iOS工程模块化实践与优化

基于Xcodeproj & Carthage 的持续集成

Amateurish VS Professional

Xcodeproj => Makefile => Gradle

Imperative => Declarative

Everyone Can Code

https://www.apple.com/everyone-can-code/

```
wc -l tools/template/universal-project.pbxproj
    1793 tools/template/universal-project.pbxproj
```

Pbxproj & Makefile & Gradle

Pain

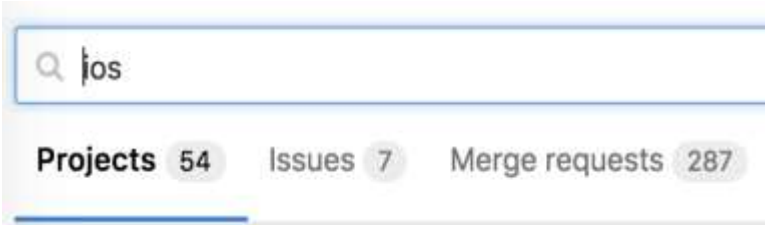
Merge hell

All in one

Pain

version	developers	source code (oc)	date
v2.1.3	1	119	2013.09.15
v3.6.3	3	441	2015.03.04
v4.23	9	1438	2016.06.24
v4.38	40+	3000+	2017.02.22
v5.18	120+	6000+	2017.10.16

54个项目仓库(oc/c++/c)



3个应用

Modular Design

1. Components
2. Data bus
3. Router
4. ....

Carthage / CocoaPods

Differences between Carthage and CocoaPods

CocoaPods is a long-standing dependency manager for Cocoa. So why was Carthage created?

Firstly, CocoaPods (by default) automatically creates and updates an Xcode workspace for your application and all dependencies. Carthage builds framework binaries using `xcodebuild`, but leaves the responsibility of integrating them up to the user. CocoaPods' approach is easier to use, while Carthage's is flexible and unintrusive.

The goal of CocoaPods is listed in its README as follows:

> … to improve discoverability of, and engagement in, third party open-source libraries, by creating a more centralized ecosystem.

By contrast, Carthage has been created as a *decentralized* dependency manager. There is no central list of projects, which reduces maintenance work and avoids any central point of failure. However, project discovery is more difficult—users must resort to GitHub's Trending pages or similar.

CocoaPods projects must also have what's known as a podspec file, which includes metadata about the project and specifies how it should be built. Carthage uses `xcodebuild` to build dependencies, instead of integrating them into a single workspace, it doesn't have a similar specification file but your dependencies must include their own Xcode project that describes how to build their products.

Ultimately, we created Carthage because we wanted the simplest tool possible—a dependency manager that gets the job done without taking over the responsibility of Xcode, and without creating extra work for framework authors. CocoaPods offers many amazing features that Carthage will never have, at the expense of additional complexity.

CocoaPods

优点

1. 大部分第三方库都写好了podspec (模块的导出)
2. 管理工具方便 pod

缺点

1. 基于源码编译(无法享受incremental build)
2. 污染Xcode编译中间文件，导致各种奇怪的编译错误需要频繁清理cache
3. 难以(无法)管理resource
4. 维护pod困难（ local repo，podspec书写...)

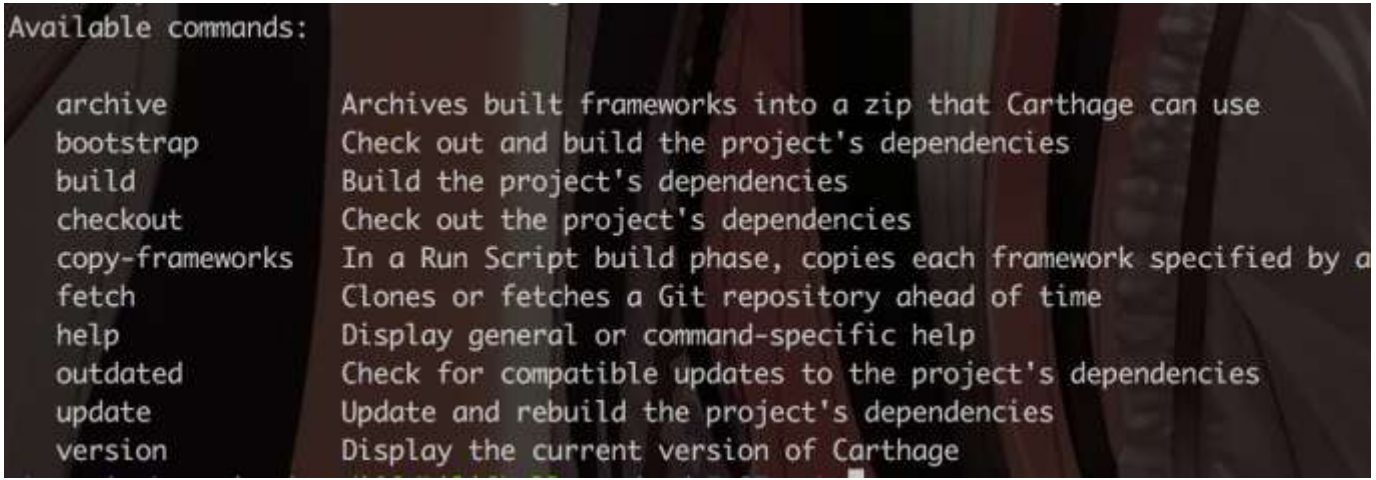
Carthage

优点

恰到好处

缺点

恰到好处Orz



What bilibili do for Carthage

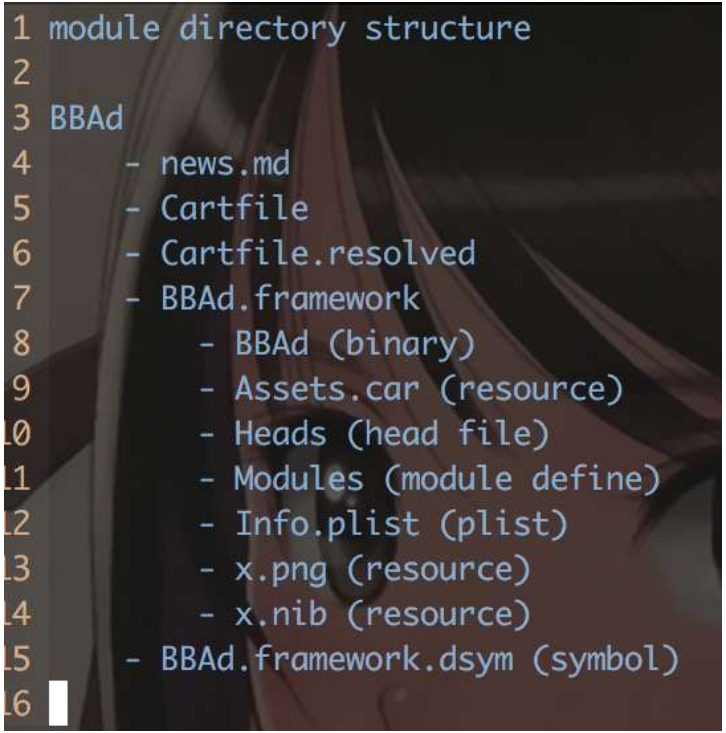
Central module server (macross)

1. Download
   a. 一个文件服务来描述模块列表https://github.com/Carthage/Carthage/blob/master/Documentation/Artifacts.md#cartfile
2. Upload
   a. 一个jenkins用来模块构建及静态检查及模块上传服务
3. Dashboard
   a. 一个用来监控模块体积，异常依赖的查询服务

What is module

1. Semver (https://semver.org/)
2. 依赖关系 (当次构建的依赖 & 对依赖的兼容关系)
3. 更新日志

1. Library
    a. 自定义 dylib 或者 static lib的方式自由集成
2. Resource
    a. 独立的资源空间，解决命名冲突
3. Symbol
    a. 统一构建拥有完整的调试符号，可以在ide中debug step into代码



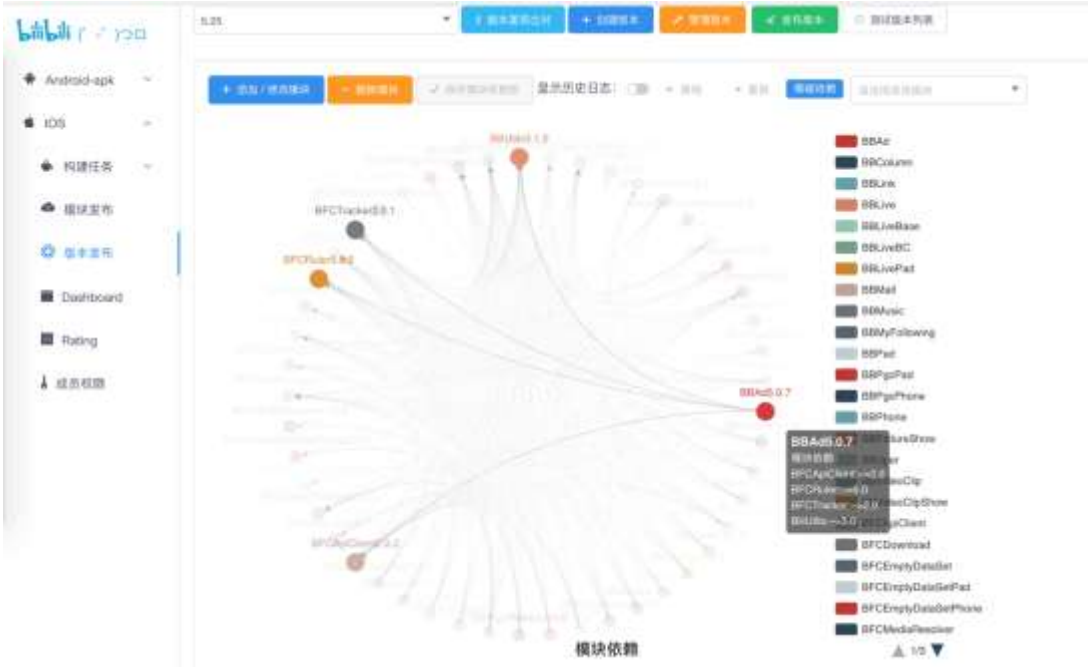
XcodeProj

https://github.com/CocoaPods/Xcodeproj

Gradle VS project.pbxproj (fuck it!)

1. Build
    a. Feature（lldb, editor, profile...)
    b. Source code OR Pre compile
2. Merge
    a. aiuohfsjklfhjaghbgaf！！(remote ref, local ref, build phase, group, ……)

Dependencies Management



Bilibili Project Structure

Main:

1. Initialize const variable
2. Initialize base lib (BFC)
3. Initialize module bus

Build Step

1. Generate project.pbxproj (sh build.sh)
2. Generate ipa (xcodebuild release)
    a. Compile modules
    b. Compile target / extensions
    c. Link target / extensions
    d. Copy resource
    e. Delete binary
    f. Resign

What bilibili do for CI

1. ignore project配置文件，基于Cartfile内容，分别相应的生成，xcode的目录结构依赖以及buildphase
2. 基于不同的target（ipa, extension) 通过生成脚本自由生成不同业务角色最合适的project项目
    a. Cartfile.filter('git').addToReference.addToLinkPhases.addToCopyPhases
    b. Cartfile.filter('binary').addToLinkPhases.addToCopyPhases
3. 一系列辅助工具使得开发与发布全自动化(strip, sign…)
    a. 根据module内的binary属性，使用不同的链接方式(dy,static)
    b. 根据构建的不同类型，选择是否裁切相应架构 (x86, arm)
    c. 根据构建的不同类型，选择相应的辅助工作的执行，符号上传，签名等
4. 模块/版本发布工具 Macross

More declarative (TODO)

1. 把最小集成单元继续从模块往下降成单个代码文件，单个函数 (模块级别的CI -> 文件级别的CI)
2. 进一步降解xcodeproj，使xcode最终退化成编辑器及调试器
3. 通过xcrun toolchain定制一套构建对构建速度进行极限优化 （ 去除module server, 去除 version)

吐槽

1. Carthage的版本冲突计算由于会递归自己历史版本算真的很慢很慢 = =
2. Carthage的寻找shared scheme的条件实在有点呆萌 = =
3. Xcode 的 static framework的link方式真的很迷很迷 = =
4. Xcode的增量编译真的很蠢很蠢 = =

Q/A