## MVC、MVP、MVVM

###### 到底什么是MVC，M、V、C之间如何通信？

V - 视图层

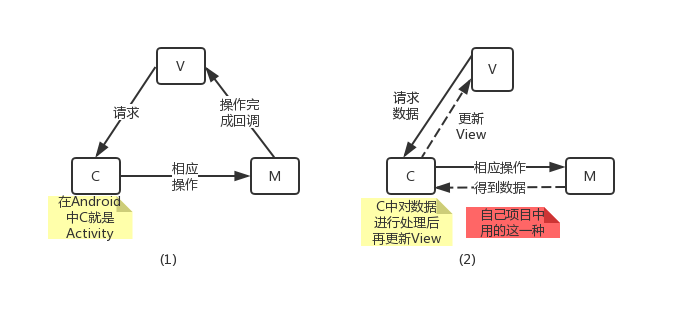
C - 控制器层

M - 数据层

* + 1. 编写MVC时的注意事项。

MVC模式如果使用不当，会导致大量的代码都集中在Controller层（也就是对应的Activity）。Android中V层的概念，我的理解应该是所有的View控件，用户操作View来调用C，然后C操作M获取数据，最后C拿到数据后更新V。是不是有点类似MVP？和MVP有什么区别呢？把C中的逻辑拿到了P中，简化了C层（Activity）。C层持有P层的实例，然后操作P层就行。

* + 1. MVC在实际项目中有哪些形式？



1. 图1是在维基百科中看到的。（号称是最标准的MVC，适用于.net）。

这块儿存在一个问题就是V要与M进行交互。但是M就是提供数据的，可以为任何一个V提供数据而不是单独的一个V。（图2的话，M为多个C提供数据，在不同的C中对数据进行处理）。

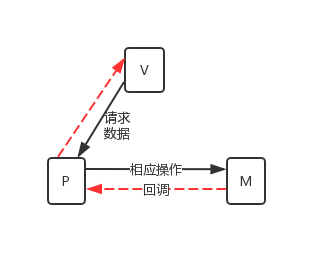
1. 图2是Android中适用的MVC类型。PAD项目使用的是图2这种方式，M和C通过观察者模式进行绑定，在Controller中请求完成后，设置model，然后model通知其它被观察者。





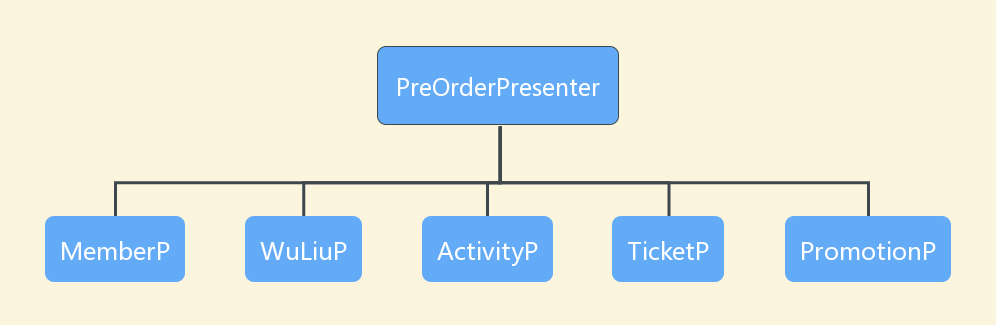
###### 什么是MVP，MVP与MVC的区别。

P - Presenter，代替了MVC的Controller，持有View的实例和Model的实例，避免V和M直接做交互。（其实和上述自己项目的MVC架构差不多，只不过自己的MVC把C和V合一起了，相当臃肿，可参考自己写的Demo）。



* + 1. MVP存在的问题

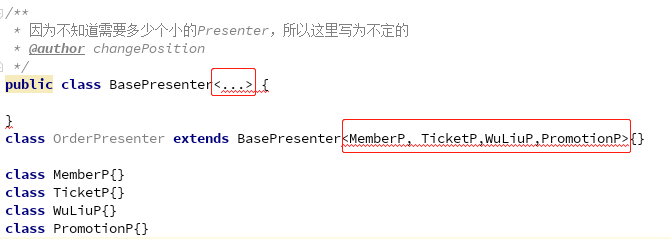
自己在项目中碰到的问题是：写着写着，P层变得和MVC中的Controller一样臃肿了，想到的办法扩展Presenter。拿PreOrderPresenter来说，在预下单模块要进行查会员信息、查券、查商品可参加的活动、地址信息、物流信息、促销计算金额等逻辑，如果都放在PreOrderPresenter中肯定会很乱（首先就是代码量多），怎么办？



这样就解决了，把一个Presenter拆为多个小的Presenter，可以想象成有一系列Presenter，哪个Activity想用哪个就持有哪个，交给当前Activity对应的总Presenter处理。

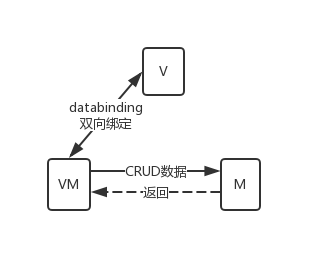
* + 1. 通过不定参数的泛型呢？

之前想过使用不定参数的泛型来实现，但是（目前）发现实现不了，所以只能手动设置（在主Presenter中手动set或者inject注入）。



* 1. 什么是MVVM？

VM - ViewModel，代替了MVP中的P，和P不同的是，VM和V之间实现了数据双向绑定，处理完业务逻辑后直接”更新”View，View有变化时直接反馈到VM中，进而改变Model。



## 网络协议

先看下网络分层（四层、五层、七层）。

|  |  |  |  |
| --- | --- | --- | --- |
| 七层模型（OSI） | 五层协议模型 | 四层模型（TCP/IP） | 各层协议、设备 |
| 应用层 | 应用层 | 应用层 | Http、Https |
| 表示层 |
| 会话层 |
| 传输层 | 传输层 | 传输层 | TCP、UDP |
| 网络层 | 网络层 | 网际层IP | IP地址、路由器、ARP |
| 数据链路层 | 数据链路层 | 网络接  口层 | MAC地址、交换机 |
| 物理层 | 物理层 |  |

五层协议的体系机构只是为了介绍网络原理而设计的，实际应用还是TCP/IP四层体系结构。

### 2.1、Http相关

###### 2.1.1、一个http请求携带了哪些信息（也就是请求报文有什么信息）？

|  |  |
| --- | --- |
| 请求行 | 请求方式（Post、Get）  请求地址（接口地址）  Http协议版本（HTTP/1.1，HTTP/2.0） |
| 请求头 | Accept（客户端接受的数据类型，image/\*，感觉就是MimeType类型）  请求体的数据类型（报文体）：application/json  请求端的IP地址  User-Agent（客户端类型，当用浏览器请求时一般是浏览器版本、如果用Http工具请求，比如App调接口，一般是请求工具的版本，比如Commons-HttpClient/3.1）  携带的内容长度：content-length  连接状态：Connection：Keep-Alive  是否支持缓存：cache-control，我们知道浏览器支持缓存，但请求工具如OkHttp也可以设置支持缓存 |
| 请求体 | Get、Post等携带的参数，拼接在url后的、设置在body中的 |

###### 2.2、Https相关

Https通过数字证书+非对称加密+对称加密来保证传输的安全。

**数字证书**：数字证书可以保证数字证书里的公钥确实是这个证书的所有者(Subject)的，或者证书可以用来确认对方的身份。

**对称加密**：客户端和服务端使用相同的密钥进行加密解密。算法公开、计算量小、加密速度快、效率高，加密的数据量大

**非对称加密**：使用公钥加密、私钥解密（所以一般通信双方会有两对密钥，双方都会充当加密、解密的角色）。加密和解密需要的时间长，速度慢，只能对少量数据加密。

综合以上，Https最终选择对称加密，需要通过一个协商的过程来生成一个密钥。

2.2.1、Https的加密具体流程

* 第一步：客户端发送一个请求，包含客户端支持的SSL/TLS版本信息、加密、解密、压缩算法等信息
* 第二步：服务端拿到信息后，返回一个消息，包含了：服务端SSL/TLS版本信息、服务端选择的加解密、压缩算法、以及数字证书
* 第三步：客户端拿到信息，使用本地CA列表进行验证证书合法性，对证书信息进行解密（用CA证书列表中的密钥），然后对解密后的信息进行签名、HASH得到A，对数字证书中的签名进行HASH得到，对比A、B是否相等来判断信息是否被修改，并且得到证书中的公钥信息Public
* 第四步：客户端生成一个随机数Random1，使用Public加密-Public(Random1)发给服务端
* 第五步：服务端接收到以后生成一个Random2，发给客户端。（注意：这里Random2并没有加密，因为我没有实际抓包，所以这块儿是很清楚，但个人感觉实际的实现方式应该是：使用上述的Random1对Random2加密或者使用私钥对Random2加密，但私钥一般不用来加密（可以，但一般不这么做））
* 第六步：客户端生成一个PreMasterSecret，使用Public加密发给服务端

最终，双方都有了三个随机数Random1、Random2、PremasterSecret，使用这三个数生成一个对称加密的密钥，进行以后数据的安全传输。

###### 2.3、TCP

2.3.1、三次握手

2.3.2、四次挥手

2.3.3、滑动窗口

2.3.4、拥塞控制

## 三、泛型

## 四、多线程

线程的分类

## 内存语义

## 设计模式

### 6.1 面向对象六大原则