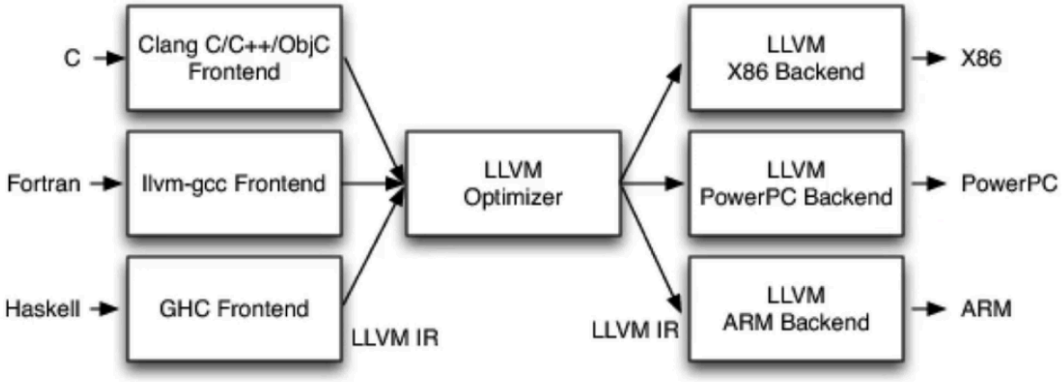
LLVM

传统编译器架构

- Frontend：前端
  - 词法分析，语法分析，语义分析，生成中间代码Optimizer
- Optimizer：优化器
  - 中间代码优化，更快，更小
- Backend：后端
  - 生成机器码
  - 如果运行在iOS上，则生成ARM架构的代码，如果运行在Win上，则生成x86，x64架构的代码

LLVM编译器架构



- 同样分为前端，优化器，后端
- 后段负责生成不同平台，不同架构的机器码
- 不同编程语言对应的前端不一样
- 不同的前端后端使用统一的中间代码LLVM IR
- 如果需要支持一种新的编程语言，那么只需要实现一个新的前端
- 如果需要支持一种新的硬件设备，那么只需要实现一个新的后端
- 优化阶段是一个通用的阶段，它针对的是统一的LLVM IR，不论是支持新的编程语言，还是支持新的硬件设备，都不需要对优化阶段进行改变。
- llvm现在被作为实现各种静态和运行时编译语言的通用基础结构。（java，python，Ruby）

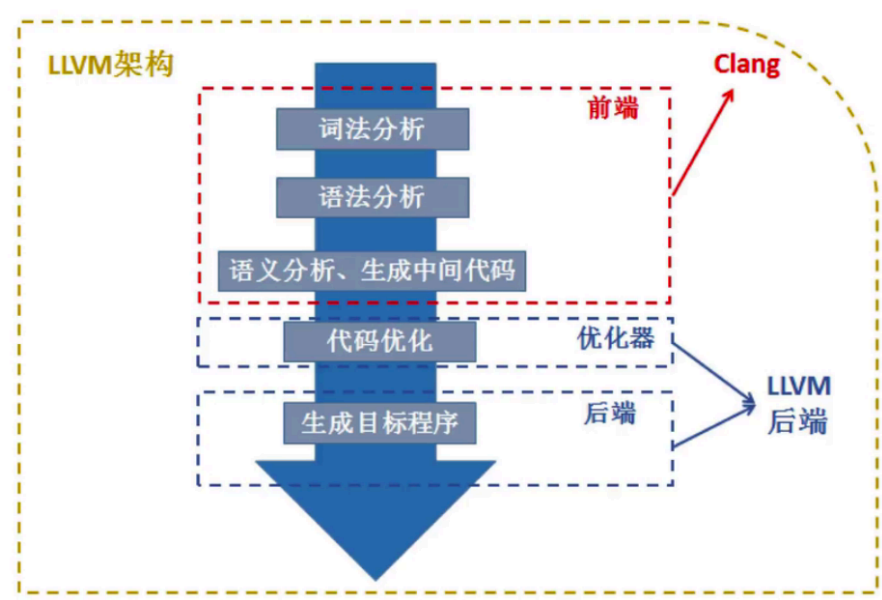
Clang

- 什么是Clang
  - LLVM项目的一个子项目

- 基于LLVM架构的c/c++/OC编译器前端

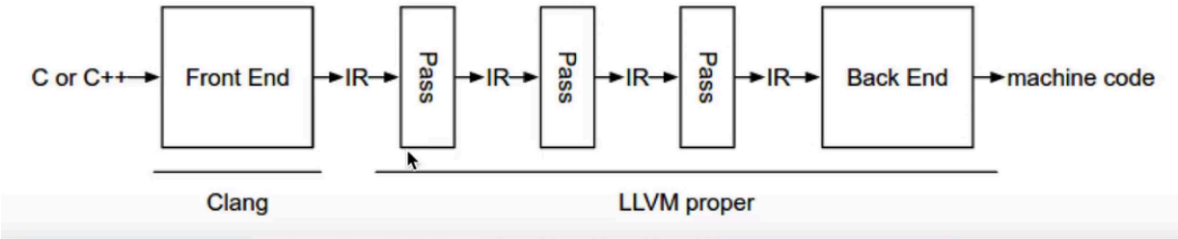
- 相比较GCC，Clang具有哪些优点
    - 编译速度快：在某些平台上，Clang的编译速度显著的快过GCC
    - 占用内存小：Clang生成的AST（语法树）所占用的内存是GCC的五分之一左右
    - 模块化设计：Clang采用基于库的模块化设计，易于IDE集成及其他用途的重用
    - 诊断信息可读性强：在编译过程中，Clang创建并保存了大量详细的元数据，有利于调试
    - 设计清晰简单，容易理解，易于扩展增强

Clang与LLVM



- 广义的LLVM
    - 整个LLVM架构

- 狭义的LLVM
    - LLVM后端（代码优化，目标代码生成）



- 左边是编程语言，最右边是机器码
- 首先通过编译器前端Clang，生成中间代码IR
- 中间代码要经过一系列的优化，优化用的是Pass
- 中间代码的优化可以自己编写，可以插入一个Pass

OC源文件的编译过程

■ 命令行查看编译的过程：$ clang -ccc-print-phases main.m

```
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, "x86_64", {5}, image
```

0. 找到main.m文件

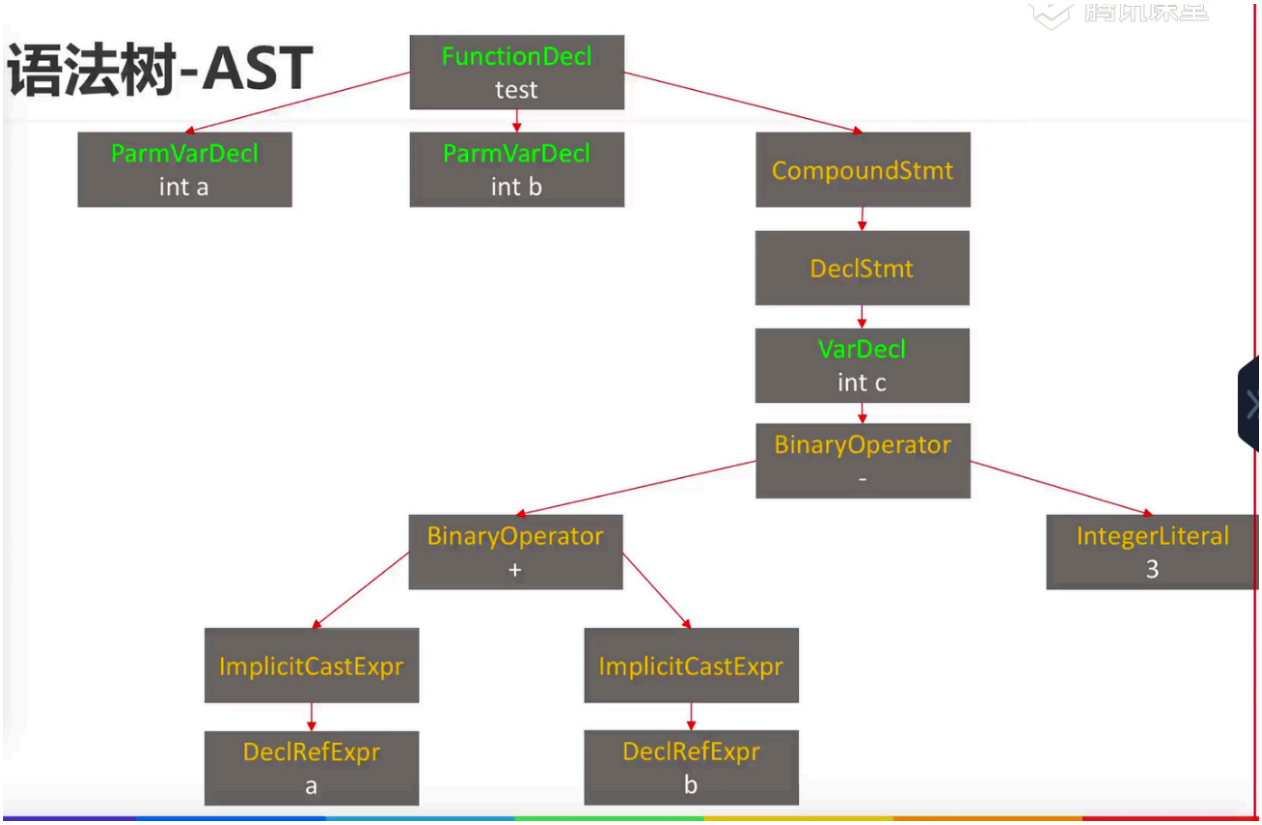
1. 预处理器，把include，import，宏定义替换掉
    - 把include中的文件拷贝到当前.m文件中
    - 查看preprocessor(预处理)的结果： $ clang -E main.m
2. 编译器编译为中间代码IR
    - 词法分析，生成Token： $ clang -fmodules -E -Xclang -dump-tokens main.m
        - ```
          ~/Desktop/Test/Test mj$ clang -fmodules -E -Xclang -dump-tokens main.m
          annot_module_include '#include <stdio.h>

          #define AGE 40

          int main(int argc, const char * argv[]) {
              int a = 10;
              int b = 20;
              int c = a + b + A'          Loc=<main.m:9:1>
          int 'int'        [StartOfLine]  Loc=<main.m:13:1>
          identifier 'main'        [LeadingSpace] Loc=<main.m:13:5>
          l_paren '('             Loc=<main.m:13:9>
          int 'int'               Loc=<main.m:13:10>
          identifier 'argc'        [LeadingSpace] Loc=<main.m:13:14>
          comma ','               Loc=<main.m:13:18>
          const 'const'    [LeadingSpace] Loc=<main.m:13:20>
          char 'char'      [LeadingSpace] Loc=<main.m:13:26>
          star '*'         [LeadingSpace] Loc=<main.m:13:31>
          identifier 'argv'        [LeadingSpace] Loc=<main.m:13:33>
          l_square '['            Loc=<main.m:13:37>
          r_square ']'            Loc=<main.m:13:38>
          r_paren ')'             Loc=<main.m:13:39>
          l_brace '{'      [LeadingSpace] Loc=<main.m:13:41>
          ```
    - 语法分析，生成语法树（AST，Abstract Syntax Tree）： $ clang -fmodules -fsyntax-only -Xclang -ast-dump main.m
        - ```
          1
          2 void test(int a, int b) {
          3     int c = a + b - 3;
          4 }
          5
          ```
        - ```
          -FunctionDecl 0x7fa372038e00 <line:21:1, line:24:1> line:21:6 test 'void (int, int)'
           |-ParmVarDecl 0x7fa372038c88 <col:11, col:15> col:15 used a 'int'
           |-ParmVarDecl 0x7fa372038d00 <col:18, col:22> col:22 used b 'int'
           `-CompoundStmt 0x7fa372039040 <line:22:1, line:24:1>
             `-DeclStmt 0x7fa372039028 <line:23:5, col:22>
               `-VarDecl 0x7fa372038ed8 <col:5, col:21> col:9 c 'int' cinit
                 `-BinaryOperator 0x7fa372039000 <col:13, col:21> 'int' '-'
                   |-BinaryOperator 0x7fa372038fb8 <col:13, col:17> 'int' '+'
                   | |-ImplicitCastExpr 0x7fa372038f88 <col:13> 'int' <LValueToRValue>
                   | | `-DeclRefExpr 0x7fa372038f38 <col:13> 'int' lvalue ParmVar 0x7fa372038c88 'a' 'int'
                   | `-ImplicitCastExpr 0x7fa372038fa0 <col:17> 'int' <LValueToRValue>
                   |   `-DeclRefExpr 0x7fa372038f60 <col:17> 'int' lvalue ParmVar 0x7fa372038d00 'b' 'int'
                   `-IntegerLiteral 0x7fa372038fe0 <col:21> 'int' 4
          -<undeserialized declarations>
          ```
3. 交给后端生成目标代码
4. 目标代码
          ```

5. 链接动态库，静态库
6. 变成适合某个架构的代码

语法树 AST



中间代码 LLVM IR

- IR有3种表示形式（但本质是等价的，就好比水可以有气体，液体，固体3种形式）
  - text：便于阅读的文本格式，类似于汇编语言，拓展名.II $ clang -S -emit-llvm main.m
  - memory：内存格式
  - bitcode：二进制格式，拓展名.bc $ clang -c -emit-llvm main.m

```
; Function Attrs: noinline nounwind optnone ssp uwtable
define void @test(i32, i32) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = add nsw i32 %6, %7
  %9 = sub nsw i32 %8, 3
  store i32 %9, i32* %5, align 4
  ret void
}
```

安装Clang


小码哥教育 SEEMYGO
源码下载
■ 下载LLVM
□ $ git clone https://git.llvm.org/git/llvm.git/
□ 大小 648.2 M，仅供参考
■ 下载clang
□ $ cd llvm/tools
□ $ git clone https://git.llvm.org/git/clang.git/
□ 大小 240.6 M，仅供参考


小码哥教育 SEEMYGO
源码编译
■ 安装cmake和ninja（先安装brew，https://brew.sh/）
□ $ brew install cmake
□ $ brew install ninja
■ ninja如果安装失败，可以直接从github获取release版放入【/usr/local/bin】中
□ https://github.com/ninja-build/ninja/releases
■ 在LLVM源码同级目录下新建一个【llvm_build】目录（最终会在【llvm_build】目录下生成【build.nin
□ $ cd llvm_build
□ $ cmake -G Ninja ../llvm -DCMAKE_INSTALL_PREFIX=LLVM的安装路径
□ 更多cmake相关选项，可以参考： https://llvm.org/docs/CMake.html
■ 依次执行编译、安装指令
□ $ ninja
□ 编译完毕后，【llvm_build】目录大概 21.05 G（仅供参考）
□ $ ninja install
□ 安装完毕后，安装目录大概 11.92 G（仅供参考）


小码哥教育 SEEMYGO
应用与实践
■ libclang、libTooling
□ 官方参考：https://clang.llvm.org/docs/Tooling.html
□ 应用：语法树分析、语言转换等
■ Clang插件开发
□ 官方参考
✓ https://clang.llvm.org/docs/ClangPlugins.html
✓ https://clang.llvm.org/docs/ExternalClangExamples.html
✓ https://clang.llvm.org/docs/RAVFrontendAction.html
□ 应用：代码检查（命名规范、代码规范）等

Clang 插件开发