VIPER 浅谈

架构腐朽

Massive View Controller

MVVM

VIPER 理论介绍

VIPER项目实践

个人实践体会

参考

架构腐朽

在介绍之前，谈谈项目开发的痛点。

在一个多年迭代开发的项目中，只要项目依然活跃，必然带来需求的不断增长，业务扩张，人员频繁调动...巨大的交付压力，新员工的磨合阶段，文档的缺乏等等，不管项目之初的架构设计是多么合理，也会存在无法满足的情况，架构已然被腐朽。当你发现代码维护越来越困难，新功能的开发痛不欲生，一个问题的解决、一个需求的增加导致不可预测的新问题的出现，代码的耦合，程序的构建时间越来越长时...我们知道需要重构了.

至今为止，我们听过太多的架构名词，经典的MVC、三层架构(UI-BLL_DAL)、四层、五层、MVVM、HMVC、SOA(面向服务)等等，以及今天的主角--VIPER. 新技术的产生往往是针对解决某一个特定域而进行的尝试。那么，VIPER 是解决什么特定域的问题呢？或者说，我们经典的MVC架构，在长期的实践中带来哪些问题?

Massive View Controller

我们都听过MVC架构，也一直在项目中实践MVC。我查了下weiki，以下是wiki上对MVC的介绍:

> ***MVC模式***（*Model-View-Controller*）*是软件工程中的一种软件架构模式，把软件系统*

> *分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。*
>
> ***MVC模式**的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式通过对复杂度的简化，使程序结构更加直观。软件系统通过对自身基本部分分离的同时也赋予了各个基本部分应有的功能*

简而言之，MVC的架构模式，简化了程序结构和复杂度，降低了程序的维护和迭代开发成本。确实，从Trygve Reenskaug在1978年提出，在Smalltalk语言中实现，到现在MVC架构在各种平台上的广泛应用，事实证明MVC确实是一个久经考验、征战沙场的老将，是一个经典的架构设计。

But！！对于MVC，你有没有听过另一种解释，叫 Massive-View-Controller？所谓的重量级视图控制器的概念？在这里不深入细谈MVC问题域，因为相信只要经历过一年以上迭代开发项目的你，在交付压力等上节提到的因素之下，你肯定亲手写过或维护过上千行乃至上万行的code文件(当然我也没有例外👅)？你没有过么？..那你可以不用继续看下去了，要么你的环境太优越了！！要么你的同事跟你一样优秀！！所以，更不用说持续迭代开发2到3年以上的项目，伴着需求不增长(毫无节制的增加需求，只会导致产品的奔溃！)，程序猿无法吐槽的交互设计，随着时间的推移，只会导致项目变得无法控制。等到了那时，为时晚矣！

回过神来，我们看看所谓的Massive－View－Controller，看看你项目中那些重量级的视图控制器，毫无疑问已经违反了单一责任原则。其实不难理解，我们可以非常容易将所有的业务逻辑写入到视图控制器中，而不是将业务逻辑写入到Model中。针对这一问题域，给视图控制器瘦身的策略很多，但不局限于此，随之而来的是The Clean Architecture的提出，MVVM、VIPER等新架构概念的词汇出现了。那么，MVVM是什么？VIPER又是什么？

MVVM

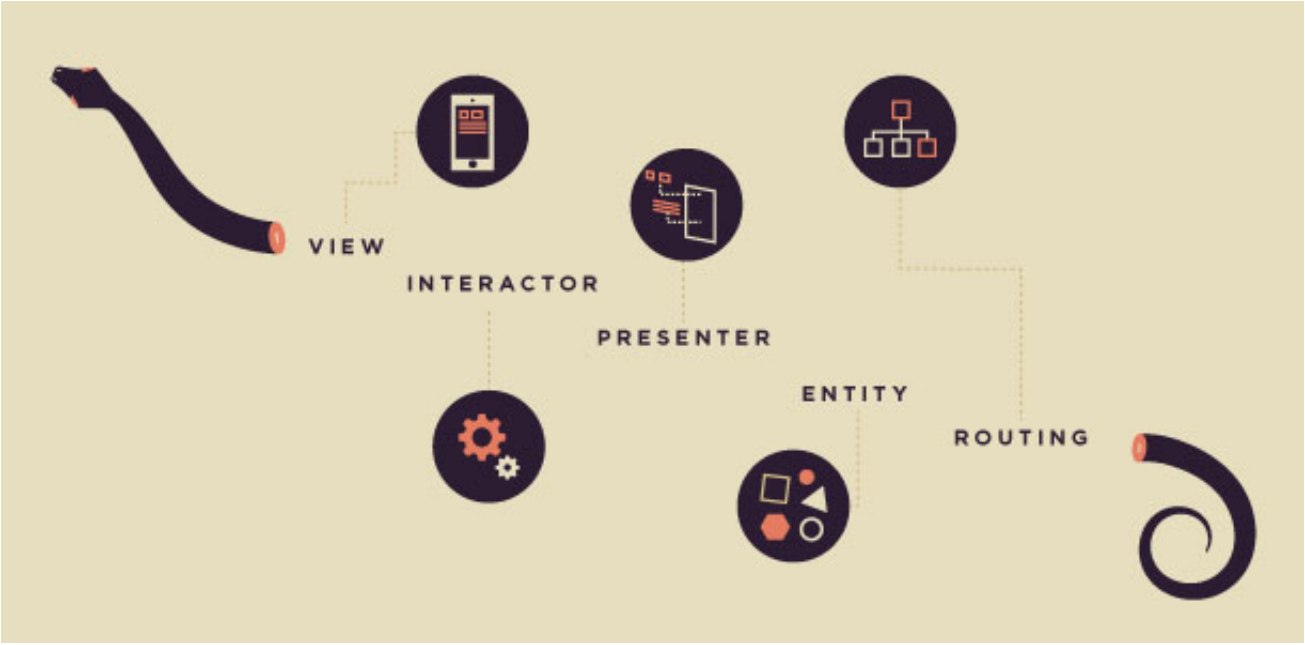
此篇不详细介绍MVVM(Model-View-ViewModel)。该架构的理念，其实是在MVC的基础上，将部分展示数据逻辑抽离到ViewModel类中。一定程度上降低了视图控制器的压力。其实在个人实践中，ViewModel类非常适合展示数据的页面中实现，数据的解析、处理、格式化等都非常适合在ViewModel中进行处理。但是在针对比较大型复杂的应用，MVVM起到的效果依然有限。对MVVM感兴趣同学，可以看看Gitbucket源代码，作者将MVVM和RAC实践了一年之后写的开源客户端，其中ViewModel的应用还是有很多学习的地方。

VIPER 理论介绍

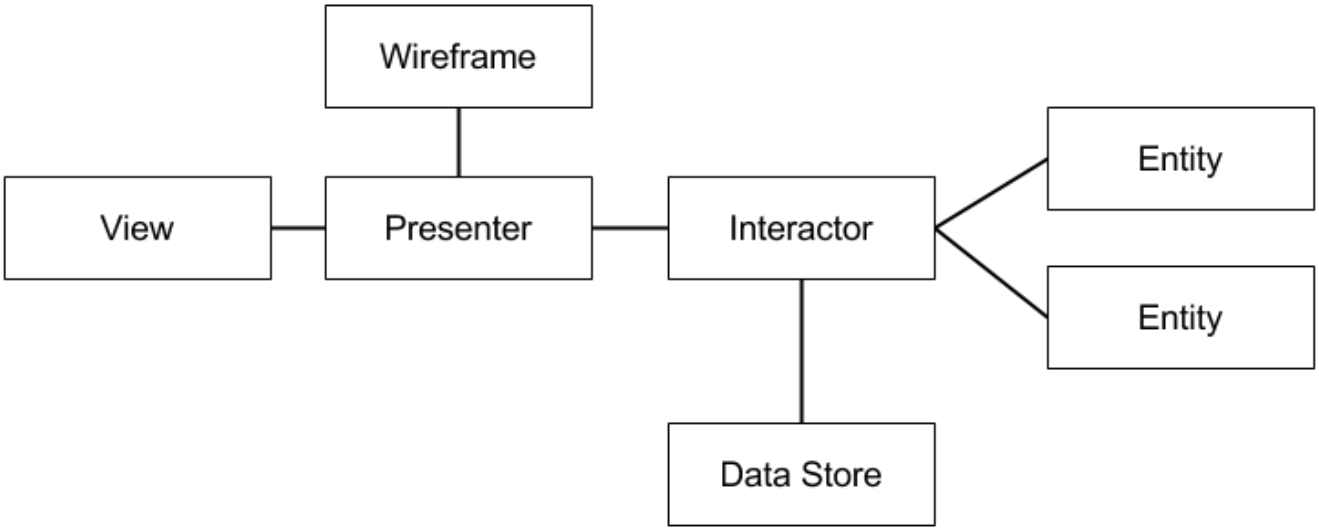
废话这么多，终于进入正题了。下面介绍一些必要的理论知识。

大家不要被VIPER这个单词吓到，它其实是View－Interactor－Presenter－Entity－

Router五个单词的简写。非常明显，按照MVC的理解，它其实将程序架构分成了5个模块。在我看来，它不是取代MVC，而是在MVC的基础上进一步划分了程序结构。VIPER的目的非常清晰，固定五个模块的设计，每个模块都遵守单一责任原则，致力于创建一个简明的应用。**本文的图片来自objc.io**



事实上，VIPER的架构，是一种基于用例的架构设计，也就是我们经常听到的TDD，测试驱动开发。虽然在绝大多数的国内开发团队，因为交付压力等原因，很少写测试用例。但不妨碍我们去尝试VIPER架构，因为就像我上面讲的一样，每一个模块都遵守单一责任原则,职责分明。5个模块的关系，如下图所示：



1. 视图(View):展示在屏幕上的视图组建，是接收用户的点击和输入第一入口
2. 展示器(Presenter):传递视图传递的数据给交互器处理，传递交互器处理的结果给视图展示，传递导航调整给路由
3. 交互器(Interactor):处理主要的业务逻辑
4. 实体(Entity):交互器使用到的基本数据模型对象
5. 路由(Router):集中处理导航视图逻辑，图中称为WireFrame线框

视图

在我看来，视图就是纯粹的视图组件，可以是直接使用UIkit中的系统组件，也可以是你自定义的视图组件。它的职责应该是五个模块中，最清晰和容易理解的吧。

展示器

展示器在整个VIPER中是非常重要的一环，在图上的依赖关系中我们可以看到，它同时起到一个中间件的作用。展示器的职责包含驱动用户界面的逻辑处理，在于传递数据流，展示相关提示，传递导航控制。可以根据用户的输入和点击事件，告诉交互器去做数据请求、数据库存储等，同时，接收交互器的处理结果，去更新用户界面的内容，进行相关必要的界面提示。此外，作为视图控制器与路由的中间件，任何的导航控制跳转，都可以通过展示器告诉路由去处理。在实际的开发中，需要注意的是，实体与展示器无任何交集。任何实体的处理逻辑都不能在展示器中出现。

简而言之，展示器应该只做驱动用户界面相关的逻辑和数据传递。

交互器

交互器在应用中代表着一个独立的用例。它具有业务逻辑以操纵模型对象（实体）执行特定的任务。交互器模块代码，与任何用户界面是剥离的，可以在任何模块、平台（iOS/OS）中进行复用。通常，我们可以在交互器中进行接口请求、数据库的增删改查、业务的复杂逻辑处理等。这个时候，我们就可以将原来的视图控制器中的业务逻辑转移到交互器模块中了。

实体

简单的数据模型对象。模型对象应该只有属性方法，以及一个或多个初始化方法。在初始化方法中，不应该做任何的数据加工处理，而是直接的赋值。只有这样，才是一个纯粹的model。

路由

在实际开发中，用线框wireframe来描述路由。顾名思义，所有的视图控制器，通过线框来串联和控制导航逻辑。在线框中，可以集中处理导航的跳转、过渡动画等相关的操作。

事实上，以上5个模块，处理视图，其他都是PONSO(Plain Old NSObject)普通的NSObject对象罢了。当你将一个视图控制器、一个复杂的模块划分成这5个模块的时候，只要你划分的足够明确，你的视图控制器肯定会相当轻量。你的视图控制器要做的也是一件事情，那就是控制视图(创建、自动布局)。通常，如果使用代码，你可能在视图控制器中需要做的只是创建视图、添加视图、视图布局，VIPER+ViewController的模块构建，保证了比MVC更粒度的划分。

最后，只要大家对这些模块的职责有了一个认识就可以了！接下来，我们还是通过code来更深入的了解下VIPER吧！

VIPER项目实践

如果你刚听说VIPER，可以具体看看示例代码Objc－VIPER－Demo。如果你只是想大致了解下，也没有关系，为了方便大家对该架构的理解，我将在萤石云的登录模块基础上，使用VIPER的思想来分析。

首先，我们来看下萤石云和萤石运动的登录界面，如下：

非常清晰，基本上APP端的登录界面都长得差不多，我们可以确定，基本上登录模块的功能，不管业务多复杂，其实核心功能无非以下几点：

因此，针对该模块的功能，我们可以设计一个登录模块协议,来表达登录模块的功能。目前只针对登录页面的进行设计，大概是这个样子的：

```
@protocol YSLoginModuleInterface <NSObject>

/**
 *  登录操作
 *
 *  @param userInfo 账号信息
 */
- (void)loginInWithInfo:(id)userInfo;

//  忘记密码操作
- (void)forgetPassword;

//  注册账户操作
- (void)registAccount;

/**
 *  第三方登录操作
 *
 *  @param loginType 第三方登录类型
 */
- (void)thirdLoginWithLoginType:(YSThirdLoginType)loginType;

/**
 *  根据用户名获取头像
 *
 *  @param accountStr 用户名
 */
- (void)fetchAvatarImageWithAccount:(NSString *)accountStr;

@optional
// 获取上次的缓存信息 用于显示
- (void)getLastCacheingInfo:(void (^)(NSString *accountStr,
NSString *password))userInfoBlock;
```

以上四个接口，明确了我们登录模块的核心功能。在确定好模块的功能之后，接下来就让我们按V－I－P—E－R的顺序，来进行依次剖析。

VIEW

视图，一个登录界面，包括但不局限于视图组件：取消按钮、登录按钮、注册按钮、忘记密码按钮、第三方按钮、头像视图、账号输入框、密码输入框等。视图组件非常清晰简单。

视图控制器，则创建这些相关的视图组件，进行自动布局，添加手势等操作。

此处的VIEW，可以理解为视图控制器。

INTERACTOR

交互器，按照TDD的思路，交互器的设计应该是通过测试用例来明确和设计对应的接口。交互起作为业务处理中心，我们需要将核心的业务逻辑搬到交互器中。在设计之前，我们来看看登录模块会包含哪些业务逻辑？我们根据登录模块的入口，来进行分析，首先我们来看看注册和忘记密码操作保护流程：

在上图中，我们可以分成清晰得看到整个登录模块的操作逻辑。为了方便大家理解，需要提到的是：

1. 注册和忘记密码流程不做深入，针对登录页面，只考虑导航跳转到对应的视图控制器流程。
2. 异常处理逻辑复杂，为了方便大家理解，暂不深入异常错误的具体处理流程。

因此，我们可以很明确，目前登录交互器需要做的事情很简单，就是：

1. 账号密码的登录处理
2. 第三方登录处理(包含微信、淘宝、京东等)
3. 获取用户头像

现在，我们来看看交互器的设计，大概长这个样子的：

```
@interface YSLoginInteractor : NSObject

/**
 *  登陆处理
 *
 *  @param loginInfo    登陆信息
 *  @param SuccessBlock 成功
 *  @param FaieldBlock  失败
 */
- (void)loginInWithInfo:(id)userInfo
           successBlock:(SuccessBlock) successBlock
            failedBlock:(FailureBlock) faieldBlock;

/**
 *  第三方登录操作
 *
 *  @param loginType 第三方登录类型
 *  @param viewController 登录所在的视图控制器
 *  @param SuccessBlock 成功
 *  @param FaieldBlock  失败
 */
- (void)thirdLoginWithLoginType:(YSThirdLoginType)loginType
                 viewController:(UIViewController *) viewController
                   successBlock:(SuccessBlock) successBlock
                    failedBlock:(FailureBlock) faieldBlock;

/**
 *  获取头像
 *
 *  @param accountStr   用户名
 *  @param successBlock 成功回调
 *  @param faieldBlock  失败回调
 */
- (void)fetchAvatarImageWithAccount:(NSString *)accountStr
                       successBlock:(SuccessBlock) successBlock
                        failedBlock:(FailureBlock) faieldBlock;

// 获取缓存账户信息
- (void)getLastCacheingInfo:(void (^)(NSString *accountStr,
NSString *password))userInfoBlock;
```

它们的实现大致可以如下：

登录

```
- (void)loginInWithInfo:(id)userInfo
           successBlock:(SuccessBlock) successBlock
            failedBlock:(FailureBlock) faieldBlock
{

    [EzvizUser login:userInfo success:^(id request, id
responseObject) {
         这里就是需要的逻辑处理
        // 包括更新本地数据缓存
        // 需要进行的关联请求操作
        // .. 处理完跟登录成功绑定的相关业务逻辑后 回调展示器，进行相关展示操
作
        successBlock(nil);

    } failure:^(NSError *error, NSString *message) {
        // 登录异常 我们需要根据错误码进行处理
        // 萤石云APP需要根据各种错误码进行导航跳转进入不同处理流程，此外无多
余的业务逻辑
        faieldBlock(error);
    }];
}
```

第三方登录

```
- (void)thirdLoginWithLoginType:(YSThirdLoginType)loginType
                 viewController:(UIViewController *) viewController
                   successBlock:(SuccessBlock) successBlock
                    failedBlock:(FailureBlock) faieldBlock
{
    // 有一个单独的第三方处理中心 获取对应的token userid等相关信息
    [YSThirdLogin logininWithLoginType:loginType success:^(id
userInfo) {
        // 根据返回的数据，调用账号密码登录接口
        [self loginInWithInfo:userInfo
                 successBlock:successBlock
                  faieldBlock:faieldBlock];

    } failure:^(NSError *error) {
        faieldBlock(error);
    }];
}
```

获取头像

```
- (void)fetchAvatarImageWithAccount:(NSString *)accountStr
                       successBlock:(SuccessBlock) successBlock
                        failedBlock:(FailureBlock) faieldBlock
{
    // 根据accontStr  获取数据库或本地文件、缓存对应的头像
    // 或者本地无缓存则数据请求等
    // 根据处理  回调
}
```

获取上次账户缓存信息

从数据库或缓存中获取对应存储信息。

从上面可以看出:具体的接口请求设计和实现不应该在交互器中实现，包括登录接口、第三方请求接口、数据缓存等都需要在底层进行封装，供交互器直接调用。交互器的主要职责，还是业务逻辑相关的内容，包括登录接口的成功逻辑处理、失败逻辑处理、头像的获取处理、缓存处理等业务逻辑。绝大多数的业务逻辑，都可以在交互器中实现。但如果涉及到页面驱动的业务逻辑，则就需要在展示器中处理。

PRESENTER

展示器，展示器的作用其实非常重要。我们从关系图中可以看到，它是视图、路由和交互器之间的纽带。那么如何设计展示器呢?

其实很简单，首先我们来看看上面分析设计的YSLoginModuleInterface协议。可以看出，该协议包含了主要的用户操作入口。而展示器的职责恰恰包含了页面驱动逻辑处理，所以展示器必须实现该协议。此外，由于展示器是纽带，它关联视图、路由以及交互器，因此它的设计是这个样子的:

```
@interface YSLoginPresenter : NSObject<YSLoginModuleInterface>

@property (nonatomic,strong)YSLoginInteractor *loginInteractor;

@property (nonatomic,strong)YSLoginViewController *userInterface;

@property (nonatomic,strong)YSLoginWireFrame *loginWireFrame;
```

在展示器中，分别与交互器 、视图控制器、线框三者强关联在一起。我们来看看，它实现协议的接口:

```
#pragma mark -protocol
/**
 *  登录操作
 *
 *  @param userInfo 账号信息
 */
- (void)loginInWithInfo:(id)userInfo
{
  //检查userInfo格式
  if (![YSCheckFormart checkUserInfo:userInfo key:kUserInfoKey])
  {
    //格式错误，则进行页面提示 所有的提示基本上都可以在展示器中进行
    [[YSProgressHUD shareInstance]showAutoDisapperWithTitle:@"输入格式错误"];
    return;
  }

  //正确，则调用交互器

  [self.loginInteractor loginInWithInfo:userInfo successBlock:^(id responseObject)
  {
    // 登录成功 交给线框进行调转
    [self showViewControllerWithViewControllerKey:kYSTabbarViewControllerKey
      params:nil];
  }
  failedBlock:^ (NSError *error)
  {
    // 1.登录失败 进行错误提示
    [[YSProgressHUD shareInstance]showAutoDisapperWithTitle:[error domain]];
    // 2.根据错误码，进行异常流程处理 因为异常处理流程主要涉及到页面的跳转和错误的提示 因此将该驱动逻辑放在展示器中是最合适的地方
    [self routerViewWithLoginErrorCode:[error code]];
  }];
}

//  忘记密码操作
- (void)forgetPassword
{
  //单纯的页面跳转，直接交给线框进行路由分发
    [self showViewControllerWithViewControllerKey:kYSRecoverViewControllerKey
```

```
      params:nil];
}

//  注册账户操作
- (void)registAccount
{
    [self
showViewControllerWithViewControllerKey:kYSRegistViewControllerKey
      params:nil];
}

/**
 *  第三方登录操作
 *
 *  @param loginType 第三方登录类型
 */
- (void)thirdLoginWithLoginType:(YSThirdLoginType)loginType
{
   // 与登录几乎一致
  [self.loginInteractor  thirdLoginWithLoginType:loginType
                                  viewController:self.userInterface
                                    successBlock:^(id
responseObject){
    // 登录成功 交给线框进行调转
    [self
showViewControllerWithViewControllerKey:kYSTabbarViewControllerKey
      params:nil];
  }
  failedBlock:^ (NSError *error)
  {
    // 1.登录失败 进行错误提示
    [[YSProgressHUD shareInstance]showAutoDisapperWithTitle:[error
domain]];
    // 2.根据错误码，进行异常流程处理 因为异常处理流程主要涉及到页面的跳转和错误的提示 因此将该驱动逻辑放在展示器中是最合适的地方
    [self routerViewWithLoginErrorCode:[error code]];
  }];
}

//...略

#pragma mark -private
- (void)showViewControllerWithViewControllerKey:(NSString *)key
params:(NSDictionary *)params
{
    // 数据对象
```

```
    YSMessageInfo *messageInfo = [YSMessageInfo
initwihMessageName:key params:params];
    // 页面跳转
    [self.loginWireFrame routingViewController:self.userInterface

                                  messageOfPage:messageInfo];
}
- (void)routerViewWithLoginErrorCode:(int)code
{
  // 根据错误码进行路由分法
  // 略
}
```

从上面可以看到，展示器的任务就是实现YSLoginModuleInterface协议，需要路由页面时则交给线框去处理，需要具体业务接口处理时，则交给交互器去处理，需要页面提示或逻辑判断进行页面导航时，则在本类中进行。此外，我们看到线框在此处的应用是根据给定一个key值以及一个messageInfo的参数进行调转。

Entity

实体，model对象。model对象比较常见，在MVC中我们就经常运用。在我们的设计中，有很多model对象。比如下面这个info对象：

```
@interface YSUserInfo : NSObject

@property (nonatomic, readonly , copy) NSString *accountStr;/**<账
号*/
@property (nonatomic, readonly , copy) NSString *passwordStr;/**<密
码*/

- (instancetype)initWithAccount:(NSString *)accountStr
                       password:(NSString *)passwordStr;
```

WireFrame

线框，导航的处理中心。在线框中，我们可以集中处理页面跳转的方式、自定义跳转动画的设置等。

```
@protocol YSWireFrameProtocol <NSObject>

/**
 *  视图控制器路由
 *
 *  @param ViewController 视图控制器
 *  @param messager       消息
 */
- (void)routingViewController:(UIViewController*)ViewController
                messageOfPage:(YSMessageInfo*)messager;

@end

@interface YSWireFrame : NSObject <YSWireFrameProtocol>
```

```
#pragma mark - 视图控制器路由控制
- (void)routingViewController:(UIViewController*)ViewController
                messageOfPage:(YSMessageInfo *)messager
{
    NSParameterAssert(messager != nil);
    NSParameterAssert(ViewController != nil);

    SEL selector = [self
selectorOfClassName:messager.messageNameStr];
    IMP imp      = [self  methodForSelector:selector];
    void (*func)(id,SEL,UIViewController*,NSDictionary*) =
(void*)imp;
    func(self ,selector,ViewController,messager.paramsDic);
}

- (SEL)selectorOfClassName:(NSString *)messageName
{
    static NSDictionary *selectordictionary = nil;
    if(!selectordictionary)
    {
        selectordictionary = @{

     kShowRegistUserOneViewController:
                                    [NSValue valueWithPointer:
@selector(showRegistUserOneViewController:params:)],

            kShowTabBarViewController:
                                     [NSValue valueWithPointer:
```

```
@selector(showTabbarViewController:params:)]

        };
    }

    NSValue *value = [selectordictionary valueForKey:messageName];

    return value ? [value
pointerValue]:@selector(emptyMothedViewController:params:);
}

- (void)emptyMothedViewController:(UIViewController
*)viewController params:(NSDictionary *)params
{
    DDLogCError(@"不存在对应的选择器方法,请注意实现！");
}

- (void)showRegistUserOneViewController:(UIViewController
*)viewController params:(NSDictionary *)params
{
    //具体跳转处理
}

- (void)showTabbarViewController:(UIViewController *)viewController
params:(NSDictionary *)params
{
    //具体跳转处理
}
...
```

在线框中，实现了跳转YSWireFrameProtocol协议。在具体实现中，创建了一个静态的字典来存储key值和对应的选择器方法。如果找不到方法，则调用默认的emptyMothedViewController方法。在具体的跳转中，可以定制化跳转风格和动画。

综上，VIPER的5个模块已经介绍完，那视图控制器如何接入呢？其实很简单：

```
@interface YSLoginViewController : UIViewController
/**
 *  实现登陆模块功能的对象，也是展示器，负责与视图控制器之间的交互。
 */
@property (nonatomic,strong)id <YSLoginModuleInterface>
eventHander;
```

通过创建工厂类方法：

```
+ (UIViewController *)YSLoginViewController
{
    HIKLoginViewController *userInterface =
[[HIKLoginViewController alloc]init];
    YSLoginInteractor   *interactor       = [[YSLoginInteractor
alloc]init];
    YSLoginPresenter    *presenter        = [[YSLoginPresenter
alloc]init];
    YSLoginWireFrame    *wireFrame        = [[YSLoginWireFrame
alloc]init];
    //视图控制器引用展示器
    userInterface.eventHander             = presenter;
    //展示器分别引用视图控制器 交互器 线框
    presenter.userInterface               = userInterface;
    presenter.loginInteractor             = interactor;
    presenter.loginWireFrame              = wireFrame;
    // 返回视图控制器
    return userInterface;
}
```

此时，我们再来看看登录模块之间的关系图：

是不是比较清晰了？我们首先分析了模块的核心功能，创建了YSLoginModuleInterface协议。依次将涉及的内容分为5个模块，每个模块只做一件事情，最后通工厂类方法，按照职责创建之间的依赖关系。只要你对VIPER有这样一个概念，那本篇的目的也就达到了。任何架构的学习和应用，也必须经过长久的实践和探索。

个人实践体会

最后，我讲一些个人实践的体会，希望对大家有所帮助。

1.首先需要对VIPER每个模块的职责非常清晰，以及清晰的区分每块代码的职责。当我在写本篇文章的时候，仔细去回顾我重构的模块时，发现我的有些模块不只干了一件事情。

2.VIPER架构目前没有成熟的应用方案，国外也有很多成功的应用案例。我刚讲述的实现方式也是个人的一种实践探索，给大家提供一个思路。不同的项目，拥有不同的问题域和关卡。在实际的应用中，只要牢记每个模块的职责，结合自己的项目进行运用才是上策。

3.VIPER架构实践，推荐在复杂的逻辑处理页面或复杂的模块中进行应用。本身简单的操作视图控制器，如果还按VIPER的架构划分，反而显得复杂了。比如整个登录模块、图片管理模块等。上面提到的只是登录模块中的登录页面，其实可以将注册、忘记密码流程整合到一起，用VIPER实现。

4.其实不管你使用VIPER，还是任何其他的框架，只要你细化每个模块，每个模块遵守单一

责任原则，保持轻量的代码文件即可。我们的目的都很简单，保证代码结构的清晰，代码块的低耦合，保证项目在迭代开发中，尽可能的降低维护和增量开发成本！

5.最后，希望本篇文章对大家有所启迪。不管你认不认同VIPER架构，任何新技术的产生本身就需要质疑和不断的改进，只要你认为对你有所帮助，那么就大胆地去实践吧！

参考

http://www.objc.io/issues/13-architecture/viper/

http://mutualmobile.github.io/blog/2013/12/04/viper-introduction/

http://mutualmobile.github.io/blog/2013/12/04/viper-introduction/

http://www.slideshare.net/kprofic/from-mvc-to-viper

http://www.slideshare.net/RajatDatta1/i-os-viper-presentation

http://code.oursky.com/viper-ios-architecture-beyond-mega-viewcontroller/

http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

http://www.mutualmobile.com/posts/meet-viper-fast-agile-non-lethal-ios-architecture-framework

http://blog.8thlight.com/uncle-bob/2011/11/22/Clean-Architecture.html