# 2016/8/28

一、数据类型

1. 值类型
2. 简单值类型：整数类型、字符类型、实数类型和布尔类型

* **整数类型**：

short：有符号16位整数，取值范围-32768-32767；

uShort：无符号16位整数，取值范围0-65535；

Int：有符号32位整数，取值范围-2147489648-2147483647；

uint：无符号32位整数，取值范围0-4294967295；

long：有符号64位整数，取值范围-2^63-2^63;

ulong：无符号64位整数，取值范围0-2^64；

* **字符类型**：

char：字符类型，用单引号，'陈'，'A'

string：字符串类型，用双引号，“你好，大家好”

* **实数类型：**

float：32位单精度实数，取值范围1.5\*10^-45-3.4\*10^38

Double：64位双精度实数，取值范围5.0\*10^-324-1.7\*10^308

* **布尔类型bool**

取值只能是ture或者false

1. 复合类型：结构类型（Struct）、枚举类型（enum）

struct person

{

string m\_name;

int m\_age;

string m\_sex;

}

enum Weekday

{

Sunday,Monday,Tuesday……,Saturday

}

1. 引用类型
2. 类Class

类是一组具有相同数据结构和相同操作的对象集合。创建类的实例必须使用关键字new来进行声明

类和结构有些类似。区别：结构是值类型，类是引用类型。值类型：每个变量直接包含自身的所有数据，每创建一个变量，就在内存中开辟一块区域；引用类型：每个变量只存储对目标存储数据的引用，每创建一个变量，就增加一个只想目标数据的指针

1. 接口Interface

应用程序相互调用所达成的协议

1. 委托delegate

用于封装某个方法的调用过程

1. 数组array

一维数组

int[] array1= new int[5]; //声明，并指定长度

int[] array2= new int[]{1, 3, 5, 7, 9}; //声明，并设置值

int[] array3 = {1, 3, 5, 7, 9};

多维数组

int[ , ] num= new int[2,3];

int[ , ] num= {{1, 2, 3 }, {4, 5, 6}};

int[ , ] num= new int[2,3]{{1, 2, 3 }, {4, 5, 6}};

二、面向对象

1. 概念

面向对象Object Oriented，OO

指在程序设计中采用封装、继承、抽象等设计方法

1. 特征
2. 抽象

抽象就是忽略一个主题中与当前目标无关的那些方面，更充分地注意与当前目标有关的方面；

过程抽象：任何一个明确定义功能的操作都可被使用者看作单个的实体看待

数据抽象：数据类型和施加于该类型对象上的操作

1. 继承

联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。

派生类（子类）

基类（父类）

1. 封装

是对象和类概念的主要特性

把过程和数据包围起来，对数据的访问只能通过已定义的接口。

1. 多态性

指允许不同类的对象对同一消息作出响应。

多态性包括参数化多态性和包含多态性

多态性语言具有灵活、抽象、行为共享、代码共享的优势。

三、访问修饰符

**Public**共有访问，不受任何限制

**Private**私有访问，只限于本类成员访问，子类、实例都不能访问，

Protected保护访问，只限于本类和子类访问，实例不能访问

Internal内部访问，只限于本项目内访问，其它不能访问

Protected internal内部保护访问，只限于本项目或是子类访问，其它不能访问

四、函数（方法）

函数就是可以完成一定功能，可以重复执行的代码块。

五、构造函数

六、常量和变量

1. 变量

变量本身被用来存储特定类型的数据，可以根据要随时改变变量中所存储的数据值。

1. 常量

常量是在编译时已知并在程序的生存期内不发生更改的不可变值。 常量使用 const 修饰符进行声明。 只有 C# 内置类型（System.Object 除外）可以声明为 const,  用户定义的类型（包括类、结构和数组）不能为 const。

七、操作符

C#常用操作符：

赋值操作符：=

复合赋值：+=，-=

算数运算符 ++ -- %

条件运算符>,<,>=,<=,!=,==

逻辑运算符&&,||,!

# 2016/9/4

一、while循环

while()

{

……

break; //结束循环，一般搭配if使用，提前结束

……

}

do

{

……

}

while( );

{

……

}

|  |  |
| --- | --- |
| 问题 | do  j* / ／ 自 加  去 吡 11 巴 这 里 判 断 。  ／ ／ 这 里 有 个 分 号  while （ 〕 〈 5).  Debug. Log (" j<5==  加 果 为 真 ， 返 回 到 缣 缍 循 环  ==j=" + j):// 条 件 为 则 行 过 e 大 挺 号 中 的 内 吝 |

这里后面的{}的语句有什么意义吗？加不加{}，应该都一样啊

二、Switch语句

三、for循环

四、List用法

using System.Collections.Generic;

List<int> m\_intGroup= new List<int>(); //

List<int> m\_intGroup\_a = new List<int>();

* 1. Add方法

m\_intGroup.Add(1);

m\_intGroup.Add(2);

m\_intGroup.Add(3);

* 1. AddRange方法

可以增加同类型的List的值到新的List末尾

m\_intGroup\_a.AddRange( m\_intGroup);

* 1. Clear方法

清除该List的所有内容，但该List的内存依然存在

m\_intGroup.Clear();

* 1. RemoveAt移除指定索引的值（匹配到的第一个）

m\_intGroup.RemoveAt(0);

* 1. Count获取列表的长度

int m\_getcount = m\_intGroup.Count;

* 1. 配合foreach遍历List

foreach(int value in m\_intGroup)

{

}

foreach(var value in m\_intGroup) //这里的var是一个泛型，m\_intGroup是什么类型的List，var就是什么型

{

}

* 1. RemoveAll

Test.RemoveAll(Predicate);

……

static bool Predicate(int data)

{

return data % 2 == 0?true : false;

}

* 1. 举例说明

List<int> m\_intGroup = new List<int>();

List<int> m\_intGroup2 = new List<int>();

// Use this for initialization

void Start ()

{

/\*

int i = 0;

do

{

i++;

} while (i < 6);

debug.log("fwefjowejfowe" + i);

\*/

m\_intGroup.Add(1);

m\_intGroup.Add(2);

m\_intGroup.Add(3);

Debug.Log("第1次遍历");

foreach (int value in m\_intGroup) //把三个元素全部输出

Debug.Log(value);

m\_intGroup2.Add(4); //先加了个元素4，

m\_intGroup2.AddRange(m\_intGroup); //再把前面的1，2，3，放到了4后面

Debug.Log("第2次遍历");

foreach (int val in m\_intGroup2)

Debug.Log(val);

m\_intGroup.Clear(); //

// Debug.Log("清空后第一个元素是："+m\_intGroup[0]); 这个时候再执行这句会报越界，因为找不到List[0]了

Debug.Log("第3次遍历");

foreach (int value in m\_intGroup) //所以呢，这里不会有任何东西输出

Debug.Log(value);

m\_intGroup.Add(5);

Debug.Log("第4次遍历"); //

foreach (int value in m\_intGroup) //加了个后，这里只会输出5

Debug.Log(value);

m\_intGroup2.RemoveAt(1); //移除了第二个元素

Debug.Log("移除后第2个元素的值是："+m\_intGroup2[1]); //原来是4，1，2，3，移除1，后，那这时候第二个就应该是2了

Debug.Log("第5次遍历");

foreach (int value in m\_intGroup2) //所以这里是5，2，3

Debug.Log(value);

Debug.Log("第一个List的最后长度是："+m\_intGroup.Count+"，第二个List最后长度是："+m\_intGroup2.Count);

}

右侧是执行输出的内容

五、Dictionary字典

//Dictionary<键名称Key，键值Value>

Dictionary<int, string> nameById = new Dictionary<int ,string>();

nameById.Add(1, "zhangsan");

nameById.Clear();

nameById.Count;

一、NGUI图集的使用

NGUI->Open->Atlas Maker

Create

Update

二、

NGUI->Create->2D UI

UI拉伸后不变形

UI Atlas->Sprite Details->Border设置出九宫格的样式

UI Sprite->Type设置成Sliced

# 2016/9/11

一、合并创建和删除

二、List.RemoveAll补充

Test.RemoveAll(Predicate);

……

static bool Predicate(int data)

{

return data % 2 == 0?true : false;

}

void ListRemoveAll()

{

Test.RemoveAll(data => Test1.Contains(data)); //移除Test和Test1中的共有元素，委托类型， 拉姆达表达式

}

三、Property属性

为了更好的封装性、

---------------Property.cs

pulic class Property

private int playerExp = 0; //私有变量 玩家经验

//公有属性

public int P\_exp

{

set

{

playerExp = value;

}

get

{

return playerExp;

}

}

------------------------------------------------------------------------------------------------------------proMain.cs

Property m\_Property = new Property();

m\_Property.P\_exp = 1000;

int getValue = m\_Property.P\_exp;

Debug.log("The Value: " + getValue);

四、Delegate委托

------------------------------------------------------------------------------------------------------------m\_delegate.cs

public delegate void Buy\_BabeeBunHandle(); //声明了一个委托类型

Buy\_BabeeBunHandle mydelegate = new Buy\_BabeeBunHandle( ByBun ); //**初始化字段必须用静态（全局唯一）**

//Buy\_BabeeBunHandle mydelegate; //如果不需要初始化，则这样写，则下方只会去买牛奶，如果还要买包子，则把买包子的函数去掉static

//委托要执行的函数

static void ByBun()

{

Debug.Log("去帮我 买包子！");

}

void ByMilk()

{

Debug.Log("去帮我 买牛奶！");

}

void Start()

{

mydelegate += ByMilk; //把买牛奶事件加到买包子后面了

mydelegate(); //执行委托

}

具体例子

--------------------------------------------------------------------------------------------------------------EventSystem.cs

using System.Collections.Generic;

//消息系统 广播

public static class Eventsystem

{

public delegate void GetTimeByServerHandle(ulong time); //

static public event GetTimeByServerHandle GetTimeByServer;

static public void RaiseGetTimeByServer(ulong time)

{

if(GetTimeByServer != null)

{

GetTimeByServer(time);

}

}

}

---------------------------------------------------------------------------------------------------------------另外一个cs，继承MonoBehavior的

void Awake()

{

EventSytem.GetTimeByServer += GetTimeByServerFunc;

}

void Strat()

{

//服务器返回了时间，通知UI界面刷新

EventSystem.RaiseGetTimeByServer(20160910);

}

void OnDestroy()

{

EventSytem.GetTimeByServer -= GetTimeByServerFunc;

}

//UI逻辑层，界面显示

void GetTimeByServerFunc(ulong time)

{

Debug.log("time=====" + time);

}

与NGUI结合讲解（需要重看视频）

一、与NGUI结合讲解委托（需要重看视频）

二、函数重载

void Start()

{

TestFunc();

TestFunc(1);

TestFunc(1, "jfiwe");

}

void TestFunc()

{}

void TestFunc(int value)

{}

void TestFunc(int value, string s)

{}

三、虚函数

public static GameObject GlobeRoot;

public GameObject OpenButton;

ShopNpc m\_ShopNpc = new ShopNpc;

void Awake()

{

UIEventListener.Get(OpenButton).onClick = OpenButtonFunc;

GlobeRoot = transform.parent.gameObject;

}

void OpenButtonFunc(GameObject ob)

{

}

//NPC的基类

public class NpcBase

{

private string path = "prefab/";

private string commonUI = "commonTalk";

public virtual void ShowTalkGUI() //在基类里面定义了一个虚函数

{

GameObject ui = Instantiate(Resources.Load(path+commonUI)) as GameObject;

ui.transform.parent = GlobleRoot.transform;

ui.transform.localposition = Vector3.zero;

ui.transform.localScale = Vector3.one;

}

}

public class ShopNpc : NpcBase

{

private string path = "prefab/";

private string shopTalk = "shopTalk";

public override void ShowTalkGUI() //override 关键字 重写父类的虚方法

{

//base.ShowTalkGUI(); //调用基类的函数

thisUI = Instantiate(Resources.Load(path + shopTalk)) as GameObject;

thisUI.transform.parent = GlobleRoot.transform;

thisUI.transform.localposition = Vector3.zero;

thisUI.transform.localScale = Vector3.one;

}

}

四、抽象方法

public abstract class NpcBase

{

private string path = "prefab/";

private string

}

五、字符串拼接的三种常见方式

1. 简单式-性能较差，简单

string strA = "A" + "B" + "C"; //简单，性能较差，在数量较多的拼接，就像循环拼接一样，性能较差

1. 拼接排列式-性能较高，比较简便

string strB = string.Format("{0}{1}{2}", "A", "B", "C"); //字符串拼接和排列

string strB = string.Format("{2}{1}{0}", "A", "B", "C"); //性能较高，比较简便

1. StringBuilder-性能较高，比较繁琐

using System.Text;

……

StringBuilder strB = new StringBuilder();

strB.Append("A");

strB.Append("B");

strB.Append("C");

strB.Remove(0, 2); //从第0个位置开始，删除2个字符，即会把AB删掉，只剩下C

# 2016/9/25

一、

//获取一下邮件数量 List<mailClass> someMail 长度 显示新邮件数量

//系统功能管理类 mailMgr List dc 数据存放在管理类中

//打开邮件界面时，会访问管理的数据

|  |  |  |
| --- | --- | --- |
| mailClass邮件类 | 发送者 | string |
|  | 时间 | ulong |
|  | 内容 | string |

其它需要邮件显示

二、字符串比较

1、基本类型

==和Equals比较的是2个变量的值

|  |  |
| --- | --- |
| 代码 | string a= "你好帅啊！";  string b="你好帅啊！";  Debug.Log("a==b:" + (a==b));  Debug.Log("a.Equals(b):" + a.Equals(b)); |
|  |  |

2、引用类型的比较

==和Equals比较的是内存中的地址是否相等

|  |  |
| --- | --- |
| 代码 | class Person  {  }    ……  Person p1=new Person();  Person p2=new Person();  Person p3=new Person();  p4=p3;    Debug.Log("p1==p2:"+(p1==p2));  Debug.Log("p1.Equals(p2):" + p1.Equals(p2));  Debug.Log("p3==p4:"+(p3==p4));  Debug.Log("p3.Equals(p4):" + p3.Equals(p4)); |
| 输出 | p1==p2:Fasle  p1.Equals(p2):Fasle  p3==p4:True  p3.Equals(p4):True |

三、关于字符串

C#中创建2个字符串对象时，内存中的地址不相同，但可以赋相同的值

**对于引用类型，==和Equals都是比较内存的地址是否相同，而对于字符串的Equals方法，C#中由于重载了Equals方法，所以使其看起来像一个值类型**

1、基本类型-比较值是否相等

|  |  |
| --- | --- |
| 代码 | string aa=new string (new char[] {'h', 'e', 'l', 'l', 'o'});  string bb=new string (new char[] {'h', 'e', 'l', 'l', 'o'});    Debug.Log("aa==bb:", (aa==bb));  Debug.Log("aa.Equals(bb):", aa.Equals(bb)); |
| 输出 | aa==bb:True  aa.Equals(bb):True  · |

2、引用类型-引用String对象有点特殊

==比较的还是地址，Equals比较的时值

|  |  |
| --- | --- |
| 代码 | object a1=aa;  object b1=bb;  Debug.Log("a1==b1:", (a1==b1));  Debug.Log("a1.Equals(b1):", (a1.Equals(b1))); |
| 输出 | a1==b1:False  a1.Equals(b1):True |

四、泛型

* 泛型类

|  |  |
| --- | --- |
| 代码 | public class c\_Genericity<T> //尖括号必须要  {  public T m\_value;  public c\_Genericity(T obj) //这里构造函数的public必须要，否则在声明的时候会报错  {  m\_value = obj;  }  }  ……  void Start()  {  c\_Genericity<string> genr=new c\_Genericity<string>("你好美啊！");  Debug.Log("m\_value=" + genr.m\_value);  } |
| 输出 | m\_value=你好美啊！ |

* 泛型方法

|  |  |
| --- | --- |
| 代码 | public T ReturnElement<T>(T obj)  {  T result=obj;  return result;  }  ……  void Start()  {  int intResult=ReturnElement<int>(11);  Debug.Log(intResult);  string strResult = ReturnElement<string>("你好美啊！");  Debug.Log(strResult);  } |
| 输出 | 11  你好美啊！ |

五、接口

接口是一个引用类型，通过接口实现多重继承

在C#中，接口的成员不能有new，Public，Private，Protect等修饰符

接口中只能声明抽象成员，所以不能对接口进行实例化，不能用new操作符声明一个接口的实例对象

接口的声明不包括数据成员，只能包括属性、方法、事件等成员

接口的成员一般都以“I”作为首字符，也是接口和类的区别之一

接口成员的默认访问级别默认就是公有，不要再指定任何修饰符

接口成员不能有static， abstract，override，virtual修饰符，使用new的话不会报错，但会警告说不需要new

在声明接口成员的时候，不需要为成员编写可执行代码

接口一旦被实现，实现类必须实现接口中的所有成员，除非实现类本身是一个抽象类

|  |  |  |  |
| --- | --- | --- | --- |
| 代码  animal.cs | //不需要继承mono  interface animal  {  void bark(); //默认公有  } | 代码  Dog.cs | public class Dog:animal //声明了后必须实现animal的所有成员  {  public void bark() //继承接口的方法需要声明是公有的  {  Debug.Log("汪汪汪！！！");  }  }    public class Cat:animal  {  public void bark()  {  Debug.Log("喵喵喵！！！");  }  } |
| 代码  InterfaceMain.cs | void Start()  {  Dog m\_dog=new Dog();  m\_dog.bark();  Cat m\_cat=new Cat();  m\_cat.bark();  } |  |  |

六、协程

C#迭代器

* IEnumerator关键字
* StartCoroutine调用协程方法
* yield关键字，延迟执行，实现迭代
* StopCoroutine停止协程

|  |  |
| --- | --- |
| 代码 | IEnumerator xiecheng()  {  Debug.Log("我是协程前面的狗");  yield return 0; //**延迟返回一帧，当遇到yield时，会交出执行权，执行本函数外面的语句，指定时间后，重新执行yield后面的内容**  yield return new WaitForSeconds(3f); //**延迟3s继续执行，new WaitForSeconds将会导致内存垃圾GC（garbage collection） 21字节的垃圾碎片的分配，所以不要滥用**  yield return new WaitForEndOfFrame(); //**本帧结束后继续执行**  Debug.Log("我是协程后面的大花猫");  }    void Start()  {  Debug.Log("啦啦啦，我是小画家");  StartCoroutine(xiecheng()); //**注意协程的调用方法**  Debug.Log("呵呵呵，我是粉刷匠");  } |
| 代码 |  |

一、延迟调用

**Invoke("函数名"，时间);**

**InvokeRepeating("函数名"，初始延迟时间， 重复间隔时间）；**

**CancelInvoke(); //停止重复调用的方法，一旦碰到CancelInvoke，立马会停止延迟调用的，不指定的话会停止所有延迟调用的方法**

在某些场景可以替代协程使用

1. 写在脚本声明周期（Start、Awake、Update）调用
2. Invoke不能接受含有参数的方法
3. Time.timeScale=0的情况下，Invoke（）无效，无法调用到，

|  |  |
| --- | --- |
| 代码 | void AFunc()  {  Debug.Log("延迟调用！");  }    void Start()  {  Debug.Log("开始！！");  Invoke("AFunc", 3f); //延迟3s执行AFunc函数  InvokeRpeating("AFunc", 1f, 60f); //1秒后，开始调用AFunc，然后每隔60s调用一次  } |
| 使用举例 | * 1. 第n关完成结算了，请求进入下一关（请求服务器）   2. 结算完成后，播放一个特效（显示通关奖励，或者胜利等），3s，播放完      * 1. InvokeRepeating做心跳包：向服务器发送消息，接收服务器消息，交互，同步一个服务器时间，比如：今晚8点，副本界面，开放倒计时，倒计时效果，活动开放后，服务器再通过消息下推通知客户端 |

二、C#异常处理

C#异常处理，关键字

|  |  |
| --- | --- |
| **关键字** | **作用** |
| **try** | 用于检查发生的异常，并帮助发送任何可能的异常 |
| **catch** | 以控制权更大的方式处理错误，可以用多个catch字句 |
| **finally** | 无论是否引发了异常，finally的代码块都将被执行 |

|  |  |
| --- | --- |
| 代码 | using System;    void Test()  {  int i=0;  i=12/i;  }    void Start()  {  try  {  Test();  }  catch(Exception ex) //Exception需要System的名空间  {  Debug.LogError("ex"+ex);  }  finally  {  Debug.Log("无论有无异常，都继续执行");  }  } |
| 输出 | 会输出一个试图除以0的异常提示 |

三、线程 Thread

|  |  |
| --- | --- |
| 进程 | 程序运行的时候，就是一个进程，进程包括运行中的程序和程序所使用的内存和系统资源。一个进程由多个线程组成 |
| 线程 | 线程是程序中的一个执行流，每个线程有自己专有的寄存器（栈指针，程序计数器等），但是代码区是共享的，就是说不同的线程可以执行相同的函数（方法） |
| 多线程 | 程序中包含多个执行流，在一个程序中同时运行多个不同的线程来执行不同的任务，也就是说允许单个程序创建多个并执行的线程来完成不同的任务  **优点：**提高CPU的利用率，当一个线程必须等待的时候，CPU可以运行其它的线程，而不是等待，这样提高效率  **缺点：**线程也需要占用内存，线程越多占用内存也越多，多线程还需要管理协调，CPU需要处理线程，线程对共享资源的访问也会有一定影响，会造成控制复杂，容易产生bug |

|  |  |
| --- | --- |
| 客户端连接服务器端 | 采用的是异步网络连接，通过IP地址和端口进行网络连接的请求（异步）->连接上  而此时主线程发完请求后，不等待服务器回复，先去做如加载资源、播放美术效果等的事情  当服务器端返回连接状态以后，不在主线程 |

静态方法

|  |  |
| --- | --- |
| 代码  m\_Thread.cs | using System.Threading;  public class m\_Thread{  public static void StartThreadFunc()  {  Debug.Log("新线程1 静态方法");  Thread.Sleep(500); //毫秒，休眠的时间,本线程让出执行权,规定时间后继续执行  Debug.Log("新线程1 静态方法 继续打印！");  }  } |

实例方法

|  |  |
| --- | --- |
| 代码  m\_Thread.cs | public void otherThreadFunc()  {  Debug.Log("新线程2 实例方法");  } |

|  |  |
| --- | --- |
| 代码  MainThread.cs | using System.Threading;    public class MainThread:MonoBehaviour  {  void Start()  {  Debug.Log("在Start函数中 这里执行的是主线程");  Thread m\_thread1=new Thread(m\_Thread.StartThreadFunc); //创建新线程1  m\_thread1.Start(); //开始执行这个新线程  Thread m\_thread2=new Thread(new m\_Thread().otherThreadFunc); //  m\_thread2.Start();    Thread m\_thread3 = new Thread(delegate(){Debug.Log("新线程3 匿名委托！");}); //匿名函数的写法  m\_thread3.Start();  Thread m\_thread4=new Thread( () => {Debug.Log("新线程4 Lambda表达式"); } );  m\_thread4.Start();  }  }    void Update()  {  Debug.Log("这里执行的是主线程");  } |

**匿名函数是一个“内联”的语句或表达式，可在需要委托类型的任何地方使用**

**Lambda表达式，是一种用于创建委托或表达式目录树类型的匿名函数**

不太易读，看起来繁琐，拉姆达表达式效率非常低

# 2016/10/23

一、程序集

1. 概念

由.NET编译生成的EXE文件，或者DLL（Dynamic Link Library动态链接库）文件，这两种就是程序集

1. 生成DLL

|  |  |
| --- | --- |
| 生成dll方法 | * 1. 自己写一个类，复制到本地某位置 e:\dll\some.cs   2. windows菜单，VS2015->tools->开发人员命令提示   3. 进入对应盘符 csc/target:library /out:e:\dll\some.dll e:\dll\some.cs |

VS2015开发人员命令提示工具

|  |  |
| --- | --- |
| 命令行代码 | e:  csc/target:library /out:e:\dll\dlltest.dll e:\dll\dlltest.cs |

二、C#反射（reflection）

1. 概念

程序运行的时候，**查看和调用**其它程序集的字段和方法等元数据的行为就叫反射

反射是.NET的重要机制，通过反射，可以在运行的时候，获得.NET中的类型（类，结构，委托，接口，枚举等）的成员、方法、属性、事件、构造函数等，还可以获得每个成员的名称。

1. 查看公共字段与方法

|  |  |
| --- | --- |
| reflex.cs | dlltest m\_test = new dlltest();  System.Type type = m\_test.GetType(); //获取当前实例的类型    System.Reflection.MethodInfo[] methds = type.GetMethods(); //获取dlltest类的公共方法  System.Reflection.FieldInfo[] fields = type.GetFields(); //获取dlltest类的公共字段    foreach(var field in methds)  {  Debug.Log("=======" + field); //另外通过m\_test.出现的提示也可以看到公共方法  }    foreach(var field in fields)  {  Debug.Log("=======" + field);  } |

三、Json（JavaScript对象表示法）

1. 概念

Json ( Javascript object Notation)是一种**轻量级的数据交换格式**，是存储和交换文本信息的语法。完全独立于语言的文本格式，一种比较特殊的string字符串形式，采用了Json的语法。

（做页游的用到的Json会比较多，也是数据交互的一个方案）

1. Json语法规则

* 数据在键值对中，就是在名称和值的对中（类似字典的键值对应）
* 数据用逗号分隔
* 花括号保存对象（大括号保存对象）
* 方括号保存数组（中括号保存数据）

1. 基本写法

Json值： "name":"Jack" 名称和值（包括字段名称）在双引号中，在中间写一个冒号，冒号后面是值（键必须是字符串），值可以是数字（整数，浮点数），字符串（在双引号中），逻辑值（false或true），数组（在中括号中），对象（在花括号中），null

eg：

对象的写法：

{"firstname":"John", "lastname":"doe"}

数组的写法：

[{"firstname":"John", "lastname":"doe"}, {"firstname":"anna", "lastname":"smith"}, {"firstname":"petar", "lastname":"jones"}]

1. 访问数据

* 访问数据

string name = emplyees[0]["lastname"];

* 修改数据

employees[0]["firstname"] = "jack";

1. 示例

|  |  |
| --- | --- |
| Jsondata.txt | [  {"monsterID":"111", "name":"frog"},  {"monsterID":"112", "name":"dog"},  {"monsterID":"113", "name":"wolf"}  ] |

1. Json解析

|  |  |
| --- | --- |
| JsonParse.cs | using LitJson;    public class JsonParse:MonoBehavior  {  void Start()  {  //resources路径读取方式  TextAsset m\_text = Resources.Load("Jsondata") as TextAsset;  string str = m\_text.text;  //Debug.Log("str="+str);    JsonData jsd = JsonMapper.ToObject(str);  Debug.Log(jsd.Count);  for(int i = 0; i < jsd.Count; i++)  {  Debug.Log( jsd[i]["monsterid"] );  Debug.Log(jsd[i]["name"]);  }  }  } |

1. 用途示例

ex一个字段（扩展字段） 道具结构 id count price

道具结构 属性-111 攻击力-222 防御值-111 魔法值-

策划调整（功能反复修改） 前后端 数据交互（protoBuf前后版本的兼容） 迭代开发 SVN（版本控制）

|  |  |  |
| --- | --- | --- |
| 迭代开发 | 第一天 | UI界面搭建-界面逻辑编写 |
|  | 第二天 | 跟后端协商数据通讯协议  数据包结构（发起者ID，聊天对象ID，日期和时间，内容（string），头像ID，**ex一个字段（扩展字段string）**） |

前后逻辑-调用发起者ID，如果删除发起者ID，这时候就会报错，需要调整所有有调用ID的地方

四、RenderQ渲染序列

1. 概念

渲染序列RenderQueue（通过渲染序列设置）

1. 注意事项

* 渲染序列的值必须是正值
* 一旦在材质上设置了渲染序列，即使shader在将来被改为其他值，它仍然会保持原来设置的值

1. 工具化

渲染序列的设置采用工具化

ToolMgr-设置摄像机-设置灯光-设置渲染序列等

工具功能（整合到工具管理类中）

|  |  |
| --- | --- |
| RenderQ.cs  文件放在摄像机上 | public GameObject gameObject\_a; //这个物体放cube  public GameObject gameObject\_b; //这个物体放投石车，并且shader设置成Unlit->Transparent    void Start()  {  SetRenderQueue(gameObject\_a.transform, 100);  SetRenderQueue(gameObject\_b.transform, 101);  }  //RenderQueue设置规则    //设置渲染序列的方法  public static void SetRenderQueue( Transform trans, int renderqueue )  {  Renderer[] m\_render = trans.GetComponentsInChildren<Renderer>();  if(m\_render != null)  {  for(int i = 0; i< m\_render.Length; i++)  {  if( m\_render[i].material != null)  {  m\_render[i].material.renderQueue = renderqueue;  }  }  }  } |

一、文件路径

1. 文件（资源-文本）存放在哪里

* **Resources资源路径**

这个路径下为只读路径，需要动态更新的资源无法放在这个文件路径；

发布时会将此文件夹的资源打包到.asset文件中，建议可以放一些预制体（prefab），因为在打包的时候会自动过滤不需要的资源，有利于减少资源包大小；

必须在主线程中加载

放在这里的资源通过Resources方法加载资源：

Resources.Load("path"+"文件名（不包括文件后缀名）")

* **dataPath 资源路径**

在移动端这个路径下的文件没有访问权限

* **streamingAssetsPath 流数据资源路径**

这个路径下的资源为只读的

和Resources的区别是Resources文件中的内容在打包时会被压缩和加密，而streamAssets文件夹中的内容会原封不动的打入包中

因此这个文件夹主要用来存放一些二进制文件

只能用WWW类来获取数据

* **persistentDataPath 持久数据路径**

（1）这个路径下的资源为可读写的，不能直接存放文件，在初次运行程序时创建生成，只能在运行时写入或者读取，无法提前把数据存入这个路径，持久存在，当程序卸载时自动删除

（2）在IOS平台上是应用程序的沙盒（固化文件夹），在安卓平台可以是程序的沙盒也可以是sdcard，在安卓打包的时候，projectsetting页面上有选项，WriteAccess可以设置这个路径是沙盒还是sdcard

（3）无内容限制，可以从streamingAssetsPath读取二进制文件或者AssetBundle读取文件来写入persistentDataPath中

（4）写好的数据，在电脑设备上可以查看到的，也可以清除掉，应用程序卸载后自动删除此文件路径

SQLite是一款轻型可视化工具，将CSV文件导成DB后缀的文件

数据库文件（存放在streamingAssetsPath路径下）

读取的数据写入沙盒文件夹的目的：进行版本验证MD5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 易变数据写入沙盒文件夹 | 版本验证 | 不需要更新 | 进游戏 | 易变数据从沙盒文件夹读取 |
|  |  | 需要更新 | 获取服务器数据  二进制数据，写入沙盒文件路径 | 进游戏 |

|  |  |
| --- | --- |
| Notion.cs | void Start()  {  Debug.LogError(Application.dataPath); //输出的是Assets文件夹的路径  Debug.LogError(Application.streamingAssetsPath); //输出的是Assets/StreamingAssets文件夹的路径  Debug.LogError(Application.persistentDataPath); //不同的平台会输出不一样的路径，输出的是  } |

1. 应用示例

热更新 HotFix

* 资源的热更新

主要用AssetsBundle资源包实现资源的热更新，美术资源，config（配置文件）

* 逻辑的热更新

第一种，Ulua方法，把逻辑代码（C#）易变（可能会更改）逻辑的部分用lua脚本来写，主逻辑的部分使用C#，结合lua脚本实现代码逻辑的热更新

第二种，L#原理，主逻辑和易变逻辑分离，脚本的部分用L#语法（和C#比较接近），把易变逻辑（脚本）当成文本格式也是利用AssetsBundle实现热更新

游戏一开始设计框架的时候就要设计出哪些内容是易变的（是需要热更新），从一开始就规划

1. 示例

|  |  |
| --- | --- |
| ResourcesPath.cs | void Start()  {  TextAsset m\_text = Resources.Load("Jsondata") as TextAsset;  } |
| streamingAssetsPath.cs | using System.IO;    void Start()  {  ReadData(); //第一种  StartCoroutine("LoadWWW"); //第二种  }    void ReadData()  {  string fileAddress = System.IO.Path.Combine(Application.streamingAssetsPath, "Jsondata.txt" );  FileInfo m\_info = new FileInfo(fileAddress);  string str = "";  if(m\_info.Exists)  {  StreamReader strReader = new StreamReader(fileAddress);  str = strReader.ReadToEnd();  Debug.Log(str);    StreamReader strReader2 = m\_info.OpenText();  str = strReader2.ReadToEnd();  Debug.Log(str);  }  }    //在安卓平台下使用WWW类来加载数据（异步，使用协程）  string \_dataStr;  IEnumerator LoadWWW()  {  string strPath = Application.streamingAssetsPath + "Jsondata.txt"  Debug.Log(strPath);    if(!strPath.Contains("file://"))  {  strPath = "[file://](NULL)" + strPath; //必须要加  }  WWW www = new WWW(strPath);  yield return www;  while(!www.isDone) //www没有加载完的话  {    }  \_dataStr = [www.text](http://www.text);  Debug.Log(\_dataStr);  } |
| persistentDataPath.cs | using System.IO;    private string readOnly = "ReadOnly.db";  private string write = "write.db";    void Start()  {  string[] paths = new string[]  {  readOnly,  write  };  LoadBypaths(paths);  }    public void LoadBypaths(string[] paths)  {  StartCoroutine("CopyFile", paths);  }    private IEnumerator CopyFile(string[] paths)  {  int length = paths.Length;  for(int i = 0; i< length; i++)  {  string path = paths[i];  string streamPath = path.Combine(Application.streamingAssetsPath, path);  if(!streamPath.Contains("file://"))  {  streamPath = "[file://](NULL)" +streamPath;  }  WWW www = new WWW(streamPath);  yield return www;  while(!www.isDone)  {  //等待加载完成  }  //将读取的数据写入沙盒文件夹，以备后续逻辑调用  string perPath = Path.Combine(Application.persistentDataPath, path);  WriteBytes(www.bytes, perPath);  }  }    private void WriteBytes(byte[] bytes, string path)  {  if(!File.Exists(path))  {  FileStream stream = new FileStream(path, FileMode.Create);  stream.Write(bytes, 0, bytes.Length);  stream.Flush(); //清除该流的所有缓冲区  stream.Close(); //关闭当前流并释放与之关联的所有资源（如套接字和文件句柄）  }  } |

# 2016/10/30

一、Navmesh

Nav Mesh Agent组件

|  |  |
| --- | --- |
| screenposToworldpos.cs | NavMeshAgent cube;    void Start()  {  cube = GetComponent<NavMeshAgent>(); //获取到自身的NavMeshAgent组件  }    void Update()  {  Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition); //  RaycastHit hit; //射线命中点  Physics.Raycast(ray, out hit); //out关键字会导致参数通过引用来传递  cube.SetDestination(hit.point); //设置目标位置  }      //以下是out用法示例  void Func(int h, out int i)  {  i=44+h;  }  void m\_main()  {  int value;  Func(1, out value);  Debug.LogError("value = " + value);  } |
| 更新：增加鼠标点击才寻路 | void Update()  {  if(Input.GetMouseButtonDown(0)) //如果点击了鼠标左键  {  Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);  Raycast hit;  if(Physics.Raycast(ray, out hit))  {  cube.SetDestination(hit.point);  }  }  } |
| 更新:给一个目标，找目标物体 | public Transform target;    void Update()  {  cube.SetDestination(target.position);  } |

跳跃别忘了设置这里

Generated Off Mesh 
Drop Height 
-lump Distance 
Links 
10 
10 

一、Table管理

Table管理类用途

* 数据获取：读取本地配置文件，游戏运行时，把数据加载到内存中（字典或者集合）容器
* 保存数据：在管理类中开辟内存空间（字典或者集合）
* 提供一些供外部调用的，获取数据的函数方法（工具）

注意事项

* 在excel中另存为CSV文件格式，然后用记事本打开，另存为UTF-8编码格式

Resources->Table

|  |  |
| --- | --- |
| TableMgr.cs | using System.Collections.Generic;  using System;    private string[][] Array;    void Start()  {  //读取CSV文件  TextAsset binAsset = Resources.Load("Table/item\_shop") as TextAsset;  string[] temp\_lineArray = binAsset.text.Split("\r"[0]); //每一行的结尾都有一个\r换行符    //测试一下读取到的内容  for(int i = 0; i < temp\_lineArray.Length - 1; i++)  Debug.Log(temp\_lineArray[i]);    //二维数组  Array = new string[temp\_lineArray.Length - 1][];  for(int i = 0; i < temp\_lineArray.Length - 1; i++)  Array[i] = temp\_lineArray[i].Split(',');    string getStr = GetDataFunc(10001, "name");  Debug.Log(getStr);  }    //通过横向索引和纵向索引（ID-title）返回对应数据。判断ID的时候前面加一个“\n"， 这个时换行符。每一个ID前面前缀  string GetDataFunc(int m\_id, string strTittle)  {  if(Array.Length <= 0)  {  return "";  }    int n\_Row = Array.Length;  int n\_Col = Array[0].Length;  for( int i = 1; i< n\_Row; i++)  {  string strId = string.Format("\n{0}", m\_id); //**注意要加上“\n”**  if(Array[i][0] == strId)  {  for(int j = 0; j < n\_Col; j++)  {  if(Array[0][j] == strTittle)  {  return Array[i][j];  }  }  }  }    return "";  } |

二、多语言

在游戏框架设定的时候考虑解决方案，文字的设计的UI的匹配问题

eg：确定、取消按钮

解决方案：多语言表，语言Key（字符串） 中文 英文 韩语 德语 其它

|  |  |
| --- | --- |
| TableMgr.cs | //多语言处理相关  public Dictionary<int, multiLanguageStruct> muLanguageTableData = new Dictionary<int, multiLanguageStruct>();  public struct multiLanguageStruct  {  public int id;  public string chinese;  public string english;  public string russian;  public string japanese;  }  //根据表名读取数据  public void GetTable(string TableName)  {  string[][] Array;  string path = String.Format("{0}{1}", "Table/", TableName);  TextAsset binAsset = Resources.Load("Table/") as TextAsset;  string[] temp\_lineArray = binAsset.text.Split("\r"[0]);  string[] lineArray = new string[temp\_lineArray.Length-1];  for(int i = 0; i < temp\_lineArray.Length-1; i++)  {  lineArray[i] = temp\_lineArray[i];  }  Array = new string[lineArray.Length][];  for(int i = 0; i < lineArray.Length; i++)  {  Array[i] = lineArray[i].Split(',');  }  SaveTableData(Array);  }  //保存配置文件数据到内存（容器）  int m\_id;  string m\_chinese;  string m\_english;  string m\_russian;  string m\_japanese;  public void SaveTableData(string[][] m\_array)  {  if(m\_array.Length <= 0)  {  return;  }  int nRow = m\_array.Length;  int nCol = m\_array[0].Length;  for( int i = 1; i < nRow; i++) //处理行  {  multiLanguageStruct m\_struct = new multiLanguageStruct();  for(int j = 0; j < nCol; j++) //处理列  {  if(j == 0)  {  string tempID = m\_array[i][j];  tempID = tempID.Replace("\n", ""); //去掉了字符串里面的\n  m\_id = int.Parse(tempID);  }  if(j == 1)  {  m\_chinese = m\_array[i][j];  }  if(j == 2)  {  m\_english = m\_array[i][j];  }  if(j == 3)  {  m\_russian = m\_array[i][j];  }  if(j == 4)  {  m\_japanese = m\_array[i][j];  }  m\_struct.id = m\_id;  m\_struct.chinese = m\_chinese;  m\_struct.english = m\_english;  m\_struct.russian = m\_russian;  m\_struct.japanese = m\_japanese;  }  mulLanguageTableData.Add(m\_struct);  }  } |

一、倒计时效果

|  |  |
| --- | --- |
| timelose.cs | using System;    public UILabel labelShow;    int someSec = 11145;    void Start()  {  string time = SetGenTime(someSec);  Debug.Log("time====="+time);  time = FormatTimeStr(someSec);  Debug.Log("time====="+time);  InvokeRepeating("UpdateTime", 0, 1f); //1秒倒计时效果  }    //第一种显示方式  public string SetGenTime(int \_genTime)  {  DateTime datetime = DateTime.Parse("00:00:00").AddSeconds(\_genTime);  string strTime = GetTimeString(dateTime.Hour) + ":" + GetTimeString(datetime.Minute) +"" + GetTimeString(datetime.Second) +"";  return strTime;  }    string GetTimeString(int \_time)  {  string str;  if(\_time >=10)  {  str = \_time.ToString();  }  else  {  str="0"+\_time.ToString();  }  return str;  }    //显示方法2  public static string FormatTimeStr(int second)  {  return string.Format("{0}:{1}:{2}", (second/3600).ToString("D2"), ((second%3600)/60).ToString("D2"), ((second%3600)%60).ToString("D2"));  }    //倒计时  void UpdateTime()  {  string time = FormatTimeStr(someSec);  labelShow.text = time;  someSec--;  } |

# 2016/11/6

回顾

本地配置文件 CSV（用的比较多）

* 显示配置，资源名，资源图片等，不需要服务器验证
* 攻击力，价格等需要服务器验证

一、游戏制作相关

游戏框架

核心玩法

系统功能

网络模块

UI由美术做预制件，美术不会？

技术人员培训美术人员使用Unity当作工具

专门研发Unity的插件工具供给美术使用

二、英雄属性统计UI

|  |  |
| --- | --- |
| propertyItem.cs | public UILabel titleInfoLbl;  public UILabel valueLbl;    public struct showInfo  {  public string titleInfo;  public string value;  } |
| PropertyElement.cs | using System.Collections.Generic;  ……  public int cellHeight; //UIGrid的cellHeight  public int height\_ElemTopBg;  public int uiItemCount; //当前Element背景高度下显示适合的Item的数量  public propertyItem item;  public UISprite elementBg; //获取元素背景Sprite对象  public UIGrid itemParent;    public int itemCount;  public int nextElePosY;    //根据数据集合，创建Item  public void StartCreateItem(List<propertyItem.ShowInfo> infoGroup)  {  itemCount = infoGroup.Count;  GameObject tempItem;  for( int i = 0; i < InfoGroup.Count; i++)  {  tempItem = Instantiate( item.gameObject) as GameObject;  tempItem.transform.parent = itemParent.transform;  tempItem.transform.localScale = Vector3.one;  tempItem.transform.localPosition = Vector3.zero;  tempItem.GetComponent<propertyItem>().titleInfoLbl.text = infoGroup[i].titleInfo;  tempItem.GetComponent<propertyItem>().valueLbl.text = infoGroup[i].value;  }  itemParent.repositionNow = true; //UIGrid组件刷新，让重新排列  }    //设置背景高度  public void SetElementBgWidth()  {  int currentHeight = elementBg.height; //  int modelUI\_ItemWidth = cellHeight \* uiItemCount;  int m\_ItemWidth = cellHeight \* itemCount; //获取所需要的  elementBg.height = m\_ItemWidth + ( currentHeight - modelUI\_ItemWidth);  nextElePosY = elementBg.height + height\_ElemTopBg; //计算后面一个元素的Position.Y  } |
| UIMainWidget.cs  主控件 | public float FirstTab\_Y; //第一个属性模块的Y位置  public propertyElement m\_Eleme; //  public GameObject Eleme\_Parent; //非Grid    Dictionary<int , List<propertyItem.showInfo>> AllInfoGroup = new Dictionary<int, List<propertyItem.showInfo>>(); //存放数据的集合  List<propertyElement> ElemeGroup = new List<propertyElement>();    float CurrentY = 0;    public void StartCreatEleme<T>(List<T> elemeInfoGroup)  {  GameObject tempEleme;  for(int i = 0; i < elemeInfoGroup.Count; i++)  {  tempEleme = Instantiate(m\_Eleme.gameObject) as GameObject;  tempEleme.transform.parent = Eleme\_Parent.transform;  tempEleme.transform.localScale = Vector3.one;  tempEleme.transform.localPosition = Vector3.zero;  propertyElement m\_tempEleme = tempEleme.GetComponent<propertyElment>();  m\_tempEleme.StartCreateItem(AllInfoGroup[i]);  m\_tempEleme.SetElemeBgWidth();  ElemeGroup.Add(m\_tempEleme);  }  UpdateElemePos();  }    void UpdateElemePos()  {  for(int i = 0; i < ElemeGroup.Count; i++)  {  if( i == 0)  {  CurrentY = FirstTab\_Y;  }  else  {  CurrentY = CurrentY - ElemeGroup[i-1].NextElemePosY;  }  ElemeGroup[i].transform.localPosition = new Vector3(0f, CurrentY, 0f);  }  } |

一、（接上）英雄属性统计UI

# 2016/11/13

一、Protocol Buffer

1. 简介

目前Unity开发手游主流的数据通讯协议（前后端的报文交互的一种格式）

google提供的一个开源序列化框架，也叫googleBuf，类似XML、JSON等数据表示语言，最大的特点是基于二进制，高效，精简短小，有一定的语法格式

且提供了生成工具，通过生成工具可以将proto文件生成相应的类

可以支持的语言：java、C++、python（原生工具不支持C#）

通过将这些类包含在我们的项目中，可以轻松的调用相关方法，来完成业务消息（报文）序列化和反序列化

前端-后端流程：

proto->C#类->构造类对象（初始化成员）->序列化（二进制）->后端->二进制数据->反序列化->类对象（C++）

服务器端-客户端流程：

构造proto C++类->序列化为二进制数据->发送到网络层->客户端收到二进制数据->反序列化为C#代码使用

1. 语法

（1）定义消息message（关键字）

message c2s\_loginggame\_requeset

{

required string name = 1; //注意这里的1-2不是赋值，而是分配标示号

required string passw = 2;

}

每个字段有一个唯一的标识号，作用是在二进制中识别各个字段，一段开始使用无法再改变

最小的标识号可以从1开始，最大到229-1 or 536870911

不要使用从19000-19999标识号，因为proto协议实现中对这个号段进行了预留

（2）关键字

**required**——表示该字段必须要设置（赋值）

**optional**——该字段可以有0个或者1个值（不超过1个）该字段可以不赋值或者赋值

**repeated**——重复的值顺序会被保留，表示该值可以重复，类似C#中的List

可以增加注释

通过导入其他的.proto文件的定义来使用它们，引用

import"myproject/other\_protos.proto"; //import"路径"

message item

{

required uint64 ItemID = 1;

required uint32 ItemCount = 2;

optional string property = 3; //作为预留的字段，属性 解析：JSON 键-值 品质-值 是否有宝石镶嵌-1 其它属于键-值

}

message c2s\_GetmyItem\_request

{

//请求道具消息

}

message c2s\_GetmyItem\_Response

{

repeated item myItemGroup = 1; //相当于List<item> myItemGroup = new List<item>();

}

1. proto解析方式

（1）源代码：将proto源代码直接复制到项目中，目前用的比较多的方法

（2）反射机制（使用proto的动态链接库dll文件），但是这种方法如果要发布在IOS平台会有问题

1. 示例

|  |  |
| --- | --- |
| gaproto.proto | message ItemInfo  {  required unit64 itemID = 1; //道具ID  required uint32 itemCount = 2; //数量  optional string GAproperty = 3; //属性  }    message c2s\_GetmyItem\_request  {  required uint32 roleId = 1;  }    message s2c\_GetmyItem\_response  {  repeated ItemInfo gaItemGroup = 1;  } |

导入proto源码后报错unsafe解决方案：

Asset目录下新建文件：

|  |  |
| --- | --- |
| smcs.rsp | -unsafe |

|  |  |
| --- | --- |
| proto\_main.cs | using proto.gaproto;  using ProtoBuf;  using System.IO;    void SendMessage()  {  c2s\_GetmyItem\_request send = new c2s\_GetmyItem\_request();  send.roleId = 1100; //角色ID  //将构造的send对象序列化为二进制  //将构造的send对象通过网络层发送方法发送出去    byte[] m\_result = Serialize(send);    //m\_result网络发送数据  }    //序列化proto数据（将数据转为二进制）  public static byte[] Serialize(IExtensible msg)  {  byte[] result;  using(var stream = new MemoryStream())  {  Serializer.Serialize(stream, msg);  result = stream.ToArray();  }  return result;  }    //反序列化  public static T DeSerialize<T>(byte[] msg)  {  T result;  using (var stream = new MemoryStream(msg))  {  result = Serializer.Deserialize<T>(stream);  }  return result;  }    void TestRrs(byte[] result) //数据包 包头-包体-包尾（或许有）  {  switch("消息ID")  {  case 1101:  s2c\_GetmyItem\_response m\_result = DeSerialize<s2c\_GetmyItem\_response>(result);  GlobalMgr.Instance.ItemMgr.myItemGroup.AddRange(m\_result.gaItemGroup);  //通知界面，显示背包 //委托 消息系统  break;  }    }    void CreatBagUI()  {  foreach(var data in GlobalMgr.Instance.ItemMgr.myItemGroup)  {  GameObject tempItem = Instantiate();  }  } |

|  |  |
| --- | --- |
| ItemMgr.cs | using proto.gaproto;  using ProtoBuf;    public List<ItemInfo> myItemGroup = new List<ItemInfo>(); |

上午的protobuf的扫尾

一、NGUI-Tween