vector3.Dot((cube2.transform.position -
    cube1.transform.position).normalized, cube1.transform.forward);
    // 通过点乘值使用反余弦函数就可以计算物体与自己的夹角
    float includedAngle = Mathf.Acos(dotValue) * Mathf.Rad2Deg;
    // 当夹角与自己小于30°且距离小于2的时候就在攻击范围内
    if (includedAngle < 30 && distance < 2)
```

```

{
    Debug.Log("在我的攻击范围内");
}
}

```

向量的叉乘

- 两个向量叉乘，得到一个新的向量，新向量跟原始两个向量都垂直，也就是得到这两个向量所确定平面的法向量⁵

Unity中的叉乘公式

- `Vector3.Cross(Vector3,Vector3)`：传进两个向量，返回的也是一个向量，计算的就是法向量⁵
- 通过叉乘可以判断物体在自己的左边还是右边
- 若是传入的第二个参数向量在第一个参数向量的左边，得到的法向量的 `y` 值小于 `0`，右边则大于 `0`

```

{
    //计算物体与自己的叉乘，返回的是一个向量
    Vector3 cross = Vector3.Cross(transform.forward * 10,
cube2.transform.position - transform.position);
    //判断向量的y值
    if (cross.y > 0)
    {
        Debug.Log("目标在右边");
    }
    else if (cross.y < 0)
    {
        Debug.Log("目标在左边");
    }
}

```

Vector3数学运算

- `Vector3.Lerp(Vector3,Vector3,float)`：==线性插值==，和数学类中的 `Lerp` 算法是一样的，做的行为就是从第一个点直线运动到第二个点，一般用于摄像机巡游，这样运动起来更自然，一般第一个参数传自己当前的位置，第二个参数传要去的位置
 - `Vector3.Lerp(transform.position, new Vector3(100, 0, 0), 0.01f)`：从自己当前位置做线性运动到(100,0,0)点
- `Vector3.Slerp(Vector3,Vector3,float)`：==球型插值==，返回的也是一个向量，做的行为就是从第一个点球形运动到第二个点
 - `Vector3.Slerp(transform.position, new Vector3(100, 0, 0), 0.01f)`：从自己当前位置做球形运动到(100,0,0)点
- `Vector3.SmoothDamp(Vector3,Vector3,ref Vector3,float,float)`：平滑阻尼运动，参数传进起点、终点、初始速度，时间（阻尼时间），最大速度（可不传），做的行为就是从初始速度开始加速到最大速度后匀速运动然后再近些减速运动

四元素

- 四元素是一个超复数，由一个实部三个虚部组成，三个虚部代表 **x**、**y**、**z** 轴，实部是角度
- 四元素旋转只能一个轴一个轴的旋转
- 了解下 Quaternion 构造函数的使用
 - `Quaternion(float x, float y, float z, float w)`：前三个参数传的就是要旋转的轴的值，且角度要除以2，即要旋转60°，则就是要计算30°的值，如下：

```
//绕z轴旋转了60°
Quaternion quaternion = new Quaternion(0, 0, Mathf.Sin(30 * Mathf.Deg2Rad),
Mathf.Cos(30 * Mathf.Deg2Rad));
cube1.transform.rotation = quaternion;
```

四元素常用API

- `Quaternion.Euler(Vector3 euler)`：传要旋转的轴的角度值，返回的是一个 `rotation` 值
 - 重载函数：`Quaternion.Euler(float x, float y, float z)`
 - `Quaternion.Euler(0,100,0)`：绕 **y** 轴转100°
- `Quaternion.AngleAxis(float angle, Vector3 axis)`：绕一个轴旋转多少度，传进一个要旋转的度数和方向
- `Quaternion.LookRotation(Vector3 forward)`：传进一个方向向量，就会使自己的 **z** 轴朝着这个方向旋转

```
//算物体与自己的方向向量，即算出一个方向
Vector3 dir = cube2.transform.position - transform.position;
//让自己朝着这个方向旋转
transform.rotation = Quaternion.LookRotation(dir);
```

- `Quaternion.Lerp(Quaternion a, Quaternion b, float t)`：与 `Vector3` 的 `Lerp` 类似，就是让自己的旋转更平滑，**a** 是当前的方向，**b** 是要旋转的方向
- 两个四元素相乘就是让两个旋转角度叠加

让摄像机随着鼠标移动

```
float x = Input.GetAxis("Mouse X");//检测鼠标的左右移动
Vector3 dir = transform.position - cube.transform.position;//得到目标物体与自己的向量
Quaternion q = Quaternion.AngleAxis(x, Vector3.up);//当鼠标操作时，让自己朝y轴旋转角度
Vector3 pos = cube.transform.position + q * dir;//四元素和向量相乘就是新的向量位置，加上物体的位置就得到自己旋转后的新位置
transform.position = pos;//让自己移动到新位置
```

- 摄像机全面巡游

```
x = Input.GetAxis("Mouse X");//检测鼠标的左右移动
Vector3 dir = transform.position - cube.transform.position;//得到目标物体与自己的向量
Quaternion q = Quaternion.AngleAxis(x, Vector3.up);//当鼠标操作时，让自己朝y轴旋转角度
Vector3 pos = cube.transform.position + q * dir;//四元素和向量相乘就是新的向量位置，加上物体的位置就得到自己旋转后的新位置
transform.position = pos;//让自己移动到新位置
```

```

y = Input.GetAxis("Mouse Y");
dir = transform.position - cube.transform.position;
Quaternion u = Quaternion.AngleAxis(y, Vector3.right);
pos = cube.transform.position + u * dir;
transform.position = pos;//让自己移动到新位置

transform.rotation = Quaternion.LookRotation(cube.transform.position -
transform.position);//让自己面朝向一直是这个方向向量，即看向物体

float scroll = Input.GetAxis("Mouse ScrollWheel");//检测鼠标滚轮
transform.Translate(Vector3.forward * scroll);//滚轮动的时候自己可以前后移动

```

发射圆形子弹

```

for (int i = 0; i < 12; i++)
{
    GameObject go = Instantiate(bullet);
    go.transform.position = transform.position;
    Vector3 dir = Quaternion.AngleAxis(30 * i, Vector3.up) *
transform.forward;//得到一个新向量
    go.transform.rotation = Quaternion.LookRotation(dir);
}

```

协程

- 因为Unity不支持线程，所以就有了协程这个概念，它是一种==“假”线程==，同步主线程执行，在主线程执行的时候开辟分支，执行完之后回到主线程
- 当前脚本失活了，协程也会照样执行，但是游戏物体失活了就不会执行

延迟函数

- `Destroy(GameObject, float)`：延迟几秒销毁
- `Invoke(string, float)`：传入函数名，延迟时间执行这个函数
- `CancelInvoke(string)`：传入函数名，取消这个 `Invoke` 执行方法，若是不传参数，就是取消所有 `Invoke`
- `InvokeRepeating(string, float, float)`：重复一定时间执行一个函数，传入一个函数名、延迟执行函数的时间、重复要执行这个函数的时间
- 当前脚本失活了，延迟函数也会照样执行，但是游戏物体失活了就不会执行

协程


```
IEnumerator Enumerator()//声明一个协程
{
    yield return new WaitForSeconds(3);//必须要用yield return返回
    //waitForSeconds()等待几秒
}
void Start()
{
    StartCoroutine("Enumerator");//启动协程
}
```

- 协程在执行的时候，主线程也在执行，相当于多线程

```
IEnumerator Enumerator()//声明一个协程
{
    Debug.Log(1)
    yield return new WaitForSeconds(3);//必须要用yield return返回
    //waitForSeconds()等待几秒
    Debug.Log(2);
}
void Start()
{
    Debug.Log(0)
    StartCoroutine("Enumerator");//启动协程
    Debug.Log(3);
}
```

- 上述代码输出结果是 0,1,3,2，当输出0的时候就进入协程中，执行打印1，然后要等待三秒延迟返回，当协程等待的时候主线程同时也在执行，执行打印3，然后三秒结束执行打印2
- 协程可以传参数，当声明协程的时候声明了参数，开启的时候就可以在名字后面加一个参数
`StartCoroutine("Enumerator", 参数);`

开启关闭协程

```
//开启关闭方式1
StartCoroutine("Enumerator");//启动协程
StopCoroutine("Enumerator");//关闭协程

//开启关闭方式2
StartCoroutine(Enumerator());//开启协程
StopCoroutine(StartCoroutine(Enumerator()));//关闭协程

//如果通过StopCoroutine(Enumerator());此方法是关闭不了的协程
//可以通过StopAllCoroutines来关闭、
StopAllCoroutines();
//但是StopAllCoroutines是关闭所有的协程，最好不要使用
```

协程的返回内容

```
yield return new WaitForSeconds(3);//等多少秒
yield return new WaitForFixedUpdate();//等固定步长0.02的秒数
yield return null;//等一帧
yield return new WaitForEndOfFrame();
yield break;//直接跳出协程
yield 1;//下一帧执行，无论填什么数字都是下一帧执行
```

异步加载

异步加载场景

```
AsyncOperation async = null;

IEnumerator Loading()
{
    async = SceneManager.LoadSceneAsync("TankBattle");//异步加载这个场景
    yield return async;//一直等待这个场景加载出来就退出这个协程
}

StartCoroutine("Loading");
```

- DontDestroyOnLoad(GameObject) :不销毁这个物体，切换场景时是会清空场景物体的，所以有些管理类挂载的物体是不应该销毁的，可以用这个函数

异步加载场景实例

```
//声明一个单例类，同一管理切换场景
using UnityEngine;
using System.Collections;
using UnityEngine.SceneManagement;

public delegate void LoadCallback(params object[] args);//一个委托变量
public class LoadScene : MonoBehaviour
{
    //单例模式
    public static LoadScene Instance;
    private void Awake()
    {
        Instance = this;
        DontDestroyOnLoad(this.gameObject);
    }

    // 统一切换场景接口，用来外部调用，参数分别为场景的名字，和加载场景完之后要做的方法
    public void LoadScene_1(string levelName, LoadCallback callback)
    {
        StartCoroutine(IELoadScene(levelName, callback));
    }

    private AsyncOperation async = null;//一个异步值，用来异步加载
    private IEnumerator IELoadScene(string levelName, LoadCallback callback)
    {
        //参数分别为场景的名字，和加载场景完之后要做的方法，使用委托来传
        async = SceneManager.LoadSceneAsync(levelName);
        yield return async;

        //加载完场景需要做的事情-1. 打开一个面板
        //假如加载的场景是主场景，要做的是打开玩家信息面板

        //总结：加载完场景需要干的事情不一样，加载什么场景,做什么事情我不管
        if (callback != null)
```

```
        {
            callBack();//如果委托变量不为空即是有这个方法，那就执行这个方法
        }
    }
}

-----

//如何外部调用
private void OnGUI()
{
    if(GUI.Button(new Rect(0, 0, 0, 0), "加载场景")){
        LoadScene.Instance.LoadScene_1("场景名", OpenPanel);
    }
}
private void OpenPanel(params object[] args)
{
    //打开一个面板或者自己声明方法来决定加载场景之后做什么
}
```

高级组件操作

Camera摄像机

摄像机静态属性

- `allCameras` : 场景中所有摄像机
- `allCamerasCount` : 场景中相机数量
- `main` : 第一个被标记为MainCamera的相机

Camera类常用方法

- 一般可以通过静态属性 `main` 点出这些方法
- `WorldToScreenPoint()` : 将世界坐标转换成屏幕的坐标，返回一个 `Vector3` 的值
- `ScreenToWorldPoint(float,float,float)` : 将屏幕坐标转世界坐标，第三个参数是指与摄像机的距离
- `Camera.main.ScreenPointToRay (Vector3)` : 从屏幕点发射一条射线
- `Camera.main.ViewportPointToRay(vector3)` : 从视口点发射一条射线
- `Camera.main.ScreenToViewportPoint(vector3)` : 屏幕坐标转视口坐标
- 屏幕坐标系原点在左下角
- GUI坐标系的原点在左上角

摄像机组件属性

- `Clear Flags` : 如何清除背景
 - `skybox` 天空盒
 - `Solid Color` 颜色填充
 - `Depth only` 只画该层，背景透明
 - `Don't Clear` 不移除，覆盖渲染
- `Culling Mask` : 选择性渲染部分场景，可以指定只渲染对应层级的对象
- `Projection` : 摄像机模式

- `Perspective` 透视模式
 - `Clipping Planes` 裁剪平面距离
 - `orthographic` 正交摄像机，一般用于2D游戏制作
 - `Size` 在正交模式下相机一半的尺寸
 - `Viewport Rect`：视口范围
 - 主要用于双摄像机游戏
 - 0~1 相当于宽高百分比
 - `Depth`：渲染顺序上的深度
 - `Redering path`：渲染路径
 - `Target Texture`：渲染纹理
 - 可以把摄像机画面渲染到一张图上，主要用于制作小地图
 - 在 `Project` 右键创建 `Render Texture`
 - `Occlusion Culling`：是否启用剔除遮挡
 - `Target Display`：用于哪个显示器，主要用来开发有多个屏幕的平台游戏
-

组件Line Renderer

- 划线组件，可以在 `Game` 视窗观察到划的线

LineRenderer类常用方法

- 这个类需要 `new` 出对象使用
- `SetPosition(int index, Vector3 position)`：设置一个点在这条线上，传入点的索引值和位置
- `positionCount`：设置线段的数量
- `startWidth`：开始位置
- `endWidth`：结束位置

组件的重要属性

- `Materials`：线的材质球
 - `Positions`：线段的点
 - `Use World Space`：是否使用世界坐标系
 - `width`：线段宽
 - `Loop`：是否重点起始自动相连
-

Rigidbody刚体组件

力Force

- 力是矢量，有大小有方向
- 牛顿第二定律 $F=m \cdot a$ ， a 是加速度， m 质量，被力的作用质量越大惯性越大
- 动量定理 $Ft=m \cdot v$ ，质量乘以速度就是动量/冲量的值，冲量 $Impulse$
- 动量守恒定理 $m1 \cdot v1=m2 \cdot v2$
- 以下这些是作用在刚体 `Rigidbody` 的组件上的

Rigidbody类

- `velocity` : 速度
- `AddForce(Vector3 force)` : 给物体添加一个力的作用, 传进的是这个力的方向, 以世界坐标为参考系
 - `AddForce(Vector3 force, ForceMode mode)` : 第二个参数是==力的模式== `ForceMode`, 有以下四个模式
 - `Force` : 只关注力的作用, 与质量相关, 因为 $F=m \cdot a$
 - `Impulse` : 关注的是冲量, 质量越大速度越小, $Ft=m \cdot v$
 - `VelocityChange` : 只关注速度的变化, 与质量就无关
 - `Acceleration` : 只关注加速度 a , 与质量无关
- `AddRelativeForce(Vector3 force)` : 相对于自身坐标系的力, 功能与 `AddForce` 一样
- `AddTorque(Vector3 torque)` : 给物体一个扭力, 传一个方向物体就会绕这个轴旋转, 就相当于旁边有一个力在推动这个物体
- `AddRelativeTorque()` : 与 `AddRelativeForce` 类似
- `AddForceAtPosition(Vector3 force, Vector3 position)` : 给一个点的表面添加一个力的作用, 使周围的物体被这个力作用, 类似爆炸的时候

根据鼠标的位置决定子弹方向

```
{
    //0.实例化预制体, 给一个初始位置
    GameObject go = Instantiate(bullet);
    go.transform.position = bulletPos.transform.position;

    //1.获取物体的刚体
    Rigidbody rigidbody = go.GetComponent<Rigidbody>();

    //2.获取鼠标在屏幕上的坐标点并将屏幕坐标转化为世界坐标
    Vector3 pos = Camera.main.ScreenToWorldPoint
        (new Vector3(Input.mousePosition.x, Input.mousePosition.y, 20));

    //3.根据鼠标点的目标点和子弹的起始点的到一个向量作为物体的方向
    Vector3 dir = pos - bulletPos.transform.position;

    //4.将子弹的方向赋值给子弹刚体作为速度方向
    rigidbody.velocity = dir.normalized * 20;
}
```

- `Input.mousePosition.x/y/z` : 获取鼠标的坐标

射线检测Physics

Ray射线类

- 需要实例化 `new` 一个 `Ray`
- `Ray` 有一个构造函数, 初始化的时候需要填入初始值, `Ray(Vector3 origin, Vector3 direction)`, 两个参数一个原点一个方向就构成这个射线
- `Ray ray = Camera.main.ScreenPointToRay()` : 将屏幕上的点形成一条射线

Physics射线检测类

- 这个类中多为静态方法，可以直接通过类名点方法使用
- `Raycast(Ray ray)`：传入一条射线，当射线触碰到==碰撞体==就会有一个返回值，返回的是布尔值
 - `Raycast(Ray ray, out RaycastHit hitInfo)`：传入一条射线和 `RaycastHit` 变量，当触碰到这个碰撞体的时候，就会返回一个布尔值和这个碰撞点的所有信息，信息包括距离、位置、刚体、碰撞体、`transform` 信息、法向量等等
 - `Raycast(Ray ray, float maxDistance, int layerMask)`：三个参数，一条射线，一个最远距离和一个 layer 值
- **layer 值**：这个值就是控件上 Layer 对应层的 ID 的2次方，就是使用十进制数控制层级，例如这个层是第0层，就是 $\text{int}\ \text{space}\ \text{layer}=2^0=1$ ，第8层就是 $\text{int}\ \text{space}\ \text{layer}=2^8=256$ ，若是同时要同时控制两个层，将这两个值相加就同时访问到了这两个层
 - `<<` 左移符号 `>>` 右移符号：将1左移8位的值 $1<<8 == 256$
- 如何多层检测：`(1 << LayerMask.NameToLayer(层名)) + (1 << LayerMask.NameToLayer(层名)) + ...`
 - 本质：2进制位运算
- `Physics.RaycastAll(Ray ray, float maxDistance, int layerMask)`：返回的是一个 `RaycastHit` 的数组，可以一条射线检测多个碰撞体
- `RaycastHit` 返回信息
 - `collider` 碰撞到的对象碰撞信息
 - `transform` 碰撞到的对象的transform信息
 - `point` 碰撞产生的位置点
 - `normal` 碰撞到的面的法向量
 - `rigidbody` 碰撞到的对象的刚体信息
 - `distance` 碰撞的对象离自己的距离

Physics球形搜索

- `OverlapSphere(Vector3 position, float radius, int layerMask)`：范围型搜索，传入一个点、半径，层级值，就可以检测这个点的碰撞体周围范围内的所有碰撞体，返回的是一个碰撞体的数组

球形搜索实例

```
collider[] colliders = Physics.OverlapSphere(transform.position, 100, 8);
for (int i = 0; i < colliders.Length; i++)
{
    Debug.Log(colliders[i].transform.name);
}
```

图片

Texture属性

- `TextureType`：图片格式，重要的就是 `Default`、`Sprite`
 - `Default`：默认图片类型
 - `Normal map`：法向贴图

- `Editor GUI` : 编辑器使用图片
 - `Sprite` : 精灵图片格式
 - `Cursor` : 鼠标贴图
 - `Cookie` : 灯光 Cookie 贴图
 - `Lightmap` : 灯光贴图
 - `Single Channel` : 单通道贴图
 - 其他的查询API
-

SpriteRenderer 2D图片渲染组件

- `Sprite` 图元资源
 - `Color` 颜色
 - `Flip` 翻转
 - `Material` 材质球 一般不去更改 除非有特殊需求
 - `DrawMode` 绘制模式
 - `Sliced9` 宫图模式
 - `Tiled` 平铺模式
 - `Simple` 拉伸模式
 - `Sorting Layer` 排序层
 - `Order in Layer` 属于哪一层
 - `Mask Interaction` 遮罩模式
-

效应器

- 一系列 `Effector 2D` 的组件，都是模仿磁铁、风力、平台、浮力等效果，查询API
-

修改代码模板

- VS中 : D:\VSIDE\Common7\IDE\Extensions\Microsoft\Visual Studio Tools for Unity\ItemTemplates\CSharp\1033
 - Unity中 : D:\Program Files\Unity2017\Editor\Data\Resources\ScriptTemplates
-

遇到问题解答

游戏的单例模式

- 在 `Awake` 中直接 `instance=this`，但这个是在要在挂载的脚本上是这样写的，因为会执行 `Awake` 函数，如果不挂载就按正常单例模式写

```
public static GeneratingEnemies Instance;//全局唯一实例
private void Awake()
{
    Instance=this;
}
```

这样可以直接使用类名点出这个实例再调用这个类的方法，一般放在管理类中，即使这个脚本不挂载在组件上也可以进行调用

为什么子弹会跟着特效乱跑

- 其实是子弹加了刚体，受到了重力的作用，一般只要其中一方有刚体就能发生触碰，不需要两方都加刚体

为什么播放了音效特效就出不来这个函数了

- 因为游戏特效也是一个游戏物体，特效只是它身上的脚本文件，要找这个物体再去获得它身上的特效脚本，然后就可以播放特效

```
private GameObject boom;
{
    boom = Instantiate<GameObject>(Resources.Load<GameObject>
("FX/Explosion_Smoke_FX"));
    boom.GetComponent<ParticleSystem>().Play();
}
```

动画系统

- 现在Unity的动画系统有两套，即老版的 `Animation` 和新版的动画状态机 `Animator`

老版的动画系统Animation

- 首先给物体添加组件 `Animation`，然后打开 `Animation` 面板，点击 `Create`，就可以创建一个动画了
- 动画的改变其实就是改变物体身上组件的在当前帧的一个状态，点击 `Add Property` 就可以选择要控制的组件
- `Animation` 面板上的参数信息
 - `New Animation` 这个按钮：创建新的动画，也可以选择要编辑的动画
 - `Samples`：动画帧数，指这个动画需要一秒多少帧，旁边的按钮分别是 `Add Keyframe`，`Add event`
 - `Add Keyframe`：添加关键帧
 - `Add event`：添加帧事件
 - `Preview`：动画预览，旁边的按钮分别是回到第一帧、回到前一帧、暂停/播放、到后一帧、到最后一帧
 - `Curves`：动画播放的曲线

动画上的几个参数

- `Wrap Mode` : 播放的状态
 - `Once` : 只播放一次
 - `Loop` : 循环播放
 - `Ping Pong` : 类似乒乓球效果

帧事件

- 当动画添加了帧事件之后,就是在播放完这一帧动画之后必须要实现的逻辑
- 添加帧事件之后,要设置帧事件的名字 `Function`,然后在脚本中要实现这个相同名字的方法

```
//例如帧事件名字 DeathEvent  
//然后在对应调用动画的脚本里实现方法  
public void DeathEvent()  
{  
    //要做的逻辑  
}
```

Animation组件

- `Animation` : 默认动画
- `Animations` : 动画组
 - `Size` : 动画组的个数,通过代码进行切换动画
- `Play Automatically` : 是否游戏启动时播放默认动画

Animation组件的API

- `bool Play()` : 播放动画,连续两次播放的是同一个动画的时候,会等上一个动画播放完毕再进行播放,如果连续两次播放的不是一个动画,会立即停止当前动画,直接播放下一个动画
 - `Play(string animation)` : 传入一个动画的名字指定播放
- `AnimationState PlayQueued(string animation)` : 队列播放,会将当前播放的动画放入队列中,即播放的是两个不同动画的时候,将动画一个一个放入队列中,当上一个动画播放完再进行下一个动画的播放
- `AnimationState CrossFade(string animation, float fadeLength)` : 两个动画播放之间有个渐入渐出的效果,当第一个动画将要播放完的时候第二个播放开始渐渐播放,第二个参数就是渐入渐出效果的时间,即两个动画切换的时间
- `AnimationState CrossFadeQueued(string animation, float fadeLength)` : 渐入渐出效果的队列播放
- `Stop()` : 停止所有在播放的动画
- `isPlaying()` : 是否在播放某个动画

动画模型

- 动画模型一般指的是直接提交的一个动画模型文件或者是一个模型上已经绑定有动画的模型文件,文件下有包括了动画文件和替身 `Avatar` 文件

动画模型上的参数

Model

- **Scale Factor**：模型文件的比例，调节大小作用

Rig

- **Animation Type**：动画类型
 - **None**：没有类型
 - **Legacy**：老版的动画类型
 - **Generic**：通用动画类型
 - **Humanoid**：人类特有类型，只有使用这个才能使用替身系统 Avatar
- **Avatar Definition**：定义一个替身文件
 - 点击 **Configure** 可以查看这个替身匹配效果，进行调整

Animation

- 这上面重点的就是动画帧匹配和帧事件
- **Start**：起始帧数
- **End**：结束帧数
- **Loop Time**：是否循环播放
- **Root Transform Rotation**：方向匹配，这个人物模型和这个动画在方向上是否匹配
- **Root Transform Position(Y)**：动画与人物在Y轴上是否匹配
- **Root Transform Position(XZ)**：动画与人物在X和Z轴上是否匹配
- **Events帧事件**：和老版动画系统添加帧事件效果一样，添加方式也一样，在动画上添加帧事件就是让动画在执行到这一帧的时候可以执行什么样的事件

Avatar替身系统

- 替身系统值得是每个人类模型一般都会带一个 **Avatar** 文件，用来匹配动画，当这个动画是按一个骨骼来做的时候，只要是骨骼相同的，那么无论是什么模型，都可以播放和使用这个动画
- 点击这个替身文件，点击 **Configure Avatar**，可以进入配置替身效果
 - 实线圆圈，完全匹配
 - 虚线圆圈，不完全匹配
 - 虚线，完全不匹配

新版动画系统Animator

- 新版的动画系统是一个动画状态机，可以配置各个动画之间的切换条件
- 在 **Project** 面板右键创建一个状态机 **Animator Controller**，双击就可以进入 **Animator** 面板

Animator面板

- **Layers**：动画层级，可以新建一个动画和修改名字
- **Parameters**：参数类型，这个参数决定了动画之间的触发条件，参数类型有四种，点击加号就可以创建对应的类型
 - **Bool** 型
 - **Float** 型
 - **Int** 型
 - **Trigger** 型
- **Entry**：起始状态，创建的第一个动画会链接这一个，就是指这个状态机开始的时候播放的动画，即默认动画

- **Any State**：任何状态下都可以转，这里链接的动画一般是一个死亡动画，因为死亡动画在任何状态下都有可能播放，那么这个动画链接在这上面的时候就可以指在任何状态下只要触发参数条件就可以转到这个链接的动画上面
- **Exit**：
- **可以在空白处点击右键创建新动画 Create State->Empty，或者直接将动画拖动进来**
 - 点击这个新建 **Empty**，在 **Inspector** 面板可以设置参数，可以编辑名字和标签 **Tag**
 - **Mation**：赋值动画文件
 - **Speed**：播放速度
 - **Transition**：过渡条件
 - **Solo**：优先过渡，即两个动画过渡参数一样的时候，这里优先过渡
 - **Mute**：禁止过渡，禁止任何动画转到这个动画来
- **可以右键点击新建 Empty，点击 Make Transition，创建一个过渡效果，这个是用来连接两个动画，创建过渡效果**
 - 这里有两个重要的地方，一个是设置过渡的平滑度，一个设置过渡触发参数
 - **Has Exit Time**：是否播放完上个动画后再进行切换下个动画
 - **Setting**：设置参数
 - **ExitTime** 切换上一动作的退出时间
 - **Fixed Duration/Transtion Duration** 动作切换时间长度
 - **Transtion offset** 下一动作所在上一动作比例
 - **Interruption sources** 打断动画切换来源
 - **Can Transition To** 是否一直转到这个动画
 - 平滑度就类似老版动画系统的 **CrossFade** 函数，就是两个动画切换之间有个渐入渐出的效果，但一般默认就行
 - **Conditions**：条件参数，这里就可以设置刚刚在 **Parameters** 上设置的参数，设置参数，然后设置参数条件，如布尔值就是当为真的时候就触发这条过渡线等等，其他线一样
- **可以右键点击空白处，点击 Create Sub-State Machine：新建一个动画层，其实就相当于一个嵌套效果，从一个动画进入另一个动画层级，一般做一个技能里包含多个动画连招的时候，就这样做**

Animator组件

- **Controller**：赋值一个状态机
- **Avatar**：赋值一个替身文件
- **Apply Root Motion**：去除动画位移，一般有些动画会带位移效果，不勾选就去掉了位移功能
- **Update Mode**：更新模式，选择更新的模式
 - **Normal**：
 - **Animate Physics**：物理更新模式
 - **Unscaled Time**：
- **Culling Mode**：剔除模式
 - **Always Animate**：总是渲染动画，无论是否在镜头之中
 - **Cull Update Transform**：当不在镜头中时就停止它的动画效果
 - **Cull Completely**：

Animator类中常用API

- `void SetBool(string name, bool value)`：设置参数条件，字符串传布尔参数名字，后面的值传参数数据，然后调用这条过渡线，例如设置了一个 `Move` 参数为真 `true` 的时候就从待机状态转到前行状态，调用这个方法的时候就从待机动画转到前行动画
 - `SetFloat()`、`SetIntger()`、`SetTrigger()` 这几个方法与上面是一样的，只是根据参数类型选择不同方法
 - `SetIntger()`：这个一般可以用来设置技能动画的播放，当传入的参数整型值为多少的时候，就播放对应的技能动画
 - `SetTrigger(string name)`：如果是 `Trigger` 类型的参数这里只需要传一个参数，因为 `Trigger` 类型的过度，是当一个动画转到另一个动画的时候，触发条件只播放一次这个动画就马上按返回过度线返回了
- `bool GetBool(string name)`：得到这个参数当前的参数值
- `AnimatorStateInfo GetCurrentAnimatorStateInfo(int layerIndex)`：得到一个动画层，可以赋值给一个 `AnimatorStateInfo` 的类对象，0就是第一层，1就是第二层等

AnimatorStateInfo类

- 有关当前动画层的信息，需要赋值一个动画层
- `bool IsName(string name)`：判断这个动画是不是当前正在播放，一般在多层动画的时候用这个来判断，单层的时候直接 `GetBool`、`GetIntger` 等等即可，输入的字符串格式为
 - 动画层名.动画名

动画融合树

- 一般写实类的游戏使用复杂动作的情况下融合树会频繁用到

1D动画融合树

- 融合树的作用就是将几个动画合在一起，通过阈值进行相互之间的切换，例如向左跑的时候要有个向左的动作，就可以设置阈值判断当阈值等多少的时候是播放向左跑的动作还是向右跑的动作还是向前跑的动作
- 在状态机空白处右键点击 `Create State -> From New Blend Tree`，双击这个新创建的融合树，就可以进入到融合树层进行设置
- `Blend Type`，设置融合树类型，是1D还是2D
- `Parameter` 控制当前混合树的参数
- `Motion` 设置要融合动画个数和阈值
- `Automate Threshold`：自动计算，通过动画片段的动画位移来计算的阈值，我们一般使用自己设置的

2D动画融合树

- 和建立1D融合树一样，在 `Type` 处改成2D，2D融合树有三种不同的设置效果
 - `2Dsimple Direction`：在同一方向上不能有多动画片段
 - `2DFreeform Directional`：在同一方向上可以有多个动画片段
 - `2Dfreeform Cartesian`：不是在方向上操作的动画片段
- 1D融合树一般必须先设置默认动画，再从默认动画切换到融合树，而2D融合树可以直接设置为默认的动画，一般添加5个动画四个方向，中间的可以设置为默认动画，这样可以做一个向前向左向后向右的动画效果。
- 需要设置两个参数X、Y，因为2D融合树是将一个直角坐标的值转化成阈值来控制

融合树的嵌套

- 可以在融合树下再链接一个融合树，这样就形成了融合树的嵌套

动画遮罩AvatarMask

- 用来限制关节的运动
- 在 Project 面板右键创建一个 Avatar Mask
- 在设置面板 Humanoid 左键点击绿色的地方，就可以设置限制动画关节不运动的部位
- 在状态机 Layers 新建一个动画层，点击动画层旁边的齿轮，局可以将 Mask 赋值给这个动画层
- Mask 遮罩的作用就是在播放一个动画的时候可以再同时播另一个动画，即做到边移动边射击这种效果
 - 例如有一个摇手的动画，而这个动画手在动的的时候，其他部位也在动，这个时候直接设置一个遮罩层，除了右手部位其他部位都限制其运动，这个时候这个动画层的所有动画都只有右手能动

Layers动画层齿轮参数

- Weight : 权重，设置这个动画层的权重，如果权重太低的时候，这个层的动画播放就不完全
- Blending : 效果
 - Override : 覆盖，把这个动画层覆盖上一个动画层
 - Additive : 融合，和上一个动画叠加在一起
- Sync : 同步层，打开后两个层所有状态同步
- IK Pass : 打开 IK 效果

IK动画逆向动画

- 逆向动画指先指定一个目标点，然后再带动四肢的运动，例如去拿一个球的时候，先取到这个球的点，然后再让手部跟着球去运动
- 一般做AR游戏等，平时很少用到
- 在 AvatarMask 处也可以限制 IK 的运动
- 当要设置一个逆向动画的时候，要把 IK Pass 打开
- 然后将设置 Ik 的参数写在 OnAnimatorIK 函数中，就跟 GUI 要写在 OnGUI 中一样

Animator类中设置IK的方法

- SetIKPositionWeight(AvatarIKGoal goal, float value) : 打开 IK 的位置权重，value 值为0的时候权重关闭，就不执行 IK，为1时打开权重
 - AvatarIKGoal 是个枚举变量，里面四个参数对应四肢
- SetIKRotationWeight(AvatarIKGoal goal, float value) : 打开 IK 的方向权重，一般位置权重和方向权重都要打开
- SetIKPosition(AvatarIKGoal goal, Vector3 goalPosition) : 设置 IK 目标点的位置
- SetIKRotation(AvatarIKGoal goal, Quaternion goalRotation) : 设置 IK 目标点的方向
- 一般这四个函数设置完，这个部位就可以跟着这个目标点运动

- `MatchTarget(Vector3 matchPosition, Quaternion matchRotation, AvatarTarget targetBodyPart, MatchTargetWeightMask weightMask, float startNormalizedTime, float targetNormalizedTime)`
 - 这个函数可以直接设置匹配，就不需要上面四个一起设置
 - `matchPosition`：匹配目标点的位置
 - `matchRotation`：匹配目标点的方向
 - `targetBodyPart`：要匹配的身体部位，`AvatarTarget` 是个枚举值，对应全身的部位
 - `weightMask`：匹配的权重值
 - `MatchTargetWeightMask(Vector3 positionXYZweight, float rotationweight)`：位置匹配的权重值，第二个参数是方向匹配的权重值
 - `startNormalizedTime`：匹配的开始时间
 - `targetNormalizedTime`：匹配的结束时间

爬墙功能

```
public class IKAnimatorJump : MonoBehaviour
{
    public GameObject target; // 目标点
    private Animator myAnimator;
    private AnimatorStateInfo stateInfo;

    private void Awake()
    {
        myAnimator = gameObject.GetComponent<Animator>();
    }
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.A))
        {
            myAnimator.SetBool("Jump", true); // 当按A键的时候就播放爬墙动画
        }
        stateInfo = myAnimator.GetCurrentAnimatorStateInfo(0);
        if (stateInfo.IsName("Base Layer.Jump Up")) // 当这个动画在播放的时候才开始匹配
        {
            // 设置匹配参数
            myAnimator.MatchTarget(target.transform.position,
            target.transform.rotation, AvatarTarget.RightHand,
            new MatchTargetWeightMask(new Vector3(1, 1, 1), 1), 0.19f,
            0.3f);
        }
    }
}
```

寻路系统

- 寻路系统使用的是A星算法，这个在各个博客都可以看到这个算法，可以去百度
- 在Unity中已经封装好了寻路系统 `Navigation`，可以自动的完成寻路

烘焙路径

- 首先要将所有障碍物设置成静态的属性，静态属性可以在物体的 Inspector 面板勾选
- 在菜单栏 window 中打开 Navigation 面板，就可以设置烘焙路径的参数了

Navigation面板

Bake面板

- Agent Radius：不可行区域的半径，根据障碍物周边来算这个半径
- Agent Height：不可行区域的高，若是有一个通道低于这个值，这判断这个通道为不可行的区域
- Max Slope：可行的最大角度，即能通过坡的最大角度，超过这个角度这个坡就判断是上不去的
- Step Height：可以走的台阶的最大高度
- Drop Height：可以跳下去的平台的最大高度
- Jump Distance：可以跳过障碍物的距离，就是指跳过一条沟，这个沟的宽
- 设置好参数后点击 Bake 就可以烘焙路径了

Areas面板

- 寻路算法是有计算消耗的，电脑会计算消耗最少的且最近的路
- 这个面板就是这事消耗层，后面的数字就是消耗

Object面板

- 设置好消耗层之后，选中障碍物，就可以在这个面板下的 Navigation Area 选择层
- Generate OffMeshLinks：打开 OffMeshLink 的静态属性，跟你单独在游戏物体上打开静态属性是一样的
- Navigation Static：也是打开静态属性

Agents 代理模板

- 设置烘焙的模板，可以在 Nav Mesh Agent 中 Agent Type 里设置

off Mesh Link组件

- 离网连接组件
- 可以让物体判断前面是沟渠还是阶梯等等，可以跳跃过

Nav Mesh Agent组件

- 寻路系统要搭配这个组件使用，给要寻路的对象添加一个 Nav Mesh Agent 的组件
- 重要的参数
 - Speed：最大的速度，物体会慢慢加速到最大速度
 - Angular Speed：旋转的角速度
 - Acceleration：加速度
 - Stopping Distance：到目标点距离多少停下
 - Obstacle Avoidance 下的参数就是类似碰撞体的参数，自己设置即可
 - Priority：设置权重
 - Auto Traversal Off Mesh Link：是否支持 off Mesh Link

- `Auto Repath`：是否自动重新规划路径，若是中途障碍物或者目标改变，就可以重新规划路径
- 这个组件需要引入一个新的命名空间 `using UnityEngine.AI;`
- 常用API
 - `bool SetDestination(Vector3 target)`：设置目标点，物体就会向这个目标点移动

Nav Mesh Obstacle组件

- 这个是设置临时障碍组件，挂载这个组件的不需要烘焙，可以当个临时的障碍物，在游戏运行的时候如果在场景上就可以阻挡物体，使用代码控制消失之后这个障碍物就消失了，这个区域就可以过去。

文件夹

Unity的特殊文件夹

- `Resources`：资源文件夹，可以通过路径直接用 API `Resources` 进行加载；工程文件打包时，不在特殊文件夹内时，跟其他文件都没有依赖关系时，此资源不会打进包；但是 `Resources` 文件夹内所有资源，无论是否跟其他资源有依赖关系，都会打进包；为了减小包体大小，`Resources` 不能乱放资源，打包时会被压缩加密，只读属性
 - `StandardAssets`：默认文件夹，此文件夹内资源会被优先编译，一般 `U3D` 自带资源都在这个文件夹下
 - `Plugins`：`Plugins` 文件夹用来放 `native` 插件。它们会被自动包含进 `build` 中去。注意这个文件夹只能是 `Assets` 文件夹的直接子目录。跟 `Standard Assets` 一样，这里的脚本会更早的编译，允许它们被之外的脚本访问
 - 在安卓平台还有个 `Android` 文件夹
 - 在苹果平台还有个 `IOS` 文件夹
 - 在 `Windows` 平台下，`native` 插件是 `dll` 文件
 - `Mac OS X` 下，是 `bundle` 文件
 - `Linux` 下，是 `.so` 文件
 - `Editor`：以 `Editor` 命名的文件夹允许其中的脚本访问 `Unity Editor` 的 API。如果脚本中使用了在 `UnityEditor` 命名空间中的类或方法，它必须被放在名为 `Editor` 的文件夹中。`Editor` 文件夹中的脚本不会在 `build` 时被包含。在项目中可以有多个 `Editor` 文件夹，打包的时候不会被打包出去，开发 `U3D` 编辑器时，编辑器脚本放在该文件夹中

注意：如果在普通的文件夹下，`Editor` 文件夹可以处于目录的任何层级。
 - `StreamingAssets`：`StreamingAssets` 文件夹也是一个==只读==的文件夹，但是它和 `Resources` 有点区别，`Resources` 文件夹下的资源会进行一次压缩，而且也==会加密==，不使用特殊办法是拿不到原始资源的。但是 `StreamingAssets` 文件夹就不一样了，它下面的所有资源不会被加密，然后是原封不动的打包到发布包中，这样很容易就拿到里面的文件。所以 `StreamingAssets` 适合放一些二进制文件，而 `Resources` 更适合放一些 `GameObject` 和 `Object` 文件
 - `Application.persistentDataPath`：固定数据路径
 - `Application.DataPath`：返回 `Assets` 文件夹的路径
-

文件夹的操作

UGUI

UI基础

常见的UI类型

- 环抱式：将主要角色放在中间，UI按钮环绕在主人物周围，例如崩坏的主界面，荣耀的游戏界面
- 弹框式：弹出任务框这些，但并未完全遮盖原界面类型的，原界面会有点暗淡
- 全屏覆盖：完全覆盖原有界面的UI类型
- 超全屏：一般塔防地图这些，可以一直拖到地图，好像没有边界一样
- 3D：炉石的UI基本都是3D类型

常见分辨率

- 13:6全面屏：2340*1080
- 18:9全面屏：2010*1080
- 4:3Pad屏：1024*768
- 16:9常规：1920*1080

Canvas

- 所有的UI都必须放在Canvas下，Canvas相当于一个画布，让所有UI在上面绘制
- UI控件在 Canvas 的显示顺序
 - 下方物体在上方物体的前面显示
 - 子物体在父物体的前面显示

Canvas组件

- **Render Mode**：渲染模式
 - **overLay**：覆盖模式，相当于将UI直接通过画布绘制在摄像机的镜头上，所以这个模式的UI在所有场景物体前面
 - **Pixel Perfect**：高像素清晰度渲染，一般像素风游戏可以让像素颗粒感更强
 - **Sort Order**：多 Canvas 时，控制 Canvas 显示顺序，值越大越在前方显示
 - **Target Display**：显示的哪个摄像机
 - **Camera**：相机显示，直接通过相机照到画布 Canvas 上
 - **Render Camera**：要赋值一个摄像机物体
 - **Plane Distance**：Canvas 到相机的距离，多 Canvas 的时候，距离近的会先渲染覆盖距离远的
 - **Sorting Layer**：排序层，粒子特效的渲染形式一般与UI不同，UI总会覆盖在粒子上，所以就要修改粒子渲染先后顺序的，在Unity右上角的Layers可以设置 **Sorting Layer**
 - 越靠下的层越在相机面前显示

- 相机模式下可以选择UI的渲染排序层
- 粒子特效选择层在最下面的一个 `Render` 里有选择
- `Order in Layer` : 值越大渲染靠前
- `World Space` : 世界模式, `Canvas` 以3D物体显示在场景中, 可以改变它的方向, 大小, 但是前两个不行

Canvas Scaler组件

- 控制UI对屏幕分辨率的缩放, 当分辨率改变的时候能自适应缩放
- `UI Scale Mode` : 拉伸类型
 - `Constant Pixel Size` : 静态模式
 - `Scale with Screen Size` : 屏幕像素模式, 一般都用这个模式
- `Reference Resolutio` : 参考分辨率, 这个UI画布画的UI根据一开始UI设计师设计的整体图的尺寸, 然后就会自适应进行缩放
- `Screen Match Mode` : 适配模式
 - 宽高自适应
 - 完全的自适应
 - 第三个不常用
- `Reference Pixels` : 与像素的比例, 多少像素等于1米

UI基础组件

Rect Transform位置组件

- `Pivot` : 轴点, 设置这个控件中间的蓝色点
- `Stretch` : 锚点, 这个控件相对于 `Canvas` 的中心点
- 其他的都与 `Transform` 组件差不多, 快捷键 `Shift`、`Alt` 快捷设置锚点和轴点

Text文本组件

- `Line Spacing` : 行间距
- `Rich Text` : 是否开启富文本

```
加粗: <b>文字</b>
斜体: <i>文字</i>
大小: <size=字号>文字</size>
修改颜色: <color=颜色名>文字</color>
           <color=#颜色数(十六进制)>文字</color>
```

- `Horizontal` : 水平溢出
- `Vertical` : 垂直溢出
- `Best Fit` : 字号自适应, 当 `Canvas` 放大缩小的时候字号自适应

还可以添加两个效果组件

- `Shadow` : 阴影
- `Outline` : 外发光

Image图片组件

- **Image Type** : 图片类型
 - **Simple** : 普通模式, 一般图标使用
 - **Preserve Aspect** : 保持图片的宽高比, 高度或宽度自适应
 - **Set Native Size** : 可以快速恢复美术提供的图片原始像素尺寸
 - **Sliced** : 裁剪模式
 - **Fill Center** : 是否使用 Sprit 图的裁剪效果
 - **Tiled** : 瓦片模式, 放大图片的时候, 会把当个图平铺, 一般做无缝连接的图使用
 - **Filled** : 填充模式, 一般可以做进度条, 血条等
 - **Fill Method** : 填充效果
 - **Fill Origin** : 填充起始点
 - **Fill Amount** : 填充百分比, 用作血条进度条的关键
 - **Clockwise** : 顺时针还是逆时针填充
 - **Preserve Aspect** : 等比缩放

Raw Image组件

- 原始图片组件: 可以显示精灵或纹理, 功能相对于Image少, 所以性能更好, 可以控制UV的偏移, 来显示精灵的一部分

图集制作

- 在项目面板新建一个图集 **Sprite Atlas**
- 把需要制作图集的图片拖到 **Objects for Packeng** 下
- **Allow Rotation** 和 **Tight Packing** 不要打开
- 一般做一个页面就把一个页面的东西打成一个图集, 会减少 **Batches** (绘制调用数)的数值, 提高效率

Mask遮罩组件

- 可以遮盖图片显示, 阿尔法值为0不显示, 为1的地方显示, 一般可做头像的显示

UI复合组件与交互

- 交互的三个必然条件
 - 场景中是否有 **EventSystem**, 且这个物体上存在有 **Event System** 的脚本
 - Canvas上挂有 **Graphic Raycaster** 射线发射器
 - 控件上有勾选 **Raycast Target** 射线接收器
- 事件监测一般都是子物体会反馈都父物体, 如果父物体没有打开射线接收, 但是子物体有的话, 点击子物体就会反馈父物体, UI的射线检测会依次检测**对象树**下的所有元素, 有任何射线检测成功, 则会反馈
 - 一般控件或者图片都是方形的, 如果要做异形按钮, 就要用一个个的方形图片打开射线检测, 然后设置成异形图片的子物体, 然后去拼成异形

Button按钮组件

- **Interactable** : 按钮是否可以交互, 如果关闭之后点击按钮就没有任何反应
- **Transition** : 按钮的交互效果
 - **Color Tint** : 交互使用颜色

- `Sprite Swap` : 交互的时候使用图片, 就是点击之后会改变图片
 - `Animation` : 交互使用动画, 即点击后会有动画效果
- `Normal` : 正常状态下的样子
- `Highlighted` : 高亮时候的样子, 就是鼠标放在上面反馈的样子
- `Pressed` : 点击时候反馈的样子
- `Disabled` : 没有开启控件交互时候显示的样子
- `Color Multiplier` : 颜色倍数 (RGBA分量乘以倍数)
- `Fade Duration` : 切换动画持续时间
- `Navigation` : 是否开启键盘导航 (`Visualize` 开启后, 能够看到导航线)
 - 横向移动
 - 纵向移动
 - 自动匹配移动
 - 指定上下左右移动
 - 手机游戏可以关闭导航
- `Button` 添加事件 `On Click`
 - `On Click` 实际上是一个委托, 当我们写方法添加到这的时候, 就相当于这个委托注册了我们添加的函数, 然后就执行了这个函数
 - 写一个脚本, 里面写一个方法, 可以挂载在 `Button` 的父对象上, 用加号添加事件, 将挂载脚本的对象拖到这里, 然后再去对应的按钮添加事件选择对应的函数, 当这个按钮被按下的时候, 就会执行这个函数

异形按钮代码设置

```
public Image img;

{
    img.alphaHitTestMinimumThreshold = 0.1f;
}
```

- 使用这句代码, 就可以解决异形按钮, 只有有颜色的地方会被触发

Button添加事件代码向

- 代码添加事件有几个步骤, 一般代码都挂载在 `Panel` 上, 然后其他的控件都是它的子物体
 - 找到物体, 使用 `transform.Find()` 查找子物体, 如果想查找子物体的子物体可以传路径进去可以
 - 获取 `Button` 组件
 - 属性 `onClick` 可以点出方法 `AddListener()` 添加事件, 传进的就是方法
 - `button.onClick.AddListener()`
- `RemoveListener()` : 删除事件
- `Invoke()` : 触发调用的所有回调函数

Toggle单选控件

- 其余参数都与 `Button` 组件的参数是一样的, 以下是不同的以下参数
- `Is On` : 用于判断当前 `Toggle` 是否为开启状态
- `Toggle Transition` : 勾切换的动画效果

- `Graphic` : 钩钩的图
- `Group` : 如果 `Toggle` 构成组, 则需要将 `ToggleGroup` 拖给 `Toggle`
 - 添加组就可以做一个单选的效果, 给一个空物体添加一个 `Toggle Group` 组价, 就可以把多个 `Toggle` 给这个物体做子物体就形成了一个组
- `Toggle` 添加事件 `on click`
 - 观察到这里的 `on click` 是一个带参数的委托, 参数为一个布尔值
 - 这里选择函数支持动态传参和静态传参
- 这里的代码添加事件跟 `Button` 差不多

InputField输入框组件

- `Text Component` : 输入文本的组件
- `Text` : 输入的文字
- `Character Limit` : 限制输入的字符个数
- `Content Type` : 限制输入字符的类型, 例如限制整型, 或者密码变成星号
- `Line Type` : 这个文本框可以输入几行, 可以单行双行或者回车结束编辑
- `Placeholder` : 提示的文字的组件
- `Caret Blink Rate` : 光标闪动的频率
- `Caret Width` : 光标的宽度
- `Custom Caret Color` : 改变光标的颜色
- `Selection Color` : 文字被选中的时候的颜色
- `Hide Mobile Input` : 打开后禁止弹出键盘, 手机设备
- `Read Only` : 让这个文字变成只读文字, 不可修改不可编辑
- `InputField` 添加事件 `on click`
 - 这里有两个委托, 一个 `Change` 委托是在你输入的过程中每输入一个字符就会执行一次
 - 还有一个委托就是当你输入完之后, 按回车或者点输入框外面就执行一次

Slider滑块组件

- `Fill Rect` : 滑块填充区域的图形
- `Handle Rect` : 滑块的图形
- `Direction` : 滑块滑动方向, 左到右还是右到左
- `Min Value & Max Value` : 最大值和最小值
- `Whole Numbers` : 是否限制滑块滑动的值为整数
- `Value` : 滑块滑动的值
- 添加事件 `on click`, 这里委托添加的参数为 `single` 单精度

Scrollbar滚动组件

- `Handle Rect` : 拖拽条对象
- `Direction` : 运行方向, 可以选择上下, 下上, 左右, 右左
- `Value` : 拖拽条对应的值 (0起始位置, 1结束位置)
- `Size` : 拖拽条占滚动条的比例, 可视区域/实际区域, 自动计算, 一般不去调
- `Number of Steps` : 按步拖拽, 固定步数分步显示所有实际区域
- 这里的事件委托是一个单精度浮点型的参数, 检测的是滑动块左右滑动时候的值

Scroll View矩形滚动组件

- **Content**：实际显示区域的UI对象，这个是重点，可视范围就是这里控制的
- **Horizontal**：是否开启横向滚动
- **Vertical**：是否开启纵向滚动
- **Movement Type**：滚动类型
 - **Unrestricted**：无边界自由滚动
 - **Elastic**：有边界带回弹效果
 - **Elastic**：回弹系数
 - **Clamped**：有边界无回弹效果
- **Inertia**：有无拖拽惯性
- **Deceleration Rate**：设置惯量时，减速率决定内容停止移动的速度。速率为0将立即停止运动。值为1表示运动永不减速。
- **Scroll Sensitivity**：滚轮或触摸板移动系数
- **Viewport**：**ScrollView**的可视区域
- **Horizontal Scrollbar**：横向滚动条
- **visibility**：可见性（可见区域与实际显示区域对比）
 - 一直显示
 - 自动隐藏
 - 自动隐藏，并支持自动扩展区域
- **Spacing**：空间，横纵滚动条交叉区域预留空间，当删除滚动条不想要的时候，设置这个它的留边值，就可以无留边滚动，做一个超全屏UI效果
- **verticalScrollbar**：与上面的一样
- 事件添加 **on click**，这里的委托参数是一个 **vector2** 的类型，这里检测的就是滑动条左右上下的值，当滑动到最底部的时候 **y** 会等0，用此来判断是否滑动到了底部，当然这个值是是根据滑动条是从上往下滑还是从下往上滑来决定的

Content Size Fitter

- 宽高自适应，挂载 **Content** 下面，有水平自适应和纵向自适应
- **Unconstrained**：无约束，不要根据布局元素驱动宽度。
- **Min Size**：根据布局元素的最小宽度改变宽度
- **Preferred Size**：根据布局元素的首选宽度改变宽度。

排列组件

Grid Layout Group表格布局组件

- 搭配 **Scroll View** 可以用作背包等等类型的UI
- **Padding**：外框的内边距
- **Cell Size**：内部元素的大小
- **Spacing**：子元素的间距
- **Start Corner**：第一个子元素位于的角
 - 左上，右上，左下，右下
- **Start Axis**：开始排列的轴方向

- 横向，纵向
- **Child Alignment**：子元素对齐方式
 - 外框的九个点位，左上对齐左下对齐左中对齐等等
- **Constraint**：固定行列数，让其中的元素只能排几列
 - 自适应，设置固定列[列数]，设置固定行[行数]

Vertical Layout Group纵向排序组件

- 与表格排列类似
- **Child Alignment**：子元素对齐方式
- **Child Controls Size**：是否锁定子元素的宽高，锁定后就不能改变子元素的宽高
- **Child Force Expand**：子元素强制自适应，锁定之后就会根据父元素宽高，子元素就等分自适应

Horizontal Layout Group横向排序组件

- 参数与纵向一样

Dropdown下拉列表

- 就是一个下拉菜单选择选项
- **value**：选项的值，选项是一个列表，所以索引值从0开始
- **Options**：添加下拉选项
- 添加事件的委托参数是一个整型的值，用来判断选择的是哪个选项

DoTween插件

类拓展

- 扩展方法可能实现向现有类型“添加”方法，而无需创建新的派生类型（继承）
- 扩展方法必须是静态方法，可以像实例方法一样调用
- 如果原始类中有同名方法，原始方法的优先级高于扩展方法
- 其实类拓展就是将新写的方法添加到原有的类中，这样我们可以往 **Transform** 里添加方法等等，但是 **MonoBehaviour** 类不可拓展
- 以下就是往 **Transform** 中添加一个说话的方法，参数列表中一定要指定这个类，且方法必须是静态的

```
public static class Twenn
{
    public static void AddSpeek(this Transform speak)
    {
        Debug.Log("");
    }
}
```

DoTween

- **DoTween** 插件其实就是类拓展写的一个插件
- **DOTween** 是一个免费的 **Unity3D** 动画插件，少量编码即可以实现常见的动画效果
- 引入新的命名空间 `using DG.Tweening;`

- 安装插件
 - Window->AssetStore下载
 - 导入Package
 - 菜单栏->Tools->DOTween Utility Panel->Setup按钮
- 更新插件
 - 删除Resources/DOTweenSettings文件
 - 删除老的DOTween安装目录Demigiant
 - 重新导入Package，再走安装流程
- 在线手册
 - [国外手册][<http://dotween.demigiant.com/documentation.php>]
 - [沈军老师翻译手册]
[<https://shenjun4unity.github.io/unityhtml/%E7%AC%AC9%E7%AB%A0%20DOTween/9.2%20%E6%96%87%E6%A1%A3.html>]
- Do Tween Animation —— 做补间动画
- Do Tween Patch —— 做补间路径

DoTween Animation图形参数

- Do Tween Component ---- DoTween 组件的添加
- Do Tween Animation ---- 做补间动画
- 所有可添加的动画效果
 - Move -- 移动 (- 世界坐标)
 - LocalMove -- 移动 (自身坐标 - 相对于父物体)
 - Rotate -- 旋转 (- 世界轴)
 - LocalRotate -- 旋转 (- 自身轴)
 - Scale -- 缩放比例
 - Color -- 颜色
 - Fade -- 淡入/淡出(- 透明度)
 - Text -- 文本
 - Punch -- 重击 (打击感)
 - Position —— 位置 (- 打击感 位移 偏移量)
 - Rotation —— 旋转 (- 打击感 角度 偏移量)
 - Scale —— 缩放比例 (- 打击感 缩放 偏移量)
 - Shake -- 摇动 (震动)
 - Position —— 位置 (- 摇动 位移 偏移量)
 - Rotation —— 旋转 (- 摇动 角度 偏移量)
 - Scale —— 缩放比例 (- 摇动 缩放 偏移量)
 - Camera -- 相机
 - Aspect —— 方向 (- 改变Camera的视角范围)
 - Background —— 背景 (- 改变Camera的背景底色)
 - Field of View —— 视野 (- 改变Camera的视野距离)
 - OrthoSize —— 正交视野大小
 - PixelRect —— 像素矩阵
 - Rect —— 矩阵

- `AutoPlay` —— 自动播放（动画）
- `AutoKill` —— 自动删除（动画）
- `Duration` —— 持续的时间
- `Ignorer TimeScale` —— 忽略时间表
- `Ease` —— 减缓（动画的过程曲线：这是一个枚举类型）
- `loops` —— 循环次数
- `ID` —— 动画的 `ID` 标示（通过 `ID`，直接用代码控制）
- `TO` —— 到达目标位置（可通过点击，设置 `Form` - 从哪里来
- `Snapping` —— 强烈 / 突然折断
- `Relative` —— 相对的
- `Events` —— 事件
 - 监听的事件：
 - `OnStart` —— 初始化（只有在第一次运行）
 - `OnPlay` —— 运行开始（每次运行开始）
 - `OnUpdate` —— 运行时每一帧
 - `OnStep` —— 每一步（运行中的每个步骤）
 - `OnComplete` —— 动画结束后

常用方法

- 直接 `transform` 点出即可
- `DoFade()` 淡入或淡出
- `DoLocalMove()` 本地坐标系，移动动画
- `DoScale()` 缩放动画
- `DoRotate()` 旋转
- `DoColor()` 颜色变化
- `DoText()` 文本逐渐展开

UI控件的自定义交互

- 基础的交互只有一些点击等功能，若要实现长按等等没法实现，所以需要自定义交互

Button的自定义交互

- 引入命名空间 `UnityEngine.EventSystems`
- 接入接口
 - `IPointerEnterHandler`：当鼠标移入到这个控件的时候执行这个接口中的方法
 - `IPointerExitHandler`：当鼠标移出这个控件的时候执行这个接口方法
 - `IPointerDownHandler`：当鼠标在控件内部按下的时候执行这个接口方法
 - `IPointerUpHandler`：必须按下接口方法执行的时候，这个抬起接口才会执行，如果按下没有执行那么抬起也不会执行
 - `IPointerClickHandler`：必须在控件内部按下抬起的时候，才会执行 `Click`，这就是有效的触发效果

- 添加添加事件的类，就是 button 的添加事件 onClick，引入两个命名空间 using System; 和 using UnityEngine.Events;，这样就跟 button 一样，有了一个 onClick 添加事件的窗口，然后就跟 button 一样可以添加事件了

```
//事件成员变量
public ButtonClickedEvent onClick;
//因为当前类需要作为成员变量显示在编辑器上，所以需要加标签[Serializable]
[Serializable]//继承自System
public class ButtonClickedEvent : UnityEngine.Event//继承自UnityEngine.Events
{
    public ButtonClickedEvent()
    {

    }
}
```

Button长按效果的实现

- 思路：当用户按下鼠标的时候，需要启动一个计时器，当计时器达到某个阈值的时候触发这个事件回调，当鼠标按键抬起的时候，需要有长按结束的事件执行

```
{
    //点击事件成员变量
    public ButtonClickedEvent onClick;
    //开始长按事件成员变量
    public UnityEngine.Event onLongPressedStart;
    //结束长按事件成员变量
    public UnityEngine.Event onLongPressedEnd;

    //判断是否在按钮图片内
    private bool _InEnter = false;
    //是否按下按钮
    private bool _InDown = false;
    //是否在长按中
    private bool _IsInLongPressed = false;

    //长按时候的计时器
    private float _PressedTime = 0f;
    //长按的阈值
    public float LongPressedTime = 3f;

    void update()
    {
        //如果没有按下鼠标，则不计
        if(!_InDown)
            return;
        //处于长按中，那么就不再计时
        if(_IsInLongPressed)
            return;

        //如果累积的时间，大于触发的阈值，则执行长按开始事件
        if(_PressedTime >= LongPressedTime)
        {
            onLongPressedStart.Invoke();
            //触发长按后，需要将按钮改为长按状态
        }
    }
}
```

```

        _IsInLongPressed = true;
    }
    //要在判断当前秒数之后才再累加，否则会多加一帧
    _PressedTime += Time.deltaTime;
}

public void OnPointerEnter(PointerEventData eventData)
{
    _InEnter = true;
}

public void OnPointerExit(PointerEventData eventData)
{
    _InEnter = false;
}

public void OnPointerDown(PointerEventData eventData)
{
    _InDown = true;
}

public void OnPointerUp(PointerEventData eventData)
{
    _InDown = false;

    //在按钮区域内
    if (_InEnter)
    {
        //在长按状态的时候就不能执行点击时候的事件
        if(!_IsInLongPressed)
            onClick.Invoke();
    }

    //长按结束后时间归零
    _PressedTime = 0f;

    //如果处于长按状态下，执行长按结束回调函数，并将长按状态关闭
    if(_IsInLongPressed)
    {
        //长按结束的时候执行结束事件
        onLongPressedEnd.Invoke();
        //长按状态关闭
        _IsInLongPressed = false;
    }
}
}
}

```

拖拽效果的实现

- 用来实现摇杆
- 接入三个接口
 - `IBeginDragHandler`：开始拖拽的时候执行接口方法
 - `IEndDragHandler`：结束拖拽的时候执行接口方法
 - `IDragHandler`：拖拽期间执行的接口方法
- 做一个让图片跟着鼠标的拖拽进行移动

```

{
    Vector2 localPos;
    //把一个屏幕坐标转换成一个本地坐标，为什么要转本地坐标呢，因为一般图片都是在Canvas下的，所以不是在世界坐标，一定是相对于Canvas的本地坐标
    RectTransformUtility.ScreenPointToLocalPointInRectangle(
        GameObject.Find("/Canvas").transform as RectTransform, //参考坐标系的父物体
        eventData.position, //鼠标点击的位置
        eventData.pressEventCamera, //当前触发事件的相机
        out localPos //当前的本地坐标
    );
    //把这个物体的坐标设置成鼠标点的坐标，这样就可以拖拽物体进行移动
    transform.localPosition = localPos;
}

```

- 摇杆的实现

```

public class DragBar :
MonoBehaviour, IPointerDownHandler, IPointerUpHandler, IDragHandler
{

    //摇杆的底座
    public GameObject dragBar;
    //摇杆的那个杆
    public Transform bar;
    //底座的半径
    public float r;

    public void OnPointerDown(PointerEventData eventData)
    {
        //按下时候摇杆出现
        dragBar.SetActive(true);
        Vector2 localPos;
        //把一个屏幕坐标转换成一个本地坐标，为什么要转本地坐标呢，因为一般图片都是在Canvas下的，所以不是在世界坐标，一定是相对于Canvas的本地坐标
        RectTransformUtility.ScreenPointToLocalPointInRectangle(
            transform as RectTransform, //参考坐标系的父物体
            eventData.position, //鼠标点击的位置
            eventData.pressEventCamera, //当前触发事件的相机
            out localPos //当前的本地坐标
        );
        //把这个物体的坐标设置成鼠标点的坐标，这样就可以拖拽物体进行移动
        dragBar.transform.localPosition = localPos;
    }

    public void OnPointerUp(PointerEventData eventData)
    {
        //底座在抬起的时候就消失了
        dragBar.SetActive(false);
        //抬起的时候中间的杆就回归原点
        bar.localPosition = Vector3.zero;
    }

    public void OnDrag(PointerEventData eventData)
    {
        Vector2 localPos;

```

//把一个屏幕坐标转换成一个本地坐标，为什么要转本地坐标呢，因为一般图片都是在Canvas下的，所以不是在世界坐标，一定是相对于Canvas的本地坐标

```
RectTransformUtility.ScreenPointToLocalPointInRectangle(  
    dragBar.transform as RectTransform, //参考坐标系的父物体  
    eventData.position, //鼠标点击的位置  
    eventData.pressEventCamera, //当前触发事件的相机  
    out localPos //当前的本地坐标  
);  
  
//用这个向量的长度去对比底座半径，如果越界了  
if (localPos.magnitude > r)  
{  
    //就使用越界向量的单位向量乘与一个半径就得到这个摇杆的临界向量  
    localPos = localPos.normalized * r;  
}  
  
//把这个物体的坐标设置成鼠标点的坐标，这样就可以拖拽物体进行移动  
bar.localPosition = localPos;  
}  
  
void Start ()  
{  
    dragBar.SetActive(false);  
}  
  
}
```

自定义交互的组件EventTrigger

- 这个组件定义了很多不同的交互效果
- 使用方法其实跟添加组件获取参数是一样的
 - 第一步找到 UI
 - 第二步给UI添加一个 EventTrigger 组件，或者获取到这个UI上的组件
 - 第三步给 Trigger 添加一个委托列表并指定交互类型
 - 第四步给 Click 添加事件

```
{  
    //找到UI物体  
    GameObject t = this.transform.Find(URL).gameObject;  
    //获取或者添加一个组件  
    EventTrigger trigger = t.GetComponent<EventTrigger>();  
    //创建一个委托列表  
    EventTrigger.Entry entry = new EventTrigger.Entry();  
    //指定交互类型  
    entry.eventID = EventTriggerType.PointerClick;  
    //把委托列表添加给这个组件  
    trigger.triggers.Add(entry);  
    //给委托列表添加事件  
    entry.callback = new EventTrigger.TriggerEvent();  
    entry.callback.AddListener(new UnityAction<BaseEventData>(callBack));  
}
```

- 也可以在指定交互类型、添加完事件之后再把委托列表添加给这个组件

游戏开发的数据处理

JSON基础

- 服务器可能是各种各样的语言写的，而 JSON 就像英语一样，是一种通用语言，可以用来与服务器交接
- JSON 全名：JavaScript Object Notation
 - 功能：JavaScript 对象标记语言，是一种跨平台，跨语言，轻量级的数据交换和存储

JSON基础格式

- 在C#中实例化类和给类赋值是以下这种方式的

```
class OneRow
{
    public string userName;
    public string password;
}
{
    OneRow obj = new OneRow();
    obj.userName = "admin";
    obj.password = "123";
}
```

- 而翻译到 JSON 中就是 {"userName":"admin","password":"123"}，逗号就是一个分隔符，最后一个是不需要加分割符了
- JSON 工具介绍：<http://www.bejson.com>，用于校验有没有语法错误，或者转换为 JSON

JSON支持的数据结构

- 数字型：short，int，long，float，double
- 字符串："abc"，"你好"，'abc'
- 布尔：true，false
- null：null
- 数组（列表）：[值1, 值2]
- 对象（字典）：{"键1": "值1", "键2": "值2"}

```
{
    "userName": "admin", //字符串格式
    "age": 999, //数字格式，不用区分整型字符型，都会识别是数字
    "iswoman": false, //布尔型
    "cell": null, //空值
    "ids": [28, 30, 31], //数组
    "grade": { //字典
        "xxx": [10, 8, 9], //字典嵌套数组
        "sss": 99,
        "aaa": "没交"
    }
}
```

JSON的符号含义

- 大括号组：对象，字典
- 中括号组：数组，列表
- 冒号：赋值，左侧是变量或键名称，右侧为值
- 逗号：元素分割符，最后一个元素后，没有逗号
- 双引号组：修饰变量（可以不加），表示string数据类型
- 单引号组：同双引号组

JSON在游戏中的使用

- 存储在服务器中的数据
- 存储在策划配置的Excel中（Excel -> JSON）
- 将Excel中的数据导出为 JSON
 - 填写Excel数据
 - 将Excel数据，导出为 .CSV
 - 通过文本编辑器，打开CSV文件内部内容，复制
 - 如果使用代码进行CSV文件转 .json 文件，记得Excel导出的CSV文件是 GB2312 的字符编码，转换前，需要将文件内容字符集转换为 UTF-8，将数据贴到转换工具上(那个网站)

C#使用JSON数据

- 数据存储（序列化）：将C#的数据格式，转化为JSON字符串，存储或传输
 - 序列化：程序数据 -> JSON 字符串
- 数据使用（反序列化）：将JSON字符串中存储的数据，转化为C#可用的数据格式，实现代码逻辑
 - 反序列化：JSON 字符串 -> 程序数据

操作使用的JSON文件内容

```
[
  {"ItemID":1,"Name":"柯尔凡的猩红月牙斧","Price":20000,"Color":4},
  {"ItemID":2,"Name":"里卡尔的染血手术刀","Price":24000,"Color":4},
  {"ItemID":3,"Name":"缚龙者巨杖","Price":14500,"Color":3},
  {"ItemID":4,"Name":"心魔战戟","Price":6000,"Color":1},
  {"ItemID":5,"Name":"迦托克，信仰之力","Price":8000,"Color":2},
  {"ItemID":6,"Name":"缚誓者，游侠将军的战刃","Price":11000,"Color":3},
  {"ItemID":7,"Name":"尼伯龙根","Price":30000,"Color":5},
  {"ItemID":8,"Name":"布林托尔，白骨裁决者","Price":4000,"Color":1}
]
//color的数字分别对应白色绿色蓝色紫色橙色
```

将JSON文件在Unity中读取出来

- 将 .json 文件拷贝到Unity中，然后跟加载资源文件一样
 - 读取 Resources 文件夹下 .json 文件的内容

```

{
    //可以将Unity中.json文件作为Unity的TextAsset数据类型加载
    TextAsset ta = Resources.Load<TextAsset>("item");
    //获取TextAsset内部的文本内容
    string json = ta.text;

    Debug.Log(json);
}

```

- TextAsset 支持的读取文件的拓展名

- .txt
- .html
- .htm
- .xml
- .bytes
- .json
- .csv
- .yaml
- .fnt

JSON工具LitJson

- 该工具用于解析 .json 文件
- 官网：<https://litjson.net/>：点 Source 跳转到 github 后点 releases 下载
- 工具的使用
 - 导入 src 文件夹中的 C# 文件
 - 删除 C# 以外的文件
 - 如果 Unity 提示警告了，删除导致 warning 的特性（中括号）代码
- 使用库中的方法要引用命名空间 LitJson
- 序列化：JsonMapper.ToJson()
 - 通用型

```

//单个
{
    //C#数据
    JsonData jsonData = new JsonData();
    jsonData["Name"] = "柯尔凡的猩红月牙斧";
    jsonData["Price"] = 20000;
    //序列化
    string toJson = JsonMapper.ToJson(jsonData);
}

//多个
{
    //先创建一个线性表
    JsonData listJosn = new JsonData();
    //每个去添加数据
    JsonData one = new JsonData();
    one["ItemID"] = 1;
    one["Name"] = "柯尔凡的猩红月牙斧";
}

```



```

JsonData tow = new JsonData();
tow["ItemID"] = 2;
tow["Name"] = "里卡尔的染血手术刀";

//将数据添加到表中
listJosn.Add(one);
listJosn.Add(tow);

string toJson = JsonSerializer.ToJson(listJosn);
Debug.Log(toJson);
//可以将数据存入本地文件中
PlayerPrefs.SetString("list", toJson);
}

```

- 泛型序列化方法

```

//定义一个自定义的数据类，加构造函数
public class ItemRow
{
    public int ItemID;
    public string Name;
    public int Price;
    public int Color;

    public ItemRow(int itemId, string name, int price, int color)
    {
        ItemID = itemId;
        Name = name;
        Price = price;
        Color = color;
    }
}

{
    //声明一个数据类的线性表
    List<ItemRow> itemRows = new List<ItemRow>();
    //添加数据
    itemRows.Add(new ItemRow(1, "柯尔凡的猩红月牙斧", 20000, 4));
    itemRows.Add(new ItemRow(2, "里卡尔的染血手术刀", 24000, 4));

    string toJson2 = JsonSerializer.ToJson(itemRows);
    Debug.Log(toJson2);
}

```

- 反序列化步骤，有两种方式，通用型和泛型

- 通用型

- 获得 JsonData (类似C#的 object) : `JsonMapper.ToObject()`
- `JsonData` 的下层数据，也是 `JsonData`，所以使用前需要进行转换
- `JsonData` 包含 `ToJson` 方法，可以直接序列化

```

{
    TextAsset ta = Resources.Load<TextAsset>("item");
    string json = ta.text;
    //将JSON字符串转化为C#变量
    //参数：读取到的JSON字符串
    //返回值：是JSON解析后的数据
    JsonData date = JsonMapper.ToObject(json);

    //data的第一层是数组，第二层是字典，所有跟数组的使用一样通过下标再通过键值就可以访问到这个数据
    int price = (int)date[1]["Price"];
    Debug.Log("里卡尔的染血手术刀的价格是：" + price);
}

```

○ 泛型

- 获得指定类型：`JsonMapper.ToObject<T>()`
- 首先要定义一个类，类变量的名字要和JSON文件中的变量名一样，一定要一样

```

public class ItemRow
{
    public int ItemID;
    public string Name;
    public int Price;
    public int Color;
}

{
    TextAsset ta = Resources.Load<TextAsset>("item");
    string json = ta.text;
    //使用定义的类型列表去每个JSON文件中的每一行的数据
    List<ItemRow> data = JsonMapper.ToObject<List<ItemRow>>(json);
    //可以遍历出所有数据
    for (int i = 0; i < data.Count; i++)
    {
        Debug.LogFormat("道具名字是：{0}，价格是：{1}", data[i].Name, data[i].Price);
    }
}

```

○ 注意的两点

- 反序列化不要求最外层是对象
- 对象内部存储对象不需要 `[System.Serializable]` 修饰Class声明

Unity中关于JSON的类：JsonUtility

- 序列化方法：`ToJson()`
- 反序列化方法：`FromJson()`
- 使用内置的类序列化方法是有要求的

```
{
    List<ItemRow> itemRows = new List<ItemRow>();
    itemRows.Add(new ItemRow(1, "柯尔凡的猩红月牙斧", 20000, 4));
    itemRows.Add(new ItemRow(2, "里卡尔的染血手术刀", 24000, 4));

    JsonUtility.ToJson(itemRows);
}
```

- 仅仅这样数据是无法读取出来的，因为这个结构最外层是个列表，但是使用内置的类最外层必须是一个对象，所以修改之后
 - 再定义一个类，类中放入这个列表

```
public class ItemTable
{
    public List<ItemRow> data = new List<ItemRow>();
}
{
    //要求最外层是对象，所以先创建一个对象
    ItemTable itemTable = new ItemTable();
    //创建一个列表
    List<ItemRow> itemRows = new List<ItemRow>();
    itemRows.Add(new ItemRow(1, "柯尔凡的猩红月牙斧", 20000, 4));
    itemRows.Add(new ItemRow(2, "里卡尔的染血手术刀", 24000, 4));
    //把列表数据加载到对象中
    itemTable.data = itemRows;
    JsonUtility.ToJson(itemRows);
}
```

- 但是这样还是无法显示出这个内容来，所以必须给几个类添加一个可视化标签
[System.Serializable]

```
[System.Serializable]
public class ItemRow
{
    public int ItemID;
    public string Name;
    public int Price;
    public int Color;

    public ItemRow(int itemId, string name, int price, int color)
    {
        ItemID = itemId;
        Name = name;
        Price = price;
        Color = color;
    }
}

[System.Serializable]
public class ItemTable
{
    public List<ItemRow> data = new List<ItemRow>();
}
```

- 最终就可以使用这个序列化的内容
- 反序列化跟也是最外层必须是对象，且跟外部工具使用方法差不多

```
{
    TextAsset textAsset = Resources.Load<TextAsset>("unityItem");
    string json = textAsset.text;

    ItemTable item = JsonUtility.FromJson<ItemTable>(json);

    for (int i = 0; i < item.data.Count; i++)
    {
        Debug.Log(item.data[i].Name);
    }
}
```

XML

- 全名：Extensible Markup Language
- 可扩展标记语言，标准通用标记语言的子集，是一种用于标记电子文件使其具有结构性的标记语言。
- 支持的数据类型
 - 数字
 - 字符串
 - 布尔
 - 数组（链表）
 - 对象
 - null
- 与JSON的对比
 - JSON 格式 {"Username" : "root", "Password" : "123"}
 - XML 格式
 - 头部一般放编码格式等

```
<!--必须要有头部-->
<?xml version="1.0" encoding="UTF-8" ?>
<!--开始根节点-->
<Root>
    <!--数据存储要从开始标签到闭合标签，数据放在中间-->
    <Username v="123">root</Username>
    <Password>123</Password>
<!--闭合根节点-->
</Root>

<!--还可以这样写-->
<?xml version="1.0" encoding="UTF-8" ?>
<Root>
    <Item ItemID="1" Name="柯尔凡的猩红月牙斧"></Item>
    <Item ItemID="2" Name="里卡尔的染血手术刀"></Item>
</Root>
```

- XML 规则
 - 所有 XML 元素都须有关闭标签
 - XML 标签对大小写敏感
 - XML 必须正确地嵌套
 - XML 必须存在一个父节点
- XML 有些字符必须转过以后才能用，否则是无法识别的
 - "<" => "<"
 - ">" => ">"
 - "&" => "&"
 - "'" => "'"
 - "\"" => """
- 拓展学习：<http://www.w3school.com.cn/xml/index.asp>
- 类库：<http://www.cnblogs.com/fish-li/archive/2013/05/05/3061816.html>
- JSON 和 XML 对比
 - JSON 比 XML 可读性高
- 同样具有很强的扩展性
 - XML 的解析得考虑子节点父节点，让人头昏眼花，而 JSON 的解析难度几乎为0。
- JSON 相对于 XML 来讲，数据的体积小，传输和解析的速度更快。
 - JSON 不使用保留字

文件的读写

- 引入两个新的命名空间
 - System.IO;
 - System.Text;

写

```
FileStream fs = new FileStream("文件路径", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
//开始写入
sw.Write("Hello world!!!!");
//清空缓冲区
sw.Flush();
//关闭流
sw.Close();
fs.Close();
```

- Flush() 的作用：你写了一个程序，其中要对硬盘上的一个文件操作，FileStream fs = new FileStream(fileName)，这样就是建立了一个文件缓冲流，换句话说的意思就是说你通过这条程序，计算机给了一块内存空间，但是呢这块内存空间不是你想干什么就干什么的，他是专门存 fileName 这个文件里面的内容的，内存空间的大小，和其他信息。简单地操作是没有办法访问的。当你要从文件里面读取一个 int 整数的时候，程序做的不仅仅是读取一个 int 型整数，他会把该数据附近的一大块数据都读出来放在刚才的那块空间中，为什么这么做呢，因为 CPU 访问硬盘比访问内存慢多了，所以一开始读出很多的数据，后面需要使用的时候直接使用读出来的，就防止了再次访问硬盘。相应的，你要往文件里面写入数据，也是先存到这个内存里，等存的东西足够多了，或者过了足够的时间，系统一次性把内容写入硬盘。

- `Flush` 的作用就是强制执行了一次把数据写出硬盘，这样，你写入的数据确实到了文件中，否则如果程序突然中断，你要写入的内容也许还没写到文件中，就造成了数据丢失。

读

- 当文本的内容比较大时，我们就不要将文本内容一次读完，而应该采用流（`Stream`）的方式来读取内容。C#为我们封装了 `StreamReader` 类

```
//使用流的形式读取文件
StreamReader sr = new StreamReader("文件路径", Encoding.Default);
string line;
while ((line = sr.ReadLine()) != null)
{
    Console.WriteLine(line.ToString());
}
```

FileMode的几个类型

- `Create`：创建文件，如果存在就执行覆盖
- `CreateNew`：创建新文件，若存在就会报错
- `Append`：追加，在原来的文件内容之后追加
- `Open`：打开文件，一般是在执行文件读取的时候使用

File静态类方法

- `bool Exists(path)`：判断文件是否存在，存在返回true,不存在返回false
- `void Copy(string path,string newpath)`：将文件复制到一个新的位置
- `void Move(string path,string newpath)`：将文件移动到一个新的位置，原来的文件不存在
- `Delete(string path)`：删除文件

Encoding编码格式

- 指定读取文件后的编码格式
 - `Default`：默认编码
 - `ASCII`、`UTF8` 等等其他编码

关于Close关闭流

- 如果你开启了文件流在读取一个文件却没有 `close`，那么这个文件一直被你的程序在占用。别人就无法再操作这个文件。当然，读还是可以的，只是无法写这个文件
- 就相当于你打开一个Word文件没有关闭，再去修改它的名字或者其他操作，系统会提示这个文件正在打开一样

老师给的配置文件使用方法

1. 修改工具包中 `Release` 中的XML表中的路径数据

- `loadUrl` —— `excel` 表所在的路径
- `outUrl` ——最终输出的二进制文件存放的路径
- `beanOutUrl` ——输出的结构类存放路径
- `containerOutUrl` ——输出的容器类存放路径

2. 如果选择 excel 表后，左边显示的不是表数据加载完毕，而是提示缺少什么文件，就安装 `AccessDatabaseEngine`
3. 一定要记得修改 excel 表中的表名，不能是 `Sheet`
4. excel 表中前五行的作用
 1. 决定主键，如果主键是int类型那么千万不要重复
 2. 字段名
 3. 类型，只有 `int` 和 `string` 类型，默认不填就是 `int`
 4. 描述字段，说明
 5. 注释，结构文件会自动生成注释
5. 从第六行开始，才是配置你的数据
6. 生成的数据\$2\$进制文件放 `Resource` 下的必须 `GameData` 文件夹，类文件随意，但是最好放规范点

代码使用

```
void Start () {
    //添加一个数据事件 在事件里去添加你要加载的表
    GameDataMgr.GetInstance().addDataInfo += InitInfo;
    //启动数据加载,这是个协程, 参数是委托
    StartCoroutine(GameDataMgr.GetInstance().analysisData(LoginOver, null));
}

public void InitInfo()
{
    //加载这个表的数据, 参数一个是类容器名, 一个是二进制文件名, 二进制文件必须在Resource下的
    GameDataMgr.GetInstance().addWaitAnalysisTable("T_roleInfoContainer",
    "t_roleInfo");
}

public void LoginOver()
{
    //获取表中一行的数据
    T_roleInfo info = GameDataMgr.GetInstance().getTable<T_roleInfoContainer>
    ().getValue(1);
    //获取整个表的数据, 形成一个线性表
    List<T_roleInfo> t_RoleInfos =
    GameDataMgr.GetInstance().getTable<T_roleInfoContainer>().getList<T_roleInfo>();
}
```

网络传输

网络参考模型

- OSI七层模型



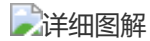
- TCP/IP模型



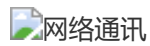
- OSI与TCP-IP模型



- TCP/IP详细图解



网络通信



- 长链接和短连接
 - 短连接 (HTTP)
 - 需要数据交换时，连接服务器，数据交换完成后，断开连接
 - 卡牌类游戏，不需要一直链接服务器，本地模拟之后再告诉服务器结果
 - 长连接 (TCP)
 - 客户端和服务端一开始会进行连接，并一直保持连接，直到不再和服务端交换数据时，会断开连接
 - 常用即使游戏就长链接，因为要一直刷新数据，一直与服务端进行链接

Unity网络请求类WWW

- 通过协程给网址发送请求，注意Unity不支持 `Https` 的协议请求，需要使用 `Http`
 - 使用协程的原因就是避免下载东西的时候主线程卡死
- 赵老师服务器地址：<http://hxsd.ucenter.honorzhao.com/user/register>

WWW下载文本

- 开启网络请求获得路径上的网页源码

```
{  
    //开启一个网络请求  
    WWW www = new WWW("https://fanyi.baidu.com/?aldtype=16047#zh/en/%E9%93%AD%E6%96%87");  
  
    yield return www;  
  
    //网络请求成功获得服务器的字符串数据  
    Debug.Log(www.text);  
}
```

WWW下载文件

- 开启网络请求下载文件，可以下载任何文件

```
void Start ()  
{  
    //协程开启网络请求，传入一个网络图片路径  
    StartCoroutine(  

```



```

Request("http://wx2.sinaimg.cn/large/d2e27164gy1fbu77brf5jj21hc0xc1kx.jpg")
);
}

IEnumerator Request(string URL)
{
    //www网络请求路径, www.bytes就是通过网络路径下载文件
    WWW www = new WWW(URL);
    yield return www;
    //引入新命名空间System.IO;这个方法就是用来写入文件, 参数是一个文件夹加要写入的文件名, 第
    二个参数就是一个二进制文件
    File.WriteAllBytes("D:\\下载测试\\t.jpg", www.bytes);
}

```

URL域名结构

- 通信协议：
 - http://
 - https://
- 主机地址：主机地址都是IP地址，但是通过DNS协议形成域名便于记忆
 - IP：39.105.153.133
 - 域名：hxsducenter.honorzhao.com
- 端口号：可以在域名后面加上端口号hxsducenter.honorzhao.com:80
 - :80：提供HTTP服务的端口
 - :443：提供HTTPS服务的端口
- 目录：服务器脚本在服务器上存储的路径
 - "/目录名"：<http://hxsducenter.honorzhao.com/user/register>中的/user/register就是目录名
- 脚本名称：用什么语言写的服务器脚本，可以在域名后面加这个脚本名称
 - 老师服务器使用PHP语言：<http://hxsducenter.honorzhao.com:80/index.php>
- URL参数：?参数名=参数值&参数名=参数值
 - 以?开头
 - 参数名=参数值
 - 多个参数以&分割
 - 可以通过参数访问目录
 - 不带参数：<http://hxsducenter.honorzhao.com/user/register>
 - 带参数：<http://hxsducenter.honorzhao.com/index.php?c=user&a=register>

HTTP请求类型（GET和POST的区别）

- 请求头：客户端向服务器发送数据的报（数据报）头
- 响应头：服务器向客户端发送回来的报头
- Get：Get的数据是通过URL地址传递的
- Post：Post的数据是通过HTTP数据头传递的
- 区别

- GET 传递的数据会被浏览器和搜索引擎记录，不安全（被记录）
- POST 传递的数据，记录在请求头部中，相对安全
- GET 能够传递的数据量受到 URL 最大长度的限制
- POST 可以传递任意长度的数据（服务器会有限制）

编码

- 如果需要在 URL 传递数据中加入特殊字符，就需要对数据进行 URL 编码
 - <https://baike.baidu.com/item/urlencode/8317412>，后面这个数字就是对特殊符号的编码
 - Unity 中对字符串的编码方法 `UnityWebRequest.EscapeURL()`

```
WWW www = new WWW("http://hxsu.ucenter.honorzhao.com:80/index.php?c=user&a=register"+ UnityWebRequest.EscapeURL("&"));
```

常见状态号

- \$200\$：成功
- \$301\$：重定向（当前页面已过时，跳转到新的页面）
- \$403\$：对被请求页面的访问被禁止（权限不足）
- \$404\$：服务器无法找到被请求的页面（服务器给的URL地址不对）
- \$500\$：服务器内部错误（服务器代码有错）
- \$502\$：服务器响应失败（访问量过大，不能提供服务的就会收到）

TCP长链接



- 网络层IP协议，找到需要链接的IP地址
- TCP/UDP协议一般处于传输层，用于数据传递
- Http协议处于应用层

Socket套接字

- Socket套接字：
<https://baike.baidu.com/item/%E5%A5%97%E6%8E%A5%E5%AD%97/9637606>
- IP协议实现主机的网络定位，就是一个地方的地址
- 操作系统的端口实现数据的流入与流出，这个地址的门
- 套接字：是将IP地址与主机端口号合并在一起后的数据，IP地址定位主机位置，端口号知道通讯入口与出口，从而就可以实现主机的数据交换
 - 例如一个主机IP地址是\$10:28:213:186\$，开放端口\$8080\$，则套接字就是：
\$10:28:213:186:8080\$
- Socket编程基于传输层实现，所以需要指定协议类型（TCP或UDP）

TCP编程方法

- 字节长度之间的换算
- 计算机只能识别\$1/0\$，同理也只能存储\$1/0\$

- 一个二进制单位叫做位 (bit)
- bit 就是计算机最小得到储存单位

单位	换算
1Byte	8Bit
1KB	1024Byte
1MB	1024KB
1GB	1024MB
1TB	1024GB

- UTF-8编码长度是从1个字节~6个字节存储，其中中文是3个字节
- 查询API：<https://msdn.microsoft.com/zh-cn/>，查询Socket的API或者其他语言API都可以在这个手册中找一下

线程

- Unity中是存在线程的，但是线程中不能对场景中的对象进行任何操作，只有主线程可以操作场景对象

```
//调用线程命名空间
using System.Threading;
using UnityEngine;

/// <summary>
/// 测试线程
/// </summary>
public class TestThread : MonoBehaviour
{
    void Start()
    {
        //创建线程，会在独立的CPU内核运行
        Thread t = new Thread(InThread);
        //让线程开始执行
        t.Start();
    }

    void InThread()
    {
        int i = 0;

        while(i < 10)
        {
            Debug.Log("这是分线程");
            i++;
        }

        //Unity中是存在线程的，但是线程中不能对场景中的对象进行任何操作，只有主线程可以操作场景对象
        //new GameObject("分线程运行结束");
        TestFunc(); //即使是这样回调，在外面去创建一个对象，也会报错，因为这个函数是被线程引用的
    }
}
```

```

        //函数运行退出时，线程将退出
    }

    void TestFunc()
    {
        new GameObject("分线程运行结束");
    }
}

```

链接（三次握手）

- 链接可以写在脚本的 `Start` 函数中

```

//同步
//创建套接字
//创建了一个套接字，参数：网络类型，基于数据流方式，基于TCP协议
Socket socket = new
Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
//调用连接方法，与远程主机建立连接。主机由主机名和端口号指定。
socket.Connect("IP地址", 端口号)

-----

//异步连接
//创建套接字
Socket socket = new
Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
//参数：主机地址、端口、链接成功后的回调函数、其他参数
socket.BeginConnect(Host, Port, _EndConnect, null);

//_EndConnect回调函数中执行
void _EndConnect(IAsyncResult ar)
{
    socket.EndConnect(ar异步连接结果);
}

-----

```

- 异步链接就不会占用主线程，万一链接时间过长就不会卡掉程序

断开（四次挥手）

- 断开可以写在脚本的 `OnDestroy` 周期函数中

```

//同步
//下次使用，会创建全新的套接字
socket.Disconnect(false);
//关闭套接字连接，释放资源
socket.Close();

-----

//异步断开
socket.BeginDisconnect(false, _EndDisconnect, null);

// EndDisconnect回调函数执行
void _EndDisconnect(IAsyncResult ar)
{

```

```
socket.EndDisconnect(ar异步断开连接结果); //有BeginConnect就需要有EndDis
socket.Close();
}
```

发送

```
//创建套接字
_socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
//链接主机与端口
_socket.Connect("hxsdsd.tcp.honorzhao.com", 8282);

string data = "分手";
//字符串, 转字节数组
//Encoding类, 根据字符串的字符集, 自动将其转换为字节数组
byte[] datab = System.Text.Encoding.UTF8.GetBytes(data);

//返回值是发送到网卡的数据字节长度
//Send是阻塞的, 在主线程中运行, 如果数据过大, 则游戏会假死
//参数: 要发送的字节数组, 发送的起始下标, 终止下标, 发送的类型
int length = _socket.Send(datab, 0, datab.Length, SocketFlags.None);

Debug.Log("向服务器发送了长度: " + length + "的数据");
```

接收

```
//这里开了一个1M的缓存区
private byte[] _Buffer = new byte[1024 * 1024]; //开一个字节缓存区
//创建套接字
_socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
_socket.Connect("hxsdsd.tcp.honorzhao.com", 8282);

//参数:
//接收缓冲区
//数据在网卡中的接收起始偏移量
//当次接收的数据长度(缓冲区是1MB, 所以接收数据的长度也是1MB)
//不对接收数据的来源进行指定(接收所有的数据)
//返回值:
//返回值是从服务器接收到的数据的长度
//代码会阻塞主线程, 直到服务器给客户端发送了数据, 接收的阻塞才会结束
int length = _socket.Receive(_Buffer, 0, _Buffer.Length, SocketFlags.None);
Debug.Log("从服务器接收到了长度: " + length + "的数据");

byte[] data = new byte[length];
//遍历接收缓存区中的数据进行数组拷贝
for (int i = 0; i < length; i++)
{
    data[i] = _Buffer[i];
}
//将二进制数据编码回字符串变成人可以看懂得数据类型
string finalData = System.Text.Encoding.UTF8.GetString(data);
```

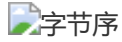
lock关键字

- 锁 (Lock) : 当一个线程, 锁住变量后, 另一个线程, 必须等待锁释放, 才能继续操作变量, 这样就能防止并行执行代码时, 出现内存数据的错误 (读写会同时执行)

数据包处理

字节序

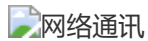
- 字节序, 针对数字, 字符串是没有什么顺序问题的



- 大端字节序: 数据的前位字节, 放在内存低地址位, 后位字节, 放在高内存地址位, 符合人类的思维方式
- 小端字节序: 将数字的后位字节, 放在内存栈的低地址位, 前位字节放在高地址位, 不符合人类思维方式, 但是计算机处理这样的内存更快
- 主机字节序: 当前计算机, 数字的字节表示方式, 与硬件和操作系统有关
- 网络字节序: 无论主机是怎样的字节序, 互联网规定, 传递数据时, 都应该转换为大端字节序进行传递。

数据包定制

- 包头: 记录有关于整个数据包的信息, 需要加密
- 包体: 原始数据, 需要加密, 防止泄露, RC4方式



- 数据打包: 对原始数据, 添加协议头的过程, 就是数据打包的过程 (前四个字节记录数据包总长度, 后面拼接包体内容, 成为一个数据包)
 - 发送一个“分手”字符串的时候, 打包为 (10分手)
- 数据解包: 接收到数据包时, 读取包头, 并根据包头记录信息, 获取到包内的原始数据的过程, 就是解包
 - 例如包头得到是10, 得到这个数据长度为10个字节, 去掉包头4个字节, 得到要接收的字符串是6个字节
- 数据粘包: 发送数据前, 如果有多个数据包需要一起发送, 则可以将数据包, 拼接在一起, 再发送, 这样的发送效率更高
 - 数据粘包有时是硬件将数据包拼接在一起, 有时是代码将数据包拼接在一起
- 数据分包: 当接到数据后, 需要将每一个定制的数据包按格式分离出来, 所写的代码就是分包代码

数据包的生命周期

- 对数据打包
- 对多个数据粘包
- 套接字: 链接、发送
- 套接字: 接收
- 对数据包进行分包
- 对分包后的数据包进行解包
- 获得数据后, 根据数据进行其他代码逻辑

心跳包

- 因为TCP是有连接的, 所以必须在两个PC间, 建立连接。但是如果长时间连接, 却又不发送数据, 则会占用互联网络的通信信道, 就有可能被网络的中间设备 (路由器, 防火墙) 将网络连接断开, 所以为了防止网络被断开, 则需要两台计算机间, 定期发送一些数据, 这样的数据就是心跳数据 (心跳)。

- 常说的网络延迟计算：服务器返回心跳时间 - 客户端发送心跳时间 (ms)

MVC分层开发思想

- MVC思想图示



- 之前的代码功能实现是一个功能模块的代码逻辑（显示处理，数据处理，逻辑判定）都写在一起，这样耦合性非常高，不利于代码的修改调用或者简写
- 代码MVC 分层实现方法，将原来的代码分成三个脚本来实现
 - 显示部分实现（ view 视图）
 - 数据处理实现（ Model 数据模型）
 - 逻辑判定实现（ Controller 控制器）

MVC是什么

- C：控制器，负责流程控制和事件响应
- V：视图，负责图形交互
- M：数据模型，负责数据处理

MVC开发步骤

- 页面预制体制作，做个页面
- 写数据处理脚本，这个脚本不挂载在场景，提供API给控制器用
 - JSON 读写操作
 - 数据的CURD操作
 - C：Create 增加数据
 - U：Update 修改数据
 - R：Read 读取数据
 - D：Delete 删除数据
- 逻辑控制（ Controller 脚本），这个脚本一般一个页面有一个且最好只有一个
 - 给UI添加事件
 - 调用显示脚本
 - 调用数据处理脚本的读写API等等
- 显示脚本（ view 脚本），这个脚本只做显示功能，找到界面上的文本文件把要显示的数据给这些，这个脚本一般就挂载在要显示的UI上，一般一个页面会有很多显示的脚本
 - 文本显示
 - 图片显示
 - 列表显示
 - 其他的资源等等

AssetBundle包

AssetBundle资源管理

- 之前很多动态资源都放在 `Resource` 文件夹下，但是当Unity项目打包出去的时候 `Resource` 文件夹下的资源都会被打包出去
 - 存储在 `Resources` 下的资源，最终会存储在游戏的主体包中，发送给用户，手机系统上，如果需要做资源的更新，是无法使用 `Resources` 即时更新（不需要关闭手机游戏，即可实现游戏更新，即热更新）
- 这时就需要引入 `AssetBundle` 文件包，简称 `AB` 包
 - 其实可以理解为一个压缩包，这个压缩包可以理解为一个文件夹，文件夹里包含了多个文件，这些文件可以分为两类
 - `serialized file`：资源被打碎放在一个对象中，最后被统一写进一个单独得文件（只有一个）
 - `resource files`：资源源文件
 - `AB` 包是独立于游戏主包存在的资源存储文件，使用内部资源时，需要单独下载和加载。

AB包和Resource的区别

- `AB` 包可以理解为可下载的 `Resource`，但其实结构是不一样的
- 存储：
 - `Resources` 内部资源存储在游戏的发布包中
 - `AB` 包存储在独立的文件中：`AB` 包存储在非特殊目录下时，不在游戏的发布包中
- 加载：
 - `Resources` 内部资源使用 `Resources.Load()`
 - `AB`包的加载：
 - 通过 `UnityWebRequest` 下载这个`AB`包文件
 - 通过名称去加载`AB`包中的内部资源

AssetBundle的定义

- `AssetBundle` 是把一些资源文件，场景文件或二进制文件以某种紧密的方式保存在一起的一些文件。
- `AssetBundle` 内部不能包含C#脚本文件，`AssetBundle` 可以配合Lua实现资源和游戏逻辑代码的更新
 - 即 `AssetBundle` 不能更新代码，只能更新资源，所以需要配合Lua实现

AB包的创建与使用

AB包的使用流程

- 指定资源的`AB`包名
- 创建`AB`包
- 上传`AB`包

- 加载AB包和包内资源

AB包的创建

插件AssetBundle Browser

- 下载 AssetBundle 插件
 - 先在 GitHub 上搜 Unity，找一下 AssetBundle，下载下来
 - 导入到项目的Asset文件夹下，其实就是一个插件
 - 可以Windows任务栏中找到 AssetBundle Browser 插件
- 在资源设置面板的右下角可以 New 一个AB文件的名字，或者选择一个文件名字
 - 也可以打开插件，直接右键新建一个AB文件或者直接将资源拖入到 Configure
- 打开插件 AssetBundle Browser 就可以在 Configure 看到新建的这个AB文件名字
- 点击 Build 在 Build Target 选择平台，Output Path 选择路径，点击 build 就可以创建一个AB文件
- AB包配置修改后或AB内部的资源修改后，都需要重新生成AB包
- 导入到AB包的资源，包内部是不存在目录的，如果把一个目录当成一个文件放入AB包中将会报错
 - 如果AB包名称如果配置为这样的结构“ui/package”，ui会作为AB包存储的父目录，package 是AB包的名称

Build下的参数信息

- Build Target：平台
- Output Path：输出路径
- Clear Folders：生成新的AB包的时候删除旧的同名AB包
- Copy to StreamingAssets：
- 以下是高级设置
- Compression：AB包的压缩方式
 - 不压缩：AB包比较大，下载较慢，加载速度快，因为CPU不用运算解压缩
 - LZMA 算法压缩：默认压缩模式，文件尺寸最小，加载速度比LZ4长，使用的时候会将包整体解压
 - LZ4 算法压缩：5.3版本以后可用，压缩比例居中，文件居中，加载速度居中，使用的时候不需要整体解压，可以指定加载资源而不用加载全部，最好选用这种压缩方式

代码创建

- 使用到的API：BuildPipeline.BuildAssetBundles()
- 参数时：打包路径，压缩选项，对象平台等
- 使用代码打包时，先判断路径是否存在，若不存在则进行创建
- BuildPipeline 在 UnityEditor 的命名空间下

```
[MenuItem("Tools/BuildAssetBundles")]
private static void BuildAllAssetBundles()
{
    string url = "AssetBundles";
    if (!Directory.Exists(url))
    {
        Directory.CreateDirectory(url);
    }
    BuildPipeline.BuildAssetBundles(url, BuildAssetBundleOptions.None,
    BuildTarget.StandaloneWindows64);
    Debug.Log("finishd");
}
```

AB包的使用的四种方式

- 主要有四种方式
 - `AssetBundle.LoadFromMemoryAsync`：当AB包在服务器上下载下来时有可能得到的是一个byte数组，这时候可以用这个将byte数组转化为AB包文件
 - `AssetBundle.LoadFromFile`：本地加载
 - `www.LoadFromCacheOrDownload`：已经弃用了，不推荐使用，与 `UnityWebRequest` 一致
 - `UnityWebRequest`：服务器网络端下载下AB包文件

本地加载AB包内部数据

- `Resource` 加载文件直接是加载文件夹下的资源，填写一个 `Resource` 路径就可
- 加载AB包分两步，第一步加载AB包文件，第二步根据资源在AB包内的名称加载AB包内的资源
 - 加载：加载的含义就是将文件状态的AB文件加载到内存中
 - 卸载：将内存中的AB文件释放掉
 - 第一步：加载AB包文件，这个路径要直接填写电脑路径加AB包名称，返回的就是AB包文件对象
 - `AB包 = AssetBundle.LoadFromFile(AB包文件路径)`
 - `AssetBundle.LoadFromFileSync(AB包文件路径)`，异步加载
 - 异步加载得到的是一个 `AssetBundleRequest` 请求类型，访问请求类型下的 `asset` 属性就会得到资源
 - 第二步：加载AB包内部资源，参数是资源在AB包中的名称
 - `资源对象 = AB包对象.LoadAsset<资源类型>("资源名称")`
 - `AB包对象.LoadAssetSync<资源类型>("资源名称")`，异步加载
- 注意**：AB包不能重复加载，但是可以卸载后再加载
 - 但可以同一个AB包重复加载资源
 - 卸载方法 `AB包对象.Unload(bool)`

```
{
    //加载这个AB包文件
    AssetBundle ab = AssetBundle.LoadFromFile("url");
    //加载这个包文件中的资源
    Sprite sprite = ab.LoadAsset<Sprite>("assetName");
    //使用包内的资源
    GameObject.Find("/Canvas/AB").GetComponent<Image>().sprite = sprite;
}
```

- 一定要在使用完之后卸载AB包对象，减少内存的占用

服务端加载AB包

- `www.LoadFromCacheOrDownload`：已经弃用了，不推荐使用，与 `UnityWebRequest` 一致
- `UnityWebRequest`：服务器网络端下载AB包文件，一共三步
 1. 创建一个访问web服务器的请求，可以根据访问不同类型资源使用不同的类，例如这里要访问AB包资源就使用的是 `UnityWebRequestAssetBundle`
 - `UnityWebRequest webRequest = UnityWebRequestAssetBundle.GetAssetBundle(url);`
 2. 开启下载请求
 - `webRequest.SendWebRequest();`
 3. 得到资源文件，同样不同的文件有不同的 `DownloadHandler` 类
 - `AssetBundle asset = DownloadHandlerAssetBundle.GetContent(webRequest);`
- 注意在开启下载请求后，最好还要做一下判空操作
 - `String.IsNullOrEmpty(webRequest.error)`，是否下载错误

```
IEnumerator Start()
{
    string url = @"http://localhost/AssetBundles/zxr/pink.unity";
    //创建请求
    UnityWebRequest webRequest = UnityWebRequestAssetBundle.GetAssetBundle(url);
    //开启下载请求
    yield return webRequest.SendWebRequest();
    //判空
    if (!String.IsNullOrEmpty(webRequest.error))
    {
        UnityEngine.Debug.Log(webRequest.error);
        yield break;
    }
    //获得AB包资源
    AssetBundle asset = DownloadHandlerAssetBundle.GetContent(webRequest);
    //使用AB包内资源
    GameObject obj = Instantiate(asset.LoadAsset<GameObject>("Sprite"));
    obj.name = "WebRequest";
}
```

AB包的依赖关系

- 如果一个AB（名称A）包中的资源，使用到了另一个AB（名称B）包的资源，那么两个AB包就产生了依赖关系。也就是A依赖于B。那么在使用A中资源的时候就要把B包一起加载出来，否则A中资源就会出现资源不完整的情况
 - 在A中的配置文件 `.manifest` 中的 `Dependencies` 就会记录这种依赖关系
- 在这个主AB包，即和储存目录同名的AB文件的配置文件中，记录了所有这个包下所有文件的依赖关系
- 如果要加载有依赖关系AB包，除了使用 `LoadFromFile` 把一个包加载，也可以先去先加载主AB包，加载数据有以下几个步骤
 - 加载主AB包
 - 根据主AB包的配置文件，获得我当前需要加载的AB所依赖的AB们

- 将所有的依赖AB们，加载进来
- 然后就是加载你想要的AB包

```
{

    //加载主AB包
    AssetBundle ab = AssetBundle.LoadFromFile(url);
    //加载主AB包的配置文件，文件类型就是AssetBundleManifest
    AssetBundleManifest abm = ab.LoadAsset<AssetBundleManifest>
("AssetBundleManifest");

    //查询某个AB包的依赖情况
    //参数：被查询依赖关系的AB包名称
    //返回值：被查询依赖关系的AB包，所依赖的所有AB包名称
    string[] names = abm.GetAllDependencies(name);

    //加载所有的依赖
    foreach (string name in names)
    {
        //加载依赖的AB包
        AssetBundle.LoadFromFile(name);
    }

    //加载这个AB包
    AssetBundle realAB = AssetBundle.LoadFromFile(url);
    //加载这个包中的资源
    GameObject prefab = realAB.LoadAsset<GameObject>(name);
    //使用
    GameObject go = GameObject.Instantiate<GameObject>(prefab);
    go.transform.SetParent(GameObject.Find("/Canvas").transform);
}
```

AB包的卸载内存的释放

- Resources 加载的资源，Unity会存储在内存中，用 Destroy 删除对象后是不会降内存的，资源的内存占用依然存在，所以要使用 Resources.UnloadUnusedAssets(); 定期的释放场景没有引用的资源
- 加载AB包，内存的占用根据AB包的压缩模式来决定
 - LZMA 算法压缩：虽然压缩后尺寸最小，但是加载会占用内存，就要跟 Resources 一样，不使用时候要释放掉
 - 卸载方法 AB包对象.Unload(bool)
 - LZ4 算法压缩：不会占用内存，可以一直挂着这个AB包，不需要去释放内存
- 加载AB包中的资源，无论是什么压缩模式都会导致内存占用的上涨，所以就和卸载方法中的参数有关
 - .Unload(false)：仅仅释放这个AB包但是不释放AB包加载出来的资源，如果仅仅释放包，资源就成为游离资源，这部分内存是没有被释放的，如果继续加载同一个包，加载同一个资源，内存会再叠加上涨，**因为这个资源虽然是同名同资源，但不是从一个包出来的，所以会叠加占用内存**
 - 同一个AB包重复加载资源，不会叠加占用内存

- ==重点==： `.Unload(true)`：不仅释放AB包，也释放AB包中的资源，并且资源占用的那部分内存也会被释放，推荐使用这种方式，但是这样在使用的资源也会被卸载，所以卸载之前要确保资源已经不需要使用了，一般可以在关卡与关卡之间调用这个API，就是转场景的时候。

AB包使用的一些问题

1. 依赖包重复问题

- 把需要的共享的资源打在一起
- 分割包，这些包不是在同一时间使用的
- 把共享的资源达成一个包，其他包只去依赖这个包

2. 图集重复问题

- 需要给图片规划一个图集，然后把这个图集下的图片打进一个包

Lua

语言的执行方式

- 编译型语言：代码在运行前需要使用编译器，先将程序源代码编译为可执行文件，再执行。
 - `C/C++`
 - `Java` , `C#`
 - `Go` , `Objective-C`
- 解释型语言（脚本语言）：需要提前安装编程语言解析器，运行时使用解析器执行代码，就可以在不编译就可以直接运行，即写即运行，开发速度快，但是效率比编译型语言慢点，解释性语言可以实现热更
 - `JavaScript` (`TypeScript`)
 - `Python` , `PHP` , `Perl` , `Ruby`
 - `SQL`
 - `Lua`

Lua文本工具安装和解析器安装

- 先安装文本工具 `Sublime`
- 安装解析器
- Lua回顾网站：<https://www.runoob.com/lua/lua-tutorial.html>

Lua语言特点

- `Lua` 是一种轻量小巧的脚本语言，用标准 `C` 语言编写并以源代码形式开放，其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。
- 特点：
 - **轻量级**: 它用标准 `C` 语言编写并以源代码形式开放，编译后仅仅一百余K，可以很方便的嵌入别的程序里。
 - **可扩展**: `Lua` 提供了非常易于使用的扩展接口和机制：由宿主语言(通常是 `C` 或 `C++`)提供一部分功能，`Lua` 可以使用它们，就像是本来就内置的功能一样。

- 使用领域：
 - 游戏行业（魔兽世界的插件，愤怒的小鸟）
 - Web 领域（Nginx 功能开发）
 - 嵌入式领域
- 使用方式：
 - 需要高性能的程序部分，使用（C/C++）实现。需要快速实现功能部分，使用 Lua 实现。
- 国内游戏行业流行的 Lua 使用方式：
 - Cocos2D-X(Cocos)：核心引擎使用 C++ 实现，提供 LuaBinding（API 的 Lua 调用），游戏逻辑开发部分可使用 Lua 开发（刀塔传奇）
 - Unity3D：使用如xLua插件（API 的 Lua 调用），实现游戏核心代码使用 C# 实现（打包时 il2cpp，编译为 C++），游戏逻辑使用 Lua 开发（王者荣耀）
- Lua 支持多返回值

Lua语法

```
--打印方法
print("hello world");

-- 单行注释

--[[
    多行注释
]]

--Lua可以不加分号

--Lua的变量不需要声明变量类型
name="不需要声明变量类型"
print(name);

--可以打印没有复制的变量，打印出来的是nil，nil==null
print(id);

--销毁变量赋值为nil，变量就会销毁
name=nil;

--这个变量是一个局部变量
local data = "局部变量"
print(data)
```

Lua变量的作用域

- 当前文件：加了 local 就只在这一个文件有效，其他地方都无用
- 当前方法(函数)
- 系统全局：不加 local 就是全局变量，整个编译器下都有效，任意地方随意访问，轻易不要使用全局变量
- 如果在一个函数中加 local，那么这个变量只在这个函数的作用域中有作用

Lua函数与方法

- 在 C# 中，函数和方法是同一个东西，都必须写在类或者接口中，外面必须包裹一层

- 但是在 Lua 中，可以定义函数也可以定义方法，方法就是像 C# 一样要有一个作用域，还可以定义一个纯函数，不属于任何表

Lua保留字

and	break	do	else
else if	end	false	for
function	if	in	local
nil	not	or	repeat
return	then	true	until
while			

Lua数据类型

- type() 函数判断变量类型，这个函数的返回值是一个 string 类型

```

local name = "Unity"

--判定变量的数据类型
print(type(name))           --string

--判定变量类型是不是字符串
print(type(name)=="string") --true

--对没有声明的变量查询类型
print(type(id))             --nil

--判定一个变量是否已经定义
print(type(id)=="nil")      --true
--判定一个变量是否已经定义第二种方法
print(id==nil)              --true

--lua的数字只有number型
print(type(123))            --number
print(type(0.12))           --number

--布尔型
print(type(true))           --boolean
print(type(false))          --boolean

--单引号字符串
print(type('Unity'))        --string

--Lua一切复杂数据皆表（table）
--C#对象，结构体，数组，列表，字典
print(type({}))              --table

--只打印一个大括号是内存地址

```

```
print({}) --table: 00A0C278
```

- `nil` : 空值
- `boolean` : 布尔型
- `number` : 所有数字皆是 `number` , 无论整型字符型
- `string` : 字符型, 可以单引号和双引号
- `function` : 函数类型
- `table` : 表, 复杂数据类型

Lua字符串的操作

- `Lua` 里的下标从1开始

```
--声明单行字符串
local str1 = 'abc'
local str2 = 'def'

--多行字符串声明
--每次换行都会插入一个换行符'\n',会占一个长度
local str3 = [[
    ghi
    jkl
]]

--字符串拼接
print(str1..str2) --abcdef

--Lua会将字符串尝试转化为数字再相加, 如果字符串不是数字就会报错, 如果是数字就会等于和
print('6'+ '7') --13

--Lua获取字符串长度
print(#str1) --3

--字符串函数使用
--string.函数名 ( )

--字符串大写化
print(string.upper(str1))

--字符串小写化
print(string.lower(str2))

--字符串反转
print(string.reverse(str1))

--字符串查找(KMP算法)
--参数1: 要查找的字符串
--参数2: 查找的内容
--返回值(多返回值): 起始位置和结束位置
print(string.find("abcdef", "cd")) -- 3 5

--字符串截取
--参数1: 被截取的字符串
--参数2: 起始下标
--返回值: 从下标到结束的字符串
```



```

print(string.sub("abcdef",3))      --cdef

--带有结束位置的截取字符串
print(string.sub("abcdef",3,5))    --cde

--截取倒数第二个字符
local str4 = 'acvdefghijk'
print(string.sub(str4,3,#str4-1))

--字符串格式化
print(string.format("I'm a Player %s,Level is %d","FYX",99))
--I'm a Player FYX,Level is 99

--字符串重复
--参数1: 重复的字符串
--参数2: 重复的次数
print(string.rep("abc",3))

--字符串修改
--参数1: 要修改的字符串
--参数2: 修改的字符串里的内容
--参数3: 要替换的内容
--返回值1: 替换后的内容
--返回值2: 替换的次数
print(string.gsub("abcdef","cd","**"))
--ab**ef      1

--字符串转ASCII
print(string.byte("A"))
--ASCII转字符
print(string.char(65))

```

Lua运算符

- 没有 ++ , --
- 没有 += , -= , *= , /=
- 次方 ^
- 不等于 ~=

Lua的表[数组]

```

-- 数组实际上就是表，现阶段只考虑数字索引
-- 这样就声明了一个数组，下标从1开始
local data = {}
-- 内部存储类型可以混合
data={"abc",123,[-1]=100,[0]=true,[4]=233,one="212",["two"]=11}
print(data[-2])

-- 获取table的长度
-- 计算表长是从1开始的，并且根据连续的索引计算，如果中间有断开的索引值，计数就停止
print("数组长度: ".. #data)

--通过下标号操作
data[0]=3.14

```

```

--通过下标索引
print(data[2])
print(data["one"])
print(data.two)

--多维table
local multi={{ "abc",123},{true,nil}}

--多维表2
local multi={
    {"abc",123},
    {true,nil},
    {'hehe'}}
}

print(multi[2][3][1])

```

- 数组实际上就是表，表不止代表数组，可以当字典、列表等等来使用，现阶段只考虑数字索引
- Lua 里的表下标从1开始，索引还可以是字符串的索引，若是用了字符串索引，就相当于字典的键值
- 表中存储的数据可以混合存储，一个数组里既可以放字符串也可以放数字其他类型
- 索引可以有间隔，跳着索引值进行查询，所以跟字典很像，不需要遍历表
- 还可以指定下标号进行赋值，下标号可以是0和负数
- 如果没有赋值的下标号那这个下标号上的值就是nil
- 数组可以声明后赋值也可以直接赋值
- 如果没有提高索引的数据，下标是从1开始
- 计算表长是从1开始的，并且根据连续的索引计算，如果中间有断开的索引值，计数就停止，一般计算表长不要这种方式，不准确，一般使用迭代器
- 修改值直接通过下标进行修改，可以连类型都修改
- 支持多维表，跟多维数组意思一样，可以在表中再套表

Lua的分支判断循环控制语句

```

--if
--Lua的if判断语句没有大括号，使用then开头end结束，前面判断条件也可以不加小括号，then和end支持无限极嵌套
if true
then

end

--if else
--同样不需要大括号，再then和end中间写个else，会自动检测并把两部分分开
if false
then

else

end

--if elseif else
--if后一定要跟then，其实这两个结合就像大括号，然后结尾是else或者end
if false
then

```

```
elseif true
then

else

end

--if嵌套
if true
then
    if true
    then
        print("进入第二层if")
    end
end
end
```

- Lua 的判断语句没有大括号，if 加 then 代表判断语句起始，else 或者 end 结尾
- if 一定要跟 then
- Lua 没有 switch 分支方法，全都用 if 来写
- 判断语句都是 then 起始 end 结尾

```
--while循环，使用do开头，end结尾
while num < 4
do

end

--repeat until 用法类似于do while但不等同于do while
repeat

until --直到条件满足时跳出循环

--老司机写法，这样写在这种语言中可以提高效率，可以不写else if，并且写在until中，可以写上break
让直行一个条件满足的时候就不执行后面的代码，提高效率
repeat
    if false
    then
        print("执行第一个分支")
        break
    end

    if true
    then
        print("执行第二个分支")
        break
    end

    if false
    then
        print("执行第三个分支")
        break
    end

until true
```

```

--for循环, 直接指定起始值和终点值就可
for i=1,10
do
    print(i)
end

--for循环, 指定增长步长
for i=1,10,2
do
    print(i)
end

```

- `while`、`for` 循环语句由 `do` 起始，`end` 结尾
- `repeat..until` 用法和 `do..while` 类似，但是 `do..while` 是直到条件不满足的时候跳出循环，而 `until` 是到条件满足时跳出循环
- `for` 循环直接指定起始值和终点值即可，遍历表要从下标1开始，如果不指定增长步长的话，它会加1加1的增长
- `for` 循环的步长可以指定负数，就是递减
- `for` 循环只能遍历规则索引的表

Lua的与或非

- `and` : 与
- `or` : 或
- `not` : 非

Lua迭代器

- `ipairs` : 续数字索引迭代器
- `pairs` : 所有数据迭代器

```

--连续数字索引迭代器
--i: 下标号
--v: 数据
--ipairs(表)
for i,v in ipairs(data)
do
    print("i和v " .. i,v)
end

--所有数据迭代器, 可以遍历表中所有数据
for k,v in pairs(data) do
    print("k: " .. k, " v:" .. v)
end

```

- 连续数字索引迭代器和for循环没什么区别，同样只能遍历连续的数字索引，中间有断开就不能再继续索引
- 所有数据迭代器可以遍历所有数据，并返回下标号和对应的值

Lua函数

```

--写一个函数， function开头 end结尾
function function_name()
    -- body
end

--函数的调用
function_name()

--本地函数的声明，其实这也像是一个委托，将一个函数赋值给fun2
local fun2 = function()
    --body
end

--可变参传参
local func3 = function ( ... )
    --body
end

--可变参传参，若要让传入参数相加做操作需要将参数转化为表
--arg是一个内置参数
local func3 = function ( ... )
    local arg = {...}

    local total = 0
    --不需要第一个参数的时候可以让他定义为一个下划线
    for _,v in pairs(arg)
    do
        total=total+v
    end

    print("输出: "..total)
end

-[[
lua中函数的传参，参数不需要定义类型，而是自己约定传参，
就是自己觉得需要传递什么，随后在调用函数的时候按想法传参就行
例如下方：参数我自己约定第一个为字符串，第二个为表，第三个为函数
那么传参的时候我就要按自己约定的传递参数即可，
为了清楚，可以将参数名称定义为自己想传类型的名称缩写，这样就更清晰
]]
function Function1(str, tabels, fun)
    --body
end

--函数可以作为数据赋值，可以作为参数传递
--传参与匿名函数
Function1(
    "字符串参数1",
    A,
    function()
        --参数传递
        print("匿名函数")
    end
)

Tmep = Function1 --赋值

```

```
MyPrint = function(arg1, arg2, arg3)    --赋值，其实类似委托
    --body
end
```

- Lua 中函数定义时从 `function` 开头，`end` 结尾
- Lua 中的函数结构实际上是一个变量，所以可以实现委托的效果
- Lua 中函数的调用只能在定义的下方调用，但编译型语言可以在上方调用下方调用都行
- 解释型语言只能在定义下方调用函数，因为他们都是自上而下进行解析，编译型语言是整体编译过后再执行的，所以无论定义上方下方调用函数都可以
- 函数调用的时候，如果传入参数比定义的函数的参数多，也会正常调用，它只会用到它需要的参数，但是参数不能少，即传参只多不少
- 参数如果是三个点 `...`，就是可变参传参，类似C#的 `params`，但是无固定的参数需要转换为 `Table`

Lua多返回值与多变量赋值

```
function fun4()
    --返回两个返回值
    return 99,999
end
--用两个变量来接收这两个返回值
local num1,num2 = fun4()

print(num1)
print(num2)

--多变量赋值
local a, b = 10.20
local c, d, e = 11, 22, "33"
```

Lua表Table的常见操作

- 表不仅可以用来代表数组列表字典等，也可以将他理解为一个对象

```
local data={"abc",123,[-1]=100,[0]=true,[4]=233,one="212",["two"]=11}
--因为function是一种数据类型
--这样就相当于将函数储存在了data表中，也就是成员方法
--这个函数就可以访问这个表中的其他变量
data.fun5=function ( ... )
    -- body
end

--self等同于this，就是把自己当作参数传递过去了
--成员方法内部可以通过self获得当前表的其他数据
data.fun6=function(self)
    --取值
    print(self.one)
end

--self需要表，则将data传过去
data.fun6(data)
--冒号表示隐式传递data到fun6中
data:fun6()
```

```

--不需要在参数里写self也可以在函数内部通过self访问表中的值
--这里的意思是，冒号帮助data的fun7函数隐式声明了一个self参数，所以可以直接在内部使用self
function data:fun7()
    --取值
    print(self.one)
end

--将表里的数据全部拼接在一起，返回的是字符串
--多参数
--参数1: 表
--参数2: 拼接的分割符号
--参数3和4: 指定索引区域的拼接
table.concat(data)

--表的插入
table.insert(data, "插入的数据")
--指定索引插入
table.insert(data, 2, "指定索引插入")

--表的移除
--不是置空操作，置空操作那个索引会消失，但是后面的数据不会向前移动
table.remove(data, 2)

--表的排序，不同类型数据无法排序
table.sort(data)

```

- 有时候这个表定义在其他文件中且是 `local`，所以需要用到 `self`
- `self` 等同于 `this`
- `.` 需要显示的传递表，`:` 隐式传递表
- Lua冒号和点的区别
 - 声明时：使用点声明方法，如果在内部需要调用成员变量，则需要使用 `self`，那么方法也需要声明 `self` 参数，而冒号会帮开发者隐式声明
 - 调用时：点相当于成员变量的方式调用，如果声明的方法上有 `self` 参数，则需要显示的传递当前表，而冒号会帮开发者隐式传递

Lua中的模块module

- 类似命名空间的意思
- 在Lua中实现模块就是用表来实现，将所有的变量，函数等等放在一个表里，就实现了一个模块

```

--定义一个模块Module,该文件的文件名为config
Module = {}

Module.var = "模块中的变量"

Module.func1 = function()
    print("模块中的函数")
end

--可指定为Module表内的函数，其实也可以不指定，直接func2即可，其实都是属于该模块的
function Module.func2()

```

```

    print("模块中的函数")
end

local function func3()
    print("相当于一个私有函数")
end

return Module

```

Lua模块(文件)的使用

- 要是想在一个文件调用另一个模块(文件)的变量或者方法，使用 `require("文件路径")`
- 路径不加文件的拓展名
- 若是在同级目录下就可以直接通过文件名查找
- 若是在下级目录就通过 `/文件夹名/文件名` 路径查找
- 若是要用 `require` 加载上级目录有两步
 - 在path里添加一个访问上级目录的路径 `package.path = package.path .. ';..\?.lua'`
 - 然后就可以访问到上级目录 `require "../HeHeBaby"`
 - 因为 `require` 查找文件是在 `package.path` 中去查找，若是没有添加上级目录的路径是默认访问不到的

```

--上级目录文件HeHeBaby
package.path = package.path .. ';..\?.lua'
require "../HeHeBaby"

--文件fileResource
require("Config")
print(config["var"])

--也可以指定变量
m = require "Config"
print(m.var)

```

- 但是要访问其他文件的变量，那个变量必须是全局，这样很不安全，所以使用 `return` 返回这个变量，就可以保证这个变量只会被访问但是不会被修改，有点像属性
- `require` 不支持多返回值
- 多次执行一个文件的内部代码，所有被加载过的文件，路径信息都会记录在 `package.loaded` 中，当再次加载的时候会判断这个信息，如果这个路径信息存在，则不会再次加载，必须要清空才可以

```

--文件fileResource
require("Config")
require("Config")--只执行一次输出

--文件fileResource
require("Config")
package.loaded["Config"]=nil
require("Config") --可以两次输出了

```


Lua中C包

- 可以引入C语言的包给Lua添加一些方法，详情查看菜鸟教程

Lua元表

- 直接打印 `table`，显示的是内存地址
- 期望打印的是表的内容，便于程序员阅读
- 现在就需要扩展 `table` 的功能，实现打印便于阅读，拓展使用的就是 `metatable` 语法特性---元表

```
local t1 = {1,2,3}

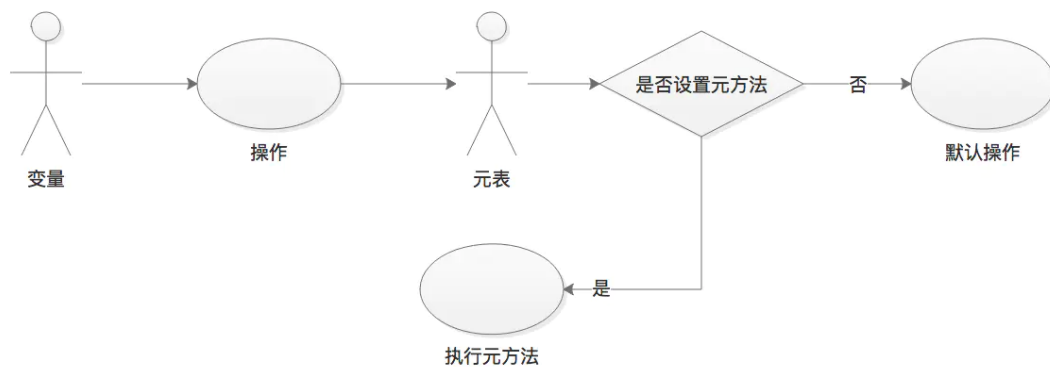
--直接打印table，显示的是内存地址
--期望打印的是'{1,2,3}'，便于程序员阅读
--现在就需要扩展t1的功能，实现打印便于阅读，拓展使用的就是metatable语法特性---元表
print(t1)    --table: 0097c228

--元表是一个table，但是这个表有特定的写法
--需求：打印利于阅读的内容，实际上是将表转化为string类型
local meta = {
    __tostring=function(t)
        local format='{ '

        for i,v in pairs(t1) do
            format = format .. tostring(v) .. ', '
        end

        format= format .. '}'
        return format
    end
}
--将meta设置为t1的功能拓展表(元表)
--设置元表的方法
setmetatable(t1,meta)
--将t1当成字符串进行打印
--如果t1设置了元表，并且元表中实现了__tostring方法，则自动调用元表的__tostring方法
print(t1)
```

- 元表的意思其实就是给某个表添加了一些方法，例如当执行打印操作时，就会先检查两个表之一中是否有元表，之后就会检查有没有一个 `__tostring` 的键，如果有就会调用对应键的值，而且对应的值往往是一个表或者一个函数，即“元方法”



- 自动调用的 `__tostring` 关键字是两个下划线 `__tostring`
- 设置元表的方法 `setmetatable(表, 元表)`，但是如果元表中有 `__metatable` 键值的话会关联失败
 - 也可已如此设置：`mytable=setmetatable({}, {})`
 - `setmetatable` 方法有返回值，返回的是普通表的数据
- `getmetatable(table)`：返回对象的元表，如果元表中有 `__metatable` 键，就会返回该键的值
- 使用 `__metatable` 可以保护元表，当 `__metatable="lock"` 时，可以禁止用户访问元表中的成员或者修改元表，此时 `getmetatable` 的时候得到的就是字符串
- 当两个表都有元表的时候，lua是怎么去调用表中的键值呢
 - 对于二元操作符，如果第一个操作数有元表，并且元表中有所需要的字段定义，比如我们这里的 `__add` 元方法定义，那么Lua就以这个字段为元方法，而与第二个值无关
 - 对于二元操作符，如果第一个操作数有元表，但是元表中没有所需要的字段定义，比如我们这里的 `__add` 元方法定义，那么Lua就去查找第二个操作数的元表
 - 如果两个操作数都没有元表，或者都没有对应的元方法定义，Lua就引发一个错误

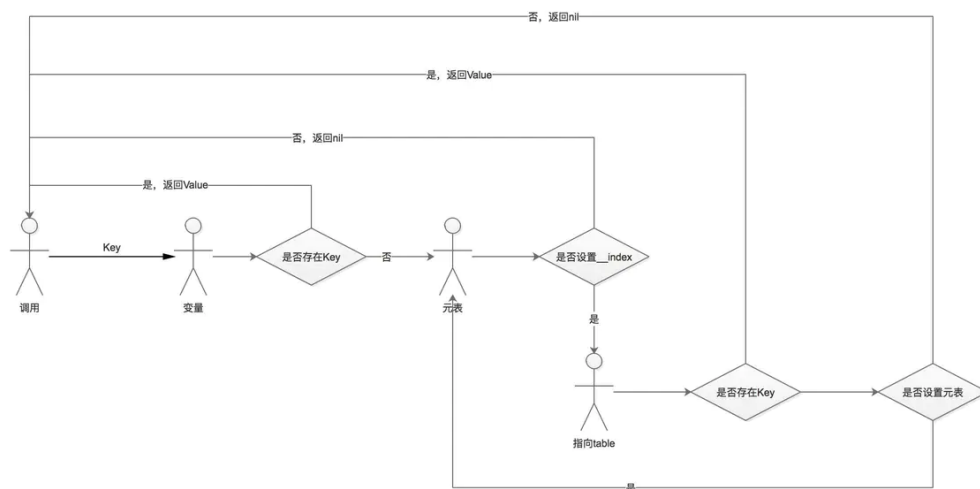
Lua元表中可设置元方法

```

__add(a, b)  --加法
__sub(a, b)  --减法
__mul(a, b)  --乘法
__div(a, b)  --除法
__mod(a, b)  --取模
__pow(a, b)  --乘幂
__unm(a)     --相反数
__concat(a, b) --连接
__len(a)     --长度
__eq(a, b)   --相等
__lt(a, b)   --小于
__le(a, b)   --小于等于
__index(a, b) --访问表中不存的字段，访问时调用，rawget(tab, i)
__newindex(a, b, c) --索引更新,更新：向表中不存在索引赋值，赋值时调用,rawset(t, key, value)
__call(a, ...) --执行方法调用，把表当作函数使用时调用，例如mytable(666)
__tostring(a) --字符串输出
__metatable  --保护元表
  
```

- `__index` 元方法
 - 当访问一个table的字段时，如果table有这个字段，则直接返回对应的值

- 当table没有这个字段，则会促使解释器去查找一个叫 `__index` 的元方法，接下来就会调用对应的元方法，返回元方法返回的值
- 如果没有这个元方法，那么就返回 `nil` 结果



- `__newindex` 元方法，当对一个table中不存在的索引赋值时，在Lua中是按照以下步骤进行的
 - Lua解释器先判断这个table是否有元表
 - 如果有了元表，就查找元表中是否有 `__newindex` 元方法；如果没有元表，就直接添加这个索引，然后对应的赋值
 - 如果有这个 `__newindex` 元方法，Lua解释器就执行它，而不是执行赋值
 - 如果这个 `__newindex` 对应的不是一个函数，而是一个table时，Lua解释器就在这个table中执行赋值，而不是对原来的table
- 有的时候，我们就不想从 `__index` 对应的元方法中查询值，我们也不想更新table时，也不想执行 `__newindex` 对应的方法，或者 `__newindex` 对应的table。那怎么办？在Lua中，当我们查询table中的值，或者更新table中的值时，不想理那该死的元表，我们可以使用 `rawget` 函数，调用 `rawget(tb, i)` 就是对table tb进行了一次“原始的（raw）”访问，也就是一次不考虑元表的简单访问；你可能会想，一次原始的访问，没有访问 `__index` 对应的元方法，可能有性能的提升，其实一次原始访问并不会加速代码执行的速度。对于 `__newindex` 元方法，可以调用 `rawset(tbl, key, value)` 函数，它可以不涉及任何元方法而直接设置table中与key相关联的value。
- 应用

```

Set = {}
mt = {} --元表

function Set.new(l)
    local set = {}
    setmetatable(set, mt)
    for _, v in ipairs(l) do
        set[v] = true --把表的数值设置为键
    end
    return set
end

--在此之后，所有由Set.new创建的集合都具有一个相同的元表，例如
--[[
    local set1 = Set.new({10, 20, 30})
    local set2 = Set.new({1, 2})
    print(getmetatable(set1))
    print(getmetatable(set2))
  
```

```

    assert(getmetatable(set1) == getmetatable(set2))
]]

=====
function Set.toString(set)
    local l = {}
    for e in pairs(set) do --单个得到的就是键，而不是值
        table.insert(l,e) --将键作为值插入l中
    end
    return "{" .. table.concat(l, ",") .. "}"
end

function Set.print(s)
    print(Set.toString(s))
end

--1 加(__add)，并集=====
function Set.union(a, b)
    --如果a的元表不是mt后者b的元表不是mt则打印错误
    if getmetatable(a) ~= mt or getmetatable(b) ~= mt then
        error("attemp to 'add' a set with a non-set value", 2)
        --error第二个参数的含义P116
    end

    local res = Set.new{}

    for k in pairs(a) do
        res[k] = true
    end

    for k in pairs(b) do
        res[k] = true
    end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}

mt.__add = Set.union

s3 = s1 + s2
Set.print(s3)

--元表混用
s = Set.new{1, 2, 3}
s = s + 8
Set.print(s + 8)

--2 乘(__mul)，交集=====
function Set.intersection(a, b)
    local res = Set.new{}
    for k in pairs(a) do
        res[k] = b[k]
    end
end

```

```

        return res
    end

    mt.__mul = Set.intersection
    --Set.print(s2 * s1)

--3 关系类=====NaN的概念=====
mt.__le = function(a, b)
    for k in pairs(a) do
        if not b[k] then
            return false
        end
    end
    return true
end

mt.__lt = function(a, b)
    return a <= b and not (b <= a)
end

mt.__eq = function(a, b)          --竟然能这么用! ? ----
    return a <= b and b <= a
end

g1 = Set.new{2, 4, 3}
g2 = Set.new{4, 10, 2}
print(g1 <= g2)
print(g1 < g2)
print(g1 >= g2)
print(g1 > g2)
print(g1 == g1 * g2)

=====
--4 table访问的元方法=====
--__index有关继承的典型示例
window = {}
window.prototype = {x = 0, y = 0, width = 100, height}
window.mt = {}

function window.new(o)
    setmetatable(o, window.mt)
    return o
end

window.mt.__index = function (table, key)
    return window.prototype[key]
end

w = window.new{x = 10, y = 20}
print(w.width)

--__index修改table默认值
function setDefault (t, d)
    local mt = {__index = function () return d end}
    setmetatable(t, mt)
end

```

```

end

tab = {x = 10, y = 20}
print(tab.x, tab.z)
setDefault(tab, 0)
print(tab.x, tab.z)

--13.4.5 只读的table
function readOnly(t)
    local proxy = {}
    local mt = {
        __index = t,
        __newindex = function(t, k, v)
            error("attempt to update a read-only table", 2)
        end
    }
    setmetatable(proxy, mt)
    return proxy
end

days = readOnly{"Sunday", "Monday", "Tuesday", "W", "T", "F", "S"}
print(days[1])
days[2] = "Noday"

```

Lua协程

线程和协程的区别

- 线程与协同程序的主要区别在于，一个具有多个线程的程序可以同时运行几个线程，而协同程序却需要彼此协作的运行。
- 在任一指定时刻只有一个协同程序在运行，并且这个正在运行的协同程序只有在明确的被要求挂起的时候才会被挂起。
- 协同程序有点类似同步的多线程，在等待同一个线程锁的几个线程有点类似协同。

基本语法

方法	描述
<code>coroutine.create()</code>	创建 <code>coroutine</code> ，返回 <code>coroutine</code> ，参数是一个函数，当和 <code>resume</code> 配合使用的时候就唤醒函数调用
<code>coroutine.resume()</code>	重启 <code>coroutine</code> ，和 <code>create</code> 配合使用
<code>coroutine.yield()</code>	挂起 <code>coroutine</code> ，将 <code>coroutine</code> 设置为挂起状态，这个和 <code>resume</code> 配合使用能有很多有用的效果
<code>coroutine.status()</code>	查看 <code>coroutine</code> 的状态 注： <code>coroutine</code> 的状态有三种： <code>dead</code> ， <code>suspended</code> ， <code>running</code> ，具体什么时候有这样的状态请参考下面的程序
<code>coroutine.wrap()</code>	创建 <code>coroutine</code> ，返回一个函数，一旦你调用这个函数，就进入 <code>coroutine</code> ，和 <code>create</code> 功能重复
<code>coroutine.running()</code>	返回正在跑的 <code>coroutine</code> ，一个 <code>coroutine</code> 就是一个线程，当使用 <code>running</code> 的时候，就是返回一个 <code>corouting</code> 的线程号

应用

```
--Lua中的协程

--定义一个协程
--参数1: 传入函数，可以直接定义一个匿名函数
Co1 =
    coroutine.create(
        function(numA, numB)
            print(numA + numB)
            print(coroutine.status(Co1))--out:running(正在运行)
            --暂停协程，跳出继续向下执行代码，yield中也可以传入参数，当作在暂停时协程的返回值
            coroutine.yield("暂停时的返回值")
            print(numA - numB)
            return "返回值"
        end
    )

print(coroutine.status(Co1))--out:suspended(暂停的未启动的)

--运行协程，启动协程
--参数:
    --1: 协程函数
    --2: 参数，传入协程函数需要的参数
--返回值: 1.Boolean执行是否成功, 2.函数的返回值
Res1, Res2 = coroutine.resume(Co1, 20, 30)
print(Res1, Res2)    --out:true,"暂停时的返回值"

print(coroutine.status(Co1))--out:suspended(暂停的未启动的)
print("协程暂停后输出")

Res3, Res4 = coroutine.resume(Co1) --继续运行被暂停的协程，可以只传函数不传参
print(Res3, Res4)    --out:true,"返回值"

print(coroutine.status(Co1)) --out:dead(死亡)

--另一个定义协程函数的方法
--该方法启动不需要通过resume来启动，可以直接调用协程即可
Co2 =
    coroutine.wrap(
        function(numA, numB)
            print(numA * numB)
        end
    )

--使用wrap创建的协程启动方法
Co2(10, 20)
```

Lua中的文件操作

- Lua `I/O` 库用于读取和处理文件。分为简单模式（和C一样）、完全模式
 - 简单模式（`simple model`）拥有一个当前输入文件和一个当前输出文件，并且提供针对这些文件相关的操作

- 完全模式 (`complete model`) 使用外部的文件句柄来实现。它以一种面对对象的形式，将所有文件操作定义为文件句柄的方法
- 简单模式在做一些简单的文件操作时较为合适。但是在进行一些高级的文件操作的时候，简单模式就显得力不从心。例如同时读取多个文件这样的操作，使用完全模式则较为合适
- 打开文件操作语句：`file = io.open (filename [, mode])`
 - `filename` 就是文件名，也可以指定路径，如果只写文件名的话就会在lua文件的同级目录下查找文件
- `mode` 的值有

模式	描述
<code>r</code>	以只读方式打开文件，该文件必须存在。
<code>w</code>	打开只写文件，若文件存在则文件长度清为0，即该文件内容会消失。若文件不存在则建立该文件。
<code>a</code>	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF符保留）
<code>r+</code>	以可读写方式打开文件，该文件必须存在。
<code>w+</code>	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
<code>a+</code>	与a类似，但此文件可读可写
<code>b</code>	二进制模式，如果文件是二进制文件，可以加上b
<code>+</code>	号表示对文件既可以读也可以写

应用

- 简单模式

```
-- 以只读方式打开文件
file = io.open("test.lua", "r")

-- 设置默认输入文件为 test.lua
io.input(file)

-- 输出文件第一行
print(io.read())

-- 关闭打开的文件
io.close(file)

-- 以附加的方式打开只写文件
file = io.open("test.lua", "a")

-- 设置默认输出文件为 test.lua
io.output(file)

-- 在文件最后一行添加 Lua 注释
io.write("-- test.lua 文件末尾注释")
```



```
-- 关闭打开的文件
io.close(file)
```

- `read` 也可以传递参数

模式	描述
"*n"	读取一个数字并返回它。例： <code>file.read("*n")</code>
"*a"	从当前位置读取整个文件。例： <code>file.read("*a")</code>
"*l" (默认)	读取下一行，在文件尾 (EOF) 处返回 <code>nil</code> 。例： <code>file.read("*l")</code>
number	返回一个指定字符个数的字符串，或在 EOF 时返回 <code>nil</code> 。例： <code>file.read(5)</code>

- `io` 的其他方法

```
--用字符串 mode 指定的模式打开一个文件，返回一个文件类型
io.open(filename: string ,[ mode: string("r")]) -> FILE*
--设置 file 为默认输入文件
io.input([file: string/FILE*])-> [FILE*]
--设置 file 为默认输出文件。
io.output([file: string/FILE*]) -> [FILE*]
--读文件 file， 指定的格式决定了要读什么，等价于 io.input():read(...)
io.read(mode: ...) -> string/number
--将参数的值逐个写入默认输出文件，等价于 io.output():write(...)
io.write(...) -> FILE* [2. errmsg: string]
--用一个分离进程开启程序 prog
io.popen(prog: string ,[ mode: string("r")]) -> file: FILE*
--将写入的数据保存到默认输出文件中，等价于 io.output():flush()
io.flush()
--如果成功，返回一个临时文件的句柄
io.tmpfile() -> FILE*
--检查 obj 是否是合法的文件句柄。
--如果 obj 它是一个打开的文件句柄，返回字符串 "file"
--如果 obj 是一个关闭的文件句柄，返回字符串 "closed file"
--如果 obj 不是文件句柄，返回 nil
io.type(obj: FILE*) -> type: string
--返回一个迭代函数，迭代函数的功能根据mode传的参数决定
io.lines([filename: string, mode: ...])
--关闭 file 或默认输出文件
io.close([file: FILE*])
```

- 完全模式

```
-- 以只读方式打开文件
file = io.open("data.txt", "r")

-- 输出文件第一行
print(file:read())

-- 关闭打开的文件
```

```

file:close()

-- 以附加的方式打开只写文件
file = io.open("data.txt", "a")

-- 在文件最后一行添加 Lua 注释
file:write("--test")

-- 关闭打开的文件
file:close()

```

Lua的错误处理/调试/垃圾回收

- [菜鸟教程](#)

Lua面向对象

```

--[[
    对象由属性和方法组成。LUA中最基本的结构是table，所以需要table来描述对象的属性
    table是引用类型

    封装：lua 中的 function 可以用来表示方法。那么LUA中的类可以通过 table + function 模拟出来。

    继承：至于继承，可以通过 metatable 模拟出来（不推荐用，只模拟最基本的对象大部分时间够用了）

    Lua 中的表不仅在某种意义上是一种对象。像对象一样，表也有状态（成员变量）；
    也有与对象的值独立的本性，特别是拥有两个不同值的对象（table）代表两个不同的对象；
    一个对象在不同的时候也可以有不同的值，但他始终是一个对象；
    与对象类似，表的生命周期与其由什么创建、在哪创建没有关系。对象有他们的成员函数，表也有
]]
--声明一个类（表）
Shape = {}

--给类（表）添加成员变量
Shape.area = 0
Shape.name = "shape"

--创建一个构造函数
function Shape:new(otable, side)
    --如果传过来一个新的表则赋值给atable，如果没有传入数据，则默认构建一个空表
    local atable = otable or {}
    --设置元表为自己，这样通过new 方法创建出来的对象（表）的元表就是Shape
    setmetatable(atable, self)
    --！！！！这是重点：给元表的__index添加值为自己
    --当创建的对象访问某个成员的时候，就会先在自身查找，如果自身查找不到，就会去元表index指向的表中去查找
    self.__index = self
    --构造函数中初始化成员
    side = side or 0
    self.area = side * side
    --返回创建的对象（新表）
    return atable
end

--设置一个析构方法

```

```

Shape.__gc = function(atable)
    --对象销毁时调用
    print(atable.name, "--destroy")
end

--给类（表）添加成员函数
function Shape:printArea()
    print("面积为", self.area)
end

-----以上就是Lua中一个类的创建-----

--创建对象（通过源表创建新表）
local myshape = Shape:new(nil, 10)

--使用成员方法
myshape:printArea()

-----以下就是Lua中的继承-----

--创建一个新类，通过源类创建，其实就是构建一个新表，和上面的创建对象差不多
Square = Shape:new()

--创建派生类的构造函数
function Square:new(otable, side)
    --这个就是若有传递表内容过来，则新表的数据就等于该表，若没有则通过父类的构造方法创建
    --类似: base.new()的意思
    local atable = otable or Shape:new(otable, side)
    --设置元表，设置index属性
    setmetatable(atable, self)
    --这里就引出了继承的效果，该类创建的对象首先会在自己的元表里查找成员属性
    --若自己没有这个属性，那么因为square又是通过shape创建的，就会去shape里查找属性
    self.__index = self
    self.name = "square"
    return atable
end

--重写方法
function Square:printArea()
    print("正方形的面积为 ", self.area)
end

--创建对象
local mysquare = Square:new(nil, 10)
mysquare:printArea()

--创建一个派生类
Rectangle = Shape:new()
--派生类的构造函数，这里可以添加新的成员
function Rectangle:new(otable, length, breadth)
    local atable = otable or Shape:new(otable)
    setmetatable(atable, self)
    self.__index = self
    self.area = length * breadth
    self.name = "rectangle"
    return atable
end

```

```
--派生类重写方法
function Rectangle:printArea()
    print("矩形面积 ", self.area)
end

--创建对象
local myrectangle = Rectangle:new(nil, 10, 20)
myrectangle:printArea()
myrectangle = nil
```

out:

```
面积为 100
正方形的面积为 100
矩形面积 200
rectangle --destroy
square --destroy
shape --destroy
```

- [参考资料](#)

Lua作业

```
--计算不规则表的长度
local table1 = {1,2,3,[0]=1,[6]=6,[9]=10,one=12,[4]={3,4,{1,2,3}}}
--递归
function count(a,con)

    if type(a)=="table" then
        --print("table")
        for _,v in pairs(a) do
            con = count(v,con)
        end
    else
        con = con + 1
    end

    return con
end

local con = 0
print("表的长度: "..count(table1,con))

--两个表的合并
local table2 = {2,6,5,7,[0]=10,two=11}
local table3 = {1,2,3,4,5}

local meta = {

    __add=function (t1,t2)
        local temp = {}

        for _,v in pairs(t1) do
            table.insert(temp,v)
        end
```

```

        for _,v in pairs(t2) do
            table.insert(temp,v)
        end

        return temp
    end,

    __tostring = function(t)

        local format='{'
        for i,v in pairs(t) do
            format = format .. tostring(v) .. ','
        end
        format= format .. '}'

        return format
    end
}

local table5 = {}
setmetatable(table5,meta)
table5 = table2 + table3
setmetatable(table5,meta)
print(table5)

--冒泡排序
local maopao = function( ... )
    local data = {...}
    for i=1,#data,1 do
        for j=i+1,#data do
            if data[i]<data[j] then
                local tmp = data[i]
                data[i]=data[j]
                data[j]=tmp
            end
        end
    end
    return data
end
end

```

热更新

- 不关闭Unity应用的前提，实现游戏资源和代码逻辑的更新。
 - 例如王者经常上线时候加载的一些小文件
- 冷更新：直接停服更新，完了需要重新下载这个游戏，一般就是大版本更新
- 因为 `Lua` 可以实时更新代码，所以用 `Lua` 实现热更

热更新

- 广义：无需关闭应用，不停机状态下修复漏洞，更新资源等，重点是更新逻辑代码。
- 狭义定义（`ios` 热更新）：无需将代码重新打包提交至 `AppStore`，即可更新客户端的执行代码，即不用下载 `app` 而自动更新程序。

- 现状：苹果禁止了 C# 的部分反射操作，禁止 JIT（即时编译，程序运行时创建并运行新代码），不允许逻辑热更新，只允许使用 AssetBundle 进行资源热更新。
 - 注意：2017年，苹果更新了热更新政策说明，上线后的项目，一旦发现使用热更新，一样会以下架处理
- 为何要热更新：缩短用户获取新版应用的客户端的流程，改善用户体验，具体到 iOS 平台的应用上，有以下几个原因
 - AppStore 的审核周期难控制
 - 手机应用更新频繁
 - 对于大型应用，更新成本太大
- 终极目标：不重新下载、不停机状态下完全变换一个应用的内容
- 每个平台如何做热更新
 - Android，PC（C#）端
 - 将执行代码预编译为 Assembly.DLL，这个文件在 Library 下的 ScriptAssemblies 文件夹的 Assembly-CSharp.dll
 - 将代码作为 TextAsset 打包进 AssetBundle
 - 运行时调用 Assembly.DLL 代码
 - 更新相应的 AssetBundle 即可实现热更新
 - iOS（Lua）
 - 苹果官方禁止 iOS 下的程序热更新；JIT 在 iOS 下无效
 - 热更新方案：Unity + Lua 插件

常见的Unity热更新插件

- sLua：最快的 Lua 插件
- toLua：由 uLua 发展而来的，第三代 Lua 热更新方案
- xLua：特性最先进的 Lua 插件，鹅厂的
- ILRuntime：纯C#实现的热更新插件

xLua的使用

- 将 xLua 的文件夹下 Assets 文件夹下的资源拷贝到项目中
- 导入命名空间 xLua;
- C#执行 Lua 代码

```
{  
    //Lua是解释型语言，所以运行Lua代码，需要在Unity中创建Lua解析器  
  
    //创建Lua解析器  
    LuaEnv luaEnv = new LuaEnv();  
  
    //将字符串作为Lua代码运行  
    luaEnv.DoString("print('hello world')");//结果LUA: hello world  
  
    //没有Lua代码再运行时，记得释放Lua解释器  
    luaEnv.Dispose();  
}
```

- C#调用 Lua 文件，必须放在 Resource 文件夹下，且必须是 txt 拓展名，因为 Unity 不支持 .lua 拓展名，但是这种系统自带的加载器很局限，必须在特点文件夹下，且还不是 Lua 文件

```
{  
    //Lua是解释型语言，所以运行Lua代码，需要在Unity中创建Lua解析器  
  
    //创建Lua解析器  
    LuaEnv luaEnv = new LuaEnv();  
  
    //使用require调用文件  
    luaEnv.DoString("require 'task' ");  
  
    //没有Lua代码再运行时，记得释放Lua解释器  
    luaEnv.Dispose();  
}
```

自定义加载器

- 自定义加载器 `LuaEnv.AddLoader()`
- 因为 `require` 是通过路径去查找的文件，所以要找到文件，就需要去自定义加载器添加路径
- 自定义加载器优先级高于内置加载器

```
{  
    //创建Lua解析器  
    LuaEnv luaEnv = new LuaEnv();  
  
    //将自定义加载器的回调函数添加到xLua中  
    //当Lua编写require代码的时候，自定义加载器的回调函数就会执行  
    luaEnv.AddLoader(MyLoader);  
  
    //test文件不在Resource文件夹下，而是在同级的一个Lua文件夹下  
    luaEnv.DoString("require 'test'");  
  
    //没有Lua代码再运行时，记得释放Lua解释器  
    luaEnv.Dispose();  
}  
byte[] MyLoader(ref string path)  
{  
    //path是回调函数街道的是require加载的那个文件的名称，没有拓张吗  
  
    //添加一个路径  
    //返回值是希望将Lua的代码文件的内容以byte[]方式返回  
    string realPath = "D:/UnityProject/xLuaProject/Assets/Lua/" + path + ".lua";  
  
    //将文件读取为byte[]  
    byte[] code = File.ReadAllBytes(realPath);  
  
    return code;  
}
```

- 但是这样添加路径太固定了，使用 `Application.dataPath` 访问到自己本身这个项目的 Asset 文件夹，所有项目内容都在 Asset 文件夹下，再通过字符串拼接去添加路径

```
string realPath = "D:/UnityProject/xLuaProject/Assets/Lua/" + path + ".lua";
//-----替换为-----
string realPath = Application.dataPath + "/Lua/" + path + ".lua";
```

- 但是还需要让加载器去判空，所以要再改造一下，如果没有找到文件就会调用自带加载器

```
byte[] MyLoader(ref string path)
{
    //path是回调函数街道的是require加载的那个文件的名称，没有拓张吗

    //返回值是希望将Lua的代码文件的内容以byte[]方式返回
    string realPath = "D:/UnityProject/xLuaProject/Assets/Lua/" + path +
".lua";

    realPath = Application.dataPath + "/Lua/" + path + ".lua";

    //通过文件路径，判断文件是否存在
    if (File.Exists(realPath))
    {
        return File.ReadAllBytes(realPath);
    }
    else
    {
        //null表示当前加载器没有加载到需要的Lua文件(xLua会继续用其他的加载器加载Lua文件)
        return null;
    }
}
```

使用AB包加载Lua脚本

- 因为AB包也不支持 .lua 拓展名，所以打进AB包的文件都需要是 .txt 文件
- 因为AB包文件夹一般是在 Asset 文件夹的同级目录，所以需要通过字符串操作获得AB包的目录

```
{
    luaEnv.AddLoader(MyLoader);
    luaEnv.DoString("require 'subfile'");
}

byte[] MyLoader(ref string path)
{
    //加载AB包
    //得到Asset的路径后减去Asset这个字符串再拼接上AB包文件夹的名称就是AB包的路径
    string abPath = Application.dataPath.Substring(0,
Application.dataPath.Length - 7) + "/AB";
    string abFile = abPath + "/lua";

    //加载Lua以TextAsset方式读取出来
    AssetBundle assetBundle = AssetBundle.LoadFromFile(abFile);
    TextAsset textAsset = assetBundle.LoadAsset<TextAsset>(path + ".lua");

    //通过文件路径，判断文件是否存在
```



```

        if (textAsset == null)
        {
            return null;
        }
        else
        {
            return textAsset.bytes;
        }
    }
}

```

单例模式的解析器LuaEnv

- 因为 `LuaEnv` 会被频繁使用，所以整个全局只需要一个解析器就够了，所以要编写一个单例模式的解析器

xLua拓展学习

[沈军老师博客][<https://shenjun-coder.github.io/LuaBook/%E7%AC%AC%E4%B8%89%E7%AB%A0%20XLua%E6%95%99%E7%A8%8B/>]

xLua与C#代码的相互调用

- `Unity` 是基于 `C#` 语言开发的，所有生命周期函数都是基于 `C#` 实现，`xLua` 本身是不存在 `Unity` 的相关生命周期函数的。如果希望 `xLua` 能够拥有生命周期函数，那么我们可以实现 `C#` 作为 `Unity` 原始调用，再使用 `C#` 调用 `Lua` 对应的方法。

```

//以下Lua调用使用的参考类
namespace Hxsd
{
    //静态类，其中包括静态变量，静态属性和静态方法
    public class TestStatic
    {
        public static int ID = 100;
        //这样写就默认写了一个属性，很方便
        public static string Name
        {
            get;
            set;
        }
        public static string OutPut()
        {
            return "static";
        }
        public static void Default(string str="abc")
        {
            Debug.Log("C#:" + str);
        }
    }
}
//非静态类，其中包括构造函数、变量、属性和方法
public class TestClassObject

```

```

{
    public string name;
    public int ID
    {
        get;
        set;
    }

    public TestClassObject()
    {

    }

    public TestClassObject(string name)
    {
        this.name = name;
    }

    public int Output()
    {
        return ID;
    }
}
}

```

Lua调用C#静态结构

- Lua 调用静态类内部结构，在 lua 中写调用 c# 的静态方法
 - CS.命名空间.类名.需要调用的结构
 - 若是没有命名空间可以直接 CS.类名

```

--调用静态类和静态结构就是通过 CS.命名空间.类名.需要调用的结构 访问
CS.Hxsd.TestStatic.ID=99
CS.Hxsd.TestStatic.Name="Unity"
print(CS.Hxsd.TestStatic.ID)
print(CS.Hxsd.TestStatic.Name)
print(CS.Hxsd.TestStatic.OutPut())

```

Lua调用C#非静态结构

- Lua 通过调用 C# 的构造方法，实现 C# 类的实例化
- 实例化之后就可以通过 实例化对象.变量/属性 访问
- 访问方法需要通过冒号去访问，因为方法内部会使用到 this 等关键字访问自己的成员变量，所以 lua 调用的时候就要用冒号将自己的实例化对象传入到方法中

```

-- Lua实例化对象
-- Lua通过调用C#的构造方法，实现C#类的实例化
local obj = CS.Hxsd.TestClassObject()
--通过对象访问属性和变量
obj.name = "名字"
obj.ID = 98
-- 在成员方法中为了保证其内部可通过this访问到成员变量，所以必须通过传递当前表的方式
print(obj:Output())
--还可以调用有参构造函数
local newObj = CS.Hxsd.TestClassObject("霸王枪")
print(newObj.name)

```

Lua调用C#继承的方法

```

//调用继承
public class TestFather
{
    public string name = "father";
    public void Talk()
    {
        Debug.Log("这是来自父对象的方法");
    }
    public virtual void Cover()
    {
        Debug.Log("这是父对象的虚方法");
    }
}

public class TestChild : TestFather
{
    public override void Cover()
    {
        Debug.Log("这是来自子对象的重写方法");
    }
}

```

- Lua 可以调用C#中继承多态的特性
- 也可以调用重载的方法，传入参数就可以了，但是 Lua 中数字不分浮点或整型，所以参数如果分整型或浮点型的化只会统一认为是数字，会调用失败
 - 所以避免使用 Lua 中可合并类型进行参数传递的重载函数

```

local father = CS.TestFather()
print(father.name)
father:Talk()
father:Cover()

local child = CS.TestChild()
child.name="child"
print(child.name)
child:Talk()
child:Cover()

```

Lua调用C#拓展的方法

```

//调用类拓展
public class TestExtension
{
    public void Output()
    {
        Debug.Log("类本身定义的方法");
    }
}
[LuaCallCSharp]
public static class MyExtension
{
    public static void Show(this TestExtension test)
    {
        Debug.Log("扩展类实现的方法");
    }
}

```

- 默认 Lua 是不能调用拓展方法的，所以需要在拓展类的上打一个标签 [LuaCallCSharp]

```

local obj = CS.TestExtension()
--调用对象本身的方法
obj:Output()
--调用拓展方法
obj:Show()

```

Lua调用C#结构体的方法

```

//结构体
public struct TestStruct
{
    public static int ID;
    public static string name;
    public string Gender
    {
        get;
        set;
    }
}

```

```

    public static string Output()
    {
        return name;
    }

    public string Talk()
    {
        return Gender;
    }
}

```

- 和类的调用时非常类似的

```

--静态结构的使用
CS.TestStruct.ID=99
CS.TestStruct.name="Unity"
print(CS.TestStruct.Output())

--实例化结构体进行调用方法，和类的调用是一样的
local obj = CS.TestStruct()
obj.Gender = "Female"
print(obj.Talk())

```

Lua使用C#枚举

```

//枚举
public enum TestEnum
{
    Male,
    Female
}

```

- 使用 `CS.枚举名.枚举值` 访问到枚举，但是这列代码的数据类型是 `userdata`
 - `CS.枚举名.__CastFrom(枚举值)`，该枚举值可以是枚举本身的值或者是字符串
- 因为 `lua` 中没有与 `enum` 对应的数据类型，所以将其放入 `userdata` 这种数据类型中，表示其是一个来自于母体语言的数据类型

```

--lua中没有与enum对应的数据类型，所以将其放入userdata这种数据类型中
--表示其是一个来自于母体语言的数据类型
print(CS.TestEnum.Male)

--其他获取枚举的方式
print(CS.TestEnum.__CastFrom(1))
print(CS.TestEnum.__CastFrom("Male"))

```

Lua使用C#泛型

```
//泛型
public class TestGenericType
{
    public void Output<T>(T data)
    {
        Debug.Log("泛型方法: " + data);
    }
    public void Output(string data)
    {
        Output<string>(data);
    }
}
```

- lua 是不支持泛型的，但是我们添加组件的时候那些方法几乎都是泛型，为了解决这个问题，unity 中是有一些重载函数的，所以我们可以使用 AddComponent 的重载函数在 lua 添加组件
- lua 中 typeof 函数是 xLua 帮我们定义的和 C# 中 typeof() 功能一样的函数

```
local obj = CS.TestGenericType()
obj:Output("Unity")

--创建一个物体在场景上
local go = CS.UnityEngine.GameObject("物体名称")
--typeof函数是xLua帮我们定义的和C#中typeof()功能一样的函数
--AddComponent因为有重载，所以我们可以使用AddComponent(Type t)方式添加组件
go:AddComponent(typeof(CS.UnityEngine.BoxCollider));
```

Lua的多返回值

```
//多返回值
public class TestOutRef
{
    public static string Func1()
    {
        return "Func1";
    }
    public static string Func2(out string r2)
    {
        r2 = "Func2 Out";
        return "Func2";
    }
}
```

- C# 中是没有多返回值的，但是 Lua 有，所以如何解决这个问题，就使用 out 关键字即可

```
print(CS.TestOutRef.Func1())
--return是第一个返回值，out是第二及n个返回值
--使用out在lua这边是不需要给函数传参的
local s1,s2 = CS.TestOutRef.Func2()
print(s1,s2)
```

- `ref` 和 `out` , 在 `xLua` 中是一样的

Lua使用C#委托

```
//C#中的委托
public delegate void TestDelegate();

public class TestDelegateClass
{
    public static TestDelegate Static;
    public TestDelegate Dynamic;

    public static void StaticFunc()
    {
        Debug.Log("静态成员方法");
    }

    public void DynamicFunc()
    {
        Debug.Log("对象成员方法");
    }
}
```

- `Lua` 去使用委托, 注册方法不仅可以注册 `C#` 代码中的方法, 还可以注册 `Lua` 脚本中的函数

```
--静态方法的注册, 跟C#中差不多
CS.TestDelegateClass.Static=CS.TestDelegateClass.StaticFunc
--使用委托也是和C#一样
CS.TestDelegateClass.Static()
--但是静态方法是和程序同生共死的, 但是Lua解析器会释放, 如果在解析器释放前不把委托置空, 就会报错, 解析器释放失败
--应该在解析器释放前, 把静态的委托变量中的委托进行置空
CS.TestDelegateClass.Static=nil

local callback = function ( )
    -- body
    print("lua中的回调函数")
end
--添加Lua自己的函数
CS.TestDelegateClass.Static=callback
CS.TestDelegateClass.Static()
CS.TestDelegateClass.Static=nil
--注册多个函数
CS.TestDelegateClass.Static=CS.TestDelegateClass.Static+callback
```

- 如何判断使用加号还是等号, 在注册前进行判空处理
- 静态委托和非静态委托的注册是一样的, 但是非静态委托是不需要进行置空处理的
- 因为静态委托变量需要手动释放, 非静态的会有C#自动帮你释放

C#与Lua代码的互相调用

C#使用Lua的变量

- `lua` 中局部变量只能在 `lua` 中使用，所以C#调用时候需要使用全局变量
- `xLua` 的解析器 `LuaEnv` 中封装了一个变量 `LuaTable _G`，存放了 `lua` 中的所有全局变量
 - `LuaTable` 是一个封装的用来与 `lua` 对照变量的一个数据类型

```
id=1
name="Unity"
isMale=true
pi=3.14
```

- 封装一个取 `Global` 的方法

```
public LuaTable Global
{
    get
    {
        //lua的解析器
        return luaEnv.Global;
    }
}
```

- 通过 `Global` 去取得 `lua` 中的全局变量

```
//加载出文件
LuaEnvSingleCase.Instance.DoString("require
'LuaCallCsharp/LuaCallClassExtend'");
//通过Global访问全局变量
int id = LuaEnvSingleCase.Instance.Global.Get<int>("id");
string name = LuaEnvSingleCase.Instance.Global.Get<string>("name");
bool isMale = LuaEnvSingleCase.Instance.Global.Get<bool>("isMale");
float pi = LuaEnvSingleCase.Instance.Global.Get<float>("pi");
```

C#使用Lua的函数

- 要在C#中使用 `lua` 函数，就要在C#代码中有个类型与之对应，才可以产生映射关系

```
--无返回值不带参
func1=function()
    print("这是Lua中的函数1")
end

--无返回值带参
func2=function(name)
    print("这是Lua中的函数2,参数是: "..name)
end

--单返回值
func3=function()
    return "这是Lua中的函数3"
end

--多返回值
```



```
func4=function()
    return "这是Lua中的函数4",99
end
```

- Lua函数的映射应该使用委托

```
//加载出文件
LuaEnvSingleCase.Instance.DoString("require
'LuaCallCsharp/LuaCallClassExtend'");
//函数都是定义在全局变量中，所以可以使用Global获得lua的全局变量(变量内部是函数结构)
Func1 func1 = LuaEnvSingleCase.Instance.Global.Get<Func1>("func1");
Func2 func2 = LuaEnvSingleCase.Instance.Global.Get<Func2>("func2");
Func3 func3 = LuaEnvSingleCase.Instance.Global.Get<Func3>("func3");
Func4 func4 = LuaEnvSingleCase.Instance.Global.Get<Func4>("func4");
//使用
func1();
func2("Unity");
Debug.Log(func3());

string name;
int id;
func4(out name, out id);
Debug.Log(name);
Debug.Log(id);
//如果释放放在这里，func1~func4的局部变量还没有被释放，释放Lua运行环境就会报错
LuaEnvSingleCase.Instance.Dispose();
//如果就想在start里释放解析器，就要手动的将func1~func4变量置空
func1 = null;
```

- 如果要释放解析器，局部变量委托就要先释放才可以释放解析器，否则就会报错
 - 第一种方法将委托变量置空
 - 第二种方法将释放方法放在另一个函数里，当这些执行完了，垃圾回收启动后再去调用这个释放方法

C#使用Lua的表

- lua 中最中要的就是表类型

```
Core={
    ID=99,
    Name="Unity",

    Func1=function()
        print("这是Lua中的Core表的func1方法")
    end,

    Func2=function(name)
        print("这是Lua中的Core表的func2方法，参数是："..name)
    end,

    Func3=function(self)
        print("这是Lua中的Core表的func3方法，成员变量Name是："..self.Name)
```

```

        end,
    }

    function Core:Func4()
        print("这是Lua中的Core表的func4方法，成员变量Name是: "..self.Name)
    end

```

- C#中使用Lua的表有两种方法，一种是导出 `LuaTable` 类型，还有一种是导出结构体类型
- 导出 `LuaTable` 类型

```

//Lua中有个隐式传递self，所以委托中必须要有能传递表类型的参数，LuaTable
public delegate void Func3Self(LuaTable self);

public void ToLuaTable()
{
    //Lua中的Table
    LuaTable core = LuaEnvSingleCase.Instance.Global.Get<LuaTable>("Core");
    Debug.Log(core.Get<int>("ID"));
    core.Set<string, string>("Name", "Unreal");
    Debug.Log(core.Get<string>("Name"));

    Func1 func1 = core.Get<Func1>("Func1");
    Func2 func2 = core.Get<Func2>("func2");
    func2("Unity");

    //因为有隐式self参数，所以必须将LuaTable再传会Lua中
    Func3Self func3 = core.Get<Func3Self>("Func3");
    Func3Self func4 = core.Get<Func3Self>("Func4");
    func3(core);
    func4(core);
}

```

- 但是这种方法性能较弱，因为 `LuaTable` 是通用数据类型，所以使用的时候需要进行一些转换，性能降低，所以就使用导出结构体类型
- 结构体要和 Lua 表进行一对一的映射
- 官方推荐的方式，是结构体，导出结构体可以避免再 `xLua` 中产生 GC，从而影响性能

```

//添加这个特性，可以让Lua中的table，导出结构体的时候不产生垃圾回收GC
[GCoptimize]
public struct TableCore
{
    [CSharpCallLua]
    public delegate void Func3Self(TableCore self);

    public int ID;
    public string Name;
    public bool IsMale;

    public Func1 func1;
    public Func2 func2;
    public Func3Self func3;
    public Func3Self func4;
}

```

```
//转换的方法
//官方推荐的方式，是结构体，导出结构体可以避免再xLua中产生GC，从而影响性能
public void ToStruct()
{
    TableCore core = LuaEnvSingleCase.Instance.Global.Get<TableCore>("Core");
    Debug.Log(core.ID);
    Debug.Log(core.Name);

    core.func1();
    core.func2("Unity");
    //因为有隐式self传参，所以必须将结构体结构再传会Lua中
    core.func3(core);
    core.func4(core);
}
```

矩阵

矩阵的定义

向量

- 点乘：计算向量夹角
- 叉乘：计算两个向量构成平面的法向量

矩阵

$$M_{32} = \begin{bmatrix} 2 & 3 \\ -8 & 12 \\ 0 & 7 \end{bmatrix}$$

- 矩阵有\$3\$行，\$2\$列，所以表示为\$M_{32}\$
- 获取固定元素\$M_{32}\$，表示获取第二行第二列元素

行矩阵

$$M_{13} = [3 \quad 9 \quad 88]$$

列矩阵

$$M_{31} = \begin{bmatrix} 3 \\ 9 \\ 88 \end{bmatrix}$$

- 矩阵实际上是一个数组存储（\$2\$维数组），向量也是一个数组（\$1\$维数组）矩阵是有行（Row）和列（Column）之分

矩阵的转置

- 矩阵的转置就是行变列列变行

$$M_{32} = \begin{bmatrix} 2 & 3 \\ -8 & 12 \\ 0 & 7 \end{bmatrix}$$

矩阵的转置：

$$M_{23}^T = \begin{bmatrix} 2 & -8 & 0 \\ 3 & 12 & 7 \end{bmatrix}$$

- 矩阵的特效： $M^{TT}=M$

矩阵的乘法

矩阵乘标量

- 矩阵和标量的乘法：矩阵的每个分量，乘以标量

$$M_{33} \times X = \begin{bmatrix} m_{11} \times X & m_{12} \times X & m_{13} \times X \\ m_{21} \times X & m_{22} \times X & m_{23} \times X \\ m_{31} \times X & m_{32} \times X & m_{33} \times X \end{bmatrix}$$

矩阵乘矩阵

- 矩阵乘法其实就是一个矩阵变换另一个矩阵
- **限制条件**：乘号左边的矩阵列数 $=$ 乘号右边的矩阵行数
 - 左列等右行
- 矩阵相乘得出的矩阵：左边矩阵的行数 \times 右边矩阵的列数
 - 左行右列
- 矩阵相乘不满足交换律，满足结合律
 - 交换律： $3 \times 4 = 4 \times 3$
 - 即矩阵不满足 $M1 \times M2 = M2 \times M1$
 - 结合律： $3 \times 4 \times 5 = (3 \times 4) \times 5 = 3 \times (4 \times 5)$
 - 即矩阵满足 $M1 \times M2 \times M3 = M1 \times (M2 \times M3)$
- 矩阵相乘运算公式

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

- 矩阵相乘技巧
 - 新矩阵的每个元素编号列出
 - 找到左边矩阵对应的行和右边矩阵对应的列，相乘再相加
 - 即 $c_{23}=a_{21} \times b_{13} + a_{22} \times b_{23}$

Unity向量乘以矩阵

- 当向量经过矩阵乘法后，我们可以理解为矩阵对向量进行了变换操作。
- 行矩阵与矩阵相乘，因为左边列数要等于右边行数，所以要左乘

$$\begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

- 列矩阵与矩阵相乘
 - Unity 矩阵与向量运算，普遍采用列矩阵右乘

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

特殊矩阵

- 方阵：行数与列数相同的矩阵，现阶段考虑 2×2 ， 3×3 ， 4×4
- 对角
 - 对角线元素：方阵中，行数和列数相同的元素，就是对角线元素
 - 即 m_{11} 、 m_{22} 、 m_{33} 行列相同的元素为对角线元素
- 非对角线元素：方阵中，除了对角线元素以外的所有其他元素

对角矩阵

- 非对角线元素的值为0，对角线元素是任意值的方阵

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

数量矩阵

- 对角线元素相等的对角矩阵

$$M = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

单位矩阵

- 对角线元素都为1的对角矩阵
 - 特性：单位矩阵乘以另一个矩阵，还是原来的矩阵

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

逆矩阵

- 逆矩阵本身就是一个变换过程
- 逆矩阵可以还原一个矩阵的变换过程

矩阵的行列式

- 由于计算逆矩阵时，行列式会作为除数。因为除法的除数不能为0，所以可以通过计算矩阵的行列式，来判定矩阵是否存在逆矩阵

- 行列式表示方式：假设有 M 矩阵， $|M|$ 表示 M 矩阵的行列式
- **注意**：行列式是一个标量
- 2×2 矩阵行列式：对角元素相乘 - 非对角元素

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

公式：

$$|M| = m_{11} \times m_{22} - m_{21} \times m_{12}$$

- 3×3 矩阵行列式：先算中间一条河，再在河两边相互对望
- 注：相乘的数一定是和中间河乘数是一样的

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

公式

$$|M| = m_{11} \times m_{22} \times m_{33} + m_{21} \times m_{32} \times m_{13} + m_{31} \times m_{12} \times m_{23} - m_{31} \times m_{22} \times m_{13} - m_{32} \times m_{23} \times m_{11} - m_{21} \times m_{12} \times m_{33}$$

代数余子式



- 余子式有正负，所以 -1^{i+j} 就是用来控制正负，数值是行列式来控制
- 算出每个分量的代数余子式组成一个新矩阵，用于转置再计算逆矩阵

标准伴随矩阵

- 代数余子式构成的矩阵的转置 C^T

逆矩阵

- 逆矩阵计算公式：逆矩阵 = 标准伴随矩阵 / 行列式
- 逆矩阵的表示方法：假设有矩阵 M ，逆矩阵就是 M^{-1}
- 逆矩阵的计算有几步
 - 先算矩阵行列式，若为 0 则不存在逆矩阵就不需要再计算
 - 计算每个位置的代数余子式
 - 将代数余子式矩阵转置，得到标准伴随矩阵
 - 标准伴随矩阵除以行列式就得到了逆矩阵
- 特点
 - 逆矩阵的逆矩阵就是原始矩阵 $M = (M^{-1})^{-1}$
 - 单位矩阵的逆矩阵就是单位矩阵本身
 - 转置矩阵的逆矩阵就是逆矩阵的转置 $(M^T)^{-1} = (M^{-1})^T$
 - 两个矩阵相乘的逆矩阵 = 后矩阵的逆矩阵 \times 前矩阵的逆矩阵 $(A \times B)^{-1} = B^{-1} \times A^{-1}$
- 逆矩阵重要的几何含义
 - 一个矩阵可以表示一个变换，而逆矩阵可以还原这个变换过程

算出一个逆矩阵

一个原始矩阵

$$M = \begin{bmatrix} 1 & 2 & 0 \\ 2 & -2 & 1 \\ 1 & 0 & -2 \end{bmatrix}$$

1.求得行列式，行列式必须不为0.....

$$|M| = 14$$

2.求得代数余子式构成的矩阵.....

$$C = \begin{bmatrix} 4 & 5 & 2 \\ 4 & -2 & 2 \\ 2 & -1 & -6 \end{bmatrix}$$

3.标准伴随矩阵 $= C^T$ ，余子式矩阵的转置.....

$$C^T = \begin{bmatrix} 4 & 4 & 2 \\ 5 & -2 & -1 \\ 2 & 2 & -6 \end{bmatrix}$$

4.逆矩阵 $M^{-1} = C^T \div |M|$

$$M^{-1} = \begin{bmatrix} 0.29 & 0.29 & 0.14 \\ 0.36 & -0.14 & -0.07 \\ 0.14 & 0.14 & -0.43 \end{bmatrix}$$

矩阵变换

- 变换 (transform)：指的是我们把一些数据，如点、方向向量甚至是颜色，通过某种方式（矩阵运算），进行转换的过程。
 - 矩阵的乘法就是一种变化过程

变换类型

线性变换

- 保留**向量加**和**标量乘**的计算
 - $f(x) + f(y) = f(x+y)$
 - $Kf(x) = f(Kx)$
 - 线性变换包含：旋转、缩放、镜像、投影，可以使用 3×3 矩阵，表示变换过程

非线性变换

- 包含平移等，可以使用 4×4 矩阵，表示变换过程

仿射变换

- 仿射变换就是合并线性变换与平移变换的变换类型，仿射变换可以使用一个 4×4 的矩阵表示。所以需要将矢量扩展到四维空间下，这就是**齐次坐标空间**，变换矩阵称为**齐次矩阵**。
 - 齐次坐标： $\begin{bmatrix} x & y & z & w \end{bmatrix}$
 - 点的列矩阵 W 分量：补1，因为点会受到平移变化的影响
 - 方向矢量列矩阵的 W 分量：补0，因为平移不会影响到方向向量的方向

齐次矩阵构成

$$\begin{bmatrix} M_{3 \times 3} & T_{3 \times 1} \\ 0_{1 \times 3} & 1 \end{bmatrix}$$

- $M_{3 \times 3}$ ：表示线性变换矩阵
- $T_{3 \times 1}$ ：表示平移变化矩阵

- $0_{1 \times 3}$ ：用于补位，使变换不受其他影响，因为如果要进行仿射变换带上平移的话 3×3 矩阵是不够的，所以要补位
- 通过齐次矩阵就可以推出以下几个变换矩阵，这些矩阵全都应用于列矩阵的相乘，即右乘一个列矩阵，如果是行矩阵就无法使用，因为行矩阵是左乘

平移矩阵

$$\begin{bmatrix} 1 & 0 & 0 & X_{\text{平移量}} \\ 0 & 1 & 0 & Y_{\text{平移量}} \\ 0 & 0 & 1 & Z_{\text{平移量}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- $M_{3 \times 3}$ ：这里是一个单位矩阵，因为平移变化是非线性，所以线性处不能影响
- $T_{3 \times 1}$ ：平移矩阵

缩放矩阵

$$\begin{bmatrix} X_s & 0 & 0 & 0 \\ 0 & Y_s & 0 & 0 \\ 0 & 0 & Z_s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- $M_{3 \times 3}$ ：缩放是线性变化，所以这里是一个线性变化矩阵
- $T_{3 \times 1}$ ：因为缩放是线性变换，所以平移变换这里为 0 ，不能对缩放产生影响

旋转矩阵

- 绕 X 轴，旋转 θ 角度

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 绕 Y 轴，旋转 θ 角度

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 绕 Z 轴，旋转 θ 角度

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

复合变换

- 例如：一个点 $P(1,1,1)$ ，需要做绕 Z 轴旋转 30° ，平移 $(5,4,2)$ ，缩放 $(3,2,1)$
 - $M_s \times M_m \times M_r \times P$
 - 即 $M_{\text{缩放}} \times M_{\text{平移}} \times M_{\text{旋转}} \times P$

- 复合变换，是存在顺序的，因为矩阵乘运算不满足乘法交换律，复合变换的顺序，决定了变换矩阵相乘的顺序
 - 所以 $M_r \times M_m \times M_s \times P$ 是不可行的，结果不一样
 - 因为矩阵满足结合律，所以可以得出变换矩阵为
 - $(M_s \times M_m \times M_r)$
 - 最后的结果是
 - $(M_s \times M_m \times M_r) \times P$
-

坐标空间

模型空间

- 模型内部点的位置都存储在模型文件内，所以点都是相对于模型空间的

世界空间

- 模型在游戏运行时，需要加载到场景中，所以点存储在世界空间中

观察（摄像机）空间

- 物体是否被投射到屏幕中，是由相机控制的，相机相对于物体的位置，决定了显示效果

裁剪空间

- 需要判定点，是否存在于摄像机裁剪视锥体下，如果存在于视锥体内，则点可以进行显示

屏幕空间

- 最终显示的设备为显示器，所以需要将点投影到显示器上，算出屏幕坐标点，由显示器显示
-

Unity着色器中常见矩阵

- `UNITY_MATRIX_MVP`：将点从模型空间下，转换到裁剪空间下，是由以下三个矩阵构成
 - `UNITY_MATRIX_M`：将点从模型空间下，转换到世界空间下
 - `UNITY_MATRIX_V`：将点从世界空间下，转换到观察空间下
 - `UNITY_MATRIX_P`：将点从观察空间下，转换到裁剪空间下
 - 以下两个互为逆矩阵
 - `_Object2World`：将点从模型到世界空间转换
 - `_World2Object`：将点从世界空间到模型转换
-

Shader着色器

CPU编程和GPU编程的区别

- `(CPU) Center Processing Unit`：逻辑编程
 - `(GPU) Graphics Processing Unit`：图形处理（矩阵运算，数据公式运算，光栅化）
 - `shader` 实际上就是 `GPU` 编程
-

GPU渲染流程

- 渲染管线

GPU渲染过程

- 渲染管道也称为渲染流水线，就是CPU准备一些数据，然后告诉GPU渲染出一个二维的图像过程。
- 渲染管道的三个阶段：应用程序阶段，几何阶段，光栅化阶段
- 应用阶段（CPU）
 - 主要就是CPU将硬盘中的数据加载后，和内存进行通讯，将需要显示的数据信息（点数据，纹理坐标，法线信息），传递给GPU处理
- 几何阶段（GPU）
 - 几何阶段最重要的工作就是“变换三维顶点坐标”，变换的工作由矩阵运算完成。
 - 物体最终显示：需要GPU进行一系列的矩阵运算，将模型空间下的网格顶点，转换到屏间下（转换点的过程是并行）
 - 非模型网格顶点信息中记录的点，GPU会以线性插值的方式获得并显示
- 光栅化阶段（GPU）
 - 使用上一个阶段（几何阶段）递过来的数据，对像素点进行染色，最终显示出来，染色过程为并行执行。

Shader着色器

- 给GPU编程的语言，代码会在GPU上并行执行（几何运算，光栅化）

Unity的Shader类型

- 固定管线 Shader：很古老显卡的编程着色器，现在很少使用，画质不好
- 顶点/片元 Shader：通用的着色器类型（重点学习），可编程
- 表面体 Shader：Unity自己创建的着色器类型，基于顶点/片元着色器二次封装实现，降低了着色器的编写难度
- 材质球和 Shader 的关系
 - Unity中Material用来关联游戏物体和Shader着色器，所以编写Shader前，需要先建立材质球

创建一个Shader

- 创建一个材质球
- 创建一个 Shader 文件，有四种类型
 - Standard Surface Shader：表面体 Shader，Unity封装的，功能强大，性能较弱
 - Unlit Shader：顶点/片元 Shader，通用，在大部分引擎都可以用
 - Image Effect Shader：图片效果 Shader
 - Compute Shader：GPU运算 Shader，不仅可以渲染图像还可以计算数据
- Shader 的代码基础结构

//着色器名称，可以在材质球上选择

```

Shader "Unlit/NewUnlitShader"
{
    //添加属性变量，才材质球中会以参数配置的方式进行显示
    Properties
    {

    }

    //可以存在多个SubShader,根据显卡性能的不同配置不同的SubShader
    //游戏引擎会从上向下执行SubShader，只会成功执行一个SubShader
    //开发人员会根据不同显卡编写不同的SubShader代码，一个SubShader执行失败就会执行下一个
    SubShader
    //可以把最新的显卡的SubShader写在上面，较老的显卡的SubShader写在下面，实现多种设备的适配
    SubShader
    {
        //Pass通道就是用来渲染图象的一个通道
        //Pass通道也可以写多个
        //一个Pass通道会产生一次绘制调用DrawCall
        //Pass通道从上往下执行且每一个通道都会执行
        Pass
        {

        }

    }
    //所有着色器都执行失败，最终使用Diffuse着色器进行显示
    Fallback "Diffuse"
}

```

固定管线Shader

- 一个基础的改变颜色的代码

```

Shader "Unlit/NewUnlitShader"
{

    Properties
    {
        //显示在材质球属性上
        //_Color是材质球变量的名称
        //"颜色"这是显示在面板上的变量文字
        //"Color"变量类型，材质球上显示为吸管
        //"(1,1,1,1)"颜色默认值
        _Color("颜色",Color) = (1,1,1,1)
    }

    SubShader
    {

        Pass
        {
            //使用固定管线着色器对物体进行染色
            //Color[]固定管线的命令
            //_Color材质球传递过来的颜色命令
            Color[_Color]
        }

    }
}

```

```
    Fallback "Diffuse"
}
```

- Shader 变量的声明为 变量名(显示在面板上的文字, 变量类型) = 默认值
- 以下是几个常用类型

```
Shader "Unlit/fixedShader"
{
    Properties
    {
        //2D纹理,white为默认值, 没有贴图就默认为白图
        _MainTex("2D纹理",2D) = "white"{}

        //rect纹理
        _RectTex("Rect纹理",Rect) = ""{}

        //Cube纹理, 应用在天空盒上
        _CubeTex("Cube纹理",Cube) = ""{}

        //用来控制一些数值, 作为阈值
        _Float("浮点数",Float) = 12

        //可以限定一个范围, 比如做一些阿尔法值得改变等等
        _Range("范围",Range(0,1)) = 1

        //齐次坐标
        _Vector("齐次坐标",vector) = (1,1,1,1)
    }
    SubShader
    {
        Pass
        {
            //设置主纹理到物体上
            SetTexture[_MainTex]
            {
                //将纹理合并到物体上
                Combine texture
            }
        }
    }
}
```

基础CPU与GPU数据传递

- 更改材质球上的参数其实就是一个简单的 CPU 传递数据到 GPU
- 上述写好 Shader 后, 将材质球赋给物体, 可以通过代码修改上面的参数

```

void Start ()
{

    //获取MeshRender上的材质球，材质球就是CPU和GPU之间的桥梁
    Material m = gameObject.GetComponent<MeshRenderer>().material;
    //这一步可以理解为将数据从CPU传递到GPU
    //参数1: 在Shader中定义的变量名称
    //参数2: 需要设定的纹理
    m.SetTexture("_MainTex", Resources.Load<Texture>(""));
}

```

着色器常用变量类型

```

//2D纹理,white为默认值，没有贴图就默认为白图
_MainTex("2D纹理",2D) = "white"{}

//rect纹理
_RectTex("Rect纹理",Rect) = ""{}

//Cube纹理，应用在天空盒上
_CubeTex("Cube纹理",Cube) = ""{}

//用来控制一些数值，作为阈值
_Float("浮点数",Float) = 12

//可以限定一个范围，比如做一些阿尔法值得改变等等
_Range("范围",Range(0,1)) = 1

//齐次坐标
_Vector("齐次坐标",Vector) = (1,1,1,1)

//类似布尔型
[Toggle(UNITY_UI_ALPHACLIP)] _BoolGray("To Gray", Float) = 0

```

顶点/片元着色器

- 顶点/片元着色器是可编程的着色器，对GPU编程

Cg语言

- Cg语言是可编程管线语言
- 可编程管线语言大概有3种
 - GLSL：支持 OpenGL 接口，使用这个开发后只能在 Linux 上运行
 - HLSL：支持 DirectX 接口，Windows 推出的，只能在微软上运行
 - Cg：C for Graphic，支持两种图形化接口，由 Nvidia 开发，Cg 允许对顶点变换和像素着色进行编程

Cg支持的7种基本数据类型

- `float` : 32 位浮点数据,一个符号位;
- `half` : 16 位浮点数据;
- `fixed` : 12 位浮点数;
- `int` : 32 位整形数据,有些 profile 会将 `int` 类型作为 `float` 类型使用;
- `bool` : 布尔数据 ;
- `string` : 字符类型 ;
- `sampler*` : 纹理对象的句柄,分为 6 类: `sampler`, `sampler1D`, `sampler2D`, `sampler3D`, `samplerCUBE` 和 `samplerRECT` ;
- Cg 是针对 GPU 编程,而 GPU 常做的是矩阵预算,即浮点数字的计算,所以常用的数据类型就是几个浮点型,其他的逻辑运算比较少,逻辑运算一般丢给 CPU 来做

Cg向量数据类型

- Cg 提供了内置的向量数据类型 (built-in vector data types), 内置的向量数据类型基于基础数据类型
 - 例如: `float4`, 表示 `float` 类型的4元向量。 `bool4`, 表示 `bool` 类型 4 元向量。
- 注意:向量最长不能超过 4 元,即在 Cg 程序中可以声明 `float1`、`float2`、`float3`、`float4` 类型的数组变量,但是不能声明超过 4 元的向量
- 向量初始化方式一般为: `float4 array = float4(1.0, 2.0, 3.0, 4.0);`

Cg矩阵数据类型

- Cg 提供矩阵数据类型,不过最大的维数不能超过 4×4 阶。
- `float1x1 matrix1` : 等价于 `float matrix1`, `x` 是字符,并不是乘号
- `float2x3 matrix2` : 表示 2×3 阶矩阵,包含 6 个 `float` 类型数据
- `float4x2 matrix3` : 表示 4×2 阶矩阵,包含 8 个 `float` 类型数据
- `float4x4 matrix4` : 表示 4×4 阶矩阵,这是最大的维数
- 矩阵的初始化方式为: `float2x3 matrix5 = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};`

Cg数组

- 在着色程序中,数组通常的使用目的是:作为从外部应用程序传入大量参数输出到 Cg 的顶点程序中的形参接口。
 - 简而言之,数组数据类型在 Cg 程序中的作用是:作为函数的形参,用于大量数据的转递。
- Cg 中声明数组变量的方式和 C 语言类似
 - `float a[10]` : 声明了一个数组,包含 10 个 `float` 类型数据
 - `float4 b[10]` : 声明了一个数组,包含 10 个 `float4` 类型向量数据
 - 对数组进行初始化的方式为: `float a[4] = {1.0, 2.0, 3.0, 4.0};`
- 获取数组长度: `float a[2] = {1.0, 2.0}; int nLen = a.length;`
- 声明多维数组以及初始化的方式如下
 - `float b[2][3] = {{0.0, 0.0, 0.0},{1.0, 1.0, 1.0}};`
- 对多维数组取长度
 - `int length1 = b.length; // length1 值为 2`
 - `int length2 = b[0].length; // length2 值为3`
- 数组和矩阵类型的区别
 - 4×4 阶数组的的声明方式为: `float M[4][4];`
 - 4×4 阶矩阵的声明方式为: `float4x4 M;`
 - 前者是一个数据结构,包含16个 `float` 类型数据,后者是一个4阶矩阵数据。
 - `float4x4 M[4]`, 表示一个数组,包含 4×4 阶矩阵数据。

- **注意：**进行数组变量声明时,一定要指定数组长度,除非是作为函数参数而声明的形参数组。

Cg结构体

- Cg 语言支持结构体 `structure` , 实际上 Cg 中的结构体的声明、使用和 C++ 非常类似 (只是类似, 不是相同)
- 声明一个结构体类型

```
struct MyAdd
{
    float val;
    float Add(float x)
    {
        return val + x;
    }
}; //结构体后要跟分号
MyAdd s; //声明一个结构体变量
```

- 注意：在当前的所有的 `profile` 版本下，如果结构体的一个成员函数使用了成员变量，则该成员变量要声明在前。此外，成员函数是否可以重载依赖于使用的 `profile` 版本。

Cg中的数据类型转换

- Cg 中的类型转换和 C 语言中的类型转换很类似。
- C 语言中类型转换可以是强制类型转换，也可以是隐式转换，如果是隐式转换，则数据类型从低精度向高精度转换。在 Cg 语言中也是如此

```
float a = 1.0;
half b = 2.0;
float c = a+b; //等价于 float c = a + (float)b
```

- Cg 语言中对于常量数据可以加上类型后缀，表示该数据的类型
- 常量的类型后缀 (`type suffix`) 有 3 种
 - `f` : 表示 `float`
 - `h` : 表示 `half`
 - `x` : 表示 `fixed`

```
float a = 1.0;
float b = a + 2.0h; //2.0h 为 half 类型常量数据,运算是需要做类型转换
```

Cg中的关系符

- 关系操作符，用于比较同类型数据之间的大小关系或者等价关系（不同类型的基础数据需要进行类型转换，不同长度的向量，不能进行比较）
- 支持的关系符和 C 语言中差不多
 - `<` : 小于
 - `<=` : 小于或等于
 - `!=` : 不等于
 - `==` : 等于
 - `>=` : 大于等于
 - `>` : 大于

Cg向量bool逻辑运算

- Cg 语言表达式允许对向量使用所有的 `boolean operator`，如果是二元操作符，则被操作的两个向量的长度必须一致。表达式中向量的每个分量都进行一对一的运算，最后返回的结果是一个 `bool` 类型的向量，长度和操作数向量一致

```
float3 a = float3(0.5, 0.0, 1.0);
float3 b = float3(0.6, -0.1, 0.9);
bool3 c = a < b;
//运算后向量c的结果为bool3(true, false, false);
```

- Cg 语言中有3种逻辑操作符（也被称为 `boolean operators`），逻辑操作符运算后的返回类型均为 `bool` 类型
 - 逻辑与：`&&`
 - 逻辑或：`||`
 - 逻辑非：`!`
- 逻辑操作符也可以对向量使用，返回的变量类型是同样长度的内置 `bool` 向量
- Cg 中的 `&&` 和 `||` 不存在 C 中的短路现象（`short-circuiting`），即只用计算一个操作数的 `bool` 值即可），而是参与运算的操作数据都进行 `bool` 分析。
 - 短路现象就是当一个条件满足的时候后续条件就不会再去执行

Cg数学操作符

- Cg 语言对向量的数学操作提供了内置的支持，Cg 中的数学操作符有
 - `*`、`/`、`-`、`+`、`%`、`++`、`--`、`*=`、`/=`、`+=`、`-=`
 - 后面四种运算符有时被归纳入赋值操作符，不过它们实际上进行数学计算,然后进行赋值，所以这里也放入数学操作符中进行说明
 - 需要注意的是：求余操作符 `%`。只能在 `int` 类型数据间进行，否则编译器会提示错误信息：
`error C1021: operands to “%” must be integral.`

Cg位移操作符

- Cg 语言中的移位操作符，功能和 C 语言中的一样，也可以作用在向量上，但向量类型必须是 `int` 类型。

```
int2 a = int2(0.0,0.0);
int2 b = a>>1;
```

Cg的Swizzle操作符

- 可以使用 Cg 语言中的 `swizzle` 操作符（`.`）将一个向量的成员取出组成一个新的向量。
`swizzle` 操作符被 GPU 硬件高效支持。
- `swizzle` 操作符后接 `x`、`y`、`z`、`w`，分别表示原始向量的第一个、第二个、第三个、第四个元素
- `swizzle` 操作符后接 `r`、`g`、`b`、`a` 的含义与前者等同。不过为了程序的易读性，建议对于表示颜色值的向量，使用 `swizzle` 操作符后接 `r`、`g`、`b`、`a` 的方式。取得值是一样的
- `Swizzle` 操作符只能对结构体和向量使用，不能对数组使用。

```
float4(a, b, c, d).xyz;// 等价于 float3(a, b, c),即取出来的是一个多元向量
float4(a, b, c, d).xyy;// 等价于 float3(a, b, b)
float4(a, b, c, d).wzyx;// 等价于 float4(d, c, b, a)
float4(a, b, c, d).w;// 等价于 float d
```


Cg条件运算符

```
expr1 ? expr2 : expr3;
```

```
if(a < 0)
{
    b = a;
}
else
{
    c = a;
}
```

Cg控制流语句

- Cg 中的控制流语句和循环语句与 C 语言类似
- 条件语句有：if、if-else、if-else 、if-else;
- 循环语句有：while、for;
- break、continue 语句可以在 for 和 while 语句中使用。

Cg关键字

- Cg 中的关键字很多都是照搬 C\C++ 中的关键字，不过 Cg 中也创造了一系列独特的关键字，这些关键字不但用于指定输入图元的数据含义(是位置信息,还是法向量信息)，本质也则对应着这些图元数据存放的硬件资源(寄存器或者纹理)，称之为语义词(Semantics)，通常也根据其用法称之为绑定语义词(binding semantics)
- 除语义词外，Cg 中还提供了三个关键字：in、out、inout。用于表示函数的输入参数的传递方式，称为**输入/输出**关键字，这组关键字可以和语义词合用表达硬件上不同的存储位置，即同一个语义词,使用 in 关键字修饰和 out 关键词修饰，表示的图形硬件上不同的寄存器
- Cg 语言还提供两个修饰符：uniform 用于指定变量的数据初始化方式；const 关键字的含义与 C\C++ 中相同，表示被修饰变量为常量变量

Cg输入数据流Uniform

- Cg 语言将输入数据流分为两类
 - Varying inputs，即数据流输入图元信息的各种组成要素
 - Uniform inputs，表示一些与三维渲染有关的离散信息数据，这些数据通常由应用程序传入，并通常不会随着图元信息的变化而变化,如材质对光的反射信息、运动矩阵等。
- 需要注意的一点是：uniform 修饰的变量的值是从外部传入的，所以在 Cg 程序(顶点程序和片段程序)中通常使用 uniform 参数修饰函数形参，不容许声明一个用 uniform 修饰的局部变量

Cg输入输出修饰符

- 参数传递是指:函数调用实参值初始化函数形参的过程
- in：修饰一个形参只是用于输入,进入函数体时被初始化,且该形参值的改变不会影响实参值,这是典型的值传递方式
- out：修饰一个形参只是用于输出的,进入函数体时并没有被初始化，这种类型的形参一般是一个函数的运行结果
- inout：修饰一个形参既用于输入也用于输出,这是典型的引用传递

Cg输入语义和输出语义的区别

- 语义：是两个处理阶段(顶点程序、片段程序)之间的输入 / 输出数据和寄存器之间的桥梁，同时语义通常也表示数据的含义，如 POSITION 一般表示参数种存放的数据是顶点位置。

- 只对两个处理阶段的输入 / 输出数据有意义，也就是说语义只有在入口函数中才有效，在内部函数(一个阶段的内部处理函数，和下一个阶段没有数据传递关系)的无效，被忽略。
- 分为输入语义和输出语义，输入语义和输出语义是有区别的

常用输入语义

- POSITION :
- NORMAL :
- BINORMAL :
- BLENDINDICES :
- BLENDWEIGHT :
- TANGENT :
- PSIZE :
- TEXCOORD0-TEXCOORD7 :
- 参数使用语义使用举例：`in float4 modelPos: POSITION`
 - 表示该参数中的数据是顶点位置坐标(通常位于模型空间)，属于输入参数，语义词 POSITION 是输入语义，如果在 OpenGL 中则对应为接受应用程序传递的顶点数据的寄存器(图形硬件上)

输出语义

- 顶点程序的输出数据被传入到片段程序中，所以顶点着色程序的输出语义词，通常也是片段程序的输入语义词，不过语义词 POSITION 除外。
- 这些语义词适用于所有的 Cg vertex profiles (几何阶段运行的回调函数) 作为输出语义和 Cg fragment profiles (光栅化阶段运行的回调函数) 的输入语义：POSITION、PSIZE、FOG、COLOR0-COLOR1、TEXCOORD0-TEXCOORD7

顶点着色程序输出语义

- 顶点着色程序必须声明一个输出变量，并绑定 POSITION 语义词，该变量中的数据将被用于，且只被用于光栅化！如果没有声明一个绑定 POSITION 语义词的输出变量就会报错
- 示例代码

```
void main_v(float4 position: POSITION,
            out float4 oposition:POSITION,//输出语义
            uniform float4x4 modelViewProj)
{
    //oposition = mul(modelViewProj,position);
}
```

- 为了保持顶点程序输出语义和片段程序输入语义的一致性，通常使用相同的 struct 类型数据作为两者之间的传递,这是一种非常方便的写法，推荐使用

```

struct VertexIn
{
    float4 position : POSITION;
    float4 normal : NORMAL;
};
struct VertexScreen
{
    float4 oPosition : POSITION;
    float4 objectPos : TEXCOORD0;
    float4 objectNormal : TEXCOORD1;
};

```

- 注意:当使用 `struct` 结构中的成员变量绑定语义时，需要注意到顶点着色程序中使用的 `POSITION` 语义词，是会被片段程序所使用的。
- 如果需要从顶点着色程序向片段程序传递数据，例如顶点投影坐标、光照信息等，则可以声明另外的参数,绑定到 `TEXCOORD` 系列的语义词进行数据传递，实际上 `TEXCOORD` 系列的语义词通常都被用于从顶点程序向片段程序之间传递数据
- 当然，也可以选择不使用 `struct` 结构，而直接在函数形参中进行语义绑定。无论使用何种方式，都要记住 `vertex program` 中的绑定语义（`POSITION` 除外）的输出形参中的数据会传递到 `fragment program` 中绑定相同语义的输入形参中。

片段着色程序输出语义

- 片段着色程序的输出语义词较少，通常是 `COLOR`。这是因为片段着色程序运行完毕后，就基本到了 `GPU` 流水线的末端了。片段程序必须声明一个 `out` 向量(三元或四元)，绑定语义词 `COLOR`，这个值将被用作该片断的最终颜色值

```

void main_f(out float4 color : COLOR)
{
    color.xyz = float3(1.0,1.0,1.0);
    color.w = 1.0;
}

```

语义绑定方法

- 入口函数输入/输出数据的绑定语义有四种方法
- 绑定语义放在函数的参数列表的参数声明后面

```
[const][in | out | inout] <type><identifier> [:<binding-semantic>][=<initializer>]
```

例如:

```
out float4 color:COLOR
```

- 绑定语义可以放在结构体的成员变量后面

```
struct <struct-tag>
{
    <type><identifier> [:<binding-semantic >];
};
```

例如:

```
struct VertexIn
{
    float4 position : POSITION;
    float4 normal : NORMAL;
};
```

- 绑定语义词可以放在函数声明的后面

```
<type> <identifier> (<parameter-list>) [:<binding-semantic>]
{
    <body>
}
```

例如:

```
float4 vert(float4 v):SV_POSITION
{
    //body
}
```

- 最后一种语义绑定的方法是,将绑定语义词放在全局非静态变量的声明后面

```
<type> <identifier> [:<binding-semantic>][=<initializer>];
```



例如:


```
float v : POSITION
```

Cg函数

- Cg 语言中的函数声明形式与 C\C++ 中相同,由返回类型(return type)、函数名、形参列表(parameter list,位于括号中,并用逗号分隔的参数表)和函数体组成,函数体包含在花括号中。如果没有返回值,则函数的返回类型是 void
- Cg 语言支持函数重载(Functon Overlaoding),其方式和 C++ 基本一致,通过形参列表的个数和类型来进行函数区分
- 所谓入口函数,即一个程序执行的入口,例如 C\C++ 程序中的 main() 函数。通常高级语言程序中只有一个入口函数,不过由于着色程序分为顶点程序和片断程序,两者对应着图形流水线上的不同阶段,所以这两个程序都各有一个入口函数。

Cg标准函数库

- 和 C 的标准函数库类似, Cg 提供了一系列内建的标准函数。这些函数用于执行数学上的通用计算或通用算法(纹理映射等)
- Cg 标准函数库主要分为五个部分
 - 数学函数(Mathematical Functions)
 -  数学函数
 - 几何函数(Geometric Functions)
 -  几何函数

- 纹理映射函数 (TextureMapFunctions)
 -  纹理函数
- 偏导数函数 (Derivative Functions)
- 调试函数 (Debugging Function)

Cg程序

- 注：变量需要导入到 Cg 代码段中

```
Shader "Unlit/fixedShader"
{
    Properties
    {
        _Color("颜色",Color)={1,1,1,1}
    }
    SubShader
    {
        Pass
        {
            //告诉Shader，我需要编写Cg代码
            //Cg开始标记
            CGPROGRAM
            //告诉GPU，vert是几何阶段运行的回调函数，几何阶段的代码写在内部
            #pragma vertex vert

            //告诉GPU，frag是光栅化阶段运行的回调函数，光栅化阶段的代码写在内部
            #pragma fragment frag

            //将材质球中的属性导入到Cg代码段中
            fixed4 _Color;

            //几何阶段需要将存储在模型空间内的点转换到裁剪空间
            float4 vert(float4 v)
            {

            }

            //光栅化阶段对像素点进行染色
            fixed4 frag()
            {

            }
            //Cg结束标记
            ENDCG
        }
    }
}
```

基础顶点着色器

```
Pass
{
    CGPROGRAM
```

```

#pragma vertex vert
#pragma fragment frag

//顶点着色器
//几何阶段需要将存储在模型空间内的点转换到裁剪空间
//参数：来自于CPU的当前需要渲染点的模型空间下的位置，POSITION用来描述参数v，表示当前点的位置来自于CPU传递，模型空间
//返回值：裁剪空间下的点的位置（屏幕空间的转换时GPU完成），SV_POSITION用来描述顶点着色器返回值，告诉GPU当前点已经转换到裁剪空间下
float4 vert(float4 v: POSITION):SV_POSITION
{
    //返回值将转后的顶点传递给GPU，光栅化阶段就会使用转换后的顶点
    return mul(UNITY_MATRIX_MVP,v);
}

EBDCG
}

```

基础片元着色器

```

Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    //片元、片段着色器（一切模型上的像素点都会执行）
    //光栅化阶段对像素点进行染色
    //参数：来自于GPU顶点着色器，SV_POSITION修饰参数，来自于顶点着色器
    //返回值：当前需要渲染点的颜色，SV_TARGET修饰返回值，告诉GPU返回值是当前需要染得颜色
    fixed4 frag(float4 v:SV_POSITION):SV_TARGET
    {
        return _Color;
    }

    EBDCG
}

```

- 首先需要导入变量到 `cg` 代码中才能使用
- 其次要编写顶点着色器，传递位置将位置信息返给片元着色器
- 片元着色器进行染色

表面体着色器

- 表面体着色器是Unity自己封装的一个着色器，自带一些效果例如光照，只需要添加函数或者修改就可以实现效果

```

Shader "Custom/NewSurfaceShader" {
    Properties {
        _Color ("Color", color) = (1,1,1,1)
    }
    SubShader {

        CGPROGRAM

```

```
// 设定表面着色器为surf,Standard: 使用标准光照
#pragma surface surf Standard fullforwardshadows
// 支持DirectX最低到9.0
#pragma target 3.0

struct Input
{
    float2 uv_MainTex;
};

//导入颜色变量
fixed4 _Color;
//参数o因为是inout类型，所以颜色对o赋值，会传递给GPU进行染色，模型空间到裁剪空间的转换，便面提着色器已经实现了
void surf (Input IN, inout SurfaceOutputStandard o)
{
    //Albedo存有材质的颜色
    o.Albedo = _Color.rgb;
    //Alpha存有颜色的透明度
    o.Alpha = _Color.a;
}
ENDCG
}
Fallback "Diffuse"
}
```

去色效果实现

- 参考文章：<https://blog.csdn.net/lyh916/article/details/45726273>
- 在Unity官网上下载对应版本的Shader包
- 精灵图片都有默认的Shader，在包里找到这个默认Shader，拖到项目中，这样就可以修改默认Shader了
- 在片元着色器中添加一行代码然后修改返回值

```
//灰色算法处理
fixed gray = (color.r + color.g + color.b) / 3.0;
return fixed4(gray, gray, gray, color.a);
```

Shader光照

标准光照的构成结构

- 自发光：材质本身发出的光，模拟环境使用的光
- 漫反射光：光照到粗糙材质后，光的反射方向随机，还有一些光发生了折射，造成材质表面没有明显的光斑



- 高光反射光：光照到材质表面后，无（低）损失直接反射给观察者眼睛，材质表面能观察到光斑



- 环境光：模拟场景光照（简单理解为太阳光）

- 裴祥凤提出的光照理论：标准光照 = 自发光 + 漫反射光 + 高光反射光 + 环境光
- 这个理论是模拟光照效果，并不是真实效果
- 以他的名字命名：Phong光照模型

逐顶点光照和逐像素光照

- 顶点着色器：会在模型渲染点上运行，其他的点会线性插值
- 片元着色器：会在模型的所有像素点上运行
- 逐顶点光照：会在顶点着色器上进行光照运算（高洛德着色）
- 逐像素光照：会在片元着色器上运行光照运算（Phong着色）
- 逐像素比逐顶点效果好，逐顶点比逐像素性能好
- 逐像素光照一般用在距离相机较劲的模型上，逐顶点一般用在距离相机有一定距离的模型上

漫反射光照模型

- 漫反射光照（兰伯特定律）

漫反射光照 = 光源的颜色 × 材质的漫反射颜色 × $MAX(0, \text{标准化后物体表面法线向量} \cdot \text{标准化后光源方向向量})$

- 几个操作数的取得
 - 光源颜色：场景中光 `GameObject` 取得
 - `_LightColor0.rgb`
 - 材质的漫反射颜色：材质球配置
 - 通过材质球变量去配置
 - `Max` 是数学函数，标准化也有函数
 - 标准化函数：`normalize`
- 表面法线向量：`CPU` 加载模型后，传递到 `GPU` 中的
 - 就是点的法线向量的意思
 - 通过模型空间下被渲染的点的法向量转化到世界空间下得到
 - `mul((float3x3)unity_ObjectToWorld, 模型空间下被渲染的点的法向量)`
 - 模型空间下被渲染的点的法向量由 `CPU` 传过来
 - 光源方向向量：场景中光 `GameObject` 取得
 - 就是光的方向
 - `_worldSpaceLightPos0.xyz`
 - 世界空间下标准化后光源方向向量：`normalize(_worldSpaceLightPos0.xyz)`
 - 取得所有数据后拿公式计算
- 漫反射逐顶点光照，所有运算都丢给顶点着色器

```
Pass
{
    //开启前向光照渲染模型，用来计算光照
    Tags{"LightMode"="ForwardBase"}

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
```



```

//需要依赖于Unity的光照Cg文件，获得光照相关的参数：光源颜色，光照方向
#include "Lighting.cginc"
//导入数据变量
fixed4 _Color;

//因为要传入和传出多个参数，所以使用结构体会比较方便
//传入参数结构体，CPU传递给顶点着色器的数据
struct c2v
{
    float4 vertex : POSITION;//来自于模型空间下被渲染的点的位置
    float3 normal : NORMAL;//来自于模型空间下被渲染的点的法向量
};

//传出参数结构体，顶点着色器传递给片元着色器的数据
//因为GPU需要知道传递的数据是什么含义，所以需要使用语义修饰被传递的数据
struct v2f
{
    float4 pos : SV_POSITION;//经过矩阵运算后，裁剪空间下点的位置
    fixed3 diffuseColor : COLOR;//经过兰伯特定律运算后，当前点的颜色
};

//参数要取得位置和法线
v2f vert(c2v data)
{
    //必须要做的：将当前需要渲染的点从模型空间转换到裁剪空间下
    float4 pos = UnityObjectToClipPos(data.vertex);
    //漫反射光照 = 光源的颜色 * 材质的漫反射颜色 * MAX(0, 标准化后物体表面法线向量 · 标准化后光源方向向量)

    //光的颜色
    fixed3 lightColor = _LightColor0.rgb;
    //材质的漫反射颜色
    fixed3 diffuseColor = _Color.rgb;
    //世界空间下标准化后物体点的表面法线向量
    //CPU拿到的法线来自于模型空间，但是光照发生在世界空间下，所以需要将法线从模型空间转换
    到世界空间下
    //unity_ObjectToWorld是4*4齐次矩阵，法线是3*1列矩阵，所以需要
    unity_ObjectToWorld强转为3*3矩阵
    fixed3 worldNormal = normalize(mul((float3x3)unity_ObjectToWorld,
    data.normal));
    //世界空间下标准化后光源方向向量
    fixed3 worldLightDir = normalize(_worldSpaceLightPos0.xyz);
    //套入兰伯特公式
    fixed3 lanbote = lightColor * diffuseColor*max(0, dot(worldNormal,
    worldLightDir));
    //使用环境光提亮下计算出的颜色
    fixed3 color = UNITY_LIGHTMODEL_AMBIENT.rgb + lanbote;

    v2f result;
    result.pos = pos;
    result.diffuseColor = color;

    //此处由于需要传递两个参数到片元着色器中，所以使用结构体传参
    return result;
}

fixed4 frag(v2f data):SV_TARGET
{

```

```

        return fixed4(data.diffuseColor,1.0);
    }

    ENDCG
}

```

- 逐像素光照，把几何运算丢给顶点着色器，而光照运算放在片元着色器

```

Pass
{

    Tags{"LightMode"="ForwardBase"}

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "Lighting.cginc"
    fixed4 _Color;

    struct c2v
    {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
    };

    struct v2f
    {
        float4 pos : SV_POSITION;
        //返回标准化后物体表面法线向量
        fixed3 worldNormal : NORMAL;//返回值不同，所以要修改返回值的结构体
    };

    v2f vert(c2v data)
    {
        v2f result;

        //因为渲染流水线分为两个阶段，vert()执行的是几何阶段的工作
        //模型空间的裁剪空间转换属于集合运算，所以还得卸载顶点着色器中
        result.pos = UnityObjectToClipPos(data.vertex);
        //法线变化属于集合运算，也需要放在顶点着色器中
        result.worldNormal = normalize(mul((float3x3)unity_ObjectToWorld,
data.normal));

        return result;
    }
    //因为是逐像素光照，所以光照运算应该写在片元着色器中
    fixed4 frag(v2f data):SV_TARGET
    {

        fixed3 worldLightDir = normalize(_worldspaceLightPos0.xyz);
        fixed3 lanbote = _LightColor0.rgb * _Color.rgb * max(0,
dot(data.worldNormal, worldLightDir ));
    }
}

```

```

        fixed3 color = UNITY_LIGHTMODEL_AMBIENT.rgb + 1*anbote;
        return fixed4(color,1.0);
    }
    ENDCG
}

```

- 半兰伯特定律：将整体颜色降低一半，再加一半，就是亮部不会受到影响，但是暗部会提亮
 - 这个 Shader 只需要把公式替换一下即可

漫反射光照 = 光源的颜色 × 材质的漫反射颜色 × ((标准化后物体表面法线向量 · 标准化后光源方向向量) × 0.5 + 0.5)

- 使用半兰伯特是解决兰伯特暗部过暗的问题

高光反射光照模型

- 高光反射光照公式

高光光照 = 光源的颜色 × 材质的高光反射颜色 × $MAX(0, \text{标准化后的观察方向向量} \cdot \text{标准化后光的反射方向})$ × 光照系数

- 几个操作数的取得
 - 光源颜色：场景中光 GameObject 取得
 - `_LightColor0.rgb`
 - 材质的高光反射颜色：材质球配置
 - 观察方向向量：相机观察点到当前渲染的点坐标所构成的方向
 - 当前需要渲染的点的坐标 - 摄像机的位置
 - `worldPoint.xyz - _WorldSpaceCameraPos.xyz`
 - 然后要标准化：`normalize(worldPoint.xyz - _WorldSpaceCameraPos.xyz)`
 - 光反射方向：用函数 `reflect` (标准化后光源方向向量 **I**, 标准化后物体表面法线向量 **N**) 计算
 - 即 `reflect` (标准化入射光的方向 **I**, 标准化当前点的法线方向 **N**)
 - **I** 和 **N** 需要归一化，即标准化
 - 标准化入射光的方向：`normalize(_worldSpaceLightPos0.xyz)`
 - 标准化当前点的法线方向：`normalize(mul((float3x3)unity_ObjectToWorld, 模型空间下被渲染的点的法向量))`
 - 标准化后光的反射方向：`normalize(reflect(I,N))`
 - 光晕系数：材质球配置
 - 光晕大小会和这个系数有关
 - 指数函数 `pow`(底数, 指数)
- 高光反射逐顶点运算

```

Pass
{

    //开启前向光照渲染模型，用来计算光照
    Tags{"LightMode"="ForwardBase"}

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "Lighting.cginc"

```

```

//导入数据变量
fixed4 _Color;
float _Gloss;

struct c2v
{
    float4 vertex : POSITION;//用于计算观察方向，模型空间
    float3 normal : NORMAL;//用于计算光的反射方向，模型空间
};

struct v2f
{
    float4 pos : SV_POSITION;//裁剪空间下的点
    fixed3 specularColor : COLOR;//高光反射算出来的颜色
};

v2f vert(c2v data)
{
    v2f result;
    //高光光照 = 光源的颜色 x 材质的高光反射颜色 x MAX(0, 标准化后的观察方向向量 · 标准化后光的反射方向) ^光照系数

    //1.光的颜色
    fixed3 lightColor = _LightColor0.rgb;
    //2.材质的高光材质颜色
    fixed3 specularColor = _Color.rgb;

    //GPU需要知道裁剪空间下的点的位置（GPU会给他做投影）
    float4 clipPoint = UnityObjectToClipPos(data.vertex);
    //为什么要取世界空间下的点，因为需要这个点计算摄像机的观察方向
    //如何将这个点转换到世界空间下，用unity_ObjectToWorld矩阵来转
    //为什么不使用float3x3转换，因为被转换的点带有齐次坐标，所以需要4x4矩阵
    float4 worldPoint = mul(unity_ObjectToWorld,data.vertex);
    //3.标准化后的观察方向向量的计算（当前需要渲染的点的位置-摄像机的位置）
    fixed3 viewDir = normalize(worldPoint.xyz - _WorldSpaceCameraPos.xyz);

    //世界空间下标准化后光源方向向量
    fixed3 worldLightDir = normalize(_worldSpaceLightPos0.xyz);
    //为什么需要世界空间下的法线，因为需要在世界空间下计算光的反射方向
    //为什么需要normalize，因为Cg的函数reflect函数需要的法线必须是标准化后的
    fixed3 worldNormal = normalize(mul((float3x3)unity_ObjectToWorld,
data.normal));
    //4.标准化后光的反射方向 reflect（入射光的方向I，当前点的法线方向N）
    fixed3 lightRefDir = normalize(reflect(worldLightDir,worldNormal));

    //套入高光反射公式
    fixed3 specular = lightColor * specularColor * pow(max(0, dot(viewDir,
lightRefDir)) ,_Gloss);

    //使用环境光提亮下计算出的颜色
    fixed3 color = UNITY_LIGHTMODEL_AMBIENT.rgb + specular;

    result.pos = clipPoint;
    result.specularColor = color;

    return result;
}

```

```

//因为是逐像素光照，所以光照运算应该写在片元着色器中
fixed4 frag(v2f data):SV_TARGET
{
    return fixed4(data.specularColor,1.0);
}
ENDCG
}

```

- 高光反射逐像素运算

```

Pass
{

    Tags{"LightMode"="ForwardBase"}

    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #include "Lighting.cginc"
    fixed4 _Color;
    float _Gloss;

    struct c2v
    {
        float4 vertex : POSITION;//用于计算观察方向，模型空间
        float3 normal : NORMAL;//用于计算光的反射方向，模型空间
    };
    struct v2f
    {
        float4 pos : SV_POSITION;//裁剪空间下的点
        float4 worldPosition:TEXCOORD0;//借用纹理的float4语义将计算好的世界空间下的顶
点，传递到片元着色器中
        fixed3 specularNormal : NORMAL;//标准化后物体表面法线向量
    };
    //参数要取得位置和法线
    v2f vert(c2v data)
    {
        v2f result;

        float4 clipPoint = UnityObjectToClipPos(data.vertex);
        float4 worldPoint = mul(unity_ObjectToWorld,data.vertex);
        fixed3 worldNormal = normalize(mul((float3x3)unity_ObjectToWorld,
data.normal));

        result.pos = clipPoint;
        result.worldPosition = worldPoint;
        result.specularNormal = worldNormal;

        return result;
    }

    fixed4 frag(v2f data):SV_TARGET
    {
        //高光光照 = 光源的颜色 x 材质的高光反射颜色 x MAX(0, 标准化后的观察方向向量 · 标
准化后光的反射方向) ^光照系数

        //1.光的颜色

```

```

        fixed3 lightColor = _LightColor0.rgb;
        //2.材质的高光材质颜色
        fixed3 specularColor = _Color.rgb;
        //3.标准化后的观察方向向量（当前需要渲染的点的位置-摄像机的位置）
        fixed3 viewDir = normalize(data.worldPosition.xyz -
        _WorldSpaceCameraPos.xyz);

        //4. 标准化后光的反射方向(reflect(光的方向, 点的法线方向))
        fixed3 worldLightDir = normalize(_worldSpaceLightPos0.xyz);
        fixed3 lightRefDir =
        normalize(reflect(worldLightDir,data.specularNormal));

        fixed3 specular = lightColor * specularColor * pow(max(0, dot(viewDir,
        lightRefDir)) ,_Gloss);

        fixed3 color = UNITY_LIGHTMODEL_AMBIENT.rgb + specular;
        return fixed4(color,1.0);
    }

    ENDCG
}

```

Phong着色

- 裴祥凤提出的光照理论：标准光照 = 自发光 + 漫反射光 + 高光反射光 + 环境光
- 所以就是将两个光照模型结合在一起

纹理采样

- 导入一个纹理到项目中
- 原始纹理（边长是 2^n ），纹理不一定是方图，如果原始图的边长不是 2^n ，游戏引擎在运行时，会自动将纹理的变长补偿为 2^n ，所以补偿是有性能损耗的。
- 贴图与渲染点对应原理：[UV 贴图](#)
- 贴纹理示例

```

Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    //将Shader材质球上的主纹理，导入到Cg语言中
    sampler2D _MainTex;
    //存储了在材质球上配置的offset和tiling值
    float4 _MainTex_ST;
    fixed4 _Color;

    struct c2v
    {
        float4 pos : POSITION;
        float4 tex : TEXCOORD0; //来自模型的UV信息
    };

    struct v2f

```

```

{
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD1;
};

v2f vert(c2v data)
{
    v2f result;
    result.pos = UnityObjectToClipPos(data.pos);

    //_MainTex_ST内的xy存储了tiling的缩放值，zw存放了offset的平移值
    //因为是2D纹理，所以只取纹理信息的xy
    result.uv = data.tex.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    return result;
}

fixed4 frag(v2f data) : SV_TARGET
{
    //通过函数，将纹理的UV解出颜色
    return tex2D(_MainTex,data.uv) * _Color;
}

ENDCG
}

```

法线贴图

- 法线贴图，对主纹理凹凸显示
- 建模原理，法线贴图：切线空间，存储切线，映射法线，法线信息存储在切线空间中
- 模型是否凹凸，是由模型顶点决定的，现在实现的法线贴图，控制凹凸，实际上是配合光照实现的，凹进去的部分，颜色偏暗，突出来的部分，颜色偏亮。
- 导入法线贴图，贴图类型转换为 Normal Map，法线纹理类型
- 重点是一个转换矩阵的生成
- 还有就是通过点乘来模拟矩阵右乘
- 具体代码看文件夹中示例

裁剪命令

- 在引擎中，模型一般不会渲染背面，因为往往用户看不到，渲染背面，还会消耗 GPU 性能，但是可以再 Shader 中控制裁剪类型
- 裁剪命令 Cull 值，编写在 Pass 通道内
 - Back：裁剪模型背面，不显示模型的背面（默认值）
 - Front：裁剪模型正面，不显示模型的正面
 - Off：不裁剪，全部都显示

```

Pass
{
    Cull Front
    Cull Back
    Cull off
    //<body>
}

```

渲染和渲染顺序

- 默认的渲染顺序原则：距离相机越近，则渲染越靠前，非透明情况下，靠前的物体会盖住靠后的物体
- 深度：被渲染的物体，到摄像机的距离
- 深度测试：比较当前渲染的物体的深度值
- 深度写入：深度测试确定需要替换颜色时，将需要显示的颜色对应的深度信息写入深度缓冲区
- 深度缓冲区：Unity 会为屏幕的每个点，设定一个存储有当前渲染的颜色的深度值（相机到颜色点物体的距离），物体如果在摄像机视角上发生重叠后，需要计算每个重叠点像素的深度值，如果被渲染物体的深度值小于已经存储在深度缓冲区内的值，则将被渲染物体的像素点颜色替换掉原有的颜色，默认的深度值，是正无穷
- 颜色缓冲区：与深度缓冲区中像素点一一对应，存储有最终显示在屏幕上的颜色信息，就一个像素点而言，应该存储有一个颜色信息。

深度测试

- Unity 是做的从前往后进行测试
- ZTest 表示深度测试通过的条件，用来修改测试通过的结果
 - Off：深度测试永远通过，同 Always
 - Less：比深度缓冲区中深度值小时，测试通过
 - LEqual（默认值）：比深度缓冲区中深度值小或相等时，测试通过
 - Equal：比深度缓冲区中深度值相等时，测试通过
 - NotEqual：比深度缓冲区中深度值不相等时，测试通过
 - Greater：比深度缓冲区中深度值大时，测试通过
 - GEqual：比深度缓冲区中深度值大于等于时，测试通过
 - Always：深度测试永远通过

```
Pass
{
    ZTest Always
    ZTest LEqual
    ZTest Off
    //<body>
}
```

- 测试成功，颜色会写入颜色缓冲区中，但是屏幕最终显示的颜色，不一定是当前物体上的颜色，前方还可能有物体

渲染类型

- 不透明物体，使用 Opaque
- 透明物体，使用 Transparent
- 写在 Pass 通道上方


```
SubShader
{
    Tags{"RenderType" = "Opaque"}
    Pass
    {

    }
}
```

深度写入

- `zwrite` 控制深度测试成功后，是否进行深度写入
 - `On`：开启写入
 - `off`：关闭写入
- 当深度写入关闭时，即使深度测试完全通过，颜色缓冲区不会被写入，必须将深度的物体，放置在 `Transparent` 队列下，颜色缓冲区的更新才会正常 `unity` 内置的透明物体 `Shader`，会关闭深度写入
- 当透明物体在前，非透明物体在后，深度测试都使用默认状态，深度写入保持开启，结果会出现透明物体正常显示，非透明物体被透明物体遮盖的部分，会显示为透明物体颜色

```
Pass
{
    Zwrite On
    Zwrite off
    //<body>
}
```

丢弃片元

实现透明图

- 当 `Alpha` 值降低的时候，需要丢弃一些片元
- 丢弃片元：`Discard`

```
fixed4 frag(v2f data) : SV_TARGET
{
    //通过函数，将纹理的uv解出颜色
    fixed4 color = tex2D(_MainTex,data.uv) * _Color;

    //丢弃片元: discard
    //当纹理采样后，取到的颜色如果Alpha透明度低于0.5 则不显示这个片元
    if(color.a<=0)
        discard;
    return color;
}
```

用噪声图实现消融效果

```
fixed4 frag(v2f data) : SV_TARGET
{
    //通过函数，将纹理的uv解出颜色
    fixed4 color = tex2D(_MainTex,data.uv) * _Color;
    //添加一个新图，噪声图
    fixed4 noise = tex2D(_NoiseTex,data.uv);

    //当纹理采样后，取到的颜色如果Alpha透明度低于阈值 则不显示这个片元
    //这样丢弃噪声图上的颜色就实现消融效果
    if(noise.r < _Flag)
        discard;
    return color;
}
```

渲染队列

- 将要渲染的物体分成一组一组的
- 渲染顺序：是从小向大依次渲染
 - Background (1000)：最早被渲染的队列值，（天空盒，背景等）
 - Geometry (2000)：默认队列，不透明物体的队列值
 - AlphaTest (2450)：透明度测试队列
 - Transparent (3000)：渲染透明物体存放的队列
 - Overlay (4000)：需要覆盖显示的队列（镜头光晕）
- 添加方法

```
SubShader
{
    Tags{"Queue" = "Transparent"}
    //Tags可以叠加使用
    //Tags{"Queue" = "Transparent" "RenderType" = "Opaque"}
    Pass
    {
    }
}
```

- 队列可以在 Shader 文件的参数面板栏可以看到

颜色混合

光的混合方式

- 加倍： $\text{Color}(r, g, b, a) \times \text{倍数} = \text{Color}(r \times \text{倍数}, g \times \text{倍数}, b \times \text{倍数}, a \times \text{倍数})$
- $\text{Color1}(r, g, b, a) + \text{Color2}(r, g, b, a) = \text{Color3}(r1 + r2, g1 + g2, b1 + b2, a1 + a2)$
- $\text{Color1}(r, g, b, a) \times \text{Color2}(r, g, b, a) = \text{Color3}(r1 \times r2, g1 \times g2, b1 \times b2, a1 \times a2)$
- 命令：`Blend SrcFactor DstFactor`，写在 Pass 通道内

颜色混合


- $\text{最终的颜色} = \text{新颜色} \times \text{SrcFactor} + \text{旧颜色} \times \text{DstFactor}$
 - Blend 系数 系数

- 旧颜色：上一个 Pass 通道，颜色缓冲区
- 新颜色：当前 Pass 通道的颜色
- 常用系数

混合颜色系数

- 做一个颜色叠加的效果（固定管线模式的），可以自己使用 cg，命令一样

```
Pass
{
    Color[_Color]
}
Pass
{
    Blend One One
    Color[_Color]
}
```

- 常常见混合类型用混合模式命令

真机打包

IOS打包

- 环境配置
 - Mac：<https://www.apple.com/cn/mac/>
 - iPhone：<https://www.apple.com/cn/iphone/>
- 打包方法谷歌百度

打包方法

- 下载Unity的 ios 支持文件，并安装
- Unity选择打包，点击 ios 平台，点击 switch platform 切换，点击 build 按钮
- 选择导出目录，最终生成 xCode 项目目录
- 点击 *.xcodproj，启动XCode项目
- 插入手机（解锁，在iPhone上信任苹果电脑）
- 配置证书（ios 系统必须有证书，才能进行真机测试）
 - 顶部菜单栏选择 xCode
 - 点击 Preference
 - 点击 Account 标签页（登陆苹果 ID）
 - 点击登陆的账号，Manager Certificate
 - 点击，加号，添加iOS的证书
- 选择项目总配置文件，配置证书，选择iPhone设备，点击运行按钮
- iPhone最终就会启动游戏
- 生成ipa，发送到AppStore审核，审核通过后发布

Android打包

环境配置

- **JDK**：Java开发环境包，安卓应用是使用 Java 开发的，所以需要 Java 的开发环境
 - 去甲骨文网站去下
- **SDK**：安卓系统开发环境包，Unity会依赖安卓系统的一些 API
 - <http://www.android-studio.org/index.php/download>，使用命令行工具可以下载
 - Android Studio是安卓的集成开发环境，自动帮我们下载安卓 SDK 的最新版本
- **NDK**：Android 原生开发工具集，因为Unity支持 i12cpp 打包方式，该方式会将 C# 代码转为 C++ 运行，所以需要 Android 的 C/C++ 开发库支持
 - <https://developer.android.google.cn/ndk>
 - 注意，Unity需要使用 NDK 的 r13b 的版本
- 先装 JDK，最好不要改变默认路径，一路下一步
- 安装Android Studio，他会一起安装 SDK，首次打开还会选择安装依赖包，选择 Standard 标准下载
- 安装 NDK，解压出来即可
- 下载Unity的打包软件，根据开发时候选择的版本，且要选择 Bug 最少的
 - 进入官方网站<https://unity.com/>
 - 页面切换后，最下方（资源->Unity旧版本）
 - 选择需要的版本，点击发行说明
 - 下载后安装
 -  Unity其他依赖包
 - 或者在打包切换平台的时候Unity会要求下载，点击下载就可以

打包步骤

- 安装Unity的 Android 打包支持
- 新建一个Android Studio项目，选择最低版本的Android版本的时候与Unity支持的最低版本安卓保持一致，一般选4.1版本，如果选高版本那么打包的游戏低版本用不了
 - 首次新建项目会下载东西
- 在Unity配置 JDK，SDK，NDK 的目录
 - Edit->Preferences->External Tools
 - 将三个DK的路径找到并配置上
- 切换到安卓平台
 - Build System：打包方式，选择Internal老版打包方式，新版的有些问题
 - 打开Development Build
 - 拖入场景点击Build打包

打包参数的设置

- 每个平台设置不太一样，这里列举Android平台的参数设置
- Company Name：公司的英文名
- Product Name：显示在应用的图标下的文字
- Default Icon：游戏图标
- Default Orientation：手机的持有方向
 - 选择某一个方向后，则游戏应用不会受到系统陀螺仪的方向判定影响，永远锁定一个方向
 - Portrait：听筒朝上

- Portrait Upside Down：听筒朝下
 - Landscape Right：听筒朝右
 - Landscape Left：听筒朝左
 - Auto Rotation：自动朝向，根据手机陀螺仪分别，下面可以勾选它限制朝向
- Icon：新的系统，可以使用更清楚的图标
 - 开启Override for Android
 - 根据不同分辨率，设置对应的图标
- Splash Image：可以修改Unity的应用启动时的图
- Other Settings：其他设置
 - Identification：鉴定
 - Package Name：设置应用的包名
 - Version：游戏的打包版本，在商店上会显示
 - Minimum API Level：支持系统的最低版本，Android Studio建项目的最低版本要和这里对应
 - Target API Level：支持的最新的版本，与下载的 `Android SDK` 有关
 - Configuration：构造
 - Scripting Runtime Version： `C#` 运行的版本（2017支持的稳定版本是3.5）
 - Scripting Backend：Unity脚本的运行方式
 - Mono：以 `C#` 虚拟机在终端上运行
 - il2cpp：打包时，会将 `C#` 编译为 `C++` 代码，终端上运行时以 `C++` 编译代码运行
 - `C++` 的运行效率更高，同时可以防止别人反编译游戏代码
 - API Compatibility Level： `C#` 兼容级别
 - Target Architecture：支持的CPU架构
 - Internet Access：是否允许访问网络，自动即可
 - Write Permission：配置写入权限，一般不做修改
 - Scripting Define Symbols： `C#` 宏，可以用于直接将 `C#` 代码从项目中去掉，可以写多个宏名，用分号分割
 - Logging：设置 `Debug` 打印信息的输出开关，使用腾讯出品的远程记录程序Bug插件 Bugly，记录错误日志
 - <https://bugly.qq.com/v2/>
 - Publishing Setting：应用在安卓商店发布时需要的证书
 - 百度： `keystore` 生成
 - Browse keystore：可以选择已经生成好的证书（证书是个文件）
 - Build system：选择 `Internal`，使用传统的安卓打包方式

宏

- `Debug` 是非常耗性能的，所以在项目正式上线的时候需要删掉 `Debug` 代码，但是当代码多的时候删除掉就非常麻烦，这就可以使用宏

```
#if 宏名
    代码段
#endif

#if 宏名

#else

#endif
```

- 在参数设置中的 Scripting Define Symbols 中写上宏名，这时候代码中的宏片段中的代码就会执行，如果删掉宏名，打包编译的时候，宏片段中的代码会直接被抹掉，即相当于没有这段代码

真机调试

- 一般语言报错在Unity中可以看出，但是在真机上没有提示，这个时候就需要一个插件 Log Viewer，在资源商店即可下载
- 下载后导入插件，发现报错，点击报错信息修改代码，错误原因是因为现在Unity的页游平台已经更换，新版本不用 OSXWebPlayer 和 windowsWebPlayer，统一为 WebGLPlayer，修改代码为
 - Application.platform != RuntimePlatform.WebGLPlayer
- 使用方法
 - 导入插件，顶部导航栏点击后场景中会自动创建一个对象
 - 保存场景，打包
 - 在真机上画个圈就可以看到 debug
 - 注意，如果需要看到 debug 的行数，一定要在打包的时候勾选 development build 开发者测试，就可以画圈的时候看到报错的行数，否则只能看到报错的文件不会显示报错的行

反编译和打包破解方法

- 破解软件<https://www.cnblogs.com/caokai520/p/7708795.html>
- 找到项目代码
 - 项目名_Data/Managed/Assembly-CSharp.dll
 - 使用软件即可破解反编译代码了

AB包管理

- 打包到手机上的时候手机是没有 Asset 目录的，所以需要将包放在 StreamingAssets 目录下，从 StreamingAsset 目录拷贝文件到可写目录 persistentData 中，使用 www 读取

```
public class Bootstrap : MonoBehaviour
{
    public GameObject Unzip;//加载资源的时候的提示框

    void Start()
    {
        //判断在项目打包中定义的宏指令是否定义
        //如果定义过TUNING: if宏判定中的代码会编译到最终的字节码中
        //如果未定义过TUNING: if宏判定中的代码相当于从未编写过
#if TUNING
        Debug.Log("Bootstrap");
```

```

Debug.Log(Application.persistentDataPath);
#endif

//当前应用的可写目录
//每次安装应用时，系统会自动帮应用创建，卸载应用时，目录会被删除
//如果AB包，存储在服务器上，应该用WWW将AB包下载到这个目录里
//Application.persistentDataPath;
//打包在原始的App中，目录中的文件，不能被代码直接读取，Resources内的可以被直接读取
//Application.streamingAssetsPath
//资源解压缩
//将存储在StreamingAssets目录下的资源，拷贝到可写目录中
string[] files = new string[4];
files[0] = "AB/AB";
files[1] = "AB/AB.manifest";
files[2] = "AB/ui";
files[3] = "AB/ui.manifest";

StartCoroutine(CopyAB(files, Run));

//不能将AB包的加载写在这里，因为只有协同执行完成后，所有的AB包才会被拷贝到可写目录下，
才能继续读取AB包
}

IEnumerator CopyAB(string[] files, UnityAction callback)
{
    //防止应用第一次被安装，可写目录下不存在AB目录
    if (!Directory.Exists(Application.persistentDataPath + "/AB/"))
    {
        Unzip.SetActive(true);

        //创建AB目录
        Directory.CreateDirectory(Application.persistentDataPath + "/AB/");

        for (int i = 0; i < files.Length; i++)
        {
            //生成原始路径（AB包存储在StreamingAssets下）
            string source = Application.streamingAssetsPath + "/" +
files[i];

            //读取文件
            WWW www = new WWW(source);
            yield return www;

            string target = Application.persistentDataPath + "/" + files[i];

            //yield继续执行，则读取成功
            //将www读取出来的文件字节流，写入可写目录下的文件中
            File.WriteAllBytes(target, www.bytes);
            www.Dispose();
        }

        Unzip.SetActive(false);
    }

    //写在这里，能够保证所有协同执行完成后，调用AB包加载
    callback();
}

void Run()
{

```

```
        AssetBundle ab = AssetBundle.LoadFromFile(Application.persistentDataPath
+ "/AB/ui");
        GameObject.Find("/Canvas/Image").GetComponent<Image>().sprite =
ab.LoadAsset<Sprite>("0029");

        GameObject.Find("/Canvas/Text").GetComponent<Text>().text = "真机调试";
        GameObject.Find("/Canvas/Text1").GetComponent<Text>().text = "再真机调试";
    }
}
```

1. 游戏物体是显示状态 [↩](#)

2. 有方向有长度 [↩](#) [↩](#)

3. 大小、位置、方向 [↩](#)

4. 游戏物体的激活即显示状态，可在游戏物体组件上勾选隐藏或非隐藏，也可以通过方法设置，游戏物体生成的时候也叫激活状态 [↩](#)

5. 游戏物体的非激活即隐藏状态，游戏物体销毁之后也是叫失活状态 [↩](#) [↩](#) [↩](#)