浅谈iOS编译过程

jianshu.com/p/5b2cce762106

引言

维基百科：
编译语言（英语：Compiled language）是一种以编译器来实现的编程语言。它不像解释型语言一样，由解释器将代码一句一句运行，而是以编译器，先将代码编译为机器码，再加以运行。理论上，任何编程语言都可以是编译式，或直译式的。它们之间的区别，仅与程序的应用有关。

> 一般可以将编程语言分为两种，编译语言和直译式语言。

前言

iOS开发使用Object-C和Swift编译语言，两者都需要通过编译器（Clang + LLVM）把代码编译器生成机器码，机器码可以直接在CPU上执行。接下来详细介绍一下这两种语言的优缺点，可以更深入的理解为什么移动端开发会采用编译语言。

编译语言和直译式语言两种编程语言的优缺点：

编译语言/直译式语言优缺点比较：

编译语言	运行速度快（被预先编译成机器码，可以直接运行）	开发、调试比较长（程序开发速度，以及除错时间时间较长）
直译式语言	开发、调试比较短（程序开发速度，以及除错时间时间较长。应用程序不能脱离其解释器，但这种方式比较灵活，可以动态地调整、修改应用程序）	运行速度慢（源代码一边由相应语言的解释器翻译"成目标代码（机器语言），一边执行）

编译语言： 像C++、Objective-C（Swift）、C、Java等都是编译语言，必须通过编译器生成机器码，机器码可以直接在CPU上执行，所以你执行效率较高。而每次运行都需要编译器生成机器码再运行，故编写、调试、排错比较繁琐。

直译式语言：像JavaScript、Python、PHP等都是直译式语言，不需要经过编译的过程，而是在执行的时候通过一个中间的解释器将代码解释为CPU可以执行的代码。所以，较编译语言来说，直译式语言效率低一些，但是编写比较灵活。

> 总之，由于移动端（iOS、Android）设备性能限制的情况下，采用编译语言进行开发是一种比较好的方式

一、iOS编译器

> 在讲解编译过程之前，需要了解苹果公司采用的编译器，编译器采用哪种方式进行编译，才能更深入的理解编译底层的整个过程。

1.Xcode编译器发展史

> Xcode3 以前： GCC；
> Xcode3： 增加LLVM，GCC(前端) + LLVM(后端)；
> Xcode4.2： 出现Clang - LLVM 3.0成为默认编译器；
> Xcode4.6： LLVM 升级到4.2版本；
> Xcode5： GCC被废弃，新的编译器是LLVM 5.0，从GCC过渡到Clang-LLVM的时代正式完成

- **1.1.为什么苹果的Xcode会使用Clang+LLVM取代GCC？**

  GCC最初是作为CNU(**GNU是"GNU is Not Unix"**)操作系统的编译器编写的，是一套由 GNU 开发的编程语言编译器，不属于苹果维护也不能完全控制开发进程，Apple为Objective-C增加许多新特性，但是GCC开发者对这些支持却不友好；Apple需要做模块化，GCC开发者却拖着迟迟不实现。
- **1.2.GCC被取代的历史必然**

  随着Apple对其IDE(也就是Xcode)性能的要求越来越高,苹果公司需要找到一个可以控制的编译器。而在在科技的历史长河中，LLVM项目于2000年在伊利诺伊大学厄巴纳 - 香槟分校开始，由Vikram Adve和Chris Lattner领导。 LLVM最初是作为研究基础设施开发的，用于研究静态和动态编程语言的动态编译技术。 LLVM是根据伊利诺伊大学/ NCSA开源许可证发布的，是一个许可的免费软件许可证。
- **1.3.GCC被取代**

  2005年，Apple Inc.聘请了Lattner并组建了一个团队，致力于LLVM系统，以便在Apple的开发系统中实现各种用途。 LLVM是Apple最新的macOS和iOS开发工具中不可或缺的一部分。

2.为什么选择LLVM编译器

2.1编译器

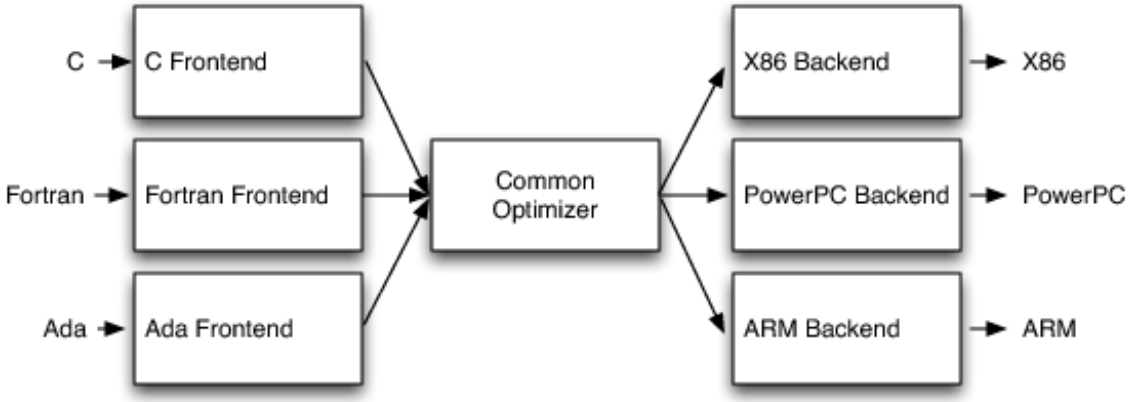
2.1.1经典编译器设计简介

传统静态编译器（如大多数C编译器）最流行的设计是三阶段设计，其主要组件是前端，优化器和后端。 前端解析源代码，检查它是否有错误，并构建一个特定于语言的抽象语法树（AST）来表示输入代码。 AST可选地转换为新的表示以进行优化，优化器和后端在代码上运行。

1111.png

2.1.2GCC编译器设计简介（同样采用三相设计）

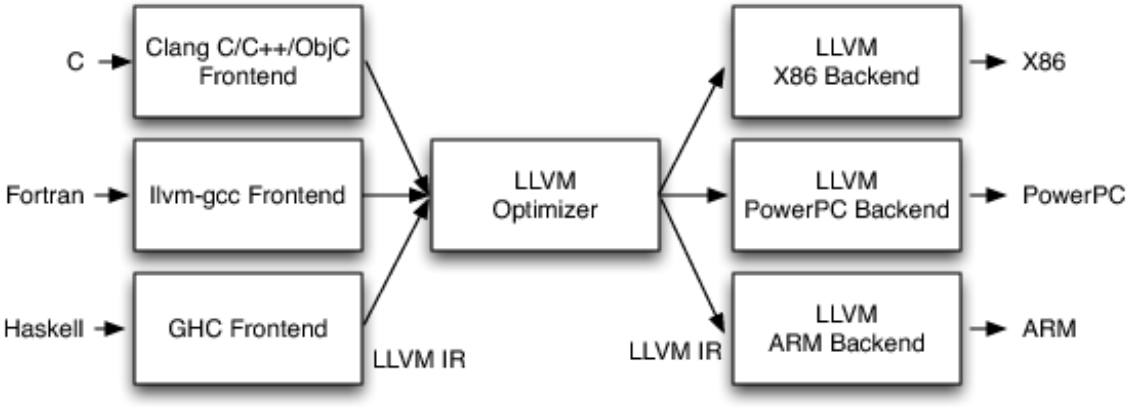
GCC编译器的三相设计中最成功的实现是对Java、 .Net语言的编译解析。



222.png

2.1.3 LLVM编译器设计简介（实现三相设计）

在基于LLVM的编译器中，前端负责解析，验证和诊断输入代码中的错误，然后将解析的代码转换为LLVM IR（通常情况。但是也有例外，通过构建AST然后将AST转换为LLVM IR）。该IR可选地通过一系列改进代码的分析和优化过程提供，然后被发送到代码生成器以生成本机机器代码，如图下图所示。



3333.png

> 为什么要使用三相设计？优势在哪？

> 首先解决了一个很大的问题：假如有N种语言（C、OC、C++、Swift...）的前端，同时也有M个架构（模拟器、arm64、x86...）的Target，是否就需要 N × M 个编译器？
> 三相架构的价值就体现出来了，通过共享优化器的中转，很好的解决了这个问题。
> 假如你需要增加一种语言，只需要增加一种前端；假如你需要增加一种处理器架构，也只需要增加一种后端，而其他的地方都不需要改动。这复用思想很牛逼吧。

2.2 LLVM编译器的组成

LLVM项目是模块化和可重用的编译器和工具链技术的集合。LLVM主要的子项目有一下几个：

> **1.LLVM核心库:**
> LLVM提供一个独立的链接代码优化器为许多流行CPU（以及一些不太常见的CPU）的代码生成支持。这些库是围绕一个指定良好的代码表示构建的，称为LLVM中间表示（“LLVM IR”）。LLVM还可以充当JIT编译器 - 它支持x86 / x86_64和PPC / PPC64程序集生成，并具有针对编译速度的快速代码优化。。
> **2.LLVM IR 生成器Clang:**
> Clang是一个“LLVM原生”C / C ++ / Objective-C编译器，旨在提供惊人的快速编译（例如，在调试配置中编译Objective-C代码时比GCC快3倍），非常有用的错误和警告消息以及提供构建优秀源代码工具的平台。
> **3.LLDB项目:**
> LLDB项目以LLVM和Clang提供的库为基础，提供了一个出色的本机调试器。它使用Clang AST和表达式解析器，LLVM JIT，LLVM反汇编程序等，以便提供“正常工作”的体验。在加载符号时，它也比GDB快速且内存效率更高。
> **4.libc ++和libc++:**
> libc ++和libc++ ABI项目提供了C ++标准库的标准符合性和高性能实现，包括对C ++ 11的完全支持。
> **5.lld项目:**
> lld项目旨在成为clang / llvm的内置链接器。目前，clang必须调用系统链接器来生成可执行文件。
>
> ***其他的就不再详细介绍了,详情可以参考(LLVM和Clang)***

总之，LLVM是Apple主导的开源框架，并提供一套使用于Apple平台的LLVM编译器，同时提供优秀的性能，所以Apple采用LLVM的方式进行编译

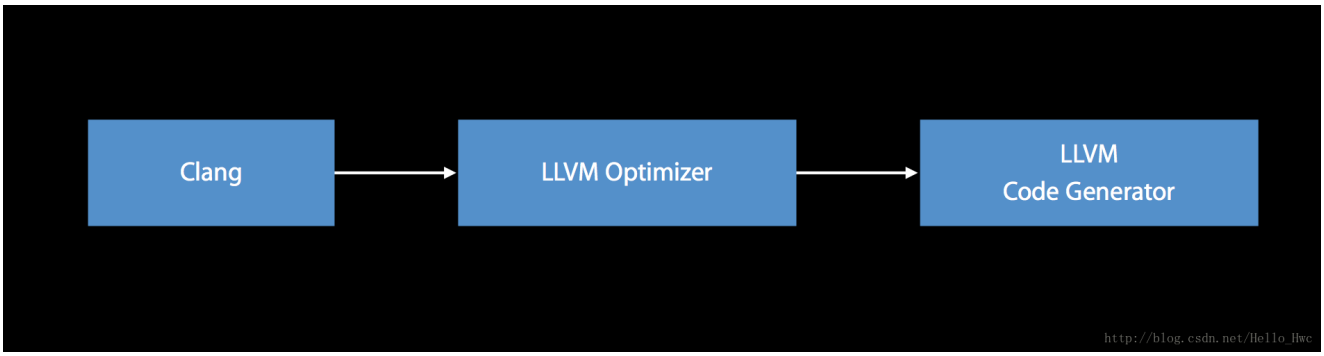
3. Clang + LLVM 的编译简单过程

3.1.Clang + LLVM 编译过程

LLVM采用三相设计，前端Clang负责解析，验证和诊断输入代码中的错误，然后将解析的代码转换为LLVM IR，后端LLVM编译把IR通过一系列改进代码的分析和优化过程提供，然后被发送到代

码生成器以生成本机机器代码。

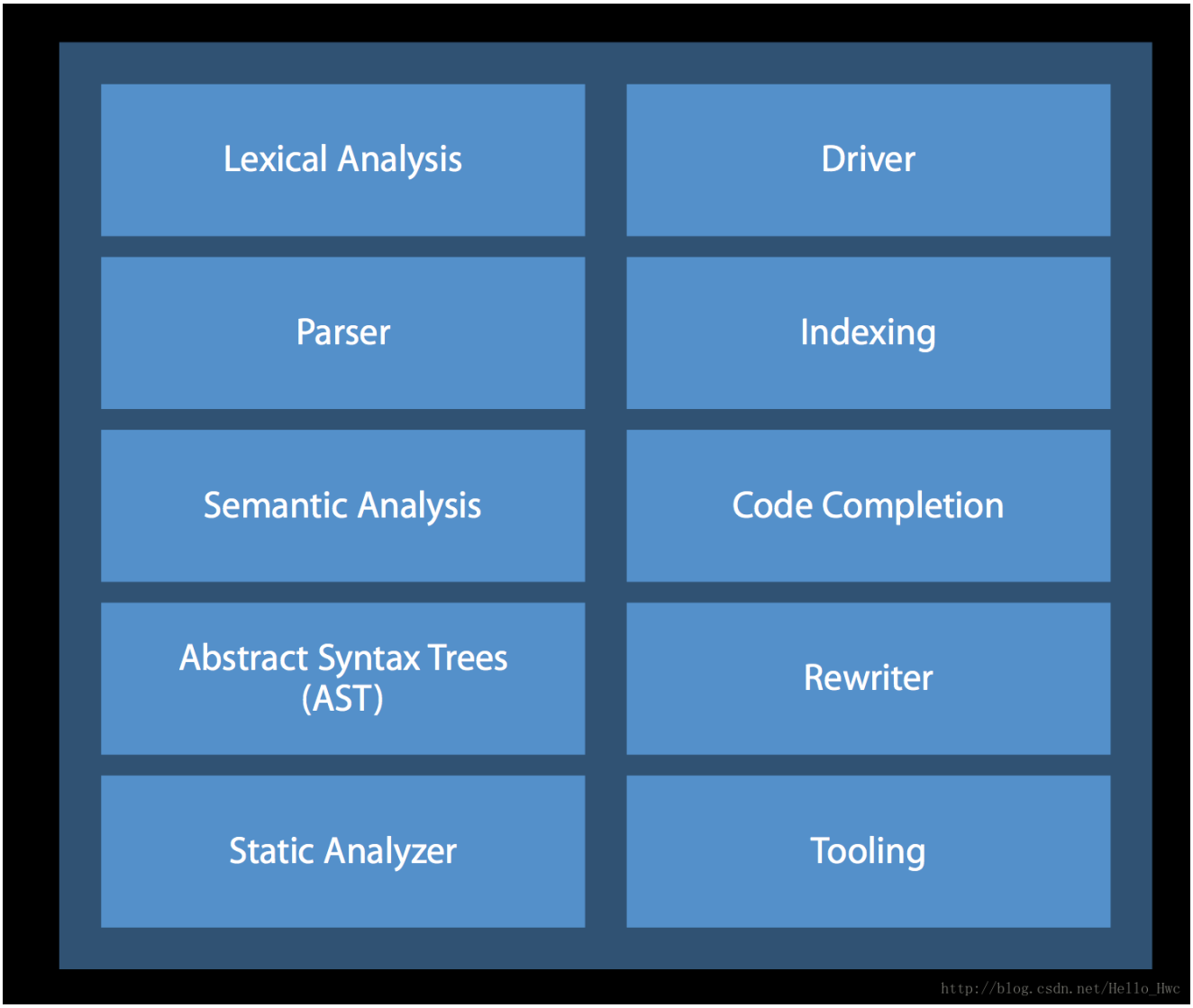
> 简单的流程如下图：



444.png

3.2 Clang **编译前端**

> 编译器前端的任务是进行：语法分析，语义分析，生成中间代码(intermediate representation)。在这个过程中，会进行类型检查，如果发现错误或者警告会标注出来在哪一行。

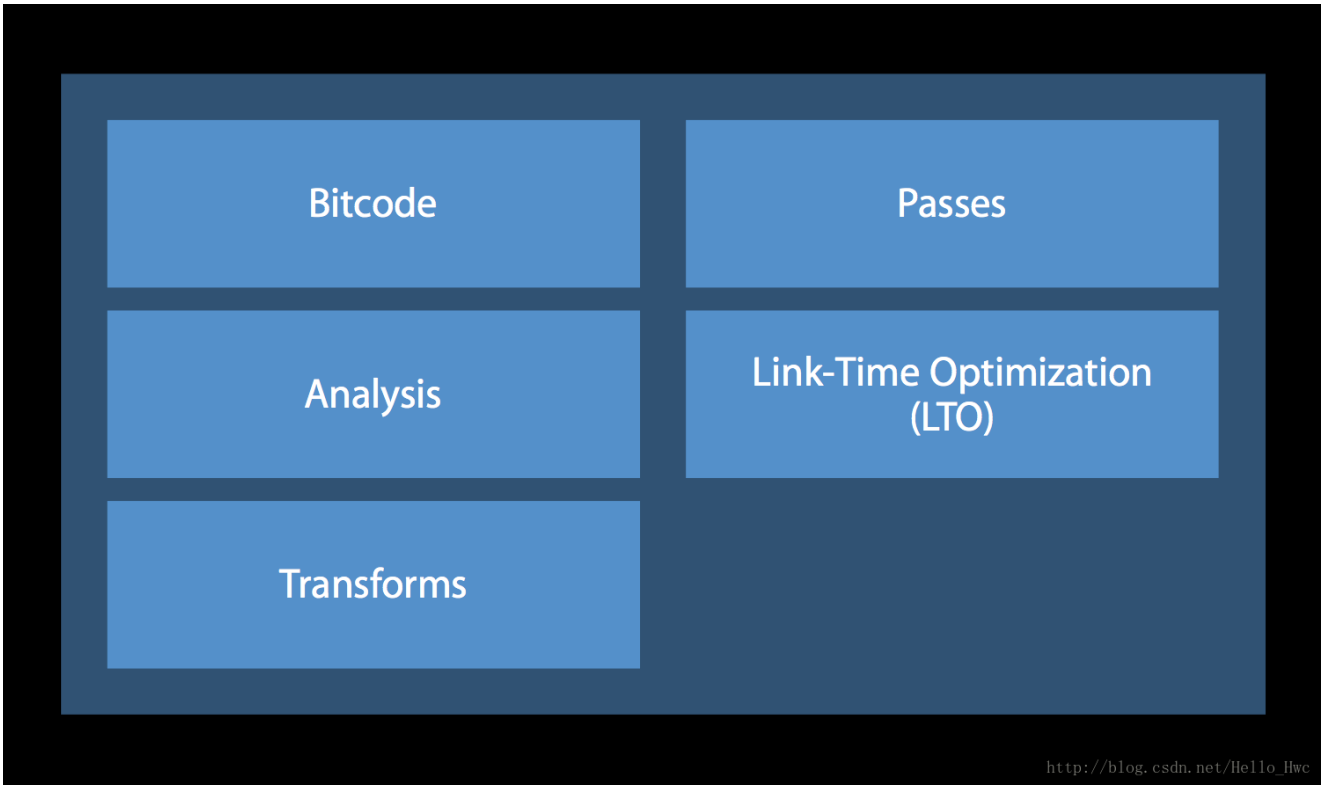


555.png

3.3 LLVM 编译后端

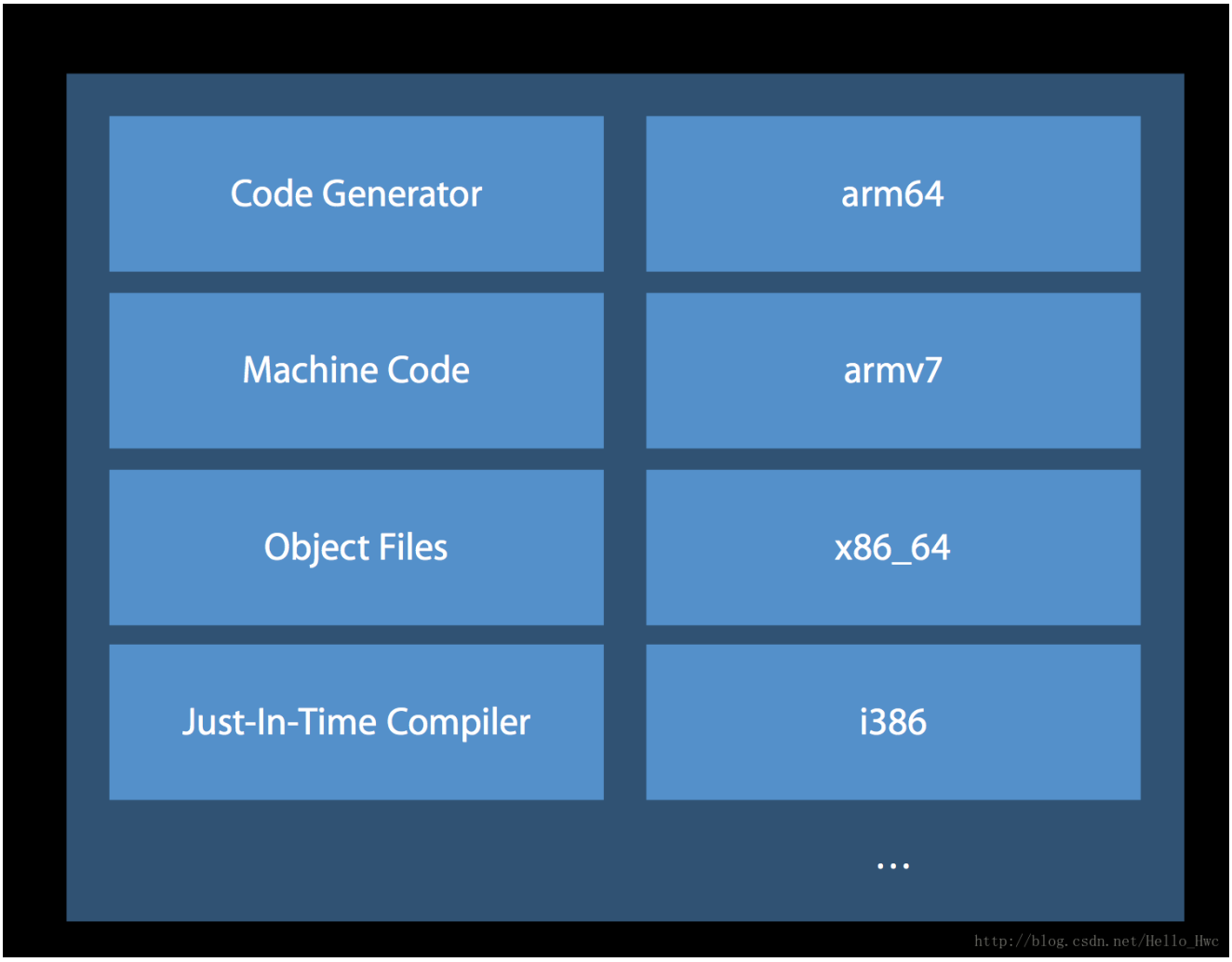
编译器后端会进行机器无关的代码优化，生成机器语言，并且进行机器相关的代码优化。iOS的编译过程，后端的处理如下：

> LVVM优化器会进行BitCode的生成，链接期优化等等。



666.png

> 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。



777.png

二、LLVM 编译详细过程原理

1.简述LLVM的使用场景

通过上方的讲解，可能对LLVM有一个简单的了解，但是感觉LLVM是底层编译器的内容，和开发没有太大的关系，在使用过程中基本上用不到，感觉遥不可及，其实LLVM在项目的使用过程中一直都在使用，只是没有发现。

Clang是LLVM的一个前端，在Xcode编译iOS项目的时候，都是使用的LLVM，其实在编写代码以及调试的时候都在接触LLVM提供的功能，例如：代码的亮度（Clang）、实时代码检查（Clang）、代码提示（Clang）、debug断点调试（LLDB）。

2.项目编译过程简介

下面来简单的讲讲整个 iOS 项目的编译过程,其中可能会有一些疑问，先保留着，后面会详细解释

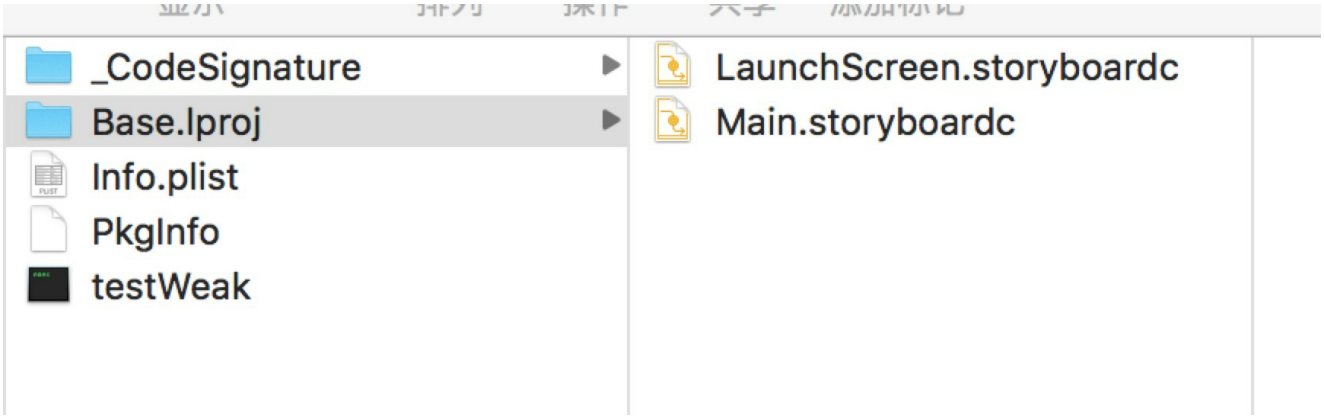
> 我们的项目是一个 target，一个编译目标，它拥有自己的文件和编译规则，在我们的项目中可以存在多个子项目，这在编译的时候就导致了使用了 Cocoapods 或者拥有多个 target 的项目会先编译依赖库。这些库都和我们的项目编译流程一致。Cocoapods 的原理解释将在文章后面一部分进行解释。

> ### iOS 项目的编译过程

1.写入辅助文件：将项目的文件结构对应表、将要执行的脚本、项目依赖库的文件结构对应表写成文件，方便后面使用；并且创建一个 .app 包，后面编译后的文件都会被放入包中；
2.运行预设脚本：Cocoapods 会预设一些脚本，当然你也可以自己预设一些脚本来运行。这些脚本都在 Build Phases 中可以看到；
3.编译文件：针对每一个文件进行编译，生成可执行文件 Mach-O，这过程 LLVM 的完整流程，前端、优化器、后端；
4.链接文件：将项目中的多个可执行文件合并成一个文件；
5.拷贝资源文件：将项目中的资源文件拷贝到目标包；
6.编译 storyboard 文件：storyboard 文件也是会被编译的；
7.链接 storyboard 文件：将编译后的 storyboard 文件链接成一个文件；
8.编译 Asset 文件：我们的图片如果使用 Assets.xcassets 来管理图片，那么这些图片将会被编译成机器码，除了 icon 和 launchImage；
9.运行 Cocoapods 脚本：将在编译项目之前已经编译好的依赖库和相关资源拷贝到包中。
10.生成 .app 包
11.将 Swift 标准库拷贝到包中
12.对包进行签名
13.完成打包

简单的Demo的Build过程图和App 内部的结构

888.jpeg



999.jpeg

在上述流程中：2 - 9 步骤的数量和顺序并不固定，这个过程可以在 Build Phases 中指定。 Phases：阶段、步骤 **。这个 Tab 的意思就是编译步骤。其实不仅我们的整个编译步骤和顺序可以被设定，包括编译过程中的编译规则（Build Rules）和具体步骤的参数（Build Settings），在对应的 Tab 都可以看到。关于整个编译流程的日志和设定，可以查看这篇文章：Build 过程，跟着它的步骤来查看自己的项目将有助于你理解整个编译流程。后面也会详细讲解这些内容。**

3.文件编译过程

3.1. 预处理

预处理顾名思义是预先处理,那预处理都做了哪些事情呢？内容如下。

- (1) import 头文件替换
  代码中会有很多 #import 宏，预处理的第一步就是将 import 引入的文件代码放入对应文件。
- (2) macro 宏展开
  带参数宏和不带参数宏
- (3)**处理其他的预编译指令（其实预编译过程也是出了预编译指令的过程）**

> 条件编译语句也是在预处理阶段完成，并且条件编译只允许编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销并提高了程序的效率,如以下代码就只会保留一个return语句：

```
#if DEBUG
    return YES;
#else
    return NO;
#endif
```

（4）总之

简单来说，“#”这个符号是编译器预处理的标志

```
$clang -E main.m
```

```
#import <Foundation/Foundation.h>
#define aa 10
int main(){
    NSObject *obj = [[NSObject alloc] init];
    id __weak obj1 = obj;
    NSLog(@"------%@--%d--",[obj1 class],aa);
}
```

```
int main(){
    NSObject *obj = [[NSObject alloc] init];
    id __attribute__((objc_ownership(weak))) obj1 = obj;
    NSLog(@"------%@--%d--",[obj1 class],10);
}
```

3.2. Lexical Analysis - 词法分析（输出token流）

使用 clang 命令 `clang -Xclang -dump-tokens main.m` **转化后的代码如下（去掉了#import <Foundation/Foundation.h>的内容）：**

> **词法分析，只需要将源代码以字符文本的形式转化成Token流的形式，不涉及交验语义，不需要递归，是线性的。**

> 什么是token流呢？可以这么理解：就是有"类型"，有"值"的一些小单元。

词法分析其实是编译器开始工作真正意义上的第一个步骤，其所做的工作主要为将输入的代码转换为一系列符合特定语言的词法单元，这些词法单元类型包括了关键字，操作符，变量等等。

可以通过下发被编译过的代码对应main.m文件，把所有的内容都一一对应起来。

```
int main(){
    NSObject *obj = [[NSObject alloc] init];
    id __attribute__((objc_ownership(weak))) obj1 = obj;
    NSLog(@"------%@--%d--",[obj1 class],10);
}
```

//编译后

```
int 'int'    [StartOfLine]  Loc=<main3.m:11:1>
identifier 'main'    [LeadingSpace] Loc=<main3.m:11:5>
l_paren '('     Loc=<main3.m:11:9>
r_paren ')'     Loc=<main3.m:11:10>
l_brace '{'     Loc=<main3.m:11:11>
identifier 'NSObject'    [StartOfLine] [LeadingSpace]   Loc=<main3.m:13:5>
star '*'     [LeadingSpace] Loc=<main3.m:13:14>
identifier 'obj'        Loc=<main3.m:13:15>
equal '='    [LeadingSpace] Loc=<main3.m:13:19>
l_square '['     [LeadingSpace] Loc=<main3.m:13:21>
l_square '['        Loc=<main3.m:13:22>
identifier 'NSObject'       Loc=<main3.m:13:23>
identifier 'alloc'   [LeadingSpace] Loc=<main3.m:13:32>
r_square ']'        Loc=<main3.m:13:37>
identifier 'init'    [LeadingSpace] Loc=<main3.m:13:39>
r_square ']'        Loc=<main3.m:13:43>
semi ';'        Loc=<main3.m:13:44>
identifier 'id'  [StartOfLine] [LeadingSpace]   Loc=<main3.m:14:5>
__attribute '__attribute__'  [LeadingSpace] Loc=<main3.m:14:8 <Spelling=<built-in>:309:16>>
l_paren '('     Loc=<main3.m:14:8 <Spelling=<built-in>:309:29>>
l_paren '('     Loc=<main3.m:14:8 <Spelling=<built-in>:309:30>>
identifier 'objc_ownership'     Loc=<main3.m:14:8 <Spelling=<built-in>:309:31>>
l_paren '('     Loc=<main3.m:14:8 <Spelling=<built-in>:309:45>>
identifier 'weak'       Loc=<main3.m:14:8 <Spelling=<built-in>:309:46>>
r_paren ')'     Loc=<main3.m:14:8 <Spelling=<built-in>:309:50>>
r_paren ')'     Loc=<main3.m:14:8 <Spelling=<built-in>:309:51>>
r_paren ')'     Loc=<main3.m:14:8 <Spelling=<built-in>:309:52>>
identifier 'obj1'    [LeadingSpace] Loc=<main3.m:14:15>
equal '='    [LeadingSpace] Loc=<main3.m:14:20>
identifier 'obj'     [LeadingSpace] Loc=<main3.m:14:22>
semi ';'        Loc=<main3.m:14:25>
identifier 'NSLog'   [StartOfLine] [LeadingSpace]   Loc=<main3.m:15:5>
l_paren '('     Loc=<main3.m:15:10>
at '@'      Loc=<main3.m:15:11>
string_literal '"------%@--%d--"'       Loc=<main3.m:15:12>
comma ','       Loc=<main3.m:15:28>
l_square '['        Loc=<main3.m:15:29>
identifier 'obj1'       Loc=<main3.m:15:30>
identifier 'class'   [LeadingSpace] Loc=<main3.m:15:35>
r_square ']'        Loc=<main3.m:15:40>
comma ','       Loc=<main3.m:15:41>
numeric_constant '10'       Loc=<main3.m:15:42 <Spelling=main3.m:10:12>>
r_paren ')'     Loc=<main3.m:15:44>
semi ';'        Loc=<main3.m:15:45>
r_brace '}'  [StartOfLine]  Loc=<main3.m:17:1>
eof ''      Loc=<main3.m:17:2>
```

> 这里，每一个符号都会标记出来其位置，这个位置是宏展开之前的位置，这样后面如果发现报错，就可以正确的提示错误位置了。

3.3.Semantic Analysis - 语法分析（输出(AST)抽象语法树）

对代码进行标记并不是Clang最终的目的，而是一个Clang的一个过程，其实标记代码为了让代码更便于转化成机器语言，标记代码转化成抽象语法树（abstract syntax tree – AST）是一个必经之路。

3.3.1 AST

使用 clang 命令 `clang -Xclang -ast-dump -fsyntax-only main.m` **，转化后的树如下**

```
-FunctionDecl 0x7fbbea20f538 <main3.m:11:1, line:17:1> line:11:5 main 'int ()'
  `-CompoundStmt 0x7fbbea20fa40 <col:11, line:17:1>
    |-DeclStmt 0x7fbbea20f6b8 <line:13:5, col:44>
    | `-VarDecl 0x7fbbea20f5e8 <col:5, col:43> col:15 used obj 'NSObject *' cinit
    |   `-ObjCMessageExpr 0x7fbbea20f688 <col:21, col:43> 'NSObject *' selector=init
    |     `-ObjCMessageExpr 0x7fbbea20f658 <col:22, col:37> 'NSObject *' selector=alloc class='NSObject'
    |-DeclStmt 0x7fbbea20f830 <line:14:5, col:25>
    | `-VarDecl 0x7fbbea20f778 <col:5, col:22> col:15 used obj1 '__weak id':'__weak id' cinit
    |   `-ImplicitCastExpr 0x7fbbea20f818 <col:22> 'id':'id' <BitCast>
    |     `-ImplicitCastExpr 0x7fbbea20f800 <col:22> 'NSObject *' <LValueToRValue>
    |       `-DeclRefExpr 0x7fbbea20f7d8 <col:22> 'NSObject *' lvalue Var 0x7fbbea20f5e8 'obj' 'NSObject *'
    `-ExprWithCleanups 0x7fbbea20fa28 <line:15:5, col:44> 'void'
      `-CallExpr 0x7fbbea20f9d0 <col:5, col:44> 'void'
        |-ImplicitCastExpr 0x7fbbea20f9b8 <col:5> 'void (*)(id, ...)' <FunctionToPointerDecay>
        | `-DeclRefExpr 0x7fbbea20f848 <col:5> 'void (id, ...)' Function 0x7fbbe94469e0 'NSLog' 'void (id,
...)'
        |-ImplicitCastExpr 0x7fbbea20fa10 <col:11, col:12> 'id':'id' <BitCast>
        | `-ObjCStringLiteral 0x7fbbea20f8a8 <col:11, col:12> 'NSString *'
        |   `-StringLiteral 0x7fbbea20f870 <col:12> 'char [15]' lvalue "------%@--%d--"
        |-ObjCMessageExpr 0x7fbbea20f908 <col:29, col:40> 'Class':'Class' selector=class
        | `-ImplicitCastExpr 0x7fbbea20f8f0 <col:30> 'id':'id' <LValueToRValue>
        |   `-DeclRefExpr 0x7fbbea20f8c8 <col:30> '__weak id':'__weak id' lvalue Var 0x7fbbea20f778 'obj1'
'__weak id':'__weak id'
        `-IntegerLiteral 0x7fbbea20f938 <line:10:12> 'int' 10
```

这个main方法的抽象树可以看出来树顶是 `FunctionDecl` **:方法声明（Function Declaration）。**

> 这里因为截取了部分代码，其实并不是整个树的树顶。真正的树顶描述应该是：TranslationUnitDecl。

详细的AST语法不多介绍，关于 AST 的详细解释可以查看：Introduction to the Clang AST**。**

3.3.2 静态分析

- **通过语法树进行代码静态分析，找出非语法性错误**
- **模拟代码执行路径，分析出**control-flow graph(CFG) **【MRC时代会分析出引用计数的错误】**

- 预置了常用Checker（检查器）

3.4. CodeGen - （Intermediate Representation，简称IR）IR中间代码生成

当通过Clang语法解析，代码没有出现报错，Clang前端就将进入最后一步：生成LLVM IR中间代码，并将生成的LLVM IR代码递交给优化器。

使用命令 `clang -S -emit-llvm main3.m -o main.ll` **生成LLVM 中间代码LLVM IR**

```
#import <Foundation/Foundation.h>
#define aa 10
int main(){

    NSObject *obj = [[NSObject alloc] init];
    NSLog(@"------%@--%d--",[obj class],aa);

}
```

```
; ModuleID = 'main3.m'
source_filename = "main3.m"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

%0 = type opaque
%struct._class_t = type { %struct._class_t*, %struct._class_t*, %struct._objc_cache*, i8* (i8*, i8*)**,
%struct._class_ro_t* }
%struct._objc_cache = type opaque
%struct._class_ro_t = type { i32, i32, i32, i8*, i8*, %struct.__method_list_t*, %struct._objc_protocol_list*,
%struct._ivar_list_t*, i8*, %struct._prop_list_t* }
%struct.__method_list_t = type { i32, i32, [0 x %struct._objc_method] }
%struct._objc_method = type { i8*, i8*, i8* }
%struct._objc_protocol_list = type { i64, [0 x %struct._protocol_t*] }
%struct._protocol_t = type { i8*, i8*, %struct._objc_protocol_list*, %struct.__method_list_t*,
%struct.__method_list_t*, %struct.__method_list_t*, %struct.__method_list_t*, %struct._prop_list_t*, i32,
i32, i8**, i8*, %struct._prop_list_t* }
%struct._ivar_list_t = type { i32, i32, [0 x %struct._ivar_t] }
%struct._ivar_t = type { i64*, i8*, i8*, i32, i32 }
%struct._prop_list_t = type { i32, i32, [0 x %struct._prop_t] }
%struct._prop_t = type { i8*, i8* }
%struct.__NSConstantString_tag = type { i32*, i32, i8*, i64 }

@"OBJC_CLASS_$_NSObject" = external global %struct._class_t
@"OBJC_CLASSLIST_REFERENCES_$_" = private global %struct._class_t* @"OBJC_CLASS_$_NSObject",
section "__DATA,__objc_classrefs,regular,no_dead_strip", align 8
@OBJC_METH_VAR_NAME_ = private unnamed_addr constant [6 x i8] c"alloc\00", section
"__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_ = private externally_initialized global i8* getelementptr inbounds ([6 x
i8], [6 x i8]* @OBJC_METH_VAR_NAME_, i32 0, i32 0), section
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
```

```
@OBJC_METH_VAR_NAME_.1 = private unnamed_addr constant [5 x i8] c"init\00", section
"__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_.2 = private externally_initialized global i8* getelementptr inbounds ([5 x
i8], [5 x i8]* @OBJC_METH_VAR_NAME_.1, i32 0, i32 0), section
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
@__CFConstantStringClassReference = external global [0 x i32]
@.str = private unnamed_addr constant [15 x i8] c"------%@--%d--\00", section
"__TEXT,__cstring,cstring_literals", align 1
@_unnamed_cfstring_ = private global %struct.__NSConstantString_tag { i32* getelementptr inbounds ([0
x i32], [0 x i32]* @__CFConstantStringClassReference, i32 0, i32 0), i32 1992, i8* getelementptr inbounds
([15 x i8], [15 x i8]* @.str, i32 0, i32 0), i64 14 }, section "__DATA,__cfstring", align 8
@OBJC_METH_VAR_NAME_.3 = private unnamed_addr constant [6 x i8] c"class\00", section
"__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_.4 = private externally_initialized global i8* getelementptr inbounds ([6 x
i8], [6 x i8]* @OBJC_METH_VAR_NAME_.3, i32 0, i32 0), section
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
@llvm.compiler.used = appending global [7 x i8*] [i8* bitcast (%struct._class_t**
@"OBJC_CLASSLIST_REFERENCES_$_" to i8*), i8* getelementptr inbounds ([6 x i8], [6 x i8]*
@OBJC_METH_VAR_NAME_, i32 0, i32 0), i8* bitcast (i8** @OBJC_SELECTOR_REFERENCES_ to i8*), i8*
getelementptr inbounds ([5 x i8], [5 x i8]* @OBJC_METH_VAR_NAME_.1, i32 0, i32 0), i8* bitcast (i8**
@OBJC_SELECTOR_REFERENCES_.2 to i8*), i8* getelementptr inbounds ([6 x i8], [6 x i8]*
@OBJC_METH_VAR_NAME_.3, i32 0, i32 0), i8* bitcast (i8** @OBJC_SELECTOR_REFERENCES_.4 to i8*)],
section "llvm.metadata"

; Function Attrs: noinline optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca %0*, align 8
  %2 = load %struct._class_t*, %struct._class_t** @"OBJC_CLASSLIST_REFERENCES_$_", align 8
  %3 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_, align 8, !invariant.load !8
  %4 = bitcast %struct._class_t* %2 to i8*
  %5 = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %4, i8* %3)
  %6 = bitcast i8* %5 to %0*
  %7 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_.2, align 8, !invariant.load !8
  %8 = bitcast %0* %6 to i8*
  %9 = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %8, i8* %7)
  %10 = bitcast i8* %9 to %0*
  store %0* %10, %0** %1, align 8
  %11 = load %0*, %0** %1, align 8
  %12 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_.4, align 8, !invariant.load !8
  %13 = bitcast %0* %11 to i8*
  %14 = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %13, i8* %12)
  notail call void (i8*, ...) @NSLog(i8* bitcast (%struct.__NSConstantString_tag* @_unnamed_cfstring_ to
i8*), i8* %14, i32 10)
  ret i32 0
}

; Function Attrs: nonlazybind
declare i8* @objc_msgSend(i8*, i8*, ...) #1

declare void @NSLog(i8*, ...) #2
```

```
attributes #0 = { noinline optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16,+fxsr,+mmx,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes
attributes

!llvm.module.flags = !{!0, !1, !2, !3, !4, !5, !6}
!llvm.ident = !{!7}

!0 = !{i32 1, !"Objective-C Version", i32 2}
!1 = !{i32 1, !"Objective-C Image Info Version", i32 0}
!2 = !{i32 1, !"Objective-C Image Info Section", !"__DATA,__objc_imageinfo,regular,no_dead_strip"}
!3 = !{i32 4, !"Objective-C Garbage Collection", i32 0}
!4 = !{i32 1, !"Objective-C Class Properties", i32 64}
!5 = !{i32 1, !"wchar_size", i32 4}
!6 = !{i32 7, !"PIC Level", i32 2}
!7 = !{!"Apple LLVM version 9.1.0 (clang-902.0.39.2)"}
!8 = !{}
```

在这里简单介绍一些 LLVM IR 的指令：

> %：局部变量
> @：全局变量
> alloca：分配内存堆栈
> i32：32 位的整数
> i32**：一个指向 32 位 int 值的指针的指针
> align 4：向 4 个字节对齐，即便数据没有占用 4 个字节，也要为其分配四个字节
> call：调用

LLVM IR 是Frontend的输出，也是LLVM Backend的输入，前后端的桥接语言, 更具生成的文件解析，其实生成的LLVM IR对Runtime进行桥接的一个文件

1.Class/Meta Class/Protocol/Category内存结构生成，并存放在指定section中（如Class：_DATA,_objc_classrefs）
2.Method/Ivar/Property内存结构生成
3.组成method_list/ivar_list/property_list并填入Class
4.Non-Fragile ABI:为每个Ivar合成OBJC_IVAR_$_ 偏移值常量
5.存取Ivar的语句（ivar = 123; int a = ivar;）转写成base + OBJC_IVAR$_的形式
6.将语法树中的ObjcMessageExpr翻译成相应版本的objc_msgSend，7.对super关键字的调用翻译成objc_msgSendSuper
8.根据修饰符strong/weak/copy/atomic合成@property 自动实现的 setter/getter
9.处理@synthesize
10.生成block_layout的数据结构
11.变量的capture(__block/__weak)
12.生成_block_invoke函数
13.ARC：分析对象引用关系，将objc_storeStrong/objc_storeWeak等ARC代码插入
14.将ObjCAutoreleasePoolStmt转译成objc_autoreleasePoolPush/Pop
15.实现自动调用[super dealloc]
16为每个拥有ivar的Class合成.cxx_destructor方法来自动释放类的成员变量，代替MRC时代的“self.xxx = nil”

3.5. Optimize - 优化IR

使用命令 `clang -O3 -S -emit-llvm main3.m -o main3.ll`

```
; ModuleID = 'main3.m'
source_filename = "main3.m"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

%struct._class_t = type { %struct._class_t*, %struct._class_t*, %struct._objc_cache*, i8* (i8*, i8*)**, %struct._class_ro_t* }
%struct._objc_cache = type opaque
%struct._class_ro_t = type { i32, i32, i32, i8*, i8*, %struct.__method_list_t*, %struct._objc_protocol_list*, %struct._ivar_list_t*, i8*, %struct._prop_list_t* }
%struct.__method_list_t = type { i32, i32, [0 x %struct._objc_method] }
%struct._objc_method = type { i8*, i8*, i8* }
%struct._objc_protocol_list = type { i64, [0 x %struct._protocol_t*] }
%struct._protocol_t = type { i8*, i8*, %struct._objc_protocol_list*, %struct.__method_list_t*, %struct.__method_list_t*, %struct.__method_list_t*, %struct.__method_list_t*, %struct._prop_list_t*, i32, i32, i8**, i8*, %struct._prop_list_t* }
%struct._ivar_list_t = type { i32, i32, [0 x %struct._ivar_t] }
%struct._ivar_t = type { i64*, i8*, i8*, i32, i32 }
%struct._prop_list_t = type { i32, i32, [0 x %struct._prop_t] }
%struct._prop_t = type { i8*, i8* }
%struct.__NSConstantString_tag = type { i32*, i32, i8*, i64 }

@"OBJC_CLASS_$_NSObject" = external global %struct._class_t
@"OBJC_CLASSLIST_REFERENCES_$_" = private global %struct._class_t* @"OBJC_CLASS_$_NSObject", section "__DATA,__objc_classrefs,regular,no_dead_strip", align 8
@OBJC_METH_VAR_NAME_ = private unnamed_addr constant [6 x i8] c"alloc\00", section "__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_ = private externally_initialized global i8* getelementptr inbounds ([6 x i8], [6 x i8]* @OBJC_METH_VAR_NAME_, i64 0, i64 0), section
```

```
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
@OBJC_METH_VAR_NAME_.1 = private unnamed_addr constant [5 x i8] c"init\00", section
"__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_.2 = private externally_initialized global i8* getelementptr inbounds ([5 x
i8], [5 x i8]* @OBJC_METH_VAR_NAME_.1, i64 0, i64 0), section
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
@__CFConstantStringClassReference = external global [0 x i32]
@.str = private unnamed_addr constant [15 x i8] c"------%@--%d--\00", section
"__TEXT,__cstring,cstring_literals", align 1
@_unnamed_cfstring_ = private global %struct.__NSConstantString_tag { i32* getelementptr inbounds ([0
x i32], [0 x i32]* @__CFConstantStringClassReference, i32 0, i32 0), i32 1992, i8* getelementptr inbounds
([15 x i8], [15 x i8]* @.str, i32 0, i32 0), i64 14 }, section "__DATA,__cfstring", align 8
@OBJC_METH_VAR_NAME_.3 = private unnamed_addr constant [6 x i8] c"class\00", section
"__TEXT,__objc_methname,cstring_literals", align 1
@OBJC_SELECTOR_REFERENCES_.4 = private externally_initialized global i8* getelementptr inbounds ([6 x
i8], [6 x i8]* @OBJC_METH_VAR_NAME_.3, i64 0, i64 0), section
"__DATA,__objc_selrefs,literal_pointers,no_dead_strip", align 8
@llvm.compiler.used = appending global [7 x i8*] [i8* bitcast (%struct._class_t**
@"OBJC_CLASSLIST_REFERENCES_$_" to i8*), i8* getelementptr inbounds ([6 x i8], [6 x i8]*
@OBJC_METH_VAR_NAME_, i32 0, i32 0), i8* getelementptr inbounds ([5 x i8], [5 x i8]*
@OBJC_METH_VAR_NAME_.1, i32 0, i32 0), i8* getelementptr inbounds ([6 x i8], [6 x i8]*
@OBJC_METH_VAR_NAME_.3, i32 0, i32 0), i8* bitcast (i8** @OBJC_SELECTOR_REFERENCES_ to i8*), i8*
bitcast (i8** @OBJC_SELECTOR_REFERENCES_.2 to i8*), i8* bitcast (i8**
@OBJC_SELECTOR_REFERENCES_.4 to i8*)], section "llvm.metadata"

; Function Attrs: ssp uwtable
define i32 @main() local_unnamed_addr #0 {
  %1 = load i8*, i8** bitcast (%struct._class_t** @"OBJC_CLASSLIST_REFERENCES_$_" to i8**), align 8
  %2 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_, align 8, !invariant.load !8
  %3 = tail call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %1, i8* %2)
  %4 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_.2, align 8, !invariant.load !8
  %5 = tail call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %3, i8* %4)
  %6 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_.4, align 8, !invariant.load !8
  %7 = tail call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*, i8*)*)(i8* %5, i8* %6)
  notail call void (i8*, ...) @NSLog(i8* bitcast (%struct.__NSConstantString_tag* @_unnamed_cfstring_ to
i8*), i8* %7, i32 10)
  ret i32 0
}

; Function Attrs: nonlazybind
declare i8* @objc_msgSend(i8*, i8*, ...) local_unnamed_addr #1

declare void @NSLog(i8*, ...) local_unnamed_addr #2

attributes #0 = { ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false"
"less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-
features"="+cx16,+fxsr,+mmx,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-
soft-float"="false" }
attributes
```

```
attributes

!llvm.module.flags = !{!0, !1, !2, !3, !4, !5, !6}
!llvm.ident = !{!7}

!0 = !{i32 1, !"Objective-C Version", i32 2}
!1 = !{i32 1, !"Objective-C Image Info Version", i32 0}
!2 = !{i32 1, !"Objective-C Image Info Section", !"__DATA,__objc_imageinfo,regular,no_dead_strip"}
!3 = !{i32 4, !"Objective-C Garbage Collection", i32 0}
!4 = !{i32 1, !"Objective-C Class Properties", i32 64}
!5 = !{i32 1, !"wchar_size", i32 4}
!6 = !{i32 7, !"PIC Level", i32 2}
!7 = !{!"Apple LLVM version 9.1.0 (clang-902.0.39.2)"}
!8 = !{}
```

这一步骤的优化是非常重要的，很多直接转换来的代码是不合适且消耗内存的，因为是直接转换，所以必然会有这样的问题，而优化放在这一步的好处在于前端不需要考虑任何优化过程，减少了前端的开发工作。

3.6. 生成Target相关汇编

使用命令 `clang -S -o - main3.m | open -f` **可以查看生成的汇编代码：**

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl  _main
    .p2align    4, 0x90
_main:
    .cfi_startproc

    pushq   %rbp
Lcfi0:
    .cfi_def_cfa_offset 16
Lcfi1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Lcfi2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movq    L_OBJC_CLASSLIST_REFERENCES_$_(%rip), %rax
    movq    L_OBJC_SELECTOR_REFERENCES_(%rip), %rsi
    movq    %rax, %rdi
    callq   _objc_msgSend
    movq    L_OBJC_SELECTOR_REFERENCES_.2(%rip), %rsi
    movq    %rax, %rdi
    callq   _objc_msgSend
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rax
    movq    L_OBJC_SELECTOR_REFERENCES_.4(%rip), %rsi
    movq    %rax, %rdi
    callq   _objc_msgSend
```

```
    leaq    L__unnamed_cfstring_(%rip), %rsi
    movl    $10, %edx
    movq    %rsi, %rdi
    movq    %rax, %rsi
    movb    $0, %al
    callq   _NSLog
    xorl    %eax, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc

    .section    __DATA,__objc_classrefs,regular,no_dead_strip
    .p2align    3
L_OBJC_CLASSLIST_REFERENCES_$_:
    .quad   _OBJC_CLASS_$_NSObject

    .section    __TEXT,__objc_methname,cstring_literals
L_OBJC_METH_VAR_NAME_:
    .asciz  "alloc"

    .section    __DATA,__objc_selrefs,literal_pointers,no_dead_strip
    .p2align    3
L_OBJC_SELECTOR_REFERENCES_:
    .quad   L_OBJC_METH_VAR_NAME_

    .section    __TEXT,__objc_methname,cstring_literals
L_OBJC_METH_VAR_NAME_.1:
    .asciz  "init"

    .section    __DATA,__objc_selrefs,literal_pointers,no_dead_strip
    .p2align    3
L_OBJC_SELECTOR_REFERENCES_.2:
    .quad   L_OBJC_METH_VAR_NAME_.1

    .section    __TEXT,__cstring,cstring_literals
L_.str:
    .asciz  "------%@--%d--"

    .section    __DATA,__cfstring
    .p2align    3
L__unnamed_cfstring_:
    .quad   ___CFConstantStringClassReference
    .long   1992
    .space  4
    .quad   L_.str
    .quad   14

    .section    __TEXT,__objc_methname,cstring_literals
L_OBJC_METH_VAR_NAME_.3:
    .asciz  "class"
```

```
    .section    __DATA,__objc_selrefs,literal_pointers,no_dead_strip
    .p2align    3
L_OBJC_SELECTOR_REFERENCES_.4:
    .quad    L_OBJC_METH_VAR_NAME_.3

    .section    __DATA,__objc_imageinfo,regular,no_dead_strip
L_OBJC_IMAGE_INFO:
    .long    0
    .long    64


.subsections_via_symbols
```

注意代码中的 .section 指令，它指定了接下来会执行的代码段。

在这篇文章中，详细解释了这些汇编指令或代码到底是如何工作的：Mach-O 可执行文件。

3.7. Link生成Executable

在最后，LLVM 将会把这些汇编代码输出成二进制的可执行文件，使用命令 clang main3.m -o main.out 即可查看

```
Segment __PAGEZERO: 0x100000000 (vmaddr 0x0 fileoff 0)
Segment __TEXT: 0x1000 (vmaddr 0x100000000 fileoff 0)
    Section __text: 0x34 (addr 0x100000f50 offset 3920)
    Section __stubs: 0x6 (addr 0x100000f84 offset 3972)
    Section __stub_helper: 0x1a (addr 0x100000f8c offset 3980)
    Section __cstring: 0xe (addr 0x100000fa6 offset 4006)
    Section __unwind_info: 0x48 (addr 0x100000fb4 offset 4020)
    total 0xaa
Segment __DATA: 0x1000 (vmaddr 0x100001000 fileoff 4096)
    Section __nl_symbol_ptr: 0x10 (addr 0x100001000 offset 4096)
    Section __la_symbol_ptr: 0x8 (addr 0x100001010 offset 4112)
    total 0x18
Segment __LINKEDIT: 0x1000 (vmaddr 0x100002000 fileoff 8192)
total 0x100003000
```

上面的代码中，每个 segment 的意义也不一样：

- `__PAGEZERO` segment 它的大小为 4GB。这 4GB 并不是文件的真实大小，但是规定了进程地址空间的前 4GB 被映射为 不可执行、不可写和不可读。
- `__TEXT` segment 包含了被执行的代码。它被以只读和可执行的方式映射。进程被允许执行这些代码，但是不能修改。
- `__DATA` segment 以可读写和不可执行的方式映射。它包含了将会被更改的数据。
- `__LINKEDIT` segment 指出了 link edit 表（包含符号和字符串的动态链接器表）的地址，里面包含了加载程序的元数据，例如函数的名称和地址。

关于 section 中的内容的研究可以查看：
Mach-O 可执行文件
Mach-O 可执行文件2

PARSING MACH-O FILES

关于更详细的编译过程可以查看：
深入剖析 iOS 编译 Clang LLVM

4. Swift的编译过程

在 Swift 编译器结构 的官方文档中描述了 Swift 编译器是如何工作的，分为如下步骤：

- **解析：**解析器是一个简单的递归下降解析器（在 lib / Parse 中实现），带有集成的手动编码词法分析器。解析器负责生成没有任何语义或类型信息的抽象语法树（AST），并针对输入源的语法问题发出警告或错误。
- **语意分析：**语义分析（在 lib / Sema 中实现）负责解析 AST 并将其转换为格式良好的完全检查形式的 AST，并在源代码中发出语义问题的警告或错误。语义分析包括类型推断，如果成功，则所得到的代码是类型检查安全的 AST 。
- **Clang导入器：**Clang导入器（在 lib / ClangImporter 中实现）导入Clang模块，并将它们导出的 C 或 Objective-C API 映射到相应的 Swift API中。结果导入的 AST 可以通过语义分析来引用。
- **SIL生成：**Swift中间语言（Swift Intermediate Language，简称SIL）是一种高级的，Swift 特有的中间语言，适用于 Swift 代码的进一步分析和优化。SIL 生成阶段（在 lib / SILGen 中实现）将类型检查的 AST 降低到所谓的 “原始” SIL。SIL的设计描述在 docs/ SIL.rst 中可以看到。
- **SIL优化：**在SIL优化（在 lib/Analysis，lib/ ARC，lib/LoopTransforms，和 lib/Transforms 中实现）执行额外的高级别，Swift 特有的优化的程序，包括（例如）自动引用计数优化，虚拟化和通用专业化。
- **LLVM IR生成：**IR生成（在 lib/IRGen 中实现）将 SIL 降到 LLVM IR，此时LLVM可以继续对其进行优化并生成机器码。

相关内容不详细讲解，可参考：https://blog.csdn.net/aas319/article/details/78606342

三、相关衍生的内容

1.Xcode 编译设置

1.1 Build Settings

这里是编译设置，针对编译流程中的各个过程进行参数和工具的配置：

- Architectures：编译目标 CPU 架构，这里比较常见的是 `Build Active Architectures Only`（只编译为当前架构，是指你在 scheme 中选定的设备的 CPU 架构），`debug` 设置为 `YES`，`Release` 设置为 `NO`。
- Assets：`Assets.xcassets` 资源组的配置。
- Build Locations：查看 Build 日志可以看到在编译过程中的目标文件夹。
- Build Options：这里是一些编译的选项设定，包含：
    - 是否总是嵌入 Swift 标准库，这个在静态库和动态库的第一篇文章中有讲，iOS 系统目前是不包含 Swift 标准库的，都是被打包在项目中。
    - c/c++/objective-c 编译器：Apple LLVM 9.0
    - 是否打开 Bitcode
    - ...
- Deployment：iOS 部署设置。说白了就是安装到手机的设置。
- Headers：头文件？具体作用不详，知道的可以说一下。
- Kernel Module：内核模块，作用不详。
- Linking：链接设置，链接路径、链接标记、Mach-O 文件类型。
- Packaging：打包设置，info.plist 的路径设置、Bundle ID 、App 显示名称的设置。
- Search Paths：库的搜索路径、头文件的搜索路径。
- Signing：签名设置，开发、生产的签名设置，这些都和你在开发者网站配置的证书相关。
- Testing：测试设置，作用不详。
- Text-Based API：基于文本的 API，字面翻译，作用不详。
- Versioning：版本管理。
- Apple LLVM 9.0 系列：LLVM 的配置，包含路径、编译器每一步的设置、语言设置。在这里 `Apple LLVM 9.0 - Warnings` 可以选择在编译的时候将哪些情况认定为错误（Error）和警告（Warning），可以开启困难模式，任何一个小的警告都会被认定为错误。
- Asset Catalog Compiler - Options：Asset 文件的编译设置。
- Interface Builder Storyboard Compiler - Options：Storyboard 的编译设置。
- 以及一些静态分析和 Swift 编译器的设定。

1.2 Build Phases

编译阶段，编译的时候将根据顺序来进行编译。这里固定的有：

- Compile Sources：编译源文件。
- Link Binary With Libraries：相关的链接库。
- Copy Bundle Resources：要拷贝的资源文件，有时候如果一个资源文件在开发过程中发现找不到，可以在这里找一下，看看是不是加进来了。

如果使用了 Cocoapods，那么将会被添加：

- [CP] Check Pods Manifest.lock：检查 Podfile.lock 和 Manifest.lock 文件的一致性，这个会再后面的 Cocoapods 原理中详细解释。
- [CP] Embed Pods Frameworks：将所有 cocoapods 打的 framework 拷贝到包中。
- [CP] Copy Pods Resources：将所有 cocoapods 的资源文件拷贝到包中。

1.3 Build Rules

编译规则，这里设定了不同文件的处理方式，例如：

- Copy Plist File：在编译打包的时候，将 info.plist 文件拷贝。
- Compress PNG File：在编译打包的时候，将 PNG 文件压缩。
- Swift Compiler：Swift 文件的编译方式，使用 Swift 编译器。
- ....

2. Cocoapods 原理

使用了 Cocoapods 后，我们的编译流程会多出来一些，虽然每个 target 的编译流程都是一致的，但是 Cocoapods 是如何将这些库导入我们的项目、原项目和其他库之间的依赖又是如何实现的仍然是一个需要了解的知识点。下面这几篇文章从不同角度解释了 Cocoapods 是如何工作的：

这些文章大部分都是讲述了 Objective-C 语言下 Cocoapods 是如何使用静态库动态库并添加依赖的。特别说明一下 Swift 的实现：

Cocoapods 对于 Swift 和 Objective-C 的操作区别其实并不是很大，Swift 是必须使用动态库的，因此在 podfile 中我们必须添加代码 `use_frameworks!` 来指明 Cocoapods 所有管理的库都将被编译成 framework 动态库。这些库的依赖过程和 Objective-C 一致，我们的主项目依赖于 `Pods-ProjectName.framework` target，而这个 pods 的 target 则依赖于其他我们使用的库。

需要说明一个概念，就是 Swift 的命名空间和 Module。

- Module：在 Swift 中，项目里的每个 target、framework 都是一个 Module。
- 命名空间：每个 Module 都拥有独立的命名空间。

因此，我们在使用 Swift 库的时候，例如 `Alamofire`，在某个文件中 `import Alamofire`，其实就是在引入这个 Module，在代码提示中可以看到，这样其中的代码才可以被我们使用。

在同一个 Module 内我们的 Swift 文件是不需要被引用的，可以直接使用其中 `public`、`internal` 描述的类属性和方法。这也说明一个问题，在同一个 Module 中，命名空间也是同一个，重复的类都会在编译的时候报错。

总之

LLVM 的编译过程是相当复杂的，中间牵扯到的技术和语言要比我们做一个简单的移动开发要复杂的多，本文也只是非常浅显的描述了相关的编译过程，其主要目的是为了在 iOS 开发中更加得心应手。

研究编译原理、深究编译过程以及其相关内容，在开发过程中是非常重要的一个环节，任何开发者都应该做深入研究。如果你看不懂相关代码，也不要觉得困难，很多代码的命名就能很清楚的表明它到底是干什么的，英语差的话翻译一下就好了。只是相对于英语好，底层代码也略懂的情况来说，学习成本比较高，不过这不应该是打败你的理由。

原理是我们构建整个代码世界的基础，不懂原理（无论哪方面）都只能让我们做一个垒砖的，而不是整个工程的控制者。