- 姓名：孙源

- ID：sunnyxx

- 公司：滴滴出行

- 博客：http://blog.sunnyxx.com

- 微博：@我就叫Sunny怎么了

- GitHub：http://github.com/forkingdog
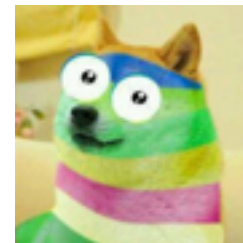
- 问：编译器可以编译程序，但编译器本身也是个程序，那它一定是由更早的编译器编译而成的，那…最早的一个编译器是哪儿来的？

- ➡️ Apple 编译器 Clang-LLVM 架构初识
- 你的源码是如何一步步成为可执行文件的?
- 我们能用 Clang 做什么有意思的事情?

- LLVM - Low Level Virtual Machine
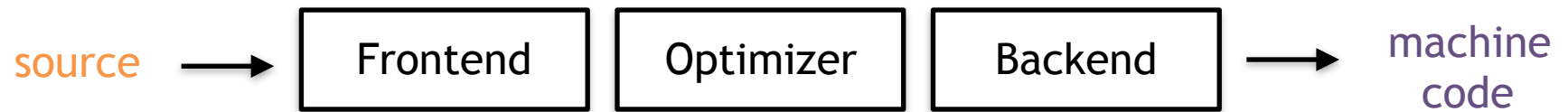- Clang /ˈklæŋ/ - C Language (Frontend for LLVM)

GCC 用的好好的，Apple 为啥要自己搞一套?

- GCC 的 Objective-C Frontend 不给力
- GCC 插件、工具、IDE 的支持薄弱
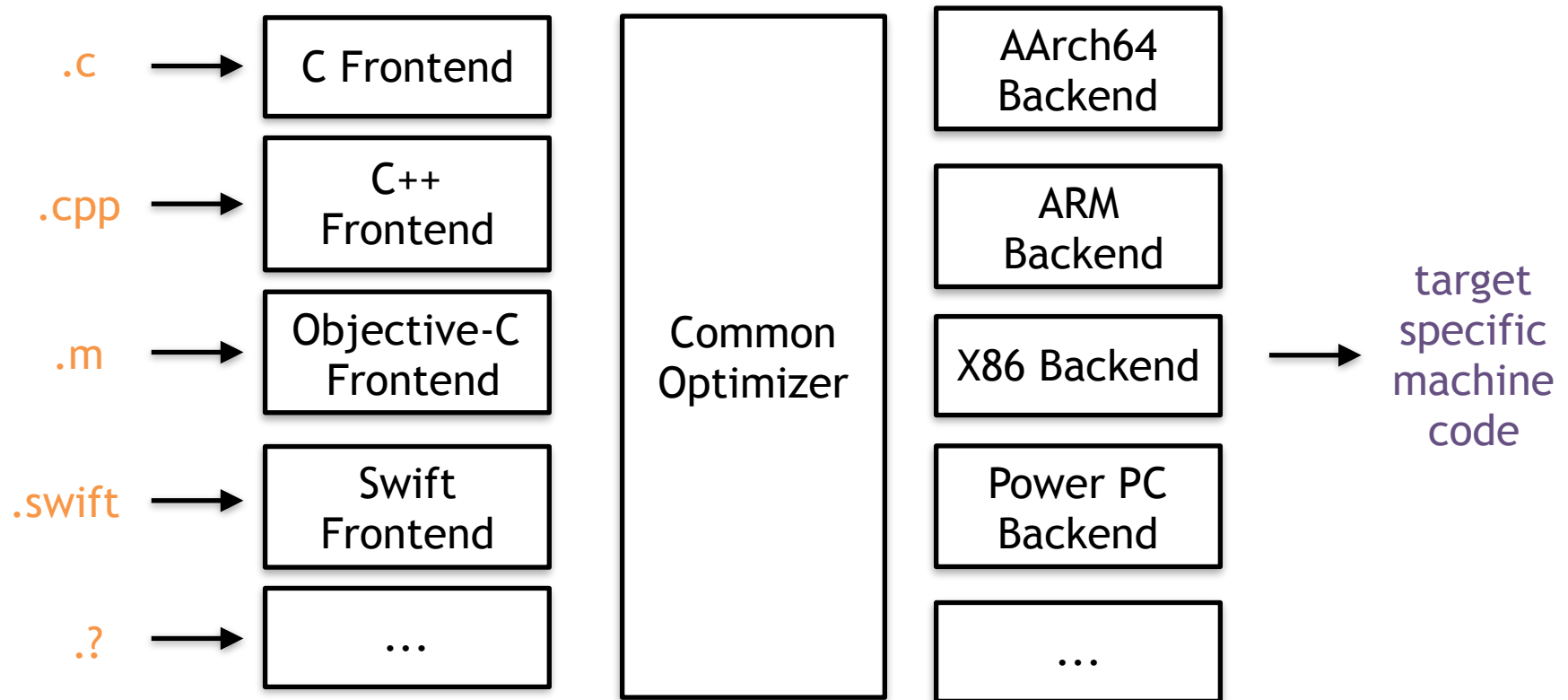- GCC 编译效率和性能
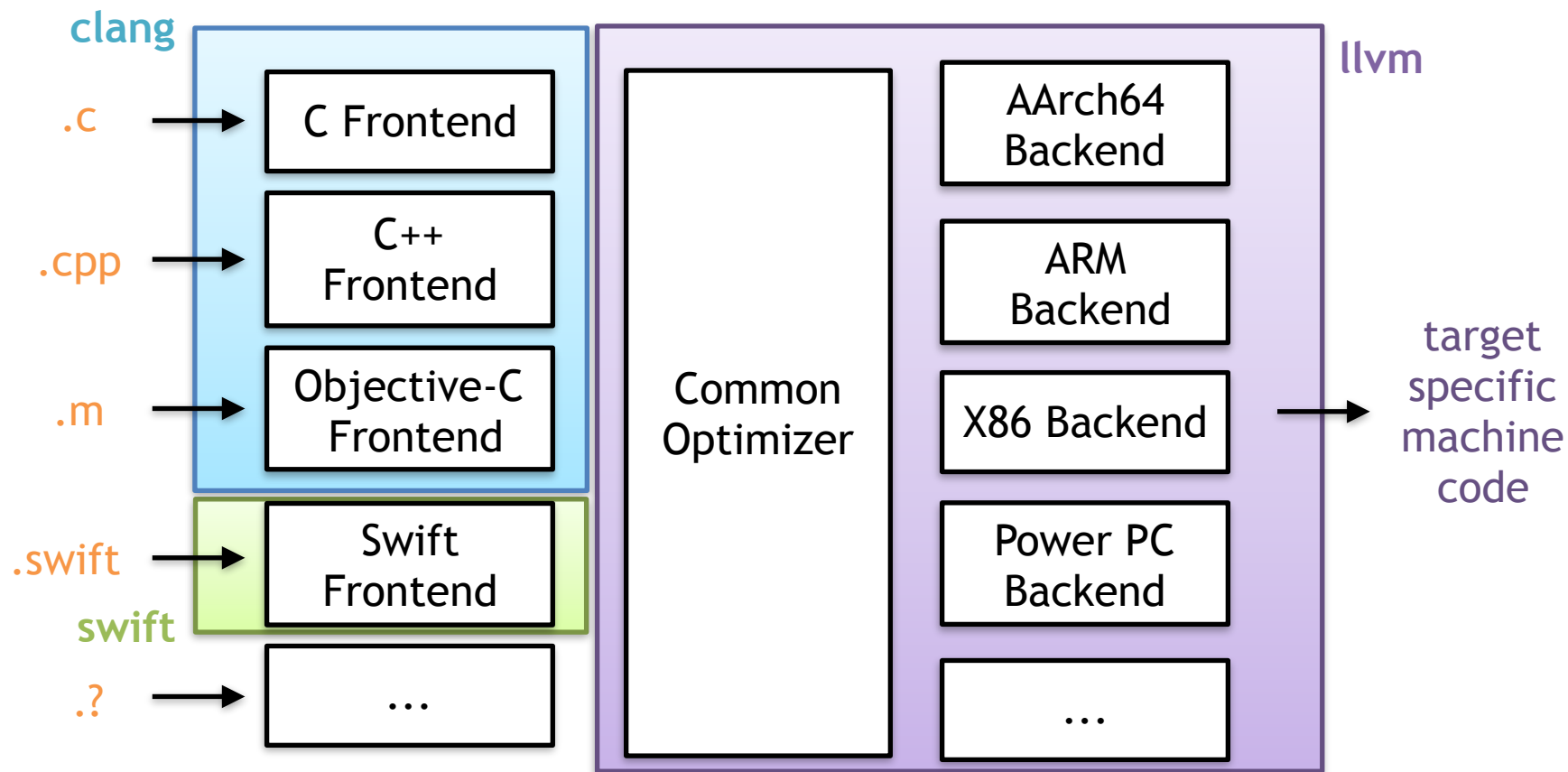- Apple 收回对工具链的控制（lldb, lld, swift...）

Three-Phase 编译器架构

source ⟶ | Frontend | | Optimizer | | Backend | ⟶ machine code

⚠️ **M (Language) * N (Target) = M * N Compilers**

Clang/Swift - LLVM 编译器架构

```
.c     ──▶  ┌──────────────┐
            │  C Frontend  │
            └──────────────┘

.cpp   ──▶  ┌──────────────┐
            │     C++      │
            │   Frontend   │
            └──────────────┘

.m     ──▶  ┌──────────────┐
            │ Objective-C  │
            │   Frontend   │
            └──────────────┘

.swift ──▶  ┌──────────────┐
            │    Swift     │
            │   Frontend   │
            └──────────────┘

.?     ──▶  ┌──────────────┐
            │     ...      │
            └──────────────┘
```

Common Optimizer

```
┌──────────────┐
│   AArch64    │
│   Backend    │
└──────────────┘

┌──────────────┐
│     ARM      │
│   Backend    │
└──────────────┘

┌──────────────┐
│ X86 Backend  │ ──▶ target specific machine code
└──────────────┘

┌──────────────┐
│  Power PC    │
│   Backend    │
└──────────────┘

┌──────────────┐
│     ...      │
└──────────────┘
```

Clang/Swift - LLVM 编译器架构

PS: Swift Frontend 中还包含一层 SIL 及 Optimizer

Clang + LLVM 代码规模

- Total 400W

- C++ 235W

```
github.com/AlDanial/cloc v 1.70   T=472.28 s (36.4 files/s, 8470.9 lines/s)
-------------------------------------------------------------------------------
Language                      files          blank        comment           code
-------------------------------------------------------------------------------
C++                            6507         302077         420778        1627278
C/C++ Header                   3198          99007         169324         406639
C                              2770          36050         148231         121953
Assembly                       1716          56701         173940         111549
Objective C                    1346          14895          70242          50831
HTML                             35           3168            288          26471
Python                          210           5986           7640          23301
Windows Module Definition        68           1521              0          14635
Objective C++                   355           4105          22984          14121
CMake                           437           2019           1462          13144
OCaml                            74           1774           2804           5722
YAML                             74            152           1319           3365
Bourne Shell                     38            349            605           3226
Perl                             14            689            482           3154
Go                               21            401            598           2988
OpenCL                          135            928           1423           2869
Pascal                           11            902           3645           1761
CUDA                             71            657           1782           1358
DOS Batch                        17            139             24            898
Lisp                              9            181            206            810
XML                              42             23              4            686
CSS                               9            144             58            660
JavaScript                        4             79            150            518
C#                                6             46             93            359
JSON                             11             52              0            357
vim script                        8             38             46            283
Bourne Again Shell                4             34             96            227
MSBuild script                    1              0              7            224
make                              7             44             18            135
C Shell                           1             13             14            118
Markdown                          3             45              0             98
Windows Resource File             1             18             11             60
Fortran 95                        1              3              0             18
Windows Message File              1              3              0             13
Rust                              3              6             11             13
INI                               1              1              0              6
NAnt script                       1              0              0              5
Fortran 90                        1              0            260              0
-------------------------------------------------------------------------------
SUM:                          17211         532250        1028545        2439861
-------------------------------------------------------------------------------
```

Swift Frontend 代码规模

- C++ 43W

```
github.com/AlDanial/cloc v 1.70  T=62.38 s (159.6 files/s, 16696.3 lines/s)
-------------------------------------------------------------------------------
Language                        files          blank        comment           code
-------------------------------------------------------------------------------
C++                               533          62726          62671         312494
Swift                            8238          54068         121746         218181
C/C++ Header                      708          25005          36851          85162
Windows Module Definition          49           1420              0          10771
Python                            110           2370           3254           9032
CMake                             168           1050           1215           6547
Markdown                           15           1971              0           6479
Objective C++                      21            843            818           3794
Bourne Again Shell                 12            373            432           2709
HTML                                3            639            141           2409
Objective C                        19            241            136            992
JSON                               35              0              0            743
Lisp                                5            109            226            732
INI                                 1            224              0            647
C                                   7            106             58            552
CSS                                 2             10              8            407
vim script                          8             50             13            271
make                                4             36              5            165
JavaScript                          1             28             19            106
D                                   3             17             12             94
Ruby                                1              7              2             87
Bourne Shell                       10             19             16             74
Perl                                1              7              3             69
Assembly                            1             14             39             30
YAML                                1              0              0             26
MUMPS                               1              1              0              2
-------------------------------------------------------------------------------
SUM:                             9957         151334         227665         662575
-------------------------------------------------------------------------------
```

看 Clang-LLVM 源码的感受

- 代码巨多、需要一定 C++ 基础

- 远离安逸的 Xcode Build System，CMake Ninja 都比较陌生

- 目录明了、分层清晰、风格规范、注释覆盖度高 (~40%)

- 代码结构朴素但合理，均以 library 的形式整合，便于组合与复用

- ✅ Apple 编译器 Clang-LLVM 架构初识
- ➡️ 你的源码是如何一步步成为可执行文件的?
- 我们能用 Clang 做什么有意思的事情?
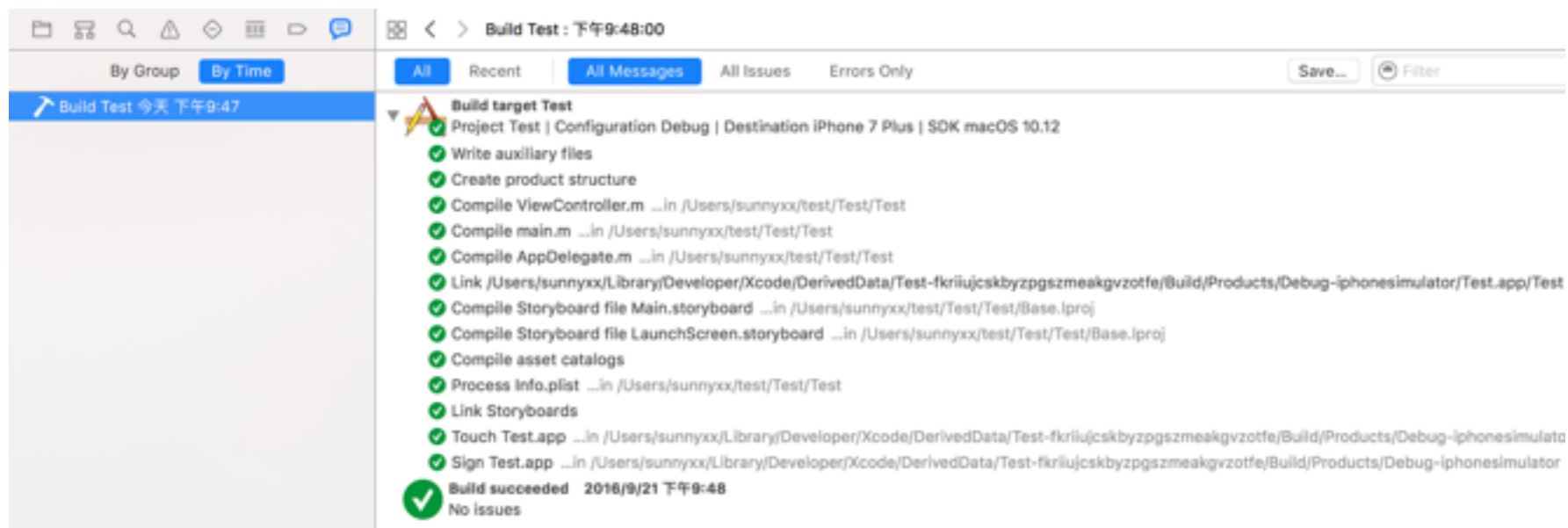
点我点我

当我们按下 Run 之后...

当我们按下 Run 之后...

当我们按下 Run 之后...

```
/Applications/Xcode.app/Contents/Developer/
 Toolchains/XcodeDefault.xctoolchain/usr/bin/
 clang -x objective-c -fobjc-arc ...... main.m
 -o main.o
```

Clang 命令

- Clang 在概念上是编译器前端，同时，在命令行中也作为一个"黑盒"的 Driver

- 封装了编译管线、前端命令、LLVM 命令、Toolchain 命令等，一个 Clang 走天下

- 方便从 gcc 迁移过来

gcc

clang

拆解编译过程

main.m

```objc
#import <Foundation/Foundation.h>

int main() {
    @autoreleasepool {
        id obj = [NSObject new];
        NSLog(@"Hello world: %@", obj);
    }
    return 0;
}
```

1. Preprocess - 预处理

- import 头文件
- macro 展开
- 处理 '#' 打头的预处理指令，如 #if

1. Preprocess - 预处理

$clang -E main.m

```
...
...
...
# 181 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
SDKs/MacOSX10.12.sdk/System/Library/Frameworks/Foundation.framework/Headers/
Foundation.h" 2 3
# 1 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
SDKs/MacOSX10.12.sdk/System/Library/Frameworks/Foundation.framework/Headers/
FoundationLegacySwiftCompatibility.h" 1 3
# 185 "/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
SDKs/MacOSX10.12.sdk/System/Library/Frameworks/Foundation.framework/Headers/
Foundation.h" 2 3
# 6 "main.m" 2

int main() {
    @autoreleasepool {
        id obj = [NSObject new];
        NSLog(@"Hello world: %@", obj);
    }
    return 0;
}
```

1. Preprocess - 预处理

$clang -E **-fmodules** main.m

```
@import Foundation;
int main() {
    @autoreleasepool {
        id obj = [NSObject new];
        NSLog(@"Hello world: %@", obj);
    }
    return 0;
}
```

2. Lexical Analysis - 词法分析

- 词法分析，也作 Lex 或者 Tokenization
- 将预处理过的代码文本转化成 Token 流
- 不校验语义

2. Lexical Analysis - 词法分析

$clang -fmodules -fsyntax-only -Xclang -dump-tokens main.m

```
int 'int' [StartOfLine] Loc=<main.m:7:1>
identifier 'main'  [LeadingSpace] Loc=<main.m:7:5>
l_paren '('     Loc=<main.m:7:9>
r_paren ')'     Loc=<main.m:7:10>
l_brace '{'   [LeadingSpace] Loc=<main.m:7:12>
at '@' [StartOfLine] [LeadingSpace]  Loc=<main.m:8:5>
identifier 'autoreleasepool'      Loc=<main.m:8:6>
l_brace '{'   [LeadingSpace] Loc=<main.m:8:22>
identifier 'id' [StartOfLine] [LeadingSpace]   Loc=<main.m:9:9>
identifier 'obj'    [LeadingSpace] Loc=<main.m:9:12>
equal '=' [LeadingSpace] Loc=<main.m:9:16>
l_square '[' [LeadingSpace] Loc=<main.m:9:18>
identifier 'NSObject'   Loc=<main.m:9:19>
identifier 'new'    [LeadingSpace] Loc=<main.m:9:28>
r_square ']'   Loc=<main.m:9:31>
semi ';'    Loc=<main.m:9:32>
...
```

3. Semantic Analysis - 语法分析

- 语法分析，在 Clang 中由 Parser 和 Sema 两个模块配合完成

- 验证语法是否正确

```
main.m:8:32: error: expected ';' at end of declaration
    id obj = [NSObject new]
                          ^
```

- **根据当前语言的语法，生成语意节点，并将所有节点组合成抽象语法树（AST）**

3. Semantic Analysis - 语法分析

```
$clang -fmodules -fsyntax-only -Xclang -ast-dump main.m
```

```
|-FunctionDecl 0x7fe881035b38 <line:6:1, line:12:1> line:6:5 main 'int ()'
| `-CompoundStmt 0x7fe88133ac28 <col:12, line:12:1>
|   |-ObjCAutoreleasePoolStmt 0x7fe88133abe0 <line:7:5, line:10:5>
|   | `-CompoundStmt 0x7fe88133abb8 <line:7:22, line:10:5>
|   |   |-DeclStmt 0x7fe88133a9e0 <line:8:9, col:32>
|   |   | `-VarDecl 0x7fe88132b728 <col:9, col:31> col:12 used obj 'id':'id' cinit
|   |   |   `-ImplicitCastExpr 0x7fe881327778 <col:18, col:31> 'id':'id' <BitCast>
|   |   |     `-ObjCMessageExpr 0x7fe881327748 <col:18, col:31> 'NSObject *'
selector=new class='NSObject'
|   |   `-CallExpr 0x7fe88133ab50 <line:9:9, col:38> 'void'
|   |     |-ImplicitCastExpr 0x7fe88133ab38 <col:9> 'void (*)(id, ...)'
<FunctionToPointerDecay>
|   |     | `-DeclRefExpr 0x7fe88133a9f8 <col:9> 'void (id, ...)' Function
0x7fe881327798 'NSLog' 'void (id, ...)'
|   |     |-ImplicitCastExpr 0x7fe88133ab88 <col:15, col:16> 'id':'id' <BitCast>
|   |     | `-ObjCStringLiteral 0x7fe88133aa90 <col:15, col:16> 'NSString *'
|   |     |   `-StringLiteral 0x7fe88133aa58 <col:16> 'char [16]' lvalue "Hello world:
%@"
|   |     `-ImplicitCastExpr 0x7fe88133aba0 <col:35> 'id':'id' <LValueToRValue>
|   |       `-DeclRefExpr 0x7fe88133aab0 <col:35> 'id':'id' lvalue Var 0x7fe88132b728
'obj' 'id':'id'
|   `-ReturnStmt 0x7fe88133ac10 <line:11:5, col:12>
|     `-IntegerLiteral 0x7fe88133abf0 <col:12> 'int' 0
```

AST
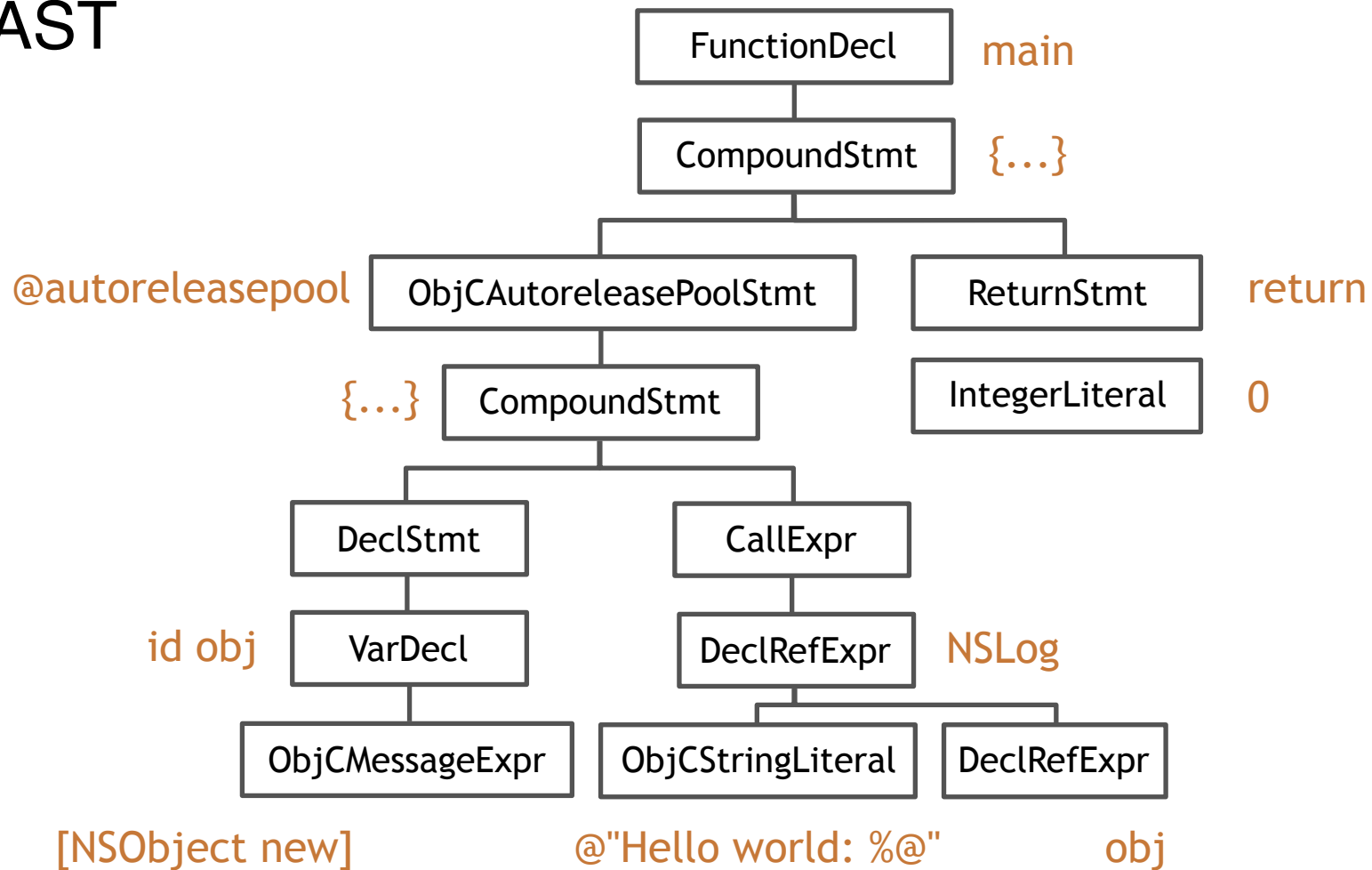
Static Analysis - 静态分析

- 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)
- 预置了常用 Checker
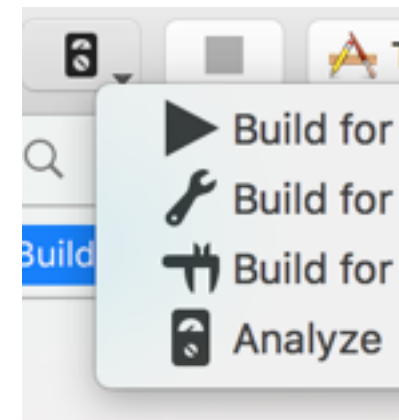
4. CodeGen - IR 代码生成

- CodeGen 负责将语法树从顶至下遍历，翻译成 LLVM IR
- LLVM IR 是 Frontend 的输出，也是 LLVM Backend 的输入，前后端的桥接语言
- **与 Objective-C Runtime 桥接**

4. CodeGen - IR 代码生成 与 Objective-C Runtime 桥接

- Class / Meta Class / Protocol / Category 内存结构生成, 并存放在指定 section 中 (如 Class： _DATA, _objc_classrefs)

- Method / Ivar / Property 内存结构生成

- 组成 method_list / ivar_list / property_list 并填入 Class

4. CodeGen - IR 代码生成   与 Objective-C Runtime 桥接

- Non-Fragile ABI：为每个 Ivar 合成 OBJC_IVAR_$_ 偏移值常量

- 存取 Ivar 的语句 (_ivar = 123; int a = _ivar;) 转写成 base + OBJC_IVAR_$_ 的形式

4. CodeGen - IR 代码生成　与 Objective-C Runtime 桥接

- 将语法树中的 ObjCMessageExpr 翻译成相应版本的 objc_msgSend，对 super 关键字的调用翻译成 objc_msgSendSuper

4. CodeGen - IR 代码生成　与 Objective-C Runtime 桥接

- 根据修饰符 strong / weak / copy / atomic 合成 @property
  自动实现的 setter / getter
- 处理 @synthesize

4. CodeGen - IR 代码生成 与 Objective-C Runtime 桥接

- 生成 block_layout 的数据结构
- 变量的 capture (__block / __weak)
- 生成 _block_invoke 函数

4. CodeGen - IR 代码生成 与 Objective-C Runtime 桥接

- ARC：分析对象引用关系，将 objc_storeStrong/objc_storeWeak 等 ARC 代码插入

- 将 ObjCAutoreleasePoolStmt 转译成 objc_autoreleasePoolPush/Pop

- 实现自动调用 [super dealloc]

- 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"

4. CodeGen - IR 代码生成　与 Objective-C Runtime 桥接

```
namespace {
struct FinishARCDealloc final : EHScopeStack::Cleanup {
  void Emit(CodeGenFunction &CGF, Flags flags) override {
    const ObjCMethodDecl *method = cast<ObjCMethodDecl>(CGF.CurCodeDecl);

    const ObjCImplDecl *impl = cast<ObjCImplDecl>(method->getDeclContext());
    const ObjCInterfaceDecl *iface = impl->getClassInterface();
    if (!iface->getSuperClass()) return;

    bool isCategory = isa<ObjCCategoryImplDecl>(impl);

    // Call [super dealloc] if we have a superclass.
    llvm::Value *self = CGF.LoadObjCSelf();

    CallArgList args;
    CGF.CGM.getObjCRuntime().GenerateMessageSendSuper(CGF, ReturnValueSlot(),
                                                      CGF.getContext().VoidTy,
                                                      method->getSelector(),
                                                      iface,
                                                      isCategory,
                                                      self,
                                                      /*is class msg*/ false,
                                                      args,
                                                      method);
  }
};
}
```

合成 [super dealloc]

CodeGen - IR 代码生成

$clang -S -fobjc-arc -emit-llvm main.m -o main.ll

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8*, align 8
  store i32 0, i32* %1, align 4
  %3 = call i8* @objc_autoreleasePoolPush() #3
  %4 = load %struct._class_t*, %struct._class_t**
@"OBJC_CLASSLIST_REFERENCES_$_", align 8
  %5 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_, align 8, !invariant.load !7
  %6 = bitcast %struct._class_t* %4 to i8*
  %7 = call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8* (i8*,
i8*)*)(i8* %6, i8* %5)
  %8 = bitcast i8* %7 to %0*
  %9 = bitcast %0* %8 to i8*
  store i8* %9, i8** %2, align 8
  %10 = load i8*, i8** %2, align 8
  notail call void (i8*, ...) @NSLog(i8* bitcast
(%struct.__NSConstantString_tag* @_unnamed_cfstring_ to i8*), i8* %10)
  call void @objc_storeStrong(i8** %2, i8* null) #3
  call void @objc_autoreleasePoolPop(i8* %3)
  ret i32 0
}
```

Optimize - 优化 IR

$clang **-O3** -S -fobjc-arc -emit-llvm main.m -o main.ll

```
define i32 @main() #0 {
  %1 = tail call i8* @objc_autoreleasePoolPush() #3
  %2 = load i8*, i8** bitcast (%struct._class_t**
@"OBJC_CLASSLIST_REFERENCES_$_" to i8**), align 8
  %3 = load i8*, i8** @OBJC_SELECTOR_REFERENCES_, align 8, !invariant.load
!7
  %4 = tail call i8* bitcast (i8* (i8*, i8*, ...)* @objc_msgSend to i8*
(i8*, i8*)*)(i8* %2, i8* %3), !clang.arc.no_objc_arc_exceptions !7
  %5 = bitcast i8* %4 to %0*
  %6 = bitcast %0* %5 to i8*
  notail call void (i8*, ...) @NSLog(i8* bitcast
(%struct.__NSConstantString_tag* @_unnamed_cfstring_ to i8*), i8* %4), !
clang.arc.no_objc_arc_exceptions !7
  tail call void @objc_release(i8* %6) #3, !clang.imprecise_release !7
  tail call void @objc_autoreleasePoolPop(i8* %1) #3, !
clang.arc.no_objc_arc_exceptions !7
  ret i32 0
}
```

LLVM Bitcode - 生成字节码

$clang -emit-llvm -c main.m -o main.bc
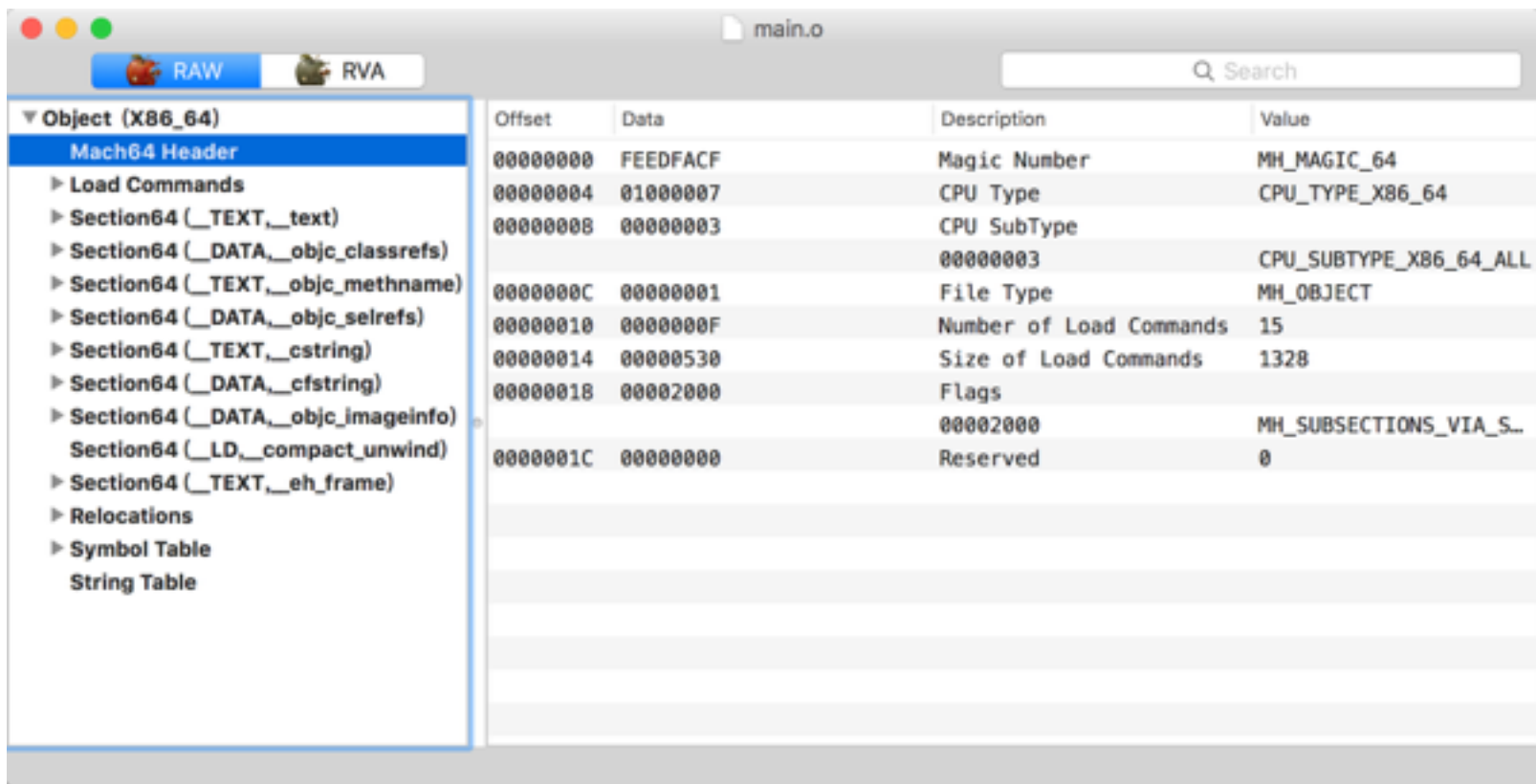
Assemble - 生成 Target 相关汇编

$clang -S -fobjc-arc main.m -o **main.s**

```
_main:                                          ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
Ltmp0:
    .cfi_def_cfa_offset 16
Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $32, %rsp
    movl    $0, -4(%rbp)
    callq   _objc_autoreleasePoolPush
    movq    L_OBJC_CLASSLIST_REFERENCES_$_(%rip), %rcx
    movq    L_OBJC_SELECTOR_REFERENCES_(%rip), %rsi
    movq    %rcx, %rdi
    movq    %rax, -24(%rbp)          ## 8-byte Spill
    callq   _objc_msgSend
    leaq    L__unnamed_cfstring_(%rip), %rcx
    movq    %rax, -16(%rbp)
    movq    -16(%rbp), %rsi
    movq    %rcx, %rdi
    movb    $0, %al
    callq   _NSLog
    leaq    -16(%rbp), %rdi
    xorl    %edx, %edx
    movl    %edx, %esi
    callq   _objc_storeStrong
    movq    -24(%rbp), %rdi          ## 8-byte Reload
    callq   _objc_autoreleasePoolPop
    xorl    %eax, %eax
    addq    $32, %rsp
    popq    %rbp
    retq
```

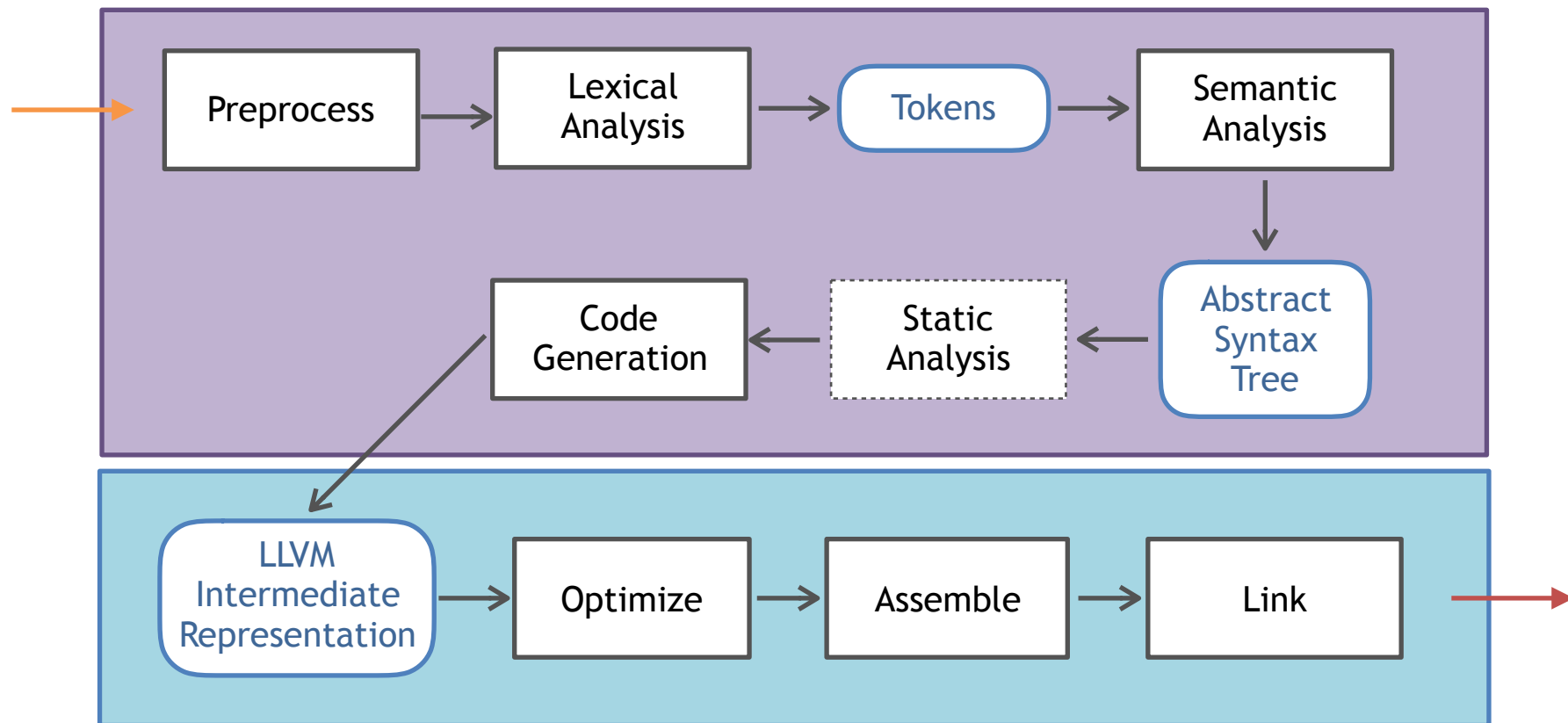# Assemble - 生成 Target 相关 Object (Mach-O)

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

Link 生成 Executable

$clang main.m -o main

$./main

```
main[13595:2214602] Hello world: <NSObject: 0x7f9b01506700>
```

总结：Clang-LLVM 下，一个源文件的编译过程

- ✅ Apple 编译器 Clang-LLVM 架构初识
- ✅ 你的源码是如何一步步成为可执行文件的?
- ➡️ 我们能用 Clang 做什么有意思的事情?

我们能在 Clang 上做什么?

- LibClang
- LibTooling
- ClangPlugin

LibClang

- 😊 C API 来访问 Clang 的上层能力，如获取 Tokens、遍历语法树、代码补全、获取诊断信息
- 😊 API 稳定，不受 Clang 源码更新影响
- 😡 只有上层的语法树访问，不能获取到全部信息

LibClang - 如何使用

- 使用原始 C API
- 脚本语言：使用官方提供的 python binding 或开源的 node-js / ruby binding
- Objective-C：开源库 ClangKit

LibClang - Demo

```objc
@interface Sark : NSObject

@property (nonatomic, strong) id passWord;
@property (nonatomic, strong) id nickName;
@property (nonatomic, strong) id netWorking;
@property (nonatomic, strong) id suuny;
@property (nonatomic, strong) id backgrond;

@end
```


用 LibClang 的 Python Binding 实现一个 Property Name Linter

LibClang - Demo

```python
import enchant, difflib
from clang.cindex import Index

if __name__ == '__main__':
    index = clang.cindex.Index.create()
    tu = index.parse(sys.argv[1])
    d = enchant.Dict("en_US")
    for c in tu.cursor.walk_preorder():
        if c and c.spelling:
            if (c.kind == clang.cindex.CursorKind.OBJC_PROPERTY_DECL):
                if (not d.check(c.spelling)):
                    best = None
                    best_ratio = 0
                    suggestions = set(d.suggest(c.spelling))
                    for sug in suggestions:
                        tmp = difflib.SequenceMatcher(None, c.spelling.lower(), sug).ratio()
                        if tmp > best_ratio:
                            best = sug
                            best_ratio = tmp
                    print "typo: " + c.spelling + ", do you mean: " + best + "?";
```

LibClang - Demo

```
$python property-linter.py main.m
```

```
typo: passWord, do you mean: password?
typo: nickName, do you mean: nickname?
typo: netWorking, do you mean: networking?
typo: suuny, do you mean: sunny?
typo: backgrond, do you mean: background?
```

LibTooling

- 😊 对语法树有完全的控制权
- 😊 **可作为一个 standalone 命令单独的使用，如 clang-format**
- 😠 需要使用 C++ 且对 Clang 源码熟悉

LibTooling - Demo

```objc
@interface Sark : NSObject
@property (nonatomic, copy) NSString *name;
- (void)becomeGay;
@end
```

实现一个简易 Objective-C -> Swift 源码转换器

LibTooling - Demo

```
|-ObjCInterfaceDecl 0x7ff94185dca0 <line:7:1, line:10:2> line:7:12 Sark
| |-super ObjCInterface 0x7ff9411a4608 'NSObject'
| |-ObjCPropertyDecl 0x7ff9411a24e0 <line:8:1, col:39> col:39 name 'NSString
*' readwrite copy nonatomic
| |-ObjCMethodDecl 0x7ff9411a2600 <line:9:1, col:18> col:1 - becomeGay
'void'
| |-ObjCMethodDecl 0x7ff9411a2688 <line:8:39> col:39 implicit - name
'NSString *'
| `-ObjCMethodDecl 0x7ff9411a2710 <col:39> col:39 implicit - setName: 'void'
|   `-ParmVarDecl 0x7ff9411a2798 <col:39> col:39 name 'NSString *'
```

创建 RecursiveASTVisitor，在 AST 中重写感兴趣节点的 Visit 方法

LibTooling - Demo

`$objc2swift test.m -- -fsyntax-only -fmodules`

```
class Sark: NSObject {
    var name: NSString?
    func becomeGay() {
    }
}
```

ClangPlugin

- 😊 对语法树有完全的控制权

- 😊 **作为插件注入到编译流程中，可以影响 build 和决定编译过程**

- 😡 需要使用 C++ 且对 Clang 源码熟悉

ClangPlugin - Demo

```
14
15   @interface AppDelegate : UIResponder <UIApplicationDelegate>
16                                              ⚠ 缺少 Objective-C 类名前缀
17   @property (strong, nonatomic) UIWindow *window;
18
19   @end
20
21
```

可以嵌入 Xcode 的 Linter，提供可识别的诊断信息

ClangPlugin - Demo

```
bool VisitObjCInterfaceDecl(clang::ObjCInterfaceDecl *D) {
    const clang::SourceManager &SM = Context->getSourceManager();
    clang::FullSourceLoc loc = Context->getFullLoc(D->getLocStart());
    if (!SM.isInSystemHeader(loc)) {
        std::string name = D->getName();
        clang::DiagnosticsEngine &DE = *Diagnostics;
        if (std::islower(name[0]) || std::islower(name[1])) {
            unsigned int id =
DE.getCustomDiagID(clang::DiagnosticsEngine::Warning, "缺少 Objective-C
类名前缀");
            DE.Report(loc.getLocWithOffset(11), id);
        }
    }
    return true;
}
```

ClangPlugin - Demo

```
24  @implementation ViewController
25
26  - (void)viewDidLoad {
27      [super viewDidLoad];
28  sunny:
29      goto sunny;                                    很遗憾，你的代码不符合公司价值观
30  }
31
32
33  - (void)didReceiveMemoryWarning {
34      [super didReceiveMemoryWarning];
```

ClangPlugin - Demo

```
24  @implementation ViewController
25
26  - (void)viewDidLoad {
27      [super viewDidLoad];
28  sunny:
29      请到 HR 处办理离职手续 sunny;          很遗憾，你的代码不符合公司价值观
30  }
```

很遗憾，你的代码不符合公司价值观
Fix-it Replace "goto" with "请到 HR...

```
33  - (void)didReceiveMemoryWarning {
34      [super didReceiveMemoryWarning];
35      // Dispose of any resources that can be recreated.
36  }
```

Like a Boss

- ✅ Apple 编译器 Clang-LLVM 架构初识
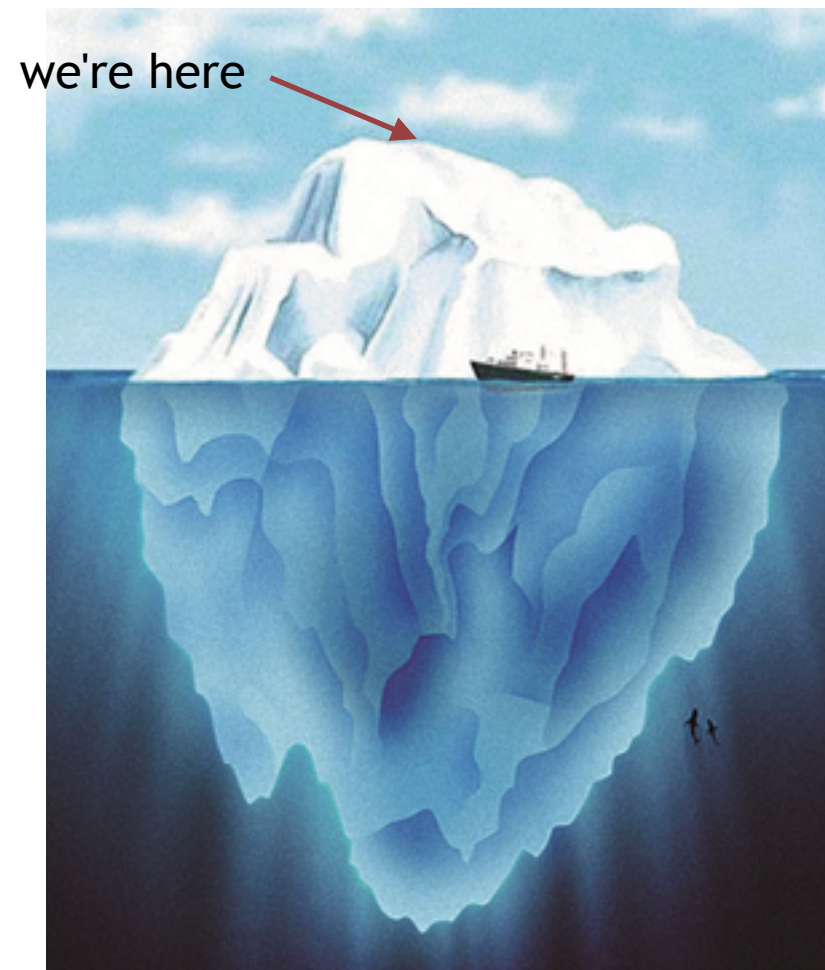- ✅ 你的源码是如何一步步成为可执行文件的?
- ✅ 我们能用 Clang 做什么有意思的事情?

Clang-LLVM 相关资料

- http://clang.llvm.org/docs/index.html

- http://blog.llvm.org/

- https://www.objc.io/issues/6-build-tools/compiler/

- http://llvm.org/docs/tutorial/index.html

- https://github.com/loarabia/Clang-tutorial

- http://lowlevelbits.org/getting-started-with-llvm/clang-on-os-x/

- https://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-i-introduction/

- http://szelei.me/code-generator/

- 《Getting Started with LLVM Core Libraries》

- 《LLVM Cookbook》

- 问：编译器可以编译程序，但编译器本身也是个程序，那它一定是由更早的编译器编译而成的，那…最早的一个编译器是哪儿来的？

手写机器码？

we're here