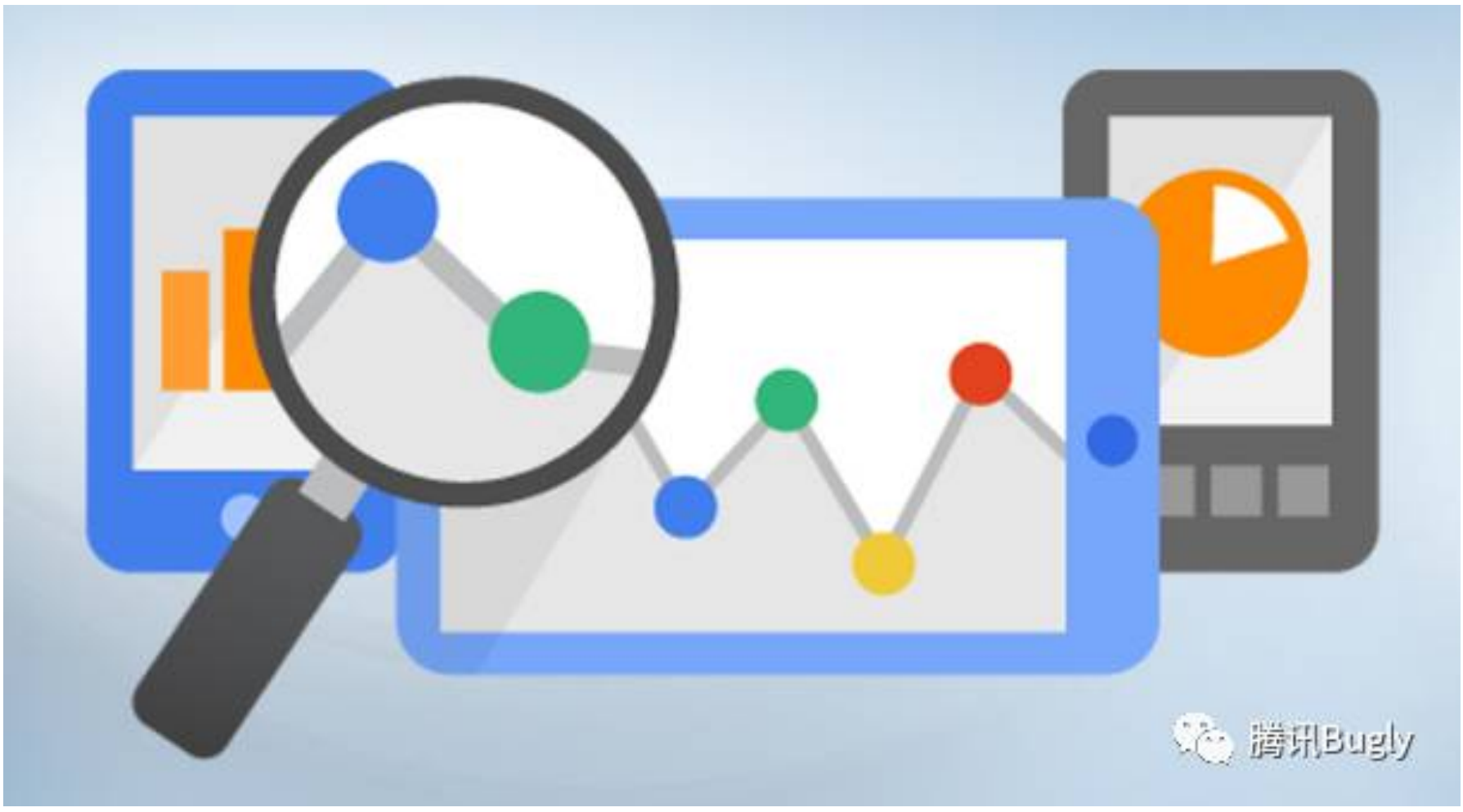
iOS App 启动性能优化

原创 2017-08-10 samsonxu 腾讯Bugly



导语

本文介绍了如何优化 iOS App 的启动性能，分为四个部分：

- 第一部分科普了一些和App启动性能相关的前置知识
- 第二部分主要讲如何定制启动性能的优化目标
- 第三部分通过在WiFi管家这个具体项目的优化过程，分享一些有用的经验
- 第四部分是关键点的总结。

【第一部分】一些小科普

因为篇幅的限制，没有办法很详尽的说明一些原理性的东西，只是方便大家了解哪些事情可能跟启动性能有关。同时，内容相对也比较入门，大神们请跳过这一部分。

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
  - 调用 `main()`
  - 调用 `UIApplicationMain()`
  - 调用 `applicationWillFinishLaunching`

2. 如何测量启动过程耗时

冷启动比热启动重要

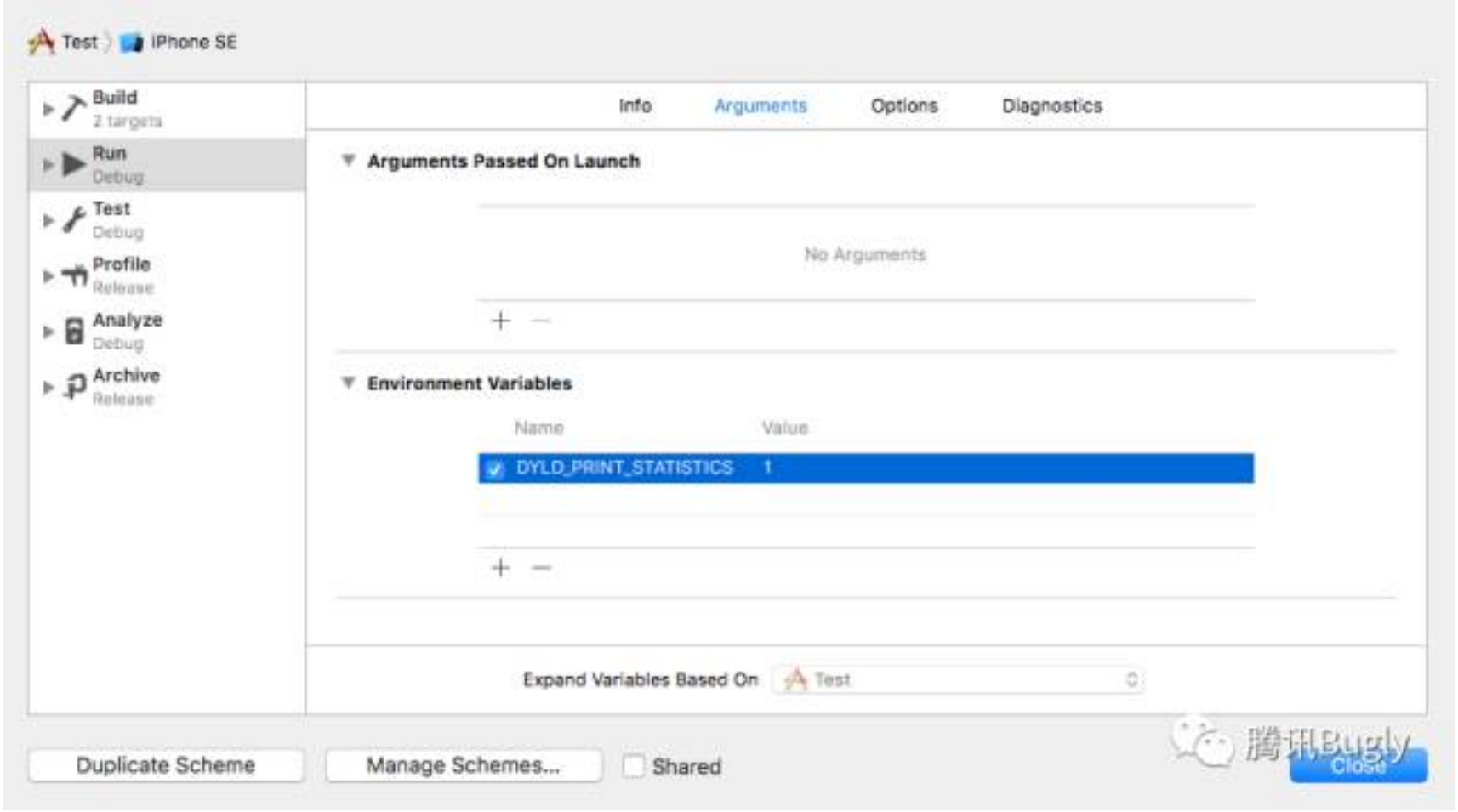
当用户按下home键的时候，iOS的App并不会马上被kill掉，还会继续存活若干时间。理想情况下，用户点击App的图标再次回来的时候，App几乎不需要做什么，就可以还原到退出前的状态，继续为用户服务。这种持续存活的情况下启动App，我们称为热启动，相对而言冷启动就是App被kill掉以后一切从头开始启动的过程。我们这里只讨论App冷启动的情况。

main()函数之前

在不越狱的情况下，以往很难精确的测量在main()函数之前的启动耗时，因而我们也往往容易忽略掉这部分数据。小型App确实不需要太过关注这部分。但如果是大型App（自定义的动态库超过50个、或编译结果二进制文件超过30MB），这部分耗时将会变得突出。所幸，苹果已经在Xcode中加入这部分的支持。

苹果提供的方法

- 在Xcode的菜单中选择 `Project` → `Scheme` → `Edit Scheme...`，然后找到 `Run` → `Environment Variables` → `+`，添加name为 `DYLD_PRINT_STATISTICSvalue` 为 `1` 的环境变量。



- 在Xcode运行App时，会在console中得到一个报告。例如，我在WiFi管家中加入以上设置之后，会得到这样一个报告：

```
Total pre-main time:  94.33 milliseconds (100.0%)
         dylib loading time:  61.87 milliseconds (65.5%)
        rebase/binding time:   3.09 milliseconds (3.2%)
            ObjC setup time:  10.78 milliseconds (11.4%)
           initializer time:  18.50 milliseconds (19.6%)
           slowest intializers :
             libSystem.B.dylib :   3.59 milliseconds (3.8%)
    libBacktraceRecording.dylib :   3.65 milliseconds (3.8%)
                    GTFreeWifi :   7.09 milliseconds (7.5%)
```

如何解读

1. main()函数之前总共使用了94.33ms
2. 在94.33ms中，加载动态库用了61.87ms，指针重定位使用了3.09ms，ObjC类初始化使用了10.78ms，各种初始化使用了18.50ms。
3. 在初始化耗费的18.50ms中，用时最多的三个初始化是libSystem.B.dylib、libBacktraceRecording.dylib以及GTFreeWifi。

main()函数之后

从 `main()` 函数开始至 `applicationWillFinishLaunching` 结束，我们统一称为main()函数之后的部分。

3. 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：

```
//AppDelegate.m
@implementation AppDelegate

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)lau
nchOptions {
    self.rootViewController = [[[MQQTabBarController alloc] init] autorelease];

    self.window = [[[UIWindow alloc] init] autorelease];
    [self.window makeKeyAndVisible];
    self.window.rootViewController = self.rootViewController;
```

```
    UITabBarController *tabBarViewController = [[[UITabBarController alloc] init] autorelease];

    NSLog(@"%s", __PRETTY_FUNCTION__);
    return YES;
}

...

//MQQTabBarController.m
@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];
}

...
@end
```

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

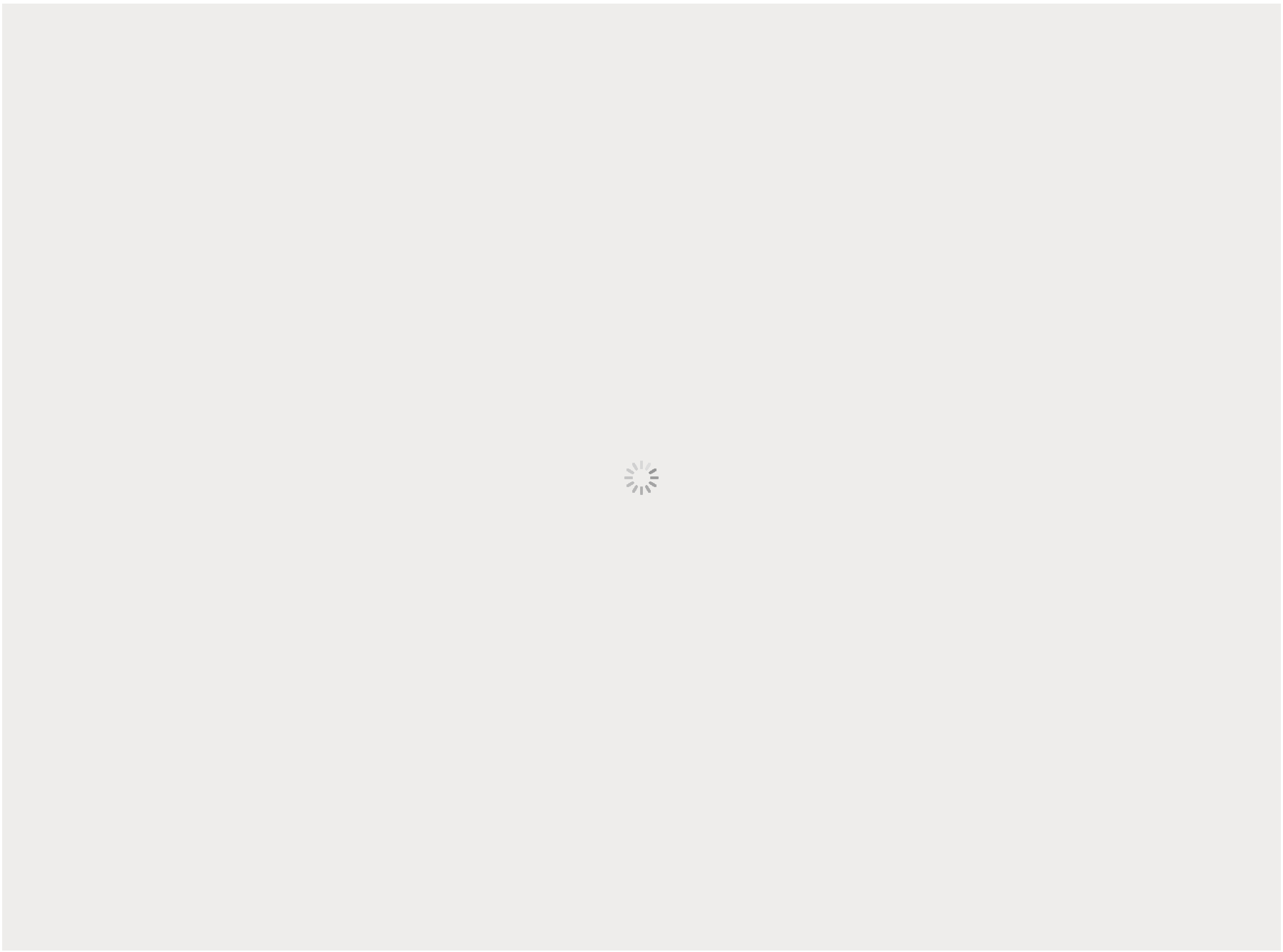
答案是:

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：

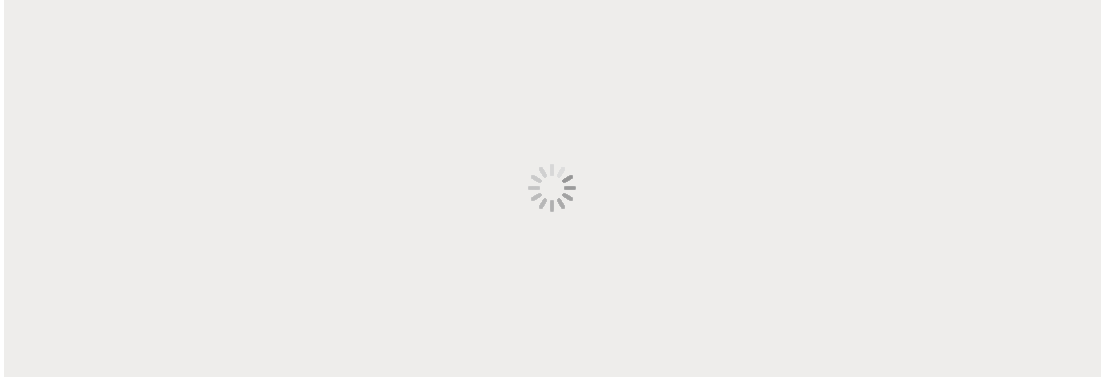
1. main()函数之前
2. main()函数之后至applicationWillFinishLaunching完成
3. App完成所有本地数据的加载并将相应的信息展示给用户
4. App完成所有联网数据的加载并将相应的信息展示给用户

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”。

【第三部分】WiFi管家启动优化实践

先show一下成果：



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

考虑到我作为一只高级程序猴，工资很高，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WiFi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。

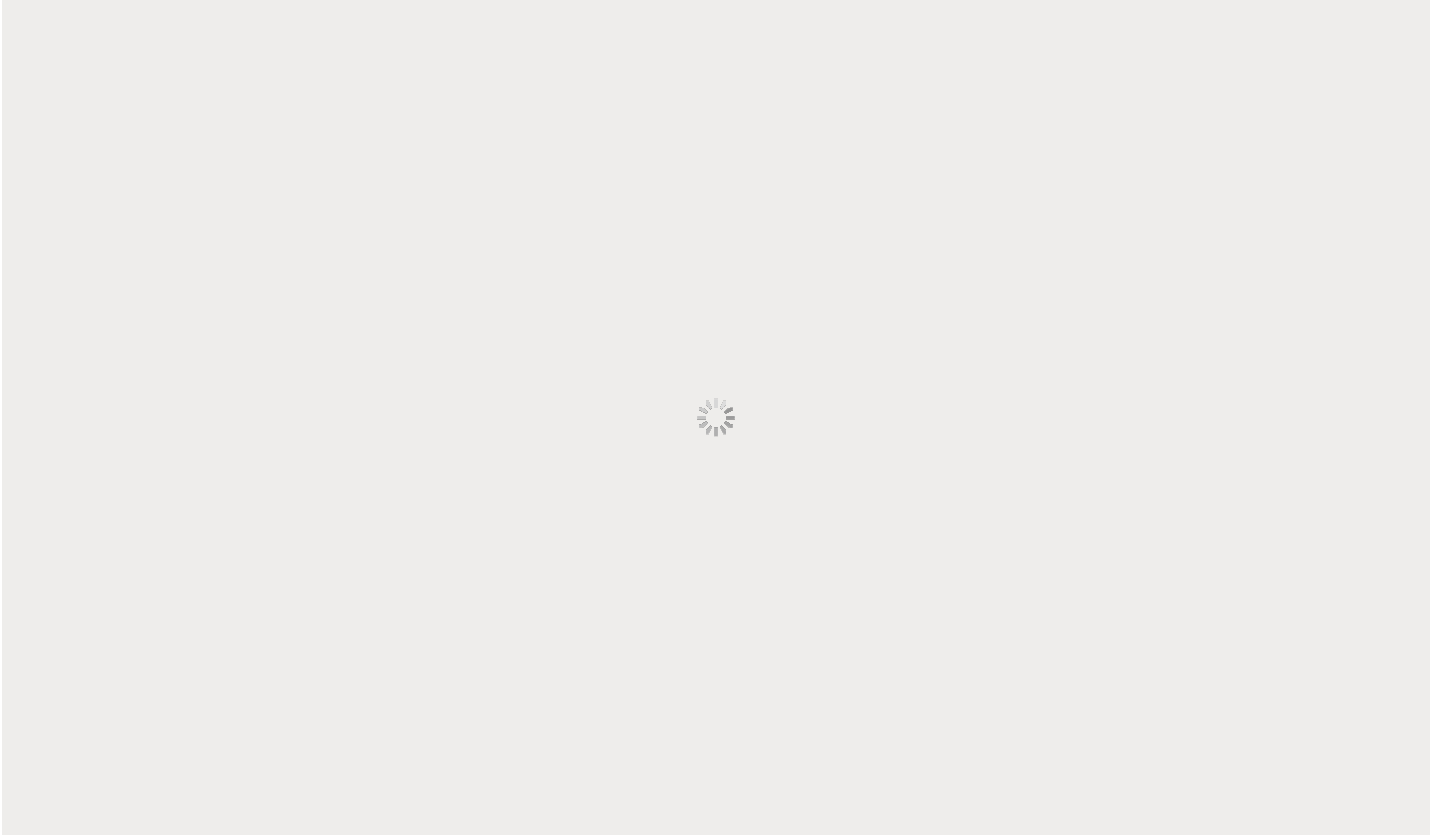
UI构架请看如下示意图，其中蓝色的部分需要在App启动的时候立即加载：

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。

改进后的结构大致如下图，其中蓝色部分需要在App启动的时候立即加载：



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。

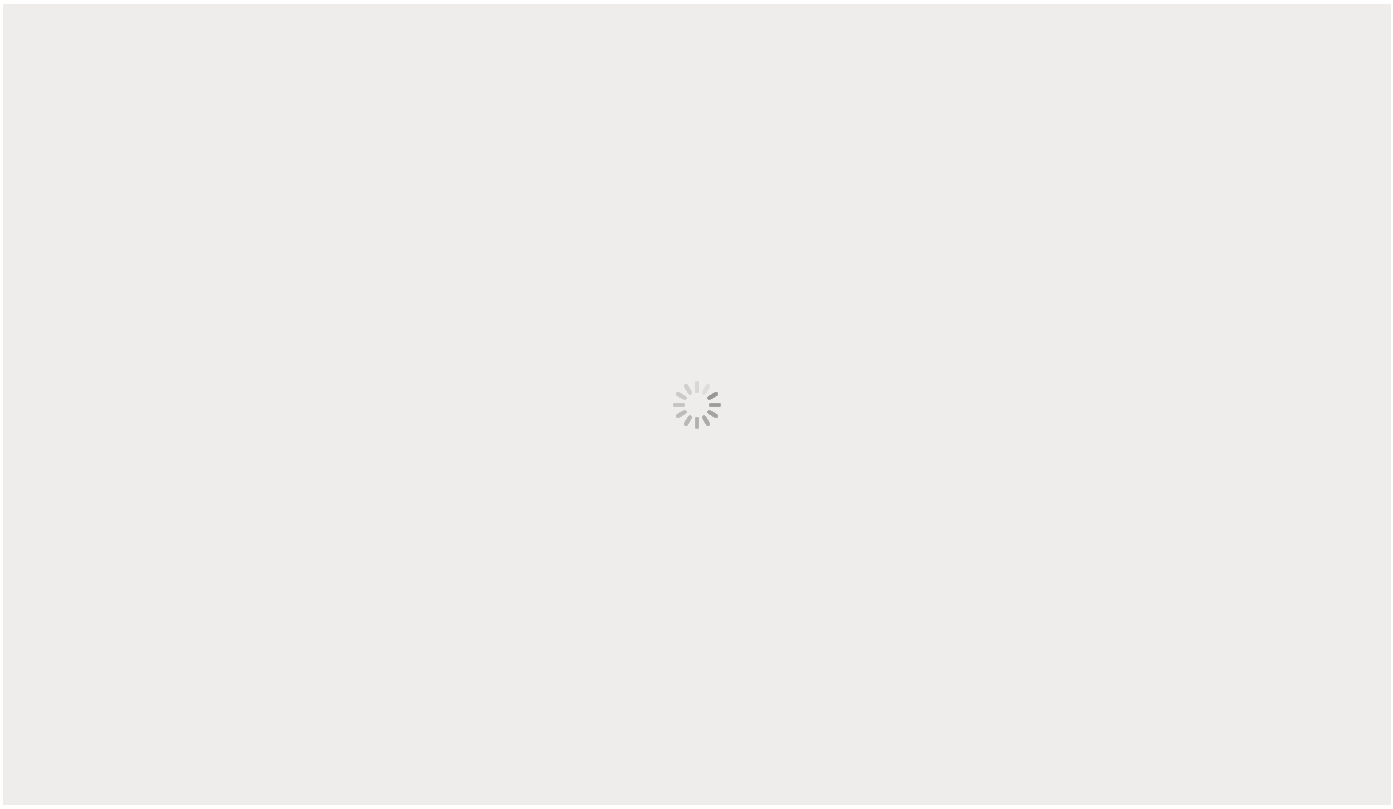
【第四部分】总结

- 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



阅读原文