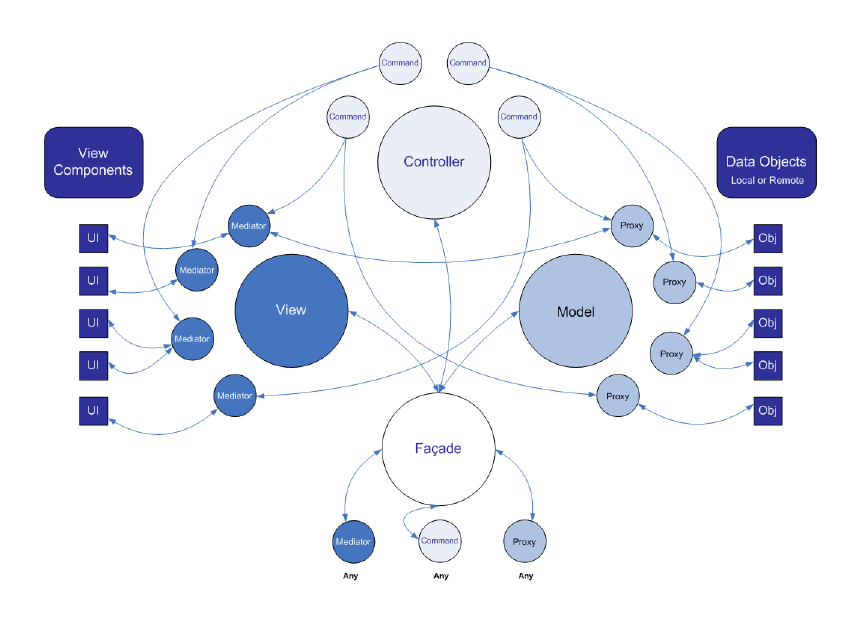
**Pure MVC for cocos2d javascript.**

**v.1.0(2014-9-10)**



## 一、研究目的

在做了一个老虎机项目后，发觉pureMVC架构的优缺点都比较明显。

优点包括：结构比较明晰，模块之间可以任意嫁接，健壮性和可扩展性强，可以实现流水线生产游戏或应用。

缺点包括：结构较为复杂，原来一两行代码可以完成的功能，可能在pureMVC里面会封装到几个类里面去执行（尤其对于javascript这种弱引用语言来说），但对于用户来说，pureMVC是一个黑盒结构，用户不需要知道它内部工作流程，只要知道它能提供哪些接口和如何使用这些接口来执行相关功能就可以了，因此，可能在工作量上，我们会花费更多的可能是学习如何去用它的API。

综上所述，如果我们能花时间去了解一下pureMVC的内部结构，制作一个自己的框架结构，会是一个一劳永逸的事情。

代码仓库：https://github.com/VirtuosZhangCH/js-pureMVCWeb.git

## 二、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。

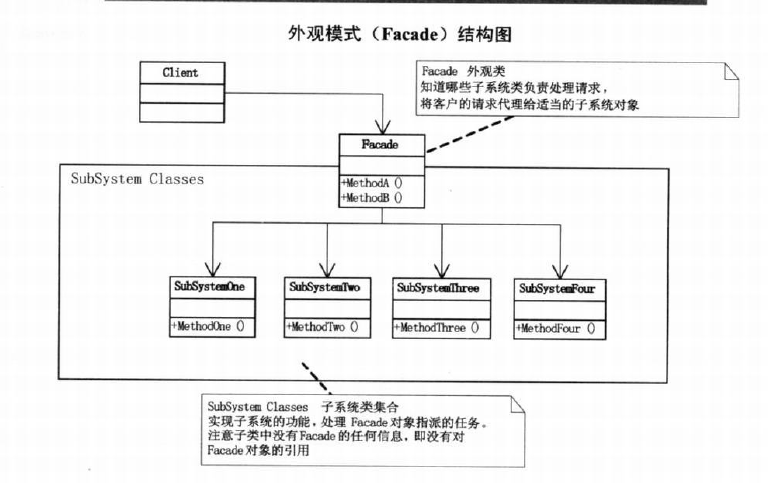
降低模块间的耦合性，各模块如何结合在一起工作对于创建易扩展，易维护的应用程序是非常重要的。

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。

在整个PureMVC中，Facade是一个封装入口单例，本身就是一个比较经典的设计模式，外观模式。

它的作用就是提供接口给用户来启动，注册MVC单例等，提供开放封闭的架构接口，使用户不需要去了解底层实现。



## 三、启动pureMVC：

var \_facade = SlotFacade.getInstance();

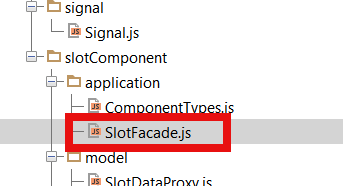
\_facade.startup(SlotDataProxy,SlotStatesProxy,layer,SlotGameViewMediator);

facade 是一个单例，启动facade一般使用继承自抽象facade的具体facade的自定义startup函数来执行，在startup函数里面传入相应的需要实例化的具体proxy，view，mediator，并在函数内部实现具体注册。

在pureMVC的官方文档里面有这么一段话：

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。

例如：我这里自己定义了一个slotFacade，它继承自Facede



startup:function(definitionOfSlotDataProxy,definitionOfSlotStatesProxy,mainView,mainViewMediator)

{

var slotDataProxy = new definitionOfSlotDataProxy();

var slotStatesProxy = new definitionOfSlotStatesProxy();

if(!slotDataProxy){

throw new Error("definitionOfSlotDataProxy must be a descendent of SlotDataProxy");

return;

}

if(!slotStatesProxy){

throw new Error("definationOfSlotStatesProxy must be a descendent of SlotStatesProxy");

return;

}

this.registerProxy(slotDataProxy);

this.registerProxy(slotStatesProxy);

this.registerMediator(new mainViewMediator(mainView));

}

#### 1.proxy

注意，我这里注册了两个proxy，proxy的职责就是封装数据模型，这些数据可以是服务器的，也可以是本地的xml，json，只要你知道协议或接口，你就可以翻译这些数据信息给前台使用。

proxy在pureMVC模式中是个被Model注册的简单的数据持有者。

具体流程可以查看Facade：：registerProxy，model是facade的一个单例成员变量，它在facade中实现对proxy的注册。

#### 2.Mediator

Mediator是视图组件（View Component）与系统其他部分交互的中介器。

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和proxy虽然都继承自Notifier，但proxy不会接受notification，只会发送notification，而mediator因为牵涉到组件之间的交互通信，所以既发送notification，又接收notification。

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。

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。

例如：我们会在ControlBarLayerMediator里面保存一个\_slotDataProxy实例，用来访问proxy的数据和一个\_slotStatesProxy实例，用来改变和访问UI状态。

#### 3.Command

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。

我所理解的Command应该类似与AbstractResponseInterpreter之类的直接解析和翻译proxy的类，然后通过SlotCommunicationController之类的类来dispatch事件出去。

## 四、下一步计划：

我基本在这个框架里面完成了view与view之间通过mediator的通信，这么做的优点就是view与view之间不需要彼此知道相互的API，真正起到脱偶的作用，proxy也已经完成的差不多了，后面我还将进一步研究一下Command这一个模块，并把command的功能实现出来。