C#和Unity基础

一、代码开发工具

1. Visual Studio

Visual Studio是微软提供的开发包工具

优点是安装和配置都比较简单，使用官方下载工具进行下载安装即可使用

缺点是mac上的功能较少，使用体验不佳

2. VS Code

VS Code是微软提供的轻量级编辑器

优点是功能和插件较多

缺点:配置相对复杂; mac上的代码提示功能有问题

VSCode的Unity开发环境配置

3. Rider

Rider是JetBrain提供的适用于Unity的开发工具

优点是功能和插件较多，JetBrain的软件界面统一，使用体验较好

缺点是安装较为复杂

二、C#基础

1. 程序结构

```
1  // using关键字在程序中引入其他的命名空间
2  using System;
3
4  // namespace关键字用来声明一个命名空间，HelloWorld是命名空间的名字
5  // 命名空间是类的集合
6  namespace HelloWorld
7  {
8      // class关键字用来定义一个类，类名是HelloWorld
9      class HelloWorld
10     {
```

```
11          // Main是函数名，Main函数是整个C#程序的入口
12          // static表示这是一个静态方法
13          // void表示方法没有返回值
14          static void Main(String[] args)
15          {
16              // 输出Hello World
17              Console.WriteLine("Hello World!");
18              // 读取键盘输入
19              Console.ReadKey();
20          }
21      }
22  }
```

2. 关键字

保留关键字						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in(generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operater	out	out(generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
上下文关键字						
add	alias	ascending	descending	dynamic	from	get

global	group	into	join	let	orderby	partial(type)
partial(method)	remove	select	set			

3. 数据类型

C#的数据可以和Swift一样使用?和??修饰符表示可空类型

3.1 值类型

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- sbyte
- short
- uint
- ulong
- ushort

3.2 引用类型

- 对象类型 Object

对象类型是C#通用类型系统中的所有数据类的最终基类，Object是System.Object的别名，类比OC的NSObject

- 动态类型 Dynamic

```
dynamic <variable_name> = value;
```

动态类型可以存储任意类型的任意值，变量的类型是在运行时检查的

- 字符型类型 String

字符型类型String是System.String的别名，类比OC的NSString

String可以使用 `""` 和 `@""` 两种形式

使用 `@""` 形式声明的字符串称为逐字字符串，会把转义字符 `\` 当做普通字符串使用

3.3 指针类型

指针类型同样是一个变量，他的值是另一个变量的内存地址

```
type * var_name;
```

4. 类型转换

C#的类型转换和OC相同，支持隐式转换和显式转换

C#还可以使用To方法进行转换

ToBoolean、ToByte、ToChar、ToDateTime、ToDecimal、ToDouble、ToInt16、ToInt32、ToInt64、ToSbyte、ToSingle、ToString、ToType、ToUInt16、ToUInt32、ToUInt64

```
int i = 100;
string s = i.ToString();
```

5. Foreach

C#的Foreach语句与OC略有不同

```
foreach (type name in array)
{
    code;
}
```

6. 函数声明

C#的函数声明结构如下

```
Access_Specifier Return_Type FunctionName(Parameter List)
{
    Function_Body
    Return_Statement
}
```

其中Access_Specifier是访问控制修饰符，Return_Type是返回值类型，FunctionName是函数名称，Parameter List是参数列表，Function_Body是函数主体，Return_Statement是返回值

注：C#的函数名一般都是首字母大写

7. 权限控制

- public：所有对象都可以访问
- private：类的内部才能够访问
- internal：同一个程序集的对象可以访问
- protected：受保护的，类的内部或类的父类和子类可以访问
- protected internal：protected和internal的并集，符合任意一条都可以访问

8. 函数数据传递方式

- 值传递

复制实参的值并赋值给形参，实参和形参互不影响

- 引用传递

复制实参的地址并传递给形参，形参改动时会影响实参

```
Func(ref val);

void Func(ref int val)
{
    val += val;
}
```

- 输出传递

输出传递可以一次返回多个值

```
Func(out val);

void Func(out int val)
{
    val = 100;
}
```

9. 数组

9.1 一维数组

```
// 声明数组
int[] array1 = new int[10];
// 初始化一个一维数组
int[] array2 = {1, 2, 3, 4, 5};
// 访问数据
array1[1];
```

9.2 多维数组

```
// 声明数组
// 声明一个二维数组
int[,] array1 = new int[2,3];
// 声明一个三维数组
int[,,] array2 = new int[2,3,4];
// 初始化一个二维数组
int[,] array3 = new int[2,3]{
    {0, 1, 2},
    {3, 4, 5}
};
// 初始化一个二维数组
int[,] array4 = new int[,]{
    {0, 1, 2},
    {3, 4, 5}
};
// 访问多维数组
array3[0, 1];
```

9.3 交错数组

```
// 声明一个包含3个数组的交错数组
int[][] array1 = new int[3][];
// 初始化一个交错数组
int[][] array2 = new int[][] {
    new int[] {1, 2, 3, 4, 5},
    new int[] {1, 2, 3, 4},
    new int[] {1, 2}
}
```

```
// 访问交错数组
array1[1][1]
```

9.4 Array

Array类是所有数组的基类，提供了一系列的数组操作

```
// 数组排序
Array.Sort(arr);
// 数组拷贝
Array.Copy(arr1, arr2, arr1.Length);
// 反转数组
Array.Reverse(arr);
```

10. 字符串

```
// 创建一个字符串
string str1 = "字符串";
// 从字符数组创建一个字符串
char[] letters = {'H', 'e', 'l', 'l', '0'};
string str2 = new string(letters);
```

10.1 C#的字符串提供了两个属性

- Chars[Int32] 获取指定下标的字符
- Length 当前字符串的字符数

10.2 字符串的常用方法

```
// 比较字符串相同
bool isSame = String.Compare(str1, str2) == 0;
// 判断字符串是否包含另一个字符串
bool isContain = str1.Contains(str2);
// 截取字符串
string substr = str1.Substring(5);
// 从数组合并字符串
string[] strArray = new string[] {
    "1",
    "23",
    "456"
```

```
}
string str1 = String.Join(" ", strArray);
```

11. 结构体

C#的结构体与Swift基本相同

使用new创建结构体对象时会调用无参构造函数，对各个字段进行赋值

```
struct Struct1
{
    string field1;
    string field2;
}
Struct1 struct1 = new Struct1();
Console.WriteLine(struct1.field1) // 此处会输出空字符串
```

结构体可以不使用new进行实例化

如果不使用new进行实例化，只有在字段被赋值之后，结构体对象的字段才可以使用，否则会报错

```
struct Struct1
{
    string field1;
    string field2;
}

Struct1 struct1;
Console.WriteLine(struct1.field1); // 由于field1未赋值，此处调用会报错
struct1.field1 = "field1";
Console.WriteLine(struct1.field1); // 由于field1已赋值，此处调用可正常输出field1
```

类和结构体的不同主要在于类是引用类型，结构体是值类型，这一点与Swift相同

12. 枚举类型

```
enum enum_name {
    enumeration list;
}
```

其中，enum_name是类型变量的名称，enumeration list是成员列表，用逗号分开

C#的枚举和OC类似，只能使用整数类型常量

13. 类

13.1 类的定义

```
<access specifier> class class_name
{
    // 成员变量/属性
   <access specifier> <data type> variable1;

   // 成员函数
   <access specifier> <return type> method1(parameter_list)
   {
       // 函数体
   }
}
```

其中，<access specifier>表示权限修饰符，class_name是类的名称

与OC不同的是，一般来说，C#使用命名空间区分类，可以不用在类名前添加前缀

13.2 对象的创建

```
// 创建一个Class1对象
Class1 object = new Class1();
```

C#类的使用与OC和Swift基本相同

13.3 构造函数

13.3.1 实例构造函数

new创建对象时可以使用这个函数进行类的创建和成员变量的初始化

13.3.2 静态构造函数

用于初始化类中的静态数据或执行只需要执行一次的操作，这个函数会在创建第一个实例或者引用静态成员变量之前调用

```
class Class1
{
```

```
    public static int field = 0;
    static Class1()
    {
        field = 1;
    }
}

Console.WriteLine(Class1.field) // 此处会输出1
```

特点

- 不能使用访问权限修饰符
- 不具有参数
- 每个类只能有一个静态构造函数
- 不能继承或重载
- 不能直接调用
- 不能控制执行时间
- 在实例构造函数之前执行

13.3.3 私有构造函数

这是一种特殊的实例构造函数，通常用在只包含静态成员的类中，如果一个类没有公共构造函数的话，其他类（嵌套类除外）将无法创建该类的实例

```
class Class1
{
    // 空的私有构造函数，可以阻止自动生成无参构造函数
    private Class1() { }
}
```

13.4 析构函数

析构函数类似OC的dealloc函数或Swift的deinit函数，用于在垃圾回收实例时执行

```
class Class1
{
    // 析构函数
    ~Class1()
    {
    }
}
```

特点

- 只能在类中定义，结构体不能定义
- 一个类只能有一个析构函数
- 不能继承或重载
- 没有返回值
- 自动调用，不能手动调用
- 不能使用访问修饰符
- 不能包含参数

13.5 this

C#中使用this表示当前对象，相当于OC和Swift的self

13.6 继承

与OC和Swift一样，C#只支持单继承

当类被sealed修饰时，这个类不能被继承

```
sealed class Class1 { }
class Class2: Class1 { } // 此处会报错
```

C#中使用base调用父类方法

13.6.1 直接重写方法

```
class Class1
{
    public void Func1()
    {
        Console.WriteLine("1");
    }
}

class Class2 : Class1
{
    public new void Func1()
    {
        // 调用父类方法
        base.Func1();
```

```
        Console.WriteLine("2");
    }
}
```

13.6.2 使用虚函数重写

```
class Class1
{
    public virtual void Func1()
    {
        Console.WriteLine("1");
    }
}

class Class2 : Class1
{
    public override void Func1()
    {
        base.Func1();
        Console.WriteLine("2");
    }
}
```

13.6.3 使用抽象类重写

```
abstract class Class1
{
    public abstract void Func1();
}

class Class2 : Class1
{
    public override void Func1()
    {
        Console.WriteLine("2");
    }
}
```

特点

- 抽象类不能创建实例
- 抽象方法没有方法体，且必须在抽象类中

- 如果子类不是抽象类，则必须要实现抽象方法

13.7 运算符重载

```
public static Class1 operator+ (Class1 a, Class1 b)
{
    return a;
}
```

运算符重载情况如下

- 可重载的运算符

+、-、!、~、++、--、*、/、%、&、|、^、<<、>>、=、==、!=、<、>、<=、>=

- 不可重载的运算符

&&、||、(typeof)var_name、+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=、.、?、?:、??、??=、..、->、=>、as、await、checked、unchecked、default、delegate、is、nameof、new、sizeof、stackalloc、switch、typeof

注:

- 以下运算符==和!=、<和>、<=和>=必须成对重载，在重载其中任意一个时，必须也要重载另外一个
- +=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=不能直接重载，但是在重载对应的二元运算符时会隐式重载

13.8 索引器

```
<access specifier> <data type> this[<data type> parameter_list]
{
    get {}
    set {}
}
```

索引器是一个特殊成员，可以支持用数组的方式访问，可以带多个参数

14. 接口

C#的接口类似OC和Swift的Protocol

```
interface Interface1
{
```

```
    int field1 { get; set; }
    int field2 { get; }
    int field3; // 此处会报错
    private int field4 { get; } // 此处会报错
    int Func1();
}

class Class1: Interface1
{
    public void Func1() { }
}

class Class2: Interface1
{
    void Interface1.Func1() { }
}

class Class3: Interface1
{
    public void Interface1.Func1() { } // 此处会报错
}
```

特点

- 接口不能实例化
- 接口可以包含事件、索引器、方法和属性
- 接口可以继承多个接口
- 类可以继承多个接口
- 默认使用internal修饰，可以使用new、public、protected、inteval和private修饰
- 接口的方法不能添加任何修饰符，默认为public
- 接口的成员变量不能使用任何修饰符，默认为public static final
- 接口不能定义字段
- 接口不能包含运算符重载、构造函数、析构函数
- 接口定义的方法不能包含方法体
- 可以在方法前加上前缀区分实现的接口，但是用前缀修饰的方法不能使用任何修饰符

15. 命名空间

C#中可以使用命名空间区分代码

使用using引入不同的命名空间

```
using System;
// 使用别名区分重名的命名空间
using sys = System;
// 添加global使命名空间用于所有文件
global using System;
// 添加static使得命名空间中的static成员和嵌套类型不需要制定类型
using static System.Console;
```

16. 特性

特性是一种用于在程序运行时传递各种元素（类、方法、结构体、枚举等）行为信息的声明性代码

使用特性可以将元数据（例如编译器指令、注释、描述、方法和类等信息）添加到程序中

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

其中，[]用来定义特性的名称和值，positional_parameters用来指定基本信息，name_parameter用来指定可选信息

16.1 预定义特性

- AttributeUsage

```
[AttributeUsage(validon, AllowMultiple = allowmultiple, Inherited = inherited)]
```

Validon用来定义特性可被放置的元素，是枚举AttributeTargets的值的组合，默认为AttributeTargets.All

成员名称	说明
All	可以对任何应用程序元素应用属性
Assembly	可以对程序集应用属性
Class	可以对类应用属性
Constructor	可以对构造函数应用属性

Delegate	可以对委托应用属性
Enum	可以对枚举应用属性

AllowMultiple标记了特性是否可以放置在同一个实体多次

Inherited标记了特性是否能被继承

- Conditional

```
[Conditional("Debug")]
public static void Func()
{
    Console.WriteLine("Conditional Func");
}
```

与#if类似，但是判断的依据是调用方的环境，而#if取决于当前环境

- Obsolete

```
[Obsolete(message, isError)]
```

标记函数为已弃用，message标记报错信息，isError标记提示为一个错误还是警告

16.2 自定义特性

```
// 定义一个特性
public class Demo1Attribute: System.Attribute {
    public Demo1Attribute() { }
}
// 定义一个自动生成构造函数的特性
public class Demo2Attribute: System.Attribute {
    public string Field1;
    public string Field2;
}
// 定义一个包含位置参数和命名参数的构造函数的特性
public class Demo3Attribute: System.Attribute {
    public string Field1;
    public string Field2;

    public Demo3Attribute(string field1) {
        Field1 = field1;
```

```
    }
}

[Demo1]
[Demo2(Field1 = "Field1", Field2 = "Field2")]
// 传入参数的方式有两种，一种是通过位置参数在构造函数中传递，另一种是通过命名参数直接传递
[Demo3("Field1", Field2 = "Field2")]
public void Func() {}
```

自定义特性需要继承自 `System.Attribute` ，命名方式一般以 `Attribute` 为后缀

C#提供了两个方法用于判断对象的特性

- IsDefine

```
Class1 obj = new Class1();
obj.GetType().IsDefined(typeof(Demo1Attribute), false)
```

判断对象是否应用了某个特性，第一个参数是检查的特性的Type对象，第二个参数bool类型用于标记是否搜索继承树来查找这个特性

- GetCustomAttributes

```
Class1 obj = new Class1();
obj.GetType().GetCustomAttributes(true);
```

用于返回对象的特性数组，返回的是一个 `object` 数组，使用时必须要强制转换，参数bool类型用于标记是否搜索继承树来查找这个特性

17. 反射

C#反射的功能

- 运行时查看视图属性
- 检查装配中的各种类型并实例化这些类型
- 在后期绑定到方法和属性
- 在运行时创建新类型，然后使用这些类型执行一些任务

```
class Class1
{
    public string Field1;
```

```
4
5       public Class1(string field1)
6       {
7           this.Field1 = field1;
8       }
9
10      public void Func1() { }
11      public void Func2(int i) { }
12  }
13
14  class Class2
15  {
16      public List<string> List1 = new List<string>();
17  }
18
19  // 通过Activator创建实例
20  object obj1 = Activator.CreateInstance(typeof(Class1), "Field1");
21  // 通过构造函数创建实例
22  Type t = typeof(Class1);
23  Type[] paramTypes = new Type[1] { typeof(string) };
24  var info1 = t.GetConstructor(paramTypes);
25  object[] param = new object[1] { "Field1" };
26  var obj2 = info1.Invoke(param);
27  // 通过反射获取Field1并赋值
28  var obj3 = new Class1("Field1");
29  var info2 = obj3.GetType().GetField("Field1");
30  info2.SetValue(obj3, "Field2");
31  Console.WriteLine(info2.GetValue(obj3));
32  // 获取list中所有元素
33  Class2 obj4 = new Class2();
34  Type type = obj4.List1.GetType();
35  int count = Convert.ToInt32(type.InvokeMember("get_Count",
    BindingFlags.InvokeMethod, null, obj4.List1, null));
36  for (int i = 0;i < count; i ++)
37  {
38      object value = type.InvokeMember("get_Item",
    BindingFlags.InvokeMethod|BindingFlags.Default, null, obj4.List1, new object[]
    {i});
39  }
40  // 调用类的方法
41  var f1 = obj1.GetType().GetMethod("Func1");
42  f1.Invoke(obj1, null);
43  var f2 = obj2.GetType().GetMethod("Func2");
44  f2.Invoke(obj1, new object[] { 100 });
```

18. 委托

```
delegate <return type> delegate-name(<parameter list>)
```

C#的委托类似于OC的block，所有委托都派生自System.Delegate

18.1 声明和使用委托

```
public delegate void Delegate1(string s); // 声明一个委托

Delegate1 d1 = new Delegate1(Func1); // 实例化委托对象并将其与Func1关联
d1("String");
```

18.2 多播委托

```
public delegate void Delegate1(string s); // 声明一个委托

// 实例化两个Delegate1
Delegate1 d1 = new Delegate1(Func1);
Delegate1 d2 = new Delegate1(Func2);
d1 += d2

d1();// 此处会调用Func1和Func2
```

委托可以使用+将多个对象分配给一个实例，可以使用-将对象从实例中移除，委托会依次调用列表中的委托

18.3 事件

```
public delegate void Delegate1(string str);
public event Delegate1 Event1;
```

在声明一个事件之前，需要声明一个委托类型，当事件触发时会调用委托

事件本质上是委托字段的一个包装器

```
public delegate void Delegate1(string str);
```

```
// 这样可以确保事件总有默认值，不需要每次检查NULL
private Delegate1 eventHangler = delegate {  };
private event Delegate1 Event1
{
    add
    {
        eventHangler += value;
    }
    remove
    {
        eventHangler -= value;
    }
}

// 调用event
private event Delegate1 Event2;

Event2?("str");
```

18.4 匿名函数

```
delegate (<parameter list>)
{
}
```

类似block变量，主要配合委托和事件使用

缺点在于添加到委托或事件容器后，无法单独移除，只能清空

18.5 Action和Func

```
// 默认Action没有参数
Action a1 = Method1;
a1();

// 带参数的Action
Action<string, string> a2 = Method2;
a("str1", "str2");

// 使用lambda表达式
Action<string, string> a3 = new Action((str1, str2) => {});

// Func
```

```
Func<string> f1 = Method1;
f1();

// 带参数的Func
Func<string, string, bool> f2 = Method2;
f2("str1", "str2")
```

Action是一个泛型委托，可以使用0~16个参数，没有返回值

Func是一个泛型委托，可以使用0~16个 参数，有返回值

19. 集合

19.1 动态数组ArrayList

```
ArrayList arr = new ArrayList();
```

功能类比NSMutableArray

19.2 哈希表Hashtable

```
Hashtable h = new Hashtable();
ht.Add("key", "value");
```

功能类比NSMutableDictionary

19.3 排序列表SortedList

```
SortedList s1 = new SortedList();
s1.Add("key", "value")
```

根据Key排序的Dictionary，还可以使用数组的方式访问

19.4 栈Stack

```
Stack s = new Stack();
s.Push('A')
```

先进后出的数据结构

19.5 队列Queue

```
Queue q = new Queue();
q.Enqueue('A');
```

先进先出的数据结构

19.6 点阵列BitArray

```
byte[] a = {60};
BitArray b = new BitArray(a);
```

紧凑型的位值数组，值都是布尔类型，1表示开启，0表示关闭

20. 多线程Thread

```
Thread th = Thread.CurrentThread;
th.Name = "主线程";
```

C#中，System.Threading.Thread类用于处理线程

```
public static void CallToChildThread()
{
    // 线程暂停
    Thread.Sleep(60);
}
ThreadStart childref = new ThreadStart(CallToChildThread);
Thread childThread = new Thread(childref);
// 创建一个子线程
childThread.Start();
Thread.Sleep(200);
// 销毁一个子线程
childThread.Abort();
```

三、Unity基础

当前SnakeUnity使用的版本是2019.4.32f1c1

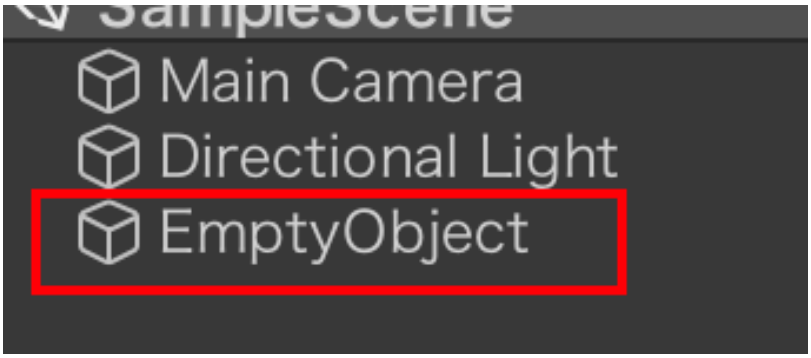
1. 基础概念

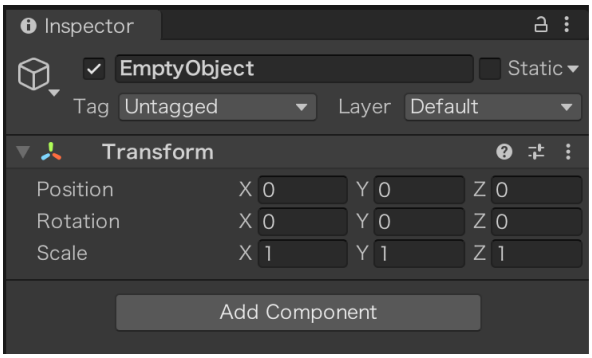
1.1 物体 GameObject

Unity场景中所有实体的基类

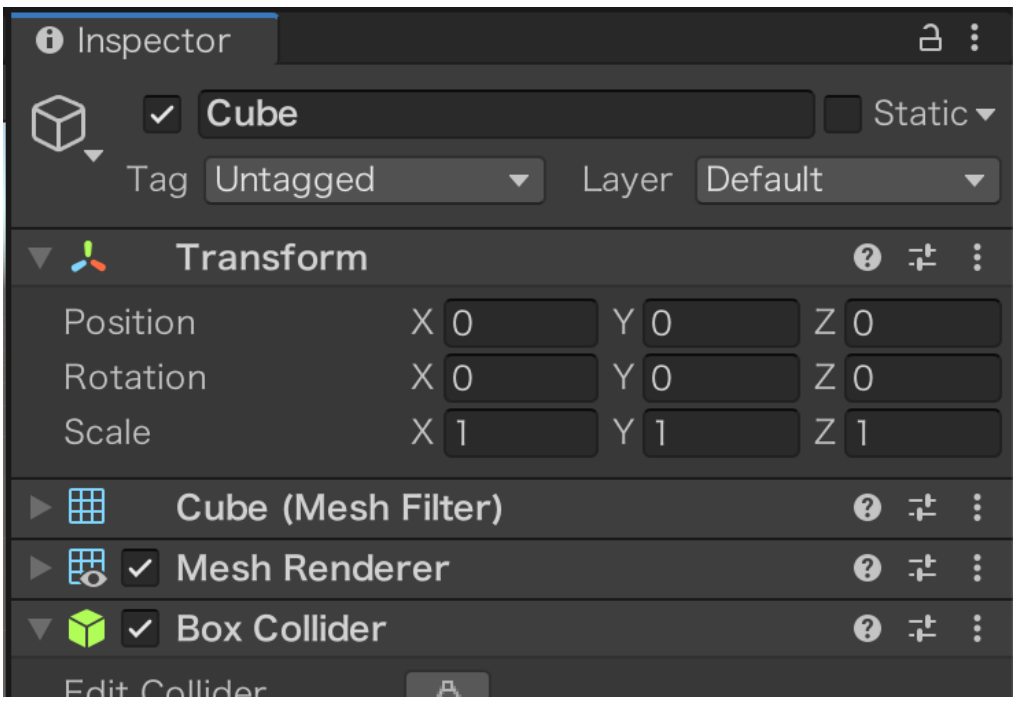
每个物体都会有一个Transform组件用于确定物体的位置、旋转、缩放

空物体EmptyObject是一个特殊的物体，空物体不包含Mesh

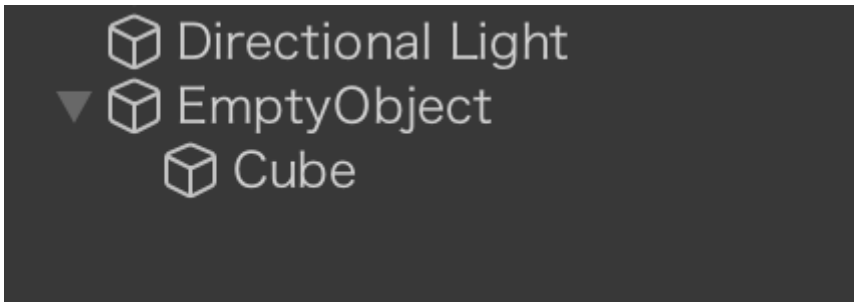




下面是一个包含Mesh的物体



物体之间可以有父子关系，子物体会跟随父物体的移动进行移动



1.2 坐标系

世界坐标系Global，以世界中心为轴，6个方向代表了上下东西南北

本地坐标系Local，以物体自己为轴，6个方向代表了上下前后左右

1.3 轴心 Pivot和几何中心 Center

轴心，一个物体的操作基准点，作用类似于锚点

几何中心，对于基本物体来说，轴心点默认位于几何中心

在Unity中，默认情况是Pivot模式，可以选择切换物体在移动、旋转和缩放时，使用哪个点作为操作基准点

1.4 组件 Component

代表一个具体功能，每个游戏物体都有多个组件构成，其中，每个游戏物体都有Transform组件，用于保存游戏物体的position，rotation、scale等信息。

1.5 矢量 Vector

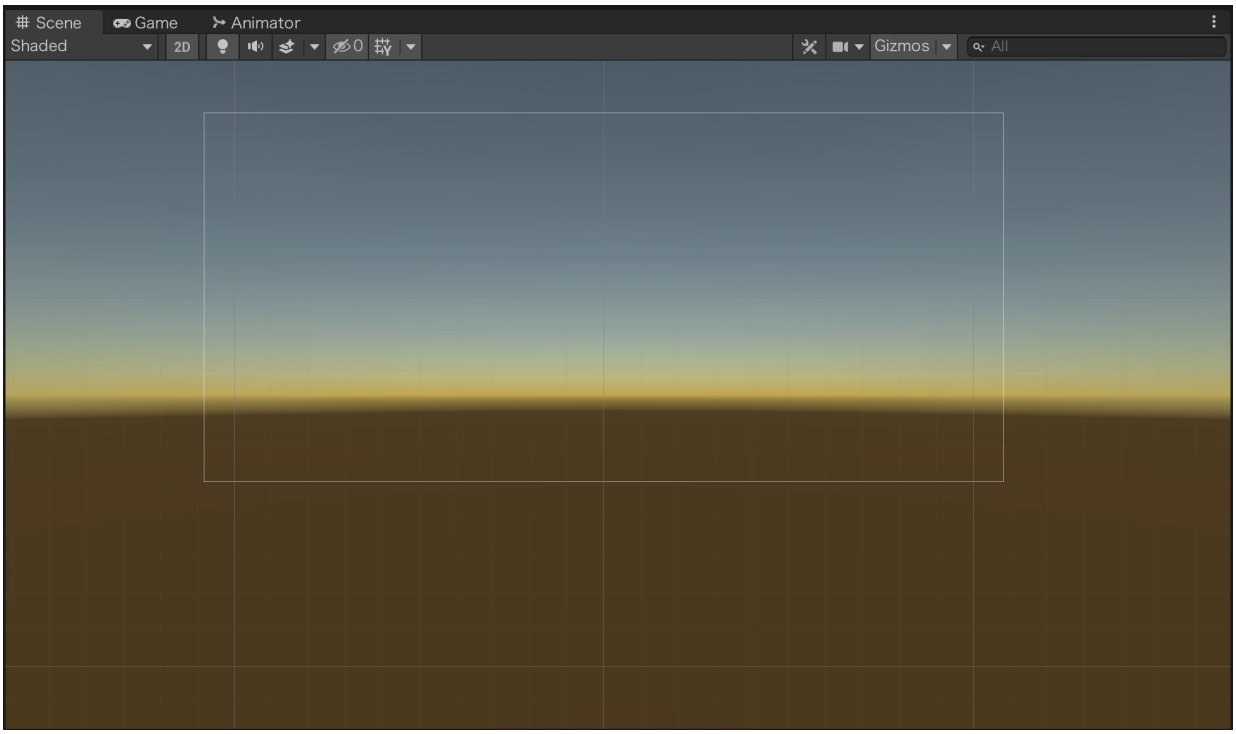
矢量是一个描述位置、方向等的基本数据结构，Unity提供了Vector2、Vector3、Vector4来处理2D、3D、4D矢量

1.6 四元组 Quaternion

Unity使用四元组描述三维方向，他不受万向节锁的影响，可以方便快捷的进行球面插值

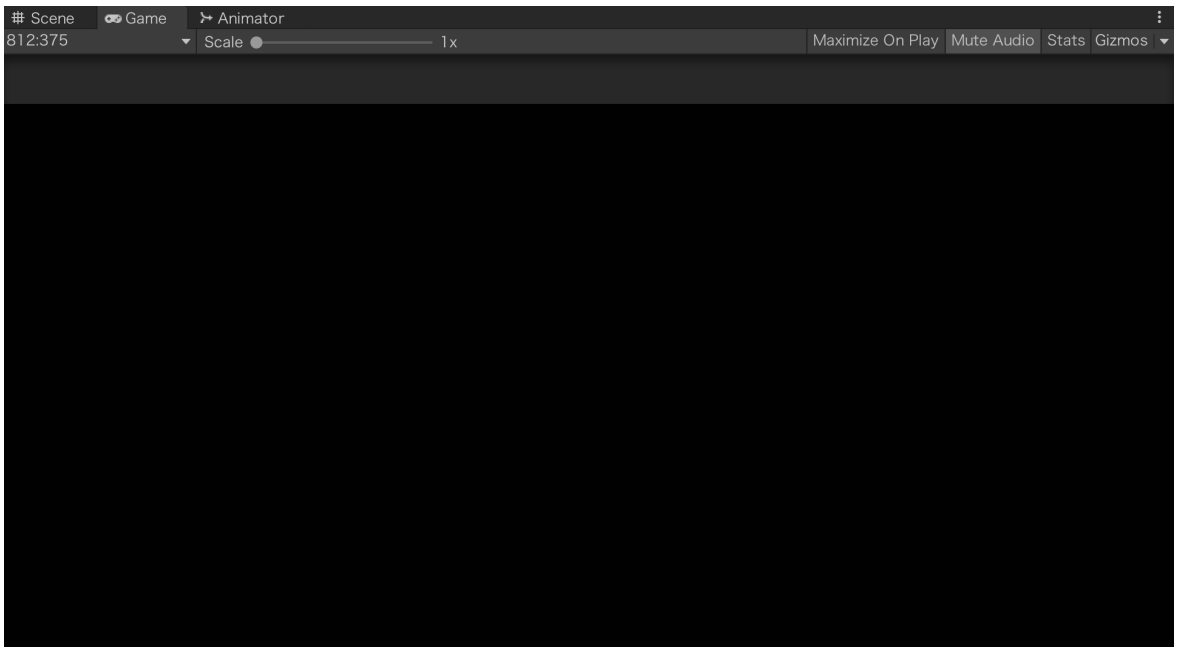
2. Unity编辑器

2.1 场景编辑 Scene



通过这个窗口可以进行各种GameObject的操作，可以可视化的进行编辑，展示的画面就是开发者看到的视野。

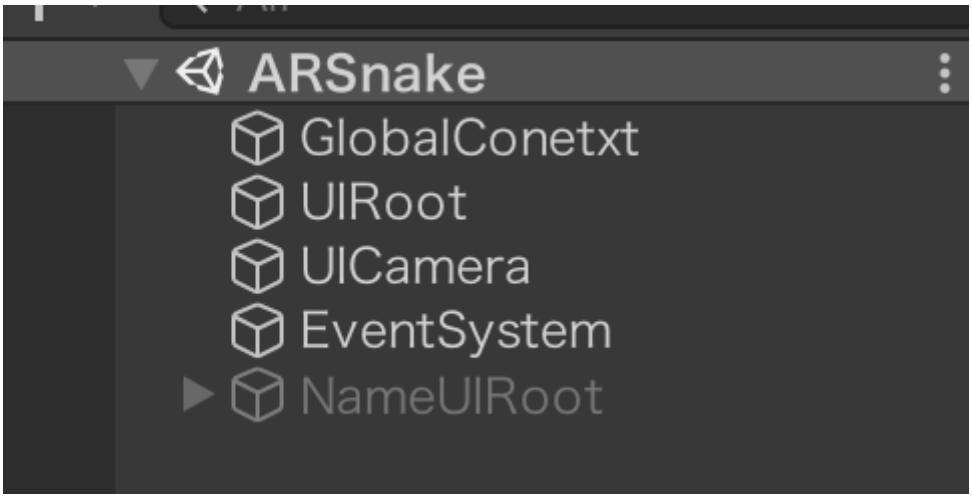
2.2 游戏运行 Game



这个窗口是显示运行中摄像机视野的图像，这个窗口不能进行编辑

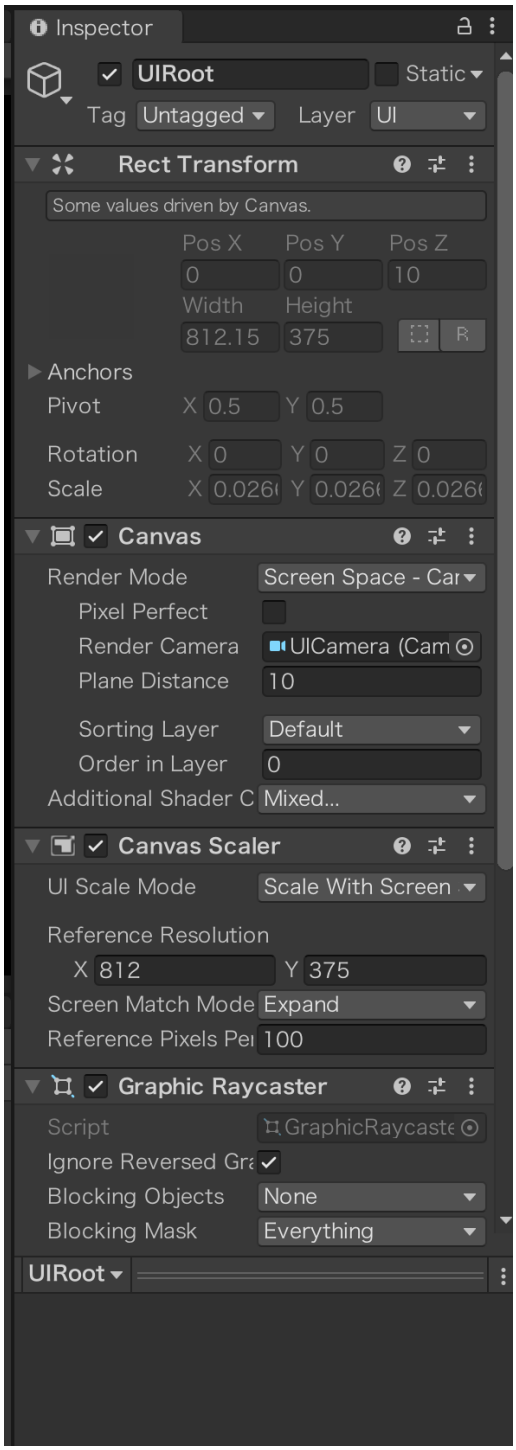
2.3 场景物体列表 Hierarcy

使用树形结构列出当前场景的GameObject，并显示GameObject的层级关系



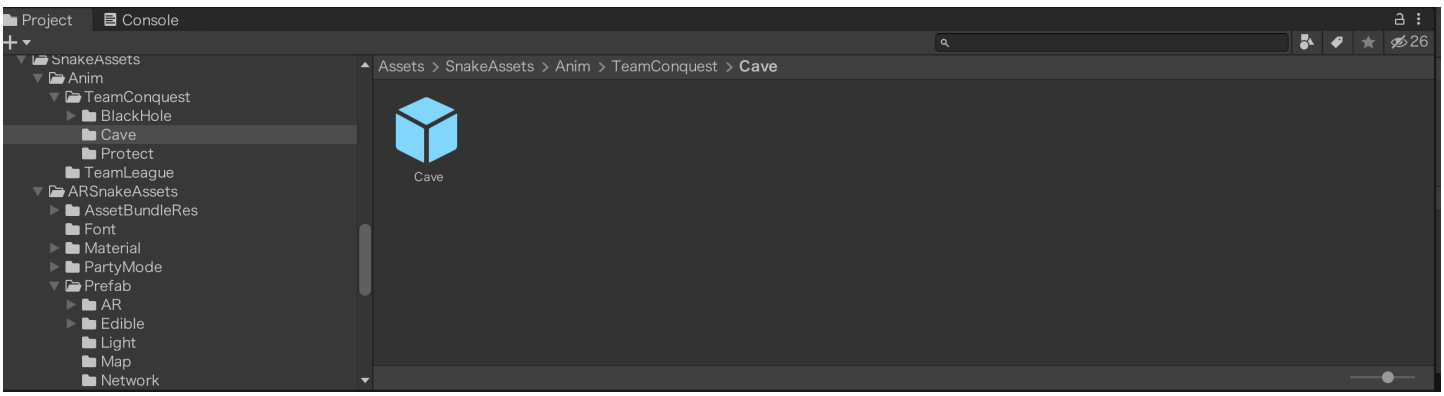
2.4 属性编辑列表 Inspector

属性面板，用于展示GameObject的各种组件的各种属性，通过 `Add Component` 添加相应的组件



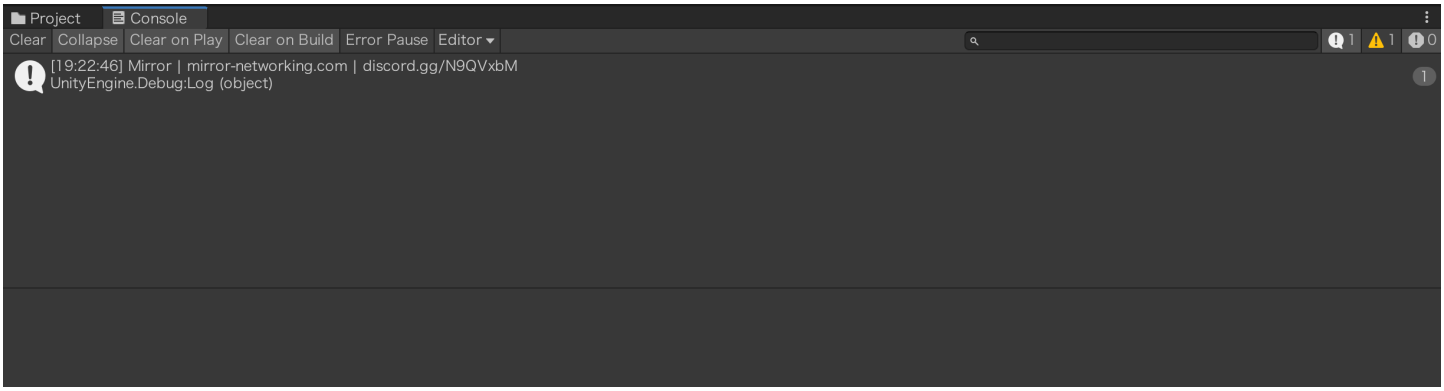
2.5 项目资源列表 Project

展示工程中的各种资源，包括场景、脚本、材质等等



2.6 控制台 Console

用于显示报错、警告和测试信息



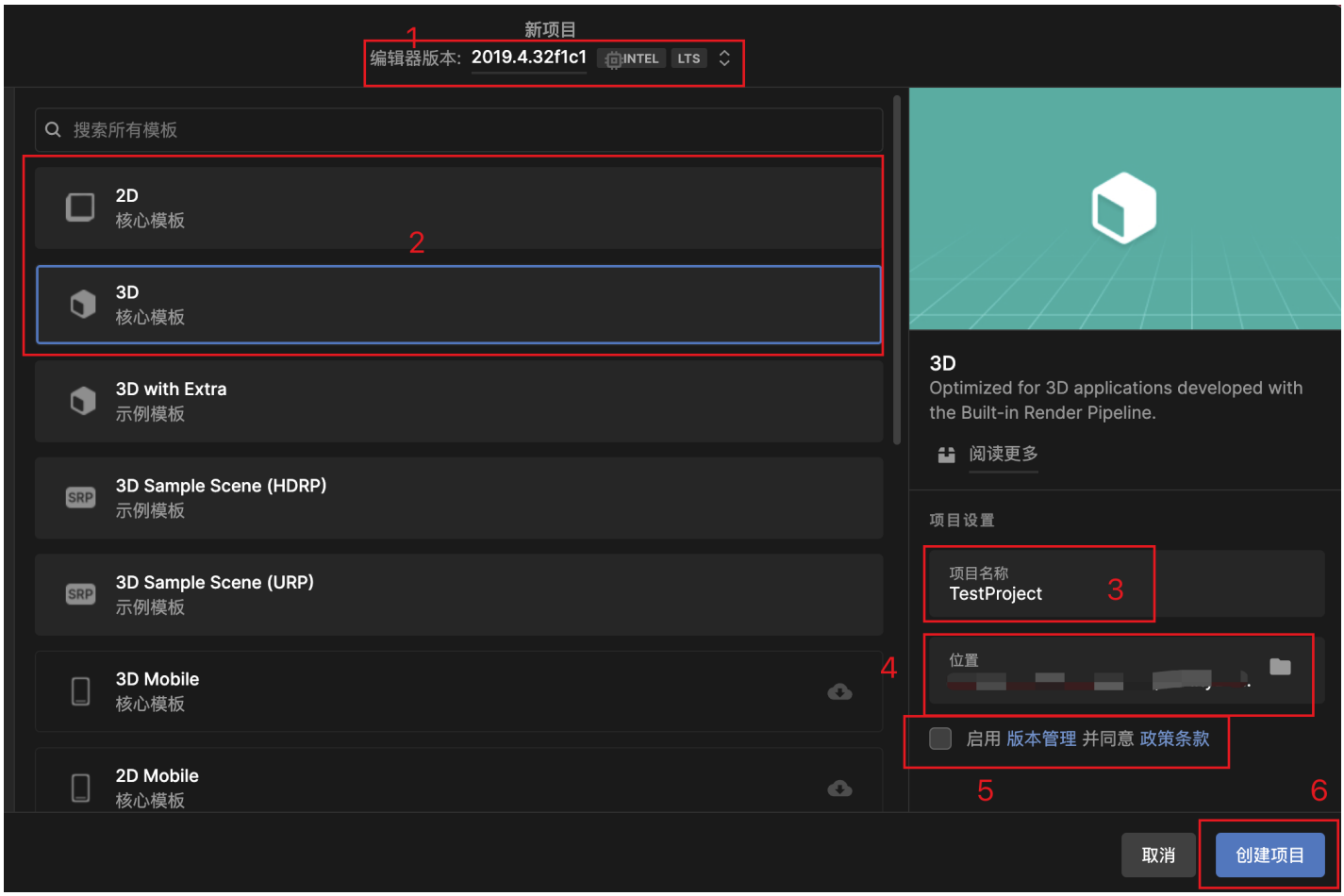
3. 创建第一个Unity工程

此处使用3D模板进行创建，2D相似

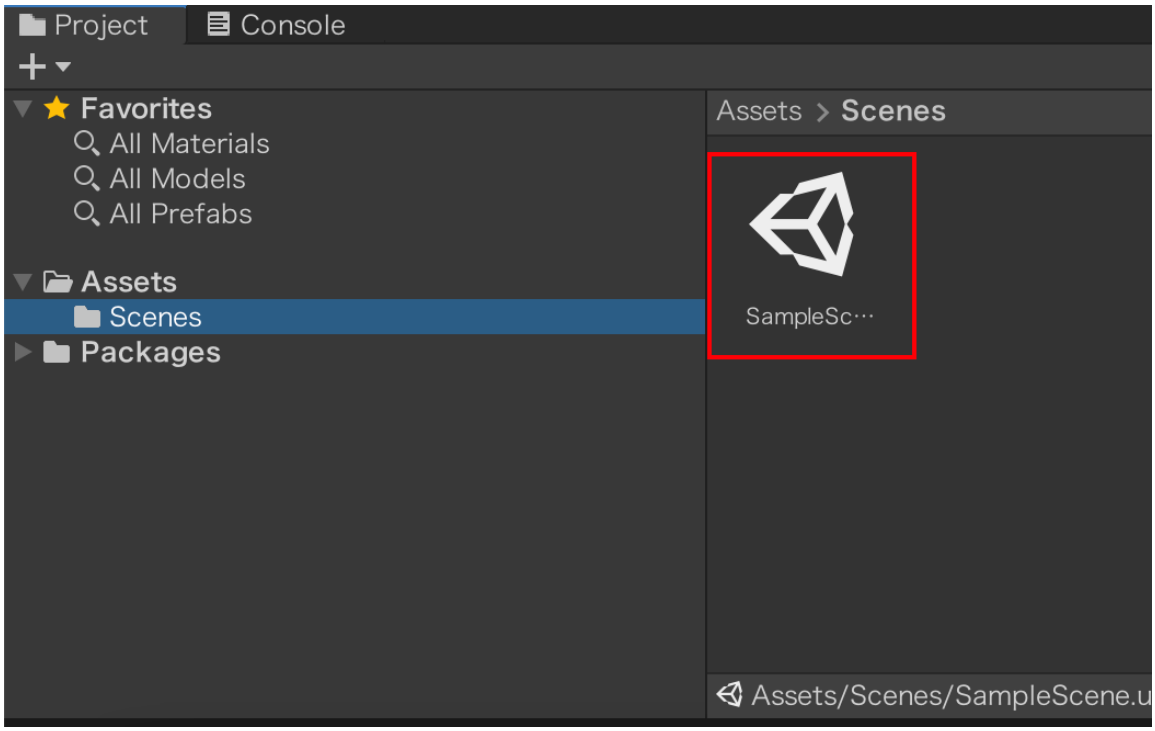
- 创建工程

选择创建新项目

根据需要选择2D或3D的核心模板，设置项目名称和位置、是否开启Unity版本管理，Unity的版本管理使用的是PlasticSCM，然后创建项目进入项目

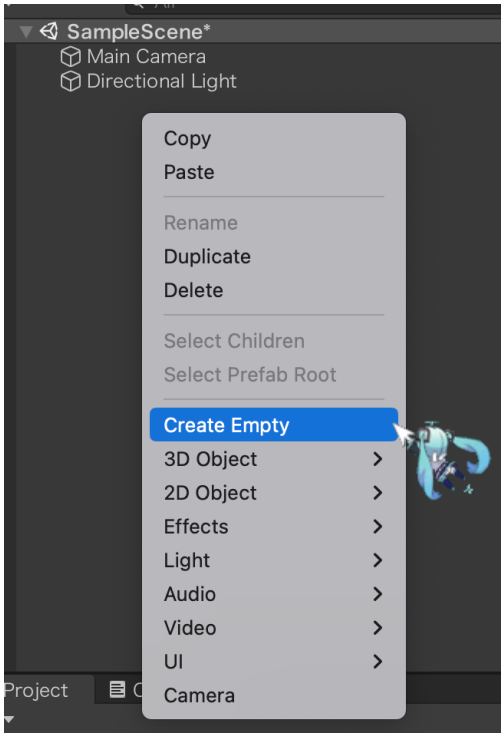


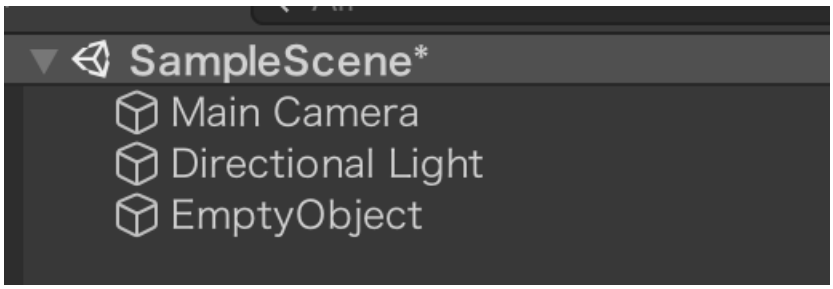
- 创建第一个物体和开始运行
  - 选择进入场景SampleScene，场景中会包含一个相机和一个光照(3D模板只会包含一个相机)



- 创建一个空物体

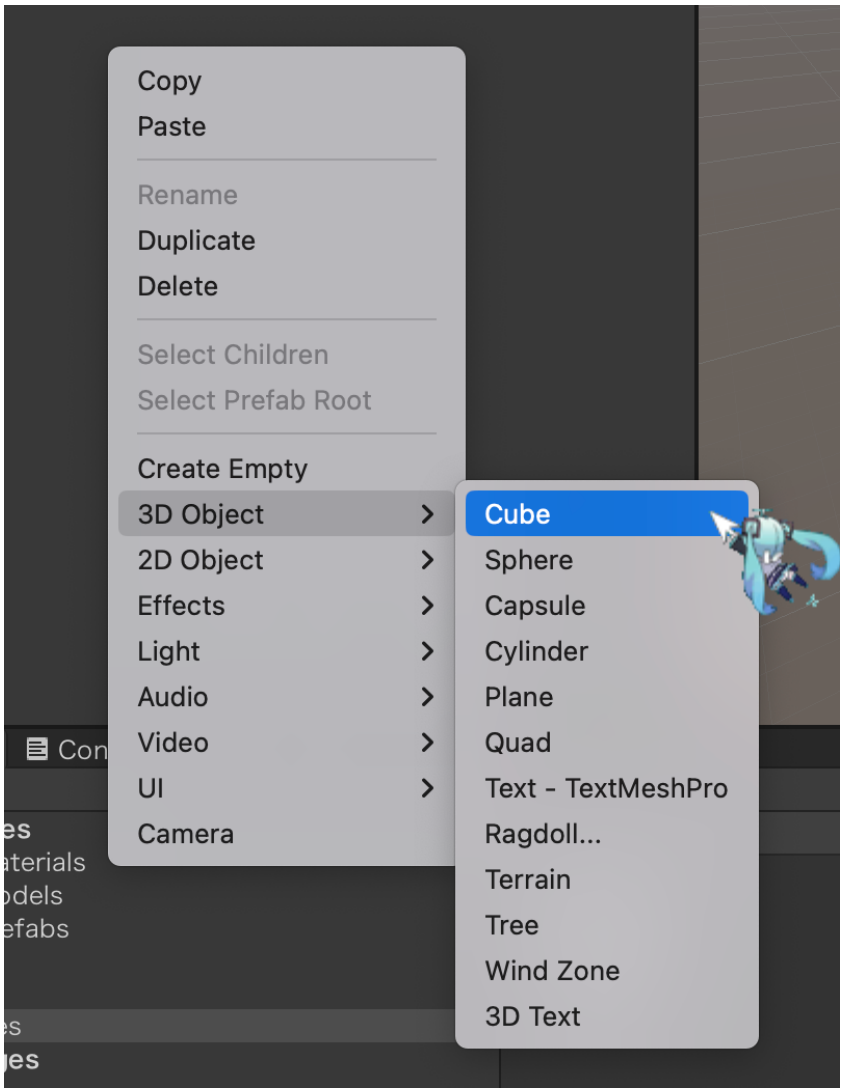
在Hierarchy中右击创建一个空对象并重命名为EmptyObject





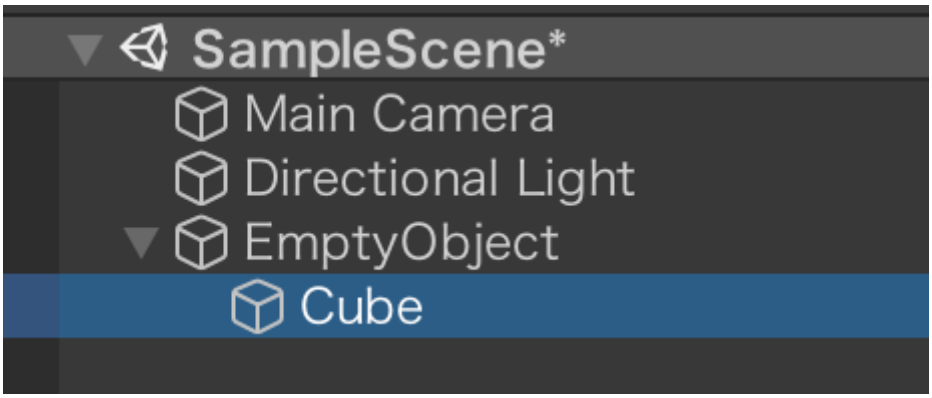
- 创建一个Cube

同样的方式创建一个Cube并重命名为Cube



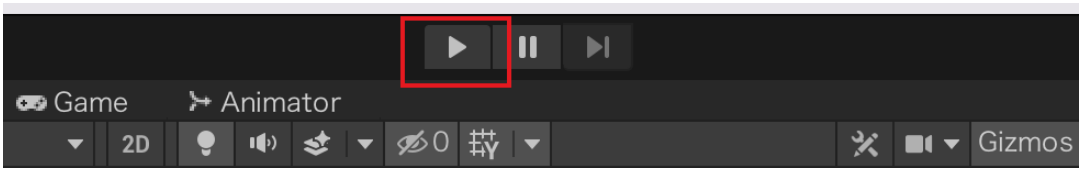
- 设置父物体

将Cube拖到EmptyObject上，使Cube成为EmptyObject的子物体，EmptyObject成为Cube的父物体

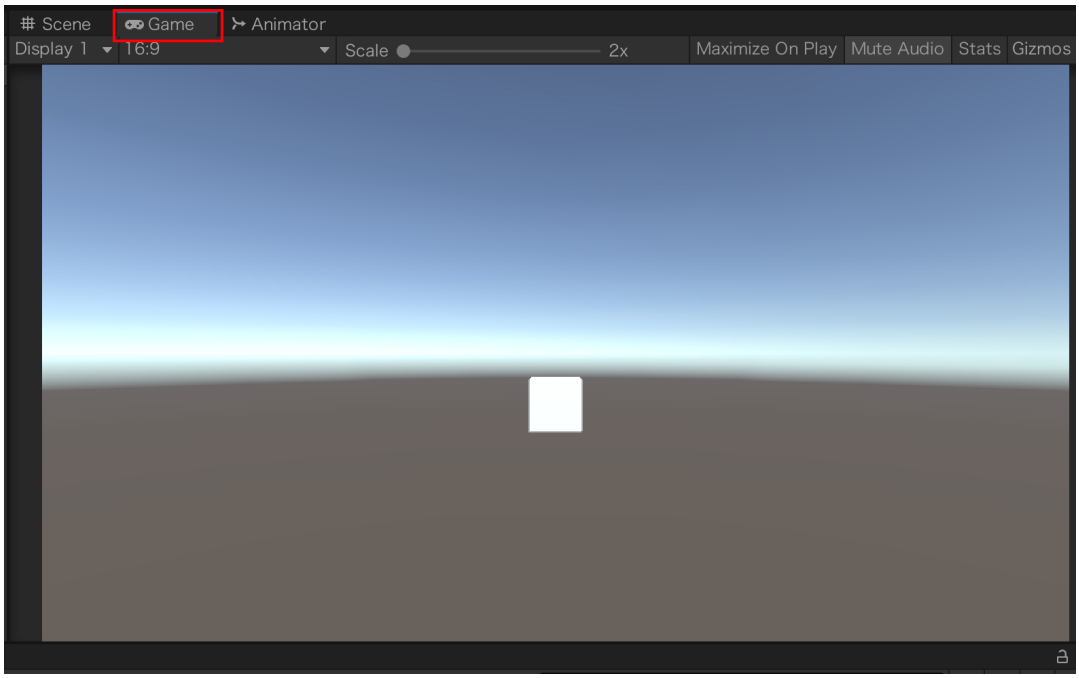


- 第一次运行

点击运行按钮

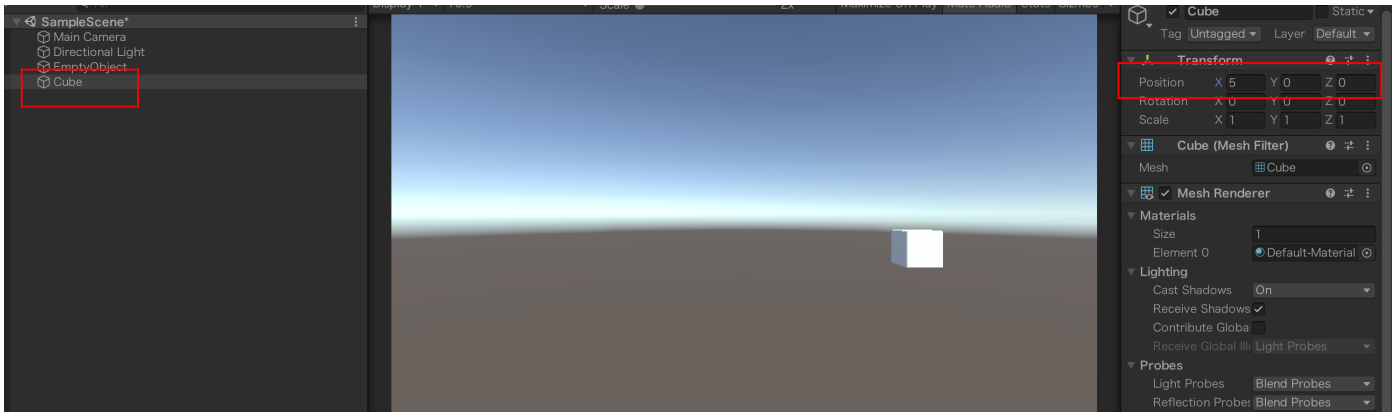


在game界面查看运行效果



- 修改物体位置

在Hierarchy中选中Cube，在Inspector中修改Transform中position的x值，可以看到物体位置发生了改变

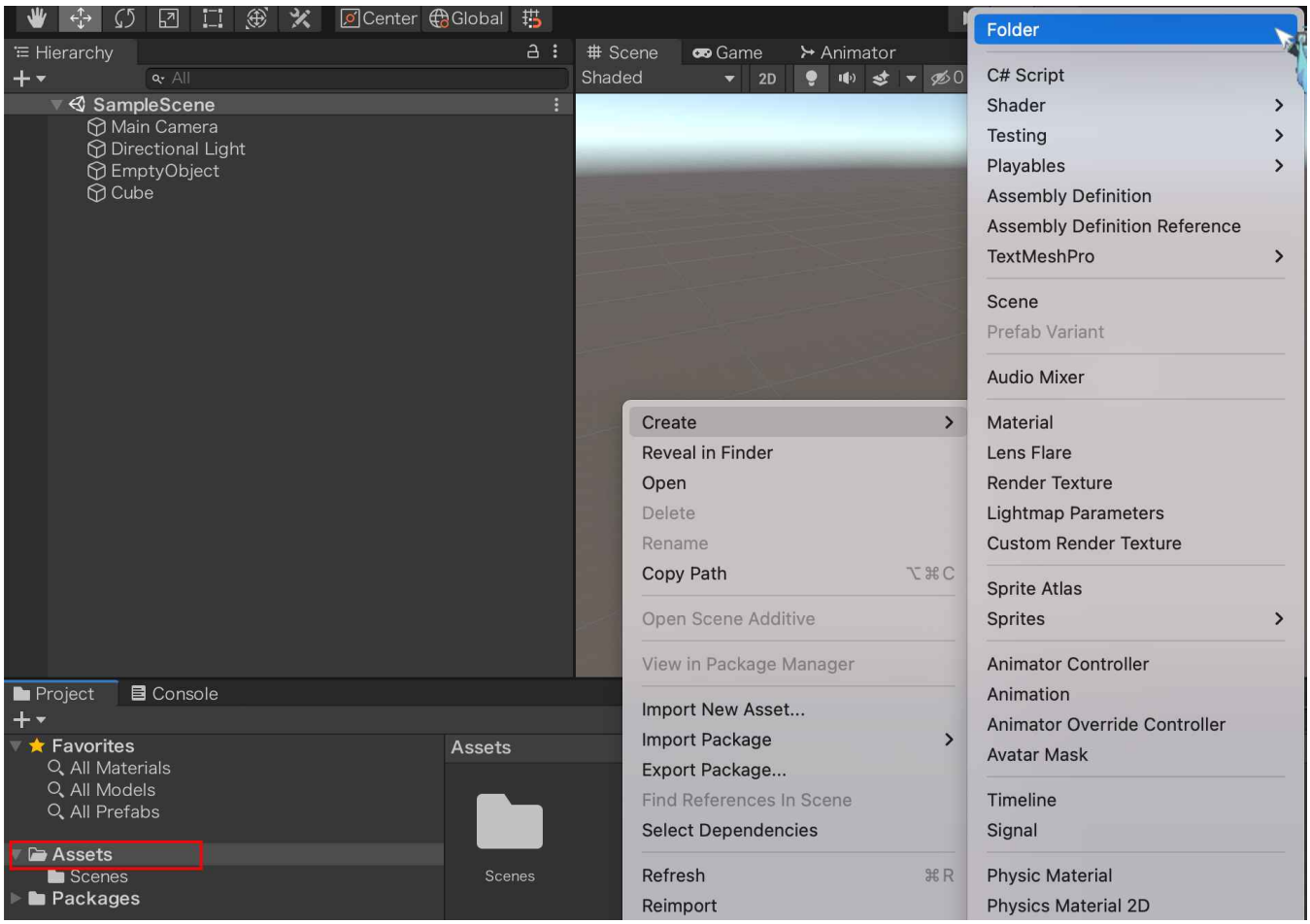


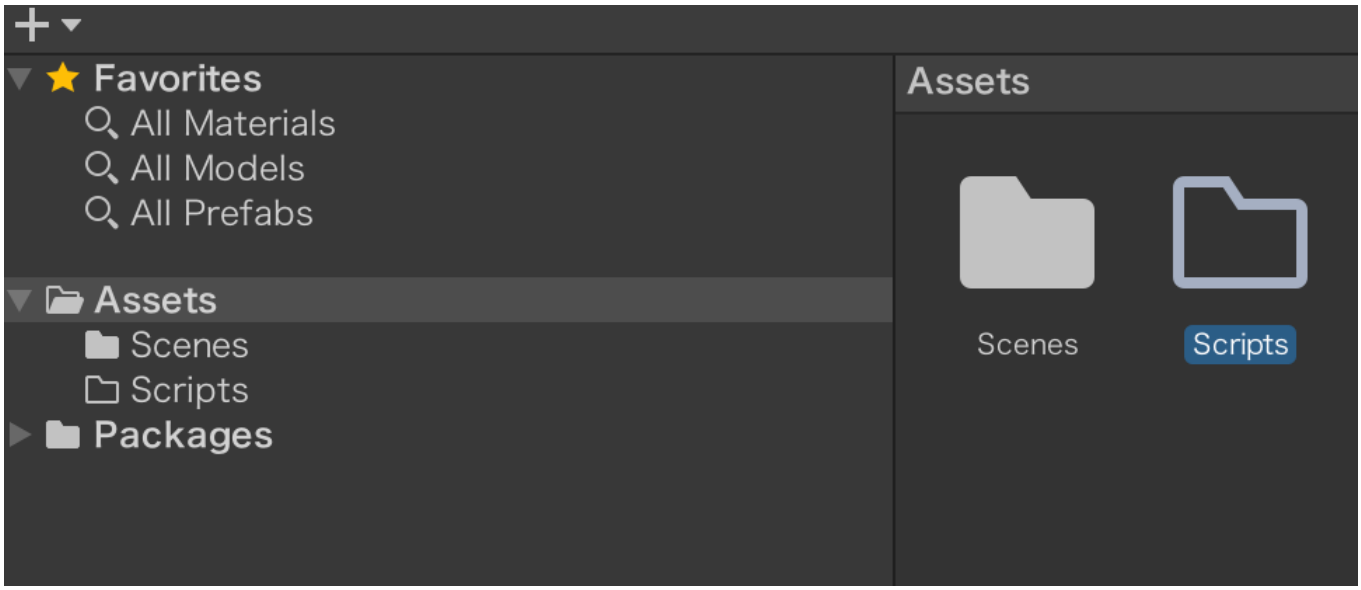
- 开始编写第一个脚本

游戏对象的行为由附加的组件控制

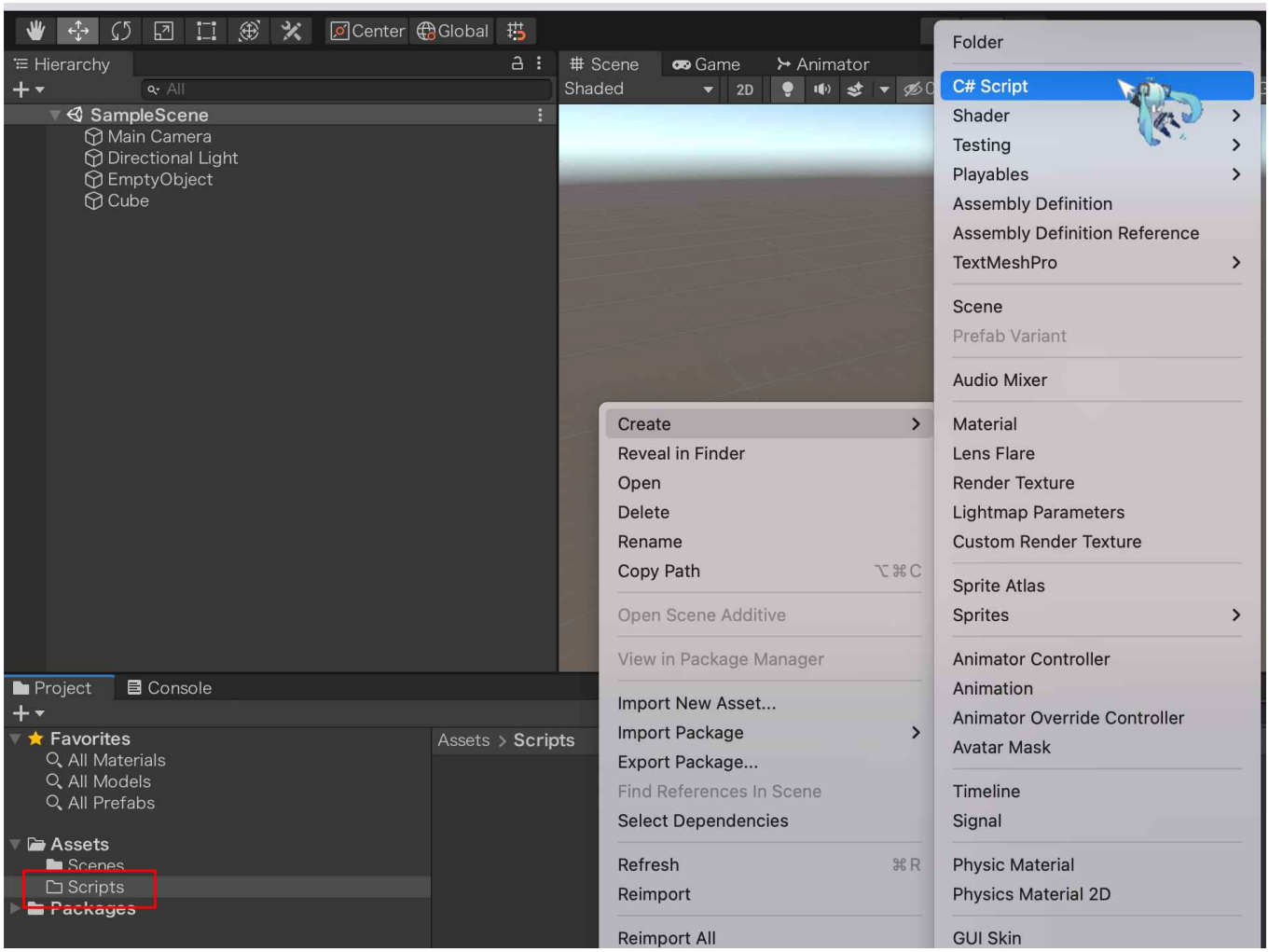
Unity支持使用C#编写脚本，此外，只要其他.NET语言能够编译兼容的DLL，也可以编写Unity

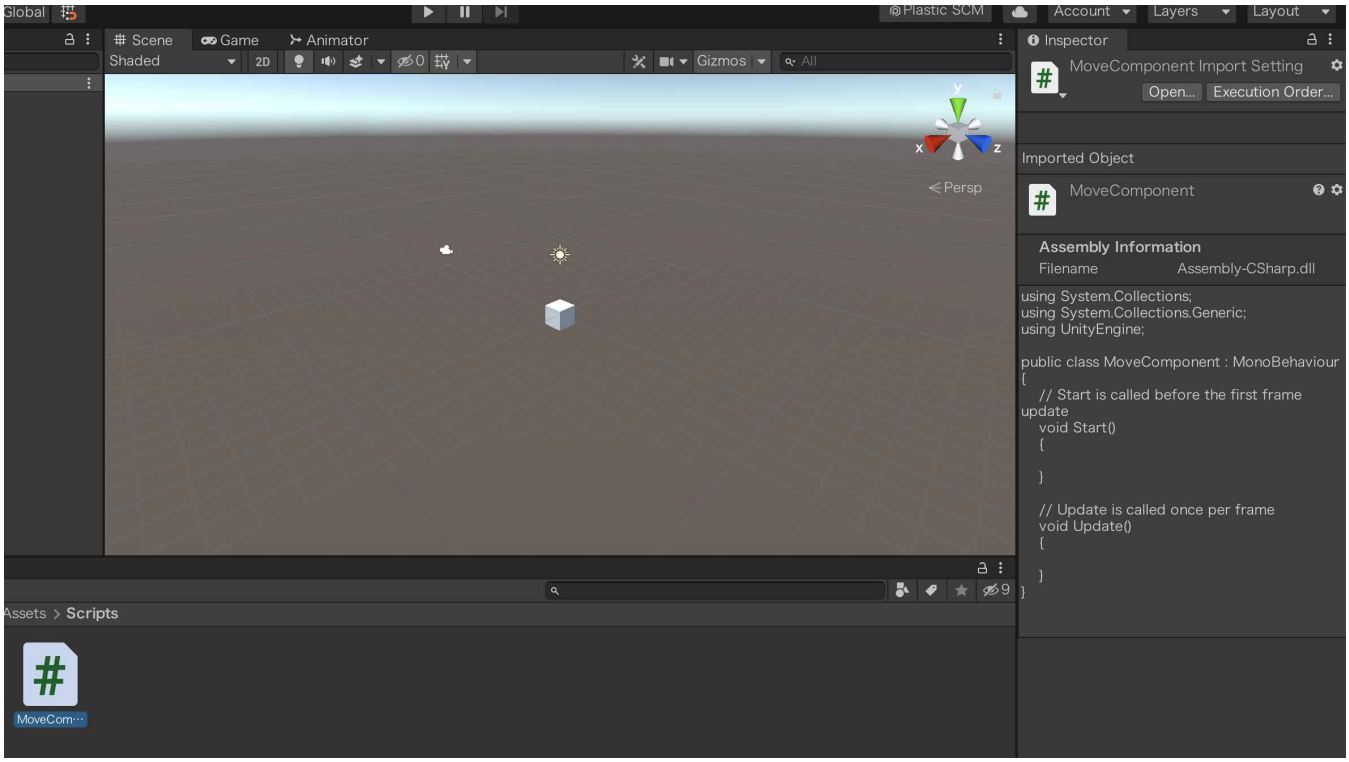
点击Assets，在右侧Assets窗口的空白处右键创建一个文件夹并重命名为Scripts用于存放脚本文件



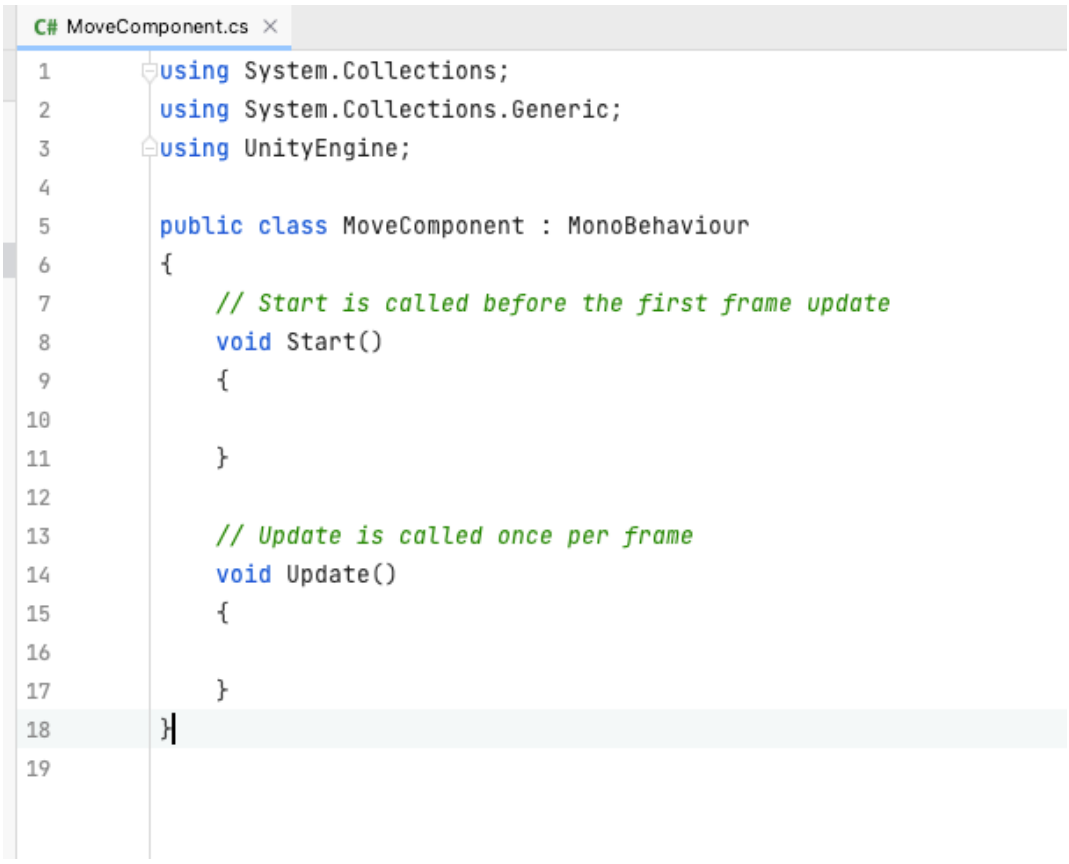


进入Scripts文件夹，在右侧Scripts的空白处右击创建一个C# 脚本并重命名为MoveComponent





点击Open在IDE中打开项目并打开MoveComponent脚本，这里我使用的是Rider，其他IDE类似



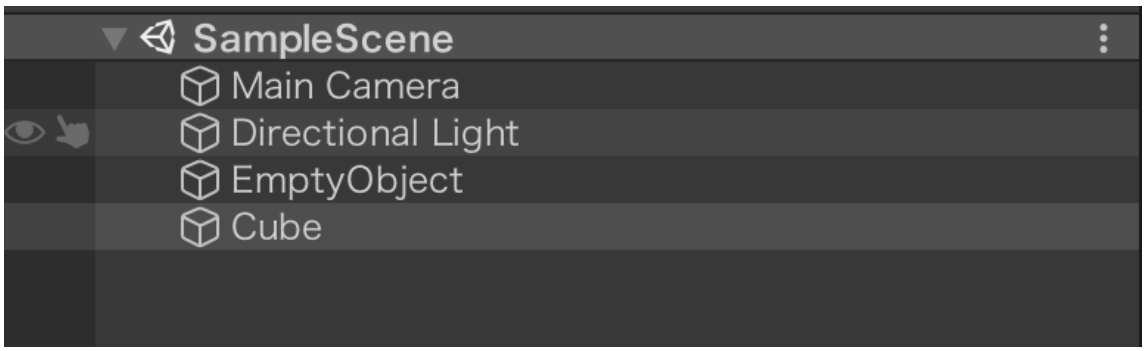
在Update函数中添加一段代码，这段代码的含义为按下w会使物体向上运动

```
if (Input.GetKey(KeyCode.Space))
{
    transform.Translate(Vector3.up * Time.deltaTime);
}
```

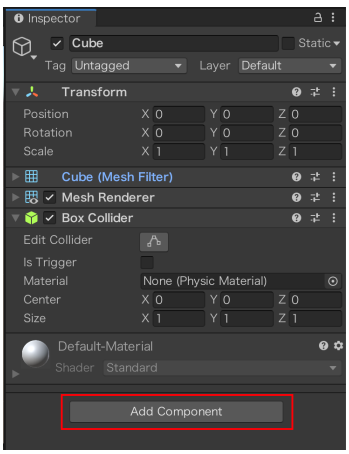
```
// Update is called once per frame
Event function
void Update()
{
    if (Input.GetKey(KeyCode.Space))
    {
        transform.Translate( translation: Vector3.up * Time.deltaTime);
    }
}
```

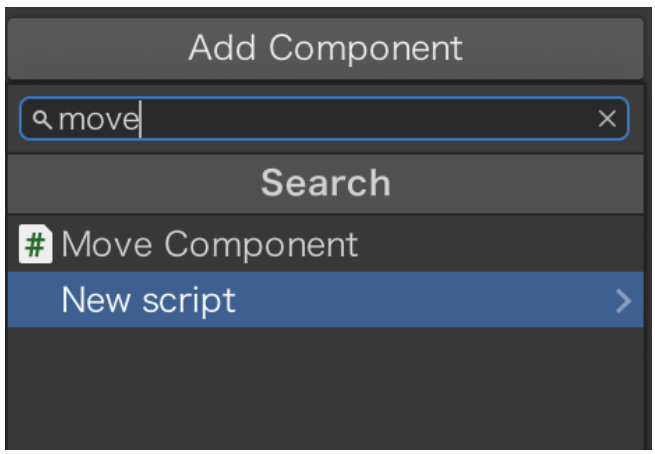
- 将脚本挂载到Cube上

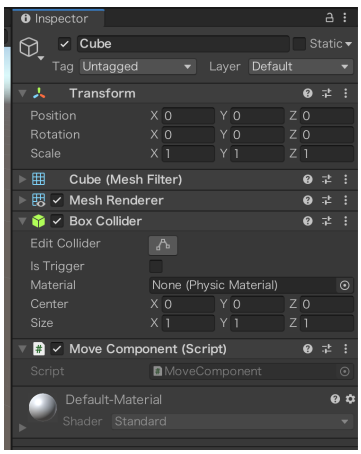
回到Unity编辑器，在Hierarchy中选中Cube



在Inspector中点击AddComponent







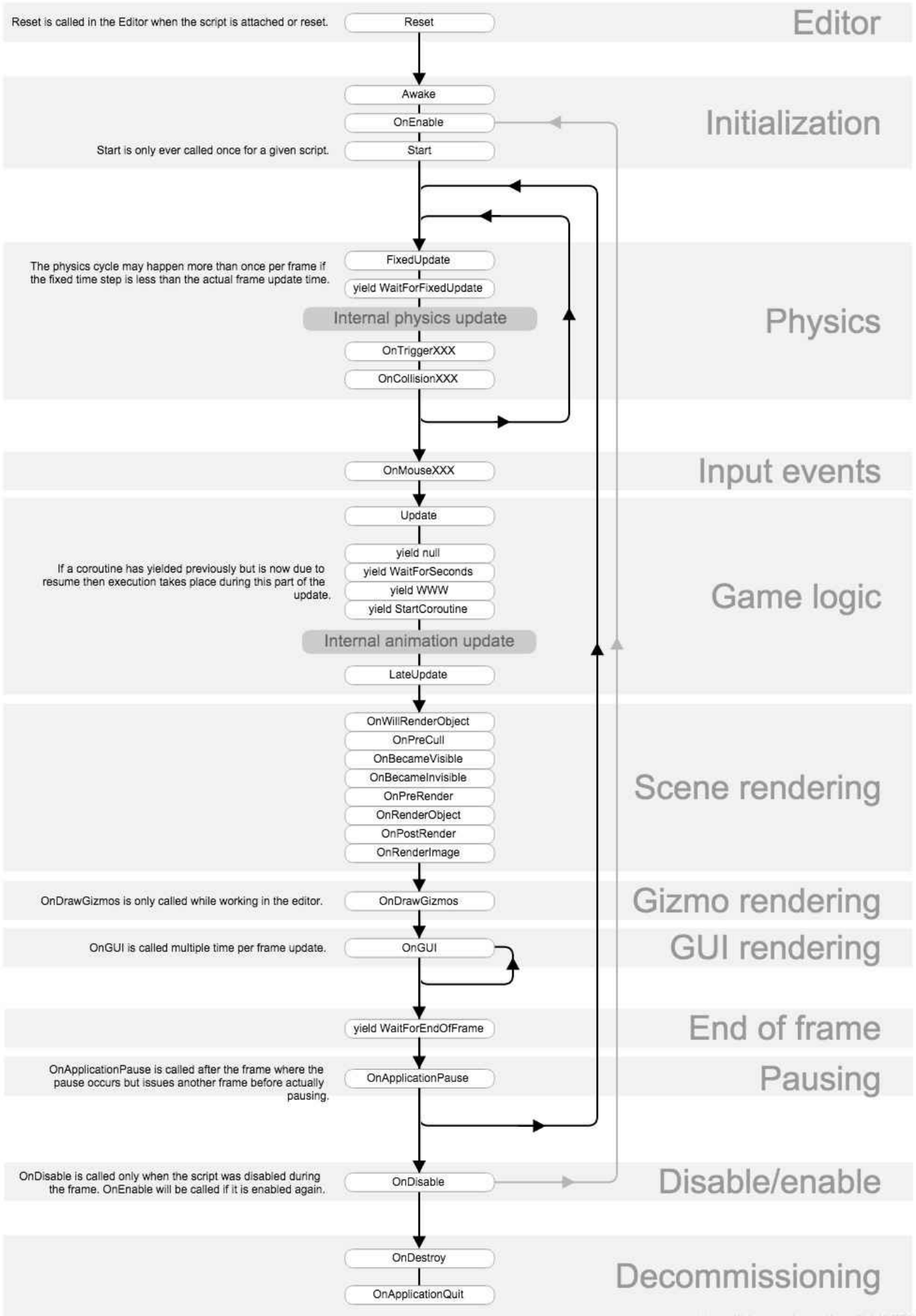
开始运行，这时按下空格键，Cube会开始向上运动

4. MonoBehavior

Unity所有的脚本都统一继承于MonoBehavior

4.1 生命周期

MonoBehavior的生命周期如下



注：其中，Phtsics、Input Event、Game Logic、Scene rendering、Gizmo rendering、End of frame、Pausing部分的函数，只会在物体被激活时调用，物体在非激活状态下不会调用这些

4.1.1 Editer 编辑器

Reset	首次被添加到对象时或点击reset按钮时调用
OnValidate	在设置脚本属性时会调用

4.1.2 Initialization 初始化

Awake	唤醒事件，脚本实例化的时候被调用，只会调用一次，通常在这里完成成员变量的初始化
OnEnable	物体变为激活时调用，实例化时会在Awake之后，Start之前调用一次
Start	在Awake之后调用，在Update第一次调用之前调用，通常把一些需要依赖Awake的变量在这里初始化，比如获取组件，同时，大多也在这个类中执行StartCourtine进行一些协程的触发

4.1.3 Physics 物理逻辑

FixedUpdate	每个物理时间间隔调用，每秒固定次数
OnTriggerXXX	处理碰撞相关逻辑
OnCollisionXXX	处理碰撞相关逻辑

4.1.4 Input 输入事件

OnMouseXXX	处理鼠标相关逻辑

4.1.5 Game Logic 游戏逻辑

Update	每个游戏帧调用，时间间隔不固定，每秒调用次数不固定
LateUpdate	每个Update执行结束后调用

4.1.6 Scene rendering 场景渲染

OnWillRenderObject	如果对象可见，则会为每个相机调用一次
OnPreCull	在相机剔除场景之前调用函数，相机是否可见对象取决于剔除
OnBecameVisible/OnBecameInvisible	在对象对于相机可见/不可见时调用
OnPreRender	在相机开始渲染场景之前调用此函数
OnRenderObject	在完成所有常规场景渲染后调用此函数，可以在这个函数使用GL类或者Graohics.DrawMeshNow绘制自定义图形
OnPostRender	在相机完成场景渲染后调用此函数
OnRenderImage	在完成场景渲染后调用此函数，可以对屏幕图像进行处理

4.1.7 Gizemo rendering 自定义渲染

OnDrawGizmos	只能在编辑模式下被调用，用于在场景中绘制调试用的UI，以实现数据可视化

4.1.8 GUI rendering GUI渲染

OnGUI	每个游戏帧至少调用两次，用于绘制GUI

4.1.9 Pausing 暂停

OnApplicationPause	程序暂停时调用

4.1.10 Enable/Disable 物体激活状态

OnEnable	物体变为激活时调用，实例化时会在Awake之后，Start之前调用一次
OnDisable	物体变为非激活时调用，在物体被销毁时会被调用一次

4.1.11 Decommissioning 物体卸载

OnDestroy	当这个脚本被销毁时调用，这个函数只会在已被激活的游戏物体上调用
OnApplicationQuit	程序退出后调用

4.1.12 生命周期函数常见问题

- Awake和Start的区别

Awake是在实例化之前就会调用，Start是在实例化之后调用，如果在Awake中执行了SetEnabled == false，则不会再执行Start函数

- Update和FixedUpdate的区别

Update的调用是根据游戏帧的间隔，因此间隔时间不固定，内部计算时间时需要使用到Time.deltatime

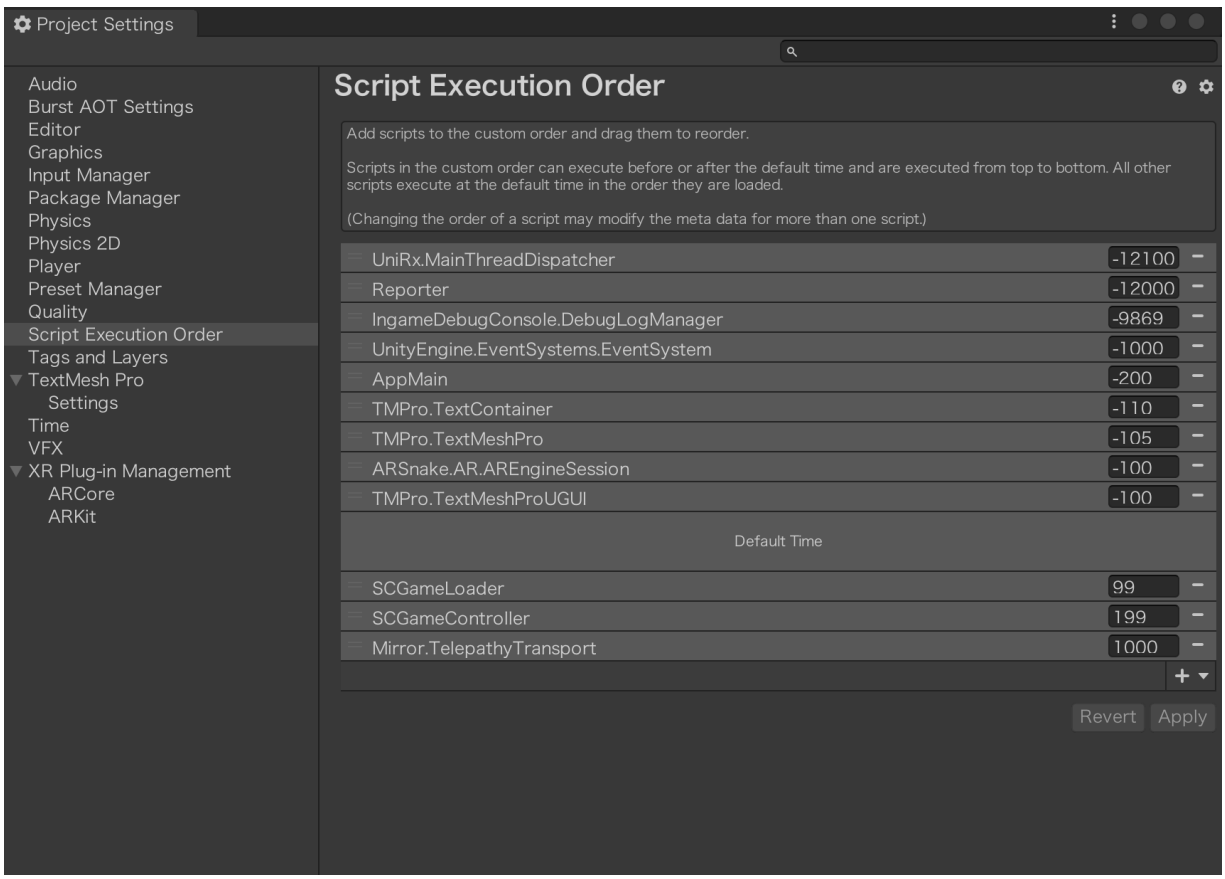
FixedUpdate的调用是根据物理帧的间隔，因此间隔时间固定

- 多个脚本的生命周期执行顺序

假设有A、B两个脚本，一般来说，触发顺序如下

A.Awake()->A.OnEnable()->B.Awake()->B.OnEnable()->A.Start()->B.Start()->A.Update()->B.Update

A、B的调用顺序先后取决于Project Settings的Script Execution Order



4.2 参数的使用

```
1 // 编辑器中显示参数
2 public string CustomName1;
3
```

```
4  // 编辑器中显示参数
5  [SerializeField]
6  private string CustomName2;
7
8  // 编辑器中隐藏参数
9  [HideInInspector]
10 public string CustomName3;
```

4.3 常用的属性和方法

- enabled: 是否已激活
- transform: 当前脚本挂载物体的Transform脚本
- gameObject: 当前脚本挂载的物体
- tag: 当前脚本挂载的物体的标签
- name: 当前脚本挂载的物体的名字
- GetComponent(System.Type type): 获取当前脚本挂载物体下类型为type的脚本
- Instantiate(Object obj): 生成对象
- Destroy(Object obj): 销毁对象

注: Destoroy销毁物体需要使用Destroy(gameObject)而不是Destroy(this)，因为this是指这个脚本，而gameObject才是挂载的物体

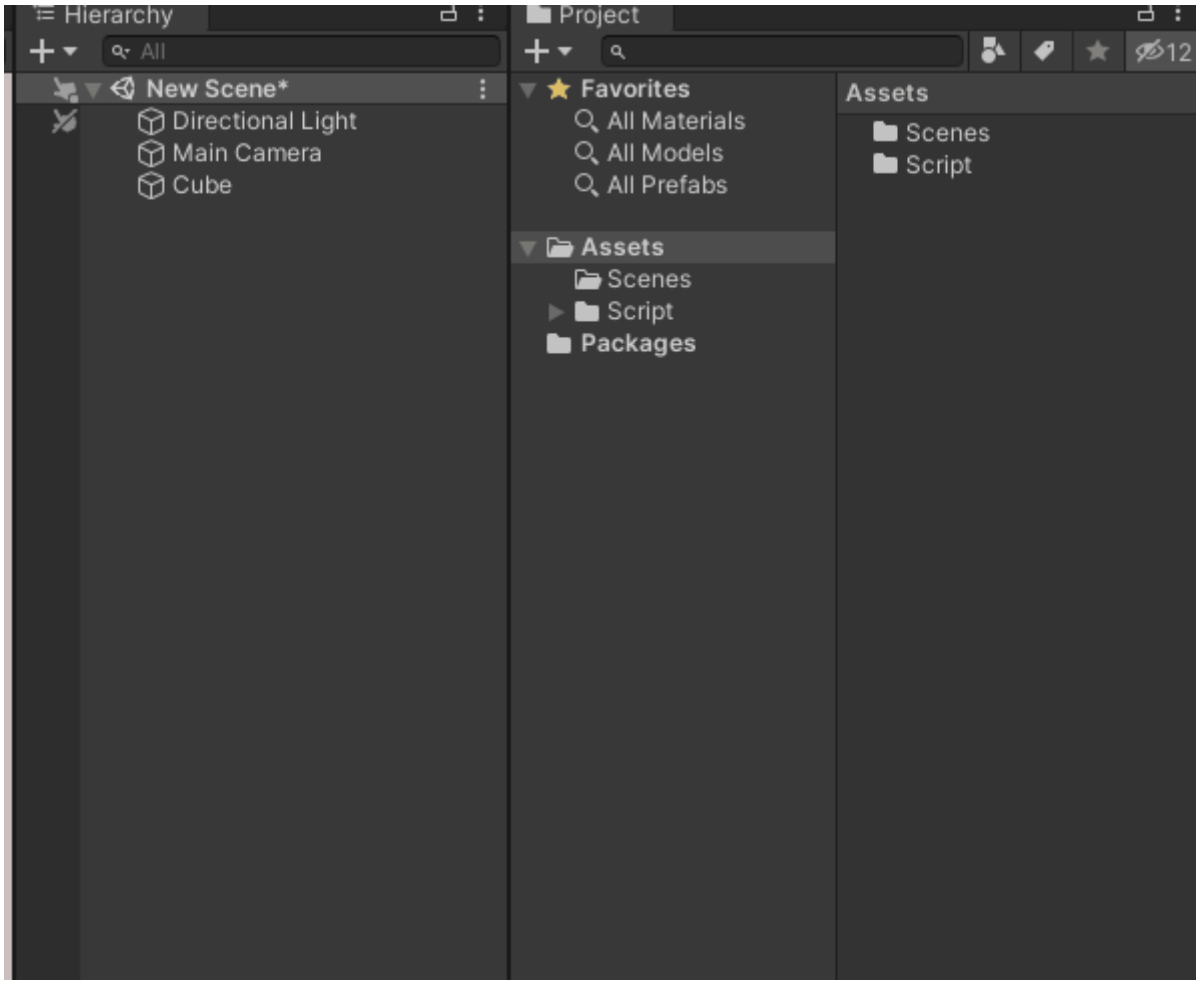
- FindObjectOfType(Object obj): 查找对象

5. 预制体

预制体是一个已包装好内容的GameObject，主要目的是可以重复利用资源

5.1 预制体的制作

创建一个对象，将其拖至存储预制体的文件夹中



5.2 预制体变体

相当于原始预制体的一个派生，可以达到预制体继承的目的

原始预制体的改动会对所有变体产生影响

5.3 预制体的使用

```
1 public GameObject prefabObject;
2 Instantiate(prefabObject, transform.position, transform.rotation);
```

5.4 预制体的复用

由于Unity中new一个对象后会分配一定的空间，用完之后即使删除对象，也没有释放内存，只有达到一定的内存量后才会触发垃圾回收GC释放内存，一次GC需要消耗大量的资源，很容易出现卡顿，因此需要使用缓冲池解决这个问题

目前AR玩法中使用的方式是使用缓存池，缓存池中会持有一个队列对象

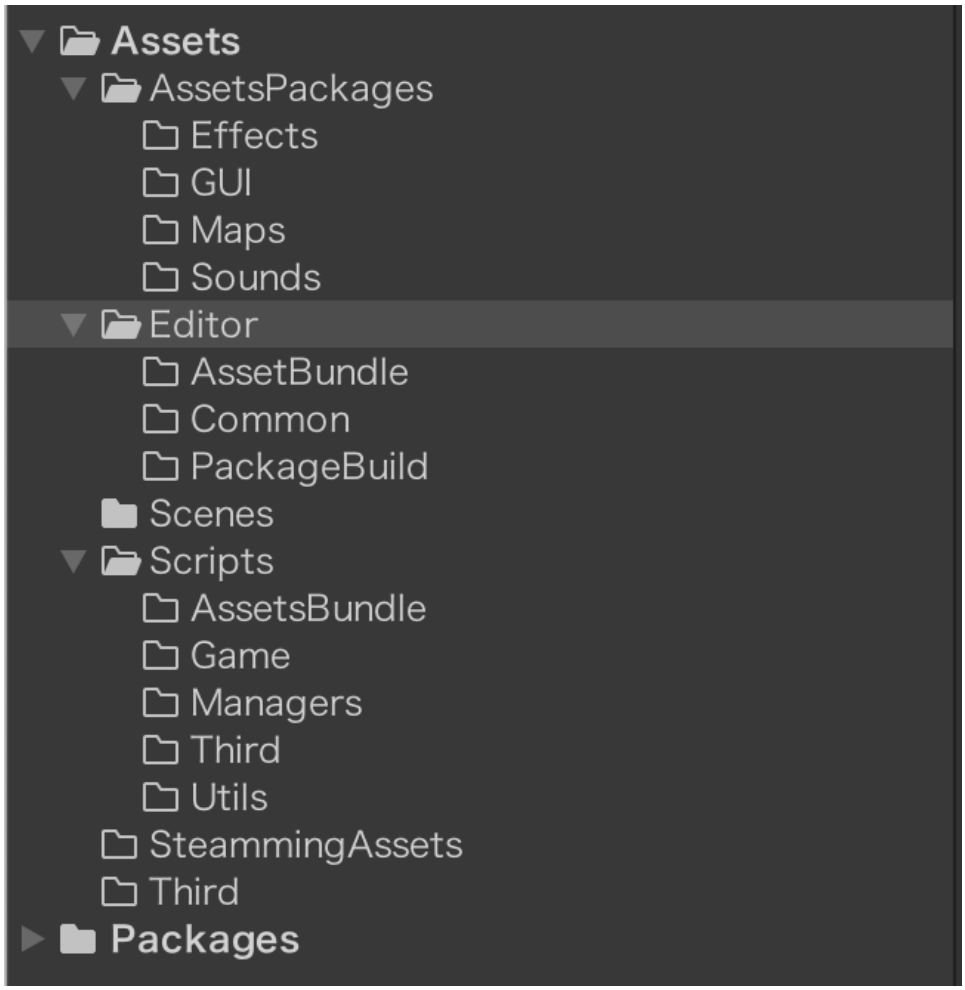
1. 生成对象时，如果队列中有未使用的对象，则出队列一个对象并使其激活，对其进行赋值；如果队列没有未使用对象，则创建一个新的对象

2. 销毁对象时，将对象入队列并使对象变为非激活

使用这种方式可以实现在游戏进行中减少物体实例的生成和销毁，可以避免频繁的触发GC

6. 项目结构

这里列举一个比较常见的项目结构



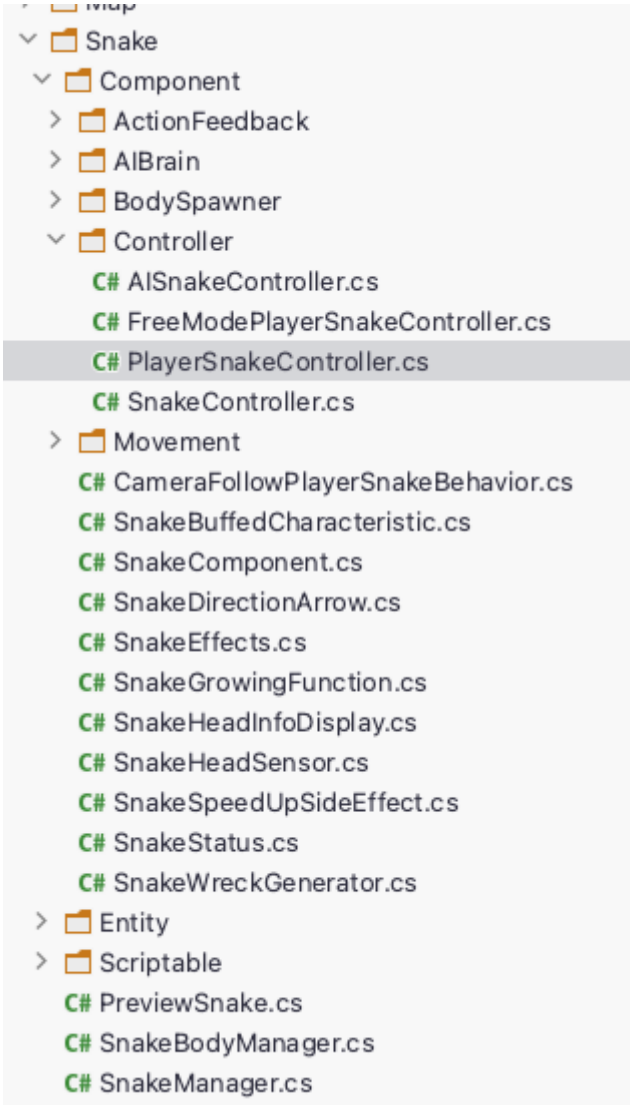
- AssetsPackage用于存放游戏资源
  - GUI用于存放GUI代码
  - Sounds用于存放音乐与音效
  - Maps存放地图相关文件
  - Effects存放粒子效果
- Editor用于存放框架与项目编辑器扩展代码
  - AssetBundle用于存放AB包打包的编辑器扩展
  - Common用于存放一些编辑器扩展的通用代码
  - PackageBuild用于存放一些打包的工具代码

- Scenes用于存放游戏场景
- Scripts用于存放游戏代码
  - Third用于存放第三方的C#代码
  - Game用于存放游戏逻辑
  - Utils用于存放工具类
  - Managers用于存放管理模块
  - AssetsBundle用于存放AB包更新代码
- Third用于存放第三方插件
- SteammingAssets用于存放打包后的AB资源

由于Unity的MonoBehavior继承自Component，为了解耦和逻辑复用，会在同一个物体下挂载多个Component实现不同的功能

目前AR玩法采用的就是这种设计

以蛇为例，区分了移动、碰撞、UI、动效、控制、AI逻辑等多个Component，这样可以实现各个功能模块之间低耦合，逻辑较为明确



7. UGUI

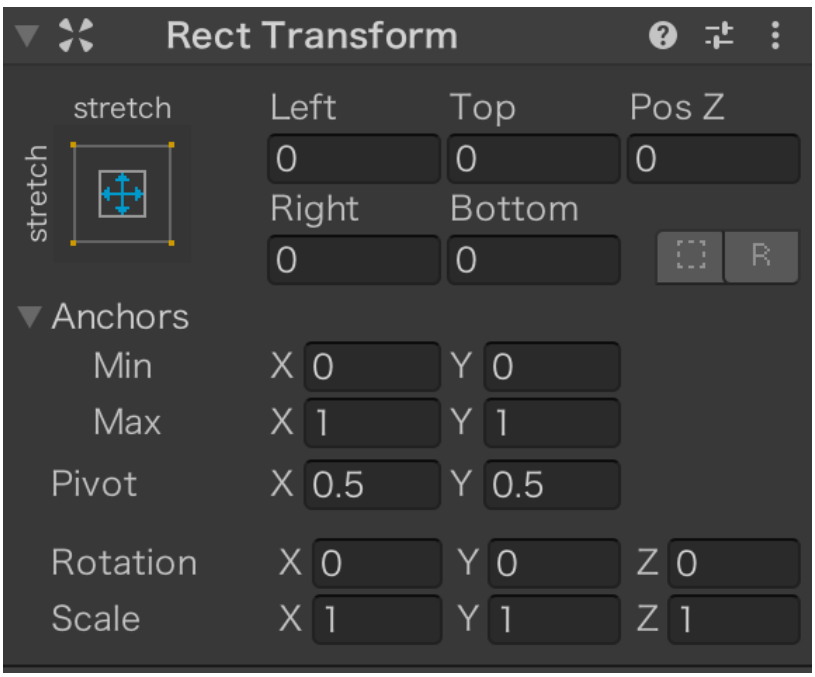
Unity的GUI框架有很多，这里举例说明一个比较常用的框架

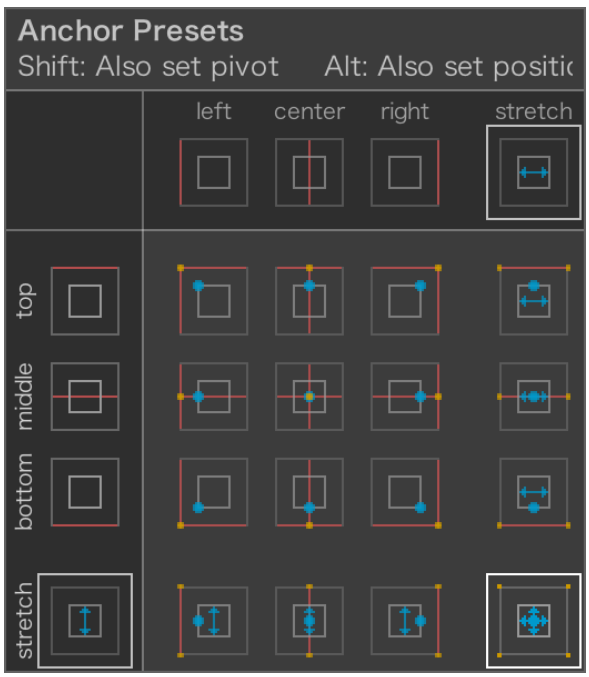
UGUI的显示主要依赖Canvas画布组件，它负责渲染自己的所有UI子对象

7.1 常用基础控件

- Text: 文本组件，类似UILabel
- Image: 图像组件，用于显示Sprite
- RawImage: 原始图像组件，用于显示Texture
- Button: 按钮组件，类似UIButton
- Toggle: 开关按钮组件，类似于UISwitch
- InputField: 输入框组件，类似UITextView
- Slider: 滑动条组件
- ScrollBar: 滑动组件，类似UIScrollView
- DropDown: 下拉框组件

7.2 布局



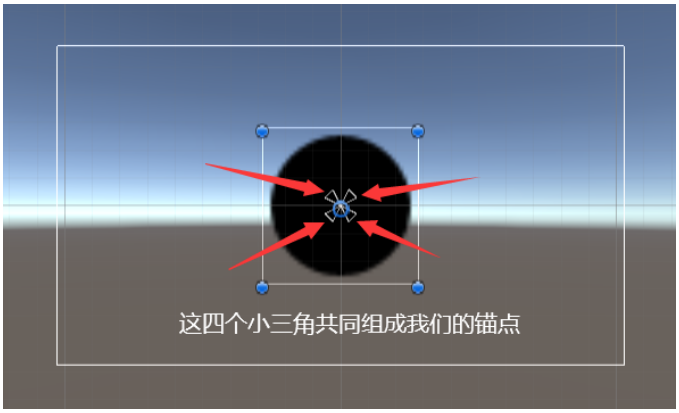


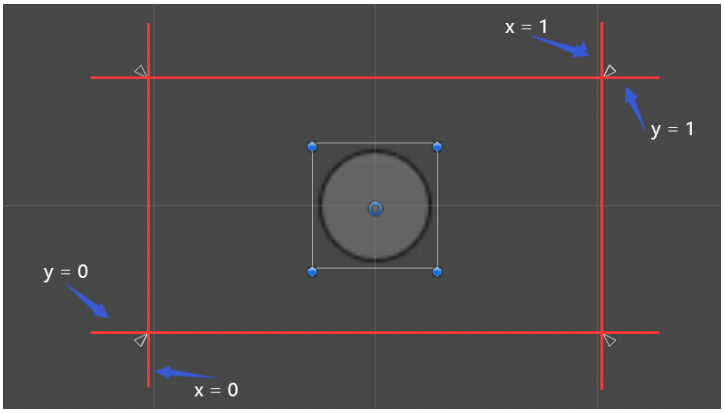
每个UGUI物体都包含一个Rect Transform组件，用于控制他的显示

stretch用于控制相对于父物体的填充方式

stretch右侧的数值控制stretch的位置信息

Anchors用于控制物体的锚点，minX、maxX、minY、maxY分别代表物体的四条边，用于计算与父物体的相对位置





Pivot也可以表示相对于父物体的位置，功能与Anchors类似

Rotation用于控制物体的旋转

Scale用于控制物体的缩放

8. Unity Attribute

- RuntimeInitializeOnLoadMethod

  标记方法用于全局初始化

  调用顺序为

  SubsystemRegistration -> AfterAssembliesLoaded -> BeforeSplashScreen -> BeforeSceneLoad -> Awake

  -> OnEnable -> AfterSceneLoad -> Start

  - RuntimeInitializeLoadType
    - AfterSceneLoad 在第一个场景的Awake执行后执行
    - BeforeSceneLoad 在第一个场景的Awake调用前执行
    - AfterAssembliesLoaded 加载完成所有程序集并初始化预加载资源时执行
    - BeforeSplashScreen 启动动画显示之前执行
    - SubsystemRegistration 用于子系统注册的回调
- HideInInspector

在属性编辑器中隐藏

- RequireComponent

添加组件之间的依赖关系，可以添加多次，每次最多添加3个依赖

当该组件被添加到对象上时，会自动添加依赖的组件

当该组件依赖的组件被删除时，会报错提示某组件存在依赖关系

```
[RequireComponent(typeof(ComponentName1), typeof(ComponentName2),
typeof(ComponentName3))]
public ComponentName: MonoBehaviour
{}
```

- SerializeField

序列化私有字段

一般情况下，unity只会序列化公共字段，但是当私有字段添加SerializeField时，他也会被序列化

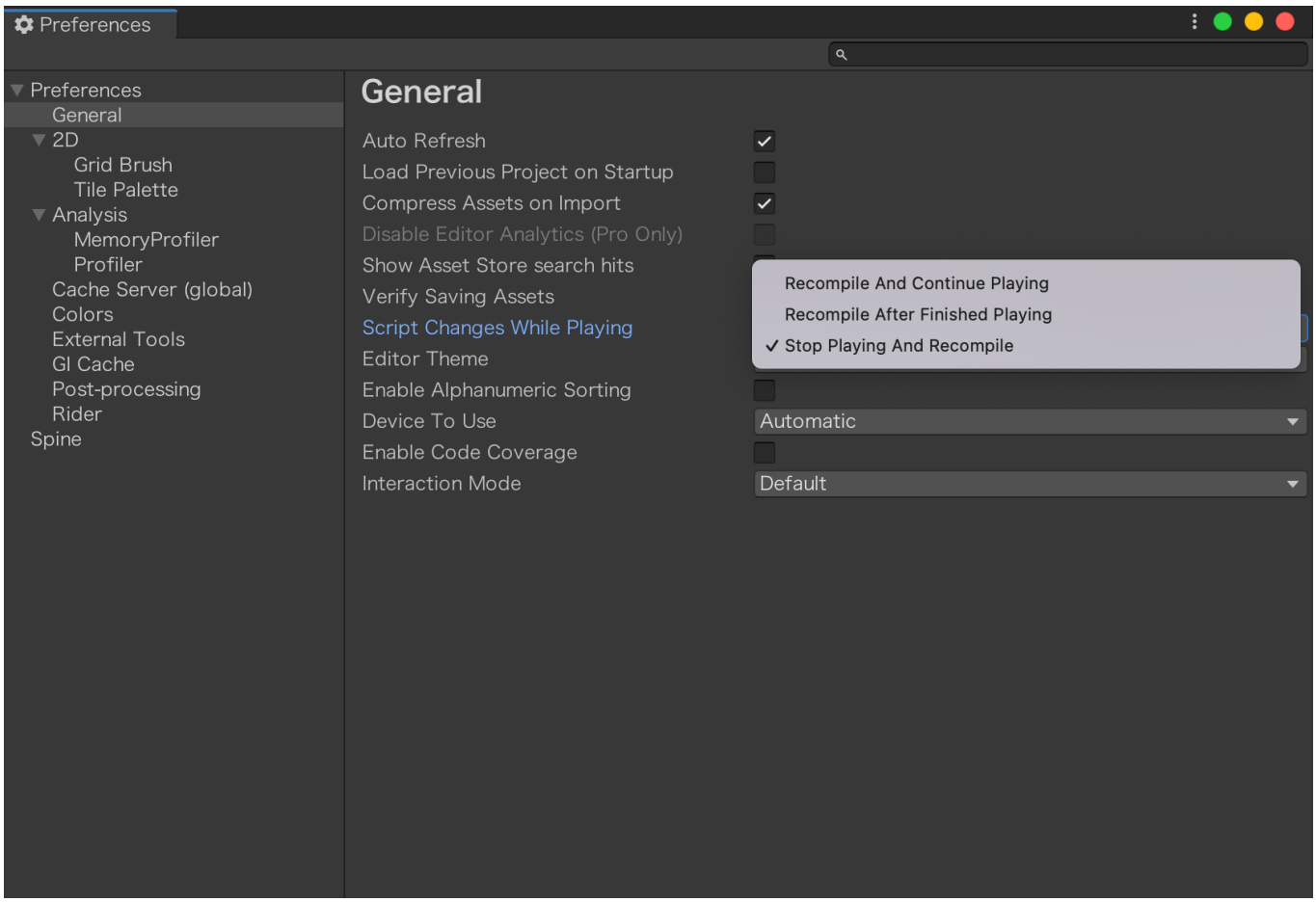
- Tooltip

属性编辑器上可以鼠标悬停查看的提示信息

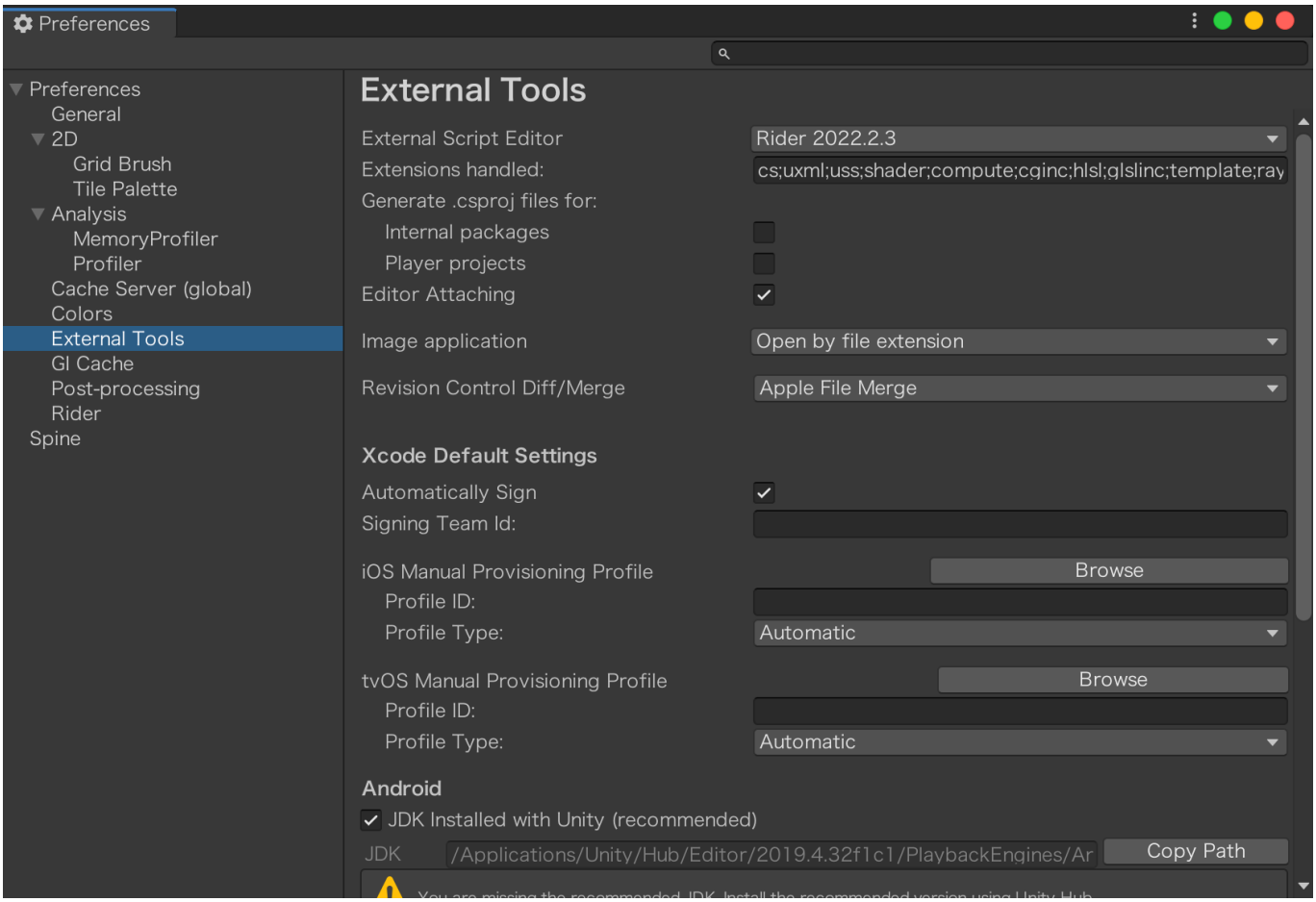
```
[Tooltip("tips")]
public string Str;
```

9. 一些其他的经验总结

- 编译设置建议修改为Stop Playing And Recompile



- 开发工具的设置在External Tools里



- 项目中有可能出现一些单例对象，重新启动后可能会为空，可以选择reload进行清除，但是启动时会更慢