iOS从编译过程出发研究代码混淆优化（一）

0x00 前言

我们常规的代码混淆是在代码阶段通过脚本语言例如Python进行混淆，我们叫它为硬编码混淆。硬编码混淆的代码可读性极差，不利于维护，如果线上有崩溃就需要把混淆过的代码做还原。所以我们想探索一下有没有更好的混淆代码的方式。我们带着好奇心从编译过程着手，看看有没有非侵入的代码混淆方案。

0x01 编译过程

首先我们查看一下编译过程，可以通过命令可以打印源码的编译阶段执行步骤

`clang -ccc-print-phases main.m`

HTTP

```
0: input, "main.m", objective-c
1: preprocessor, {0}, objective-c-cpp-output
2: compiler, {1}, ir
3: backend, {2}, assembler
4: assembler, {3}, object
5: linker, {4}, image
6: bind-arch, "arm64", {5}, image


0: 输入文件，找到源文件
1: 预处理阶段，该阶段会进行宏的替换、头文件的导入
2: 编译阶段，进行词法分析、语法分析、检测语法是否正确，生成IR
3: 后端，LLVM会通过一个一个Pass去优化，最终生成汇编代码
4: 生成目标文件
5: 链接动态库和静态库，生成可执行文件
6: 通过不同的架构生成对应的可执行文件
```

可以主要分为：

1、预处理

2、词法分析

3、语法分析

4、生成中间IR代码

5、优化

6、汇编代码

7、生成目标文件

8、链接生成可执行文件

一、预处理

1、头文件引入, 递归将头文件引用替换为头文件中的实际内容, 所以尽量减少头文件中的#import, 使用@class替代, 把#import放到.m文件中

```C
#import "" 引用自定义头文件
#import <> 引用系统头文件
#include ""
```

2、条件编译

```C
#if bool条件
#elif 否则
#endif 结束
#ifdef 判断某个宏是否被定义
#ifndef 判断某个宏是否没被定义

#if !DEBUG
 abc();
#endif
```

3、宏替换

MTABC用到的地方会被替换成3

```C
#define MTABC 3
```

4、注释处理, 在预处理的时候, 注释被删除

预处理主要是处理一些宏定义，比如 `#define`、`#include`、`#if` 等。预处理的实现有很多种，有的编译器会在词法分析前先进行预处理，替换掉所有 # 开头的宏，而有的编译器则是在词法

分析的过程中进行预处理。当分析到 # 开头的单词时才进行替换。虽然先预处理再词法分析比较符合直觉，但在实际使用中，GCC 使用的却是一边词法分析，一边预处理的方案。

例如：

C++

```
1
2  #include <stdio.h>
3
4  #define MTABC 3
5
6  #if MTABC==1
7  #define MTEGF 4
8  #else
9  #define MTEGF 5
10 #endif
11
12 int main(int argc, const char * argv[]) {
13     int a = 1;
14     int b = 2;
15     int c = a + b + MTABC + MTEGF;
16     printf("%d",c);
17     return 0;
18 }
```

查看预处理preprocess结果

Shell

```
1  Clang -E main.m
```

C++

```
# 此处省略一大片
extern int __vsnprintf_chk (char * restrict, size_t, int, size_t,
       const char * restrict, va_list);
# 400 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/stdio.h" 2 3 4
# 2 "main.m" 2
# 15 "main.m"
int main(int argc, const char * argv[]) {
    int a = 1;
    int b = 2;
    int c = a + b + 3 + 5;
    printf("%d",c);
    return 0;
}
```

二、词法分析

编译器不能像人一样，一眼就看明白源代码的内容，它只能比较傻的逐个单词分析。词法分析要做的就是把源代码分割开，形成若干个单词。这个过程并不像想象的那么简单。比如举几个例子:

1. `int a` 表示一个整数，而 `inta` 只是一个变量名。
2. `int b()` 表示一个函数而非整数 b，`int b ()` 也是一个函数。
3. `a =` 没有具体价值，它可以是一个赋值语句，还可以是 `a == 1` 的前缀，表示一个判断。

词法分析的主要实现原理是状态机，它逐个读取字符，然后根据读到的字符的特点转换状态。

Shell

```
clang -fmodules -E -Xclang -dump-tokens main.m
```

Objective-C

```

annot_module_include '#include <stdio.h>

'               Loc=<main.m:1:1>
int 'int'         [StartOfLine]       Loc=<main.m:15:1>
identifier 'main'         [LeadingSpace]       Loc=<main.m:15:5>
l_paren '('               Loc=<main.m:15:9>
int 'int'               Loc=<main.m:15:10>
identifier 'argc'         [LeadingSpace]       Loc=<main.m:15:14>
comma ','               Loc=<main.m:15:18>
const 'const'         [LeadingSpace]       Loc=<main.m:15:20>
char 'char'         [LeadingSpace]       Loc=<main.m:15:26>
```

```
star '*'         [LeadingSpace]        Loc=<main.m:15:31>
identifier 'argv'         [LeadingSpace]        Loc=<main.m:15:33>
l_square '['              Loc=<main.m:15:37>
r_square ']'              Loc=<main.m:15:38>
r_paren ')'               Loc=<main.m:15:39>
l_brace '{'         [LeadingSpace]        Loc=<main.m:15:41>
int 'int'         [StartOfLine] [LeadingSpace]        Loc=<main.m:16:5>
identifier 'a'         [LeadingSpace]        Loc=<main.m:16:9>
equal '='         [LeadingSpace]        Loc=<main.m:16:11>
numeric_constant '1'         [LeadingSpace]        Loc=<main.m:16:13>
semi ';'               Loc=<main.m:16:14>
int 'int'         [StartOfLine] [LeadingSpace]        Loc=<main.m:17:5>
identifier 'b'         [LeadingSpace]        Loc=<main.m:17:9>
equal '='         [LeadingSpace]        Loc=<main.m:17:11>
numeric_constant '2'         [LeadingSpace]        Loc=<main.m:17:13>
semi ';'               Loc=<main.m:17:14>
int 'int'         [StartOfLine] [LeadingSpace]        Loc=<main.m:18:5>
identifier 'c'         [LeadingSpace]        Loc=<main.m:18:9>
equal '='         [LeadingSpace]        Loc=<main.m:18:11>
identifier 'a'         [LeadingSpace]        Loc=<main.m:18:13>
plus '+'         [LeadingSpace]        Loc=<main.m:18:15>
identifier 'b'         [LeadingSpace]        Loc=<main.m:18:17>
plus '+'         [LeadingSpace]        Loc=<main.m:18:19>
numeric_constant '3'         [LeadingSpace]        Loc=<main.m:18:21 <Spelling
=main.m:3:15>>
plus '+'         [LeadingSpace]        Loc=<main.m:18:27>
numeric_constant '5'         [LeadingSpace]        Loc=<main.m:18:29 <Spelling
=main.m:11:15>>
semi ';'               Loc=<main.m:18:34>
identifier 'printf'         [StartOfLine] [LeadingSpace]        Loc=<main.m:19
:5>
l_paren '('               Loc=<main.m:19:11>
string_literal '"%d"'               Loc=<main.m:19:12>
comma ','               Loc=<main.m:19:16>
identifier 'c'               Loc=<main.m:19:17>
r_paren ')'               Loc=<main.m:19:18>
semi ';'               Loc=<main.m:19:19>
return 'return'         [StartOfLine] [LeadingSpace]        Loc=<main.m:20:5>
numeric_constant '0'         [LeadingSpace]        Loc=<main.m:20:12>
semi ';'               Loc=<main.m:20:13>
r_brace '}'         [StartOfLine]        Loc=<main.m:21:1>
eof ''               Loc=<main.m:23:1>
```

可以用开源工具PEGKit 做词法分析

三、语法分析

语法分析比词法分析更复杂，因为所有 C 语言支持的语法特性都必须被语法分析器正确的匹配，这个难度比纯新手学习 C 语言语法难上很多倍。不过这个属于业务复杂性，无论采用哪种解决方案都不可避免，因为语法规则的数量就是这么多。

luck.m

Objective-C

```
int luck(int a, int b) {
  int e = 4;
  int f = 2;
  int c = a + b + e + f;
  return c;
}
```

Shell

```
clang -fmodules -fsyntax-only -Xclang -ast-dump luck.m
```

Gherkin

```
TranslationUnitDecl 0x14b00aa08 <<invalid sloc>> <invalid sloc> <undeserialized declarations>

#此处省略很多行

`-FunctionDecl 0x135a926d0 <line:23:1, line:28:1> line:23:5 luck 'int (int, int)'
  |-ParmVarDecl 0x135a92540 <col:10, col:14> col:14 used a 'int'
  |-ParmVarDecl 0x135a925c0 <col:17, col:21> col:21 used b 'int'
  `-CompoundStmt 0x135a92b70 <col:24, line:28:1>
    |-DeclStmt 0x135a92838 <line:24:3, col:12>
    | `-VarDecl 0x135a927b0 <col:3, col:11> col:7 used e 'int' cinit
    |   `-IntegerLiteral 0x135a92818 <col:11> 'int' 4
    |-DeclStmt 0x135a928f0 <line:25:3, col:12>
    | `-VarDecl 0x135a92868 <col:3, col:11> col:7 used f 'int' cinit
    |   `-IntegerLiteral 0x135a928d0 <col:11> 'int' 2
    |-DeclStmt 0x135a92af8 <line:26:3, col:24>
    | `-VarDecl 0x135a92920 <col:3, col:23> col:7 used c 'int' cinit
    |   `-BinaryOperator 0x135a92ad8 <col:11, col:23> 'int' '+'
    |     |-BinaryOperator 0x135a92a68 <col:11, col:19> 'int' '+'
    |     | |-BinaryOperator 0x135a929f8 <col:11, col:15> 'int' '+'
    |     | | |-ImplicitCastExpr 0x135a929c8 <col:11> 'int' <LValueToRValue>
    |     | | | `-DeclRefExpr 0x135a92988 <col:11> 'int' lvalue ParmVar 0x135a92540 'a' 'int'
    |     | | `-ImplicitCastExpr 0x135a929e0 <col:15> 'int' <LValueToRValue>
    |     | |   `-DeclRefExpr 0x135a929a8 <col:15> 'int' lvalue ParmVar 0x135a925c0 'b' 'int'
    |     | `-ImplicitCastExpr 0x135a92a50 <col:19> 'int' <LValueToRValue>
    |     |   `-DeclRefExpr 0x135a92a18 <col:19> 'int' lvalue Var 0x135a927b0 'e' 'int'
    |     `-ImplicitCastExpr 0x135a92ac0 <col:23> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x135a92a88 <col:23> 'int' lvalue Var 0x135a92868 'f' 'int'
    `-ReturnStmt 0x135a92b60 <line:27:3, col:10>
      `-ImplicitCastExpr 0x135a92b48 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x135a92b10 <col:10> 'int' lvalue Var 0x135a92920 'c' 'int'
```

TranslationUnitDecl 根节点，表示一个编译单元

节点主要有三种：Type类型，Decl声明，Stmt陈述

ObjCInterfaceDecl OC中Interface声明

FunctionDecl 函数声明

ParmVarDecl 参数声明

CompoundStmt 具体语句

DeclStmt 语句声明

VarDecl 变量声明

IntegerLiteral 整数字面量

BinaryOperator 操作符

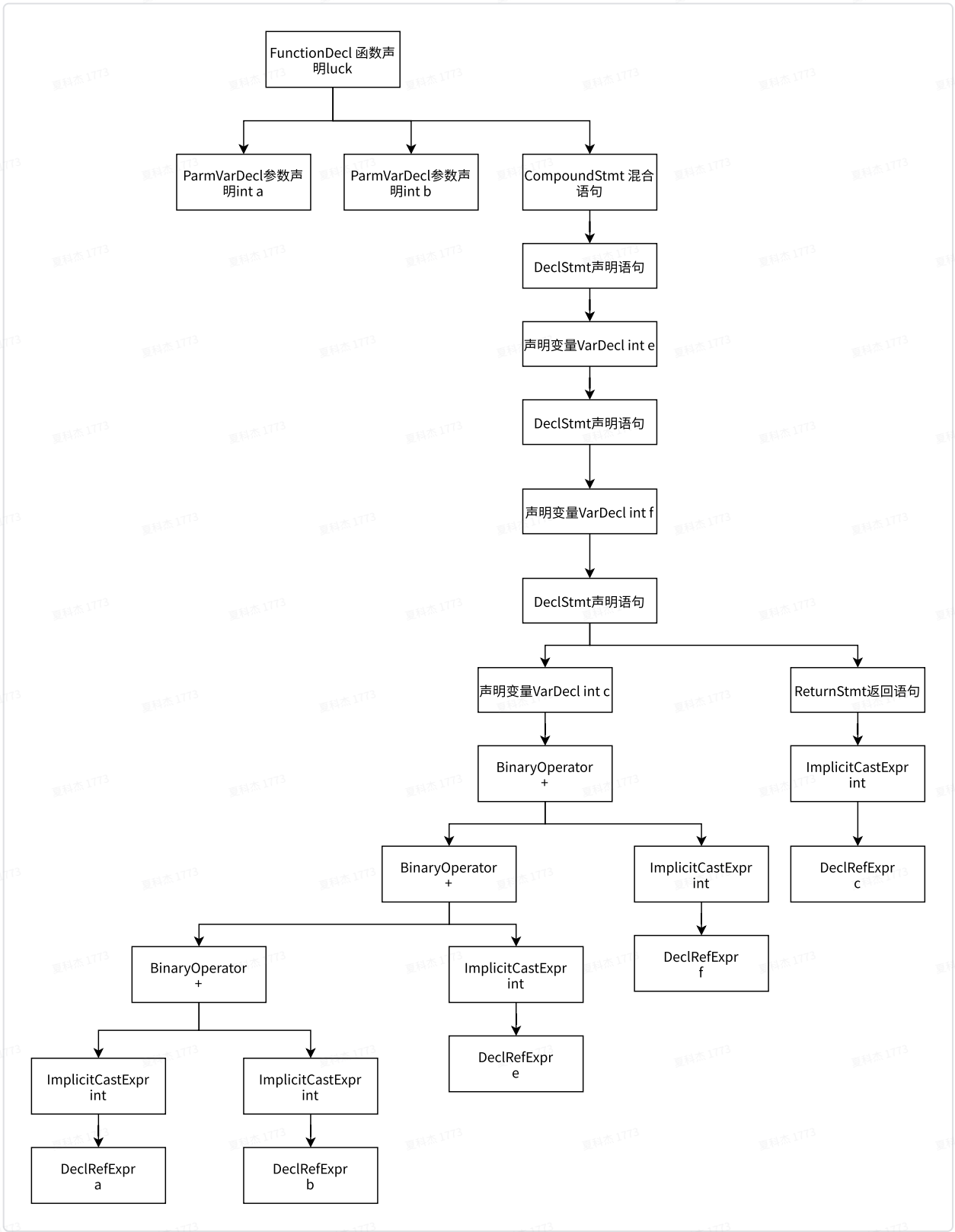
ImplicitCastExpr 隐式转换

DeclRefExpr 引用类型声明

ReturnStmt 返回语句

使用Clang的API可针对抽象语法树（AST）进行相应的分析及处理

对应的语法树模型如下：



四、生成中间IR代码

IR，即 Intermediate language ，LLVM IR有三种表现形式，类似于水有三种形态，气态，液态和固态。

1、文本格式

可读的汇编语言表示，一般以`.ll`的形式存在

Shell

```
clang main.m -S -emit-llvm
```

生成了带ll后缀的文件main.ll，用文本编辑器查看

Rust

```
; ModuleID = 'main.m'
source_filename = "main.m"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx12.0.0"

@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1

; Function Attrs: noinline optnone ssp uwtable
define i32 @main(i32 %0, i8** %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  %8 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  store i32 1, i32* %6, align 4
  store i32 2, i32* %7, align 4
  %9 = load i32, i32* %6, align 4
  %10 = load i32, i32* %7, align 4
  %11 = add nsw i32 %9, %10
  %12 = add nsw i32 %11, 3
  %13 = add nsw i32 %12, 5
  store i32 %13, i32* %8, align 4
  %14 = load i32, i32* %8, align 4
  %15 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.str, i64 0, i64 0), i32 %14)
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline optnone ssp uwtable "frame-pointer"="non-leaf" "min-legal-vector-width"="0" "no-trapping-math"="true" "probe-stack"="__chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="apple-m1" "target-feature
```

```
s"="+aes,+crc,+crypto,+dotprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+sm4,+v8.5a,+zcm,+zcz" }
attributes #1 = { "frame-pointer"="non-leaf" "no-trapping-math"="true" "probe-stack"="__chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="apple-m1" "target-features"="+aes,+crc,+crypto,+dotprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+sm4,+v8.5a,+zcm,+zcz" }

!llvm.module.flags = !{!0, !1, !2, !3, !4, !5, !6, !7, !8, !9, !10, !11, !12, !13, !14}
!llvm.ident = !{!15}

!0 = !{i32 2, !"SDK Version", [2 x i32] [i32 12, i32 3]}
!1 = !{i32 1, !"Objective-C Version", i32 2}
!2 = !{i32 1, !"Objective-C Image Info Version", i32 0}
!3 = !{i32 1, !"Objective-C Image Info Section", !"__DATA,__objc_imageinfo,regular,no_dead_strip"}
!4 = !{i32 1, !"Objective-C Garbage Collection", i8 0}
!5 = !{i32 1, !"Objective-C Class Properties", i32 64}
!6 = !{i32 1, !"Objective-C Enforce ClassRO Pointer Signing", i8 0}
!7 = !{i32 1, !"wchar_size", i32 4}
!8 = !{i32 1, !"branch-target-enforcement", i32 0}
!9 = !{i32 1, !"sign-return-address", i32 0}
!10 = !{i32 1, !"sign-return-address-all", i32 0}
!11 = !{i32 1, !"sign-return-address-with-bkey", i32 0}
!12 = !{i32 7, !"PIC Level", i32 2}
!13 = !{i32 7, !"uwtable", i32 1}
!14 = !{i32 7, !"frame-pointer", i32 1}
!15 = !{!"Apple clang version 13.1.6 (clang-1316.0.21.2.5)"}
```

2、二进制格式

一般以`.bc`的形式存在。方便JIT编译器快速加载。生成命令：

Shell

```
clang main.m  -c -emit-llvm
```

可以用llvm-dis或clang 命令 转main.ll文件。

将LLVM Bitcode（BC）转换成人可阅读的IR文件（LL）

Shell

```
clang -emit-llvm -S main.bc -o main.o.ll
```

Clang 还有很多常用工具如：opt、llc、llvm-mc、lli、llvm-link、llvm-as等

3、内存表示

`Instruction`类等，基于LLVM开发需要用到许多的类。`include "llvm/IR/XX"`

> **LLVM Language Reference Manual — LLVM 15.0.0git documentation**
> https://llvm.org/docs/LangRef.html?highlight=instruction#instruction-reference

IR语法

LLVM IR采用的是基于寄存器的满足RISC（精简指令集）架构以及load/store模式，也就是说只能通过将load和store 指令来进行CPU和内存间的数据交换。

讲一下IR关键字的意义：

- ; 注释，以;开头直到换行符
- define 函数定义
- declare 函数声明
- i32 所占bit位为32位
- ret 函数返回
- alloca 在当前执行的函数的栈空间分配内存，当函数返回时，自动释放内存
- align 内存对齐
- load 读取数据
- store 写入数据
- icmp 整数值比较，返回布尔值结果
- br 选择分支，根据cond来转向label
- label 代码标签
- %0，%1分别为函数参数

LLVM IR 标识符有两种基本类型：

- 全局标识符（函数，全局变量）

  以’@’字符开头
- 本地标识符（寄存器名称，类型）

  以’%’字符开头

有名字的值(Named Values)

```
格式：[%@][-a-zA-Z.][-a-zA-Z._0-9]*
例如：
%luck
@jjkkl
%a.really.long.identifier

```

无名字的值(Unnamed Values)

```
格式：用无符号数值作为它们的前缀
例如
%12
@2
%44
```

IR的语法格式是标准的三元格式： 操作符+操作数1+操作数2。

五、PASS优化

LLVM的优化是对中间代码IR进行优化。由多个PASS来完成，每个PASS完成特定的工作。

PASS即为一层一层相互独立的IR优化器。显然它的一个用处就是插桩，在Pass遍历LLVM IR的同时，自然就可以往里面插入新的代码。这一步可以用来做代码优化和代码混淆。

在iOS Xcode 的编译可以设置（BuildSettings->Code Generation->Optimization Level）里也可以设置优化级别-O1，-O3，-Os，-O0，还可以写些自己的 Pass，官方有比较完整的 Pass 教程:https://llvm.org/docs/WritingAnLLVMPass.html

这一步在后续llvm代码混淆非常重要，这个后面详谈。

六、汇编代码

我们通过中间代码.ll文件、.bc文件最终生成汇编代码。

Shell

```
clang -S -fobjc-arc main.bc -o main.s
clang -S -fobjc-arc main.ll -o main.s
```

Assembly language

```

        .section        __TEXT,__text,regular,pure_instructions
        .build_version macos, 12, 0       sdk_version 12, 3
        .globl        _main                             ; -- Begin function main
        .p2align        2
_main:                                  ; @main
        .cfi_startproc
; %bb.0:
        sub         sp, sp, #64
        stp         x29, x30, [sp, #48]             ; 16-byte Folded Spill
        add         x29, sp, #48
        .cfi_def_cfa w29, 16
        .cfi_offset w30, -8
        .cfi_offset w29, -16
        mov         w8, #0
        str         w8, [sp, #16]                   ; 4-byte Folded Spill
        stur         wzr, [x29, #-4]
        stur         w0, [x29, #-8]
        stur         x1, [x29, #-16]
        mov         w8, #1
        stur         w8, [x29, #-20]
        mov         w8, #2
        str         w8, [sp, #24]
        ldur         w8, [x29, #-20]
        ldr         w9, [sp, #24]
        add         w8, w8, w9
        add         w8, w8, #3
        add         w8, w8, #5
        str         w8, [sp, #20]
        ldr         w9, [sp, #20]
                                        ; implicit-def: $x8
        mov         x8, x9
        adrp         x0, l_.str@PAGE
        add         x0, x0, l_.str@PAGEOFF
        mov         x9, sp
        str         x8, [x9]
        bl         _printf
        ldr         w0, [sp, #16]                   ; 4-byte Folded Reload
        ldp         x29, x30, [sp, #48]             ; 16-byte Folded Reload
        add         sp, sp, #64
```

```
41          ret
42          .cfi_endproc
43                                          ; -- End function
44          .section        __TEXT,__cstring,cstring_literals
45  l_.str:                                 ; @.str
46          .asciz          "%d"
47
48          .section        __DATA,__objc_imageinfo,regular,no_dead_strip
49  L_OBJC_IMAGE_INFO:
50          .long           0
51          .long           64
52
53  .subsections_via_symbols
```

七、生成目标文件

目标文件

Shell

```
1  clang -fmodules -c main.m -o main.o
```

查看目标文件

Shell

```
1  xcrun nm -nm main.o
```

目标文件内容

Shell

```
1
2                   (undefined) external _printf
3  0000000000000000 (__TEXT,__text) external _main
4  0000000000000000 (__TEXT,__text) non-external ltmp0
5  0000000000000074 (__TEXT,__cstring) non-external l_.str
6  0000000000000074 (__TEXT,__cstring) non-external ltmp1
7  0000000000000077 (__DATA,__objc_imageinfo) non-external ltmp2
8  0000000000000080 (__LD,__compact_unwind) non-external ltmp3
```

八、链接生成可执行文件

链接器把编译产生的.o文件和（.dylb .a）文件，生成一个mach-o文件。

```Shell
clang main.o -o main
```

查看目标文件main

```Shell
xcrun nm -nm main
```

```Objective-C
(undefined) external _printf (from libSystem)
0000000100000000 (__TEXT,__text) [referenced dynamically] external __mh_execute_header
0000000100003f34 (__TEXT,__text) external _main
```

从目标文件能看出，_printf 方法来自libSystem库

常用转换命令

```Plain Text
.c -> .ll: clang -emit-llvm -S a.c -o a.ll
.c -> .bc: clang -emit-llvm -c a.c -o a.bc
.ll -> .bc: llvm-as a.ll -o a.bc
.bc -> .ll: llvm-dis a.bc -o a.ll
.bc -> .s: llc a.bc -o a.s
```

0x02 后续

后续分享

0x03 常见错误

1、 'stdio.h' file not found

Makefile

```
clang luck.m  -c -emit-llvm

luck.m:1:10: fatal error: 'stdio.h' file not found
#include <stdio.h>
         ^~~~~~~~~
1 error generated.
```

Bash

```
vim ~/.bashrc
添加
export SDKROOT=$(xcrun  --show-sdk-path)

source ~/.bashrc
```

名词解释：

GCC（GNU Compiler Collection，GNU编译器套装），是一套由 GNU 开发的编程语言编译器。GCC 原名为 GNU C 语言编译器，因为它原本只能处理 C语言。GCC 快速演进，变得可处理 C++、Fortran、Pascal、Objective-C、Java, 以及 Ada 等他语言。

`Clang`是`LLVM`项目中的一个子项目，它是基于`LLVM`架构的轻量级编译器，诞生之初是为了替代GCC，提供更快的编译速度。它是负责编译C、C++、Objecte-C语言的编译器，属于整个`LLVM`架构中的前端，对于开发者而言，研究Clang可以给我们带来很多好处。

LLVM (Low Level Virtual Machine，底层虚拟机) 提供了与编译器相关的支持，能够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。简而言之，可以作为多种编译器的后台来使用。

苹果公司一直使用 GCC 作为官方的编译器。GCC 作为一款开源的编译器，一直做得不错，但 Apple 对编译工具会提出更高的要求。原因主要有以下两点：

其一，是 Apple 对 Objective-C 语言（包括后来对C语言）新增很多特性，但 GCC 开发者并不买 Apple的账——不给实现，因此索性后来两者分成两条分支分别开发，这也造成 Apple 的编译器版本远落后于 GCC 的官方版本。其二，GCC 的代码耦合度太高，很难独立，而且越是后期的版本，代码质量越差，但 Apple 想做的很多功能（比如更好的 IDE 支持），需要模块化的方式来调用 GCC，但 GCC一直不给做。

参考

详解三大编译器:gcc、llvm 和 clang-51CTO.COM

大前端开发者需要了解的基础编译原理和语言知识 - 掘金

The LLVM Compiler Infrastructure Project

https://github.com/itod/pegkit