**谈谈MVC和MVVM**

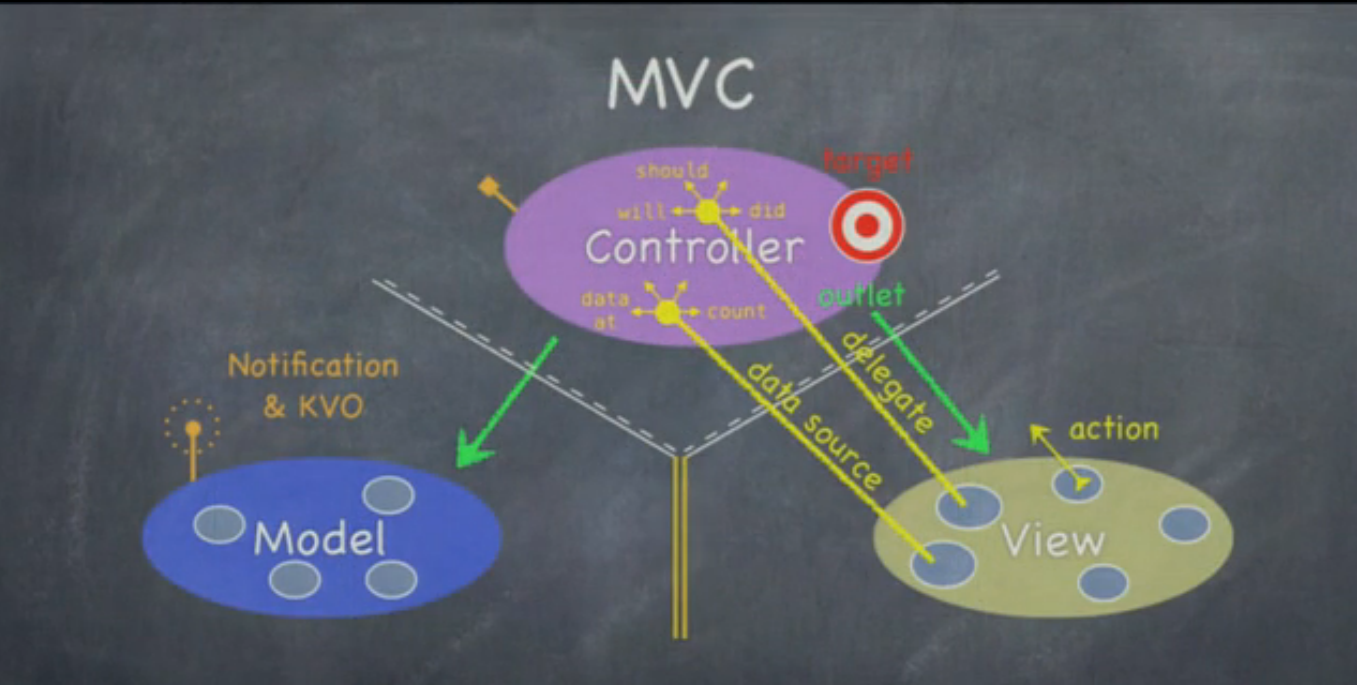
参考：<http://www.jianshu.com/p/f4faa720f00d>

工作到现在，对代码稍微有点认识，if else 会写那么点。今天就和大家谈谈所理解的MVC，以及现在比较流行的MVVM。首先我们应该明白，计算机实现一个功能核心代码就那么点。也许我们经常会听到对于同一个问题，菜鸟的实现的真的就是几行，可是大牛却多出了很多文件。这个是为什么？因为大牛的“经验”比较多。。。这里面最主要的目的就是为了维护和可扩展。在设计模式里面，如果你能遵循单一原则，你的代码就已经很好了。

**MVC**

做iOS开发，一直被教导一定要按MVC模式开发。可是MVC到底是神马？至少在我工作初也不懂是咋回事！如果网上一搜就是千篇一律的M是数据模型,V是视图，C是控制器。然后巴拉巴拉的讲他们之间是怎么通信的。然并无卵用！iOS的MVC展现形式还是有点特殊的，特别是controller和view紧密在一起，controller还必须负责view的展示。在服务器端，view根据model直接生成HTML，然后直接扔给浏览器去渲染和展示，通过Ajax或者js post告诉服务器controller view的响应事件，controller真的做的只是业务数据的处理，出来的结果其实还是数据，根本没有去做UI相关的事情。也可以说服务器根本就没有view。。但它有个浏览器，帮他管理着view和controller之间的交互。而在我们iOS中，controller可不仅仅只是处理数据了，还负责view的管理以及事件的传递。 MVC本质就是将数据展示和数据进行隔离，提高代码的复用性和扩展性。好吧，我也说点并无什么卵用的。

看看斯坦福老爷爷的一张图：



这就是我们所认识的MVC。我们可以看到，Controller可以和Model通信，也可以和View进行通信。继续看Controller和Model的关系，绿色的箭头代表Controller可以直接进行对Model进行访问，也就是说Model对于Controller来说就是透明的。但是Model并不知道Controller是谁。如果Model发生了变化，那么就通过Notification和KVO的方式传递给Controller。同样的Controller和View之间也是这种关系，View对Controller来说就是透明的。Controller可以直接根据Model决定View的展示。View如果接受响应事件则通过delegate，target-action，block等方式告诉Controller的状态变化。Controller进行业务的处理，然后再控制View的展示。

到这里你会发现Model和View并不能直接的进行通信，都必须通过Controller。那这样Model和View就是相互独立的。View只负责页面的展示，Model只是数据的存储，那么也就达到了解耦和重用的目的。

说这么多不如几行代码来的实在。我们以一个简单的例子来看下：

我们假设苹果根据买iPhone的人给予不同的优惠，学生优惠20%，it民工优惠50%，其他不优惠。

//客户类

typedef NS\_ENUM(NSInteger, CustomerType) {

CustomerTypeStudent,

CustomerTypeiT,

CustomerTypeOther,

};

@interface Customer : NSObject

@property (nonatomic, assign) CustomerType customerType;

@end

//iPhone类

@interface iPhone : NSObject

@property (nonatomic, strong) NSString \*name;

@property (nonatomic, strong) NSString \*price;

@end

//VC类

@interface ViewController ()

@property (nonatomic, strong) iPhone \*iphone;

@property (nonatomic, strong) Customer \*customer;

@property (weak, nonatomic) IBOutlet UILabel \*lblName;

@property (weak, nonatomic) IBOutlet UILabel \*lblPrice;

@property (weak, nonatomic) IBOutlet UILabel \*lblDiscount;

@end

@implementation ViewController

- (void)viewDidLoad {

[super viewDidLoad];

self.title = @"手机优惠";

self.lblName.text = self.iphone.name;

self.lblPrice.text = self.iphone.price;

if (self.customer.customerType == CustomerTypeStudent) {

self.lblDiscount.text = @"优惠20%";

}

else if (self.customer.customerType == CustomerTypeiT) {

self.lblDiscount.text = @"优惠50%";

}

else {

self.lblDiscount.text = @"没有优惠";

}

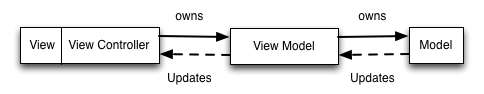
}

@end

这个就是我们最正常的开发，我们的逻辑都是在vc里面写的。这样写有木有错呢？一点没错，controller本来就是用来处理业务的。由于这个例子比较简单，只是做了个优惠判断，所以我们看不出有啥坏处。有点开发经验的都知道，如果业务复杂起来，再加上其他乱七八糟的验证，controller就会变得很大，越来越难以维护。这个也是MVC比较明显的缺点。

**MVVM**

既然controller越来越臃肿，越来越难以维护，我们怎么去优化和瘦身呢？回头再仔细看看我们所谓的业务逻辑，是干什么的？无非就是根据几个数据得出一个数据用来控制view的显示。比如展示的是什么文案，按钮能不能响应，页面能不能跳转等等。那MVVM就干了这件事，帮忙分担一下controller里面的部分业务逻辑。MVVM更合理的应该叫做MV-CM。



这个时候，controller将不再直接和真实的model进行绑定了，而通过ViewModel,viewModel进行持有真实的Model。

我们来看看刚刚那例子怎么修改：

//新建一个viewModel

//.h文件

@interface viewModel : NSObject

@property (nonatomic, strong) NSString \*name;

@property (nonatomic, strong) NSString \*price;

@property (nonatomic, strong) NSString \*discount;

- (id)initWithCustomer:(Customer \*)customer iphone:(iPhone \*)iphone;

@end

//.m文件

@interface viewModel ()

@property (nonatomic, strong) iPhone \*iphone;

@property (nonatomic, strong) Customer \*customer;

@end

@implementation viewModel

- (id)initWithCustomer:(Customer \*)customer iphone:(iPhone \*)iphone

{

if (self = [super init]) {

\_customer = customer;

\_iphone = iphone;

[self bindData];

}

return self;

}

- (void)bindData

{

self.name = \_iphone.name;

self.price = \_iphone.price;

if (self.customer.customerType == CustomerTypeStudent) {

self.discount = @"优惠20%";

}

else if (self.customer.customerType == CustomerTypeiT) {

self.discount = @"优惠50%";

}

else {

self.discount = @"没有优惠";

}

}

@end

//VC

@interface ViewController ()

@property (nonatomic, strong) viewModel \*viewModel;

@property (weak, nonatomic) IBOutlet UILabel \*lblName;

@property (weak, nonatomic) IBOutlet UILabel \*lblPrice;

@property (weak, nonatomic) IBOutlet UILabel \*lblDiscount;

@end

@implementation ViewController

- (void)viewDidLoad {

[super viewDidLoad];

// Do any additional setup after loading the view, typically from a nib.

self.title = @"手机优惠";

self.lblName.text = self.viewModel.name;

self.lblPrice.text = self.viewModel.price;

self.lblDiscount.text = self.viewModel.discount;

}

- (void)didReceiveMemoryWarning {

[super didReceiveMemoryWarning];

// Dispose of any resources that can be recreated.

}

@end

看到修改完的代码，你会发现VC里面已经省去了不少的代码。一切都和viewModel进行交流。这里我只是展示一个最简单的数据展示，如果有其他响应事件，是需要viewModel开放方法来进行处理的，并要通知VC处理结果的。

**关于MVVM的优点：**

* 方便测试

在MVC下，Controller基本是无法测试的，里面混杂了个各种逻辑，而且分散在不同的地方。有了MVVM我们就可以测试里面的viewModel，来验证我们的处理结果对不对（Xcode7的测试已经越来越完善了）。

* 便于代码的移植

比如iOS里面有iPhone版本和iPad版本，除了交互展示不一样外，业务逻辑的model是一致的。这样，我们就可以以很小的代价去开发另一个app。（以前做公司iPad的时候就深深感觉到，全部在VC里面是多么的痛苦和重新开发一个没有啥区别）。

* 兼容MVC

MVVM是MVC的一个升级版，目前的MVC也可以很快的转换到MVVM这个模式。VC可以省去一大部分展示逻辑。

**缺点：**

* 类会增多

每个VC都附带一个viewModel，类的数量\*2

* viewModel会越来越庞大

我们把逻辑给了viewModel，那势必Model也会变得很复杂，里面的属性和方法越来越多。可能重写的方法比较多，因为涉及到一些数据的转换以及和controller之间的通信。

* 调用复杂度增加

由于数据都是从viewModel来，想想突然来了一个新人，一看代码，不知道真实的模型是谁。比如常用tableview的数据源，一般都是一个数组，如果不断的通过viewModel去取，沟通上没有那么直接。况且每封一层，意味着要写很多代码去融合他们的转换。

最后说下ReactiveCocoa这个框架，这个虽然和MVVM经常一起出现，这个框架主要是帮我们实现model和view的绑定机制。后面会有文章来介绍它。