# **PureMVC框架在Unity中的应用（一）**

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

链接：<http://gad.qq.com/article/detail/287236>

前言：

这篇文章分为两部分，第一部分是理论，解读PureMVC框架原理，第二部分是实践，通过一个简单的例子，演示如何在Unity中应用PureMVC框架。(文章的内容很长，坚持看完，一定会有所收获的:)

我一直认为，框架的使用，需要你工作一段时间以后再去接触会比较好，就像孩子踢球一样，刚开始可以随心所欲的踢，没有什么中前，中后场的概念，球在哪儿，就一股脑儿的追上去抢，几十双脚噼里啪啦的，其实，这就是大多数人刚开始做游戏开发时候的状态，没什么框架，代码硬怼，不注重性能，扩展性，重用性等等，BUG层出不穷，只要功能出来就好，慢慢的，随着开发经验的不断丰富，你就会开始思考，当下做的这些事儿，有没有更方便，更有效的方法。

这时候你会开始注意到，”技战术“的作用有多么的重要！ 说点儿题外话，如果你在工作中，碰到技术很好的大神，而且又乐于分享，一定要多向他们请教，因为你真的可以少走很多弯路，并且能够更快的提升自己，没有什么比时间更有价值了，这是我一直以来都梦寐以求的，也许是我运气不太好，也许是我的工作历程和很多人都不同的缘故......扯得有些远了，无论是有经验的大神告诉你的，还是你通过搜索引擎查找，你都会听到MVC这个框架，这是个诞生了将近40年的经典框架。

没有完美的框架，MVC也是，在MVC的基础上，又演变出来了MVP，MVVM，但在本篇文章中，我将只介绍MVC框架的原理和应用，理解了MVC框架，将会更容易的理解MVP，MVVM，甚至是其它一些框架的设计，比如ECS。

那么简单说，MVC （Model:数据 View:视图组件 Controller:控制逻辑）的职责是：

将数据，视图组件和控制逻辑进行分离。让程序便于修改，更具有扩展性，灵活性，可重用性。高内聚，低耦合，一直是追求的目标。

但是传统经典的MVC模型虽然将数据，视图组件和控制逻辑进行了分离，但耦合性还是比较高，所以就有了今天要说的PureMVC。

他在传统MVC基础上做了许多的改进，通过结合多个“设计模式”的应用，让耦合性变得更低，也变得更加的易用，在扩展性，灵活性，重用性方面也做得更好。

设计模式的存在，其实很重要的一个职责就是解决耦合性。PureMVC用到的这些设计模式，贯穿了整个游戏框架，即便你项目中使用的不是MVC框架，你都离不开这些设计模式的应用，下面是PureMVC中使用到的设计模式：

1.代理设计模式

2.中介者设计模式

3.外观设计模式

4.观察者设计模式

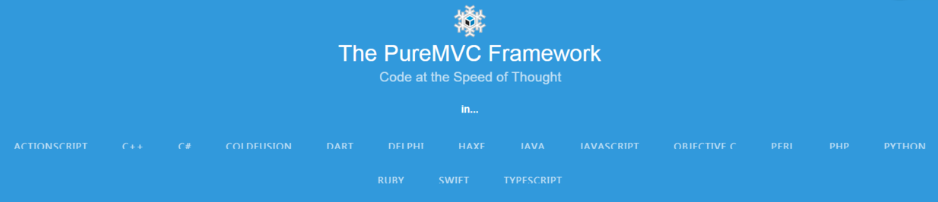
5.命令设计模式

6.单例设计模式

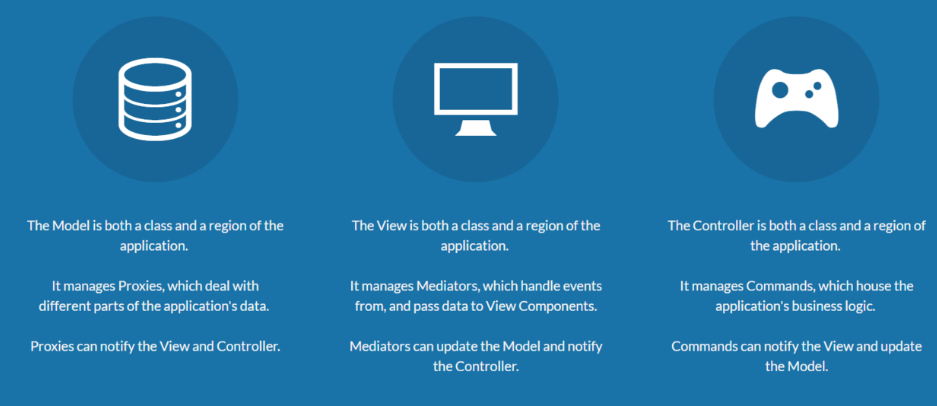
直入主题：

[http://puremvc.org/](http://puremvc.org/)

这是PureMVC的官方网站



可以看到，PureMVC支持了大部分的主流语言，你可以很容易的在项目中引入PureMVC。

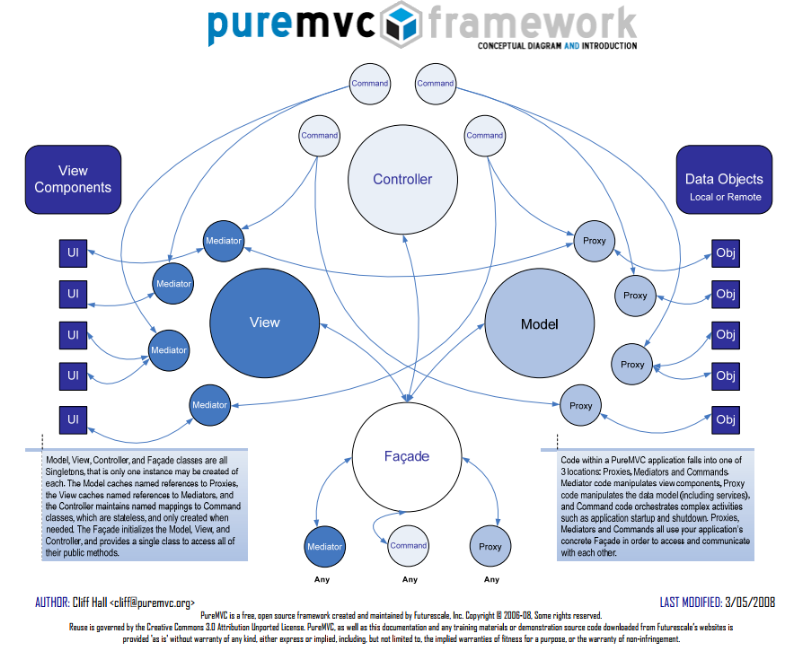


这是对MVC三层结构的说明和描述。



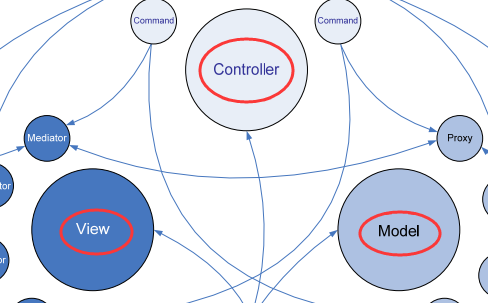
这里是Best Practises，实践手册，提供了6种语言版本，中文版本的翻译还可以，能表述出来核心思想就够了。

http://puremvc.org/docs/PureMVC\_Conceptual\_and\_Intro.pdf



这是PureMVC的概念图

我们先看下最核心的三个层，Model，View，Controller



我们发现，Model，View，Controller之间并没有任何的直接通信。

这里有一点需要注意，就是”改变一个不影响其它“，这也是为什么不能用传统的，将数据，视图组件和控制逻辑耦合在一起的做的原因，我仅是改变了一个UI组件的位置，不应该让数据和控制逻辑也进行编译，或者我对控制逻辑的调整，不应该影响到数据部分，而且这也不利于多人协作开发。

在PureMVC中，Model,View,Controller是三个单例模式类，三者合称为核心层或核心角色，换句话说就是Manager管理类，他们分别定义了字典用于保存引用，以及Register,Add,Remove,Retrieve等方法，将使用到的具体层(Model,View,Controller)，保存到字典中进行统一管理，这样，在我需要获取某一个具体层时，我可以通过key直接访问到它们。

简要的代码如下：

public interface IView ｛

void RegisterObserver (string notificationName, IObserver observer);

void RemoveObserver (string notificationName, object notifyContext);

void NotifyObservers (INotification note);

...........

void RegisterMediator (IMediator mediator);

IMediator RetrieveMediator (string mediatorName);

IMediator RemoveMediator (string mediatorName);

bool HasMediator (string mediatorName);

.........

｝

public class View : IView ｛

 public static IView Instance ｛

 get ｛ .... ｝

 ｝

 protected IDictionary<string,IMediator> m\_mediatorMap;

 protected IDictionary&lt;string,IList<IObserver>> m\_observerMap;

 public virtual void RegisterMediator (IMediator mediator) ｛

 .........

 ｝

｝

（我们先不要理会上面什么是Mediator，什么是Observer？ 后面会讲到，Model和Controller类也是大致类似的结构）

在开发的过程中，我要在Contoller中，获取View以及Model的对象，修改Mode，更新View，或是在View，我要获取Model，进行一些初始化或是修改的操作（当然，在PureMVC中，并不建议这样做），在业务逻辑很多的情况下，Model,View,Controller之间的频繁的调用就会非常多，耦合性会变高，如何解决呢？

我们通过一个上层的接口来负责所有核心层（Model,View,Controller)的管理和操作，在PureMVC中，使用了Facade设计模式，即外观设计模式。

外观设计模式的定义如下：

"为一组子系统或是接口提供一个统一的界面，以简化其复杂度，降低耦合性"

”Facade提供了与核心层通信的唯一接口，以简化开发复杂度。“

这样上面的Controller中，获取View和Model，或者View中，获取Model，均统一使用Facade进行管理。这样就降低了MVC三层之间的耦合性。对于使用者来说，只需要知道Facade类存在就可以了。

**具体Facade是什么样子的?**

Facade类应被当成抽象类, 永远不被直接实例化，你应该具体编写 Facade 的子类，添加或重写 Facade 的方法来实现具体的应用。这个类命名为“ApplicationFacade”（当然，命名随你喜欢）。

如前所述，它主要负责访问和通知 Model、View 和 Controller 。即管理这三者。

Facade 是 Model、View 和 Controller 三者的“经纪人”。实际编写代码时你并不用导入这三者的类文件，也不用直接使用它们。Facade 类已经在构造方法包含了对核心 MVC 三者单例的构造。

Facade类在构造方法中初始化了 Model、View 和Controller 对象，并把对它们的引用保存在成员变量。

这样，Facade就可以访问 Model、View 和 Controller 了。这样把对核心层的操作都集中在 Facade，避免开发者直接操作核心层。

这些是文档上的描述，有些啰嗦，但重复可以加深你的理解。

**初始化Facade？**

一般地，实际的应用程序都有一个 Facade 子类，这个 Facade 类对象负责初始化 Proxy，Mediator和Command，建立 Command 与 Notification 名之间的映射，或是通过执行一个 Command 来注册所有的 Proxy和 Mediator。

这时候就提到了Command，Proxy和Mediator，他们的对应关系其实就是：

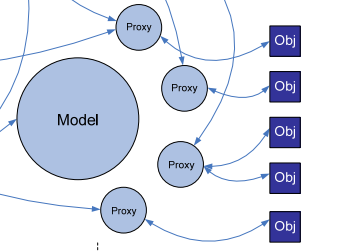
**Proxy=>Model**

**Mediator=>View**

**Command=>Controller**

但为什么要增加Proxy，Mediator和Command三个概念呢，其实也是为了降低耦合性。

**什么是Model和Proxy？**



从上面的截图可以看到Model和Proxy的关系图。

Model即数据(Data Object)，游戏是基于数据驱动的，比如角色的数据，包括HP，MP，金币，等级，经验，技能等等，在PureMVC中，是通过Proxy来进行管理，Proxy即代理设计模式，“为其它对象提供代理以控制该对象的访问”，即代理人，在PureMVC中被用来控制和管理数据模型的。

\*Data Object即是以任意结构存储数据的对象。

也就是说，我们不会直接和Model通信，对Model的增删改查均是通过Proxy来处理的。

关于Proxy代理模式，比如球星-C罗，他的Proxy代理就是C罗团队，有什么商业合作事宜均通过C罗团队进行接洽，这样，C罗就不需要一个人去面对来自四面八方的合作沟通成本（降低耦合性），同时，团队也可以帮助C罗处理很多的事务，不需要每件事儿都要经由C罗的过目(一定程度上隐藏了其内部的实现细节），从代码角度，也满足”改变一个不影响其它“，我对部分数据的修改，不应该影响到其它的数据。

我们继续看上面的示意图，看下Model的箭头，他只和Facade进行交互。上面提到过，这是为了降低耦合性。

旁边的一众Obj即Model,对应着Proxy,但并不是一个Model对应一个Proxy,如果是这样，就太繁琐了，比如一个模块中，可能包括很多种不同的Model数据，你可以定义多个不同的Model，但可以通过一个Proxy进行管理，这样更方便。

通常会以同步的方式取得或设置Model数据。Proxy 可能会提供访问 DataObject 部分属性或方法的 API，也可能直接提供 Data Object 的引用(但不建议这样做，我们应该保护Model，提供相应的接口来访问）。如果提供了更新 Data Object 的方法，那么在数据被修改时可能会发送一个 Notifidation 通知系统的其它部分。

这里Notification通知，其实就是观察者模式，当一个对象发生改变的时候，同时也需要有很多其它的对象要对此做出响应，这时候就要使用观察者模式了，发布-订阅的模式，比如我们订阅了某个微信公众号，公众号发表了一篇文章，所有订阅的用户都可以收到提醒，这在游戏中无处不在，当Model发生变化的时候，通知View组件进行更新。那么在View中，就会有相应的方法来处理Notification通知，并进行相应的逻辑处理。

Proxy只发送Notification通知（在数据改变的时候），他并不处理Notification通知，他并不关心View组件如何变化。

Proxy 对象不应该通过引用、操作 Mediator 对象来通知系统它的 DataObject（数据对象）发生了改变。

也就是说，Mediator可以获取Proxy，但Proxy中不应该获取Mediator，如果要通知View层进行更新，发送Notification通知即可。

（Proxy 不关心这些 Notification 被发出后会影响到系统的什么）

这样Proxy和Mediator之间只存在单向耦合。

Proxy中也包含了一定的逻辑处理的部分，我们把 Domain Logic（域逻辑）尽可能放在 Proxy 中实现，这样尽可能地做到 Model 层与相关联的 View 层、Controller 层的分离。

比如计算扣税的函数，如果你将他放在Mediator或是Command中实现，那么就相当于把这部分代码耦合了，比如你的View要重建，或是别的Command也要使用该扣税函数，那么这部分代码就无法得到复用，所以放在Proxy中是更为合适的。

**关于数据类型的转换**

因为 Model(Data Object) 通常是一个复杂的数据结构，我们经常需要引用它的一部分属性并将类型转化成我们需要的数据。

通过getter 和 setter 属性，它可以很好地帮助我们解决这种频繁的类型转换问题。

可能需要定义不同的多个类型 getter 来取得 Data Object 某部分的数据。

public ArrayCollection searchResultAC

｛

 get ｛return data as ArrayCollection;｝

 ｝

这是官方文档的例子，但在移动端，还是建议缓存起来，避免每次都进行转换来消耗内存，引起GC。

**什么是View 与 Mediator ?**

View即视图组件，和Model层一样，也是通过”中介“来进行管理，View是由Mediator来操作具体的视图组件(View Component)。包括：添加事件监听器 ，发送或接收 Notification ，直接改变视图组件的状态。

Mediator(中介者设计模式）：

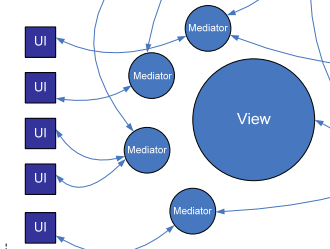
“用一个中介对象来封装一系列的对象交互“（重交互， 强逻辑）

\*View Component即UI上的各种控件，按钮，列表，滚动条等等。

这样做实现了把视图和控制它的逻辑分离开来。对于View中组件的定义和初始化都在Mediator中定义和实现，这样即使UI重建，也只是更改Mediator。

因为 Mediator 也会经常和 Proxy 交互，所以经常在 Mediator 的构造方法中取得Proxy 实例的引用并保存在 Mediator 的属性中，这样避免频繁的获取 Proxy 实例带来的性能开销。

如图：



这里可以看到，通常View和Mediator是一对一的关系，但有些View会相对复杂，有多个子UI组成，Mediator中也可以有多个View Component引用（同一功能的不同子UI）。

但如果Mediator过于庞大，就要进行拆分，在拆分后的子模块的 Mediator 里处理要比全部放在一起更好。这部分工作需要慢慢的重构。

**转化 View Component 类型**

（这部分和Model是一样的处理方式 ）

这个 Mediator 子类会在构造函数中把它的 View Component 传参给父类，它会在内部赋值给一个 protect 属性：viewComponent，并传化为Object 类型

Mediator 被构造之后，你可以通过调用它的 setViewComponent 函来动态给它的 View Component 赋值（修改）。之后，每一次需要访问这个 Object 的 API 时，你都要手动把这个 Object转化成它的真正类型。这是一项烦琐重复的工作。

和上面的Model一样，Mediator中保存了View的引用，我们最缓存下来。

**Mediator通常要做的事：**

1.检查或修改 View Component 的属性

2.检查或修改 Proxy 对象公布的属性

3.发送一个或多个 Notification ，通知别的 Mediator 或Command 作出响应（甚至有可能发送给自身）。

但要注意一点，业务逻辑(Business Logic)应该放在Command中，而非Mediator中！

实际上Mediator并不处理复杂的逻辑。像Model那样，域逻辑的部分，可以放在Mediator中实现，减少与Controller的耦合性，也提高了重用性。

注意：不要将检测或是对VC(View Component以及Proxy）属性的修改当作是业务逻辑(Business Logic)

**下面是一些有用的经验：**

1.如果有多个的 Mediator 对同一个事件做出响应，那么应该发送一个 Notification，然后相关的 Mediator 做出各自的响应。(观察者模式）

（比如说，你当前屏幕上显示了3个UI，每个UI上都显示着玩家的金钱数量，当金钱发生变化的时候，Proxy应该发送一个相应的Notification通知，然后3个UI接受通知并进行View的更新）

2.如果一个 Mediator 需要和其他的 Mediator 进行大量的交互，那么一个好方法是利用 Command 把交互步骤定义在一个地方。

3.不应该让一个 Mediator 直接去获取调用其他的 Mediator，在Mediator 中定义这样的操作本身就是错误的。可以看上面的概念图，Mediator和Mediator之间不会直接进行通信的，这样就违背了降低耦合性的初衷。

当一个View的改变会影响到另外一个View组件，发送Notification通知即可。

4.Proxy 是有状态的，当状态发生变化时发送 Notification 通知Mediator，将数据的变化反映到视图组件上。

将这些多次使用到的“请求“，通过command实现，使之更加的独立，提高重用性。

**Proxy设计模式 vs Mediator设计模式？**

前面提到的两个设计模式，两者所做的事情非常的相似，但定义上，Proxy更侧重于控制数据的访问，相当于真实数据的代表，而Mediator则更侧重于数据的交互（封装了一系列对象的交互），强逻辑，比如AB之间交互的中间人，那么对于UI的交互是相对复杂繁琐的，所以使用Mediator来负责处理View上的操作。

在《大话设计模式》中举了个蛮不错的例子来说明Mediator，即联合国，类似的还有环境保护组织，我们日常能接触的房产中介，负责房屋的勘察，审核，买卖，缴税，过户等（交互，强逻辑）工作。

如果直接让我们和房东联系，很多不懂的知识外，还有法律上的风险。

另一个例子，4S店，我们买卖，售后等都要去4S店进行处理，在Mediator中，A和B进行交互，A和B都”认识“Mediator中介者，我们去找4S店，4S店负责和汽车的生产商沟通。

**什么是Controller 与 Command ？**

(终于说到最后一部分了:)

Controller保存了所有Command的映射，Command 类是无状态的，只在需要时才被创建。

这里使用到了Command命令设计模式，即将一个“请求”,"行为”封装为一个对象，将逻辑的部分进行独立封装，提高复用性，对View或Mediator的修改也不会影响到Command本身。通过Facade顶层接口，可以在Proxy，Mediator，Command之间，相互访问和通信。

Command 可以获取 Proxy 和Mediator对象并与之交互，发送 Notification，执行其他的 Command。经常用于复杂的或系统范围的操作，如应用程序的“启动”和“关闭”。应用程序的业务逻辑应该在这里实现。

Facade 需要在启动时初始化 Controller，建立 Notification 与 Command的映射。

Controller 会注册侦听每一个 Notification，当被通知到时，Controller 会实例化一个该 Notification 对应的 Command 类的对象。最后，将 Notification 作为参数传递给execute 方法。具体可以参考Command基类的实现。

也就是说，Command的执行是通过发送Notification通知操作的。

Command 对象是无状态的；只有在需要的时候（ Controller 收到相应的Notification）才会被创建，并且在被执行（调用 execute 方法）之后就会被删除。所以不要在那些生命周期长的对象（long-living object）里引用 Command 对象。

在运行中，可以通过Command来初始化Proxy和Mediator，即注册到Facade中。

比如：

public class ModelPrepCommand : SimpleCommand ｛

 //由 MacroCommand 调用

 public override void Execute (INotification notification) ｛

 IFacade.registerProxy (new SearchProxy ());

 IFacade.registerProxy (new PrefsProxy ());

 IFacade.registerProxy (new UsersProxy ());

 ｝

｝

（这里就不贴Command部分的截图上，大家能看得懂Model和Mediator后，Command这里就比较清晰了）

两种类型的Command：

Command 要实现 ICommand 接口。在 PureMVC 中有两个类实现了ICommand 接口：SimpleCommand、MacroCommand。SimpleCommand 只有一个 execute 方法，execute 方法接受一个Inotification 实例做为参数。实际应用中，你只需要重写这个方法就行了。

MacroCommand 让你可以顺序执行多个 Command。每个执行都会创建一个 Command 对象并传参一个对源 Notification 的引用。MacroCommand 在构造方法调用自身的 initializeMacroCommand 方法。实际应用中，你需重写这个方法，调用 addSubCommand 添加子 Command。你可以任意组合 SimpleCommand 和 MacroCommand 成为一个新的 Command。

其它要介绍的：

**Business Logic（业务逻辑）和 Domain Logic（域逻辑）？**

在程序的很多地方你都可以放置代码（Command，Mediator 和Proxy）；不可避免地会不断碰到一个问题：哪些代码应该放在哪里？确切的说，Command 应该做什么？

程序中的逻辑分为 Business Logic（业务逻辑）和 Domain Logic（域逻辑），首先需要知道这两者之间的差别。

Business Logic（业务逻辑）要协调 Model 与视图状态(View)。

Model 通过使用 Proxy 来保证数据的完整性、一致性 。Proxy 集中程序的Domain Logic（域逻辑），并对外公布操作数据对象的 API。它封装了所有对数据模型的操作，不管数据是客户端还是服务器端的，对程序其他部分来说就是数据的访问是同步还是异步的。

Mediator 和 Proxy 可以提供一些操作接口让 Command 调用来管理 ViewComponent 和Model( Data Object)，同时对 Command 隐藏具体操作的细节。

**Observer 与 Notification ？**

PureMVC的通信是使用观察者模式以一种松耦合的方式来实现的，几乎在游戏开发中，无处不在的设计模式，你只需要使用一个非常简单的方法从 Proxy,Mediator, Command 和 Facade 发送 Notification，甚至不需要创建一个Notification 实例。

Facade 和 Proxy 只能发送 Notification，Mediators 既可以发送也可以接收 Notification，Notification 被映射到 Command，同时 Command 也可以发送 Notification。这是一种“发布/订阅”机制 ，所有的观察者都可以收到相同的通知。例如多个书刊订阅者可以订阅同一份杂志，当杂志有新刊出版时，所有的订阅者都会被通知。

Facade 保存了 Command 与 Notification 之间的映射。当 Notification（通知）被发出时，对应的 Command（命令）就会自动地由 Controller 执行。Command 实现复杂的交互，降低 View 和 Model 之间的耦合性。

**定义Notification常量**

当这些 Notification 的名称常量需要被其他的程序访问时，我们可以使用单独的“ApplicationConstants”类来存放这些 Notification 名称常量定义。不管什么时候，都应该把 Notification（通知）名称定义为常量，需要引用一个 Notification 时就使用它的名称常量，这样做可以避免一些编译时无法发现的错误。因为编译器可以检查常量；而使用字符串，如果你手误输入错误的字符串，编译器也不法知道，也无从报错。

**Mediator发送、声明、接收Notification**

当用 View 注册 Mediator 时，Mediator 的 listNotifications 方法会被调用，以数组形式返回该 Mediator 对象所关心的所有 Notification。之后，当系统其它角色发出同名的 Notification（通知）时，关心这个通知的Mediator 都会调用 handleNotification 方法并将 Notification 以参数传递到方法。

这里Mediator是被通知者，当Proxy数据进行改变时，Mediator接收到通知，并对UI进行更新。

**Proxy发送，但不接收Notification**

在很多场合下 Proxy 需要发送 Notification（通知），比如：Proxy 从远程服务接收到数据时，发送 Notification 告诉系统；或当 Proxy 的数据被更新时，发送 Notification 通知视图组件进行更新等等。

如果让 Proxy 也侦听 Notification（通知）会导致它和 View（视图）层、Controller（控制）层的耦合度太高。

View 和 Controller 必须监听 Proxy 发送的 Notification，因为它们的职责是通过可视化的界面使用户能与 Proxy 持有的数据交互。不过对 View 层和 Controller 层的改变不应该影响到 Model 层。

最后是文档中View Component 和 Mediator 交互的小例子：

假如有一个含表单的 LoginPanel 组件。对应有一个 LoginPanelMediator，负责与 LoginPanel 交互并响应它的输入信息发送登录请求。

LoginPanel 和 LoginPanelMediator 之间的协作表现为：LoginPanel 在用户输入完信息要登录时发送一个 TRY\_LOGIN 的事件，LoginPanelMediator 处理这个事件，处理方法是发送一个以组件包含的 LoginVO 为“报体”的 Notification（通知）。

这里省略LoginPanel.mxml的部分，视图的部分各不相同。

LoginPanel 组件包含有一个用用户表单输入新创建的 LoginVO 对象，当用户单击“Login”按钮时触发一个事件，接下来的事情由LoginPanelMediator 接管。

这样 View Component 的角色就是简单收集数据，收集完数据通知系统。可以完善的地方是只有当 username 和 password 都有内容时才让 login按钮可用（enable），这样可以避免恶意登录。

View Component 对外隐藏自己的内部实现，它由 Mediator 使用的整个API 包括：一个 TRY\_LOGIN 事件，一个 LoginVO 属性和 Panel 的状态属性。

LoginPanelMediator 会对 LOGIN\_FAILED 和 LOGIN\_SUCCESS 通知做出反应，设置 LoginPanel 的状态。

LoginPanelMediator.as:

package com.me.myapp.view

｛

import flash.events.Event;

import org.puremvc.as3.interfaces.\*;

import org.puremvc.as3.patterns.mediator.Mediator;

import com.me.myapp.model.LoginProxy;

import com.me.myapp.model.vo.LoginVO;

 import com.me.myapp.ApplicationFacade;

import com.me.myapp.view.components.LoginPanel;

// LoginPanel视图的Mediator

public class LoginPanelMediator extends Mediator implements IMediator

 ｛

 public static const NAME:String = 'LoginPanelMediator';

 public function LoginPanelMediator( viewComponent:LoginPanel )

｛

     super( NAME, viewComponent );

     LoginPanel.addEventListener( LoginPanel.TRY\_LOGIN,

    onTryLogin );

 ｝

 // 列出该Mediator关心的Notification

 override public function listNotificationInterests( ) : Array

｛

    return [

    LoginProxy.LOGIN\_FAILED,

     LoginProxy.LOGIN\_SUCCESS

 ];

 ｝

 // 处理Notification

 override public function handleNotification( note:INotification ):void

｛

 switch ( note.getName() ) ｛

     case LoginProxy.LOGIN\_FAILED:

     LoginPanel.loginVO = new LoginVO( );

     loginPanel.loginStatus =

    LoginPanel.NOT\_LOGGED\_IN;

     break;

     case LoginProxy.LOGIN\_SUCCESS:

     loginPanel.loginStatus = LoginPanel.LOGGED\_IN;

     break;

 ｝

 ｝

 // 用户单击Login按， 登。 钮 尝试 录

 private function onTryLogin ( event:Event ) : void ｛

     sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );

 ｝

 // 把viewComponent转 类 化成它真正的 型。

 protected function get loginPanel() : LoginPanel ｛

     return viewComponent as LoginPanel;

｝

 ｝

｝

主要看LoginPanelMediator.as，针对视图事件的注册，监听，处理等逻辑均在Mediator中

处理，登录的结果，成功LOGIN\_SUCCESS和LOGIN\_FAILED失败均在Mediator中处理。

由Mediator来负责处理View的事件交互。

Mediator可以直接修改View相关的属性和参数，如上面的例子中，对loginVO进行修改，但如果涉及到对Proxy数据进行修改，应该放在Command中，提高可重用性，降低View与Model之间的耦合性。

**最后小小总结下：**

View和Model之间是单向依赖关系，View必须知道Model是什么，View也是基于Model的数据来显示视图上的内容。而Model并不在乎View上的内容。

Proxy和Mediator在职责上，均是代理，中介者的角色，负责与其它组件进行通信。而他们的注册都由

Facade来统一进行管理。

Proxy和Mediator中不应该包含大量的业务逻辑，业务逻辑部分应该放在Command中处理，对于数据本身的一些操作，应该放在Proxy和Mediator中。

虽然Mediator中可以访问任意的Proxy，并进行修改，但不建议这样做，由 Command 来做这些工作可以实现 View 和 Model 之间的松耦合。这样Command可以被View的其它部分重用。

理论的部分终于讲完了，下面我们通过一个Unity例子来实际应用下PureMVC！

(感谢阅读，如文中有误，欢迎指正)