曹理鹏@iCocos

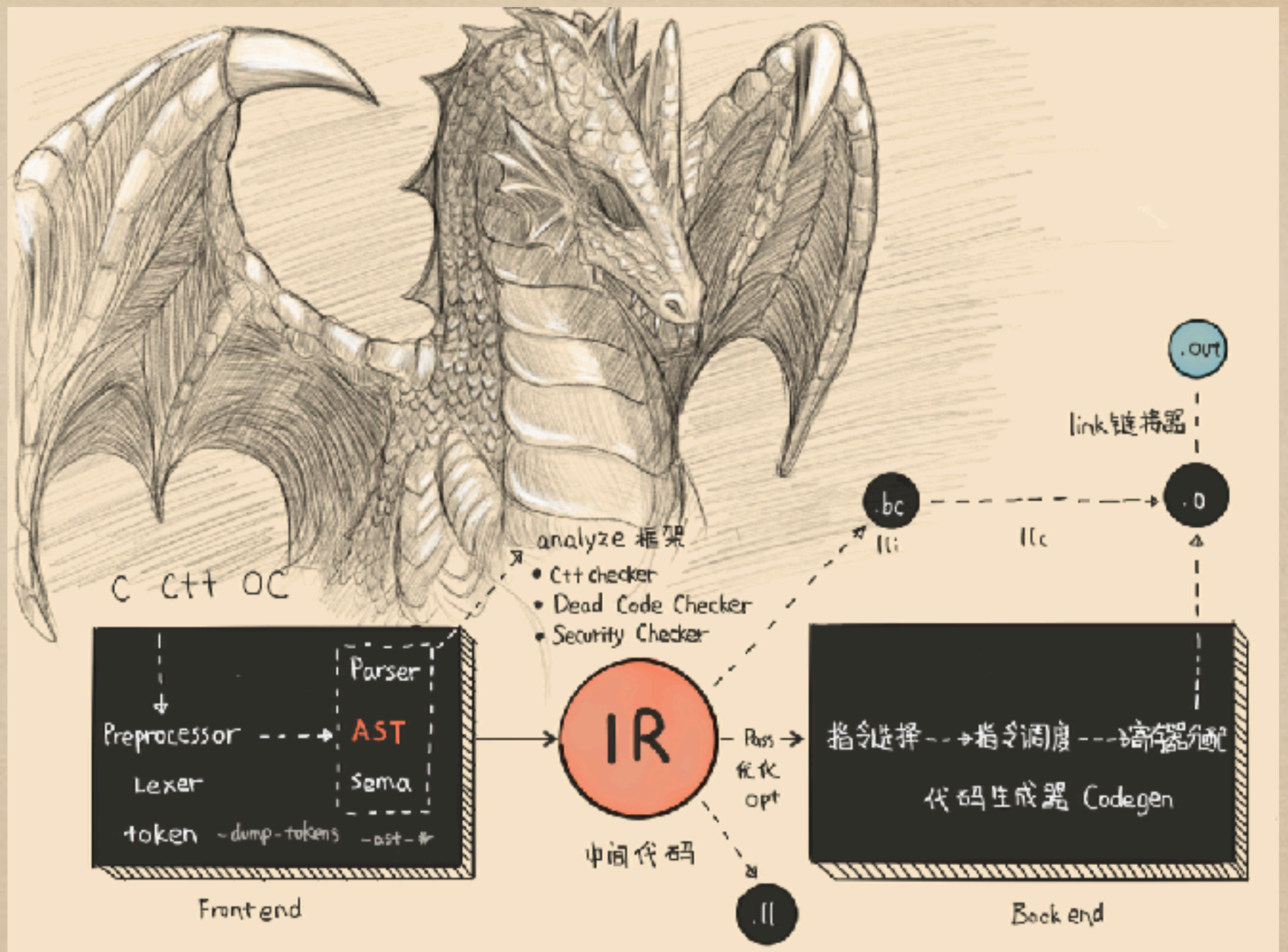
- 曹理鹏
- 江西九江
- 刻骨铭心D幸福
- iOS（Objective-C），逆向工程, H5, PHP, Java
- Swift
- 博客:
  - https://icocos.github.io
  - http://al1020119.github.io/
  - https://github.com/al1020119
- 爱好:
  - 音乐，运动（打球），学习（代码）
- 座右铭:
  - 不战胜自己，何以改变未来!

App的整个启动过程

Swift && Objective-C

- App的整个启动过程大概分为

- 编译、汇编、链接(静态链接)、代码签名，启动执行（动态链接）、Main、AppDelegate（启动方案）、显示窗口和相关控件。



App的前后端的编译流程（OC版）

按大点的范围我们可以分为三个过程,编译,链接,装载,呈现(语言本身除外)

- 编译：编译系统读取文本字符串，解读所蕴含的意义，然后会翻译成机器能看懂的汇编语言：目标文件。
- 链接：每个类，每个文件会被编译成不同的目标文件，链接器把这每个目标文件串起来，相互调用，最后生成可执行文件。
- 装载：把已经生成的可执行文件放到操作系统里，在系统专属的进程与内存控制下，找到机器可以识别的汇编代码入口，开始按着汇编去执行机器码，并且能与操作系统级别的各种系统Api对接起来。
- 呈现：整个编译，链接，装载完成之后，就会重新回到Main函数，清空之前的所有现场，并开始根据代码的逻辑进行显示我们的App。

常规思维

- 按我们之前的学习和思维方式我们可以将整个过程以Main函数为分水岭（一切代码的开始？）。
  - 严格来说其实是以中间层代码为分水岭划分整个启动的两个大阶段

- 这里分别根据Main函数之前和之后所做的事情去理解,当然这中间有一个比较重要的就是资源的加载过程！

整个文件和资源的加载过程

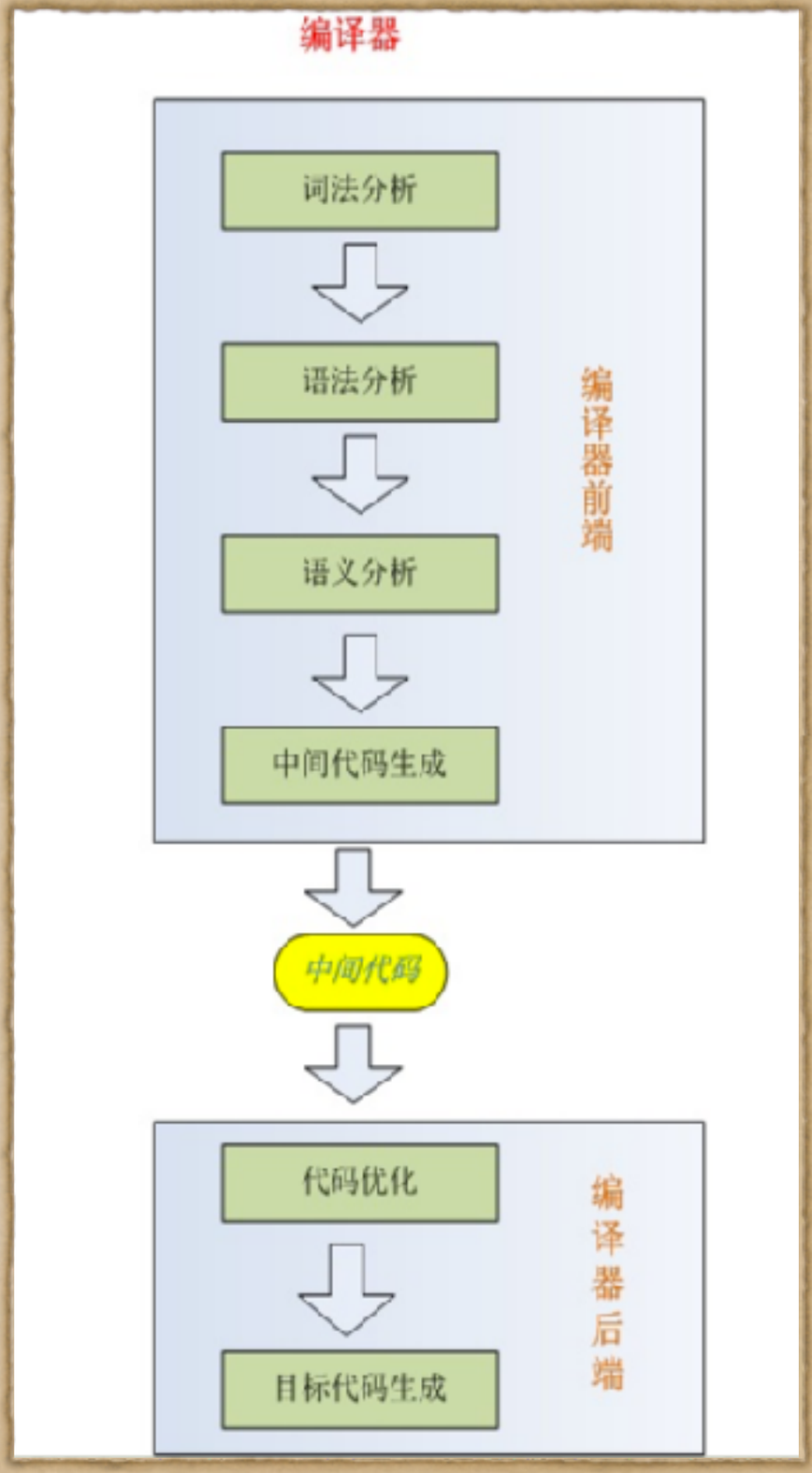
- 编译信息写入辅助文件，创建文件架构 .app 文件
- 处理文件打包信息
- 执行 CocoaPod 编译前脚本，checkPods Manifest.lock
- 编译.m文件，使用 CompileC 和 clang 命令
- 链接需要的 Framework
- 编译 xib
- 拷贝 xib，资源文件
- 编译 ImageAssets
- 处理 info.plist
- 执行 CocoaPod 脚本
- 拷贝标准库
- ……

Main之前

- 编译、
- 汇编、
- 链接(静态链接)、
- 代码签名、
- 启动执行（动态链接）、
- ……

编译

- 编译工作是由编译器来完成的，主要进行预处理、词法分析、语法分析、语义分析、生成中间代码、目标代码生成（优化）.



解释型与编译型

- 编译型语言：在程序执行之前，有一个单独的编译过程，将程序翻译成机器语言，以可执行文件的形式存在.以后执行这个程序的时候，就不用再进行翻译了（C/C++/OC/Swift）。

- 解释型语言(脚本语言)：是在运行的时候将程序翻译成机器语言，所以运行速度相对于编译型语言要慢(Java，C#, javascript, PHP)。一般都是以文本形式存在,类似于一种命令.

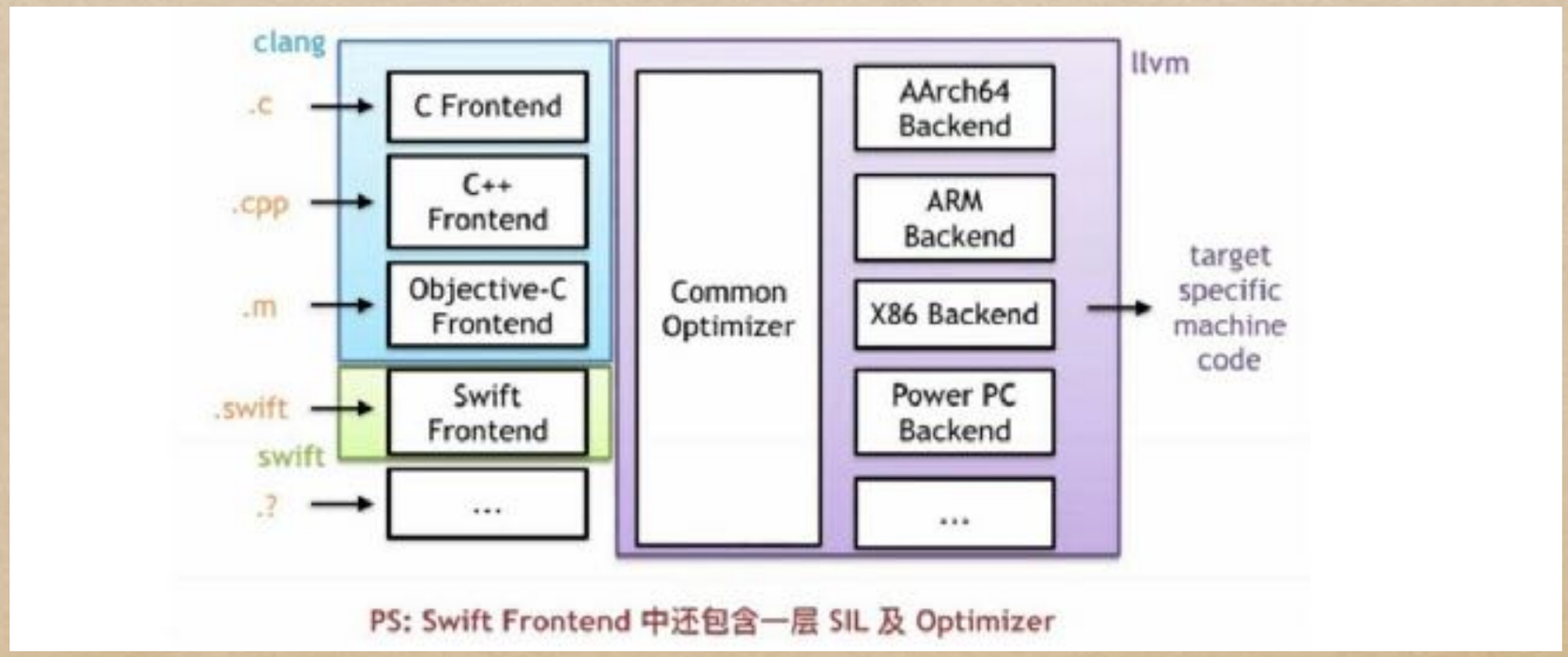
LLVM && Clang

- 苹果在早期使用GCC（GCC原名为GNU（GNU Compiler Collection） C语言编译器，它原本只能处理C语言，后来扩展了Objective-C、Java等语言）作为官方的编译器，后来由于各种限制，使用了一套自己的编译器，那就是LLVM（Low Level Virtual Machine）。

* Xcode 3.1实现了llvm-gcc compiler, Xcode 3.2实现了Clang 1.0,克里斯（Swift她爹）再接再励, Xcode 4.0实现了Clang 2.0.后来的Mac OS X 10.6 Snow Leopard即大量使用LLVM的编译技术

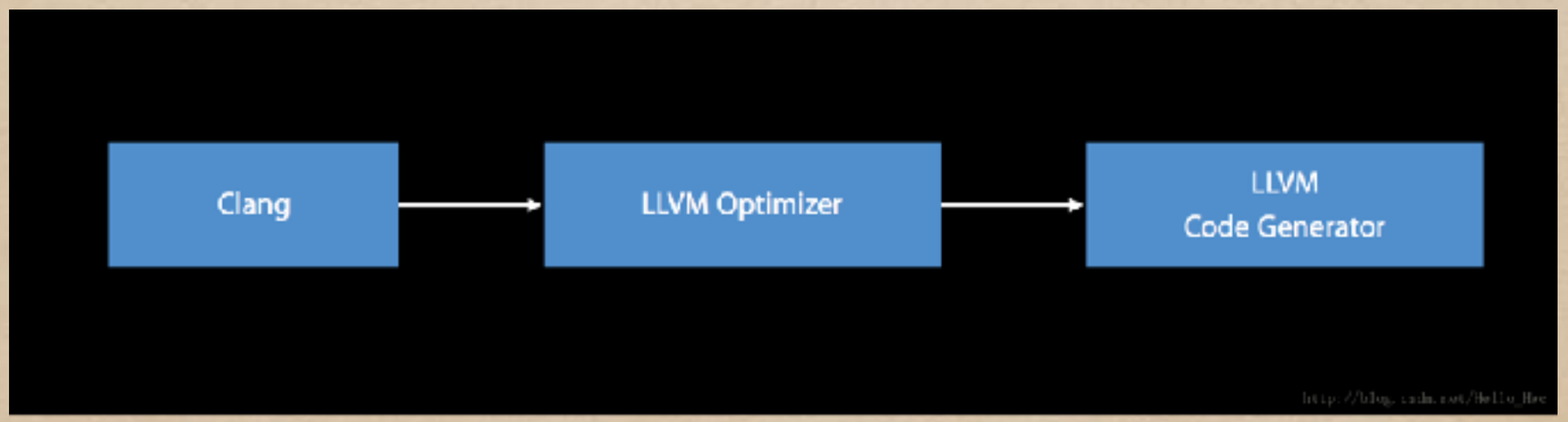
编译器前后端

- ~ 编译器前端负责产生机器无关的中间代码
- ~ 编译器后端负责对中间代码进行优化并转化为目标机器代码。



Clang / LLVM

- LLVM（Low Level Virtual Machine）：
  - - 编译器的后台——进行程序语言的编译期优化、链接优化、在线编译优化、代码生成（优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)）
- Clang:
  - - 编译器（LLVM）的前端（Clang是LLVM的子项目）— 是一个C++编写、基于LLVM、发布于LLVM BSD许可证下的C/C++/Objective C/Objective C++编译器，其目标（之一）就是超越GCC
- 编译器（LLVM）前端和编译器的后台：可以以我们后面提到的中间码生成为分界线。前边的环节就叫编译前端，后面的环节叫做编译后端



- 不管是OC还是swift，都是采用Clang作为编译器前端，LLVM(Low level vritual machine)作为编译器后端，
- 只是在Xcode 9之后使用Swift重写编译器并进行了应用。

Clang / LLVM 总结

- lldb 是 Clang / LLVM 的内置链接器，clang必须调用链接器来产生可执行文件。
- LLVM是编译器和工具链技的集合，Clang才是真正的编译器，Clang必须调用链接器(内置lldb)来产生可执行文件。

- LVVM优化器会进行BitCode的生成，链接期优化，有新的后端架构还是可以用这份优化过的 bitcode 去生成。
- LLVM机器码生成器会针对不同的架构，比如arm64等生成不同的机器码。

预编译

- 预编译主要用来处理那些源文件中以#开头的预编译命令，比如：
- 文件包含 #include，#import
- 宏定义 #define
- 条件编译 #if #else #endif
- 错误、警告处理 #error #warning
- 编译器控制 #pragma
- /* */ & //删除注释，注释是对编译完全没用，不会参与编译
- 添加行号，给每行代码添加行号，万一编译报错，也方便追查
- 其他 #line
- .....

词法分析

- 把代码或者输入切（扫描器逐行去扫描整个代码文件 + 算法）成一个个独立的Token也叫词法符号（类PHP）,比如关键字，标识符，字面量，运算符号等

    - // find a Class
    - id matchClass(id *s) {
    - }

- 关键词：比如for while if static等会被词法分析识别成单词

语法分析

- 语法是否正确,将所有节点组成抽象语法树 AST（编译器在尝试读懂你写的代码了），抽象语法树每个节点都是一个表达式

- 经过了词法分析你得到的token流，会按顺序输入语法分析器，语法分析器会尝试解读，最终将我们希望表达的自然语义，构建成了一个逻辑上的计算机能识别，能执行，能遍历的结构--树状结构

  - AST语法树，指的是源代码语法所对应的树状结构。也就是说，对于一种具体编程语言下的源代码，通过构建语法树的形式将源代码中的语句映射到树中的每一个节点上。

- 如果此时我们写代码不严谨，哪里少了个括号，哪里赋值没写值就直接回车，这时候语法树都是无法生成的，就会直接编译报错。

语义分析

- 语法分析只能完成语法层面的分析，无法对整个语句的真正意义进行判别，比如，讲一个浮点数赋值指针类型的时候，语义分析器就会发现类型不匹配，编译器提出相应的错误警告。

- 语义分析主要做的事情就是类型检查、以及符号表管理.这种静态分析，会遍历整个语法树，把每个节点的表达式都标识类型，并且验证是否合法，

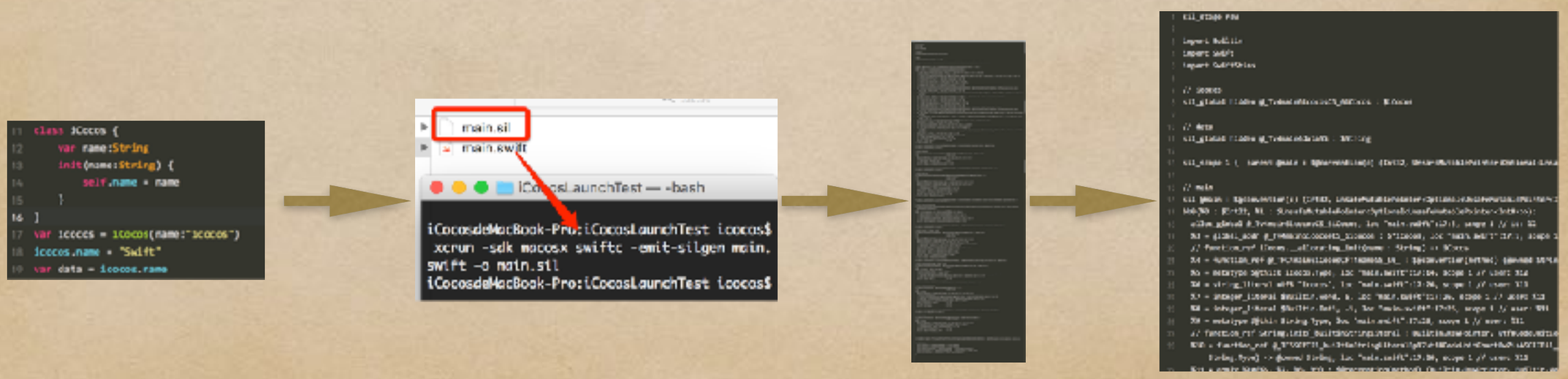
Swift 中间语言（SIL）

(Swift Intermediate Language)

生成 Swift 抽象语法树（AST）之后，就会从 AST 生成 Swift 中间语言（Swift Intermediate Language，SIL）

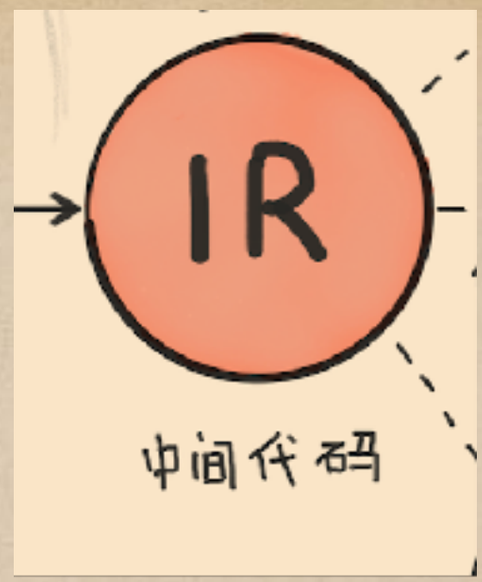
- Swift 编译器提供命令行工具：swiftc。通过控制命令行参数能获取到 Swift 源码编译到各个阶段的结果，从而进行分析和做一些特殊的处理。

新建一个命令行项目在main.swift里面验证SIL——使用命令：xcrun -sdk macosx swiftc -emit-silgen main.swift -o main.sil



- 通用的LLVM 中间表述（LLVM IR）

  - (LLVM Intermediate Representation)



- 中间代码生成

  - CodeGen会负责将语法树自顶向下遍历逐步翻译成 LLVM IR(IR 是编译过程的前端的输出后端的输入.

  - Pass 是 LLVM 优化工作的一个节点，一个节点做些事，一起加起来就构成了 LLVM 完整的优化和转化.

- 前面提到了：编译器前端负责产生机器无关的中间代码，编译器后端负责对中间代码进行优化并转化目标代码的生成与优化

汇编

- 目标代码需要经过汇编器处理，才能变成机器上可以执行的指令，生成对应的.o文件。

- 当你写Objective-C代码时，它们最终转换成机器码—ARM处理器能理解的原始的0和1指令。在Objective-C和机器码之间，还有一种可直接理解的汇编语言，所以其实整个过程中是需要走一遍汇编指令的。

- 汇编语言通过汇编器转译成CPU可识别的01机器码。

- 如果你想了解更多关于iOS开发中汇编相关的知识可以点击这里：iOS汇编教程：理解ARM（http://www.jianshu.com/p/544464a5e630）

```
.386
STACK   SEGMENT USE16 STACK
        DB 200 DUP(0)
STACK   ENDS
DATA    SEGMENT USE16
BUF1    DB 0AH,0DH,'1111111111! $'
BUF2    DB 0AH,0DH,'2222222222! $'
BUF3    DB 0AH,0DH,'Other Character! $',0AH,0DH
DATA    ENDS
CODE    SEGMENT USE16
ASSUME  DS: DATA,CS: CODE,SS: STACK
START:  MOV AX,DATA
    MOV DS,AX
    MOV AH,1
    INT 21H
    CMP AL,'1'
    JE A1
    CMP AL,'2'
    JE A2
    JMP A3
A1: LEA DX,BUF1
    MOV AH,9
    INT 21H
    JMP EXIT
A2: LEA DX,BUF2
    MOV AH,9
    INT 21H
    JMP EXIT
A3: LEA DX,BUF3
    MOV AH,9
    INT 21H
EXIT:   MOV AH,4CH
    INT 21H
CODE    ENDS
END START
```

链接

- 链接器（这里指的是静态链接器）将多个目标文件合并为一个可执行文件，在 OS X 和 iOS中的可执行文件是 Mach-O（Mach-O 被划分成一些 segement，每个 segement 又被划分成一些 section。），又分为静态链接和动态链接

- 静态链接

- 静态链接：在编译链接期间发挥作用，把目标文件和静态库一起链接形成可执行文件

- 静态链接的代码在编译后的静态链接过程就将插头和插排一个个插好，运行时直接执行二进制文件；

动态链接（windows中的dll）

- 动态链接：链接过程推迟到运行时再进行。
- 而动态链接需要在程序启动时去完成“插插销”的过程，所以在我们写的代码执行前，动态连接器需要完成准备工作。
- 如果多个程序都用到了一个库，那么每个程序都要将其链接到可执行文件中，非常冗余，动态链接的话，多个程序可以共享同一段代码，不需要在磁盘上存多份拷贝，但是动态链接发生在启动或运行时，增加了启动时间，造成一些性能的影响。
- 静态库不方便升级，必须重新编译，动态库的升级更加方便。

代码签名

- 每次build之后，都会发现工程目录下多了一个.app文件
- 右键ipa，重命名为.zip，双击zip文件，解压缩后会得到一个文件夹，查看ipa包内容。

- .o 文文件，.m文件编译后的产物。
- .a文件
- 需要link的framework
- __TEXT 代码段
- __DATA 数据段
- dSYM

- 在 .app目录中，有又一个叫_CodeSignature的子目录，这是一个 plist文件，里面包含了程序的代码签名，你的程序一旦签名，就没有办法更改其中的任何东西，包括资源文件，可执行文件等，iOS系统会检查这个签名。

- 签名过程本身是由命令行工具 codesign 来完成的。如果你在 Xcode中build一个应用，这个应用构建完成之后会自动调用codesign 命令进行签名，这也是Link之后的一个关键步骤。

启动

- dyld（动态链接器）进行动态链接，进行符号和地址的一个绑定。
- dyld 主要在启动过程中主要做了以下事情：
  - 加载所依赖的dylibs
  - Fix-ups：Rebase修正地址偏移，因为 OS X和 iOS 搞了一个叫 ASLR的东西来做地址偏移（随机化）来避免收到攻击
  - Fix-ups：Binding确定 Non-Lazy Pointer地址，进行符号地址绑定。
  - ObjC runtime初始化：加载所有类
  - Initializers：执行load 方法和attribute((constructor))修饰的函数

Main函数之前补充

- 其实Main函数事前还做了非常多我们不知道的事情，包括代码层的。
- 1.+ load
- 2.动态链接库（framework：otool），objc和runtimelibdispatch(GCD)，libsystem_c(C语言库)，libsystem_blocks(Block)
- 3.ImageLoader（非图片），二进制文件。包含被编译过的符号、代码。
- 4.默认的有缓存机制（不会玩命初始化）

- 总结：
  - 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。

Main之后

- 在Main函数开始其实做的事情就是比较偏代码层，也相对比较好理解了。

- 执行Main函数->执行UIApplicationMain函数->初始化UIApplication并设置代理对象，开启事件循环->同时监听视图事件->根据不同的方案将显示到用户的眼前

Main函数之后的启动方案

- main函数
- 执行UIApplicationMain函数
- 创建UIApplication对象
- 创建UIApplicationDelegate对象并复制
- 创建应用程序的Main Runloop循环
- UIApplicationDelegate对象开始处理监听到的事件
- 读取配置文件info.plist，设置程序启动的一些属性，
- 如果info.plist文件中配置了启动storyboard文件名，则加载storyboard文件,并根据storyboard中启动配置加载和显示窗口，显示对应的控件。
- 如果没有配置，则根据application:didFinishLaunchingWithOptions:代码来创建UIWindow—>UIWindow的

控制器View的生命周期

loadView： 加载view

viewDidLoad： view加载完毕

viewWillAppear： 控制器的view将要显示

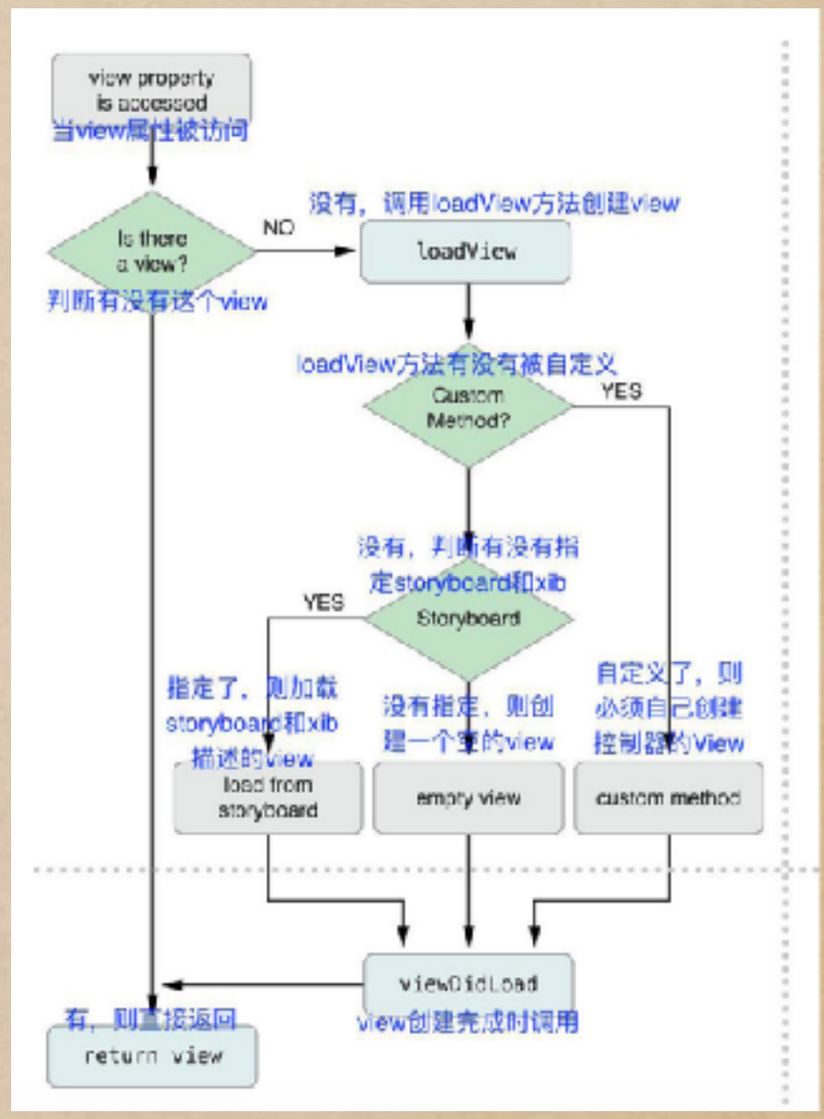
viewWillLayoutSubviews： 控制器的view将要布局子控件

viewDidLayoutSubviews： 控制器的view布局子控件完成

viewDidAppear: 控制器的view完全显示

viewWillDisappear： 控制器的view即将消失的时候

viewDidDisappear： 控制器的view完全消失的时候



OC中的Main函数

- 我们先来看OC中Main函数

  - int main(int argc, char * argv[]) {
  - @autoreleasepool {
  - return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
  - }
  - }

Main函数

- 我们都知道，main函数是C语言程序的入口，同样的它也是Objective C的入口

- 从上面的代码我们可以知道：UIApplicationMain如果返回，则程序会结束，所以它会在结束时返回或者永远不会返回，而传入的可控参数则是最后一个参数，是一个类的名字。

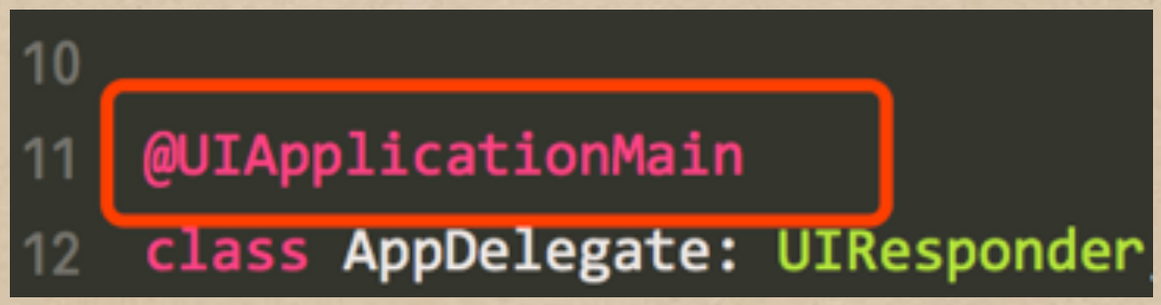
- 但是实际上，UIApplicationMain函数永远不会返回（这里是因为RunLoop在起作用），它负责对于建立程序执行所需要的类，进行初始化，并使程序进入事件等待的循环中，而最后一个参数则指明了程序执行所需的代理类。

Swift中的Main函数

- In Xcode, Mac templates default to including a “main.swift” file, but for iOS apps the default for new iOS project templates is to add @UIApplicationMain to a regular Swift file. This causes the compiler to synthesize a mainentry point for your iOS app, and eliminates the need for a “main.swift” file.

Main函数

- 意思是，Swift项目中添加了@UIApplicationMain到swift文件中，使得编译器合成了一个app入口，所以不需要main.swift文件。AppDelegate文件中多了个@UIApplicationMain的标志，启动app并放置断点，会发现其实main函数还是存在的。



- 我们可以通过调用堆栈看出Swift的执行顺序是

- ->start()

- ->main()

- ->top_level_code()

top_level_code

- 相对于C语言项目，多出来top_level_code()，在main.swift中的（非声明）代码会直接作为top_level_code()代码执行。此处要注意在Swift语言本身并不需要入口函数，程序入口是指定为main.swift中的非声明代码。在具体编译环节，ios/osx的入口均采用约定的main()函数，为了兼容以前的入口方法，将Swift语言程序在编译环节处理成隐式入口函数top_level_code(),再由main()调用。

UIApplicationMain

- 我们来回忆一下上面提到的Main函数中的两个参数

  - 这个方法将根据第三个参数初始化一个UIApplication或其子类的对象并开始接收事件 (传入nil，意味使用默认的UIApplication)。最后一个参数指定了AppDelegate类作为应用的委托，它被用来接收类似didFinishLaunching或者didEnterBackground这样的与应用生命周期相关的委托方法。

- 虽然这个方法标明为返回一个int，但是其实它并不会真正返回。它会一直存在于内存中，直到用户或 者系统将其强制终止。

@UIApplicationMain

- 前面我们提到了关于@UIApplicationMain标签：

```swift
//  AppLaunch.swift
//  AppLaunch
//
//  Created by iCocos on 2017/6/28.
//  Copyright © 2017年 iCocos. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.
        return true
    }
```

- 这个标签做的事情就是将被标注的类作为委托，去创建一个UIApplication并启动整个程序。在编译的时候，编译器将寻找这个标记的类，并自动插入像main函数这样的模板代码。

UIApplication

- 说明是UiApplication：
  - - UiApplication代表一个应用程序，每一个应用程序都有一个UIApplication全局对象，而且这个对象是单例的，我们在程序中可以通过[UIApplication sharedApplication]获得这个对象，进行一些应用级的操作。

- UiApplication的作用：
  - - UIApplication是应用程序的开始，它维护了一个在本应用程序中打开的Window列表，负责初始化显示UIWindow，并负责加载应用程序的第一个UIView到UIWindow窗口中，在随后的操作中，我们也可以更换UIWindow窗口的显示内容。
  - - UIApplication还被赋予一个代理对象，在实际编程中，我们一般并不直接与UIApplication打交道，而是和其代理对象UIApplicationDelegate打交道，UIApplication负责监听接收事件，而由UIApplicationDelegate决定应用程序如何去响应这些事件（生命周期：程序启动和关闭，系统事件：来电、记事项警告）等等。

UIApplicationDelegate

- 所有的移动操作系统都有个致命的缺点:app很容易受到打扰.比如一个来电或者锁屏会导致app进入后台甚至被终止;还有很多其他类似的情况会导致app受到干扰,在app受到干扰时,会产生一些系统事件,这时UIApplication会通知它的delegate对象,让delegate来处理这些系统事件.

```swift
// 当应用程序启动完毕的时候就会调用(系统自动调用)
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
    [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    return true
}
// 即将失去活动状态的时候调用(失去焦点，不可交互)
func applicationWillResignActive(_ application: UIApplication) {}
//应用程序进入后台的时候调用 一般在该方法中保存应用程序的数据，以及状态
func applicationDidEnterBackground(_ application: UIApplication) {}
// 应用程序即将进入前台的时候调用 一般在该方法中恢复应用程序的数据,以及状态
func applicationWillEnterForeground(_ application: UIApplication) {}
//重新获取焦点(能够和用户交互)
func applicationDidBecomeActive(_ application: UIApplication) {}
// 应用程序即将被销毁的时候会调用该方法 注意:如果应用程序处于挂起状态的时候无法调用该方法
func applicationWillTerminate(_ application: UIApplication) {}
```

启动过程验证

- 在2016 WWDC中有提到：如果我们想知道程序启动的时间，则可以在工程的scheme中添加环境变量DYLD_PRINT_STATISTICS，如图1所示。这样在调试时，可以在控制台打印出程序启动过程中各个阶段所消耗的时间。

- 另外，如果嫌弃控制台输出的乱七八糟打印太多，可以在上面相同的位置添加OS_ACTIVITY_MODE ,value 设置为disable即可

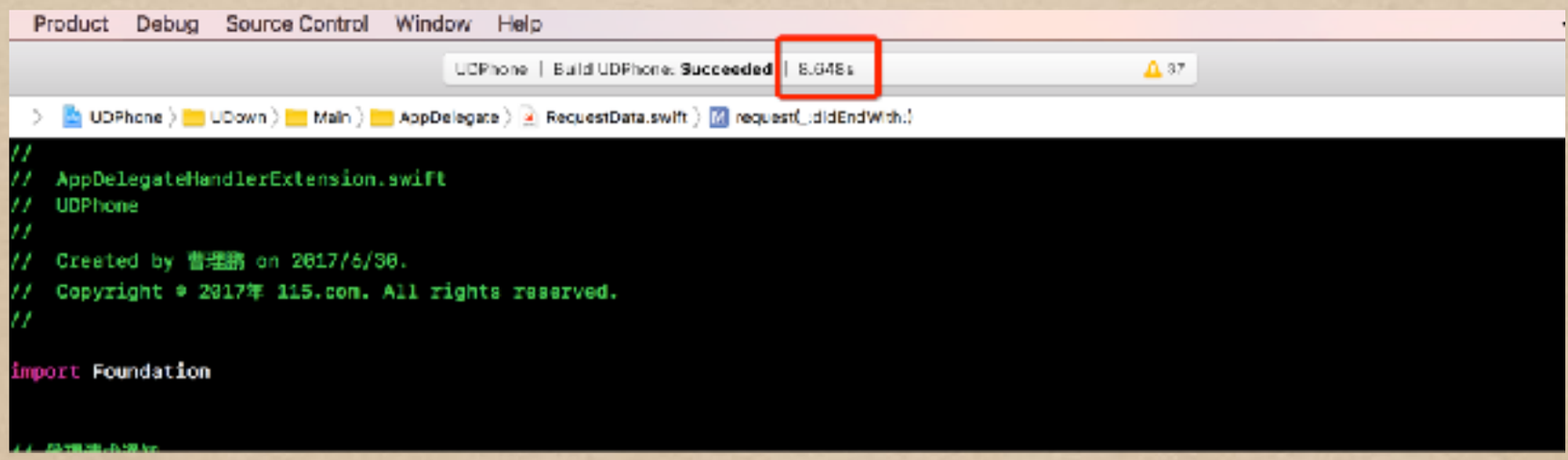
- 同时Swift 编译器提供命令行工具：swiftc。（查看swiftc --help）

查看编译时间

关闭XCode，终端输入以下指令

$ defaults write com.apple.dt.Xcode ShowBuildOperationDuration YES

然后，重启XCode，然后编译，你会在这里看到编译时间。



优化编译和启动方案

基础层面

实战与配置相关

- 使用DYLD_PRINT_STATISTICS测试启动加载时间。
- Swift编译优化（Product > Scheme > Edit Scheme > BuildBuild Opitions选项中，去掉Find Implicit Dependencies.）
- 编译时长优化（Build Setting > Swift Compile - Code Generation > Optimization Level ->Debug None[-Onone] Release Fast[-O]）
- Debug模式下，不生成dsym文件（借助XCode和LLDB进行调试）。
- Debug开启Build Active Architecture Only（XCode -> Build Settings -> Build Active Architecture Only 改为YES），只编译当前的版本，高版本Xcode自动开启。
- 减少自定义的动态库集成（系统库已经极度的优化了），苹果推荐在6个以内。
- 精简原有的Objective-C类和代码（消息和转发机制：动态链接），重定位和绑定(Rebase/binding)：指针修正。
- 移除静态的初始化操作（类的注册,分类的注册），合理利用+initialize替换原有+load方法进行初始代码工作。
- 使用更多的Swift代码（静态类型没有消息与转发，减少对象调用方法的查找时间，即使有缓存）。
- 减少动态：动态派发，private或final关键字，或者在开启Whole Module Optimization选项，声明为internal级别的没有被重载的方法下，将直接调用，在编译时确定。
- 合并动态库（FIXME）。

代码与实现相关

- 向前声明：@class CLASSNAME，而不是#import CLASSNAME.h
- 常用头文件放到预编译文件里（pch文件是预编译文件，执行XCode build之前就已经被预编译，并且引入到每一个.m文件里了）
- 纯代码方式而不是storyboard加载首页UI。
- 对didFinishLaunching里的函数考虑能否挖掘可以延迟加载或者懒加载。
- 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。
- 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。
- 安装包大小的优化（清理无用资源，代码或使用工具）

拓展篇

其他常见语言编译与执行过程

java的启动过程：

- java代码也是要经过编译前端的全部流程，只不过到中间码这一步产生了分歧
- Java需要一个Java Runtime Environment，这里面就有JVM，Java运行环境就是可以识别这种字节码的运行环境，如果一个设备，内部安装好了Java Runtime Environment，他不需要通过汇编来执行代码，Java运行环境可以直接将字节码输入，然后在java自己的虚拟机JVM里来运行。
    - 词法分析
    - 语法分析
    - 生成AST
    - 生成字节码（这个东西其实对应的就类似LLVM中的IR中间码）
    - 目标设备必须具备 Java Runtime Environment
    - 通过Java运行环境来执行字节码
- JAVA的机制是，一次编译生成后，字节码可以每次运行的时候直接使用

JavaScript的编译过程

- js也是需要进行编译的，但是使用的不同引擎，可能内部的执行流程完全不一样。
  js代码会直接在运行的时候输入给JSCore，JSCore也会进行如下的步骤
- 预处理->词法分析->语法分析->生成语法树->生成字节码
- JavaScript的机制是，每次运行的时候，再进行编译生成字节码，然后执行，下次运行，又要重新编译生成字节码，

PHP的执行过程

- PHP实现了一个典型的动态语言执行过程：拿到一段代码后，经过词法解析、语法解析等阶段后，源程序会被翻译成一个个指令(opcodes)，然后ZEND虚拟机顺次执行这些指令完成操作。PHP本身是用C实现的，因此最终调用的也都是C的函数，实际上，我们可以把PHP看做是一个C开发的软件。

- PHP的执行的核心是翻译出来的一条一条指令，也即opcode。

资料

- 2016 WWDC（苹果提供的启动优化方案）
- https://developer.apple.com/videos/play/wwdc2016/406/

- 优化 App 的启动时间（各个阶段优化与处理）
- http://ios.jobbole.com/90331/

- iOS 程序 main 函数之前发生了什么
- http://blog.sunnyxx.com/2014/08/30/objc-pre-main/

- 点击 Run 之后发生了什么？（Build类似）
- http://www.jianshu.com/p/d5cf01424e92

- Xcode编译性能优化（各个阶段优化实战与对比）
- http://blog.csdn.net/qq_25131687/article/details/52194034

感谢聆听

-Q&A-

资源地址：https://github.com/al1020119/iOS-App-Start-Up-Process