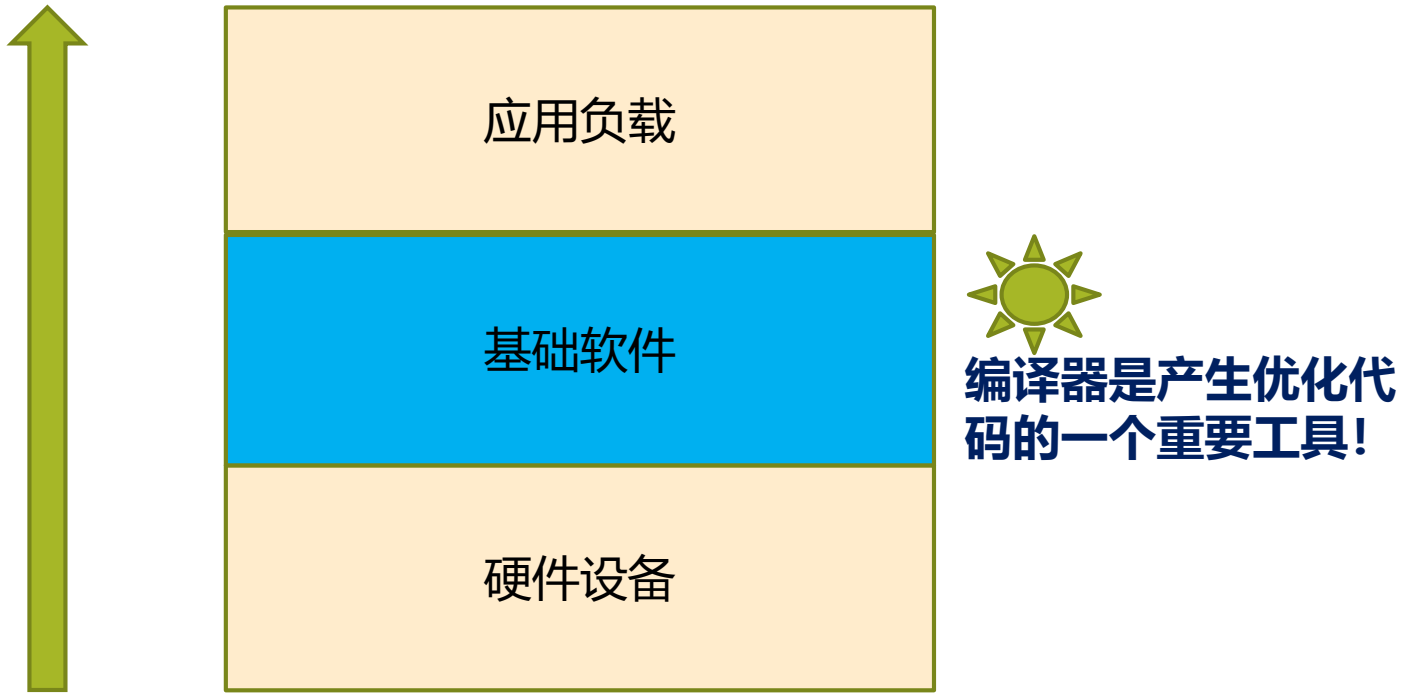
编译器概述

黄 波
bhuang@dase.ecnu.edu.cn

SOLE 系统优化实验室
华东师范大学

本次课的关注点

Scale up
全栈思维

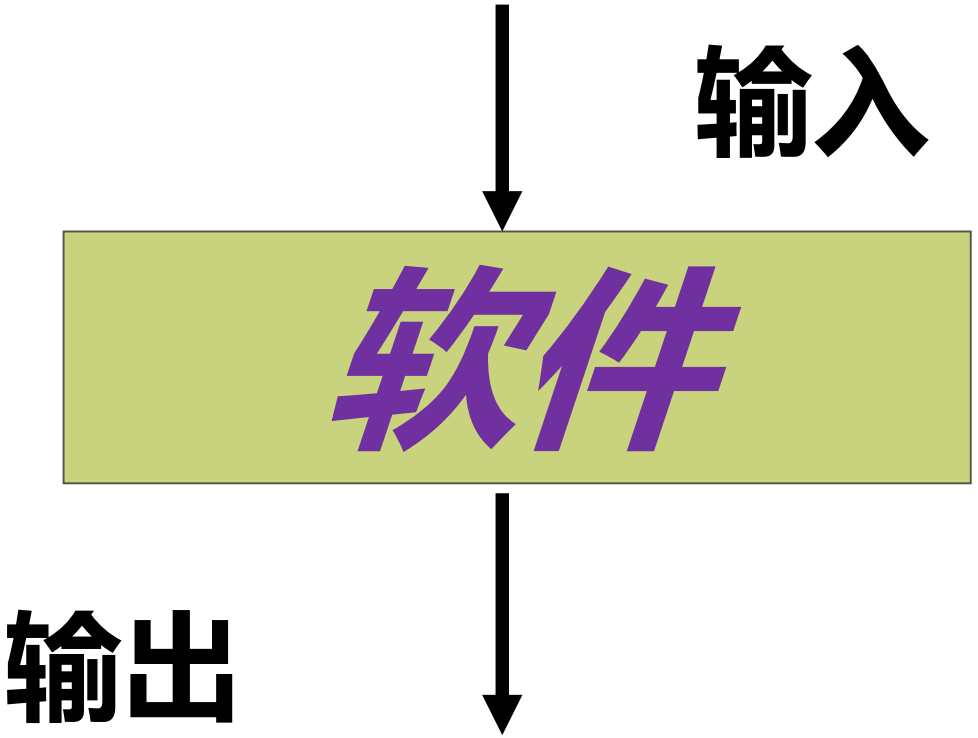


编译器是产生优化代码的一个重要工具！

本次课程的内容

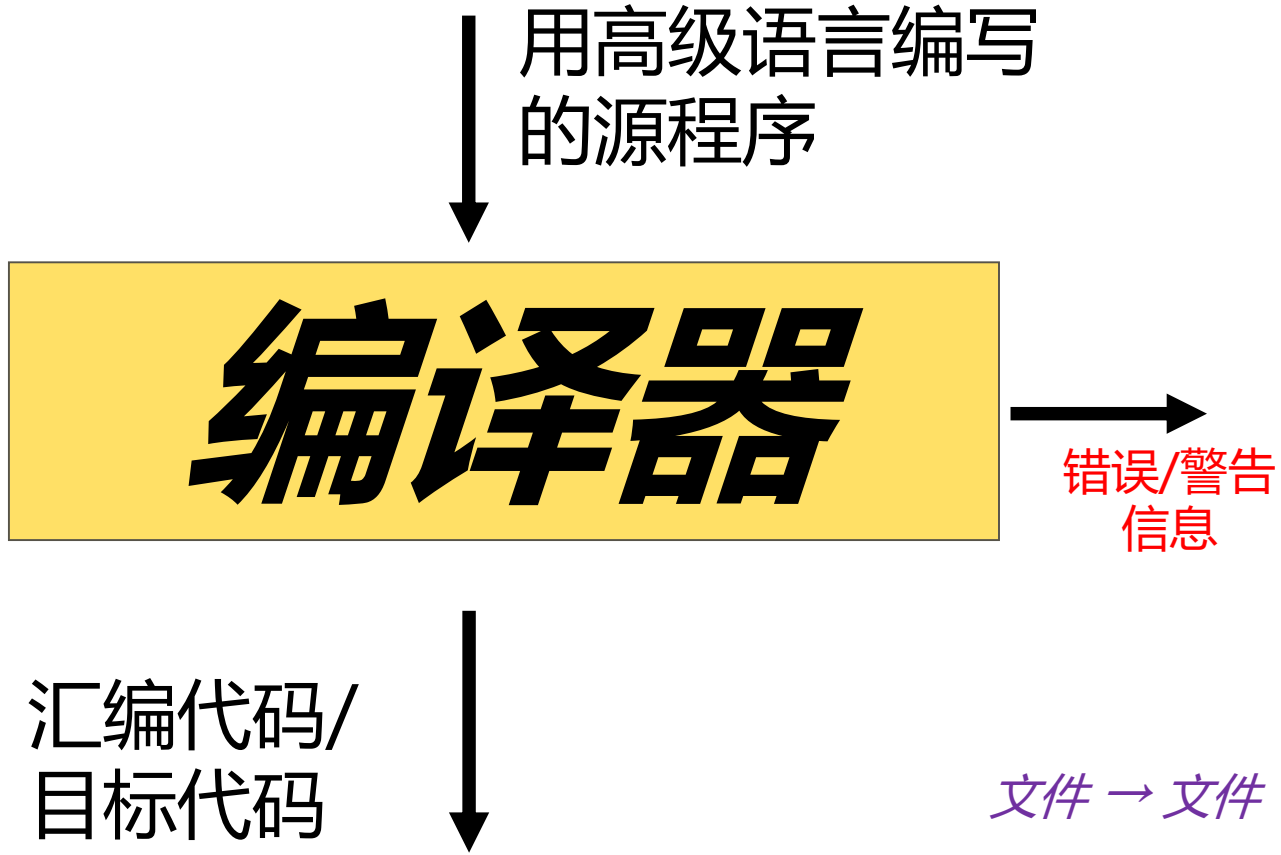
- 编译器的定义与分类
- 程序的中间表示
- 符号表
- 程序运行时的内存组织
- 程序分析和优化
- 交叉编译
- 用编译器优化程序的迭代循环

软件



Q: 请枚举一下软件的输入输出有哪些形式?

编译器的传统定义



文件 → 文件

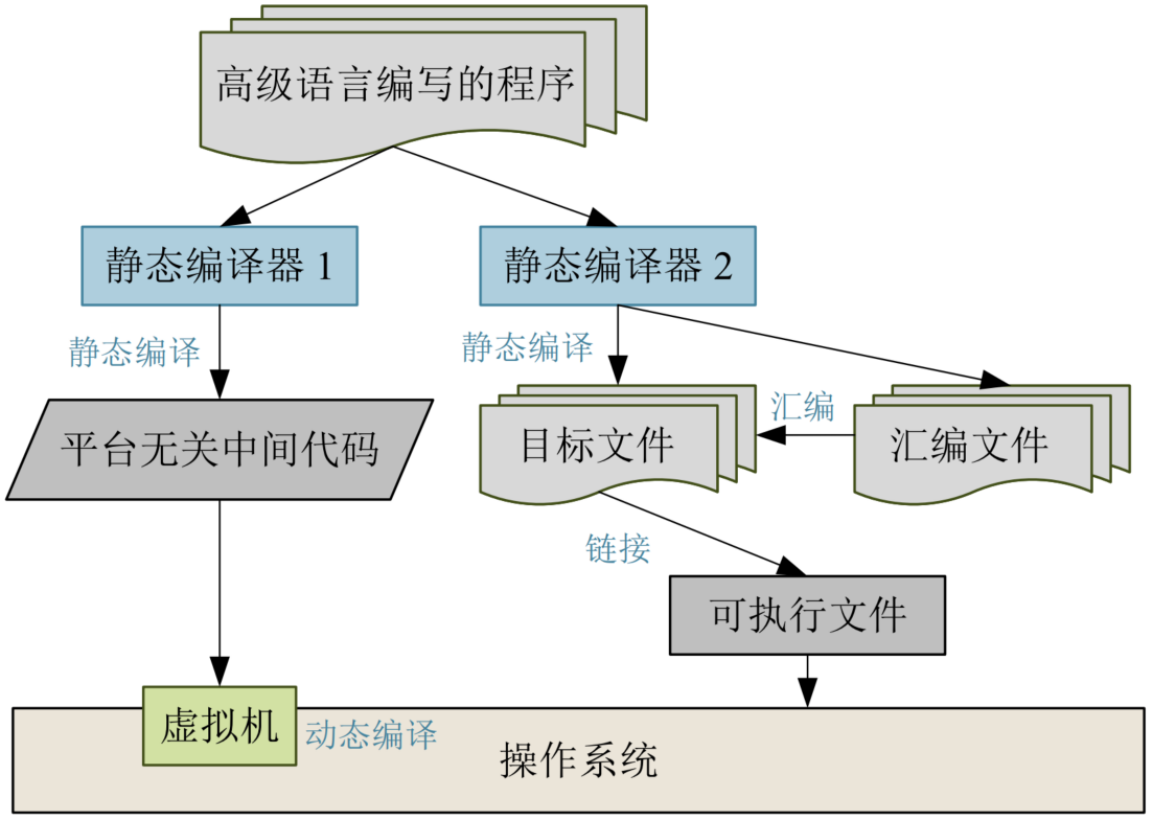
编译器的广义定义

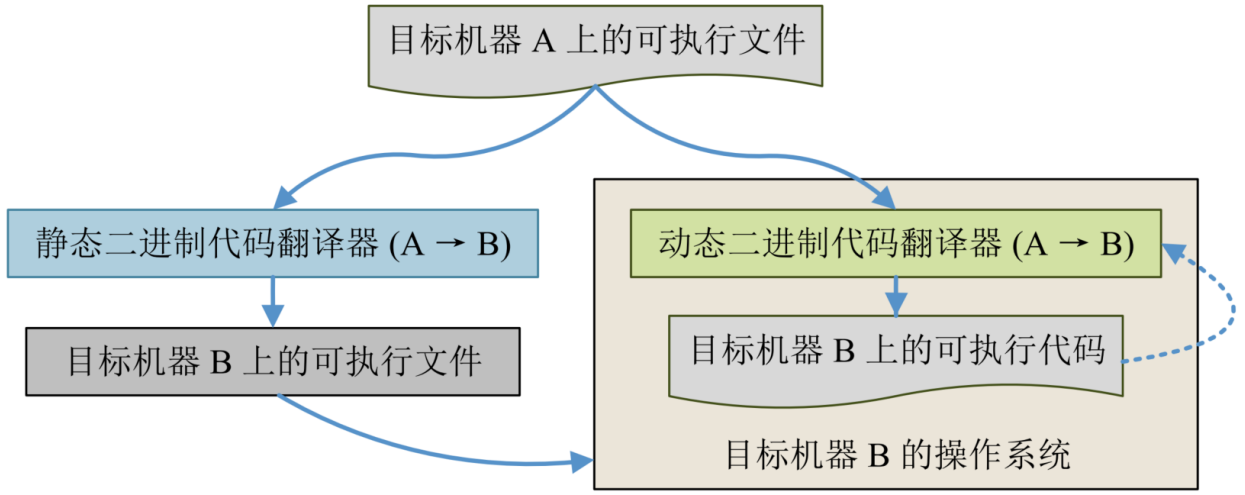
从广义来说编译器可以认为是进行**语义等价**程序变换的软件系统，这个程序变换包括如下几种类型：

- 把用高级语言编写的程序转换成
  - 目标代码
  - 汇编程序
  - 中间语言/中间表示
  - 用另外一种高级语言编写的程序
- 把中间语言转换成目标平台上能执行的代码
- 把一种汇编程序转换成另外一种汇编程序
- 把一种二进制执行码转换成另外一种二进制执行码

Q: 请举一些课堂上没有讲过的广义编译器的例子？

静态编译器 vs. 动态编译器



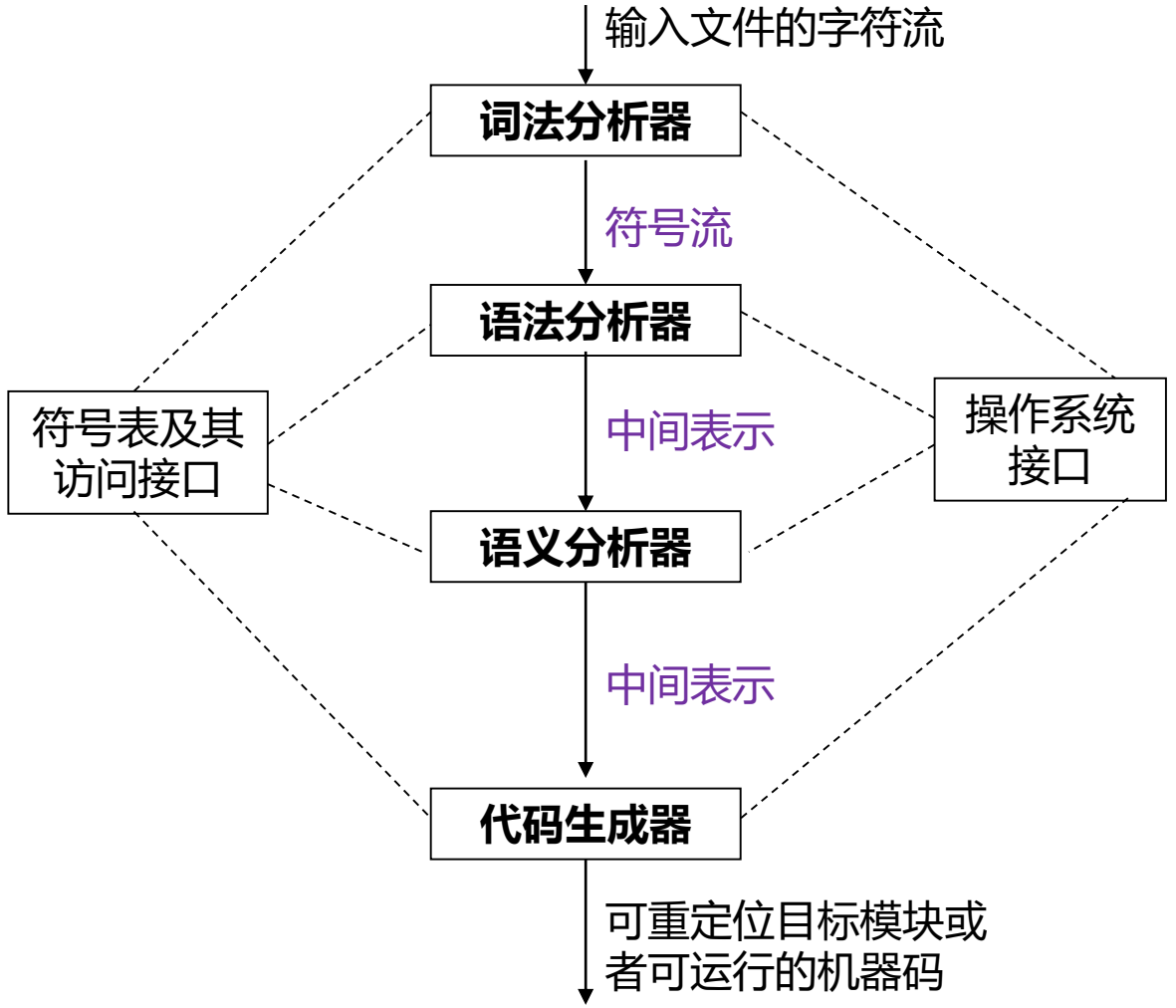


静态：*编译器生成代码，所生成代码的运行在编译之后*
动态：*编译器的运行和所生成代码的运行"同时"发生*

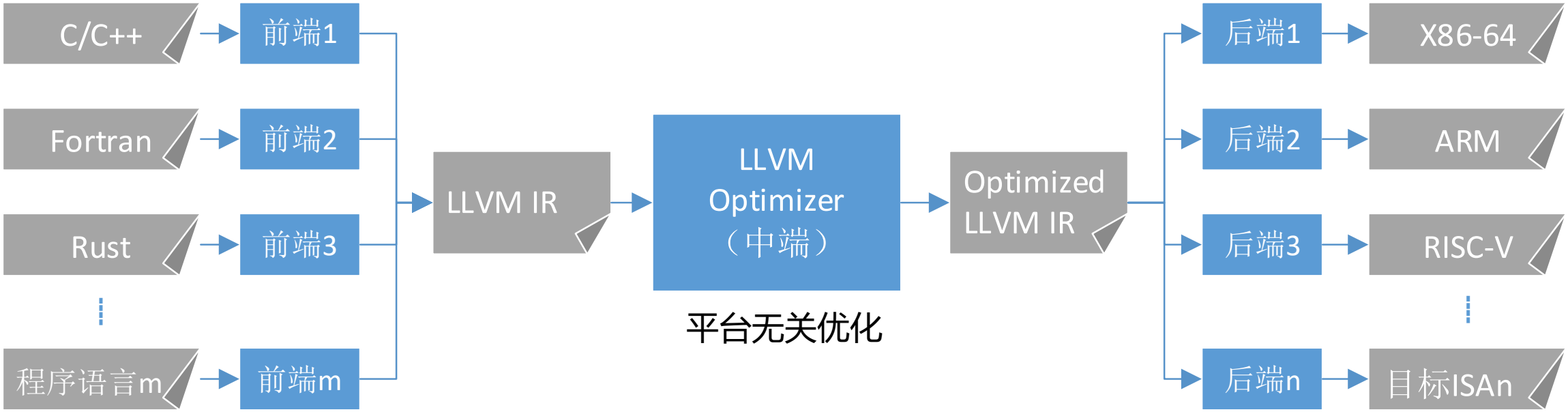
编译器 vs. 解释器

- 什么是解释器?
- 编译器和解释器有何异同点?

一个简单编译器的高层结构



LLVM编译器的基本框架



为什要有程序中间表示（IR）？

- 程序的中间表示既是编译器进行程序分析与优化的对象，同时也是编译器进行程序变换与优化的结果呈现，它是编译器基础设施的一个重要组成部分
- 通过一定的抽象使得IR不带有源程序所用高级语言的细节以及目标体系架构的细节，从而做到中立，即语言无关和平台无关
- 不同的程序中间表示有不同的优缺点，在编译器及编译器基础设施的设计与实现中需要根据实际需求选择合适的程序中间表示

程序中间表示的设计思考

- 被编译的程序语言
- 目标平台
- 做哪些优化
- 是否平台相关/无关
- IR结构及可表达性
- 一种IR还是多种IR
- 调试功能
- 导出/重新加载功能

- **高层IR（HIR）**：HIR通常可以比较直观地表达高级语言的语法与语义，所以在整个编译过程的早期阶段或者程序预处理阶段用得比较多，基于HIR比较容易进行相关性分析及高层优化。抽象语法树是一种常见的HIR
- **中层IR（MIR）**：MIR一般具有语言无关及平台无关的特性，很多编译优化都可以基于MIR进行
- **低层IR（LIR）**：LIR一般都与具体的目标平台相关，基于LIR比较容易进行目标指令集架构相关的底层优化

LLVM IR：LLVM的程序中间表示

LLVM IR本身可以看成是一种程序语言，它比较接近于底层中间表示，跟C比较相似。简单来说LLVM IR包括模块、函数、基本块、指令等语法单位，并支持整数、浮点数、指针、标号、数组、结构、向量等多种**数据类型**（Data types）：

- **整数**：用i<位数>的形式表示，e.g. i64，i1
- **浮点数**：float表示32位的浮点数类型，double表示64位的浮点数类型
- **指针**：用<类型>*表示，具体表示指向<>里面那个类型的指针，e.g. i8*
- **标号**：在LLVM IR中标记一个基本块（basic block）的入口
- **数组**：[<数目> x <类型>]，e.g. [5 x i32]
- **结构**：{类型列表}，e.g. {i32, float}
- **向量**：< <数目> x <类型> >，e.g. <10 x i8>

寄存器：%<名字>，其中名字可以是数字，也可以是标识符

指令

(a) <目标操作数> = <操作码> <源操作数列表>
(b) <操作码> <操作数列表>

LLVM IR 示例

$clang –O1 fib.c -c -emit-llvm

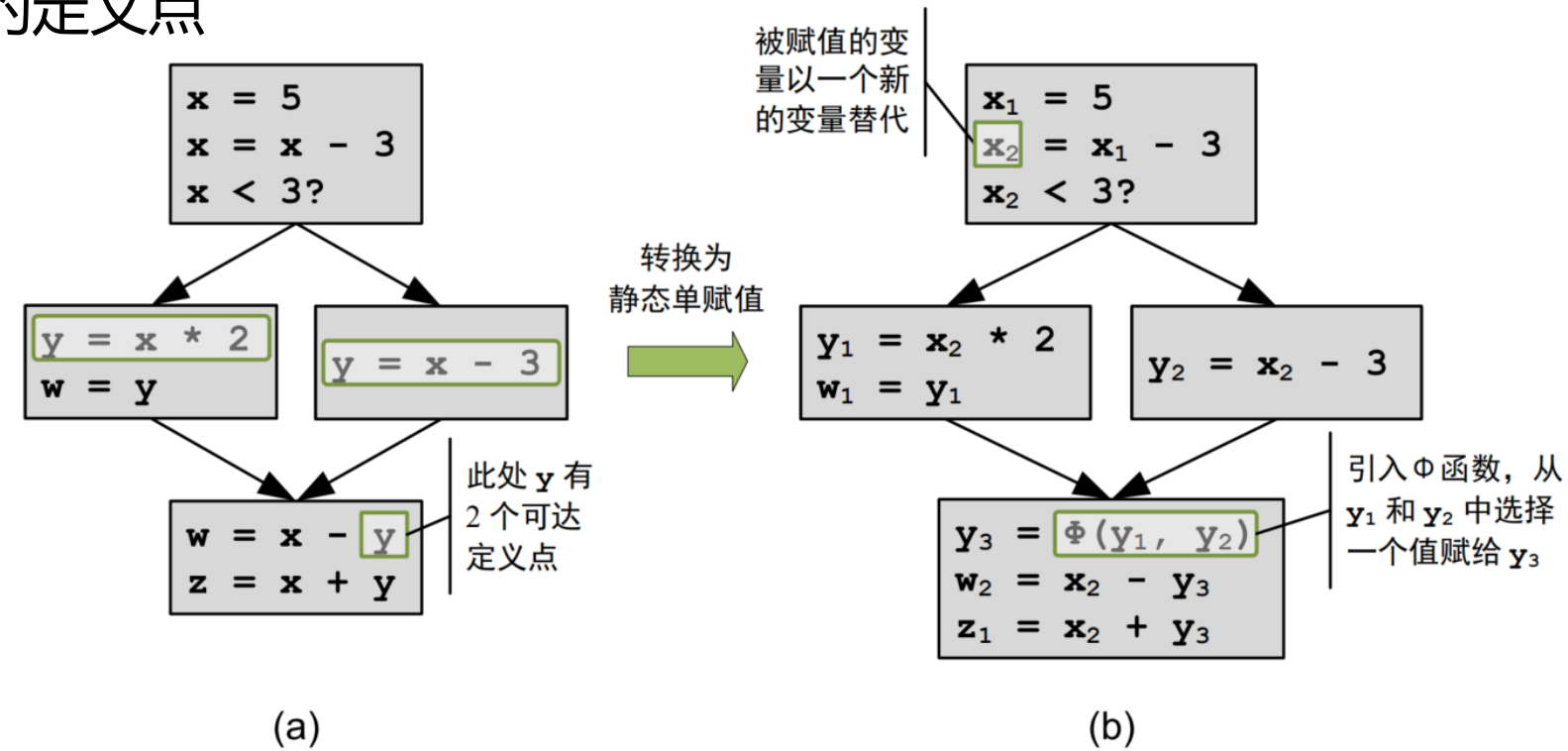
```
int64_t fib(int64_t n) {
  if (n < 2)
        return n;
  return (fib(n-1) + fib(n-2));
}
```

```
; Function Attrs: nounwind readnone uwtable
define dso_local i64 @fib(i64 %0) local_unnamed_addr #0 {
  %2 = icmp slt i64 %0, 2
  br i1 %2, label %9, label %3

3:                                                ; preds = %1
  %4 = add nsw i64 %0, -1
  %5 = call i64 @fib(i64 %4)
  %6 = add nsw i64 %0, -2
  %7 = call i64 @fib(i64 %6)
  %8 = add nsw i64 %7, %5
  br label %9

9:                                                ; preds = %1, %3
  %10 = phi i64 [ %8, %3 ], [ %0, %1 ]
  ret i64 %10
}
```

静态单赋值（SSA）

- LLVM IR内的每个变量都只有唯一的一个可达定义点（reaching definition）
- 当对同一个变量的两个或多个定义汇聚时，需要引入一个Φ函数(Φ-function)来产生一个新的定义点

```
x = 5
x = x - 3
if x < 3:
    y = x * 2
    w = y
else:
    y = x - 3
w = x - y
z = x + y
```



Q: Φ函数在代码生成时该如何处理？

欢迎进入LLVM IR的世界（自学篇）

LLVM Home | Documentation » Reference »

LLVM Language Reference Manual

https://llvm.org/docs/LangRef.html

符号表

- 源程序中的变量名和函数名都会被存储在符号表内
- 符号表内的每一个符号都有它的作用范围
- 一个符号从首次可见到最后一次可见之间的间隔叫做这个符号的生命周期
- 访问符号表的便捷性和效率对编译器的效率起着至关重要的作用
- 在编译器进行代码生成时，符号表中的符号变量最终会跟某个存储空间绑定。特别地，编译器把全局变量分配到全局数据区内，而把局部变量分配到此变量所属函数的运行栈（Stack）内。在编译器进行优化时，也可能把符号变量跟某个寄存器进行绑定

Q: 符号/变量的可见范围一定连续么？

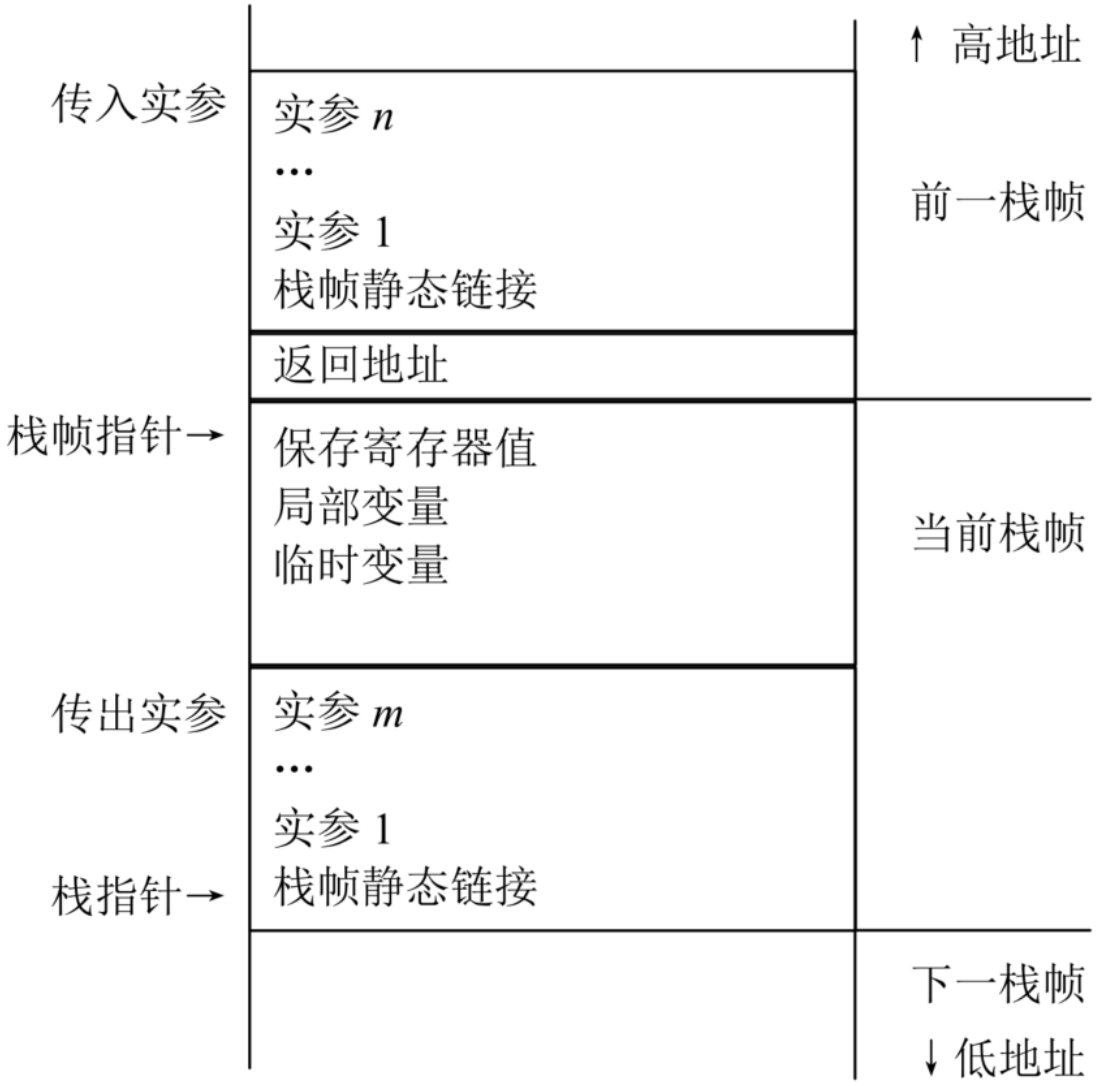
示例：同名变量的可见范围

Q: 这个程序的运行结果是什么?

```c
#include <stdio.h>
int main() {
    int value = 0;
    for(int i = 0; i < 10; i++) {
        int sum = 0;
        for(int j = 0; j < 10; j++) {
            int sum = j;
            sum += j;
            value += sum;
        }
        value += sum;
    }
    printf("value = %d \n", value);
    return 0;
}
```

程序运行时的内存组织

Q: 对栈内变量的访问方式?



存储绑定

名字->地址

- 全局变量
  - 确定的可重定向的地址
  - 全局指针 + 偏移量
  - 寄存器
- 本地变量
  - 栈指针 + 偏移量*
  - 帧指针 – 偏移量*
  - 寄存器

*此处偏移量为正

影响内存布局的因素

- 偏移量限制
- 对齐方式
- 访问的邻近性
- 代码大小

符号寄存器

- 通过寄存器分配来映射到物理寄存器

编译器优化示例

(a) 源程序
(b) clang -O0 产生的LLVM IR
(c) clang -O1 产生的LLVM IR
(d) clang -O2 产生的LLVM IR

```c
int f(int a, int b) {
    return a + b;
}

int main() {
    int c = 1, d = 2;
    return f(c, d);
}
```

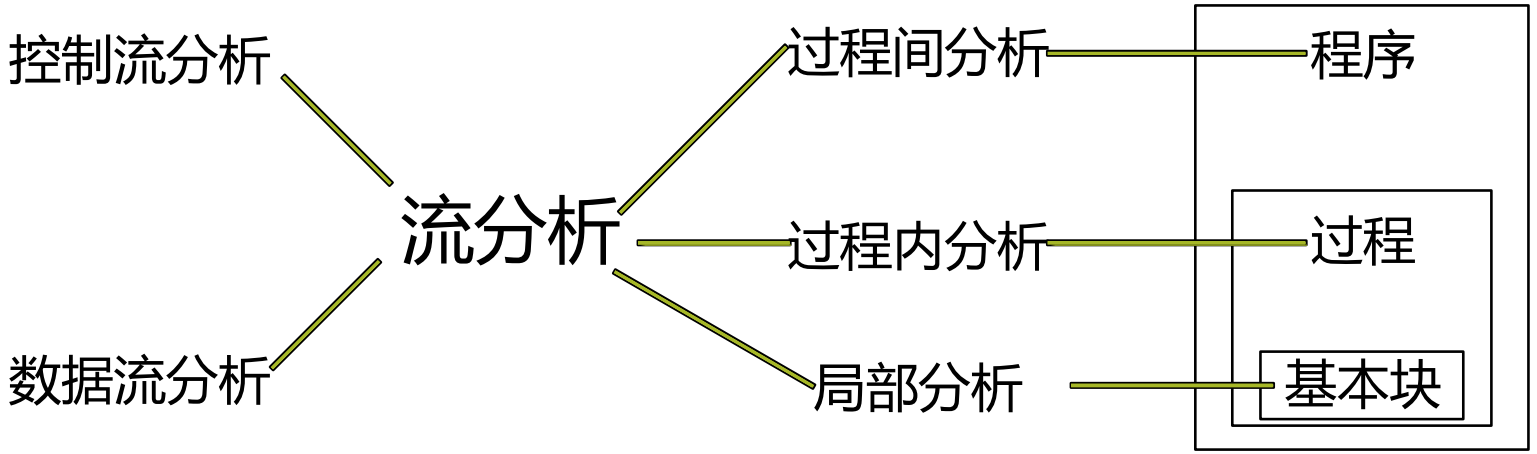
```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @f(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw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

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4
  store i32 2, i32* %3, align 4
  %4 = load i32, i32* %2, align 4
  %5 = load i32, i32* %3, align 4
  %6 = call i32 @f(i32 %4, i32 %5)
  ret i32 %6
}
```

```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @f(i32 %0, i32 %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
  %1 = call i32 @f(i32 1, i32 2)
  ret i32 %1
}
```

```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @f(i32 %0, i32 %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local i32 @main() local_unnamed_addr #0 {
  ret i32 3
}
```

程序分析技术

- 理论上说，程序分析进行得越彻底，所得到的程序信息则越准确，被分析程序可以被优化的潜在机会相应就越多
- 对程序的分析从大类上分为：
  - **控制流分析**：通过构建控制流图来确定被分析程序的控制结构
  - **数据流分析**：基于控制流图来分析变量或者表达式的值如何沿着各控制路径进行传播



程序优化

- **本质**：保证程序**语义等价**前提下的程序变换
- **可能效果**：性能提升、资源占用减少、能耗降低
- **发生阶段**：在编译器前端、中端和后端都可以进行不同的优化
- **顺序相关**：如果按照不同的顺序来进行多个优化，优化效果可能会有差异
- **中间表示相关**：有些优化需要特定的IR形式，或者在特定IR下会更加高效

交叉编译

定义

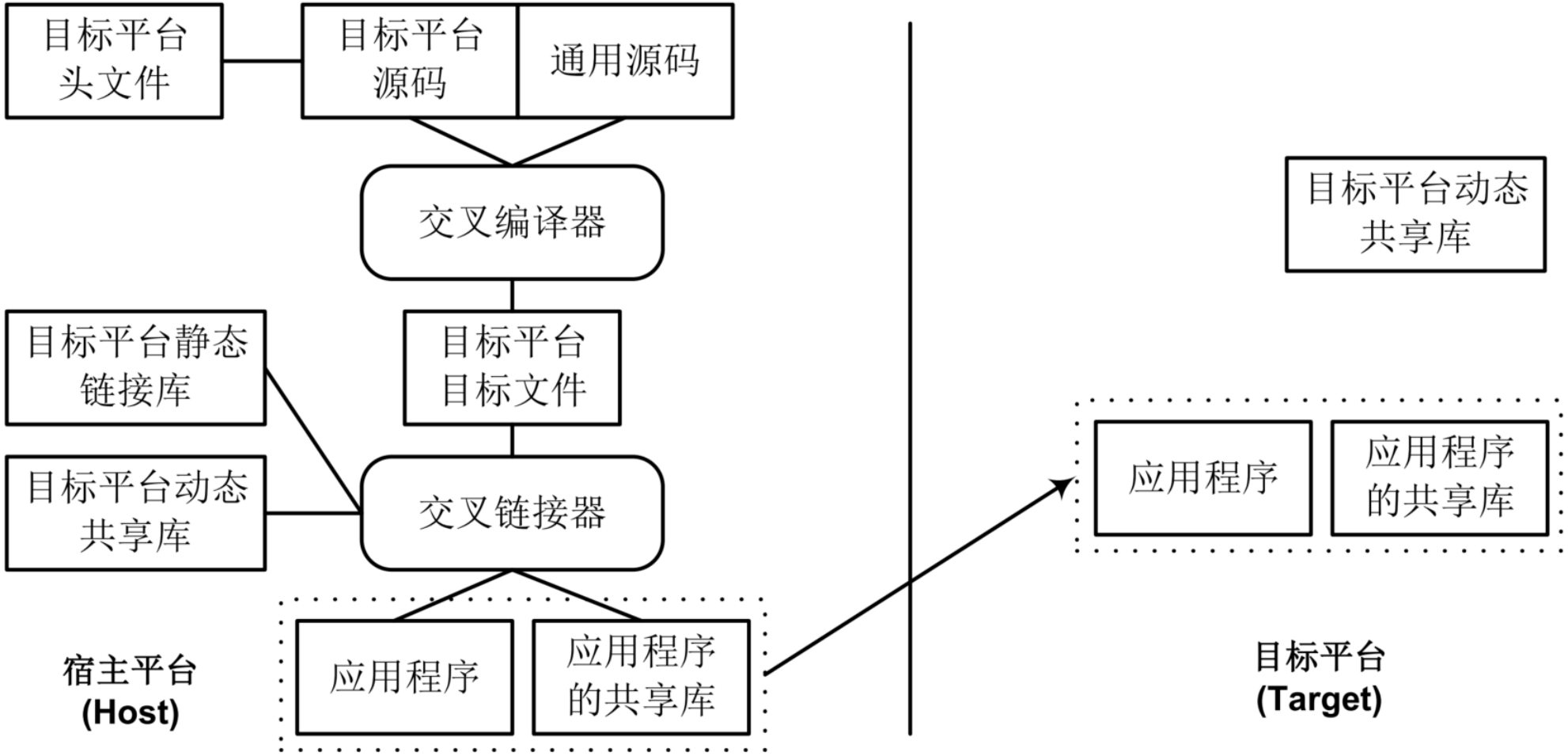
- **宿主平台（Host）**
  开发平台（运行软件开发所需的开发工具的平台）
- **目标平台（Target）**
  运行开发出来的应用程序的平台
- **本地编译（Native compilation）**
  宿主平台和目标平台相同或者兼容
- **交叉编译（Cross compilation）**
  宿主平台与目标平台不相同或者不兼容

动机

- 宿主平台比目标平台的算力高
- 有些目标平台的算力及内存不足以运行一个完整的编译器，甚至没有操作系统
- 需要为多个目标平台开发程序
- 构建目标平台的初始编译器必需在别的开发平台上进行

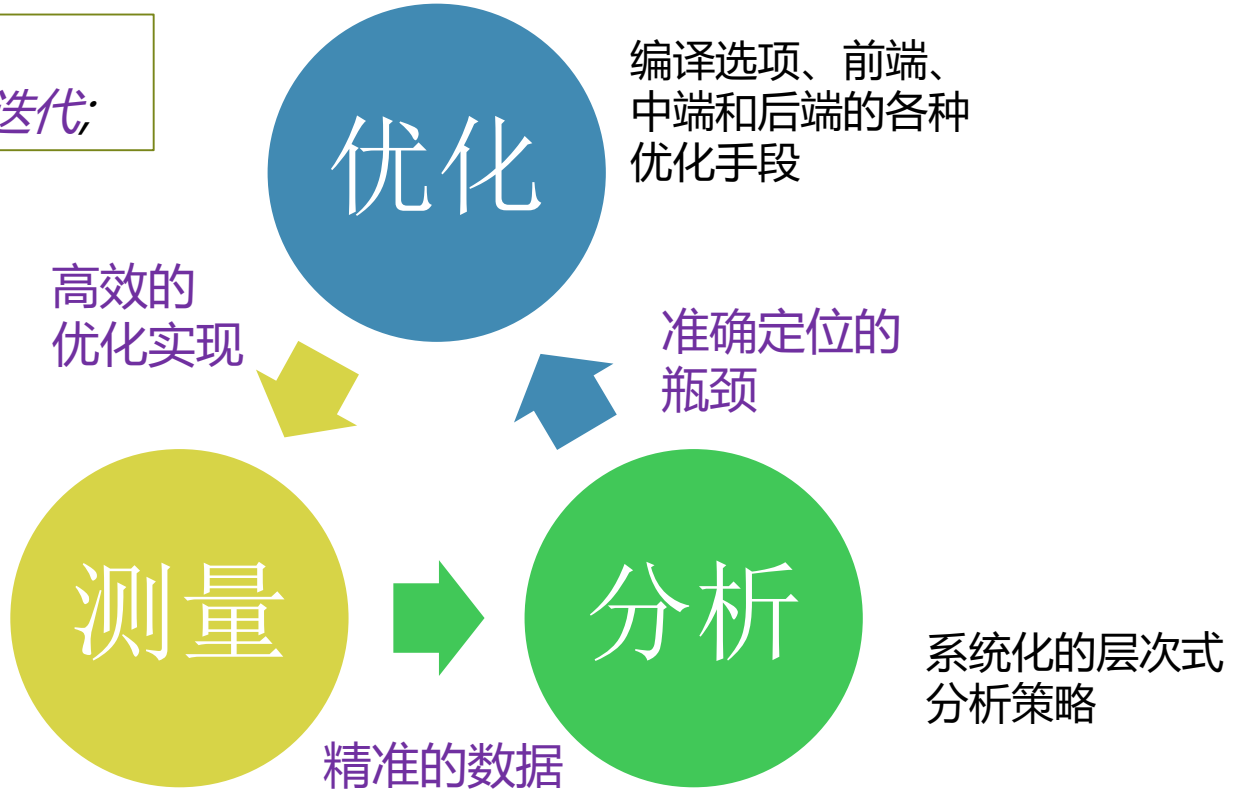
Q: 你理解中的平台包含哪些方面？

用交叉编译开发程序的流程示意图



编译器优化的迭代循环

```
while (优化目标未达到)
    进行一轮优化的循环迭代;
```



本次课程总结

- 传统编译器是一个系统软件，它能把用高级语言编写的源程序转换成目标平台上的可执行代码
- 一个编译器通常包括编译器前端、编译器中端和编译器后端
- 程序分析和程序优化是基于程序的中间表示来进行的
- 符号表是编译器基础设施内的一个重要数据结构
- 理解存储绑定以及函数调用规范对理解编译器所生成的代码非常关键
- 编译器可以分为本地编译器和交叉编译器
- 程序的优化是一个循环迭代的过程，直到程序优化目标达成


Q
Q
&
&
A
A

为什么这些公司需要搭建编译团队？

公司类别	公司举例	为什么这些公司要组建编译优化团队？
芯片	Intel, AMD, Nvidia, ARM、Ampere、壁仞、海光、平头哥、瑞芯微、寒武纪 、地平线…	围绕芯片构建软件生态的刚需； 提升性能/效能以提升芯片的竞争力
系统	Apple、Microsoft、Honeywell、华为、中兴、VIVO、OPPO、小米、大疆…	软硬协同，确保系统竞争力
互联网	Google、Amazon、腾讯、阿里、百度、字节跳动、网易、…	提升数据中心的运营效率、提升用户体验； 提升公司内部的软件开发效率
软件、服务	Cadence、MathWorks、Github、蚂蚁金服、麒麟软件、爱加密、…	提升软件和服务的竞争力，提升用户体验； 提升内部工作效率
智能驾驶	Apple、Tesla、Waymo、百度、NIO蔚来、…	端边云无缝协同；优化智能驾驶算法、提升智能驾驶芯片和SoC的利用效率、确保安全
人工智能	Sima.ai、商汤、云天励飞、深信科创、…	优化AI算法落地，提升方案竞争力； 自研深度学习框架，提升开发效率
…	…	…

理解编译器的产品需求

- 输入：源程序文件（何种语言）、中间代码、可执行文件?
- 输出：源程序文件（何种语言）、中间代码、可执行文件?
- 运行平台：本地编译、交叉编译?
- 目标平台：指令集架构、操作系统、仿真器/物理机?
- 指标：正确性、易用性、编译时间、性能、生成文件大小?
- 测试集：自研+基准测试程序?
- 发布形式：开源、部分开源、免费使用、许可形式?
- 时间节点：Pre-Alpha, Alpha, Beta, Golden?
- 支持方式：Bug汇报渠道、反馈时延、线下支持、更新方式?