> 感谢up主 ZOMI酱：https://space.bilibili.com/517221395
> 《AI编译器开发》没点基础还真看不懂

01 编译器基础概念

1. 什么是编译器?
2. 为什么 AI 框架需要引入编译器?
3. AI 框架和 AI 编译器之间什么关系?

编译器与解释器

编译器（Compiler）和解释器（Interpreter）是两种不同的工具，都可以将编程语言和脚本语言转换为机器语言。虽然两者都是将高级语言转换成机器码，但是其最大的区别在于：**解释器在程序运行时将代码转换成机器码，编译器在程序运行之前将代码转换成机器码**。

JIT和AOT编译方式

目前，程序主要有两种运行方式：**静态编译和动态解释**。

- **静态编译的代码程序在执行前全部被翻译为机器码**，通常将这种类型称为 AOT（Ahead of time），即“提前编译”；
- **动态解释的程序则是对代码程序边翻译边运行**，通常将这种类型称为 JIT（Just in time），即“即时编译”。

AOT 程序的典型代表是用 C/C++ 开发的应用，其必须在执行前编译成机器码，然后再交给操作系统具体执行；而 JIT 的代表非常多，如 JavaScript、Python 等动态解释的程序。

特点	JIT（即时编译）	AOT（提前编译）
优点	1．可以根据当前硬件情况**实时编译生成最优机器指令** 2．可以根据当前程序的运行情况**生成最优的机器指令序列** 3．当程序**需要支持动态链接**时，只能使用JIT的编译方式 4．可以**根据进程中内存的实际情况调整代码**，使内存能够更充分的利用	1．在程序运行前编译，可以**避免在运行时的编译性能消耗和内存消耗** 2．可以在程序运行初期就达到**最高性能** 3．可以显著**加快程序的执行效率**
缺点	1．编译需要占用运行时Runtime的资源，会导致进程执行时候卡顿 2．编译占用运行时间，对某些代码编译优化不能完全支持，需在流畅和时间权衡 3．在编译准备和识别频繁使用的方法需要占用时间，初始编译不能达到最高性能	1．在程序运行前编译会使程序安装的时间增加 2．将提前编译的内容保存起来，会占用更多的内存 3．牺牲高级语言的一致性问题

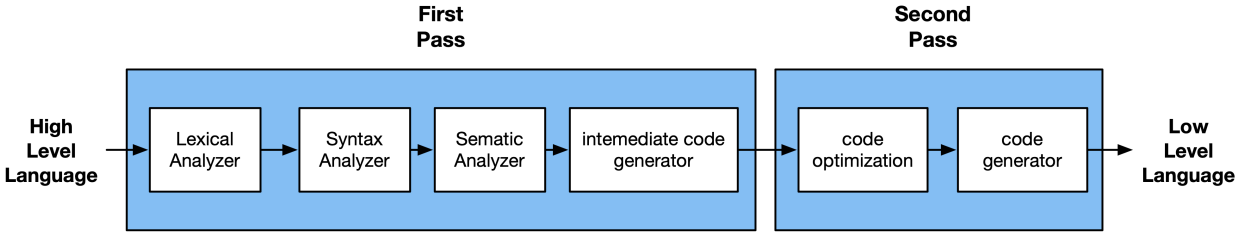
在AI框架中区别

目前主流的 AI 框架，都会带有前端的表达层，再加上 AI 编译器对硬件使能，因此 AI 框架跟 AI 编译器之间关系非常紧密，部分如 MindSpore、TensorFlow 等 AI 框架中默认包含了自己的 AI 编译器。目前 PyTorch2.X 版本升级后，也默认自带 Inductor 功能特性，可以对接多个不同的 AI 编译器。

编译方式	描述	典型代表
AOT（提前编译）	**静态编译**的代码程序在执行前全部被翻译为机器码，适合移动、嵌入式深度学习应用。	1．推理引擎：训练后的AI模型提前固化，用于推理部署。 2．静态图生成：神经网络模型表示为统一的IR描述，运行时执行编译后的内容。
JIT（即时编译）	**动态解释**的程序边翻译边运行，适合需要实时优化的场景。	1．PyTorch JIT：将Python代码实时编译成本地机器代码，优化和加速深度学习模型。 2．Jittor：基于动态编译JIT，使用元算子和统一计算图的深度学习框架，实现高效操作和自动优化。

Pass 和中间表示 IR

Pass 主要是对源程序语言的一次完整扫描或处理。在编译器中，Pass 指所采用的一种结构化技术，用于完成编译对象（IR）的分析、优化或转换等功能。Pass的执行就是编译器对编译单元进行分析和优化的过程，Pass 构建了这些过程所需要的分析结果。

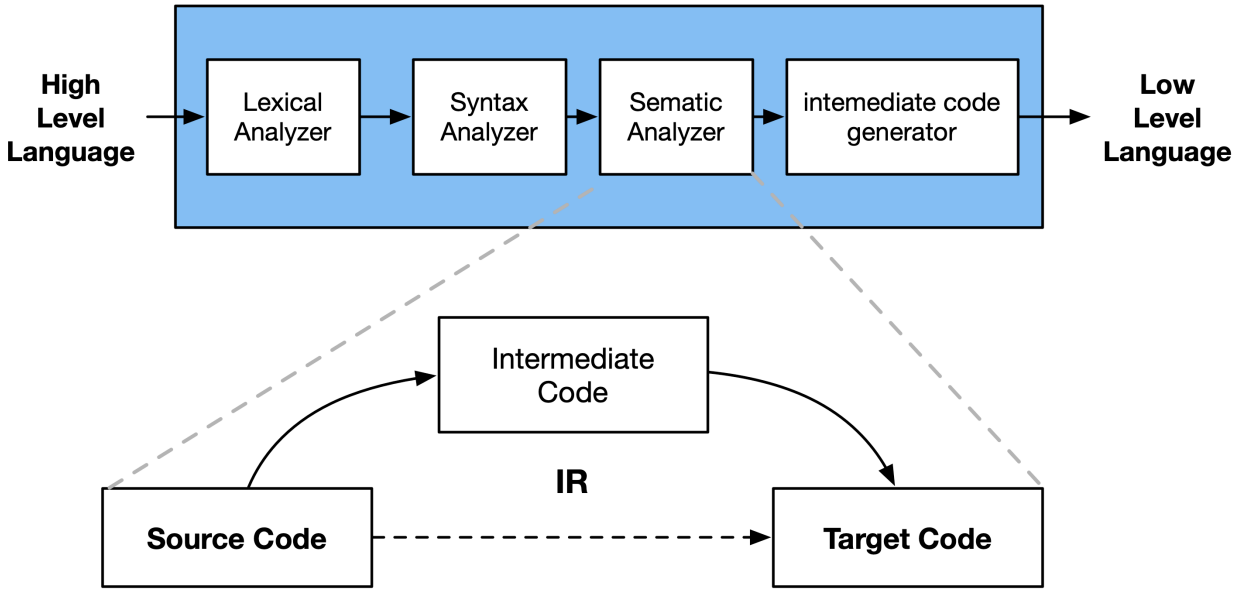


在编译器 LLVM 中提供的 Pass 分为三类：**Analysis pass、Transform pass 和 Utility pass**。

Pass 类型	描述	功能	常见例子
Analysis Pass	计算相关IR单元的高层信息，但不对其进行修改。这些信息可以被其他Pass使用，或用于调试和程序可视化。	1. 从IR单元中挖掘并存储信息 2. 提供查询接口供其他Pass访问 3. 提供invalidate接口以处理信息失效	Basic Alias Analysis、Scalar Evolution Analysis
Transform Pass	使用Analysis Pass的分析结果，以某种方式改变和优化IR。	1. 改变IR中的指令和控制流 2. 可能会减少函数调用，暴露更多优化机会	Inline Pass
Utility Pass	功能性的实用程序，不属于Analysis Pass或Transform Pass。	1. 执行特定任务，如提取basic block	extract-blocks Pass

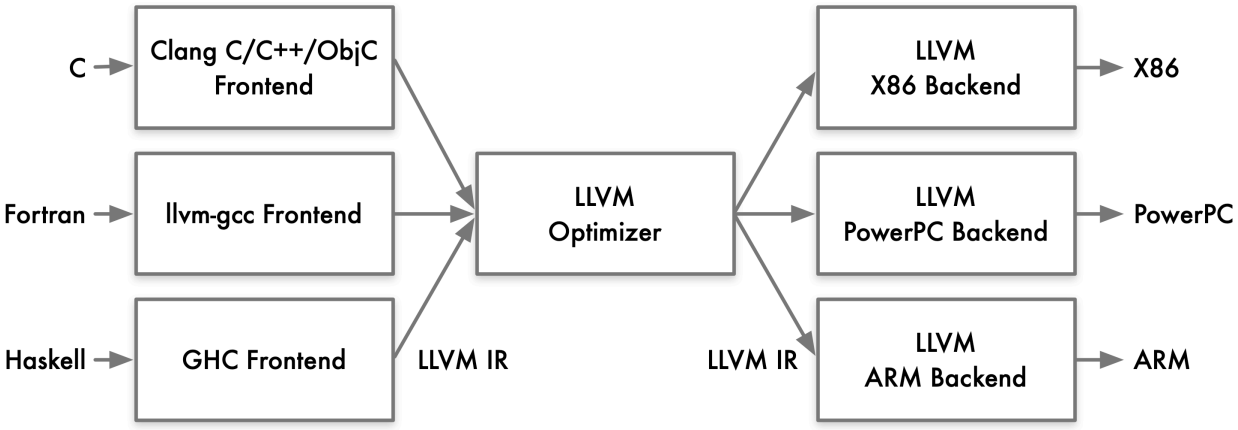
IR（Intermediate Representation）中间表示，是编译器中很重要的一种数据结构。编译器在完成前端工作以后，首先生成其自定义的 IR，并在此基础上执行各种优化算法，最后再生成目标代码。

如图所示，在编译原理中，通常将编译器分为前端和后端。其中，**前端会对所输入的程序进行词法分析、语法分析、语义分析，然后生成中间表达形式 IR。后端会对 IR 进行优化，然后生成目标代码。**



例如：LLVM 把前端和后端给拆分出来，在中间层明确定义一种抽象的语言，这个语言就叫做 IR。定义了 IR 以后，前端的任务就是负责最终生成 IR，优化器则是负责优化生成的IR，而后端的任务就是把 IR 给转化成目标平台的语言。LLVM 的 IR 使用 LLVM assembly language 或称为 LLVM language 来实现 LLVM IR的类型系统，就指的是 LLVM assembly language 中的类型系统。

因此，编译器的前端，优化器，后端之间，唯一交换的数据结构类型就是 IR，通过 IR 来实现不同模块的解耦。有些IR还会为其专门起一个名字，比如：Open64的IR通常叫做WHIRL IR，方舟编译器的IR叫做MAPLE IR，LLVM则通常就称为LLVM IR。



IR 在通常情况下有两种用途，1）**一种是用来做分析和变换**，2）**一种是直接用于解释执行。**

编译器中，基于 IR 的分析和处理工作，前期阶段可以基于一些抽象层次比较高的语义，此时所需的 IR 更接近源代码。而在编译器后期阶段，则会使用低层次的、更加接近目标代码的语义。基于上述从高到低的层次抽象，IR 可以归结为三层：高层 HIR、中间层 MIR 和 底层 LIR。

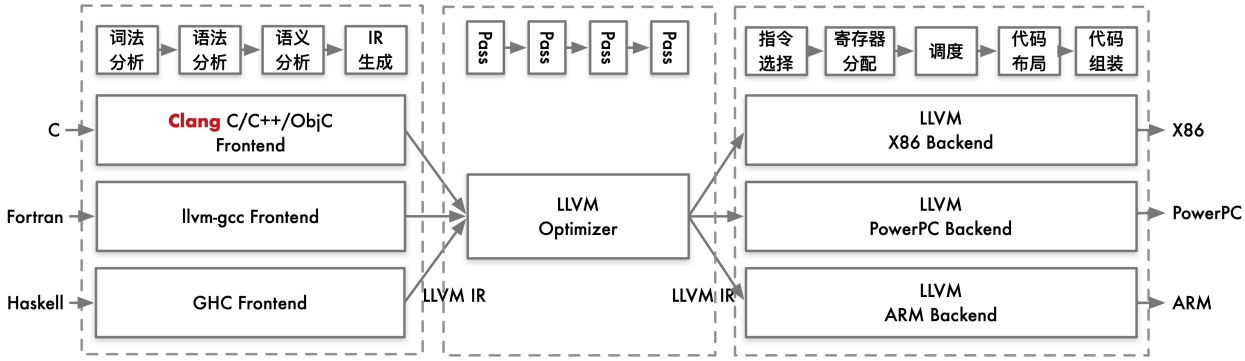
IR 类型	描述	用途	特点
HIR (High IR)	基于源程序语言执行代码的分析和变换。	用于IDE、代码翻译工具、代码生成工具等，执行高层次代码优化（如常数折叠、内联关联）。	1. 准确表达源程序语言的语义 2. 可以使用AST和符号表
MIR (Middle IR)	独立于源程序语言和硬件架构执行代码分析和优化。	用于编译优化算法，执行通用优化（如算术优化、常量和变量传播、死代码删除）。	1. 与源程序代码和目标程序代码无关 2. 通常基于三地址代码（TAC）
LIR (Low IR)	依赖于底层具体硬件架构做优化和代码生成。	用于执行与具体硬件架构相关的优化，生成机器指令或汇编代码。	1. 指令通常与机器指令一一对应 2. 体现了具体硬件架构的底层特征

> 三地址代码 TAC 的特点：最多有三个地址（也就是变量），其中赋值符号的左边是用来写入，右边最多可以有两个地址和一个操作符，用于读取数据并计算。

多层 IR 和单层 IR 比较起来，具有较为明显的优点：

- 可以提供更多的源程序语言的信息
- IR表达上更加地灵活，更加方便优化
- 使得优化算法和优化Pass执行更加高效

如在 LLVM 编译器里，会根据抽象层次从高到低，采用了前后端分离的三段结构，这样在为编译器添加新的语言支持或者新的目标平台支持的时候，就十分方便，大大减小了工程开销。而 LLVM IR 在这种前后端分离的三段结构之中，主要分开了三层 IR，IR 在整个编译器中则起着重要的承上启下作用。从便于开发者编写程序代码的理解到便于硬件机器的理解。



总结

1. 解释器是一种计算机程序，将每个高级程序语句转换成机器代码
2. 编译器把高级语言程序转换成机器码，即将人可读的代码转换成计算机可读的代码
3. Pass 主要是对源程序语言的一次完整扫描或处理
4. 中间表示 IR 是编译器中的一种数据结构，负责串联起编译器内各层级和模块

02 传统编译器发展

编译器与编程语言几乎是同步发展起来的，发展过程可以分为几个阶段：

- 第一阶段：20世纪50年代，出现了第一个编译程序，将算术公式翻译成机器代码，为高级语言的发展奠定了基础。
- 第二阶段：20世纪60年代，出现多种高级语言和相应的编译器，如 Fortran、COBOL、LISP、ALGOL等，编译技术也逐渐成熟和规范化
- 第三阶段：20世纪70年代，出现了结构化程序设计方法和模块化编程思想，以及面向对象的语言和编译器，如Pascal、C、Simula 等，编译技术也开始注重工程代码的可读性和可维护性。
- 第四阶段是20世纪80年代，出现了并行计算机和分布式系统，以及支持并行和分布式的语言和编译器，如Ada、Prolog、ML等，编译技术也开始考虑程序的并行和分布能力。
- 第五阶段：20世纪90年代，出现了互联网和移动设备等新兴平台，以及支持跨平台和动态特性的语言和编译器，如Java、C#、Python等，编译技术也开始关注程序的安全性和效率。
- 第六阶段：21世纪第一个10年，出现了以 Lua 为首的 Torch 框架，用于解决爆炸式涌现的AI应用和AI算法研究，之后又推出 TensorFLow、PyTorch、MindSpore、Paddle等 AI 框架，随着 AI 框架和 AI 产业的发展，出现了如 AKG、MLIR 等 AI 编译器。

传统编译器之争

目前主流如 LLVM 和 GCC 等经典的开源编译器，通常分为三个部分，前端(frontEnd)，优化器(Optimizer)和后端(backEnd)。

1. Front-End：主要负责词法和语法分析，将源代码转化为抽象语法树，即将程序划分为基本的组成部分，检查代码的语法、语义和语法，然后生成中间代码
2. Optimizer：优化器则是在前端的基础上，对得到的中间代码进行优化（如去掉冗余代码、子表达式消除等工作），使代码更加高效
3. Back-end：后端则是将已经优化的中间代码，针对具体的硬件生成目标代码，转换成为包括代码优化器和代码生成器

类型	GCC	Clang/LLVM
许可证	GNU GPL	Apache 2.0
代码模块化	**一体化架构**	**模块化**
支持平台	Unix, Windows, MAC	Unix, MAC
代码生成	**高效，有很多编译器选项可以使用**	**高效，LLVM 后端使用了 SSA 表单**
语言独立类型系统	没有	有
构建工具	**Make Base**	**CMake**
解析器	**最早采用 Bison LR，现在改为递归下解析器**	**手写的递归下降解析器**
链接器	LD	lld
调试器	GDB	LLDB

总结

- 编译技术是计算机科学皇冠上的一颗明珠，作为基础软件中的核心技术
- 编译器能够识别高级语言程序代码中的词汇、句子以及各种特定的格式和数据结构
- 编译过程，是将源代码程序转换成机器能够识别的二进制码
- 传统编译器通常分为三个部分，前端(frontEnd)，优化器(Optimizer)和后端(backEnd)