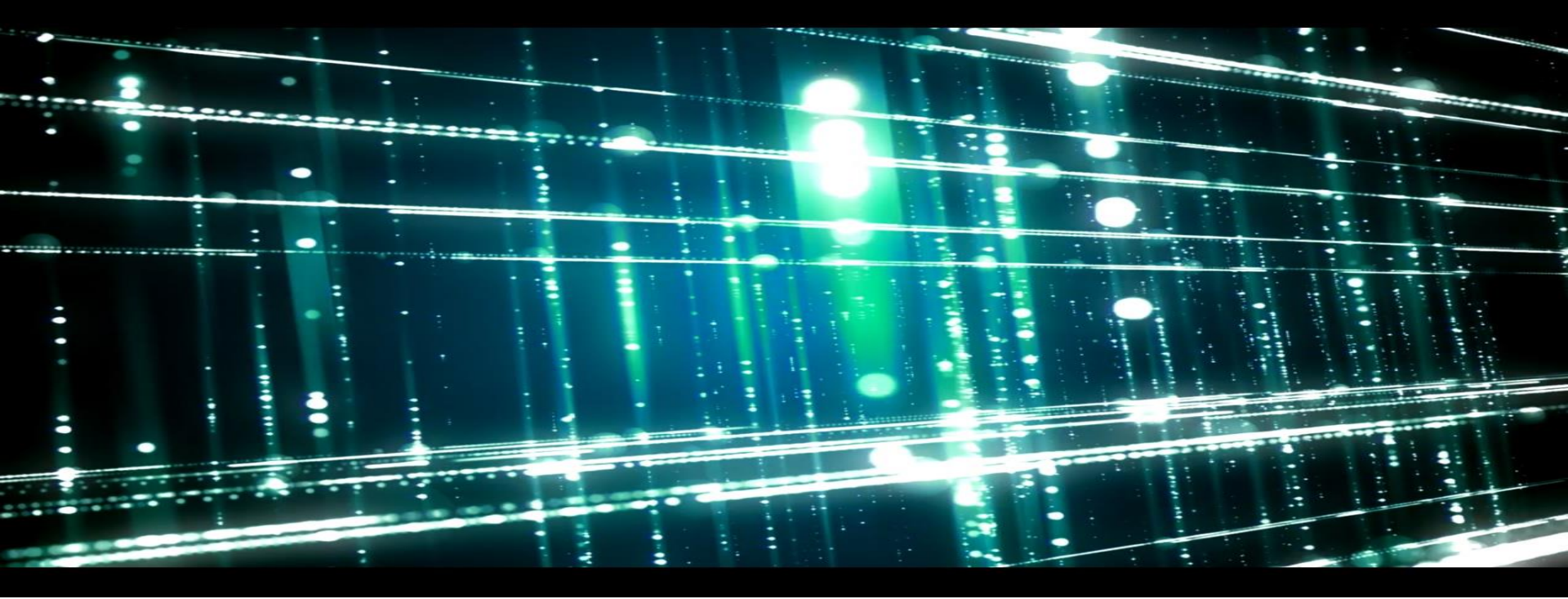


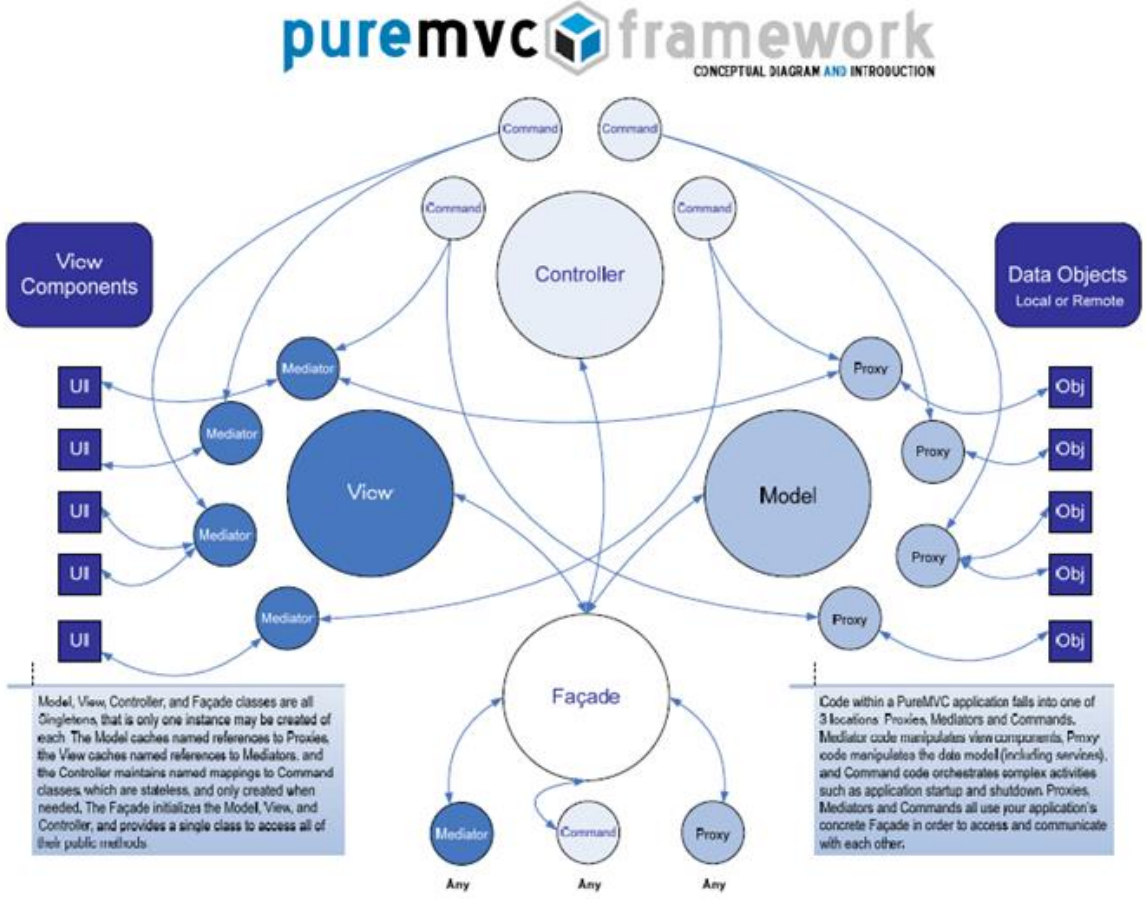
开源架构源码分析

PureMVC框架设计(下篇)　讲师:刘国柱

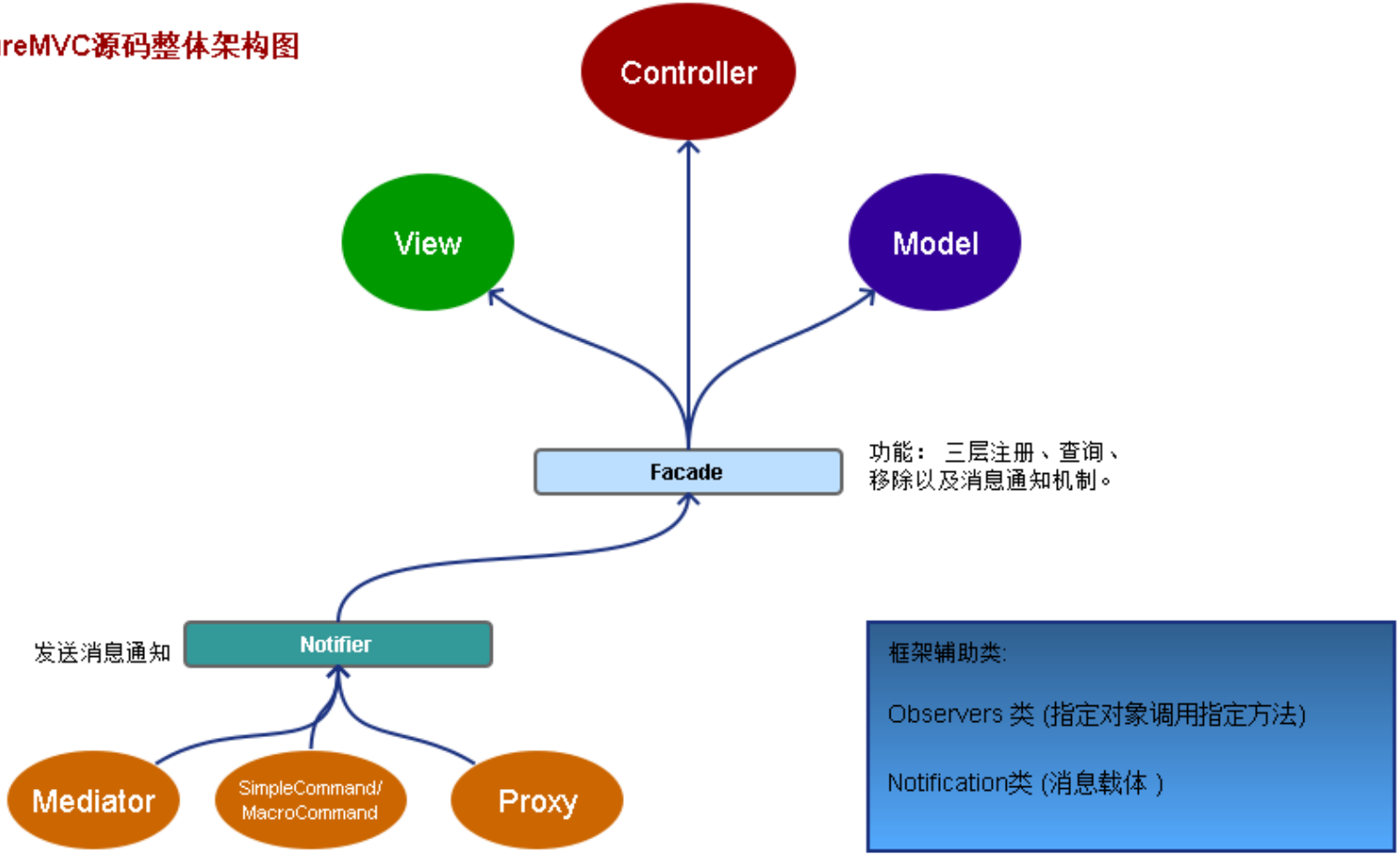
目录

PureMVC 整体架构图

- 原版PureMVC 整体架构图。



PureMVC源码整体架构图



目录

PureMVC主要设计模式：Mediator模式

- Mediator 中介者模式/迪米特法则(LoD) 也叫最少知识原则。[J&DP]

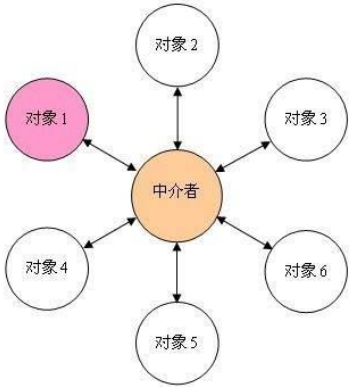
定义1： 如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。[J&DP]

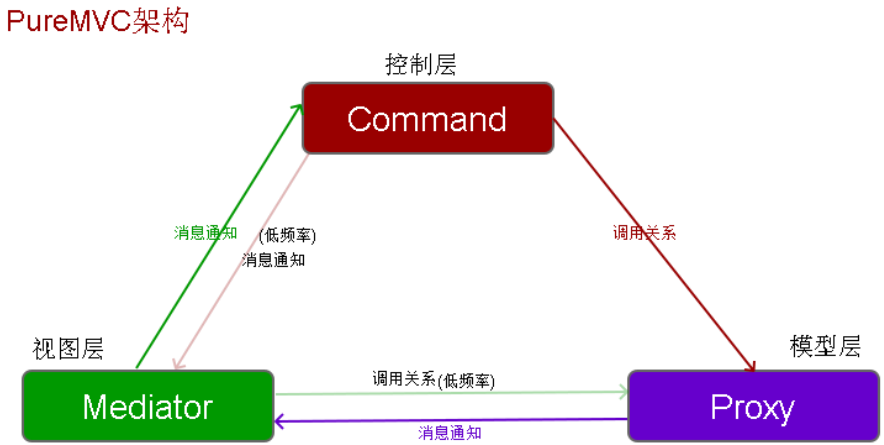
定义2： 定义一个中介对象来封装系列对象之间的交互。中介者使各个对象不需要显示地相互引用，从而使其耦合性松散，可以独立地改变他们之间的交互。

J&DP: 《Java与模式》 作者: 彦宏，电子工业出版社

PureMVC主要设计模式：Mediator模式(续)

- Mediator的设计用意在于通过一个媒介对象，完成一组对象的交互，避免对象间相互引用，产生复杂的依赖关系。



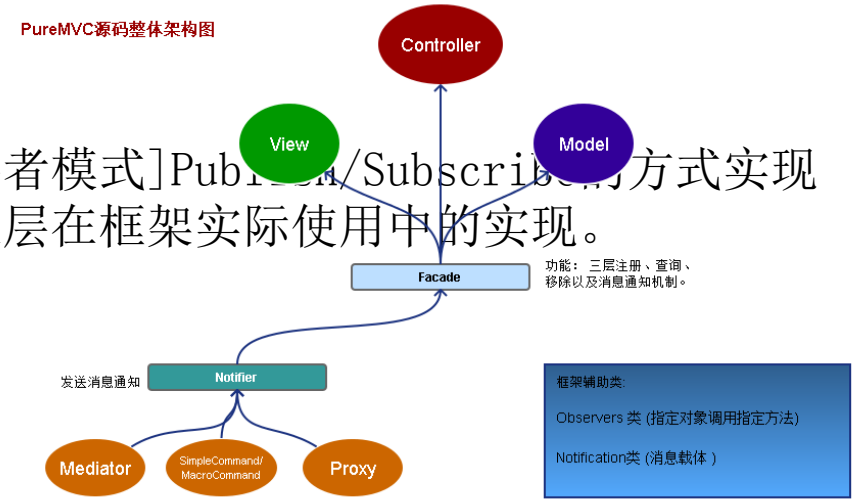


PureMVC主要设计模式：观察者模式

- 观察者模式（也叫发布-订阅[Publish/Subscribe]模式）

定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使他们能够自动更新自己。[DP]

其三层的通信，采用框架本身[观察者模式]Publish/Subscribe的方式实现事件机制来驱动，并由Facade类统筹三层在框架实际使用中的实现。

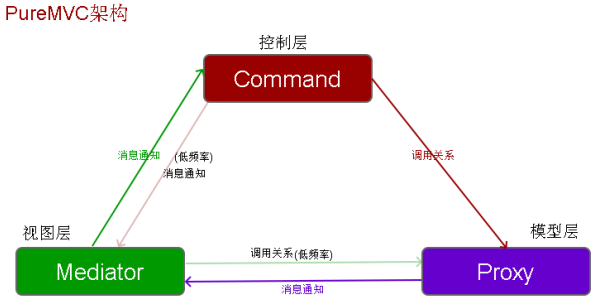


DP: 《设计模式》 作者: Erich Gamm 等，机械工业出版社

PureMVC主要设计模式:观察者模式(续)

- 事件机制起到了至关重要的作用。事件机制可以让当前对象专注于处理其职责范围内的事务，而不必关心超出部分由谁来处理以及怎样处理。

- 当前对象只需要广播一个事件，就会有对此事件感兴趣的其他对象出来接手下一步的工作，当前对象与接手对象之间不存在直接依赖，甚至感知不到彼此的存在，这是事件机制被普遍认为是一种松耦合机制的重要原因。

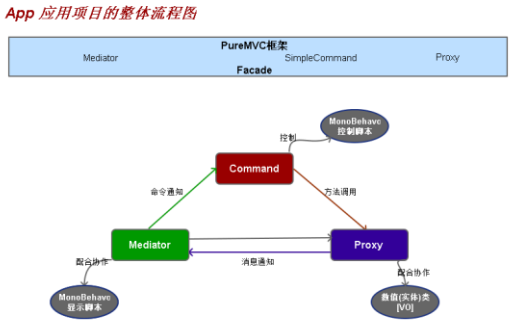


PureMVC主要设计模式：Facade模式

- Facade （门面模式/外观模式）

  为子系统中的一组接口提供一个一致的界面，此模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。[DP]

作用：使得架构外部的类与脚本，更方便的与架构内部的核心类通讯，但相互之间不直接沟通。（生活中： 中小学的 “课代表”、政府的“信访办”等 ）



PureMVC主要设计模式：其他

- 代理模式(Proxy)：

  为其他对象提供一种代理以控制对这个对象的访问。[DP]

- 命令模式(Command)：

  将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。[DP]

  命令模式的优点：把请求一个操作的对象与知道怎么执行一个操作的对象分隔开。

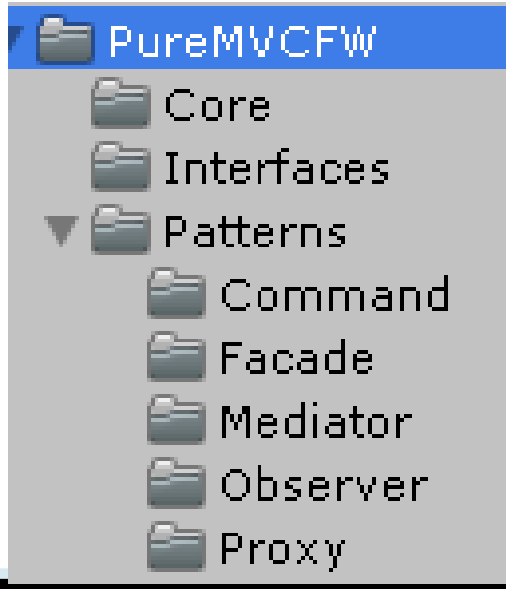
目录

源代码目录结构分析(1)

- Puremvc源码目录结构

  - Core目录:
  - Interfaces目录：

接口的名称比较好理解，使用者可以实现这些接口来完成自己定制的功能。



源代码目录结构分析(2)

- Patterns目录：
  - Command

1> 在puremvc里面提供了2种command，分别为SimpleCommand和MacroCommand，显然SimpleCommand是对应一个消息进行的处理,在创建具体的commmand的时候需要覆盖SimpleCommand的execute方法。

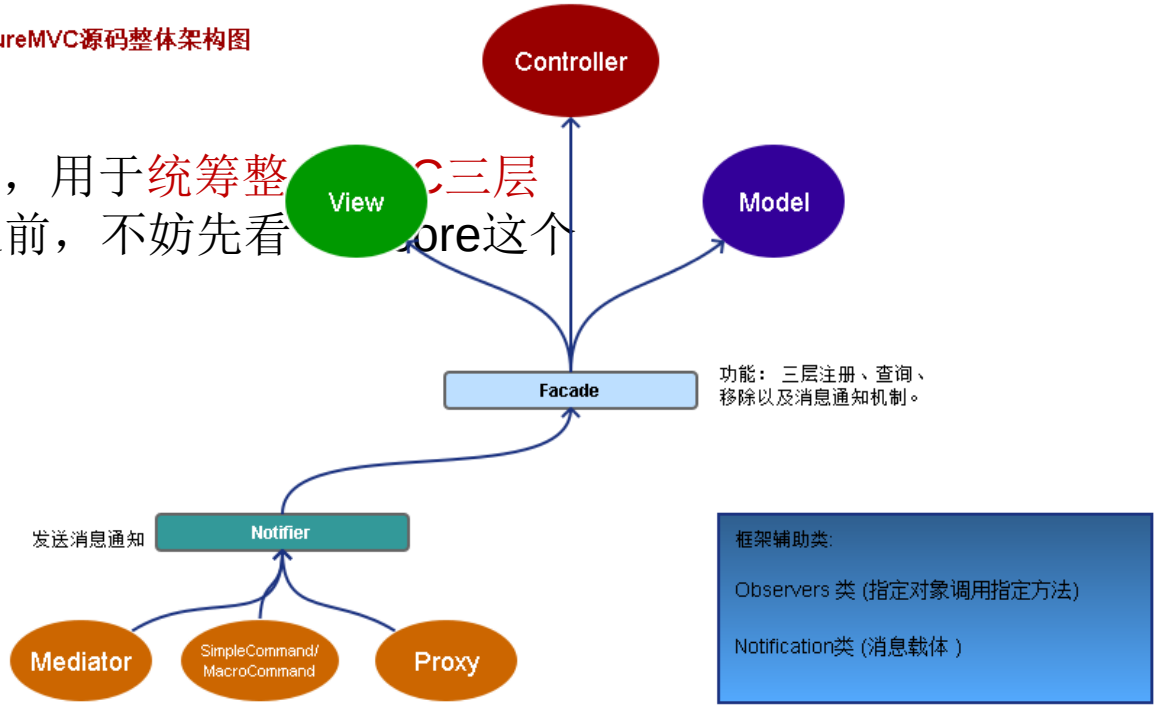
2> MacroCommand则是加载了一个SimpleCommand的队列,按先进先出的顺序执行队列中的SimpleCommand，无论是SimpleCommand和MacroCommand都需实现Icommand接口的execute方法，以便在控制器中能有统一的方法来执行逻辑操作。

源代码目录结构分析(3)

- Patterns目录：
  - Facade

Facade作为一个门面，用于统筹整[illegible]C三层的实现，在查看Facade之前，不妨先看[illegible]ore这个包中的三个类。

  - Mediator

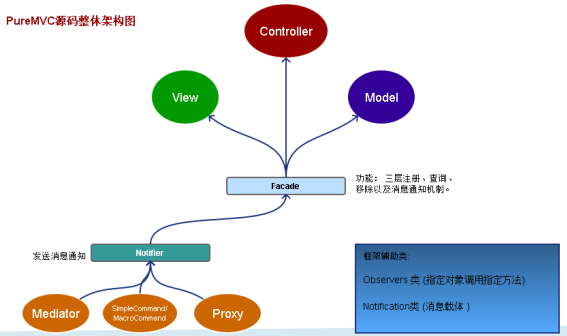


源代码目录结构分析(4)

- Patterns目录：
  - Observer

1> 观察者模式的实现是贯通这个框架的关键。下面对该包下的三个类:Notification,Notifier,Observer进行一下讲述。

2> Notification 类是在pureMVC中传递的消息体，其name属性定义了唯一的消息名，其data属性允许在消息体中附加数据,使得框架的各部分相互作用变得更生动。



源代码目录结构分析(5)

- Patterns目录：
  - Observer（续）

    3> Notifier 类是消息的发送者。 command,proxy,mediator这三个类都继承了Notifier，故在三者中都可对消息(sendNotification)进行传递。

    4> *Observer 类是重点，注意到此类有2个关键属性，notify与context,以及一个关键的方法notifyObserver，对消息响应的统一封装。

  - Proxy

目录

三大核心类分析：Model(1)

- 先分析最简单的 Model.cs 类，得出如下公共特性。

  共同点：

  A: 面向接口编程。

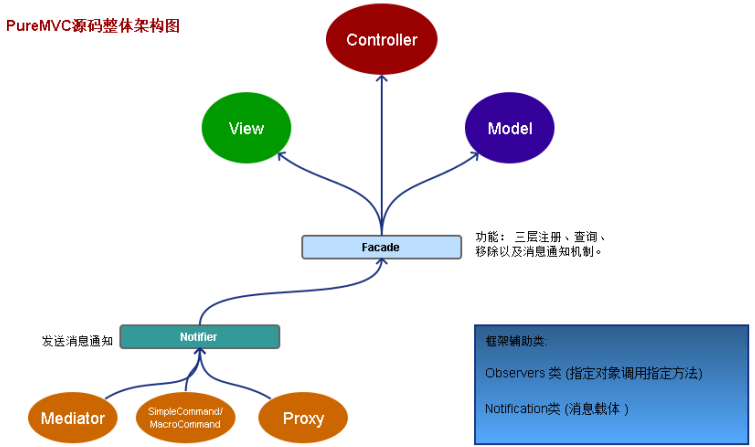
  B: 存在4个方法：

  注册方法 RegisterXXX();

  查询方法 RetrieveXXX();

  删除方法 RemoveXXX();

  是否存在方法 HasXXX();



三大核心类分析：Model(2)

- 先分析最简单的 Model.cs 类，得出如下公共特性（续）

  共同点：

  C: 都是“线程安全”、“延迟加载”的单例类。 [Demo]

  D: 很多方法都是“虚方法”(Virtual)，方便通过子类继承的方式，来重载“修改”框架既有实现。[Demo]

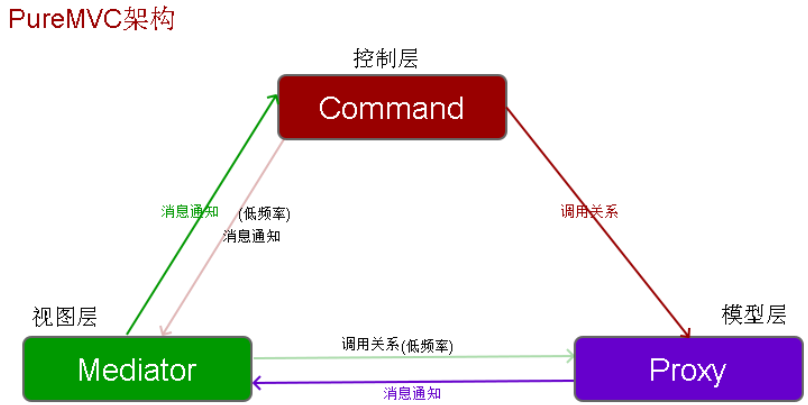
  E: 注册“类实例”时候，调用 OnRegister() 方法。[Demo]

  移除“类实例”，调用 OnRemove() 方法。

三大核心类分析：Model(3)

- 不同点：

Model.cs 类中是没有处理发来消息的定义，即不处理消息。这也就解释PureMVC应用层架构图中，Proxy只能被“方法调用”的原因了。



三大核心类分析：Controller(1)

- 本类分重点字段与方法进行讲解:
  - 重点字段“m_commandMap”定义的类型 IDictionary<string,Type>，表明Controller 的Type 类型，可以运用反射技术动态调用方法。
  - 增加了一个字段“m_view”,View实例的引用。

```
4  using System;
5  using System.Collections.Generic;
6  using PureMVC.Interfaces;
7  using PureMVC.Patterns;
8
9  namespace PureMVC.Core
10 {
11     public class Controller : IController
12     {
13         protected IView m_view;                                  //IView 的引用
14         protected IDictionary<string, Type> m_commandMap;        //Command 类引用的（通知名）映射
15         protected static volatile IController m_instance;        //接口实例
16         protected readonly object m_syncRoot = new object();//锁对象
17         protected static readonly object m_staticSyncRoot = new object();//静态锁
18
19         protected Controller()
20         {
21             m_commandMap = new Dictionary<string, Type>();
22             InitializeController();                              //初始化
23         }
```

三大核心类分析:Controller(2)

- 本类分重点字段与方法进行讲解(续):

  重点方法：

  - RegisterCommand();

    两个功能：

    1> 处理注册的“消息”问题。

    通过 View实例中的RegisterObserver() 方法，实现消息的注册。即实现“消息字符串” 与 对应“执行方法”的绑定。这里通过反射的技术，实现“字符串消息”与“执行方法”的对应关系。

    2> 注册本身。

三大核心类分析:Controller(3)

- 本类分重点字段与方法进行讲解(续):

  重点方法：

  - ExecuteCommand()

    两个功能：

    1> 根据消息字符串，得到对应注册（集合中的）类的Type类型。

    2> 根据Type 类型，得到类实例，进而调用执行接口方法：Execute() 方法。

```
48          public virtual void ExecuteCommand(INotification note)
49          {
50              Type commandType = null;
51
52              lock (m_syncRoot)
53              {
54                  if (!m_commandMap.ContainsKey(note.Name)) return;
55                  commandType = m_commandMap[note.Name];
56              }
57
58              object commandInstance = Activator.CreateInstance(commandType);
59
60              if (commandInstance is ICommand)
61              {
62                  ((ICommand) commandInstance).Execute(note);
63              }
64          }
```

三大核心类分析:View(1)

- 本类功能点可以分两大部分：
  - 消息中心： m_observerMap 字段，以及对应的注册、通知、移除方法。
  - m_mediatorMap 字段以及对应的注册、查询、移除、是否存在方法。

“消息中心”功能分析：

1> RegisterObserver() 注册观察者。

把“消息名称”与“IObserver”实例作为一个“键值对”进行存储。一个“消息名称”，对应一个“IObserver”实例集合（即：IList<Iobserver>）

三大核心类分析:View(2)

- “消息中心”功能分析：
  2> Observer 类代码分析 [功能：对指定对象，调用指定方法]

  - 字段“通知上下文”： m_notifyContext （object 类型）
  - 字段“通知方法名称”： m_notifyMethod （string 类型）其中的 NotifyObserver() 方法，就是使用反射技术，调用“通知上下文”的“通知方法”。

三大核心类分析:View(3)

- “消息中心”功能分析（续）：
  3> NotifyObservers() “通知观察者”方法。
  - 按照通知名称，查询出对应"IObserver" 集合，且实例化对应集合拷贝IList<IObserver> 数据。
  - 循环遍历“IObserver”集合中每一项，且执行（Observer 实例中的)方法“NotifyObserver()”，实现调用（注册时）定义的每一个方法。

三大核心类分析:View(4)

- “消息中心”功能分析（续）：
  NotifyObservers() 方法的详细描述：

1> NotifyObservers 方法进行消息广播，是“消息中心”的关键。Controller和Mediator都是双向接收消息(proxy只能发送消息，而不能接收)。

2> 关于controller的registerCommand方法以及view的registerMediator方法，对事件的响应形式都是以Observer统一起来。在view的notifyObservers方法，其实就是遍历消息与Observer关联的哈希表，当Observer对目前发送的消息感兴趣时，则调用其保存的方法函数，执行实际在command或者mediator的相应方法。

三大核心类分析:View(5)

- “消息中心”功能分析（续）：

4> RemoveObserver() "移除通知"方法

方法的参数是“通知名称”与“通知上下文”，所以本方式可以实现，移除特定的对象。 [即： Observer 中的“上下文”]

三大核心类分析:View(6)

- 本类核心字段功能点分析：（m_mediatorMap 字段）

1> RegisterMediator()

- 注册“消息名称”与对应“Mediator子类对象”到字段“m_mediatorMap”中。
- 获取“Mediator子类对象” 中ListNoticatiionInterests() 返回的字符串集合。
- 实例化“Observer”对象，然后调用“RegisterObserver()”方法，从而实现通过一个“消息”，调用Mediator子类对象的HandleNotification 方法的功能实现。（即： 将消息与Observer之间的对应持久在一个字典表中）

三大核心类分析:View(7)

- 本类核心字段功能点分析（续）：

2> RemoveMediator()

- 根据Mediator子类对象名称，首先(集合中)查询出对应IMediator 的引用。
- 得到这个IMediator 引用的所有“感兴趣”方法集合[通过ListNotificationInterests()]
- 调用“RemoveObserver()”方法，移除注册的对应所有消息。
- 移除对于“m_mediatorMap”的引用。
- 调用Meidiator对象引用的 OnRemove() 方法。
- 返回此Mediator对象引用 （此步骤同Model 的移除方法）

目录

入口与外围类分析:Facade(1)

- Facade 类分析 [门面模式/外观模式]
  - 理解core包的controller，proxy，view 3层的实现之后，facadee里面的代码理解就十分容易了，facade只是做为一个外壳，统一管理3个层次的实现，
  - Facade模式，对应了GoF中的Facade模式，是一种将复杂且庞大的内部实现暴露为一个简单接口的设计模式，例如对大型类库的封装。
  - 在PureMVC中，Facade是与核心层（Model,View,Controller）进行通信的唯一接口，目的是简化开发复杂度。

入口与外围类分析:Facade(2)

- Facader 类分析 (续)
  - 具备三大核心类的引用（作为字段），在自身构造函数中进行实例化。

```
Facade()-->InitializeFacade()
          -->InitializeModel();
          -->InitializeController();
          -->InitializeView();
```

  - 整合三大类（模型、视图、控制三层）所有主要核心方法，即：“注册”、“查询”、“移除”、“是否存在”方法。

入口与外围类分析:Facade(3)

- 辅助类（方法）分析
  - 分析“消息载体封装类”： Notification 类（名称、内容、类型）
  - SendNotification() 方法分析

1> 此方法最终转为调用View实例中的 “NotifyObservers()”方法，从而完成消息与方法执行的对应关系。

2> SendNotification() 流程分析：

Façade 实例的NotifyObservers()--> View 实例的NotifyObservers();

入口与外围类分析:外围类(1)

- 辅助类（方法）分析(续)
  - Mediator、SimpleCommand、MacroCommand、与Notifier 类关系

1> Mediator 定义了m_mdiatorName 字段，保存名称m_ViewComponet 字段 ,保存数据体定义虚方法如下：

- ListNotificationInterests();
- HandleNotification()
- OnRegister();
- OnRemove();

入口与外围类分析:外围类(2)

- 辅助类（方法）分析(续)
  - Notifier

    定义 SendNotifacation()

    1> 代理保有 Facade 的引用。

    2> 间接的调用 Facade 类中的SendNotifacation()方法，从而简化“外围类”调用发送消息的成本。


谢谢大家！
刘老师讲Unity系列教程