LLVM

Definition

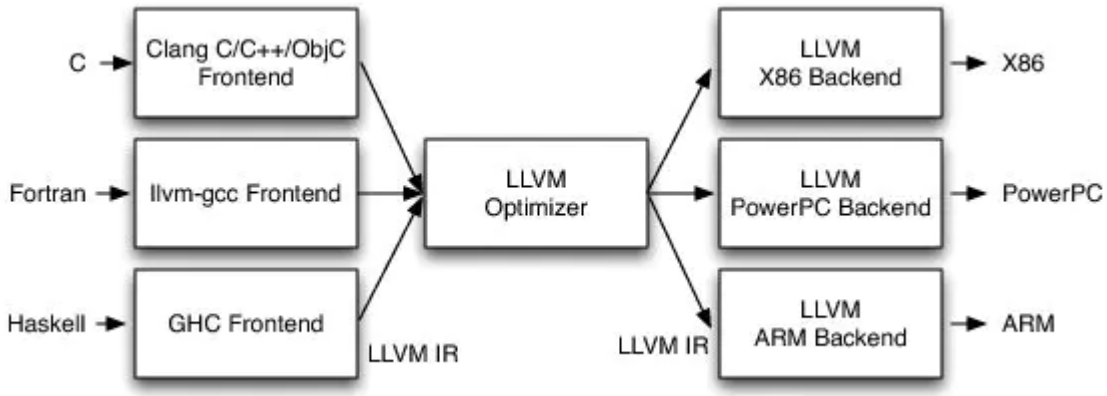
一套模块化可重用的编译器和工具链的集合

包括但不限于前端、后端、优化器、汇编器、链接器、libc++标准库、Compiler-RT和JIT引擎

LLVM IR

传统静态编译器分为三个阶段：

前端，中端(优化)，后端



为了支持一种新的编程语言，只需要重新实现一个新的前端，以及支持一种新的目标架构就只需要重新实现一个后端即可。

LLVM IR（intermediate representation）

本质上是一种和编程语言与目标机器架构无关的通用中间表示。可以理解为是一组类RISC的虚拟指令集

- 以三地址码形式组织指令
- 假设有无数的寄存器可用

基础架构组成

- 前端：

把程序源代码转换为LLVM IR的编译器步骤。包括词法分析，语法分析，语义分析组件，LLVM IR生成器等

Clang执行了所有与前端相关的步骤，并提供了一个插件接口和一个单独的静态分析工具

- 中间表示

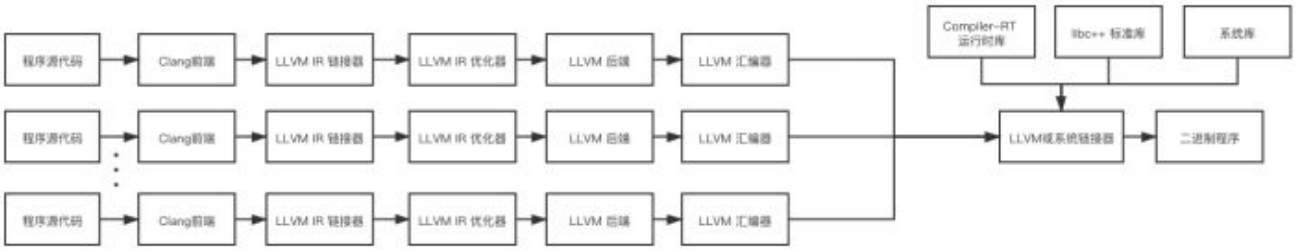
LLVM IR可以以可读文本代码和二进制代码两种形式呈现。LLVM库中提供了对IR进行构造、组装和拆卸的接口。LLVM优化器也在IR上进行操作，并在IR上进行了大部分优化

- 后端

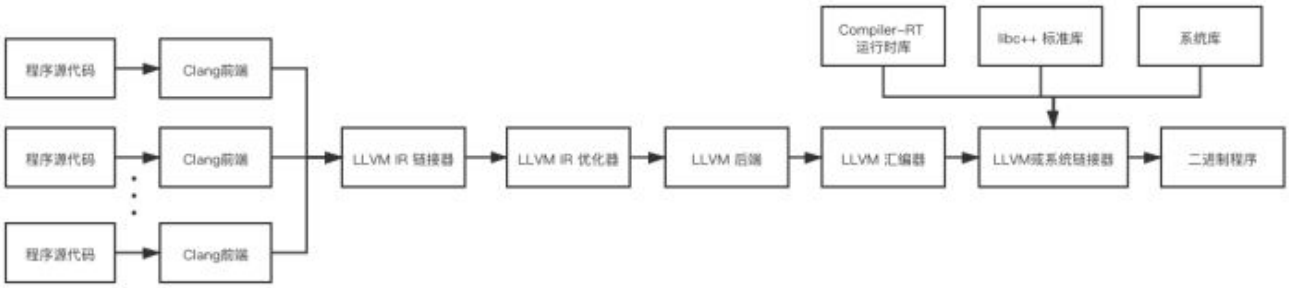
负责汇编码或机器码生成的步骤，将LLVM IR转换为特定机器架构的汇编代码或而二进制代码，包括寄存器分配、循环转换、窥视孔优化器、特定机器架构的优化和转换等步骤

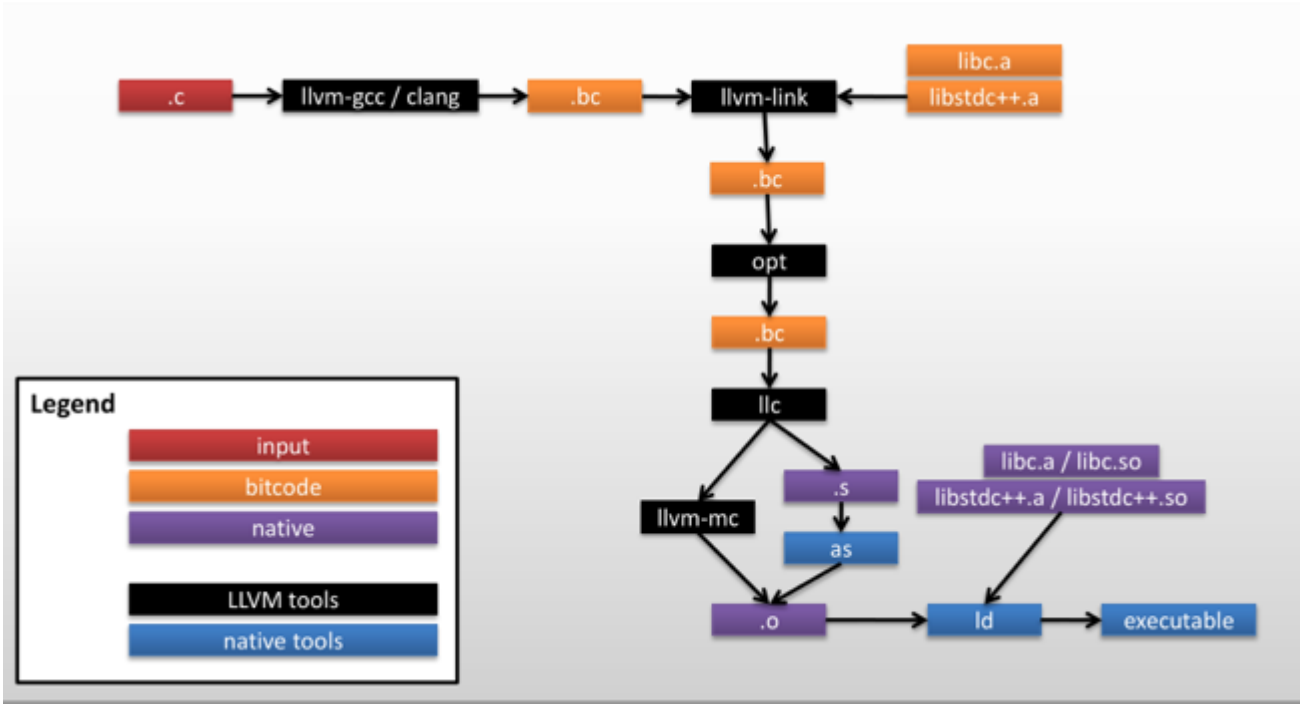
两种连接方式：

源代码内部链接由LLVM或者系统连接器完成：



内部链接由LLVM IR连接器完成：（开启链接优化时候采用）





更详细流程：

- C语言代码经过了Clang前端生成LLVM IR。

LLVM IR通常是以.bc结尾，即bitcode形式保存的。是序列化的数据用于保存在磁盘上。

而LLVM IR其实有三种表示形式：

1 .bc

2 .ll 人类可读形式保存

3 .内存表示形式

- 如果在llvm-link之前，使用了O2等优化，还会经历一个优化阶段如内联，死代码消除等。
- 然后进入llvm-link阶段，其实就是链接，把多个.bc文件合并成一个.bc文件并且做链接时优化。
- 然后经过llvm-optimizer再经历一次优化
- 然后就可以进入代码生成阶段，即后端
- 图中走的是llc，即代码生成。而其实还可以走一个过程叫lli，是解释执行LLVM IR

那其实llc可以理解为一个编译器，负责把LLVM编译生成汇编文件.s，然后再用GNU的as（系统的汇编器）从而产生目标文件object，即.o文件。

- 或者可以走llvm-mc，这是一个llvm本身的集成汇编器，可以直接到o
- 有了.o，就可以走GNU 的链接器ld，生成可执行文件了

LLVM的中间数据结构

- LLVM IR 一套虚拟指令集
- AST 前端语法分析器/语义分析器的产出数据结构
- DAG LLVM IR 在转化成特定机器架构的汇编代码时候，先转换为DAG形式，然后再转换回三地址码形式以进行指令调度
- MCModule 为了实现汇编器和链接器，LLVM使用MCModule类将程序表示保存在对象文件的上下文中(.o)

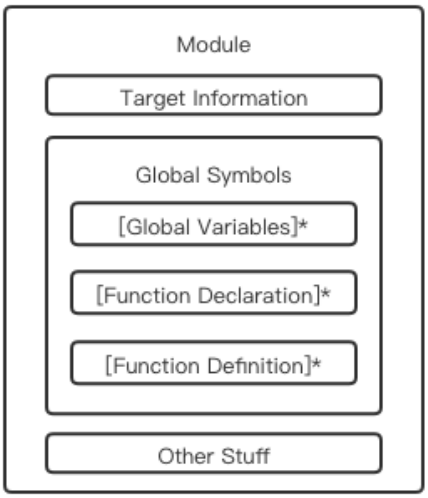
不同编译阶段的中间数据结构有以下两种存在方式：

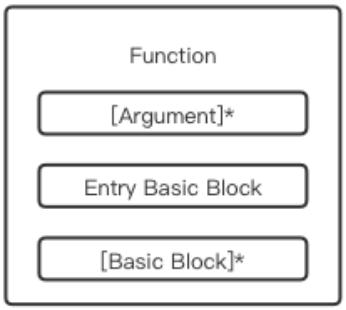
- 内存中：需要编译驱动程序的帮助，将一个阶段的输出数据结构作为下一个阶段的输入数据结构
- 文件中：独立命令之间多数以文件为媒介进行交互，比如汇编器与链接器通过可重定向的`.o`对象文件进行交互

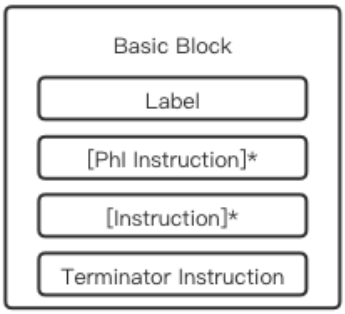
LLVM IR结构

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.14.0"

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}

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

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "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,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 9.0.0 (tags/RELEASE_900/final)"}
```

分号是注释 第7-11行是main函数 第14-23行是add函数 第25行是函数属性 第27-32行是模块元信息







- Module

是一个LLVM IR的顶层容器。对应于前端的每个翻译单元，每个模块由目标机器信息，全局符号以及元信息组成。

- Function

  编程语言中的函数，包括函数签名和若干个基本块，函数内的第一个基本块叫做入口基本块。

- BasicBlock（基本块）

  是一组顺序执行的指令集合，只有一个入口和一个出口，非头尾指令执行时不会违背顺序跳转到其他指令上去。每个基本块最后一条指令一般是跳转指令（跳转到其它基本块上去），函数内最后一个基本块的最后条指令是函数返回指令

- Instruction (指令)

  是LLVM IR中的最小可执行单位，每一条指令都单占一行

Target Information

```
; ModuleID = 'add.c'
source_filename = "add.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.14.0"
```

- `ModuleID`：编译器用于区分不同模块的ID
- `source_filename`：源文件名
- `target datalayout`：目标机器架构数据布局
- `target triple`：用于描述目标机器信息的一个元组，一般形式是`<architecture>-<vendor>-<system>[-extra-info]`

需要关注的是`target datalayout`，它由`-`分隔的一列规格组成

- `e`：内存存储模式为小端模式
- `m:o`：目标文件的格式是Mach格式
- `i64:64`：64位整数的对齐方式是64位，即8字节对齐
- `f80:128`：80位扩展精度浮点数的对齐方式是128位，即16字节对齐
- `n8:16:32:64`：整型数据有8位的、16位的、32位的和64位的
- `S128`：128位栈自然对齐

标识符与变量

- 全局标识符 以@开头
- 局部标识符 以%开头

局部标识符的分类方案：

按照是否命名分类：

- 命名局部变量：顾名思义，比如`%tmp`

- 未命名局部变量：以带前缀的无符号数字值表示，比如`%1`、`%2`，按顺序编号，函数参数、未命名基本块都会增加计数

按照分配方式分类：

- 寄存器分配的局部变量：此类局部变量多采用`%1 = some value`形式进行分配，一般是接受指令返回结果的局部变量，
- 栈分配的局部变量：使用`alloca`指令在栈帧上分配的局部变量，比如`%2 = alloca i32`，`%2`也是个指针，访问或存储时必须使用`load`和`store`指令

```
define i32 @main() {
    %1 = alloca i32, align 4
    %tmp = alloca i32, align 4
    store i32 1, i32* %1, align 4
    store i64 2, i32* %tmp, align 8
    %2 = add nsw i32 %1, %tmp
    %result = add nsw i32 %1, %2
}
```

其中`%1`是栈分配的未命名局部变量，`%tmp`是栈分配的命名局部变量，`%2`是寄存器分配的未命名局部变量，`%result`是寄存器分配的命名局部变量

ref

https://zhuanlan.zhihu.com/p/102250532