IOS移动APP网络层设计方案

前言

网络层在一个App中也是一个不可缺少的部分，工程师们在网络层能够发挥的空间也比较大。另外，苹果对网络请求部分已经做了很好的封装，业界的AFNetworking也被广泛使用。其它的ASIHttpRequest, MKNetworkKit啥的其实也都还不错，但前者已经弃坑，后者也在弃坑的边缘。在实际的App开发中，Afnetworking已经成为了事实上各大App的标准配置。

网络层在一个App中承载了API调用，用户操作日志记录，甚至是即时通讯等任务。我接触过一些App（开源的和不开源的）的代码，在看到网络层这一块时，尤其是在看到各位架构师各显神通展示了各种技巧，我非常为之感到兴奋。但有的时候，往往也对于其中的一些缺陷感到失望。

关于网络层的设计方案会有很多，需要权衡的地方也会有很多，甚至于争议的地方都会有很多。但无论如何，我都不会对这些问题做出任何逃避，我会在这篇文章中给出我对它们的看法和解决方案，观点绝不中立，不会跟大家打太极。

这篇文章就主要会讲这些方面：

1. 网络层跟业务对接部分的设计
2. 网络层的安全机制实现
3. 网络层的优化方案

网络层跟业务对接部分的设计

在安居客App的架构更新换代的时候，我深深地感觉到网络层跟业务对接部分的设计有多么重要，因此我对它做的最大改变就是针对网络层跟业务

对接部分的改变。网络层跟业务层对接部分设计的好坏，会直接影响到业务工程师实现功能时的心情。

在正式开始讲设计之前，我们要先讨论几个问题：

1. 使用哪种交互模式来跟业务层做对接?
2. 是否有必要将API返回的数据封装成对象然后再交付给业务层?
3. 使用集约化调用方式还是离散型调用方式去调用API?

这些问题讨论完毕之后，我会给出一个完整的设计方案来给大家做参考，设计方案是鱼，讨论的这些问题是渔，我什么都授了，大家各取所需。

> 使用哪种交互模式来跟业务层做对接?

这里其实有两个问题：

1. 以什么方式将数据交付给业务层?
2. 交付什么样的数据给业务层?

以什么方式将数据交付给业务层?

iOS开发领域有很多对象间数据的传递方式，我看到的大多数App在网络层所采用的方案主要集中于这三种：Delegate，Notification，Block。KVO和Target-Action我目前还没有看到有使用的。

目前我知道边锋主要是采用的block，大智慧主要采用的是Notification，安居客早期以Block为主，后面改成了以Delegate为主，阿里没发现有通过Notification来做数据传递的地方（可能有），Delegate、Block以及target-action都有，阿里iOS App网络层的作者说这是为了方便业务层选择自己合适的方法去使用。这里大家都是各显神通，每次我看到这部分的时候，我都喜欢问作者为什么采用这种交互方案，但很少有作者能够说出个条条框框来。

然而在我这边，我的意见是以Delegate为主，Notification为辅。原因如下：

- 尽可能减少跨层数据交流的可能，限制耦合
- 统一回调方法，便于调试和维护

- 在跟业务层对接的部分只采用一种对接手段（在我这儿就是只采用delegate这一个手段）限制灵活性，以此来交换应用的可维护性

> 尽可能减少跨层数据交流的可能，限制耦合

什么叫跨层数据交流？就是某一层（或模块）跟另外的与之没有直接对接关系的层（或模块）产生了数据交换。为什么这种情况不好？严格来说应该是大部分情况都不好，有的时候跨层数据交流确实也是一种需求。之所以说不好的地方在于，它会导致代码混乱，破坏模块的封装性。我们在做分层架构的目的其中之一就在于下层对上层有一次抽象，让上层可以不必关心下层细节而执行自己的业务。

所以，如果下层细节被跨层暴露，一方面你很容易因此失去邻层对这个暴露细节的保护；另一方面，你又不可能不去处理这个细节，所以处理细节的相关代码就会散落各地，最终难以维护。

说得具象一点就是，我们考虑这样一种情况：A<-B<-C。当C有什么事件，通过某种方式告知B，然后B执行相应的逻辑。一旦告知方式不合理，让A有了跨层知道C的事件的可能，你 就很难保证A层业务工程师在将来不会对这个细节作处理。一旦业务工程师在A层产生处理操作，有可能是补充逻辑，也有可能是执行业务，那么这个细节的相关处理代码就会有一部分散落在A层。然而前者是不应该散落在A层的，后者有可能是需求。另外，因为B层是对A层抽象的，执行补充逻辑的时候，有可能和B层针对这个事件的处理逻辑产生冲突，这是我们很不希望看到的。

那么什么情况跨层数据交流会成为需求？在网络层这边，信号从2G变成3G变成4G变成Wi-Fi，这个是跨层数据交流的其中一个需求。不过其他的跨层数据交流需求我暂时也想不到了，哈哈，应该也就这一个吧。

严格来说，使用Notification来进行网络层和业务层之间数据的交换，并不代表这一定就是跨层数据交流，但是使用Notification给跨层数据交流开了一道口子，因为Notification的影响面不可控制，只要存在实例就存在被影响的可能。另外，这也会导致谁都不能保证相关处理代码就在唯一的那个地方，进而带来维护灾难。作为架构师，在这里给业务工程师限制其操作的灵活性是必要的。另外，Notification也支持一对多的情况，这也给代码

散落提供了条件。同时，Notification所对应的响应方法很难在编译层面作限制，不同的业务工程师会给他取不同的名字，这也会给代码的可维护性带来灾难。

手机淘宝架构组的侠武同学曾经给我分享过一个问题，在这里我也分享给大家：曾经有一个工程师在监听Notification之后，没有写释放监听的代码，当然，找到这个原因又是很漫长的一段故事，现在找到原因了，然而监听这个Notification的对象有那么多，不知道具体是哪个Notificaiton，也不知道那个没释放监听的对象是谁。后来折腾了很久大家都没办法的时候，有一个经验丰富的工程师提出用hook（Method Swizzling）的方式，最终找到了那个没释放监听的对象，bug修复了。

我分享这个问题的目的并不是想强调Notification多么多么不好，Notification本身就是一种设计模式，在属于他的问题领域内，Notification是非常好的一种解决方案。但我想强调的是，对于网络层这个问题领域内来看，架构师首先一定要限制代码的影响范围，在能用影响范围小的方案的时候就尽量采用这种小的方案，否则将来要是有什么奇怪需求或者出了什么小问题，维护起来就非常麻烦。因此Notification这个方案不能作为首选方案，只能作为备选。

那么Notification也不是完全不能使用，当需求要求跨层时，我们就可以使用Notification，比如前面提到的网络条件切换，而且这个需求也是需要满足一对多的。

所以，为了符合前面所说的这些要求，使用Delegate能够很好地避免跨层访问，同时限制了响应代码的形式，相比Notification而言有更好的可维护性。

> 然后我们顺便来说说为什么尽量不要用block。

- block很难追踪，难以维护

我们在调试的时候经常会单步追踪到某一个地方之后，发现尼玛这里有个block，如果想知道这个block里面都做了些什么事情，这时候就比较蛋疼了。

```
- (void)someFunctionWithBlock:(SomeBlock *)block
{
    ... ...

 -> block();  //当你单步走到这儿的时候，要想知道block里面都做了哪些事情的话，就很麻烦。

    ... ...
}
```

- block会延长相关对象的生命周期

block会给内部所有的对象引用计数加一，这一方面会带来潜在的retain cycle，不过我们可以通过Weak Self的手段解决。另一方面比较重要就是，它会延长对象的生命周期。

在网络回调中使用block，是block导致对象生命周期被延长的其中一个场合，当ViewController从window中卸下时，如果尚有请求带着block在外面飞，然后block里面引用了ViewController（这种场合非常常见），那么ViewController是不能被及时回收的，即便你已经取消了请求，那也还是必须得等到请求着陆之后才能被回收。

然而使用delegate就不会有这样的问题，delegate是弱引用，哪怕请求仍然在外面飞，，ViewController还是能够及时被回收的，回收之后指针自动被置为了nil，无伤大雅。

- block在离散型场景下不符合使用的规范

block和delegate乍看上去在作用上是很相似，但是关于它们的选型有一条严格的规范：当回调之后要做的任务在每次回调时都是一致的情况下，选择delegate，在回调之后要做的任务在每次回调时无法保证一致，选择block。在离散型调用的场景下，每一次回调都是能够保证任务一致的，因此适用delegate。这也是苹果原生的网络调用也采用delegate的原因，因为苹果也是基于离散模型去设计网络调用的，而且本文即将要介绍的网络层架构也是基于离散型调用的思路去设计的。

在集约型调用的场景下，使用block是合理的，因为每次请求的类型都不一样，那么自然回调要做的任务也都会不一样，因此只能采用block。

AFNetworking就是属于集约型调用，因此它采用了block来做回调。

就我所知，目前大部分公司的App网络层都是集约型调用，因此广泛采取了block回调。但是在App的网络层架构设计中直接采用集约型调用来为业务服务的思路是有问题的，因此在迁移到离散型调用时，一定要注意这一点，记得迁回delegate回调。关于离散型和集约型调用的介绍和如何选型，我在后面的集约型API调用方式和离散型API调用方式的选择？小节中有详细的介绍。

所以平时尽量不要滥用block，尤其是在网络层这里。

> 统一回调方法，便于调试和维护

前面讲的是跨层问题，区分了Delegate和Notification，顺带谈了一下Block。然后现在谈到的这个情况，就是另一个采用Block方案不是很合适的情况。首先，Block本身无好坏对错之分，只有合适不合适。在这一节要讲的情况里，Block无法做到回调方法的统一，调试和维护的时候也很难在调用栈上显示出来，找的时候会很蛋疼。

在网络请求和网络层接受请求的地方时，使用Block没问题。但是在获得数据交给业务方时，最好还是通过Delegate去通知到业务方。因为Block所包含的回调代码跟调用逻辑放在同一个地方，会导致那部分代码变得很长，因为这里面包括了调用前和调用后的逻辑。从另一个角度说，这在一定程度上违背了single function, single task的原则，在需要调用API的地方，就只要写API调用相关的代码，在回调的地方，写回调的代码。

然后我看到大部分App里，当业务工程师写代码写到这边的时候，也意识到了这个问题。因此他们会在block里面写个一句话的方法接收参数，然后做转发，然后就可以把这个方法放在其他地方了，绕过了Block的回调着陆点不统一的情况。比如这样：

```
[API callApiWithParam:param successed:^(Response *response){
    [self successedWithResponse:response];
} failed:^(Request *request, NSError *error){
    [self failedWithRequest:request error:error];
}];
```

这实质上跟使用Delegate的手段没有什么区别，只是绕了一下，不过还是没有解决统一回调方法的问题，因为block里面写的方法名字可能在不同的ViewController对象中都会不一样，毕竟业务工程师也是很多人，各人有各人的想法。所以架构师在这边不要贪图方便，还是使用delegate的手段吧，业务工程师那边就能不用那么绕了。Block是目前大部分第三方网络库都采用的方式，因为在发送请求的那一部分，使用Block能够比较简洁，因此在请求那一层是没有问题的，只是在交换数据之后，还是转变成delegate比较好，比如AFNetworking里面：

```
    [AFNetworkingAPI callApiWithParam:self.param successed:^(Response
*response){
        if ([self.delegate
respondsToSelector:@selector(successWithResponse:)]) {
            [self.delegate successedWithResponse:response];
        }
    } failed:^(Request *request, NSError *error){
        if ([self.delegate
respondsToSelector:@selector(failedWithResponse:)]) {
            [self failedWithRequest:request error:error];
        }
    }];
```

这样在业务方这边回调函数就能够比较统一，便于维护。

综上，对于以什么方式将数据交付给业务层？这个问题的回答是这样：

尽可能通过Delegate的回调方式交付数据，这样可以避免不必要的跨层访问。当出现跨层访问的需求时（比如信号类型切换），通过Notification的方式交付数据。正常情况下应该是避免使用Block的。

交付什么样的数据给业务层?

我见过非常多的App的网络层在拿到JSON数据之后，会将数据转变成对应的对象原型。注意，我这里指的不是NSDictionary，而是类似Item这样的对象。这种做法是能够提高后续操作代码的可读性的。在比较直觉的思路里面，是需要这部分转化过程的，但这部分转化过程的成本是很大的，主要成本在于：

1. 数组内容的转化成本较高：数组里面每项都要转化成Item对象，如果Item对象中还有类似数组，就很头疼。
2. 转化之后的数据在大部分情况是不能直接被展示的，为了能够被展示，还需要第二次转化。
3. 只有在API返回的数据高度标准化时，这些对象原型（Item）的可复用程度才高，否则容易出现类型爆炸，提高维护成本。
4. 调试时通过对象原型查看数据内容不如直接通过NSDictionary/NSArray直观。
5. 同一API的数据被不同View展示时，难以控制数据转化的代码，它们有可能会散落在任何需要的地方。

其实我们的理想情况是希望API的数据下发之后就能够直接被View所展示。首先要说的是，这种情况非常少。另外，这种做法使得View和API联系紧密，也是我们不希望发生的。

在设计安居客的网络层数据交付这部分时，我添加了reformer（名字而已，叫什么都好）这个对象用于封装数据转化的逻辑，这个对象是一个独立对象，事实上，它是作为Adaptor模式存在的。我们可以这么理解：想象一下我们洗澡时候使用的莲蓬头，水管里出来的水是API下发的原始数据。reformer就是莲蓬头上的不同水流挡板，需要什么模式，就拨到什么模式。

在实际使用时，代码观感是这样的：

先定义一个protocol：

```
@protocol ReformerProtocol <NSObject>
- (NSDictionary)reformDataWithManager:(APIManager *)manager;
@end
```

在Controller里是这样：

```
@property (nonatomic, strong) id<ReformerProtocol> XXXReformer;
@property (nonatomic, strong) id<ReformerProtocol> YYYReformer;

#pragma mark - APIManagerDelegate
- (void)apiManagerDidSuccess:(APIManager *)manager
{
    NSDictionary *reformedXXXData = [manager
```

```
fetchDataWithReformer:self.XXXReformer];
    [self.XXXView configWithData:reformedXXXData];

    NSDictionary *reformedYYYData = [manager
fetchDataWithReformer:self.YYYReformer];
    [self.YYYView configWithData:reformedYYYData];
}
```

在APIManager里面，fetchDataWithReformer是这样：

```
- (NSDictionary)fetchDataWithReformer:(id<ReformerProtocol>)reformer
{
    if (reformer == nil) {
        return self.rawData;
    } else {
        return [reformer reformDataWithManager:self];
    }
}
```

- 要点1：reformer是一个符合ReformerProtocol的对象，它提供了通用的方法供Manager使用。

- 要点2：API的原始数据（JSON对象）由Manager实例保管，reformer方法里面取Manager的原始数据(manager.rawData)做转换，然后交付出去。莲蓬头的水管部分是Manager，负责提供原始水流（数据流），reformer就是不同的模式，换什么reformer就能出来什么水流。

- 要点3：例子中举的场景是一个API数据被多个View使用的情况，体现了reformer的一个特点：可以根据需要改变同一数据来源的展示方式。比如API数据展示的是“附近的小区”，那么这个数据可以被列表（XXXView）和地图（YYYView）共用，不同的view使用的数据的转化方式不一样，这就通过不同的reformer解决了。

- 要点4：在一个view用来同一展示不同API数据的情况，reformer是绝佳利器。比如安居客的列表view的数据来源可能有三个：二手房列表API，租房列表API，新房列表API。这些API返回来的数据的value可能一致，但是key都是不一致的。这时候就可以通过同一个reformer来做数据的标准化输出，这样就使得view代码复用成为可能。这体现了reformer另外一个特点：同一个reformer出来的数据是高度标准化

的。形象点说就是：只要莲蓬头不换，哪怕水管的水变成海水或者污水了，也依旧能够输出符合洗澡要求的淡水水流。举个例子：

```
- (void)apiManagerDidSuccess:(APIManager *)manager
{
    // 这个回调方法有可能是来自二手房列表APIManager的回调，也有可能是租房，也有可能是新房。但是在Controller层面我们不需要对它做额外区分，只要是同一个reformer出来的数据，我们就能保证是一定能被self.XXXView使用的。这样的保证由reformer的实现者来提供。
    NSDictionary *reformedXXXData = [manager fetchDataWithReformer:self.XXXReformer];
    [self.XXXView configWithData:reformedXXXData];
}
```

- 要点5：有没有发现，使用reformer之后，Controller的代码简洁了很多？而且，数据原型在这种情况下就没有必要存在了，随之而来的成本也就被我们绕过了。

reformer本质上就是一个符合某个protocol的对象，在controller需要从api manager中获得数据的时候，顺便把reformer传进去，于是就能获得经过reformer重新洗过的数据，然后就可以直接使用了。

更抽象地说，reformer其实是对数据转化逻辑的一个封装。在controller从manager中取数据之后，并且把数据交给view之前，这期间或多或少都是要做一次数据转化的，有的时候不同的view，对应的转化逻辑还不一样，但是展示的数据是一样的。而且往往这一部分代码都非常复杂，且跟业务强相关，直接上代码，将来就会很难维护。所以我们可以考虑采用不同的reformer封装不同的转化逻辑，然后让controller根据需要选择一个合适的reformer装上，就像洗澡的莲蓬头，需要什么样的水流（数据的表现形式）就换什么样的头，然而水（数据）都是一样的。这种做法能够大大提高代码的可维护性，以及减少ViewController的体积。

总结一下，reformer事实上是把转化的代码封装之后再从主体业务中拆分了出来，拆分出来之后不光降低了原有业务的复杂度，更重要的是，它提高了数据交付的灵活性。另外，由于Controller负责调度Manager和View，因此它是知道Manager和View之间的关系的，Controller知道了这个关系之后，就有了充要条件来为不同的View选择不同的Reformer，并用这个Reformer去改造Mananger的数据，然后ViewController获得了经过

reformer处理过的数据之后，就可以直接交付给view去使用。Controller因此得到瘦身，负责业务数据转化的这部分代码也不用写在Controller里面，提高了可维护性。

所以reformer机制能够带来以下好处：

- 好处1：绕开了API数据原型的转换，避免了相关成本。
- 好处2：在处理单View对多API，以及在单API对多View的情况时，reformer提供了非常优雅的手段来响应这种需求，隔离了转化逻辑和主体业务逻辑，避免了维护灾难。
- 好处3：转化逻辑集中，且将转化次数转为只有一次。使用数据原型的转化逻辑至少有两次，第一次是把JSON映射成对应的原型，第二次是把原型转变成能被View处理的数据。reformer一步到位。另外，转化逻辑在reformer里面，将来如果API数据有变，就只要去找到对应reformer然后改掉就好了。
- 好处4：Controller因此可以省去非常多的代码，降低了代码复杂度，同时提高了灵活性，任何时候切换reformer而不必切换业务逻辑就可以应对不同View对数据的需要。
- 好处5：业务数据和业务有了适当的隔离。这么做的话，将来如果业务逻辑有修改，换一个reformer就好了。如果其他业务也有相同的数据转化逻辑，其他业务直接拿这个reformer就可以用了，不用重写。另外，如果controller有修改（比如UI交互方式改变），可以放心换controller，完全不用担心业务数据的处理。

> 在不使用特定对象表征数据的情况下，如何保持数据可读性?

不使用对象来表征数据的时候，事实上就是使用NSDictionary的时候。事实上，这个问题就是，如何在NSDictionary表征数据的情况下保持良好的可读性?

苹果已经给出了非常好的做法，用固定字符串做key，比如你在接收到KeyBoardWillShow的Notification时，带了一个userInfo，他的key就都是

类似UIKeyboardAnimationCurveUserInfoKey这样的，所以我们采用这样的方案来维持可读性。下面我举一个例子：

PropertyListReformerKeys.h

```
extern NSString * const kPropertyListDataKeyID;
extern NSString * const kPropertyListDataKeyName;
extern NSString * const kPropertyListDataKeyTitle;
extern NSString * const kPropertyListDataKeyImage;
```

PropertyListReformer.h

```
#import "PropertyListReformerKeys.h"

... ...
```

PropertyListReformer.m

```
NSString * const kPropertyListDataKeyID = @"kPropertyListDataKeyID";
NSString * const kPropertyListDataKeyName = @"kPropertyListDataKeyName";
NSString * const kPropertyListDataKeyTitle = @"kPropertyListDataKeyTitle";
NSString * const kPropertyListDataKeyImage = @"kPropertyListDataKeyImage";

- (NSDictionary *)reformData:(NSDictionary *)originData fromManager:
(APIManager *)manager
{
    ... ...
    ... ...

    NSDictionary *resultData = nil;

    if ([manager isKindOfClass:[ZuFangListAPIManager class]]) {
        resultData = @{
            kPropertyListDataKeyID:originData[@"id"],
            kPropertyListDataKeyName:originData[@"name"],
            kPropertyListDataKeyTitle:originData[@"title"],
            kPropertyListDataKeyImage:[UIImage
imageWithUrlString:originData[@"imageUrl"]]
        };
    }

    if ([manager isKindOfClass:[XinFangListAPIManager class]]) {
        resultData = @{
            kPropertyListDataKeyID:originData[@"xinfang_id"],
```

```
            kPropertyListDataKeyName:originData[@"xinfang_name"],
            kPropertyListDataKeyTitle:originData[@"xinfang_title"],
            kPropertyListDataKeyImage:[UIImage
imageWithUrlString:originData[@"xinfang_imageUrl"]]
        };
    }

    if ([manager isKindOfClass:[ErShouFangListAPIManager class]]) {
        resultData = @{
            kPropertyListDataKeyID:originData[@"esf_id"],
            kPropertyListDataKeyName:originData[@"esf_name"],
            kPropertyListDataKeyTitle:originData[@"esf_title"],
            kPropertyListDataKeyImage:[UIImage
imageWithUrlString:originData[@"esf_imageUrl"]]
        };
    }

    return resultData;
}
```

PropertListCell.m

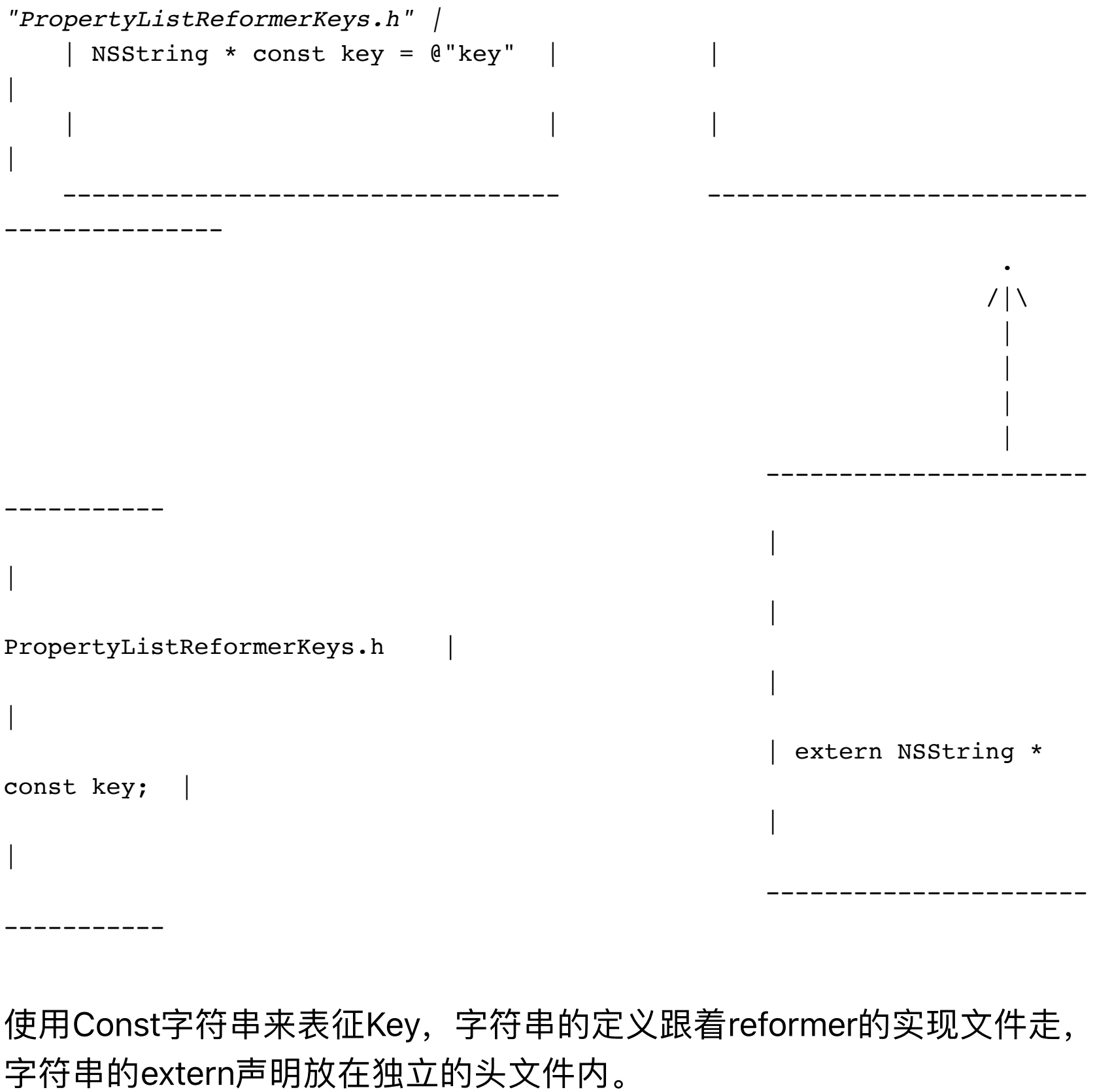
```
#import "PropertyListReformerKeys.h"

- (void)configWithData:(NSDictionary *)data
{
    self.imageView.image = data[kPropertyListDataKeyImage];
    self.idLabel.text = data[kPropertyListDataKeyID];
    self.nameLabel.text = data[kPropertyListDataKeyName];
    self.titleLabel.text = data[kPropertyListDataKeyTitle];
}
```

这一大段代码看下来，我如果不说一下要点，那基本上就白写了哈：

我们先看一下结构：

```
    ---------------------------------          ------------------------
--------------
    |                                 |          |
|
    | PropertyListReformer.m          |          | PropertyListReformer.h
|
    |                                 |          |
|
    | #import PropertyListReformer.h | <------- |  #import
```



使用Const字符串来表征Key，字符串的定义跟着reformer的实现文件走，字符串的extern声明放在独立的头文件内。

这样reformer生成的数据的key都使用Const字符串来表示，然后每次别的地方需要使用相关数据的时候，把PropertyListReformerKeys.h这个头文件import进去就好了。

另外要注意的一点是，如果一个OriginData可能会被多个Reformer去处理的话，Key的命名规范需要能够表征出其对应的reformer名字。如果reformer是`PropertyListReformer`，那么Key的名字就是`PropertyListKeyXXXX`。

这么做的好处就是，将来迁移的时候相当方便，只要扔头文件就可以了，只扔头文件是不会导致拔出萝卜带出泥的情况的。而且也避免了自定义对象带来的额外代码体积。

另外，关于交付的NSDictionary，其实具体还是看view的需求，reformer的设计初衷是：通过reformer转化出来的可以直接是View，或者是view直接可以使用的对象（包括NSDictionary）。比如地图标点列表API的数据，通过reformer转化之后就可以直接变成MKAnnotation，然后MKMapView就可以直接使用了。这里说的只是当你的需求是交付NSDictionary时，如何保证可读性的情况，再强调一下哈，reformer交付的是view直接可以使用的对象，交付出去的可以是NSDictionary，也可以是UIView，跟DataSource结合之后交付的甚至可以是UITableViewCell/UICollectionViewCell。不要被NSDictionary或所谓的转化成model再交付的思想局限。

综上，我对交付什么样的数据给业务层？这个问题的回答就是这样：

对于业务层而言，由Controller根据View和APIManager之间的关系，选择合适的reformer将View可以直接使用的数据（甚至reformer可以用来直接生成view）转化好之后交付给View。对于网络层而言，只需要保持住原始数据即可，不需要主动转化成数据原型。然后数据采用NSDictionary加Const字符串key来表征，避免了使用对象来表征带来的迁移困难，同时不失去可读性。

> 集约型API调用方式和离散型API调用方式的选择？

集约型API调用其实就是所有API的调用只有一个类，然后这个类接收API名字，API参数，以及回调着陆点（可以是target-action，或者block，或者delegate等各种模式的着陆点）作为参数。然后执行类似startRequest这样的方法，它就会去根据这些参数起飞去调用API了，然后获得API数据之后再根据指定的着陆点去着陆。比如这样：

集约型API调用方式：

```
[APIRequest startRequestWithApiName:@"itemList.v1" params:params
success:@selector(success:) fail:@selector(fail:) target:self];
```

离散型API调用是这样的，一个API对应于一个APIManager，然后这个APIManager只需要提供参数就能起飞，API名字、着陆方式都已经集成入

APIManager中。比如这样：

离散型API调用方式：

```
@property (nonatomic, strong) ItemListAPIManager *itemListAPIManager;

// getter
- (ItemListAPIManager *)itemListAPIManager
{
    if (_itemListAPIManager == nil) {
        _itemListAPIManager = [[ItemListAPIManager alloc] init];
        _itemListAPIManager.delegate = self;
    }

    return _itemListAPIManager;
}

// 使用的时候就这么写：
[self.itemListAPIManager loadDataWithParams:params];
```

集约型API调用和离散型API调用这两者实现方案不是互斥的，单看下层，大家都是集约型。因为发起一个API请求之后，除去业务相关的部分（比如参数和API名字等），剩下的都是要统一处理的：加密，URL拼接，API请求的起飞和着陆，这些处理如果不用集约化的方式来实现，作者非癫即痴。然而对于整个网络层来说，尤其是业务方使用的那部分，我倾向于提供离散型的API调用方式，并不建议在业务层的代码直接使用集约型的API调用方式。原因如下：

- 原因1：当前请求正在外面飞着的时候，根据不同的业务需求存在两种不同的请求起飞策略：一个是取消新发起的请求，等待外面飞着的请求着陆。另一个是取消外面飞着的请求，让新发起的请求起飞。集约化的API调用方式如果要满足这样的需求，那么每次要调用的时候都要多写一部分判断和取消的代码，手段就做不到很干净。

前者的业务场景举个例子就是刷新页面的请求，刷新详情，刷新列表等。后者的业务场景举个例子是列表多维度筛选，比如你先筛选了商品类型，然后筛选了价格区间。当然，后者的情况不一定每次筛选都要调用API，我们先假设这种筛选每次都必须要通过调用API才能获得数据。

如果是离散型的API调用，在编写不同的APIManager时候就可以针对不同

的API设置不同的起飞策略，在实际使用的时候，就可以不必关心起飞策略了，因为APIMananger里面已经写好了。

- 原因2：便于针对某个API请求来进行AOP。在集约型的API调用方式下，如果要针对某个API请求的起飞和着陆过程进行AOP，这代码得写成什么样。。。噢，尼玛这画面太美别说看了，我都不敢想。

- 原因3：当API请求的着陆点消失时，离散型的API调用方式能够更加透明地处理这种情况。

当一个页面的请求正在天上飞的时候，用户等了好久不耐烦了，小手点了个back，然后ViewController被pop被回收。此时请求的着陆点就没了。这是很危险的情况，着陆点要是没了，就很容易crash的。一般来说处理这个情况都是在dealloc的时候取消当前页面所有的请求。如果是集约型的API调用，这个代码就要写到ViewController的dealloc里面，但如果是离散型的API调用，这个代码写到APIManager里面就可以了，然后随着ViewController的回收进程，APIManager也会被跟着回收，这部分代码就得到了调用的机会。这样业务方在使用的时候就可以不必关心着陆点消失的情况了，从而更加关注业务。

- 原因4：离散型的API调用方式能够最大程度地给业务方提供灵活性，比如reformer机制就是基于离散型的API调用方式的。另外，如果是针对提供翻页机制的API，APIManager就能简单地提供`loadNextPage`方法去加载下一页，页码的管理就不用业务方去管理了。还有就是，如果要针对业务请求参数进行验证，比如用户填写注册信息，在离散型的APIManager里面实现就会非常轻松。

综上，关于集约型的API调用和离散型的API调用，我倾向于这样：对外提供一个BaseAPIManager来给业务方做派生，在BaseManager里面采用集约化的手段组装请求，放飞请求，然而业务方调用API的时候，则是以离散的API调用方式来调用。如果你的App只提供了集约化的方式，而没有离散方式的通道，那么我建议你再封装一层，便于业务方使用离散的API调用方式来放飞请求。

> 怎么做APIManager的继承?

如果要做成离散型的API调用，那么使用继承是逃不掉的。BaseAPIManager里面负责集约化的部分，外部派生的XXXAPIManager负责离散的部分，对于BaseAPIManager来说，离散的部分有一些是必要的，比如API名字等，而我们派生的目的，也是为了提供这些数据。

我在这篇文章里面列举了种种继承的坏处，呼吁大家尽量不要使用继承。但是现在到了不得不用继承的时候，所以我得提醒一下大家别把继承用坏了。

在APIManager的情况下，我们最直觉的思路是BaseAPIManager提供一些空方法来给子类做重载，比如`apiMethodName`这样的函数，然而我的建议是，不要这么做。我们可以用IOP的方式来限制派生类的重载。

大概就是长这样：

BaseAPIManager的init方法里这么写：

```
// 注意是weak。
@property (nonatomic, weak) id<APIManager> child;

- (instancetype)init
{
    self = [super init];
    if ([self conformsToProtocol:@protocol(APIManager)]) {
        self.child = (id<APIManager>)self;
    } else {
        // 不遵守这个protocol的就让他crash，防止派生类乱来。
        NSAssert(NO, "子类必须要实现APIManager这个protocol。");
    }
    return self;
}
```

protocol这么写，把原本要重载的函数都定义在这个protocol里面，就不用在父类里面写空方法了：

```
@protocol APIManager <NSObject>

@required
- (NSString *)apiMethodName;
...

@end
```

然后在父类里面如果要使用的话，就这么写：

```
[self requestWithAPIName:[self.child apiMethodName] ......];
```

简单说就是在init的时候检查自己是否符合预先设计的子类的protocol，这就要求所有子类必须遵守这个protocol，所有针对父类的重载、覆盖也都以这个protocol为准，protocol以外的方法不允许重载、覆盖。而在父类的代码里，可以不必遵守这个protocol，保持了未来维护的灵活性。

这么做的好处就是避免了父类写空方法，同时也给子类带上了紧箍咒：要想当我的孩子，就要遵守这些规矩，不能乱来。业务方在实现子类的时候，就可以根据protocol中的方法去一一实现，然后约定就比较好做了：不允许重载父类方法，只允许选择实现或不实现protocol中的方法。

关于这个的具体的论述在这篇文章里面有，感兴趣的话可以看看。

> 网络层与业务层对接部分的小总结

这一节主要是讲了以下这些点：

1. 使用delegate来做数据对接，仅在必要时采用Notification来做跨层访问
2. 交付NSDictionary给业务层，使用Const字符串作为Key来保持可读性
3. 提供reformer机制来处理网络层反馈的数据，这个机制很重要，好处极多
4. 网络层上部分使用离散型设计，下部分使用集约型设计
5. 设计合理的继承机制，让派生出来的APIManager受到限制，避免混乱
6. 应该不止这5点...

网络层的安全机制

判断API的调用请求是来自于经过授权的APP

使用这个机制的目的主要有两点：

1. 确保API的调用者是来自你自己的APP，防止竞争对手爬你的API
2. 如果你对外提供了需要注册才能使用的API平台，那么你需要有这个

机制来识别是否是注册用户调用了你的API

> 解决方案：设计签名

要达到第一个目的其实很简单，服务端需要给你一个密钥，每次调用API时，你使用这个密钥再加上API名字和API请求参数算一个hash出来，然后请求的时候带上这个hash。服务端收到请求之后，按照同样的密钥同样的算法也算一个hash出来，然后跟请求带来的hash做一个比较，如果一致，那么就表示这个API的调用者确实是你的APP。为了不让别人也获取到这个密钥，你最好不要把这个密钥存储在本地，直接写死在代码里面就好了。另外适当增加一下求Hash的算法的复杂度，那就是各种Hash算法（比如MD5）加点盐，再回炉跑一次Hash啥的。这样就能解决第一个目的了：确保你的API是来自于你自己的App。

一般情况下大部分公司不会出现需要满足第二种情况的需求，除非公司开发了自己的API平台给第三方使用。这个需求跟上面的需求有一点不同：符合授权的API请求者不只是一个。所以在这种情况下，需要的安全机制会更加复杂一点。

这里有一个较容易实现的方案：客户端调用API的时候，把自己的密钥通过一个可逆的加密算法加密后连着请求和加密之后的Hash一起送上去。当然，这个可逆的加密算法肯定是放在在调用API的SDK里面，编译好的。然后服务端拿到加密后的密钥和加密的Hash之后，解码得到原始密钥，然后再用它去算Hash，最后再进行比对。

保证传输数据的安全

使用这个机制的主要目的有两点：

1. 防止中间人攻击，比如说运营商很喜欢往用户的Http请求里面塞广告...
2. SPDY依赖于HTTPS，而且是未来HTTP/2的基础，他们能够提高你APP在网络层整体的性能。

> 解决方案：HTTPS

目前使用HTTPS的主要目的在于防止运营商往你的Response Data里面加

广告啥的（中间人攻击），面对的威胁范围更广。从2011年开始，国外业界就已经提倡所有的请求（不光是API，还有网站）都走HTTPS，国内差不多晚了两年（2013年左右）才开始提倡这事，天猫是这两个月才开始做HTTPS的全APP迁移。

关于速度，HTTPS肯定是比HTTP慢的，毕竟多了一次握手，但挂上SPDY之后，有了链接复用，这方面的性能就有了较大提升。这里的性能提升并不是说一个请求原来要500ms能完成，然后现在只要300ms，这是不对的。所谓整体性能是基于大量请求去讨论的：同样的请求量（假设100个）在短期发生时，挂上SPDY之后完成这些任务所要花的时间比不用SPDY要少。SPDY还有Header压缩的功能，不过因为一个API请求本身已经比较小了，压缩数据量所带来的性能提升不会特别明显，所以就单个请求来看，性能的提升是比较小的。不过这是下一节要讨论的事儿了，这儿只是顺带说一下。

安全机制小总结

这一节说了两种安全机制，一般来说第一种是标配，第二种属于可选配置。不过随着我国互联网基础设施的完善，移动设备性能的提高，以及优化技术的提高，第二种配置的缺点（速度慢）正在越来越微不足道，因此HTTPS也会成为不久之后的未来App的网络层安全机制标配。各位架构师们，如果你的App还没有挂HTTPS，现在就已经可以开始着手这件事情了。

网络层的优化方案

网络层的优化手段主要从以下三方面考虑：

1. 针对链接建立环节的优化
2. 针对链接传输数据量的优化
3. 针对链接复用的优化

这三方面是所有优化手段的内容，各种五花八门的优化手段基本上都不会逃脱这三方面，下面我就会分别针对这三方面讲一下各自对应的优化手段。

1. 针对链接建立环节的优化

在API发起请求建立链接的环节，大致会分这些步骤：

1. 发起请求
2. DNS域名解析得到IP
3. 根据IP进行三次握手（HTTPS四次握手），链接建立成功

其实第三步的优化手段跟第二步的优化手段是一致的，我会在讲第二步的时候一起讲掉。

> 1.1 针对发起请求的优化手段

其实要解决的问题就是网络层该不该为此API调用发起请求。

- 1.1.1 使用缓存手段减少请求的发起次数

对于大部分API调用请求来说，有些API请求所带来的数据的时效性是比较长的，比如商品详情，比如App皮肤等。那么我们就可以针对这些数据做本地缓存，这样下次请求这些数据的时候就可以不必再发起新的请求。

一般是把API名字和参数拼成一个字符串然后取MD5作为key，存储对应返回的数据。这样下次有同样请求的时候就可以直接读取这里面的数据。关于这里有一个缓存策略的问题需要讨论：什么时候清理缓存？要么就是根据超时时间限制进行清理，要么就是根据缓存数据大小进行清理。这个策略的选择要根据具体App的操作日志来决定。

比如安居客App，日志数据记录显示用户平均使用时长不到3分钟，但是用户查看房源详情的次数比较多，而房源详情数据量较大。那么这个时候，就适合根据使用时长来做缓存，我当时给安居客设置的缓存超时时间就是3分钟，这样能够保证这个缓存能够在大部分用户使用时间产生作用。嗯，极端情况下做什么缓存手段不考虑，只要能够服务好80%的用户就可以了，而且针对极端情况采用的优化手段对大部分普通用户而言是不必要的，做了反而会对他们有影响。

再比如网络图片缓存，数据量基本上都特别大，这种就比较适合针对缓存大小来清理缓存的策略。

另外，之前的缓存的前提都是基于内存的。我们也可以把需要清理的缓存存储在硬盘上（APP的本地存储，我就先用硬盘来表示了，虽然很少有手机硬盘的说法，哈哈），比如前面提到的图片缓存，因为图片很有可能在很长时间之后，再被显示的，那么原本需要被清理的图片缓存，我们就可以考虑存到硬盘上去。当下次再有显示网络图片的需求的时候，我们可以先从内存中找，内存找不到那就从硬盘上找，这都找不到，那就发起请求吧。

当然，有些时效性非常短的API数据，就不能使用这个方法了，比如用户的资金数据，那就需要每次都调用了。

- 1.1.2 使用策略来减少请求的发起次数

这个我在前面提到过，就是针对重复请求的发起和取消，是有对应的请求策略的。我们先说取消策略。

如果是界面刷新请求这种，而且存在重复请求的情况（下拉刷新时，在请求着陆之前用户不断执行下拉操作），那么这个时候，后面重复操作导致的API请求就可以不必发送了。

如果是条件筛选这种，那就取消前面已经发送的请求。虽然很有可能这个请求已经被执行了，那么取消所带来的性能提升就基本没有了。但如果这个请求还在队列中待执行的话，那么对应的这次链接就可以省掉了。

以上是一种，另外一种情况就是请求策略：类似用户操作日志的请求策略。

用户操作会触发操作日志上报Server，这种请求特别频繁，但是是暗地里进行的，不需要用户对此有所感知。所以也没必要操作一次就发起一次的请求。在这里就可以采用这样的策略：在本地记录用户的操作记录，当记录满30条的时候发起一次请求将操作记录上传到服务器。然后每次App启动的时候，上传一次上次遗留下来没上传的操作记录。这样能够有效降低用户设备的耗电量，同时提升网络层的性能。

> 小总结

针对建立连接这部分的优化就是这样的原则：能不发请求的就尽量不发请

求，必须要发请求时，能合并请求的就尽量合并请求。然而，任何优化手段都是有前提的，而且也不能保证对所有需求都能起作用，有些API请求就是不符合这些优化手段前提的，那就老老实实发请求吧。不过这类API请求所占比例一般不大，大部分的请求都或多或少符合优化条件，所以针对发送请求的优化手段还是值得做的。

> 1.2 & 1.3 针对DNS域名解析做的优化，以及建立链接的优化

其实在整个DNS链路上也是有DNS缓存的，理论上也是能够提高速度的。这个链路上的DNS缓存在PC用户上效果明显，因为PC用户的DNS链路相对稳定，信号源不会变来变去。但是在移动设备的用户这边，链路上的DNS缓存所带来的性能提升就不太明显了。因为移动设备的实际使用场景比较复杂，网络信号源会经常变换，信号源每变换一次，对应的DNS解析链路就会变换一次，那么原链路上的DNS缓存就不起作用了。而且信号源变换的情况特别特别频繁，所以对于移动设备用户来说，链路的DNS缓存我们基本上可以默认为没有。所以大部分时间是手机系统自带的本地DNS缓存在起作用，但是一般来说，移动设备上网的需求也特别频繁，专门为我们这个App所做的DNS缓存很有可能会被别的DNS缓存给挤出去被清理掉，这种情况是特别多的，用户看一会儿知乎刷一下微博查一下地图逛一逛点评再聊个Q，回来之后很有可能属于你自己的App的本地DNS缓存就没了。这还没完，这里还有一个只有在中国特色社会主义的互联网环境中才会有的问题：国内的互联网环境由于GFW的存在，就使得DNS服务速度会比正常情况慢不少。

基于以上三个原因所导致的最终结果就是，API请求在DNS解析阶段的耗时会很多。

那么针对这个的优化方案就是，索性直接走IP请求，那不就绕过DNS服务的耗时了嘛。

另外一个，就是上面提到的建立链接时候的第三步，国内的网络环境分北网通南电信（当然实际情况更复杂，这里随便说说），不同服务商之间的连接，延时是很大的，我们需要想办法让用户在最适合他的IP上给他提供服务，那么就针对我们绕过DNS服务的手段有一个额外要求：尽可能不要让用户使用对他来说很慢的IP。

所以综上所述，方案就应该是这样：本地有一份IP列表，这些IP是所有提供API的服务器的IP，每次应用启动的时候，针对这个列表里的所有IP取ping延时时间，然后取延时时间最小的那个IP作为今后发起请求的IP地址。

针对建立连接的优化手段其实是跟DNS域名解析的优化手段是一样的。不过这需要你的服务器提供服务的网络情况要多，一般现在的服务器都是双网卡，电信和网通。由于中国特色的互联网ISP分布，南北网络之间存在瓶颈，而我们App针对链接的优化手段主要就是着手于如何减轻这个瓶颈对App产生的影响，所以需要维护一个IP列表，这样就能就近连接了，就起到了优化的效果。

我们一般都是在应用启动的时候获得本地列表中所有IP的ping值，然后通过NSURLProtocol的手段将URL中的HOST修改为我们找到的最快的IP。另外，这个本地IP列表也会需要通过一个API来维护，一般是每天第一次启动的时候读一次API，然后更新到本地。

如果你还不熟悉NSURLProtocol应该怎么玩，看完官方文档和这篇文章以及这个Demo之后，你肯定就会了，其实很简单的。另外，刚才提到那篇文章的作者(mattt)还写了这个基于NSURLProtocol的工具，相当好用，是可以直接拿来集成到项目中的。

不用NSURLProtocol的话，用其他手段也可以做到这一点，但那些手段未免又比较愚蠢。

2. 针对链接传输数据量的优化

这个很好理解，传输的数据少了，那么自然速度就上去了。这里没什么花样可以讲的，就是压缩呗。各种压缩。

3. 针对链接复用的优化

建立链接本身是属于比较消耗资源的操作，耗电耗时。SPDY自带链接复用以及数据压缩的功能，所以服务端支持SPDY的时候，App直接挂SPDY就可以了。如果服务端不支持SPDY，也可以使用PipeLine，苹果原生自带这个功能。

一般来说业界内普遍的认识是SPDY优于PipeLine，然后即便如此，SPDY能够带来的网络层效率提升其实也没有文献上的图表那么明显，但还是有性能提升的。还有另外一种比较笨的链接复用的方法，就是维护一个队列，然后将队列里的请求压缩成一个请求发出去，之所以会存在滞留在队列中的请求，是因为在上一个请求还在外面飘的时候。这种做法最终的效果表面上看跟链接复用差别不大，但并不是真正的链接复用，只能说是请求合并。

还是说回来，我建议最好是用SPDY，SPDY和pipeline虽然都属于链接复用的范畴，但是pipeline并不是真正意义上的链接复用，SPDY的链接复用相对pipeline而言更为彻底。SPDY目前也有现成的客户端SDK可以使用，一个是twitter的CocoaSPDY，另一个是Voxer/iSPDY，这两个库都很活跃，大家可以挑合适的采用。

不过目前业界趋势是倾向于使用HTTP/2.0来代替SPDY，不过目前HTTP/2.0还没有正式出台，相关实现大部分都处在demo阶段，所以我们还是先SPDY搞起就好了。未来很有可能会放弃SPDY，转而采用HTTP/2.0来实现网络的优化。这是要提醒各位架构师注意的事情。嗯，我也不知道HTTP/2.0什么时候能出来。

> 渔说完了，鱼来了

这里是我当年设计并实现的安居客的网络层架构代码。当然，该脱敏的地方我都已经脱敏了，所以编不过是正常的，哈哈哈。但是代码比较齐全，重要地方注释我也写了很多。另外，为了让大家能够把这些代码看明白，我还附带了当年介绍这个框架演讲时的PPT。(补充说明一下，评论区好多人问PPT找不着在哪儿，PPT也在上面提到的repo里面，是个key后缀名的文件，用keynote打开)

然后就是，当年也有很多问题其实考虑得并没有现在清楚，所以有些地方还是做得不够好，比如拦截器和继承。而且当时的优化手段只有本地cache，安居客没有那么多IP可以给我ping，当年也没流行SPDY，而且API也还不支持HTTPS，所以当时的代码里面没有在这些地方做优化，比较原始。然而整个架构的基本思路一直没有变化：优先服务于业务方。另外，安居客的网络层多了一个service的概念，这是我这篇文章中没有讲

的。主要是因为安居客的API提供方很多，二手房，租房，新房，X项目等等API都是不同的API team提供的，以service作区分，如果你的app也是类似的情况，我也建议你设计一套service机制。现在这些service被我删得只剩下一个google的service，因为其他service都属于敏感内容。

另外，这里面提供的PPT我很希望大家能够花时间去看看，在PPT里面有些更加细的东西我在博客里没有写，主要是我比较懒，然后这篇文章拖的时间比较长了，花时间搬运这个没什么意思，不过内容还是值得各位读者去看的。关于PPT里面大家有什么问题的，也可以在评论区问，我都会回答。

总结

第一部分主要讲了网络层应当如何跟业务层进行数据交互，进行数据交互时采用怎样的数据格式，以及设计时代码结构上的一些问题，诸如继承的处理，回调的处理，交互方式的选择，reformer的设计，保持数据可读性等等等等，主要偏重于设计（这可是艺术活，哈哈哈）。

第二部分讲了网络安全上，客户端要做的两点。当然，从网络安全的角度上讲，服务端也要做很多很多事情，客户端要做的一些边角细节的事情也还会有很多，比如做一些代码混淆，尽可能避免代码中明文展示key。不过大头主要就是这两个，而且也都是需要服务端同学去配合的。主要偏重于介绍。（主要是也没啥好实践的，google一下教程照着来就好了）。

第三部分讲了优化，优化的所有方面都已经列出来了，如果业界再有七七八八的别的手段，也基本逃离不出本文的范围。这里有些优化手段是需要服务端同学配合的，有些不需要，大家看各自情况来决定。主要偏重于实践。

最后给出了我之前在安居客做的网络层架构的主要代码，以及当时演讲时的PPT。关于代码或PPT中有任何问题，都可以在评论区问我。

这一篇文章出得比较晚，因为公司的事情，中间间隔了一个礼拜，希望大家谅解。另外，隔了一个礼拜之后我再写，发现有些地方我已经想不起来当初是应该怎么行文下去的了，然后发之前我把文章又看了几遍，尽可能把断片的地方抹平了，如果大家读起来有什么地方感觉奇怪的，或者讲到

一半就没了的，那应该就是断片了。在评论区跟我说一下，我补上去。

然后如果有需要勘误的地方，也请在评论区指出，帮助我把错的地方订正回来，如果有没讲到的地方，但你又特别想要了解的，也可以在评论区提出来，我会补上去。说不定看完之后你脑袋里还会有很多个问号，也请在评论区问出来哈，说不定别人也有跟你一样的问题，他就能在评论区找到答案了。

在第二篇文章的评论区里面出现了喷子，遇到这种情况我怎么可能删帖呢？那根本就不是我的风格哇，哈哈哈。我肯定是会喷回去的，并且还会把链接传播给周围人，发动周围朋友来看:"快看，这儿有2B，哈哈哈"。

嗯，所以评论的时候你一定要想清楚哈，我写代码的实力不差，打嘴仗的实力那可比写代码强多了。评论区同样欢迎切磋。

有任何问题建议直接在评论区提问，这样后来的人如果有相同的问题，就能直接找到答案了。提问之前也可以先看看评论区有没有人问过类似问题了。

所有评论和问题我都会在第一时间回复，QQ上我是不回答问题的哈。