



## 刘坤的技术博客

- [BlueBox](#)
- [所有文章](#)

嗨，我是刘坤，一名来自中国的 IOS 开发者，现就职于杭州阿里，花名‘念纪’，沉淀技术，寻求创意

[GitHub](#) [RSS](#)

### 友情链接

- [Casatwy Taloyum](#)
- [gf&zjの盗梦空间](#)
- [明弈](#)

## dyld与ObjC

dyld 是ios上的二进制加载器，如何剖析这个过程呢？

### 0x10 dyld

dyld是加载mach-o的库。一切都从 `_dyld_start` 开始，拉到源码看下，这是个汇编方法（arm64）：

```
274  -- .data
275  .align 3
276  __dso_static:
277  .quad  __dso_handle
278
279  .text
280  .align 2
281  .globl  __dyld_start
282  __dyld_start:
283  mov     x28, sp
284  and     sp, x28, #~15           // force 16-byte alignment of stack
285  mov     x0, #0
286  mov     x1, #0
287  stp     x1, x0, [sp, #-16]!     // make aligned terminating frame
288  mov     fp, sp                // set up fp to point to terminating frame
289  sub     sp, sp, #16           // make room for local variables
290  ldr     x0, [x28]              // get app's mh into x0
291  ldr     x1, [x28, #8]          // get argc into x1 (kernel passes 32-bit int
292  add     x2, x28, #16           // get argv into x2
293  adrp    x4, __dso_handle@page
294  add     x4, x4, __dso_handle@pageoff // get dyld's mh in to x4
295  adrp    x3, __dso_static@page
296  ldr     x3, [x3, __dso_static@pageoff] // get unslid start of dyld
297  sub     x3, x4, x3             // x3 now has slide of dyld
298  mov     x5, sp                // x5 has &startGlue
299
300  // call dyldbootstrap::start(app_mh, argc, argv, slide, dyld_mh, &startGlue)
301  bl      __ZN13dyldbootstrap5startEPK12macho_headeriPPKc1S2_Pm
302  mov     x16, x0                // save entry point address in x16
303  ldr     x1, [sp]
304  cmp     x1, #0
305  b.ne    Lnew
306
307  // LC_UNIXTHREAD way, clean up stack and jump to result
308  add     sp, x28, #8           // restore unaligned stack pointer without ap
309  br      x16                   // jump to the program's entry point
```

找到 `dyldbootstrap::start` 这个方法，看到最后调用到了 `dyld::_main` 这个方法。 `dyld::_main`的源码较长，里面就整个加载过程，有兴趣同学可以下载dyld来看，过程大概如下：

1. 设置运行环境，环境变量
2. 实例化Image
3. 加载共享缓存
4. 动态库的版本化重载
5. 加载插入的动态库
6. link主程序
7. link插入的动态库
8. weakBind
9. initialize

10. 1. main

本文会对其中几步进行描述：

## 0x11 实例化可执行文件

在 `dyld::_main` 中找到了 `instantiateFromLoadedImage`，这个方法就是实例化的过程。

```
1 static ImageLoader* instantiateFromLoadedImage(const macho_header* mh, uintptr_t slide, const char* path)
2 {
3     // try mach-o loader
4     if ( isCompatibleMachO((const uint8_t*)mh, path) ) {
5         ImageLoader* image = ImageLoaderMachO::instantiateMainExecutable(mh, slide, path, gLinkContext);
6         addImage(image);
7         return image;
8     }
9
10    throw "main executable not a known format";
11 }
```

从这个方法中，我们大致可以看到加载有三部：

`isCompatibleMachO => instantiateMainExecutable => addImage`

字面意思已经挺明确了：`* isCompatibleMachO` 是检查mach-o的subtype是否是当前cpu可以支持；`* instantiateMainExecutable` 就是实例化可执行文件，这个期间会解析LoadCommand，这个之后会发送 `dyld_image_state_mapped` 通知；`* addImage` 添加到 `allImages` 中。

## 0x12 link过程都做了什么

实例化之后就是动态链接的过程 link 这个过程就是将加载进来的二进制变为可用状态的过程。简单来说就是：

`rebase => binding`

这些过程的信息都存储在LoadCommand的 `LC_DYLD_INFO` 这个cmd中。解析出来会得到这样的信息：

```
1 struct dyld_info_command {
2     uint32_t cmd; /* LC_DYLD_INFO or LC_DYLD_INFO_ONLY */
3     uint32_t cmdsize; /* sizeof(struct dyld_info_command) */
4     uint32_t rebase_off; /* file offset to rebase info */
5     uint32_t rebase_size; /* size of rebase info */
6     uint32_t bind_off; /* file offset to binding info */
7     uint32_t bind_size; /* size of binding info */
8     uint32_t weak_bind_off; /* file offset to weak binding info */
9     uint32_t weak_bind_size; /* size of weak binding info */
10    uint32_t lazy_bind_off; /* file offset to lazy binding info */
11    uint32_t lazy_bind_size; /* size of lazy binding infs */
12    uint32_t export_off; /* file offset to export info */
13    uint32_t export_size; /* size of export infs */
14 };
```

这里面分别记录了哪些地址需要被 rebase, binding 等。

什么是rebase? 为什么要做rebase?

rebase就是针对“mach-o在加载到内存中不是固定的首地址”这一现象做数据修正的过程。

什么是binding?

binding就是将这个二进制调用的外部符号进行绑定的过程。比如我们objc代码中需要使用到NSObject, 即符号 `_OBJC_CLASS_$_NSObject`，但是这个符号又不在我们的二进制中，在系统库 Foundation.framework中，因此就需要binding这个操作将对应关系绑定到一起。

什么是lazyBinding?

lazyBinding就是在加载动态库的时候不会立即binding, 当时当第一次调用这个方法的时候再实施binding。做到的方法也很简单：通过 `dyld_stub_binder` 这个符号来做。lazy binding的方法第一次会调用到 `dyld_stub_binder`，然后 `dyld_stub_binder` 负责找到真实的方法，并且将地址bind到桩上，下一次就不用再bind了。

什么是weakBinding?

weakBind 这个我也没有太搞懂什么时候会有weakBind的符号，应是在c++中的场景。oc没看到过。但是从代码中可以看出这一步会对所有含有弱符号的镜像合并排序进行bind。

篇幅有限，这块就讲这些。有兴趣的同学可以撸源码！

## 0x13 initialize

这一步就是执行initialize的方法的时候了。也就是C++ 中的 `attribute((constructor))` 方法。编译在mach-o里面会有一个section记录了这些方法，如下：

▼ Shared Library (ARM64_ALL)	Offset	Data	Description	Value
*Mach64 Header				
▶ Load Commands				
▶ Section64 (__TEXT,__text)				
▶ Section64 (__TEXT,__stubs)				
▶ Section64 (__TEXT,__stub_helper)				
▶ Section64 (__TEXT,__cstring)				
▶ Section64 (__TEXT,__objc_classname)				
▶ Section64 (__TEXT,__objc_methname)				
▶ Section64 (__TEXT,__objc_methtype)				
Section64 (__TEXT,__gcc_except_tab)				
Section64 (__TEXT,__const)				
Section64 (__TEXT,__unwind_info)				
▼ Section64 (__DATA,__got)				
*Non-Lazy Symbol Pointers				
▼ Section64 (__DATA,__la_symbol_ptr)				
*Lazy Symbol Pointers				
▼ Section64 (__DATA,__mod_init_func)				
*Module Init Func Pointers				
Section64 (__DATA,__const)				
	000101C8	0000000000008B54	Pointer	0x8B54 ( _testABCD)

上图的步骤中我们也可以看到，这一步是在main函数之前的。

## 0x20 ObjC

上面都是在讲dyld, 那么是如何与ObjC关联起来的呢？ ObjC的运行时是什么时候启动的呢？ +Load方法是什么时候调用的呢？

### 0x21 objc的启动

翻一下objc的源码，发现了 objc\_init这个方法，实现看起来很简单，贴一下源码：

```
1 void _objc_init(void)
2 {
3     static bool initialized = false;
4     if (initialized) return;
5     initialized = true;
6
7     // 各种初始化
8     environ_init();
9     tls_init();
10    static_init();
11    lock_init();
12    // 看了一下exception_init是空实现!! 就是说objc的异常是完全采用c++那一套的。
13    exception_init();
14    // 注册dyld事件的监听
15    _dyld_objc_notify_register(&map_2_images, load_images, unmap_image);
16 }
```

这个方法是什么时候调用的呢,断点一下：

0 _objc_init
1 _os_object_init
2 libdispatch_init
3 libSystem_initializer
4 ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext const&)
5 ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&)
6 ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&, unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitUpwards&)
7 ImageLoader::recursiveInitialization(ImageLoader::LinkContext const&, unsigned int, char const*, ImageLoader::InitializerTimingList&, ImageLoader::UninitUpwards&)
8 ImageLoader::processInitializers(ImageLoader::LinkContext const&, unsigned int, ImageLoader::InitializerTimingList&, ImageLoader::UninitUpwards&)
9 ImageLoader::runInitializers(ImageLoader::LinkContext const&, ImageLoader::InitializerTimingList&)
10 dyld::initializeMainExecutable()
11 dyld::_main(macho_header const*, unsigned long, int, char const**, char const**, char const**, unsigned long*)
12 _dyld_start

根据这张图，再结合dyld的知识，原来：

objc\_init是在 libsystem 中的一个initialize方法 libsystem\_initializer中初始化了 libdispatch, 然后libdispatch调用了\_os\_object\_init, 最终调用了 \_objc\_init.

如下：

```
1 _os_object_init(void)
2 {
3     _objc_init();
4     Block_callbacks_RR callbacks = {
5         sizeof(Block_callbacks_RR),
6         (void (*)(const void *))&objc_retain,
7         (void (*)(const void *))&objc_release,
8         (void (*)(const void *))&_os_objc_destructInstance
9     };
10 }
```

```

9  };
10 _Block_use_RR2(&callbacks);
11 #if DISPATCH_COCOA_COMPAT
12 const char *v = getenv("OBJC_DEBUG_MISSING_POOLS");
13 _os_object_debug_missing_pools = v && !strcmp(v, "YES");
14 #endif
15 }

```

\_dyld\_objc\_notify\_register 这个方法在苹果开源的dyld里面可以找到，然后看到调用了dyld::registerObjCNotifiers这个方法：

```

1 void registerObjCNotifiers(_dyld_objc_notify_mapped mapped, _dyld_objc_notify_init init, _dyld_objc_notify_unmapped unmapped)
2 {
3     // record functions to call
4     sNotifyObjCMapped = mapped;
5     sNotifyObjCInit = init;
6     sNotifyObjCUnmapped = unmapped;
7
8     // call 'mapped' function with all images mapped so far
9     // 第一次先触发一次ObjCMapped
10    try {
11        notifyBatchPartial(dyld_image_state_bound, true, NULL, false, true);
12    }
13    catch (const char* msg) {
14        // ignore request to abort during registration
15    }
16 }

```

从字面意思可以明白，传进来的分别是 map, init, unmap事件的回调。dyld的事件通知有以下几种，分别会在特定的时机发送：

```

1 enum dyld_image_states
2 {
3     dyld_image_state_mapped = 10, // No batch notification for this
4     dyld_image_state_dependents_mapped = 20, // Only batch notification for this
5     dyld_image_state_rebased = 30,
6     dyld_image_state_bound = 40,
7     dyld_image_state_dependents_initialized = 45, // Only single notification for this
8     dyld_image_state_initialized = 50,
9     dyld_image_state_terminated = 60 // Only single notification for this
10 };

```

大家可能奇怪，上面第一次触发mapped的为啥发送的是 bound 事件。因为此Mapped的非彼mapped。ObjCMapped实际上是在 binding结束之后执行的。

## 0x22 ObjC map images

下面再来看看 map\_2\_images，就是这个ObjCMapped 干了啥，简单来说就是对这个二进制中的ObjC相关的信息进行初始化。关键信息可以看\_read\_images：

- init classes map: 第一次调用时会初始化一个全局的一个Map: gdb\_objc\_realized\_classes 用来存放class
- readClasses: 这一步会把class从二进制里面读出来，然后将 class\_ro\_t 替换为 class\_rw\_t。class\_ro会放在class\_rw\_t里面。然后把class加入到第一步创建的gdb\_objc\_realized\_classes里面。

注意，这个时候，虽然放到了gdb\_objc\_realized\_classes 但是class还没有realized, 后面会有realize的步骤 附图：

The diagram illustrates the process of converting a read-only class structure into a mutable one. It starts with a memory dump of the `_OBJC_CLASS_$_IAMTEST_Student` structure, showing fields like `_OBJC_METACLASS_$_IAMTEST_Student`, `_OBJC_CLASS_$_IAMTEST_StudentBase`, `_objc_empty_cache`, `0x0`, and `0x100076550`. Below this, the `class_ro_t` structure is defined, containing fields for flags, instance start/end, instance size, reserved space, ivar layout, name, method list, base method list, protocol list, base protocols, ivars, weak ivar layout, base properties, and base methods. A green arrow points to the `class_rw_t` structure, which is a mutable version of `class_ro_t`. It contains the same fields as `class_ro_t` but also includes `firstSubclass`, `nextSiblingClass`, `demangledName`, and methods for setting and clearing flags, and changing flags. The `setFlags` method uses `OSAtomicOr32Barrier` to set flags, and the `clearFlags` method uses `OSAtomicXor32Barrier` to clear flags. The `changeFlags` method uses `OSAtomicCompareAndSwap32Barrier` to change flags.

- fix selector: selector的唯一性
- read protocols: 读取protocol. 看读取protocol的源码可以发现:
  - ① protocol is an objc\_object!,
  - ② protocol具有唯一性
  - ③ protocol的isa都指向: OBJC\_CLASS\$Protocol
- realizeClasses: 这一步的意义就是动态链接好class, 让class处于可用状态, 主要操作如下:
  - ① 检查ro是否已经替换为rw,没有就替换一下。
  - ② 检查类的父类和metaClass是否已经realize,没有就先把它们先realize
  - ③ 重新layout ivar. 因为只有加载好了所有父类, 才能确定ivar layout
  - ④ 把一些flags从ro拷贝到rw
  - ⑤ 链接class的 nextSiblingClass 链表
  - ⑥ attach categories: 合并categories的method list、 properties、 protocols到 class\_rw\_t 里面
- read categories: 读取categories, 然后attach

## 0x23 ObjC load\_images

load\_images这一步很简单, 就是调用+Load. 前面也看到了, 这个方法是在监听dyld\_image\_state\_dependents\_initialized 这个事件的时候会执行。因此 +Load和 constructor的执行时机是差不多的。

DEBUG一下会发现+Load是在 constructor之前执行的, 为什么呢?

```

1  we are about to initialize this image
2  uint64_t t1 = mach_absolute_time();
3  fState = dyld_image_state_dependents_initialized;
4  oldState = fState;
5  // 当dependent的initializer执行完成之后, 发送dyld_image_state_dependents_initialized事件, 这个时候接收到事件就开着执行Load了
6  context.notifySingle(dyld_image_state_dependents_initialized, this, &timingInfo);
7
8  // 而constructor在这里。
9  bool hasInitializers = this->doInitialization(context);
10
11 // let anyone know we finished initializing this image
12 fState = dyld_image_state_initialized;
13 oldState = fState;
14 context.notifySingle(dyld_image_state_initialized, this, NULL);

```

因此 +Load是还要在constructor之前执行的哦, 但也就是紧挨着执行的。


## 0x30 小结

看了dyld和objc的源码, 感觉学到很多, 本文主要讲了objc怎么run起来的。包括大概的流程, 但是实际上还有很多细节没有详细讲, 比如ObjC的class在二进制中是什么样? 加载到runtime又什么样, 有兴趣的同学可以下载源码详细挖掘, 推荐使用MachOView这个工具来看二进制的结构是怎么样的, 对于理解加载很有帮助。

Share

## Comments

0条评论    [blog.cnbluebox.com](#)

 登录 ▾


 推荐 2     分享

最新发布 ▾



开始讨论...

通过以下方式登录

或注册一个 **DISQUS** 帐号 


姓名

来做第一个留言的人吧！

在 **BLOG.CNBLUEBOX.COM** 上还有


数字证书 - 刘坤的博客

2条评论 • 4年前

 **lizhijiang** — “用公钥解密”这个描述会不会引起误解？改成“用公钥验证”是不是好一点？

你的下拉刷新是否“抖”了一下 - 刘坤的技术博客

2条评论 • 3年前

 **陈大敏** — 有个问题是下拉的时候在加载过程中，就不能再往上拉了。

iOS开发同学的arm64汇编入门

3条评论 • 4个月前

 **刘坤** — 多谢指正！

使用AppleScript制作一个AppIcon自动生成器

1条评论 • 3年前

 **roro** — 请问一下 appleScript 能不能实现以脚本的方式 创建 xcode project

 订阅     在您的网站上使用 **Disqus**添加 **Disqus**添加     隐私

Copyright © 2017 刘坤 Design credit: [Shashank Mehta](#)