iOS群整理收集：335930567

前言：先自己尝试去回答，回答不了再看参考答案，你才能学的更多!

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.如何把异步线程转换成同步任务进行单元测试?
5.介绍下App启动的完成过程?
6.比如App启动过慢，你可能想到的因素有哪些?
7.0x8badf00d表示是什么?
8.怎么防止反编译?
9.说说你遇到到的技术难点?
10.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?

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 文件有什么作用

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。

如何将文件一一对应

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

于是我抽了几个小时的时间将这些命令封装到一个应用中，也为以后解决bug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。


dSYM文件分析工具
请将xcarchive文件拖拽到窗口中并选中任意一个版本进行分析
10.xcarchive
7.xcarchive
8.xcarchive
9.xcarchive
请选择该archive文件对应的编译类型
armv7
选中xcarchive的UUID:
1D9DCCF1-EC70-3E81-9F9C-C0F9E144EDC5
请输入错误信息的内存地址:
0x1bd8f9
分析
有可能错误的地方:
-[DiagnoseMainViewController refreshDiagForWhoLable] (in EkuKangDA) (DiagnoseMainViewController.m:829)

2.多线程有哪几种？你更倾向于哪一种?

技术方案	简介	语言	线程生命周期	使用频率
pthread	• 一套通用的多线程API • 适用于Unix\Linux\Windows等系统 • 跨平台\可移植 • 使用难度大	C	程序员管理	几乎不用
NSThread	• 使用更加面向对象 • 简单易用，可直接操作线程对象	OC	程序员管理	偶尔使用
GCD	• 旨在替代NSThread等线程技术 • 充分利用设备的多核	C	自动管理	经常使用
NSOperation	• 基于GCD（底层是GCD） • 比GCD多了一些更简单实用的功能 • 使用更加面向对象	OC	自动管理	经常使用

第一种：pthread

.特点:

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

b.使用语言：c语言

c.使用频率：几乎不用

d.线程生命周期：由程序员进行管理

第二种：NSThread

a.特点:

1）使用更加面向对象

2）简单易用，可直接操作线程对象

b.使用语言：OC语言

c.使用频率：偶尔使用

d.线程生命周期：由程序员进行管理

第三种：GCD

a.特点：
1）旨在替代NSThread等线程技术
2）充分利用设备的多核（自动）
b.使用语言：C语言
c.使用频率：经常使用
d.线程生命周期：自动管理

第四种：NSOperation
a.特点：
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
b.使用语言：OC语言
c.使用频率：经常使用
d.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
开启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

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。

倾向于：**NSOperation**
NSOperation相对于GCD：
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.如何把异步线程转换成同步任务进行单元测试?
在项目中的应用：强制把异步任务转换为同步任务来方便进行单元测试
下面是 Parse 的一段代码:

```
@interface PFEventuallyQueueTestHelper : NSObject {
  dispatch_semaphore_t events[PFEventuallyQueueEventCount];
```

```
}

- (void)clear;
- (void)notify:(PFEventuallyQueueTestHelperEvent)event;
- (BOOL)waitFor:(PFEventuallyQueueTestHelperEvent)event;
```

注释是这样写的：

PFEventuallyQueueTestHelper gets notifications of various events happening in the command cache, // so that tests can be synchronized. See CommandTests.m for examples of how to use this.

强制把异步任务转换为同步任务来方便进行单元测试。这个用途信号量是最合适的用途。但注意并不推荐应用到除此之外的其它场景！

这种异步转同步便于单元测试的用法类似于下面的写法：

```
#define WAIT_FOREVER [self waitForStatus:XCTAsyncTestCaseStatusSucceeded timeout:DBL_MAX];
#define NOTIFY [self notify:XCTAsyncTestCaseStatusSucceeded];
- (void)testInstallationMutated {
  NSDictionary *dict = [self jsonWithFileName:@"TestInstallationSave"];
  AVInstallation *installation = [AVInstallation currentInstallation];
  [installation objectFromDictionary:dict];
  [installation setObject:@(YES) forKey:@"enableNoDisturb"];
  [installation saveInBackgroundWithBlock:^(BOOL succeeded, NSError *error) {
    XCTAssertNil(error);
    NOTIFY;
  }];
  WAIT;
}
```

信号量属性底层工具，他虽然非常强大，但在多数需要使用它的场合，最好从设计角度重新考虑，看是否可以不用，应该优先考虑使用诸如操作队列这样的高级工具。通常可以通过增加一个分派队列配合 **dispatch_suspend** ，或者通过其它方式分解操作来避免使用信号量。信号量并非不好，只是它本身是锁，能不使用就不用。尽量用 **cocoa** 框架中的高级抽象，信号量非常接近底层。所以除了上面的例子是最佳应用场景外，不推荐应用到除此之外的其它场景！

《关于**dispatch_semaphore**的使用》 中有这样的描述：

关于信号量，一般可以用停车来比喻。

停车场剩余4个车位，那么即使同时来了四辆车也能停的下。如果此时来了五辆车，那么就有一辆需要等待。

信号量的值就相当于剩余车位的数目，dispatch_semaphore_wait函数就相当于来了一辆车，

dispatch_semaphore_signal，就相当于走了一辆车。停车位的剩余数目在初始化的时候就已经指明了（dispatch_semaphore_create（long value））

调用一次dispatch_semaphore_signal，剩余的车位就增加一个；调用一次dispatch_semaphore_wait剩余车位就减少一个；

当剩余车位为0时，再来车（即调用dispatch_semaphore_wait）就只能等待。有可能同时有几辆车等待一个停车位。有些车主

没有耐心，给自己设定了一段等待时间，这段时间内等不到停车位就走了，如果等到了就开进去停车。而有些车主就像把车停在这，

所以就一直等下去。

《**GCD dispatch_semaphore 信号量 协调线程同步**》也有类似的比喻：
以一个停车场是运作为例。为了简单起见，假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆不受阻碍的进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入一辆，如果又离开两辆，则又可以放入两辆，如此往复。

在这个停车场系统中，车位是公共资源，每辆车好比一个线程，看门人起的就是信号量的作用。 更进一步，信号量的特性如下：信号量是一个非负整数（车位数），所有通过它的线程（车辆）都会将该整数减一（通过它当然是为了使用资源），当该整数值为零时，所有试图通过它的线程都将处于等待状态。在信号量上我们定义两种操作： Wait（等待） 和 Release（释放）。 当一个线程调用Wait（等待）操作时，它要么通过然后将信号量减一，要么一直等下去，直到信号量大于一或超时。Release（释放）实际上是在信号量上执行加操作，对应于车辆离开停车场，该操作之所以叫做“释放”是因为加操作实际上是释放了由信号量守护的资源。

这个比喻里可以用一个表格来表示：

喻体	本体	代码
车位	信号量	`dispatch_semaphore_t`
剩余几个车位	最大并发线程	`dispatch_semaphore_t`
看门人起的作用	信号量的作用	`dispatch_semaphore_t`
车	线程	代码
耐心的极限时间	超时时间	`dispatch_semaphore_wait`
逛街结束走了，离开车位	signal+1	`dispatch_semapho...signal`

5.介绍下App启动的完成过程?

1. App启动过程

- 解析Info.plist
  - 加载相关信息，例如如闪屏
  - 沙箱建立、权限检查

- **Mach-O加载**
  - 如果是胖二进制文件，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加载的方法）
  - 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
- 加载类扩展（Category）中的方法
- C++静态对象加载、调用ObjC的 +load 函数

- 程序执行

· 1.main函数
· 2.执行UIApplicationMain函数
· 1.创建UIApplication对象
· 2.创建UIApplicationDelegate对象并复制
· 3.读取配置文件info.plist，设置程序启动的一些属性，(关于info.plist的内容可网上搜索下)
· 4.创建应用程序的Main Runloop循环
· 3.UIApplicationDelegate对象开始处理监听到的事件
· 1.程序启动成功之后，首先调用application:didFinishLaunchingWithOptions:方法,
· 如果info.plist文件中配置了启动storyboard文件名，则加载storyboard文件。
· 如果没有配置，则根据代码来创建UIWindow--->UIWindow的rootViewController-->显示

6.比如App启动过慢，你可能想到的因素有哪些?

1. 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
如果有这样这样的代码:

```
@implementation MQQTabBarController

- (void)viewDidLoad {
        NSLog(@"%s", __PRETTY_FUNCTION__);
        [super viewDidLoad];
        // Do any additional setup after loading the view.

        UIViewController *tab1 = [[[MQQTab1ViewController alloc] init] autorelease];
        tab1.tabBarItem.title = @"red";
        [self addChildViewController:tab1];

        UIViewController *tab2 = [[[MQQTab2ViewController alloc] init] autorelease];
        tab2.tabBarItem.title = @"blue";
        [self addChildViewController:tab2];

        UIViewController *tab3 = [[[MQQTab3ViewController alloc] init] autorelease];
        tab3.tabBarItem.title = @"green";
        [self addChildViewController:tab3];
}
```

```
1 那么-[MQQTabBarController viewDidLoad]、
2 -[AppDelegate application:didFinishLaunchingWithOptions:]、
3 -[MQQTab1ViewController viewDidLoad]、
4 -[MQQTab2ViewController viewDidLoad]、
5 -[MQQTab2ViewController viewDidLoad] 完成的先后顺序是怎样的呢?
```

答案是:

1. -[MQQTabBarController viewDidLoad]
2. -[MQQTab1ViewController viewDidLoad]
3. -[AppDelegate application:didFinishLaunchingWithOptions:]
4. -[MQQTab2ViewController viewDidLoad] （点击了第二个tab之后加载）
5. -[MQQTab3ViewController viewDidLoad] （点击了第三个tab之后加载）

一般而言，大部分情况下我们都会把界面的初始化过程放在viewDidLoad，但是这个过程会影响消耗启动的时间。特别是在类似TabBarController这种会嵌套childViewController的ViewController的情况，它也会把部分children也初始化，因此各种viewDidLoad会递归的进行。
最简单的解决的方法，是把viewController延后加载，但实际上这属于一种掩耳盗铃，确实，applicationWillFinishLaunching的耗时是降下来了，但用户体验上并没有感觉变快。
更好一点的解决方法有点类似facebook，主视图会第一时间加载，但里面的数据和界面都会延后加载，这样用户就会阶段性的获得视觉上的变化，从而在视觉体验上感觉App启动得很快。

【第二部分】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。苹果。

- 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。

4. 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。

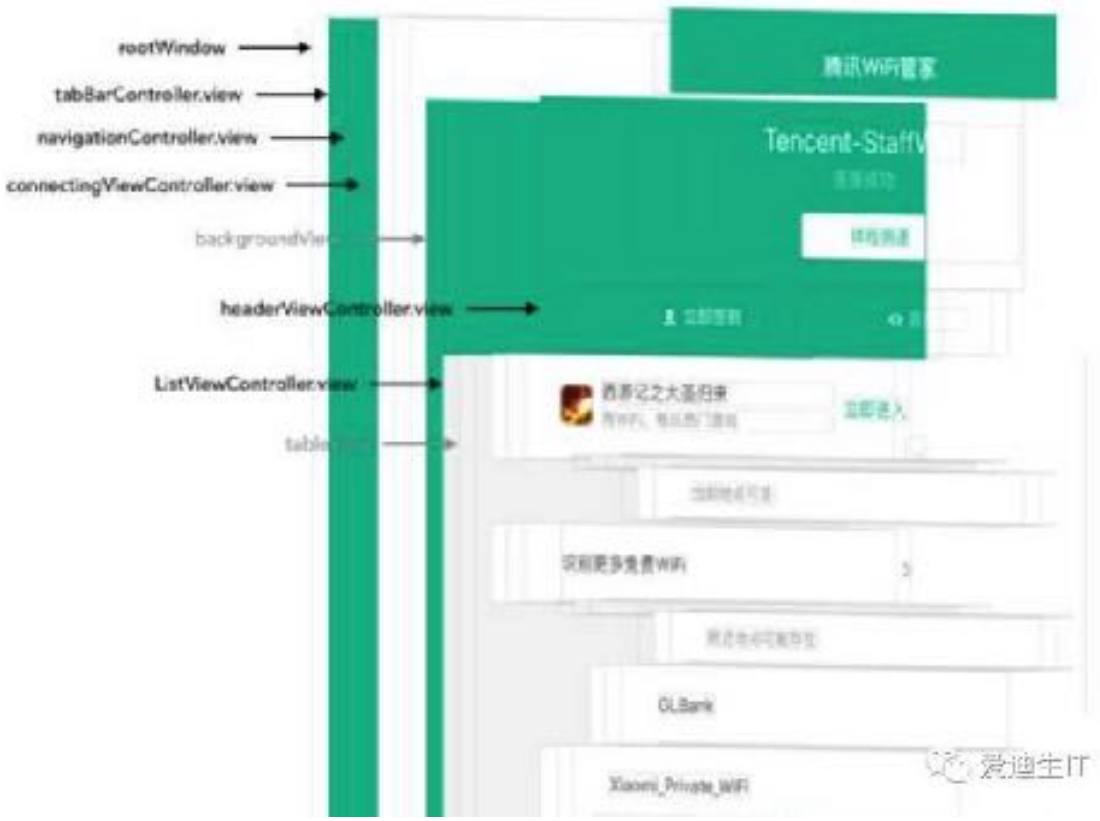
6. 优化rootViewController加载

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：



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。

对所有启动相关的模块打锚点计算耗时后，发现tabBarController和connectingViewController分别占用了applicationWillFinishLaunching耗时的31%和24%。加载耗费了大量时间，这跟它所需要承载的逻辑任务似乎并不对称。于是检查相关代码进行深入分析，发现了几个问题比较严重：

1. 有些程序员可能架构意识不是太强，直接在**tabBarController**的启动过程中插入了各种奇怪的业务，例如检查**WiFi**连接状态变化、配置拉取，而这些业务显然应该在另外的某些地方统一处理，而不应该在一个**ViewController**上。

2. 由于一些历史原因，连接页的视图控制器**connectingViewController**包含了三个**childViewController**：**WiFiViewController**、**3GViewController**、**errorViewController**，分别在**WiFi**状态、**3G**状态和出错状态下展示界面（三选一，其中一个展示的时候其它两个视图会隐藏）。

3. 大部分**view**都是直接加载完的。有些界面的加载非常复杂，比如再进入**App**时会展示一个检查**WiFi**可用性和安全性的动画，由于需要叠加较多图片，这部分视图的加载耗时较多。

由于随着几次改版之后，连接页的UI架构已经变得很不合理，历史包袱还是比较重的，而且耦合比较严重，几乎无法改动，因此决定重构。至于tabBarController，检查代码后决定简单的把不相关的业务做一些迁移，优化childViewController的加载过程，不作重构。

由于本篇主要讲启动性能优化，重构涉及的软件工程和设计模式方面的东西就不详细论述了，对启动优化的过程，主要是使用了更合理的分层结构，使得启动得以在更短的时间内完成。
至此，WiFi管家的启动性能基本优化完毕。

7. 挖掘最后一点性能优化
由于WiFi管家是一个具有WiFi连接能力的App，因此有可能在后台过程中完成

冷启动过程（实际上是在用户进入系统的WiFi设置时，iOS会启动WiFi管家，以便请求WiFi密码）。在这种情况下，整个rootViewController都是不需要加载的。

【第四部分】总结

- 利用DYLD_PRINT_STATISTICS分析main()函数之前的耗时
  - 重新梳理架构，减少动态库、ObjC类的数目，减少Category的数目
  - 定期扫描不再使用的动态库、类、函数，例如每两个迭代一次
  - 用dispatchonce()代替所有的__attribute__((constructor))函数、C++静态对象初始化、ObjC的+load
  - 在设计师可接受的范围内压缩图片的大小，会有意外收获
- 利用锚点分析applicationWillFinishLaunching的耗时
  - 将不需要马上在applicationWillFinishLaunching执行的代码延后执行
  - rootViewController的加载，适当将某一级的childViewController或subviews延后加载
  - 如果你的App可能会被后台拉起并冷启动，可考虑不加载rootViewController
- 不应放过的一些小细节
  - 异步操作并不影响指标，但有可能影响交互体验，例如大量网络请求导致数据拥堵
  - 有时候一些交互上的优化比技术手段效果更明显，视觉上的快决不是冰冷的数据可以解释的，好好和你们的设计师谈谈动画

7.0x8badf00d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。

8.怎么防止反编译?

1.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

2.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

3.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

4.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

5.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

6.借助第三方APP加固，例如：网易云易盾

9.说说你遇到到的技术难点?

你遇到的问题难度，一定程度决定了你的水平。如实反映就好。只是一道承上启下的问题!

以下问题是群友提供的:

问题1.地理空间距离计算优化

不管是“离我最近”还是“智能排序”，都涉及到计算用户位置与各个团购单子或者商家的距离（注：在智能排序中距离作为一个重要的参数参与排序打分）。以筛选商家为例，北京地区有5~6w个POI（本文将商家称之为POI），当用户进入商家页，请求北京全城+所有品类+离我最近/智能排序时，我们筛选服务需要计算一遍用户位置与北京全城所有POI的距离。

这种大量计算距离的场景十分消耗资源，从测试来看目前5w个点仅计算一遍距离就需要7ms，而到100w的时候就需要140多ms，随着数据的快速增长，筛选服务的性能将会非常堪忧。

如何优化距离的计算，进而提高计算速度、降低cpu使用率已经迫在眉睫。移动后台团购组在此方向上进行了些许探索，下文将分为4部分展开：1）地理空间距离计算原理；2）Lucene使用的距离计算公式；3）优化方案；4）实际应用。

2 地理空间距离计算原理

地理空间距离计算方法较多，目前我们使用的可以分为两类：1）球面模型，这种模型将地球看成一个标准球体，球面上两点之间的最短距离即大圆弧长，这种方法使用较广，在我们服务端被广泛使用；2）椭球模型，该模型最贴近真实地球，精度也最高，但计算较为复杂，目前客户端有在使用，但实际上我们的应用对精度的要求没有那么高。

下面着重介绍我们最常用的基于球面模型的地理空间距离计算公式，推导也只需要高中数学知识即可。

该模型将地球看成圆球，假设地球上有A(ja,wa)，B(jb,wb)两点（注：ja和jb分别是A和B的经度，wa和wb分别是A和B的纬度），A和B两点的球面距离就是AB的弧长，AB弧长=R*角AOB（注：角AOB是A跟B的夹角，O是地球的球心，R是地球半径，约为6367000米）。如何求出角AOB呢？可以先求AOB的最大边AB的长度，再根据余弦定律可以求夹角。

如何求出AB长度呢？

1）根据经纬度，以及地球半径R，将A、B两点的经纬度坐标转换成球体三维坐标；

$$A\begin{cases} Xa=RCos(wa)Cos(ja) \\ Ya=RCos(wa)Sin(ja) \\ Za=RSin(wa) \end{cases}$$

$$B\begin{cases} Xb=RCos(wb)Cos(jb) \\ Yb=RCos(wb)Sin(jb) \\ Zb=RSin(wb) \end{cases}$$

2）根据A、B两点的三维坐标求AB长度;

$$AB^2=(Xa-Xb)^2+(Ya-Yb)^2+(Za-Zb)^2$$
$$=...=2R^2(1-Cos(wa)Cos(wb)Cos(jb-ja)-Sin(wa)Sin(wb))$$

3）根据余弦定理求出角AOB;

$$AB^2=AO^2+BO^2-2AO*BO*Cos(\triangle AOB)$$
$$Cos(\triangle AOB)=\frac{AB^2-AO^2-BO^2}{-2AO*BO}=\frac{AB^2-2R^2}{-2R^2}=1-\frac{AB^2}{2R^2}$$

4）AB弧长=R*角AOB.

#3 Lucene使用的地理空间距离算法

团购app后台使用lucene来筛选团购单子和商家，lucene使用了spatial4j工具包来计算地理空间距离，而spatial4j提供了多种基于球面模型的地理空间距离的公式，其中一种就是上面我们推导的公式，称之为球面余弦公式。还有一种最常用的是Haversine公式，该公式是spatial4j计算距离的默认公式，本质上是球面余弦函数的一个变换，之前球面余弦公式中有cos(jb-ja)项，当系统的浮点运算精度不高时，在求算较近的两点间的距离时会有较大误差，Haversine方法进行了某种变换消除了cos(jb-ja)项，因此不存在短距离求算时对系统计算精度的过多顾虑的问题。

1）Haversine公式代码

```
public static double distHaversineRAD(double lat1, double
lon1, double lat2, double lon2) {
        double hsinX = Math.sin((lon1 - lon2) * 0.5);
        double hsinY = Math.sin((lat1 - lat2) * 0.5);
        double h = hsinY * hsinY +
                (Math.cos(lat1) * Math.cos(lat2) * hsinX
* hsinX);
        return 2 * Math.atan2(Math.sqrt(h), Math.sqrt(1 -
h)) * 6367000;
    }
```

2）Haversine公式性能

目前北京地区在线服务有5w个POI，计算一遍距离需要7ms。现在数据增长特别快，未来北京地区POI数目增大到100w时，我们筛选服务仅计算距离这一项就需要消耗144多ms，性能十分堪忧。（注：本文测试环境的处理器为2.9GHz Intel Core i7，内存为8GB 1600 MHz DDR3，操作系统为OS X10.8.3，实验在单线程环境下运行）

POI数目	耗时（ms）
5w	7
10w	14
100w	144

#4 优化方案

通过抓栈我们发现消耗cpu较多的线程很多都在执行距离公式中的三角函数例如atan2等。因此我们的优化方向很直接---消减甚至消除三角函数。基于此方向我们尝试了多种方法，下文将一一介绍。

##4.1 坐标转换方法

1）基本思路

之前POI保存的是经纬度（lon,lat），我们的计算场景是计算用户位置与所有筛选出来的poi的距离，这里会涉及到大量三角函数计算。一种优化思路是POI数据不保存经纬度而保存球面模型下的三维坐标（x,y,z），映射方法如下：

$$x = Math.cos(lat)\ Math.cos(lon);$$

$$y = Math.cos(lat)\ Math.sin(lon);$$

$$z = Math.sin(lat);$$

那么当我们求夹角AOB时，只需要做一次点乘操作。比如求(lon1,lat1）和 (lon2,lat2）的夹角，只需要计算x1x2 + y1y2 + z1*z2，这样避免了大量三角函数的计算。

在得到夹角之后，还需要执行arccos函数，将其转换成角度，AB弧长=角AOB*R（R是地球半径）。

此方法性能如下：

POI数目	耗时（ms）
5w	3
10w	8
100w	88

2）进一步优化

我们的业务场景是在一个城市范围内进行距离计算，因此夹角AOB往往会比较小，这个时候sinAOB约等于AOB，因此我们可以将Math.acos(cosAOB)R 改为Math.sqrt(1 - cosAOBcosAOB)*R，从而完全避免使用三角函数，优化后性能如下。

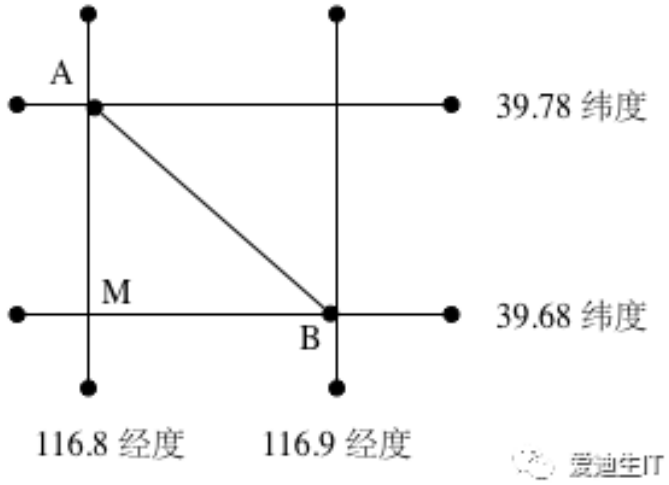
POI数目	耗时（ms）
5w	0.2
10w	0.5
100w	4

##4.2 简化距离计算公式方法

1）基本思路

我们的业务场景仅仅是在一个城市范围内进行距离计算，也就是说两个点之间的距离一般不会超过200多千米。由于范围小，可以认为

经线和纬线是垂直的，如图所示，要求A（116.8，39,78）和B（116.9，39.68）两点的距离，我们可以先求出南北方向距离AM，然后求出东西方向距离BM，最后求矩形对角线距离，即sqrt(AMAM + BMBM)。



南北方向AM = R纬度差Math.PI/180.0;

东西方向BM = R经度差Cos<当地纬度数* Math.PI/180.0>

这种方式仅仅需要计算一次cos函数。

```
public static double distanceSimplify(double lat1, double
lng1, double lat2, double lng2, double[] a) {

     double dx = lng1 - lng2; // 经度差值

     double dy = lat1 - lat2; // 纬度差值

     double b = (lat1 + lat2) / 2.0; // 平均纬度
```

```
    double Lx = toRadians(dx) * 6367000.0*
Math.cos(toRadians(b)); // 东西距离

    double Ly = 6367000.0 * toRadians(dy); // 南北距离

    return Math.sqrt(Lx * Lx + Ly * Ly);  // 用平面的矩形
对角距离公式计算总距离

    }
}
```

我们对这个方法的有效性和性能进行验证。

1.1）有效性验证

我们首先检验这种简化是否能满足我们应用的精度，如果精度较差将不能用于实际生产环境。

我们的方法叫distanceSimplify，lucene的方法叫distHaversineRAD。下表是在不同尺度下两个方法的相差情况。

测试点对	distanceSimplify（米）	distHaversineRAD（米）	差别（米）
（39.941，116.45）（39.94，116.451）	140.0285167225230	140.02851671981400	0.0
（39.96，116.45）（39.94，116.40）	4804.421262839180	4804.421153907680	0.0
（39.96，116.45）（39.94，117.30）	72444.81551882200	72444.54071519510	0.3
（39.26，115.25）（41.04，117.30）	263525.6167839860	263508.55921886700	17.1

可以看到两者在百米、千米尺度上几乎没有差别，在万米尺度上也仅有分米的差别，此外由于我们的业务是在一个城市范围内进行筛选排序，所以我们选择了北京左下角和右上角两点进行比较，两点相距有260多千米，两个方法差别17m。从精度上看该优化方法能满足我们应用需求。

1.2）性能验证

POI数目	耗时（ms）
5w	0.5
10w	1.1
100w	11

2）进一步优化

我们看到这里计算了一次cos这一三角函数，如果我们能消除此三角函数，那么将进一步提高计算效率。

如何消除cos三角函数呢?

采用多项式来拟合cos三角函数，这样不就可以将三角函数转换为加减乘除了嘛！

首先决定多项式的最高次数，次数为1和2显然都无法很好拟合cos函数，那么我们选择3先尝试吧，注：最高次数不是越多越好，次数越高会产生过拟合问题。

使用org.apache.commons.math3这一数学工具包来进行拟合。中国的纬度范围在10~60之间，即我们将此区间离散成Length份作为我们的训练集。

```
public static double[] trainPolyFit(int degree, int
Length){
    PolynomialCurveFitter polynomialCurveFitter =
PolynomialCurveFitter.create(degree);
    double minLat = 10.0; //中国最低纬度
    double maxLat = 60.0; //中国最高纬度
    double interv = (maxLat - minLat) / (double)Length;
    List<WeightedObservedPoint> weightedObservedPoints =
new ArrayList<WeightedObservedPoint>();
    for(int i = 0; i < Length; i++) {
```

```
        WeightedObservedPoint weightedObservedPoint = new
WeightedObservedPoint(1,  minLat + (double)i*interv,
Math.cos(toRadians(x[i])));

weightedObservedPoints.add(weightedObservedPoint);
    }
    return
polynomialCurveFitter.fit(weightedObservedPoints);
}
```

```
public static double distanceSimplifyMore(double lat1,
double lng1, double lat2, double lng2, double[] a) {
     //1) 计算三个参数
     double dx = lng1 - lng2; // 经度差值
     double dy = lat1 - lat2; // 纬度差值
     double b = (lat1 + lat2) / 2.0; // 平均纬度
     //2) 计算东西方向距离和南北方向距离(单位：米)，东西距离采用三
阶多项式
     double Lx = (a[3] * b*b*b  + a[2]* b*b  +a[1] * b +
a[0] ) * toRadians(dx) * 6367000.0; // 东西距离
     double Ly = 6367000.0 * toRadians(dy); // 南北距离
     //3) 用平面的矩形对角距离公式计算总距离
     return Math.sqrt(Lx * Lx + Ly * Ly);
```

```
}
```

我们对此优化方法进行有效性和性能验证。

2.1）有效性验证

我们的优化方法叫distanceSimplifyMore，lucene的方法叫distHaversineRAD，下表是在不同尺度下两个方法的相差情况。

测试点对	distanceSimplifyMore（米）	distHaversineRAD（米）	差别（米）
（39.941，116.45）（39.94，116.451）	140.0242769266660	140.02851671981400	0.0
（39.96，116.45）（39.94，116.40）	4804.113098854450	4804.421153907680	0.3
（39.96，116.45）（39.94，117.30）	72438.90919479560	72444.54071519510	5.6
（39.26，115.25）（41.04，117.30）	263516.676171262	263508.559218867000	8.1

可以看到在百米尺度上两者几乎未有差别，在千米尺度上仅有分米的区别，在更高尺度上如72千米仅有5.6m米别，在264千米也仅有8.1米区别，因此该优化方法的精度能满足我们的应用需求。

2.2）性能验证

POI数目	耗时（ms）
5w	0.1
10w	0.3
100w	4

#5 实际应用

坐标转换方法和简化距离公式方法性能都非常高，相比lucene使用的Haversine算法大大提高了计算效率，然而坐标转换方法存在一些缺点：

a）坐标转换后的数据不能被直接用于空间索引。lucene可以直接对经纬度进行geohash空间索引，而通过空间转换变成三维数据后不能直接使用。我们的应用有附近范围筛选功能（例如附近5km的团购单子），通过geohash空间索引可以提高范围筛选的效率；

b）坐标转换方法增大内存开销。我们会将坐标写入倒排索引中，之前坐标是2列（经度和纬度），现在变成3列（x,y,z），在使用中我们往往会将这数据放入到cache中，因此会增大内存开销；

c）坐标转换方法增大建索引开销。此方法本质上是将计算从查询阶段放至到索引阶段，因此提高了建索引的开销。

基于上述原因我们在实际应用中采用简化距离公式方法（通过三次多项式来拟合cos三角函数），此方法在团购筛选和商家筛选的距离排序、智能排序中已经开始使用，与之前相比，筛选团购时北京全城美食品类距离排序响应时间从40ms下降为20ms。

问题2.iOS应用架构，我的架构设计?

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BU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

10.说说你了解的第三方原理或底层知识?

Runtime、Runloop、block

SD原理、YYCache、GCD源码分析、JSPatch原理等！

以上面试题，部分答案来自群主笔记！仅供参考！

面试题＋学习视频持续更新，扫描下方二维码获取最新消息，谢谢大家的支持！