AI编译器

1、编译器的基本概念

①编译器和解析器

> 编译器compiler:
>
> 解析器interpreter:

Compiler and Interpreter - 编译器与解析器

	Interpreter 解释器	Compile 编译器
程序步骤	1、创建代码 2、没有文件链接或机器代码生成 3、源语句在执行过程中逐行执行	1、创建代码 2、将解析或分析所有语言语句的正确性 3、将把源代码转换为机器码 4、链接到可运行程序 5、运行程序
Input	每次读取一行	整个程序
Output	不产生任何的中间代码	生成一个目标代码
工作机制	编译和执行同时进行	编译在执行之前完成
存储	不保存任何机器代码	存储编译后的机器代码在机器上
执行	程序执行是解释过程的一部分，因此是逐行执行的	程序执行与编译是分开的，它只在整个输出程序编译后执行
生成程序	不生成输出程序，所以他们在每次执行过程中都要评估源程序	生成可以独立于原始程序运行的输出程序(以exe的形式)
修改	直接修改就可运行	如果需要修改代码，则需要修改源代码，重新编译
运行速度	慢	快
内存	它需要较少的内存，因为它不创建中间对象代码	内存需求更多的是由于目标代码的创建
错误	解释器读取一条语句并显示错误。你必须纠正错误才能解释下一行	编译器在编译时显示所有错误和警告。因此，不修正错误就不能运行程序
错误监测	容易	难
编程语言	PHP, Perl, Python, Ruby	C, C++, C#, Scala, Java

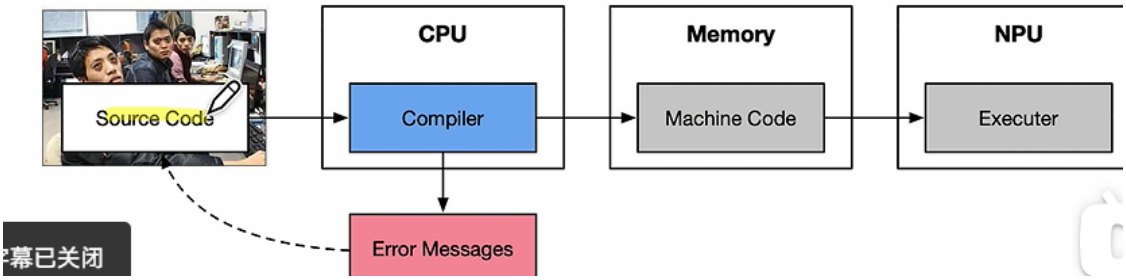
②JIT和AOT区别

> 程序主要有俩种运行方式：
>
> 静态编译：程序执行前全部被编译为**机器码**，称为AOT （ahead of time），即提前编译
>
> 动态解释：程序边编译边运行，通常将这种类型称为Jlt(just in time)，即”即时编译“

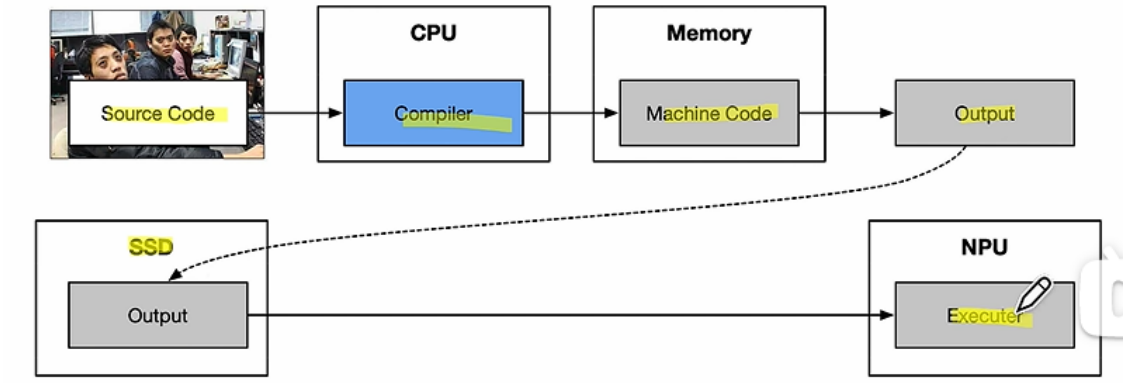
	Just in time	Ahead of time
优点	1. 可以根据当前硬件情况实时编译生成最优机器指令 2. 可以根据当前程序的运行情况生成最优的机器指令序列 3. 当程序需要支持动态链接时，只能使用JIT 4. 可以根据进程中内存的实际情况调整代码，使内存能够更充分的利用	1. 在程序运行前编译，可以避免在运行时的编译性能消耗和内存消耗 2. 可以在程序运行初期就达到最高性能 3. 可以显著的加快程序的启动
缺点	1. 编译需要占用运行时资源，会导致进程卡顿 2. 编译占用运行时间，对某些代码编译优化不能完全支持，需在流畅和时间权衡 3. 在编译准备和识别频繁使用的方法需要占用时间，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. 牺牲高级语言的一致性问题 3. 将提前编译的内容保存会占用更多的外

③什么时候用JIT什么时候用AOT

离线运算一般用到AOT



写出神经网络源码--》AI框架会对其进行编译--》转为机器码--》机器码丢给CPU，GPU，NPU等执行，直到收敛为止，然后将神经网络模型丢出。



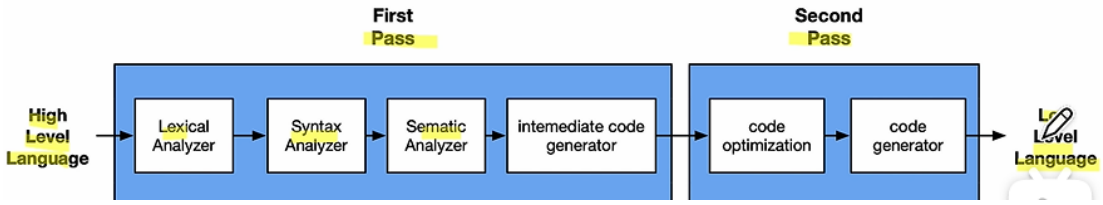
将模型保存到磁盘中，当需要运算时，把模型交给GPU等进行执行运算。

④编译器的pass和IR

pass对源程序进行扫描，然后将其变成低级语言

Pass

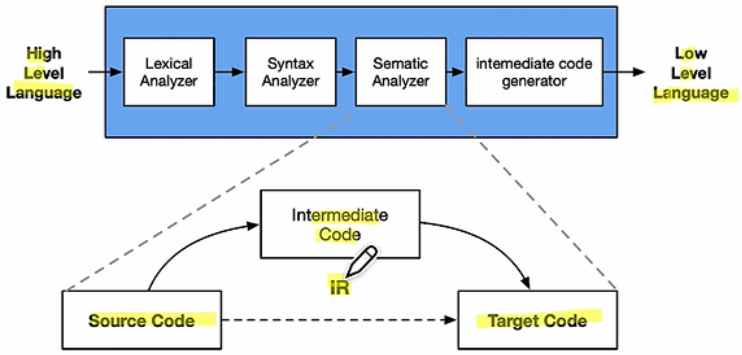
- Pass：One complete scan or processing of the source program.
  对源程序的一次完整扫描或处理



IR：中间表达

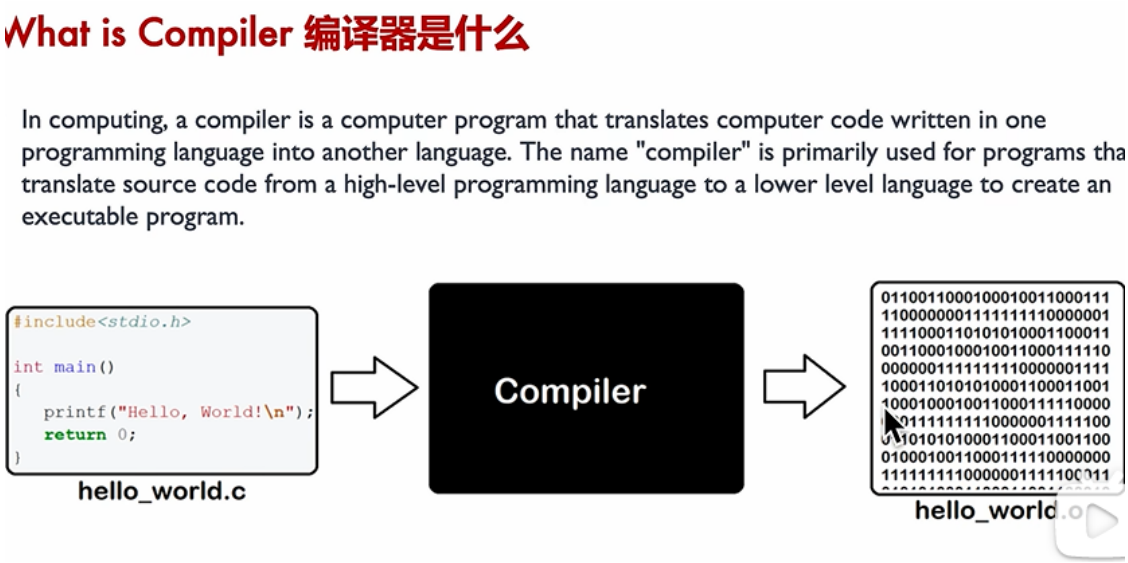
将源代码通过中间表达变成目标有代码

- IR: An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code.



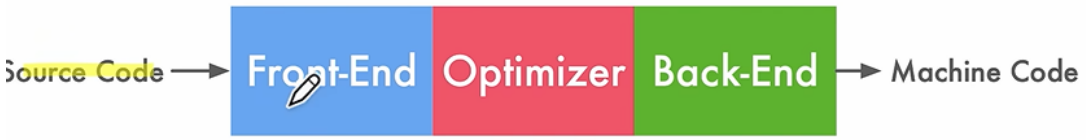
2、开源编译器简介

编译器：将高级语言转换为二进制代码的机器语言



编译器执行的过程

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；



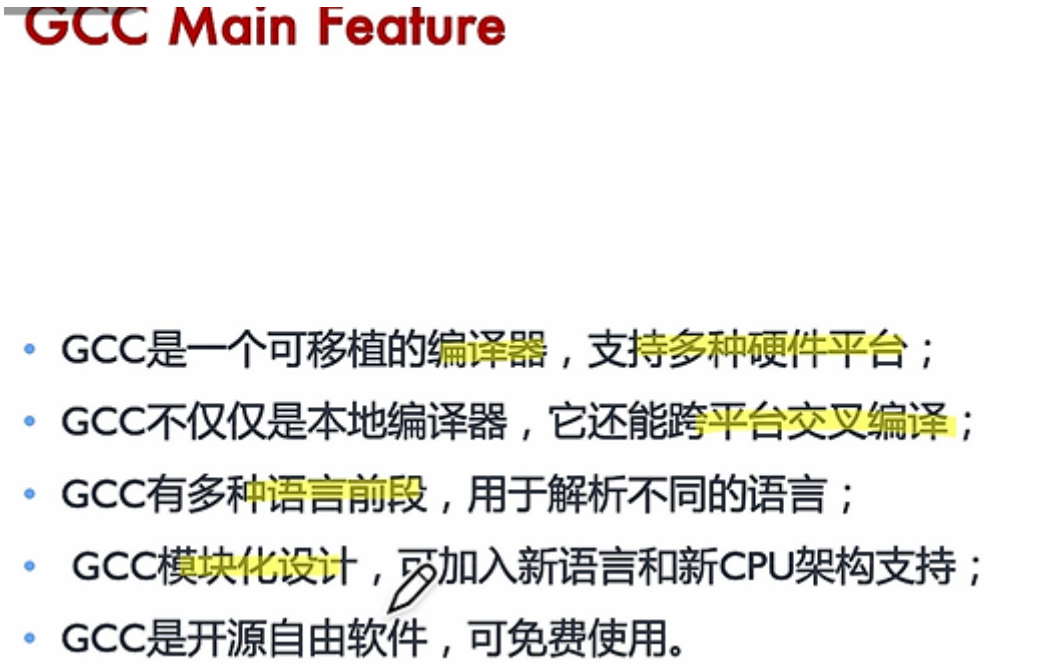
- Optimizer：优化器则是在前端的基础上，对得到的中间代码进行优化，使代码更加高效；
- Back-end：后端则是将已经优化的中间代码转化为针对各自平台的机器代码；

GCC编译器主要基于GNU或Linux

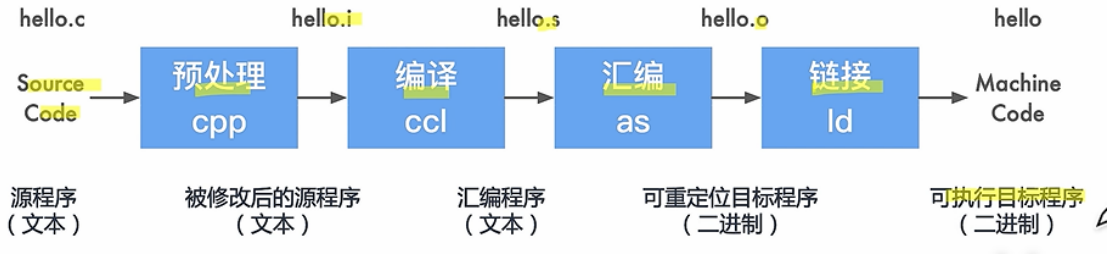
LLVM基于苹果操作系统

微软的是基于Visual　studio的

3、GCC编译过程



1、GCC编译流程



①预处理：包括宏定义，文件包含，条件编译三部分。预处理过程读入源代码，检查包含预处理指令的语句和宏定义，并对其进行响应和替换。预处理过程还会删除程序中的注释和多余空白字符，最后 生成.i文件。

```
1  #include<stdio.h>
2
3  #define HELLOWORLD ("Hello World\n")   //宏定义
4
5  int main(){
6      printf(HELLOWORLD);
7      return 0;
8  }
```

```
问题   输出   调试控制台   终端
    + FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\bai\project\C++\first_C++> gcc -E .\hello1.cpp -o .\hello.i
```

进行预处理之后

```
944
945  # 5 ".\\hello1.cpp"
946  int main(){
947      printf(("Hello World\n"));
948      return 0;
949  }
950
```

②编译

通过编译器将.i文件编译后形成汇编指令。

- **编译器（Compiling）**：编译器会将预处理完的 .i 文件进行一些列的语法分析，并优化后生成对应的汇编代码。会生成 .s 文件。

```
PS C:\Users\bai\project\C++\first_C++> gcc -S .\hello.i -o .\hello.s
```

```
main:
    pushq   %rbp
    .seh_pushreg    %rbp
    movq    %rsp, %rbp
    .seh_setframe   %rbp, 0
    subq    $32, %rsp
    .seh_stackalloc 32
    .seh_endprologue
    call    __main
    leaq    .LC0(%rip), %rcx
    call    puts
    movl    $0, %eax
    addq    $32, %rsp
    popq    %rbp
    ret
    .seh_endproc
    .ident  "GCC: (x86_64-win32-seh-rev0, Built by MinGW-W
    .def    puts;   .scl    2;  .type   32; .endef
```

③汇编：

汇编器：汇编器会将编译器生成的.s汇编程序汇编为机器语言或指令，也就是机器可以执行的二进制程序。会生成.o文件 二进制文件

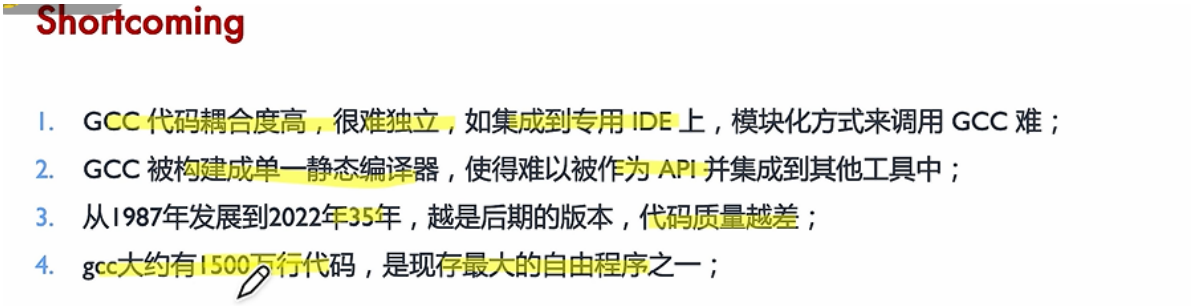
```
gcc -c hello.s -o hello.o
```

④链接：

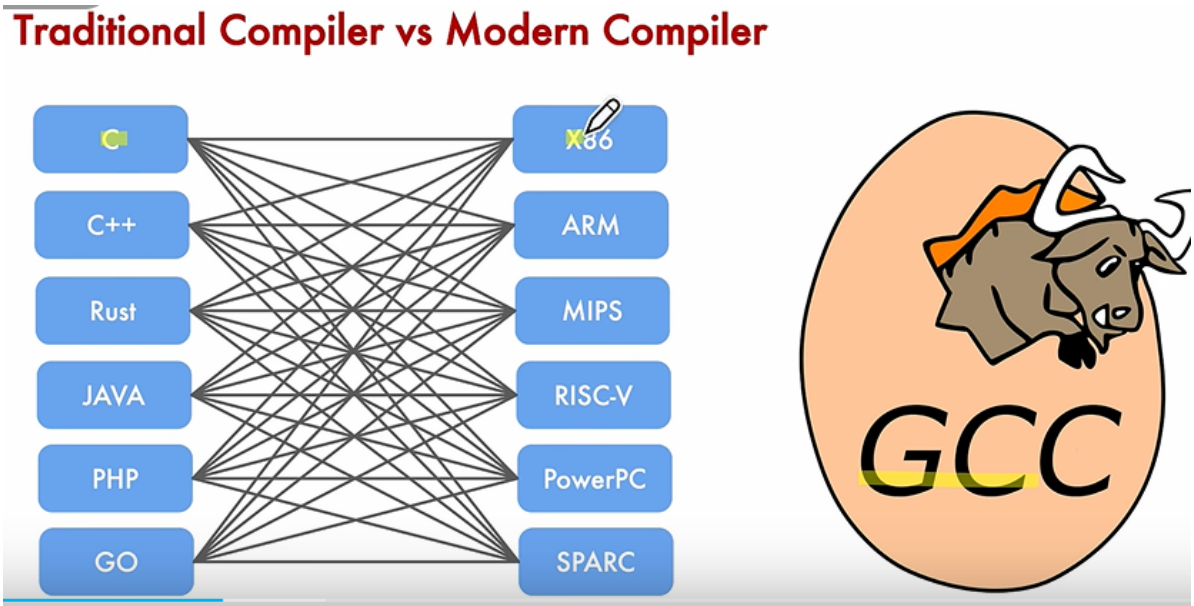
连接器会来链接程序运行所需要的目标文件，以及依赖的库文件，最后生成可执行的文件，以二进制的形式存在磁盘中。因为可执行的程序可能用到的不仅仅一个目标文件。

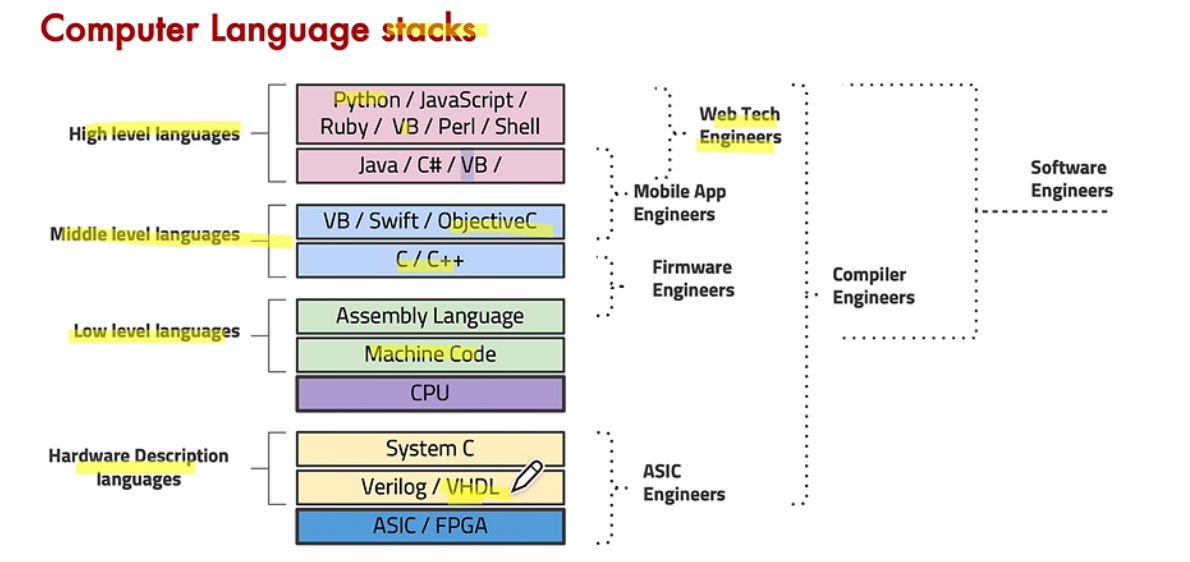
- **链接器（Linking）：链接器会来链接程序运行的所需要的目标文件，以及依赖的库文件，最后生成可执行文件，以二进制形式存储在磁盘中。**

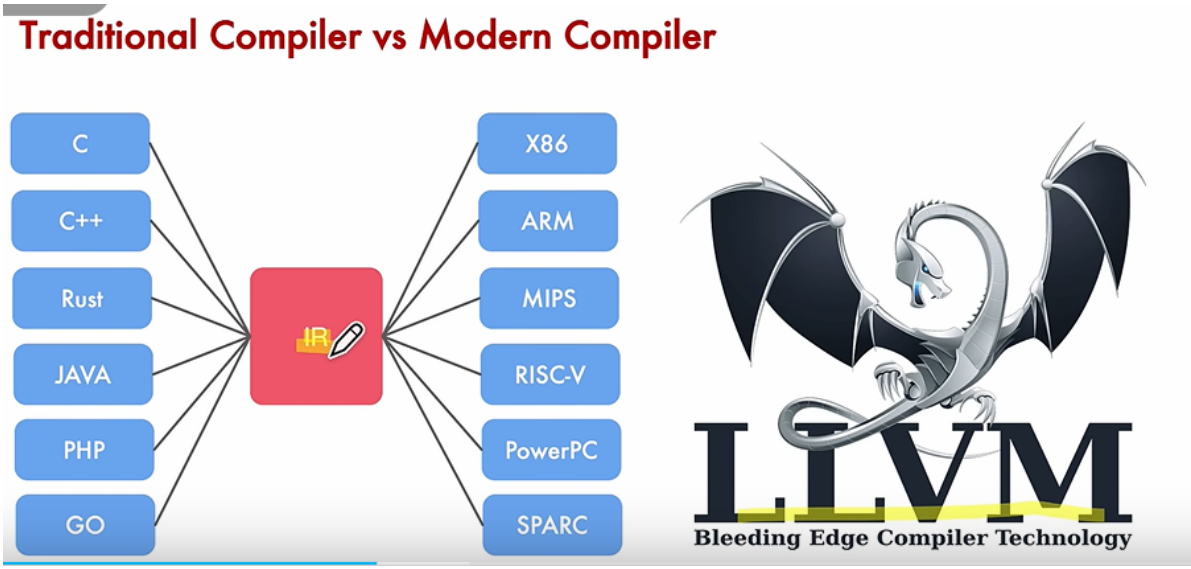
```
PS C:\Users\bai\project\C++\first C++> gcc hello.o -o hello
```

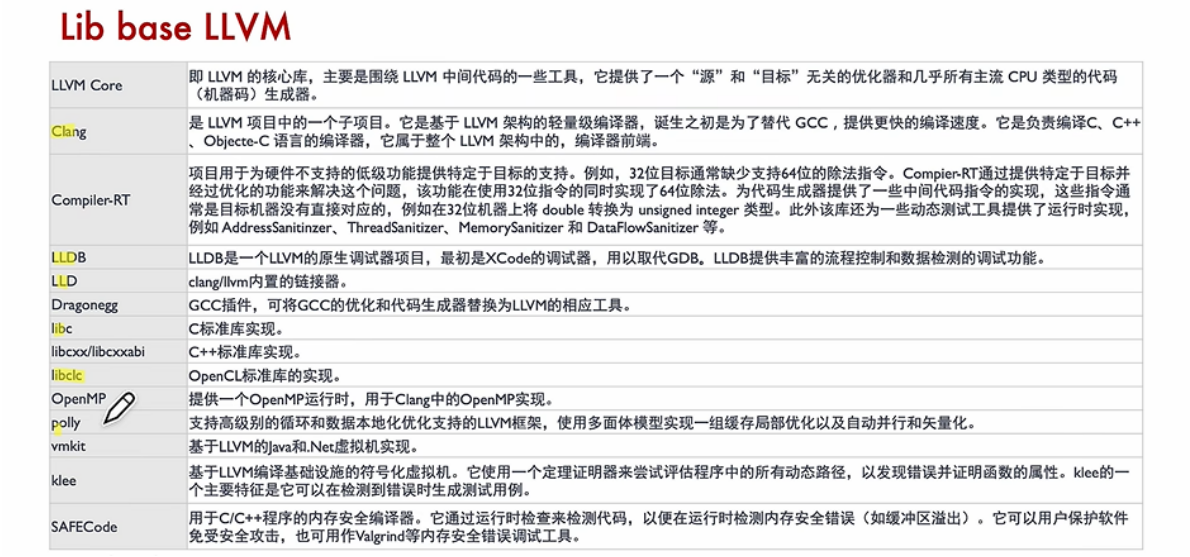


04LLVM架构和原理









LLVM Core	即 LLVM 的核心库，主要是围绕 LLVM 中间代码的一些工具，它提供了一个“源”和“目标”无关的优化器和几乎所有主流 CPU 类型的代码（机器码）生成器。
Clang	是 LLVM 项目中的一个子项目。它是基于 LLVM 架构的轻量级编译器，诞生之初是为了替代 GCC，提供更快的编译速度。它是负责编译C、C++、Objecte-C 语言的编译器，它属于整个 LLVM 架构中的，编译器前端。
Compiler-RT	项目用于为硬件不支持的低级功能提供特定于目标的支持。例如，32位目标通常缺少支持64位的除法指令。Compier-RT通过提供特定于目标并经过优化的功能来解决这个问题，该功能在使用32位指令的同时实现了64位除法。为代码生成器提供了一些中间代码指令的实现，这些指令通常是目标机器没有直接对应的，例如在32位机器上将 double 转换为 unsigned integer 类型。此外该库还为一些动态测试工具提供了运行时实现，例如 AddressSanitinzer、ThreadSanitizer、MemorySanitizer 和 DataFlowSanitizer 等。
LLDB	LLDB是一个LLVM的原生调试器项目，最初是XCode的调试器，用以取代GDB。LLDB提供丰富的流程控制和数据检测的调试功能。
LLD	clang/llvm内置的链接器。
Dragonegg	GCC插件，可将GCC的优化和代码生成器替换为LLVM的相应工具。
libc	C标准库实现。
libcxx/libcxxabi	C++标准库实现。
libclc	OpenCL标准库的实现。
OpenMP	提供一个OpenMP运行时，用于Clang中的OpenMP实现。
polly	支持高级别的循环和数据本地化优化支持的LLVM框架，使用多面体模型实现一组缓存局部优化以及自动并行和矢量化。
vmkit	基于LLVM的Java和.Net虚拟机实现。
klee	基于LLVM编译基础设施的符号化虚拟机。它使用一个定理证明器来尝试评估程序中的所有动态路径，以发现错误并证明函数的属性。klee的一个主要特征是它可以在检测到错误时生成测试用例。
SAFECode	用于C/C++程序的内存安全编译器。它通过运行时检查来检测代码，以便在运行时检测内存安全错误（如缓冲区溢出）。它可以用户保护软件免受安全攻击，也可用作Valgrind等内存安全错误调试工具。

什么是LLVM

> 可以说LLVM是一个编译器

What is LLVM

- LLVM is a Compiler
- LLVM is a Compiler Infrastructure
- LLVM is a series of Compiler Tools
- LLVM is a Compiler Toolchain
- LLVM is an open source C++ implementation
- LLVM 项目发展为一个巨大的编译器相关的工具集合