|  |
| --- |
| 1、API设计  安全机制：调用者进行安全验证的方式；数据传输够不够安全。  接口协议标准化。  使用统一的命名规范，编程规范。  接口版本控制迭代，就是要解决接口的兼容性的问题，需要做好接口的版本控制。  面向接口编程。  2、框架分层  原则：低耦合高内聚。  简单的划分：数据管理、数据加工、数据展示。相应的三层架构：数据层、业务层、展示层。  数据层：缓存 + 网络API。（缓存策略设计的点：缓存数据保存的位置、缓存数据的有效时间。）  业务层：数据加工者，主要就是从数据层获取数据，然后经过业务逻辑处理后转化成展示层需要的数据。  展示层：作为数据展示者，它只要关心数据如何展示就可以了。  1）保持规范性：定义好开发规范，包括书写规范、命名规范、注释规范等，并按照规范严格执行；  2）保持单一性：布局就只做布局，内容就只做内容，各自分离好，每个方法、每个类，也只做一件事情；  3）保持简洁性：保持代码和结构的简洁，每个方法，每个类，每个包，每个文件，都不要塞太多代码或资源，感觉多了就应该拆分。  3、原生框架与混合框架：  原生开发固然体验比较好，但是开发周期相对于网页通常比较长，对于混合开发模式；  1）一般情况下；从用户体验的角度出发；为了提高用户体验；一般本地的一级页面，以及改动量比较小的页面，需要做成原生的。  2）基于公司实际情况出发，经常变动版本；改动比较大的或者详情页面我们可以做成网页形式，便于我们版本迭代更新  3）复杂的软件必须有清晰合理的架构，否则无法后期扩展和维护；通常情况下；我们会结合业界比较成熟的一些设计模式。  4、MVC  § 视图（View）：用户界面。  § 控制器（Controller）：业务逻辑  § 模型（Model）：数据保存  它们的通信方式：  1.View传送指令到Controller  2.Controller完成业务逻辑后，要求 Model改变状态  3.Model将新的数据发送到 View，用户得到反馈.  所有的通信都是单向的。  5、MVP  MVP模式作为一种新型模式，是从经典的模式MVC演变而来，它们的基本思想有相通的地方，将 Controller 改名为Presenter，同时改变了通信方向。  1. 各个部分之间的通信，都是双向的。  2. View 与 Model 不发生联系，都通过 Presenter传递。  3. View 非常薄，不部署任何业务逻辑，称为"被动视图",即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。  6、MVVM  MVVM (数据绑定) 最早是微软提出的；模式将 Presenter 改名为ViewModel，基本上与 MVP 模式完全一致。  简单的说，ViewModel就是View与Model的连接器，View与Model通过ViewModel实现双向绑定。  唯一的区别是，它采用双向绑定：View的变动，自动反映在 ViewModel，反之亦然.  谷歌推出了Data binding这个框架，Data Binding是一个 support包，因此与 Android M没什么关系，可以轻松的实现MVVM，Data binding解决了 Android UI 编程中的一个痛点。 |

# API设计

## 数据—API

一个APP最重要的就是数据，而数据的来源就是API。

## 安全机制

设计API首先要考虑的就是API的安全机制。

主要存在两个漏洞：

1. 是因为缺少对调用者进行安全验证的方式；
2. 是因为数据传输不够安全。那么，制定API的安全机制，主要就是为了解决这两个问题：

A.保证API的调用者是经过自己授权的App；——>通过加设计签名的方式，设计签名算法。

B.保证数据传输的安全。——>采用HTTPS，其添加了SSL安全协议。

## （3）接口协议标准化

API返回的数据，一般是采用JSON格式进行传输。比如，前后端平台对Date的处理不同，导致错误，最好偶读方法是，统一使用时间戳。

## （4）使用统一的命名规范，编程规范。

## （5）接口版本控制迭代

问题：经常出现，因为接口变动导致旧版本的APP出错的问题。这类问题，就是要解决接口的兼容性的问题，需要做好接口的版本控制。

实现上，一般有两种做法：

每个接口有各自的版本，一般为接口添加个version的参数；

整个接口系统有统一的版本，一般在URL中添加版本号，比如http://api.domain.com/v2。

平时小版本的更新，就采用第一种方式，我们的做法是根据不同版本号做不同分支处理。大版本的更新，则用第二种方式，这时候，基本就是一套全新的接口系统了，跟旧版本是相对独立的。

当版本越来越多时，维护就会成为一个大问题，我们没那么多精力去维护所有版本，因此，太旧的版本一般就不会再维护了。这时候，如果有用户还在使用即将废弃的旧版本，需要提醒用户升级到新版本。

# 框架分层

## 背景

早期的时候，Android就是将所有操作都放在Activity里完成，包括界面数据处理、业务逻辑处理、调用API。后来发现Activity越来越臃肿，代码越来越复杂，很难维护。于是就开始思考如何拆分，如何才能做到低耦合高内聚。

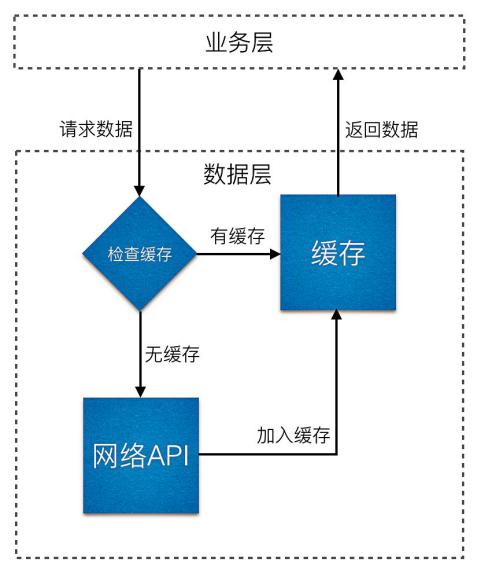
## 逻辑分层

前面也说过，一个App的核心就是数据，那么，从App对数据处理的角色划分出发，最简单的划分就是：数据管理、数据加工、数据展示。相应的也就有了三层架构：数据层、业务层、展示层。它们之间的关系如下图，数据层是三层中的最底层，往下，它接入API；往上，它向业务层交付数据。业务层夹在中间，属于数据的加工厂，将数据层提供上来的数据加工成展示层需要展示的数据。展示层处于三层中的最上层，主要就是将从业务层取得的数据展示到界面上。



## ****（3）数据层****

数据层是数据管理者，主要任务就是封装API，并将数据结果交付给上层，中间会再加个数据缓存。整个主流程如下图：



1. 业务层向数据层请求数据；
2. 数据层检查缓存中有没有请求需要的数据；
3. 如果有缓存数据，则直接返回缓存数据；
4. 如果没有缓存数据，则从网络API获取数据，并将数据加入缓存，然后返回数据。

调用网络API时，还要判断网络状态，根据不同状态做不同处理。如果网络不可用，就无需发起请求了。

网络可用时，也要区分是连接WIFI还是连接移动网络。连接移动网络时，一般需要限制调用比较耗流量的请求。连接WIFI时，则无需设置这种限制，而且还可以预先请求一些接口，比如请求当前分页数据时，可以将下一页的数据也预先请求。

缓存也需要缓存策略，不同的接口需要做不同的缓存处理。

首先，缓存只适用于获取数据的接口，对于修改数据的接口则不适用。

其次，不同接口缓存时间一般也不同，对于很少变动的数据缓存时间可以设置长一些，而频繁变动的数据缓存时间则比较短，甚至不进行缓存。

最后，缓存数据因为比较多，我们一般保存在数据库，而对于调用频率高、最新的数据，还会在内存中也拥有一份缓存，不过缓存时间比较短。请求缓存数据时，会先检查内存缓存中有没有，有则直接将缓存的数据返回，没有才从数据库获取。

（缓存策略设计的点：缓存数据保存的位置、缓存数据的有效时间。）

那么，如何将数据交付给业务层呢？这是整个数据层模块与外部交互的部分，当与外部交互的时候，一般都要符合面向接口编程的原则，因此只要提供开放的数据接口就可以了。

## ****（4）业务层****

业务层是数据加工者，主要就是从数据层获取数据，然后经过业务逻辑处理后转化成展示层需要的数据。业务层因为夹在数据层和展示层中间，起着承上启下的作用。也因此，业务层很容易沦落为只是一个数据的中转站，主要就是因为对业务层具体的作用和职责没有理解清楚。

这里用一个例子来说明业务层具体的工作吧，就举个用户注册的例子。用户注册时，界面上需要用户提供手机号、短信验证码、密码、确认密码。那么，最简单的操作就是，带上这些参数调用数据层的注册接口。好了，问题来了，注册接口并没有提供确认密码的参数。那好，调用注册接口之前先判断下密码和确认密码是否一致，不一致则返回错误提示给用户，一致了才调用注册接口。好了，第二个问题来了，用户等网络请求等了一段时间后，请求结果返回说手机号少了一位。下一次，又等了一段时间，这次又返回说手机号多了一位。就因为一个小错误要让用户等那么久，用户肯定有意见。后台也有意见，各种非法的请求都发过来，是嫌服务器压力不够大啊。那好，调用接口之前对这些参数做有效性检查吧，手机号要规范，短信验证码只能为六位数字，密码不能少于六位。终于注册成功了，第三个问题又来了，注册接口是没有返回用户的accessToken的，只有登录接口才会返回。让用户手动再登录一下？这用户体验不太好啊。正确的姿势应该是注册成功后再自动调用一次登录接口，如果因为网络问题第一次登录失败，后面还需要再自动调用多一次，如果还是调用失败，才让用户手动登录。

上面的例子中，对参数的有效性检查，注册成功后的自动登录，都属于业务逻辑的处理，也就是说都是业务层的工作。

业务层交付给展示层的数据也是通过接口的方式，不过，和数据层交付给业务层时不同的是：交付给展示层的数据应该是通过异步回调返回的。因为获取数据是一个比较耗时的任务，通过异步回调才不会阻塞UI主线程。

## ****（5）展示层****

展示层作为数据展示者，它只要关心数据如何展示就可以了。不过，数据如何展示却不是那么简单。展示层是三层架构中最复杂的一层了，要考虑的东西远远多于其他两层，涉及的东西包括但不限于界面布局、屏幕适配、图片资源、文本资源、颜色资源等等。在开发一段时间后，展示层出现代码混乱是最常见的。因此，做好展示层，就需要保持高质量的代码。要保持高质量代码，我觉得至少应该遵循几条基本的原则：

1）保持规范性：定义好开发规范，包括书写规范、命名规范、注释规范等，并按照规范严格执行；

2）保持单一性：布局就只做布局，内容就只做内容，各自分离好，每个方法、每个类，也只做一件事情；

3）保持简洁性：保持代码和结构的简洁，每个方法，每个类，每个包，每个文件，都不要塞太多代码或资源，感觉多了就应该拆分。

所谓无规矩不成方圆，展示层的设计，要从开发规范开始。一份好的开发规范，是保证代码有较高的可读性的基础。iOS方面，苹果已经有一套[Coding Guidelines](https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/CodingGuidelines.html" \t "https://blog.csdn.net/fengxinziyangyang/article/details/_blank)，主要属于命名方面的规范。当我们制定自己的开发规范时，首先就要遵守苹果的这份规范，在此基础上再加上自己的规范。Android方面，我也在我的博客中分享过一套（[Android技术积累:开发规范](http://keeganlee.me/post/android/20150709" \t "https://blog.csdn.net/fengxinziyangyang/article/details/_blank)），主要分为书写规范、命名规范、注释规范三部分。

最重要的不是开发规范的制定，而是开发规范的执行。如果没有按照开发规范去执行，那开发规范就等于形同虚设，那代码混乱的问题依然得不到解决。

说到单一性，面向对象设计中，有一个基本原则就是单一职责原则，它规定一个类应该只有一个发生变化的原因。保持单一性是减低耦合度的关键标准，其目的就是各方面的解耦。而我这里说的单一性不只是规定类的单一，也包括界面的单一、方法的单一、资源文件的单一等。

界面的单一，首先是界面的布局和界面的数据应该分离。另外，界面数据的获取和展示也应该分离。一句话，保持界面的单一性就是要保持界面上每个维度都做好分离，从界面的布局，到数据的获取，数据的检查，数据的展示。

方法的单一，则表现为一个方法是对一个行为的封装。行为又可以拆分为多个步骤，每个步骤其实也是更细化的行为。因此，方法嵌套方法是一种常态。那么，保持方法的单一性，关键不在于怎么定义这个方法的行为，而在于这个行为要怎么拆分成更细的行为。举个例子，通常在Activity的onCreate方法，做初始化操作，细分出来就分为了：控件的初始化、逻辑变量的初始化、数据的初始化。数据的初始化又可以再细分：数据的获取、数据的展示。每个细化的行为都应该封装为一个独立的方法，这样，才真正符合方法的单一性。

资源文件的单一，主要是指Android的各类资源文件，包括存放字符串的strings.xml，存放字符串数组的arrays.xml，存放颜色值的colors.xml，存放尺寸值的dimens.xml，等等。资源文件的单一，是说所有相关的资源信息要在资源文件里定义并引用到代码或布局文件里，而不是在代码或布局文件里直接定义。这样做，可以很方便地做各种适配和修改，比如支持国际化，比如不同分辨率的屏幕用不同尺寸值。iOS则没有提供和Android一样的资源文件分离的机制，但可以参考Android的做法自己去实现。

# MVC—MVP—MVVC

## 原生框架与混合框架：

原生开发固然体验比较好，但是开发周期相对于网页通常比较长，对于混合开发模式；我们要考虑以下几个方面：

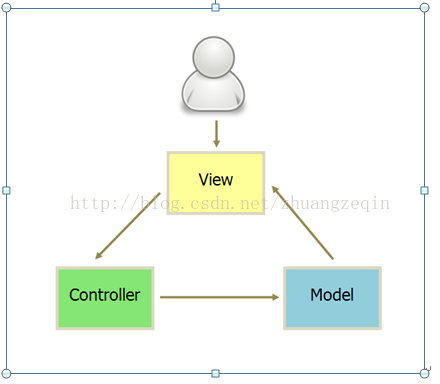
1）一般情况下；从用户体验的角度出发；为了提高用户体验；一般本地的一级页面，以及改动量比较小的页面，需要做成原生的。

2）基于公司实际情况出发，经常变动版本；改动比较大的或者详情页面我们可以做成网页形式，便于我们版本迭代更新

3）复杂的软件必须有清晰合理的架构，否则无法后期扩展和维护；通常情况下；我们会结合业界比较成熟的一些设计模式。

## （2）MVC

MVC是最常见的软件架构之一，业界有着广泛应用，也是早期我们APP 设计时最常见的一种架构模式.



§ 视图（View）：用户界面。

§ 控制器（Controller）：业务逻辑

§ 模型（Model）：数据保存

MVC模式的意思是，软件可以分成三个部分。它们的通信方式

1.View传送指令到 Controller

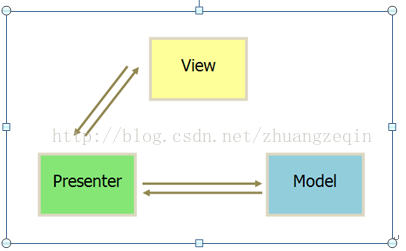
2.Controller完成业务逻辑后，要求 Model改变状态

3.Model将新的数据发送到 View，用户得到反馈.

所有的通信都是单向的。

## （2）MVP

MVP模式作为一种新型模式，是从经典的模式MVC演变而来，它们的基本思想有相通的地方，将 Controller 改名为Presenter，同时改变了通信方向。



1. 各个部分之间的通信，都是双向的。

2. View 与 Model 不发生联系，都通过 Presenter传递。

3. View 非常薄，不部署任何业务逻辑，称为"被动视图",即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

4模型与视图完全分离，我们可以修改视图而不影响模型；

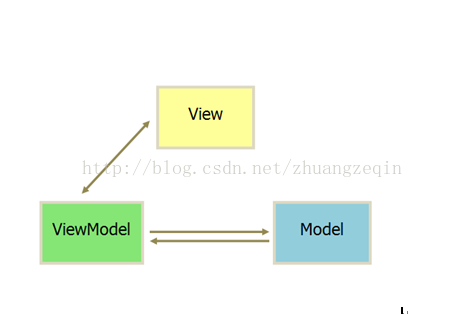
5可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter内部；

6我们可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁；

7如果我们把逻辑放在Presenter中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）

## （3）MVVM

MVVM (数据绑定) 最早是微软提出的；模式将 Presenter 改名为ViewModel，基本上与 MVP 模式完全一致。



简单的说，ViewModel就是View与Model的连接器，View与Model通过ViewModel实现双向绑定。

唯一的区别是，它采用双向绑定：View的变动，自动反映在 ViewModel，反之亦然.

谷歌推出了Data binding这个框架，Data Binding是一个 support包，因此与 Android M没什么关系，可以轻松的实现MVVM，Data binding解决了 Android UI 编程中的一个痛点。

## （1）MVP模式

MVP是一种使用广泛的基础架构模式，使用基于事件驱动的应用框架。MVP从更早的MVC框架演变过来的一种框架，与MVC有一定的相似性。

MVP框架由3部分组成：View负责显示，Presenter负责逻辑处理，Model提供数据。

MVP与MVC之间最主要的区别在控制层上，在MVP框架中，View与Model并不直接交互，所有的交互放在Presenter中；而在MVC里，View与Model会直接产生一定的交互。

MVP的Presenter是框架的控制者，承担了大量的逻辑操作，而MVC的Controller更多时候承担一种转发的作用。

因此在App中引入MVP的原因，是为了将此前在Activty中包含的大量逻辑操作放到控制层中，避免Activity的臃肿。MVP模式使得View与Model的耦合性更低，降低了Presenter对View的依赖，实现了关注点分离的初衷，方便开发人员的编码和测试工作。

具体到Android App中，我一般将App根据程序的结构进行纵向划分，对应MVP分别为模型层，UI层和逻辑层。UI层一般包括Activity，Fragment，Adapter等直接和UI相关的类，UI层的Activity在启动之后实例化相应的Presenter，App的控制权后移，由UI转移到Presenter，两者之间的通信通过BroadCast、Handler或者接口完成，只传递事件和结果。举个简单的例子，UI层通知逻辑层（Presenter）用户点击了一个Button，逻辑层（Presenter）自己决定应该用什么行为进行响应，该找哪个模型（Model）去做这件事，最后逻辑层（Presenter）将完成的结果更新到UI层。

## （2）MVP模式存在的问题

# 架构设计的目的

# 5、架构设计之模块化

## （1）为什么要模块化

一切是为提高生产力。通过设计使程序模块化，做到模块内部的高聚合和模块之间的低耦合。可以提高程序的开发效率，以及后续的测试、定位问题，以及维修修改代码。

良好的模块化设计对于Android客户端开发来说，在方很多方面都有重要意义。例如，合理的模块组织层次，会增加代码的可读性；恰到好处的粒度控制，能让模块灵活组织，增加拓展性；模块间互相独立，会让代码复用性、移植性增强，增加技术储备的可用性；众多可独立测试的模块，会让代码的调试、Debug效率更高。

但合理的模块化设计并不容易，除了设计本身的难度意外，更大的挑战是协作开发的设计统一和架构膨胀时的有效控制。同时由于软件设计高复杂度的特性，规定设计标准需要非常繁杂、抽象、难以理解的文档，可执行性较差。

## （2）模块的划分

Android中的模块划分方法，总的来说有职责拆分法和职责组合法两种方法。

拆分法是指把Android应用整体逐级拆解，由大到小分为A级、B级、C级等等。例如，把faceu作为一个软件项目，那么拍照、直播、IM等是A级模块，A级模块将会依赖其各自的子模块，也就是B级模块，其他级模块以此类推。

组合法是指以Android系统提供的Api为基础，逐级组合成父模块直到整个应用，由小到大分为一级、二级、三级等等。也就是说，一级模块只依赖Android（能单独编译），二级模块只依赖Android和一级模块，三级模块只能依赖Android、一级和二级，其他级别模块以此类推。

两种方法虽然方向不同，一种是从父模块到子模块，一种是从子模块到父模块，但本质上殊途同归。之所以一分为二，是因为在实际的工作中，模块化工作往往是滞后的，这种从两头到中间汇合的方式可操作性比较好，类似于打通隧道。由于开发是由需求驱动的，所以模块的拆分要与需求拆分对应，从这个层面来看，拆分法往往是前期架构、设计起步时采用的方法，而组合法是后期为了满足移植、拓展等需求变化而进行的软件优化和细化。

## （3）模块化的程度

模块化程度判定的唯一标准就是“独立可用模块”的多少。

从组合法来看，模块有很多分级别，想要“独立可用模块”多，最关键的就是一级模块要多，因为所有的模块都是由一级模块直接或间接组合出来的，只有一级模块多才能组合出更多的高级模块。其次，模块组合的层次要多、要错落有致，这样在组合高级模块时就会有更多次高级的子模块可用，比较简洁、拓展性强。最后，别为了模块化而去模块化，保障组合出来的各级模块都是“可用”模块。

从拆分法来看，思路大体一致。

1）模块依赖

模块间要组合、要协作，就可能产生依赖，而依赖是模块化过程中要解决的头号难题。

理想情况下，依赖关系只存在于父模块与其直接子模块，也就是平级间不存在依赖、子对父不存在依赖、超越父子层级（爷爷）间不存在依赖。

可以容忍情况，模块通过定义代理实现对外的依赖。比如，想把下拉刷新设计为一个一级模块，同时打印、UI工具类也都是一级模块，而下拉刷新与他们平级，不能依赖它们。这种情况下可以定义一个代理类，代理类中转调用其他模块的方法，这样在下拉刷新移植的时候需要改动的地方就集中在一个点上，方便移植。如果下拉刷新模块只有一个类，那么可以定义若干代理方法并转调用打印、UI工具类方法。

严格来讲，上述下拉刷新的情况是可以通过设置回调的方式，设计成无依赖模块的，代理的方式是对不可避免的过度依赖基础模块的妥协。越是解耦，越是从子模块直接依赖转化为靠父模块建立子模块依赖关系，依赖本身的复杂度越高。

2）标准

a.任何类最多只能依赖三个非android类（控制类除外，视图树类除外）

b.任何模块整体只能有一个public类（允许该public具有内部public类）。

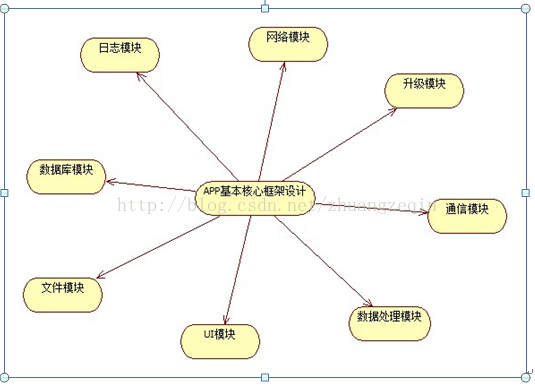
c.任何父模块最多只能依赖三个子模块。（控制模块除外，视图树模块除外）

3）量化

对于标准中的abc三个标准，全部达到100%则为S，90%为A，80%为B，70%为C，60%为D。例如，abc达成率分别为90%、80%、70%，则模块化等级为C。

## （4）APP框架设计包括哪些内容

APP框架搭建的设计；主要的核心思想还是分层思想，通常设计下，会包括以下内容:(如下图)



# 6、组件化框架设计