App架构经验总结

架构因人而异，不同的架构师大多会有不同的看法；架构也因项目而异，不同的项目需求不同，相应的架构也会不同。然而，有些东西还是通用的，是所有架构师都需要考虑的，也是所有项目都会有的需求，比如API如何设计？架构如何分层？开发环境和生产环境如何分离？这几年，我负责研发过的App，有餐饮类的、社交类的、智能家居类的、电商类的、新闻媒体类的等等。当有了一定的经验之后，你总会有一些自己的心得体会。而以下内容就是根据我的这些经历提炼出来的关于以上几个问题方面的经验总结，内容不多，旨在抛砖引玉。

从API开始

一个App，最核心的东西，其实就是数据，而数据的主要来源，就是API。我之前负责的项目，因为API的坑已经受过了不少苦，因此，之后对App项目的架构设计我都会先从API开始。

制定安全机制

设计API第一个需要考虑的是API的安全机制。我负责的上一个项目，因为API的安全问题，就被人攻击了两次。之后经过分析，主要存在两个漏洞：一是因为缺少对调用者进行安全验证的方式，二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：

保证API的调用者是经过自己授权的App；
保证数据传输的安全。

第一个问题的解决方案，我主要采用设计签名的方式。对每个客户端，Android、iOS、WeChat，分别分配一个AppKey和AppSecret。需要调用API时，将AppKey加入请求参数列表，并将AppSecret和所有参数一起，根据某种签名算法生成一个签名字符串，然后调用API时把该签名字符串也一起带上。服务端收到请求之后，根据请求中的AppKey查询相应的AppSecret，按照同样的签名算法，也生成一个签名字符串，当服务端生成的签名和请求带过来的签名一致的时候，那就表示这个请求的调用者是经过自己授权的，证明这个请求是安全的。而且，每个端都有一个Key，也方便不同端的标识和统计。为了防止AppSecret被别人获取，这个AppSecret一般写死在代码里面。另外，签名算法也需要有一定的复杂度，不能轻易被别人破解，最好是采用自己规定的一套签名算法，而不是采用外部公开的签名算法。另外，在参数列表中再加入一个时间戳，还可以防止部分重放攻击。

第二个问题的解决方案，主要就是采用HTTPS了。HTTPS因为添加了SSL安全协议，自动对请求数据进行了压缩加密，在一定程序可以防止监听、防止劫持、防止重发，主要就是防止中间人攻击。苹果从iOS9开始，默认就采用HTTPS了。而关于在Android中如何使用HTTPS，Google官方也给出了很多安全建议。不过，大部分App并没有按照安全建议去实现，主要就是没有对SSL证书进行安全性检查，这就成为了一个很大的漏洞，中间人利用此漏洞用假证书就可以通过检查，从而可以劫持到所有数据了。因此，为了安全考虑，建议对SSL证书进行强校验，包括签名CA是否合法、域名是否匹配、是不是自签名证书、证书是否过期等。

接口协议标准化

API返回的数据，一般都是采用JSON格式进行传输。然而，JSON的值只有六种数据类型：

Number：整数或浮点数
String：字符串
Boolean：true 或 false
Array：数组包含在方括号[]中
Object：对象包含在大括号{}中
Null：空类型

我遇到过的，关于API的坑有大部分就是因为JSON数据和实体对象转化时出错导致的，而且是各种各样的错误都有，其中不乏有一些很奇葩的错误。

最麻烦的就是处理Date类型，因为JSON本身没有Date类型，因此，JSON库将Date类型的数据序列化时会转为String。这时，不同环境，不同平台，以及用不同的JSON解析库，转换后的结果经常会不同。比如，你在开发机上可能得到的结果是”2016-1-1 17:11:11”，但放到服务器后结果却变成了“Jan 1,2016 5:11:11 PM” ，客户端进行反序列化时无疑会失败。后来，我取消了所有Date类型，统一采用时间戳表示，就再没有转化的烦恼了。

另外，接口的开发人员有时候会将一些数据错误地转换为了String，导致客户端使用时因类型错误而异常。例如，本来是数字的1，被转成了"1"，客户端做运算时就会出错，或用switch判断时也会出错，或其他无法转换的情况发生时；例如，为空时JSON正确地表示应该是null，但如果转为了String就变成了"null"，那问题就来了，我遇到的因为这个错误的转换导致的程序奔溃已经好几次了，第一次的时候，查了一整天才定位到问题所在。

还有，因为接口的开发人员不同，很多时候还会出现不同接口同一个意思的参数名称却不同。比如，对于有分页数据的接口，一般都有当前页的参数，A开发人员可能将参数命名为currentPage，第一页是从0开始；B开发人员在另一个接口则命名为currPage，第一页却从1开始；C开发人员在另一个接口又命名为presentPage，第一页又是从0开始。客户端的开发人员看到也是醉了。

每个技术团队一般都会有一份接口协议文档，主要内容包括每个接口的描述、入参、输出结果等，但一般并不严谨，很多地方没有统一标准，从而容易出现很多坑。因此，有一份统一标准且严格执行的接口协议非常重要。协议的内容除了规定每个接口，包括接口中每个数据具体的数据类型，还需要规定一套共用的数据字典，以及其他需要统一定义的信息，比如签名算法等。一旦有了这份统一标准且严格执行的接口协议，很多问题都将迎刃而解。

接口版本控制

我们已经不止一次因为接口发生变动而导致旧版本的App出错的问题，而且变动不一定是修改了接口本身，有可能是底层增加了一种新的数据结构，接口把新数据也返回给客户端了，但客户端旧版本是解析不了的，从而就导致出错了。

为了解决接口的兼容性问题，需要做好接口版本控制。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。

架构分层

API的设计完成之后，接下来我就会考虑App项目的整体架构了。整体如何架构，我也曾经做过不少尝试。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，如何才能做到松耦合高内聚。

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。业务层夹在三层中间，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。

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：

业务层向数据层请求数据；
数据层检查缓存中有没有请求需要的数据；
如果有缓存数据，则直接返回缓存数据；
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。曾经，我们没有对移动网络状态下的请求进行限制，结果，测试时流量DuangDuangDuang地一下子就不见了十几M。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。

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。对于接口的参数需要说明一下，上面提到的参数有appKey、version、currentPage这几个，还有签名sign、时间戳time，其实可以分为两类：系统参数和业务参数。像appKey、version、sign、time这些属于系统参数，而

currentPage，或username之类的则属于业务参数。数据层开放的数据接口的参数只需要包含业务参数就可以了，业务层并不需要关心系统参数是什么，系统参数在数据层内部封装API时指定就可以了。

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。

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：

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；
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；
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，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），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。
环境分离

每个App项目，至少都会有两个环境：测试环境和生产环境。多的甚至有四个环境：开发环境、测试环境、预生产环境和生产环境。开发人员经常需要在环境之间切换，测试人员也同样。经常出现测试人员今天需要测试环境的最新版本，叫App开发人员打包一个给她，明天需要切换到生产版本，再叫App开发人员打包一个生产环境的给她。我们知道，一个App，在一台手机上要么只能是测试环境的，要么只能是生产环境的。测试人员要测试两个环境，只能不断替换不同环境的同个App，这实在太麻烦了。为了解决此问题，最好的方案就是环境分离，不同环境有不同的App。

一个App的唯一标识，Android是用包名，iOS是用Bundle Identify。那么，在一个系统想安装不同环境的App，只要每个环境App的包名和Bundle Identify不同即可。比如，生产版的包名和Bundle Identify命名为com.mydomain.myapp，测试版的包名和Bundle Identify则命名为com.mydomain.myapp.beta，这样，Android和iOS都会识别为两个不同的App了。

不过，只改包名和Bundle Identify是不够的，应用图标和应用名称也要修改，不然安装之后很难区分哪个App是哪个环境的。一般做法就是，非生产环境的App图标就是在生产图标的基础上添加一个环境标签，同时App的应用名称也是在生产的基础上添加环境后缀名。另外，因为包名和Bundle Identify不同了，微信、微博、百度地图等这些第三方平台也都需要为不同环境的App分别申请不同的appID。

实现上，最笨的方法就是拷贝当前工程，然后修改，缺陷很明显，维护成本很高。不过，好在Android和iOS都有很方便的修改方式。

Android有了Gradle，可以设置多个不同的Flavors，每个Flavor都有一个applicationId属性，其实就是App的包名。比如，生产版和测试版的设置如下：

```
productFlavors {
    myapp {
        applicationId "com.mydomain.myapp"
    }
    myappBeta {
        applicationId 'com.mydomain.myapp.beta'
    }
}
```

这样，其实就有两个App了。然后，源代码新建一个和main同级的目录，命名为myappBeta，然后，将图标、名称和第三方设置之类的，和main保持一样的位置、文件名、属性等，就可以替换成环境相关的了。

iOS则可以通过创建多个环境的Target来实现环境分离，不同Target可以设置不同的Bundle Identify、Bundle display name、更换图标。另外，每个Target也各自有自己的一份plist文件的，环境变量和第三方设置之类的，都可以设置在相应的plist文件里。