什么是 MVVM 模式?

一个常规软件或者 APP 一般都是服务于某种商业或者非商业述求，我们平常称为“业务需求”。随着业务需求的扩张、一个软件会变得越来越庞大，越来越复杂。所以一般都会有一套完整的架构设计、研发流程以及质量管理体系来保证整个研发过程。关于“架构设计”，这是一个很大的话题，伴随着我们的业务需求，它会涉及到方方面面，我们今天来谈一谈其中的一个基础环节——MVVM模式。

MVC 是最常见的客户端软件架构之一，它历史悠久，简单好用，易于理解，而且目前常见的 iOS 和 Android 开发，SDK 和与其搭配的 IDE 工具都是默认以 MVC 的方式来使用的。但是我个人更喜欢 MVVM 模式，也一直坚持使用 MVVM 模式来工作了很多年。

最常见的客户端架构有三种:

- MVC: Model-View-Controller
- MVP: Model-View-Presenter
- MVVM: Model-View-ViewModel

在 MVC 里面，Model 是数据模型；View 是视图或者说就是我们的软件界面需要去展示的东西；Controller 是用来控制Model的读取、存储，以及如何在View上 展示数据，更新数据的逻辑控制器。

- 对应到 iOS 开发中，View 约等于 Storyboard，Controller 就是 ViewController，Model 的话是需要我们自己去创建的一些实体（Entity）Class。
- 对应到 Android 开发中，View 约等于 Layout 中的 xml ，Controller 就是 Activity，Model 同上。
- 对应到 React-Native 开发中，View 约等于 Component 中的 render 函数部分，Controller 就是整个 Component，Model 同上。

这里为什么要说 View 约等于 Storyboard 或者 Layout 呢？因为 Storyboard 和 Layout 都被设计为一种界面（View）的描述语言，它描述了整个 View 应该长成什么样子，应该具有哪些控件。程序启动时，系统会首先读取 Storyboard 或者 Layout 文件，通过渲染引擎去把这种 View 的描述语言渲染

成一个真正的 View，此时此刻的 View 才是 MVC 中的那个真正的 V。网络上还有一些文章论调，说 Activity 并不能算真正的 Controller 等等，在这里我们就不纠结这些细节了。

VC 到 VM 的转变

我们前面有提到一个常见的 Controller（下称 VC ）中会包含 Model 的读取、存储等功能，除此之外还会有一些控件的事件绑定和响应，甚至还有网络请求等等。

一个 VC 在包含了大量的业务逻辑后，代码就会变得特别的臃肿、不易阅读和修改。于是后来就慢慢延伸出了 MVP 和 MVVM 模式。MVP 这里我们就跳过了，直接讲 MVVM 模式。MVVM 模式顾名思义是由 Model、View 和 ViewModel（下称 VM ）组成，之前的 VC 变成了 VM。怎么来理解 VM 呢?

对于一个 Model ，比如我们要存储和显示一个人的信息，这个人具有姓名、年龄、性别这三个属性。这个Model的伪代码如下:

```
class Person {
    String  name;
    int     age;
    int     gender;
}
```

Model 的数据模型，和我们的业务需求或者说业务实体（Entity）是一一映射关系。而 ViewModel 顾名思义，就是一个 Model of View，它是一个 View 信息的存储结构，ViewModel 和 View 上的信息是一一映射关系。

以一个软件的登陆场景为例子，假设这个登录界面上有如下逻辑:

- 用户名输入框
- 密码输入框
- 登陆按钮，点击登陆按钮按钮置灰，显示 loading 框
- 登陆成功，页面触发跳转
- 登陆失败，loading 框消失，在界面上显示错误信息
- 错误信息可以分为两种情况： 1、密码错误；2、没有网络

那么我们下面来定义这样一个 ViewModel：

```
class LoginViewModel {
    String  userId;
    String  password;
    bool    isLoading;
    bool    isShowErrorMessage;
    String  errorMessage;
}
```

界面初始化

由于 LoginView 和 LoginViewModel 是映射关系，也称为绑定关系，那么 LoginViewModel 是怎样的数据，View 就按照怎样的数据来进行显示。界面第一次打开时，整个 LoginViewModel 的初始值为：

```
{
    userId:  '',
    password: '',
    isLoading: false,
    isShowErrorMessage: false,
    errorMessage: ''
}
```

那么此时界面为：

- 用户名输入框显示为空白字符串
- 密码输入框显示为空白字符串
- loading 框因为 isLoading = false，所以不显示
- 错误信息框 因为 isShowErrorMessage = false，所以不显示
- 错误信息框里面的文字为空白字符串

触发登陆

接下来，用户输入用户名和密码，点击登录按钮，在登陆事件里面触发网络通信逻辑，同时设定 isLoading = true，伪代码如下：

```
function onLoginButtonClick() {
    request(url, ...);
```

```
    loginViewModel.isLoading = true;
}
```

此时 LoginViewModel 的值为：

```
{
    userId:  'this is user id',
    password: 'this is password',
    isLoading: true,
    isShowErrorMessage: false,
    errorMessage: ''
}
```

随着 isLoading 值的变化，因为 ViewModel 和 View 存在绑定关系，那么此时界面动态变化为：

- 用户名输入框显示为刚刚输入的字符串
- 密码输入框显示为刚刚输入的字符串
- 因为isLoading = true，所以显示 loading 框
- 因为isLoading = true，登陆按钮置灰，不可点击
- 错误信息框 因为 isShowErrorMessage = false，所以不显示
- 错误信息框里面的文字为空白字符串

登录失败

接下来我们假设登陆失败，服务器返回密码错误，那么此时在服务器逻辑的相应代码里面我们去设定 isLoading = false，isShowErrorMessage = true，以及对应的errorMessage，伪代码如下：

```
request(url, {
    success: function() {
        ...
    },

    fail: function(err) {
        if(err.code == 1000)  { // 假设1000表示密码错误
            loginViewModel.isLoading = false;
            loginViewModel.isShowErrorMessage = true;
            loginViewModel.errorMessage = '密码错误';
        }
```

```
    }
})
```

此时 LoginViewModel 的值为：

```
{
    userId:  'this is user id',
    password: 'this is password',
    isLoading: false,
    isShowErrorMessage: ture,
    errorMessage: '密码错误'
}
```

接下来依然是触发界面变化，根据绑定关系重新更新显示内容：

- 用户名输入框显示为刚刚输入的字符串
- 密码输入框显示为刚刚输入的字符串
- 因为isLoading = false，隐藏 loading 框
- 因为isLoading = false，登陆按钮置为正常，可以点击
- 因为 isShowErrorMessage = true，显示错误信息框
- 错误信息框里面的文字为“密码错误”

重新登录

用户修改密码后，重新点击登陆按钮：

```
function onLoginButtonClick() {
    loginViewModel.isLoading = true;
    loginViewModel.isShowErrorMessage: false,
    request(url, ...);
}
```

到这里相信大家都会知道了，错误提示框消失，显示 loading 框，登陆按钮置灰。

所以这就是 MVVM 模式的神奇之处，让你不要去关心如何去显示登录框，如何去置灰一个按钮，如何去显示错误提示框又如何去隐藏它等等。当然，这里说的“不关心”并不代表不需要知道这些，完全不处理这些逻辑，只是在

架构上给你一种更清晰，更简单的原则，那就是：

> “当任何外部事件发生时，永远只操作 ViewModel 中的数据”

这里外部事件主要指界面点击、文字输入、网络通信等等事件。因为绑定关系的存在，ViewModel 变成啥样，界面就会自动变成啥样。

单向绑定与双向绑定

随着 ViewModel 的变化，View 会自动变化，那么 View 变化后，ViewModel 会自动变化么？比如用户在“用户名输入框”输入文字后，LoginViewModel 中的 userId 会自动存储输入值么？然后用户又删掉部分输入的内容， userId 再次立即变化么？

这里就需要引入一个新的概念了：**单向绑定与双向绑定**。

- 所谓“单向绑定”就是 ViewModel 变化时，自动更新 View
- 所谓“双向绑定”就是在单向绑定的基础上 View 变化时，自动更新 ViewModel

我们把前面登陆按钮点击后这一过程的代码再来梳理下，单向绑定模式下的伪代码如下：

```
function onUserIdTextViewChanged(textView) {
    loginViewModel.userId = textView.text;
}

function onPasswordTextViewChanged(textView) {
    loginViewModel.password = textView.text;
}

function onLoginButtonClick() {
    loginViewModel.isLoading = true;
    loginViewModel.isShowErrorMessage: false,
    login(loginViewModel.userId,  loginViewModel.password);
}
```

大家可以看到，我们需要非常明确的在 TexView 变化事件里面去重新设定 LoginViewModel 中的值，而双向绑定模式下，根据绑定关系，这一过程就隐藏性的自动完成了。既然“双向绑定”那么智能、简单，为什么还需要“单

向绑定”呢？因为在真实的“业务需求”下，实际情况是非常复杂的，虽然 ViewModel 可以和 View 形成映射关系，但是它们之间的值却不一定能直接划等号。

比如在界面上要填写性别，我们通常会提供一个下拉列表框，让用户选择。这个选择框里面至少有“未知”、“男”和“女”三种字符串值，而我们的 ViewModel 一般情况下并不直接存储这些字符串。因为 ViewModel 中的数据很大一部分情况下是来自于数据库、来自于服务器，而数据库和服务器中几乎是不可能直接把性别字符串存储在数据模型中的。一般会建立一个 int 类型的字段，用 0 表示未知；用 1 表示男人；用 2 表示女人。

那么问题来了，在 ViewModel 中一个 gender 属性类型为 int，值为 0 或者 1 或者 2 时，与其绑定的 View 怎么知道该如何来显示为“未知”、“男”或者“女”呢？

所以“属性转换器”应运而生，在给 View 绑定 ViewModel 时，发现属性值不匹配，那么就需要设定一个属性转换器。反之亦然，当性别选择下拉列表框被用户改变时，用户选择了“男”，在双向绑定模式下，那么 View 依然需要在一个属性转换器的帮助下，把“男”转换为 1，然后设定到 ViewModel 中。

上面只是最简单的一种在绑定不匹配时涉及到属性转换的情况，但是真实的世界往往会更加的错综复杂，双向绑定下的属性转换器随着业务需求的迭代常常会变得越来越庞大，而且因为绑定关系触发 ViewModel 和 View 的动态变化过程是隐藏不可见的，也给调试带来了极大的麻烦。

所以后来大家在长年累月的使用过程中，发现单向绑定可能会是更合适的一种做法。

把数据的请求与处理放在 ViewModel 中

针对前面的登陆代码，我们再来做一次优化，得到一个更加合理的版本：

```
class LoginViewModel {
    String  userId;
    String  password;
    bool    isLoading;
    bool    loginStatus;
    String  errorMessage;
```

```
    Login() {
        request(url, this.userId, this.password, {
            success: function() {
                ...
            },
            failed: function() {
                this.isLoading = false;                 //触发绑定关系，隐藏登陆 lo
                this.isShowErrorMessage = true;         //触发绑定关系，显示错误提示
                this.errorMessage = '密码错误';          //触发绑定关系，设置错误提示
            }
        });
    }
}
```

可以看到，我们把整个登陆过程放在了 LoginViewModel 中，那么登陆按钮点击后这一套响应过程也相应的有所调整：

```
function onUserIdTextViewChanged(textView) {
    loginViewModel.userId = textView.text;
}

function onPasswordTextViewChanged(textView) {
    loginViewModel.password = textView.text;
}

function onLoginButtonClick() {
    loginViewModel.isLoading = true;                //触发绑定关系，显示登陆 loadi
    loginViewModel.isShowErrorMessage: false;       //触发绑定关系，隐藏错误提示框
    loginViewModel.login();                         //开始登陆
}
```

大家看到没有，上面这段代码再也不处理任何的数据逻辑，不关心数据库、不关心网络调用，也完全不关心界面随着数据和逻辑的变化应该如何去设置控件属性状态等等。让我们再来复习一下 MVVM 的核心原则：

> “当任何外部事件发生时，永远只操作 ViewModel 中的数据”

上面这段代码它不属于 Model，不属于 View，也不属于 ViewModel，那它应该写在哪里呢？

- iOS 下依然写在 ViewController 中

- Android 下依然写在 Activity 或者 Fragment 中
- ReactNative 下依然写在 Component 中
- 微信小程序 下依然写在 Page 中

所以 MVC 中的 C，其实一直都默默的存在着，只是变得弱化了，一定要完整的讲的话，那就是 **Model-View-Controler-ViewModel** 模式。只有在理想的双向绑定模式下，Controller 才会完全的消失。

上帝之手

讲到这里，我们已经讲解完了整个 MVVM 模式的核心原理和使用原则，不涉及任何平台、任何语言。或者说只要你遵循以上的原则，写出来的代码都是 MVVM 模式的。但是还有一个最大的疑问，我们一直没有去探讨。那就是：

> ViewModel 是如何与 View 形成的绑定关系，凭什么 ViewModel 中的数据变化了，View 就会动态变化？

我们之所以要把这个问题放在最后讲是因为数据绑定和动态变化这都是由具体的 MVVM 平台或者框架来实现的，跟 MVVM 模式没有直接关系。或者说为了达成 MVVM 模式，不同的平台、不同的框架提供了不同的实现版本来完成这一目标。当然，他们也都大同小异，核心原理差不多。

先来说说第二个疑问：“为什么会动态变化”？其实这里是有一只“上帝之手”在帮你实现这个过程，当“上帝”发现某个数据变化了，然后根据“映射”关系去帮你修改那个控件的属性。但是，我们的登陆界面是否要显示 loading 框，是否要显示错误提示，这些都是我们的业务需求，“上帝”也并不知道你到底想要怎样。所以，所谓的绑定或则映射关系是需要开发者明确来告诉“上帝”的。怎么告诉“上帝”呢？这就需要你或通过配置文件或通过代码语句来设定。

我们来总结一下这两个问题的答案：

- 开发者在代码或者配置文件中设定 ViewModel 和 View 的映射关系
- “上帝之手”在整个软件的运行过程中监控 ViewModel，自动变化这一切

设定映射关系

目前在几大手机 APP 开发平台下，Android 的 MVVM 框架是做得最完善、最智能、最方便的。让我们来看一看 Android 是怎么实现 MVVM 的。

假设我们要实现一个“时钟”界面，这个界面可以显示“当前时间”，如果用户开启了闹钟功能，且同时显示“闹钟时间”。这个界面的简化描述文件（Layout 文件）如下：

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <import type="android.view.View" />
        <variable
            name="mainVM"
            type="com.company.app.viewModel.MainViewModel" />
    </data>

    <RelativeLayout>
        <TextView
            android:id="@+id/timeTextView"
            android:text="@{mainVM.timeText}" />

        <TextView
            android:id="@+id/alarmTimeTextView"
            android:visibility='@{mainVM.alarmEnable ? View.VISIBLE : View.
            android:text='@{mainVM.alarmTimeText}' >
    </RelativeLayout>
</layout>
```

它具有以下几部分内容：

- 该 Layout 具有哪些控件（布局属性这里就隐去了），可以看到这里有两个 TextView，也就是文本展示控件
- 该 Layout 和哪个 ViewModel 具有映射关系
- 该 Layout 中的哪些控件的哪些属性和 ViewModel 中的哪些属性是映射关系

```
<variable
    name="mainVM"
    type="com.company.app.viewModel.MainViewModel" />
```

这几行指定了 Layout 和 MainViewModel 类具有映射关系，且 MainViewModel 在整个绑定关系中的对象名字叫做 mainVM。

```
<TextView
    android:id="@+id/timeTextView"
    android:text="@{mainVM.timeText}" />
```

这几行指定了控件 timeTextView 的 text 属性 和 mainVM 中的 timeText 属性具有映射关系

```
<TextView
    android:id="@+id/alarmTimeTextView"
    android:visibility='@{mainVM.alarmEnable ? View.VISIBLE : View.GONE }'
    android:text='@{mainVM.alarmTimeText}' >
```

这几行中，和上面 timeTextView 控件的绑定关系类似，稍微不一样的是第三行。这一行描述了 alarmTimeTextView 控件的 visibility 属性和 mainVM 中的 alarmEnable 属性绑定在一起。

visibility 属性用来控制 alarmTimeTextView 控件是显示还是隐藏。即 alarmEnable 为 true 则显示，为 false 则隐藏。但是有一个小问题就是在 Android 平台上，一个控件的 visibility 属性并不是 bool 值，而是一个enum 值，分别是 VISIBLE、INVISIBLE 和 GONE。在映射时，这里就需要一个属性转换器了。

Android 的属性转换器比较直接，就是让你直接在绑定语句里面写上一些简单的逻辑代码。如果你的属性转换器过于复杂，还允许把这些逻辑写在一个正常的代码文件中，然后通过 Layout 顶部的 Import 语句引入。

```
@{mainVM.alarmEnable ? View.VISIBLE : View.GONE }
```

这行代码大家看字面意思肯定就能理解：如果 alarmEnable 为 true，则 visibility 的值 为 VISIBLE， 否则为 GONE。

生成上帝之手

Android 平台的 MVVM 映射关系基本上就是这样来指定，那它的上帝之手又在哪里呢？特别的简单，Android 的 IDE 在编译代码的时候，会根据 Layout 文件中的绑定关系，自动生成一批 Java 代码插入到你的源代码工程

中，这个过程你完全不用关心。你的 Layout 变化后，下一次编译时也会自动更新这些隐藏代码。

所以 Android 平台下的 MVVM 模式是在框架代码和 IDE 工具的辅助下，来实现了整个工作机制。想了解细节的朋友看一看这篇官方文档。
https://developer.android.com/topic/libraries/data-binding/index.html

其他平台的 MVVM 模式

而 iOS 平台下想实现 MVVM 就没有那么轻松了，有一个叫做 ReactiveCocoa 的第三方库实现了大部分 MVVM 工作机制，但是还没有像 Android 那么傻瓜化，还需要你手动写代码来指定绑定关系。有兴趣的朋友可以看这篇文章：

- https://www.raywenderlich.com/74106/mvvm-tutorial-with-reactivecocoa-part-1
- https://www.raywenderlich.com/74131/mvvm-tutorial-with-reactivecocoa-part-2

如何在 React-Native 下来实现 MVVM，大家可以研究一下一个叫做 MobX 的库。

那如何在微信小程序里面来实现 MVVM 呢？且看下回分解。