先进编译实验室
Advanced Compiler

编译论坛 | 第一期

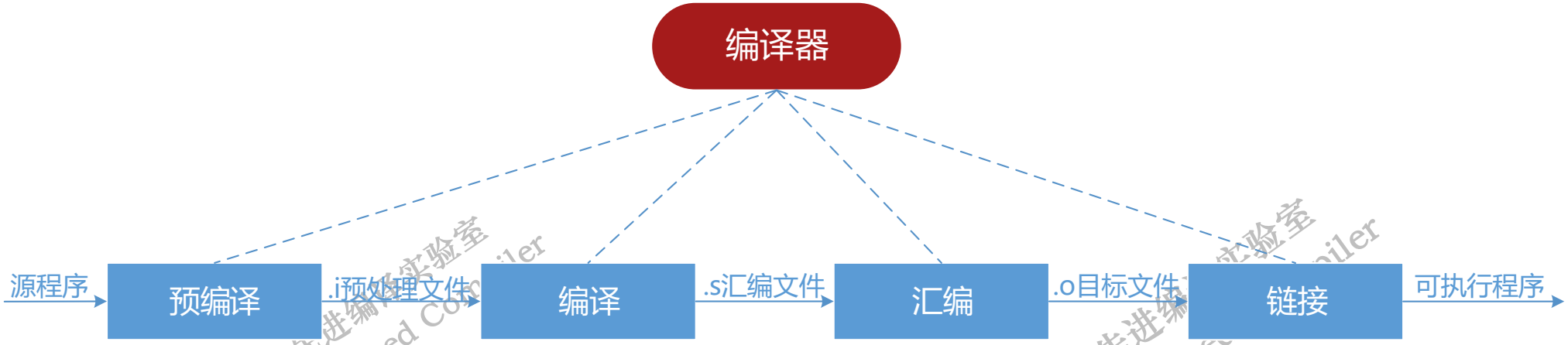
编译器概述

嘉宾：柴赟达

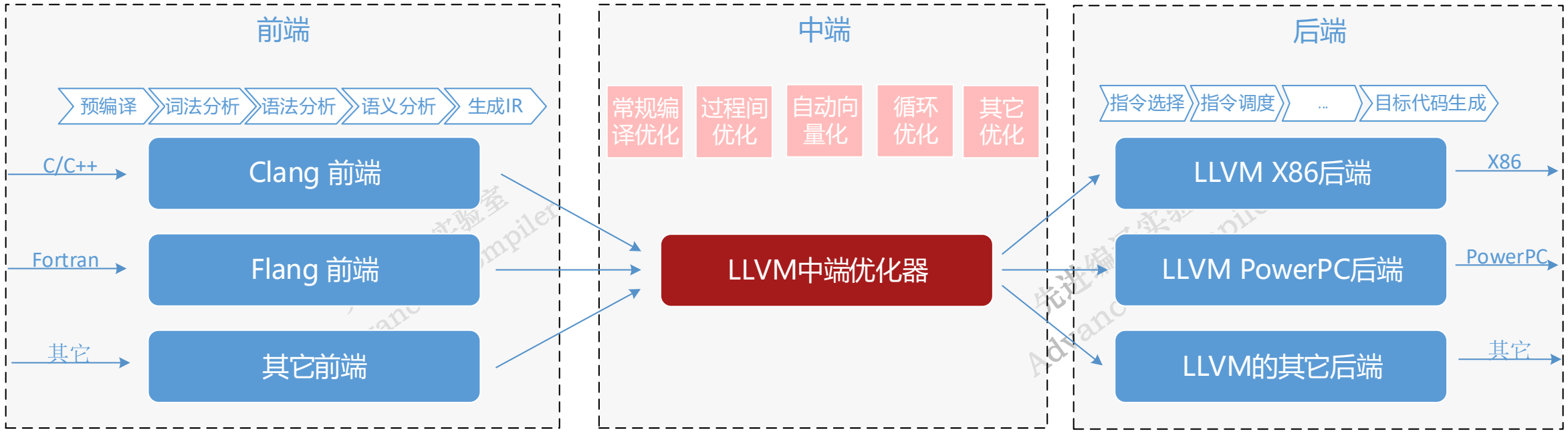
先进编译实验室
Advanced Compiler

分享内容

CONTENTS



01 序言-LLVM编译器架构



分享内容

CONTENTS

- **预编译**

预编译主要处理：

(1) 文件包含

(2) 宏展开

(3) 条件编译

(4) 删除注释

预编译所完成的基本上是对程序源代码的替代与整理工作，经过此类替代整理，生成一个没有宏定义、条件编译指令、特殊符号、用户注释的输出文件。

编译命令：clang -E hello.c -o hello.i

```
#include <stdio.h>
#define N 1024
int main ()
{
  int a,b,c,d[N];
  a = 2;
  b = 4;
  c = a + b*3;
  printf("Hello!");//test for pre compiler
  return c;
}
```

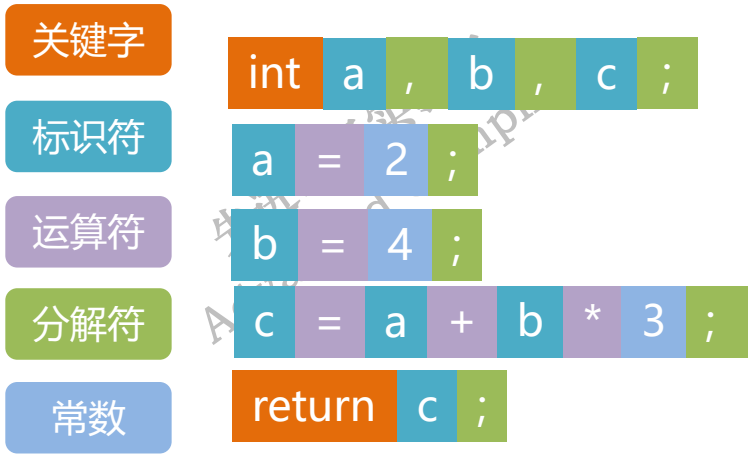
```
//hello.i
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;
……

int main ()
{
  int a,b,c,d[1024];
  a = 2;
  b = 4;
  c = a + b*3;
  printf("Hello!");
  return c;
}
```

- **词法分析**

词法分析器读入程序的源代码字符流，扫描、分解字符串，识别出一个个的单词，包括如下类型：

① 关键字，如int、float、if、 sizeof等；
② 标识符，用来表示各种名字，如变量、数组名、函数名等
③ 运算符，包括算术运算符、逻辑运算符、关系运算符等
④ 分解符，包括“，、；”等符号
⑤ 常数，包括整形、浮点型、字符型等



查看LLVM词法分析过程的编译命令

clang -fmodules -fsyntax-only -Xclang -dump-tokens file.c

- **语法分析**

语法分析的任务是将词法分析生成的单词组合成语法短语，同时分析这些短语是否符合高级程序设计语言中的语法规则。有下面的规则来定义表达式：

① 标识符是表达式；

② 常数是表达式；

③ 若表达式1和表达式2都是表达式，那么表达式1+表达式2以及表达式1 * 表达式2也都是表达式

有下面的规则来定义赋值语句：

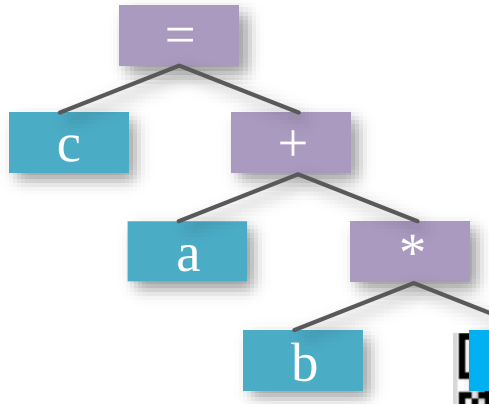
① <表达式> = n

② <表达式> = <表达式> “+” <表达式>

③ <表达式> = <表达式> “* ” <表达式>

④ <赋值语句> = <标识符> “=” <表达式>

其中语句c = a + b * 3，依据高级程序设计语言中约定的赋值语句及表达式的定义规则表示为抽象语法树形式。将字符串格式的源代码转化为树状的数据结构，更容易被计算机理解和处理。



查看LLVM语法分析过程的编译命令：

clang -fmodules -fsyntax-only -Xclang -ast-dump file.c

- **语义分析**

语义分析阶段的任务是审查源代码有无语义错误，源代码中有些语法成分，按照语法规则去判断是正确的，但不符合语义规则，比如使用了没有声明的变量。

语义分析主要的任务可归结为以下四类：

① 完成静态语义审查和处理；

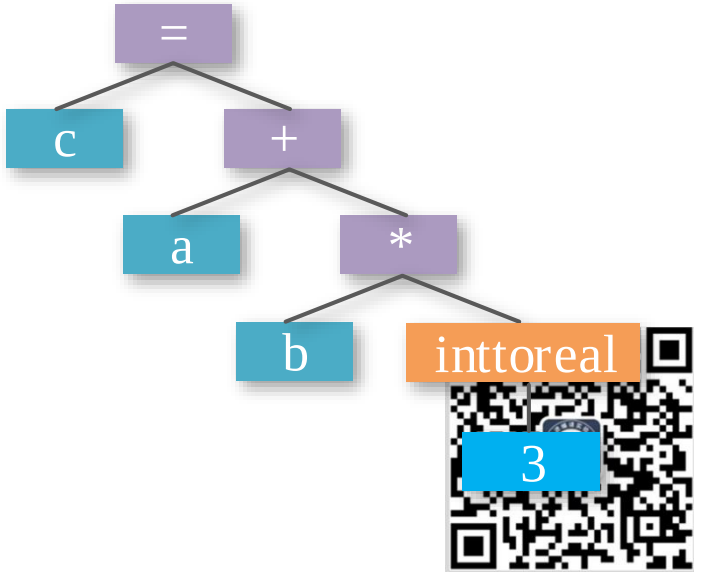
② 上下文相关性审查；

③ 类型匹配审查；

④ 类型转换。

比如语句c=a+b*3中，运算符*的两个运算对象分别是b和3，如果b是实型变量，3是整型常数，语义分析阶段执行类型审查之后，会自动地将整型量转换为实型量以完成同类型的数据运算，体现在语法分析所得到的语法树上，即增加一个运算符结点(inttoreal)。

```
int b,c ;
c = a+b*3 ;
```



编译器前端将高级程序设计语言编写的源代码翻译到统一中间表示，优化人员可以通过编译选项指定预处理、语言和模式等对程序的前端编译过程予以干预。

	选项	功能
预处理	-include <file>	将隐式的#include添加到预定义缓冲区中，在对源文件进行预处理之前读取该缓冲区
	-I <directory>	将指定的目录添加到include文件的搜索路径中
	-F <directory>	将指定的目录添加到框架include文件的搜索路径中
前端	-fsyntax-only	防止编译器生成代码,只是语法级别的说明和修改
	-dump-tokens	词法分析过程，拆分内部代码段为各种单词序列（token）
	-ast-dump	语法分析过程，构建抽象语法树AST,然后对其进行拆解和调试
语言和模式	-x <language>	将后续输入文件视为具有<language>类型的语言
	-std=<standard>	指定要编译的语言标准
	-stdlib=<library>	指定要使用的C ++标准库，例如libstdc ++和libc++。如果未指定，将使用平台默认值
	-fno-builtin	禁用内建函数的处理和优化
编译阶段	-E	运行预编译阶段，生成预编译文件
	-emit-llvm -S	运行编译阶段，生成.ll中间代码文件

分享内容

CONTENTS

先进编译实验室
Advanced Compiler

- **中间代码**

LLVM 中间表示的三种格式：在内存中的编译中间语言；硬盘上存储的二进制中间语言，以.bc结尾；以及可读的中间代码格式，以.ll结尾

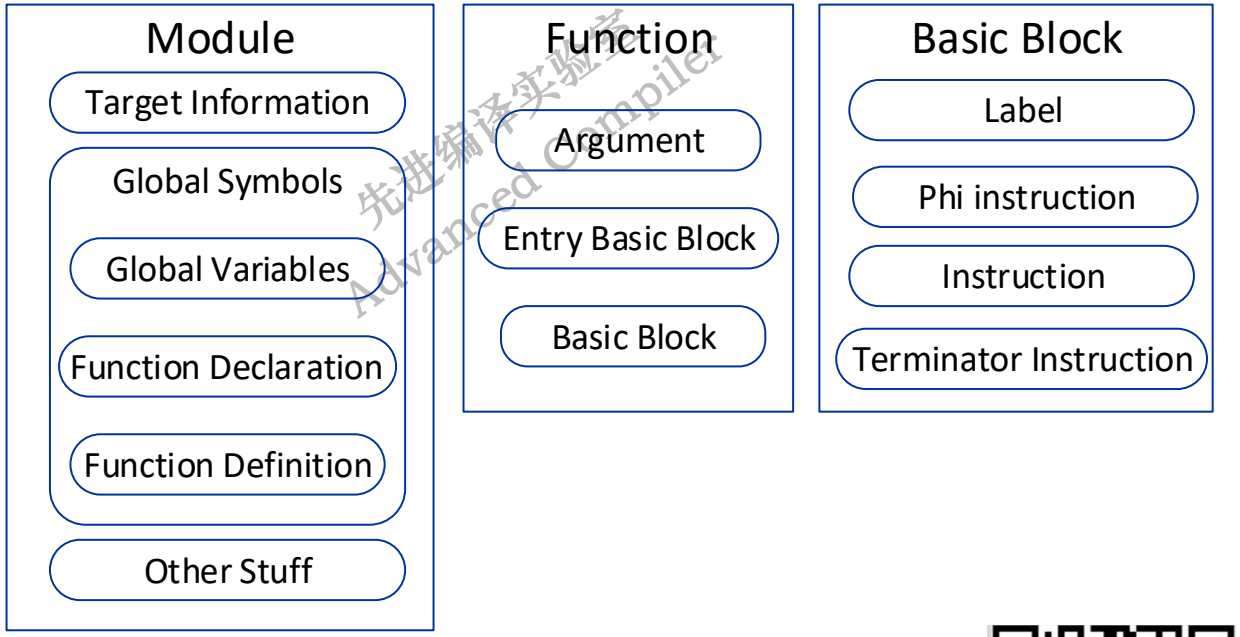
LLVM 中间代码结构包括四个部分：

① 模块（Module）是LLVM IR的顶层容器，对应于编译前端的每个翻译单元。每个模块由目标机器信息、全局符号（全局变量和函数）及元信息组成。

② 函数（Function）就是编程语言中的函数，包括函数签名和若干个基本块，函数内的第一个基本块叫做入口基本块。

③ 基本块（BasicBlock）是一组顺序执行的指令集合，只有一个入口和一个出口，非头尾指令执行时不会违背顺序跳转到其他指令上去。每个基本块最后一条指令一般是跳转指令（跳转到其它基本块上去），函数内最后一个基本块的最后一条指令是函数返回指令。

④ 指令（Instruction）是LLVM IR中的最小可执行单位。



- **中间代码**

生成LLVM中间代码的编译命令为：clang -emit-llvm -S file.c -o file.ll

LLVM中间代码为采用静态单赋值形式的中间表示，使用的指令集为LLVM虚拟指令集。 LLVM虚拟指令集由操作指令和内建指令两部分组成。内建指令是指以llvm.开头的指令，比如预取内建指令llvm.prefetch，在使用时被转换成一条或多条操作指令。常见的操作指令有四则运算指令、逻辑运算指令等，如LLVM中间代码的加法操作：

file.c:

```
int a , b, c ;
a = 2 ;
b = 4 ;
c = a + b *3 ;
return c ;
```

file.ll:

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 2, i32* %a, align 4
  store i32 4, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %mul = mul nsw i32 %1, 3
  %add = add nsw i32 %0, %mul
  store i32 %add, i32* %c, align 4
  %2 = load i32, i32* %c, align 4
  ret i32 %2
}
```

代码优化阶段的任务是对中间代码进行变换或改造，目的是使生成的目标代码更为高效（省时间和省空间）。

-O优化选项用以控制编译器在对程序编译时的优化级别，常用的有-O0、-O1、-O2、-O3、-Ofast。

选项	所包含的优化选项
-O1	在-O0的基础上添加添加-instcombine、-simplifycfg、-loops、-loop-unroll等
-O2	在-O1的基础上添加-inline、-fvectorize、-fslp-vectorize等
-O3	在-O2的基础上添加-aggressive-instcombine、-callsite-splitting、-domtree等
-Ofast	在-O3的基础上添加：-fno-signed-zeros、-freciprocal-math、-ffp-contract=fast、-menable-unsafe-fp-math、-menable-no-nans、-menable-no-infs、-mreassociate、-fno-trapping-math、-ffast-math、-ffinite-math-only等

死代码删除

死代码是指在程序操作过程中不可能被执行到的代码，或是代码可以被执行，但是对计算结果不起任何作用，比如代码段。

```
#include <stdio.h>
#include <stdlib.h>
int main ( )
{
  int a,b,c;
  a = rand();
  b = rand();
  c = a + b*3;//S1
  b = a;//S2
  return c;
}
```

clang dead.c -emit-llvm -S

关闭死代码删除优化：

```
%retval = alloca i32, align 4
%a = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
store i32 0, i32* %retval, align 4
%call = call i32 @rand() #2
store i32 %call, i32* %a, align 4
%call1 = call i32 @rand() #2
store i32 %call1, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%mul = mul nsw i32 %1, 3
%add = add nsw i32 %0, %mul
store i32 %add, i32* %c, align 4
%2 = load i32, i32* %a, align 4
store i32 %2, i32* %b, align 4
%3 = load i32, i32* %c, align 4
ret i32 %3
```

opt file.ll -dse -S

开启死代码删除优化：

```
%a = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
%call = call i32 @rand() #2
store i32 %call, i32* %a, align 4
%call1 = call i32 @rand() #2
store i32 %call1, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%mul = mul nsw i32 %1, 3
%add = add nsw i32 %0, %mul
store i32 %add, i32* %c, align 4
%2 = load i32, i32* %c, align 4
ret i32 %2
```


过程间优化

过程间优化是涉及程序中多个过程的程序变换与优化，过程间分析阶段为过程间优化提供足够的信息，用于支持过程间优化阶段的各类程序变换。

内联优化是过程间优化中最常用的一种方法，它是指如果在循环中调用了另一个过程，则过程间优化会将该过程内联到函数体内，并且会重新对过程排序以获得更好的内存布局。

```
#include <stdio.h>
#include <stdlib.h>
#define N 256
int add(int* a,int* b) {
  int c;
  c = *a + *b;
  return c;
}
int main() {
  int sum,i;
  int a[N],b[N];
  for(i=0;i<N;i++){
    a[i] = rand()%10;
    b[i] = rand()%10;
  }
  for(i=0;i<N;i++){
    sum += add(&a[i],&b[i]);
  }
  printf("%d",sum);
}
```

clang test.c -emit-llvm -S -O1 -o test.ll

opt test.ll -S -inline -o test-new.ll

内联优化前

```
for.body7:                                        ; preds = %for.body, %for.body7
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body7 ], [ 0, %for.body ]
  %sum.027 = phi i32 [ %add, %for.body7 ], [ undef, %for.body ]
  %arrayidx9 = getelementptr inbounds [256 x i32], [256 x i32]* %a, i64 0, i64 %indvars.iv
  %arrayidx11 = getelementptr inbounds [256 x i32], [256 x i32]* %b, i64 0, i64 %indvars.iv
  %call12 = call i32 @add(i32* nonnull %arrayidx9, i32* nonnull %arrayidx11)
  %add = add nsw i32 %call12, %sum.027
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, 256
  br i1 %exitcond.not, label %for.end15, label %for.body7, !llvm.loop !9
```

内联优化后

```
for.body7:                                        ; preds = %for.body7, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body7 ], [ 0, %for.body ]
  %sum.027 = phi i32 [ %add, %for.body7 ], [ undef, %for.body ]
  %arrayidx9 = getelementptr inbounds [256 x i32], [256 x i32]* %a, i64 0, i64 %indvars.iv
  %arrayidx11 = getelementptr inbounds [256 x i32], [256 x i32]* %b, i64 0, i64 %indvars.iv
  %2 = load i32, i32* %arrayidx9, align 4, !tbaa !2
  %3 = load i32, i32* %arrayidx11, align 4, !tbaa !2
  %add.i = add nsw i32 %3, %2
  %add = add nsw i32 %add.i, %sum.027
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %exitcond.not = icmp eq i64 %indvars.iv.next, 256
  br i1 %exitcond.not, label %for.end15, label %for.body7, !llvm.loop !9
```

```
clang test.c -O2 -Rpass=inline
test.c:17:12: remark: add inlined into main with (cost=-25, threshold=337) [-Rpass=inline]
  sum += add(&a[i],&b[i]);
```

自动向量优化

自动向量化是指编译器自动的将串行代码转化为向量代码的一种优化变换。向量计算是一种特殊的并行计算方式，相比于标量执行时每次仅操作一个数据，它可以在同一时间对多个数据执行相同的操作，从而获得数据级并行。LLVM目前支持两种自动向量化方法，分别是循环级向量化和基本块级向量化。

- 循环级向量化

通过扩大循环中的指令以获得多个连续迭代中操作的向量执行，在LLVM中通过选项-fvectorize开启循环向量化，并且当打开-O2选项及高于-O2优化级别的选项即自动开启循环向量化优化。

```
#include <stdio.h>
#define N 1280
int main(){
    int sum = 0;
    int a[N];
    int i,j;
    for(i=0;i<N;i++){
        a[i] = i;
    }
    for(j=0;j<N;j++){
        sum = sum + a[j];
     }
     printf("sum = %d",sum);
}
```

clang -O1 -fvectorize test-vec.c -emit-llvm -S

```
.......
vector.body28:                                    ; preds = %vector.body, %vector.body28
  %index30 = phi i64 [ %index.next31, %vector.body28 ], [ 0, %vector.body ], !dbg !21
  %vec.phi = phi <4 x i32> [ %6, %vector.body28 ], [ zeroinitializer, %vector.body ]
  %4 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %index30, !dbg !21
  %5 = bitcast i32* %4 to <4 x i32>*, !dbg !22
  %wide.load = load <4 x i32>, <4 x i32>* %5, align 16, !dbg !22, !tbaa !12
  %6 = add <4 x i32> %wide.load, %vec.phi, !dbg !23
  %index.next31 = add i64 %index30, 4, !dbg !21
.......
```

- 基本块级向量化

基本块级向量化算法的思想来源于指令级并行，通过将基本块内可以同时执行的多个标量操作打包成向量操作来实现并行，与循环级向量并行发掘方法不同，基本块级向量化发掘方法主要是在基本块内寻找同构语句，发掘基本块内指令的并行机会。在LLVM中加入选项-fslp-vectorize以开启基本块级向量化。

```
#include <stdio.h>
#define N 10240
int main(){
   int a[N],b[N],c[N];
   int i;
   for (i = 0; i < 10240;i++){
      b[i] = i;
      c[i] = i+1;
   }
   for (i = 0; i < 10240;i+=4){
      a[i] = b[i] + c[i];
      a[i+1] = b[i+1] + c[i+1];
      a[i+2] = b[i+2] + c[i+2];
      a[i+3] = b[i+3] + c[i+3];
   }
return a[100];
}
```

clang -O2 -fslp-vectorize SLP.c

```
……
%arrayidx7 = getelementptr inbounds [10240 x i32], [10240 x i32]* %b, i64 0, i64 %indvars.iv, !dbg !22
%arrayidx9 = getelementptr inbounds [10240 x i32], [10240 x i32]* %c, i64 0, i64 %indvars.iv, !dbg !23
%arrayidx12 = getelementptr inbounds [10240 x i32], [10240 x i32]* %a, i64 0, i64 %indvars.iv, !dbg !24
%5 = bitcast i32* %arrayidx7 to <4 x i32>*, !dbg !22
%6 = load <4 x i32>, <4 x i32>* %5, align 16, !dbg !22, !tbaa !12
%7 = bitcast i32* %arrayidx9 to <4 x i32>*, !dbg !23
%8 = load <4 x i32>, <4 x i32>* %7, align 16, !dbg !23, !tbaa !12
%9 = add nsw <4 x i32> %8, %6, !dbg !25
%10 = bitcast i32* %arrayidx12 to <4 x i32>*, !dbg !26
store <4 x i32> %9, <4 x i32>* %10, align 16, !dbg !26, !tbaa !12
```

优化后的中间代码代码示意如下：

```
for (int i = 0;  i < LEN;  i+=4){
    a[i:3] = b[i:3] + c[i:3];
}
```

基本块级向量化算法主要用于发掘基本块内的并行性，它要遍历所有的向量方案得到最优解，所以复杂度高于循环级向量化，具有更强的向量发掘能力，经转化后的向量代码程序性能会有很大的提升。

循环优化

循环优化是编译器中重要的优化手段之一，本节选取LLVM编译器支持的几种典型的循环优化，包括循环展开、循环分布和循环剥离进行介绍。

- 循环展开

循环展开是指将循环体代码复制多次的实现，通过增大指令调度的空间来减少循环分支指令的开销、增加数据引用的局部性，从而提高循环执行性能的一种循环变换技术。在LLVM中通过选项-funroll-loops打开循环展开优化。

```
#include <stdio.h>
#define N 1280
int main(){
  int sum = 0;
  int a[N];
  int i,j;
  for(i=0;i<N;i++){
    a[i] = i;
  }
  for(j=0;j<N;j++){
    sum = sum + a[j];
  }
  printf("sum = %d",sum);
}
```

clang unroll.c -O1 -funroll-loops

```
for.body3:                                        ; preds = %for.body, %for.body3
  %indvars.iv = phi i64 [ %indvars.iv.next.7, %for.body3 ], [ 0, %for.body ]
  %sum.018 = phi i32 [ %add.7, %for.body3 ], [ 0, %for.body ]
  %arrayidx5 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %indvars.iv
  %9 = load i32, i32* %arrayidx5, align 16, !tbaa !2
  %add = add nsw i32 %9, %sum.018
  %indvars.iv.next = or i64 %indvars.iv, 1
  %arrayidx5.1 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %indvars.iv.next
  %10 = load i32, i32* %arrayidx5.1, align 4, !tbaa !2
  %add.1 = add nsw i32 %10, %add
  %indvars.iv.next.1 = or i64 %indvars.iv, 2
  %arrayidx5.2 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %indvars.iv.next.1
  %11 = load i32, i32* %arrayidx5.2, align 8, !tbaa !2
  %add.2 = add nsw i32 %11, %add.1
  %indvars.iv.next.2 = or i64 %indvars.iv, 3
  %arrayidx5.3 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %indvars.iv.next.2
  %12 = load i32, i32* %arrayidx5.3, align 4, !tbaa !2
  %add.3 = add nsw i32 %12, %add.2
  %indvars.iv.next.3 = or i64 %indvars.iv, 4
……
```

- 循环分布

循环分布是指将循环内的一条或多条语句移到单独一个循环中，以满足某些特定的需求。在LLVM中通过选项-mllvm -enable-loop-distribute打开循环分布优化。

```
#include <stdio.h>
#include <stdlib.h>
#define N 1280
int main(){
    int A[N],B[N],C[N];
    int i;
    for(i=0;i<N;i++){
        B[i] = rand();
        C[i] = rand();
    }
    for(i=1;i<N;i++){
        A[i] = i;
        B[i] = 2 + B[i];
        C[i] = 3 + C[i - 1];
    }
    for(i=0;i<N;i++){
        printf("%d",B[i]);
        printf("%d",A[i]);
        printf("%d",C[i]);
    }
}
```

clang -O1 LoopDistribute.c
-mllvm -enable-loop-distribute

```
for.body6.ldist1:                                 ; preds = %for.body, %for.body6.ldist1
  %indvars.iv56.ldist1 = phi i64 [ %indvars.iv.next57.ldist1, %for.body6.ldist1 ], [ 1, %for.body ]
  %arrayidx8.ldist1 = getelementptr inbounds [1280 x i32], [1280 x i32]* %A, i64 0, i64 %indvars.iv56.ldist1
  %3 = trunc i64 %indvars.iv56.ldist1 to i32
  store i32 %3, i32* %arrayidx8.ldist1, align 4, !tbaa !2
  %arrayidx10.ldist1 = getelementptr inbounds [1280 x i32], [1280 x i32]* %B, i64 0, i64 %indvars.iv56.ldist1
  %4 = load i32, i32* %arrayidx10.ldist1, align 4, !tbaa !2
  %add.ldist1 = add nsw i32 %4, 2
  store i32 %add.ldist1, i32* %arrayidx10.ldist1, align 4, !tbaa !2
  %indvars.iv.next57.ldist1 = add nuw nsw i64 %indvars.iv56.ldist1, 1
  %exitcond59.ldist1 = icmp eq i64 %indvars.iv.next57.ldist1, 1280
  br i1 %exitcond59.ldist1, label %for.body6.preheader, label %for.body6.ldist1
for.body6.preheader:                              ; preds = %for.body6.ldist1
  %load_initial = load i32, i32* %C63, align 16
  br label %for.body6
for.body6:                                        ; preds = %for.body6.preheader, %for.body6
  %store_forwarded = phi i32 [ %load_initial, %for.body6.preheader ], [ %add15, %for.body6 ]
  %indvars.iv56 = phi i64 [ 1, %for.body6.preheader ], [ %indvars.iv.next57, %for.body6 ]
  %add15 = add nsw i32 %store_forwarded, 3
  %arrayidx17 = getelementptr inbounds [1280 x i32], [1280 x i32]* %C, i64 0, i64 %indvars.iv56
  store i32 %add15, i32* %arrayidx17, align 4, !tbaa !2
  %indvars.iv.next57 = add nuw nsw i64 %indvars.iv56, 1
  %exitcond59 = icmp eq i64 %indvars.iv.next57, 1280
  br i1 %exitcond59, label %for.body23, label %for.body6
```

- 循环剥离

循环剥离常用于将循环中数据首地址不对齐的引用，以及循环末尾不够装载到一个向量寄存器的数据剥离出来，使剩余数据满足向量化对齐性要求。在LLVM编译器优化分析中，循环展开优化通常与循环剥离配合使用，优化人员可通过选项-mllvm unroll-peel-count写定剥离数值，

```
#include <stdio.h>
#define N 1280
int main(){
  int a[N],b[N],c[N];
  int i;
  for(i=0;i<N;i++){
    a[i] = i;
    b[i] = i+3;
  }
  for(i=0;i<N-2;i++){
    c[i+2] = a[i+2] + b[i+2];
  }
    return c[8];
}
```

clang test-peel.cpp -O2

-mllvm -unroll-peel-count=2

剥离前，不符合向量对齐要求

{ c[2] c[3] c[4] c[5] } { c[6] c[7] c[8] c[9] }

剥离后，符合向量对齐要求

{ c[2] c[3] } { c[4] c[5] c[6] c[7] }

先进编译实验室
Advanced Compiler

```
for.body5.peel.next35:                            ; preds = %for.body
  %arrayidx8.peel.phi.trans.insert = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 2
  %.pre = load i32, i32* %arrayidx8.peel.phi.trans.insert, align 8, !tbaa !2
  %arrayidx11.peel.phi.trans.insert = getelementptr inbounds [1280 x i32], [1280 x i32]* %b, i64 0, i64 2
  %.pre55 = load i32, i32* %arrayidx11.peel.phi.trans.insert, align 8, !tbaa !2
  %add12.peel = add nsw i32 %.pre55, %.pre                                              i=0
  %arrayidx15.peel = getelementptr inbounds [1280 x i32], [1280 x i32]* %c, i64 0, i64 2
  store i32 %add12.peel, i32* %arrayidx15.peel, align 8, !tbaa !2
  %arrayidx8.peel37 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 3
  %17 = load i32, i32* %arrayidx8.peel37, align 4, !tbaa !2
  %arrayidx11.peel38 = getelementptr inbounds [1280 x i32], [1280 x i32]* %b, i64 0, i64 3
  %18 = load i32, i32* %arrayidx11.peel38, align 4, !tbaa !2
  %add12.peel39 = add nsw i32 %18, %17                                                  i=1
  %arrayidx15.peel40 = getelementptr inbounds [1280 x i32], [1280 x i32]* %c, i64 0, i64 3
  store i32 %add12.peel39, i32* %arrayidx15.peel40, align 4, !tbaa !2
  br label %vector.body66

vector.body66:                                    ; preds = %vector.body66, %for.body5.peel.next35
  %index70 = phi i64 [ 0, %for.body5.peel.next35 ], [ %index.next71, %vector.body66 ]
  %offset.idx74 = or i64 %index70, 2
  %19 = add nuw nsw i64 %offset.idx74, 2
  %20 = getelementptr inbounds [1280 x i32], [1280 x i32]* %a, i64 0, i64 %19
  %21 = bitcast i32* %20 to <4 x i32>*
  %wide.load = load <4 x i32>, <4 x i32>* %21, align 16, !tbaa !2
……
```

浮点优化

若优化人员想提高浮点数据的运算性能，可以利用LLVM编译器的-ffast-math选项开启较为激进的浮点优化，该选项中包含了多种浮点优化方法，比如浮点数据归约向量化、除法运算优化和忽略浮点数0的正负号等。

```
#include <stdio.h>
#define N 128
int main( ){
  float sum = 0;
  float a[N];
  int i,j;
  for(i=0;i<N;i++){
     a[i] = i;
  }
  for(j=0;j<N;j++) {
     sum = sum + a[j];
  }
  printf("sum = %f",sum);
}
```

```
[llvm@2021]$ clang ffast.cpp -fvectorize -O1 -Rpass-missed=loop-vectorize -Rpass=loop-vectorize
ffast.cpp:7:15: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
        for(i=0;i<N;i++){
ffast.cpp:10:18: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
        for(j=0;j<N;j++) {
```

```
[llvm@2021]$ clang ffast.cpp -fvectorize -O1 -ffast-math -Rpass=loop-vectorize
ffast.cpp:7:15: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
        for(i=0;i<N;i++){
ffast.cpp:10:18: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
        for(j=0;j<N;j++) {
```

数据预取优化

LLVM编译器支持自动数据预取和手动添加预取内建指令两种预取方式，其中自动数据预取是编译器对程序分析后在中间代码中自动插入预取指令，当前LLVM编译器仅支持AArch64平台和PowerPC平台的自动数据预取优化。手动添加的预取函数为：

```
__builtin_prefetch(addr， rw， locality)
```

预取的内存地址　预取读/写　预取数据的局部性

```
#include <stdio.h>
int main( ){
   int arr[10];
   int i;
   for(i=0;i<10;i++){
      __builtin_prefetch(arr+i,1,3);
   }
   for(i=0;i<10000;i++){
      arr[i%10] = i;
   }
   for(i=0;i<10;i++){
      printf("%d\n",arr[i]);
   }
}
```

```
for.body:                                         ; preds = %for.cond
  %arraydecay = getelementptr inbounds [10 x i32], [10 x i32]* %arr, i64 0, i64 0
  %1 = load i32, i32* %i1, align 4
  %idx.ext = sext i32 %1 to i64
  %add.ptr = getelementptr inbounds i32, i32* %arraydecay, i64 %idx.ext
  %2 = bitcast i32* %add.ptr to i8*
  call void @llvm.prefetch.p0i8(i8* %2, i32 1, i32 3, i32 1)
  br label %for.inc
for.inc:                                          ; preds = %for.body
  %3 = load i32, i32* %i1, align 4
  %inc = add nsw i32 %3, 1
  store i32 %inc, i32* %i1, align 4
  br label %for.cond, !llvm.loop !2
for.end:                                          ; preds = %for.cond
  store i32 0, i32* %i2, align 4
  br label %for.cond3
```

	选项	功能
内联优化	-inline	打开内联函数功能
	-finline-functions	对合适的函数进行内联
	-inline-aggressive	在链接时优化期间开启激进的内联优化
循环优化	-funroll-loops	打开循环展开
	-fno-unroll-loops	关闭循环展开
	-mllvm -unroll-count	确定展开次数
	-mllvm -unroll-peel-count	设置循环剥离计数
	-mllvm -enable-loop-distribute	打开循环分布优化
向量化	-fvectorize	开启循环向量化优化
	-fslp-vectorize	开启基本块级向量化
	-interleave-loops	在循环向量化过程中启用循环跨幅访存
浮点优化	-ffast-math	开启一系列的浮点优化功能
	-freciprocal-math	允许将除法运算转换为对倒数的乘法运算（包含于-ffast-math）
	-fno-signed-zeros	忽略浮点零的符号（包含于-ffast-math）
数据预取	-mllvm -loop-data-prefetch	开启预取访问(针对AArch64和PowerPC)
优化信息选项	-Rpass=vectorize	显示循环向量化和SLP向量化有关的信息
	-Rpass=loop-unroll	显示循环展开和循环剥离的优化信息
	-Rpass-missed=loop-unroll	显示循环展开失败的信息
	-Rpass=loop-distribute	显示循环分布的信息
	-Rpass-analysis=loop-distribute	显示循环分布的分析信息

分享内容

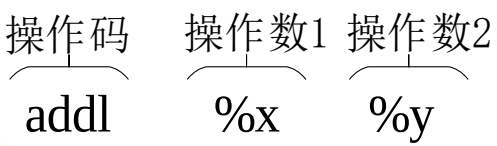
CONTENTS

先进编译实验室
Advanced Compiler

目标代码生成

该过程是把中间代码变换成特定机器上的目标代码，形式上包括：绝对指令代码、可重定位的指令代码、汇编指令代码。这是编译的最后阶段，它的工作与硬件系统结构和指令含义有关，涉及到硬件系统功能部件的运用、机器指令的选择、各种数据类型变量的存储空间分配以及寄存器分配等。在LLVM中使用clang -S file.c命令进行编译，生成特定平台的汇编代码.s文件。

以汇编文件中的加法指令为例，该汇编指令由一个操作码和两个操作数组成，操作码addl为加法操作，操作数%x和操作数%y执行加法运算，并将结果放置在%x中。



file.c

```
a = 2;
b = 4;
c = a + b * 3;
```

file.ll

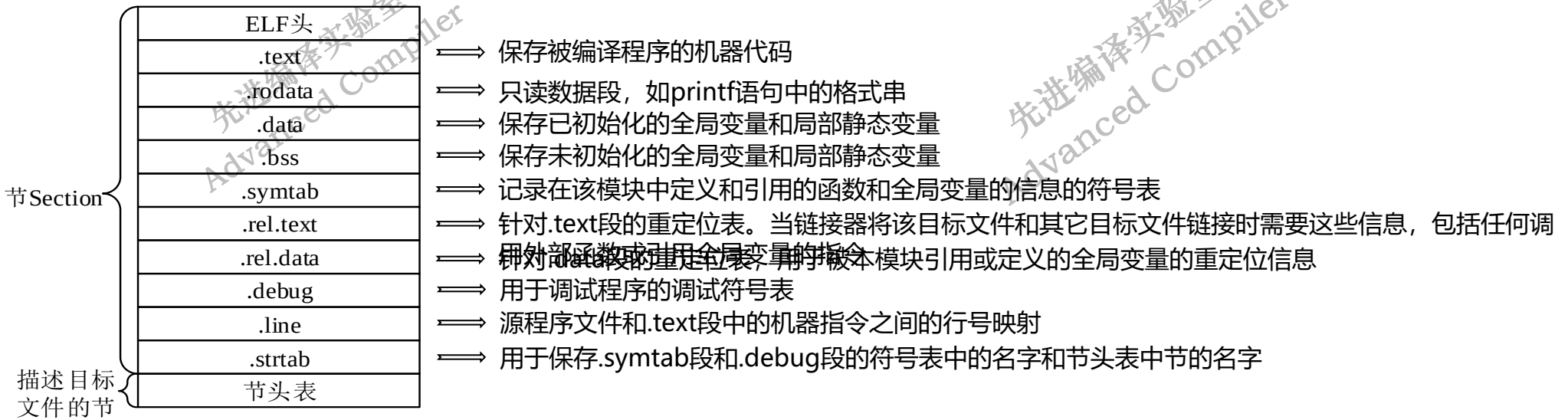
```
store i32 2, i32* %a, align 4
store i32 4, i32* %b, align 4
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%mul = mul nsw i32 %1, 3
%add = add nsw i32 %0, %mul
store i32 %add, i32* %c, align 4
```

file.s

```
movl   $2, -8(%rbp)
movl   $4, -12(%rbp)
movl   -8(%rbp), %eax
imull  $3, -12(%rbp), %ecx
addl   %ecx, %eax
movl   %eax, -16(%rbp)
```

目标文件格式

目标文件是源代码编译后但未链接的中间文件，它跟可执行文件的内容与结构很相似，从广义上看目标文件和可执行文件的格式几乎是一样的，在Linux操作系统下都是按照可执行可链接文件格式（Executable Linkable Format,ELF）进行存储。 ELF目标文件格式的最前部是ELF头，包含描述整个文件的基本属性，包括ELF文件版本、目标机器型号、程序入口地址等。节头表描述目标文件中各个节的信息，在ELF头和节头表之间是节本身，其中包含十个段。



这十段内容组成了可重定位目标文件，有了这些目标文件之后，通过静态链接就可以将它们组合起来，形成一个可以运行的程序。

在代码生成阶段可以通过数据选项选择对数据的处理方式，通过目标平台选项生成指定平台的代码，通过后端选项打开不同后端支持的优化功能。

	选项	功能
编译阶段	-S	运行编译阶段，生成.s汇编文件
后端选项	-mavx2	在选项-mavx的基础上增加支持AVX2内置函数和指令集
	-msse	支持MMX和SSE内置函数和代码生成
	-msse2	在选项-msse的基础上增加SSE2内置函数和代码生成
	-mfentry	在函数入口插入对fentry的调用
	-mllvm -disable-x86-lea-opt	关闭LEA优化
数据选项	-malign-double	在structs中将双精度对齐为双字，仅适用于x86
	-mdouble=<value>	指定double类型数据的位数
目标平台选项	-march=<cpu>	指定Clang为特定处理器生成代码
	--cuda-host-only	只编译CUDA的主机端代码

分享内容

CONTENTS

先进编译实验室
Advanced Compiler

汇编器是将汇编代码转变为机器可以执行的指令，每一个汇编语句都对应一条机器指令，由.s汇编文件经汇编器生成.o目标代码文件。

```
clang -c file.s -o file.o
llvm-mc -filetype=obj file.s -o file.o
```

若有变量定义在其它目标代码文件中，则只有运行链接的时候才能确定绝对地址，所以现代的编译器可以将一个源代码文件编译成一个可重定位的目标文件，最终由链接器将这些目标文件链接起来形成可执行文件。

链接的功能是将一个或多个目标文件以及库文件合并为一个可执行文件。链接可以在源代码翻译成机器代码即编译的时候完成，也可以在程序装入内存时完成，甚至可以在程序运行时完成，根据不同的完成时期可将链接分为静态链接和动态链接。

```
clang a.o b.o -o ab.out
```

静态链接

```
//a.c程序
extern int shared;
int main(void){
  int a=100;
  add(&a,&shared);
  printf("%d",a);
}

//b.c程序
int shared=1;
void add(int* a,int* b){
  *a = *a + *b;
}
```

```
clang a.c –c –o a.o
clang b.c –c –o b.o
clang a.o b.o –o ab.out
```

静态链接指链接器将外部函数所在的静态链接库直接拷贝到目标可执行程序中，这样在执行该程序时这些代码会被装入到该进程的虚拟地址空间中。

动态链接

```
//hello.h:
void hello(char *s);
#endif
//hello.c:
void hello(char *s)
{
    printf("Hello %s\n",s);
}
//main.c:
#include "hello.h"
int main(int argc,char **argv)
{
    hello("ZZ");
    return 0;
}
```

```
clang -c -fPIC hello.c
clang -shared –fPIC hello.o -o libhello.so
clang main.c -L. -lhello -o a.out
```

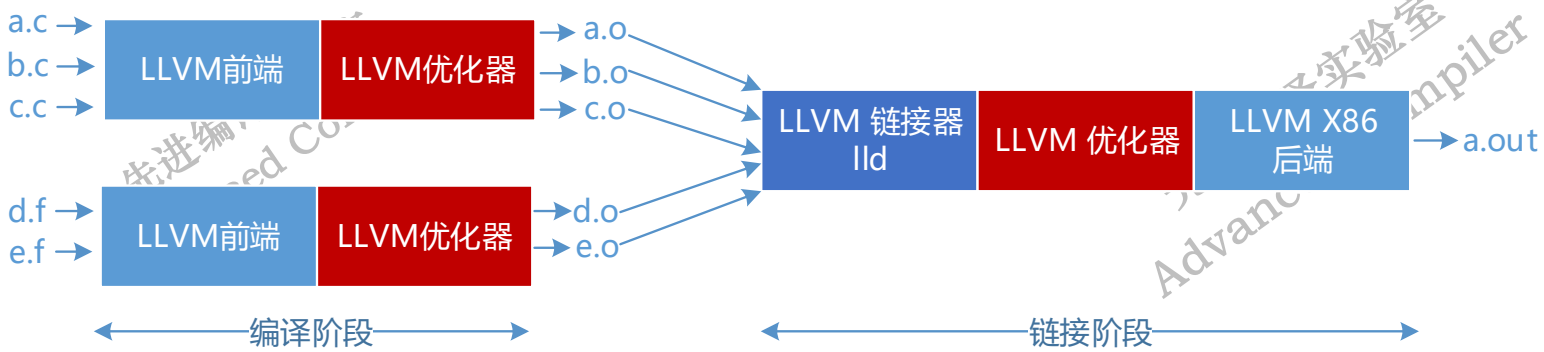
动态链接是把程序拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序。

动态链接与静态链接的对比

类型 / 特点	静态链接	动态链接
链接时机	形成可执行程序前	程序执行时
方式	地址与空间分配和符号解析与重定位	装载时重定位和地址无关代码技术
库扩展名	.a	.so
优点	程序的启动、运行速度快，方便移植	节省内存和磁盘空间
缺点	浪费内存和磁盘空间、模块更新困难	增加程序执行时链接开销，可移植性差

链接时优化

链接时优化是链接期间的程序优化，多个中间文件通过链接器合并在一起组合为一个程序，缩减代码体积，并通过对整个程序的分析以实现更好的运行时性能。优化人员通过选项-flto指示LLVM编译器生成含有LLVM比特码的.o文件，将代码生成延迟到链接阶段，并在链接阶段对代码实现进一步地优化。



当链接器检测到.o文件为LLVM比特码时，会将所有的比特码文件读入内存并链接起来，然后再进行跨文件地内联、常量传播和更激进地死代码消除等优化。

选项-flto后可跟参数：

-flto=full指链接时优化将分散的目标文件的LLVM IR组合到一个大的LLVM目标文件中，然后对其整体分析、优化并生成机器码。

-flto=thin是把目标文件分开，根据需要才从其它目标文件中导入功能，使用选项-flto=thin链接的速度要快于使用选项-flto=full。

链接时优化

比如该示例。

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
--- a.c ---
#include "a.h"
static signed int i = 0;
void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
return data;
}
--- main.c ---
#include <stdio.h>
#include "a.h"
void foo4(void) {
  printf("Hi\n");
}
int main() {
   return foo1();
}
```

```
clang -flto -c a.c -o a.o
clang -c main.c -o main.o
clang -flto a.o main.o -o main
```

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);
--- a.c ---
#include "a.h  "
static signed int i = 0
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
return data;
}
--- main.c ---
#include <stdio.h>
#include "a.h"
int main() {
   return foo1();
}
```

数学库

程序在编译过程中经常链接数学库，数学库是开展科学计算、工程计算等必备的核心基础软件，数学函数的性能、可靠性和精度对上层应用程序尤其是科学计算程序的解算至关重要。使用数学库中提供的各类数学函数，能够缩短应用程序的开发周期，并获取库函数所带来的性能收益。


```
#include <stdio.h>
#include <math.h>
#define PI 3.1415927
int main() {
  double a = (30*PI/180);
  a = sin(a);
  printf("%lf\n",a);
}
```

```
clang math.c -lm
```


BLAS（Basic Linear Algebra Subprograms）基本线性代数库是一组高质量的基本向量、矩阵运算子程序。由于BLAS涉及最基本的向量、矩阵运算应用程序的开发者只需要运用适当的技术将计算过程抽象为矩阵、向量的基本运算，就可以调用相应的BLAS库函数而不必考虑与计算机体系结构相关的性能优化问题。

数学库

BLAS从结构上分为三部分包括：

Level 1 BLAS：向量和向量，向量和标量之间的运算；

Level 2 BLAS：向量和矩阵间的运算；

Level 3 BLAS：矩阵和矩阵之间的运算。

BLAS库函数的命名由三部分组成：数据类型、矩阵类型、操作类型。

BLAS库支持的数据类型

精度	字母代号	描述
Single real	s	单精度实数
Double real	d	双精度实数
Single complex	c	单精度复数
Double complex	z	双精度复数

BLAS库支持的矩阵类型

矩阵类型	字母代号	描述
General matrix	ge	普通矩阵
General band matrix	gb	带状矩阵
Symmetric matrix	sy	对称矩阵
Hermitian matrix	he	自共轭矩阵
Hermitian band matrix	hb	自共轭带状矩阵
Triangular packed	tr	三角矩阵
Triangular band	tb	三角带状矩阵

BLAS库支持的常用函数操作

函数	描述
dot	标量运算
axpy	向量-向量操作
mv	矩阵-向量乘积运算
sv	矩阵-向量操作解线性方程组
mm	矩阵-矩阵乘积运算
sm	使用矩阵-矩阵操作解线性方程组

比如函数dgemm代表双精度实数的普通矩阵乘积运算

数学库

英特尔数学内核库（Intel Math Kernel Library,MKL）为英特尔包含有BLAS计算功能的数学核心函数库。当在C语言中使用该数学库编程时需引用头文件mkl_cblas.h。

使用Intel 的ICC编译器对代码进行编译，需要添加选项-mkl=<arg>选项，其中不同的参数表示的含义不同，包括：

- -mkl或-mkl=parallel：并行链接Intel(R) MKL库，这也是使用-mkl选项时的默认值；
- -mkl=sequential：采用串行Intel(R) MKL库链接；
- -mkl=cluster：使用Intel(R) MKL Cluster库和Intel(R) MKL序列库链接。

先进编译实验室
Advanced Compiler

```
#include<mkl_cblas.h>
#include<iostream>
using namespace std;
void init_arr(int N,double* a);
int main(int argc,char* argv[]) {
  int i,j;
  int N=1000;
  double alpha=1.0;
  double beta=0.;
  int incx = 1;
  int incy = N;
  double* a;
  double* b;
  double* c;
  a=(double*) malloc( sizeof(double)*N*N );
  b=(double*) malloc( sizeof(double)*N*N );
  c=(double*) malloc( sizeof(double)*N*N );
  init_arr(N,a);
  init_arr(N,b);
  cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,N,N,N,alpha,b,N,a,N,beta,c,N);
  free(a);
  free(b);
  free(c);
  return 0;
}
void init_arr(int N, double* a){
  int i,j;
  for (i=0; i< N;i++) {
    for (j=0; j<N;j++) {
      a[i*N+j] = (i+j+1)%10;
    }
  }
}
```

数学库优化

在性能要求苛刻的情况下，函数库提供的性能有时并不能满足优化人员的需要，因此需要优化人员提升库的性能。优化人员可以根据一些平台无关的算法，在考虑平台的相关特性后自行对库函数进行优化，以达到对于性能的要求。

通常代码中的abs数学函数都是标量运算，本示例在没有数据依赖的情况下，将标量abs运算改为向量abs运算，这样可以同时处理4个abs值求解。

先进编译实验室
Advanced Compiler

未向量化示例abs.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 40960000
int main(){
  int ref[N],cur[N];
  int sum,local;
  sum = 0.0;
  int i;
for(i=0;i<N;i++){
    ref[i] = rand()%100;
    cur[i] = rand()%100;
  }
for(i=0;i<N;i++){
    local = ref[i] - cur[i];
    sum += abs(local);
  }
 printf("sum = %d\n",sum);
}
```

手工向量化示例abs-vec.c：

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <x86intrin.h>
#define N 40960000
int main(){
  __m128i v0,v1,v2,v3,v4;
  int ref[N],cur[N];
  int A[4];
  int sum = 0;
  int i;
  for(i=0;i<N;i++){
    ref[i] = rand()%100;
    cur[i] = rand()%100;
  }
for(i=0;i<N/4;i++){
    v0 = _mm_load_epi32(ref + 4*i);
    v1 = _mm_load_epi32(cur + 4*i);
    v2 = _mm_set_epi32(0,0,0,0);
    v3 = _mm_sub_epi32(v0,v1);
    v4 = _mm_abs_epi32(v3);
    _mm_store_epi32(A,v4);
    sum += A[0] + A[1] + A[2] + A[3];
  }
  printf("sum = %d \n",sum);
}
```

	选项	功能
汇编	-c	运行编译、汇编阶段，但不运行链接，生成目标.o文件
链接	-o	运行编译、汇编、链接阶段，生成可执行文件
	-Bstatic	静态链接用户生成的库
	-l<库文件>	指明需链接的库名，如库名为libxyz.a，则可用-lxyz指定
	-L<库目录>	指定需要链接的库的目录地址
	-shared-libsan	动态链接sanitizer程序运行时
	-flto=<value>	将链接时优化的模式设置为full或thin
	-lm	链接基础数学库

分享完毕，感谢聆听！

先进编译实验室
Advanced Compiler

参考文献：

[1] Kai Nacke.Learn LLVM 12, A beginner’s guide to learning LLVM compiler tools and core libraries with C++[M]. Packt Publishing Ltd.,2021

[2] PANDEY M,SARDA S.LLVM cookbook [M].Packt Publishing Ltd.,2015.

[3] 李彭勇. 链接时死代码删除与基于模式匹配的机器码翻译[D].中国科学技术大学,2015.

[4] 俞甲子，石凡，潘爱民.程序员的自我修养[M].电子工业出版社.2009.

先进编译实验室
Advanced Compiler