# **PureMVC框架在Unity中的应用（二）**

作者：[Mitty](http://gad.qq.com/user/index?id=363714)

链接：<http://gad.qq.com/article/detail/287289>

前言：

这篇文章是第二部分，通过一个简单的小例子，演示如何在Unity中应用PureMVC框架，代码我上传到了github上。有兴趣的朋友可以直接clone下来运行。

**https://github.com/kenrivcn/PureMVC\_Demo.git**

在代码库中，PureMVCFramework提供的是源码的形式，而非DLL，这样更有助于大家的理解，也方便大家自己重构和底层优化。

这是例子截图：

（建议大家把代码clone运行下，简单看下代码的结构，这样会更容易理解）



**例子的规则：**

程序启动后，随机12个道具，玩家点击“随机获得“按钮后，会从12个道具中随机出来一个，然后更新游戏次数和游戏总奖励（随机出来的道具价格累加），并弹出奖励窗口，显示具体获得的道具信息，点击Back按钮返回后，重新随机道具池。

首先，从View组件上，我们需要2个View，一个是上面截图的View，我定义为MainPanelView：

MainPanelView包含了如下UI元素：

1.一个随机道具列表（这里，每个随机道具其实都是一个View，但我们并不会为此对应创建一个Mediator，而是放在MainPanelViewMediator中统一管理）

2.”随机获取“按钮

3.游戏次数的标签

4.奖励金额的标签

第二个View是显示游戏的奖励窗口，如下图：



这个View的内容更简单，一个显示奖励的Text文本，一个返回到MainPanelView的按钮。我定义为RewardTipView.

那么现在，我们有两个View组件:

**MainPanelView**

**RewardTipView**

View和外部的通信我们都是通过Mediator来负责的，那么我们也需要创建2个Mediator类：

**MainPanelViewMediator**

**RewardTipViewMediator**

这两个Mediator中分别定义了View的引用。

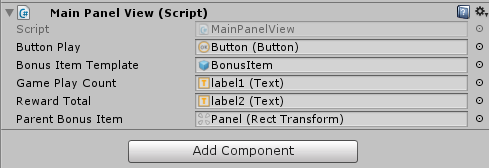
这里，我们将MainPanelView和RewardTipView制作成两个Prefab预设，并进行动态加载。然后定义

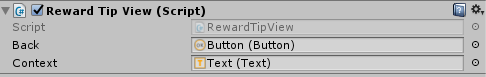
**MainPanelView.cs**

**RewardTipView.cs**

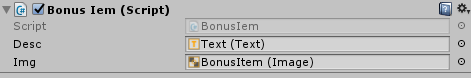
两个脚本附加在预设上，绑定我们需要”操作“的控件。

如图：





在MainPanelView中，包含了12个随机的道具，我定义为BonusItem，我们事先创建好一个BonusItem对象并隐藏，在实际生成中，我克隆BonusItem即可。我新建了一个BonusItem.cs 挂在它上面，初始化需要“操作”的控件。



View和Mediator部分就完成了。

**下面，我们再看看Model和Proxy，即数据的部分（Data Object）**

首先，我们要随机12个道具，每个道具的数据结构很简单，只包含三个字段：

1.ID

2.名称

3.价格

所以，我们定义Bonus的Model如下：

public class BonusModel ｛

    public int Id ｛ get; set; ｝

    public string Name｛get;set;｝

    public int Reward ｛ get; set; ｝

    public BonusModel(int \_id,string \_name,int \_reward)

    ｛

        Id = \_id;

        Name = \_name;

        Reward = \_reward;

    ｝

｝

(Tips:如果表示纯数据类型的Model，建议声明成struct，内存开销更小）

Model定义完成以后，在对应的Proxy中，我们定义Model的引用，在这里，我们维护一个BonusModel的列表，动态的生成，刷新，以及其它相关Model的操作，我们均在Proxy中实现，对应的Proxy，定义为BonusProxy.

但只有BonusModel还不够，我们还需要保存玩家的数据（游戏次数和奖励金额），所以，还需要定义另一个Model来保存他们，这里我定义为PlayerDataModel：

定义如下：

/// <summary>

/// 保存玩家数据

/// </summary>

public class PlayerDataModel ｛

    public int PlayGameCount ｛ get; set; ｝//游戏次数

    public int RewardTotal ｛ get; set; ｝//奖励总价

｝

对应的Proxy，定义为PlayerDataProxy，负责对PlayerDataModel的操作。

现在，Model,View部分都完成了，下一步就是分析Controller(Command)逻辑控制的部分了。

**逻辑控制部分有以下几种行为：**

**1.初始化**

启动程序后要做的工作，我们通过Command命令来做一些初始的操作，操作有：

(1) 加载MainPanelView，并注册绑定Mediator

(2) 发送Notification通知“开始随机12个道具”，这个行为我封装在另外一个Command中

初始化操作的Command，我们定义为StartUpCommand：

using System.Collections;

using System.Collections.Generic;

using PureMVC.Interfaces;

using PureMVC.Patterns;

using UnityEngine;

/// <summary>

/// 启动

/// 2.随机生成奖励池

/// </summary>

public class StartUpCommand : PureMVC.Patterns.SimpleCommand ｛

 public override void Execute (INotification notification) ｛

 //create ui

 GameObject obj = GameObjectUtility.Instance.CreateGameObject ("\_Prefabs/MainPanelView");

 //bind mediator

 Facade.RegisterMediator (new MainPanelMediator (obj));

 //更新12个道具

 SendNotification (MyFacade.REFRESH\_BONUS\_ITEMS);

 ｝

｝

StartUpCommand 中，首先创建MainPanelView UI组件并完成Mediator的绑定。初始化完成以后，SendNotification (MyFacade.REFRESH\_BONUS\_ITEMS);

发送REFRESH\_BONUS\_ITEMS通知，开始随机12个道具。

步骤(2)是发送Notification通知“开始随机12个道具”（REFRESH\_BONUS\_ITEMS），这部分业务逻辑不适合放在StartUpCommand中实现，因为玩家玩过一次后，要重新再次随机12个道具，所以这部分的业务逻辑要复用，那么我们将他定义在另一个叫RefreshBonusPoolCommand中实现，专门用于重新随机12个道具。(代码就不贴了）

**2.开始随机**

并完成初始化操作以后，我们现在就可以点击“随机获取”按钮，开始玩了。

点击按钮以后：

执行一个Command（即发送Notification通知），这里定义为PlayCommand，在PlayCommand类中，生成一个随机数（0-12），然后通过随机数，获取PlayerDataProxy中保存的BonusModel数组指定引用，再获取PlayerDataProxy，修改玩家的数据（游戏次数，奖励金额）

代码如下：

using System.Collections;

using System.Collections.Generic;

using PureMVC.Interfaces;

using PureMVC.Patterns;

using UnityEngine;

public class PlayCommand : PureMVC.Patterns.SimpleCommand ｛

    public override void Execute (INotification notification) ｛

        //开始随机

        BonusProxy bonus = MyFacade.Instance.RetrieveProxy (BonusProxy.NAME) as BonusProxy;

        int id = Random.Range (0, bonus.BonusLists.Count);

        Debug.Log ("result:" + bonus.BonusLists[id].Name + "," + bonus.BonusLists[id].Reward);

        //改变数值 并发送消息

        PlayerDataProxy playerData = Facade.RetrieveProxy (PlayerDataProxy.NAME) as PlayerDataProxy;

        if (playerData != null) ｛

            playerData.GetReward (bonus.BonusLists[id].Reward, bonus.BonusLists[id].Name);

            Debug.Log ("================PlayCommand");

        ｝

    ｝

｝

在PlayCommand中，获取随机到的道具信息，然后再获取PlayerDataProxy ，完成数据更新操作。

在上面的代码中：

PlayerDataProxy playerData = Facade.RetrieveProxy (PlayerDataProxy.NAME) as PlayerDataProxy;

        if (playerData != null) ｛

            playerData.GetReward (bonus.BonusLists[id].Reward, bonus.BonusLists[id].Name);

            Debug.Log ("================PlayCommand");

        ｝

GetReward 方法中，我们传入随机生成道具的奖励金额和道具名称，那么在GetReward方法中要做哪些操作呢？

更改PlayerData数据，然后发送通知，让MainPanelView UI组件进行更新。

代码如下：

public void GetReward (int reward, string info) ｛

        PlayerData.PlayGameCount++;

        PlayerData.RewardTotal += reward;

        //发送消息 更新MainPanelView UI组件 通知订阅者

        SendNotification (MyFacade.UPDATE\_PLAYER\_DATA, info + reward);

    ｝

我们发送了一个UPDATE\_PLAYER\_DATA通知，并将info+reward拼接在一起，传递出去。

**谁会接受UPDATE\_PLAYER\_DATA通知？**

1）某个View组件

2）某个Command

在这里，只有MainPanelView View组件会接受UPDATE\_PLAYER\_DATA通知，它是如何配置的呢？

我上面并没有贴出MainPanelViewMediator的代码，大家可以参考github上的查看。

在Mediator中需要override一个父类的方法：

  public override IList<string> ListNotificationInterests () ｛

        IList<string> list = new List<string> () ｛ MyFacade.UPDATE\_PLAYER\_DATA ｝;

        return list;

    ｝

在List中加入我们要进行监听的所有事件。

然后我们在MainPanelViewMediator中，还要override另外一个方法：

public override void HandleNotification (INotification notification) ｛

 switch (notification.Name) ｛

            case MyFacade.UPDATE\_PLAYER\_DATA:

//更新UI

                    if (playerData != null) ｛

                        View.GamePlayCount.text = string.Format ("游戏次数:｛0｝", playerData.PlayerData.PlayGameCount);

                        View.RewardTotal.text = string.Format ("游戏总奖励:｛0｝", playerData.PlayerData.RewardTotal);

                        //show reward tip view

                        SendNotification (MyFacade.REWARD\_TIP\_VIEW, notification.Body);

                    ｝

//更新UI组件的显示

｝

｝

在HandleNotification 完成UI的更新后，看看最后一行代码：

SendNotification (MyFacade.REWARD\_TIP\_VIEW, notification.Body);

我又发送了另外一个Notification通知REWARD\_TIP\_VIEW，这是启动了另外一个Command，用于弹出奖励窗口，并显示奖励信息。

代码如下：

using System.Collections;

using System.Collections.Generic;

using PureMVC.Interfaces;

using PureMVC.Patterns;

using UnityEngine;

public class RewardTipCommand : PureMVC.Patterns.SimpleCommand ｛

 public override void Execute (INotification notification) ｛

 //显示结算结果

 RewardTipViewMediator mediator = Facade.RetrieveMediator (RewardTipViewMediator.NAME) as RewardTipViewMediator;

 if (mediator == null) ｛

 GameObject obj = GameObjectUtility.Instance.CreateGameObject ("\_Prefabs/RewardTipView");

 mediator = new RewardTipViewMediator (obj);

 Facade.RegisterMediator (mediator);

 ｝

 //update reward tip view

 SendNotification (MyFacade.UPDATE\_REWARD\_TIP\_VIEW, notification.Body);

 ｝

｝

在RewardTipCommand 当中，我们要先判断我们是否创建过RewardTipView组件，如果没有，则通过Resource.Load进行加载，并绑定Mediator。

在代码的最后一行：

SendNotification (MyFacade.UPDATE\_REWARD\_TIP\_VIEW, notification.Body);

我发送了另外一个Notification通知，这里是为了演示功能，因为你可以直接调用Mediator来进行UI更新，notification.Body就是我们传递过来的奖励信息。

在RewardTipViewMediator中,注册UPDATE\_REWARD\_TIP\_VIEW通知：

public override IList<string> ListNotificationInterests () ｛

 IList<string> list = new List<string> () ｛ MyFacade.UPDATE\_REWARD\_TIP\_VIEW ｝;

 return list;

 ｝

public override void HandleNotification (INotification notification) ｛

 switch (notification.Name) ｛

 case MyFacade.UPDATE\_REWARD\_TIP\_VIEW:

 if (!View.isActiveAndEnabled) ｛

 View.gameObject.SetActive (true);

 ｝

 string text = notification.Body as string;

 //update text

 View.SetText (text);

 break;

 ｝

 ｝

并完成内容的更新。

最后一步，我在RewardTipView UI组件上点击Back按钮时，要发送一个Notification通知，让我们重新再次随机12个道具，这个行为已经封装到了一个Command中了，前面提到的RefreshBonusPoolCommand。

SendNotification (MyFacade.REFRESH\_BONUS\_ITEMS);

这样，例子的整个流程就都走完了。

现在，看看我们定义了哪些类：

**Model和Proxy：**

**BonusModel->BonusProxy**

**PlayerDataModel->PlayerDataProxy**

**View和Mediator：**

**MainPanelView->MainPanelViewMediator**

**RewardTipView->RewardTipViewMediator**

**Controller和Command：**

**PlayCommand**

**RefreshBonusPoolCommand**

**RewardTipCommand**

**StartUpCommand**

上面只是讲到了这些类的职责，但我如何绑定Proxy，Mediator，Controller？ 如何在Mediator中获取Proxy，又如何在Command中同时获取Mediator和Proxy呢，如果你还记得上一篇讲到的Facade的话，它就是用来管理上面我们增加的这些“类”的。

(当然，我们需要创建一个自定义的Facade来做这些操作。)

在使用之前，我们需要通过Facade进行注册绑定，将他们保存在对应的哈希表中，绑定的方式有几种？

**1.直接在自定义的Facade类中注册绑定：**

在Facade基类中有三个virtual方法：

**InitializeModel();**

**InitializeController();**

**InitializeView();**

通过函数名就可以知道，分别是初始化Model，Controller，View的，这也是代码的执行顺序，View是基于数据驱动，总是要在最后才完成初始化。

在自定义的Facade中，我们可以将Model，View，Controller的类在这些方法中完成绑定。

如下：

protected override void InitializeController()

    ｛

        base.InitializeController();

        //注册Command

        RegisterCommand(START\_UP, typeof(StartUpCommand));

 RegisterCommand(REFRESH\_BONUS\_ITEMS, typeof(RefreshRewardPoolCommand));

        RegisterCommand(PLAY, typeof(PlayCommand));

 RegisterCommand (REWARD\_TIP\_VIEW, typeof(RewardTipCommand));

    ｝

protected override void InitializeModel()

    ｛

        base.InitializeModel();

        RegisterProxy(new BonusProxy(BonusProxy.NAME));

        RegisterProxy(new PlayerDataProxy(PlayerDataProxy.NAME));

    ｝

通常在游戏开发的过程中，我不会在初始化的时候，就把所有用到的UI都加载到内存当中（当然，UI不多的情况是可以的，在例子中我们也可以这样做），所以采用动态加载的形式，比如放在Command中。

**2.通过Command进行注册**

实际上，我们总是要这样去使用，我们总会有这样的需求，要动态的加载和释放一些Model，View和Controller，比如游戏战斗中的相关数据，你在启动的时候就加载，显然是不合理的，比如我们在战斗Loading的时候，通过Command来注册战斗中需要的所有Model，View，Controller，在退出战斗回到菜单的时候，我们也要通过Command来Remove掉那些Model，View和Controller(Facade中提供了注册，移除，获取等等方法）。

下面是Facade自定义类的代码：

using System;

using System.Collections;

using System.Collections.Generic;

using PureMVC.Interfaces;

using UnityEngine;

/// <summary>

/// PureMVC 核心类 Facade

/// 只需要创建一个即可

///

/// 负责完成proxy,mediator,command的初始化工作

/// 获取也均通过facade

///

///

/// 重写virtual方法

///

/// 执行顺序：

/// Model->Controller->View->Facade的顺序

/// 重写Facade一定要调用base.InitializeFacade()

///

/// </summary>

public class MyFacade : PureMVC.Patterns.Facade ｛

    public const string START\_UP = "start\_up";

    public const string CREATE\_BONUS\_ITEMS = "create\_bonus\_items";

    public const string REFRESH\_BONUS\_ITEMS = "refresh\_items"; //

    public const string UPDATE\_PLAYER\_DATA = "update\_player\_data";

    public const string PLAY = "play";

    public const string REFRESH\_BONUS\_UI = "refresh\_bonus\_ui";

    public const string UPDATE\_REWARD\_TIP\_VIEW = "update\_reward\_tip\_view";

    public const string REWARD\_TIP\_VIEW = "reward\_tip\_view";

    /// <summary>

    /// 静态初始化

    /// </summary>

    static MyFacade () ｛

        m\_instance = new MyFacade ();

    ｝

    /// <summary>

    /// 获取单例

    /// </summary>

    /// <returns></returns>

    public static MyFacade GetInstance () ｛

        return m\_instance as MyFacade;

    ｝

    /// <summary>

    /// 启动MVC

    /// </summary>

    public void Launch () ｛

        //通过command命令启动游戏

        SendNotification (MyFacade.START\_UP);

    ｝

    /// <summary>

    /// 初始化Controller，完成Notification和Command的映射

    /// </summary>

    protected override void InitializeController () ｛

        base.InitializeController ();

        //注册Command

        RegisterCommand (START\_UP, typeof (StartUpCommand));

        RegisterCommand (REFRESH\_BONUS\_ITEMS, typeof (RefreshRewardPoolCommand));

        RegisterCommand (PLAY, typeof (PlayCommand));

        RegisterCommand (REWARD\_TIP\_VIEW, typeof (RewardTipCommand));

    ｝

    /// <summary>

    /// 初台化View,Initializes the view.

    /// View在Model和Controll之后运行

    /// UI的创建我放到Command中执行

    /// </summary>

    protected override void InitializeView () ｛

        base.InitializeView ();

    ｝

    /// <summary>

    /// 注册Proxy

    /// </summary>

    protected override void InitializeFacade () ｛

        base.InitializeFacade ();

    ｝

    /// <summary>

    /// 初始化Model 数据模型  Proxy

    /// </summary>

    protected override void InitializeModel () ｛

        base.InitializeModel ();

        //也可以放在Command中

        RegisterProxy (new BonusProxy (BonusProxy.NAME));

        RegisterProxy (new PlayerDataProxy (PlayerDataProxy.NAME));

    ｝

｝

在最上面，我们定义了一些常量：

public const string START\_UP = "start\_up";

    public const string CREATE\_BONUS\_ITEMS = "create\_bonus\_items";

    public const string REFRESH\_BONUS\_ITEMS = "refresh\_items";//

    public const string UPDATE\_PLAYER\_DATA = "update\_player\_data";

    public const string PLAY = "play";

    public const string REFRESH\_BONUS\_UI = "refresh\_bonus\_ui";

    public const string UPDATE\_REWARD\_TIP\_VIEW = "update\_reward\_tip\_view";

    public const string REWARD\_TIP\_VIEW = "reward\_tip\_view";

这些常量来自于Event和Notification，这些建议放在一个单独的常量类中定义。

**如何启动PureMVC框架？**

PureMVC和MonoBehaviour是无关的，所以，我们需要启动PureMVC框架，来执行StartCommand命令。

可以简单的创建一个继承自MonoBehaviour的类：

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class App : MonoBehaviour ｛

    void Awake () ｛

        //启动PureMVC，完成Controller，Proxies，Mediators的初始化工作

        MyFacade.GetInstance ().Launch ();

    ｝

｝

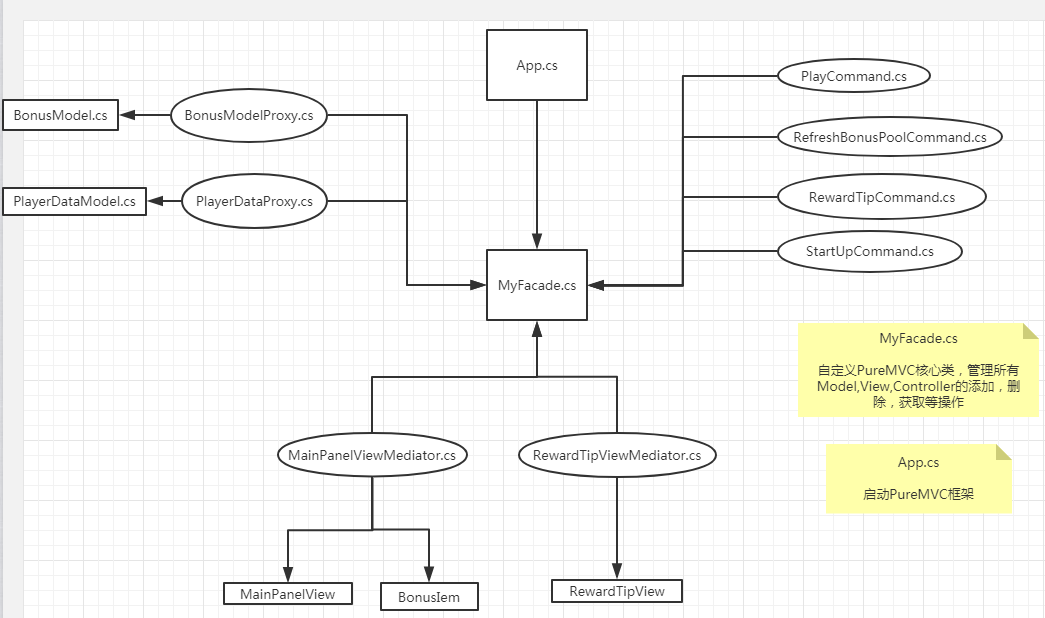
在Launch方法中，即是发送了一条通知：

SendNotification(MyFacade.START\_UP);

执行StartCommand命令。

这样，整个PureMVC就运转起来啦！

Demo结构图示：



最后，说说PureMVC在使用过程中，需要注意哪些地方？

**1.装箱和拆箱**

 发送Notification通知时，赋带的参数是object类型的，如果你传递的实参是值类型，比如struct,那么就要注意装箱过程带来的性能消耗，每一次装箱的过程都要在拖管堆中进行内存分配有字段的复制，如果传递的数据比较大，建议使用引用类型。

**2.反射**

 反射是非常好用的功能，但却要付出性能的代价，他要在程序集中进行搜索，在上面的例子中并没有提到反射，因为他足够简单，但在实际开发中，我们通常要有一个UIManager类来统一的管理UI的创建，删除，显示，隐藏等等操作，我们通过定义的一些枚举来加载指定的UI，问题是如何去动态的绑定UI对应的Mediator（我们不考虑switch这种做法），Class本身不能作为参数，所以有的代码是通过反射的形式来动态获取类型的特征，完成类型的创建并绑定。

比如：

现在有一个UIViewA组件，对应的Mediator就约定为 “View组件名称“+Mediator（也可以自行配置，这里主要是方便使用），所以UIViewA对应的Mediator就是UIViewMediator，有了Mediator的字符串名称，通过反射查找到具体的类型特征，创建并绑定，使用起来很方便，虽然这种情况下使用反射对性能影响不是很大，但"勿以善小而不为"，性能开销的地方要尽量的避免，反射可以通过将类型信息存储在字典中来代替。

但一定要注意在释放时，也要从字典中删除掉。

**3.Proxy，Mediator，Command创建**

在使用的过程中会发现，Proxy，Mediator，Command每次的创建，重复添加一些要override的方法，定义相关常量，绑定等等操作会有些繁琐，建议使用模板生成Template Generator和代码片断snippet这两个可以提高开发效率的工具，让这些繁琐的工作变得更加的快捷方便。

好啦，PureMVC的解读到此结束，感谢阅读，如文中有误，欢迎指正。