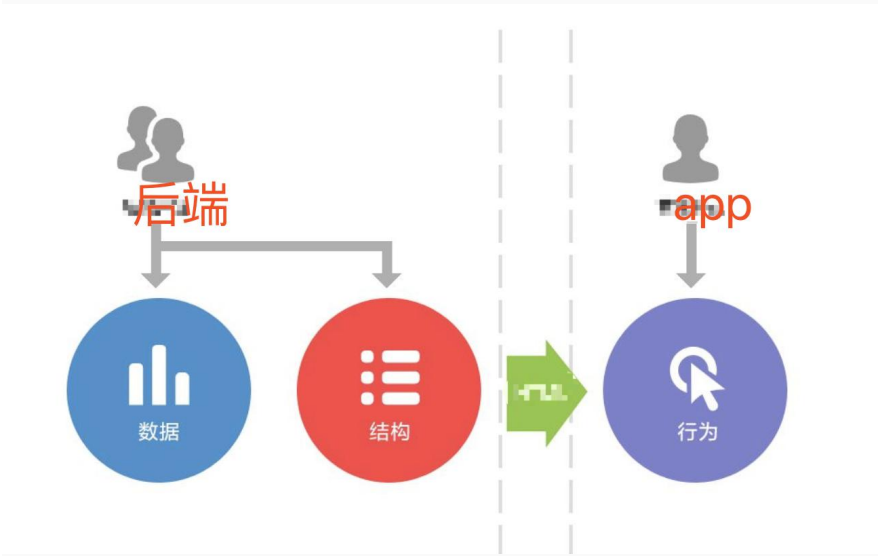
APP 架构之后端接口设计方案之初探

App 与服务器的接口设计需要考虑很多地方，这里整理项目中遇到的和使用到的一些接口设计原则，抛砖引玉。



1 设计思想

APP 对服务器端要求是比较严格的，在移动端有限的带宽条件下，要求接口响应速度要快，所有在开发过程中尽量选择效率高的框架，对数据要求也比较严格，app 需要什么数据就传什么数据，不可多传，过多的数据量影响处理速度，最重要的是影响传输效率。接口要规范，以面向对象的思想设计接口。

2 app 后端和 java web 后端的区别

其实对于后台开发来说原理都差不多。只不过 app 的后台开发和 web 不一样的地方在于传输数据格式不一样，一般来说 web 访问后返回的是一个 html 页面，少部分是 json 格式；而一般 app 的后台开发大部分直接传 json 格式数据（也有不是 json 格式的，看项目的选择，但一般来说都是 json），少部分会直接返回 html5 的页面。

还有一个不同点在于登录验证和数据加密，一般 web 是使用 session 验证登录状态，而 app 则使用 token 来验证登录状态（token 是自己定义的一个和用户 ID 相关的加密字符串，传入后台后从数据库查询用户信息）。还有如果对安全性要求较高，app 传输数据时可能会对数据进行加密，而 web 一般没有这一步，web 的加密一般是使用 https。

至于说 android 和 ios 的开发环境不一样那是指的 app 开发，和后台无关。app 的后台和 java web 的后台没有本质区别。app 的一个后台可以即提供给 android，也可以同时提供给 iOS，它就是把 app 提交的数据处理后插入数据库和从数据库查出数据处理后传给 app。

3 安全机制的设计

3.1 服务端 token 方式-类似 session

现在，大部分 App 的接口都采用 RESTful 架构，RESTFul 最重要的一个设计原则就是，客户端与服务器的交互在请求之间是无状态的，也就是说，当涉及到用户状态时，每次请求都要带上身份验证信息。实现上，大部分都采用 token 的认证方式，一般流程是：

1、用户用密码登录成功后，服务器返回 token 给客户端；

2、客户端将 token 保存在本地，发起后续的相关请求时，将 token 发回给服务器；

3、服务器检查 token 的有效性，有效则返回数据，若无效，分两种情况：

token 错误，这时需要用户重新登录，获取正确的 token

那么这种方式的缺点就是 token 过期的问题，客户端用户调接口时有可能登入已经过期了，解决的办法就是接口规范了，也就是后台要返回用户登入是否过期的字段，客户端通过这个字段判断是否跳转到登入页面，再发起一次认证请求，获取新的 token。

然而，此种验证方式存在一个安全性问题：当登录接口被劫持时，黑客就获取到了用户密码和 token，后续则可以对该用户做任何事情了。用户只有修改密码才能夺回控制权。

如何优化呢？第一种解决方案是采用 HTTPS。HTTPS 在 HTTP 的基础上添加了 SSL 安全协议，自动对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，安全性可以提高很多。不过，SSL 也不是绝对安全的，也存在被劫持的可能。另外，服务器对 HTTPS 的配置相对有点复杂，还需要到 CA 申请证书，而且一般还是收费的。而且，HTTPS 效率也比较低。一般，只有安全要求比较高的系统才会采用 HTTPS，比如银行。而大部分对安全要求没那么高的 App 还是采用 HTTP 的方式。

我们目前的做法是给每个接口都添加签名。给客户端分配一个密钥，每次请求接口时，将密钥和所有参数组合成源串，根据签名算法生成签名值，发送请求时将签名一起发送给服务器验证。类似的实现可参考 OAuth1.0 的签名算法。这样，黑客不知道密钥，不知道签名算法，就算拦截到登录接口，后续请求也无法成功操作。不过，因为签名算法比较麻烦，而且容易出错，只适合对内的接口。如果你们的接口属于开放的 API，则不太适合这种签名认证的方式了，建议还是使用 OAuth2.0 的认证机制。

我们也给每个端分配一个 appKey，比如 Android、iOS、微信三端，每个端分别分配一个 appKey 和一个密钥。没有传 appKey 的请求将报错，传错了 appKey 的请求也将报错。这样，安全性方面又加多了一层防御，同时也方便对不同端做一些不同的处理策略。

3.2 客户端 token 方式

客户端生成 token 传给服务端校验，一致就通过用户验证。
通过时间戳+用户唯一标识+MD5 加密=token(算法自定义),并且把时间戳传给后台，后台通 过后台系统的时间戳和客户端传过去的时间戳可以规定当前用户在 1 分钟内这次接口可以正常使用，也就是 说，当黑客从路由获取到连接后，只有 1 分钟的时间可以使用这次接口(时间后台可以自

定义)，这样很大程度 上确保了接口的安全性，同时，这种方式也可以有效的解决用户登入过期，也就是使用 session 的方式的不足，但是有个问题，如果客户端和服务端系统时间不一致，就不能这样用了，所以这个时间戳如何获取，也是一个关键点，可能通过接口从后台接口获取，也可以使用其他方式，有好的建议可以一起探讨

3.3 手机验证码登陆

现在越来越多 App 取消了密码登录，而采用手机号+短信验证码的登录方式，我在当前的项目中也采用了这种登录方式。这种登录方式有几种好处：

不需要注册，不需要修改密码，也不需要因为忘记密码而重置密码的操作了；

用户不再需要记住密码了，也不怕密码泄露的问题了；

相对于密码登录其安全性明显提高了。

4 接口数据的设计

接口的数据一般都采用 JSON 格式进行传输，不过，需要注意的是，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

所以，传输的数据类型不能超过这六种数据类型。以前，我们曾经试过传输 Date 类型，它会转为类似于"2016 年 1 月 7 日 09 时 17 分 42 秒 GMT+08:00"这样的字符串，这在转换时会产生问题，不同的解析库解析方式可能不同，有的可能会转乱，有的可能直接异常了。要避免出错，必须做特殊处理，自己手动去做解析。为了根除这种问题，最好的解决方案是用毫秒数表示日期。

另外，以前的项目中还出现过字符串的"true"和"false"，或者字符串的数字，甚至还出现过字符串的"null"，导致解析错误，尤其是"null"，导致 App 奔溃，后来查了好久才查出来是该问题导致的。这都是因为服务端对数据没处理好，导致有些数据转为了字符串。所以，在客户端，也不能完全信任服务端传回的数据都是对的，需要对所有异常情况都做相应处理。

服务器返回的数据结构，一般为：

```
{ code：0, message: "success", data: { key1: value1, key2: value2, ... } }
```

code: 返回码，0 表示成功，非 0 表示各种不同的错误

message: 描述信息，成功时为"success"，错误时则是错误信息

data: 成功时返回的数据，类型为对象或数组

不同错误需要定义不同的返回码，属于客户端的错误和服务端的错误也要区分，比如 1XX 表示客户端的错误，2XX 表示服务端的错误。这里举几个例子：

0：成功

100：请求错误

101：缺少 appKey

102：缺少签名

103：缺少参数

200：服务器出错

201：服务不可用

202：服务器正在重启

错误信息一般有两种用途：一是客户端开发人员调试时看具体是什么错误；二是作为 App 错误提示直接展示给用户看。主要还是作为 App 错误提示，直接展示给用户看的。所以，大部分都是简短的提示信息。

data 字段只在请求成功时才会有数据返回的。数据类型限定为对象或数组，当请求需要的数据为单个对象时则传回对象，当请求需要的数据是列表时，则为某个对象的数组。这里需要注意的就是，不要将 data 传入字符串或数字，即使请求需要的数据只有一个，比如 token，那返回的 data 应该为：

```
// 正确
data: { token: 123456 }
// 错误
data: 123456
```

常用的 HTTP 状态码有：

```
200 OK

201 Created

204 No Content

304 Not Modified

400 Bad Request

401 Unauthorized

403 Forbidden

404 Not Found
```

```
405 Method Not Allowed

410 Gone

415 Unsupported Media Type

422 Unprocessable Entity

429 Too Many Requests

500 Internal Server Error

503 Service Unavailable
```

5 接口版本的设计

接口不可能永远不变，它会随着需求的变化而做出相应的变动。接口的变化一般会有几种：
数据的变化，比如增加了旧版本不支持的数据类型
参数的变化，比如新增了参数
接口的废弃，不再使用该接口了
为了适应这些变化，必须得做接口版本的设计。实现上，一般有两种做法：
1.每个接口有各自的版本，一般为接口添加个 version 的参数。
2.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.demo.com/v2。
大部分情况下会采用第一种方式，当某一个接口有变动时，在这个接口上叠加版本号，并兼容旧版本。App 的新版本开发传参时则将传入新版本的 version。
如果整个接口系统的根基都发生变动的话，比如微博 API，从 OAuth1.0 升级到 OAuth2.0，整个 API 都进行了升级。

有时候，一个接口的变动还会影响到其他接口，但做的时候不一定能发现。因此，最好还要有一套完善的测试机制保证每次接口变更都能测试到所有相关层面。

6 撰写接口文档

文档先行。

好的文档，和好的接口同样重要。接口文档需要被很容易地找到和访问。大部分开发者会在进行接口开发之前，检查并查看接口文档。如果这些接口文档是写在 PDF 文档里，或者需要登录才能查看，那将不仅仅是难于查找，还不利于搜索。

接口文档应该描述完整的 Request/Response Cycle，并附上具体的例子。最好是，这些例子应该是真实可以访问的，比如把链接复制到浏览器里执行，或者用 curl 执行。GitHub 和 Stripe 的接口文档都写得很不错。

一旦你发布了一个 API，那意味着，在没有通知调用者的情况下，你有责任不去破坏该接口的已有功能。如果你在今后修改该接口，需要及时更新接口文档，并且在发布接口的更新之前，及时通知你的接口调用者。