

## 05 | 链接器：符号是怎么绑定到地址上的？

戴铭 2019-03-21



00:00

讲述：冯永吉

大小：15.22M

16:36

你好，我是戴铭。

你是不是经常会好奇自己参与的这么些项目，为什么有的编译起来很快，有的却很慢；编译完成后，有的启动得很快，有的却很慢。其实，在理解了编译和启动时链接器所做的事儿之后，你就可以从根儿上找到这些问题的答案了。

所以，在今天这篇文章中，我就重点和你讲解一下链接器相关的知识。**简单地说，链接器最主要的作用，就是将符号绑定到地址上。**理解了这其中的原理后，你就可以有针对性地去调整和优化项目了。

同时，掌握了链接器的作用，也将有助于你理解后面文章中，关于符号表、加载相关的内容。

现在，我们就从 iOS 开发的起点，也就是编写代码和编译代码开始说起，看看链接器在这其中到底发挥了什么作用。

### iOS 开发为什么使用的是编译器？

我们都知道，iOS 编写的代码是先使用编译器把代码编译成机器码，然后直接在 CPU 上执行机器码的。之所以不使用解释器来运行代码，是因为苹果公司希望 iPhone 的执行效率更高、运行

速度能达到最快。

那**为什么说用解释器运行代码的速度不够快呢**？这是因为解释器会在运行时解释执行代码，获取一段代码后就会将其翻译成目标代码（就是字节码（Bytecode）），然后一句一句地执行目标代码。

也就是说，解释器，是在运行时才去解析代码，这样就比在运行之前通过编译器生成一份完整的机器码再去执行的效率要低。

这时你一定会纳闷了，既然编译器效率这么高，那为什么还有人用解释器呢？所谓事有利弊，解释器可以在运行时去执行代码，说明它具有动态性，程序运行后能够随时通过增加和更新代码来改变程序的逻辑。

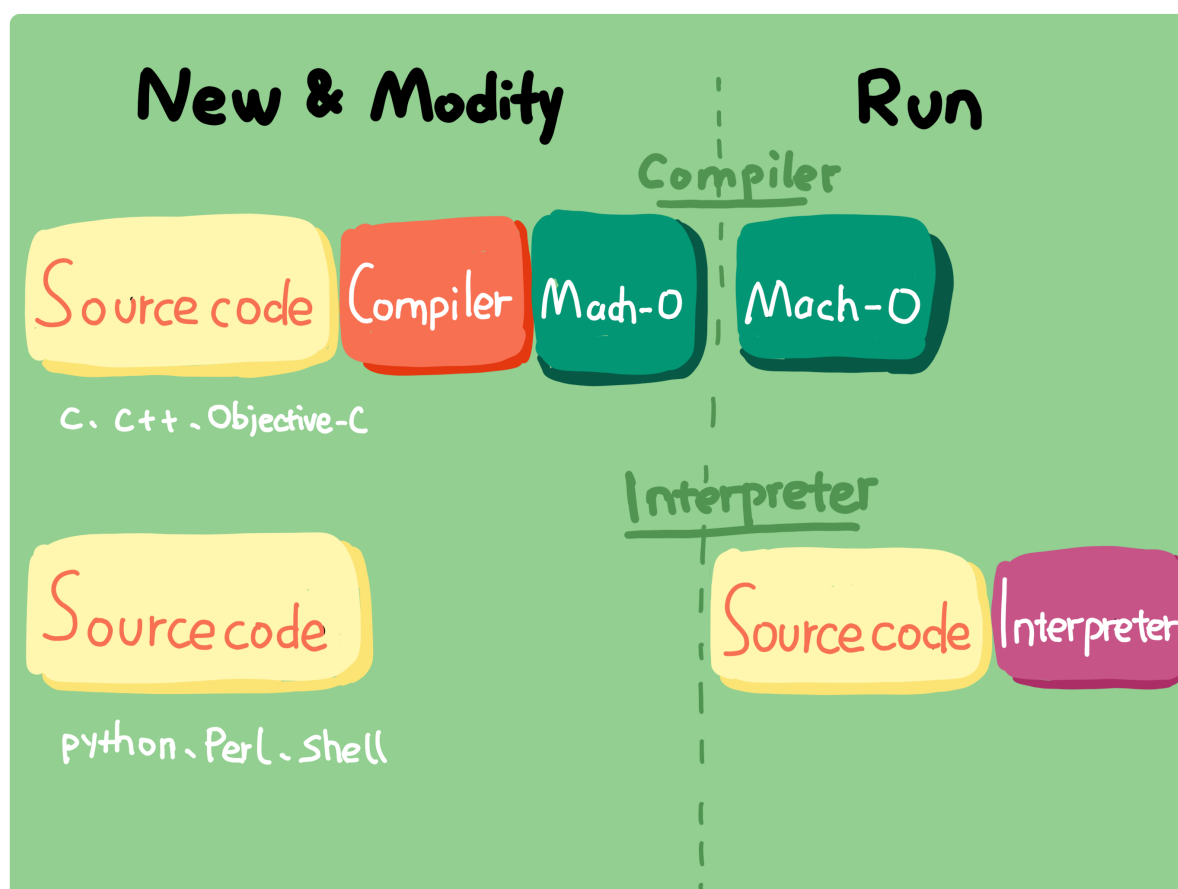
也就是说，你写的程序跑起来后不用重新启动，就可以看到代码修改后的效果，这样就缩短了调试周期。程序发布后，你还可以随时修复问题或者增加新功能，用户也不用一定要等到发布新版本后才可以升级使用。所以说，使用解释器可以帮我们缩短整个程序的开发周期和功能更新周期。

那么，使用编译器和解释器执行代码的特点，我们就可以概括如下：

采用编译器生成机器码执行的好处是效率高，缺点是调试周期长。

解释器执行的好处是编写调试方便，缺点是执行效率低。

编译器和解释器的比较图示如下：



明确了 iOS 开发使用编译器的原因以后，你还需要了解 **iOS 开发使用的到底是什么编译器**。

现在苹果公司使用的编译器是 LLVM，相比于 Xcode 5 版本前使用的 GCC，编译速度提高了 3 倍。同时，苹果公司也反过来主导了 LLVM 的发展，让 LLVM 可以针对苹果公司的硬件进行更多的优化。

总结来说，LLVM 是编译器工具链技术的一个集合。而其中的 lld 项目，就是内置链接器。编译器会对每个文件进行编译，生成 Mach-O（可执行文件）；链接器会将项目中的多个 Mach-O 文件合并成一个。

LLVM 的编译过程非常复杂。如果你有兴趣的话，可以通过[官方手册](#)查看完整的编译过程。

这里，我先简单为你总结下编译的几个主要过程：

首先，你写好代码后，LLVM 会预处理你的代码，比如把宏嵌入到对应的位置。

预处理完后，LLVM 会对代码进行词法分析和语法分析，生成 AST。AST 是抽象语法树，结构上比代码更精简，遍历起来更快，所以使用 AST 能够更快速地进行静态检查，同时还能更快地生成 IR（中间表示）。

最后 AST 会生成 IR，IR 是一种更接近机器码的语言，区别在于和平台无关，通过 IR 可以生成多份适合不同平台的机器码。对于 iOS 系统，IR 生成的可执行文件就是 Mach-O。

下图展示了编译的主要过程。

# Source code

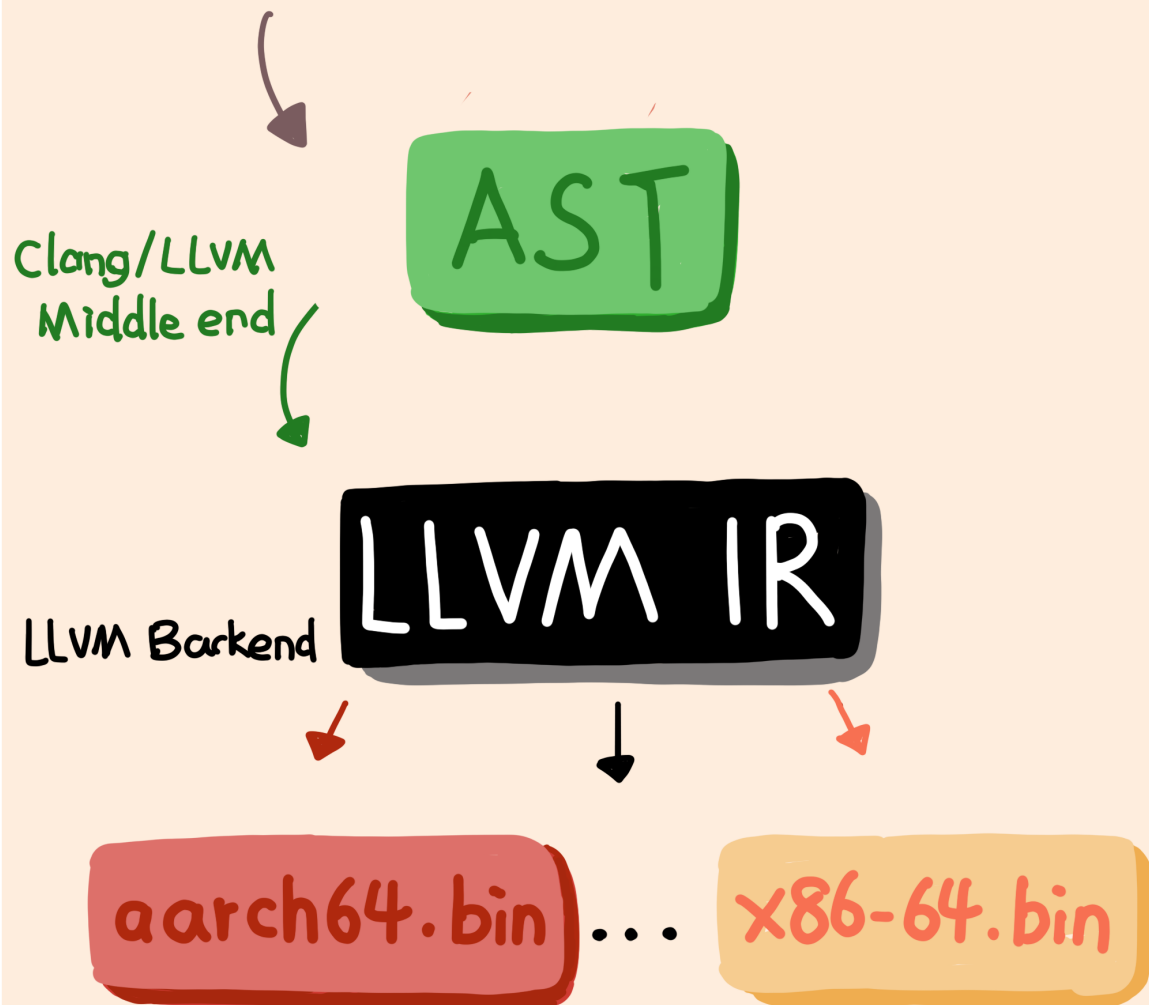


图 2 编译的主要过程

## 编译时链接器做了什么？

Mach-O 文件里面的内容，主要就是代码和数据：代码是函数的定义；数据是全局变量的定义，包括全局变量的初始值。不管是代码还是数据，它们的实例都需要由符号将其关联起来。

为什么呢？因为 Mach-O 文件里的那些代码，比如 if、for、while 生成的机器指令序列，要操作的数据会存储在某个地方，变量符号就需要绑定到数据的存储地址。你写的代码还会引用其他的代码，引用的函数符号也需要绑定到该函数的地址上。

而链接器的作用，就是完成变量、函数符号和其地址绑定这样的任务。而这里我们所说的符号，就可以理解为变量名和函数名。

## 那为什么要让链接器做符号和地址绑定这样一件事儿呢？不绑定的话，又会有什么问题？

如果地址和符号不做绑定的话，要让机器知道你在操作什么内存地址，你就需要在写代码时给每个指令设好内存地址。写这样的代码的过程，就像你直接在和不同平台的机器沟通，连编译生成 AST 和 IR 的步骤都省掉了，甚至优化平台相关的代码都需要你自己编写。

这件事儿看起来挺酷，但可读性和可维护性都会很差，比如修改代码后对地址的维护就会让你崩溃。而这种“崩溃”的罪魁祸首就是代码和内存地址绑定得太早。

另外，绑定得太早除了可读性和可维护性差之外，还会有更多的重复工作。因为，你需要针对不同的平台写多份代码，而这些代码本可以通过高级语言一次编译成多份。既然这样，那我们应该怎么办呢？

我们首先想到的就是，用汇编语言来让这种绑定滞后。随着编程语言的进化，我们很快就发现，采用任何一种高级编程语言，都可以解决代码和内存绑定过早产生的问题，同时还能扫掉使用汇编编写程序的烦恼。

现在，我们已经通过反证法，理解了在一个文件里把符号和地址绑定在一起的必要性。接下来，我们再看看**链接器为什么还要把项目中的多个 Mach-O 文件合并成一个**。

其实，这个问题也好回答。

你肯定不希望一个项目是在一个文件里从头写到尾的吧。项目中文件之间的变量和接口函数都是相互依赖的，所以这时我们就需要通过链接器将项目中生成的多个 Mach-O 文件的符号和地址绑定起来。

没有这个绑定过程的话，单个文件生成的 Mach-O 文件是无法正常运行起来的。因为，如果运行时碰到调用在其他文件中实现的函数的情况时，就会找不到这个调用函数的地址，从而无法继续执行。

链接器在链接多个目标文件的过程中，会创建一个符号表，用于记录所有已定义的和所有未定义的符号。链接时如果出现相同符号的情况，就会出现“ld: duplicate symbols”的错误信息；如果在其他目标文件里没有找到符号，就会提示“Undefined symbols”的错误信息。

说完了链接器解决的问题，我们再一起来看看**链接器对代码主要做了哪几件事儿**。

去项目文件里查找目标代码文件里没有定义的变量。

扫描项目中的不同文件，将所有符号定义和引用地址收集起来，并放到全局符号表中。

计算合并后长度及位置，生成同类型的段进行合并，建立绑定。

对项目不同文件里的变量进行地址重定位。

你在项目里为某项需求写了一些功能函数，但随着业务的发展，一些功能被下掉了或者被其他负责的同事在另一个文件里用其他函数更新了功能。那么这时，你以前写的那些函数就没有用武之地了。日长月久，无用的函数越来越多，生成的 Mach-O 文件也就越来越大。

这时，链接器在整理函数的符号调用关系时，就可以帮你理清有哪些函数是没被调用的，并自动去除掉。那这是怎么实现的呢？

链接器在整理函数的调用关系时，会以 main 函数为源头，跟随每个引用，并将其标记为 live。跟随完成后，那些未被标记 live 的函数，就是无用函数。然后，链接器可以通过打开 Dead code stripping 开关，来开启自动去除无用代码的功能。并且，这个开关是默认开启的。

说完了编译时链接器的基本功能，接下来我们再说一说动态库链接，这也是链接器的一大作用。

## 动态库链接

在真实的 iOS 开发中，你会发现很多功能都是现成可用的，不光你能够用，其他 App 也在用，比如 GUI 框架、I/O、网络等。链接这些共享库到你的 Mach-O 文件，也是通过链接器来完成的。

链接的共用库分为静态库和动态库：静态库是编译时链接的库，需要链接进你的 Mach-O 文件里，如果需要更新就要重新编译一次，无法动态加载和更新；而动态库是运行时链接的库，使用 dyld 就可以实现动态加载。

Mach-O 文件是编译后的产物，而动态库在运行时才会被链接，并没参与 Mach-O 文件的编译和链接，所以 Mach-O 文件中并没有包含动态库里的符号定义。也就是说，这些符号会显示为“未定义”，但它们的名字和对应的库的路径会被记录下来。运行时通过 dlopen 和 dlsym 导入动态库时，先根据记录的库路径找到对应的库，再通过记录的名字符号找到绑定的地址。

dlopen 会把共享库载入运行进程的地址空间，载入的共享库也会有未定义的符号，这样会触发更多的共享库被载入。dlopen 也可以选择是立刻解析所有引用还是滞后去做。dlopen 打开动态库后返回的是引用的指针，dlsym 的作用就是通过 dlopen 返回的动态库指针和函数符号，得到函数的地址然后使用。

**使用 dyld 加载动态库，有两种方式：**有程序启动加载时绑定和符号第一次被用到时绑定。为了减少启动时间，大部分动态库使用的都是符号第一次被用到时再绑定的方式。


加载过程开始会修正地址偏移，iOS 会用 ASLR 来做地址偏移避免攻击，确定 Non-Lazy Pointer 地址进行符号地址绑定，加载所有类，最后执行 load 方法和 Clang Attribute 的 constructor 修饰函数。

每个函数、全局变量和类都是通过符号的形式定义和使用的，当把目标文件链接成一个 Mach-O 文件时，链接器在目标文件和动态库之间对符号做解析处理。

下面，我们就通过一个例子来看看 dyld 的链接过程。


**第一步：**先编写多个文件。

## Boy.h

 复制代码


```
1 c
2 #import <Foundation/Foundation.h>
3 @interface Boy : NSObject
4 - (void)say;
5 @end
6
```

## Boy.m

 复制代码


```
1 c
2 #import "Boy.h"
3 @implementation Boy
4 - (void)say
5 {
6     NSLog(@"hi there again!\n");
7 }
8 @end
9
```

## SayHi.m

 复制代码

```
1 c
2 #import "Boy.h"
3 int main(int argc, char *argv[])
4 {
5     @autoreleasepool {
6         Boy *boy = [[Boy alloc] init];
7         [boy say];
8         return 0;
9     }
10 }
11
```


**第二步：**编译多个文件。

 复制代码

```
1 xcrun clang -c Boy.m
2 xcrun clang -c SayHi.m
3
```

**第三步：**将编译后的文件链接起来，这样就可以生成 a.out 可执行文件了。

备注：a.out 是编译器的默认名字。

 复制代码

```
1 xcrun clang SayHi.o Boy.o -Wl,`xcrun -show-sdk-path`/System/Library/Frameworks/Foundation.framework/
2
```

符号表会规定它们的符号，你可以使用 nm 工具查看。

我们先用 nm 工具看一下 SayHi.o 文件：

 复制代码


```
1 xcrun nm -nm SayHi.o
2
3
4          (undefined) external _OBJC_CLASS_$_Boy
5          (undefined) external _objc_autoreleasePoolPop
6          (undefined) external _objc_autoreleasePoolPush
7          (undefined) external _objc_msgSend
8 0000000000000000 (__TEXT,__text) external _main
9
```

\_OBJC\_CLASS\_\$\_Boy，表示 Boy 的 OC 符号。

(undefined) external，表示未实现非私有。如果是私有的话，就是 non-external。

external \_main，表示 main() 函数，处理 0 地址，记录在 \_\_TEXT,\_\_text 区域里。

接下来，我们再看看 Boy.o 文件：


 复制代码

```
1 xcrun nm -nm Boy.o
2
3
4          (undefined) external _NSLog
5          (undefined) external _OBJC_CLASS_$_NSObject
6          (undefined) external _OBJC_METAClass_$_NSObject
7          (undefined) external ____CFCConstantStringClassReference
8          (undefined) external __objc_empty_cache
9 0000000000000000 (__TEXT,__text) non-external -[Boy say]
10 0000000000000060 (__DATA,__objc_const) non-external l_OBJC_METAClass_ro_$_Boy
11 00000000000000a8 (__DATA,__objc_const) non-external l_OBJC_$_INSTANCE_METHODS_Boy
12 00000000000000c8 (__DATA,__objc_const) non-external l_OBJC_CLASS_ro_$_Boy
13 0000000000000110 (__DATA,__objc_data) external _OBJC_METAClass_$_Boy
14 0000000000000138 (__DATA,__objc_data) external _OBJC_CLASS_$_Boy
15
```

因为 undefined 符号表示的是该文件类未定义，所以在目标文件和 Foundation framework 动态库做链接处理时，链接器会尝试解析所有的 undefined 符号。




链接器通过动态库解析成符号会记录是通过哪个动态库解析的，路径也会一起记录下来。你可以再用 nm 工具查看 a.out 符号表，对比 boy.o 的符号表，看看链接器是怎么解析符号的。

 复制代码

```
1 xcrun nm -nm a.out
2
3
4          (undefined) external _NSLog (from Foundation)
5          (undefined) external _OBJC_CLASS_$_NSObject (from CoreFoundation)
6          (undefined) external _OBJC_METACLASS_$_NSObject (from CoreFoundation)
7          (undefined) external ____CFCConstantStringClassReference (from CoreFounda
8          (undefined) external __objc_empty_cache (from libobjc)
9          (undefined) external _objc_autoreleasePoolPop (from libobjc)
10         (undefined) external _objc_autoreleasePoolPush (from libobjc)
11         (undefined) external _objc_msgSend (from libobjc)
12         (undefined) external dyld_stub_binder (from libSystem)
13 0000000010000000 (__TEXT,__text) [referenced dynamically] external __mh_execute_header
14 00000000100000e90 (__TEXT,__text) external _main
15 00000000100000f10 (__TEXT,__text) non-external -[Boy say]
16 00000000100001130 (__DATA,__objc_data) external _OBJC_METACLASS_$_Boy
17 00000000100001158 (__DATA,__objc_data) external _OBJC_CLASS_$_Boy
18
```

进行对比的时候，我们可以重点关注哪些 undefined 的符号，有了更多信息，就可以知道在哪个动态库能够找到它。

我们可以通过 otool 工具来找到符号所需库在哪儿。


 复制代码

```
1 xcrun otool -L a.out
2
3 a.out:
4   /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation (compatibility \
5   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1238.0.0)
6   /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation (compati
7   /usr/lib/libobjc.A.dylib (compatibility version 1.0.0, current version 228.0.0)
8
```

从 otool 工具输出的结果可以看到，这些 undefined 符号需要的两个库分别是 libSystem 和 libobjc。查看 libSystem 库的话，你可以看到常用的 GCD 的 libdispatch，还有 Block 的 libsystem\_blocks。

dylib 这种格式，表示是动态链接的，编译的时候不会被编译到执行文件中，在程序执行的时候才 link，这样就不用算到包大小里，而且不更新执行程序就能够更新库。

我们可以打印看看什么库被加载了：

 复制代码

```
1 (export DYLD_PRINT_LIBRARIES=; ./a.out )
2
```

```
4 dyld: loaded: /Users/didi/Downloads/./a.out
5 dyld: loaded: /System/Library/Frameworks/Foundation.framework/Versions/C/Foundation
6 dyld: loaded: /usr/lib/libSystem.B.dylib
7 dyld: loaded: /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundat
8 ...
9
```

数一下，被加载的库还挺多的。

因为 Foundation 还会依赖一些其他动态库，这些依赖的其他库还会再依赖更多的库，所以相互依赖的符号会很多，需要处理的时间也会比较长。

这里系统上的动态链接器会使用共享缓存，共享缓存在 /var/db/dyld/。当加载 Mach-O 文件时，动态链接器会先检查是否有共享缓存。每个进程都会在自己的地址空间映射这些共享缓存，这样做可以起到优化 App 启动速度的作用。

而关于动态链接器的作用顺序是怎样的，你可以先看看 Mike Ash 写的这篇关于 dyld 的博客：[Dynamic Linking On OS X](#)。这篇博客里面，很详细地讲解了 dyld 所做的事情。


简单来说，dyld 做了这么几件事儿：

先执行 Mach-O 文件，根据 Mach-O 文件里 undefined 的符号加载对应的动态库，系统会设置一个共享缓存来解决加载的递归依赖问题；

加载后，将 undefined 的符号绑定到动态库里对应的地址上；

最后再处理 +load 方法，main 函数返回后运行 static terminator。

调用 +load 方法是通过 runtime 库处理的。你可以通过一个[可编译的开源 runtime 库](#)来了解 runtime，从源码层面去看程序启动时 runtime 做了哪些事情。在 debug-objc 下创建一个类，在 +load 方法里断点查看走到这里调用的堆栈如下：

 复制代码

```
1 0  +[someclass load]
2 1  call_class_loads()
3 2  ::call_load_methods
4 3  ::load_images(const char *path __unused, const struct mach_header *mh)
5 4  dyld::notifySingle(dyld_image_states, ImageLoader const*, ImageLoader::InitializerTir
6 11 _dyld_start
7
```

在 load\_images 方法里断点 p path 可以打印出所有加载的动态链接库，这个方法 hasLoadMethods 用于快速判断是否有 +load 方法。

prepare\_load\_methods 这个方法会获取所有类的列表然后收集其中的 +load 方法，在代码里可以发现 Class 的 +load 是先执行的，然后执行 Category。

为什么这样做呢？我们通过 `prepare_load_methods` 这个方法可以看出，在遍历 Class 的 `+load` 方法时会执行 `schedule_class_load` 方法，这个方法会递归到根节点来满足 Class 收集完整关系树的需求。

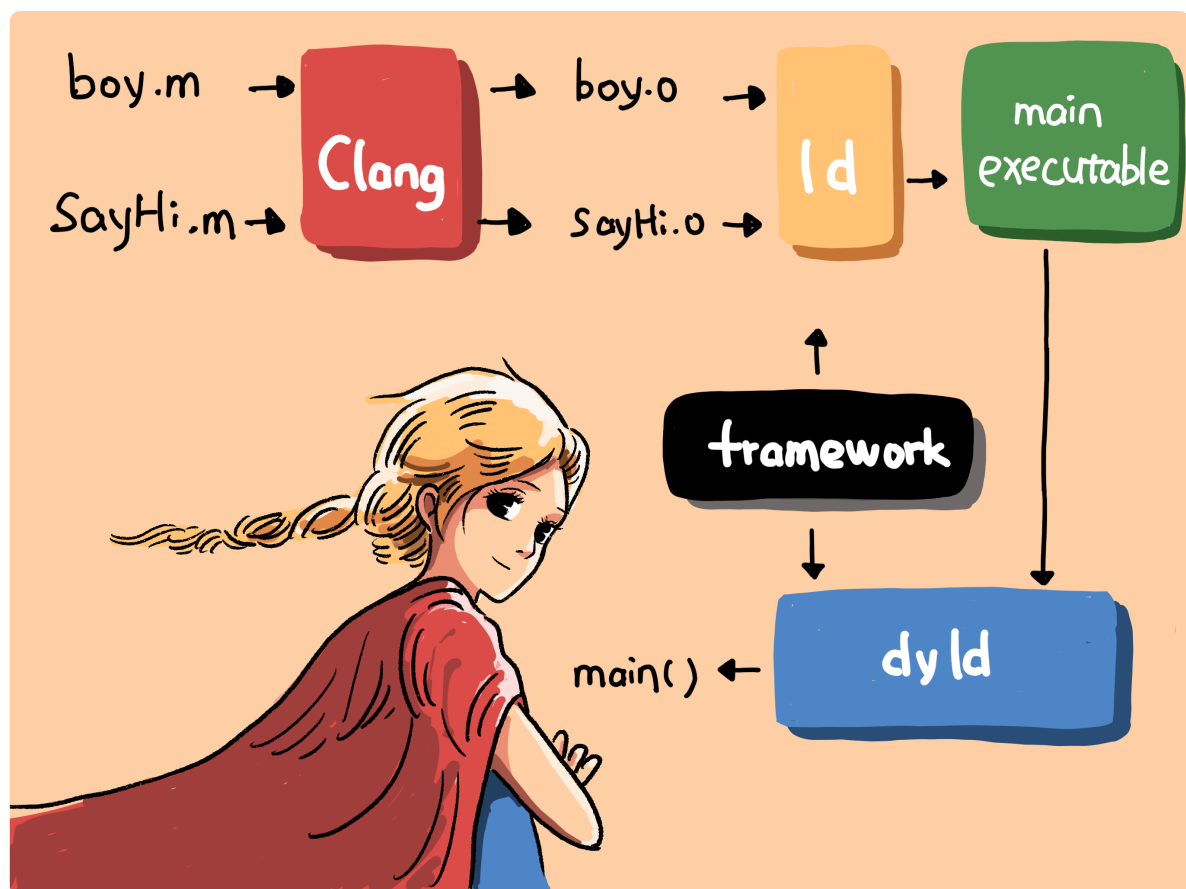
最后，`call_load_methods` 会创建一个 `autoreleasePool` 使用函数指针来动态调用类和 Category 的 `+load` 方法。

如果你了解 Cocoa 的 Foundation 库的话，可以通过 GNUStep 源码来学习。比如，`NSNotificationCenter` 发送通知是按什么顺序发送的，你可以查看 `NSNotificationCenter.m` 里的 `addObserver` 方法和 `postNotification` 方法，看看观察者是怎么添加的，以及是怎么被遍历通知到的。

最后说一句，dyld 是开源的，地址是：<https://github.com/opensource-apple/dyld>

## 小结

今天这篇文章，我与你介绍了链接器是什么，为什么需要链接器，以及链接器在编译时和程序启动时会做的事情。总体来看，从编译、链接、执行、动态库加载到 `main` 函数开始执行的过程如下图所示。



编译阶段由于有了链接器，你的代码可以写在不同的文件里，每个文件都能够独立编成 Mach-O 文件进行标记。编译器可以根据你修改的文件范围来减少编译，通过这种方式提高每次编译的速度。

了解了这种链接机制，你也能够明白，文件越多，链接器链接 Mach-O 文件所需绑定的遍历操作就会越多，编译速度也会越慢。

了解程序运行阶段的动态库链接原理，会让你更多地了解程序在启动时做的事情，同时还能够对你有一些启发。

比如，在开发调试阶段，是不是代码改完以后可以先不去链接项目里的所有文件，只编译当前修改的文件动态库，通过运行时加载动态库及时更新，看到修改的结果。这样调试的速度，不就能够得到质的提升了么。而具体怎么做，我会在第 6 篇文章 “App 如何通过注入动态库的方式实现极速编译调试？” 中和你详细说明。

再比如，你可以逆向找出其他 App 里你感兴趣功能的使用方法，然后在自己的程序里直接调用它，最后将那个 App 作为动态库加载到自己的 App 里。这样，你感兴趣的这个功能，就能够在自己的程序里起作用了。

其实，使用链接器不仅能提高开发效率，还可以让你发挥想象力再去做些其他有趣的事情。

## 课后小作业

请你写一段代码，在 App 运行时通过 `dlopen` 和 `dlsym` 链接加载 bundle 里的动态库。

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

---

© 版权归极客邦科技所有，未经许可不得转载



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

## 精选留言(8)



springday

老师，iOS目前支持AOT么。另外，感觉iOS目前编辑时间过长，是不是应该采用flutter的预编辑机制呢。

👍 1    2019-03-21



난너하나만

再比如，你可以逆向找出其他 App 里你感兴趣功能的使用方法，然后在自己的程序里直接调用它，最后将那个 App 作为动态库加载到自己的 App 里。这样，你感兴趣的这个功能，就能够在你自己的程序里起作用了。

这需要越狱机吧？

👍 1    2019-03-21

---



langzuxiaozi

dlopen dlsym 审核会被拒，是有办法解决这个问题吗？

👍 1    2019-03-21

---



Chouee

基础不扎实，看得有点吃力。

👍    2019-03-21

---



Trust me

dyld3可以预绑定系统库的 符号 加快启动时间

👍    2019-03-21

---



zeroskylian

在 App 运行时通过 dlopen 和 dlsym 链接加载 bundle 里的动态库。

这个是不是只能用于调试，记得iOS不允许在沙盒中加载动态库

👍    2019-03-21

---



绝影

太棒了，有料。深入浅出，文末还有总结，支持这样文章结构！

👍    2019-03-21

---



Gargit

睡前大概浏览一下，明天再细读里头不懂的概念

👍    2019-03-21