# 1. 第一课 课程介绍 讲稿（持续更新）

大家好，我是凉鞋，我们在这个视频对《Unity 游戏框架搭建 决定版》课程进行一个简单的介绍。

首先先介绍一下自己，我是凉鞋，开源框架 QFramework 的作者，刚毕业半年的时候开始接触的 Unity，接触 Unity 两个月后开始着手写 QFramework，一直持续至今，已有 6 年。而《Unity 游戏框架搭建》系列是笔者写 QFramework 时的思考与总结，此系列目前有：

* 2016 ~ 2017 年写的 《Unity 游戏框架搭建 2017》免费专栏。
* 2017 ~ 2018 年写的 《Unity 游戏框架搭建 2018》免费专栏和视频课程。
* 2018 ~ 2019 年写的 《Unity 游戏框架搭建 2019》收费专栏和视频课程（只能在 siki 学院观看）。
* 2019 ~ 2021 年写的 《Unity 游戏框架搭建 2020》以及 2021 版小班培训课程（暂时未对外开放，只能小班观看）。

笔者几乎是每年重写一版，而每次重写都会在原有的基础上引入大量的新的内容，这些新的内容是笔者每年自己在框架方面提升的部分，也就是说笔者是每年都在不断地突破自己，直到今年，笔者设计了一套自洽且完备的系统设计方法论，设计完此系统设计方法论后，笔者觉得很难在框架的架构方面上再有新的突破了，所以笔者决定制作一套 Unity 游戏框架搭建 决定版 来介绍这套系统设计方法论。

而这个系统设计方法论暂时命名为 QFramework System Design Architecture，即 QFramework 系统设计架构，笔者使用它进行了大量的实践和验证，不管是团队协作、开发效率、扩展性、稳定性、测试性、纸上设计，其表现都非常出色，重点是笔者在 .Net Core Web 端、React 前端、Unity 客户端都成功实践过这套架构。

而此框架搭建的决定版则是给大家演示，如何从一个空项目慢慢把架构演化成 QFramework System Design Architecture 的，在演化的过程中，会充分展现笔者的思考过程，以及所有需要掌握的关键的 C#、数据结构与算法、面向对象相关的知识。

不管是 Unity 初学者还是高手，都可以从此课程中找到与自己匹配的架构演化阶段，从而提升自己的编程、程序设计、架构能力。

而此课程最终产出的框架，也可以作为公司或个人项目的基础框架进行迭代，从而迭代出来符合个人项目和公司需求的属于自己的框架。

此课程所涉及的工具类只有 3 个，即：

* Singleton 模板
* IOC 容器
* TypeEventSystem（基于类型的事件机制）

实现这三个工具的课程内容只占 10% 左右，剩下的大部分内容则是实现这三个工具类的 C# 核心基础、面向对象设计和建模、架构理论相关，所以此课程的技术门槛低，只要能自己写个小的 Unity 游戏 Demo ，就可以学习此课程。

本课程会按照如下的顺序进行讲解：

* 开发架构演化以及相关基础知识补充
* QFramework 系统设计架构的最佳实践

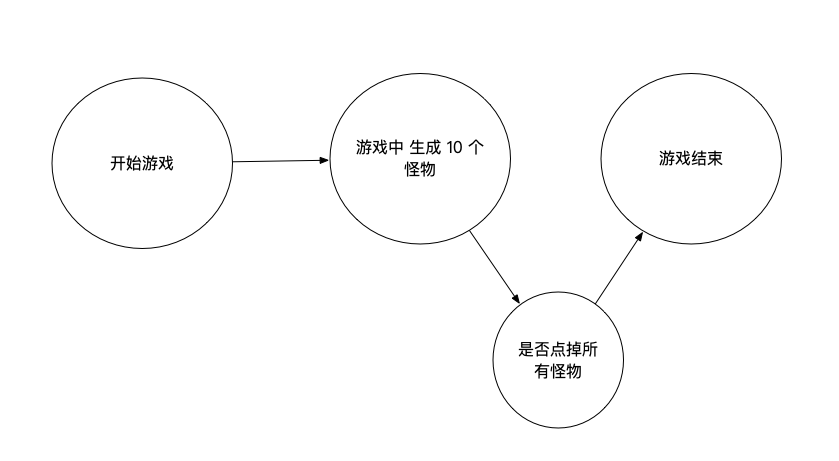
学完此课程，并且掌握 QFramework System Design Architecture，大家在架构设计方面上会打好良好的基础，再去看一些大部著，容易事半功倍。而能够掌握这套架构的同学，则可以应付工作中遇到的 90% 的架构或框架相关的问题，市面上的很多开源框架比如 PureMVC、StrangeIOC、uFrame、ET、GameFramework 等都会更快上手，因为很多设计原理都是相同的。

OK，此视频的内容就这些，拜拜。

# 2. 第二课 无架构的游戏《点点点》快速实现

大家好，我是凉鞋，从此课开始，我们开始着手架构的演化，为了清晰地体现架构，我们先制作一个小游戏《点点点》

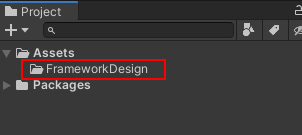
游戏的大体流程如下：



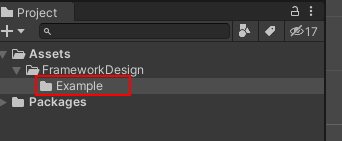
流程非常简单，用于体现我们的架构足够了。

OK，接下来，我们来着手实现它。

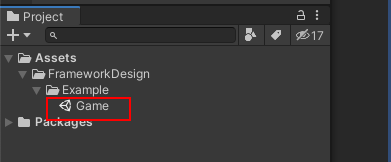
首先我们创建一个目录，叫做 FrameworkDesign ，如下所示：



接着我们再创建一个目录，叫 Example，如下所示：

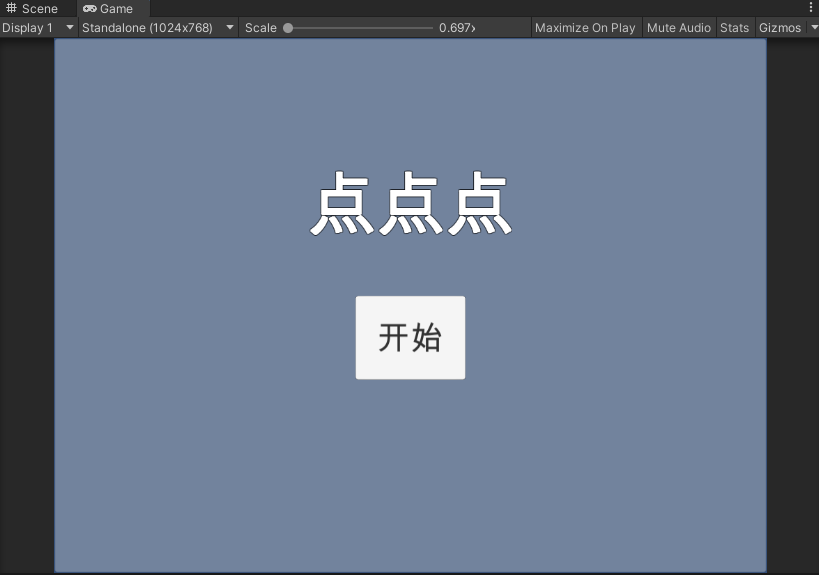


Example 中创建一个场景，如下:

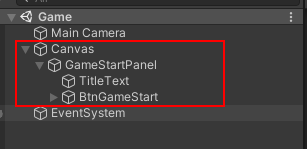


创建好场景之后，双击打开场景。

我们先实现开始拼游戏界面，如下:



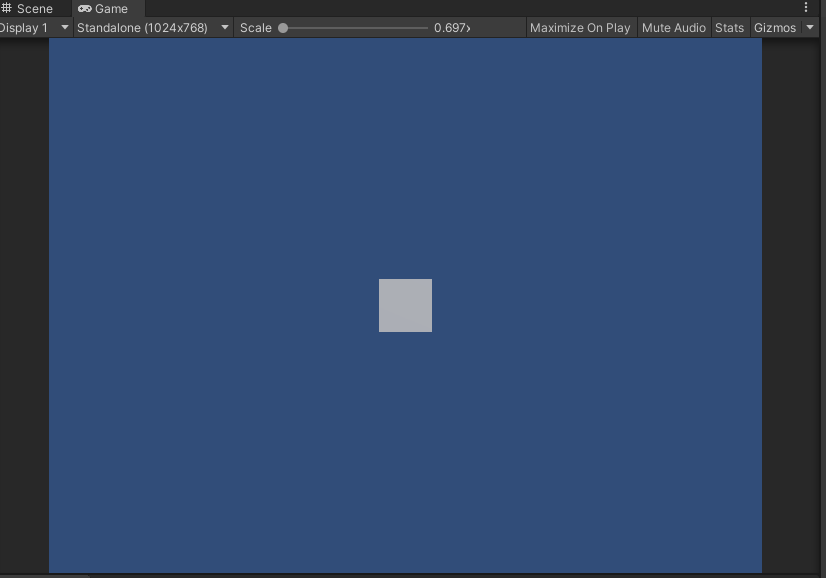
场景结构如下：



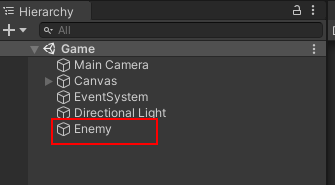
这里我们不要太在意具体的样式，关注逻辑结构即可。

现在开始游戏界面拼好了，接下来我们来实现核心游戏部分的内容。

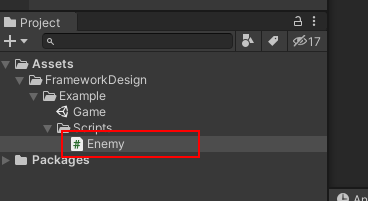
我们先在场景中创建 1 个 Cube，如下：



这个 Cube 的名字叫做 Enemy，如下：

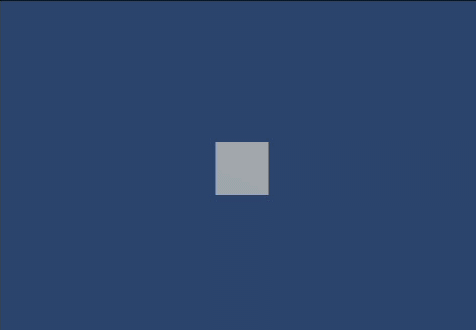


我们创建一个脚本，叫做 Enemy，所在目录如下：



代码如下:

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
 }  
 }  
}

将此脚本挂在 Enemy GameObject 上，运行游戏，结果如下：  


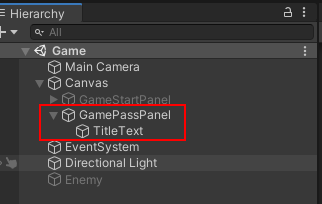
这样我们就完成了，消除一个敌人的功能。

再接下来我们实现，消除完这个敌人后，显示游戏通关界面的功能。

首先需要拼一下游戏通关界面，如下：



场景结构如下：



非常简单。

这样，我们的界面就拼完了。

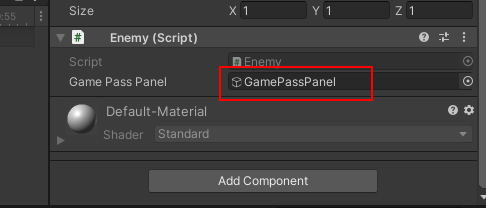
接下来，我们来实现，消除敌人之后，显示游戏通关界面的功能。

首先我们在 Enemy 脚本上增加一个 GameObject 类型的成员变量，用于存储 GamePassPanel 这个界面，并且实现打开游戏通关界面的代码。

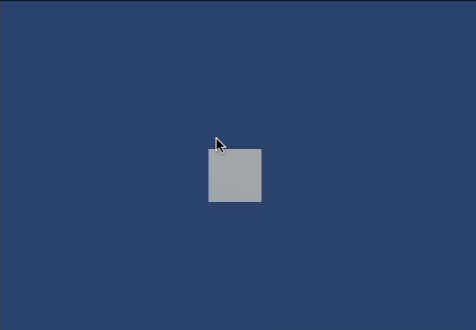
如下：

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 public GameObject GamePassPanel;  
   
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 // 显示游戏通关界面  
 GamePassPanel.SetActive(true);  
 }  
 }  
}

然后我们讲 GamePassPanel 这个 GameObject 拖到，Enemy 的 Enemy 组件上，如下：



这样我们就实现好了，消除敌人后，显示游戏通关的功能，运行 Unity 后，结果如下：



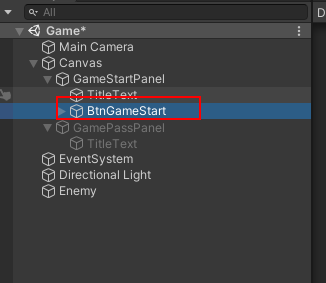
OK，这样，我们就实现好了这个功能。

接下来，我们来把前边的开始界面的流程穿起来。

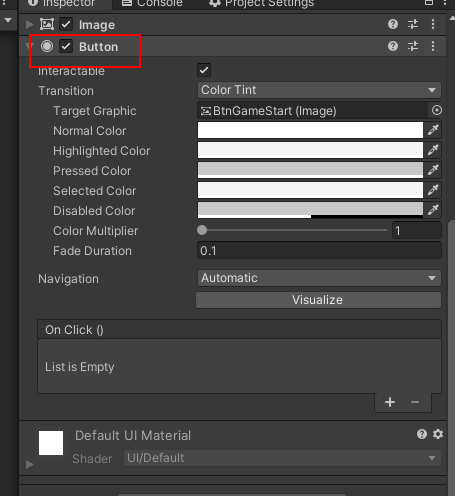
我们要实现点击游戏开始之后，然后显示敌人，然后消除敌人之后显示游戏结束。

点击游戏开始显示敌人的功能实现如下。

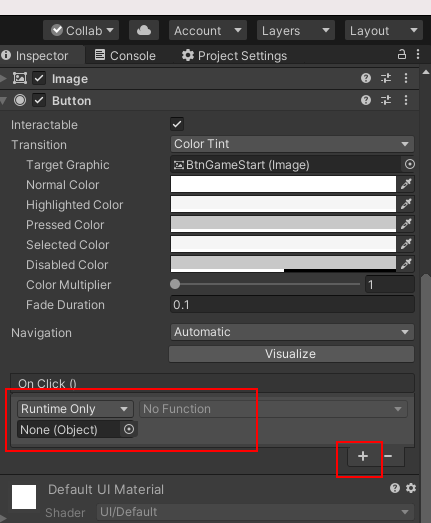
我们选定 BtnGameStart,如下所示：



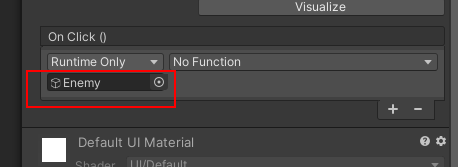
然后再 监视面板 （Inspector）上可以看到 Button 组件，如下：



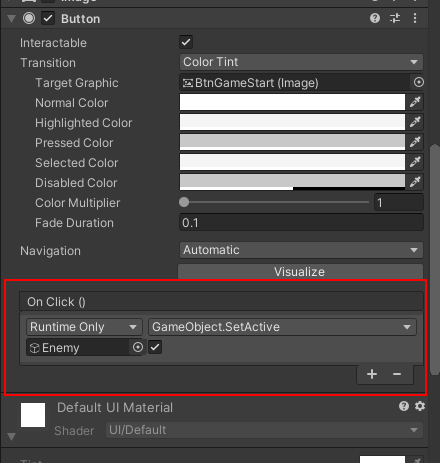
然后增加 OnClick 事件，如下：



将 Enemy 拖拽到空槽中，如下：

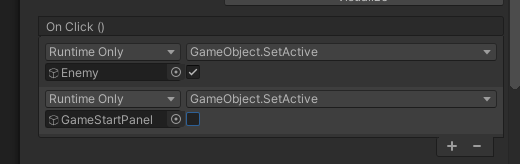


然后选择 GameObject.SetActive 方法，并把值设置为 true，如下所示；



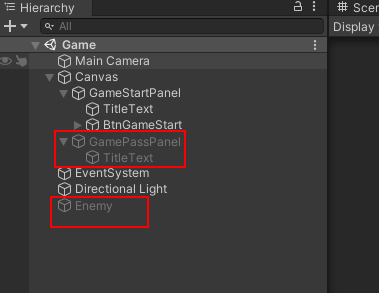
OK，这样点击开始游戏显示敌人的功能就完成了。

但是显示敌人还不够，还需要将 GameStartPanel 自己隐藏，这个功能的实现也比较简单，再加个 OnClick 事件就可以了，这里直接给出结果，如下：

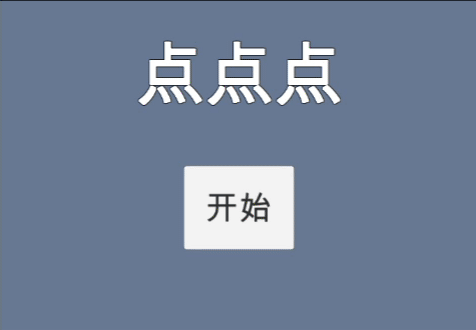


OK，这样就实现完了。

接下来将场景里的 GameObject 的显示状态设置如下：



运行游戏，结果如下：

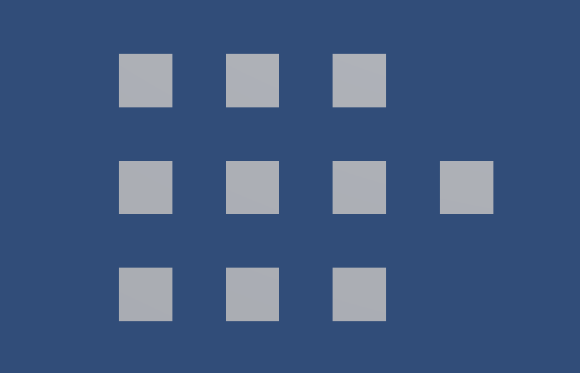


OK，到此游戏的基本流程算是完成了。

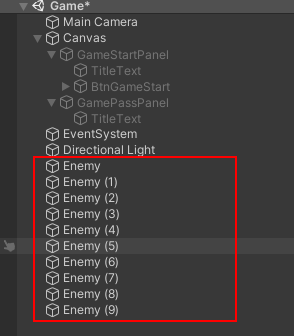
最后，还剩下一个功能，就是要创建 10 个 敌人，然后 10 个敌人全部消灭之后，才显示游戏通关。

我们接下来实现这个功能。

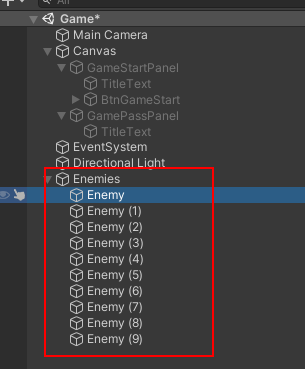
首先，复制 9 个 Enemy GameObject 将位置大致摆成如下：



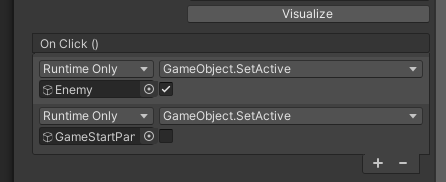
场景结构如下所示:



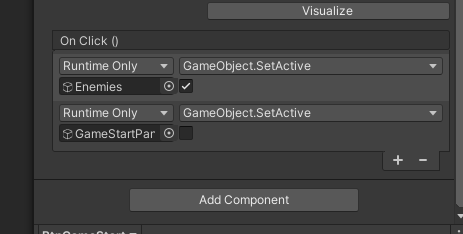
这样场景比较乱，而且新增的功能影响到了之前的 点击游戏开始显示敌人 这个功能，所以我们创建一个空的 GameObject 叫做 Enemies，然后把这 10 个 Enemy 设置成 Enemies 的子节点，如下所示：



我们再回看下 BtnGameStart 的 OnClick 部分内容，如下：



我们把第一个事件里的 Enemy 替换成 Enemies，如下：

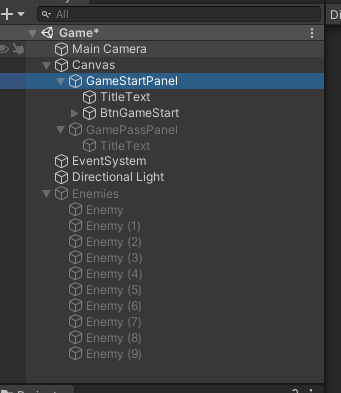


这样原来的功能就修复好了。

最后，我们再 Enemy 脚本上加上一个逻辑，如下：

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 public GameObject GamePassPanel;  
  
  
 /// <summary>  
 /// 击杀数量   
 /// </summary>  
 private static int mKillCount = 0;  
   
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
  
 mKillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (mKillCount == 10)  
 {  
 // 显示游戏通关界面  
 GamePassPanel.SetActive(true);  
 }  
 }  
 }  
}

代码不是很难，就是一个静态变量的使用，接着我们设置好场景的 GameObject 显示状态，如下：

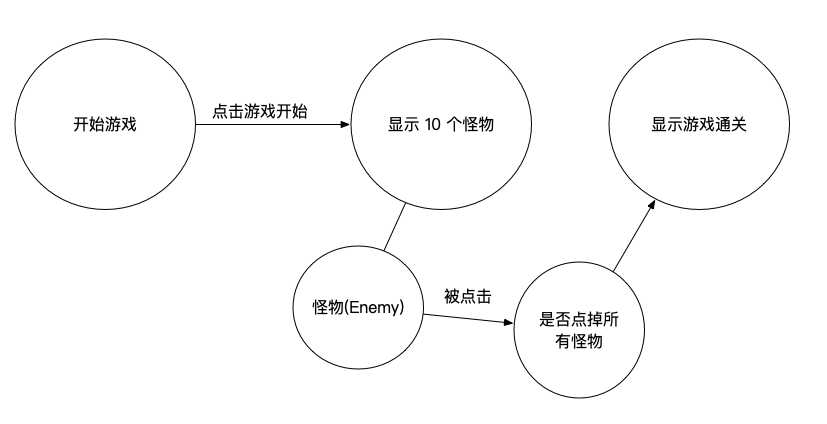


运行游戏，结果如下:



OK，到此，我们完成了此游戏的全部功能。

接下来，笔者整理了一下游戏的逻辑，如下图所示：



游戏实现了之后，我们发现 Unity 做游戏是真的方便，整个游戏所需要的脚本只有一个类，也就是 Enemy 类，其他的逻辑都可以在编辑器上拖拖拽拽就可以完成，这也就是为什么 Unity 的用户基数如此庞大的原因。

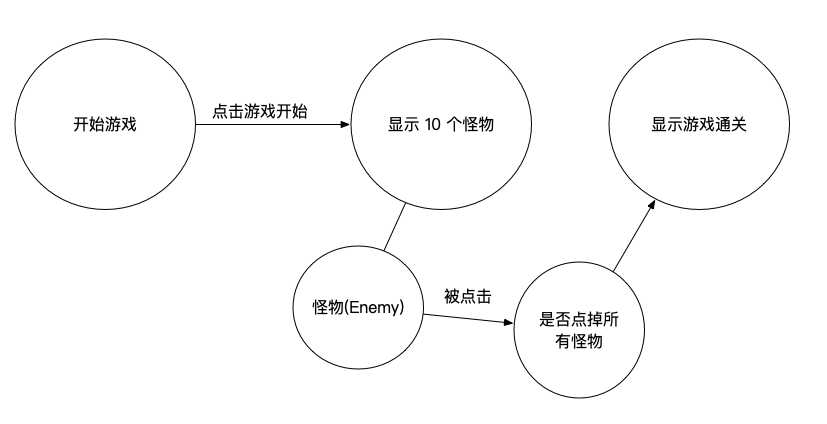
到此我们做好了学习此门课程的基本准备，也就是一个无架构的游戏《点点点》实现。

从下堂课开始，我们来开始进行架构的演化。

此课程的内容就这些，我们下堂课再见，拜拜。

# 3. 第三课 树结构 与 无架构项目的优缺点

在上一堂课，我们实现了一个 无架构的小游戏《点点点》，其逻辑如下图所示：



游戏的运行流程如下：



在此篇，我们来开始演化这个游戏的架构。

OK，首先这里大家可能会问什么是架构？

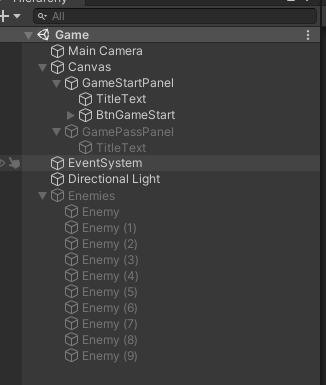
在初期，我们可以把架构简单理解为结构。

哪些结构呢？

在 Unity 中有 场景结构、项目结构，在一般开发的时候有代码结构等。

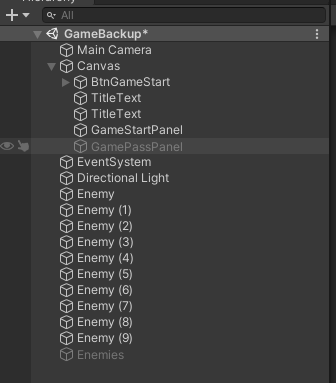
我们看下《点点点》这个游戏目前的结构。

场景结构如下：



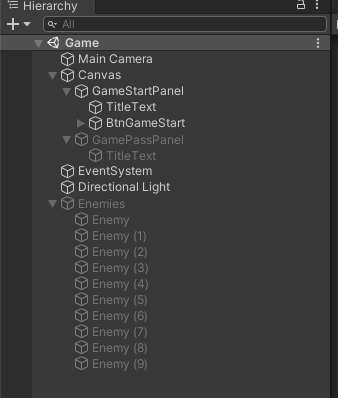
目前来看，不是很乱，不是很乱的原因是因为根节点比较少，这是因为笔者在上一堂课在做的过程中就考虑了场景的可维护性，当然这个是笔者养成的习惯了。

笔者先临时把这些场景打乱，如下：



OK，这个场景相对乱了一些，可能场景中的元素比较少，大家觉得不乱，笔者在这只说一点，就是 场景中的两个TitleText 不知道哪个是在哪个 Panel 下的，这就是乱。

OK，笔者把场景改回来，如下:

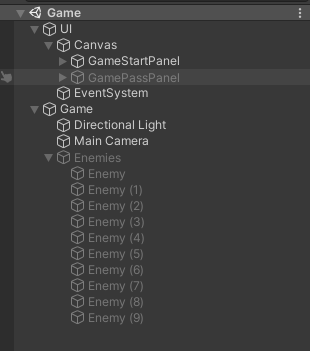


这样场景是层次分明，逻辑清晰的。

那么是什么造成了这两者的区别呢？

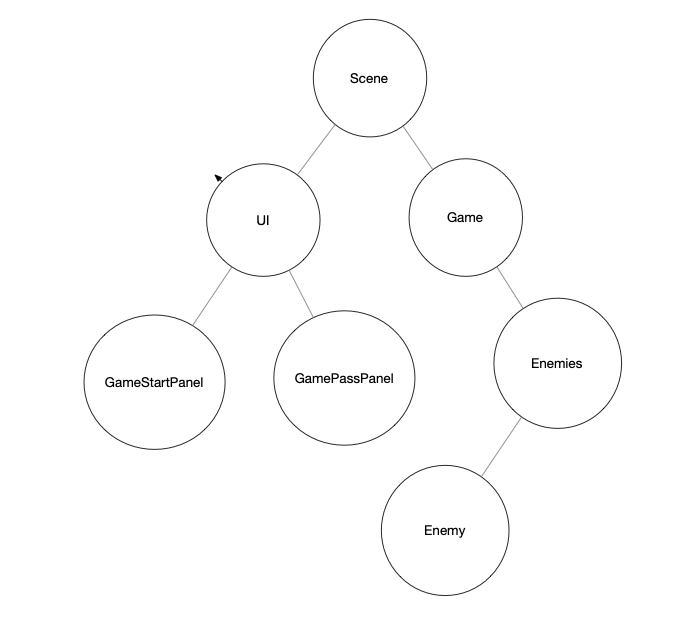
答案很简单，就是父子结构，或者说树状结构。

接下来笔者使用树结构再做进一步整理，整理之后如下：



笔者把整个场景的内容分为了两个根节点，分别是 UI 和 Game。

我们用一张图表示如下：



OK，这张图还是比较清晰的。

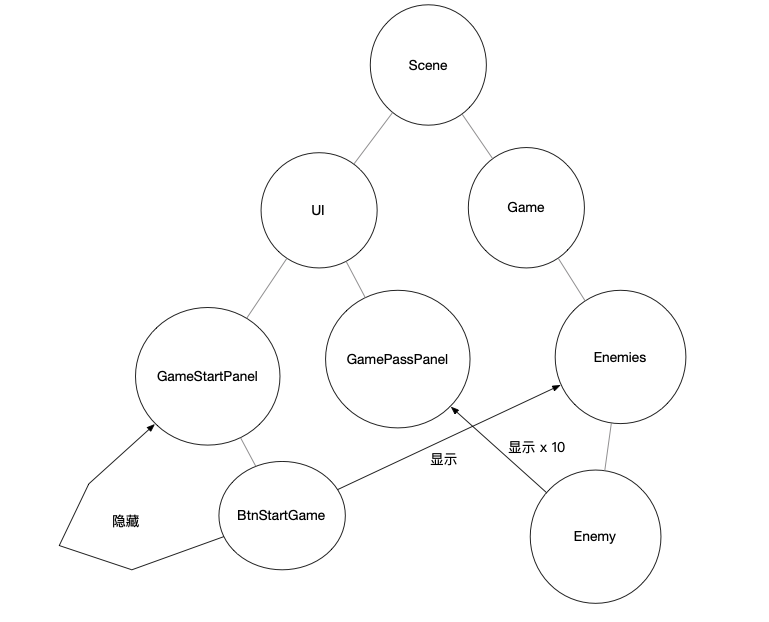
为什么要介绍树状结构呢？

因为大部分的视图结构都是树状结构，比如 iOS 的 UIKit 和 Android 的 Activity 还有前端的 HTML（本质是 XML)，基本上都是树状结构。

所以在理想情况下，一个 App 或一个游戏最好是只有一颗树。

OK，以上这张图，只是表现了各个节点的物理关系。

接下来，我们在这张图上补充引用关系，引用关系说白了就是谁调用了谁，补充之后如下图所示



OK，看以上图的引用关系，现在由于元素的数量比较少，所以看着还是比较容易理解的。

而我们在刚接触 Unity 的时候，所做出来的游戏或者 App 就是通过连线加上拖拽并编写少量的脚本的方式实现的，这种方式优点很明显，就是做东西很快也简单也很快。

缺点在我们刚接触到 Unity 的时候可能感受不到，笔者在这里简单列出来一下。

1. 难维护
2. 团队难以协作
3. 随着项目增长很快就发现新增功能比较困难（可扩展性差）

这三个缺点笔者在这里简单解释一下

1. 难维护

由于拖拽的时候，对象之间的引用关系是随机的没有规律可循的，所以当时间久了或者比人来接手次项目时要理清其中的逻辑所花费的时间会很多。

1. 团队难以协作

很多的拖拽操作和逻辑是在场景中操作的，并且也会记录到场景里，所以在使用 git 或者 svn 或者官方的 plastics 的时候比较难以管理，容易造成冲突。

1. 随着项目增长很快就发现新增功能比较困难（可扩展性差）

这个原因和第一条一样，拖拽的时候，对象之间的引用关系是随机的没有规律可循的，这样会造成大量的耦合（牵一发动全身），后边当扩展功能的时候，花的时间越来越久。

而接下来的架构演化的方向就是针对以上的这些缺点针对性地进行解决。

OK，课程的内容就这些。

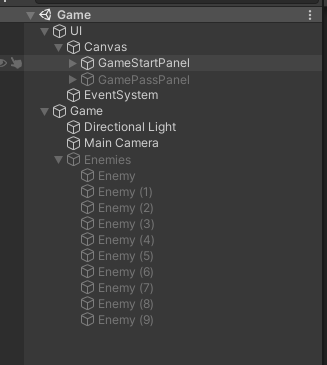
接下来笔者来解答一个问题：无架构的项目要完全杜绝吗？

答：不用完全杜绝，笔者目前是在家工作，有的时候能接到一些学生的毕业设计代做的单子，这些单子的特点就是体量非常小，基本上 1 ~ 3 天都能搞定，这个时候笔者就会不考虑什么架构的问题，直接上手做就可以，笔者管这种项目叫做一次性项目，也就是做了一次之后以后不用考虑维护啥的，当然了，凭着笔者多年的编码习惯，无架构的项目也会写得很清晰。但是当超过一定规模的项目一定要有一些架构设计比如 MVC、单例 等等，不然会项目乱到崩溃，bug 频出。

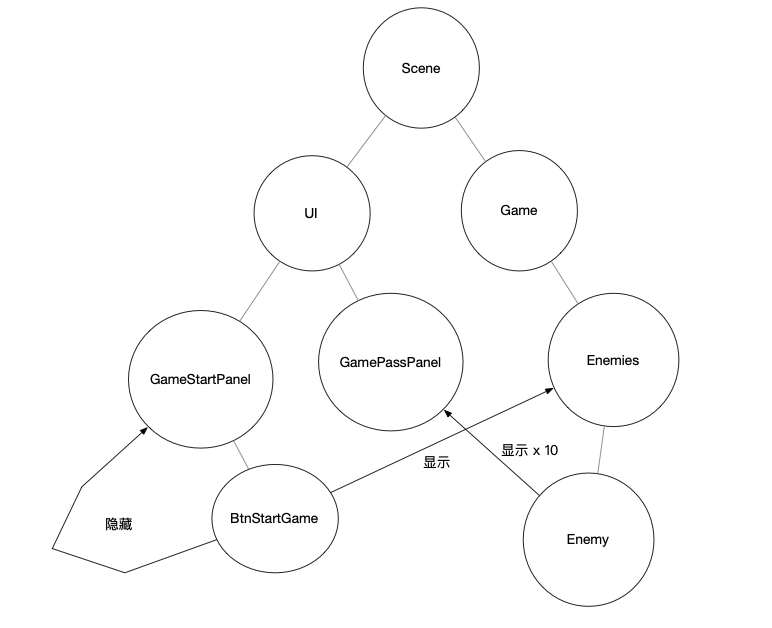
我们下一趟课再见，拜拜。

# 4. 第四课 对象之间的交互 和 模块化

在上一堂课，笔者整理了一下《点点点》的场景结构，如下：



补充了一张元素之间的调用关系图，如下：



并且分析了，一次性项目的优缺点，如下：  
优点：快、简单  
缺点：

* 难维护
* 难协作
* 难扩展（新增功能）

在这堂课，我们接着继续我们的演化过程。

一次性项目的缺点有三个：难维护、难协作、难扩展

而解决这三个的问题的途径只有一个，就是我们耳朵都快听出来茧子的低耦合、高内聚。

什么是耦合、内聚？

按照笔者的理解，耦合就是对象、类的双向引用、循环引用，内聚就是同样类型的代码放在一起。

具体详尽的解释或者严谨的解释请大家自行查询。

那么低耦合就是要尽量少地出现对象类的双向引用、循环引用。  
高内聚就是尽量地把同类的逻辑尽可能放在一起。

而笔者认为耦合、内聚这样的术语太过抽象太过严谨，而且太泛泛了，范围太广了。

所以在框架或者架构层面，我们只要关注两个问题就可以了，对象之间要如何交互？（低耦合），如何做到模块化（高内聚）。

对象之间的交互 和 模块化，是整个框架搭建系列的终极问题，是笔者自己总结出来的。

对象之间的交互一般有三种

* 方法调用，例如：A 调用 B 的 SayHello 方法
* 委托或者回调，例如：界面监听子按钮的点击事件
* 消息或事件，例如：服务器像客户端发送通知

模块化也一般有三种

* 单例，例如：Manager Of Managers
* IOC，例如：Extenject、uFrame 的 Container、StrangeIOC 的绑定等等
* 分层，例如：MVC、三层架构、领域驱动分层等等

当然可以做模块化的还有 Entity Component、门面模式 等等，但是笔者只介绍以上的三种，因为了解以上的三种足够了，其他的也很容易上手。

OK，现在引出来了 对象之间的交互 和 模块化 这两个概念。

我们来看看当前的项目在对象之间的交互和模块化方面是怎么样的。

首先对象之间的交互方面只使用了方法调用。

在模块化方面，没有用任何的模块化方式。

那么接下来，我们来重点方法调用这种对象之间的交互方式。

新手大部分都是先接触的对象之间的交互方式是方法调用，因为简单。方法调用是所有语言都会支持的交互方式（废话）。

但是在这里，我们考虑的角度有一点不一样，要成功完成一次方法调用，那么需要调用方能够获取到被调用方。

比如 A 对象想要调用 B 对象的 SayHello 方法，代码如下:

public class A  
{  
 public B B;  
   
 void Start()  
 {  
 B.SayHello();  
 }  
}

那么要确保 A 能够访问到 B 对象。

也就是说方法调用的前提是 A 要引用 B 对象才能调用 B 的 SayHello 方法。

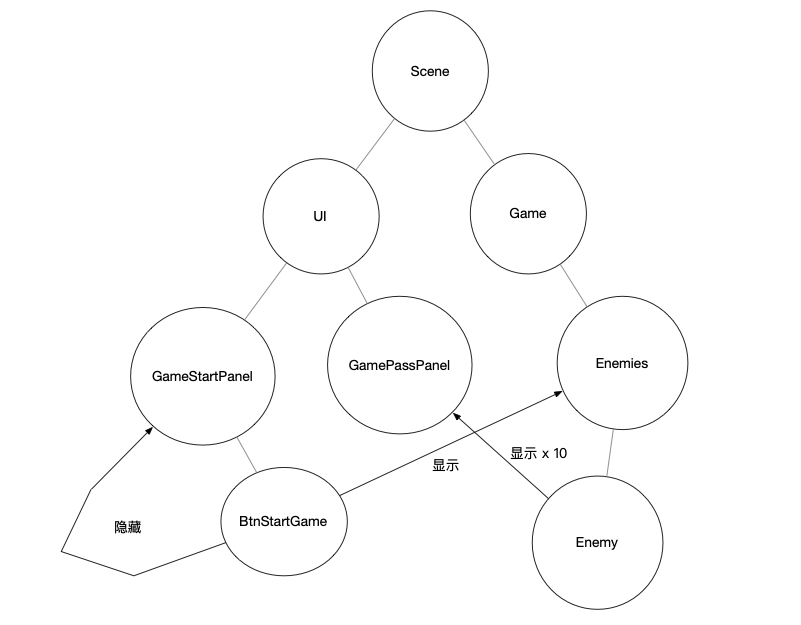
总结如下：

对象之间的交互

* 方法调用
  + A 需要持有 B 才能调用 B 的方法
* 委托 & 回调
  + ？
* 消息 & 事件
  + ？

由于方法调用，需要 A 持有 B，那么方法调用必然会造成至少单向引用的关系，而笔者在课程的开始说过，耦合就是双向引用或循环引用。

那么越多无限制地使用方法调用，耦合的概率就会越高，那么什么是无限制地使用方法调用呢？我们做的一次性的项目就是，对象和对象之间的关系是随机的，我们再看下下图。



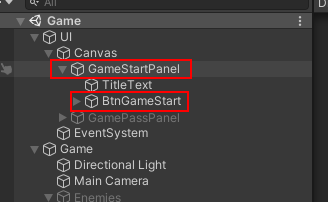
我们把图中的调用关系列出来，如下:

* BtnStartGame 调用 GameStartPanel 的 SetActive 方法
* BtnStartGame 调用 Enemies 的 SetActive 方法
* Enemy 调用 GamePassPanel 的 SetActive 方法

这三条调用关系每一条都是有很大问题的。

首先第一条：BtnStartGame 调用 GameStartPanel 的 SetActive 方法

我们先看下 BtnStartGame 和 GameStartPanel 的物理关系，如下：

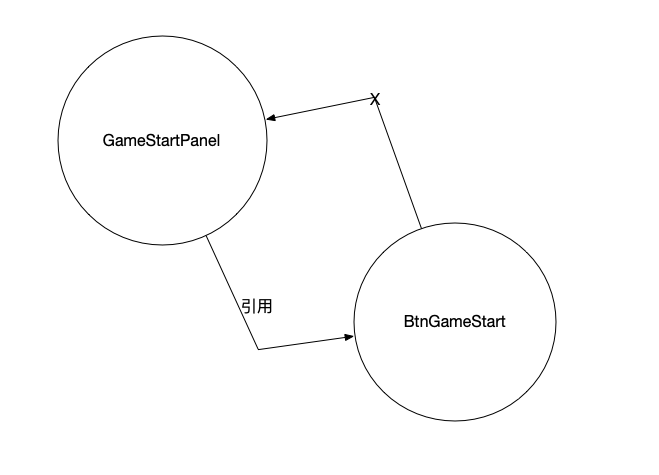


可以看到，BtnStartGame 是 GameStartPanel 的子节点，所以它俩是父子关系。

笔者在课程的开始说过，耦合就是双向引用或循环引用，所以父子节点之间的关系也不能是双向引用或循环引用。

那么既然父子节点不能耦合，那么想要引用，必然只能是单向的，而这里的共识就是 父节点可以引用子节点，但是子节点不能引用父节点。

如下图所示：

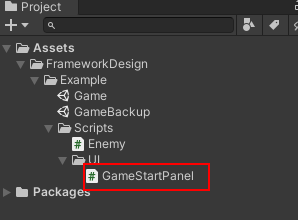


这是一个共识。

那么如果子节点想要调用父节点的方法怎么办？

答案很简单，就是使用我们对象之间的交互中的第二种方式，委托或回调。

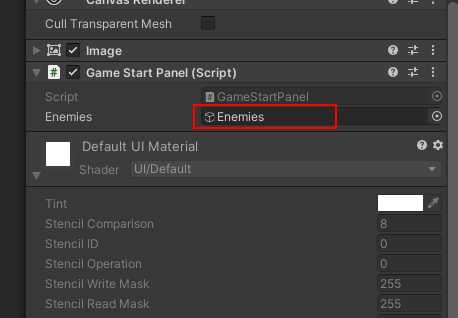
我们来尝试实现一下，首先我们为 GameStartPanel GameObject 创建一个脚本，叫做 GameStartPanel，所在位置如下：



代码如下：

using UnityEngine;  
using UnityEngine.UI;  
  
namespace FrameworkDesign.Example  
{  
 public class GameStartPanel : MonoBehaviour  
 {  
 /// <summary>  
 /// Enemy 的父节点  
 /// </summary>  
 public GameObject Enemies;  
   
 void Start()  
 {  
 transform.Find("BtnGameStart").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 gameObject.SetActive(false);  
 Enemies.SetActive(true);  
 });  
 }  
 }  
}

然后 我们在 GameStartPanel 上挂上这个脚本，并把 Enemies GameObject 拖到 此组件的 Enemies 上，如下：

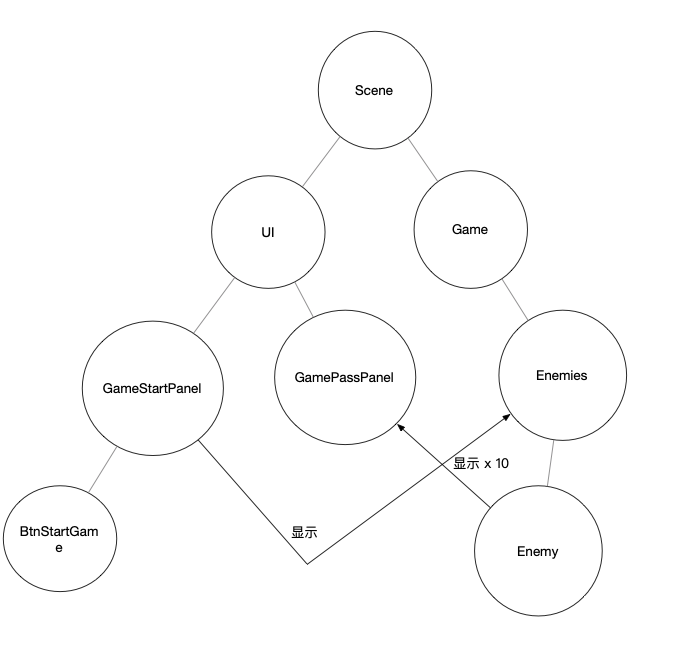


我们运行一下游戏，看下结果。

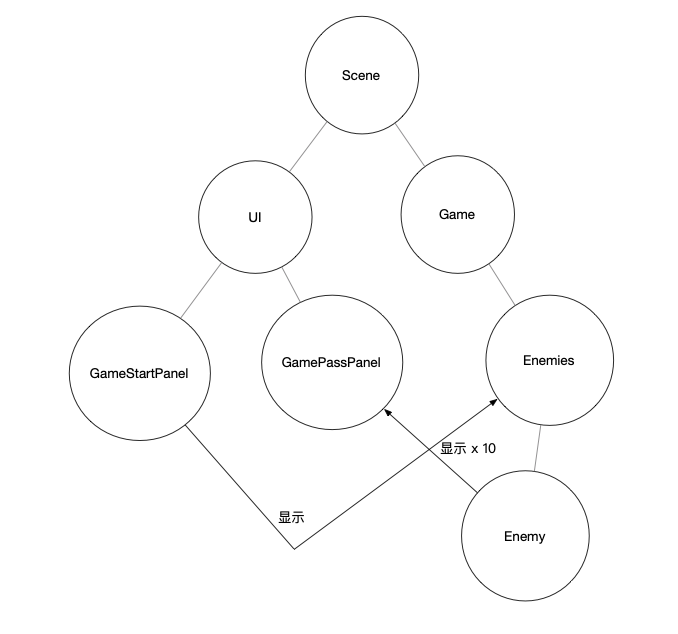


OK，没什么问题，说明改动成功。

接下来，我们更新以下 引用关系图，如下：



OK，这里 BtnStartGame ，可以清除掉了，因为它仅仅只是一个 UI 组件，并没有承担业务逻辑，清除之后如下：



OK，这样第一个引用关系的问题就处理完了。

简单总结下：

* 父节点可以引用子节点，子节点要调用父节点的方法时，可以用事件或委托。

再总结一下对象之间的交互：  
对象之间的交互

* 方法调用
  + A 需要持有 B 才能调用 B 的方法
* 委托 & 回调
  + A 需要持有 B 才能注册 B 的委托
* 消息 & 事件
  + ？

委托的话一般会造成单向引用，单向引用是弱耦合关系，所以危害比较少。但是要注意委托 & 回调尽量避免嵌套调用，因为嵌套调用容易造成冲击波（意大利面条）代码，如下：

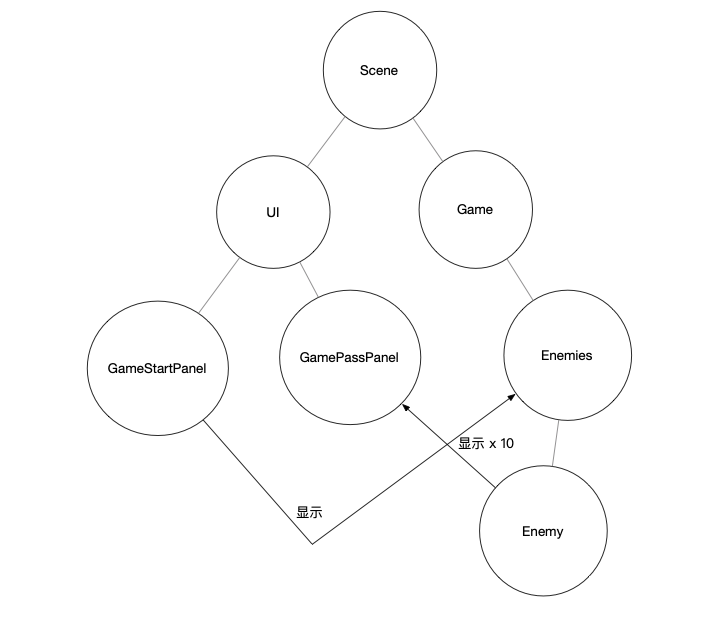
void Start()  
{  
 Http.Get("abc.com",(result)=>{  
 Http.Get("abc.com",(result1)=>{  
 Http.Get("abc.com",(result2)=>{  
 Http.Get("abc.com",(result3)=>{  
 for(int i = 0 ;i < 10 ;i ++)  
 {  
 for(int j = 0;j < 10 ;j ++)  
 {  
 // 冲击波本波  
 }  
 }  
 })  
 })   
 })   
 })  
}

* BtnStartGame 调用 GameStartPanel 的 SetActive 方法
* BtnStartGame 调用 Enemies 的 SetActive 方法
* Enemy 调用 GamePassPanel 的 SetActive 方法

OK，接下来，我们处理第二调用关系：BtnStartGame 调用 Enemies 的 SetActive 方法

由于上边的改动，现在 BtnStartGame 不调用 Enemies 的 SetActive 方法了，变成了 GameStartPanel 调用了 Enemies 的 SetActive 方法了，当然这也有问题。

我们看关系图，如下：



我们发现，GameStartPanel 的父节点是 UI，而 Enemies 的父节点是 Game，说明 GameStartPanel 和 Enemies 是不同业务类型的对象，GameStartPanel 负责 UI 相关的逻辑，Enemies 则是负责游戏相关逻辑，而不同业务模块之间的对象的交互，叫做跨模块的对象交互。

在很多情况下，一个游戏会由一个团队去开发完成，那么说到团队就必然要说到分工问题，而在分工的时候，往往会把 UI 的制作和游戏性（Gameplay) 逻辑分给不同的人来负责，那么这个时候，直接跨模块调用方法是很容易造成问题的。

所以方法调用行不通，委托能不能行得通？

也行不通，因为委托注册的前提也是要持有对象，最好的方式就是没有任何引用关系。

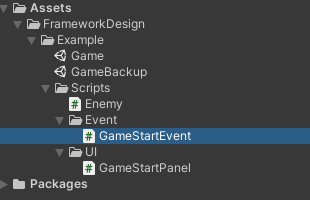
而使用对象之间的交互方式中的 消息&事件 能够满足要求。

我们尝试在项目中引入事件，这个事件叫做游戏开始事件，即 GameStartEvent，代码如下:

using System;  
  
namespace FrameworkDesign.Example  
{  
 public static class GameStartEvent  
 {  
 private static Action mOnEventTrigger;  
  
 /// <summary>  
 /// 注册事件  
 /// </summary>  
 public static void Register(Action onEvent)  
 {  
 mOnEventTrigger += onEvent;  
 }  
  
 /// <summary>  
 /// 注销事件  
 /// </summary>  
 /// <param name="onEvent"></param>  
 public static void UnRegister(Action onEvent)  
 {  
 mOnEventTrigger -= onEvent;  
 }  
  
 /// <summary>  
 /// 触发事件  
 /// </summary>  
 public static void Trigger()  
 {  
 mOnEventTrigger?.Invoke();  
 }  
  
 }  
}

代码不难，就是一个静态类，封装了一个委托而已。

代码所在的位置如下：

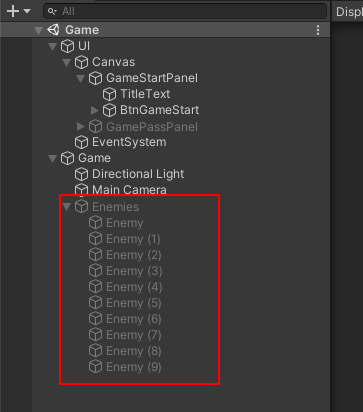


接下来我们在项目中使用这个事件。

我们修改 GameStartPanel 脚本的代码为如下：

using UnityEngine;  
using UnityEngine.UI;  
  
namespace FrameworkDesign.Example  
{  
 public class GameStartPanel : MonoBehaviour  
 {  
 /// <summary>  
 /// Enemy 的父节点  
 /// </summary>  
 public GameObject Enemies;  
  
 void Start()  
 {  
 transform.Find("BtnGameStart").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 gameObject.SetActive(false);  
  
 // 触发事件  
 GameStartEvent.Trigger();  
 });  
 }  
 }  
}

现在触发事件的代码有了，接下来要写处理事件的代码。

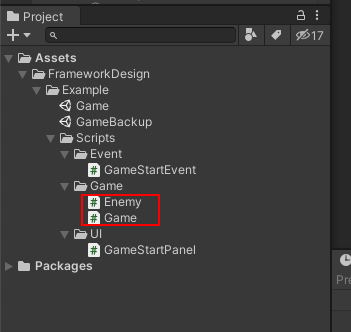
由于事件的注册需要在 Awake 或者 Start 里进行，而接收方 Enemies 的初始状态为不显示状态，如下所示：  


不显示状态没有机会调用 Awake 和 Start，所以我们在 Enemies 的父节点 Game 上挂一个脚本，这个脚本就叫做 Game，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 }  
  
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 }  
 }  
}

代码很简单。

代码所在的位置如下



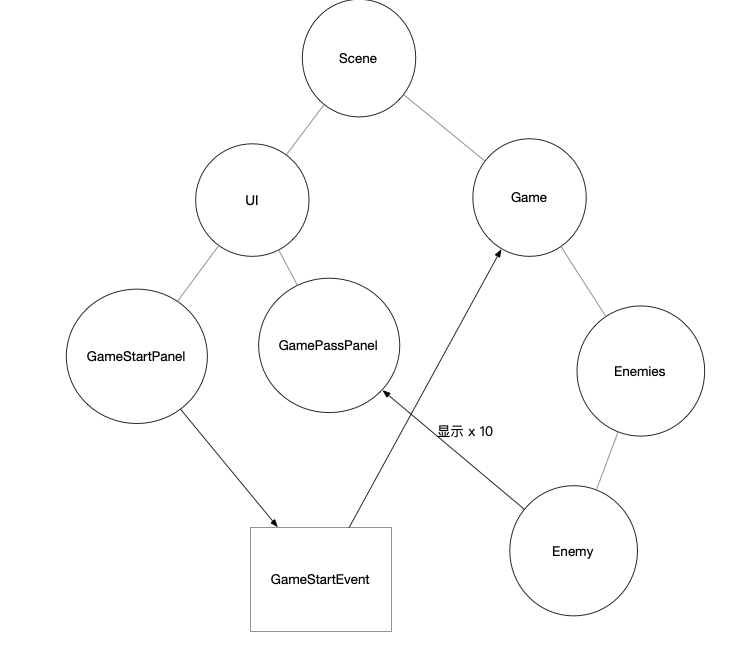
笔者顺便把 Enemy 脚本也放到了 Game 目录下。

将此脚本挂在 Game GameObject 上，运行 Unity 结果如下：



逻辑没有问题。

我们再更新以下关系图，如下：



OK，这样 UI 模块和 Game 模块通信的时候只要相关的负责人约定一下事件就可以了。

这样，第二个调用关系就处理好了，如下：

* BtnStartGame 调用 GameStartPanel 的 SetActive 方法
* BtnStartGame 调用 Enemies 的 SetActive 方法
* Enemy 调用 GamePassPanel 的 SetActive 方法

我们总结一下 对象之间的交互方法，如下：  
 对象之间的交互

* 方法调用
  + A 需要持有 B 才能调用 B 的方法
* 委托 & 回调
  + A 需要持有 B 才能注册 B 的委托
* 消息 & 事件
  + A 不需要持有 B

最后我们剩下一个调用关系，即 Enemy 调用 GamePassPanel 的 SetActive 方法

实际上第三个调用关系的问题和第二个调用关系的问题是一样的，都是跨模块的通信。

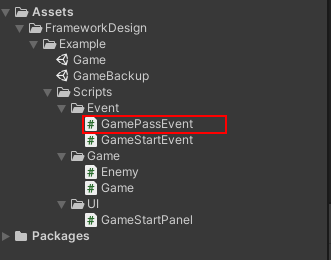
我们只要按照同样的方式处理即可。

先创建一个事件，叫做 GamePassEvent，代码如下：

using System;  
  
namespace FrameworkDesign.Example  
{  
 public static class GamePassEvent  
 {  
 private static Action mOnEventTrigger;  
  
 /// <summary>  
 /// 注册事件  
 /// </summary>  
 public static void Register(Action onEvent)  
 {  
 mOnEventTrigger += onEvent;  
 }  
  
 /// <summary>  
 /// 注销事件  
 /// </summary>  
 /// <param name="onEvent"></param>  
 public static void UnRegister(Action onEvent)  
 {  
 mOnEventTrigger -= onEvent;  
 }  
  
 /// <summary>  
 /// 触发事件  
 /// </summary>  
 public static void Trigger()  
 {  
 mOnEventTrigger?.Invoke();  
 }  
  
 }  
}

代码和 GameStartEvent 很像。

代码所在位置如下：



在 Enemy 脚本中触发此事件，如下：

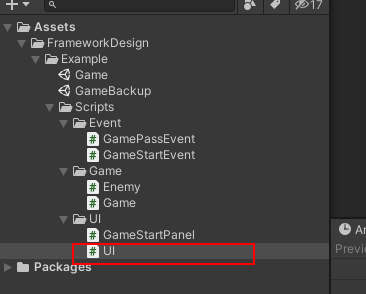
using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 击杀数量   
 /// </summary>  
 private static int mKillCount = 0;  
   
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
  
 mKillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (mKillCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

代码中，笔者去掉了用于存储 GamePassPanel 的成员变量。

同样，我们需要在 UI GameObject 上挂一个 UI 脚本，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class UI : MonoBehaviour  
 {  
 void Start()  
 {  
 GamePassEvent.Register(OnGamePass);  
 }  
  
 private void OnGamePass()  
 {  
 transform.Find("Canvas/GamePassPanel").gameObject.SetActive(true);  
 }  
   
 private void OnDestroy()  
 {  
 GamePassEvent.UnRegister(OnGamePass);  
 }  
 }  
}

代码所在的位置如下：



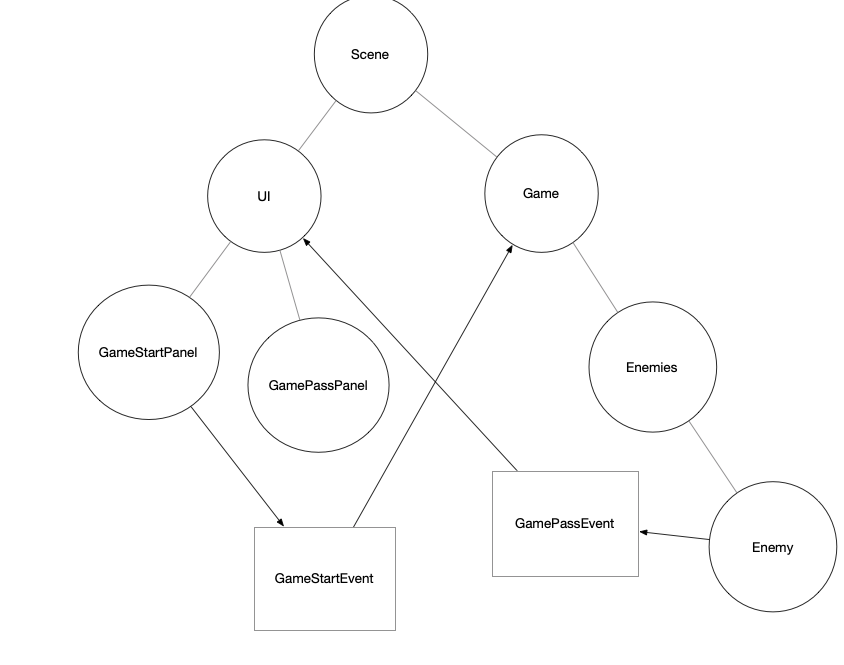
OK，这样我们就处理掉了第三个调用的问题，如下：

* BtnStartGame 调用 GameStartPanel 的 SetActive 方法
* BtnStartGame 调用 Enemies 的 SetActive 方法
* Enemy 调用 GamePassPanel 的 SetActive 方法

我们运行下游戏，结果如下：



我们再更新下关系图，如下：



OK，到此对象之间的交互的三种方式我们都接触到了。

这里总结一下：

1. 子节点通知父节点用委托或事件
2. 父节点调用子节点可以直接方法调用
3. 跨模块通信用事件
4. 耦合就是双向引用或循环引用

对象之间的交互一般有三种

* 方法调用，例如：A 调用 B 的 SayHello 方法
* 委托或者回调，例如：界面监听子按钮的点击事件
* 消息或事件，例如：服务器像客户端发送通知

模块化也一般有三种

* 单例，例如：Manager Of Managers
* IOC，例如：Extenject、uFrame 的 Container、StrangeIOC 的绑定等等
* 分层，例如：MVC、三层架构、领域驱动分层等等

对象之间的交互的引用关系

* 方法调用
  + A 需要持有 B 才能调用 B 的方法
* 委托 & 回调
  + A 需要持有 B 才能注册 B 的委托
* 消息 & 事件
  + A 不需要持有 B

OK，这堂课的内容就这些。

# 5. 第五课 表现和数据要分离

在上一堂课，我们通过架构这个概念慢慢引出来了项目结构、场景结构等概念，然后整理了一下场景结构，然后用树结构的方式去整理场景节点。

在之后提出了一次性项目的优缺点：

* 优点：
  + 快、简单
* 缺点
  + 难维护
  + 难协作
  + 难扩展

然后引出了对象之间的通信这样的概念，总结如下：

* 方法调用，例如：A 调用 B 的 SayHello 方法
* 委托或者回调，例如：界面监听子按钮的点击事件
* 消息或事件，例如：服务器像客户端发送通知

然后通过用这三种交互方式解决了《点点点》项目中的不良的引用关系。

最后得出一些使用方法，如下:

* 子节点通知父节点用委托或事件
* 父节点调用子节点可以直接方法调用
* 跨模块通信用事件
* 耦合就是双向引用或循环引用
* 对象之间的交互一般有三种

对象之间的交互方式所需的引用关系：

* 方法调用
  + A 需要持有 B 才能调用 B 的方法
* 委托 & 回调
  + A 需要持有 B 才能注册 B 的委托
* 消息 & 事件
  + A 不需要持有 B

简单在文中也说了一下常见的模块化也有三种:

* 单例，例如：Manager Of Managers
* IOC，例如：Extenject、uFrame 的 Container、StrangeIOC 的绑定等等
* 分层，例如：MVC、三层架构、领域驱动分层等等

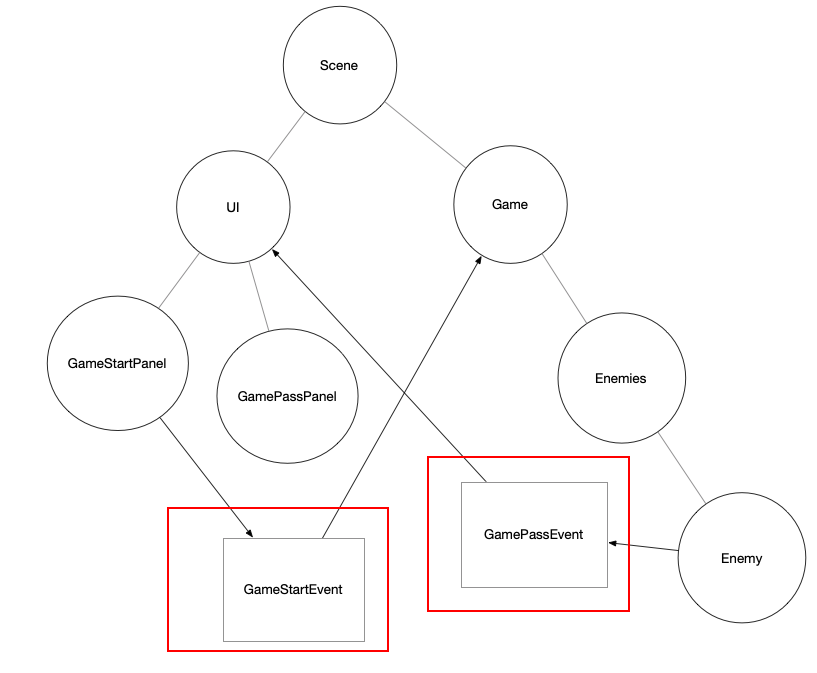
而对象之间的交互 和 模块化这两个概念是演化子 低耦合、高内聚 而来的，这是笔者自己总结的。

耦合就是对象双向引用、循环引用，内聚就是同类逻辑或代码放在一起。

OK，这些是上堂课介绍的概念，内容有点多，但是理论的基本概念也就这些了，希望大家反复琢磨这些。

在这堂课，我们接着往下演化《点点点》这个项目的架构。

在上堂课，我们在《点点点》项目中创建了两个事件，GameStartEvent 和 GamePassEvent，如下图所示：



通过这两个事件，解除掉了项目中几乎所有的不良的引用关系。

但是这两个事件的实现有一些问题，我们看下这两个事件的实现代码，如下：

GameStartEvent.cs

using System;  
  
namespace FrameworkDesign.Example  
{  
 public static class GameStartEvent  
 {  
 private static Action mOnEventTrigger;  
  
 /// <summary>  
 /// 注册事件  
 /// </summary>  
 public static void Register(Action onEvent)  
 {  
 mOnEventTrigger += onEvent;  
 }  
  
 /// <summary>  
 /// 注销事件  
 /// </summary>  
 /// <param name="onEvent"></param>  
 public static void UnRegister(Action onEvent)  
 {  
 mOnEventTrigger -= onEvent;  
 }  
  
 /// <summary>  
 /// 触发事件  
 /// </summary>  
 public static void Trigger()  
 {  
 mOnEventTrigger?.Invoke();  
 }  
  
 }  
}

GamePassEvent.cs

using System;  
  
namespace FrameworkDesign.Example  
{  
 public static class GameStartEvent  
 {  
 private static Action mOnEventTrigger;  
  
 /// <summary>  
 /// 注册事件  
 /// </summary>  
 public static void Register(Action onEvent)  
 {  
 mOnEventTrigger += onEvent;  
 }  
  
 /// <summary>  
 /// 注销事件  
 /// </summary>  
 /// <param name="onEvent"></param>  
 public static void UnRegister(Action onEvent)  
 {  
 mOnEventTrigger -= onEvent;  
 }  
  
 /// <summary>  
 /// 触发事件  
 /// </summary>  
 public static void Trigger()  
 {  
 mOnEventTrigger?.Invoke();  
 }  
  
 }  
}

可以发现，这两个代码几乎是重复的，除了类名不一样之外。

那么作为框架搭建课程，这样的代码是肯定不允许存在的，所以我们需要做一步优化。

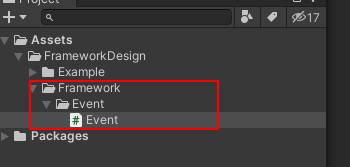
两个类的实现代码完全一样，就只有类名或类型不一样的时候，而且还需要不断**扩展**（未来会增加各种事件）的时候，这时候就用 泛型 + 继承 来提取，继承解决扩展的问题，泛型解决实现代码一致，类不一致的问题，这是一个重构技巧。

我们实现一个新的类，叫做 Event 代码如下:

using System;  
  
namespace FrameworkDesign  
{  
 public class Event<T> where T : Event<T>  
 {  
 private static Action mOnEventTrigger;  
  
 /// <summary>  
 /// 注册事件  
 /// </summary>  
 public static void Register(Action onEvent)  
 {  
 mOnEventTrigger += onEvent;  
 }  
  
 /// <summary>  
 /// 注销事件  
 /// </summary>  
 /// <param name="onEvent"></param>  
 public static void UnRegister(Action onEvent)  
 {  
 mOnEventTrigger -= onEvent;  
 }  
  
 /// <summary>  
 /// 触发事件  
 /// </summary>  
 public static void Trigger()  
 {  
 mOnEventTrigger?.Invoke();  
 }  
 }  
}

这里代码有点难，吃力的童鞋，可以自行写个 Example，来掌握下泛型、泛型约束这两个概念。

代码所在的位置如下：



然后接下来，我们用 Event 类重写 GameStartEvent 和 GamePassEvent 的代码，代码如下：

GameStartEvent.cs

namespace FrameworkDesign.Example  
{  
 public class GameStartEvent : Event<GameStartEvent>  
 {  
 }  
}

GamePassEvent.cs

namespace FrameworkDesign.Example  
{  
 public class GamePassEvent : Event<GamePassEvent>  
 {  
 }  
}

代码一下子少了很多，这样我们扩展一个事件的时候就非常容易了，主要创建个类，继承 Event 就可以。

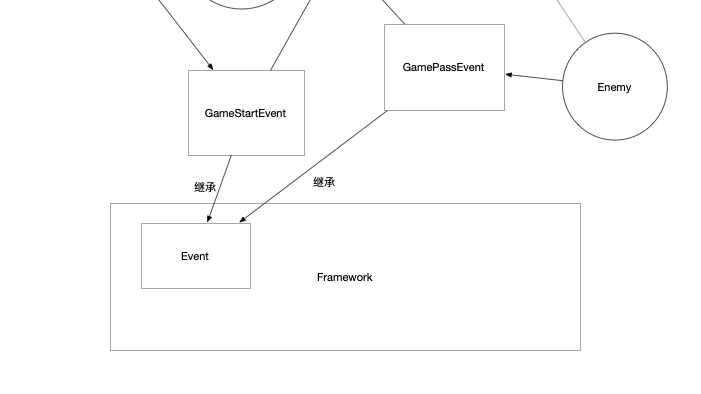
接下来我们看下，游戏的运行结果，看看有没有受到影响，结果如下：



没有被影响。

好了现在项目没啥问题了，我们通过引入事件这个概念，让项目更容易维护、跟容易协作。

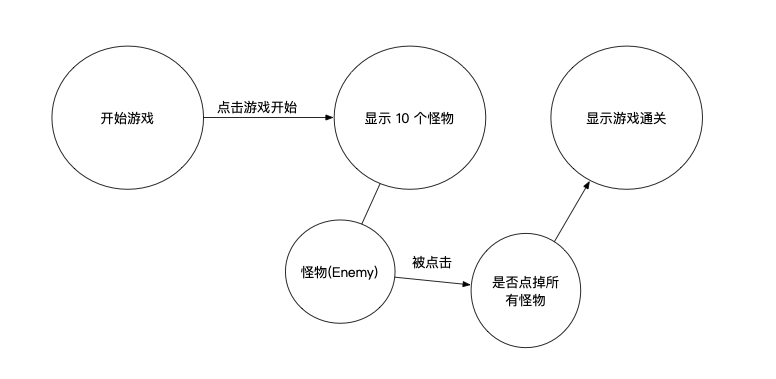
更新下引用关系图，如下：



（这么画其实很不严谨，但是目前内容不多一张图就够了，就先不升级新的图了）

OK，接下来，我们接着分析《点点点》游戏的其他问题。

我们看下游戏的流程图，如下：



我们尝试分析一下游戏中所需要的数据部分，首先游戏需要 10 个怪物，这个 10 个怪物的数量是第一个游戏数据，这个游戏的数据决定了通关条件，比如点击 10 次才能通关，那么这里就有一个计数的逻辑，这部分逻辑我们写在了 Enemy 里，代码如下:

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 击杀数量   
 /// </summary>  
 private static int mKillCount = 0;  
   
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
  
 mKillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (mKillCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

这里有一个比较严重的问题，就是游戏的核心数据与表现的部分没有分离，而是写在了一起，目前看起来没啥问题，但是未来，我们可能给这个游戏加上分数、最高分、金币、拥有的道具等功能，这些功能所需的数据在大多数情况下需要在多个场景、界面、游戏物体之间是**共享的**，这些数据不但需要在空间上共享，还需要再时间上也需要共享（需要存储起来），所以在这里，开发者的共识就是把数据的部分会抽离出来，单独放在一个地方进行维护，而比较常见的开发架构就是使用 MVC 的开发架构，我们先不用 MVC 的开发架构，而只用 MVC 中的其中一个概念，就是 Model。

Model 就是管理数据、存储数据，管理数据就是可以通过 Model 对象或类可以对数据进行增删改查，有的时候还可以进行存储。

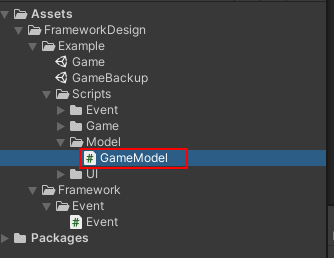
这里我们创建一个叫做 GameModel 的类，代码如下:

namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public static int KillCount = 0;  
 }  
}

代码非常简单，这里我们顺便再加上未来可能要加的金币、分数、最高分等数据，代码如下：

namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public static int KillCount = 0;  
  
 public static int Gold = 0;  
  
 public static int Score = 0;  
  
 public static int BestScore = 0;  
 }  
}

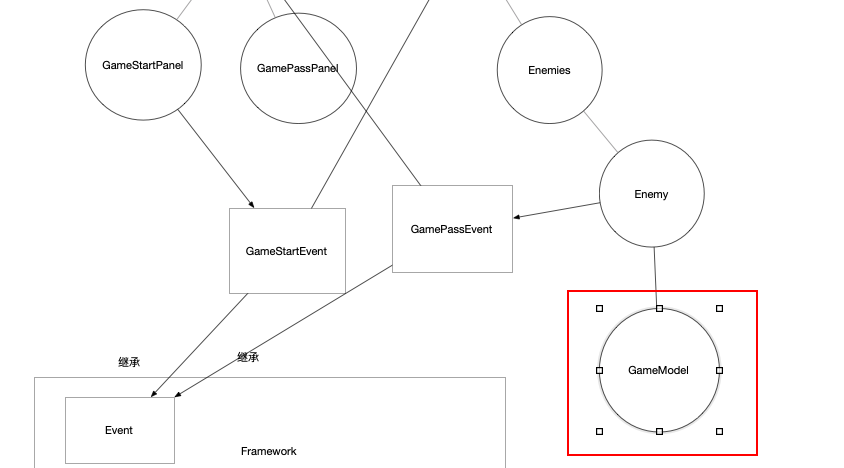
代码所在位置如下：



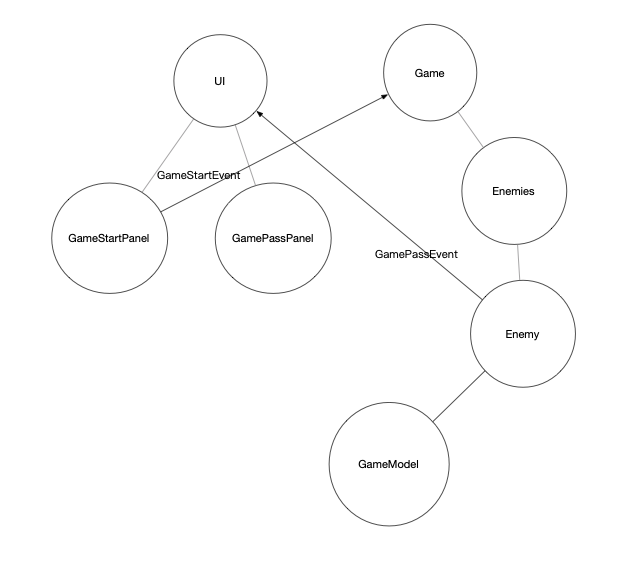
OK，接下来我们把 KillCount 数据应用到 Enemy 中，代码如下：

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
  
 GameModel.KillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

这样我们的数据就改好了，现在最起码游戏里的数据和表现是分开的，引用关系图如下：



现在的引用关系图有点 Hold 不住了，我们重新画一张，把树中的不是关键的节点都去掉，然后框架的引用也去掉，把事件也去掉换成用连线表示，新的图如下



OK，新的图又回归了简洁。

我们再看下 Enemey.cs 代码，如下：

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
  
 GameModel.KillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

我们的游戏判断是否结束的逻辑写在了 Enemy 里，这个也是由很大的问题的。

这个问题是不该在这里实现的逻辑，实现在了这里，为什么这么说呢？

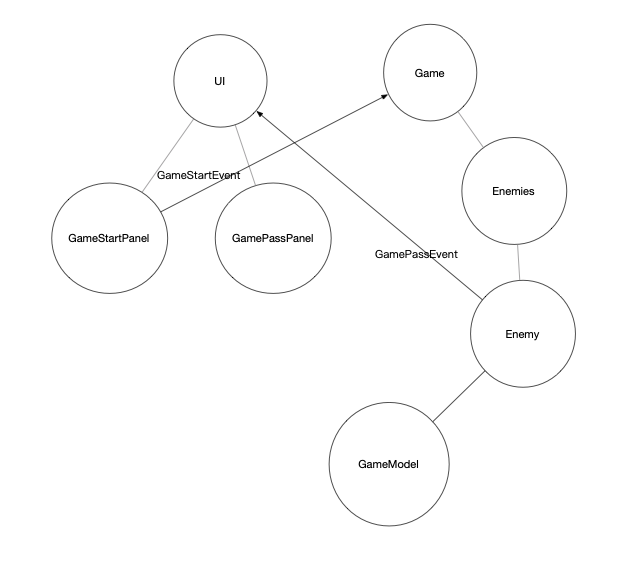
游戏通关的判断，其实是游戏的规则，那么判断一个游戏是否要通关是一个敌人该负责的事情么？ 打个比方，假如我们下一盘象棋，象棋的输赢条件是应该是棋子来判断么？显然不是，棋子没有那么智能，棋子的职责仅仅是被移动或者吃掉或者是传达信息而已，那么是谁来判断象棋的胜负条件呢？答案是双方玩家根据规则去判断胜负条件，而这个规则是属于一场游戏的，规则、游戏这些概念理解起来比较抽象，但是在我们项目中，确确实实存在一个游戏这样的概念，也就是 Game 类，代码如下:

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 }  
  
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 }  
 }  
}

这里是实现游戏结束判断条件的最佳位置，而不是 Enemy。如果在 Enemy 里实现游戏结束的判断，很不符合逻辑，不是自洽的，不是面向对象的，不是对真实世界的抽象描述。如果我们写的代码不是符合逻辑的抽象，那么别人看这个代码的时候就会增加理解成本，也就会让项目更难以维护，记住这些代码以前笔者是怎么写的，以后要避免。

OK，接下来，我们把 Enemy 里的游戏结束的条件判断迁移到 Game 类里。

但是这里有很多阻碍，我们看下关系图，如下：



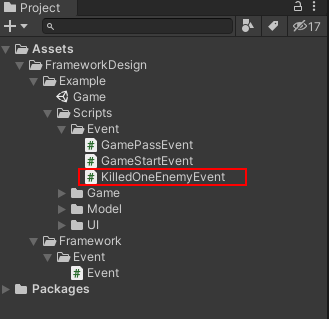
Enemy 每次被点击的时候，需要告诉 Game 类，Enemy 被点击了，而 Game 和 Enemy 隔着一个 Enemies，而且 Game 和 Enemy 是 1 对 10 的关系，这是一种子节点向父节点通信的例子，这种情况下可以用委托或者事件来实现，但是用委托不现实，如果想用委托 Game 或者 Enemies 就需要维护一个 Enemy 数组，所以用事件是最佳选择。

这里我们增加一个事件，名字叫做 KilledOneEnemyEvent，意思是击杀了一个敌人，代码如下：

namespace FrameworkDesign.Example  
{  
 public class KilledOneEnemyEvent : Event<KilledOneEnemyEvent>  
 {  
   
 }  
}

代码很简单。

代码所在的位置如下：



我们在 Enemy 中触发此事件，代码如下:

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 KilledOneEnemyEvent.Trigger();  
 }  
 }  
}

在 Game 类中监听，代码如下:

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 KilledOneEnemyEvent.Register(OnEnemyKilled);  
 }  
   
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
 private void OnEnemyKilled()  
 {  
 GameModel.KillCount++;  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 KilledOneEnemyEvent.UnRegister(OnEnemyKilled);  
  
 }  
 }  
}

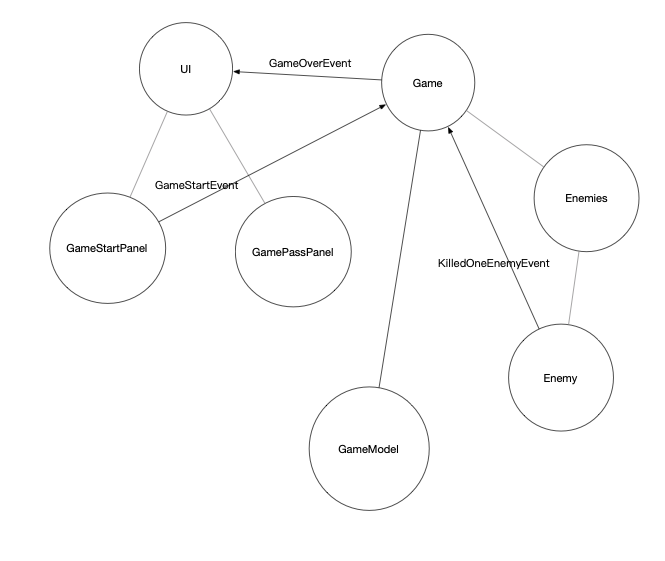
代码不难。

我们来运行一下游戏，结果如下：



游戏的逻辑没问题。

再跟新一下引用关系图，如下：



很清晰。

这样我们又多了一个事件，即 KilledOneEnemyEvent，而现在整个项目做到了正确的代码放到了正确的位置这一个原则，这一个原则更严谨的叫法就是单一职责原则，实际上代码对业务抽象得合理自然就符合单一职责原则了。

项目目前也有了一些架构的东西，比如表现和数据分离，单一职责原则，事件等，笔者也渐入佳境，哈哈哈。

到这里大家也应该掌握了对象之间交互的三种方式：方法调用、委托 & 回调、事件的用法，什么时候该用什么，这个是需要大家根据当时的条件和环境自己判断的，这也是一门艺术。

OK，在这堂课我们引出了 Model 这个概念，接下来我们要进入一个围绕 Model 学习的一个阶段。

这堂课的内容 就这些。

总结：

* 用泛型 + 继承 提取 Event 工具类
* 子节点通知父节点也可以用事件（根据情况）
* 表现和数据分离
* 正确的代码要放在正确的位置

我们下堂课继续，拜拜。

# 6. 第六课 交互逻辑 与 表现逻辑

先贴出来上一堂课的总结，如下：

总结：

* 用泛型 + 继承 提取 Event 工具类
* 子节点通知父节点也可以用事件（根据情况）
* 表现和数据分离
* 正确的代码要放在正确的位置

在上一堂课，我们使用了 Model 把游戏中的数据抽了出来，代码如下：

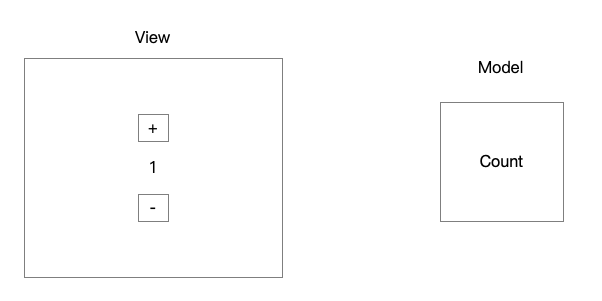
namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public static int KillCount = 0;  
  
 public static int Gold = 0;  
  
 public static int Score = 0;  
  
 public static int BestScore = 0;  
 }  
}

说到 Model 我们就不得不说到 MVC 架构模式了，MVC 想必大家耳熟能详，但是又会觉得有的时候写代码分不清这个代码应该写在 View 还是 Controller 还是 Model 里，造成这样的愿意是因为没有学习两个概念，学完了这两个概念，自然就能够用用好 MVC 了。

这两个概念是：

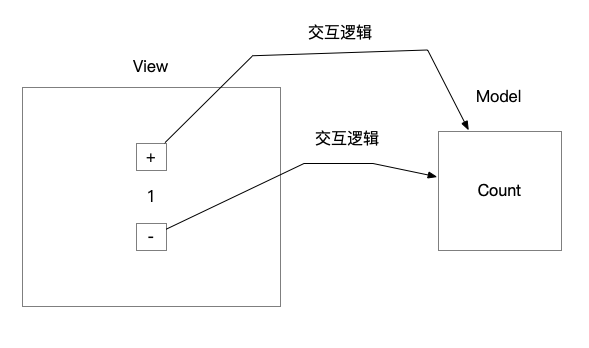
* 交互逻辑
* 表现逻辑

我们先看一张图，如下：



这是一个计数器 App，按下 + 号 数字就会递增，按下 - 号 数字就会递减，非常简单。那么这里的交互逻辑是什么呢？

答案是如下图:

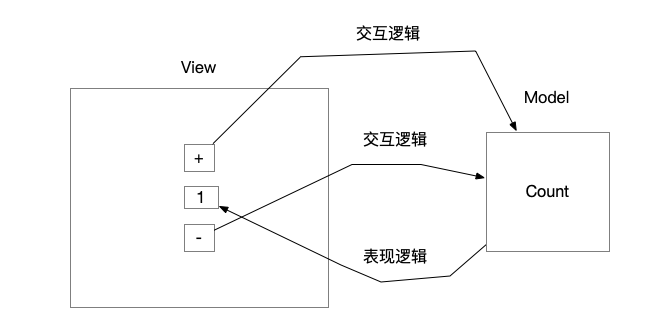


就是当用户按下 + 或者 - 时，程序会更改 Model 里的 Count 数据，这个就叫做交互数据。

用伪代码表述如下:

public class Controller   
{  
 void Start()  
 {  
 View.BtnAdd.onClick.AddListener(()=>{  
 // 交互逻辑  
 Model.Count++;  
 });  
   
 View.BtnSub.onClick.AddListener(()=>{  
 // 交互逻辑  
 Model.Count--;  
 });  
 }  
}

而表现逻辑的图，如下所示



当变更数据之后或者初始化时，从 Model 里查询数据显示到 View 上的，就是表现逻辑。

伪代码表示如下：

public class Controller   
{  
 void Start()  
 {  
 // 表现逻辑  
 View.CountText.text = Model.Count.ToString();  
   
 View.BtnAdd.onClick.AddListener(()=>{  
 // 交互逻辑  
 Model.Count++;  
   
 // 表现逻辑  
 View.CountText.text = Model.Count.ToString();  
 });  
   
 View.BtnSub.onClick.AddListener(()=>{  
 // 交互逻辑  
 Model.Count--;  
   
 // 表现逻辑  
 View.CountText.text = Model.Count.ToString();  
 });  
 }  
}

OK，到此交互逻辑和表现逻辑这两个概念应该解释清楚了，我们简单总结下：

* 交互逻辑：View -> Model
* 表现逻辑: Model -> View

从字面意义上也比较容易理解，一个 App 的交互，就是用户的输入和操作等等，这叫做交互逻辑，一个 App 的展现，一般都是展现数据的，展现数据的逻辑也叫做表现逻辑。

OK，关于这两个概念的介绍就说到这里。

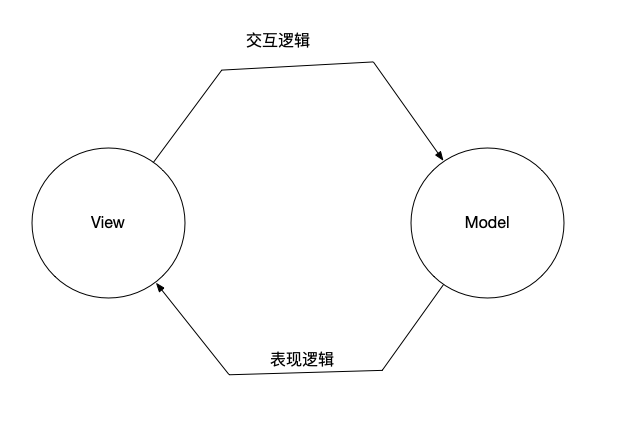
那么为什么要介绍这两个概念呢？

因为在很多时候，我们不会真的去用 MVC 开发架构，而是使用表现（View）和数据（Model）分离这样的思想，而我们只要知道 View 和 Model 之间有两种逻辑，即交互逻辑 和 表现逻辑，我们就不用管中间到底是 Controller、还是 ViewModel、还是 Presenter。只需要想清楚交互逻辑 和 交互逻辑如何实现的就可以了。

换句话说，一般 Controller 会负责两种逻辑，即交互逻辑和表现逻辑。这也是为什么 Controller 会写得越来越臃肿的原因，这是因为 Controller 负责了两种逻辑，里边拥有两种逻辑的代码，随着项目规模越来越大，自然就代码越来越多了。

而解决 Controller 臃肿问题的比较好的解决方法，就是引入 Command 这个概念，也就是命令，这个命令和命令模式的命令是一回事，不过这个笔者现在先不讲，因为关于交互逻辑和表现逻辑还有一些事情没讲完。

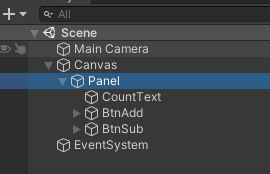
现在我们画一张图，来表达一下交互逻辑和表现逻辑，如下：



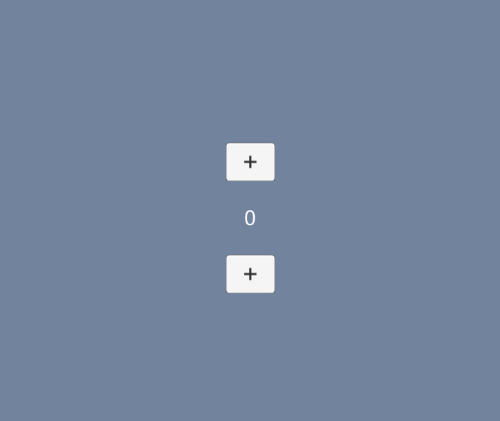
很清晰，OK，到这里，大家只要理解这两个概念就够了。

接下来笔者做一个小的练习，就是在项目中实现一下上边的计数器应用。

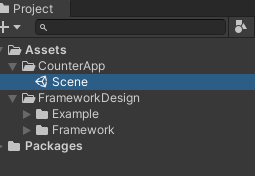
先创建一个场景，场景结构如下：



界面如下:



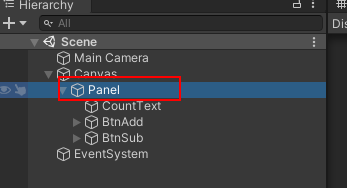
场景所在位置如下：



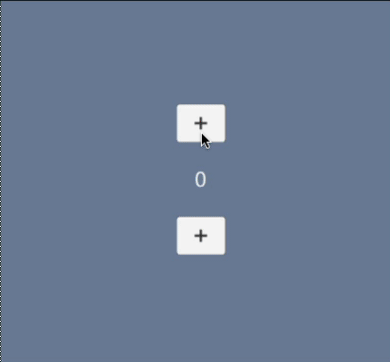
然后我们创建一个脚本，叫做 CounterViewController，代码如下:

using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count++;  
   
 // 表现逻辑  
 UpdateView();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count--;  
   
 // 表现逻辑  
 UpdateView();  
 });  
   
 // 表现逻辑  
 UpdateView();  
 }  
  
 void UpdateView()  
 {  
 transform.Find("CountText").GetComponent<Text>().text = CounterModel.Count.ToString();  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static int Count = 0;  
 }  
}

代码不难，这里要简单说一下，为什么要起名叫 ViewController，这是因为，Unity 里是比较难分 View 和 Controller 的，View 的话可以理解为感兴趣的组件的引用，比如这个示例中的 BtnSub，BtnAdd，CountText 等，而只要笔者继承 MonoBehaviour 就可以比较容易地直接获取到这些组件，所以单独创建一个 View 脚本来做这些会比较浪费，所以就叫做 ViewController 了，即，一个脚本由于是挂在 GameObject 上，所以他又有 View 的职责，也有 Controller 的职责。

OK，我们把这个脚本挂在 场景的 Panel 节点上，如下：  


然后运行 Unity 结果如下：



OK，很简单的一个练习，这个练习接下来还会用到。

这堂课的内容就这些，我们下堂课再见，拜拜。

总结：

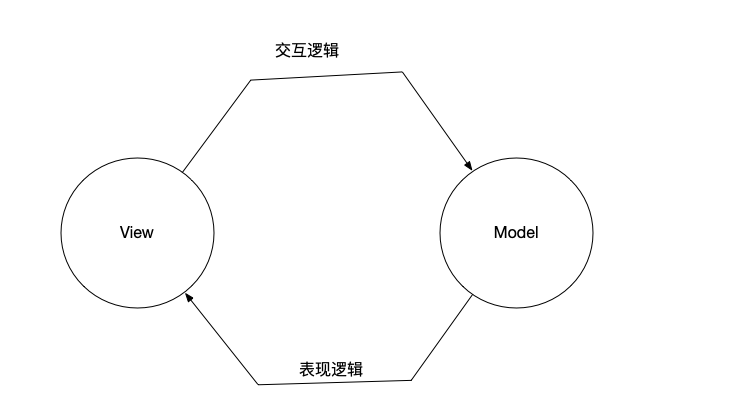
* 交互逻辑：View -> Model
* 表现逻辑: Model -> View

# 7. 第七课 表现逻辑优化-引入 BindableProperty

再上一堂课我们接触了两个概念，即 交互逻辑 和 表现逻辑 如下：

* 交互逻辑：View -> Model
* 表现逻辑: Model -> View

如下图所示：



CounterViewController 的代码如下：

using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count++;  
   
 // 表现逻辑  
 UpdateView();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count--;  
   
 // 表现逻辑  
 UpdateView();  
 });  
   
 // 表现逻辑  
 UpdateView();  
 }  
  
 void UpdateView()  
 {  
 transform.Find("CountText").GetComponent<Text>().text = CounterModel.Count.ToString();  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static int Count = 0;  
 }  
}

OK，我们了解了交互逻辑和表现逻辑，再加上我们之前我们了解的对象之间的交互的三种方式，来看看 View 和 Model 对象怎样交互比较好，或者说 交互逻辑 和 表现逻辑 怎样实现会比较好。

这里提醒一下各位童鞋，就是笔者前边介绍的理论概念虽然理解起来比较简单，但是这些概念在此课程的接下来的内容里会用的很重，就比如对象之间的交互的三种方式马上就要用起来了。

我们先看下表现逻辑。

在 CounterViewController 的代码里，表现逻辑是在交互逻辑完成之后主动调用的，代码如下：

using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count++;  
   
 // 表现逻辑  
 UpdateView();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count--;  
   
 // 表现逻辑  
 UpdateView();  
 });  
   
 // 表现逻辑  
 UpdateView();  
 }  
  
 void UpdateView()  
 {  
 transform.Find("CountText").GetComponent<Text>().text = CounterModel.Count.ToString();  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static int Count = 0;  
 }  
}

所以此时，表现逻辑是用的是对象之间的交互方式中的 方法调用方式，即 Controller 获取 Model 对象，然后把 Model 对象的 Count 数据显示个 View。

而接下来我们尝试用对象之间的交互的第二种方式来实现一下，也就是委托调用方式。

实际上 View 对象只需要监听 CounterModel 的数据变化就可以，我们先把 CounterModel 的代码改成如下；

public static class CounterModel  
 {  
 private static int mCount = 0;  
  
 public static Action<int> OnCountChanged ;  
   
 public static int Count  
 {  
 get => mCount;  
 set  
 {  
 if (value != mCount)  
 {  
 mCount = value;  
 OnCountChanged?.Invoke(value);  
 }  
 }  
 }  
 }

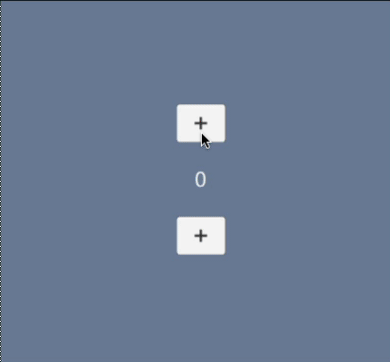
代码不难，增加了一个 OnCountChanged 委托，然后将 Count 变量变成了属性。

这样的话，在 ViewController 里只需要监听 OnCountChanged 委托就可以了。

我们把代码改成如下：

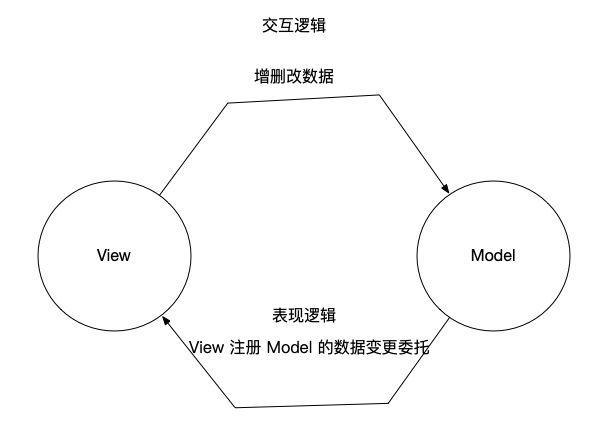
using System;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 CounterModel.OnCountChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count--;  
 });  
   
 OnCountChanged(CounterModel.Count);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newCount)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newCount.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 CounterModel.OnCountChanged -= OnCountChanged;  
 }  
 }  
  
 public static class CounterModel  
 {  
 private static int mCount = 0;  
  
 public static event Action<int> OnCountChanged ;  
   
 public static int Count  
 {  
 get => mCount;  
 set  
 {  
 if (value != mCount)  
 {  
 mCount = value;  
 OnCountChanged?.Invoke(value);  
 }  
 }  
 }  
 }  
}

代码不难，只是引入了对象之间交互的第二种：委托&回调 而已，我们看下游戏的运行结果，如下：



OK，试验完成。

我们画一下 View 和 Model 交互的图，如下：



接着看，对象之间的交互的第三种：事件&消息。

其实非常简单，我们把 Model 的代码改成如下：

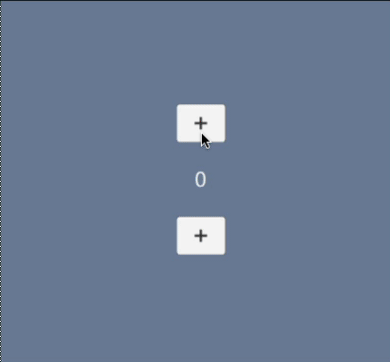
public static class CounterModel  
 {  
 private static int mCount = 0;  
   
 public static int Count  
 {  
 get => mCount;  
 set  
 {  
 if (value != mCount)  
 {  
 mCount = value;  
   
 // 触发事件  
 OnCountChangedEvent.Trigger();  
 }  
 }  
 }  
 }  
  
 // 事件定义  
 public class OnCountChangedEvent : Event<OnCountChangedEvent>  
 {  
   
 }

代码不难，就是把之前的委托删掉了，然后改成了用事件，这里事件发现一个缺陷，就是之前笔者设计的 Event 工具类，不支持携带参数，这个要在后边完善下。

我们接下来把 ViewController 的代码改成如下:

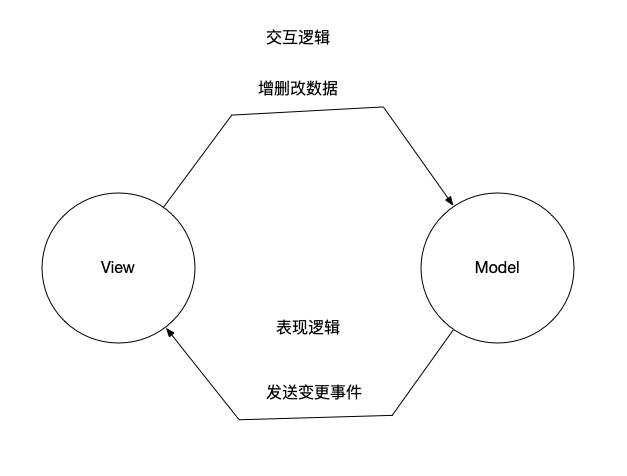
using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 OnCountChangedEvent.Register(OnCountChanged);  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count--;  
 });  
   
 OnCountChanged();  
 }  
  
 // 表现逻辑  
 private void OnCountChanged()  
 {  
 transform.Find("CountText").GetComponent<Text>().text = CounterModel.Count.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 OnCountChangedEvent.UnRegister(OnCountChanged);  
 }  
 }  
  
 public static class CounterModel  
 {  
 private static int mCount = 0;  
   
 public static int Count  
 {  
 get => mCount;  
 set  
 {  
 if (value != mCount)  
 {  
 mCount = value;  
   
 // 触发事件  
 OnCountChangedEvent.Trigger();  
 }  
 }  
 }  
 }  
  
 // 事件定义  
 public class OnCountChangedEvent : Event<OnCountChangedEvent>  
 {  
   
 }  
}

变化不是很多，运行 Unity 结果如下：



没有啥变化，说明功能是没问题的。

我们再画一下，View 和 Model 的交互图，如下：



OK，这样方法调用、委托&回调、事件&消息 我们都试过了，现在可以总结出来一点经验了。

通过对比发现，使用委托&回调 和 事件&消息 的方式可以让 Controller 的代码量更少，我们对比一下。

使用方法调用的方式：

transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count++;  
   
 // 表现逻辑  
 UpdateView();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 CounterModel.Count--;  
   
 // 表现逻辑  
 UpdateView();  
 });  
   
 // 表现逻辑  
 UpdateView();

每次按钮点击之后，都需要主动调用一下表现逻辑，而 CounterApp 的数据不多，只有一个 Count，当有更多数据的时候，这种方式会造成很多的代码调用，而且还容易造成疏忽，等等，其实严重的问题有两个：

* 表现逻辑需要交互逻辑之后主动调用，会增加开发者的心智负担。
* 当数据很多时容易让 Controller 变得臃肿，会增加维护成本。

我们再看下，使用委托&回调 或者 事件时候的代码，如下：

transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count--;  
 });

会发现，代码精简了很多，当按钮点击的时候，我们只需要考虑让数据发生变化即可，不用关心表现逻辑是怎么样的，当数据变化时自然就会触发表现逻辑，这就是所谓的数据驱动。所以通过以上比较和分析发现，表现逻辑的实现使用 委托&回调 或者 事件是比主动调用更合适的。

那么具体哪个跟合适呢？

这里笔者以经验告诉大家，如果是单个数值变化，用委托的方式更合适，比如金币、分数、等级、经验值等等，如果是颗粒度较大的更新用事件比较合适，比如从服务器拉取了一个任务列表数据，然后任务列表数据存到了 Model，此时 Model 的任务列表数据发生了变更，这个时候就向 View 发送个事件比较合适。

不过从此篇开始到现在，笔者发现了两个问题，第一个问题是我们的 Event 工具类是无法传参的。

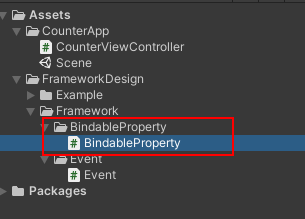
第二个就是每个数据都实现一遍如下代码是很不聪明的：

public static class CounterModel  
 {  
 private static int mCount = 0;  
  
 public static Action<int> OnCountChanged ;  
   
 public static int Count  
 {  
 get => mCount;  
 set  
 {  
 if (value != mCount)  
 {  
 mCount = value;  
 OnCountChanged?.Invoke(value);  
 }  
 }  
 }  
 }

上述代码中，一个 Count 就要写一个 mCount 变量，然后一个 OnCountChanged 委托，然后还有值比较的代码，所以这里笔者引入一个工具类，叫做 BindableProperty，也就是可绑定的属性，具体代码如下：

using System;  
  
namespace FrameworkDesign  
{  
 public class BindableProperty<T> where T : IEquatable<T>  
 {  
 private T mValue;  
  
 public T Value  
 {  
 get => mValue;  
 set  
 {  
 if (!mValue.Equals(value))  
 {  
 mValue = value;  
 OnValueChanged?.Invoke(value);  
 }  
 }  
 }  
  
 public Action<T> OnValueChanged;  
 }  
}

代码所在的位置如下：



代码量不是很多，而且这个版本的 BindableProperty 是简化的版本，所以代码有些粗糙，但是足够阐述概念了。

我们把 BindableProperty 尝试应用到我们的 CounterApp 里，首先需要修改 Model 部分的代码，如下：

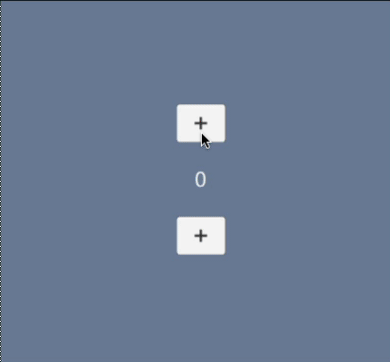
public static class CounterModel  
 {  
 public static BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }

代码一下子少了很多，而且很清爽。

然后对应地 ViewController 的部分代码如下：

using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 CounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value--;  
 });  
   
 OnCountChanged(CounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 CounterModel.Count.OnValueChanged -= OnCountChanged;  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

代码不难，我们看下 Unity 的运行结果，如下：



OK，没啥问题。

BindableProperty 是一个小工具，其思想借鉴自 UniRx 的 ReactiveProperty。

而这个小工具可以完善的部分很多，这些等笔者演化完架构部分之后再完善或者重新设计。

BIndableProperty 是 数据 + 数据变更事件 的合体，它既存储了数据充当 C# 中的 属性这样的角色，也可以让别的地方监听它的数据变更事件，这样会减少大量的样板代码，样板代码就是不用 BindableProperty 之前，如果在增加一个数据，比如金币，那就还需要写一套一模一样的代码，比如增加一个 mGold、OnGoldChanged 委托、Gold 的数值比较逻辑，这种代码叫做样板代码，样板代码有点浪费时间，所以很多框架都会用代码生成或者泛型的方式去减少样板代码的编写时间。

OK，现在我们的 CounterApp 是一个数据驱动的 App 了。

最后再说一点，我们之前有说过，子节点调用父节点时可以用委托或事件，那么 Model 和 View 为啥也可以用委托和事件呢？

答案很简单，就是数据一般是一个项目的底层，子节点和父节点是一个自底向上的关系，而 Model 和 View 也是一个底层和上层的关系，所以我们可以把这条结论编程，自底向上的逻辑可以用委托或者事件，而自顶向下的逻辑可以用方法调用。

此课的内容就讲到这里，我们下一课再见。

总结：

* 表现逻辑 适合用 事件 或 委托
* 表现逻辑用方法调用会造成很多问题，Controller 臃肿难维护、
* Model 和 View 是自底向上的关系
* 自底向上用事件或委托
* 自顶向下用方法调用
* Event 工具类不能传参
* BindableProperty 是 数据 + 数据变更事件，可以节省代码量

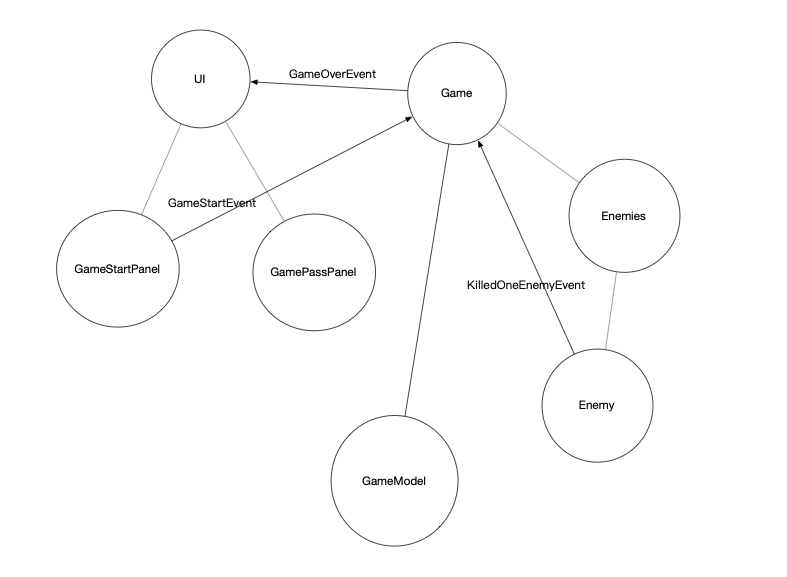
# 8. 第八课《点点点》使用 BindableProperty

在上一堂课，我们通过优化 CounterApp 的表现逻辑，引出了 BindableProperty 这个工具类，然后通过尝试对象之间交互的三种方式去实现 CounterApp 的表现逻辑，我们的到了一些结论，如下：

* 表现逻辑 适合用 事件 或 委托
* 表现逻辑用方法调用会造成很多问题，Controller 臃肿难维护、
* Model 和 View 是自底向上的关系
* 自底向上用事件或委托
* 自顶向下用方法调用（也还可以用事件，比如《点点点》里就用到）
* Event 工具类不能传参
* BindableProperty 是 数据 + 数据变更事件，可以节省代码量

在这一堂课，我们尝试在《点点点》中使用 BindableProperty。

我们先看下 《点点点》的引用关系图，如下：



非常简单。

我们再看下 GameModel 的代码，如下：

namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public static int KillCount = 0;  
  
 public static int Gold = 0;  
  
 public static int Score = 0;  
  
 public static int BestScore = 0;  
 }  
}

我们把这四个变量都改成 BindableProperty，代码如下:

namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public static BindableProperty<int> KillCount = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public static BindableProperty<int> Gold = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public static BindableProperty<int> Score = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public static BindableProperty<int> BestScore = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

这个时候会发现有编译错误，在 Game 类中，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 ...  
   
 private void OnEnemyKilled()  
 {  
 GameModel.KillCount++; // 编译错误  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount == 10) // 编译错误  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
  
 ...  
 }  
}

我们改下代码，如下:

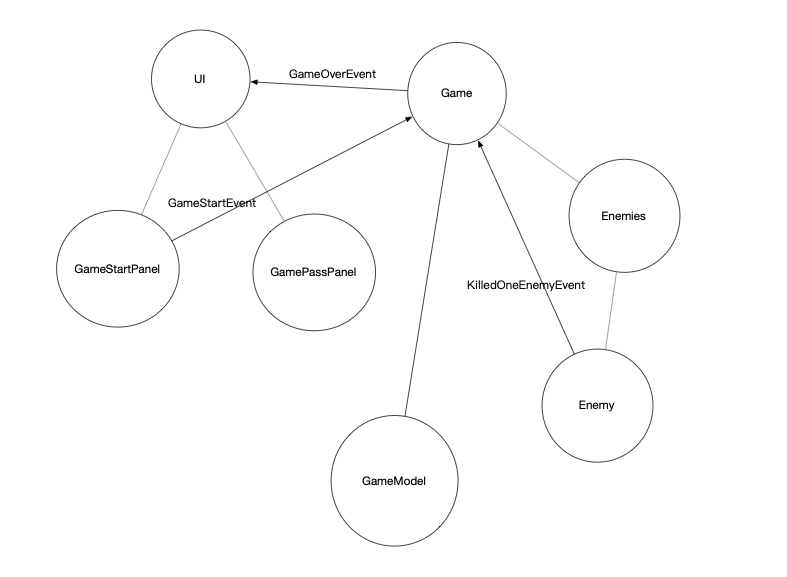
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 ...  
   
 private void OnEnemyKilled()  
 {  
 GameModel.KillCount.Value++;  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount.Value == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
  
 ...  
 }  
}

这样编译错误就解决了，我们运行一下游戏，结果如下：

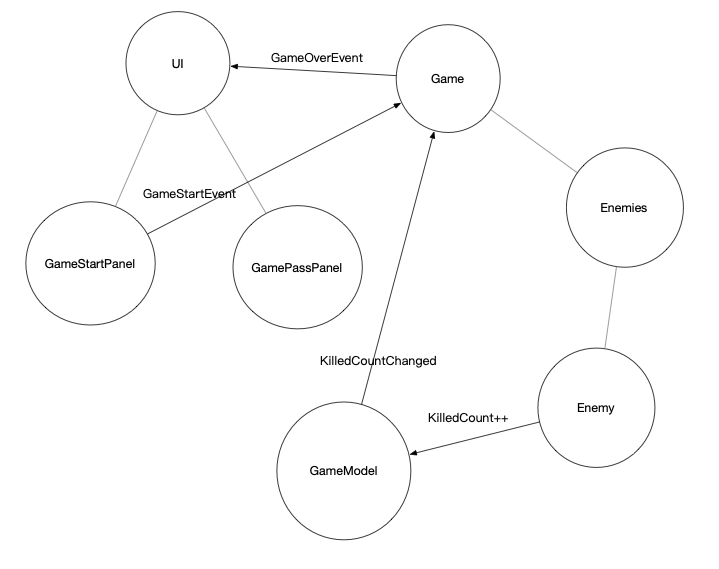


没什么问题。

OK，我们这个时候看下引用关系图，如下：



通过引入 BindableProperty 之后，我们可以发现，其实《点点点》游戏中的一个事件，我们是可以去掉的，而这个事件就是 KilledOneEnemyEvent，而 KilledOneEnemyEvent 是可以由 GameModel.KilledCount 的变更事件替代，如下：



OK，这里我们也把代码更新下。

首先在 Enemy 里对 GameModel.KillCount 进行 减减操作，代码如下：

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 GameModel.KillCount.Value--;  
 }  
 }  
}

然后，让 Game 类监听 GameModel.KillCount 的数据变更事件，而不是 KilledOneEnemyEvent，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 GameModel.KillCount.OnValueChanged += OnEnemyKilled;  
 }  
   
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
 private void OnEnemyKilled(int killedCount)  
 {  
  
 // 十个全部消灭再显示通关界面  
 if (killedCount == 10)  
 {  
 // 触发游戏通关事件  
 GamePassEvent.Trigger();  
 }  
 }  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 GameModel.KillCount.OnValueChanged -= OnEnemyKilled;  
 }  
 }  
}

代码很简单。

最后，我们删除一下，KilledOneEnemyEvent 类。

删除中。。。

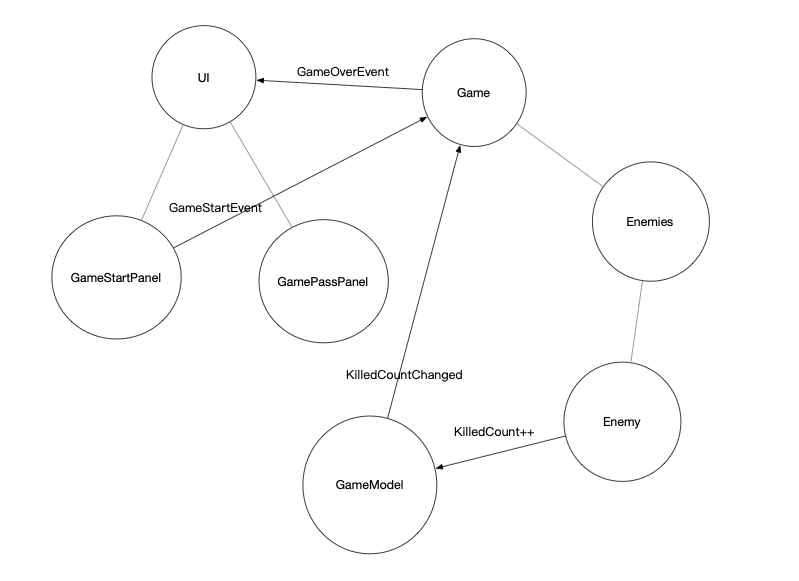
删除完了之后，我们运行下 Unity 测试一下，结果如下：



OK，没有变化。

到此，我们就完成了《点点点》项目中引入 BindableProperty 的操作。

再贴一下当前的关系图，如下：



这里简单说一下，数据变更事件 是 事件中的一种，除了数据变更事件外，我们还接触到了跨模块的事件，当然跨模块的事件会和 KilledOneEnemyEvent 一样，会被未来的其他的概念或工具替代掉，这个以后介绍。

然后看下以上的引用关系图，这里很明显是有表现逻辑和交互逻辑的。

图中的交互逻辑是 KilledCount++ 的部分，而表现逻辑是 KilledCountChanged 部分，应该比较容易分辨。

OK，到此，此课的内容就结束了。

下一堂课开始，我们来优化 CounterApp 的 交互逻辑 部分。

我们下一堂课再见，拜拜。

# 9. 第九课 交互逻辑优化-引入 Command

在这堂课，我们来优化一下 CounterApp 的交互逻辑。

先贴出来当前的代码，如下：

using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 CounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value--;  
 });  
   
 OnCountChanged(CounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 CounterModel.Count.OnValueChanged -= OnCountChanged;  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

其中交互逻辑的代码如下：

using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 ...  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value++;  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑：这个会自动触发表现逻辑  
 CounterModel.Count.Value--;  
 });  
   
 ...  
   
 }  
  
 ...  
 }  
  
 ...  
}

交互逻辑其实只有两行代码，看起来没什么课优化的。

不过，交互逻辑可并不是只是包含数据操作这么简单，在我们一般的项目开发中，交互逻辑的代码量是非常多的，除了有数据操作之外，有的时候还有查询其他 Model 的数据代码，有的时候还需要访问服务器等等。

比如，一下的伪代码是非常容易出现的，如下：

using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 ...  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 Http.Post("https://aabbcc.com/api/sub\_count",(response,err)=>{  
   
 if(err){  
 // 处理错误等，弹出提示啥的  
 } else {  
 CounterModel.Count.Value = int.Parse(response);  
   
 // 可能有成就判断  
 //（这种代码其实不合理，一般是由成就系统去监听 Count 的变更事件）  
 if (CounterModel.Count.Value == 100)  
 {  
 // 达成 100 次点击成就提示 等等  
 }  
   
 // 10 金币点一次（直接减去 10 ，或者再请求一次服务器）  
   
 CounterModel.Gold.Value -= 10;  
 }  
 });  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 Http.Post("https://aabbcc.com/api/sub\_count",(response,err)=>{  
 CounterModel.Count.Value = int.Parse(response);  
 });  
 });  
   
 ...  
   
 }  
  
 ...  
 }  
  
 ...  
}

以上代码仅仅是伪代码，总之是想表达，交互逻辑的代码量是很多的。

而一个 Controller 如果有很多交互逻辑，那么会越来越臃肿，越来越难维护，而比较容易想到的做法是像网络请求的接口都封装成 Service 层，等等，但是这个只能缓解问题，但是不能根治问题。

假如我们定义了一个 CounterService，代码如下:

public class CouterService  
{  
 public static void AddCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/add\_count",(response,err)=>{  
 // ...  
 });  
 }  
   
 public static void SubCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/sub\_count",(response,err)=>{  
 // ...  
 });  
 }  
}

那么未来，如果增加相关的功能，代码就会变成如下：

public class CouterService  
{  
 public static void AddCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/add\_count",(response,err)=>{  
 // ...  
 });  
 }  
   
 public static void SubCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/sub\_count",(response,err)=>{  
 // ...  
 });  
 }  
   
 public static void MultiplyCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/mul\_count",(response,err)=>{  
 // ...  
 });  
 }  
   
 public static void DivideCount(Action succeed)  
 {  
 Http.Post("https://aabbcc.com/api/divide\_count",(response,err)=>{  
 // ...  
 });  
 }  
   
 // 下边是接口实现 x 100  
}

代码会变得很多，那么大家可能会想到，根据业务模块再去拆分几个 Service 就可以了，这样也是可以的，但是未来还是需要不断地去改动这类代码的。这里笔者想说的是，通过给类增加方法去增加功能的这种扩展方式是有瓶颈的，10 个以内的方法还好说（一般说 7 个），但是超过 10 个了就会很难维护，当然 10 这个数字是笔者大致的预估，并不是很准确，但是大概意思是这样的。

而笔者在研究 PureMVC、StrangeIOC、uFrame 等框架，甚至是 .Net Core 的 DDD 实现之后，发现是有解决方案的，甚至它们的解决方式是完全一致的，就是引入 Command，即命令模式。

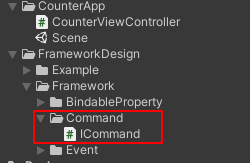
这里笔者不深讲命令模式，简单一句话概括就是，本来用一个方法来写的逻辑，改成用对象来实现，而这个对象只有一个执行方法。

直接上代码吧。

我们先定义一个接口，叫做 ICommand，代码如下：

namespace FrameworkDesign  
{  
 public interface ICommand  
 {  
 void Execute();  
 }  
}

代码所在位置如下：



然后我们实现 CounterModel.Count++ 操作，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct AddCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterModel.Count.Value++;  
 }  
 }  
}

很简单。

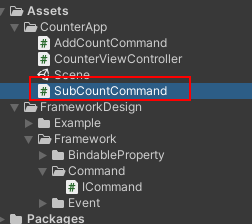
代码所在的位置如下：



再实现 CountModel.Count-- 操作，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct SubCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterModel.Count.Value--;  
 }  
 }  
}

代码所在的位置如下：

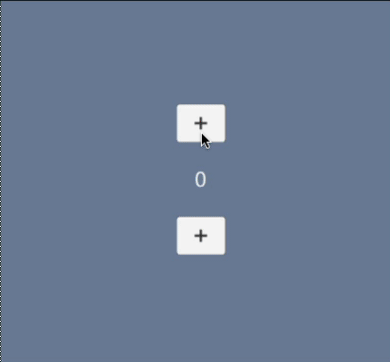


这里讲个很小的知识点，就是大家发现，笔者的 Command 是用的是 struct 实现的，而不是用的 class，这是因为游戏里边的交互逻辑有很多，如果每一个都用去 new 一个 class 的话，会造成很多性能消耗，比如 new 一个对象所需要的寻址操作、比如对象回收需要的 gc 等等，而 struct 内存管理效率要高很多，所以 Unity 官方的 Vector3、Vector2 等都是用 struct 实现的。

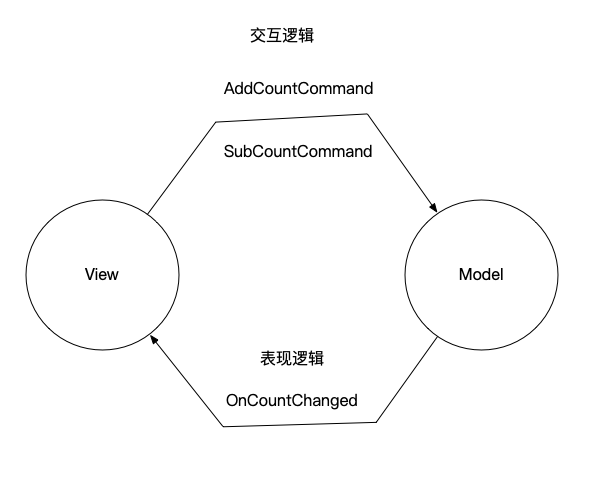
OK，我们接下来把 CounterApp 的交互逻辑更改一下，代码如下:

using System;  
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 CounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(CounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 CounterModel.Count.OnValueChanged -= OnCountChanged;  
 }  
 }  
  
 public static class CounterModel  
 {  
 public static BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

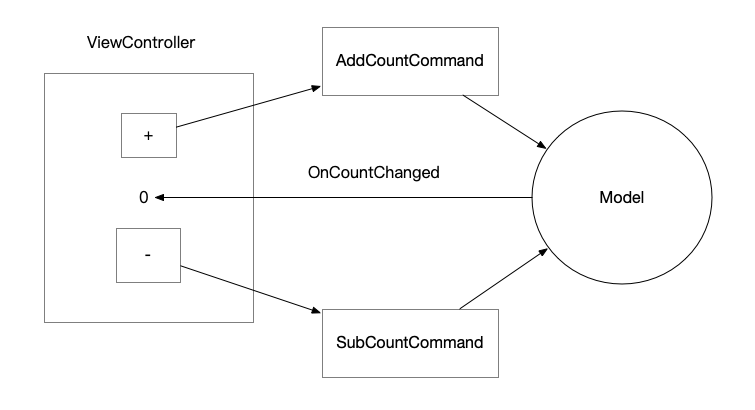
代码不难，运行结果如下：



OK，我们更新一下 CounterApp 的 View 和 Model 的图，如下：

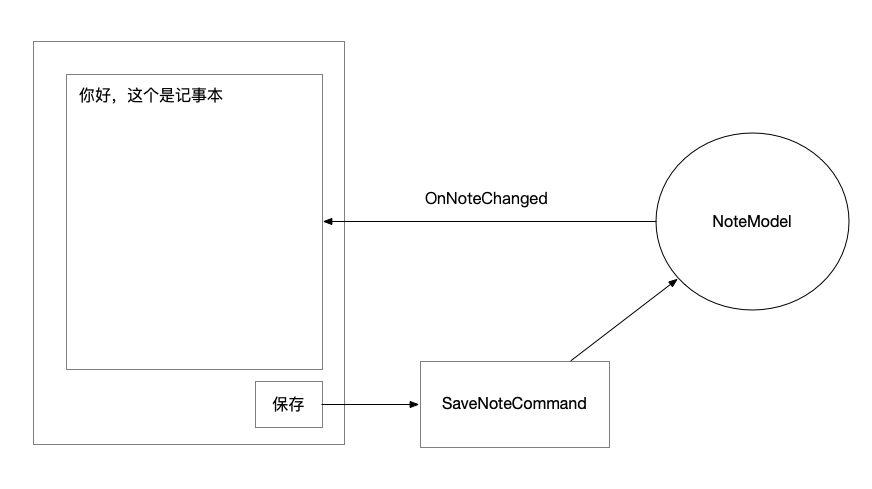


实际上，我们可以把 CounterApp 的实现用一张图给描述出来，如下：



非常清晰，这就是引入 Command 的第一个好处，就是在一张纸上就可以描述出来 App 的大致实现，同样，也可以直接先在纸上实现一个 App，然后再用代码实现。

比如我们来实现一个简单的记事本 App，用图画就是如下：

   
还可以实现很多。

使用 Command 符合读写分离原则（Comand Query Responsibility Segregation），简写为 CQRS ，了解服务端的童鞋应该是听说过这个概念的，而这个概念在 StrangeIOC、uFrame、PureMVC、Loxodon Framework 都有实现，而在微服务领域比较火的 DDD（领域驱动设计）的实现一般也会实现 CQRS。

由于是演化阶段，所以暂时对 Command 还有 CQRS 不做深入介绍了，这个会在此课的后续章节中单独介绍。

这里只需要记住一点，Command 模式就是逻辑的调用和执行是分离的，我们知道一个方法的调用和执行是不分离的，因为一旦你调用方法了，方法也就执行了，而 Command 模式能够做到调用和执行在空间和时间上是能分离的。

空间分离的方法就是调用的地方和执行的地方放在两个文件里。

时间分离的方法就是调用的之后，Command 过了一点时间才被执行。

而 Command 模式由于有了调用和执行分离这个特点，所以我们可以用不同的数据结构去组织 Command 调用，比如命令队列，再比如用一个命令的堆栈，来实现撤销功能（ctrl + z），这些有能力的童鞋可以自己尝试下，而关于 Command 模式的资料大家请自行查阅，当然笔者会在后边的章节会介绍。

Command 在目前的职责是分担 Controller 的 交互逻辑 的 职责，让很多比较混乱的交互逻辑代码 从 Controller 迁移到 Command 里。

还有 Command 是 QFramework 系统设计架构里的非常重要的一个概念。

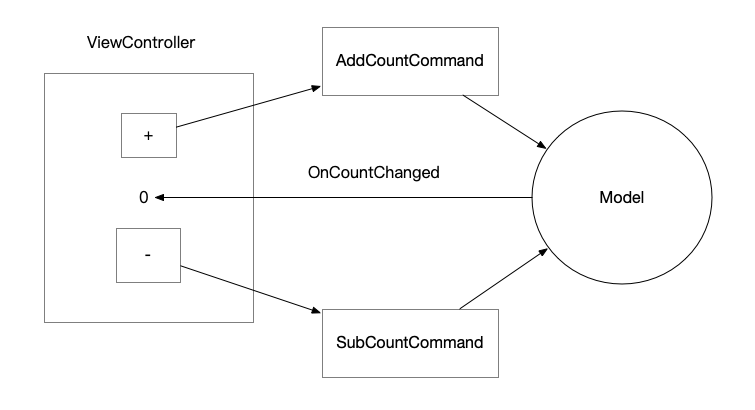
OK，这堂课的内容就说到这里，我们下一趟课再见，拜拜。

总结：

* 交互逻辑 会有很多 会让 Controller 臃肿
* 很多 Unity 框架的交互逻辑是由 Command 实现的
* Command 模式可以让逻辑的调用和执行在空间和时间上分离
* Command 分担 Controller 的交互逻辑
* struct 比 class 有更好的内存管理效率
* CQRS 读写分离

# 10. 第十课 有趣的尝试 CounterApp 编辑器扩展版本

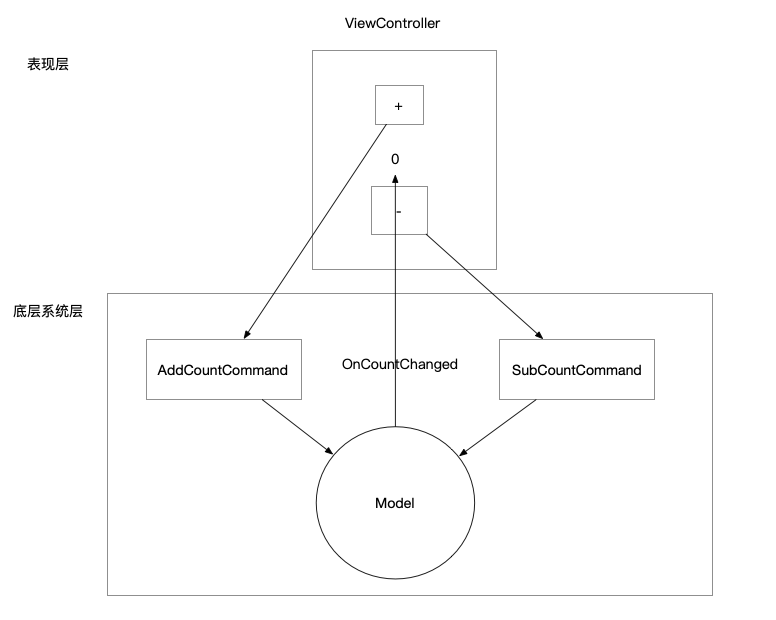
在上一堂课， 我们在 CounterApp 中引入了 Command，如下图所示。

   
通过引入 Command 和 BindableProperty 还有 Event 工具类，我们的在第一课提到的 QFramework 系统设计架构已经达到了一点雏形。

虽然目前仅仅是雏形阶段，但是我们还是能提会到这套架构的利好的部分的，比如底层系统和表现层弱耦合关系。

那么在 CounterApp 中哪里属于表现层，哪里属于底层系统层呢？

很简单，看如下图即可：



很简单，就是除了 ViewController 其他的部分都算作底层系统层，而 ViewController 则算作表现层。

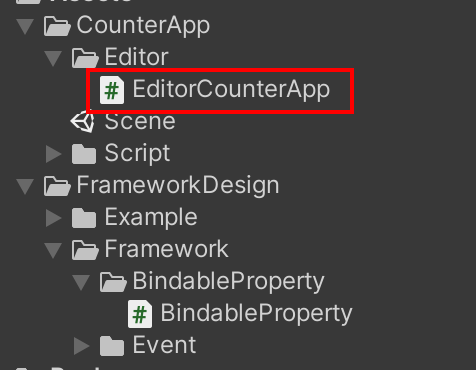
那么分这两个层有什么意义呢？答案是，底层系统层是可以共享给别的展现层使用的。

比如我们目前的 ViewController 是由 Unity 的 MonoBehaviour（UGUI) 实现的，当然我们也可以在其他的支持 C# 的平台使用这套底层系统层。

就比如，我们可以在不改变底层系统代码的情况下，我们可以很快实现一个 编辑器扩展版本的 CounterApp。

我们接下来就做一下这个有趣的尝试。

创建个脚本，位置如下：

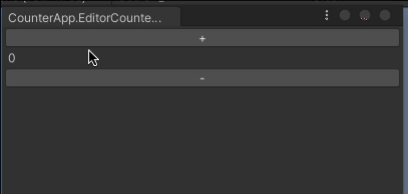


代码如下:

using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 new AddCountCommand()  
 .Execute();  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterModel.Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 new SubCountCommand()  
 .Execute();  
 }  
 }  
 }  
}

代码不难，当然如果对编辑器扩展不太熟悉的童鞋，建议手敲一下。

运行结果如下：



OK，这样我们很快就实现了一个 编辑器扩展版本的 CounterApp。

这个是 QFramework 系统设计架构的第一个利好的地方，就是切换表现层非常方便 或者说 底层系统比较容易共享。

做到这点是因为我们引入了 Command，Command 把原来 Controller 中负责操作数据的 交互逻辑 剥离出来了，而这个交互逻辑大部分情况下和视图是没啥关系的，所以如果从 Controller 剥离出来，是可以提高底层系统代码的复用率的。

如果交互逻辑直接是在 Controller 里实现，而 Controller 一般都会持有 View 对象，或者和 View 层有一定的耦合，这样的话想做到底层系统共享是很费劲的，引入了 Command 恰好解决了这个问题，同时我们发现用户的一次操作其实可以封装成一个 Command，比如用户点击 + 按钮，那么就是发送 AddCountCommand，而用户点击 - 按钮，那么就是发送 SubCountCommand，这样可以提高代码的维护性，因为一个操作封装成一个 Command 其实可以提高代码的易读性的。当别人接手此项目的时候，有哪些操作，只要看 Command 就一目了然。

除了以上的利好，其实在项目开发的前期阶段，目前的架构也是有好处的。

在项目前期，更多的是要做很多数值和玩法上的验证，而数值和玩法一般情况下是和美术无关的，这个时候可以用此架构写一套系统，然后使用类似用 xml 写界面的工具可以快速搭建一套数值和玩法的原型，等数值和玩法验证了差不多之后，就直接可以把这套视图扔掉，再用正式的 UGUI 或者其他界面制作工具去拼一套新的界面，然后接入这套系统即可。

这套架构还有一套用法，就是用于分工上，比如让一些非常有经验的开发者或者主程先写底层系统，然后没有经验的初学者或者新手写表现层，由于表现层和底层的交互方式已经是确定下来了，表现层到系统层用 Command，系统层通过事件或者委托通知表现层，在通知的时候可以推送数据，也可以让表现层收到通知后自己去查询数据。

以上不管是哪种，笔者都亲自实践过，证明是可行的，而且非常时候团队只有一两个高级开发者剩下都是初学者这样的团队，这要高级开发者掌握了这套架构并且能够设计底层系统，那么剩下的就是表现层的工作量了，而表现层的工作量其实是非常多的，但是大部分的技术含量并不高，除非和渲染有些关系，这些都可以交给初学者去做。

而等团队中的初学者慢慢成长也可以慢慢写底层系统的时候，可以将一个大的模块包括底层系统交给初学者去做。

OK，虽然有这些利好，但是架构还没有演化完，还有很多问题，目前只有表现层能够替换的，但是一个系统所涉及的 API 并不是只有表现层的 API，很多也是底层的 API，比如数据存储 API 等，这些我们后边学习一些概念之后继续支持。

这堂课的内容就这些，我们下一趟课再见，拜拜。

总结：

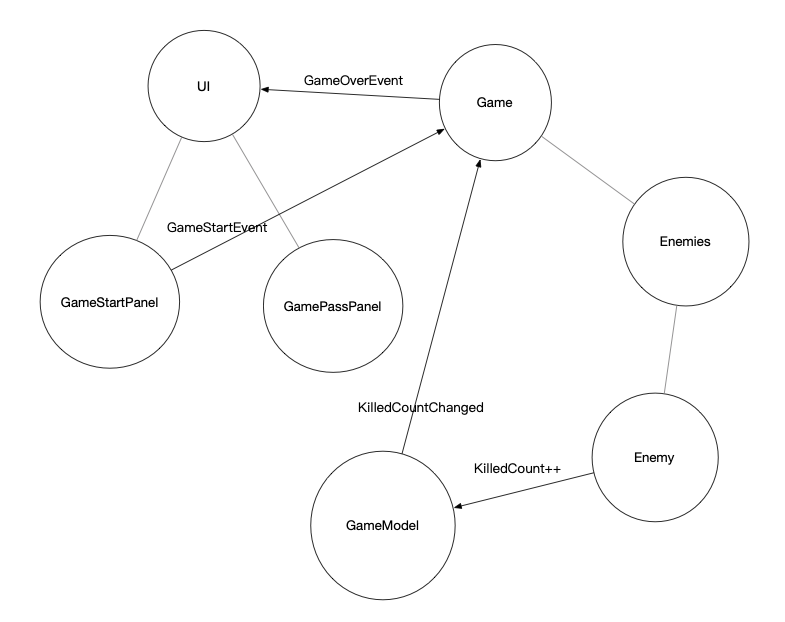
* 表现层 到 底层系统层 用 Command
* 底层系统 到 表现层 用委托或者事件（通知）
* 表现层是可替换的

# 11. 第十一课《点点点》使用 Command

我们在 CounterApp 中引入了 Command，并且通过 编辑器扩展版本的 CounterApp 知道了一些引入 Command 之后一些利好的部分，比如：比较容易替换表现层、在团队流程、分工方面提供了便利。

在这一堂课，我们来尝试在《点点点》中引入 Command。

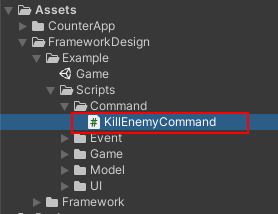
我们先看下《点点点》目前的引用关系图，如下：



在《点点点》中，操作数据的部分只有图中的 KilledCount++ 部分，所以我们目前只需实现一个 KillEnemyCommand 即可，代码如下：

namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 GameModel.KillCount.Value++;  
 }  
 }  
}

代码所在的位置如下：



然后，我们把 Enemy 的代码改成如下:

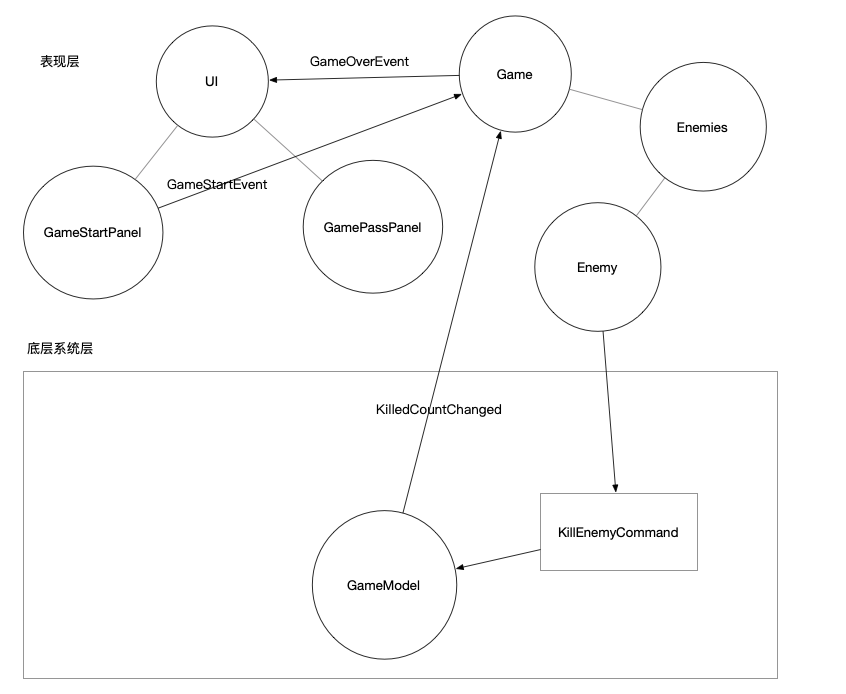
using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 // 用 Command  
 new KillEnemyCommand()  
 .Execute();  
 }  
 }  
}

代码很简单，我们来运行一下游戏，结果如下：



没有什么变化。

接着我们更新以下《点点点》的引用关系图，不过这次我们将此图改成 表现层-底层系统层的风格，如下:



由于目前《点点点》所涉及的数据部分比较少，所以有种头重脚轻的感觉，不过没关系后边还有分数、金币等数据呢。

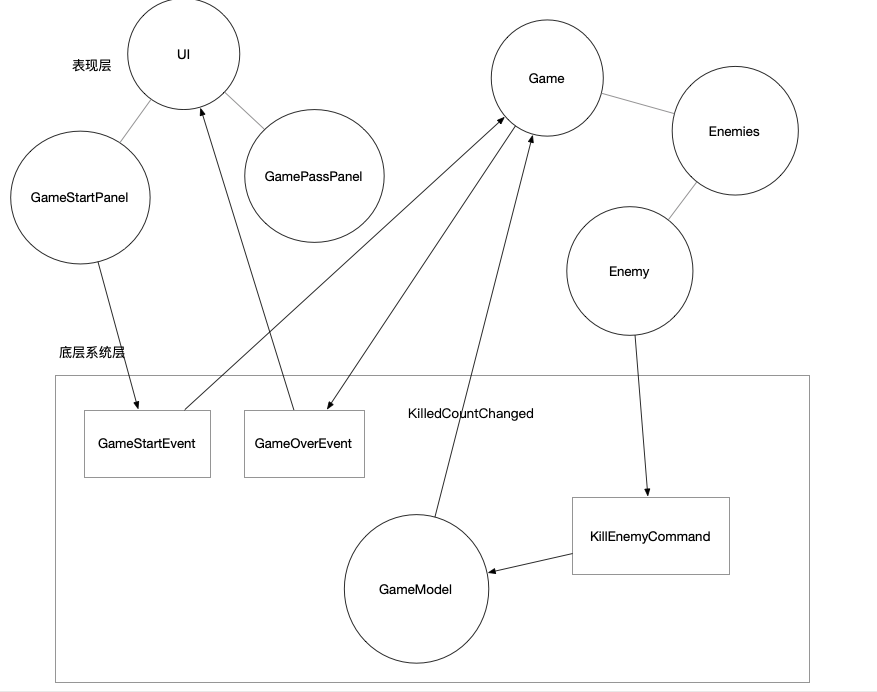
现在我们的底层系统层有如下元素：

* Command
* BindableProperty
* Model

其中 BindableProperty 实际上是 数据 + 事件，而数据的部分其实是由 Model 负责的，所以底层系统层的元素如下：

* Command
* Event
* Model

而《点点点》游戏中，还有 GameStartEvent、GameOverEvent，这些事件其实是应该归入到底层系统层的，所以我们把图改成如下：

   
这样的图是更合理的，事件是应该是属于底层系统层的。

而 GameStartEvent 和 GameOverEvent，实际上是游戏的状态变更事件，游戏从 未开始状态 -> 游戏中状态 -> 游戏结束状态。而这个状态我们并没有用代码存储在 Model 里，但是实际上它是存在的，而游戏的状态其实也是数据，我们其实可以在 GameModel 中用一个枚举表示出来，伪代码如下：

namespace FrameworkDesign.Example  
{  
  
 public enum GameStates  
 {  
 NotStart,  
 Started,  
 Over  
 }  
   
 public class GameModel  
 {  
 /// <summary>  
 /// 游戏的状态  
 /// </summary>  
 public static GameStates State = GameStates.NotStart;  
   
 public static BindableProperty<int> KillCount = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 ...  
 }  
}

当然这个只是一个伪代码而已，只是为了说明，《点点点》确实是存在这么一个状态的，而这个状态如果写出来的话是 Enum 类型的数据，而当我们点击游戏开始 按钮的时候，其实是改变了游戏的状态的，也就是游戏的数据，而我们这个时候发出去的 GameStartEvent，实际上是数据（状态）变更的事件，所以说它应该是在底层系统层的这个结论是没问题的。

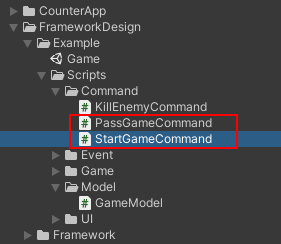
表现层 只能往 系统层 发送 Command 或者做数据查询，并没有说可以发送事件这么一说，而且事件只能是由底层系统层向表现层发送，当然这个规定是笔者规定的，是 QFramework 系统设计架构的一个架构原则。

而如果这样的话，我们就只能创建两个 Command 了，一个是 StartGameCommand，一个是 PassGameCommand，代码如下:

namespace FrameworkDesign.Example  
{  
 public struct StartGameCommand : ICommand  
 {  
 public void Execute()  
 {  
 GameStartEvent.Trigger();  
 }  
 }  
}

namespace FrameworkDesign.Example  
{  
 public struct PassGameCommand : ICommand  
 {  
 public void Execute()  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
}

代码所在的位置如下：



两个 Command 做的事情很简单，仅仅只是触发了一个事件而已，当然这里其实隐含了一些概念的，这些触发的事件是状态（数据）变更事件，和我们之前接触到的 BindableProperty 的数据变更事件（委托）是一样的事件，只不过我们在项目中没有明显地声明 GameState 这个变量，当然了以后我们想实现再来一局，或者游戏暂停这样的功能可能就需要声明了，但是目前还不需要。

我们先更改 GameStartPanel 的代码，把 StartGameCommand 应用起来，代码如下：

using UnityEngine;  
using UnityEngine.UI;  
  
namespace FrameworkDesign.Example  
{  
 public class GameStartPanel : MonoBehaviour  
 {  
 /// <summary>  
 /// Enemy 的父节点  
 /// </summary>  
 public GameObject Enemies;  
   
 void Start()  
 {  
 transform.Find("BtnGameStart").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 gameObject.SetActive(false);  
   
 // 发送 Command  
 new StartGameCommand()  
 .Execute();  
 });  
 }  
 }  
}

然后是 Game 脚本中，当判断击杀敌人为 10 时，发送 PassGameCommand，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 GameModel.KillCount.OnValueChanged += OnEnemyKilled;  
 }  
   
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
 private void OnEnemyKilled(int killedCount)  
 {  
  
 // 十个全部消灭再显示通关界面  
 if (killedCount == 10)  
 {  
 // 发送 Command  
 new PassGameCommand()  
 .Execute();  
 }  
 }  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 GameModel.KillCount.OnValueChanged -= OnEnemyKilled;  
 }  
 }  
}

代码不难，我们运行下游戏，结果如下:



OK，没问题。

不过，刚刚我们在 Game 脚本中接触到了游戏通关的规则判断的逻辑，代码如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 ...  
   
 private void OnEnemyKilled(int killedCount)  
 {  
  
 // 十个全部消灭再显示通关界面  
 if (killedCount == 10)  
 {  
 // 发送 Command  
 new PassGameCommand()  
 .Execute();  
 }  
 }  
  
 ...  
 }  
}

我们思考下，这部分逻辑是放在表现层好还是放在底层系统层好？

答案很明显，是底层系统层更好一些，因为表现层只负责表现和接收用户操作就可以了，单一职责嘛。

所以这部分的逻辑，我们可以迁移到底层系统层。

而这部分的判断，是在每次击杀敌人之后进行的，所以比较适合放在 EnemyKillCommand 里进行。

代码如下：

namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 GameModel.KillCount.Value++;  
  
 // 十个全部消灭再显示通关界面  
 if (GameModel.KillCount.Value == 10)  
 {  
 // 发送 Command  
 new PassGameCommand()  
 .Execute();  
 }  
 }  
 }  
}

代码不难。

这里还有个问题，由于 PassGameCommand，原来是在 Game 脚本里发送的，而发送的代码迁移到了 KillEnemyCommand 了，没有别的地方引用它了，所以 PassGameCommand 直接删掉就可以了，然后这里的代码直接改成触发 GamePassEvent 即可，代码如下：

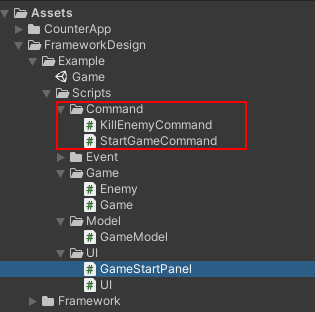
namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 GameModel.KillCount.Value++;  
   
 if (GameModel.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

而此时，Game 脚本就没必要监听 KillCount 的变更事件了，代码变成如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class Game : MonoBehaviour  
 {  
 void Start()  
 {  
 GameStartEvent.Register(OnGameStart);  
 }  
   
 private void OnGameStart()  
 {  
 transform.Find("Enemies").gameObject.SetActive(true);  
 }  
  
 private void OnDestroy()  
 {  
 GameStartEvent.UnRegister(OnGameStart);  
 }  
 }  
}

代码干净了很多，表现层做好表现层的事情就可以了。

目前的代码目录如下：



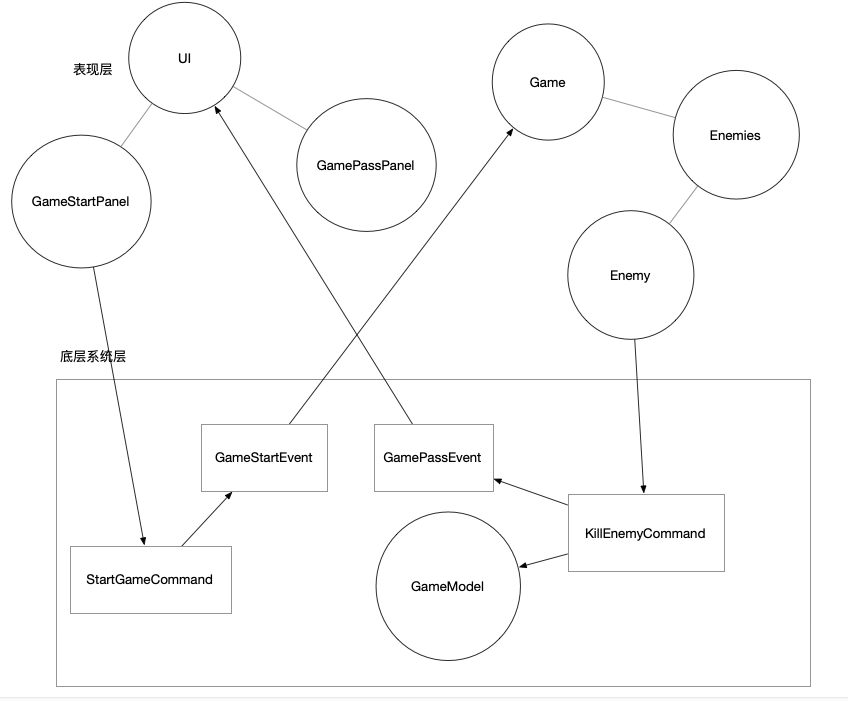
笔者删除掉了 PassGameCommand。

运行 Unity 结果如下：



没啥问题。

接着我们更新以下《点点点》的图，如下：



OK，到此，我们的《点点点》中引入 Command 就完成了。

大家可以发现，底层系统层里的东西越来越多，规矩也越来越多了，这里简单总结一下。

* 事件由系统层 向 表现层 发送
* 表现层只能用 Command 改变底层系统层的状态（数据）
* 表现层可以直接查询数据

有了这三个规矩（架构使用原则），我们离 QFramework 系统设计架构更进了一步。

这堂课的内容就这些，本来下一堂课想尝试实现一下 编辑器扩展版本的《点点点》的，但是光写界面就有一点工作量，所以就不写了。

对象之间的交互是很有用的一个概念，它适用于 表现层 和 底层系统层、也适用于 View 和 Model、也适用于父节点、子节点都是可以的。

而 Command 是对象之间的交互的第四种方式。

关于对象之间的交互的主题就到这里结束了，接下来我们需要关注第二个非常重要的问题，就是如何做模块化。

我们下一趟课再见，拜拜。

# 12. 模块化优化-引入单例

从这一堂课开始，我们来关注如何做模块化的问题。

而在前几堂课中，笔者把《点点点》和 CounterApp 都分成了，表现层 和 底层系统层 这两个层级。

表现层的模块化很容易，就是根据平台提供的概念即可，比如 Unity 的 Scene 或者 Prefab，或者由团队和个人自己规定的，登录模块表现层、战斗模块表现层、商店模块表现层等，还是比较容易的。

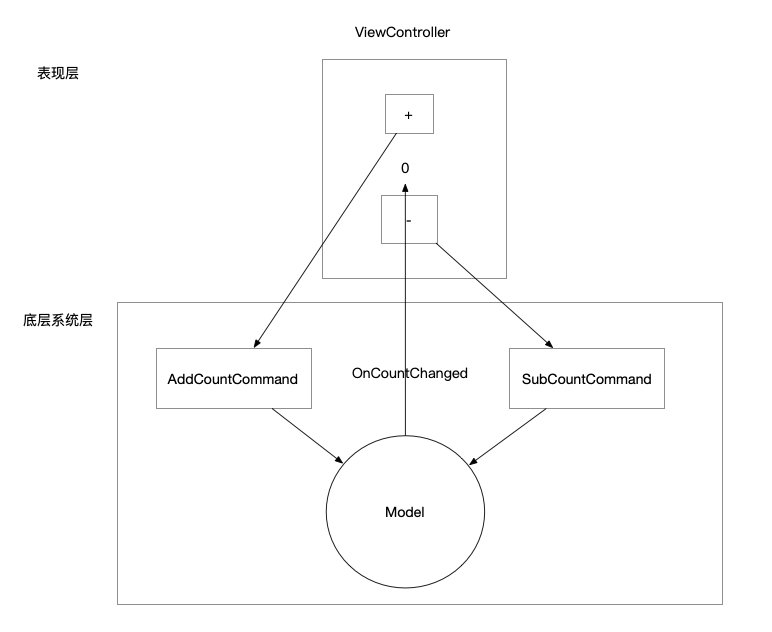
而底层系统层的模块化是相对困难的，或者说这里是有一点门道的，所以接下来重点说底层系统层的模块化。

模块化部分笔者分为两个角度去分析：

* 模块对象如何获取？
* 如何增加一个模块？

而目前不管是《点点点》还是 CounterApp，其 底层系统层 要考虑模块化的部分只有 Model 部分。

如下图所示：



目前只有一个 Model 可以算作一个代码模块，Command 算作操作，Command 可以扩展，但是 Command 对象是不可以获取的，而 Model 是共享的，会被很多地方引用，并且自身也有状态所以当做一个代码模块是没问题的。

代码如下：

public static class CounterModel  
 {  
 public static BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }

这里 CounterModel 是一个静态类。

一个静态类，从模块化的两个角度分析如下：

* 静态类可以直接获取
* 扩展一个静态类就是加上一个 static 关键字

所以静态类这种形式在模块化方面是非常方便的，这就是为啥《点点点》和 CounterApp 的 Model 都用的是静态的原因。

在开发效率上，前期用静态类做为模块化的方式是很快的，但是项目到了一定规模，就会出现一些问题，比如模块之间的互相引用关系混乱、随机且没有规律，这句话听起来是不是和一次性项目的带来的问题一模一样？

不管是对象之间的引用，还是模块之间的互相访问，如果没有限制、没有规律都是会造成问题的。

而静态类由于在技术上并没有限制别的地方对它的访问，所以使用静态类充当一个 Model 会留下很多隐患。

另外实现一个静态类是非常方便的，只要使用 C# 的语法关键字 static 即可，但是当我们打开一个项目的时候会发现多多少少都会有一些类用到 static，不过这些类是一个模块还是一个常量类等，所有通过语法关键字 static 去扩展的模块，需要思考一下才能识别这个类到底是一个模块还是工具类、还是常量类等。

OK，现在问题很清晰，即：

* 静态类没有访问限制。
* 使用 static 去扩展模块，其模块的识别度不高。

我们接下来要解决的就是这两个问题。

首先静态类没有访问限制，那么我们就稍微加上限制即可。

使用 static 去扩展模块，其模块的识别度不高，那么我们加上一定的识别度即可。

而比较容易实现的方式就是使用单例。

如果一个模块类是单例类，那么他的访问必须需要通过 模块类.Instance 获取，这增加了模块获取的一点难度，虽然这个难度仅仅是增加了一些代码量而已，但是也是增加了难度。

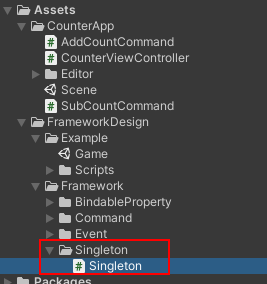
而如果一个类是单例类，那么我们就可以比较容易识别，这个单例类是一个模块类。

接下来实现一个单例的工具类，即：泛型单例，代码如下：

using System;  
using System.Reflection;  
  
namespace FrameworkDesign  
{  
 public class Singleton<T> where T : class  
 {  
 public static T Instance  
 {  
 get  
 {  
 if (mInstance == null)  
 {  
 // 通过反射获取构造  
 var ctors = typeof(T).GetConstructors(BindingFlags.Instance | BindingFlags.NonPublic);  
 // 获取无参非 public 的构造  
 var ctor = Array.Find(ctors, c => c.GetParameters().Length == 0);  
  
 if (ctor == null)  
 {  
 throw new Exception("Non-Public Constructor() not found in " + typeof(T));  
 }  
  
 mInstance = ctor.Invoke(null) as T;  
 }  
  
 return mInstance;  
 }  
 }  
  
 private static T mInstance;  
 }  
}

代码需要掌握一些 C# 反射的知识，这部分大家请自行查阅。

代码所在的位置如下：



OK，这样单例的工具类就写好了。

接下来我们在 CounterApp 中使用它，代码如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 void Start()  
 {  
 // 注册  
 CounterModel.Instance.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(CounterModel.Instance.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 CounterModel.Instance.Count.OnValueChanged -= OnCountChanged;  
 }  
 }  
  
 public class CounterModel : Singleton<CounterModel>  
 {  
 // 需要声明一个非 Public 构造  
 private CounterModel(){}  
   
 public BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

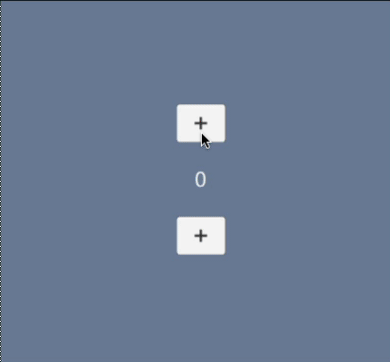
AddCountCommand.cs

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct AddCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterModel.Instance.Count.Value++;  
 }  
 }  
}

SubCountCommand.cs

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct SubCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterModel.Instance.Count.Value--;  
 }  
 }  
}

代码不难，运行结果如下：



OK，没什么变化。

同样，我们在《点点点》里也使用一下 Singleton。

代码如下:

GameModel.cs

namespace FrameworkDesign.Example  
{  
 public class GameModel : Singleton<GameModel>  
 {  
 private GameModel(){}  
   
 public BindableProperty<int> KillCount = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Gold = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Score = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> BestScore = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

KillEnemyCommand.cs

namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 GameModel.Instance.KillCount.Value++;  
   
 if (GameModel.Instance.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

代码也不是很难，运行结果如下：



没什么变化。

不过，我们的问题还没有根治，在没有使用单例之前，我们的模块化有两个问题，如下：

* 静态类没有访问限制。
* 使用 static 去扩展模块，其模块的识别度不高。

而现在我们使用了 Singleton 之后，我们还有一个问题:

* 单例没有访问限制。

当然 Singleton 相比静态类确实是在模块化方面好了那么一点点。

我们在下一堂课继续解决模块化剩余的一个问题。

这堂课的内容就这些，我们下一趟课再见，拜拜。

# 13. 模块化优化-引入 IOC 容器

再上一堂课，我们通过引入 Singleton 改善了模块化方面的两个问题。

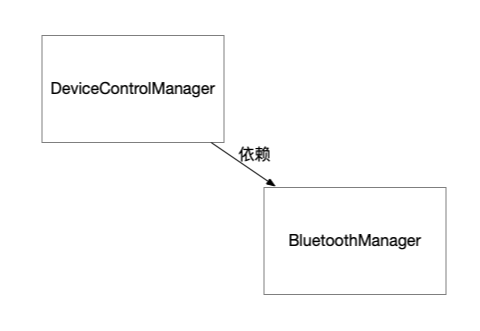
在这一堂课，我们接着继续改善 Singleton 留下的一个问题：

* 单例类没有访问限制

首先说一下，为什么单例类没有访问限制是一个问题。

假如我们有一个单例类叫做 BluetoothManager（蓝牙管理），另一个类叫做 DeviceControllManager（设备管理）。

而 DeviceControllerManager 在业务中我们假定是需要依赖 BluetoothManager 的，其依赖关系图如下：

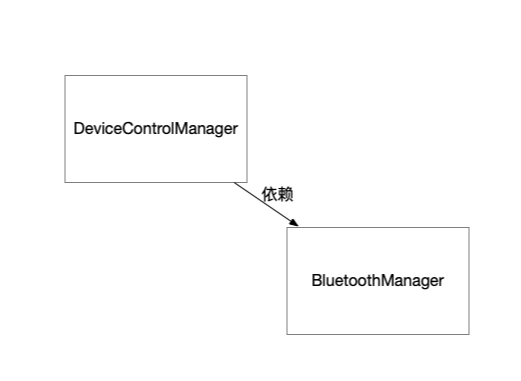


很清晰。

那么现在问题出现了，虽然 DeviceControlManager 和 BluetoothManager 都继承了 Singleton，但是从高低层关系来讲，DeviceControlManager 是比 BluetoothManager 高一个层级的。

而 DeviceControlManager 比 BluetoothManager 高一个层级这么一个信息是需要其他人去翻源码才能知道的，这就会造成一些问题。比如其他人在不知道这条信息的情况下，可能会调用 BluetoothManager 破坏掉 BluetoothManager 的原来状态，也可能造了一个和 DeviceControlMananger 差不多的轮子造成代码重复等等。

而这个时候可以通过增加一张所谓的“架构草图”来解决这个问题，比如笔者如下图：



而这个架构草图，是需要花时间和精力去维护的，并且当项目中的代码发生变更，是需要去更新这个架构草图的，这是一件很麻烦的事情，再加上游戏项目的需求变更是比一般项目的需求变更更多的。

所以单例类没有访问限制这一点是会随着项目规模越来越大，会让项目越来越混乱直到崩溃的。

当然单例类在很多中小型项目是最佳选择，因为单例类易上手，而且能够作为模块的识别标识，加上其技术实现也不难。

OK，总之单例是有一定风险的。

那么再往下一个比较好的工具，是使用 IOC 容器，这里先不说 IOC 是个什么玩意，笔者先用大白话说一下其技术原理，而具体理论内容后边会深入介绍。

IOC 容器，大家可以理解为是一个字典，这个字典以 Type 为 key，以对象即 Instance 为 value，非常简单。

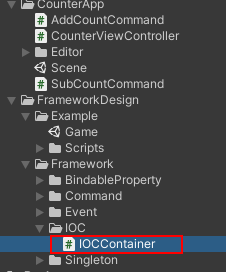
而 IOC 容器最少有两个核心的 API，即根据 Type 注册实例，根据 Type 获取实例。

我们先实现一个简单的 IOC 容器，代码如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 public class IOCContainer  
 {  
 /// <summary>  
 /// 实例  
 /// </summary>  
 public Dictionary<Type, object> mInstances = new Dictionary<Type, object>();  
  
 /// <summary>  
 /// 注册  
 /// </summary>  
 /// <param name="instance"></param>  
 /// <typeparam name="T"></typeparam>  
 public void Register<T>(T instance)  
 {  
 var key = typeof(T);  
  
 if (mInstances.ContainsKey(key))  
 {  
 mInstances[key] = instance;  
 }  
 else  
 {  
 mInstances.Add(key,instance);  
 }  
 }  
  
 /// <summary>  
 /// 获取  
 /// </summary>  
 public T Get<T>() where T : class  
 {  
 var key = typeof(T);  
   
 object retObj;  
   
 if(mInstances.TryGetValue(key,out retObj))  
 {  
 return retObj as T;  
 }  
  
 return null;  
 }  
 }  
}

代码的逻辑不难，其思想也很简单，就是类型为 Key，而 Object 为 Value。

代码所在的位置如下：

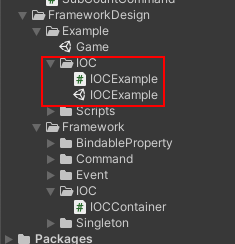


我们来为 IOCContainer 写一个简单的示例，代码如下：

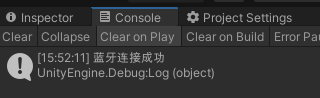
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class IOCExample : MonoBehaviour  
 {  
 void Start()  
 {  
 // 创建一个 IOC 容器  
 var container = new IOCContainer();  
   
 // 注册一个蓝牙管理器的实例  
 container.Register(new BluetoothManager());  
   
 // 根据类型获取蓝牙管理器的实例  
 var bluetoothManager = container.Get<BluetoothManager>();  
   
 //连接蓝牙  
 bluetoothManager.Connect();  
 }  
  
 public class BluetoothManager  
 {  
 public void Connect()  
 {  
 Debug.Log("蓝牙连接成功");  
 }  
 }  
 }  
}

代码不难，一行一行读取注释即可。

代码所在位置如下：



将此脚本挂在任意 GameObject 上，运行 Unity ，结果如下:



OK，这就是基本的用法，不是很难，在下一趟课，笔者尝试把 IOCContainer 引入到 CounterApp 和 《点点点》 中，这堂课的内容就这些，我们下一趟课再见，拜拜。

# 14. 《点点点》和 CounterApp 引入 IOC 容器

再上一堂课，我们实现了一个简单的 IOC 容器，并且把它的基本用法介绍了一下。

在这一堂课，我们来尝试在《点点点》和 CounterApp 中引入 IOC 容器。

首先我们在 CounterApp 中引入 IOC 容器。

我们先创建一个 CounterApp 类，用于注册全部模块，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp  
 {  
 private static IOCContainer mContainer = null;  
  
 // 确保 Container 是有实例的  
 static void MakeSureContainer()  
 {  
 if (mContainer == null)  
 {  
 mContainer = new IOCContainer();  
 Init();  
 }  
 }  
  
 // 这里注册模块  
 private static void Init()  
 {  
 mContainer.Register(new CounterModel());  
 }  
   
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureContainer();  
 return mContainer.Get<T>();  
 }  
 }  
}

代码不难，但是有一定的逻辑。

接着我们把 CounterApp 类应用起来，代码如下：

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 private CounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = CounterApp.Get<CounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
 }  
  
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel  
 {  
 public BindableProperty<int> Count = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

AddCountCommand.cs

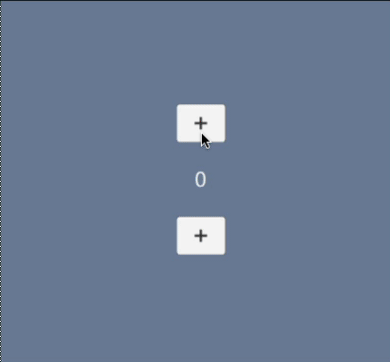
using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct AddCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterApp.Get<CounterModel>().Count.Value++;  
 }  
 }  
}

SubCountCommand.cs

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct SubCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterApp.Get<CounterModel>().Count.Value--;  
 }  
 }  
}

代码其实也不是很难，总之就是获取 CounterModel 只能通过 CounterApp 获取了，增加了一下 CounterModel 获取的限制。

运行结果如下：

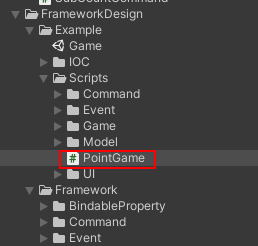


接下来，《点点点》项目也是如此，我们先创建一个类，叫做 PointGame，代码如下:

namespace FrameworkDesign.Example  
{  
 public class PointGame   
 {  
 private static IOCContainer mContainer = null;  
  
 // 确保 Container 是有实例的  
 static void MakeSureContainer()  
 {  
 if (mContainer == null)  
 {  
 mContainer = new IOCContainer();  
 Init();  
 }  
 }  
  
 // 这里注册模块  
 private static void Init()  
 {  
 mContainer.Register(new GameModel());  
 }  
   
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureContainer();  
 return mContainer.Get<T>();  
 }  
 }  
}

代码和 CounterApp 差不多，不同的地方只有一行，也就是注册 Model 的部分。

代码所在的位置如下:



接着我们把，PointGame 应用到《点点点》游戏中，代码如下：

GameModel.cs

namespace FrameworkDesign.Example  
{  
 public class GameModel  
 {  
 public BindableProperty<int> KillCount = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Gold = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Score = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> BestScore = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

KillEnemyCommand.cs

namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 var gameModel = PointGame.Get<GameModel>();  
   
 gameModel.KillCount.Value++;  
   
 if (gameModel.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

代码也不是很难，运行结果如下:  
 

逻辑上没啥问题。

不过我们发现 CounterApp 和 PointGame 有大量重复代码，如下:

CounterApp.cs

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp  
 {  
 private static IOCContainer mContainer = null;  
  
 // 确保 Container 是有实例的  
 static void MakeSureContainer()  
 {  
 if (mContainer == null)  
 {  
 mContainer = new IOCContainer();  
 Init();  
 }  
 }  
  
 // 这里注册模块  
 private static void Init()  
 {  
 mContainer.Register(new CounterModel());  
 }  
   
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureContainer();  
 return mContainer.Get<T>();  
 }  
 }  
}

PiontGame.cs

namespace FrameworkDesign.Example  
{  
 public class PointGame   
 {  
 private static IOCContainer mContainer = null;  
  
 // 确保 Container 是有实例的  
 static void MakeSureContainer()  
 {  
 if (mContainer == null)  
 {  
 mContainer = new IOCContainer();  
 Init();  
 }  
 }  
  
 // 这里注册模块  
 private static void Init()  
 {  
 mContainer.Register(new GameModel());  
 }  
   
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureContainer();  
 return mContainer.Get<T>();  
 }  
 }  
}

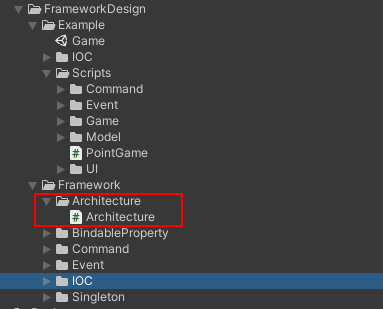
这两个重复的代码，我们需要整理一下。

创建一个类，名字叫 Architecture.cs ，代码如下:

namespace FrameworkDesign  
{  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> where T : Architecture<T>, new()  
 {  
 #region 类似单例模式 但是仅在内部课访问  
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
 }  
 }  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

代码有一点变化，在笔者编写 Architecture 的时候目标很简单，就是想办法把 Init 交给子类去实现，然后给子类提供必备的注册方法，即 Register。

代码所在位置如下：



接着我们将 Architecture 引入到 CounterApp 类和 PointGame 类中，代码如下:

CounterApp.cs

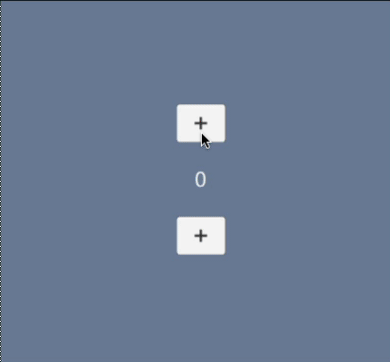
using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 Register(new CounterModel());  
 }  
 }  
}

代码精简了很多，然后再看下 PointGame 类，代码如下:

namespace FrameworkDesign.Example  
{  
 public class PointGame : Architecture<PointGame>  
 {  
 // 这里注册模块  
 protected override void Init()  
 {  
 Register(new GameModel());  
 }  
 }  
}

同样也是精简了很多，不用写很多样板代码了。

接着看下运行结果，如下：





两个项目都没什么变化，说明是没有问题的。

OK，到此我们在《点点点》和 CounterApp 中引入了 IOC 容器，同时我们还得到了一个 Architecture.cs 工具类，收获颇丰。

可能到这里大家觉得引入 IOC 容器之后比直接使用 Singleton 相比麻烦了很多，还有实现一个 CounterApp.cs 或者 PointGame.cs ，甚至每次从 IOC 容器里查询字典也造成了更多的性能消耗，这么做有什么意义呢？

答案很简单，就是仅仅是为了增加模块访问的限制。就是一个模块不可以随便访问的，必须通过项目的 Architecture 类（CounterApp 或 PointGame）获取，仅仅是这一点就可以让项目规避了 Singleton 所带来的风险。

当然引入了 IOC 之后还可以往上增加更多的限制，这些限制是接下来课程中的重点。

还有一点好处就是，我们在统一的地方去注册模块，这样更有利于让开发人员拥有宏观的视角，而统一注册模块的地方可以充当一个“架构草图”，省去了维护一个“架构草图”的麻烦。

除以上好处之外，还有更多的好处比如更符合 SOLID 原则、更好的纸上设计体验、可以增加层级、增加中间件模式、增加事件中心等等，这些会在后边介绍。

这堂课的内容就这些，我们下一堂课再见，拜拜。

# 15. IOC 容器的隐藏功能-注册接口模块

在上一堂课，我们在《点点点》和 CounterApp 中引入了 IOC 容器。

通过引入 IOC 容器，我们提取出来了一个新的工具类，即 Architecture 代码如下：

namespace FrameworkDesign  
{  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> where T : Architecture<T>, new()  
 {  
 #region 类似单例模式 但是仅在内部课访问  
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
 }  
 }  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

代码不难。

可以看得出来 IOC 容器是一个很方便的模块管理工具。

除了可以用来注册和获取模块，IOC 容器一般还会有一个隐藏的功能，即：注册接口模块

什么意思呢？

我们先看下之前写的 IOCExample 代码如下：

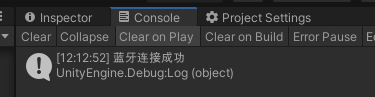
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class IOCExample : MonoBehaviour  
 {  
 void Start()  
 {  
 // 创建一个 IOC 容器  
 var container = new IOCContainer();  
   
 // 注册一个蓝牙管理器的实例  
 container.Register(new BluetoothManager());  
   
 // 根据类型获取蓝牙管理器的实例  
 var bluetoothManager = container.Get<BluetoothManager>();  
   
 //连接蓝牙  
 bluetoothManager.Connect();  
 }  
  
 public class BluetoothManager  
 {  
 public void Connect()  
 {  
 Debug.Log("蓝牙连接成功");  
 }  
 }  
 }  
}

这里 Container 注册了 BluetoothManager 对象。

其实还可以写成如下代码:

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class IOCExample : MonoBehaviour  
 {  
 void Start()  
 {  
 // 创建一个 IOC 容器  
 var container = new IOCContainer();  
   
 // 根据接口注册实例  
 container.Register<IBluetoothManager>(new BluetoothManager());  
   
 // 根据接口获取蓝牙管理器的实例  
 var bluetoothManager = container.Get<IBluetoothManager>();  
   
 //连接蓝牙  
 bluetoothManager.Connect();  
 }  
   
 /// <summary>  
 /// 定义接口  
 /// </summary>  
 public interface IBluetoothManager  
 {  
 void Connect();  
 }  
  
 /// <summary>  
 /// 实现接口  
 /// </summary>  
 public class BluetoothManager : IBluetoothManager  
 {  
 public void Connect()  
 {  
 Debug.Log("蓝牙连接成功");  
 }  
 }  
 }  
}

代码很简单，核心的代码已经写了注释，运行结果如下：



代码运行正确。

而 抽象-实现 这种形式注册和获取对象的方式是符合**依赖倒置原则**的。

依赖倒置原则（Dependence Inversion Principle）：程序要依赖于抽象接口，不要依赖于具体实现。

当然现在是在架构演化阶段，所以先不深入依赖倒置原则。

这里简单说一下符合依赖倒置原则，也就是以 抽象-实现 这种形式注册和获取对象 有什么好处吧。

好处如下：

* 接口设计与实现分成两个步骤，接口设计时可以专注于设计，实现时可以专注于实现。
* 接口设计时专注于设计可以减少系统设计时的干扰。
* 实现是可以替换的，比如一个接口叫 IStorage，其实现可以是 PlayerPrefsStorage、EasySaveStorage、EdtiroPrefsStorage，等切换时候只需要一行代码就可以切换了。
* 比较容易测试（单元测试等）
* 当实现细节（比如 PlayerPrefsStorage）发生变化时，其引用接口（比如 IStorage）类里的代码不会跟着改变，降低耦合。
* 等等

好处有很多，当然不好的地方就是可能要多写一点代码，但是对于大规模项目来说这点代码量来换取更健康的项目状态是很划算的。

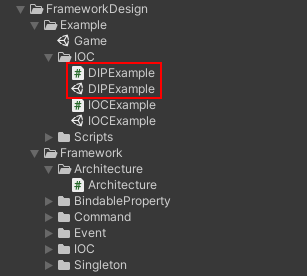
OK，说了这么多理论，接下来写一个小例子，来展示一下这些好处。

这里我们创建一个 DIPExample.cs 脚本，DIP 是依赖倒置原则的意思，代码如下:

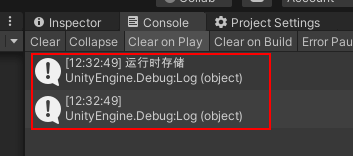
#if UNITY\_EDITOR  
using UnityEditor;  
#endif  
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class DIPExample : MonoBehaviour  
 {  
 // 1. 设计模块接口  
 public interface IStorage  
 {  
 void SaveString(string key, string value);  
 string LoadString(string key, string defaultValue = "");  
 }  
  
 // 2. 实现接口  
 // 运行时存储  
 public class PlayerPrefsStorage : IStorage  
 {  
 public void SaveString(string key, string value)  
 {  
 PlayerPrefs.SetString(key, value);  
 }  
  
 public string LoadString(string key, string defaultValue = "")  
 {  
 return PlayerPrefs.GetString(key, defaultValue);  
 }  
 }  
  
 // 3. 实现接口  
 // 编辑器存储  
 public class EditorPrefsStorage : IStorage  
 {  
 public void SaveString(string key, string value)  
 {  
#if UNITY\_EDITOR  
 EditorPrefs.SetString(key, value);  
#endif  
 }  
  
 public string LoadString(string key, string defaultValue = "")  
 {  
#if UNITY\_EDITOR  
 return EditorPrefs.GetString(key, defaultValue);  
#else  
 return ""  
#endif  
 }  
 }  
  
  
 // 4. 使用  
 private void Start()  
 {  
 // 创建一个 IOC 容器  
 var container = new IOCContainer();  
  
 // 注册运行时模块  
 container.Register<IStorage>(new PlayerPrefsStorage());  
  
 var storage = container.Get<IStorage>();  
   
 storage.SaveString("name", "运行时存储");  
  
 Debug.Log(storage.LoadString("name"));  
   
 // 切换实现  
 container.Register<IStorage>(new EditorPrefsStorage());  
  
 storage = container.Get<IStorage>();  
   
 Debug.Log(storage.LoadString("name"));  
 }  
 }  
}

代码很简单，不管是模块的设计过程，还是切换实现，还是解耦这个示例都有所体现。

代码所在的位置如下：



将此脚本挂到任意 GameObject 上，运行 Unity 结果如下：



结果是没有问题的。

OK，这就是 IOC 容器提供的**注册接口模块** 功能。

依赖倒置原则是 SOLID 中的字母 D。

单一职责原则是 SOLID 中的字母 S。

还剩下：

* 开闭原则（O）
* 里式替换原则（L）
* 接口分离原则（I）

总之离 SOLID 原则的支持更进了一步。

而由于 IOC 容器支持注册接口模块，我们接下来能做的事情更多了，这些我们接下来慢慢学习。

OK，这堂课的内容就到这里，我们下一趟课再见，拜拜。

# 16. 《点点点》和 CounterApp 支持接口模块

在上一堂课，笔者介绍了 IOC 容器的隐藏功能，即：注册接口模块，并且简单介绍了其用法和一些好处。

在这一堂课，我们尝试在《点点点》和 CounterApp 中支持接口模块。

首先是 CounterApp 的 CounterModel 代码如下：

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 ...  
  
 public interface ICounterModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel : ICounterModel  
 {  
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

代码很简单。

然后是 CounterApp.cs 中的注册模块的代码变更，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 Register<ICounterModel>(new CounterModel());  
 }  
 }  
}

接着是 CounterViewController 中获取 CounterModel 的代码，如下：

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = CounterApp.Get<ICounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
 }  
  
 ...  
}

最后是，两个 Command，代码如下：

AddCountCommand.cs

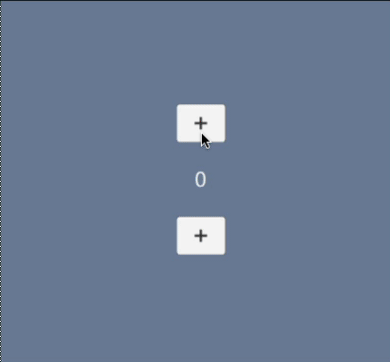
using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct AddCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterApp.Get<ICounterModel>().Count.Value++;  
 }  
 }  
}

SubCountCommand.cs

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct SubCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterApp.Get<ICounterModel>().Count.Value--;  
 }  
 }  
}

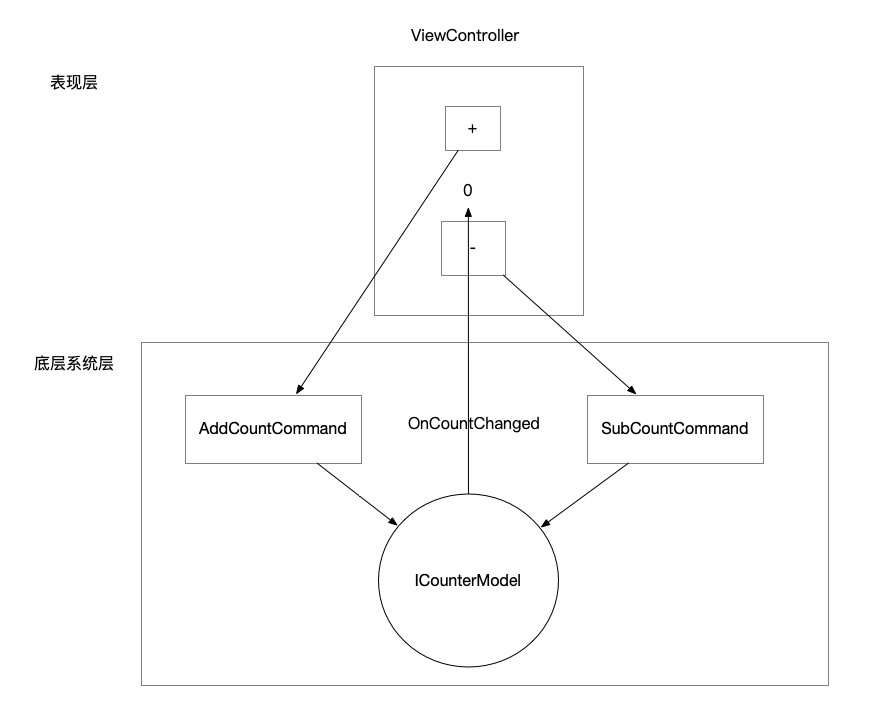
代码都不难，就是更改了下注册方式。

运行结果如下：



没啥变化。

这里我们来展示下好久没有展示过的草图，如下：



非常清晰，改动也非常少，仅仅是从 CounterModel 变成了 ICounterModel，而不要小看这个改动，当我们知道 ICounterModel 是一个接口的时候，我们看这张图时候的心智负担会少很多，因为我们不用考虑具体实现。

接着我们让 《点点点》也支持一下接口模块。

同样，首先是 GameModel 代码如下:

namespace FrameworkDesign.Example  
{  
 public interface IGameModel  
 {  
 BindableProperty<int> KillCount { get; }  
 BindableProperty<int> Gold { get; }  
 BindableProperty<int> Score { get; }  
 BindableProperty<int> BestScore { get; }  
 }  
  
 public class GameModel : IGameModel  
 {  
 public BindableProperty<int> KillCount { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Gold { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Score { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> BestScore { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

然后是 PointGame，代码如下：

namespace FrameworkDesign.Example  
{  
 public class PointGame : Architecture<PointGame>  
 {  
 // 这里注册模块  
 protected override void Init()  
 {  
 Register<IGameModel>(new GameModel());  
 }  
 }  
}

最后是 KillEnemyCommand.cs 代码如下：

namespace FrameworkDesign.Example  
{  
 public struct KillEnemyCommand : ICommand  
 {  
 public void Execute()  
 {  
 var gameModel = PointGame.Get<IGameModel>();  
   
 gameModel.KillCount.Value++;  
   
 if (gameModel.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

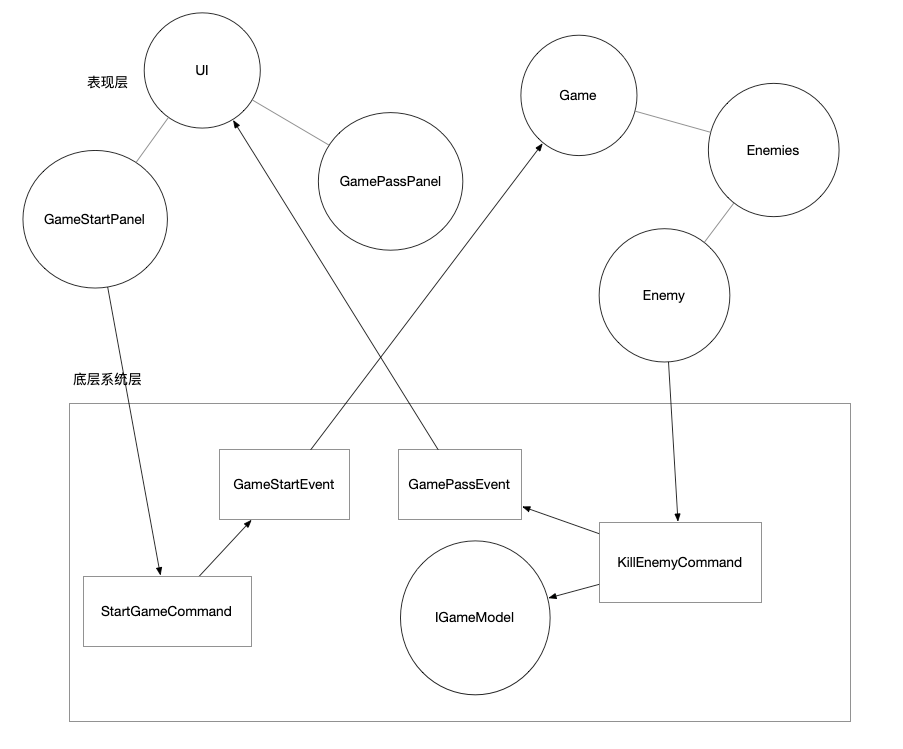
代码不难。

我们运行一下 Unity，结果如下：



没啥问题。

同样，我们更新一下引用关系图，如下：

   
很清晰。

到此，《点点点》和 CounterApp 支持接口模块的任务就完成了。

这堂课的内容就这些，我们下一趟课再见，拜拜。

# 17. CounterApp 支持数据存储

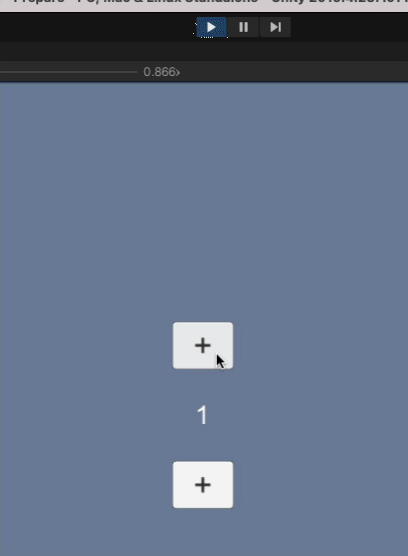
在上一堂课，我们尝试让《点点点》和 CounterApp 支持接口模块。

在这堂课我们尝试在 CounterApp 中引入数据存储功能。

其代码的实现非常简单，只需要将 CounterModel 改成如下即可：

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 ...  
   
 public interface ICounterModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel : ICounterModel  
 {  
 public CounterModel()  
 {  
 Count.Value = PlayerPrefs.GetInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 PlayerPrefs.SetInt("COUNTER\_COUNT", count);  
 };  
 }  
   
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

我们运行一下项目，如下：



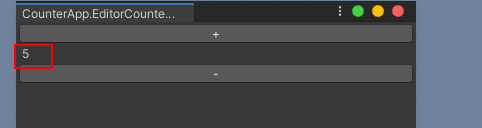
OK，功能实现成功，很简单。

接下来，我们来尝试跑一下原来我们实现的 EditorCounterApp。

这里我们先让 EditorCounterApp 也支持下接口模块，代码如下:

using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 new AddCountCommand()  
 .Execute();  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 new SubCountCommand()  
 .Execute();  
 }  
 }  
 }  
}

代码不难，我们打开一下 EditorCounterApp，如下：



会发现，也是 5，说明功能也支持了。

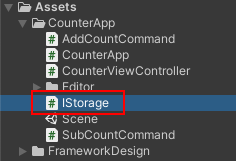
但是这里，我们不想让 EditorCounterApp 和 CounterApp 用同一套存储，而是 EditorCounterApp 使用 EditorPrefs 而 CounterApp 使用 PlayerPrefs。

要实现也不难，我们实现一个 IStorage 接口和对应的实现即可，代码如下:

#if UNITY\_EDITOR  
using UnityEditor;  
#endif  
using UnityEngine;  
  
namespace CounterApp  
{  
 public interface IStorage  
 {  
 void SaveInt(string key, int value);  
  
 int LoadInt(string key, int defaultValue = 0);  
 }  
  
 public class PlayerPrefsStorage : IStorage  
 {  
 public void SaveInt(string key, int value)  
 {  
 PlayerPrefs.SetInt(key, value);  
 }  
  
 public int LoadInt(string key, int defaultValue = 0)  
 {  
 return PlayerPrefs.GetInt(key, defaultValue);  
 }  
 }  
  
 public class EditorPrefsStorage : IStorage  
 {  
 public void SaveInt(string key, int value)  
 {  
#if UNITY\_EDITOR  
 EditorPrefs.SetInt(key, value);  
#endif  
 }  
  
 public int LoadInt(string key, int defaultValue = 0)  
 {  
#if UNITY\_EDITOR  
 return EditorPrefs.GetInt(key, defaultValue);  
#else  
 return 0;  
#endif  
 }  
 }  
}

代码不难，这些我们在 DIPExample 里都实现过了，但是有一点区别，DIPExample 实现的 IStorage 是字符串的存储，而目前的 IStorage 是 int 存储。

代码所在的位置如下：



OK，实现了之后，我们先在 CounterApp 中注册，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 Register<ICounterModel>(new CounterModel());  
 Register<IStorage>(new PlayerPrefsStorage());  
 }  
 }  
}

这里我们默认先注册 PlayerPrefsStorage。

接着，我们更改下 CounterModel 的代码，如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 ...  
  
 public interface ICounterModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel : ICounterModel  
 {  
 public CounterModel()  
 {  
 var storage = CounterApp.Get<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
   
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

代码很简单。就是将原来的 PlayerPrefs 改成 IStorage。

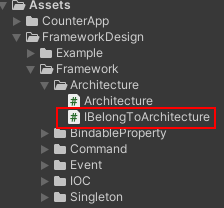
不过这样会有一个很大的问题。

由于我们的 CounterApp 是一个类似单例一样的类，所以在第一次调用 CounterApp.Get 的时候，会调用 CounterApp 的 Init 方法，而 CounterModel 又是在 Init 方法中注册的，所以就会造成无限的递归调用，从而造成堆栈溢出。

而这里笔者为了解决这个问题，会引入一个叫做 IBelongToArchitecture 这样的一个接口，意思是我属于 Architecture，代码如下：

namespace FrameworkDesign  
{  
 public interface IBelongToArchitecture  
 {  
 IArchitecture Architecture { get; set; }  
 }  
}

代码所在位置如下:



代码很简单，注意，此时还没有 IArchitecture 接口，所以我们需要在 Architecture 工具类中声明一个接口，如下：

namespace FrameworkDesign  
{  
  
 public interface IArchitecture  
 {  
 // 提供一个获取 Utility 的 API  
 T GetUtility<T>() where T : class;  
 }  
   
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 ...  
 // 提供一个注册 Model 的 API  
 public void RegisterModel<T>(T instance) where T : IBelongToArchitecture  
 {  
 // 需要给 Model 赋值一下  
 instance.Architecture = this;  
 mContainer.Register<T>(instance);  
 }  
   
 // 新增  
 public T GetUtility<T>() where T : class  
 {  
 return mContainer.Get<T>();  
 }  
 }  
}

由于目前 IArchitecture 仅仅只是为了在 Model 中能够访问 Utility，所以这里提供了一个 GetUtility 这样的一个方法，同时也增加了一个叫做 RegisterModel 方法，是为了在注册 Model 时，Model 与 Architecture 可以互相访问，从而 Model 可以通过 Architure 对象去获取 Utility，而不是通过 CounterApp 去获取 Utility。

OK，这样的话，就能避免堆栈溢出了。

我们把 Model 的代码改下，如下：

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 ...  
 // 继承 IBelongToArchitecture  
 public interface ICounterModel : IBelongToArchitecture  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
  
 public class CounterModel : ICounterModel  
 {  
 public CounterModel()  
 {  
 // 通过 Architrecture 获取  
 var storage = Architecture.GetUtility<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
   
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public IArchitecture Architecture { get; set; }  
 }  
}

接着更改下 CounterApp.cs 的代码，如下：

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 RegisterModel<ICounterModel>(new CounterModel());  
   
 Register<IStorage>(new PlayerPrefsStorage());  
 }  
 }  
}

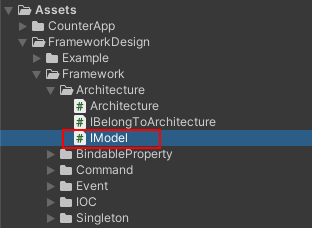
代码很简单。

不过这里还是有一个很严重的问题，就是在 CounterModel 的构造中获取 Architecture 是不允许的，因为此时还没有对 Architecture 进行赋值，而对 Architecture 进行赋值是在注册完模块之后进行的，所以这里还是有问题。

这里笔者的解决方案是再引入一个 IModel 接口，代码如下:

namespace FrameworkDesign  
{  
 public interface IModel : IBelongToArchitecture  
 {  
 void Init();  
 }  
}

代码所在的位置如下:



对应的，我们需要更改下 Architecture.cs 的代码，如下:

using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
  
 public interface IArchitecture  
 {  
 // 提供一个获取模块的 API  
 T GetUtility<T>() where T : class;  
 }  
   
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 /// <summary>  
 /// 是否已经初始化完成  
 /// </summary>  
 private bool mInited = false;  
   
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
   
 // 提供一个注册 Model 的 API  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.Architecture = this;  
 mContainer.Register<T>(instance);  
   
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
   
 #region 类似单例模式 但是仅在内部课访问  
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
   
  
 public T GetUtility<T>() where T : class  
 {  
 return mContainer.Get<T>();  
 }  
 }  
}

这里引入了，一个 Model 初始化的机制，分别有判断是否已经初始化的 mInited 变量，还有用于记录将要初始化的 Model 的 mModels List，还增加了初始化逻辑以及相应的判断等。

增加了这些代码让 Architecture 复杂了很多，不过没关系，这些代码不难。

到此我们的 IModel 算是实现完了。

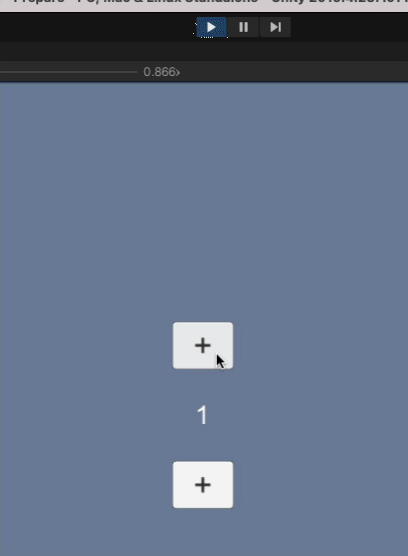
接下来，我们来尝试使用一下 IModel。

首先 CounterModel 的代码改成如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 ...  
  
 public interface ICounterModel : IModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel : ICounterModel  
 {  
 public void Init()  
 {  
 // 通过 Architrecture 获取  
 var storage = Architecture.GetUtility<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
   
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public IArchitecture Architecture { get; set; }  
  
 }  
}

代码中用 Init 替代了原来的构造，这个时候就可以保证 Architecture 是已经有值的了。

到此，我们终于可以看运行结果了，结果如下：



没啥问题。

接下来，我们来实现 EditorCounterApp，代码如下:

using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 // 需要在这里切换一下 Storage 的实现  
 CounterApp.RegisterUtility<IStorage>(new EditorPrefsStorage());  
   
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 new AddCountCommand()  
 .Execute();  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 new SubCountCommand()  
 .Execute();  
 }  
 }  
 }  
}

代码中有一个地方是需要切换 IStorage 的实现，但是此时我们还没有提供 RegisterUtility 这样的方法。

所以我们接下来在 Architecture 实现 RegisterUtility 这个方法即可，代码如下:

using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance);  
   
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class;  
 }  
   
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 /// <summary>  
 /// 是否已经初始化完成  
 /// </summary>  
 private bool mInited = false;  
   
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
   
 // 提供一个注册 Model 的 API  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.Architecture = this;  
 mContainer.Register<T>(instance);  
   
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
   
 #region 类似单例模式 但是仅在内部课访问  
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public static void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
   
 public void RegisterUtility<T>(T instance)  
 {  
 mContainer.Register<T>(instance);  
 }  
   
 public T GetUtility<T>() where T : class  
 {  
 return mContainer.Get<T>();  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

代码变化很多，首先是对外提供了一个静态的注册模块 API，即 Register，Register 原来是非静态的。然后提供了 RegisterUtility 成员方法，以提供在 CounterApp 类的内部使用。

代码不难。

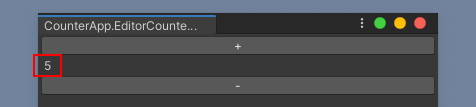
我们接着更改一下 CounterApp 的代码，改成如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 RegisterModel<ICounterModel>(new CounterModel());  
   
 RegisterUtility<IStorage>(new PlayerPrefsStorage());  
 }  
 }  
}

然后再修改下 EditorCoutnerApp 的代码，如下:

using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 // 需要在这里切换一下 Storage 的实现  
 CounterApp.Register<IStorage>(new EditorPrefsStorage());  
   
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 new AddCountCommand()  
 .Execute();  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 new SubCountCommand()  
 .Execute();  
 }  
 }  
 }  
}

我们运行一下 EditorCounterApp，结果如下:



这里发现还是 5，说明是有问题的。

这是因为在 CounterModel 的初始化调用是在第一次使用 CounterApp 类的时候，而我们在调用 CounterApp.RegisterUtility 的时候，已经调用完 CounterModel 的初始化了。

所以在 CounterModel 初始化时获取的 IStorage，其实还是 PlayerPrefsStorage。

代码如下:

public class CounterModel : ICounterModel  
 {  
 public void Init()  
 {  
 // 此时还是 IStorage  
 var storage = Architecture.GetUtility<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
   
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public IArchitecture Architecture { get; set; }  
  
 }

在这里，我们就需要在 Architecture 的 注册模块之后 与 调用模块初始化 之前对外提供一个委托，用于做一些配置。

代码如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance);  
   
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class;  
 }  
   
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 ...  
 #region 类似单例模式 但是仅在内部课访问  
  
 /// <summary>  
 /// 增加注册  
 /// </summary>  
 public static Action<T> OnRegisterPatch = architecture=> { };  
   
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 调用  
 OnRegisterPatch?.Invoke(mArchitecture);  
   
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
 ...  
 }  
}

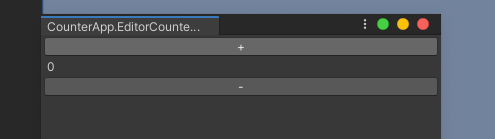
代码不难。

接下来修改下 EditorCounterApp 代码如下：

using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 // 需要在这里切换一下 Storage 的实现  
 CounterApp.OnRegisterPatch += architecture =>  
 {  
 architecture.RegisterUtility<IStorage>(new EditorPrefsStorage());  
 };  
   
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 new AddCountCommand()  
 .Execute();  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 new SubCountCommand()  
 .Execute();  
 }  
 }  
 }  
}

这样的话，就应该能够正常运行了。

打开 EditorCounterApp 结果如下:



此时是正确的。

再打开 CounterApp 结果如下:



结果是正确的。

这样的话，CounterApp 增加存储功能就实现好了。

虽然过程很坎坷，代码修改量也比较多，但是这些代码都是一些常规的逻辑代码。每一步的代码修改都对应着要解决的问题。

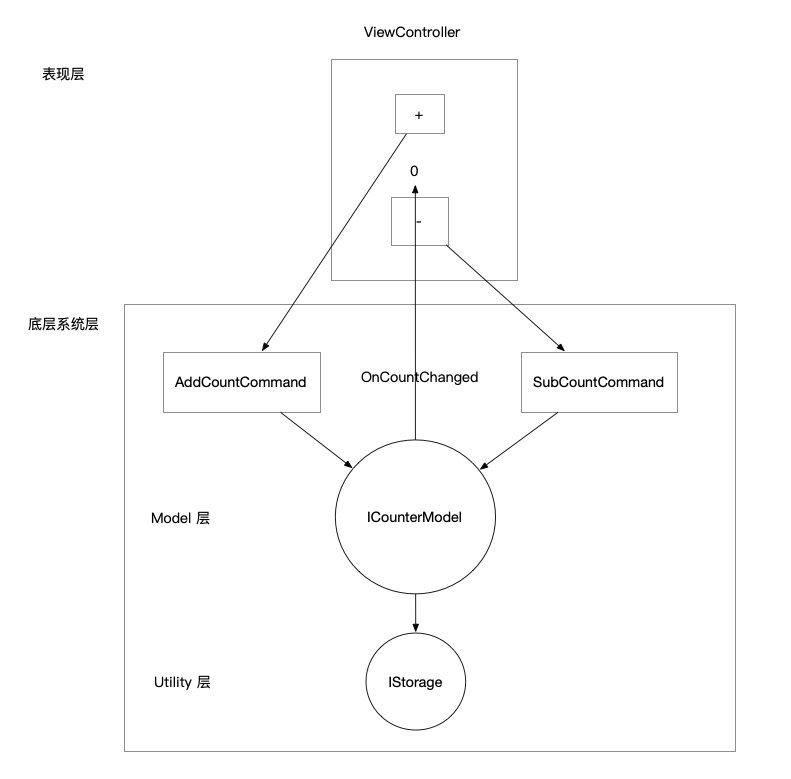
当然虽然这一篇文章就包含了几次修改，但是在现实中，笔者是花了很久才一点点摸索出以上的修改方案的。

这里还要提一点，如果在 CounterApp 内部中使用宏来区分平台的话会简单很多，就不用像上边的代码绕来绕去了。

但是宏是有风险的，尤其是在重构的过程中，而架构的演化是要一直进行重构的，所以笔者就不怎么使用宏来做框架的设计。

OK，总之，我们的 Architecture 工具类变得更强大了一点，同时几乎对 CounterApp 和 《点点点》的项目代码没什么影响。

在架构层面，我们引入了一个 Utility 层，而现在的 CounterApp 图 则变成如下：

   
同样，在这张图上，我们只需要考虑接口即可，不需要考虑具体的实现。

OK，这堂课的内容就这些，我们下一趟课再见，拜拜。

# 18. 接口的阉割技术

在上一堂课，我们实现了 CounterApp 对数据存储的支持，并且让 EditorCounterApp 和 CounterApp 分别使用不同的存储 API。

由于 IOC 容器支持接口模块，所以我们目前阶段的架构有了更好的兼容性和测试性，而引入了 Utility 层，使得当前的架构的代码复用性更高，可配置性更强。

在这一堂课，我们先喘口气休息一下，不去接着演化当前的架构了，而是学习一个 C# 的设计技巧，这个技巧笔者起名为接口的阉割。

其原理很简单，就是利用 C# 接口的显示实现，来达到接口方法在子类阉割的目的，而要想调用这个方法必须通过接口而不是方法所在的对象。

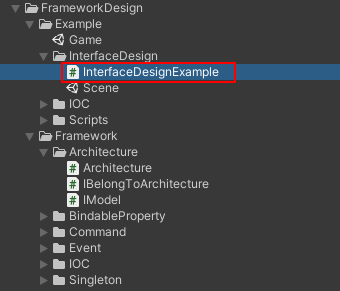
直接看代码就知道了。

我们来写一个示例，代码如下:

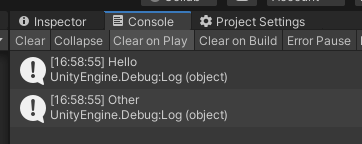
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 /// <summary>  
 /// 1. 定义接口  
 /// </summary>  
 public interface ICanSayHello  
 {  
 void SayHello();  
 void SayOther();  
 }  
   
 public class InterfaceDesignExample : MonoBehaviour,ICanSayHello  
 {  
 /// <summary>  
 /// 接口的隐式实现  
 /// </summary>  
 public void SayHello()  
 {  
 Debug.Log("Hello");  
 }  
  
 /// <summary>  
 /// 接口的显式实现  
 /// </summary>  
 void ICanSayHello.SayOther()  
 {  
 Debug.Log("Other");  
 }  
   
 // Start is called before the first frame update  
 void Start()  
 {  
 // 隐式实现的方法可以直接通过对象调用  
 this.SayHello();  
   
 // 显式实现的接口不能通过对象调用  
 // this.SayOther() // 会报编译错误  
   
 // 显式实现的接口必须通过接口对象调用  
 (this as ICanSayHello).SayOther();  
 }  
 }  
}

代码不难，大家读注释即可。

代码所在的位置如下:



运行的结果如下：



我们在看下代码，如下：

using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 /// <summary>  
 /// 1. 定义接口  
 /// </summary>  
 public interface ICanSayHello  
 {  
 void SayHello();  
 void SayOther();  
 }  
   
 public class InterfaceDesignExample : MonoBehaviour,ICanSayHello  
 {  
 /// <summary>  
 /// 接口的隐式实现  
 /// </summary>  
 public void SayHello()  
 {  
 Debug.Log("Hello");  
 }  
  
 /// <summary>  
 /// 接口的显式实现  
 /// </summary>  
 void ICanSayHello.SayOther()  
 {  
 Debug.Log("Other");  
 }  
   
 // Start is called before the first frame update  
 void Start()  
 {  
 // 隐式实现的方法可以直接通过对象调用  
 this.SayHello();  
   
 // 显式实现的接口不能通过对象调用  
 // this.SayOther() // 会报编译错误  
   
 // 显式实现的方法必须通过接口对象调用  
 (this as ICanSayHello).SayOther();  
 }  
 }  
}

这个例子是展示了接口的显式实现和隐式实现的特性。

如果要想调用一个对象的显式实现的方法，那么必须要将此对象强制转换成接口对象才能调用，这样就增加了调用显式实现的方法的成本，所以可以理解为这个方法被阉割了。

而这种技巧，笔者一般会在两种情况下使用，一种是接口-抽象类-实现类的情况，另一种就是接口-静态扩展来实现方法的访问规则的时候。

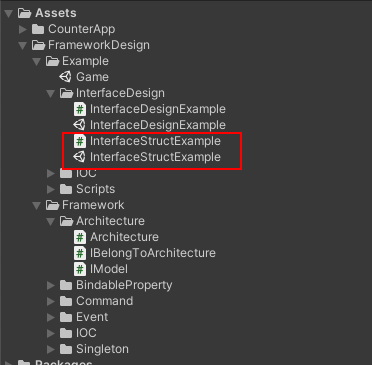
这两种情况，笔者都写个示例实现一下。

第一种就是比较常见的 接口-抽象类-实现类的结构，示例代码如下：

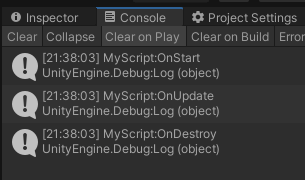
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class InterfaceStructExample : MonoBehaviour  
 {  
 /// <summary>  
 /// 接口  
 /// </summary>  
 public interface ICustomScript  
 {  
 void Start();  
 void Update();  
 void Destroy();  
 }  
   
 /// <summary>  
 /// 抽象类  
 /// </summary>  
 public abstract class CustomScript : ICustomScript  
 {  
 protected bool mStarted { get; private set; }  
 protected bool mDestroyed { get; private set; }  
   
 /// <summary>  
 /// 不希望子类访问 Start 方法，因为有可能破坏状态  
 /// </summary>  
 void ICustomScript.Start()  
 {  
 OnStart();  
 mStarted = true;  
 }  
   
 void ICustomScript.Update()  
 {  
 OnUpdate();  
 }  
  
 void ICustomScript.Destroy()  
 {  
 OnDestroy();  
 mDestroyed = true;  
 }  
   
 /// <summary>  
 /// 希望子类实现 OnStart 方法  
 /// </summary>  
 protected abstract void OnStart();  
 protected abstract void OnUpdate();  
 protected abstract void OnDestroy();  
 }  
   
 /// <summary>  
 /// 由用户扩展的类  
 /// </summary>  
 public class MyScript : CustomScript  
 {  
 protected override void OnStart()  
 {  
 Debug.Log("MyScript:OnStart");  
 }  
  
 protected override void OnUpdate()  
 {  
 Debug.Log("MyScript:OnUpdate");  
 }  
  
 protected override void OnDestroy()  
 {  
 Debug.Log("MyScript:OnDestroy");  
 }  
 }  
  
 /// <summary>  
 /// 测试脚本  
 /// </summary>  
 private void Start()  
 {  
 ICustomScript script = new MyScript();  
 script.Start();  
 script.Update();  
 script.Destroy();  
 }  
 }  
}

代码不难，这是一种比较常见的 接口-抽象类-实现类 的继承结构。

代码所在的位置如下：



将此脚本挂在任意 GameObject 上，运行 Unity 结果如下：



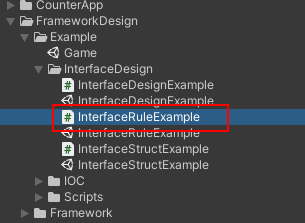
接着我们看第二种情况，即接口 + 扩展时用于限制方法的访问规则。

我们同样写一个例子，代码如下：

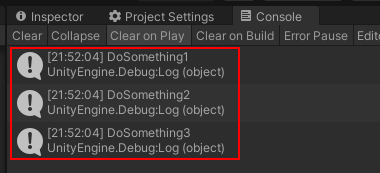
using UnityEngine;  
  
namespace FrameworkDesign.Example  
{  
 public class CanDoEveryThing   
 {  
 public void DoSomething1()  
 {  
 Debug.Log("DoSomething1");  
 }  
  
 public void DoSomething2()  
 {  
 Debug.Log("DoSomething2");  
 }  
  
 public void DoSomething3()  
 {  
 Debug.Log("DoSomething3");  
 }  
 }  
  
 public interface IHasEveryThing  
 {  
 CanDoEveryThing CanDoEveryThing { get; }  
 }  
  
 public interface ICanDoSomething1 : IHasEveryThing  
 {  
   
 }  
  
 public static class ICanDoSomeThing1Extensions  
 {  
 public static void DoSomething1(this ICanDoSomething1 self)  
 {  
 self.CanDoEveryThing.DoSomething1();  
 }  
 }  
  
 public interface ICanDoSomething2 : IHasEveryThing  
 {  
   
 }  
   
 public static class ICanDoSomeThing2Extensions  
 {  
 public static void DoSomething2(this ICanDoSomething2 self)  
 {  
 self.CanDoEveryThing.DoSomething2();  
 }  
 }  
  
 public interface ICanDoSomething3 : IHasEveryThing  
 {  
  
 }  
   
 public static class ICanDoSomeThing3Extensions  
 {  
 public static void DoSomething3(this ICanDoSomething3 self)  
 {  
 self.CanDoEveryThing.DoSomething3();  
 }  
 }  
  
 public class InterfaceRuleExample : MonoBehaviour  
 {  
 public class OnlyCanDo1 : ICanDoSomething1  
 {  
 CanDoEveryThing IHasEveryThing.CanDoEveryThing { get; } = new CanDoEveryThing();  
 }  
   
 public class OnlyCanDo23 : ICanDoSomething2,ICanDoSomething3  
 {  
 CanDoEveryThing IHasEveryThing.CanDoEveryThing { get; } = new CanDoEveryThing();  
 }  
  
 private void Start()  
 {  
 var onlyCanDo1 = new OnlyCanDo1();  
 // 可以调用 DoSomething1  
 onlyCanDo1.DoSomething1();  
 // 不能调用 DoSomething2 和 3 会报编译错误  
 // onlyCanDo1.DoSomething2();  
 // onlyCanDo1.DoSomething3();  
  
  
 var onlyCanDo23 = new OnlyCanDo23();  
 // 不可以调用 DoSomething1 会报编译错误  
 // onlyCanDo23.DoSomething1();  
 // 可以调用 DoSomething2 和 3  
 onlyCanDo23.DoSomething2();  
 onlyCanDo23.DoSomething3();  
 }  
 }  
}

代码涉及到静态扩展这个 C# 语法概念，这个如果大家不熟悉需要自行攻克下。

OK，代码所在位置如下：



讲此脚本挂在任意 GameObject 上，运行 Unity，结果如下：



结果是没啥问题。

OK，到此接口的阉割技术就介绍完了。

这里简单总结一下：

* 接口的阉割技术核心的知识是接口的显式实现
* 接口+抽象类+具体类 时，不想被乱调用一些方法时可以用接口阉割技术。
* 接口+静态扩展时，想要通过实现某个接口来获得具体方法的访问权限时可以用接口阉割技术。

而两种常见的接口阉割的情况我们接下来在演化阶段的时候会遇到。

接口的阉割笔者认为是和 C# 的 public、private、internal、protected 关键字的用途是一样的，都是用于控制 C# 成员方法的访问权限，而接口的阉割技术则是以上 4 种控制访问权限之外的第五种。

当然接口的显式声明这个 C# 特性并不只是为了用作阉割使用，它有它本来的用途，就是当需要实现多个签名一致的方法时，可以通过接口的显式声明来区分到底哪个方法是属于哪个接口的。

当然本来的目的大家自行查阅即可。

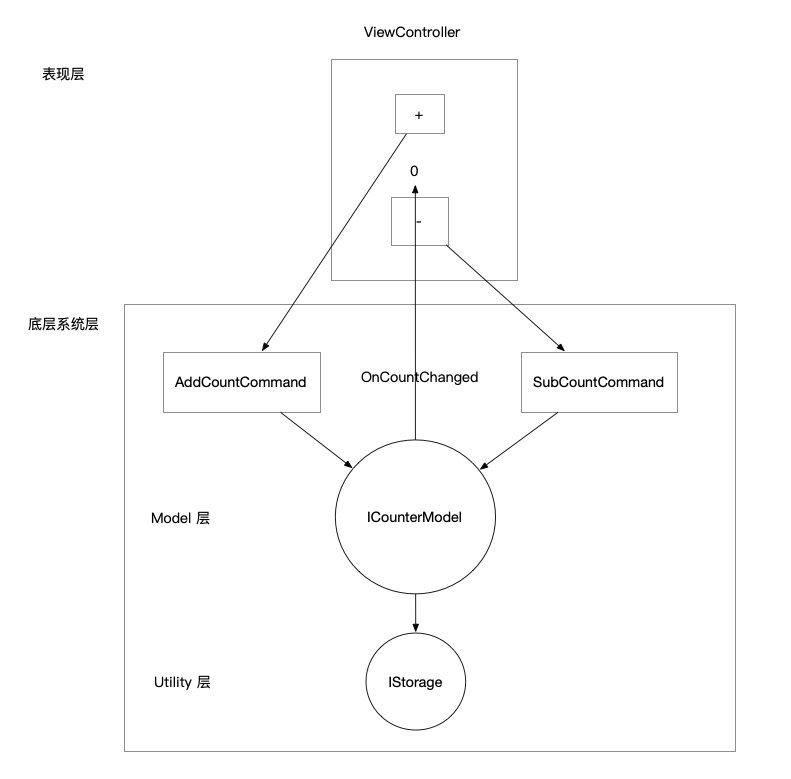
这堂课的内容就这些，我们下一堂课再见，拜拜。

# 19. 引入 System 层

在上一堂课，笔者介绍了接口阉割技术和两种应用接口阉割技术的情况。

在这一堂课我们来对当前的架构做一个梳理。

我们先看下 CounterApp 的草图，如下：



目前从这张图中，我们可以将整个 App 分为三个层级，如下：

* 表现层：即 ViewController 或者 MonoBehaviour 脚本等
* Model 层：管理数据，提供数据的增删改查
* Utility 层：工具层，提供一些必备的基础工具，比如数据存储、网络链接、蓝牙、序列化反序列化等

目前是有三个层级了，而 QFramework 系统设计架构 最终是有四个层级，而第四个层级其实叫做 System 层，即系统层。

接下来笔者就介绍一下 System 层。

首先我们在最开始只有表现层的代码，即 ViewController 的代码。

然后再多个 ViewController 可能会出现数据需要共享的情况，所以我们主张数据和表现要分离，所以就单独把数据部分提取了出来，这部分提取出来的代码我们放在了 Model 层。

而一旦有了 Model 层，就需要考虑对象之间的交互这样的问题，也就是 ViewController 对象 和 Model 对象如何交互。

在考虑时引入了两个概念，即表现逻辑和交互逻辑，表现逻辑就是当 Model 数据变更时，视图来显示变更后的数据，也就是 Model->View，而交互逻辑则是接收用户输入然后改变 Model 数据，也就是 View->Model。

在这里表现逻辑我们可以用 方法调用、委托、事件三种方式实现，而经过比较委托和事件是更合适的。

而交互逻辑并没有尝试方法调用、委托、和事件三种方式，而是直接引入了 Command，因为使用 Command 是可以减少 Controller 的负担的，而 Command 也是大多数 MVC 框架的选择。

引入 Command 在加上分了 Model 和 ViewController 两个层级之后，会发现系统的跨平台性提高了很多，我们可以在不用改变底层代码的情况下去把一个 App 移植到编辑器版本。

到这里，我们的对象之间的交互专题就完成了。

在这个阶段，我们的 Command 是分担了 Controller 的交互逻辑的部分逻辑，当然并不是所有的交互逻辑都适合用 Command 来分担的，还有一部分交互逻辑是需要交给 System 层来分担。

首先我们要理解一个事情，即 Command 是没有状态的，因为 Command 相当于是一个方法，只要调用然后执行一次就可以不用了，所以 Command 是没有状态的。

那么什么是有状态的呢？

有很多，比如我们创建了一个角色控制脚本，在这个脚本里创建了一些角色的数据，伪代码如下:

public class PlayerController : MonoBehaviour  
{  
 public bool Walking;  
 public int JumpCount;  
 public int MaxJumpCountl;  
   
 ...  
}

这个角色控制脚本很明显是有状态的，有没有状态我们可以理解为这个对象需不需要维护数据。

比如，我们的《点点点》中的 GameModel 和 CounterApp 中的 CounterModel 是有状态的，因为它们都维护数据。

而 Command 是没有状态的，因为 Command 是不需要维护数据的，我们看一下伪代码，如下:

public class DoSomethingCommand : ICommand  
{  
 private readonly int mCount;  
   
 public DoSomethingCommand(int count)  
 {  
 mCount = count;  
 }  
   
 public void Execute()  
 {  
 for(int i = 0;i < mCount;i++)  
 {  
 Debug.Log("DoSomeThing")  
 }  
 }  
}

以上代码虽然有个成员变量 mCount，但是这个 mCount 只负责从构造中接收参数，然后再 Execute 方法执行完之后就没啥用处了，所以 DoSomethingCommand 是没有状态的，因为他的成员变量仅仅是用作数据接收，然后再 Execute 中使用。

到这里一个对象有没有状态应该是比较容易分辨了。

接着我们再看一个例子，比如 CounterApp 中的 AddCountCommand 代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public struct AddCountCommand : ICommand  
 {  
 public void Execute()  
 {  
 CounterApp.Get<ICounterModel>().Count.Value++;  
 }  
 }  
}

AddCountCommand 只更改了 ICountModel 中的 Count 的值。

但是现实情况下，Command 除了要更改 Model 的值，还有可能是开启了某些服务，而这些服务是有状态的，比如计时服务，或者蓝牙链接服务等，很明显这些服务如果写成对象他们肯定都是有状态的对象，而这些服务既包含要维护的数据，也包含一些对外提供的方法，那么把这些服务归类为 Model 层 或者 Utility 层都是不合适的，而直接把这些服务的逻辑分撒在表现层实现，也会造成大量的重复代码，而 Command 仅仅是表现层和底层系统层交互方式的一种，而且不能有状态，所以这部分代码就只能再创建一个层级来负责了，而这个层级就是 System 层。如果我们把蓝牙服务和计时器服务用 System 层实现的话，那就应该是 BluetoothSystem 和 TimerSystem。

BluetoothSystem 伪代码可能如下:

public interface IBluetoothSystem : ISystem  
{  
 bool Connected {get;set;}  
 void ConnectCar();  
 void DisconnectCar();  
}  
  
public class BluetoothSystem : IBluetoothSystem  
{  
 public bool Connected {get;set;}  
   
 private IBLEConnection mConnection;  
   
 public void ConnectCar()  
 {  
 var bluetooth = Architecture.GetUtility<IBluetoothUtility>();  
 mConnection = bluetooth.Connect("qoskal");  
 // ...  
 Connected = true;  
 }  
   
 public void DisconnectCar()  
 {  
 var bluetooth = Architecture.GetUtility<IBluetoothUtility>();  
   
 if (mConnection != null) {  
 mConnection.Disconnect()  
 mConnection = null;  
 }  
 // ...  
 Connected = false;  
 }  
   
 IArchitecture Architecture {get;set;}  
}

以上是伪代码，真实代码会复杂一些，不过用于介绍 System 层来说足够了。

而在表现层如果想要使用 IBluetoothSystem 的话，还是按照管理，需要写两个 Command，代码如下;

public class ConnectCarCommand : ICommand  
{  
 public void Execute()  
 {  
 var bluetoothSystem = Architecture.GetSystem<IBluetoothSystem>();  
 bluetoothSystem.ConnectCard();  
   
 }  
   
 IArchitecture Architecture {get;set;}  
}  
  
public class ConnectCarCommand : ICommand  
{  
 public void Execute()  
 {  
 var bluetoothSystem = Architecture.GetSystem<IBluetoothSystem>();  
 bluetoothSystem.DisconnectCard();  
   
 }  
   
 IArchitecture Architecture {get;set;}  
}

而如果想监听 IBluetoothSystem 的连接状态是否变更，则可以让 IBluetoothSystem 的 Connected 变量使用 BindableProperty，代码如下:

public interface IBluetoothSystem : ISystem  
{  
 BindableProperty<bool> Connected {get;set;}  
 void ConnectCar();  
 void DisconnectCar();  
}  
  
public class BluetoothSystem : IBluetoothSystem  
{  
 public BindableProperty<bool> Connected {get;set;} = new BindableProperty<bool>() {  
 Value = false;  
 }  
   
 private IBLEConnection mConnection;  
   
 public void ConnectCar()  
 {  
 var bluetooth = Architecture.GetUtility<IBluetoothUtility>();  
 mConnection = bluetooth.Connect("qoskal");  
 // ...  
 Connected = true;  
 }  
   
 public void DisconnectCar()  
 {  
 var bluetooth = Architecture.GetUtility<IBluetoothUtility>();  
   
 if (mConnection != null) {  
 mConnection.Disconnect()  
 mConnection = null;  
 }  
 // ...  
 Connected = false;  
 }  
   
 IArchitecture Architecture {get;set;}  
}

当然以上这些都是伪代码，这些伪代码只是为了说明引入 System 层是有一些必要的，虽然 System 层的利用率针对不同的项目是不一样的，有可能使用率很高，也可能根本用不上，但是我们在架构层面是需要提供这么一个层级的。

OK，引入 System 层之后，我们再次梳理一下当前的架构如下：

* 表现层：即 ViewController 或者 MonoBehaviour 脚本等
* System 层：系统层，有状态，负责即提供 API 又有状态的对象，比如网络服务、蓝牙服务、商城系统等。
* Model 层：管理数据，有状态，提供数据的增删改查。
* Utility 层：工具层，无状态，提供一些必备的基础工具，比如数据存储、网络链接、蓝牙、序列化反序列化等。

这样，我们的 QFramework 的系统设计架构中的四个层级就都接触到了。

然后表现层改变 System、Model 层级的状态用 Command，System 层 和 Model 层 通知 表现层用事件或者委托，表现层查询状态时可以直接获取 System 和 Model 层。

一般情况下，表现层根本不用访问 Utility 层，同样 Utility 层也不用通知表现层，除非 Utility 层设计得有问题。

OK，到此我们的架构部分的规范差不多都制定好了，剩下的事情就是要把实现架构的工具类都进行强化。

而我们到此也全部接触到了模块化的三种方式：

* 单例
* IOC 容器（主从结构）
* 分层

而我们的 QFramework 系统设计架构是三种方式都结合在了一起使用。

OK，关于架构的梳理部分就介绍到这里。

这一堂课的内容就这些，我们下一趟课再见，拜拜。

# 20. 实现 System 层的支持

在上一堂课，我们引入了 System 层。

Model 层负责管理数据，Utility 层负责提供工具 API，既提供 API 又有状态的代码我们放在 System 层。

System 层虽然我们在理论上进行了引入，但是在实现层面并没有对其进行实现，所以在这一堂课，我们来实现 Architecture 对 System 层的支持。

首先与 Model 一样，我们也定义一个接口叫做 ISystem，代码如下:

namespace FrameworkDesign  
{  
 public interface ISystem : IBelongToArchitecture  
 {  
 void Init();  
 }  
}

代码不难，与 IModel 的实现差不多。

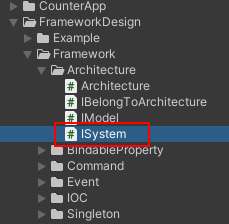
定义完接口之后，我们需要做的就是让 Architecture 对 ISystem 进行支持，支持的代码和之前支持 IModel 的差不多，这里直接贴出代码，如下：

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册系统  
 /// </summary>  
 void RegisterSystem<T>(T instance) where T : ISystem; // 新增  
   
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance);  
   
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class;  
 }  
   
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 /// <summary>  
 /// 是否已经初始化完成  
 /// </summary>  
 private bool mInited = false;  
   
 /// <summary>  
 /// 用于初始化的 Systems 的缓存  
 /// </summary>  
 private List<ISystem> mSystems = new List<ISystem>(); // 新增  
   
 // 提供一个注册 Model 的 API  
 public void RegisterSystem<T>(T instance) where T : ISystem // 新增  
 {  
 // 需要给 Model 赋值一下  
 instance.Architecture = this;  
 mContainer.Register<T>(instance);  
   
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mSystems.Add(instance);  
 }  
 }  
   
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.Architecture = this;  
 mContainer.Register<T>(instance);  
   
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
   
 #region 类似单例模式 但是仅在内部课访问  
  
 /// <summary>  
 /// 注册补丁  
 /// </summary>  
 public static Action<T> OnRegisterPatch = architecture=> { };  
   
 private static T mArchitecture = null;  
   
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 调用  
 OnRegisterPatch?.Invoke(mArchitecture);  
   
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
   
 // 初始化 System  
  
 foreach (var architectureSystem in mArchitecture.mSystems) // 新增  
 {  
 architectureSystem.Init();  
 }  
   
 // 清空 System  
 mArchitecture.mSystems.Clear(); // 新增  
   
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public static void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
   
 public void RegisterUtility<T>(T instance)  
 {  
 mContainer.Register<T>(instance);  
 }  
   
 public T GetUtility<T>() where T : class  
 {  
 return mContainer.Get<T>();  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

Architecture 的代码现在变得有点多了，代码中的新增部分笔者已经通过注释标记出来了，代码不难。

这里要注意一点，由于 Model 层是比 System 层更底层的，所以 System 层是可以直接访问 Model 层的，所以这里需要确保 Model 的初始化是在 System 初始化之前进行。

代码所在的位置如下：



OK，这样，Architecture 对 ISystem 的支持就完成了。

不过，虽然我们实现了 ISystem 的支持，但是我们要考虑到，在 System 层中有可能直接访问 Model 层，所以我们需要提供一个能够通过 IArchitecture 获取 Model 的方法，  
也就是 GetModel 方法，代码如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册系统  
 /// </summary>  
 void RegisterSystem<T>(T instance) where T : ISystem;  
  
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance);  
  
  
 /// <summary>  
 /// 获取 Model  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 /// <returns></returns>  
 T GetModel<T>() where T : class, IModel; // 新增  
  
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class;  
 }  
  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 ...  
  
 public T GetModel<T>() where T : class, IModel  
 {  
 return mContainer.Get<T>();  
 }  
  
 ...  
 }  
}

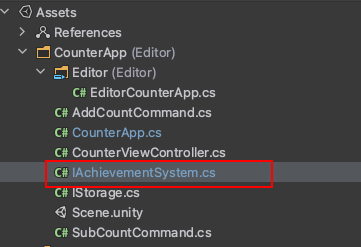
这样，在 System 获取 Model 的功能就实现好了。

接下来，我们来实践一下新的功能，我们尝试在 CounterApp 中实现一个简单的成就系统，成就系统的名字叫 AchievementSystem，代码如下:

using System.Collections;  
using System.Collections.Generic;  
using CounterApp;  
using UnityEngine;  
  
namespace FrameworkDesign  
{  
 public interface IAchievementSystem : ISystem  
 {  
  
 }  
  
 public class AchievementSystem : IAchievementSystem  
 {  
 public IArchitecture Architecture { get; set; }  
  
 public void Init()  
 {  
 var counterModel = Architecture.GetModel<ICounterModel>();  
  
 var previousCount = counterModel.Count.Value;  
  
 counterModel.Count.OnValueChanged += newCount =>  
 {  
 if (newCount >= 10 && previousCount < 10)  
 {  
 Debug.Log("解锁 10 次点击成就");  
 }  
 else if (newCount >= 20 && previousCount < 20)  
 {  
 Debug.Log("解锁 20 次点击成就");  
 }  
  
 previousCount = newCount;  
 };  
 }  
 }  
}

代码非常简单，这里只是将解锁的成就输出了一下，实际上可以改成给表现层发送解锁成绩的事件，但是为了课程的篇幅这里就先用输出日志来简化一下。

代码所在的位置如下:



我们将 IAchievementSystem 注册到 CounterApp 中，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class CounterApp : Architecture<CounterApp>  
 {  
 protected override void Init()  
 {  
 RegisterSystem<IAchievementSystem>(new AchievementSystem());  
  
 RegisterModel<ICounterModel>(new CounterModel());  
   
 RegisterUtility<IStorage>(new PlayerPrefsStorage());  
 }  
 }  
}

代码不难。

然后运行一下 CounterApp，当笔者点击到 10 次和 20 次的时候，输出的结果如下：



OK，到此，我们的简化版本的成就系统就实现完了，这堂课的内容就这些，我们下一趟课再见拜拜。

# 21. 表现层的 IController 接口定义与实现

在上一堂课，我们实现了 System 层，并且尝试实现了一个简单的成就系统。

目前，我们已经定义并实现了给用户使用的几个接口，如下：

* ICommand
* ISystem
* IModel

除了以上三个接口，用户还需要两个接口，一个是表现层的 IController，一个是工具层的 IUtility，然后 ICommand 接口目前还不是很够用，因为在 Command 接口里，还是需要通过单例的方式获取 Architecture 对象。

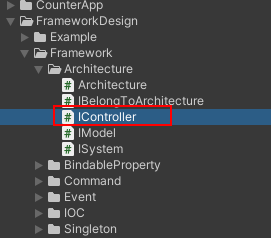
在这一堂课，我们来实现这些接口。

首先第一个接口是 IController，IController 与 MVC 中的 Controller 是一个意思。

我们知道在表现层 需要向底层系统层 发送 Command，也需要监听底层系统层发过来的事件，还需要通过底层系统层的 Model 或者 System 来查询一些数据，这些功能最好是用一个 IController 来定义一下，我们目前能够实现的就是查询数据的功能，代码如下。

namespace FrameworkDesign  
{  
 public interface IController : IBelongToArchitecture  
 {  
 }  
}

代码非常简单，所在的位置如下:



代码不是很难，由于表现层的对象会时常进行创建和销毁，所以表现层的对象注册到 Architecture 是没有意义的，而这里定义 IController 接口的意义就是标记一下这个表现层的对象是属于表现层的，而在表现层对象中，我们去访问 Architecture 中的 System 或者 Model 就不需要使用单例的形式获取了。

OK 定义好了 IController 之后，我们尝试使用一下，我们先看下 CounterApp 中的 CounterViewController 脚本，代码如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = CounterApp.Get<ICounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
 }  
   
 ...  
}

我们可以看到 CounterViewController 对象里访问 Model 是通过 CounterApp 静态类的，这里我们加上一个 IController,代码如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour,IController  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = Architecture.GetModel<ICounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 ...  
  
 public IArchitecture Architecture { get; set; }   
 }  
  
 ...  
}

增加了 IController 但是脚本会报错的，这是因为 Architecture 没有在任何地方进行赋值。所以当通过 Architecture 访问 Model 的时候会报空指针异常。所以我们需要给 Architecture 进行赋值，而给 Architecture 进行赋值我们就需要获取到 CounterApp 对象，但是现在我们目前无法得到 CounterApp 对象只能获取 CounterApp 的类。

为了能够获取 CounterApp 的对象，我们就需要让 CounterApp 是一个类似单例类的形式，代码如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 public interface IArchitecture  
 {  
 void RegisterSystem<T>(T system) where T : ISystem;  
   
 void RegisterModel<T>(T model) where T : IModel;  
  
 void RegisterUtility<T>(T utility);  
  
 T GetModel<T>() where T : class, IModel;  
   
 T GetUtility<T>() where T : class;  
 }  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 ...  
   
 private static T mArchitecture;  
  
 // 新增  
 public static IArchitecture Interface  
 {  
 get  
 {  
 if (mArchitecture == null)  
 {  
 MakeSureArchitecture();  
 }  
  
 return mArchitecture;  
 }  
 }  
   
 ...  
   
 }  
}

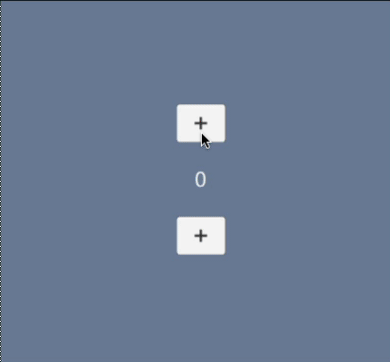
代码很简单，就是类似单例的一个实现，但是这里返回的是接口类型而不是实例的类型。

这样写完之后，我们就可以在 CounterViewController 里获取到 IArchitecture 实例了，代码如下:

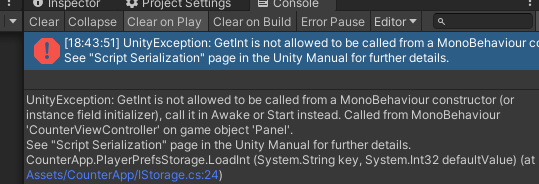
using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour,IController  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = Architecture.GetModel<ICounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
  
   
 public IArchitecture Architecture { get; set; } = CounterApp.Interface; // 修改部分  
 }  
  
 ...  
}

OK，这样的话 IController 就接入成功了。

我们运行一下 CounterApp 结果如下:



结果没啥问题，但是会报一个错误，如下：



这个错误是下边这行代码造成的:  
public IArchitecture Architecture { get; set; } = CounterApp.Interface; // 修改部分

这行代码会在构造时期调用，而在一个构造阶段，是不能调用 Unity 的一些 API 的。

所以我们不能让用户这样使用 IController，得另想别的办法。

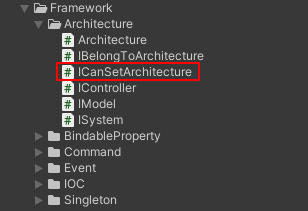
最简单的方法就是用 .Net 4.x 提供的 Lazy 类，但是笔者自己的实现没有使用 Lazy 类，而是把 IBelongToArchitecture 更改了一下，改成代码如下:

namespace FrameworkDesign  
{  
 public interface IBelongToArchitecture  
 {  
 IArchitecture GetArchitecture();  
 }  
}

原来的 IBelongToArchitecture 是可以 get 和 set Architecture 接口的，而 set Architecture 的部分，我们需要拆出来，拆成 ICanSetArchitecture 接口，代码如下:

namespace FrameworkDesign  
{  
 public interface ICanSetArchitecture  
 {  
 void SetArchitecture(IArchitecture architecture);  
 }  
}

代码所在的位置如下:



改了之后会触发一些编译错误，我们先不修改。

由于 set Architecture 和 get Architecture 的接口分离了，所以 IModel 和 ISystem 仅仅是实现 IBelongToArchitecture 是不够用的，因为 IModel 和 ISystem 都是和 Architecture 互相持有的，所以 IModel 和 ISystem 都需要继承 ICanSetArchitecture，代码如下:

namespace FrameworkDesign  
{  
 public interface IModel : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Init();  
 }  
}

namespace FrameworkDesign  
{  
 public interface ISystem : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Init();  
 }  
}

但是仅仅是这样是不够的，IBelongToArchitecture 和 ICanSetArchitecture 由于已经分离了，所以需要在实现这两个接口类中声明一个 IArchitecture 变量，而这个变量有一定的逻辑，每次交给子类实现会造成一些样板代码，所以这里笔者为 Model 和 System 都创建了一个抽象类，代码如下:

namespace FrameworkDesign  
{  
 public interface IModel : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Init();  
 }  
  
 public abstract class AbstractModel : IModel  
 {  
 private IArchitecture mArchitecture = null;  
 public IArchitecture GetArchitecture()  
 {  
 return mArchitecture;  
 }  
  
 public void SetArchitecture(IArchitecture architecture)  
 {  
 mArchitecture = architecture;  
 }  
  
 void IModel.Init()  
 {  
 OnInit();  
 }  
  
 protected abstract void OnInit();  
  
 }  
}

namespace FrameworkDesign  
{  
 public interface ISystem : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Init();  
 }  
  
 public abstract class AbstractSystem : ISystem  
 {  
 private IArchitecture mArchitecture = null;  
 public IArchitecture GetArchitecture()  
 {  
 return mArchitecture;  
 }  
  
 public void SetArchitecture(IArchitecture architecture)  
 {  
 mArchitecture = architecture;  
 }  
  
 void ISystem.Init()  
 {  
 OnInit();  
 }  
  
 protected abstract void OnInit();  
 }  
}

代码不难，这里使用了之前介绍的接口阉割技术。

接下来我们修改 Architecture，让 Architecture 支持以上的改动，代码如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 ...  
  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 ...  
  
 // 提供一个注册 Model 的 API  
 public void RegisterSystem<T>(T instance) where T : ISystem  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this); // 修改  
   
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mSystems.Add(instance);  
 }  
 }  
  
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this); // 修改  
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
  
 ...  
 }  
}

Architecture 部分改动不是很大，就两个地方。

接着是其他的部分，首先是 CounterViewController.cs 代码如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour,IController  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = GetArchitecture().GetModel<ICounterModel>(); // 修改  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new AddCountCommand()  
 .Execute();  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 new SubCountCommand()  
 .Execute();  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
  
 public IArchitecture GetArchitecture() // 修改  
 {  
 return CounterApp.Interface;  
 }  
 }  
  
  
 public interface ICounterModel : IModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel :AbstractModel /\* 新增 \*/, ICounterModel  
 {  
 protected override void OnInit() // 修改 Init -> OnInit  
 {  
 // 通过 Architrecture 获取  
 var storage = GetArchitecture().GetUtility<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
  
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

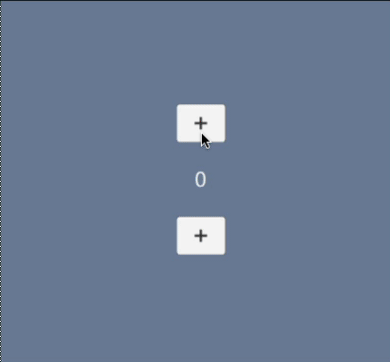
OK，都标记了出来。

最后是成就系统的代码，如下:

using System.Collections;  
using System.Collections.Generic;  
using CounterApp;  
using UnityEngine;  
  
namespace FrameworkDesign  
{  
 public interface IAchievementSystem : ISystem  
 {  
  
 }  
  
 public class AchievementSystem : AbstractSystem, IAchievementSystem  
 {  
  
 protected override void OnInit() // 修改  
 {  
 var counterModel = GetArchitecture().GetModel<ICounterModel>(); // 修改  
  
 var previousCount = counterModel.Count.Value;  
  
 counterModel.Count.OnValueChanged += newCount =>  
 {  
 if (newCount >= 10 && previousCount < 10)  
 {  
 Debug.Log("解锁 10 次点击成就");  
 }  
 else if (newCount >= 20 && previousCount < 20)  
 {  
 Debug.Log("解锁 20 次点击成就");  
 }  
  
 previousCount = newCount;  
 };  
 }  
 }  
}

这样全部的编译错误就改好了。

我们这次运行一下 CounterApp 结果如下:



这次没有任何报错。

OK，到这里，我们的 IController 的支持就全部完成了，这堂课的内容就这些，我们下一堂课再见，拜拜。

# 22. IUtiilty 实现 与 ICommand 完善

在上一堂课我们完成了 IController 接口的定义和实现。

在这一堂课，我们接着完成剩下的接口，先看下剩下哪些接口：

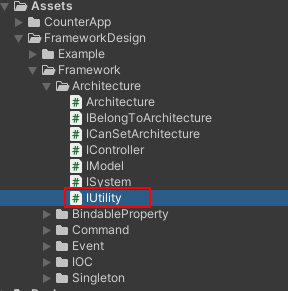
* IController
* IUtility
* ICommand

我们先实现 IUtility 的定义，代码如下：

namespace FrameworkDesign  
{  
 public interface IUtility  
 {  
  
 }  
}

代码很简单，由于工具类一般是没有生命周期的，所以不需要 Init 方法，然后也不需要访问 Architecture 所以就是一个接口即可。

代码所在位置如下:

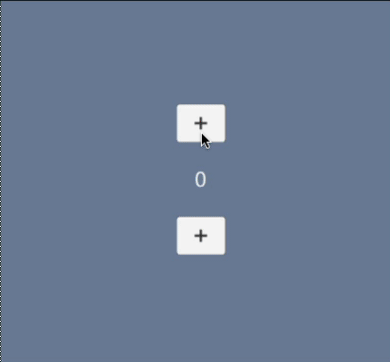


接着我们让 Architecture 支持 IUtility 接口，代码如下:

#if UNITY\_EDITOR  
using UnityEditor;  
#endif  
using FrameworkDesign;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public interface IStorage : IUtility // 新增  
 {  
 void SaveInt(string key, int value);  
  
 int LoadInt(string key, int defaultValue = 0);  
 }  
 ...  
}

之后所有的编译错误就修复了。

我们运行一下 CounterApp 结果如下:



结果正确。

接着我们来完善 ICommand 接口。

首先我们来看下 ICommand 接口，如下:

namespace FrameworkDesign  
{  
 public interface ICommand  
 {  
 void Execute();  
 }  
}

代码很简单。

在 Command 里，我们是通过单例的方式访问 Architecture 对象的。

而在其他的对象比如 Model、System 里都是与 Architecture 对象是互相持有的，所以我们的 ICommand 也要保持一致，所以 ICommand 接口代码如下:

namespace FrameworkDesign  
{  
 public interface ICommand : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Execute();  
 }  
}

同样我们需要一个抽象 Command 类，来简化 Command 的扩展，代码如下:

namespace FrameworkDesign  
{  
 public interface ICommand : IBelongToArchitecture, ICanSetArchitecture  
 {  
 void Execute();  
 }  
  
 public abstract class AbstractCommand : ICommand  
 {  
 private IArchitecture mArchitecture;  
  
 public IArchitecture GetArchitecture()  
 {  
 return mArchitecture;  
 }  
  
 public void SetArchitecture(IArchitecture architecture)  
 {  
 mArchitecture = architecture;  
  
 }  
  
 void ICommand.Execute()  
 {  
 OnExecute();  
 }  
  
 protected abstract void OnExecute();  
 }  
}

代码不难。

这样也会造成一些编译错误，我们都逐个修复一下，代码如下:

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class AddCountCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 GetArchitecture().GetModel<ICounterModel>().Count.Value++;  
 }  
 }  
}

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class SubCountCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 GetArchitecture().GetModel<ICounterModel>().Count.Value--;  
 }  
 }  
}

namespace FrameworkDesign.Example  
{  
 public class KillEnemyCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 var gameModel = GetArchitecture().GetModel<IGameModel>();  
   
 gameModel.KillCount.Value++;  
   
 if (gameModel.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

namespace FrameworkDesign.Example  
{  
 public interface IGameModel : IModel  
 {  
 BindableProperty<int> KillCount { get; }  
 BindableProperty<int> Gold { get; }  
 BindableProperty<int> Score { get; }  
 BindableProperty<int> BestScore { get; }  
 }  
  
 public class GameModel : AbstractModel, IGameModel  
 {  
 public BindableProperty<int> KillCount { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Gold { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> Score { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 public BindableProperty<int> BestScore { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
  
 protected override void OnInit()  
 {  
   
 }  
 }  
}

namespace FrameworkDesign.Example  
{  
 public class StartGameCommand : AbstractCommand, ICommand  
 {  
 protected override void OnExecute()  
 {  
 GameStartEvent.Trigger();  
 }  
 }  
}

接下来是重点部分。

我们要更改 Command 的调用方式，以前的调用方式代码如下:

new KillEnemyCommand()  
 .Execute();

就是创建一个 Command 的同时直接调用执行方法。

现在，笔者想要将 Command 发送给 Architecture，让 Architecture 去执行 Command，所以我们将 Architecture 的代码改成如下:

using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册系统  
 /// </summary>  
 void RegisterSystem<T>(T instance) where T : ISystem;  
  
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance) where T : IUtility;  
  
  
 /// <summary>  
 /// 获取 Model  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 /// <returns></returns>  
 T GetModel<T>() where T : class, IModel;  
  
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class, IUtility;  
  
 /// <summary>  
 /// 发送命令  
 /// </summary>  
 void SendCommand<T>() where T : ICommand, new(); // 新增  
  
 /// <summary>  
 /// 发送命令  
 /// </summary>  
 void SendCommand<T>(T command) where T : ICommand; // 新增  
 }  
  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 /// <summary>  
 /// 是否已经初始化完成  
 /// </summary>  
 private bool mInited = false;  
  
 /// <summary>  
 /// 用于初始化的 Systems 的缓存  
 /// </summary>  
 private List<ISystem> mSystems = new List<ISystem>();  
  
 // 提供一个注册 Model 的 API  
 public void RegisterSystem<T>(T instance) where T : ISystem  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this);  
  
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mSystems.Add(instance);  
 }  
 }  
  
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this);  
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
  
 #region 类似单例模式 但是仅在内部课访问  
  
 /// <summary>  
 /// 注册补丁  
 /// </summary>  
 public static Action<T> OnRegisterPatch = architecture => { };  
  
 private static T mArchitecture = null;  
  
 /// <summary>  
 /// 接口  
 /// </summary>  
 public static IArchitecture Interface  
 {  
 get  
 {  
 if (mArchitecture == null)  
 {  
 MakeSureArchitecture();  
 }  
  
 return mArchitecture;  
 }  
 }  
  
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 调用  
 OnRegisterPatch?.Invoke(mArchitecture);  
  
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
  
 // 初始化 System  
  
 foreach (var architectureSystem in mArchitecture.mSystems)  
 {  
 architectureSystem.Init();  
 }  
  
 // 清空 System  
 mArchitecture.mSystems.Clear();  
  
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public static void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
  
 public void RegisterUtility<T>(T instance) where T : IUtility  
 {  
 mContainer.Register<T>(instance);  
 }  
  
 public T GetModel<T>() where T : class, IModel  
 {  
 return mContainer.Get<T>();  
 }  
  
 public T GetUtility<T>() where T : class, IUtility  
 {  
 return mContainer.Get<T>();  
 }  
  
 public void SendCommand<T>() where T : ICommand, new() // 新增  
 {  
 var command = new T();  
 command.SetArchitecture(this);  
 command.Execute();  
 }  
  
 public void SendCommand<T>(T command) where T : ICommand // 新增  
 {  
 command.SetArchitecture(this);  
 command.Execute();  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

代码不难，就是执行 Command 的代码。

接着我们需要把所有调用 Command 的地方都更改一下，代码如下:

using FrameworkDesign;  
using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow,IController // +  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 // 需要在这里切换一下 Storage 的实现  
 CounterApp.OnRegisterPatch += architecture =>  
 {  
 architecture.RegisterUtility<IStorage>(new EditorPrefsStorage());  
 };  
   
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 GetArchitecture().SendCommand<AddCountCommand>(); // -+  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 GetArchitecture().SendCommand<SubCountCommand>(); // -+  
 }  
 }  
  
 public IArchitecture GetArchitecture() // +  
 {  
 return CounterApp.Interface;  
 }  
 }  
}

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour,IController  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = GetArchitecture().GetModel<ICounterModel>();  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 GetArchitecture().SendCommand<AddCountCommand>(); // -+  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 GetArchitecture().SendCommand<SubCountCommand>(); // -+  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
  
 public IArchitecture GetArchitecture()  
 {  
 return CounterApp.Interface;  
 }  
 }  
  
  
 public interface ICounterModel : IModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel :AbstractModel, ICounterModel  
 {  
 protected override void OnInit()   
 {  
 // 通过 Architrecture 获取  
 var storage = GetArchitecture().GetUtility<IStorage>();  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
  
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }  
}

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour,IController // +  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 GetArchitecture().SendCommand<KillEnemyCommand>(); // -+  
 }  
  
 public IArchitecture GetArchitecture() //+  
 {  
 return PointGame.Interface;  
 }  
 }  
}

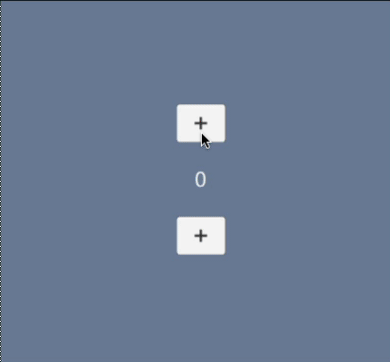
using UnityEngine;  
using UnityEngine.UI;  
  
namespace FrameworkDesign.Example  
{  
 public class GameStartPanel : MonoBehaviour,IController // +  
 {  
 void Start()  
 {  
 transform.Find("BtnGameStart").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 gameObject.SetActive(false);  
   
 GetArchitecture().SendCommand<StartGameCommand>(); // -+  
 });  
 }  
  
 public IArchitecture GetArchitecture() // +  
 {  
 return PointGame.Interface;  
 }  
 }  
}

Ok，这样就全部改完了。

改动量不少。

不过大家跟着逐个修改即可。

最后看下游戏的运行结果，如下:





没啥问题。

最后这里提一句，原来的 ICommand 扩展时只需要实现接口，所以是可以用 struct 来实现，但是现在我们扩展 ICommand 都需要用 AbstractCommand，这里必须用 class 来实现。使用 class 确实会比 struct 多一些消耗，如果这些消耗实在是影响了性能，可以考虑在 AbstractCommand 中实现一下对象池。

OK，到此，所有的接口都完善好了，如下:

* IController
* IUtility
* ICommand

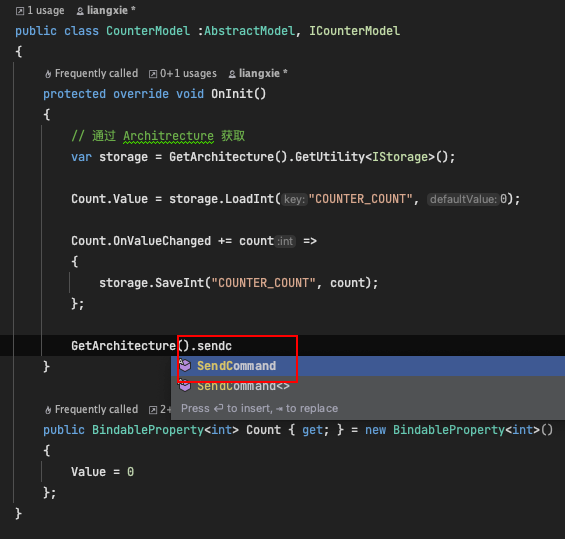
这堂课的内容就这些，我们下一趟课再见，拜拜。

# 23. 架构使用规则实现

在上一堂课，我们实现了 IUtility 和 ICommand。

在这一堂课，我们来接着改进 Architecture 的使用。

目前，我们的 Architecture 的使用有一个问题，就是 Model 层可以发送 Command，如下图所示：



我们看到图中 CounterModel 是可以发送 SendCommand 的。

而在之前的课程中笔者与说过，Command 是 表现层 与 底层系统层交互的方式。而在 Model 层使用 Command 是不合理的。

由于目前在技术上 Model 层是可以发送 Command 的，所以在实际开发中，其他开发者会很容易写出来不符合规范的代码，从而造成项目的风险。

为此，我们需要限制 Model 层，让 Model 层不能发送 Command。

首先，我们先把 GetArchitecture() 这个方法从 AbstractModel 里阉割掉，代码如下:

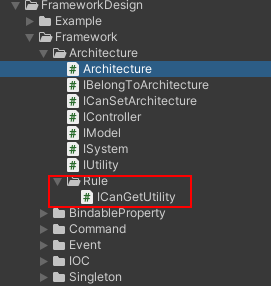
namespace FrameworkDesign  
{  
 public interface IModel : IBelongToArchitecture,ICanSetArchitecture  
 {  
 void Init();  
 }  
  
 public abstract class AbstractModel : IModel  
 {  
 private IArchitecture mArchitecture = null;  
 IArchitecture IBelongToArchitecture.GetArchitecture()  
 {  
 return mArchitecture;  
 }  
  
 void ICanSetArchitecture.SetArchitecture(IArchitecture architecture)  
 {  
 mArchitecture = architecture;  
 }  
  
 void IModel.Init()  
 {  
 OnInit();  
 }  
  
 protected abstract void OnInit();  
  
 }  
}

这样子类就无法使用 GetArchitecture()，为什么要阉割掉呢？因为 IArchitecture 是可以做所有事情的一个接口对象。而我们接下来想要让子类只能做 IArchitecture 的一部分事情，这就需要用到我们之前学过的 接口阉割技术 的第二种情况。

接下来，我们来实现一个规则接口，这一个接口是 Model 层可以用的几个能力，叫做 ICanGetUtiilty 代码如下:

namespace FrameworkDesign  
{  
 public interface ICanGetUtility : IBelongToArchitecture  
 {  
 }  
  
 public static class CanGetUtilityExtension  
 {  
 public static T GetUtility<T>(this ICanGetUtility self) where T : class, IUtility  
 {  
 return self.GetArchitecture().GetUtility<T>();  
 }  
 }  
}

代码不难，所在位置如下:



接下来，我们让 IModel 继承此接口，代码如下:

namespace FrameworkDesign  
{  
 public interface IModel : IBelongToArchitecture,ICanSetArchitecture,ICanGetUtility // 新增  
 {  
 void Init();  
 }  
  
 public abstract class AbstractModel : IModel  
 {  
 private IArchitecture mArchitecture = null;  
 IArchitecture IBelongToArchitecture.GetArchitecture()  
 {  
 return mArchitecture;  
 }  
  
 void ICanSetArchitecture.SetArchitecture(IArchitecture architecture)  
 {  
 mArchitecture = architecture;  
 }  
  
 void IModel.Init()  
 {  
 OnInit();  
 }  
  
 protected abstract void OnInit();  
  
 }  
}

这样我们就能在 CounterModel 中通过 this 来调用 GetUtility 了，代码如下:

public interface ICounterModel : IModel  
 {  
 BindableProperty<int> Count { get; }  
 }  
   
 /// <summary>  
 /// 不需要是单例了  
 /// </summary>  
 public class CounterModel :AbstractModel, ICounterModel  
 {  
 protected override void OnInit()   
 {  
 // 通过 Architrecture 获取  
 var storage = this.GetUtility<IStorage>(); // 通过 this 调用  
   
 Count.Value = storage.LoadInt("COUNTER\_COUNT", 0);  
  
 Count.OnValueChanged += count =>  
 {  
 storage.SaveInt("COUNTER\_COUNT", count);  
 };  
 }  
  
 public BindableProperty<int> Count { get; } = new BindableProperty<int>()  
 {  
 Value = 0  
 };  
 }

代码很简单，而且其实调用 GetUtility 更方便了一些，以前是通过 GetArchitecture() 调用 GetUtility 现在是通过 this 调用 GetUtility，使用 this 肯定比使用 GetArchitecture() 更容易一些，可以较少很多心智负担。

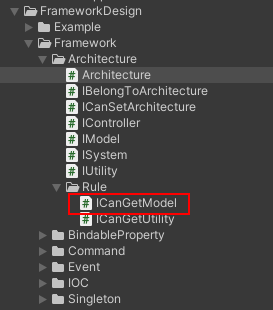
OK，这样 IModel 的第一个使用规则就实现好了。

接下来尝试一下定义 ISystem 的规则。

我们知道 ISystem 可以获取 Model，也可以获取 Utiilty，这里我们实现一下获取 Model 的规则，代码如下:

namespace FrameworkDesign  
{  
 public interface ICanGetModel : IBelongToArchitecture  
 {  
 }  
  
 public static class CanGetModelExtension  
 {  
 public static T GetModel<T>(this ICanGetModel self) where T : class, IModel  
 {  
 return self.GetArchitecture().GetModel<T>();  
 }  
 }  
}

代码所在位置如下:



接着我们来让 ISystem 继承 ICanGetModel 和 ICanGetUtility 代码如下:

namespace FrameworkDesign  
{  
 public interface ISystem : IBelongToArchitecture,ICanSetArchitecture,ICanGetModel,ICanGetUtility // 新增  
 {  
 void Init();  
 }  
  
 public abstract class AbstractSystem : ISystem  
 {  
 private IArchitecture mArchitecture = null;  
 IArchitecture IBelongToArchitecture.GetArchitecture() // -+  
 {  
 return mArchitecture;  
 }  
  
 void ICanSetArchitecture.SetArchitecture(IArchitecture architecture) // -+  
 {  
 mArchitecture = architecture;  
 }  
  
 void ISystem.Init()  
 {  
 OnInit();  
 }  
  
 protected abstract void OnInit();  
 }  
}

更改之后，我们接着修改成就系统的代码，如下:

using System.Collections;  
using System.Collections.Generic;  
using CounterApp;  
using UnityEngine;  
  
namespace FrameworkDesign  
{  
 public interface IAchievementSystem : ISystem  
 {  
  
 }  
  
 public class AchievementSystem : AbstractSystem, IAchievementSystem  
 {  
  
 protected override void OnInit()  
 {  
 var counterModel = this.GetModel<ICounterModel>(); // -+  
  
 var previousCount = counterModel.Count.Value;  
  
 counterModel.Count.OnValueChanged += newCount =>  
 {  
 if (newCount >= 10 && previousCount < 10)  
 {  
 Debug.Log("解锁 10 次点击成就");  
 }  
 else if (newCount >= 20 && previousCount < 20)  
 {  
 Debug.Log("解锁 20 次点击成就");  
 }  
  
 previousCount = newCount;  
 };  
 }  
 }  
}

OK，这样 ISystem 的使用规则也加好了。

接着再看下 ICommand ，ICommand 是可以获取 Model、获取 System，获取 Utility。

这里获取 System 的代码没有在 Architecture 里实现，所以我们先实现这个。

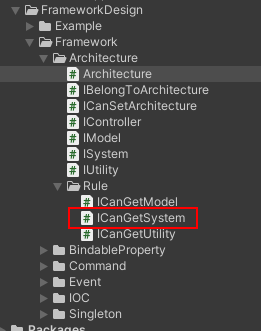
using System;  
using System.Collections.Generic;  
  
namespace FrameworkDesign  
{  
 public interface IArchitecture  
 {  
 /// <summary>  
 /// 注册系统  
 /// </summary>  
 void RegisterSystem<T>(T instance) where T : ISystem;  
  
 /// <summary>  
 /// 注册 Model  
 /// </summary>  
 void RegisterModel<T>(T instance) where T : IModel;  
  
  
 /// <summary>  
 /// 注册 Utility  
 /// </summary>  
 void RegisterUtility<T>(T instance) where T : IUtility;  
  
  
 /// <summary>  
 /// 获取 System  
 /// </summary>  
 T GetSystem<T>() where T : class, ISystem; // 新增  
   
 /// <summary>  
 /// 获取 Model  
 /// </summary>  
 T GetModel<T>() where T : class, IModel;  
  
 /// <summary>  
 /// 获取工具  
 /// </summary>  
 T GetUtility<T>() where T : class, IUtility;  
  
 /// <summary>  
 /// 发送命令  
 /// </summary>  
 void SendCommand<T>() where T : ICommand, new();  
  
 /// <summary>  
 /// 发送命令  
 /// </summary>  
 void SendCommand<T>(T command) where T : ICommand;  
 }  
  
 /// <summary>  
 /// 架构  
 /// </summary>  
 /// <typeparam name="T"></typeparam>  
 public abstract class Architecture<T> : IArchitecture where T : Architecture<T>, new()  
 {  
 /// <summary>  
 /// 是否已经初始化完成  
 /// </summary>  
 private bool mInited = false;  
  
 /// <summary>  
 /// 用于初始化的 Systems 的缓存  
 /// </summary>  
 private List<ISystem> mSystems = new List<ISystem>();  
  
 // 提供一个注册 Model 的 API  
 public void RegisterSystem<T>(T instance) where T : ISystem  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this);  
  
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mSystems.Add(instance);  
 }  
 }  
  
 /// <summary>  
 /// 用于初始化的 Models 的缓存  
 /// </summary>  
 private List<IModel> mModels = new List<IModel>();  
  
 public void RegisterModel<T>(T instance) where T : IModel  
 {  
 // 需要给 Model 赋值一下  
 instance.SetArchitecture(this);  
 mContainer.Register<T>(instance);  
  
 // 如果初始化过了  
 if (mInited)  
 {  
 instance.Init();  
 }  
 else  
 {  
 // 添加到 Model 缓存中，用于初始化  
 mModels.Add(instance);  
 }  
 }  
  
 #region 类似单例模式 但是仅在内部课访问  
  
 /// <summary>  
 /// 注册补丁  
 /// </summary>  
 public static Action<T> OnRegisterPatch = architecture => { };  
  
 private static T mArchitecture = null;  
  
 /// <summary>  
 /// 接口  
 /// </summary>  
 public static IArchitecture Interface  
 {  
 get  
 {  
 if (mArchitecture == null)  
 {  
 MakeSureArchitecture();  
 }  
  
 return mArchitecture;  
 }  
 }  
  
 // 确保 Container 是有实例的  
 static void MakeSureArchitecture()  
 {  
 if (mArchitecture == null)  
 {  
 mArchitecture = new T();  
 mArchitecture.Init();  
  
 // 调用  
 OnRegisterPatch?.Invoke(mArchitecture);  
  
 // 初始化 Model  
 foreach (var architectureModel in mArchitecture.mModels)  
 {  
 architectureModel.Init();  
 }  
  
 // 清空 Model  
 mArchitecture.mModels.Clear();  
  
 // 初始化 System  
  
 foreach (var architectureSystem in mArchitecture.mSystems)  
 {  
 architectureSystem.Init();  
 }  
  
 // 清空 System  
 mArchitecture.mSystems.Clear();  
  
 mArchitecture.mInited = true;  
 }  
 }  
  
 #endregion  
  
 private IOCContainer mContainer = new IOCContainer();  
  
 // 留给子类注册模块  
 protected abstract void Init();  
  
 // 提供一个注册模块的 API  
 public static void Register<T>(T instance)  
 {  
 MakeSureArchitecture();  
 mArchitecture.mContainer.Register<T>(instance);  
 }  
  
 public void RegisterUtility<T>(T instance) where T : IUtility  
 {  
 mContainer.Register<T>(instance);  
 }  
  
 public T GetSystem<T>() where T : class, ISystem // 新增  
 {  
 return mContainer.Get<T>();  
 }  
   
 public T GetModel<T>() where T : class, IModel  
 {  
 return mContainer.Get<T>();  
 }  
  
 public T GetUtility<T>() where T : class, IUtility  
 {  
 return mContainer.Get<T>();  
 }  
  
 public void SendCommand<T>() where T : ICommand, new()  
 {  
 var command = new T();  
 command.SetArchitecture(this);  
 command.Execute();  
 }  
  
 public void SendCommand<T>(T command) where T : ICommand  
 {  
 command.SetArchitecture(this);  
 command.Execute();  
 }  
  
 // 提供一个获取模块的 API  
 public static T Get<T>() where T : class  
 {  
 MakeSureArchitecture();  
 return mArchitecture.mContainer.Get<T>();  
 }  
 }  
}

代码很简单。

接着定义 ICanGetSystem 接口，如下:

namespace FrameworkDesign  
{  
 public interface ICanGetSystem : IBelongToArchitecture  
 {  
   
 }  
  
 public static class CanGetSystemExtension  
 {  
 public static T GetSystem<T>(this ICanGetSystem self) where T : class, ISystem  
 {  
 return self.GetArchitecture().GetSystem<T>();  
 }  
 }  
}

代码所在位置如下:



定义好了之后，我们在 ICommand 中加上规则，如下:

namespace FrameworkDesign  
{  
 // 新增  
 public interface ICommand : IBelongToArchitecture, ICanSetArchitecture,ICanGetSystem,ICanGetModel,ICanGetUtility  
 {  
 void Execute();  
 }  
  
 public abstract class AbstractCommand : ICommand  
 {  
 private IArchitecture mArchitecture;  
  
 IArchitecture IBelongToArchitecture.GetArchitecture() // -+  
 {  
 return mArchitecture;  
 }  
  
 void ICanSetArchitecture.SetArchitecture(IArchitecture architecture) // -+  
 {  
 mArchitecture = architecture;  
  
 }  
  
 void ICommand.Execute()  
 {  
 OnExecute();  
 }  
  
 protected abstract void OnExecute();  
 }  
}

代码不难。

接着修复下几个编译错误。

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class AddCountCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 this.GetModel<ICounterModel>().Count.Value++; // -+  
 }  
 }  
}

using FrameworkDesign;  
  
namespace CounterApp  
{  
 public class SubCountCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 this.GetModel<ICounterModel>().Count.Value--; // -+  
 }  
 }  
}

namespace FrameworkDesign.Example  
{  
 public class KillEnemyCommand : AbstractCommand  
 {  
 protected override void OnExecute()  
 {  
 var gameModel = this.GetModel<IGameModel>(); // -+  
   
 gameModel.KillCount.Value++;  
   
 if (gameModel.KillCount.Value == 10)  
 {  
 GamePassEvent.Trigger();  
 }  
 }  
 }  
}

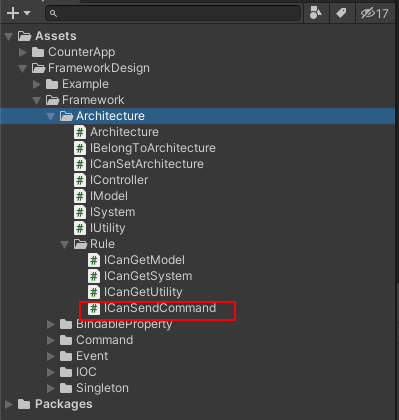
ICommand 的规则加好了。

最后是 IController，IController 可以获取 Model、获取 System、发送 Command。

这里发送 Command 的规则我们需要定义一下，如下:

namespace FrameworkDesign  
{  
 public interface ICanSendCommand : IBelongToArchitecture  
 {  
   
 }  
  
 public static class CanSendCommandExtension  
 {  
 public static void SendCommand<T>(this ICanSendCommand self) where T : ICommand, new()  
 {  
 self.GetArchitecture().SendCommand<T>();  
 }  
   
 public static void SendCommand<T>(this ICanSendCommand self,T command) where T : ICommand  
 {  
 self.GetArchitecture().SendCommand<T>(command);  
 }  
 }  
}

代码所在位置如下:



接着，我们在 IController 接口上，加上规则，如下:

namespace FrameworkDesign  
{  
 public interface IController : IBelongToArchitecture,ICanGetSystem,ICanGetModel,ICanSendCommand  
 {  
 }  
}

由于 IController 没有对应的抽象 Controller，所以只增加接口即可，IBelongToArchitecture 的阉割需要在子类实现。

我们修改下 CounterViewController 代码如下:

using FrameworkDesign;  
using UnityEngine;  
using UnityEngine.UI;  
  
namespace CounterApp  
{  
 public class CounterViewController : MonoBehaviour,IController  
 {  
 private ICounterModel mCounterModel;  
   
 void Start()  
 {  
 // 获取  
 mCounterModel = this.GetModel<ICounterModel>(); // -+  
   
 // 注册  
 mCounterModel.Count.OnValueChanged += OnCountChanged;  
  
 transform.Find("BtnAdd").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 this.SendCommand<AddCountCommand>(); // -+  
 });  
  
 transform.Find("BtnSub").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 // 交互逻辑  
 this.SendCommand<SubCountCommand>(); // -+  
 });  
   
 OnCountChanged(mCounterModel.Count.Value);  
 }  
  
 // 表现逻辑  
 private void OnCountChanged(int newValue)  
 {  
 transform.Find("CountText").GetComponent<Text>().text = newValue.ToString();  
 }  
  
 private void OnDestroy()  
 {  
 // 注销  
 mCounterModel.Count.OnValueChanged -= OnCountChanged;  
  
 mCounterModel = null;  
 }  
  
 IArchitecture IBelongToArchitecture.GetArchitecture() // -+  
 {  
 return CounterApp.Interface;  
 }  
 }  
  
 ...  
}

代码不难，IController 要求用户自己实现 GetArchitecture() 的阉割，当然我们也可以实现一个 IController 的抽象类 AbstractController，来简化使用，不过这里也留给用户自己实现即可。

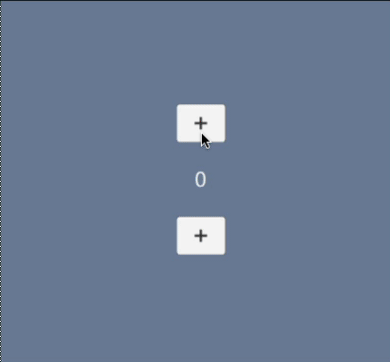
其他的 Controller 修改:

using UnityEngine;  
  
// 写命名空间是个好习惯  
namespace FrameworkDesign.Example  
{  
 public class Enemy : MonoBehaviour,IController  
 {  
 /// <summary>  
 /// 点击自己则销毁  
 /// </summary>  
 private void OnMouseDown()  
 {  
 Destroy(gameObject);  
   
 this.SendCommand<KillEnemyCommand>(); // -+  
 }  
  
 IArchitecture IBelongToArchitecture.GetArchitecture() // -+  
 {  
 return PointGame.Interface;  
 }  
 }  
}

using UnityEngine;  
using UnityEngine.UI;  
  
namespace FrameworkDesign.Example  
{  
 public class GameStartPanel : MonoBehaviour,IController  
 {  
 void Start()  
 {  
 transform.Find("BtnGameStart").GetComponent<Button>()  
 .onClick.AddListener(() =>  
 {  
 gameObject.SetActive(false);  
   
 this.SendCommand<StartGameCommand>(); // -+  
 });  
 }  
  
 IArchitecture IBelongToArchitecture.GetArchitecture() //-+  
 {  
 return PointGame.Interface;  
 }  
 }  
}

using FrameworkDesign;  
using UnityEditor;  
using UnityEngine;  
  
namespace CounterApp  
{  
 public class EditorCounterApp : EditorWindow,IController  
 {  
 /// <summary>  
 /// 打开窗口  
 /// </summary>  
 [MenuItem("EditorCounterApp/Open")]  
 static void Open()  
 {  
 // 需要在这里切换一下 Storage 的实现  
 CounterApp.OnRegisterPatch += architecture =>  
 {  
 architecture.RegisterUtility<IStorage>(new EditorPrefsStorage());  
 };  
   
 var editorCounterApp = GetWindow<EditorCounterApp>();  
 editorCounterApp.name = nameof(EditorCounterApp);  
 editorCounterApp.position = new Rect(100, 100, 400, 600);  
 editorCounterApp.Show();  
 }  
  
 private void OnGUI()  
 {  
 if (GUILayout.Button("+"))  
 {  
 this.SendCommand<AddCountCommand>(); //-+  
 }  
   
 // 由于实时刷新 所以直接就就渲染数据即可  
 GUILayout.Label(CounterApp.Get<ICounterModel>().Count.Value.ToString());  
  
 if (GUILayout.Button("-"))  
 {  
 this.SendCommand<SubCountCommand>(); //-+  
 }  
 }  
  
 IArchitecture IBelongToArchitecture.GetArchitecture() // -+  
 {  
 return CounterApp.Interface;  
 }  
 }  
}

到此所有的接口都增加完了限制，最后我们测试一下两个项目。





没啥问题。

这堂课，我们对 IModel、ISystem、ICommand、IController 都逐个增加了使用规则，用了这些规则在技术上可以规范架构的使用，也可以减少一个架构的使用说明文档，一举两得。

这一堂课的内容就这些，我们下一趟课再见，拜拜。