

IOS工程治理

Created and last modified by 严超6 大约6小时以前

- 一、背景&目标
- 二、问题&现状
 - 工程架构
 - 沙盒存储
 - 历史代码
 - 资源文件
 - 提供给RN、Flutter基础能力缺少统一
 - 启动管控
 - OOM问题
- 三、竞品分析
- 四、工程组件化改造
 - 4.1 组件模块划分
 - 4.2 理清组件依赖关系
 - 4.3 壳工程轻量化
 - 4.3.1 资源治理
 - 4.3.2 Target治理
 - 4.3.3 主工程三方库
 - 4.3.4 初始化类CVideoGoAppDelegate过于臃肿
 - 4.4 同层组件解耦方案
 - 4.5 组件二进制化
 - 4.5.1 动态库还是静态库？
 - 4.5.2 IOT构建静态库模式
 - 4.5.3 调研组件二进制化方案
 - 4.6 防腐化
 - 组件依赖关系防腐化
 - 1、提升本地编译效率
 - 2、组件定时构建二进制
 - 3、新MCI系统构建卡口
 - 资源重复防腐化
 - 1、图片资源
- 五、其他治理
 - 5.1 包体积治理
 - 5.1.1 资源治理
 - 5.1.2 代码的治理
 - 5.1.3 包体积监控
 - 5.2 沙盒存储管理
 - 5.3 RN、Flutter基建能力建设
 - 现有原生提供给RN能力缺少文档，代码中很多能力缺少注释说明，需要建立一个长期维护保障机制
 - RNBridge文档
 - 5.4 启动流程治理
 - 5.4.1 竞品分析
 - 5.4.2 Pre-main 阶段和main() 阶段优化方案
 - 工程中动态库过多（77个）
 - 动态库懒加载
 - 二进制重排
 - 4.5.3 after-main 阶段的优化方案
 - 4.5.4 建立监控机制
 - 4.6 耗电治理
 - 4.8 OOM，占用内存过高治理
 - 4.8.1 OMM
 - 4.8.2 一些开源工具
 - 4.8.3 大厂自研OOM治理
- 六、行动步骤

一、背景&目标

萤石云app经过近10年的业务发展，已经是一个中大型app。

几十个开发人员同时在工程中进行开发，很多研发人员吐槽工程架构太大，每次开发都需要拉取所有的代码，代码量过大导致编译速度慢、工程结构也不清晰、祖传代码、多技术栈开发基建能力弱，还有种种问题，严重影响开发效率；

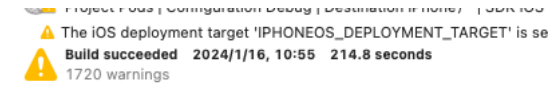
萤石云工程进行治理变成了非常紧急且必要的事情，在新的一年里，希望治理好萤石云的工程，提高内部开发人员对萤石云工程的满意度。并且对每一项治理工作都做到长期方案，形成闭环。

二、问题&现状

工程架构

- 非组件化工程，全源码开发模式，工程相当于只是做了文件夹分离，就是形式上的分离，区分主工程仓和组件仓，只要申请仓库权限，任何人在工程中都可以代码随意修改，也可以修改工程配置相关。
- 每个项目都在一个大工程中开发，需要在菊豆平台拉取所有工程中的组件，60+个数组件带来合并分支的长时间等待。
- 研发提交代码没有任何的校验，经常出现提交代码有误，导致其他人checkout出有问题代码导致相互影响，或者导致App包构建失败。
- 代码量大，全源码参与编译，首次编译和二次编译时间太长，对开发效率影响太大。
以interl5 macmin机器，萤石云6.14.0版本为例，首次编译时间为674秒左右，二次编译为214秒左右





- 5. 组件依赖关系混乱，没有任何的依赖关系卡口，所有的组件代码随意引用。
下层依赖上层（D层依赖C层、YS业务层），同层之间相互依赖（D层、C层、YS业务层相互依赖），跨层依赖（YS层依赖D层）等等
工程依赖如下：[萤石云组件依赖关系梳理](#)
- 6. 组件划分不明确，缺少命名规范，职责不明确，组件颗粒度不足，不清楚需要在哪个组件中进行代码开发。[IOS现有组件负责人整理](#)



- 比如：
- 7. 创建组件流程复杂，需要填写jira单等操作，新功能代码选择随意选择一个组件中编写，新组件创建层级是否合理也缺少卡口。
 - 8. 项目发布后，主工程代码冲突频繁，主工程改动造成代码冲突合并一般是一人合并存在风险； 组件分支冲突，目前是手动合并代码到trunk上，缺少合并后到trunk的检验机制。

沙盒存储

- 1. 沙盒存储混乱，有些文件直接存在根目录下，调用方没有控制存储的期限和存储最大空间限制
- 2. 存在一些遗留的存储的数据文件，没有使用，也无法清除掉。
- 3. 没有一套标准的沙盒存储规范，导致任何人都可以随意对沙盒数据进行操作。

历史代码

- 1. 历史代码删除不干净，页面不使用或者用RN、Flutter替换掉了代码未删除重复代码，完全看开发人员自觉，缺少校验机制
- 2. 业务调用量很少的业务代码过多存在，或者A/B Test方案后，需要通过业务埋点等方式梳理出那些客户使用量很少的业务与产品经理协商去除
- 3. 对历史代码不敢删除，对业务理解不清楚，担心删除会引起其他问题等
- 4. 相同功能三方库引入缺乏管控，比如YYImageKit和SDWebImage；重复代码未删除，[IOS重复文件治理](#)

资源文件

- 1. 不使用的图片资源未及时删除，后续更加不会删除了
- 2. 工程中存在过大资源文件未云端化
- 3. 资源文件重复问题
- 4. 现在包体积监控还是半人工的方式，通过手动跑一下脚本输出包体积变化

提供给RN、Flutter基础能力缺少统一

- 1. 现有提供给RN、Flutter的能力无法统一查看，比如有个写RN或者Flutter代码的人想知道原生提供过什么能力，是未知的
- 2. 提供的方法中存在一个功能多个类似方法等情况，并且缺少方法注释，调用方使用的时候处于懵逼的状态

启动管控

- 1. 简单地全部堆积到didFinishLaunchingWithOptions方法中，代码臃肿而混乱
- 2. 启动流程管控缺失，任何人都可以在启动流程中增加代码，缺少监控机制和合理性评审
- 3. 目前只有启动耗时

OOM问题

- 1. bugly信息有限，并且当我们在调试阶段遇到这种崩溃的时候，从设备设置->隐私->分析与改进中是找不到普通类型的崩溃日志，所以难以定位产生问题的原因
- 2. 没有线上监控OOM崩溃以及内存异常增长的有效方案
- 3. 开发过程中没有提前预防的功能和措施
- 4. 现在萤石云中只有首页和预览页单独的内存占用统计，整个app的内存监控和其他业务的内存情况是缺失的。

三、竞品分析

	公司	备注	目录结构
https://juejin.cn/post/7008796749598834702	阿里	Alibaba iOS 工程架构腐化治理实践	<ul style="list-style-type: none">背景架构腐化会产生哪些问题？<ul style="list-style-type: none">问题一：模块打包复杂度高<ul style="list-style-type: none">工程环境混杂环境不兼容&模块构建失败每年浪费了90人日的开发资源问题二：主工程打包慢问题三：工程环境不稳定问题四：Swift开发寸步难行问题五：历史代码清理困难架构腐化治理的困难与策略<ul style="list-style-type: none">影响范围广，治理难推动数据化分析，自顶向下推动解决方案<ul style="list-style-type: none">纵观全局，理清模块依赖关系依赖倒置、分层治理自动化修复架构和业务合作治理长效保障机制<ul style="list-style-type: none">架构优化

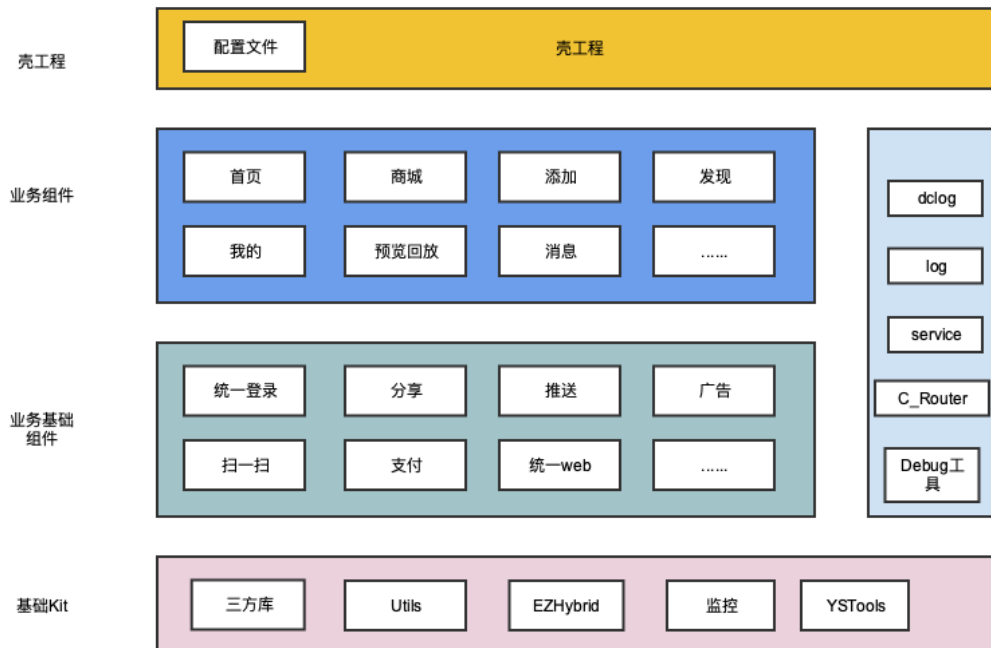
			<ul style="list-style-type: none">收敛模块工程CocoaPod和Xcode编译卡口Devops构建卡口
https://mp.weixin.qq.com/s/znfSzA-_CFsEb59ZfPcZJg	网易	网易严选APP工程架构演进	<p>2. 什么是APP架构-想做什么？</p> <p>2.1 APP架构的标准</p> <p>2.2 APP架构的指导思想</p> <p>2.2.1 分层</p> <p>2.2.2 模块化</p> <p>3. 现状分析-为什么做？</p> <p>3.1 工程文件耦合</p> <p>3.1.1 主工程代码堆积</p> <p>3.1.2 历史代码清理困难</p> <p>3.1.3 集中式管理</p> <p>3.2 工程环境复杂</p> <p>3.2.1 业务耦合</p> <p>3.2.2 复杂度高</p> <p>4. 演进方案-如何做？</p> <p>4.1 工程优化</p> <p>4.1.1 纵观全局，理清模块依赖关系</p> <p>4.1.2 模块化、颗粒度解耦</p> <p>4.1.3 提高安全性，保障项目完整</p> <p>4.2 腐效治理</p> <p>4.2.1 主工程文件剥离</p> <p>4.2.2 无用/重复资源删除</p> <p>4.2.3 渐进化修复、自动化检查</p> <p>4.3 长效保障机制</p> <p>4.3.1 CI辅助分析并设置卡口</p> <p>4.3.2 基建能力搭建</p> <p>5. 方案落地-最佳实践</p>
https://www.hellobit.com.cn/doc/2021/12/28/560.html	得物	得物 iOS 工程演进之路	<ul style="list-style-type: none">工程演进规划工程健康包体积大小治理工程代码治理Crash 治理启动流程治理工程化围绕组件化的基础设施组件管理（包含组件创建、发版等组件工具）二进制（包含二进制调试工具、组件上游依赖查询工具、裙带源码组件切换工具）持续交付（包含编译成功节点切换工具）组件化IPO 模型为什么不使用基于注册的组件化方案组件拆分粒度小规模业务和团队中等规模业务和团队大规模业务和团队好的架构没有 Common 没有 Core，也不应该有大组件的存在为什么不允许存在公共模块为什么不允许存在大组件Argument List Too Long容器化视图渲染事件通信应用总结

四、工程组件化改造

主要目标如下：

- 1. 组件按需拉取
- 2. 理清组件依赖关系，组件提测
- 3. 防腐化

4.1 组件模块划分



FREE TRIAL

You have **1000 days left** in your [Gliffy Confluence Plugin](#) free trial
[Purchase a license](#) | Admins, enter your [Gliffy my.atlassian.com](#) license in the UPM

采用组件分层的治理，先从最底层的基础Kit开始。

4.2 理清组件依赖关系

所有梳理完成的组件podSpec文件中 s.dependency都是正确的。

1，自动化工具分析出组件间的依赖关系

- (1) 找出模块间的所有文件
- (2) 使用正则匹配找到import的外部头文件，得到所有外部引用头文件集合
- (3) 根据Pods目录，匹配出头文件所属的外部模块名称

2，组件依赖关系可视化

- (1) 将组件依赖关系通过可视化表现出来，更加清晰明了

3，组件依赖关系变化监控

- (1) 对组件依赖关系进行监控管理，对新增依赖关系需要进行审核

4.3 壳工程轻量化

4.3.1 资源治理

主工程先进行SVN提交锁死，禁止添加资源文件，权限只开放给给别人，并且对现有工程中图片进行梳理，将所有图片归纳到业务组件中

4.3.2 Target治理

YSAISiriExtension、YSAISiriExtensionUI、YSScreenRecordExtension、myServiceNoti、YSWidget五个工程中Target相关功能代码放置在各个Target中，每个Target的业务独立，不依赖外部相关代码

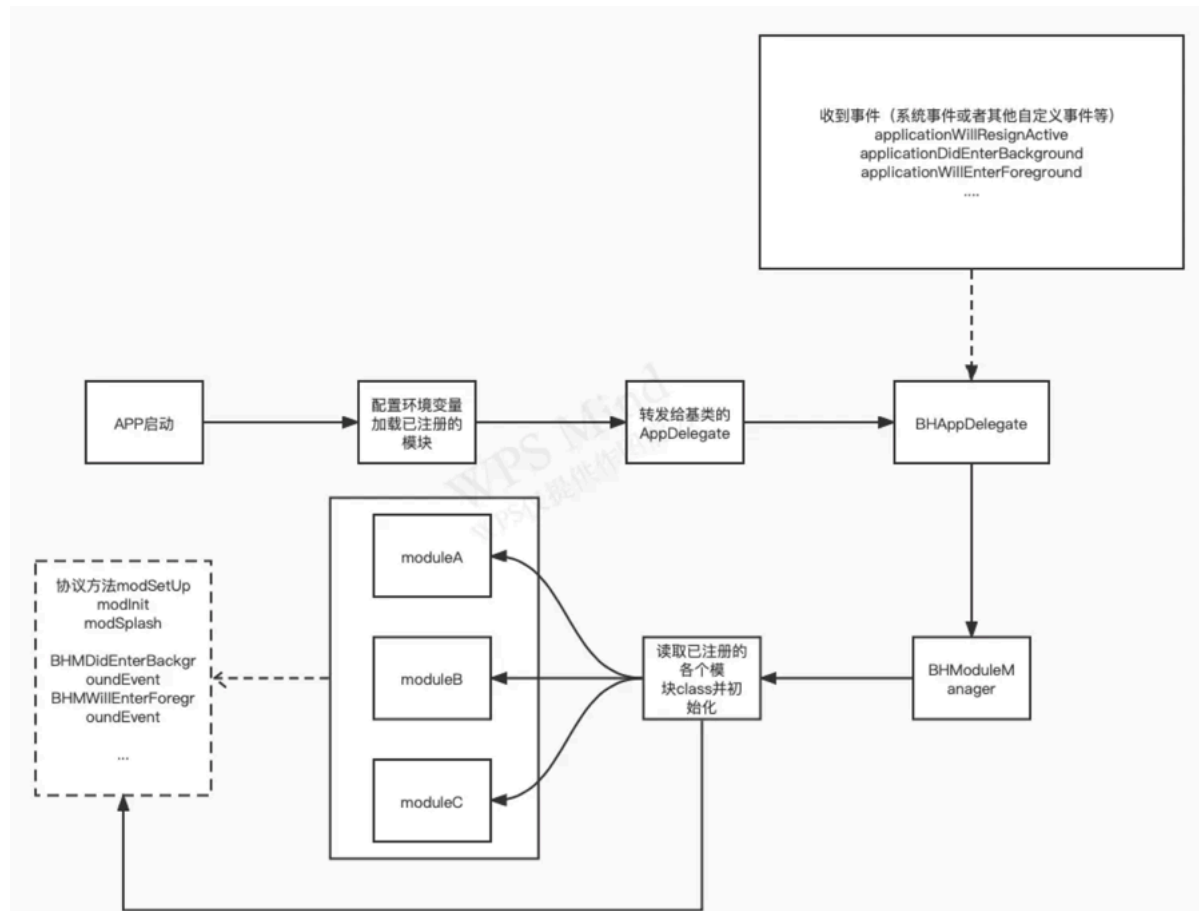
4.3.3 主工程三方库

主工程中不允许存放任何的三方库（EZDSDK.framework、TXFFmpeg.xcframework、TXSoundTouch.xcframework等），相关业务代码和库放到对应组件中

4.3.4 初始化类CVideoGoAppDelegate过于臃肿

阿里的Beehive的AppDelegate解耦方案：

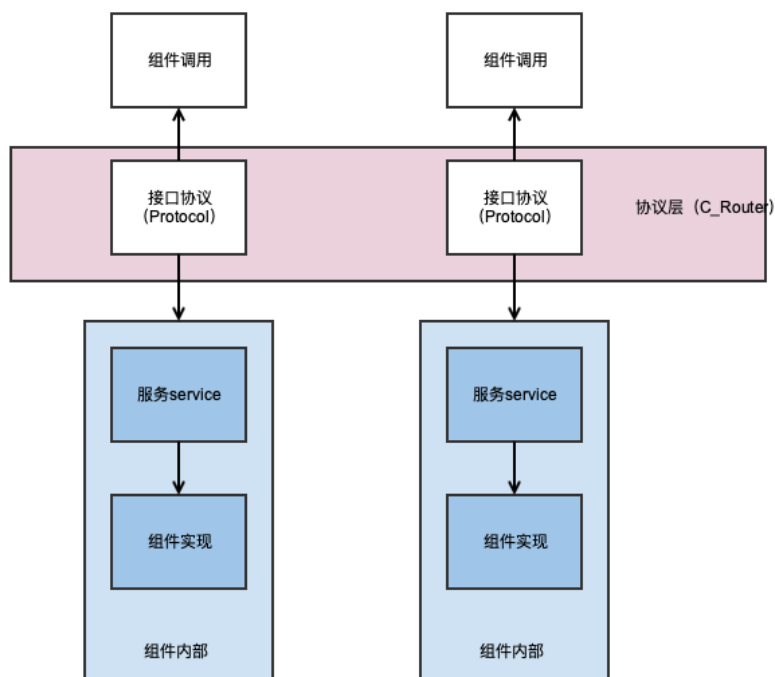
BHAppDelegate接管原来的AppDelegate，BeeHive可以监控Application生命周期事件，例如DidBecomeActive、WillEnterBackground等。



4.4 同层组件解耦方案

- URL Scheme (MGJRouter)
- Target - Action (CTMediator)
- Protocol - Class 匹配 (**BeeHive**) <https://blog.csdn.net/yezuiqingxin/article/details/126018623>

我们现有工程中使用的组件化方案是BeeHive的服务注册方案



FREE TRIAL

You have **1000 days** left in your **Gliffy Confluence Plugin** free trial
[Purchase a license](#) | Admins, enter your [Gliffy my.atlassian.com](#) license in the UPM

4.5 组件二进制化

4.5.1 动态库还是静态库?

组件二进制的实现就是打包成动态库/静态库，由于过多的动态库会导致启动速度减慢得不偿失，此外iOS对于动态库的表现形式只有framework，若想做源码与二进制切换时，引入头文件的地方也不得不进行更改。

例如：

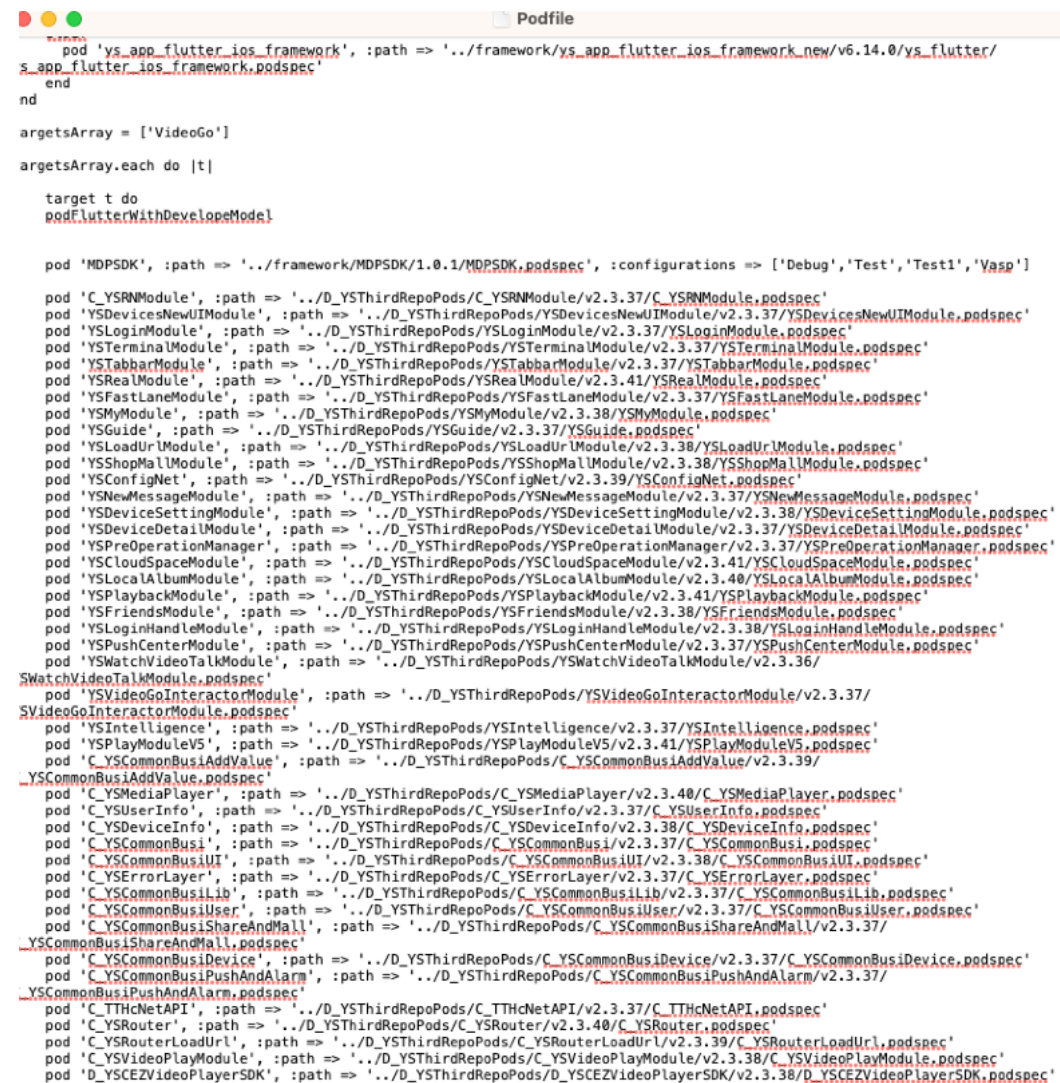
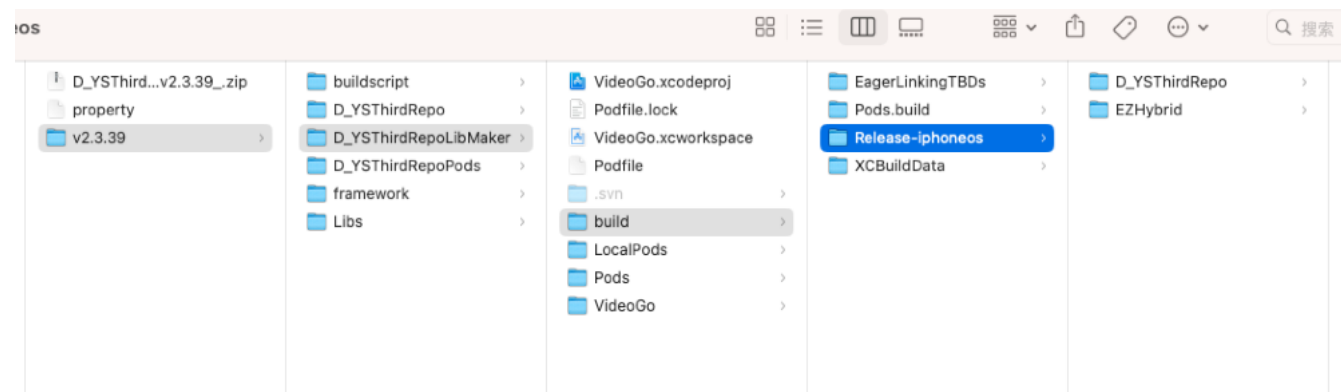
```
import <ABC.h> // 源码引用
import <ABCBinary/ABC.h> // 动态库引用
```

而打包成静态库.a文件（注意不要打包成framework形式）则不需要更改引用代码，

所以综上所述，我们选择打包成静态库的方式不需修改引用代码、缩小体积提升编译速度。

4.5.2 IOT构建静态库模式

1. checkout App主工程，根据工程中ini文件生成podfile文件，到SVN上拉出所有组件源码，执行pod install通过
2. checkout 下编译静态库工程，根据前一步的ini文件生成新的podfile文件到工程中
3. 工程中一个组件对应一个target，执行命令xcodebuild -project Pods.xcodeproj -target ' + _compoName_ + ' -UseModernBuildSystem=NO >>xx.log，编译这个对应的组件，会自动拉出组件依赖的组件进行编译



这种模式有什么问题呢？

- 1, 组件编译还是依赖原有整个开发工程, 编译前需要拉下整个工程的代码, 确保原有整个开发工程能正常编译通过才能正常构建组件包, 稳定性太差。
(由于未对构建脚本进行优化处理, 目前还是采用拉取全部组件的方式进行项目开发)

2, 组件依赖还是处于混乱的状态。

当然也有优点

- 1, 由于现有组件太多, 理清所有组件的关系需要的时间比较长, 所以可以先用这种方式快速保障组件的按需拉取。**后续再进一步优化。**

4.5.3 调研组件二进制化方案

下面列出研究过的一些主流方案以及最后没有采用的原因, 这些方案有各自的局限性, 但是也给了我不少启发, 思考过程跟最终方案一样有价值。

3.1、Carthage

Carthage可以将一部分不常变的库打包成framework, 再引如到主工程, 这样可以减少开发过程中的编译时间。Carthage 可以比较方便地调试源码。因为我们目前已经大规模使用CocoaPods, 转用 Carthage 来做包管理需要做大量的转换工作, 变动太大, 不满足我们的无侵入、无影响现有的业务, 所以不考虑这个方案了。

3.2、cocoapods-packager

cocoapods-packager 可以将任意的 pod 打包成 Static Library, 省去重复编译的时间, 一定程度上可以加快编译时间, 但是也有自身的问题:

- 优化不彻底, 只能优化第三方和私有 Pod 的编译速度, 对于其他改动频繁的业务代码无能为力
- 私有库和第三方库的后续更新很麻烦, 当有源码修改后, 需要重新打包上传到内部的 Git 仓库
- 过多的二进制文件会拖慢 Git 的操作速度 (目前还没部署 Git 的 [LFS](#))
- 难以调试源码, 不共享编译缓存
- 打包成 Static Library 过程缓慢, 需要通过pod lint, 各个组件间又层层嵌套依赖, 在现有阶段来说, 是难以实现的。

3.3、cocoapods-binary

Cocoapods-Binary (Cocoapods 官方推荐的二进制插件), 是一个**即时生成二进制包并缓存**, 而非像 CocoaPods-Packager 仅仅针对单个私有库的。原理是通过 CocoaPods 提供的 pre_install hook 在 pod install 的 prepare 阶段拦截到当前的 pod install context, 进而 fork 出一份独立的 installer 以完成将预编译源码 clone 至 Pod/_Prebuild 目录下, 同时也存在几个不足之处:

- 单私有源, 无法实现服务端缓存, 在没有对应二进制包版本时, pod install 后会额外去做二进制的生成, 一定程度上会影响 pod install 的速度。
- 开发者切回源码调试, 二进制缓存会一并清空, 需求重新编译。
- 多个项目、不同分支的相同组件依旧无法共享
- 只支持framework, 对我们项目现状需要比较大的头文件引用方式改动。

3.4、cocoapods-bin 双私有源

该插件进行二进制化的策略是采用双私有源, 即2个源地址, 一个静态服务器保存预先打好包的framework, 一个是我们现在保存源码的服务地址, 在install的时候去选择使用下载那个, 是个很不错的项目, 深受启发。

优点:

- 源码和二进制文件之间可以来回切换, 速度比较快
- 不影响未接入二进制化方案的业务团队
- 无二进制版本时, 自动采用源码版本
- 接近原生 CocoaPods 的使用体验

对于在我们项目中存在的不足之处:

- 不支持指定分支, :podspec => ".git" 方式的引用, 对需要支持多个分支、多个业务线的项目是致命的。
- Archive二进制文件时, 只能去spec仓库下载源码, 无法根据指定的分支去下载依赖库, 导致编译失败、错乱的问题
- 依赖的组件需要推送到spec仓库, 很多私有库并没有推送到仓库, 且对于频繁改动的私有库, 推送到仓库的verify很慢且与我们的开发习惯不符。
- 不支持a静态文件输出, 项目中大量类似 #import "IMYPublic.h" 需要一个一个库去编译替换为#import <IMYPublic/IMYPublic.h>, 想想那110多个组件库~
- 只支持一套环境, 对于有Debug/Release/Dev开发环境需求的无法满足
- 不支持二进制组件的源码调试
- 不能流畅的支持频繁变动的业务组件, 操作会异常繁琐。
- 针对于我们的项目, 目前存在较大的障碍, 无法使用起来。

拉取必要的壳工程和基本配置, 根据模块的依赖关系动态生成podfile文件进行组件构建
CocoaPods自动校验podSpec中依赖关系

```
inhibit_all_warnings!

platform:ios, '11.0'

install! 'cocoapods', :deterministic_uuids => false

targetsArray = ['VideoGo']

targetsArray.each do |t|

  target t do
    pod 'D_YSThirdRepo', :svn =>'https://192.0.0.241/Civil-Platform/Mobile/libraries/D_YSThirdRepo/v2.3.39'

  end

end

end

[Mac-mini:VideoGo yanchao$ pod install
Analyzing dependencies
[!] Unable to find a specification for `EZHybrid` depended upon by `D_YSThirdRepo`

You have either:
 * out-of-date source repos which you can update with `pod repo update` or with `pod install --repo-update`.
 * mistyped the name or version.
 * not added the source repo that hosts the Podspec to your Podfile.

[Mac-mini:VideoGo yanchao$
```

4.6 防腐化

组件依赖关系防腐化

1、提升本地编译效率

- (1) 本地编译工具xcode配置优化 [iOS萤石云项目本地二次编译速度提升方案\(针对低配\)](#)
- (2) 业务模块的组件依赖关系梳理清楚后, 只拉取业务模块依赖的相关组件进行开发, 加快本地编译速度
- (3) 不需要开发的依赖组件使用二进制库参与开发

2、组件定时构建二进制

- (1) 组件定时构建二进制，保证组件podSpec组件依赖正确
- (2) 所有组件都必须先编译二进制通过才能进入集成区，保障模块互相之间提交的代码不影响

3、新MCI系统构建卡口

- (1) CI流程中增加代码静态扫描、单元测试、代码审核等卡口，保障代码提交的稳定性
- (2) 增加通知机制，对全链路进行实时监控，劣化状态通知到责任人

资源重复防腐化

1、图片资源

五、其他治理

5.1 包体积治理

包体积大小 = 资源+代码，包体积大小主要由资源大小与代码大小组成

5.1.1 资源治理

- 1. 资源压缩：可以使用(TinyPNG压缩)进行有效压缩
- 2. 重复资源合并：由于组件化的存在，每个组件可能会存在自己使用到的图片，导致大量相似、甚至重复图片存在。通过使用脚本扫描出重复资源，然后手工对资源进行合并。
- 3. 云端下载：只要APP首次启动时不需要加载该资源，或者使用频率不高，或者业务中大文件，该资源就可以走云端下载；

5.1.2 代码的治理

三方库裁剪

可以通过我们对 LinkMap 进行了解析，从而找到使用的第三方库及具体使用的业务场景。

- 1. 对只需要三方库一小部分功能的场景，我们对三方库进行定制化缩减，如移除不需要的功能。
- 2. 针对为解决同类需求，引入多个相似第三方库的场景，限定只用某一个。
- 3. 三方库的多种架构进行裁剪。

无效代码去除

业务场景的不断迭代或者改造下架，会沉积大量的废弃的代码文件，有些在业务场景下不会被用到或者被使用到的很少，但是仍然会被编译打包进安装包中占用其体积，而且会持续的腐化项目。

对于工程内的代码文件资源，可以使用以下方案：

- 1. 基于Mach-O检测：WBBlades <https://github.com/wuba/WBBlades>
- 2. 基于源码检测：Fui
- 3. 使用工具检测：AppCode

重复代码合并

找由于复制粘贴产生的重复代码，然后抽出中间组件，各组件对中间组件进行依赖实现。

5.1.3 包体积监控

目前萤石云包体积监控的方案还是靠人工的方式进行输入，后续将输入接入到现有MCI流程中，完成全自动化建设

[萤石云APP体积控制方案](#)

5.2 沙盒存储管理

目前工程沙盒存储无权限管控，方案计划如下：

<http://nvwa.hikvision.com.cn/pages/viewpage.action?pageId=669390588>

5.3 RN、Flutter基建能力建设

现有原生提供给RN能力缺少文档，代码中很多能力缺少注释说明，需要建立一个长期维护保障机制

[RNBridge文档](#)

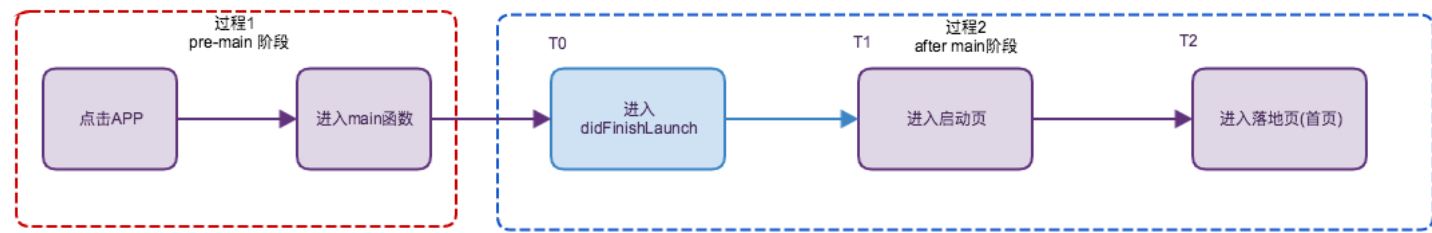
5.4 启动流程治理

5.4.1竞品分析

方案地址	大厂	备注
https://zhuanlan.zhihu.com/p/51838614	美团	

启动性能问题的治理目标:

- 1. 解决存量问题: 优化当前性能瓶颈点, 优化启动流程, 缩短冷启动时间。
- 2. 管控增量问题: 冷启动流程规范化, 通过代码范式和文档指导后续冷启动过程代码的维护, 控制时间增量。
- 3. 完善监控: 完善冷启动性能指标监控, 收集更详细的数据, 及时发现性能问题。



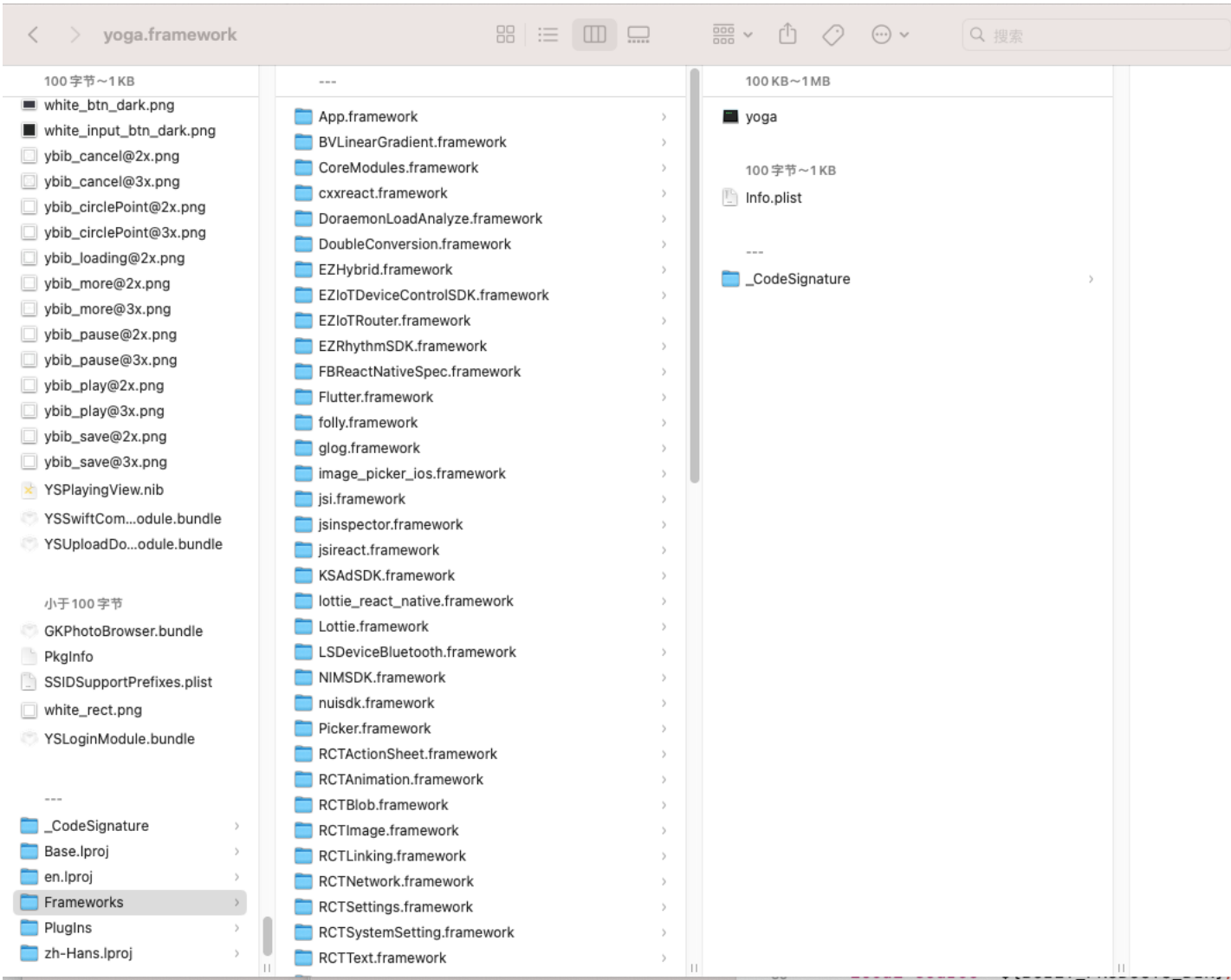
从以上的流程可以看到, 主要分为两个过程:

过程1 (pre-main) : 在调用main函数之前, 大部分的工作都是操作系统进行的工作。开发者可以控制和影响有限。这个过程, 我们需要深入操作系统在main之前的行为路径。

过程2 (after-main) : 在调用main函数之后, 到用户进入到落地页 (绝大部分场景下, 首页是落地页), 这个过程, 就是我们需要治理的最主要的过程; (该我们可以通过打点的方式, 记录关键的时间T0、T1、T2)

5.4.2 Pre-main 阶段和main() 阶段优化方案

工程中动态库过多 (77个)



动态库懒加载

我们知道, pre-main 过程中, 有dylib的加载步骤, 而动态库加载是需要耗时的, 苹果建议我们自定义的动态库不要超过 6 个, 因此, 尽量减少启动过程中的动态库加载有助于启动耗时的优化。减少启动过程中的动态库加载主要有以下方案:

- 动态库转静态库;
- 多个动态库进行合并;
- 减少启动过程中需要加载的动态库数量;

58同城的一种动态库懒加载的方案:

1、动态库懒加载方案

所谓动态库懒加载是指, 在启动的过程中并不加载该动态库, 而是在业务真正使用到该动态库中的内容时才进行加载, 从而减少启动耗时。在 Cocoapods 1.2 之前存在配置动态库懒加载的入

口，升级到 1.8 之后没有了动态库懒加载的配置入口，我们需要在pod install之后生成的配置文件中配置。使用 Cocoapods 管理的项目，在pod install之后，会生成Pods-xxx-frameworks.sh和Pods-xxx.adhoc/debug/release.xcconfig这两个文件，其中Pods-xxx-frameworks.sh文件脚本负责架构剔除和重签名等功能，而Pods-xxx.adhoc/debug/release.xcconfig文件则负责静态库和动态库的链接配置，我们自定义的动态库想要进行懒加载，只需要修改xxx.xcconfig配置文件，将需要懒加载的动态库从配置文件中移除，这样保证懒加载的动态库参与签名和拷贝，但是不参与链接。

2、动态库懒加载后的调用方式

由于采用动态库懒加载后动态库在编译时没有参与链接，原有的代码调用方式会报找不到对应动态库符号的错误，因此，原有动态库的调用方式需要修改成Runtime动态调用的形式，在使用某个动态库中的类时，先动态获取该类，如果获取不到，则通过dlopen的方式动态加载该动态库：

```
dlopen([path UTF8String], RTLD_LAZY);
```

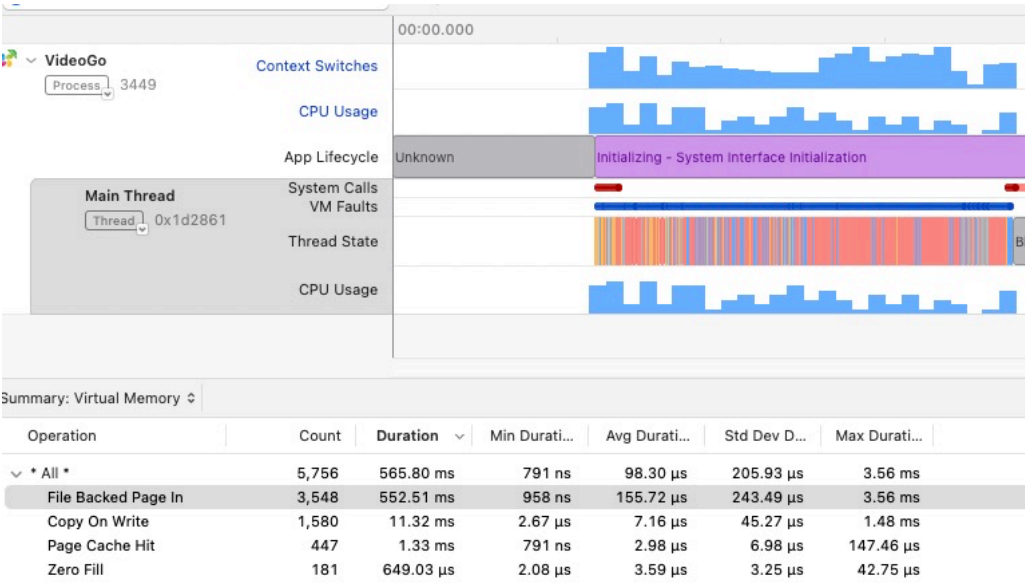
动态库懒加载后，将减少启动过程中dlopen带来的损耗，同时减少 rebase/bind 的时间，以及避免了该懒加载动态库内 load、constructor 等函数在启动过程中执行。

二进制重排

工程中二进制重排文件比较老，需要重新再生成一份



通过system trace验证pageDefault次数和耗时是否减少



xib/Storyboard治理

用纯代码的方式，而不是 xib/Storyboard，来加载首页视图 [萤石云工程xib治理](#)

4.5.3 after-main 阶段的优化方案

- 1. 现有的启动项堆积严重，拖慢启动速度。
- 2. 增加新的启动任务缺乏添加规范，杂乱无章，修改风险大，都是各个在启动流程中修改代码。

面对这两个问题，我们首先梳理目前启动流程中所有的启动任务，

启动顺序	业务	负责人	时机是否合适

4.5.4 建立监控机制

为了可以监控到日常开发过程中启动耗时变化，监控了启动过程中的方法调用耗时，通过每天构建对当天版本和昨天版本的差异分析耗时原因

4.6 耗电治理

目前没有找到有效的直接监控手段，可以通过研发开发工具分析耗电根因，后续大版本进行对比测试来进行闭环

研发分析工具：

<https://mp.weixin.qq.com/s/1ay9tXGmutiZNSNJfteP9Q>

4.8 OOM，占用内存过高治理

4.8.1 OMM

https://www.jianshu.com/p/c72809b6e534

- omm主要有：
- 内存泄漏
- 内存泄漏造成内存被持久占用无法释放，对OOM的影响可大可小，多数情况下并非泄漏的类直接造成大内存占用，而是无法释放的类引用了比较大的资源造成连锁反应最终形成OOM。
- 内存异常增长
- 缩放、绘制分辨率高的大图片，播放 gif 图，以及渲染本身 size 过大的视图（例如超长的 TextView）等，都会占用大量内存，有时会不恰当的操作会造成内存的异常增长出现OOM，尽管这部分内存可能一会就被释放掉，并不会长久的占用内存，可能在解析、渲染的过程中发生 OOM。

4.8.2 一些开源工具

内存泄漏（一般是采用MLeaksFinder+FBRetainCycleDetector结合的方式）

工具	简介	优缺点	
MLeaksFinder	MLeaksFinder 是 WeRead团队开源的一款检测 iOS 内存泄漏的框架对代码没有侵入性，而且其使用非常简单，只需要引入项目中，如果有内存泄漏，2秒后自动弹出 alert 来显示捕捉的信息	MLeaksFinder 虽然帮我们找到了内存泄漏的对象，但是我们具体不知道引起循环引用的链条	
FBRetainCycleDetector	FBRetainCycleDetector是FaceBook开源的用于检测强引用循环的工具。默认是在DEBUG环境中启用，当然你也可以通过设置 RETAIN_CYCLE_DETECTOR_ENABLED以始终开启。使用这个工具可以传入应用内存里的任意一个 Objective-C 对象，FBRetainCycleDetector 会查找以该对象为根节点的强引用树中有没有循环引用。		萤石云有接入，debug 环境下开启

内存异常增长

	Allocation	FBAallocationTracker	OOMDetector
	苹果官方提供的Allocation内存分析工具，在开发调试阶段，可以用Allocation详细分析App各模块内存占用。Allocation对App的内存监控比较全面，能监控到所有堆内存以及部分VM内存分配。	FBAallocationTracker是Facebook开源的内存分析工具，它的原理是用 Method Swizzling替换原本的alloc方法，这样可以在App运行时记录所有OC实例的分配信息，帮助App在运行阶段发现一些OC对象的异常增长问题。相比Allocation，FBAallocationTracker对App性能影响较低，可以在App中独立运行。	OOMDetector是腾讯研发的一个内存监控组件。 通过Hook系统底层的内存分配方法，能够记录到进程所有内存分配的堆栈信息，同时组件能够在对性能流畅度影响不大的情况下保证在App中独立运行，可以方便分析和监控线上用户的内存问题。 1，监控OOM，Dump引起爆内存的堆栈 2，大内存分配监控 监控单次大块内存分配，提供分配堆栈信息
使用场景	连接Mac用	App中独立运行	App中独立运行
监控范围	所有内存对象	只监控OC对象	所有内存对象
性能影响	性能影响高	性能影响低	性能影响低

4.8.3 大厂自研OOM治理

		方案
https://code84.com/886050.html	抖音	
https://blog.csdn.net/lihui49/article/details/127978667	百度	<p>在子线程开启定时器，每隔几秒采样一次内存phys_footprint字段数据</p> <p>1、建立App内存实时监控</p> <p>通过线上内存采样打点，获取app不同机型在使用过程中的内存值，然后经过上报的数据在服务端进行数据聚合，我们可得到不同机型的内存最大值</p> <p>2、重点页面内存预测</p> <p>在建立实时内存监控的基础上，通过经验数据直接预测页面的内存涨幅。</p> <p>存在一种情况，当打开一个新的页面时，这个时候采集周期还没有到，内存上涨了特别多，内存管控还未生效app可能就会发生OOM。通过页面内存计算，打开新页面之前知道页面可能增长到多少，如果进入到危险水位，实时释放内存以降低OOM率。</p> <div><pre>graph LR; A[进入P1页面] --> B[push进入P2页面]; B --> C[跳转其他页面]; D[采集内存M1] --> A; E[采集内存M2] --> B; F[M2-M1] --> B</pre></div> <p>3、梳理不同机型上OOM的内存值</p> <p>通过在不同机型上主动触发OOM的机制，拿到在不同手机上触发OOM的内存值。</p>

		4、确定内存管控危险水位阈值 不同机型上主动触发OOM值比采集到的最大内存值小的话，那么采集到的最大内存值就是内存管控阈值。
--	--	--

六、行动步骤

由于工程治理是一个特别庞大的工作量，涉及到的人员众多，所以对以上方案推进落地进行不同的负责人划分：

问题	总负责	计划	备注
工程组件化改造	严超	Q1: 1, 按照IOT组件二进制化方案，将所有组件二进制化，保障后续项目开发哪个组件拉取哪个组件分支源码开发 Q2: Q3: Q4:	结合新的MCI系统完成工程组件化改造
包体积治理			
沙盒存储管理	王樂		
RN、Flutter基建能力建设			
启动流程治理			
OOM，占用内存过高治理			

对于所有问题必须有长期有效的治理管理机制，不能靠人工的方式进行管控，必须使用工具链来完成防劣化

附件：
<https://max.book118.com/html/2023/0812/5303100224010310.shtml> 抖音，防劣化
<https://www.it120.vip/yq/12566.html> 百度 包体积
https://zhuanlan.zhihu.com/p/437678980?utm_id=0 字节 稳定性问题
<https://segmentfault.com/a/1190000040277799> 打造一套客户端功能最全的 APM 监控系统


[工程治理](#)
[萤石云工程治理](#)
[萤石云视频APP启动速度优化方案设计](#)
[APP组件化容器化动态化设计方案](#)

赞同

成为第一个赞同者

无标签

评论

 **黄飞龙** 发表：

主工程先进行SVN提交锁死，禁止添加资源文件，权限只开放给个别人，并且对现有工程中图片进行梳理，将所有图片归纳到业务组件中

hi, 提一个建议，主工程不放资源文件是合理的，所有图片归纳到业务组件中，是否会造成资源冗余，把图片资源单独打一个组件，业务组件进行依赖，把这一步直接自动化掉是否会更合理些。