目录

iOS逆向开发：dyld动态链接

- 最新版本：`v0.5.1`
- 更新时间：`20240617`

简介

整理关于iOS逆向期间涉及到的dyld动态链接的各种内容。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/ios_re_dyld_link: iOS逆向开发：dyld动态链接

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- iOS逆向开发：dyld动态链接 book.crifan.org
- iOS逆向开发：dyld动态链接 crifan.github.io

离线下载阅读

- iOS逆向开发：dyld动态链接 PDF
- iOS逆向开发：dyld动态链接 ePub
- iOS逆向开发：dyld动态链接 Mobi

版权和用途说明

鸣谢

感谢我的老婆**陈雪**的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

其他

作者的其他电子书

本人 `crifan` 还写了其他 `150+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

关于作者

关于作者更多介绍，详见：

关于CrifanLi李茂 – 在路上

dyld概述

- `dyld` = `DYLD` = **DY**namic **L**oa**D**ing

dyld库中的函数

- dyld库中的函数
  - dyld库文件：文件位置
    - `/cores/dyld`
    - `/usr/lib/dyld`
  - dyld库文件：内部结构
    - section
      - `dyld.__TEXT.__text`
  - dyld库 内部的
    - 函数
      - C函数
        - NSVersionOfRunTimeLibrary
        - NSVersionOfLinkTimeLibrary
        - _NSGetExecutablePath
        -
        - statfs64
        -
        - dyld开头的函数
          - dyld_stub_binder
          - dyld_program_sdk_at_least
          - dyld_shared_cache_file_path
        - _dyld开头的函数
          - _dyld_start
          - _dyld_get_all_image_infos
      - 类的函数
        - dyld类的函数
          - dyld::notifySingle
          - dyld::initializeMainExecutable
          - dyld::_main
          - dyld::registerAddCallback
        - ImageLoaderMachO类的函数
          - ImageLoaderMachO::doModInitFunctions
          - ImageLoaderMachO::doInitialization
        - ImageLoader类的函数
          - ImageLoader::recursiveInitialization
          - ImageLoader::processInitializers
          - ImageLoader::runInitializers
        - dyldbootstrap类的函数
          - dyldbootstrap::start
        - dyld3类的函数
          - dyld3::dladdr
          - dyld3::AllImages::dlopen
          - dyld3::OverflowSafeArray
          - dyld3::MachOFile::forEachLoadCommand
          - dyld3::MachOFile::getUuid

        - dyld4类的函数
            - dyld4::RuntimeState::initialize
            - dyld4::APIs::_libdyld_initialize
    - 全局变量
        - dyld`_main_thread
        - dyld`initialPoolContent
        - gProcessInfo
            - dyld4::gProcessInfo
            - dyld::gProcessInfo
- 相关
    - **libdyld.dylib** 动态库
        - 文件位置
            - `/usr/lib/system/libdyld.dylib`
        - 内部函数
            - dlopen
            - dlopen_internal
            - dlopen_preflight
            - dladdr
            - _dyld_image_count
            - _dyld_get_image_name
            - _dyld_get_image_header
            - _dyld_image_slide
            - _dyld_register_func_for_add_image
            - _dyld_register_func_for_remove_image
            - __dyld_private ?
    - getsectdata

dyld

- dyld头文件位置：`mach-o/dyld.h`
  - 导入头文件

```
#import <mach-o/dyld.h>
```

dyld版本

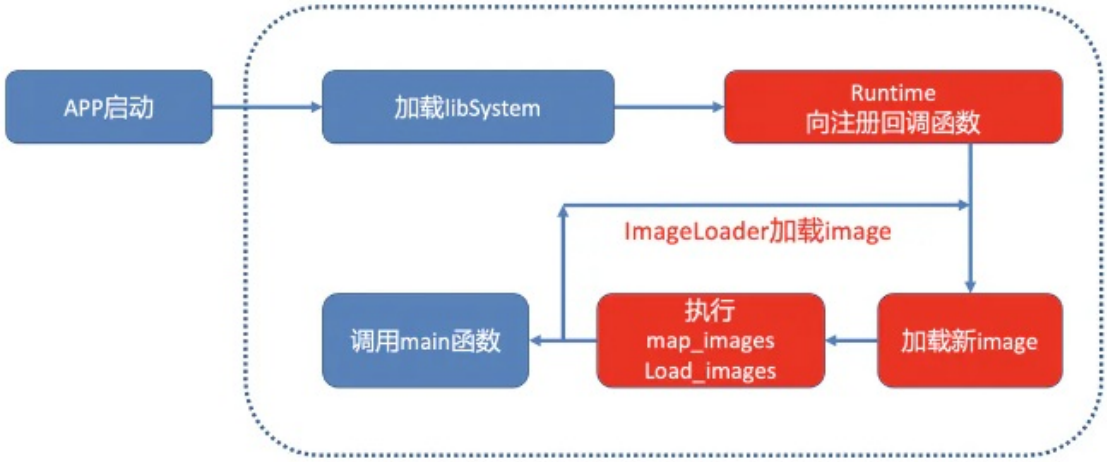
dyld源码

- dyld源码
  - 离线下载
    - https://opensource.apple.com/tarballs/dyld/
      - dyld-1125.5
        - `zip`：https://github.com/apple-oss-distributions/dyld/archive/refs/tags/dyld-1125.5.zip
        - `tag.gz`：https://github.com/apple-oss-distributions/dyld/archive/refs/tags/dyld-1125.5.tar.gz

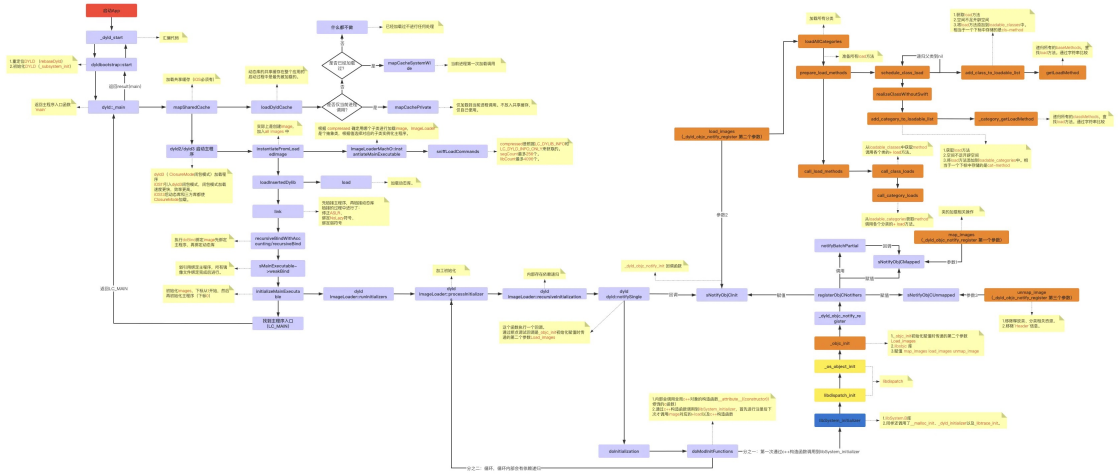
dyld加载过程

iOS的app启动过程

- iOS的app启动过程
  - 概述

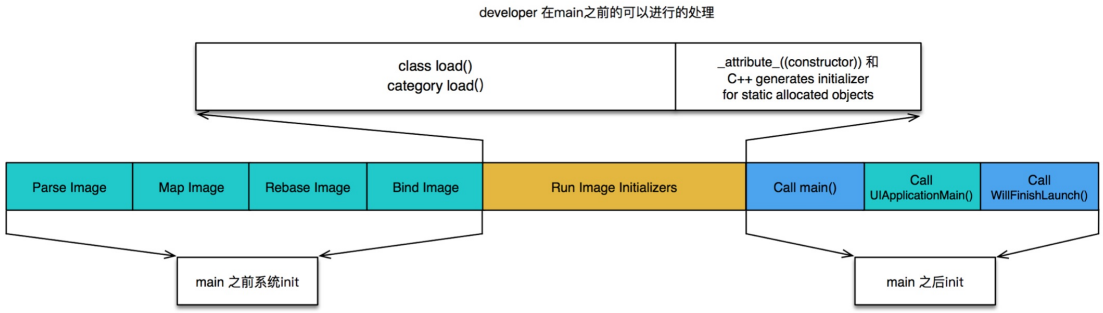


  - 详细



- iOS的app启动的不同阶段
  - `Pre-main` 阶段

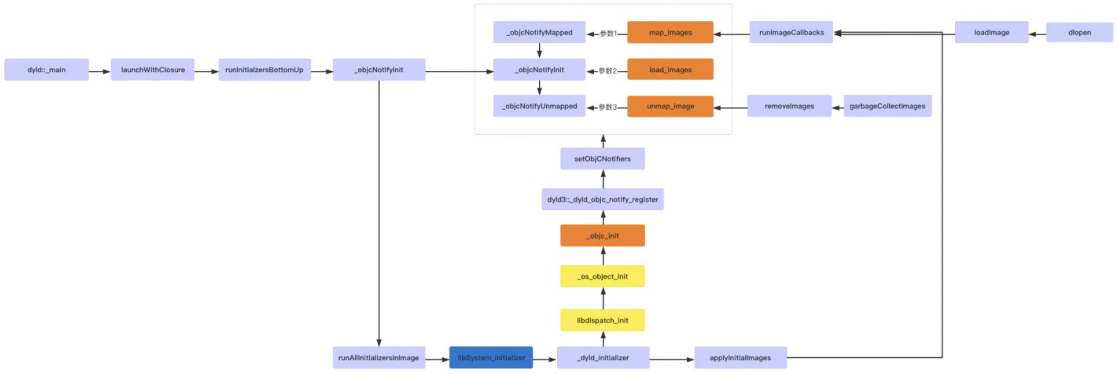
  - `main` 阶段



- iOS的app启动调用函数

dyld加载过程

- dyld2 vs dyld3

- dyld加载过程
  - 图
    - 

  - 文字版
    - 图

- `DYLD` 动态链接器：记载所有库和可执行文件。
- `DYLD` 加载流程：
  - 系统内核调用 `_dyld_start`
    - 重定位 `DYLD` （ `rebaseDyld` ）
    - 初始化 `DYLD` （ `_subsystem_init` ）
    - 调用 `dyld main` 函数（ `dyld::_main` ）：
      - 加载共享缓存（ `mapSharedCache` ）
        - 实际调用 `loadDyldCache` 分为三种情况
          - 仅加载到当前进程调用 `mapCachePrivate` 。不放入共享缓存，仅自己使用。
          - 已经加载过不进行任何处理。
          - 当前进程第一次加载调用 `mapCacheSystemWide`
      - `dyld2/dyld3` （ `ClosureMode` 闭包模式）加载程序（ `iOS11` 引入 `dyld3` 闭包模式，闭包模式加载速度更快，效率更高。 `iOS13后` 动态库和三方库都使 `ClosureMode` 加载。）：
        - `dyld3` ：
          - 找到/创建 `mainClosure`
          - 通过 `launchWithClosure` 启动主程序，成功后返回 `result` (主程序入口 `main`)。逻辑和 `dyld2` 启动主程序逻辑基本相同。
        - `dyld2` ：启动主程序
          - 实例化主程序 `instantiateFromLoadedImage` （实际上是创建 `image` ）
            - 调用 `sniffLoadCommands` 生成相关信息，比如 `compressed` 。根据 `compressed` 来确定使用哪个类来实例化。
              - `compressed` 是根据 `LC_DYLIB_INFO` 和 `LC_DYLD_INFO_ONLY` 来获取的。
              - `segCount` 最多 `256` 个。
              - `libCount` 最多 `4096` 个。
            - 实例化生成 `image` ，加入 `all images` 中。
          - 插入&加载动态库 `loadInsertedDylib`
            - 根据上下文初始化配置调用load加载动态库。
          - 链接主程序和链接插入动态库(`link` ，主程序链接在前)
            - 修正 `ASLR` 、绑定 `NoLazy` 符号、绑定弱符号
          - 初始化主程序 `initializeMainExecutable` （核心方法）
            - 初始化 `images` ，下标从 `1` 开始，然后再初始化主程序（下标 `0` ）调用 `dyld ImageLoader::runInitializers` ：
              - `dyld ImageLoader::processInitializers` ：
                - `dyld ImageLoader::recursiveInitialization` ：
                  - `dyld dyld::notifySingle` ：
                    - 这个函数执行一个回调。
                    - 通过断点调试回调是 `_objc_init` 初始化赋值时传递的第二个参数 `Load_images`
                      - `Load_images` 中调用了 `call_load_methods` 函数
                        - `call_class_loads` ：调用各个类的 `+ load` 方法。
                        - `call_category_loads` :调用各个分类的 `+ load` 方法。
                  - `doInitialization`
                    - 最终会调用到 `doModInitFunctions`
                      - 内部会调用全局 `c++` 对象的构造函数（ `attribute((constructor))` 修饰的 `c` 函数）
                      - 会首先调用 `libSystem_initializer` 构造函数进行回调的注册。
                        - 回调有三个参数（ `map_images` 、 `load_images` 、 `unmap_image` ）
                          - `map_images` 进行类的加载，在回调函数注册后就立马调用。
                          - `load_images` 在 `notifySingle` 循环中调用。
                          - `unmap_image` 在异常/回收/检查镜像文件的时候调用。
            - 找到主程序入口 `LC_MAIN` ，然后返回主程序（ `main` ）。
- `load` 、 `C++` 构造函数、 `main` 调用总结：
  - `Dyld` 初始化 `image` 是按 `Link Binary With Libraries` 顺序逐个初始化的，从下标 `1` 开始，最后再初始化主程序（下标 `0` ）。可以理解为是按 `image` 进行分组的。
  - `image` 内部是先加载所有类的 `+ load` ，再加载分类的 `+ load` ，最后加载 `C++` 全局构造函数。（ `类load->分类load->C++构造函数` ）。 `+load` 是 `objc` 中调用的， `C++全局构造函数` 是在 `dyld` 中调用的。(在不考虑二进制重排等的优化下， `image` 内部的顺序默认是按 `Compile Sources` 中顺序进行的)。
  - `main` 函数是在 `dyld` 返回入口函数（ `main` ）之后才调用的。

dyld加载过程1

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 23.2
  * frame #0: 0x000000018d124174 libdyld.dylib`dladdr
```

```
    frame #1: 0x000000010b0bfbac AwemeCore`___lldb_unnamed_symbol13025$$AwemeCore + 80
    frame #2: 0x000000018d041c38 libobjc.A.dylib`CALLING_SOME_+initialize_METHOD + 20
    frame #3: 0x000000018d04742c libobjc.A.dylib`initializeNonMetaClass + 644
    frame #4: 0x000000018d0471f8 libobjc.A.dylib`initializeNonMetaClass + 80
    frame #5: 0x000000018d047ba8 libobjc.A.dylib`initializeAndMaybeRelock(objc_class*, objc_object*, mutex_tt<false>&
, bool) + 284
    frame #6: 0x000000018d05450c libobjc.A.dylib`lookUpImpOrForward + 700
    frame #7: 0x000000018d0448a8 libobjc.A.dylib`object_setClass + 104
    frame #8: 0x000000018d29da70 CoreFoundation`_CFRuntimeCreateInstance + 580
    frame #9: 0x000000018d2bcb74 CoreFoundation`__CFStringCreateImmutableFunnel3 + 1944
    frame #10: 0x000000018d2bcef8 CoreFoundation`CFStringCreateWithCString + 92
    frame #11: 0x000000018d29f610 CoreFoundation`__CFInitialize + 812
    frame #12: 0x0000000104fddfac dyld`ImageLoaderMachO::doImageInit(ImageLoader::LinkContext const&) + 248
    frame #13: 0x0000000104fde580 dyld`ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) + 40
    frame #14: 0x0000000104fd95d0 dyld`ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&, unsigned
 int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) + 548
    frame #15: 0x0000000104fd953c dyld`ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&, unsigned
 int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) + 400
    frame #16: 0x0000000104fd8334 dyld`ImageLoader::processInitializers(ImageLoader::LinkContext const&, unsigned int
, ImageLoader::InitializerTimingList&, ImageLoader::UninitedUpwards&) + 184
    frame #17: 0x0000000104fd83fc dyld`ImageLoader::runInitializers(ImageLoader::LinkContext const&, ImageLoader::Ini
tializerTimingList&) + 92
    frame #18: 0x0000000104fca3d0 dyld`dyld::initializeMainExecutable() + 136
    frame #19: 0x0000000104fcedb4 dyld`dyld::_main(macho_header const*, unsigned long, int, char const**, char const*
*, char const**, unsigned long*) + 4616
    frame #20: 0x0000000104fc9208 dyld`dyldbootstrap::start(dyld3::MachOLoaded const*, int, char const**, dyld3::Mach
OLoaded const*, unsigned long*) + 396
    frame #21: 0x0000000104fc9038 dyld`_dyld_start + 56
```

==

- dyld`_dyld_start
  - dyld`dyldbootstrap::start
    - dyld`dyld::_main
      - dyld`dyld::initializeMainExecutable
        - dyld`ImageLoader::runInitializers
          - dyld`ImageLoader::processInitializers
            - dyld`ImageLoader::recursiveInitialization
              - dyld`ImageLoaderMachO::doInitialization
                - dyld`ImageLoaderMachO::doImageInit
                  - ...
                    - libdyld.dylib`dladdr

dyld加载过程2

- _dyld_start
  - dyldbootstrap: : start
    - dyld: : _main
      - dyld: : initializeMainExecutable
        - ImageLoader::runInitializers
          - ImageLoader::processInitializers
            - ImageLoader::recursiveInitialization
              - Dyld ::notifySingle
                - libobjC.a. dylib load_images
                  - +[ViewController load]

dyld加载过程3

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 6.1
  * frame #0: 0x0000000102495310 libAwemeDylib.dylib`_logos_method$_ungrouped$NSString$stringByAppendingString$(self=
"https://", _cmd="stringByAppendingString:", aString=0x0000000000000000) at AwemeDylib.xm:175:29
    frame #1: 0x0000000108532cf8 AwemeCore`___lldb_unnamed_symbol3548$$AwemeCore + 520
```

```
frame #38: 0x00000001c2ecf60c FrontBoardServices`-[FBSSerialQueue _performNextFromRunLoopSource] + 28
frame #39: 0x00000001bdcd0a00 CoreFoundation`__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__ + 24
frame #40: 0x00000001bdcd0958 CoreFoundation`__CFRunLoopDoSource0 + 80
frame #41: 0x00000001bdcd00f0 CoreFoundation`__CFRunLoopDoSources0 + 180
frame #42: 0x00000001bdccb23c CoreFoundation`__CFRunLoopRun + 1080
frame #43: 0x00000001bdccaadc CoreFoundation`CFRunLoopRunSpecific + 464
frame #44: 0x00000001c7c6b328 GraphicsServices`GSEventRunModal + 104
frame #45: 0x00000001c1dd863c UIKitCore`UIApplicationMain + 1936
frame #46: 0x000000010efec094 AwemeCore`awemeMain + 200
frame #47: 0x0000000102267ca4 Aweme`___lldb_unnamed_symbol21$$Aweme + 12
frame #48: 0x00000001bdb54360 libdyld.dylib`start + 4
```

- libdyld.dylib`start 开始的调用顺序
  - app相关逻辑：Aweme `___lldb_unnamed_symbol21$$Aweme、AwemeCore` awemeMain
    - 然后才是其他系统常见函数
      - UIKitCore`UIApplicationMain
        - ...

相关函数

NSVersionOfRunTimeLibrary

NSVersionOfLinkTimeLibrary

_NSGetExecutablePath

_dyld_start

_dyld_get_all_image_infos

dyld_program_sdk_at_least

dyld_shared_cache_file_path

相关变量

gProcessInfo

libdyld.dylib

相关函数

dlopen

dlopen_internal

dlopen_preflight

dlsym

dladdr

_dyld_image_count

_dyld_get_image_name

_dyld_get_image_header

_dyld_image_slide

_dyld_register_func_for_add_image

_dyld_register_func_for_remove_image

dyld_stub_binder

反汇编代码

```
libdyld.dylib`dyld_stub_binder:
->  0x18d122dc0 <+0>:   stp    x29, x30, [sp, # 0x10]
    0x18d122dc4 <+4>:   mov    x29, sp
    0x18d122dc8 <+8>:   sub    sp, sp, #0xf0             ; =0xf0
    0x18d122dcc <+12>:  stp    x0, x1, [x29, # 0x10]
    0x18d122dd0 <+16>:  stp    x2, x3, [x29, # 0x20]
    0x18d122dd4 <+20>:  stp    x4, x5, [x29, # 0x30]
    0x18d122dd8 <+24>:  stp    x6, x7, [x29, # 0x40]
    0x18d122ddc <+28>:  stp    x8, x9, [x29, # 0x50]
    0x18d122de0 <+32>:  stp    q0, q1, [x29, # 0x80]
    0x18d122de4 <+36>:  stp    q2, q3, [x29, # 0xa0]
    0x18d122de8 <+40>:  stp    q4, q5, [x29, # 0xc0]
    0x18d122dec <+44>:  stp    q6, q7, [x29, # 0xe0]
    0x18d122df0 <+48>:  ldr    x0, [x29, #0x18]
    0x18d122df4 <+52>:  ldr    x1, [x29, #0x10]
    0x18d122df8 <+56>:  bl     0x18d1246e4               ; _dyld_fast_stub_entry(void*, long)
    0x18d122dfc <+60>:  mov    x16, x0
    0x18d122e00 <+64>:  ldp    x0, x1, [x29, # 0x10]
    0x18d122e04 <+68>:  ldp    x2, x3, [x29, # 0x20]
    0x18d122e08 <+72>:  ldp    x4, x5, [x29, # 0x30]
    0x18d122e0c <+76>:  ldp    x6, x7, [x29, # 0x40]
    0x18d122e10 <+80>:  ldp    x8, x9, [x29, # 0x50]
    0x18d122e14 <+84>:  ldp    q0, q1, [x29, # 0x80]
    0x18d122e18 <+88>:  ldp    q2, q3, [x29, # 0xa0]
    0x18d122e1c <+92>:  ldp    q4, q5, [x29, # 0xc0]
    0x18d122e20 <+96>:  ldp    q6, q7, [x29, # 0xe0]
    0x18d122e24 <+100>: mov    sp, x29
    0x18d122e28 <+104>: ldp    x29, x30, [sp], #0x10
    0x18d122e2c <+108>: add    sp, sp, #0x10             ; =0x10
    0x18d122e30 <+112>: br     x16
```

相关内容

dyld_shared_cache

相关环境变量

- dyld相关环境变量
  - `DYLD_LIBRARY_PATH`
  - `DYLD_PRINT_STATISTICS`
  - `DYLD_PRINT_LIBRARIES`
  - `DYLD_INSERT_LIBRARIES`
  - `DYLD_IMAGE_SUFFIX`

相关函数

getsect

_getsectbynamefromheader_64

getsegmentdata

相关工具

dyldinfo

Mach-O中

- Mach-O中有关dyld的内容
  - Header的flag
    - `MH_DYLDLINK`
  - LC_开头的dyld相关的Load Command
    - `LC_DYLD_CHAINED_FIXUPS`
    - `LC_DYLD_EXPORTS_TRIE`
    - `LC_LOAD_DYLINKER`
      - `/usr/lib/dyld`
    - `LC_DYLD_INFO`
    - `LC_DYLD_INFO_ONLY`
    - `LC_DYLD_ENVIRONMENT`

附录

下面列出相关参考资料。

参考资料

- 【未解决】iOS反越狱检测：strcmp
- 【未解决】研究抖音越狱检测逻辑：___lldb_unnamed_symbol13025即sub_5BABB5C
- 【已解决】Xcode的lldb中动态调试objc_msgSend第二个参数函数名或属性值
- 【记录】分析和研究：当stringByAppendingString参数为空时的AwemeCore相关函数代码段
- 【已解决】研究抖音越狱检测逻辑：open_dprotected_np
- 【未解决】研究抖音是否实现了Method Swizzling的Hook检测
-
- IOS APP startup optimization (VII) : Detailed analysis of dyLD loading process - Moment For Technology (mo4tech.com)
- iOS逆向攻防实战 - 掘金 (juejin.cn)
-