LLVM

什么是LLVM

官网: https://llvm.org/

The LLVM Project is a collection of modular and reusable compile and toolchain technologies

LLVM 项目是模块化，可重用的编译器以及工具链技术的集合

美国计算机协会 (ACM) 将其2012 年软件系统奖项颁给了LLVM，之前曾经获得此奖项的软件和技术包括:Java、Apache、 Mosaic、the World Wide Web、Smalltalk、UNIX、Eclipse等等

创始人:Chris Lattner，亦是Swift之父

有些文章把LLVM当做Low Level Virtual Machine(低级虚拟机)的缩写简称，官方描述如下

The name “LLVM” itself is not an acronym; it is the full name of the project.

“LLVM”这个名称本身不是首字母缩略词; 它是项目的全名。

`LLVM` 包含三部分，分别是 `LLVM suite` 、 `Clang` 和 `Test Suite` 。

1. `LLVM suite` ，LLVM 套件，它包含了 LLVM 所需要的所有工具、库和头文件，一个汇编器、解释器、位码分析器和位码优化器，还包含了可用于测试 LLVM 的工具和 clang 前端的基本回归测试。
2. `Clang` ，俗称为 Clang 前端，该组件将 `C` ， `C++` ， `Objective C` ，和 `Objective C++` 代码编译到 LLVM 的位码中。一旦编译到 LLVM 位代码中，就可以使用 LLVM 套件中的工具来操作程序。
3. `Test Suite` ，测试套件，这是一个可选的工具，它是一套带有测试工具的程序，可用于进一步测试 LLVM 的功能和性能。

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

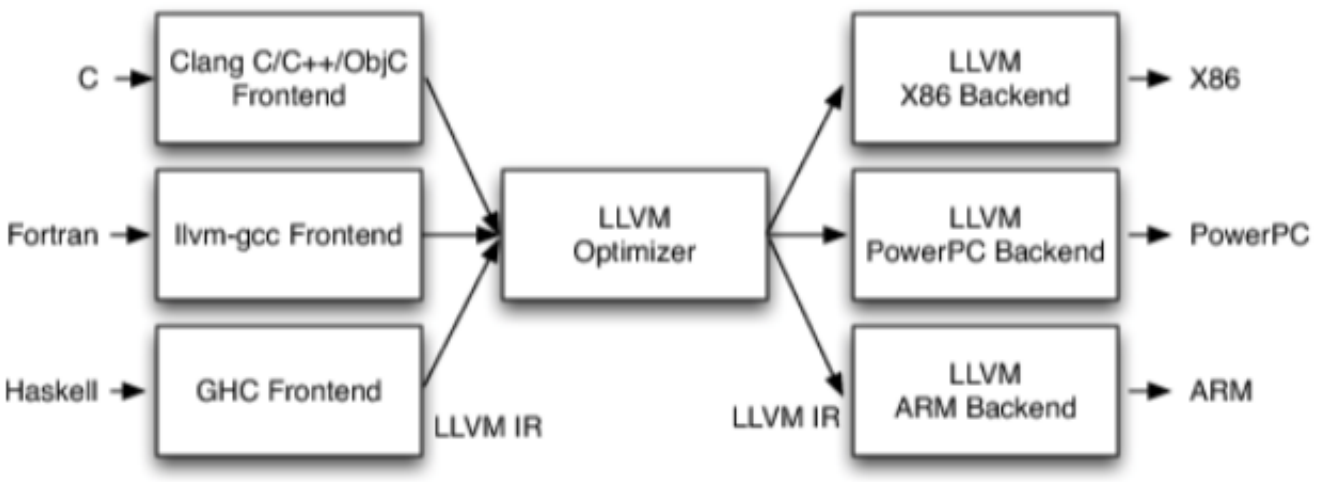
- **Optimizer:优化器**

中间代码优化 （执行速度更快，体积更小）

- **Backend:后端**

  生成机器码（x86,PC,ARM）

LLVM架构



不同的前端后端使用统一的中间代码LLVM Intermediate Representation (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做修改

相比之下，GCC的前端和后端没分得太开，前端后端耦合在了一起。所以GCC为了支持一门新的语言，或者为了支持一个新的目标平台，就 变得特别困难

Xcode编译器发展简史

- 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

Clang

LLVM项目的一个子项目

基于LLVM架构的C/C++/Objective-C编译器前端。

官网：https://clang.llvm.org/

相比于GCC，Clang具有如下优点

编译速度快:在某些平台上，Clang的编译速度显著的快过GCC(Debug模式下编译OC速度比GCC快3倍)

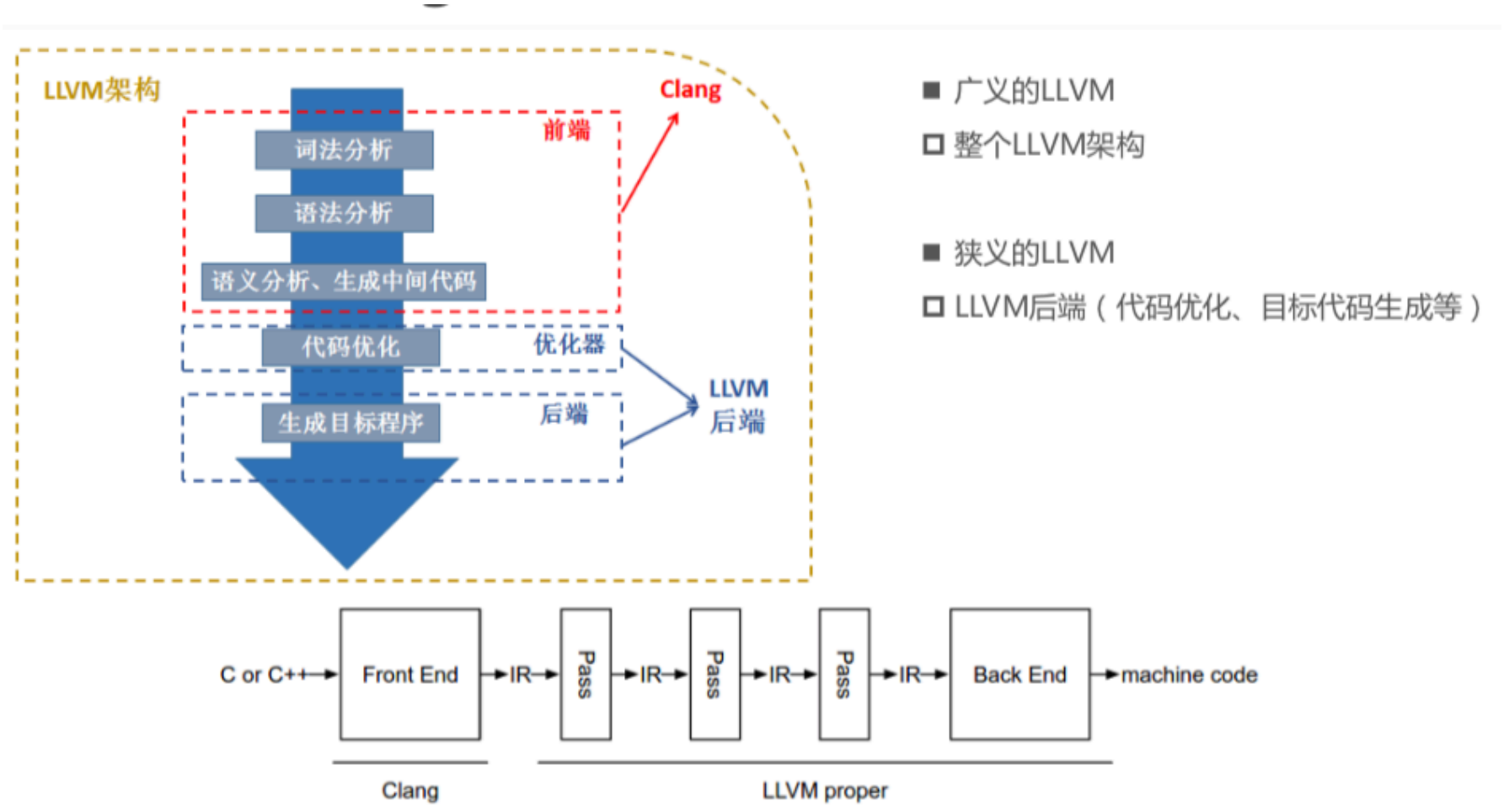
占用内存小:Clang生成的AST所占用的内存是GCC的五分之一左右

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

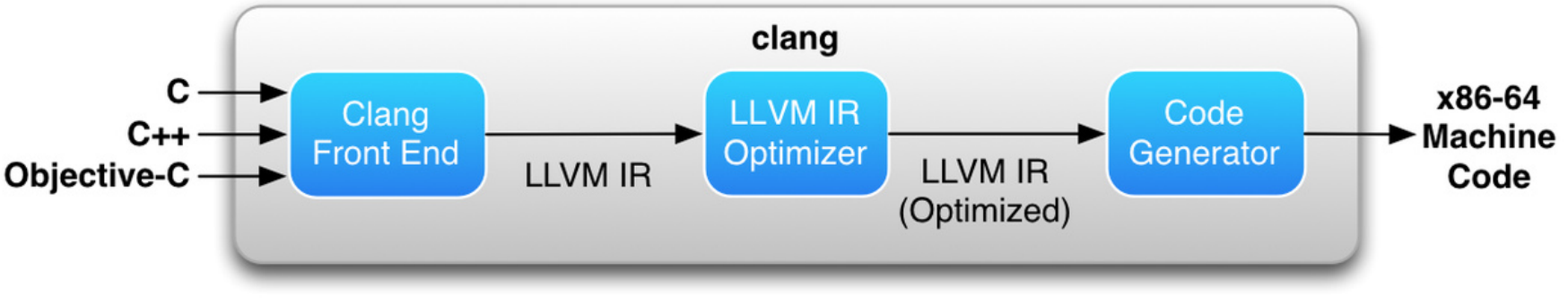
诊断信息可读性强:在编译过程中，Clang 创建并保留了大量详细的元数据 (metadata)，有利于调试和错误报告

设计清晰简单，容易理解，易于扩展增强

Clang 与LLVM 关系



LLVM整体架构，前端用的是clang，广义的LLVM是指整个LLVM架构，一般狭义的LLVM指的是LLVM后端（包含代码优化和目标代码生成）。



`Objective-C` 与 `swift` 都采用 `Clang` 作为编译器前端，编译器前端主要进行语法分析、语义分析、生成中间代码，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。

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，根据不同的系统架构生成不同的机器码。

`C++`， `Objective-C` 都是编译语言。编译语言在执行的时候，必须先通过编译器生成机器码。

源代码（c/c++）经过clang—> 中间代码(经过一系列的优化，优化用的是Pass) —> 机器码

pass

Pass就是LLVM系统转化和优化的工作的一个节点，当然我们也可以写一个这样的节点去做一些自己的优化工作或者其它的操作.多个pass一起完成了LLVM的优化与代码转换工作,每个pass都会完成指定的优化工作。

LLVM的Pass架构具有良好的可重用性与控制性。它允许我们嵌入自己编写的pass或者关闭一些默认提供的pass。同时，单个pass的开发以及调试也很独立，所以不必担心破坏整个LLVM的源码结构，只需要学会操作LLVM IR即可实现自己的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

```
$ 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，处理include、import、宏定义

2.编译器编译，编译成ir中间代码

3.后端，生成目标代码

4.汇编

5.链接其他动态库静态库

6.编译成适合某个架构的代码

查看preprocessor(预处理)的结果:$ clang -E main.m

```
# 1 "main.m"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 353 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.m" 2
.
.
.
  int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"Hello, World!");
}
return 0;
}
```

词法分析

词法分析，生成Token: $ clang -fmodules -E -Xclang -dump-tokens main.m

将代码切成一个个 token，比如大小括号，等于号还有字符串等。是计算机科学中将字符序列转换为标记序列的过程。

举一个🌰

```
void test(int a, int b){
    int c = a + b - 3;
}
```

```
void 'void'  [StartOfLine]  Loc=<main.m:18:1>
identifier 'test'    [LeadingSpace] Loc=<main.m:18:6>
l_paren '('     Loc=<main.m:18:10>
int 'int'       Loc=<main.m:18:11>
identifier 'a'   [LeadingSpace] Loc=<main.m:18:15>
comma ','       Loc=<main.m:18:16>
int 'int'    [LeadingSpace] Loc=<main.m:18:18>
identifier 'b'   [LeadingSpace] Loc=<main.m:18:22>
r_paren ')'     Loc=<main.m:18:23>
l_brace '{'     Loc=<main.m:18:24>
int 'int'    [StartOfLine] [LeadingSpace]   Loc=<main.m:19:5>
identifier 'c'   [LeadingSpace] Loc=<main.m:19:9>
equal '='    [LeadingSpace] Loc=<main.m:19:11>
identifier 'a'   [LeadingSpace] Loc=<main.m:19:13>
plus '+'     [LeadingSpace] Loc=<main.m:19:15>
identifier 'b'   [LeadingSpace] Loc=<main.m:19:17>
minus '-'    [LeadingSpace] Loc=<main.m:19:19>
numeric_constant '3'     [LeadingSpace] Loc=<main.m:19:21>
semi ';'        Loc=<main.m:19:22>
r_brace '}'  [StartOfLine]  Loc=<main.m:20:1>
eof ''      Loc=<main.m:20:2>
```

可以看出，词法分析的时候，将上面的代码拆分一个个token，后面数字表示某一行的第几个字符。

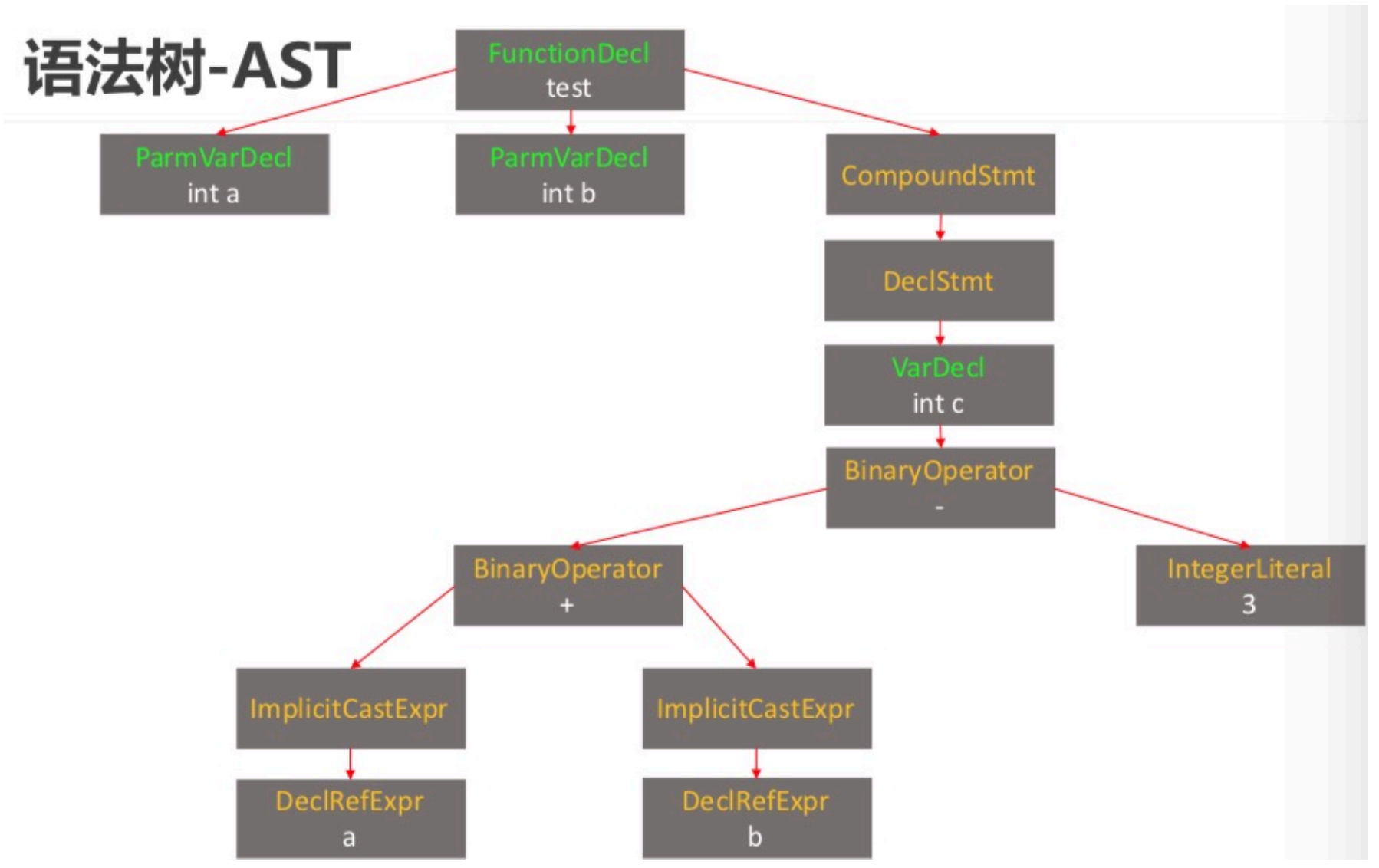
语法树-AST

语法分析，验证语法是否正确，然后将所有节点组成抽象语法树 AST 。由 Clang 中 Parser 和 Sema 配合完成。

生成语法树(AST，Abstract Syntax Tree): $ clang -fmodules -fsyntax-only -Xclang -ast-dump main.m

```
|-FunctionDecl 0x7fa1439f5630 <line:18:1, line:20:1> line:18:6 test 'void (int, int)'
| |-ParmVarDecl 0x7fa1439f54b0 <col:11, col:15> col:15 used a 'int'
| |-ParmVarDecl 0x7fa1439f5528 <col:18, col:22> col:22 used b 'int'
| `-CompoundStmt 0x7fa142167c88 <col:24, line:20:1>
|   `-DeclStmt 0x7fa142167c70 <line:19:5, col:22>
|     `-VarDecl 0x7fa1439f5708 <col:5, col:21> col:9 c 'int' cinit
|       `-BinaryOperator 0x7fa142167c48 <col:13, col:21> 'int' '-'
|         |-BinaryOperator 0x7fa142167c00 <col:13, col:17> 'int' '+'
|         | |-ImplicitCastExpr 0x7fa1439f57b8 <col:13> 'int' <LValueToRValue>
|         | | `-DeclRefExpr 0x7fa1439f5768 <col:13> 'int' lvalue ParmVar 0x7fa1439f54b0 'a' 'int'
|         | `-ImplicitCastExpr 0x7fa1439f57d0 <col:17> 'int' <LValueToRValue>
|         |   `-DeclRefExpr 0x7fa1439f5790 <col:17> 'int' lvalue ParmVar 0x7fa1439f5528 'b' 'int'
|         `-IntegerLiteral 0x7fa142167c28 <col:21> 'int' 3

`-<undeserialized declarations>
```



静态分析(Static Analysis)

使用它来表示用于分析源代码以便自动发现错误

LLVM IR

生成中间代码 IR，CodeGen 会负责将语法树自顶向下遍历逐步翻译成 LLVM IR，IR 是编译过程的前端的输出，后端的输入。

LLVM IR有3种表示形式（本质是等价的）

text:便于阅读的文本格式，类似于汇编语言，拓展名.ll， $ 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

text格式方便阅读

```
; Function Attrs: noinline nounwind optnone ssp uwtable
define void @test(i32, i32) #2 {
  %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
}
```

```
IR基本语法
注释以分号；开头
全局标识符以@开头，局部标识符以%开头
alloca，在当前函数栈帧中分配内存
i32，32bit，4个字节的意思
align，内存对齐
store，写入数据
load，读取数据
```

官方语法参考 https://llvm.org/docs/LangRef.html

应用与实践

源码下载

```
下载LLVM
$ git clone https://git.llvm.org/git/llvm.git/

下载clang
$ cd llvm/tools
$ git clone https://git.llvm.org/git/clang.git/

备注：clang是llvm的子项目，但是它们的源码是分开的，我们需要将clang放在llvm/tools目录下。
```

源码编译

这里我们在终端敲出的clang是xcode默认内置clang编译器，我们自己要进行LLVM开发的话，需要通过LLVM的源码

编译成属于我们自己的clang编译器，

```
首先安装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（专门用于存放编译的目标代码）

$ cd llvm_build
$ cmake -G Ninja ../llvm -DCMAKE_INSTALL_PREFIX=LLVM的安装路径
最终会在【llvm_build】目录下生成【build.ninja】
生成build.ninja，就表示编译成功，-DCMAKE_INSTALL_PREFIX 表示编译好的东西放在指定的路径(
本机为/Users/xxx/xxx/LLVM/llvm_release)
，-D表示参数。
```

更多cmake相关选项，可以参考 https://llvm.org/docs/CMake.html

接下来依次执行编译、安装指令

```
$ ninja
编译完毕后，  【llvm_build】目录大概 21.05 G
$ ninja install
```

另一种方式是通过Xcode编译，生成Xcode项目再进行编译，但是速度很慢。

方法如下： 在llvm同级目录下新建一个【llvm_xcode】目录

```
$ cd llvm_xcode
$ cmake -G Xcode ../llvm
```

Clang 三大件分别是 `LibClang` 、 `Clang Plugins` 和 `LibTooling`

LibClang：

libclang 供了一个相对较小的 API，它将用于解析源代码的工具暴露给抽象语法树（AST），加载已经解析的 AST，遍历 AST，将物理源位置与 AST 内的元素相关联。

libclang 是一个稳定的高级 C 语言接口，隔离了编译器底层的复杂设计，拥有更强的 Clang 版本兼容性，以及更好的多语言支持能力，对于大多数分析 AST 的场景来说，libclang 是一个很好入手的选择。

优点

1. 可以使用 C++ 之外的语言与 Clang 交互。
2. 稳定的交互接口和向后兼容。
3. 强大的高级抽象，比如用光标迭代 AST，并且不用学习 Clang AST 的所有细节。

缺点

1. 不能完全控制 Clang AST。

Clang Plugins：

Clang Plugin 允许你在编译过程中对 AST 执行其他操作。Clang Plugin 是动态库，由编译器在运行时加载，并且它们很容易集成到构建环境中。

LibTooling：

LibTooling 是一个独立的库，它提供了完整的参数解析方案,允许使用者很方便地搭建属于你自己的编译器前端工。

它的优点与缺点一样明显，它基于 C++ 接口，读起来晦涩难懂，但是提供给使用者远比 libclang 强大全面的 AST 解析和控制能力，同时由于它与 Clang 的内核过于接近导致它的版本兼容能力比 libclang 差得多，Clang 的变动很容易影响到 LibTooling。

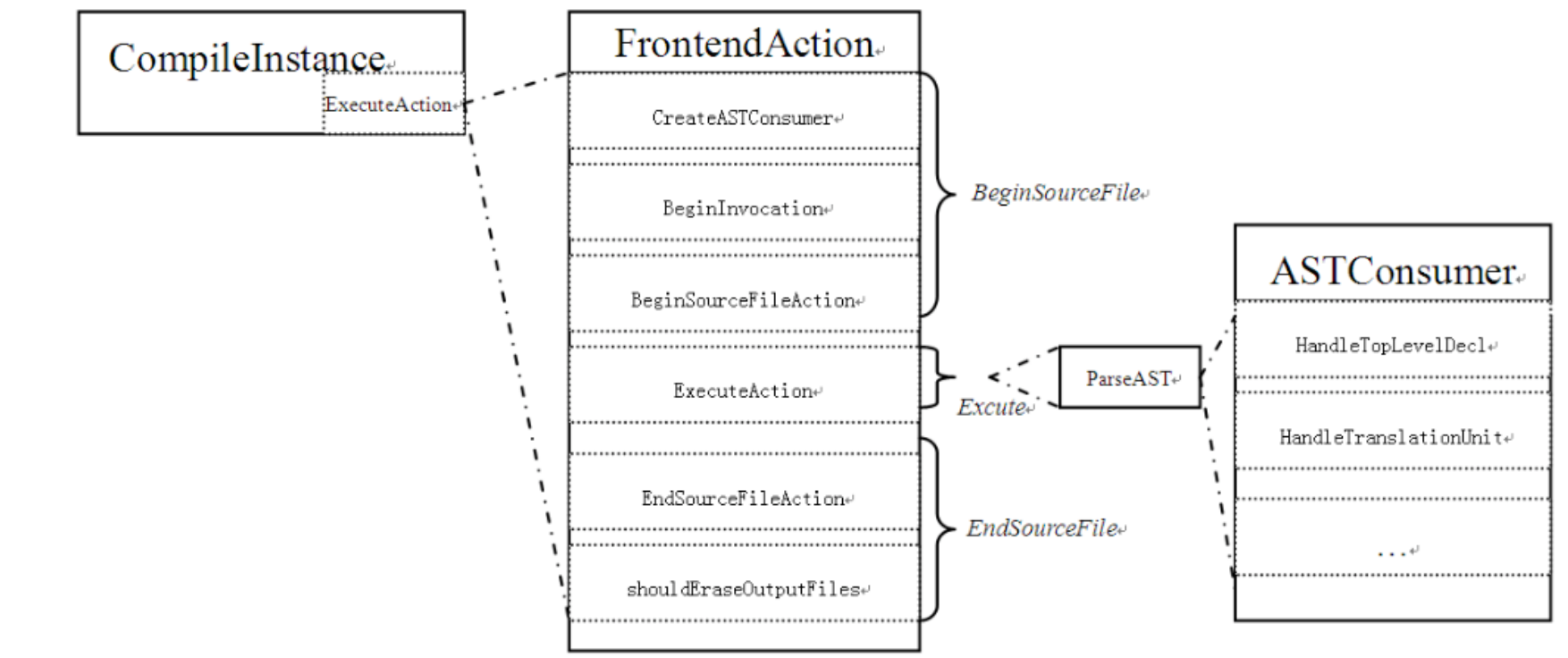
编写插件

1. 在 /llvm/tools/clang/tools 目录下新建插件目录。
2. 修改 /llvm/tools/clang/tools 目录下的 CMakeLists.txt 文文件，新增 add_clang_subdirectory(xxPlugin) (复制最后一行修改下括号内容)。
3. 在 xxPlugin 目录下新建一个名为 xxPlugin.cpp 的文文件。
4. 在 xxPlugin 目录下新建一个名为 CMakeLists.txt 的文件，内容

```
add_llvm_library(xxPlugin MODULE xxPlugin.cpp PLUGIN_TOOL clang)
```

5 目录文文件创建完成之后，利用 cmake 重新生生成一一下 Xcode 项目。在 llvm_xcode 目录下执行 $ cmake -G Xcode ../llvm 。

6 插件源代码在 Xcode 项目中的 Loadable modules 目录下可以找到，这样就可以直接在 Xcode 里编写插件代码。



上图是 `Clang Plugin` 执行的过程，分别有 `CompilerInstance` 、 `FrontendAction` 和 `ASTConsumer` 。

CompilerInstance：是一个编译器实例，综合了一个 Compiler 需要的 objects，如 Preprocessor，ASTContext（真正保存 AST 内容的类），DiagnosticsEngine，TargetInfo 等。

FrontendAction：是一个基于 Consumer 的抽象语法树(Abstract Syntax Tree/AST)前端 Action 抽象基类，对于 Plugin，我们可以继承至系统专门提供的 `PluginASTAction` 来实现我们自定义的 Action，我们重载 `CreateASTConsumer()` 函数返回自定义的 `Consumer` ，来读取 AST Nodes。

```
unique_ptr <ASTConsumer> CreateASTConsumer(CompilerInstance &CI, StringRef InFile) {
    return unique_ptr <QTASTConsumer> (new QTASTConsumer);
}
```

ASTConsumer：是一个读取抽象语法树的抽象基类，我们可以重载下面两个函数：

- `HandleTopLevelDecl()`：解析顶级的声明（像全局变量，函数定义等）的时候被调用。
- `HandleTranslationUnit()`：在整个文件都解析完后会被调用。

除了上面提到的这几个类，还有两个比较重要的类，分别是`RecursiveASTVisitor`和`MatchFinder`。

RecursiveASTVisitor：是一个特别有用的类，使用它可以访问任意类型的 AST 节点。

- `VisitStmt()`：分析表达式。
- `VisitDecl()`：分析所有声明。

MatchFinder：是一个 AST 节点的查找过滤匹配器，可以使用`addMatcher`函数去匹配自己关注的 AST 节点。

验证：我们可以在终端中使用命令的方式进行验证

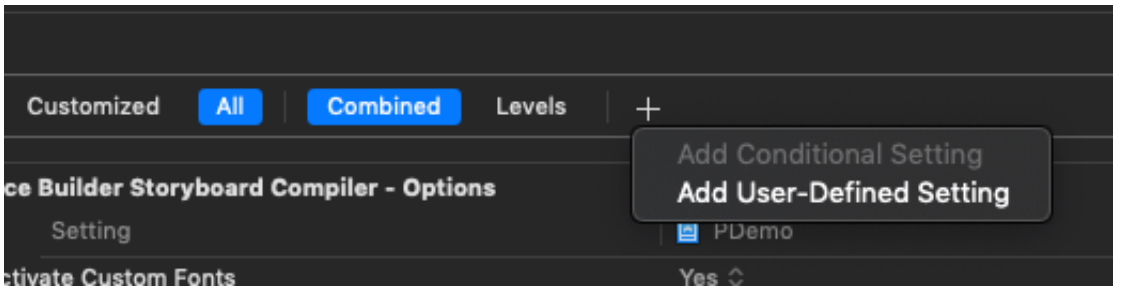
```
译的clang文件路径 -isysroot
ions/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator12.1.sdk/
-load -Xclang 插件(.dylib)路径 -Xclang -add-plugin -Xclang 插件名 -c 资源文件(.h或者.m)
```

打开需要加载插件的`Xcode`项目，在`Build Settings`栏目中的`OTHER_CFLAGS`添加上如下内容：

```
-Xclang -load -Xclang (.dylib)动态库路径 -Xclang -add-plugin -Xclang 插件名字 （namespace 的名字，名字不对则无法使用插件）
```

设置编译器：

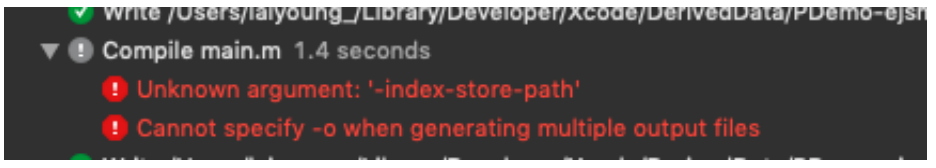
在`Build Settings`栏目中新增两项用户定义的设置



分别是`CC`和`CXX`。

`CC`对应的是自己编译的`clang`的绝对路径，`CXX`对应的是自己编译的`clang++`的绝对路径。

如果👆的步骤都确认无误之后，在编译的时候如果遇到了下图这种错误



则可以在`Build Settings`栏目中搜索`index`，将`Enable Index-Wihle-Building Functionality`的`Default`改为`NO`。

Clang 的开源给了我们更多的操作空间，我们可以利用clang的API针对语法树(AST)进行相应的分析和处理，进一步完善我们的需求，也能更好地提升我们代码的规范和质量。

参考

推荐文章：

- 深入研究Clang
- LLVM每日谈

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

1、https://clang.llvm.org/docs/ClangPlugins.html

2、https://clang.llvm.org/docs/ExternalClangExamples.html

3、https://clang.llvm.org/docs/RAVFrontendAction.html

应用:代码检查(命名规范、代码规范)等

Pass开发

官方参考:https://llvm.org/docs/WritingAnLLVMPass.html

应用:代码优化、代码混淆等

开发新的编程语言

1、 https://llvm-tutorial-cn.readthedocs.io/en/latest/index.html

2、https://kaleidoscope-llvm-tutorial-zh-cn.readthedocs.io/zh_CN/latest/