**基于LLVM的编译原理和工具调研**

****

**小组成员：王若晗2052534，黄晨冉2050664**

**韩嘉睿2051973，莫益萌2051828**

**页数：15**

**课程：编译原理**

**完成日期：2021.12**

**基于LLVM的编译器发展和工具调研**

1. **引言**

随着编译技术的不断发展，编译器工具也经历了相当大的变革过程。从IBM公司研发出Fortran，打响了编译器开发的第一枪；到GNU通过自由软件和开放许可证开发出允许互相合作的开源编译器GCC，点燃了创新的火把，也为编译器发展铺平道路；再到如今涌现出更灵活、性能更佳的LLVM和颇受瞩目的AI编译器，拓宽了编译器的应用边界。编译器研究者们一直在不断努力着，探索着编译器发展的更多可能性，为开发者提供更广泛、效率更高的技术支持。LLVM的主要发起人Chris Lattner说：“令人兴奋的是，编译器或者编程语言工程师会迎来一个崭新的时代：过去和现在都有无数的技术诞生，这些技术正在改变世界，有幸参与这场变革浪潮非常令人激动。”

LLVM是一个模块化和可重用的编译器和工具链技术的集合。它提供了通用的中间代码层，实现了前后端分离，具有很强的模块性，易于开发者们合作与创新，同时也降低了编译器的开发难度。因此，LLVM成为催生出多种语言和新的前端，一跃成为了当下最领先的开源软件技术之一。针对其发展、设计、编译过程和常用工具，本文将从不同方面对LLVM进行介绍。

本文第一部分对编译器发展进行概述，从Fortran，编译技术发展，GNU和GCC，LLVM四个方面阐述了编译器工具大致的发展历程；第二部分着重研究了LLVM的设计，分析了整体架构设计和LLVM独有的IR设计两个方面；第三部分对LLVM的编译过程进行了调研，从前端、优化层、后端三个方面对LLVM编译流程进行介绍，并详细研究了所涉及到的技术细节；第四部分介绍了LLVM相关工具，主要研究了LLVM的常用工具以及其重要的前端Clang。

1. **编译器发展概述**

## 2.1 第一个编译器：Fortran

如果想了解计算机架构和驱动计算机架构的编译器方面的现状，我们应首先了解清楚编译器在这七十年间发生的巨大改变。

让我们先把目光回到1957年的第一个编译器IBM Fortran。

在上世纪五十年代,几乎所有的程序都是用机器语言或汇编语言编写的。当时,程序员把生成一个有效程序看作是一项复杂而又富有创造力的艺术活动，他们精力主要都花在克服当时计算机引发的各种障碍上：如没有索引寄存器，没有嵌入式浮点操作等。虽然有为程序员提供的"自动编程"系统，但这些系统使用起来开销过大，通常会因此降低5-10倍的机器运行速度。同时，由于编程门槛过高，花在程序员上的投资已接近计算机本身的价值，而且计算机的使用时间有1/4都是花在程序的排错上，因此开发工作变得既复杂又艰苦。

基于上述两种因素，1953年末，IBM公司通过了John Backus的提议，成立了一个研究小组，去研究开发一种更加经济有效的编程方法，以改变当时IBM生产的704计算机编程效率低下的状况。功夫不负有心人，这个研究小组在在1954年中期开发出了一种有相当功能和灵活性的编程语言，被称作Fortran。虽然初衷只是为了IBM内部研究使用，但它的中期报告引起了许多IBM客户的兴趣，因此IBM也顺水推舟，将Fortran语言和编译器安装到了704计算机上，同时为客户提供了编程指南和编译程序。而没过多久，其它供应商也开发出了面向其它机器的Fortran编译器，这样，Fortran就拥有了机器代码无法想象的可移植性。

从这⼀刻起，编译器开始迅猛发展起来。

## 2.2 编译原理和技术的发展

在Fortran诞生的同时期，Noam Chomsky开始了对自然语言结构的研究。他根据语言文法的难易程度以及识别它们所需要的算法来对语言分类，归纳总结出了Chomsky架构。Chomsky架构包括了文法的四个层次：0型文法、1型文法、2型文法和3型文法，且后者均为其前者的特殊情况。**2型文法（或上下文无关文法）被证明是程序设计语言中最有用的，而且今天它已代表着程序设计语言结构的标准方式。基于他的发现，人们进行了对**分析问题（parsing problem）的研究，提出了**有限状态自动机（Finite Automaton）和正则表达式（Regular Expression）。它们**同上下文无关文法紧密相关，并且引出了表示程序设计语言的单词的符号方式。

随着分析问题的研究深入，人们开始研究编译器的自动构造。这些程序最初被称为编译器的编译器（Compiler-compiler），但更确切地应称为分析程序生成器（Parser Generator），这是因为它们仅仅能够自动处理编译的一部分。其中，最著名的是由Steve Johnson在1975年为Unix系统编写的**Yacc**。类似的，有限状态自动机的研究也发展出一种称为扫描程序生成器的工具，现在仍被广泛使用的**Lex便是**这其中的佼佼者。

自展技术也是编译器的开发与发展中不可忽视的一环。所谓自展，实际上就是用一个功能不太完善的编译器支持的语言写代码，然后放到这个编译器上去编译，产生一个比自己更完善的编译器的过程。因此，它能够不断进行自我的迭代更新，从而不断得到优化。从20世纪60年代起，就不断有人使用自展技术来构造编译程序。在1971年，通过自展技术生成的PASCAL编译程序诞生了。这是一个里程碑式的事件，这也使得自展的影响越来越大。

**同时，随着嵌入式系统的广泛发展，**交叉编译的概念开始出现和流行起来。如果在一种计算机环境中运行的编译程序，能够编译出在另外一种环境下运行的代码，我们就称这种编译器**支持交叉编译**，它能够有效解决运行目标环境中资源有限，无法进行直接本地编译的问题。还有并行技术和并行语言的发展，和系统芯片设计方法以及关键EDA技术的研究，也带动了专用语言及其编译技术的不断优化。

## 2.3 GNU和GCC

在20世纪80年代，当时主流的Unix系统是收费的商业软件，且只开放了一部分源码。而MIT 教授理查德·斯托曼（Richard Stallman）认为，应将软件源码看成人类共同拥有的知识财富，能够公开自由地交换、修改。因此，他在1985年提出了 GNU 计划，希望发展出一个完全开源和免费的操作系统来取代Unix，这个操作系统就叫GNU’s Not Unix，也就是所谓的GNU。

但是，GNU 在开发完全免费的操作系统上并未取得成功，直到 20 世纪 90 年代由林纳斯·本纳第克特·托瓦兹开发了 Linux 操作系统，GNU 才算完成了任务。而GNU计划中开发的一些系统级的自由软件，却获得了远超过GNU计划本身的影响力，有的被移植到各大操作系统上，一直广泛使用至今，GCC便是其中之一。

GCC原名为GNU C Compiler，是**GNU中的C语言编译器**模块。它在最初开发的时候，仅仅是一个C程序语言的编译器，GCC的意思也只是 GNU C Compiler 而已。但它的出色之处在于其执行效率比普通编译器平均高出 20%-30%，同时完全免费并开放源代码，因此它很快成为了历史上最优秀的C语言编译器。

GCC不仅功能非常强大，结构也异常灵活。经过这么多年的发展，GCC已经不仅能支持C语言，它现在还支持 Ada、C++、Java、Objective-C、Pascal、COBOL 以及函数式编程和逻辑编程的 Mercury 语言等。因此，现在的GCC已经变成了 GNU Compiler Collection，也就是GNU 编译器家族。它也成为编译与创建其他操作系统的主要编译器，包括[BSD](http://zh.wikipedia.org/wiki/BSD)家族、[Mac OS X](http://zh.wikipedia.org/wiki/Mac_OS_X)、[NeXTSTEP](http://zh.wikipedia.org/wiki/NeXTSTEP)与[BeOS](http://zh.wikipedia.org/wiki/BeOS)等。目前，GCC已发展到了 12.x 的版本，在很长一段时间里，几乎所有开源软件和自由软件的开发中都能看到它的身影。无疑，GCC处在开源软件的核心地位。

## 2.4 LLVM

在GCC诞生后的很长一段时间里，Apple一直使用GCC作为官方的编译器。虽然GCC作为开源世界的编译器标准一直做得不错，但Apple却对编译工具提出了更高的要求。例如，Apple的Objective-C语言新增了许多特性，同时许多功能需要用模块化的方式来调用GCC进行开发，而GCC却无法满足Apple的需求。于是，Apple转向于寻找一个高效的、模块化的、协议更放松的开源的编译器替代品。

与此同时，在UIUC攻读计算机硕士和博士的Chris Lattner凭着自己强烈的兴趣，不断地研究探索关于编译器的未知领域，发表了一篇又一篇的论文。他在硕士毕业论文里提出了一套完整的在编译时、链接时、运行时甚至是在闲置时优化程序的编译思想，并与 Vikram Adve教授共同启动了LLVM项目。LLVM最初的含义是Low Level Virtual Machine，这是因为Chris Lattner本来只是想写一个底层虚拟机，即使现在LLVM已经和虚拟机毫无关系了。和其它编译器相比，LLVM的特殊之处在于它具有语言无关的中间代码层，因此可以作为多种语言的后端，提供针对多种CPU的优化和代码生成功能。LLVM 在Chris Lattner读博士时更加成熟，他使用 GCC 作为前端来对用户程序进行语义分析产生 IF（Intermidiate Format），然后使用LLVM分析结果完成代码优化和生成。这项研究让他在 2005 年毕业时就成为了业界小有名气的编译器专家，他也因此早早地被 Apple 盯上，最终成为其编译器项目的骨干。

进入Apple之后，Chris Lattner首先将LLVM与OpenGL栈进行结合，使得OpenGL栈能够产出更高效率的图形代码。同时，LLVM 的链接优化被直接加入到Apple的代码链接器上，推动了Mac OS X 10.6 Snow Leopard的新功能发展。

最初，Apple使用LLVM时，使用的是GCC作为前端，但时日愈长，GCC系统就愈发显现出其庞大而笨重的缺点来。因此，Apple决定从零开始写C、C++、Objective-​ ​C语言​​的前端Clang，以求完全替代掉GCC。Clang于2007年开始开发，最早完成的是C编译器，之后两年也相继实现了对Objective-C和C++语言的支持。Clang一个重要的特性是编译快速、占内存少，却能生成比GCC质量跟高的代码。得益于自身健壮的​​架构​​和Apple的大力支持，Clang越来越全能，支持的项目也越来越多，如Mac OS X 10.6时代的Xcode和Interface Builder等，皆由Clang编译。Clang的加入代表着LLVM真正走向成熟，而后来被广泛使用的Swift语言，也是基于LLVM作为编译器框架所开发的。

现在，LLVM已经成为了一个模块化和可重用的编译器和工具链技术的集合，一个功能强大的编译器工具套装。并且随着LLVM的发展，它的名字原意“底层虚拟机”也越来越不相符，因此开发者们抛弃了这个意涵，使得LLVM成为了一个单纯的品牌名称。LLVM不仅拥有很强的灵活性和可移植性，还兼具现代化的设计，优秀的模块化支持，以及丰富的工具和函数库。我们不仅可以使用它作为工具进行编译，还可以使用它编写自己的编译器。

从LLVM的发展历程中可以看到，LLVM在短短五年内取得了快速的发展，这充分反映了Apple对于产品技术的远见和超强的执行力。也正因如此，LLVM一跃成为了最领先的开源软件技术之一。我们将在后文对LLVM进行全面和详细的介绍。

1. **LLVM的设计**

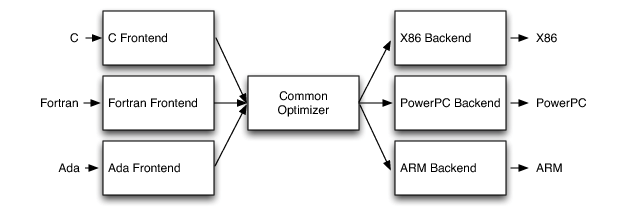
## 3.1 整体架构设计

对于传统静态编译器而言，最流行的设计是三阶段设计，其主要组件是前端（Frontend）、优化器（Optimizer）和后端（Backend），其结构如下图所示。



前端负责解析源代码，通过语法分析，语义分析，构建特定于语言的抽象语法树来表示输入代码。优化器负责对代码进行转换来优化代码的运行时间，例如消除一些冗余计算。而后端负责将代码映射到目标指令集，生成机器语言，并进行机器相关的代码优化。

LLVM采用的也是这种经典的三阶段设计，但与传统编译器不同的是，LLVM对于不同的语言都提供了同一种中间表示IR。前端使用不同的编译工具对代码文件作词法分析以形成抽象语法树AST，然后将分析好的代码转换成LLVM的中间表示IR（Intermediate Representation）；中间部分的优化器只对中间表示IR进行操作，通过一系列的pass来优化IR；后端负责将优化好的IR解释成对应平台的机器码，其具体结构如图所示：



这种设计的好处在于，移植编译器以支持新的源语言只需要实现一个新的前端，现有的优化器和后端都可以重复使用。例如，如果需要支持N个目标和M种源语言，LLVM只需要N+M个编译器，而对于前后端和解析器没有解耦的GCC来说，却需要N\*M个编译器。因此，这样的设计给编译器的开发带来了很大的便利性和可移植性。

## 3.2 LLVM IR设计

LLVM IR 作为一种编译器 IR，它的两个基本原则指导着核心库的开发：采用静态单赋值形式（Static Single Assignment SSA）表示，将代码组织为三地址指令序列，并使用数量无限制的寄存器让优化能够快速执行；同时将整个程序的 IR 都存储到磁盘，使得链接时的优化易于实现。

首先，我们先明确静态单赋值的概念。当程序中的每个变量都有且只有一个赋值语句时，这个程序就被称作是SSA形式的。在LLVM IR 中，每个变量都在使用前都必须先定义，且每个变量只能被赋值一次。以 1 \* 2 + 3 为例：



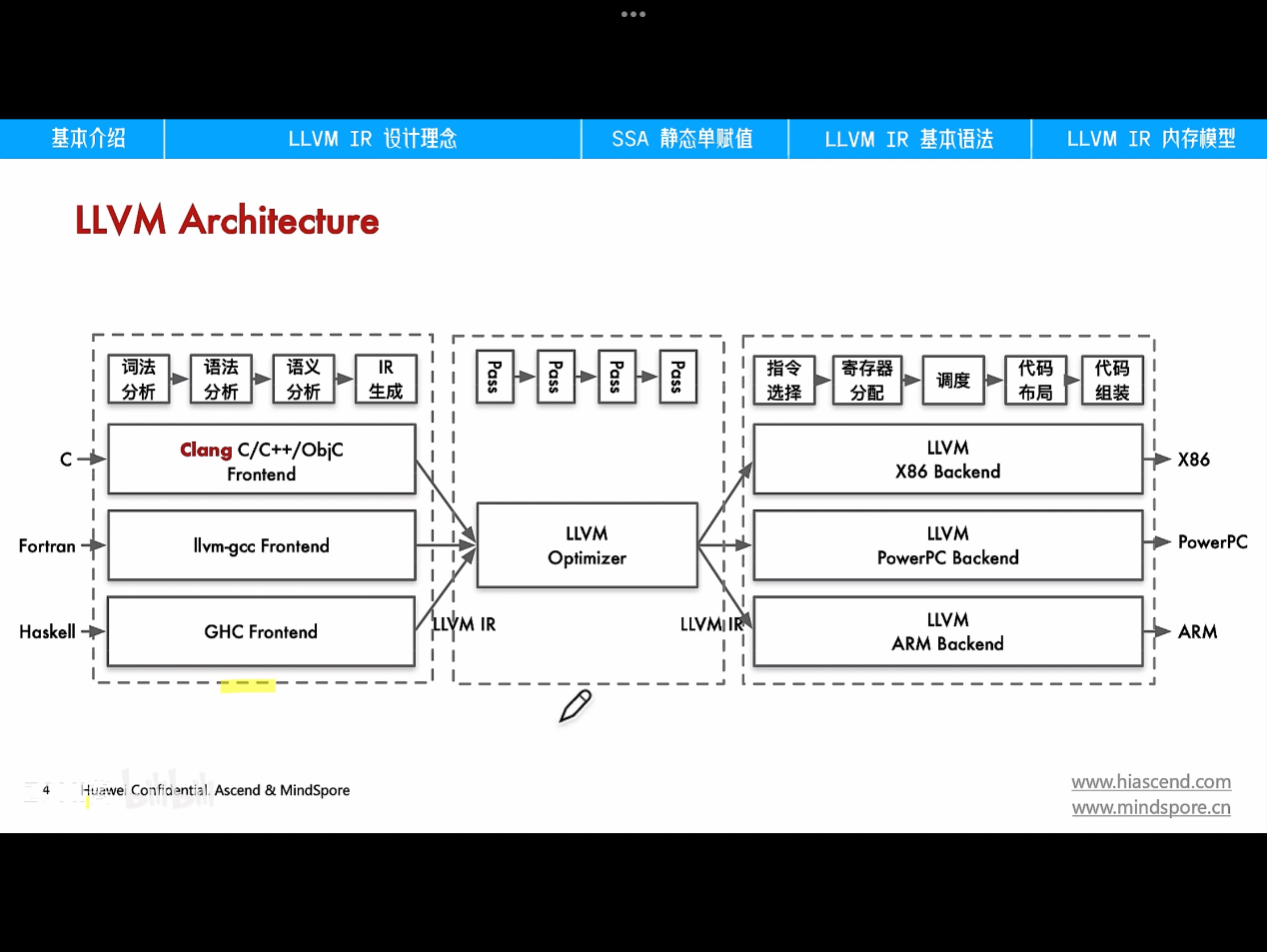
每个值只有单一赋值定义了它。因此，每使用一个值都可以立刻向后追溯到给出其定义的唯一的指令，这就极大地方便了编译器的正向和反向的遍历。同时，这也简化了编译器的优化过程。这是因为SSA形式建立了平凡的use-def链，可以简单地追溯到每个值的到达使用之处。

LLVM的基本语法类似于精简指令集（RISC）的底层虚拟指令集。它和真实精简指令集一样，支持简单指令的线性序列，例如添加、相减、比较和分支；它的指令都是三地址形式，能够接受一定数量的输入，然后在不同的寄存器中存储计算结果。但与大多数精简指令集不同的是，LLVM 使用强类型的简单类型系统，并剥离了机器差异。同时，它也不使用固定的命名寄存器，而是统一使用以 ％ 字符命名的临时寄存器。

在表示形式上，LLVM有三种完全等价的中间格式。分别是在内存中的编译中间语言，比如无法通过文件的形式得到的指令类等）；在硬盘上存储的二进制中间语言（格式为.bc）；以及格式为.ll的，人类可读的代码语言。

最后，如果需要自己对优化层进行更改，那么了解LLVM的内存模型是很有必要的。LLVM IR 文件的基本单位称为module，Module类聚合了整个翻译单元用到的所有数据，它是LLVM术语中的“module”的同义词。一个module中可以拥有多个顶层实体，比如function和global variavle。Function类包含有关函数定义和声明的所有对象，一个function define中至少有一个Basic Block，这个类包含了封装了LLVM指令序列，可通过begin()/end()访问它们。而每个Basic Block中有若干instruction，它表示LLVM IR的运算原子，并且都以terminator instruction结尾。

1. **LLVM的编译过程**



上图是更加细化的LLVM整体架构图，中间的每一个箭头都代表一个IR转换。可以看出，在LLVM架构中，IR不止存在于三个阶段之间，也存在于每一个阶段内部，这也使得LLVM的模块化程度非常高，能够大量复用代码来提高使用效率。

基于2.2提到的LLVM的整体架构设计，我们将从前端、优化层、后端这三个阶段对LLVM的编译过程进行详细的分析。

## 4.1 LLVM前端

LLVM的前端将源代码变换成编译器的中间表示LLVM IR，它处于代码生成之前。首先，词法分析是编译程序的第一步。词法分析器把程序代码的输入流切分成token，这个过程被成为符号化。一个token可以是一个字符串，由一个或多个同一范畴的字符组成。然后，语法分析器根据语言的语法规则来解析代码，决定输入的代码是否能够根据既定的语法组成token流，解析完成后构建出一棵AST抽象语法树。AST 的使用集中在语义分析阶段，在这个阶段，编译器会检查程序和语言元素是否正确使用，并基于 AST 生成符号表。在验证正确后，AST 还是代码生成的基础。通过以上的操作，现在所有的必要信息都存储于AST这一数据结构中，最后一步就是通过LLVM内建的API将AST转换为对应的LLVM IR。至此，前端的工作便完成了。

## 4.2 LLVM优化层

在优化层中，每一次优化都被称之为Pass。优化通常由分析 Pass 和转换 Pass 组成，分析 Pass指的是负责发掘性质和优化机会，而转换 Pass指的是生成必需的数据结构，以方便后者的后续利用。LLVM具有非常多的Pass类型，我们可以在LLVM的官方文档中对每种Pass的详细定义进行查看。例如，Aggressive Dead Code Elimination会将程序没有利用到的死代码进行消除，Merge Duplicate Global Constants将重复的全局常量合并到一个共享的常量中，这都可以优化中间代码的运行时间。

在转换Pass和分析Pass之间，有两种主要的依赖类型：显式依赖和隐式依赖。显式依赖指的是转换Pass需要一种分析，则Pass管理器自动地安排它所依赖的分析Pass在它之前运行；而隐式依赖将转换或者分析Pass要求IR代码运用特定表达式，需要手动地以正确的顺序把这个Pass加到Pass队列中。

在进行优化时，我们需要调用Pass API。Pass类是实现优化的主要资源。然而我们并不直接使用它，而是通过子类来使用。当实现一个Pass时，我们应该选择适合自己的Pass的最佳粒度和适合此粒度的最佳子类。常见的子类有ModulePass、FunctionPass、BasicBlockPass等。

## 4.3 LLVM后端

后端由一套分析和转换 Pass 组成，它们的任务是代码生成，即将LLVM IR变换为目标代码（或者汇编）。

在整个后端流水线中，用到了四种不同层次的指令表示：内存中的LLVM IR、SelectionDAG 节点、MachineInstr、和 MCInst。LLVM的后端流水线过程如图所示：



图中白色的Passes是一些非必需的Pass，而灰色的Passes则是必需的Passes，也被称为Super Pass，我们也将对它们进行深入分析。

Instruction Selection指的是指令选择，它将内存中的LLVM IR变换为目标特定的Selection DAG节点，DAG指的是有向无环图，每个DAG都能够表示单一基本块的计算。在每个DAG中，节点表示指令，而边编码了指令间的数据流依赖。通过这个过程，LLVM代码生成程序库能够运用基于树的模式匹配指令选择算法。

Instruction Scheduling指的是指令调度。第1次指令调度也称为前寄存器分配（RA）调度，它对指令排序，同时尝试发现尽可能多的指令层次的并行，然后将指令变换为MachineInstr的三地址表示形式。而第2次指令调度也被称为后寄存器分配（RA）调度，因为已经进行了寄存器分配，因此此时可获得真实的寄存器信息。如果发现某些类型寄存器存在延迟，此时可以通过指令调度再次改进指令顺序。

Register Allocation指的是寄存器分配。我们在上文中提到，LLVM IR具有两个特性，其中一个是LLVM IR拥有无限数量的寄存器集。这个性质一直保持着，直到寄存器分配。寄存器分配将无限的虚拟寄存器引用转换为有限的目标特定的寄存器集，如果寄存器不够，则将其挤出（spill）到内存进行缓存。

最后一个阶段是Code Emission，代码输出阶段，它将指令从 MachineInstr 表示变换为更加适合汇编器和连接器的MCInst 实例。之后，我们就能够输出汇编代码或者二进制块的特定目标代码格式。

1. **LLVM工具**

## 5.1 常用工具概述

作为一个编译器套装，LLVM具有相当丰富的工具和库，我们也对一些常用的工具进行了了解。

首先是opt，这是一个在IR级别做程序优化的工具，输入和输出都是同一类型的LLVM IR。在设计编译器时，我们需要经常性地调用这个工具来验证自己的优化pass是否正确。

llc是微观意义上的LLVM编译器。不同于gcc的编译器，它的输入是LLVM IR，输出是汇编文件或者是目标文件，可以通过指令来进行指定。若生成是目标文件，llcjiu 会调用LLVM中的汇编输出的代码库来工作，这个汇编器和gcc中的汇编器也不同，它输入的是一种后端的中间表示MI。除此之外，还可以用-On来指定优化级别或一些其它参数。

与之对应，llvm-mc是微观意义上的LLVM汇编器，它输入汇编文件，输出目标文件。同时，它也可以通过指定特殊参数来反汇编。可以发现，llc和llvm-mc都会调用到输出目标文件的库，也就是MCObjectStreamer。

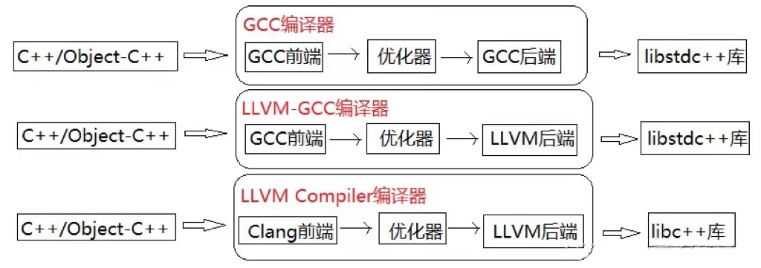
此外，我们还会用到lli——LLVM IR的解释器和llvm-link，即LLVM的链接器。针对于LLVM IR，也有汇编器llvm-as和反汇编器llvm-dis。

## 5.2 Clang

Clang是现在LLVM项目中一个非常重要的前端工具，它包括Clang前端和Clang静态分析器等。Clang前端的目的是输出代码对应的抽象语法树AST，并将代码编译成LLVM Bitcode，接着在后端使用LLVM编译成平台相关的机器语言。

最初，在Chris Lattner最初被Apple招致麾下时,Apple采用的是LLVM-GCC架构，即使用GCC作为前端，LLVM作为后端。但是，GNU组织出台了《GCC 运行环境豁免条款》从根本上限制了 LLVM-GCC 的开发，这让Apple公司下定决心，从新开始编写C编译器。因此，由Chris Lattner牵头，开发出了基于LLVM架构的Clang编译器，而以Clang为前端，LLVM为后端的LLVM Compiler 1.0也就此诞生。

下图是Apple的编译器发展过程：



相比于 GCC，Clang 具有编译速度快，占用内存小，采用模块化设计，诊断信息可读性强，设计清晰简单，容易理解，易于扩展增强等特点。在某些平台上，Clang 的编译速度显著的快过 GCC；Clang 生成的 AST 所占用的内存只有GCC 的五分之一左右；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；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；同时，它的学习曲线也比代码基础古老的 GCC 相比更加平缓。

当前，Clang 还处在不断完善的过程中。Clang目前支持的语言只有C，C++，Objective-C和Objective-C++，而不支持Fortran，Pascal，Java，Ada和Go等主流语言。同时,目前Clang 对 C++ 的支持依然落后于 GCC，还需要加强对 C++的全方位支持。同时，Clang目前只支持Linux，Windows和Mac OS平台上使用，而GCC流行的时间比较长，被广泛地使用在了更多的平台上。

1. **总结**

GCC是GNU和Linux系统的官方编译器，也是编译和创建其他UNIX操作系统的主要编译器。在很长一段时间里，它都是跨平台软件的编译器首选。

而LLVM包含一系列模块化的编译器组件和工具链，它的主要特点是三段式，即将前端、优化层和后端分离，同时采用通用的中间表示层IR，大大提高了模块化程度和使用效率。而Clang是Apple研发的基于LLVM框架的前端编译器，它支持C，C++，Objective-C或Objective-C++语言，具有比GCC更优的性能。

与GCC相比，Clang优势在于具有更优的模块化和复用性，更高的编译效率和更少的内存占用，更清晰的设计降低了它的学习成本。同时，Clang支持序列化其生成的AST到硬盘上并被其他程序所读入，也为用户提供了更清晰的诊断信息。但同时GCC广泛流行了如此之久，也有其独特的竞争力。相比于Clang，GCC支持更多语言前端，更多平台，更多语言扩展和更多的CPU 目标架构。

如果说GCC就像橡皮泥，能够塑造出任何优秀的作品，但基本没有复用性可言；LLVM就像乐高积木，虽然更加粗犷，但是模块化和复用性极佳。虽然GCC和LLVM都采用了三段式结构，但GCC的前后端是耦合的，是一个完整的可执行文件，没有给其它语言的开发者提供代码重用的接口。即使它是开源的，源代码重用的难度也比较大。

而LLVM从一开始设计时就是为了解决编译器代码重用的问题，它制定了 LLVM IR 这一中间代码表示语言，并在各种应用场景中大量应用。例如在 IDE 中调用 LLVM 进行实时的代码语法检查，对静态语言、动态语言的编译、优化等。因此，LLVM的模块化程度很高，我们可以很轻易地抽取LLVM的组件（以库的形式）用于其它领域。如抽取LLVM JIT用于 MapD 这样的 GPU 数据库，或者抽取LLVM的整个后端（优化与CodeGen）用于 TVM 这样的深度学习推理框架。这样，LLVM就不仅仅是一个编译器后端，而是可以为需要JIT / CodeGen 功能的所有领域服务了。

**在此基础上，LLVM还能够快速地可定制化**。由于在LLVM中编写优化Pass非常方便，所以针对各种各样的应用，能够高效地**编写针对自己特定应用的优化Pass。同时LLVM使用C++11编写并有良好的代码组织**，因此更加易读和易用。

除此之外，LLVM的License是BSD，相比于GCC使用的GPLv3，给使用者提供了更大的自由度。GPL虽然出发点是代码的开源和免费使用，但却不允许修改后和衍生的代码做为闭源的商业软件发布和销售；而BSP允许使用者随心所欲地进行二次开发，可以自由的使用和修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。

**目前来说，如果需要开发项目，LLVM-GCC是一个更加安全的选择，稳定性已经达到了足够的高度，而Clang/LLVM更自由，可玩性更强，具有更好的编译性能，更高的易使用性，但尚未完全成熟。**

**LLVM仍在不断地发展中，官方文档中提出了许多关于LLVM优化方向的构想：在新版本中，LLVM会**提供更多流行的API来保持它的灵活性；继续加强它的模块化和子集化特性；同时不断提高优化器和代码生成器生成的代码质量，添加能够更好地支持新语言和目标结构的IR，并且新增更好的支持来执行高级语言的优化。

让我们共同期待LLVM的光明未来。

1. **参考资料**

The **LLVM** Compiler Infrastructure <https://llvm.org/>

Chris Lattner's Homepage <https://nondot.org/~sabre/>

LLVM for Grad Students <https://www.cs.cornell.edu/~asampson/blog/llvm.html>

LLVM tutorial <https://llvm.org/docs/tutorial/index.html>

Llvm-project <https://github.com/llvm/llvm-project>

LLVM基本概念入门 [LLVM基本概念入门 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/140462815)

详解三大编译器 [详解三大编译器：gcc、llvm 和 clang - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/357803433)

LLVM官方文档阅读 [(159条消息) LLVM官方文档阅读(一)LLVM介绍\_鱼肖浓++的博客-CSDN博客](https://blog.csdn.net/weixin_43296779/article/details/121989541?ops_request_misc=&request_id=&biz_id=102&utm_term=llvm&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-3-121989541.nonecase&spm=1018.2226.3001.4187)

LLVM编译原理和使用 [(159条消息) LLVM编译原理和使用\_光@的博客-CSDN博客](https://blog.csdn.net/yayaayaya123/article/details/83993041?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522166903034516800182162663%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=166903034516800182162663&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-83993041-null-null.142%5ev66%5econtrol,201%5ev3%5econtrol_2,213%5ev2%5et3_esquery_v3&utm_term=llvm&spm=1018.2226.3001.4187)

对LLVM的认识和理解 [(159条消息) 对LLVM的认识与理解（LLVM与传统编译器对比）\_What’smean的博客-CSDN博客](https://blog.csdn.net/weixin_42492218/article/details/121203965?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-0-121203965-blog-90300638.pc_relevant_multi_platform_whitelistv3&spm=1001.2101.3001.4242.1&utm_relevant_index=3)