第3章

编 译 系 统

3.1 编译系统概述

第一台计算机出现在20世纪40年代，它使用由0、1序列组成的机器语言编程，这个序列明确地告诉计算机以什么样的顺序执行哪些技术。人们就开始了机器语言的程序设计：指定数据区编制一条条指令。由于任何人也无法记住并自如地编排二进制码(只有1和0的数字串)，所以用八进制、十六进制数写程序，输入后转换为二进制。因此，要在智能芯片上运行智能算法，需要先把开发者的高级语言编译成汇编指令，再通过汇编器最终生成二进制机器码，才能使程序在芯片上运行，在这个过程中主要是靠编译器来完成。

机器语言是用二进制代码表示的计算机能直接识别和执行的一种机器指令集合，如图3.1所示是一个机器语言例子。

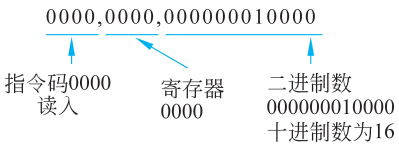

图3.1 机器语言

机器语言的缺点是可读性很差，难以理解。编程人员要首先熟记所用计算机的全部指令代码和代码的含义，编程困难并且效率低，缺少移植性。不同型号的计算机其机器语言是不同的。

汇编语言用字母和数字表示对应的0、1串指令及数据。如用“MOV A，16”表示“0000，0000，00000010000”；汇编语言和机器语言一一对应，比机器语言可读性好、编程效率高、调试性好。但汇编语言仍不具有移植性，且与人类语言差异很大。

因此，计算机公司提出了多种高级语言，从而便于开发人员开发程序，常见的开发语言有C、C++、Python和Java等。而如何把高级语言对应到汇编语言和机器语言这些低级语言，就需要编译器来完成。

编译器可以把高级语言编译成目标文件，并通过连接器把多个目标文件连接起来，从而产生出可执行文件，如图3.2所示。

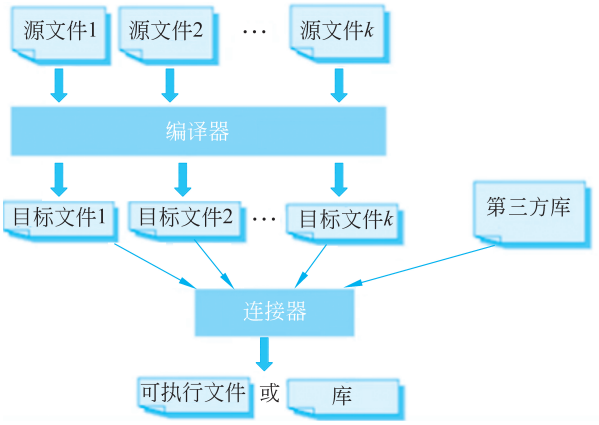

图 3.2　编译器产生出可执行文件

3.2　编　译　器

3.2.1　编译器流程说明

高级语言编译生成可以在芯片上运行二进制机器码的过程如图 3.3 所示。

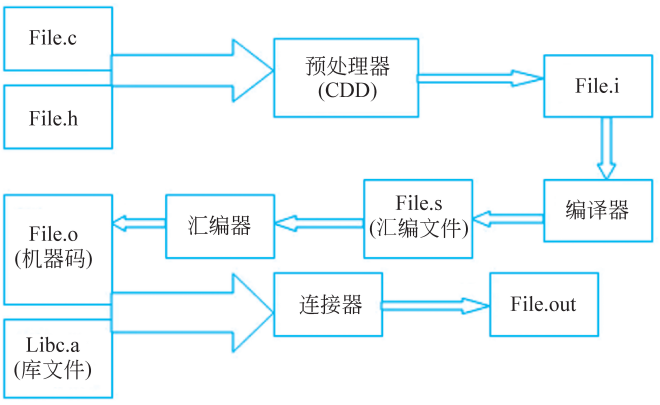

图 3.3　编译器流程图

(1) 编译器(Compiler)：将源语言翻译成目标语言。重要任务是在翻译过程中发现源语言中存在的错误。相比于解释器，目标语言执行的速度快。

编译器逐行扫描高级语言程序源程序，编译的过程如下。

① 词法分析(Lexical Analysis)。识别关键字、字面量、标识符（变量名、数据名）、运算符、注释行(给人看的，一般不处理)、特殊符号(续行、语句结束、数组)等 6 类符号，分别归类等待处理。

② 语法分析 (Syntax Analysis)。一个语句看作一串记号 (Token)流，由语法分析器进行处理。按照语言的文法检查判定是否是合乎语法的句子。如果是合法句子就以内部

格式保存，否则报错。直至检查完整个程序。

③ 语义分析(Semantic Analysis)。语义分析器对各句子的语法做检查：运算符两边类型是否相兼容；该做哪些类型转换（例如，实数向整数赋值要“取整”）；控制转移是否到不该去的地方；是否有重名或者使语义含糊的记号，等等。如果有错误，则转出错处理，否则可以生成执行代码。

④ 中间代码生成。中间代码是向目标码过渡的一种编码，其形式尽可能和机器的汇编语言相似，以便下一步的代码生成。但中间码不涉及具体机器的操作码和地址码。采用中间码的好处是可以在中间码上做优化。

⑤ 优化。对中间码程序做局部优化和全局（整个程序）优化，目的是使运行更快，占用空间最小。局部优化是合并冗余操作，简化计算，例如 x:=0 可用一条“清零”指令替换。全局优化包括改进循环、减少调用次数和快速地址算法等。

⑥ 代码生成。由代码生成器生成目标机器的目标码（或汇编）程序，其中包括数据分段、选定寄存器等工作，然后生成机器可执行的代码。

(2) 解释器(Interpreter)：利用用户的输入执行源程序中指定的操作。相比于编译器，错误诊断效果好，解释器是逐条语句执行。

第一步先作词法分析，建立内部符号表；再作语法和语义分析，并进行类型检查(解释语言的语义检查一般比较简单，因为它们往往采用无类型或动态类型系统)。完成检查后把每一语句压入执行堆栈，并立即解释执行。因为解释执行时只看到一条语句，无法对整个程序进行优化。但是解释执行占用空间很少。

操作系统的命令、Visual Basic、Java、JavaScript 都是解释执行的(其中有些语言也可以编译执行)。解释器不大，工作空间也不大，不过，解释执行难于优化、效率较低，这是这类语言的致命缺点。

(3) 预处理器(Preprocessor)：把源程序聚合在一起，把宏的缩写转换为源语言的语句。预处理器是由特殊的预处理器命令行控制的，例如在 C 语言中它们是以“#”符号开头的源文件行。预处理器的一般操作：从源文件中删除所有的预处理器命令行，并在源文件中执行这些预处理器命令所指定的转换操作。

(4) 汇编器(Assembler)：将汇编语言程序处理后生成可重定位的机器代码。汇编语言是一种以处理器指令系统为基础的低级语言，采用助记符表达指令操作码，采用标识符表示指令操作数。

(5) 连接器(Linker)：解决外部内存地址问题。将多个可重定位的目标文件以及库文件连接到一起，形成真正在机器上运行的代码。

高级语言源程序经编译后得到目标码程序，还不能立即装入机器执行，因为程序中如果用到标准函数(它们生成的目标码已存放在模块库中)，还需对编译后得到的目标模块进行连接。连接程序 (Linker)找出需要连接的外部模块，然后到模块库中找出被调用的模块，调入内存并连接到目标模块上，形成可执行程序。

(6) 加载器(Loader)：把所有的可执行目标文件放到内存中执行。

执行时，把可执行程序加载到内存中合适的位置（此时得到的是内存中的绝对地址）即可执行。

传统的三段编译器设计中，前端负责解析源码、检查源码错误以及建立抽象语法树来生成编译中间件(IR)；优化器负责将中间件进行逻辑重组以及优化；后端负责按照代码运行环境生成机器码并进行连接优化，libc 参与静态库连接以及运行时的动态库调用，如图 3.4 所示。

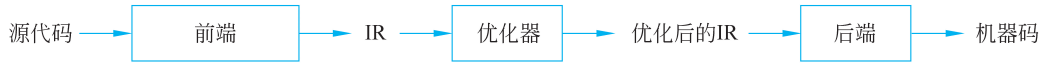

图 3.4　三段式编译器架构

3.2.2　连接过程说明

连接过程是将多个可重定位的目标文件以及库文件连接到一起，形成真正在机器上运行的代码。

1. 目标文件

以 Linux 系统为例，目标文件有如下几类。

(1) 可重定位文件。如 .o 文件，包含代码和数据，可以被连接成可执行文件或共享目标文件，静态连接库属于这一类。

(2) 可执行文件。如/bin/bash 文件，包含了可以直接执行的程序，一般没有扩展名。

(3) 共享目标文件。如.so 文件，包含代码和数据，可以跟其他可重定位文件和共享目标文件连接产生新的目标文件，也可以跟可执行文件结合作为进程映像的一部分。

目标文件由许多段组成，其中主要的段如下。

(1) 代码段(.text)。保存编译后得到的指令数据。

(2) 数据段(.data)。保存已经初始化的全局静态变量和局部静态变量。

(3) 只读数据段(.rodata)。保存只读变量和字符串常量，有些编译器会把字符串常量放到“.data”段。

(4) BSS 段(.bss)。保存未初始化的全局变量和局部静态变量。

(5) 重定位表。连接器在处理目标文件时，需要对目标文件中某些部位进行重定位，即代码段和数据段中那些绝对地址的引用位置，这些重定位信息记录在重定位表里。每个需要重定位的代码段或数据段都会有一个相应的重定位表，如.rel.text 是针对“.text”段的重定位表，“.rel.data”是针对“.data”段的重定位表。

2. 静态连接

几个目标文件进行连接时，每个目标文件都有其自身的代码段、数据段等，连接器需要将它们各个段合并到输出文件中，具体有两种合并方法。

(1) 按序叠加。将输入的目标文件按照次序叠加起来。这种方法会产生很多零散的段，而且每个段有一定的地址和空间对齐要求，会造成内存空间大量的内部碎片。所以这个方法现在较少用。

(2) 第二种方法分为两个步骤进行。

① 空间与地址分配。扫描所有输入的目标文件,获得各个段的长度、属性和位置,收集它们符号表中所有的符号定义和符号引用,统一放到一个全局符号表中。此时,连接器可以获得所有输入目标文件的段长度,将它们合并,计算出输出文件中各个段合并后的长度与位置并建立映射关系。

② 符号解析与重定位。经过步骤①后,输入文件中的各个段在连接后的虚拟地址已经确定了,连接器开始计算各个符号的虚拟地址。各个符号在段内的相对地址是固定的,连接器只需要给它们加上一个偏移量,调整到正确的虚拟地址即可。

3. 可执行文件的装载

可执行文件只有被装载到内存以后才能运行,最简单的办法是把所有的指令和数据全部装入内存,但这可能需要大量的内存。为了更有效地利用内存,根据程序运行的局部性原理,可以把程序中最常用的部分驻留内存,将不太常用的数据放在硬盘中,即动态装入。

现在大部分操作系统采用的是页映射的方法进行程序装载,将内存和所有硬盘中的数据和指令按页为单位划分成若干个页,以后所有的装载和操作的单位就是页。

4. 动态连接

静态连接有如下缺点。

(1) 浪费内存和磁盘空间。在多进程操作系统下,每个程序内部都保留了公用的库函数及其他数量可观的库函数及辅助数据结构,浪费大量空间。

(2) 程序开发和发布困难。一个程序如果使用了很多第三方的静态库,那么程序中一旦有任何库的更新,整个程序就要重新连接并重新发布给客户,非常不方便。

动态连接可以解决空间浪费和更新困难的问题,程序运行时才对目标文件进行连接。使用了动态连接之后,系统会首先加载该程序依赖的其他的目标文件,如果其他目标文件还有依赖,系统会按照同样方法将它们全部加载到内存。当所需要的所有目标文件加载完毕之后,如果依赖关系满足,系统开始进行连接工作,包括符号解析及地址重定位等。完成之后,系统把控制权交回给原程序,程序开始运行。此时如果运行第二个程序,它依赖于一个已经加载过的目标文件,则系统不需要重新加载目标文件,而只要将它们连接起来即可。

3.3 常见编译器

3.3.1 GCC 编译器介绍

GCC(GNU Compiler Collection,GNU 编译器套件)是由 GNU 开发的编程语言编译器。GNU 编译器套件包括 C、C++、Objective-C、FORTRAN、Java、Ada 和 Go 语言前端,也包括了这些语言的库(如 libstdc++、libgcj 等)。

GCC 编译器是 Linux 系统下最常用的 C/C++ 编译器，大部分 Linux 发行版中都会默认安装。

GCC 最基本的用法是：gcc [options] [filenames]。其中，options 就是编译器所需要的参数，filenames 给出相关的文件名称。

-c，只编译，不连接成为可执行文件，编译器只是由输入的.c 等源代码文件生成扩展名的目标文件，通常用于编译不包含主程序的子程序文件。

-o output_filename，确定输出文件的名称为 output_filename，同时这个名称不能和源文件同名。如果不给出这个选项，gcc 就给出预设的可执行文件 a.out。

-g，产生符号调试工具(GNU 的 gdb)所必要的符号信息，要想对源代码进行调试，我们就必须加入这个选项。

-O，对程序进行优化编译、连接，采用这个选项，整个源代码会在编译、连接过程中进行优化处理，这样产生的可执行文件的执行效率可以提高，但是，编译、连接的速度就相应地要慢一些。

-O2，比-O 更好的优化编译、连接，当然整个编译、连接过程会更慢。

-Idirname，将 dirname 所指出的目录加入到程序头文件目录列表中，是在预编译过程中使用的参数。

GCC 编译器可以在 x86 平台上面运行，也可以在 ARM 和 RISC-V 平台上应用，其编译的指令如下。

```
在 x86 平台上：gcc main.c -o main
在 ARM 平台上:arm-linux-gcc main.c -o main
在 RISC-V 平台上:riscv-gcc  main.c -o main
```

以上若只有一个文件，可直接用后缀-o 生成可执行文件 main；但若是几个文件时，需要先单独编译各个文件，然后再通过连接产生出可执行文件 main，如下。

```
gcc -o a.o -c a.c
gcc -o main.o -c main.c
gcc -o main a.o main.o
```

GCC 也支持华为公司的 openEuler 操作系统，GCC 针对 openEuler 支持基于开源 GCC-10.3 版本(https://gcc.gnu.org，2021 年 4 月发行)开发，并进行了优化和改进，实现软硬件深度协同优化，挖掘 OpenMP、SVE 向量化、数学库等领域极致性能，是一种 Linux 下针对鲲鹏 920 处理器的高性能编译器。GCC 针对 openEuler 默认使用场景为 TaiShan 服务器、鲲鹏 920 处理器、ARM 架构，操作系统为 CentOS 7.6、openEuler 20.09 等。GCC 是一个单一的可执行程序编译器，前段、IR 和后端没有明确的分界，耦合严重，难以独自发展，在整个编译过程中，中间诸多信息都无法被其他程序重用。

3.3.2　LLVM 编译器介绍

LLVM(http://llvm.org/)是构架编译器的框架系统，用 C++ 编写而成，用于优化以

任意程序语言编写的程序的编译时间、连接时间、运行时间以及空闲时间，对开发者保持开放，并兼容已有脚本。LLVM 核心库提供了与编译器相关的支持，可以作为多种语言编译器的后台来使用。能够进行程序语言的编译期优化、连接优化、在线编译优化、代码生成。LLVM 的项目是一个模块化和可重复使用的编译器和工具技术的集合。LLVM 是伊利诺伊大学的一个研究项目，计划启动于 2000 年，提供一个现代化的、基于 SSA 的编译策略，能够同时支持静态和动态的任意编程语言的编译目标。目前 LLVM 已经被苹果公司 iOS 开发工具、Xilinx Vivado、Facebook、Google 等各大公司采用。

LLVM 在继承了传统三段式设计的情况下，将优化器的输入输出接口、数据进行归一，即不同语言的前端解析后生成相同语法规则的中间件，经过优化后输出通用的代码给不同的后端进行目标代码生成。由于代码目标运行平台有限，后端相对固定，前端的输入格式固定，所以对于一门新语言的编译器开发，LLVM 具有方便快捷的集成能力，同样还有多种的前端作为开发的样例和对比，进而推动了 LLVM 框架的繁荣发展。LLVM 相对 GCC 编译器有着较快的编译速度，目标程序有着较好的性能表现，在编译错误上也有着更加友好的提示。

前端： LLVM 最初被用来取代 GCC 中的代码产生器，许多 GCC 的前端已经可以与其运行，LLVM 目前支持 Ada、C 语言、C ++ 、D 语言、FORTRAN、Haskell、Julia、Objective-C、Rust 及 Swift 的编译，它使用许多的编译器，有些来自 4.0.1 及 4.2 版本的 GCC。

中间端： LLVM 的核心是中间端表达式(Intermediate Representation，IR)，是一种类似汇编的底层语言。IR 是一种强类型的精简指令集(Reduced Instruction Set Computing，RISC)，并对目标指令集进行了抽象。例如，目标指令集的函数调用惯例被抽象为 call 和 ret 指令加上明确的参数。另外，IR 采用无限个数的暂存器，使用如%0、%1 等形式表达。LLVM 支持 3 种表达形式：人类可读的汇编，在 C++ 中对象形式和序列化后的 bitcode 形式。

后端： LLVM 已经支持多种后端指令集，包括 ARM、Qualcomm Hexagon、MIPS、NVIDIA(LLVM 中称为 NVPTX)、PowerPC、AMD TeraScale、AMDGPU、SPARC、SystemZ、RISC-V、WebAssembly、x86、x86-64 和 XCore。

3.3.3 TVM 编译器

有关深度学习编译器框架，近年有名的是华盛顿大学陈天奇提出的 TVM(Tensor Virtual Machine)框架，它旨在缩小以生产力为中心的深度学习框架与以性能和效率为中心的硬件后端之间的差距。TVM 与深度学习框架合作，为不同的后端提供端到端编译。TVM 与 LLVM 的架构非常相似。TVM 针对不同的深度学习框架和硬件平台，实现了统一的软件栈，以尽可能高效的方式，将不同框架下的深度学习模型部署到硬件平台上。TVM 的设计目的是分离算法描述、调度和硬件接口。该原则受到 Halide 的计算/调度分离思想的启发，而且通过将调度与目标硬件内部函数分开而进行了扩展。这一额外分离使支持新型专用加速器及其对应新型内部函数成为可能。TVM 具备两个优化层：一个是计算图优化层，用于解决第一个调度挑战；另一个是具备新型调度基元的张量优化

层，以解决剩余的 3 个挑战。通过结合这两个优化层，TVM 从大部分深度学习框架中获取模型描述，执行高级和低级优化，生成特定硬件的后端优化代码，如 CPU、GPU 和基于 FPGA 的专用加速器。实现了如下功能。

(1) 构建了一个端到端的编译优化堆栈，允许将高级框架(如 Caffe、MXNet、PyTorch、Caffe2、CNTK)专用的深度学习工作负载部署到多种硬件后端上(包括 CPU、GPU 和基于 FPGA 的加速器)。

(2) 提供深度学习工作负载在不同硬件后端中的性能可移植性的主要优化挑战，并引入新型调度基元(Schedule primitive)以利用跨线程内存重用、新型硬件内部函数和延迟隐藏。

(3) 在基于 FPGA 的通用加速器上对 TVM 进行评估，以提供关于如何最优适应专用加速器的具体案例。

3.3.4　方舟编译器

方舟编译器改变了系统及应用的编译和运行机制，直接将高级语言编译成机器码，让手机能直接听懂“高级语言”，消除了虚拟机动态编译的额外开销，提升了手机运行效率。同时，方舟编译器还能够理解程序特征、使用适合的指令来执行程序，因此能够极大程度地发挥出芯片的能力。目前，方舟编译器聚焦在 Java 代码性能上，未来，方舟编译器将覆盖多种编程语言(包括 C/C++ 、JS 等)，多种芯片架构(包括 CPU、GPU、IPU 等)，覆盖更广的业务场景。

方舟编译器主要有如下特点。

(1) 方舟编译器将手机开发中的多种语言 Java/C/C++ 实现了统一的中间表示 IR，将 Java/C/C++ 等混合代码一次编译成机器码直接在手机上运行，告别 Java 的 JNI 额外开销，使得不同语言代码在开发者环境中能够统一编译成同一套可直接执行的机器码，从而彻底消除混合语言互相调用的开销。

(2) 方舟编译器直接将代码优化从手机环节搬到了开发者环境，利用开发者环境更强大的算力，可以实现更先进和精细的优化算法，来达到更强大的优化效果。

(3) 方舟编译器采用了引用计数法(Reference Counting，RC)来进行内存的实时回收，并且配合使用了专门的消除环算法(消除对象互相引用带来的无法回收问题)，来避免 Android 虚拟机集中式回收带来的系统卡顿。相比 Android 虚拟机集中式内存回收，方舟编译器的内存回收是实时的而非集中式的，且不需要暂停应用进程，这样便大大消除了卡顿。

方舟编译器整体架构分为编译器输入、编译器处理和编译器输出。编译器处理采用了目前业界主流的三阶段设计。方舟编译器的整体架构如图 3.5 所示。

方舟编译器(OpenArkCompiler)是为支持多种编程语言、多种芯片平台的联合编译、运行而设计的统一编程平台，包含编译器、工具链、运行时等关键部件，并在 https://gitee.com/openarkcompiler 网站进行开源。

OpenArkCompiler 2.0 主要提供对 Java、C 语言的编译和运行支持，代码仓为 https://gitee.com/openarkcompiler/OpenArkCompiler。

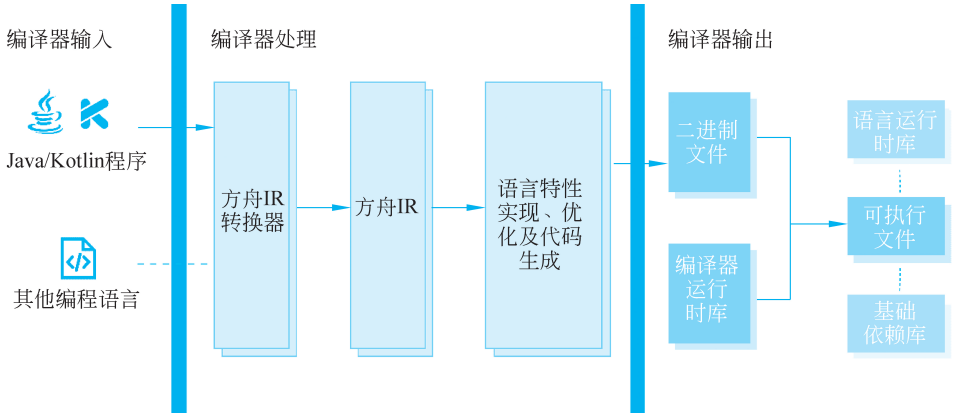

图 3.5 方舟编译器的整体架构

OpenArkCompiler 3.0 主要结合 OpenHarmony 和 HarmonyOS 面向多设备开发和运行的多语言应用的需求，新增对 JavaScript/TypeScript 语言、平台无关的应用分发格式、跨设备轻量级运行时的支持。目前在 OpenHarmony 开放的代码仓如下。

运行时公共组件：https://gitee.com/openharmony/ark_runtime_core。

JavaScript 运行时：https://gitee.com/openharmony/ark_js_runtime。

JavaScript/TypeScript 前端编译器：https://gitee.com/openharmony/ark_ts2abc。

3.3.5 毕昇编译器

毕昇编译器是华为编译器实验室针对鲲鹏等通用处理器架构场景，打造的一款高性能、高可信及易扩展的编译器工具链，增强和引入了多种编译优化技术，支持 C/C++/FORTRAN 等编程语言。

毕昇编译器基于开源 LLVM 开发，并进行了优化和改进，LLVM 是一种涵盖多种编程语言和目标处理器的编译器，毕昇编译器聚焦于对 C、C++、FORTRAN 语言的支持，利用 LLVM 的 Clang 作为 C 和 C++ 的编译和驱动程序，Flang 作为 FORTRAN 语言的编译和驱动程序。

毕昇编译器的运行平台是鲲鹏 920 硬件平台，支持的操作系统有 openEuler 21.03、openEuler 20.03 (LTS)、CentOS 7.6、Ubuntu 18.04、Ubuntu 20、麒麟 v10 和 UOS 20。

使用毕昇编译器，例如编译运行 C/C++ 程序，命令如下。

```
clang  [command line flags]  main.c  -o  main.o
clang++  [command line flags]  main.cpp  -o  main.o
```

毕昇编译器的默认选项：支持 LLVM 的所有优化等级(O0/O1/O2/O3/Ofast)，支持 Clang 的默认编译选项和 Flang 的默认编译选项，支持 fsanitize = address/leak/memory 等选项。

毕昇编译器除 LLVM 通用功能和优化外，对中端及后端的关键技术点进行了深度优化，并集成 Autotuner 特性支持编译器自动调优。初始编译阶段发生在调优开始之前，

Autotuner 首先会让编译器对目标程序代码做一次编译，在编译的过程中，毕昇编译器会生成一些包含所有可调优结构的 YAML 文件，告诉我们在这个目标程序中哪些结构可以用来调优，例如文件(Module)、函数(Function)、循环(Loop)。例如，循环展开是编译器中最常见的优化方法之一，它通过多次复制循环体代码，达到增大指令调度的空间，减少循环分支指令的开销等优化效果。若以循环展开次数(Unroll factor)为对象进行调优，编译器会在 YAML 文件中生成所有可被循环展开的循环作为可调优结构。当可调优结构顺利生成之后，调优阶段便会开始：①Autotuner 首先读取生成好的可调优结构的 YAML 文件，从而产生对应的搜索空间，也就是生成针对每个可调优代码结构的具体的参数和范围；②调优阶段会根据设定的搜索算法尝试一组参数的值，生成一个 YAML 格式的编译配置文件(Compilation config)，从而让编译器编译目标程序代码产生二进制文件；③最后 Autotuner 将编译好的文件以用户定义的方式运行并取得性能信息作为反馈；④经过一定数量的迭代之后，Autotuner 将找出最终的最优配置，生成最优编译配置文件，以 YAML 的形式存储。