有返回值的可以外部传参的Lambda表达式写法

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Cavalry : MonoBehaviour {

    void Start () {
        LambdaFun("哈哈", s =>
                              {
                                  if (s.Equals("哈哈"))
                                      return "game";
                                  else
                                      return "over";
                              });
    }

    public void LambdaFun(string str,Func<string,string> func)
    {
        Debug.Log(func(str));

    }
}

```

使用Comparsion泛型委托进行自定义排序

```
namespace System
{
    public delegate int Comparison<T>(T x, T y);
}
```

```
using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using System;

public class MyClass {}
public class Text : MonoBehaviour {

    List<MyClass> mcs = new List<MyClass>();
    void Start () {
        mcs.Sort(Sort);//Sort方法里传递实现排序的方法(委托)

        //Lambda表达式的写法
        mcs.Sort(
                (MyClass m1, MyClass m2) =>
                {
                    return -1;
                }
                );
    }

    //直接实现排序方法 和Comparsion委托类型保持一致 实现排序功能
    int Sort(MyClass m1,MyClass m2)
    {
        return 1;
    }

}

```

Attribute 特性

将元数据或声明信息与代码（程序集 类型 方法 属性）相关联

简单理解 特性就是在类的名称 属性 方法等添加一个标记

从而使得这些类属性和方法具有某些统一的特征

元数据

编译器会编译源文件和分析源文件

当检查通过后 会生成程序集 程序集是由多个模块组成

而模块中主要一个信息 就是元数据

元数据：描述源代码中定义的类型和成员 以及 源代码中引用的类型和成员

Attribute特性:

1 特性可以向程序中添加元数据信息

2 特性可以把一个或多个特性应用到整个程序集 部分模块或者更小的程序元素(类和属性)

3 特性可以像方法和属性一样接受参数

4 特性可以使用反射(reflection)检查自己的元数据或其他程序内的元数据

.NET框架提供的两种类型的特性

1 预定义特性

Conditional　Obsolete　　AttributeUsage(配合自定义特性使用)

2 自定义特性

Conditional 预定义特性

此特性会引起方法的条件编译

使用Conditional是封闭#if和#endif内部方法的替代方法 更整洁 更别致 减少出错机会

```
#define WW//文件头 添加预编译指令符号
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using System.Diagnostics;//Conditional命名空间


public class Class1
{
    [Conditional("WWW")]//Conditional 依赖指令的预处理器符号
    public static void DoLog(string msg)
    {
        UnityEngine.Debug.Log(msg);
    }
```

```
16
17      [Conditional("WWW")]//Conditional 依赖指令的预处理器符号
18      public static void DoLog2(string msg)
19      {
20          UnityEngine.Debug.Log(msg);
21      }
22  }
23
24  public class Cavalry : MonoBehaviour {
25
26      void Start () {
27          Class1.DoLog("helloWorld");
28      }
29  }
30
```

Obsolete 预定义特性

过时的 标记不应使用的程序实体

当一个新的方法存在后 还想要保留旧方法 可以添加obsolete特性到旧方法上

Unity中对obsolete特性的使用

```
// 摘要:
//     The Renderer attached to this GameObject. (Null if there is none attached).
[Obsolete("Property renderer has been deprecated. Use GetComponent<Renderer>() instea
public Component renderer { get; }
//
// 摘要:
//     The Rigidbody attached to this GameObject. (Null if there is none attached).
[Obsolete("Property rigidbody has been deprecated. Use GetComponent<Rigidbody>() inst
public Component rigidbody { get; }
//
// 摘要:
//     The Rigidbody2D that is attached to the Component's GameObject.
[Obsolete("Property rigidbody2D has been deprecated. Use GetComponent<Rigidbody2D>()
public Component rigidbody2D { get; }
public string tag { get; set; }
public Transform transform { get; }
```

```
1  using System.Collections;
```

```
using System.Collections.Generic;
using UnityEngine;
using System;


public class Class1
{
    [Obsolete]
    public void Fun1() {}

    [Obsolete("过时了 别用了 自己写函数去吧 或者调用Fun3")]
    public void Fun2(){}
    [Obsolete("Fun3不能用 GameOver",true)]
    public void Fun3(){}
}

public class Cavalry : MonoBehaviour {

    void Start () {

        Class1 c1 = new Class1();
        c1.Fun1();//警告
        c1.Fun2();//警告
        c1.Fun3();//错误
    }
}

```

自定义特性

开发者可以通过创建自定义特性 来存储声明信息

所有的自定义特性都必须继承于 Attribute

声明自定义特性必须使用到AttributeUsage

AttributeUsage 有三个属性 在定义时把它放在定制属性前面

第一个属性：

ValidOn

通过这个属性 可以定义特性可以出现在何种程序实体之前

通过|可以把若干个AttributeTargets连接起来

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

[AttributeUsage(AttributeTargets.Class
    |AttributeTargets.Struct|
    AttributeTargets.Property)]
public class MyAttribute : Attribute
{
    public string desc;

    //任何自定义特性至少包含一个字符串参数类型的构造函数
    public MyAttribute(string desc)
    {
        this.desc = desc;
    }
}

[MyAttribute("888")]
public class Class1
{
    [MyAttribute("888")]//出错
    private int a;
    [MyAttribute("888")]
    public int A
    {
        get { return a; }
        set { a = value; }
    }
}

[MyAttribute("888")]
public struct MyClass2
{
}

```

AttributeTargets.Assembly 可以对整个程序集添加特性

Module 可以对模块应用特性

Class 可以对类应用特性

Struct 可以对结构应用特性

Constructor 可以对构造器应用特性

Method 可以对方法应用特性

Property 可以对属性应用特性

Field 可以对字段应用特性

Event 可以对事件应用特性

Interface 可以对接口应用特性

Parameter 可以对参数应用特性

Delegate 可以对委托应用特性

ReturnValue 可以对返回值应用特性

GenericParamet 可以对泛型参数应用特性

All 可以以上任何引用程序实体添加特性

第二个属性：

AllowMultiple

这个属性标记我们的定制特性是否能被重复的放在同一个程序实体前多次

默认情况下 一个特性只能作用于一个程序实体1次 添加AllowMultiple可以多次添加

```
[AttributeUsage(AttributeTargets.All,AllowMultiple=true)]
public class MyAttribute : Attribute
{
    public string desc;

    //任何自定义特性至少包含一个字符串参数类型的构造函数
    public MyAttribute(string desc)
    {
        this.desc = desc;
```

```
    }
}
[MyAttribute("888")]
[MyAttribute("888")]
[MyAttribute("888")]
[MyAttribute("888")]
public class Class1
{

}

```

第三个属性：

Inherited

当特性被放在一个基类上 它是否能被派生 被继承

```

using System;

[AttributeUsage(AttributeTargets.All,AllowMultiple=true,Inherited=true
)]
public class MyAttribute : Attribute
{
    public string desc;

    //任何自定义特性至少包含一个字符串参数类型的构造函数
    public MyAttribute(string desc)
    {
        this.desc = desc;
    }
}

[MyAttribute("888")]
public class Class1
{

}

//当添加了Inherited 子类具备和父类相同的特性
```

```
public class MyClass2:Class1
{
}
```

项目中Debug标记特性

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

[AttributeUsage(AttributeTargets.All,AllowMultiple=true,Inherited=true
)]
public class DeBugInfo : Attribute
{
    public int bugNum;
    public string developer;
    public string lastView;

    public string message;
    public string Message
    {
        get { return message; }
        set { message = value; }
    }

    public DeBugInfo(int bg,string dev,string view)
    {
        this.bugNum = bg;
        this.developer = dev;
        this.lastView = view;
    }
}


[DeBugInfo(45, "Jim", "2018-6-1",Message="I quit")]
[DeBugInfo(5, "Tom", "2018-2-14",Message = "这哥们离职了")]
[DeBugInfo(1, "JACK", "2018-1-12", Message = "这哥们离职了")]
public class Student
```

```
{
    protected int age;
    protected string name;
    public Student(int l, string w)
    {
        age = l;
        name = w;
    }

    [DeBugInfo(2,"Timy","2018-5-26",Message="returnValue is Null")]
    public string GetInfo()
    {
        return name +":"+ age.ToString();
    }
}


public class Cavalry : MonoBehaviour {
    void Start () {
    }
}

```

Unity自定义的特性

```
using System;

public class Text : MonoBehaviour {

    [Tooltip("这是一个int值")]//弹出框特性
    public int num;

    [Space(30)]//和上一个变量的显示位置增加空隙
    public string name;

    [Header("这是一个年龄值")]//在Inspector窗口增加描述
    public int age;

    [Range(0,100)]//限制数值的取值范围 同时会增加一个滑动条
```

```
    public float angle;

    [SerializeField]//会被执行序列化的字段 同时会显示在Inspector窗口
    private float x;

    [NonSerialized]//不会被执行序列化 且不会显示在inspector窗口
    public float y;

    [HideInInspector]//隐藏在Inspector窗口的显示
    public int data;

    [MultilineAttribute]//多行显示的字符串
    public string text;

    [TextAreaAttribute]//把字符串在Inspector窗口编辑区变成一个TextArea
    public string text2;

    [ContextMenuItem("Function","Fun")]
    public int clickMe;
    public void Fun()
    {
        Debug.Log(1000);
    }

    public MyClass1 c1;
}
[Serializable]//使自定义类型 可以进行序列化
public class MyClass1
{
    public int x;
    public int y;
}
```

反射 Reflection

反射的定义：审查元数据并收集关于它的类型信息的能力

反射的用途：

1 它允许在运行时查看特性(Attribute)

2 它允许审查集合中的各种类型 以及实例化这些类型

3 它允许延迟绑定的方法和属性

4 它允许在运行时创建新类型 然后使用这些类型执行任务

反射和解析元数据需要用的类型：

System.Reflection.Assembly

定义和加载程序集 在 程序集中列出模块 以及从程序集中查找类型和创建类型实例

EventInfo

获取事件信息

FieldInfo

获取字段信息（名称 访问修饰符 和实现详细信息）

ConstructorInfo

获取构造器 的名称 参数 访问修饰符 和 实现详细信息

MethodInfo

获取方法 的名称 参数 访问修饰符 和 实现详细信息

PropertyInfo

获取属性 信息

MemberInfo

获取特性

System.Type

获取类的类型

通过反射获取系统string的类型信息

```
using System.Reflection;//反射的命名空间
using System;


public class Text : MonoBehaviour {

```

```
    void Start()
    {
        string n = "HelloWorld";
        Type t= n.GetType();

        foreach (MemberInfo member in t.GetMembers())
        {
            Debug.Log(member.MemberType + ":" + member.Name);
        }

    }


```

反射得到类型的所有构造函数

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using System.Reflection;//反射的命名空间
using System;


public class ClassA
{
    public ClassA() {}
    public ClassA(int x) { }
    public ClassA(string s,float y) { }
}

public class Text : MonoBehaviour {
    void Start()
    {
        Type t = typeof(ClassA);

        //获取一个类中所有的构造器（构造函数）
        ConstructorInfo[] infos= t.GetConstructors();

```

```
        foreach (ConstructorInfo info in infos)
        {
            Debug.Log(info.Name);
            //打印构造方法中的参数列表
            foreach (ParameterInfo p in info.GetParameters())
            {
                Debug.Log(p.Name + ":" + p.ParameterType);
            }
            Debug.Log("");
        }
    }
}
```

反射得到构造器 同时构造器或者Activator 动态实例化对象

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
using System.Reflection;//反射的命名空间
using System;

public class ClassA
{
    public float x;
    public string s;
    public ClassA() {}
    public ClassA(int x) { }
    public ClassA(string s,float y)
    {
        this.s = s;
        this.x = y;
    }

    public void Show()
    {
        Debug.Log(x + ":" + s);
    }
}

```

```
public class Text : MonoBehaviour {
    void Start()
    {
        Type t = typeof(ClassA);
        Type[] pt = new Type[2];
        pt[0] = typeof(string);
        pt[1] = typeof(float);
        //通过构造器的参数列表类型获取对应的构造器
        ConstructorInfo ci=t.GetConstructor(pt);
        object[] obj = new object[] { "helloWorld", 2.0f };
        //使用构造器动态实例化对象
        object o= ci.Invoke(obj);
        ((ClassA)o).Show();

        //通过Acitvator的CreateInstance方法动态实例化对象
        object o2=Activator.CreateInstance(t, "你好中国", 4.0f);
        ((ClassA)o2).Show();
    }
}
```