寄语

很多时候，我们看待技术的方向总是以目前所处的环境、或者项目。但我们应该意识到，我们是以开发者自居，而不是某某公司的开发者自居。我们学习储备技术应该从自身出发。让自己技术范畴的选择更加的宽阔，而不是随着经验的累积反而变的更加狭隘，缺少了创造性和选择性。我们是独一无二的，我们应该做独一无二的事！用全力以赴的态度去追逐，你会发现那些曾经你偷偷设想的事正一步一步朝你走来。人最珍贵的就是我们的思想，去开拓的思考，像守护生命一样，守护你坚守的梦想。

开发者技术交流群:637919808

阿里iOS面试题

1.dSYM你是如何分析的?
2.多线程有哪几种？你更倾向于哪一种?
3.单例弊端?
4.介绍下App启动的完成过程?
5.比如App启动过慢，你可能想到的因素有哪些?
6.0x8badf00d表示是什么?
7.怎么防止反编译?
8.说说你了解的第三方原理或底层知识?

1、dSYM 你是如何分析的?

dSYM是什么?

什么是 dSYM 文件Xcode编译项目后，我们会看到一个同名的 dSYM 文件，dSYM 是保存 16 进制函数地址映射信息的中转文件，我们调试的 symbols 都会包含在这个文件中，并且每次编译项目的时候都会生成一个新的 dSYM 文件，位于 /Users/<用户名>/Library/Developer/Xcode/Archives 目录下，对于每一个发布版本我们都很有必要保存对应的 Archives 文件 (AUTOMATICALLY SAVE THE DSYM FILES 这篇文章介绍了通过脚本每次编译后都自动保存 dSYM 文件)。

dSYM文件有什么用?

当我们软件 release 模式打包或上线后，不会像我们在 Xcode 中那样直观的看到用崩溃的错误，这个时候我们就需要分析 crash report 文件了，iOS 设备中会有日志文件保存我们每个应用出错的函数内存地址，通过 Xcode 的 Organizer 可以将 iOS 设备中的 DeviceLog 导出成 crash 文件，这个时候我们就可以通过出错的函数地址去查询 dSYM 文件中程序对应的函数名和文件名。大前提是我们需要有软件版本对应的 dSYM 文件，这也是为什么我们很有必要保存每个发布版本的 Archives 文件了。

如何将文件--对应?

每一个 xx.app 和 xx.app.dSYM 文件都有对应的 UUID，crash 文件也有自己的 UUID，只要这三个文件的 UUID 一致，我们就可以通过他们解析出正确的错误函数信息了。1.查看 xx.app 文件的 UUID，terminal 中输入命令：dwarfdump --uuid xx.app/xx (xx代表你的项目名)2.查看 xx.app.dSYM 文件的 UUID，在 terminal 中输入命令：dwarfdump --uuid xx.app.dSYM 3.crash 文件内第一行 Incident Identifier 就是该 crash 文件的 UUID。
dSYM工具

将命令封装到应用，在解决bug上提供了便利

使用步骤:1.将打包发布软件时的xcarchive文件拖入软件窗口内的任意位置(支持多个文件同时拖入，注意：文件名不要包含空格)2.选中任意一个版本的xcarchive文件，右边会列出该xcarchive文件支持的CPU类型，选中错误对应的CPU类型。3.对比错误给出的UUID和工具界面中给出的UUID是否一致。 4.将错误地址输入工具的文本框中，点击分析。

2、多线程有哪几种？你更倾向哪一种?

1、多线程分类

- pthread

  1、一套通用的多线程API
  2、适用于Unix\Linux\Windows等系统
  3、跨平台、可移植
  4、使用难度大

5、使用语言：C语言
6、使用频率：几乎不使用
7、线程生命周期：由开发者进行管理

- NSThread
  1、面向对象
  2、简单易用，可直接操作线程
  3、使用语言：OC语言
  4、使用频率：偶尔使用
  5、线程生命周期：有开发者管理

- GCD
  1、替换NSThread等线程技术
  2、充分利用了设备多核（自动）
  3、使用语言：C语言
  4、使用频率：经常使用
  5、线程生命周期：自动管理

- NSOperation
  1、基于GCD(底层是GCD)
  2、比GCD多了一些更简单实用的功能
  3、使用更加面向对象
  4、使用语言：OC语言
  5、使用频率：经常使用
  6、线程生命周期：自动管理

多线程的原理：同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象思考：如果线程非常非常多，会发生什么情况？ CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优点：能适当提高程序的执行效率能适当提高资源利用率（CPU、内存利用率）

多线程的缺点：线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能线程越多，CPU在调度线程上的开销就越大程序设计

更加复杂：比如线程之间的通信、多线程的数据共享

2、你更倾向于哪一种？倾向于GCD

GCD 技术是一个轻量的，底层实现隐藏的神奇技术，我们能够通过GCD和block轻松实现多线程编程，有时候，GCD相比其他系统提供的多线程方法更加有效，当然，有时候GCD不是最佳选择，另一个多线程编程的技术 NSOprationQueue 让我们能够将后台线程以队列方式依序执行，并提供更多操作的入口，这和 GCD 的实现有些类似。 这种类似不是一个巧合，在早期，MacOX 与 iOS 的程序都普遍采用Operation Queue来进行编写后台线程代码，而之后出现的GCD技术大体是依照前者的原则来实现的，而随着GCD的普及，在iOS 4 与 MacOS X 10.6以后，Operation Queue的底层实现都是用GCD来实现的

那这两者直接有什么区别呢?

1、GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2、在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3、NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4、我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5、在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6、我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。总的来说，Operation queue

提供了更多你在编写多线程程序时需要的功能，并隐藏了许多线程调度，线程取消与线程优先级的复杂代码，为我们提供简单的API入口。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。但是我认为当我们的需求能够以更简单的底层代码完成的时候，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

3、你更倾向于哪一种？倾向于NSOperationNSOperation

1、NSOperation拥有更多的函数可用，具体查看api。NSOperationQueue是在GCD基础上实现的，只不过是GCD更高一层的抽象

2、在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系。

3、NSOperationQueue支持KVO。可以监测operation是否正在执行（isExecuted）、是否结束（isFinished），是否取消（isCanceld）

4、GCD 只支持FIFO 的队列，而NSOperationQueue可以调整队列的执行顺序（通过调整权重）。NSOperationQueue可以方便的管理并发、NSOperation之间的优先级。
使用NSOperation的情况：各个操作之间有依赖关系、操作需要取消暂停、并发管理、控制操作之间优先级，限制同时能执行的线程数量.让线程在某时刻停止/继续等。
使用GCD的情况：一般的需求很简单的多线程操作，用GCD都可以了，简单高效。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。当需求简单，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

3.单例弊端?

优点：
1、一个类只被实例化一次，提供了对唯一实例的受控访问。
2、节省系统资源
3、允许可变数目的实例

缺点
1、一个类只有一个对象，可能造成责任过重，在一定程度上违背了“单一

职责原则”
2、由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。
3、滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

4、APP启动的完整过程

1、APP启动过程

- 解析Info.plist
- 加载相关信息，例如闪屏
- 沙盒建立、权限检查
- **Mach-O加载**
- 如果是二进制文件，寻找合适当前CPU类别的部分
- 加载所有依赖的Mach-O文件（递归调用Mach-o加载方法）
- 定位内部、外部指针引用，例如字符串，函数等
- 执行声明为attribute(constructor)的C函数
- 加载类的扩展中的方法
- C++静态对象加载，调用Objc的+ load函数

2、程序执行

- main函数
- 执行UIApplicationMain函数
- 创建UIApplication对象
- 创建UIApplicationDelegate对象并复制
- 读取配置文件info.plist,设置程序启动的一些属性
- 创建应用程序的Main Runloop循环
- UIApplicationDelegate对象开始处理监听事件
- 程序启动之后，首先调用application.didFinishLaunchingWithOptions:方法
- 如果info.plist中配置了启动的storyBoard的文件名，则加载storyboard文件
- 如果没有配置，则根据代码创建UIWindow ->rootViewController->显示

5、造成APP启动过慢，你可能想到的原因有哪些?

- 影响启动性能的因素**App启动过程中每一个步骤都会影响启动性能，但是有些部分所消耗的时间少之又少，另外有些部分根本无法避免，考虑到投入产出比，我们只列出我们可以优化的部分：main()函数之前耗时的影响因素
- 动态库加载越多，启动越慢。
- ObjC类越多，启动越慢
- C的constructor函数越多，启动越慢
- C++静态对象越多，启动越慢
- ObjC的+load越多，启动越慢实验证明，在ObjC类的数目一样多的情况下，需要加载的动态库越多，App启动就越慢。同样的，在动态库一样多的情况下，ObjC的类越多，App的启动也越慢。需要加载的动态库从1个上升到10个的时候，用户几乎感知不到任何分别，但从10个上升到100个的时候就会变得十分明显。同理，100个类和1000个类，可能也很难查察觉得出，但1000个类和10000个类的分别就开始明显起来。同样的，尽量不要写attribute((constructor))的C函数，也尽量不要用到C++的静态对象；至于ObjC的+load方法，似乎大家已经习惯不用它了。任何情况下，能用dispatch_once()来完成的，就尽量不要用到以上的方法。main()函数之后耗时的影响因素
- 执行main()函数的耗时
- 执行applicationWillFinishLaunching的耗时
- rootViewController及其childViewController的加载、view及其subviews的加载applicationWillFinishLaunching的耗时

优化目标：

- 优化的目标由于每个App的情况有所不同，需要加载的数据量也有所不同，事实上我们无法使用一种统一的标准来衡量不同的App。
- 应该在400ms内完成main()函数之前的加载
- 整体过程耗时不能超过20秒，否则系统会kill掉进程，App启动失败400ms内完成main()函数前的加载的建议值是怎样定出来的呢？其实我也没有太深究过这个问题，但是，当用户点击了一个App的图标时，iOS做动画到闪屏图出现的时长正好是这个数字，我想也许跟这个有

关。针对不同规模的App，我们的目标应该有所取舍。例如，对于像手机QQ这种集整个SNG的代码大成撸出来的App，对动态库的使用在所难免，但对于WiFi管家，由于在用户连接WiFi的时候需要非常快速的响应，所以快速启动就非常重要。那么，如何定制优化的目标呢？首先，要确定启动性能的界限，例如，在各种App性能的指标中，哪一此属于启动性能的范畴，哪一些则于App的流畅度性能?
我认为应该首先把启动过程分为四个部分：
1、main()函数之前

2、main()函数之后至applicationWillFinishLaunching完成

3、App完成所有本地数据的加载并将相应的信息展示给用户

4、App完成所有联网数据的加载并将相应的信息展示给用户1+2一起决定了我们需要用户等待多久才能出现一个主视图，同时也是技术上可以精确测量的时长，1+2+3决定了用户视觉上的等待出现有用信息所需要的时长，1+2+3+4决定了我们需要多少时间才能让我们需要展示给用户的所有信息全部出现。淘宝的iOS客户端无疑是各部分都做得非常优秀的典型。它所承载的业务完全不比微信和手机QQ少，但几乎瞬间完成了启动，并利用缓存机制使得用户马上看到“貌似完整”的界面，然后立即又刷新了刚刚联网更新回来的信息。也就是说，无论是技术上还是视觉上，它都非常的“快”。

1. 移除不需要用到的动态库因为WiFi管家是个小项目，用到的动态库不多，自动化处理的优势不大，我这里也就简单的把依赖的动态移除出项目，再根据编译错误一个一个加回来。如果有靠谱的方法，欢迎大家补充一下

2、 移除不需要用到的类项目做久了总有一些吊诡的类像幽灵一样驱之不去，由于【不要相信产品经理】的思想作怪，需求变更后，有些类可能用不上了，但却因为担心需求再变回来就没有移除掉，后来就彻底忘记要移除了。为了解决这个历史问题，在这个过程中我试过多种方法来扫描没有用到的类，其中有一种是编译后对ObjC类的指针引用进行反向扫描，可惜实际上收获不是很明显，而且还要写很多例外代码来处理一些特殊情况。后来发现一个叫做fui（Find Unused Imports）的开源项目能很好的分析出不再使用的类，准确率非常高，唯一的问题是它处理不了动态库和静态库里提供的类，也处理不了C++的类模板。使用方法是在Terminal中cd到项目所在的目录，然后执行fui find，然后等上那么几分钟（是的你没有看

错，真的需要好几分钟甚至需要更长的时间），就可以得到一个列表了。由于这个工具还不是100%靠谱，可根据这个列表，在Xcode中手动检查并删除不再用到的类。实际上，日常对代码工程的维护非常重要，如果制定好一套半废弃代码的维护方法，小问题就不会积累成大问题。有时候对于一些暂时不再使用的代码，我也很纠结于要不要svn rm，因为从代码历史中找删除掉的文件还是不太方便。不知道大家有没有相关的经验可以分享，也请不吝赐教

3、合并功能类似的类和扩展（Category）由于Category的实现原理，和ObjC的动态绑定有很强的关系，所以实际上类的扩展是比较占用启动时间的。尽量合并一些扩展，会对启动有一定的优化作用。不过个人认为也不能因为它占用启动时间而去逃避使用扩展，毕竟程序员的时间比CPU的时间值钱，这里只是强调要合并一些在工程、架构上没有太大意义的扩展。

1. 压缩资源图片压缩图片为什么能加快启动速度呢？因为启动的时候大大小小的图片加载个十来二十个是很正常的，图片小了，IO操作量就小了，启动当然就会快了。事实上，Xcode在编译App的时候，已经自动把需要打包到App里的资源图片压缩过一遍了。然而Xcode的压缩会相对比较保守。另一方面，我们正常的设计师由于需要符合其正常的审美需要生成的正常的PNG图片，因此图片大小是比较大的，然而如果以程序员的直男审美而采用过激的压缩会直接激怒设计师。解决各种矛盾的方法就是要找出一种相当靠谱的压缩方法，而且最好是基本无损的，而且压缩率还要特别高，至少要比Xcode自动压缩的效果要更好才有意义。经过各种试验，最后发现唯一可靠的压缩算法是TinyPNG，其它各种方法，要么没效果，要么产生色差或模糊。但是非常可惜的是TinyPNG并不是完全免费的，而且需要通过网络请求来压缩图片（应该是为了保护其牛逼的压缩算法）。为了解决这个问题，我写了一个类来执行这个请求，请参见阅读原文里的SSTinyPNGRequest和SSPNGCompressor。因为这个项目只有我一个人在用所以代码写得有点随意，有问题可以私聊也可以在评论里问，有改进的方法也非常欢迎指正。另外说明一下，使用这个类需要你自行到 https://tinypng.com/developers 申请APIKey，每一个用户每月有500张图片压缩是免费的，而每个邮箱可以注册一个用户，你懂的。

   5、优化applicationWillFinishLaunching随着项目做的时间长了，applicationWillFinishLaunching里要处理的代码会越积越多，WiFi管家的

iOS版本有一段时间没有控制好，里面的逻辑乱得有点丢人。因为可能涉及到一些项目的安全性问题，这里不能分享所有的优化细节及发现的思路。仅列出在applicationWillFinishLaunching中主要需要处理的业务及相关问题的改进方案。这里大部分都是一些苦逼活，但有一点特别值得分享的是，有一些优化，是无法在数据上体现的，但是视觉上却能给用户较大的提升。例如在【各种业务请求配置更新】的部分，经过分析优化后，启动过程并发的http请求数量从66条压缩到了23条，如此一来为启动成功后新闻资讯及其图片的加载留出了更多的带宽，从而保证了在第一时间完成新闻资讯的加载。实际测试表明，光做KPI的事情是不够的，人还是需要有点理想，经过优化，在视觉体验上进步非常明显。另外，过程中请教过SNG的大牛们，听说他们因为需要在applicationWillFinishLaunching里处理的业务更多，所以还做了管理器管理这些任务，不过因为WiFi管家是个小项目，有点杀鸡用牛刀的感觉，因此没有深入研究。

6、 优化rootViewController加载考虑到我作为一只程序猴，工资还行，为了给公司节约成本，在优化之前，当然需要先测试一下哪些ViewController的加载耗时比较大，然后再深入到具体业务中看哪些部分存在较大的优化空间。同时，先做优化效果明显的部分也有利于增强自己的信心。在开始讲述问题之前，我们先来看一下wife管家的UI层次结构：一个看似简单的界面由于承载了很多业务需求，代码量其实已经非常惊人。这里我不具体讲述这些惊人的业务量了，抽象而言可WiFi管家的UI架构总体而言基于TabBarController的框架，三个tab分别是“连接”、“发现”及“我的”。App启动的时候，根据加载原理，会加载TabBarController、第一个Tab（“连接”）的ViewController及其所有childViewController。UI构架请看如下示意图，其中蓝色的部分需要在App启动的时候立即加载

6、0x8badf00d表示是什么?

- 0x8badf00d: 读做 “ate bad food”! (把数字换成字母，是不是很像 :p)该编码表示应用是因为发生watchdog超时而被iOS终止的。 通常是应用花费太多时间而无法启动、终止或响应用系统事件。
- 0xbad22222: 该编码表示 VoIP 应用因为过于频繁重启而被终止
- 0xdead10cc: 读做 “dead lock”!该代码表明应用因为在后台运行时占用系统资源，如通讯录数据库不释放而被终止 。

- 0xdeadfa11: 读做 “dead fall”! 该代码表示应用是被用户强制退出的。根据苹果文档, 强制退出发生在用户长按开关按钮直到出现 “滑动来关机”, 然后长按 Home按钮。强制退出将产生 包含0xdeadfa11 异常编码的崩溃日志, 因为大多数是强制退出是因为应用阻塞了界面。

7、怎么防止反编译?

- **本地数据加密**：iOS应用防反编译加密技术之一：对NSUserDefaults，sqlite存储文件数据加密，保护帐号和关键信息
- **URL编码加密**：iOS应用防反编译加密技术之二：对程序中出现的URL进行编码加密，防止URL被静态分析
- **网络传输数据加密**：iOS应用防反编译加密技术之三：对客户端传输数据提供加密方案，有效防止通过网络接口的拦截获取数据
- **方法体，方法名高级混淆**：iOS应用防反编译加密技术之四：对应用程序的方法名和方法体进行混淆，保证源码被逆向后无法解析代码
- **程序结构混排加密**：iOS应用防反编译加密技术之五：对应用程序逻辑结构进行打乱混排，保证源码可读性降到最低
- **借助第三方APP加固**，例如：网易云易盾

8、iOS应用架构

可从以下4个方面回答:
iOS应用架构谈 view层的组织和调用方案？
iOS应用架构谈 网络层设计方案？
iOS应用架构谈 动态部署方案？
iOS应用架构谈 本地持久化方案?

> **什么样app的架构叫好架构?**
> 代码整齐，分类明确，没有common，没有core
> 不用文档，或很少文档，就能让业务方上手
> 思路和方法要统一，尽量不要多元
> 没有横向依赖，万不得已不出现跨层访问
> 对业务方该限制的地方有限制，该灵活的地方要给业务方创造灵活实现的条件
> 易测试，易拓展

> 保持一定量的超前性
> 接口少，接口参数少
> 高性能

- 第一类：精简型应用架构
  这类架构的文章分析主要还是围绕MVC展开，以苹果自带UIViewController优劣为出发点，再结合主流的MVP，MVVM，MVCS等变种进行分析演变。这类的探讨重点在于M，V，C三类角色的定义以及之间的数据事件流向的规范。很多小型应用所面临的问题及其架构层面的解决方案都集中在这一类。
- 第二类：综合型应用架构
  对于用户量级在千万级或以上的应用来说，MVC这一层面的思考已无法应对业务疯狂增长所带来的负担。这类应用往往需要专业资深的架构师出面进行深层次的思考设计，业内不少大厂如淘宝，天猫，携程等都做过一些分享。不过到了这一层级的战斗，不光考验架构师的技术积累，更重要的是架构师对于业务的整体理解。我姑且把这类架构名之为：综合型应用架构。综合型应用架构一般不会提到MVC，更多是在探讨“层”与“模块”的划分和耦合。后面我会就几个经典样本做下详尽深入的分析。
- 第三类：深度优化的综合型应用架构
  综合型应用架构是应对大规模业务增长的必经之路，一旦架构成型，后期业务膨胀会不停的打磨架构本身，产品本身对体验质量的追求会要求架构师和技术团队不停的优化架构细节。这种优化可以分为两块，第一是组件或模块划分的粒度越来越细，第二是组件模块的深度优化，比如网络层的深度优化，sqlite优化（多线程，FTS，安全等），数据加密，HotFix，Hybrid等，一些开源的第三方库已不能满足要求，需要团队自己重造轮子。这一层面的架构设计涉及面广，对架构师，团队技术人员的技术深度和业务理解能力有较高要求，短短一篇技术文章往往只能走马观花的介绍个大概，每一次优化几乎都可以作为一个专题来讲解。

*第四类：组织型应用架构
这类架构在第三类的基础之上更进了一步，除了关注系统层面的架构设计之外，更对团队或部门之间协作方式，各系统模块的演进方式，产品发布流程等都做了规范。除去业务膨胀带来的压力，人员增长，各团队协作依

赖增强等都会对app的质量，迭代速度产生影响，这些问题也需要从架构层面去解决。这类结合技术架构和组织架构的分享还比较少。
以上四种类型的架构又可以看做一般App从简至繁，公司规模随之增长的演进过程，技术圈绝大部分的架构类分享文章都可以归为上述四类。
对于什么是架构的学术定义，似乎大家并不太在意，更关心的是如何解决自身项目当下的问题。虽然在我看来第一类架构更像是在讨论设计模式，但这里面确实又有非常多的知识可以深入挖掘，这里就把所有“解决应用整体设计问题”的讨论都归类于架构这一话题。
值得一提的是，架构师的视野和积累一般都受限于自己所经历项目及业务的规模。如果有机会，工程师还是应该尽可能去BAT这类巨头级公司历练一下，知识深度和广度的构建绝非纸上可得。

可从以下方面回答：（举例说明，携程开发提供）

- 核心功能SDK化
- 通讯、定位、Hybrid、数据库、登录、分享、基础库等
- 直接提供给其他BU独立App使用
- 公用业务功能组件化
- 地图、日历、城市、图片、通讯录等13个公共组件
- 减少各BUG重复开发工作量
- 性能数据指标采集：
- 网络性能:网络服务成功率、平均耗时、耗时分布
- 定位:获取经纬度成功率、城市定位成功率
- 启动时间、内存、流量等指标
- 多种纬度:系统、App版本、网络状况、位置等
- 网络优化
- 使用TCP长连接实现网络服务
- 根据网络状况2G/3G/4G/WIFI进行调优参数
- 根据连接/读/写不同阶段使用重试机制
- 使用IP列表避免DNS解析失败或者劫持
- 根据网络延迟选择服务端IP(使用Ping)

- 使用ProtocolBuffer+Gzip减少Payload

8.说说你了解的第三方原理或底层知识?

Runtime、Runloop、block
SD原理、AFN、YYCache、GCD源码分析、JSPatch、GPUImage等等

以上资料由iOS开发者提供。此资料只作为分享资料，不用于商业贩卖！