使用反射得到类的属性 方法和字段

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

//自定义类型
public class ClassA
{
    public float x;
    public string s;
    private int a;
    public int A
    {
        get { return a; }
    }

    public void Show()
    {
        Debug.Log(s);
    }
}

public class Text : MonoBehaviour {

    void Start()
    {
        ClassA nc = new ClassA();
        Type t = nc.GetType();
        //通过反射得到类型的属性
        PropertyInfo[] infos=t.GetProperties();
        foreach (PropertyInfo pi in infos)
        {
            Debug.Log(pi.Name);
        }

        //通过反射获取到类型的方法信息
        MethodInfo[] mis= t.GetMethods();
```

```
        foreach (MethodInfo mi in mis)
        {
            Debug.Log(mi.Name +":"+ mi.ReturnType);
        }

        //通过反射获取类型的字段信息
        FieldInfo[] fis= t.GetFields();
        foreach (FieldInfo fi in fis)
        {
            Debug.Log(fi.Name);
        }

    }
}
```

用反射生成对象 并调用属性 方法和字段进行操作

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

public class ClassA
{
    public float x;
    public string s;
    private int a;
    public int A
    {
        get { return a; }
        set { a = value; }
    }

    public void Show()
    {
        Debug.Log(s);
        Debug.Log(x);
        Debug.Log(a);
```

```
    }

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

public class Text : MonoBehaviour {

    void Start()
    {
        ClassA a = new ClassA();
        Type t = a.GetType();

        //通过Activator动态创建对象
        object obj= Activator.CreateInstance(t);

        //获得x字段 并且给x字段进行赋值
        FieldInfo fi=t.GetField("x");
        fi.SetValue(obj, 1000);
        //获得s字段 并且给s字段进行赋值
        FieldInfo fi2 = t.GetField("s");
        fi2.SetValue(obj,"再见");

        //获得A属性 并给A属性进行赋值 注意保证A属性可以Set
        PropertyInfo pi=t.GetProperty("A");
        pi.SetValue(obj, 500, null);

        //获得成员方法 调用成员方法 打印成员信息
        MethodInfo mi=t.GetMethod("Show");
        mi.Invoke(obj,null);

    }
}
```

反射查看特性（Attribute）信息

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

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute:Attribute
{
    public string url;
    private string title;
    public string Title
    {
        get { return title;}
        set { title = value;}
    }

    public HelpAttribute(string url)
    {
        this.url = url;
    }
}


[HelpAttribute("MyClass1 Infomation",Title="game")]
public class MyClass1
{
    [HelpAttribute("Fun Infomation",Title="xxx")]
    public void Fun()
    {

    }
}

public class Text : MonoBehaviour {

    void Start()
    {
        System.Type t = typeof(MyClass1);

        object[] attributes=t.GetCustomAttributes(true);

```

```
43
44          for (int i = 0; i < attributes.Length; i++)
45          {
46              HelpAttribute a = attributes[i] as HelpAttribute;
47              Debug.Log(string.Format("url:{0}", a.url));
48              Debug.Log(string.Format("title:{0}", a.Title));
49          }
50
51          foreach (MethodInfo function in t.GetMethods())
52          {
53              foreach (Attribute a in
function.GetCustomAttributes(true))
54              {
55                  HelpAttribute info = (HelpAttribute)a;
56                  if (info != null)
57                  {
58                      Debug.Log(string.Format("url:{0},title:
{1},forMethod:{2}",
59                          info.url, info.Title, function.Name));
60                  }
61              }
62          }
63      }
64  }
```

System.Reflection.Assembly类

Assembly类可以获得程序集的信息

也可以动态加载程序集 以及在程序集中查找类型信息 并创建该类型的实例

使用Assembly类可以降低程序之间的耦合 有利于软件结构的合理化

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System;
5  using System.Reflection;
6  using MyDll;
7
8  public class Text : MonoBehaviour {
9
```

```
    void Start()
    {
        //加载自定义程序集 MyDll
        Assembly ass = Assembly.Load("MyDll");
        //加载程序集后通过类名获取对应的类型 必须是类的全名
        Type t = ass.GetType("MyDll.Class1");
        object o=Activator.CreateInstance(t);
        MethodInfo mi = t.GetMethod("Add");
        Debug.Log(mi.Invoke(o, new object[] { 100, 300 }));

        //通过程序集的具体路径 加载
        Assembly ass2= Assembly.LoadFrom(Application.dataPath +
"/MyDll.dll");
        Type[] aa = ass2.GetTypes();
        foreach (Type type in aa)
        {
           object obj= Activator.CreateInstance(type);
           Debug.Log(obj);
        }
    }
}
```

Action委托和Func委托

使用委托两步走

1 定义委托

2 创建和使用委托

.NET已经提供好了定义的委托 可以直接使用

Func和Action 两个系统委托的比较

Action

1 Action是无返回值的泛型委托

2 Action表示无参的 无返回值的委托

3 Action<int,string> 表示传入参数int string 无返回值的委托

Func

1 Func是有返回值的泛型委托

2 Func<int> 表示无参 返回值为int的委托

3 Func<object,string,int> 表示参数类型为object,string 返回值类型为int的委托

例子1：

Action无参的委托

```
using System;
public class Text : MonoBehaviour {

    void Start()
    {
        Action action = Fn;
        action();
    }

    void Fn()
    {
        Debug.Log("game");
    }
}
```

例子2

Action<T>

```
public class Text : MonoBehaviour {

    void Start()
    {
        Action<int> action = Fn;
        action(50);
    }

    void Fn(int a)
    {
        Debug.Log(a);
```

```
    }
}
```

例子3

Action<T1,T2>

```
public class Text : MonoBehaviour {

    void Start()
    {
        Action<int,string> action = Fn;
        action(50,"hel");
    }

    void Fn(int a,string text)
    {
        Debug.Log(a);
    }
}
```

例子4

Action的Lambda表达式

```
public class Text : MonoBehaviour {

    void Start()
    {
        Action<string[]> action = (string[] x) =>
                                  {
                                      Debug.Log(x[2]);
                                  };
        action(new string[] { "www", "wwz", "xxx" });
    }

    void Fn(int a,string text)
    {
        Debug.Log(a);
```

```
    }
}
```

例子5

Action作为参数使用

```
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        Fn(100, x => { Debug.Log(x); });
    }

    void Fn(int a,Action<int> action)
    {
        action(a);
    }

}
```

例子6

Func<TResult>

```
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        Func<int> func = Fn;
        int a = func();
    }

    int Fn()
    {
```

```
        return 10;
    }
}
```

例子7

Func<T,TResult>

```
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        Func<string,int> func = Fn;

    }

    int Fn(string s)
    {
        return 10;
    }
}
```

例子8

Func作为参数使用

```
using System;

public class Text : MonoBehaviour {

    void Start()
    {
        Fn("game", w => {
            if (w.Equals("start"))
                return 100;
            else
                return 200;
```

```
            });
    }

    int Fn(string s,Func<string,int> func)
    {
        return func(s);
    }
}
```

C# 异常处理

异常是在程序执行期间出现的问题

C#的异常是对程序运行时出现的特殊情况的一种响应 比如数据越界 尝试除以0

异常处理关键词：

try: 一个try语句块 标识了一个将被执行的代码 其可能会出现异常情况 后跟一个或多个catch语句块

catch： 程序通过异常处理捕获异常

： finally语句块 用于执行固定的代码语句 不管是否出现异常 finally语句块一定会执行

throw： 当问题出现时 程序抛出一个异常

```
try{
    //可能会引起异常的代码
}catch(Exception e1){
    //错误处理代码
}
catch(Exception e2){
    //错误处理代码
}
......
finally{
   //最终一定会执行的语句
```

```
}
```

因为0是不能被当做除数的 所以在除法运算里 当除数是一个变量时

就需要考虑到变量等于0的异常情况

```
public class Text : MonoBehaviour {

    void Start()
    {
        int x=100;
        int y=0;
        int a=0;

        try
        {
            a = x / y;
        }
        catch (System.DivideByZeroException e)
        {
            Debug.Log(e);
        }
        finally
        {
            Debug.Log(a);
        }
    }
}
```

常用异常处理的类

System.Exception：所有异常类的根父类

两个子类：

1 SystemException 预定义的系统异常的基类

2 ApplicationException 应用程序生成的异常 自定义异常也需要继承此类

预定义异常类

System.IO.IOException 处理IO 文件流的错误

System.IndexOutOfRangeException 处理当方法指向超出范围的数组索引时产生的错误

System.NullReferenceException 处理当使用一个空对象时产生的错误

System.DivideByZeroException 尝试除以0生成的错误

System.InvalidCastException 类型转换时引发的错误

System.OutOfMemoryException 处理内存不足时产生的错误

System.StackOverflowException 栈溢出生成的错误