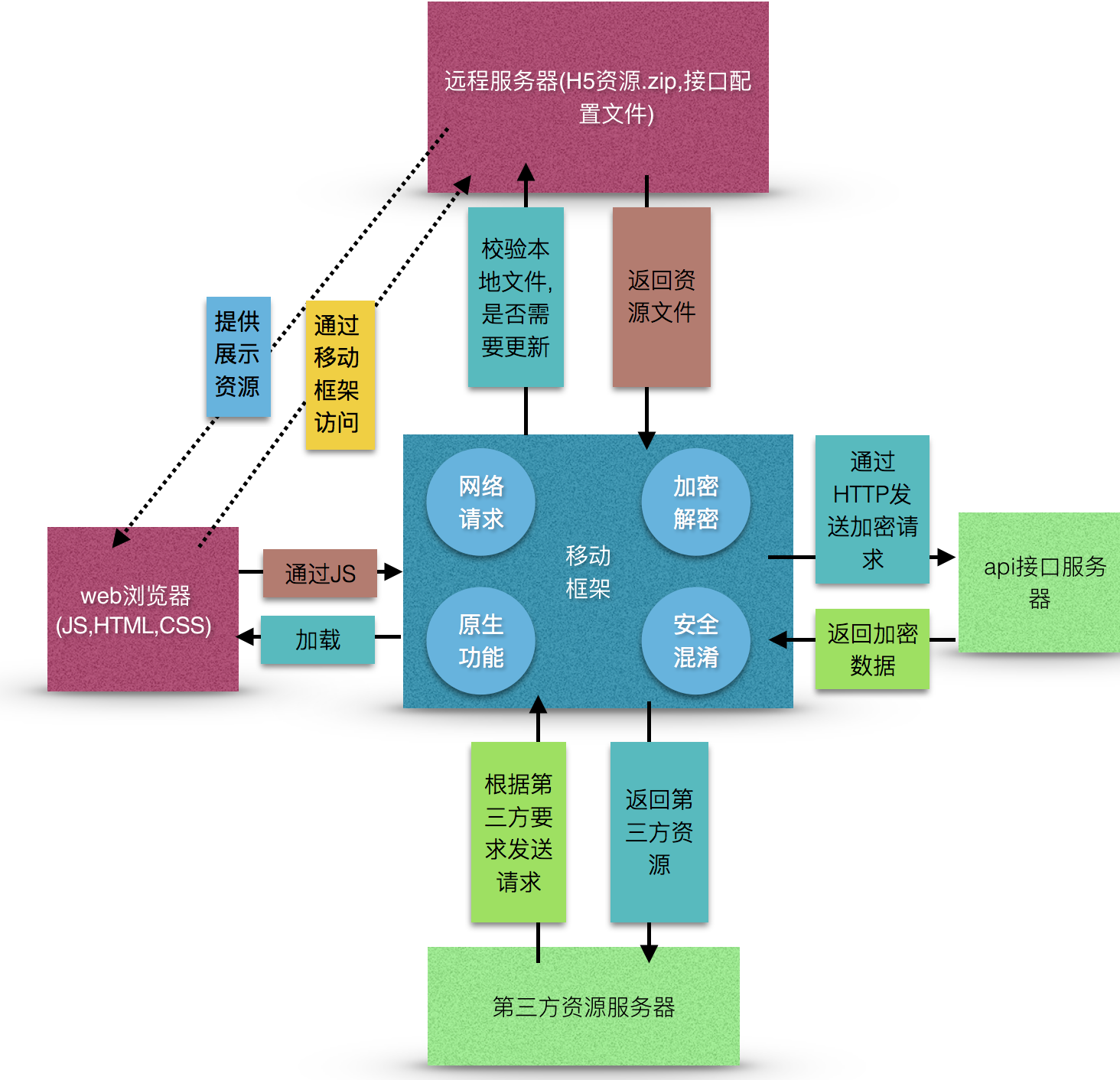
**移动端整体架构设计**

**一、设计原理图**

****

**二.整体架构设计思路**

1.资源包文件不需要再app中,app的安装包较小.

2.app初始启动,进行与远程服务器的文件对比校验,动态加载相关app最新资源文件.

3.原生框架提供JS所需的功能接口.

4.原生框架不需要知道任何有关业务需求方面事件,做到MVC彻底分离,减少移动安全的风险.

5.原生框架在服务器资源有变动时可以实时加载新数据,保证bug,随见随改.

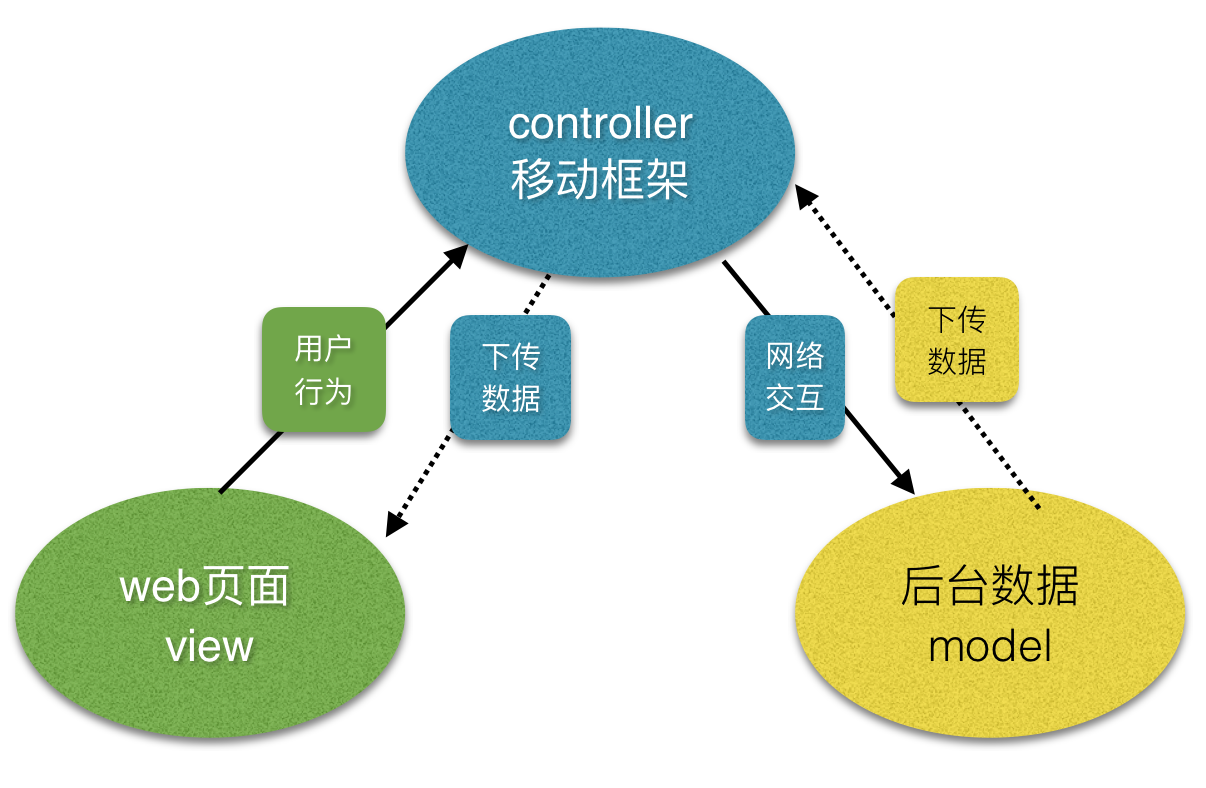
6.功能项动态加载,动态更新,灵活的配置app页面,极大地提高开发效率,为用户和开发者提供方便.

7.api接口随时更新.

8.减少本迭代和维护产生的开发成本.

9.把服务器,页面,原生框架以MVC设计模式彻底分离.

**三、整体架构设计**

****

采用经典的mvc框架

* **层次**
* **作用**
* **设计原则**

1. **模型层(model):后台接口数据**

封装了应用的一系列数据, 并定义了操作, 处理这些数据的逻辑和计算规则。

通过Notification,KVO对控制器进行反馈

1. **视图层(view):前端页面展示**

视图对象是一个应用中, 用户可以看到的对象. 视图对象知道如何绘制自己, 也能够响应用户的操作. 视图对象的主要目的之一是将应用模型对象中的数据显示出来, 并允许用户编辑该数据

视图通过不能直接操作模型层, 通过target-action, delegate, dataSource和控制器进行反馈

1. **控制器层(controller):原生框架接口**

控制器层是在视图层和若干个模型层的中间人

c可以直接操作模型层和视图层

**四、移动端架构设计方案**

1、API

一个App，最核心的东西就是数据，而数据主要来源是API

制定安装机制

1)、保证API的调用者是经过自己授权的App

采用设计签名的方式。对每个客户端，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。

采用签名算法：有一定的复杂度，不能轻易被别人破解，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。

2)、保证数据传输安全

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。

3)、接口协议标准化

一般都是采用JSON或者XML

4)、接口版本控制

每个接口有各自的版本，一般为接口添加个version的参数；根据App的版本号进行版本迭代

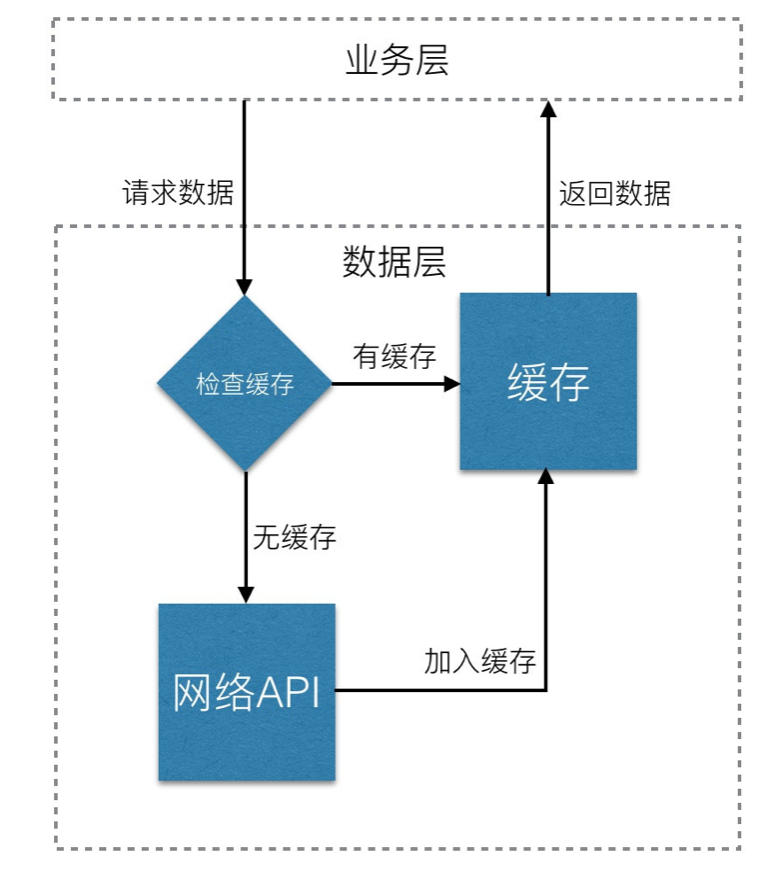
整个接口系统有统一的版本，一般在URL中添加版本号，

2、架构分层



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。

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。

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。

3、安全

App内所有交互数据全部加密，App本身提供混淆视听，加固，签名效验等。

架构采用MVP以及中间人模式设计，以下是简要设计思维导图

