# Unity3D开发之基于委托的消息框架开发

我们在开发项目的时候会发现，我们经常要在不同模块不同类之间调用函数。比如：我们在点击场景一个物体后，就需要对UI界面做一定的变化。通常最简单的方式是我们直接在面板绑定持有对更改界面的函数的对象。可是如果项目比较大，类比较多的情况下，我们会发现项目变得异常混乱。更改一点需求可能会牵一发而动全身。其实这就是代码的耦合性太高。

一.静态委托事件

如果项目很小，只有几处跨模块调用代码，我们通常会在方法所在的类内声明一个静态委托事件，具体形式以函数形式为准。然后我们绑定委托后，就可以直接调用该委托。代码如下：

//无参无返回值

public delegate void MyDelegate();

public static MyDelegate myDelegate;

private void Start(){

myDelegate += Test1;

}

void Test1() {

Debug.Log("Run");

}

当我们在外部需要调用此函数时，直接调用该类.myDelegate()就可以了。但是，当我们项目中有很多模块之间交互，这时候我们就不能一个个声明了，假如一百个调用我们难不成要声明一百个委托？这时候我们就需要将我们的委托事件收集起来并且根据键值来调用对应的委托。

二.基于消息的委托机制

我们下面要做的处理其实就是省略了对委托命名的过程并且将所有委托事件集中管理起来。通过字典的键值来寻找对应事件。首先，我们确定，基于消息机制的默认设定是消息发送是单向的，不存在返回值。所以我们就可以省略了有返回值的委托声明。然后，对于函数，一般分为有参和无参。所以，首先我们需要声明两个形式的委托。

//有参数

public delegate void NotificationDelegate(EventArgs eventArgs);

//无参数

public delegate void NotificationNoArgDelegate();

然后我们就在一个单例类里声明两个字典，分别存储有参和无参的委托事件。键值是需要我们在外部自己设定类型，一般我们都使用枚举类型，直观明了。代码如下：

public class MessageCenter : MonoBehaviour {

public static MessageCenter Instance = null;

private Dictionary<EventId, NotificationDelegate> messageListeners = new Dictionary<EventId, NotificationDelegate>();//有参数

private Dictionary<EventId, NotificationNoArgDelegate> messageNoArgsListeners = new Dictionary<EventId, NotificationNoArgDelegate>();//无参数

private void Awake(){

Instance = this;

}

/// <summary>注册 有参数函数</summary>

/// <param name="id">事件id</param>

/// <param name="notificationDelegate">有参数事件委托</param>

public void RegisterEvent(EventId id, NotificationDelegate notificationDelegate) {

if (!messageListeners.ContainsKey(id)) messageListeners.Add(id,notificationDelegate);

else{

NotificationDelegate notification = messageListeners[id];

notification += notificationDelegate;

}

}

/// <summary>注册委托 无参数函数</summary>

/// <param name="id">事件id</param>

/// <param name="notificationNoArgsDelegate">无参数事件委托</param>

public void RegisterEvent(EventId id, NotificationNoArgDelegate notificationNoArgsDelegate) {

if (!messageNoArgsListeners.ContainsKey(id)) messageNoArgsListeners.Add(id, notificationNoArgsDelegate);

else {

NotificationNoArgDelegate notification = messageNoArgsListeners[id];

notification += notificationNoArgsDelegate;

}

}

/// <summary>移除监听</summary>

/// <param name="id">事件id</param>

/// <param name="notificationDelegate">有参数事件委托</param>

public void RemoveEvent(EventId id,NotificationDelegate notificationDelegate){

if (messageListeners.ContainsKey(id)){

NotificationDelegate notification = messageListeners[id];

notification -= notificationDelegate;

if (notification == null) messageListeners.Remove(id);

}

}

/// <summary>移除监听</summary>

/// <param name="id">事件id</param>

/// <param name="notificationDelegate">无参数委托事件</param>

public void RemoveEvent(EventId id, NotificationNoArgDelegate notificationNoArgsDelegate) {

if (messageListeners.ContainsKey(id)) {

NotificationNoArgDelegate notification = messageNoArgsListeners[id];

notification -= notificationNoArgsDelegate;

if (notification == null) messageNoArgsListeners.Remove(id);

}

}

/// <summary>消息事件分发</summary>

/// <param name="id">事件ID</param>

/// <param name="eventArgs">事件参数</param>

public void DispatchMessage(EventId id, EventArgs eventArgs){

if (!messageListeners.ContainsKey(id))

return;

messageListeners[id](eventArgs);

}

/// <summary>消息事件分发 无参数</summary>

/// <param name="id">事件id</param>

public void DispatchMessage(EventId id){

if (!messageNoArgsListeners.ContainsKey(id))

return;

messageNoArgsListeners[id]();

}

}

上面是实现了我们核心的消息中心类。下面我们就可以在项目里应用他了。比如我们的属性UI面板里有一个函数：

private void Start(){

//注册

MessageCenter.Instance.RegisterEvent(EventId.SelectGameObject, Test1);

MessageCenter.Instance.RegisterEvent(EventId.SelectGameObject, Test2);

}

void Test1(){

Debug.Log("选择了一个物体");

}

void Test2(EventArgs args){

GameObject go = (args as GameObjectArgs).go;

Debug.Log("选择了一个物体"+go.name);

}

而对于消息参数类型，我们选择统一继承自Systerm下的EventArgs，如果我们要传一个Gameobject，我们可以自己写一个参数类，如下：

public class GameObjectArgs:EventArgs{

public GameObject go;

public GameObjectArgs(GameObject go){

this.go = go;

}

}

相比之前我写的万能应用框架传参来说，虽然写的步骤能稍微多一些，但是减除了拆装箱操作，也是提升了很多整体的性能。我们注册上方法后，调用就直接使用DispatchMessage方法。

//调用

void Test3(){

MessageCenter.Instance.DispatchMessage(EventId.SelectGameObject);

MessageCenter.Instance.DispatchMessage(EventId.SelectGameObject,new GameObjectArgs(new GameObject("试验")) as EventArgs);

}

结尾：其实这篇文章是大上个月和我朋友讨论我那个万能应用框架时反思出来的。他说我那个使用反射机制是可以将代码简单归一化，但是缺点就是反射效率低，传参也是有拆装箱。他建议我使用基于委托的消息机制。回来我就不断想这个事情。当时写出来的时候我也是有些不愿意用，因为我要不断声明事件的键值，也就是添加枚举成员，还要为不同的参数写不同形式的类，我觉得也很麻烦。后来项目中使用StrangeIoc后，我发现这不就是StrangeIoc的简版吗？只不过没有对model view controller三个模块划分。于是我更加坚信我这个小框架的可使用性。如果有什么不正确的地方，欢迎指点。

---------------------

原文：https://blog.csdn.net/qq\_33994566/article/details/81870742