# **游戏架构之模块间通信(消息机制)**

作者：[萧然](http://gad.qq.com/user/index?id=2248441)

链接：<http://gad.qq.com/article/detail/287237>

一、

先谈一谈个人对游戏框架的一点理解，顾名思义，框架是一个项目的骨架，如同大树的主干，搭建框架，在此基础上再加入各个功能模块，构成有一个完整的项目。如同一棵树有一个健壮的主干，再从主干上生长出一个一个的分支，最终长成一颗枝繁叶茂的大树。此外，框架会设定好模块的基本格式，更加有利于功能的模块化；框架还负责各个模块之间的交互，每个模块作为一个独立的个体，内部是独立运行的，如果模块间需要进行一些交互，则需要通过框架来实现，避免模块间直接通信，最终模块关系错综复杂，难以维护。

二、

模块间的数据交互、信息传递是框架中比较重要的一部分，最近根据做过的几个项目和一些资料，编写了一套简单的模块间信息传递机制，在此之前也发过几篇关于模块封装的博文，组装到一起，应该也是可以用了。

关于消息机制(消息/广播/通知···有多个叫法，不过实现的功能都是类似的)，大概原理是：

1. 消息：由唯一消息ID、消息体组成（有的写法也会将消息ID分离出消息体，不包含在消息体内，这样方便消息转发都多个不同模块，但不便于管理）。消息ID，用int值表示，根据需求划分一定数量的ID给每个模块，模块内部单独管理；消息体：数据信息的载体，一般是一个子类，这样方便不同模块自定义数据格式。

要注意一点，跨模块消息，A模块需要B模块的数据，就需要注册B模块的消息，这样B在发送消息之后，只要注册过这条消息的模块，都会接收到消息，这也要求模块内定义ID后，不能随意变动ID，建议采用枚举表示，使用时将枚举转为Int。

2. 建立消息中心，保存所有的消息及对应接收回调函数，各模块通过管理者将消息注册到消息中心，有对外的发消息接口，供各模块调用，当然同样要有注销接口。在收到消息之后执行对应的回调函数，将参数传递到注册过消息的多个具体模块，模块内部自行处理。

3. 各个模块管理者，在脚本运行开始，注册所需要的消息，在脚本待销毁的时候注销，提供一个消息接收回调，消息中心会将消息下发到回调，然后内部处理消息。

4. 关于消息中心保存记录消息，我用的字典Dictionary<int，委托>保存对应的ID和回调，利用委托的一个优点就是委托的“+=”和“-=”，比如有多个模块注册了同一个消息，可以将callback+=newCallback，这样来把所有的回调记录下来，在注销时减掉。

但委托减法具有不可预测的结果，虽然改成Event事件可以避免程序报错，但结果与委托一样也会有这种问题，为了避免出现问题，在使用减法时，每次只减掉一个元素(即 a-= b，不要a-=(b+c)  )，就不会发生意外了，可以忽略代码里的警告了

官方文档解释  [Code Inspection: Delegate subtractions](http://www.jetbrains.com/help/resharper/2018.2/DelegateSubtraction.html)

http://www.jetbrains.com/help/resharper/2018.2/DelegateSubtraction.html

Demo代码如下，写的比较简单，实际项目需要再完善~~

//消息中心主要负责注册、注销消息，发送消息到对应模块的回调

using System;

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public delegate void CallbackDele(Msg msg);

//消息体 父类

public class Msg

｛

public int msgId ｛ get; protected set; ｝

public object sender ｛ get; protected set; ｝

｝

//消息中心

public class NotifyManager : MonoBehaviour

｛

//单例

static NotifyManager instance;

public static NotifyManager Instance

｛

get

｛

if (instance == null)

｛

GameObject newObj = new GameObject("NotifyCenter");

instance = newObj.AddComponent<NotifyManager>();

DontDestroyOnLoad(newObj);

｝

return instance;

｝

｝

//记录已注册消息

Dictionary<int, CallbackDele> callbackDic = new Dictionary<int, CallbackDele>();

//记录待处理事件

Queue<Action> todoCallback = new Queue<Action>();

//注册消息

public bool Attach(CallbackDele callback, int[] msgIds)

｛

if (callback == null)

return false;

for (int i = 0; i < msgIds.Length; i++)

｛

Attach(callback, msgIds[i]);

｝

return true;

｝

public bool Attach(CallbackDele callback, int msgId)

｛

if (callback == null)

return false;

if (!callbackDic.ContainsKey(msgId))

｛

callbackDic.Add(msgId, callback);

｝

else

｛

callbackDic[msgId] += callback;

｝

return true;

｝

//注销消息

public bool Detach(CallbackDele callback, int[] msgIds)

｛

if (callback == null)

return false;

for (int i = 0; i < msgIds.Length; i++)

｛

Detach(callback, msgIds[i]);

｝

return true;

｝

public bool Detach(CallbackDele callback, int msgId)

｛

if (!callbackDic.ContainsKey(msgId) || callback == null)

return false;

//委托减法具有不可预测的结果:官方文档解释

//http://www.jetbrains.com/help/resharper/2018.2/DelegateSubtraction.html

//每次减掉一个委托，不会发生意外，可忽略该警告

callbackDic[msgId] -= callback;

if (callbackDic[msgId] == null)

callbackDic.Remove(msgId);

return true;

｝

//通知/广播/分发消息

public bool PostMsg(Msg msg = null)

｛

if (msg.msgId > 0 && callbackDic.ContainsKey(msg.msgId) && null != callbackDic[msg.msgId])

｛

//加入队列

lock (todoCallback)

｛

todoCallback.Enqueue(() => callbackDic[msg.msgId](msg));

｝

return true;

｝

return false;

｝

//刷新待处理消息事件

void Update()

｛

if (todoCallback.Count == 0)

return;

lock (todoCallback)

｛

while (todoCallback.Count > 0)

｛

todoCallback.Dequeue()();

｝

todoCallback.Clear();

｝

｝

｝

//每个消息对应唯一ID，每个模块分配一定数量的ID，定义模块的起始ID

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class MsgIdSetting

｛

public const int mgrIdSpan = 100;

｝

public enum MgrId

｛

//分模块划分消息ID，定义Id起点个长度，每个模块单独管理自己的Id

//0~~99

demo01MgrId = 0,

//100~~199

demo02MgrId = MsgIdSetting.mgrIdSpan \* 1,

//200~~299

demo03MgrId = MsgIdSetting.mgrIdSpan \* 2,

// ··· ···

｝

//单例模板，每个模块管理者继承模板

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

//模块管理者单例

public abstract class MgrSingle<T> : MonoBehaviour where T : MonoBehaviour

｛

public static T Instance ｛ get; protected set; ｝

protected void Awake()

｛

if(Instance != null)

DestroyImmediate(Instance.gameObject);

Instance = this as T;

OnAwake();

｝

protected abstract void OnAwake();

｝

//测试Demo

//模块管理者需要定义自己的消息体格式，消息ID，在指定的时机注册、注销所需消息

//任何脚本都可以发送消息，发送后会执行对应注册的callback回调

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public enum Demo01MsgId

｛

//模块消息Id，获取起点Id，依次取值

dufaultId = MgrId.demo01MgrId,

creatRole = MgrId.demo02MgrId + 1,

deleteRole = MgrId.demo03MgrId + 2,

// ······

｝

public class Demo01Msg : Msg

｛

//模块自定义消息体

public Demo01Msg(int newmsgId, string newname, bool newsexual, int newage, object newsender = null)

｛

msgId = newmsgId;

name = newname;

sexual = newsexual;

age = newage;

sender = newsender;

｝

public string name;

public bool sexual;

public int age;

｝

public class Demo01Manager : MgrSingle<Demo01Manager>

｛

//Awake

protected override void OnAwake()

｛

NotifyManager.Instance.Attach(Callback, new int[] ｛ (int)Demo01MsgId.creatRole, (int)Demo01MsgId.deleteRole ｝);

｝

void OnDestroy()

｛

NotifyManager.Instance.Detach(Callback, new int[] ｛ (int)Demo01MsgId.creatRole, (int)Demo01MsgId.deleteRole ｝);

｝

void Callback(Msg msg)

｛

if (msg == null || msg.msgId <= 0)

｛

Debug.Log("Receive A Empty Msg");

｝

else

｛

switch(msg.msgId)

｛

case (int)Demo01MsgId.creatRole:

Demo01Msg creatRoleMsg = msg as Demo01Msg;

Debug.Log("Creat Role: " + creatRoleMsg.name + "-" + creatRoleMsg.sexual + "-" + creatRoleMsg.age);

break;

case (int)Demo01MsgId.deleteRole:

Demo01Msg deleteRoleMsg = msg as Demo01Msg;

Debug.Log("Delete Role: " + deleteRoleMsg.name + "-" + deleteRoleMsg.sexual + "-" + deleteRoleMsg.age);

break;

default:

Debug.LogWarning("Receive A Msg Without Callback");

break;

//······

｝

｝

｝

public void Update()

｛

if (Input.GetKeyDown(KeyCode.Space))

SendMsg();

｝

void SendMsg()

｛

Demo01Msg msg = new Demo01Msg((int)Demo01MsgId.creatRole, "XiaoMing", true, 18, this);

NotifyManager.Instance.PostMsg(msg);

｝

｝

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Demo02Manager : MgrSingle<Demo02Manager>

｛

//Awake

protected override void OnAwake()

｛

NotifyManager.Instance.Attach(Callback, (int)Demo01MsgId.creatRole);

｝

void OnDestroy()

｛

NotifyManager.Instance.Detach(Callback, (int)Demo01MsgId.creatRole);

｝

void Callback(Msg msg)

｛

if (msg == null)

｛

Debug.Log("Receive A Empty Msg");

｝

else

｛

switch (msg.msgId)

｛

case (int)Demo01MsgId.creatRole:

Demo01Msg creatRoleMsg = msg as Demo01Msg;

Debug.Log("Demo02 Receive Demo01 Msg: Creat Role");

break;

default:

Debug.LogWarning("Receive A Msg Without Callback");

break;

//······

｝

｝

｝

｝

三、

以上只是一种比较常见的消息机制，在此基础上还可以进行改进、封装，因为涉及到之前的项目，这里不粘代码了，简单说一下设计思路吧

1. 消息分类：这一点与上面Demo一样，按模块对消息进行分类

2. 消息中心：每个模块的管理者作为一个消息中心，负责本模块消息的 注册、注销、处理。总消息中心，不处理具体消息，只负责不同模块间的消息流转。

要注册一条消息，模块先判断是否是本模块消息，是的话直接注册到本模块，若不是，则转发到上级的消息中心，消息中心再将消息识别下发到对应的模块，对应模块进行注册。

原本所有的消息都是在消息中心进行处理，现在在模块内部处理，消息中心只负责将消息下发到对应的模块即可。

比如说北京有一个快递中心，一天，在朝阳区的A要寄快递给海淀区的B，A找到朝阳区的快递员上门取件，快递员取件后将快递送到快递中心，再由快递中心识别快递物品，委派海淀区的快递员将快递配送给B。但第二天，朝阳区的A想要寄快递给同在朝阳区的C，同样朝阳区的快递员会上门取件，然后将快递送到快递中心，经识别后将快递委派给朝阳区的快递员，再配送给C。这样就显得比较繁琐了，快递中心的负荷也会非常大。

快递中心感觉这样好心累，要进行改革，于是增大了快递员的权利，可以直接处理自己负责地区的快递，无需再经过快递中心。这样A在寄快递给C的时候，朝阳区的快递员从A取件之后，发现这是朝阳地区内的快递，是寄给C的，就可以直接配送给C，省时又省力。如果A再给B寄快递，朝阳区的快递员取件之后，识别快递是其他地区的，就直接将快递送到快递中心，快递中心收到之后，只需识别是海淀区的，无需关注收件人是谁，再将快递流转到海淀区的快递员，由该快递员配送到C手中。这样来，整个快递流程就完美了~~

3. 记录消息及其回调：

网上搜到的大都是用委托或事件来记录消息的，前面也提到过，利用“+=”“-=”计算可以记录一条消息和多个回调。

也想过用每个消息用一个List来记录所有的回调，但明显这样是不可取的。

这里介绍另一种记录方式：

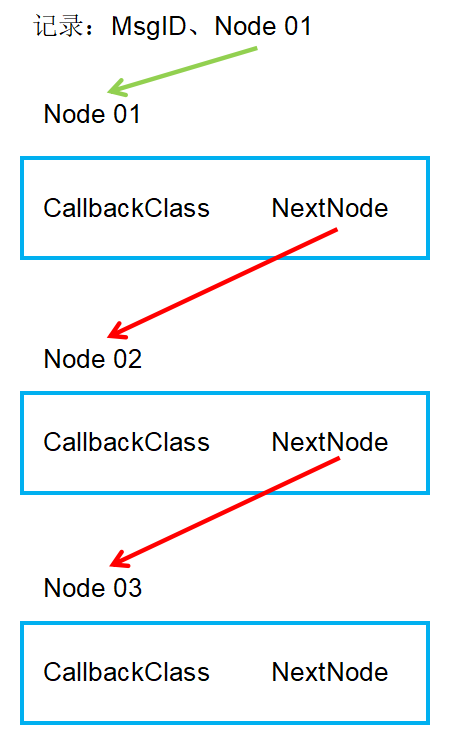
记录的不是具体某个回调函数，而是采用链表的方式记录回调函数所在的类。

3.1 写一个父类或接口，定义一个Callback函数，所有的模块管理者，都重写或实现该函数，用作消息的回调。

3.2 定义一个消息节点类(包含两个属性，本节点的回调脚本，下一个节点Next)，注册消息的第一个回调类后，其Next指向第二个回调类，依次类推，记录一个消息的所有回调类。

3.3 只需记录消息ID和第一个节点，获取第一个节点后依次获取节点的Next节点，知道Next节点为空，即遍历完所有节点。

3.4 在收到消息之后，遍历所有的节点，执行回调类的回调函数。



关于游戏架构，消息/通知机制只是其中的一部分，还有很多很多需要去学习去实践，希望以上的内容可以帮助到大家。