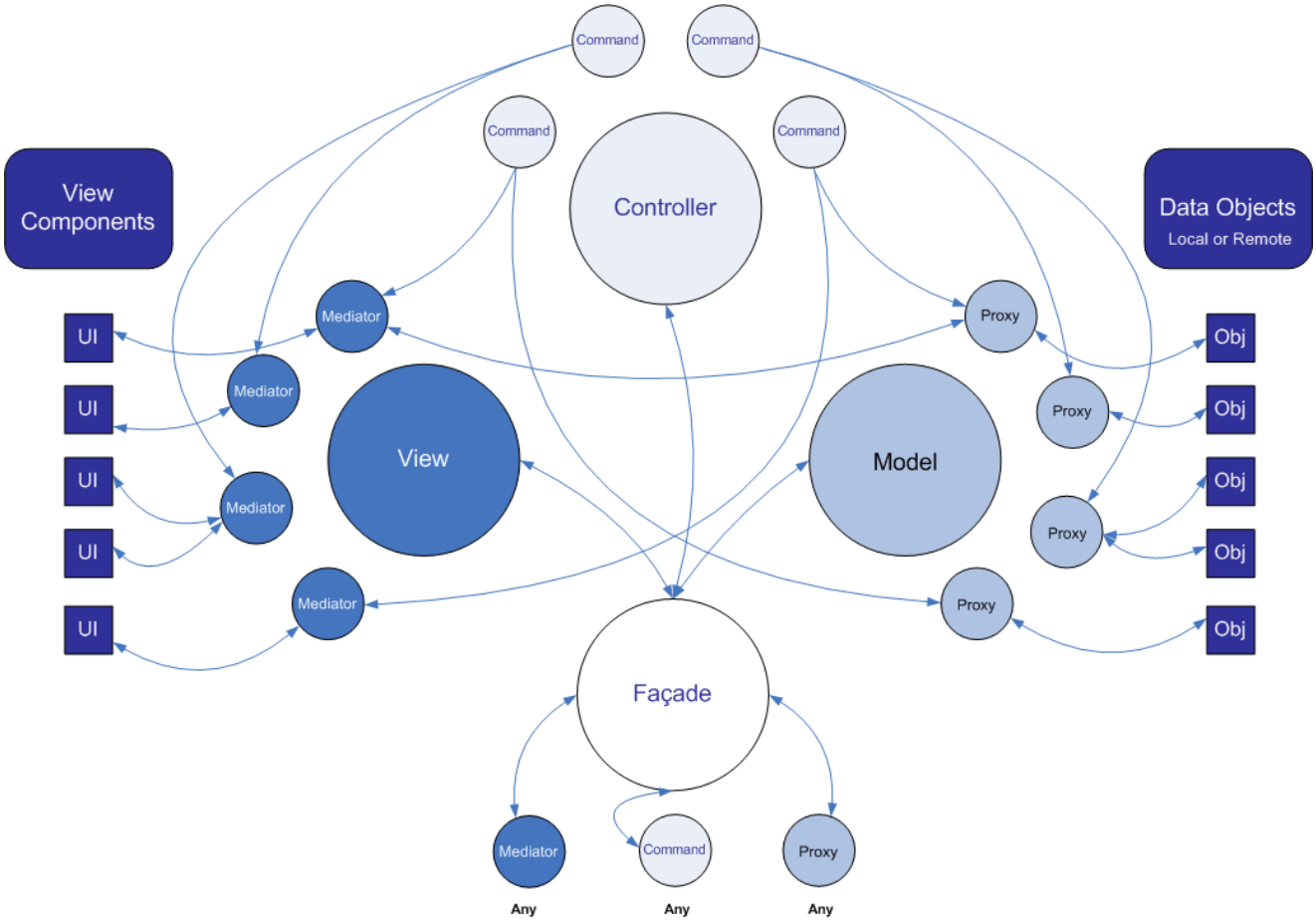
实现
术语阐述
及
最佳实践

用 PureMVC 创建健壮、易扩展、易维护的客户端程序
附 ActionScript 3 及 MXML 实例



作者: Cliff Hall <cliff@puremvc.org>
翻译: 张泽远 <51ajax.net@gmail.com>
Tamt <www.itamt.cn>
最后更新: 5/19/2008

PureMVC 结构 4

- Model 与 Proxy 4
- View 与 Mediator 4
- Controller 与 Command 4
- Façade 与 Core 5
- Observer 与 Notification 5
- Notification可以被用来触发Command的执行 5
- Mediator发送、声明、接收Notification 6
- Proxy发送，但不接收Notification 6

Façade 7

- 具体Façade是什么样子的? 7
- 为程序创建Façade 7
- 初始化Façade 10

Notification 12

- Event与Notification 12
- 定义Notification和Event常量 13

Command 14

- SimpleCommand和MacroCommand的使用 15
- 降低Command与Mediator, Proxy的耦合度 15
- 复杂的操作与业务逻辑 16

Mediator 21

- Mediator的职责 21
- 转化View Component类型 22
- 监听并响应View Component 23
- 在Mediator里处理Notification 25
- Mediator和Proxy之间、Mediator和其他Mediator之间的耦合 27
- 用户与View Component和Mediator的交互 28

Proxy 33

- Proxy的职责 34
- 转换数据对象 34
- 避免对Mediator的依赖 36
- 封装域逻辑 37
- 与Remote Proxy通信 38

启示

PureMVC 是一个定位于设计高性能 RIA 客户端的基于模式的框架。现在它已经被移植到其他的平台上，包括服务器端环境。本篇文档论述针对于客户端。

PureMVC 在不同平台语言下的阐述、实现，PureMVC 所使用的模式在“四人帮”的《**设计模式：可复用面向对象软件的基础**》一书中有很好的论述。

强烈推荐。

PureMVC 结构

PureMVC框架的目标很明确，即把程序分为低耦合的三层：Model、View和Controller。

降低模块间的耦合性，各模块如何结合在一起工作对于创建易扩展，易维护的应用程序是非常重要的。

在PureMVC实现的经典MVC元设计模式中，这三部分由三个单例模式类管理，分别是Model、View和Controller。三者合称为核心层或核心角色。

PureMVC中还有另外一个单例模式类——Façade，Façade提供了与核心层通信的唯一接口，以简化开发复杂度。

Model 与 Proxy

Model 保存对 Proxy 对象的引用，Proxy 负责操作数据模型，与远程服务通信存取数据。

这样保证了 Model 层的可移植性。

View 与 Mediator

View 保存对 Mediator 对象的引用 。由 Mediator 对象来操作具体的视图组件（View Component，例如 Flex 的 DataGrid 组件），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。

这样做实现了把视图和控制它的逻辑分离开来。

Controller 与 Command

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

PureMVC 结构

Controller 与 Command

Command 可以获取 Proxy 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。

Façade 与 Core

Façade 类应用单例模式，它负责初始化核心层（Model，View 和 Controller），并能访问它们的 Public 方法。

这样，在实际的应用中，你只需继承 Façade 类创建一个具体的 Façade 类就可以实现整个 MVC 模式，并不需要在代码中导入编写 Model，View 和 Controller 类。

Proxy、Mediator 和 Command 就可以通过创建的 Façade 类来相互访问通信。

Observer 与 Notification

PureMVC 的通信并不采用 Flash 的 EventDispatcher/Event，因为 PureMVC 可能运行在没有 Flash Event 和 EventDispatcher 类的环境中，它的通信是使用观察者模式以一种松耦合的方式来实现的。

你可以不用关心 PureMVC 的 Observer/Notification 机制是怎么实现的，它已经在框架内部实现了。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。

Notification可以被用来触发Command的执行

Facade 保存了 Command 与 Notification 之间的映射。当 Notification（通知）被

PureMVC 结构

Notification可以被用来触发Command的执行

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。

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。

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 告诉系统。

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。

例如，一个后台管理程序和一个面向用户程序可能共用一个 Model 类。如果只是用例不同，那么 View/Controller 通过传递不同的参数就可以共用相同的 Model 类。

Façade

MVC 元设计模式的核心元素在 PureMVC 中体现为 Model 类、View 类和 Controller 类。为了简化程序开发，PureMVC 应用了 Façade 模式。

Façade 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。Façade 类已经在构造方法包含了对核心 MVC 三者单例的构造。

一般地，实际的应用程序都有一个 Façade 子类，这个 Façade 类对象负责初始化 Controller（控制器），建立 Command 与 Notification 名之间的映射，并执行一个 Command 注册所有的 Model 和 View。

具体Façade是什么样子的?

Façade 类应被当成抽象类，永远不被直接实例化。针对具体的应用程序，你应该具体编写 Façade 的子类，添加或重写 Façade 的方法来实现具体的应用。按照惯例，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 Command，Mediator 和 Proxy。

通常，不同的运行平台都会创建视图结构，尽管创建过程不一样。（比如 Flex 中 MXML 程序负责实例化所有子视图组件，Flash 影片在 Stage 上构建可视对象）。视图结构构建完毕时，整个 PureMVC 机制也已经安置妥当。

创建的 Facade 子类也被用来简化“启动”的过程。应用程序调用 Facade 子类的 startup 方法，并传递自身的一个引用即完成启动，使得应用程序不需要过多了解 PureMVC。

为程序创建Façade

Façade 类的内容很简单。思考下面的例子：

Façade

为程序创建Façade

ApplicationFacade.as:

```
package com.me.myapp
{
	import org.puremvc.as3.interfaces.*;
	import org.puremvc.as3..patterns.facade.*;

	import com.me.myapp.view.*;
	import com.me.myapp.model.*;
	import com.me.myapp.controller.*;

	// MyApp 程序的 Façade 类
	public class ApplicationFacade extends Façade implements IFacade
	{
		//定义 Notification（通知）常量
		public static const STARTUP:String		= "startup";
		public static const LOGIN:String		= "login";

		//得到 ApplicationFacade 单例的工厂方法
		public static function getInstance() : ApplicationFacade
		{
			if ( instance == null ) instance = new ApplicationFacade( );
			return instance as ApplicationFacade;
		}

		//注册 Command，建立 Command 与 Notification 之间的映射
		override protected function initializeController( ) : void
		{
			super.initializeController();
			registerCommand( STARTUP,  StartupCommand );
```

Façade

为程序创建Façade

```
                registerCommand( LOGIN,  LoginCommand );
                registerCommand( LoginProxy.LOGIN_SUCCESS,
            GetPrefsCommand );
        }

        //启动 PureMVC，在应用程序中调用此方法，并传递应用程序本身的引用
        public function startup( app:MyApp ) : void
        {
                sendNotification( STARTUP, app );
        }
    }
}
```

上述代码需要注意以下几点：

- ApplicationFacade 继承自 PureMVC 的 Façade 类，Façade 类实现了 IFacade 接口。
- 这个例子里 ApplicationFacade 没有重写构造方法 。如果重写构造方法，应该在构造方法里先调用父类的构造方法。
- 类方法 getInstance 用于返回 ApplicationFacade 的单例，并将实例保存在父类的一个 protect 变量中。在返回该实例之前必须先把它转化为 ApplicationFacade 类型。
- 定义了 Notification 名称常量。Façade 类是整个系统其他角色相互访问通信的核心，所以在这里定义 Notification（通知）名称常量是最合适的。

Façade

为程序创建Façade

- 初始化 Controller（控制器），并建立 Command 与 Notification 之间的映射，当 Notification（通知）发出时相关的 Command（命令）就会被执行。

- 提供一个带有应用程序类型参数的 startup 方法，该参数能过 Notification 传递到 StartupCommand。

实现这些只需要继承父类很少的功能。

初始化Façade

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

那么，在具体的应用程序中，Façade 是何时何地初始化的呢？请查看下面的 Flex 代码：

MyApp.mxml:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="façade.startup(this)”>

    <mx:Script>
    <![CDATA[
            //获取 ApplicationFacade
            import com.me.myapp.ApplicationFacade;
            private var facade:ApplicationFacade = ApplicationFacade.getInstance();
    ]]>
```

Façade

初始化Façade

```
    </mx:Script>

    <!—视图层的其他代码-->
</mx:Application>
```

上面这个程序很简单：

程序运行时创建视图结构，得到 ApplicationFaçade 实例，调用它的 startup 方法。

注意：在 AIR 中，我们会使用“applicationComplete”代替这里的“creationComplete”；在 Flash 中，我们可以把对 startup 的调用放在第 1 帧或文档类里。

请注意例子中以下几点：

- 我们使用了 MXML 标签，这种普遍的方式，创建程序的界面。以<mx:Application>标签开始，包含组件或容器，不管是 Flex 内嵌的组件还是自定义的。
- 声明并初始化了一个私有变量，（用于）获取 ApplicationFacade 单例。
- 我们初始化这个变量时使用了 getInstance 类方法，在应用程序的 creationComplete 事件触发时，Facade 和相关的 Model，View，Controller 都已经实例化完毕（尽管此时还没有创建任何 Mediator 和 Proxy）。
- 我们在 Application 标签中指定了对 creationComplete 事件的处理过程：调用 startup 方法，并传参主程序的引用。

Façade

初始化Façade

注意：除了顶层的 Application，其他视图组件都不用（不应该）和 Façade 交互。

顶层的 Application（或 Flash 里的 Movie）构建视图结构、初始化 Façade，然后“启动”整个 PureMVC 机制。

Notification

PureMVC 使用了观察者模式，所以各层之间能以一种松耦合的方式通信，并且与平台无关。

ActionScript 语言本身没有提供 flash.events 包中的事件模型。况且 PureMVC 框架并不是只针对 AS 语言，它被移植到其他的一些平台像 C#、J2ME，所以它不会使用这些只有在 Flash 平台上才有的类，它采用自己的通信机制（即 Notification）。

Notification（通知）机制并不仅仅是 Event（事件）机制的替代品，它们的工作方式有本质上的不同。但这两者相互协作可以提高视图组件的可重用性 ，甚至，如果设计得当，视图组件可以和 PureMVC“脱耦”。

Event 与 Notification

Event 是由实现 IeventDispatcher 接口的 Flash 显示对象广播的，Event 会在整个显示对象层中“冒泡”，这样可以让父级（或父级的父级，等）对象处理事件。

Event 机制是一个“责任链”的形式：除了那些可以直接引用事件发起者（dispatcher）并侦听它事件的对象，只有和 dispatcher 是父子关系的对象才会接收到事件，并对事件做出响应动作。

Notification

Event 与 Notification

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。

Notification（通知）有一个可选的“报体”，“报体”可以是任意 ActionScript 对象。[译注：“报体”被用来在 Notification 中携带参数，比如在上一个例子中我们发送了一个携带“app”参数，名字叫 STARTUP 的 Notification。]

与 Flash Event 不同，不需要自定义一个 Notification 类来传值（注：Notification 使用 Object 类型的"报体"来传值）。当然，你可以自定义 Notification 类以强类型的方式交互。这样做的好处是编译时有类型检查，但却会导致你必须管理很多 Notification 类。

Notification 另有一个可选的“类型”参数，用于让接收者作为鉴别依据。

举例，有一个文档编辑器程序，当一个文档对象被打开时，程序会创建对应的 Proxy 对象和 Mediator 对象（Mediator 对象由视图组件使用）。Proxy 对象就可能需要在发送的 Notification 中加入“类型”参数，以让 Mediator 能够唯一标识文档对象。

所有注册了这个 Proxy 对象 Noification 的 Mediator 都会接收到这个 Notification（通知），但它们可以根据 Notification 中“类型”参数决定是否做出反应动作。

定义Notification和Event常量

如前所述，公共的 Notification 名称常量很适合定义在 Façade 中。所有与 Notification 交互的参与者都是 Facade 的协作者（collaborator）。

Notification

定义Notification和Event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。

永远不要把 Event 的名称定义在 Façade 类里。应该把 Event 名称常量定义在那些发送事件的地方，或者就定义在 Event 类里。

在物理层面上审视 Application，如果，View Component 和 Data Object 在和相应的 Mediator 和 Proxy 通信时是通过触发 Event（事件）而不是通过调用方法或发送 Notification，那么 View Component 和 Data Object 就可以保持重用性。

如果一个 View Component（或 Data Object）触发了一个对应 Mediator（或 Proxy）正在侦听的 Event，那么只有这一对协作者（view component－mediator，data object－proxy）需要知道事件的名称，接下来 Mediator（或 Proxy）与 PureMVC 系统其他部分的通信是通过 Notification 进行。

虽然协作者间（Mediator/View，或 Proxy/Data）的关系是有必要紧耦合了。但“协作者对”与程序结构的其它部分形成了松耦合，这样在需求变更时代码的修改范围能有效的缩小。

Command

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

Command

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。

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。

SimpleCommand和MacroCommand的使用

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。

降低Command与Mediator, Proxy的耦合度

通过发送 Notification 通知 Controller 来执行 Command，而且只能由 Controller 实例化并执行 Command。

为了和系统其他部分交互与通信，Command 可能需要：

- 注册、删除 Mediator、Proxy 和 Command，或者检查它们是否已经注册。

Command

降低Command与Mediator, Proxy的耦合度

- 发送 Notification 通知 Command 或 Mediator 做出响应。
- 获取 Proxy 和 Mediator 对象并直接操作它们。

Command 使我们可以很容易地切换视图元素状态，或传送数据给它。

Command 可以调用多个 Proxy 执行事务处理，当事务结束后，发送 Notification 或处理异常和失败。

复杂的操作与业务逻辑

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。

Command 管理应用程序的 Business Logic（业务逻辑），与 Domain Logic（域逻辑）相区别，Business Logic（业务逻辑）要协调 Model 与视图状态。

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。

Command 可能被用于实现一些复杂、必须按照一定顺序的系统行为，上一步动作的结果可能会流入一下个动作。.

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

Command

复杂的操作与业务逻辑

我们这里谈论的 View Component 时就是指像按钮这种用户直接交互的小东西。而 Data Object 则是指能以任意结构存储数据的对象。

Command 与 Mediator 和 Proxy 交互，应避免 Mediator 与 Proxy 直接交互。请看下面这个用于程序“启动”的 Command：

StartupCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.controller.*;

    // 程序开始时执行的 MacroCommand.
    public class StartupCommand extends MacroCommand
    {
        //添加子 Command 初始化 MacroCommand.
       override protected function initializeMacroCommand() : void
       {
            addSubCommand( ModelPrepCommand );
            addSubCommand( ViewPrepCommand );
       }
    }
}
```

这是一个添加了两个子 Command 的 MacroCommand，执行时两个版子命令会按照“先进先出”（FIFO）的顺序被执行。

Command

复杂的操作与业务逻辑

这个复合命令定义了 PureMVC 在“开启”（startup）时的动作序列。但具体的，我们应该做什么？按照什么顺序？

在用户与数据交互之前，Model 必须处于一种一致的已知的状态。一旦 Model 初始化完成，视图就可以显示数据允许用户操作与之交互。

因此，一般“开启”（startup）过程有两个主要的动作：Model 初始化与 View 初始化。

ModelPrepCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.observer.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.*;
    import com.me.myapp.model.*;

    //创建 Proxy 对象，并注册。
    public class ModelPrepCommand extends SimpleCommand
    {
        //由 MacroCommand 调用
        override public function execute( note : INotification ) : void

        {
            facade.registerProxy( new SearchProxy() );
            facade.registerProxy( new PrefsProxy() );
            facade.registerProxy( new UsersProxy() );
        }
```

```
    }
}
```

Model 的初始化通常比较简单：创建并注册在“开启”过程中需要用到的 Proxy。

上面这个 ModelPrepCommand 类是一个 SimpleCommand 例子，它功能就是初始化 Model，它是前面那个 MacroCommand 的第一个子命令，所以它会最先被执行。

通常具体的 Façade 对象，它创建并注册了多个在“启动”（startup）过程中会用到的 Proxy 类。注意这里 Command 并没有操作或初始任何的 Model 数据。Proxy 的职责才是取得，创建，和初始化数据对象。

ViewPrepCommand.as:

```
package com.me.myapp.controller
{
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.observer.*;
    import org.puremvc.as3.patterns.command.*;

    import com.me.myapp.*;
    import com.me.myapp.view.*;

    //创建 Mediator，并把它们注册到 View.
    public class ViewPrepCommand extends SimpleCommand
    {
        override public function execute( note : INotification ) : void
                                    {
            var app:MyApp = note.getBody() as MyApp;
```

Command

复杂的操作与业务逻辑

```
            facade.registerMediator( new ApplicationMediator( app ) );
        }
    }
}
```

这个 SimpleCommand 的功能是初始化 View。它是之前那个 MacroCommand 的最后一个子命令，所以它会被最后一个执行。

注意这个命令唯一创建并注册的 Mediator 是 ApplicationMediator，ApplicationMediator 被用于操作 Appliction View。

更深入一点：它向创建 Mediator 对象时，向 Mediator 构造函数传参 Notification 的“报体”。这个“报体”是个对 Application 的引用，它是在最初的“启动通知”（STARTUP Notification）发出时由 Application 自身发送的。（请查看之前的 MyApp 范例。）

Application 是个有点特殊的视图组件（View Component），它包含其他所有视图组件（View Component）并在启动时创建它们。

为了和系统其他部分通信，视图组件需要使用 Mediator。创建 Mediator 对象需要引用此 Mediaotr 管理的视图组件(View component)的引用，这些 Mediator 和 View Component 的对应关系只有 Application 知道。

比较特殊的是 Application 的 Mediator，它是唯一的被允许知道 Application 一切的类，所以我们会在 Application Mediator 的构造函数中创建其他的 Mediator 对象。

上面的三个 Command 初始化了 Model 和 View。这样做 Command 不需要知道太多 Model 和 View 的细节。

当 Model 或 View 的实现发生改变时就只需要改变 Model 和 View 和相应部分即可，而不需要改变 Command 的逻辑。

Command 的业务逻辑应该避免被 Model 或 View 的变化而受到影响。

Command

复杂的操作与业务逻辑

Model 应该封装域逻辑(domain logic)，保证数据的完整性。Command 则处理事务或业务逻辑，协调多个 Proxy 的操作，处理异常等。

Mediator

Mediator 是视图组件（View Component，例如 Flex 的 DataGrid 或 Flash 的 MovieClip）与系统其他部分交互的中介器。

在基于 Flash 的应用程序中，Mediator 侦听 View Component 来处理用户动作和 Component 的数据请求。Mediator 通过发送和接收 Notification 来与程序其他部分通信。

Mediator的职责

Flash、Flex 和 AIR 框架都提供了丰富强大的交互 UI 组件。你可以扩展这些组件或者编写自己的组件为用户提供更强大的交互方式和数据呈现方式。

在不远的将来，会有越来越多的平台运行 ActionScript 语言。PureMVC 也已经有移植到其他平台上，包括 Silverlight 和 J2ME，拓宽了 PureMVC 技术的 RIA 开发道路。

PureMVC 的目标之一就是保持平台无关性，不管你使用什么技术什么 UI 组件什么数据结构都能应用 PureMVC 框架。

对基于 PureMVC 的应用程序来说，View Component 可以是任意的 UI Component，不用管所处的框架是什么（Flex，Java 还是 C#），也不用管它有多少个组件。一个 View Component 应该把尽可能自己的状态和操作封装起来，对外只提供事件、方法和属性的简单的 API。

Mediator 保存了一个或多个 View Component 的引用，通过 View Component 自身提供的 API 管理它们。

Mediator

Mediator的职责

Mediator 的主要职责是处理 View Component 派发的事件和系统其他部分发出来的 Notification（通知）。

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。

转化 View Component 类型

PureMVC 的 Mediator 基类在构造方法中提供两个参数：name（名称）和一个 Object 类型的对象。

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。

在 Mediator 被构造之后，你可能通过调用它的 setViewComponent 函数来动态给它的 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。

ActionScript 语言提供隐式 setter 和 getter。隐式的 setter 和 getter 看起来像方法，但对外是属性。实践证明 setter 和 getter 对解决频繁转换类型问题是很好的解决方法。

一种常用做法是在具体的 Mediator 中提供一个隐式 getter 来对 View Component 进行类型转换，注意给这个 getter 命名一个合适的名字。

示例：

Mediator

转化 View Component 类型

```
protected function get controlBar() : MyAppControlBar
{
        return viewComponent as MyAppControlBar;
}
```

之后，在 Mediator 的其他地方，我们不再这样做了：

```
MyAppControlBar ( viewComponent ).searchSelction =
MyAppControlBar.NONE_SELECTED;
```

我们会这样：

```
controlBar.searchSelction =
MyAppControlBar.NONE_SELECTED;
```

监听并响应 View Component

通常一个 Mediator 只对应一个 View Component，但却可能需要管理多个 UI 控件，比如一个 ApplicationToolBar 和它包含的 button 或 control。我们可以把一组相关的 Control（比如 from）放在一个 View Component 里，把这组相关的控件当作一个 View Component，对 Mediator 来说，这些控件是这个 View Component 的属性，应尽可能的封装它们的操作与之交互。

Mediator 负责处理与 Controller 层、Model 层交互，在收到相关 Notification 时更新 View Component。

在 Flash 平台上，一般在构造方法中或 setViewComponent 方法被调用后，给 View Component 添加事件监听器。

```
        controlBar.addEventListener( AppControlBar.BEGIN_SEARCH,
onBeginSearch );
```

Mediator

监听并响应 View Component

Mediator 按需求对事件做出响应。

一般地，一个 Mediator 的事件响应会有以下几种处理：

- 检查事件类型或事件的自定义内容。
- 检查或修改 View Component 的属性（或调用提供的方法）。
- 检查或修改 Proxy 对象公布的属性（或调用提供的方法）。
- 发送一个或多个 Notification，通知别的 Mediatora 或 Command 作出响应（甚至有可能发送给自身）。

下面是一些有用的经验：

- 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。
- 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。
- 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。
- 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。

Mediator

在 Mediator 里处理 Notification

与给视图添加严格的事件监听器相比，Mediator 与 PureMVC 系统的其它部分关联起来是简单而且自动化的。

在 Mediator 实例化时，PureMVC 会调用 Mediator 的 listNotificationInterests 方法查询其关心的 Notification，Mediator 则在 listNotificationInterests 方法中以数据形式返回这些 Notification 名称。

最简单的方法是返回一个由 Notification 名称组成的匿名数组。前面提过，Notification 名称通常以常量形式定义在 Façade 类中。

下面是个例子：

```
override public function listNotificationInterests() : Array
{
    return [
            ApplicationFacade.SEARCH_FAILED,
            ApplicationFacade.SEARCH_SUCCESS
           ];
}
```

当这个数组里的一个 Notification 被系统的其他部分（也可能是 Mediator 对象自身）发出时，Mediator 对象的 handleNotification 函数会被调用，并传进 Notification 参数。

在 handleNotification 函数里，使用的是“switch/case”而不是“if/else if”的分支控制，因为前者更易读，而且增加、删除一个 Notification 比较方便。

本质上来讲，在收到一个 Notification 时 Mediator 是所要操作的是很少的。有时候（偶尔），我们需要从 Notification 里获取有关 Proxy 的信息，但记住，不应该让处理 Notification 的方法负责复杂逻辑。业务逻辑应该放在 Command 中而非在 Mediator 中。

Mediator

在 Mediator 里处理 Notification

```
override public function handleNotification( note : INotification ) : void
{
    switch ( note.getName() )
    {
            case ApplicationFacade.SEARCH_FAILED:
                    controlBar.status = AppControlBar.STATUS_FAILED;
                    controlBar.searchText.setFocus();
                    break;

            case ApplicationFacade.SEARCH_SUCCESS:
                    controlBar.status = AppControlBar.STATUS_SUCCESS;
                    break;

            }
    }
}
```

还有，一般一个 Mediator（handleNotification 方法）处理的 Notification 应该在 4、5 个之内。

还要注意的是，Mediator 的职责应该要细分。如果处理的 Notification 很多，则意味着 Mediator 需要被拆分，在拆分后的子模块的 Mediator 里处理要比全部放在一起更好。

[译注：Mediator 需要侦听来自 View Component 的 Event，也需要侦听来自其它地方（比如：Proxy 的数据更新）的 Notification，Mediator 在侦听这两者的方式上有没有什么区别呢？] Mediator 监听 Notification 与监听 Event 之间的关键不同在于，使用一个简单的预定义的处理 Notification 的方法，即 handleNotification 方法。

Mediator

在 Mediator 里处理 Notification

通常对于每一个事件都需要添加一个监听方法，一般这些方法只是发送 Notification，处理逻辑应该不复杂，也不应该涉及太多的 View Component 细节，因为对视图 View Component 的处理应该尽可能的封装起来。

对于 Notification，Mediator 有唯一的一个处理方法，这个方法中它会（通过 switch/case）处理所有的 Notification。

最好是把对所有 Notification 的处理放在 handleNotification 方法中，使用 switch/case 来区分 Notification 名称。

已经有很多关于“switch/case”用法的争论，很多开发者认为它有局限性，因为所有的情况都在一个函数里处理了。不过，单一的 Notification 处理函数以及“switch/case”分支风格是 PureMVC 特意选定的。这样做可以限定 Mediator 的代码量，并让代码保持 PureMVC 推荐的结构。

Mediator 是被用来负责 View Component 和系统其他部分的通信的。

就像一个翻译员在联合国大会（UN conference）上给大使们翻译对话。她应当尽量少做翻译（转发信息）之外的事情，偶尔可以引用一下比喻、事实。Mediator 在 PureMVC 中也是这样的一个角色。

Mediator 和 Proxy 之间、Mediator 和其他 Mediator 之间的耦合

View 本质上是显示 Model 的数据并让用户能与之交互，我们期望一种单向依赖，即 View 依赖于 Model，而 Model 却不依赖于 View。View 必须知道 Model 的数据是什么，但 Model 却并不需要知道 View 的任何内容。

虽然 Mediator 可以任意访问 Proxy，通过 Proxy 的 API 读取、操作 Data Object，但是，由 Command 来做这些工作可以实现 View 和 Model 之间的松耦合。

Mediator

Mediator 和 Proxy 之间、Mediator 和其他 Mediator 之间的耦合

同样的情况，虽然 Mediator 可以从 View 获取其他的 Mediator，通过 API 访问、操作它们。但这样是很不好的，它会导致 View 下成员的相互依赖，这违反了“改变一个不影响其他”的目的。

如果一个 Mediator 要和其他 Mediator 通信，那它应该发送 Notification 来实现，而不是直接引用这个 Mediator 来操作。

Mediator 对外不应该公布操作 View Component 的函数。而是自己接收 Notification 做出响应来实现。

如果在 Mediator 里有很多对 View Component 的操作（响应 Event 或 Notification），那么应该考虑将这些操作封装为 View Component 的一个方法，提高可重用性。

如果一个 Mediator 有太多的对 Proxy 及其数据的操作，那么，应该把这些代码重构在 Command 内，简化 Mediator，把业务逻辑（Business Logic）移放到 Command 上，这样 Command 可以被 View 的其他部分重用，还会实现 View 和 Model 之间的松耦合提高扩展性。

用户与 View Component 和 Mediator 的交互

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）。

LoginPanel.mxml:

Mediator

用户与 View Component 和 Mediator 的交互

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml"
    title="Login" status="”{loginStatus}”>

    <!—
    The events this component dispatches. Unfortunately we can’t use
    the constant name here, because MetaData is a compiler directive
     -->
    <mx:MetaData>
            [Event('tryLogin')];
    </mx:MetaData>

    <mx:Script>
    <![CDATA[
            import com.me.myapp.model.vo.LoginVO;
            // 表单项与LoginVO对象的属性双向绑定。
            [Bindable] public var loginVO:LoginVO = new LoginVO();
            [Bindable] public var loginStatus:String = NOT_LOGGED_IN;

            //定义Event名称常量
            public static const TRY_LOGIN:String='tryLogin';
            public static const LOGGED_IN:String='Logged In';
            public static const NOT_LOGGED_IN:String='Enter Credentials';
    ]]>
    </mx:Script>

    <mx:Binding source="username.text"
destination="loginVO.username"/>
    <mx:Binding source="password.text"
destination="loginVO.password"/>
```

Mediator

用户与 View Component 和 Mediator 的交互

```
    <!—The Login Form -->
    <mx:Form id="loginForm" >
        <mx:FormItem label="Username:">
                <mx:TextInput id="username"
text="{loginVO.username}" />
        </mx:FormItem>
        <mx:FormItem label="Password:">
                <mx:TextInput id="password"
text="{loginVO.password}"
                        displayAsPassword="true" />
        </mx:FormItem>
        <mx:FormItem >
          <mx:Button label="Login" enabled="{loginStatus == NOT_LOGGED_IN}"
                click="dispatchEvent( new Event(TRY_LOGIN, true ));"/>
        </mx:FormItem>
   </mx:Form>
</mx:Panel>
```

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。

Mediator

用户与 View Component 和 Mediator 的交互

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
        {
```

Mediator

用户与 View Component 和 Mediator 的交互

```
                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按钮，尝试登录。
        private function onTryLogin ( event:Event ) : void  {
                sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );
        }

        // 把viewComponent转化成它真正的类型。
        protected function get loginPanel() : LoginPanel  {
                return viewComponent as LoginPanel;
```

Mediator

用户与 View Component 和 Mediator 的交互

```
            }
        }
    }
```

注意 LoginPanelMediator 在构造方法中给 LoginPanel 注册了一个侦听方法——onTryLogin，当用户单击 Login 按钮时这个方法会被执行。在 onTryLogin 方法里发送了一个 LOGIN 的 Notification（通知，携带参数 LoginVO 对象）。

早先（在 ApplicationFacade 中）我们已经把 LoginCommand 注册到这个 Notification 上了。LoginCommand 会调用 LoginProxy 的“登录”方法，传参 LoginVO。LoginProxy 把“登录”请求远程服务，之后发送 LOGIN_SUCCESS（登录成功）或 LOGIN_FAILED（登录失败）的 Notification。这些类的定义请参见“Proxy”章节。

LoginPanelMediator 把 LOGIN_SUCCESS 和 LOGIN_FAILED 注册自己的关心的 Notification，当这两个 Notification 被发送时，Mediaotr 作为响应把 LoginPanel 的 loginStatus 设置为 LOGGED_IN（登录成功时）或 NOT_LOGGED_IN（登录失败时），并清除 LoginVO 对象。

Proxy

一般来说，Proxy Pattern（代理模式）被用来为控制、访问对象提供一个代理。在基于 PureMVC 的应用程序，Proxy 类被设计用来管理程序数据模型。

一个 Proxy 有可能管理对本地创建的数据结构的访问。它是 Proxy 的数据对象。

在这种情况下，通常会以同步的方式取得或设置数据。Proxy 可能会提供访问 Data Object 部分属性或方法的 API，也可能直接提供 Data Object 的引用。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。

Remote Proxy 被用来封装与远程服务的数据访问。Proxy 维护那些与 Remote service（远程服务）通信的对象，并控制对这些数据的访问。

Proxy

在这种情况下，调用 Proxy 获取数据的方法，然后等待 Proxy 在收到远程服务的数据后发出异步 Notification。

Proxy 的职责

Proxy 封装了数据模型，管理 Data Object 及对 Data Object 的访问，不管数据来自哪里，什么类型。

在 PureMVC 中，Proxy 是个被 Model 注册的简单的数据持有者。

虽然 Proxy 类已经是完全可用的了，但是通常对于具体的应用你应该编写 Proxy 的子类，增加操作方法。

通常 Proxy Pattern 有以下几种类型：

- *Remote Proxy*，当 Proxy 管理的数据存放在远程终端，通过某种服务访问。
- *Proxy and Delegate*，多个 Proxy 共享对一个服务的访问，由 Delegate 封装对服务的控制访问，确保响应正确的返回给相应的请求者。
- *Protection Proxy*，用于数据对象的访问有不同的权限时。
- *Virtual Proxy*，对创建开销很大的数据对象进行管理。
- *Smart Proxy*，首次访问时载入数据对象到内存，并计算它被引用的次数，允许锁定确保其他对象不能修改。

转换数据对象

Proxy 基类的构造方法接受一个名称(name)和一个 Object 类型的参数，Object 类型的参数用来设置 Proxy 管理的数据模型，在构造方法完成后也可以调用

Proxy

转换数据对象

setData 方法来设置。

就像 Mediator 和它的 View Component 一样，为了访问它的属性和方法，你会经常需要把这个 Data Object 转化成它真正的类型。看起来只是重复繁琐了一些，但更严重的是可能会暴露过多的 Data Object 细节。

另外，因为 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。

再一次的，ActionScript 语言支持隐式的 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。

一个很好的惯用做法是在你具体的 Proxy 类中引入一个适当命名的隐式 getter，用来把 Data Object 转化它真正的类型。

另外，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。

比如：

```
public function get searchResultAC () : ArrayCollection
{
        return data as ArrayCollection;
}

public function get resultEntry( index:int ) : SearchResultVO
{
        return searchResultAC.getItemAt( index ) as
SearchResultVO;
}
```

Proxy

转换数据对象

在别的 Mediator 中，我们不用再这样做了：

```
var item:SearchResultVO =
ArrayCollection ( searchProxy.getData() ).lastResult.getItemAt( 1 ) as
SearchResultVO;
```

而可以这样：

```
var item:SearchResultVO = searchProxy.resultEntry( 1 );
```

避免对 Mediator 的依赖

Proxy 不监听 Notification，也永远不会被通知，因为 Proxy 并不关心 View 的状态。但是，Proxy 提供方法和属性让其它角色更新数据。

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

它应该采取的方式是发送 Notification（这些 Notification 可能被 Command 或 Mediator 响应）。Proxy 不关心这些 Notification 被发出后会影响到系统的什么。

把 Model 层和系统操作隔离开来，这样当 View 层和 Controller 层被重构时就不会影响到 Model 层。

但反过来就不是这样了：Model 层的改变很难不影响到 View 层和 Controller 层。毕竟，它们存在的目的就是让用户与 Model 层交互的。

Proxy

封装域逻辑

Model 层中的改变总会造成 View/Controller 层的一些重构。

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。

Proxy 不仅仅用来管理数据对象的访问，而且用来封装数据对象的操作使得数据维持在一个合法的状态。

比如，计算营业税是一个Domain Logic（域逻辑），它应该放在Proxy中实现而不是 Mediator 或 Command。

虽然可以放在任意一个中实现，但是把它放在 Proxy 中实现不仅仅是出于逻辑（logic）上的考虑，这样做还可以保持其它层更轻便、更易被重构。

一个 Mdeiator 可能获取（retrieve）这个 Proxy 对象；调用它的营业税计算方法，传参一些表单项目数据。如果把真正的计算放在 Mediator 的话，就是把 Domain Logic（域逻辑）嵌在 View 层了。对营业税的计算是 Domain Model（域模型）中的一条规则。View 仅仅是把它看成 Domain Model 的一个属性，当用户的输入正确这个属性对 View 就可用。

假设你现在正在工作的程序项目是一个嵌在浏览器中桌面级的 RIA 解决方案。但新的版本可能是简化了用例（use case）的嵌在 PDA 中的解决方案，但仍然完全需要当前程序项目的 Model 层。

如果已经有正确的分离操作，我们就可以完全重用 Model 层而只需开发新的 View 层和 Controller 层。

虽然把对营业税的计算放在 Mediator 上看起来很有效而且在写代码时也很容易；你可能只需要把营业税计算出来交给 Model 而已。

然而你却需要在程序的不同版本中重复的付出，在每个新的 View 层复制粘贴营业税计算逻辑，所以最好把这段逻辑放在 Model 层。

Proxy

与 Remote Proxy 通信

Remote Proxy 对象是一个从远程位置（Remote location）获取 Data Object 的 Proxy。这通常意味着我们与它的交互是以异步的方式。

Proxy 获取数据的方式取决于客户端平台、远程服务（remote service）的实现和开发人员的选择。在 Flash/Flex 环境中，我们可能会使用 HTTPService，WebService，RemoteObject，DataService 或者 XMLSocket 来从 Proxy 中发送服务请求。

根据需要，Remote Proxy 可能动态的发送请求，响应时会设置一个属性或调用一个方法； 或只在构造方法中发送一次请求，然后提供访问数据的 get/set 方法。

在 Proxy 中有很多东西可以优化以提高与远程服务通信的效率。

比如：缓存（服务器返回的）数据以减少网络通信的“废话”；只发送改变的数据，减少带宽的浪费。

如果请求是由系统其它角色动态调用 Remote Proxy 的方法而发出的，那 Proxy 在结果返回来时应该发送一个 Notification。

注意关心这个 Notification 的角色有可能并不是发起这个数据请求的那个角色。

举例，调用远程服务的查询并显示返回的结果，这个过程通常会有以下几步：

- 一个 View Component 触发一个事件发起一个查询请求。
- 它的 Mediator 响应：获取相应的 RemoteProxy，设置它的 searchCriteria 属性。
- Proxy 的 searchCriteria 属性其实是一个隐式 setter，它会保存赋值，通过内部的 HTTPService（它会侦听 result 和 fault 事件）初始查询请求。

Proxy

与 Remote Proxy 通信

- 当服务返回结果时，HTTPService 会触发 ResultEvent 事件，Proxy 响应，把结果保存在公共属性中。

- Proxy 然后发送一个 Notification 表示请求成功，这个 Notification 会绑定一个对数据对象的引用作为“报体”。

- 关心这个 Notification 的另外一个 Mediator 就会响应这个 Notification，把“报体”中的数据赋值给它所操作的 View Component 的 dataProvider 属性。

再来，假设有一个 LoginProxy，它有一个 LoginVO（一个 Value Object；简单的数据载体类）。LoginVO 可能看起来像这样：

```
package com.me.myapp.model.vo
{
	//把这个AS3 VO映射到Remote Class
	[RemoteClass(alias="com.me.myapp.model.vo.LoginVO")]

	[Bindable]
	public class LoginVO
	{
	    public var username: String;
	    public var password: String;
	    public var authToken: String;//登录权限允许时服务器设置此值
	}
}
```

LoginProxy 对外提供方法设置登录数据，登录（logging in），退出（logging out），获取“登录”时用到的权限标识。

Proxy

与 Remote Proxy 通信

LoginProxy:

```
package com.me.myapp.model
{
    import mx.rpc.events.FaultEvent;
    import mx.rpc.events.ResultEvent;
    import mx.rpc.remoting.RemoteObject;
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.proxy.Proxy;
    import com.me.myapp.model.vo.LoginVO;

    // 用于用户登录的Proxy
    public class LoginProxy extends Proxy implements IProxy {
            public static const NAME:String             = 'LoginProxy';
            public static const LOGIN_SUCCESS:String    = 'loginSuccess';
            public static const LOGIN_FAILED:String     = 'loginFailed';
            public static const LOGGED_OUT:String       = 'loggedOut';
            private var loginService: RemoteObject;

            public function LoginProxy ()  {
                    super( NAME, new LoginVO ( ) );
                    loginService = new RemoteObject();
                    loginService.source = "LoginService";
                    loginService.destination = "GenericDestination";
                    loginService.addEventListener( FaultEvent.FAULT,
    onFault );
                    loginService.login.addEventListener( ResultEvent.RESUL
    T, onResult );
            }
```

Proxy

与 Remote Proxy 通信

```
// 隐式getter，转化data的类型
public function get loginVO( ) : LoginVO {
        return data as LoginVO;
}

//如果logivVO中包含了authToken（授权标识）表示用户登录成功
public function get loggedIn():Boolean {
        return ( authToken != null );
}

// 取得authToken
public function get authToken():String {
        return loginVO.authToken;
}

//设置用户的权限标识，登录，退出，或继续尝试登录。
public login( tryLogin:LoginVO ) : void {
        if ( ! loggedIn )          {
                loginVO.username= tryLogin.username;
                loginVO.password = tryLogin.password;
        } else {
                logout();
                login( tryLogin );
        }
}

// 退出，简单地清空LoginVO
public function logout( ) : void
```

Proxy

与 Remote Proxy 通信

```
            {
                if ( loggedIn ) loginVO = new LoginVO( );
                sendNotification( LOGGED_OUT );
            }

            //通知系统登录成功
            private function onResult( event:ResultEvent ) : void
            {
                setData( event.result ); // immediately available as
    loginVO
                sendNotification( LOGIN_SUCCESS, authToken );
            }

            //通知系统登录失败
            private function onFault( event:FaultEvent) : void
            {
                sendNotification( LOGIN_FAILED,
    event.fault.faultString );
            }
    }
}
```

一个 LoginCommand 会获取 LoginProxy，设置登录的数据，调用登录函数，呼叫登录服务。

接下来，可能一个 GetPrefsCommand 会响应 LOGIN_SUCCESS（登录成功）这个 Notification，从 Notificaiton 的“报体”中获取 authToken（授权标识），接着呼叫下一个服务，获取用户的（比如）配置信息（preferences）。

Proxy

与 Remote Proxy 通信

LoginCommand:

```
package com.me.myapp.controller {
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.command.*;
    import com.me.myapp.model.LoginProxy;
    import com.me.myapp.model.vo.LoginVO;

    public class LoginCommand extends SimpleCommand {
            override public function execute( note: INotification ) : void  {
                    var loginVO : LoginVO = note.getBody() as LoginVO;
                    var loginProxy: LoginProxy;
                    loginProxy = facade.retrieveProxy( LoginProxy.NAME ) as
                    LoginProxy;
                    loginProxy.login( loginVO );
            }
    }
}
```

GetPrefsCommand:

```
package com.me.myapp.controller {
    import org.puremvc.as3.interfaces.*;
    import org.puremvc.as3.patterns.command.*;
    import com.me.myapp.model.LoginProxy;
    import com.me.myapp.model.vo.LoginVO;

    public class GetPrefsCommand extends SimpleCommand {
            override public function execute( note: INotification ) : void  {
                    var authToken : String = note.getBody() as String;
                    var prefsProxy : PrefsProxy;
```

Proxy

与 Remote Proxy 通信

```
                    prefsProxy = facade.retrieveProxy( PrefsProxy.NAME ) as
                    PrefsProxy;
                    prefsProxy.getPrefs( authToken );
                }
            }
        }
```