2017-01-17

今日头条iOS客户端启动速度优化

应用启动时间，直接影响用户对一款应用的判断和使用体验。头条主app本身就包含非常多并且复杂度高的业务模块（如新闻、视频等），也接入了很多第三方的插件，这势必会拖慢应用的启动时间，本着精益求精的态度和对用户体验的追求，我们希望在业务扩张的同时最大程度的优化启动时间。

技术调研

先说结论，t(App总启动时间) = t1(main()之前的加载时间) + t2(main()之后的加载时间)。 t1 = 系统dylib(动态链接库)和自身App可执行文件的加载；
t2 = main方法执行之后到AppDelegate类中的- `(BOOL)Application:(UIApplication *)Application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions`方法执行结束前这段时间，主要是构建第一个界面，并完成渲染展示。

main()调用之前的加载过程

App开始启动后，系统首先加载可执行文件（自身App的所有.o文件的集合），然后加载动态链接库dyld，dyld是一个专门用来加载动态链接库的库。 执行从dyld开始，dyld从可执行文件的依赖开始，递归加载所有的依赖动态链接库。
动态链接库包括：iOS 中用到的所有系统 framework，加载OC runtime方法的libobjc，系统级别的libSystem，例如libdispatch(GCD)和libsystem_blocks (Block)。

其实无论对于系统的动态链接库还是对于App本身的可执行文件而言，他们都算是image（镜像），而每个App都是以image(镜像)为单位进行加载的，那么image究竟包括哪些呢?

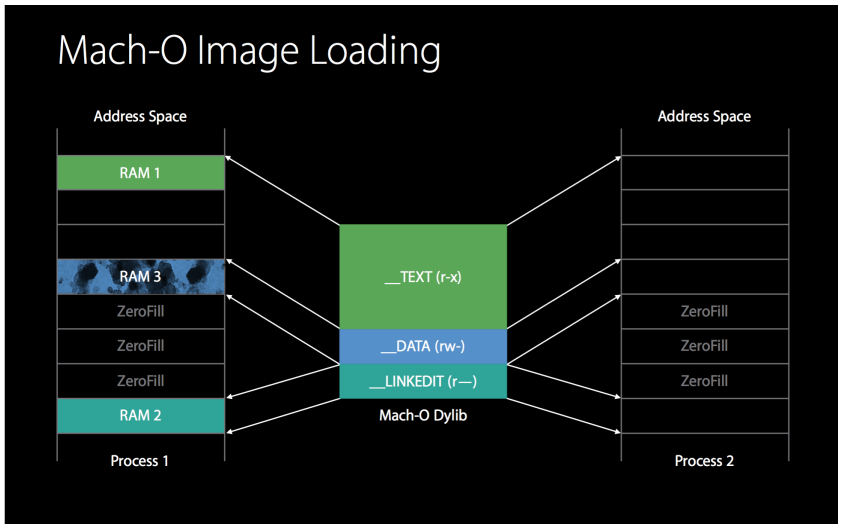
什么是image

1.executable可执行文件 比如.o文件。
2.dylib 动态链接库 framework就是动态链接库和相应资源包含在一起的一个文件夹结构。
3.bundle 资源文件 只能用dlopen加载，不推荐使用这种方式加载。

除了我们App本身的可行性文件，系统中所有的framework比如UIKit、Foundation等都是以动态链接库的方式集成进App中的。

系统使用动态链接有几点好处：

代码共用：很多程序都动态链接了这些 lib，但它们在内存和磁盘中中只有一份。 易于维护：由于被依赖的 lib 是程序执行时才链接的，所以这些 lib 很容易做更新，比如libSystem.dylib 是 libSystem.B.dylib 的替身，哪天想升级直接换成libSystem.C.dylib 然后再替换替身就行了。 减少可执行文件体积：相比静态链接，动态链接在编译时不需要打进去，所以可执行文件的体积要小很多。



如上图所示，不同进程之间共用系统dylib的`_TEXT`区，但是各自维护对应的`_DATA`区。

所有动态链接库和我们App中的静态库.a和所有类文件编译后的.o文件最终都是由dyld(the dynamic link editor)，Apple的动态链接器来加载到内存中。每个image都是由一个叫做ImageLoader的类来负责加载（一一对应），那么ImageLoader又是什么呢?

什么是ImageLoader

image 表示一个二进制文件(可执行文件或 so 文件)，里面是被编译过的符号、代码等，所以 ImageLoader 作用是将这些文件加载进内存，且每一个文件对应一个ImageLoader实例来负责加载。两步走： 在程序运行时它先将动态链接的 image 递归加载 (也就是上面测试栈中一串的递归调用的时刻)。 再从可执行文件 image 递归加载所有符号。

当然所有这些都发生在我们真正的main函数执行前。

动态链接库加载的具体流程

动态链接库的加载步骤具体分为5步：

1. load dylibs image 读取库镜像文件
2. Rebase image
3. Bind image
4. Objc setup
5. initializers

load dylibs image

在每个动态库的加载过程中， dyld需要：

1. 分析所依赖的动态库
2. 找到动态库的mach-o文件
3. 打开文件
4. 验证文件
5. 在系统核心注册文件签名
6. 对动态库的每一个segment调用mmap()

通常的，一个App需要加载100到400个dylibs， 但是其中的系统库被优化，可以很快的加载。 针对这一步骤的优化有：

1. 减少非系统库的依赖
2. 合并非系统库
3. 使用静态资源，比如把代码加入主程序

rebase/bind

由于ASLR(address space layout randomization)的存在，可执行文件和动态链接库在虚拟内存中的加载地址每次启动都不固定，所以需要这2步来修复镜像中的资源指针，来指向正确的地址。 rebase修复的是指向当前镜像内部的资源指针； 而bind指向的是镜像外部的资源指针。
rebase步骤先进行，需要把镜像读入内存，并以page为单位进行加密验证，保证不会被篡改，所以这一步的瓶颈在IO。bind在其后进行，由于要查询符号表，来指向跨镜像的资源，加上在rebase阶段，镜像已被读入和加密验证，所以这一步的瓶颈在于CPU计算。
通过命令行可以查看相关的资源指针:

> xcrun dyldinfo -rebase -bind -lazy_bind myApp.App/myApp

优化该阶段的关键在于减少__DATA segment中的指针数量。我们可以优化的点有：

1. 减少Objc类数量， 减少selector数量
2. 减少C++虚函数数量
3. 转而使用swift stuct（其实本质上就是为了减少符号的数量）

Objc setup

这一步主要工作是:

1. 注册Objc类 (class registration)
2. 把category的定义插入方法列表 (category registration)
3. 保证每一个selector唯一 (selctor uniquing)

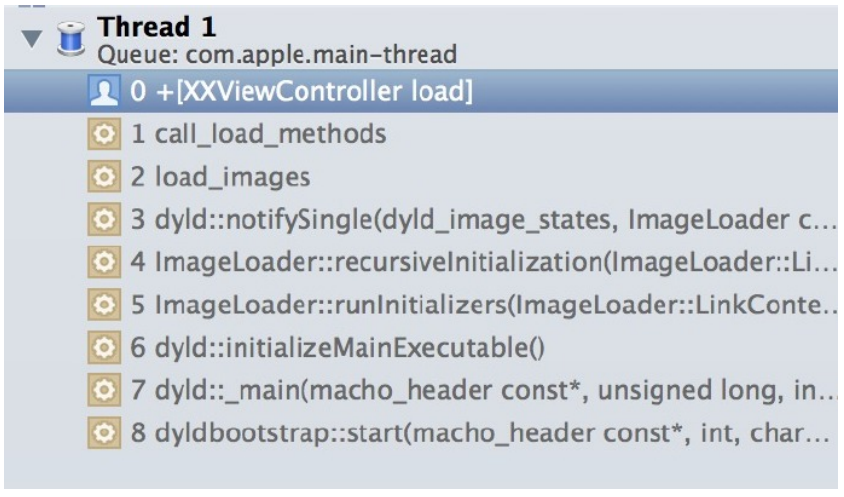
由于之前2步骤的优化，这一步实际上没有什么可做的。

initializers

以上三步属于静态调整(fix-up)，都是在修改__DATA segment中的内容，而这里则开始动态调整，开始在堆和堆栈中写入内容。 在这里的工作有：

1. Objc的+load()函数
2. C++的构造函数属性函数 形如**attribute**((constructor)) void DoSomeInitializationWork()
3. 非基本类型的C++静态全局变量的创建(通常是类或结构体)(non-trivial initializer) 比如一个全局静态结构体的构建，如果在构造函数中有繁重的工作，那么会拖慢启动速度

Objc的load函数和C++的静态构造函数采用由底向上的方式执行，来保证每个执行的方法，都可以找到所依赖的动态库。



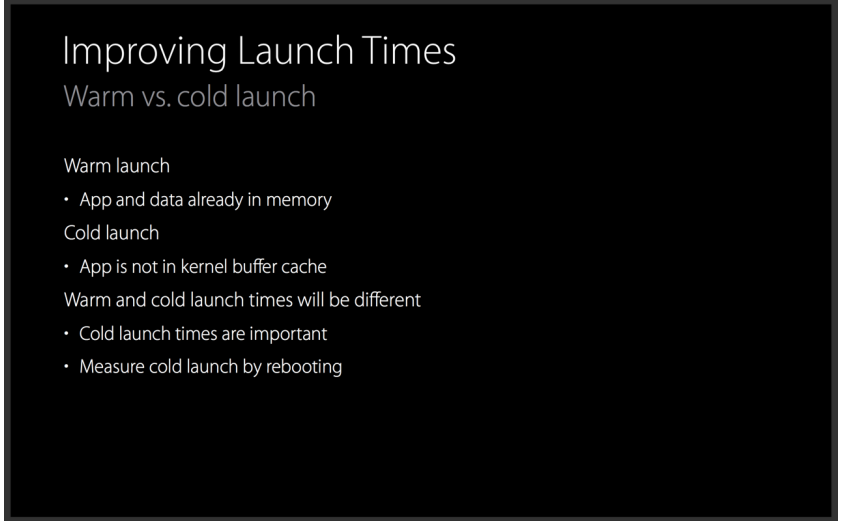
上图是在自定义的类XXViewController的+load方法断点的调用堆栈，清楚的看到整个调用栈和顺序：

1. dyld 开始将程序二进制文件初始化
2. 交由 ImageLoader 读取 image，其中包含了我们的类、方法等各种符号
3. 由于 runtime 向 dyld 绑定了回调，当 image 加载到内存后，dyld 会通知 runtime 进行处理
4. runtime 接手后调用 map*images 做解析和处理，接下来 load*images 中调用 call*load*methods 方法，遍历所有加载进来的 Class，按继承层级依次调用 Class 的 +load 方法和其 Category 的 +load 方法

至此，可执行文件中和动态库所有的符号(Class，Protocol，Selector，IMP，…)都已经按格式成功加载到内存中，被 runtime 所管理，再这之后，runtime 的那些方法(动态添加 Class、swizzle 等等才能生效)。

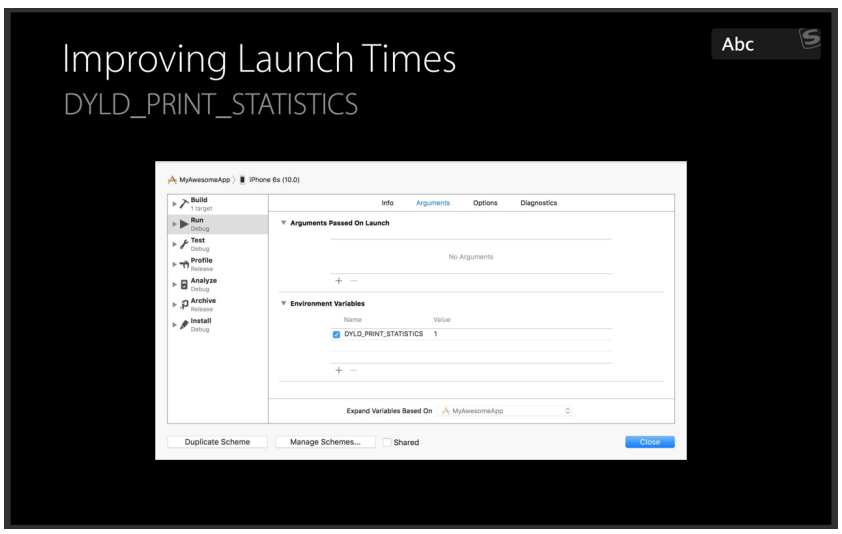
整个事件由 dyld 主导，完成运行环境的初始化后，配合 ImageLoader 将二进制文件按格式加载到内存， 动态链接依赖库，并由 runtime 负责加载成 objc 定义的结构，所有初始化工作结束后，dyld 调用真正的 main 函数。

如果程序刚刚被运行过，那么程序的代码会被dyld缓存，因此即使杀掉进程再次重启加载时间也会相对快一点，如果长时间没有启动或者当前dyld的缓存已经被其他应用占据，那么这次启动所花费的时间就要长一点，这就分别是热启动和冷启动的概念，如下图所示:

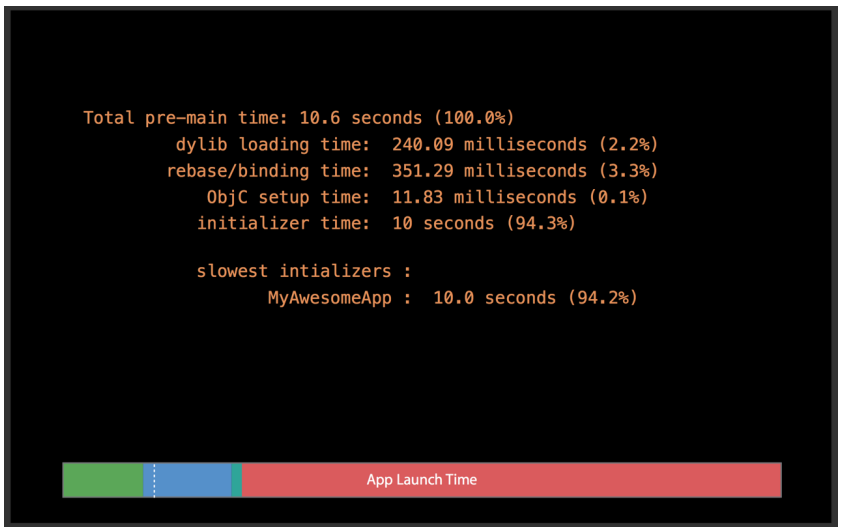


main()之前的加载时间如何衡量

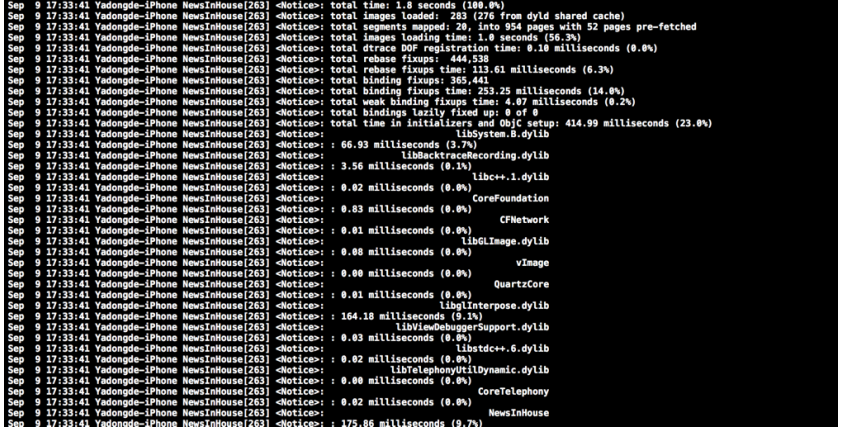
那么问题就来了，那怎么衡量main()之前也就是time1的耗时呢，苹果官方提供了一种方法，那就是在真机调试的时候勾选dyld*PRINT*STATISTICS选项。



会得到如下形式的输出：



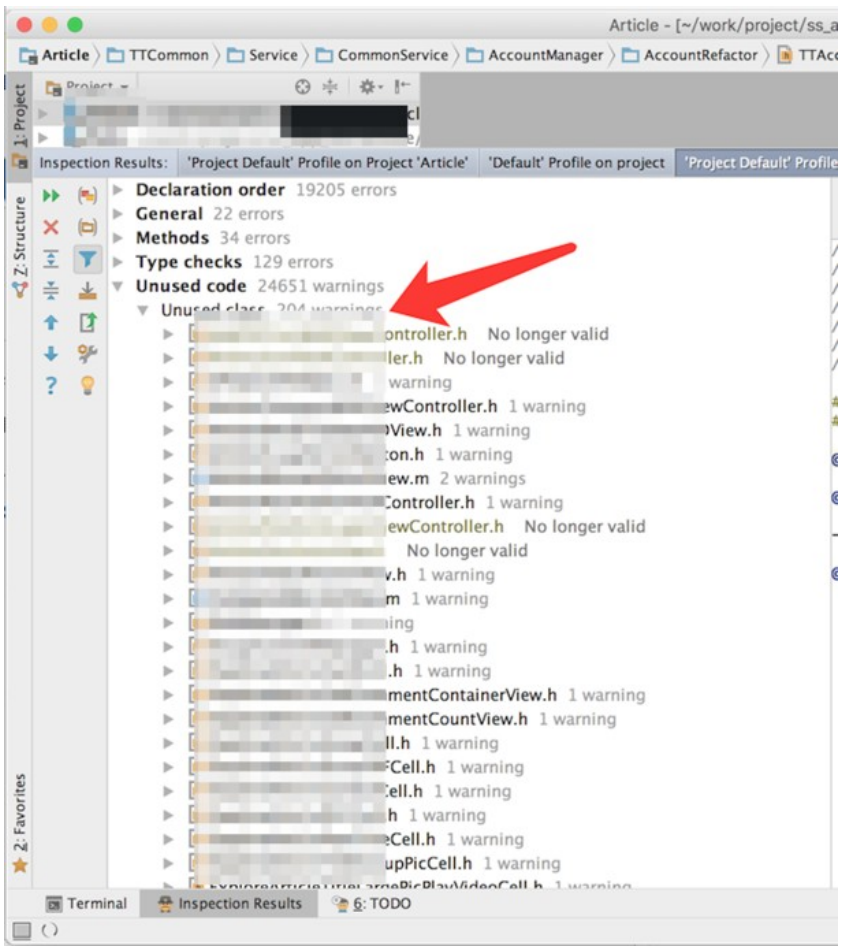
由此可见对于系统级别的动态链接库，因为苹果做了优化，所以耗时并不多，在这个awesome的例子中，自身App中的代码占用了整体时间的94.2% 我们应用中一次典型的Log如下：



由此可见，最多的用时还是在image加载和OC类的初始化，共占用总时长的79.3%，精简framework的引入和OC类有优化的空间。

总结一下：对于main()调用之前的耗时我们可以优化的点有：

1. 减少不必要的framework，因为动态链接比较耗时
2. check framework应当设为optional和required，如果该framework在当前App支持的所有iOS系统版本都存在，那么就设为required，否则就设为optional，因为optional会有些额外的检查
3. 合并或者删减一些OC类，关于清理项目中没用到的类，使用工具AppCode代码检查功能，查到当前项目中没有用到的类如下：



1. 删减一些无用的静态变量

2. 删减没有被调用到或者已经废弃的方法

方法见：

```
http://stackoverflow.com/questions/35233564/how-to-find-unused-code-in-xcod
https://developer.Apple.com/library/ios/documentation/ToolsLanguages/Concep
```

3. 将不必须在+load方法中做的事情延迟到+initialize中

4. 尽量不要用C++虚函数(创建虚函数表有开销)

main()调用之后的加载时间

在main()被调用之后，App的主要工作就是初始化必要的服务，显示首页内容等。而我们的优化也是围绕如何能够快速展现首页来开展。 App通常在AppDelegate类中的- (BOOL)Application:(UIApplication *)Application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions方法中创建首页需要展示的view，然后在当前runloop的末尾，主动调用CA::Transaction::commit完成视图的渲染。
而视图的渲染主要涉及三个阶段：

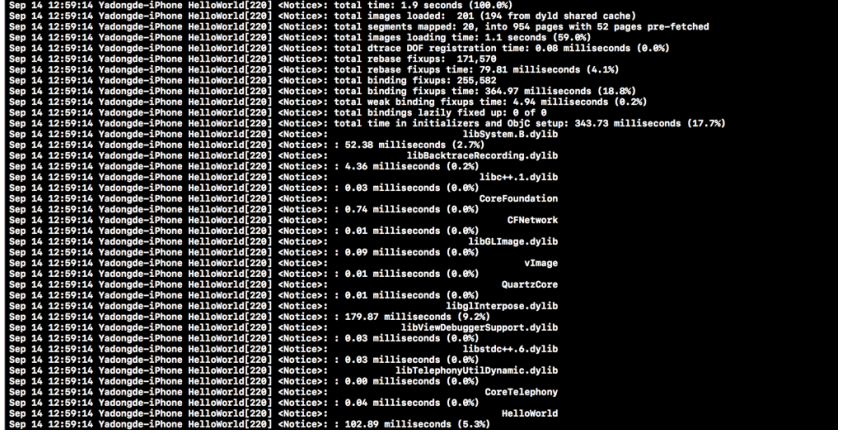
1. 准备阶段 这里主要是图片的解码
2. 布局阶段 首页所有UIView的- (void)layoutSubViews()运行
3. 绘制阶段 首页所有UIView的- (void)drawRect:(CGRect)rect运行
   再加上启动之后必要服务的启动、必要数据的创建和读取，这些就是我们可以尝试优化的地方

因此，对于main()函数调用之前我们可以优化的点有：

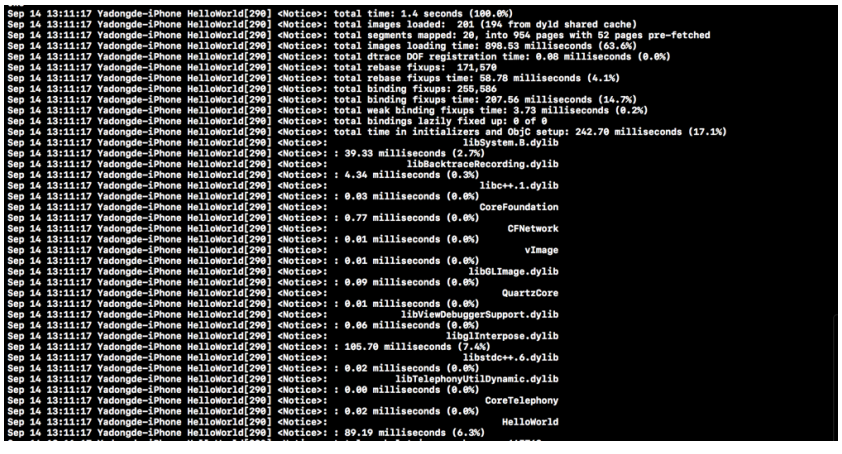
1. 不使用xib，直接视用代码加载首页视图
2. NSUserDefaults实际上是在Library文件夹下会生产一个plist文件，如果文件太大的话一次能读取到内存中可能很耗时，这个影响需要评估，如果耗时很大的话需要拆分(需考虑老版本覆盖安装兼容问题)
3. 每次用NSLog方式打印会隐式的创建一个Calendar，因此需要删减启动时各业务方打的log，或者仅仅针对内测版输出log
4. 梳理应用启动时发送的所有网络请求，是否可以统一在异步线程请求

实测数据

建立了一个空的HelloWorld工程，只加入了pods中的代码，不包含主端的业务逻辑代码，一次典型的冷启动基本接近2s iPhone6 iOS9.3.5系统测试主要时间在加载动态库，类/方法的初始化还有符号地址绑定阶段。



一次典型的热启动数据如下：可以看到因为系统做了缓存方面的优化，比冷启动快了500ms加上头条主端业务逻辑代码之后一次典型的热启动耗时2.1s。



以上用时均为main()之前的加载耗时。

main()函数之后加载时间优化记录

NSUserDefaults是否是瓶颈

苹果官方文档提到NSUserDefaults加载的时候是整个plist配置文件全部load到内存中，目前头条主端当中NSUserDefaults存储了200多项缓存数据，因此怀疑可能拖慢启动速度，但是测试结果显示并不会。 通过符号断点+[NSUserDefaults standardUserDefaults]确定最早一次的+load()从执行到结束耗时1.8ms，可见NSUserDefaults的初始化仅耗时1.8ms，并不是启动耗时的瓶颈。

如何找到拖慢启动应用时长的瓶颈

为了找到瓶颈，我们在启动之后的didFinishLauhcning方法开始执行到首页列表页的NewsListViewController的viewDidAppear方法，几乎每个可能比较耗时的流程进行拆分和统计，得到统计数据之后发现： 主要耗时在首页UI构造和渲染(storyboard加载，tabBar/topBar渲染，开屏广告加载/cell注册/日志模块初始化这几个步骤)。

具体优化点

因此，针对于今日头条这个App我们可以优化的点如下：

1. 纯代码方式而不是storyboard加载首页UI。
2. 对didFinishLaunching里的函数考虑能否挖掘可以延迟加载或者懒加载，需要与各个业务方pm

和rd共同check 对于一些已经下线的业务，删减冗余代码。
对于一些与UI展示无关的业务，如微博认证过期检查、图片最大缓存空间设置等做延迟加载

3. 对实现了+load()方法的类进行分析，尽量将load里的代码延后调用。
4. 上面统计数据显示展示feed的导航控制器页面(NewsListViewController)比较耗时，对于viewDidLoad以及viewWillAppear方法中尽量去尝试少做，晚做，不做。

优化结果

之前曾经有一位同事已经做了一定的优化，比如启动之后展示闪屏广告图的同时初始化首页的列表页，当广告展示完成之后列表页也就渲染完成了。经过这一次优化之后的main()之后的启动总时长通过上线之后收集数据的验证达到了预期的效果。

性能优化 iOS 更新日期:2017-01-17

标签

- 性能优化
- iOS
- 通用技术
- 前端
- 问题追查
- 服务端
- Android
- WWDC2017
- DATA

近期文章

- Weex在内涵发现页中的工程实践
- 多域名微信授权方案
- Android非UI线程更新UI的探索
- 光流基础概念
- 给前端工程师讲设计终篇

最新评论