

# clang编译工作流程

曾雷杰

2025年2月17日

# 简单例子（一个Project内部含三个文件）

add.h

```
1  #ifndef __ADD_H__
2  #define __ADD_H__
3
4  int add(int a, int b);
5
6  #define ADD(a, b) ((a) + (b))
7
8  #endif
```

add.c

```
1  #include "add.h"
2
3  int add(int a, int b)
4  {
5      return a + b;
6  }
```

main.c

```
1  #include "add.h"
2
3  int g1 = 10;
4  int g2;
5
6  int main()
7  {
8      int a = 10;
9      int c;
10
11     c = add(a, g1);
12
13     c = ADD(c, g2);
14
15     return c;
16 }
```

# clang编译-常见方式



# clang工作流程-动态链接编译

编译程序:

```
$clang -o main main.c add.c  
$./main  
$echo $?
```

查看文件属性:

```
$file main  
main: ELF 64-bit LSB executable, x86-64, version 1  
(SYSV), dynamically linked, interpreter /lib64/ld-  
linux-x86-64.so.2, for GNU/Linux 3.2.0,  
BuildID[sha1]=d94481eab5583bf110ae0fd5291f35  
8eb2eac0d6, not stripped
```

可执行程序依赖:

```
$ldd main  
linux-vdso.so.1 => (0x00007fff88d4b000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6  
(0x00007f7193a00000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f7193c45000)
```

编译详细内部:

```
$clang -o main main.c add.c --verbose
```

简化结果:

编译器  
(Compiler)

```
clang -cc1 -emit-obj -o /tmp/ccjLnMMc.o main.c
```

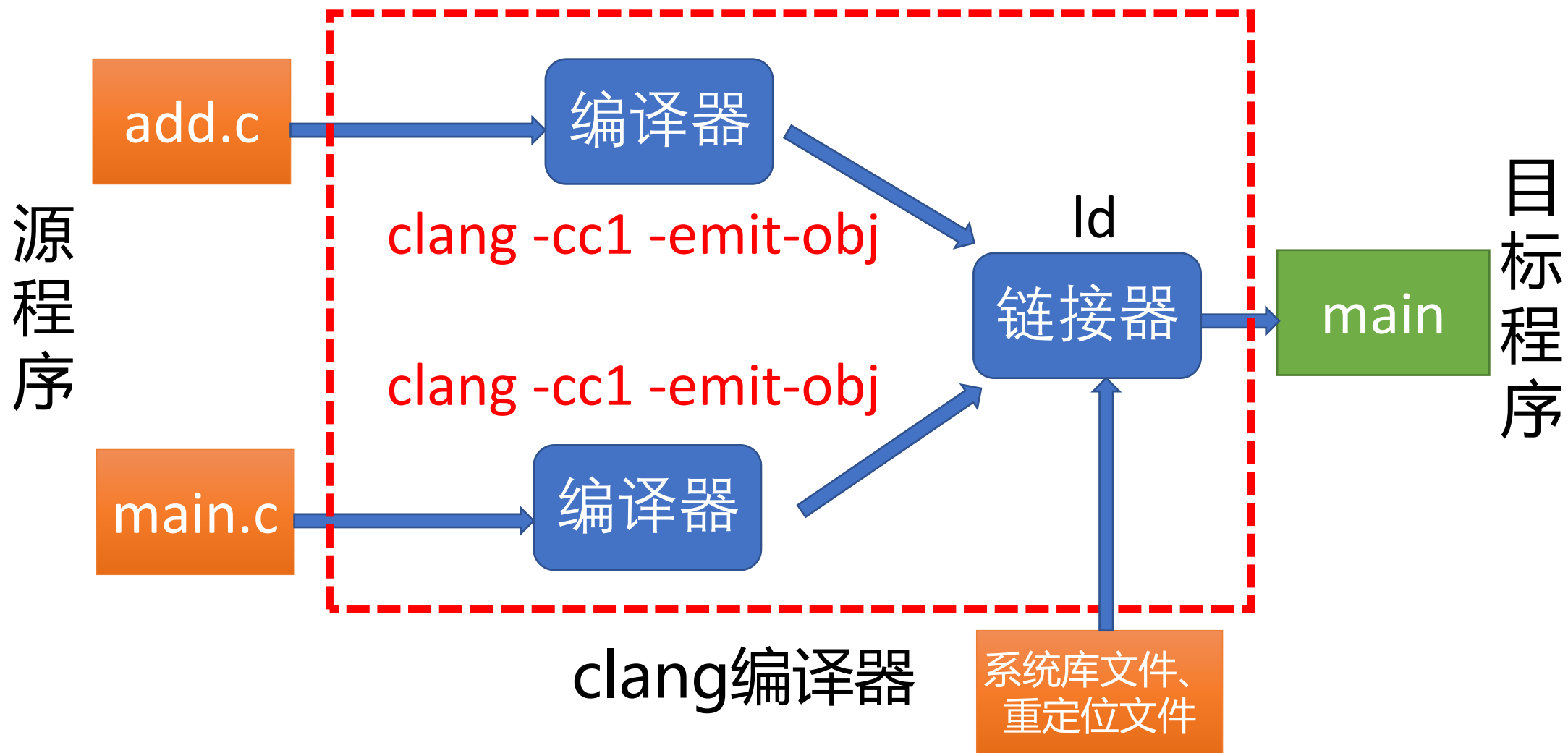
```
clang -cc1 -emit-obj -o /tmp/ccjLnMMc.o add.c
```

```
/usr/bin/ld -dynamic-linker -o main Scrt1.o crti.o  
crtbeginS.o  
/tmp/cc3FZAn.o /tmp/ccH4iZ0z.o -lgcc -lgcc_s c  
crtendS.o crtn.o
```

链接器  
(Linker)

# clang编译-内部详细过程

串行编译



# clang 工作过程概述

源程序

clang -E  
预处理

源代码  
(不含  
预处理  
宏)

clang -fsyntax-only -Xclang -dump-tokens

Token流

clang -fsyntax-only -Xclang -ast-dump

抽象语法树AST

clang -c -S -emit-llvm

LLVM IR(文本)

clang -c -emit-llvm

LLVM IR(二进制)

llvm-link 链接

llc clang -c -S

clang -c -S

汇编语言

clang -c

目标文件

clang -c

clang

Id 链接

lli  
JIT运行

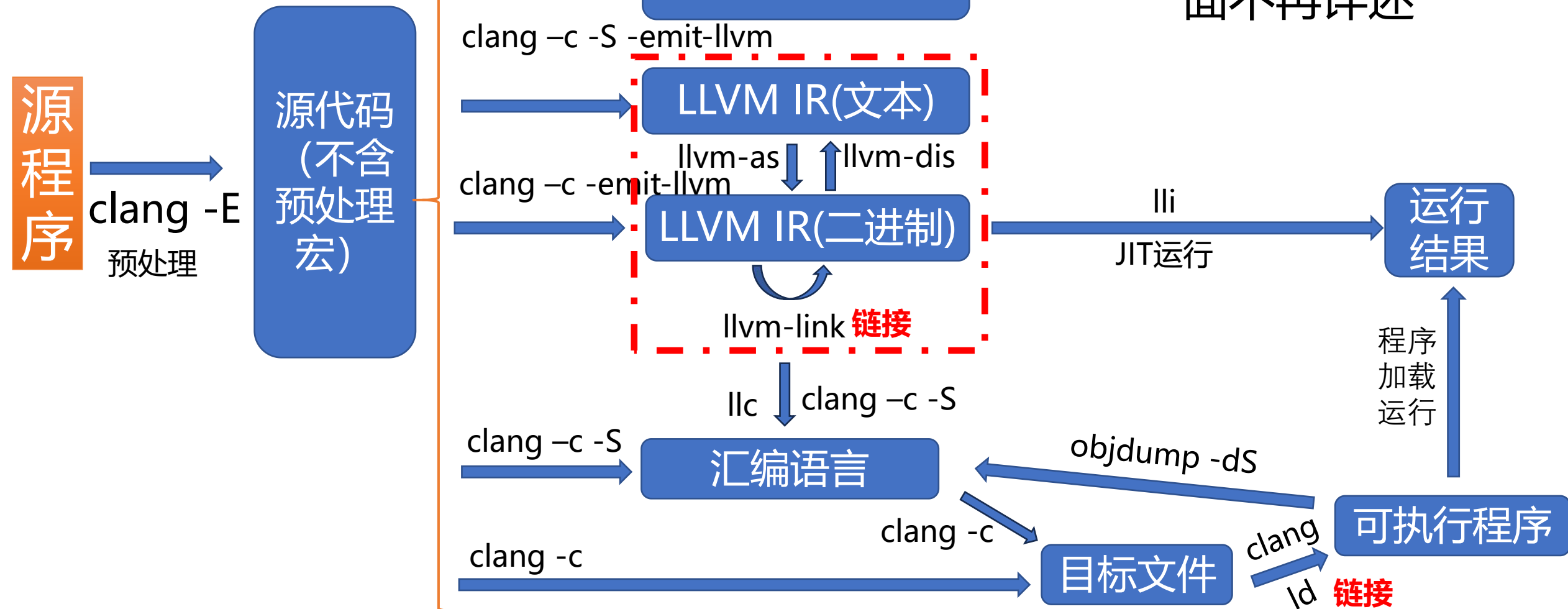
运行  
结果

程序  
加载  
运行

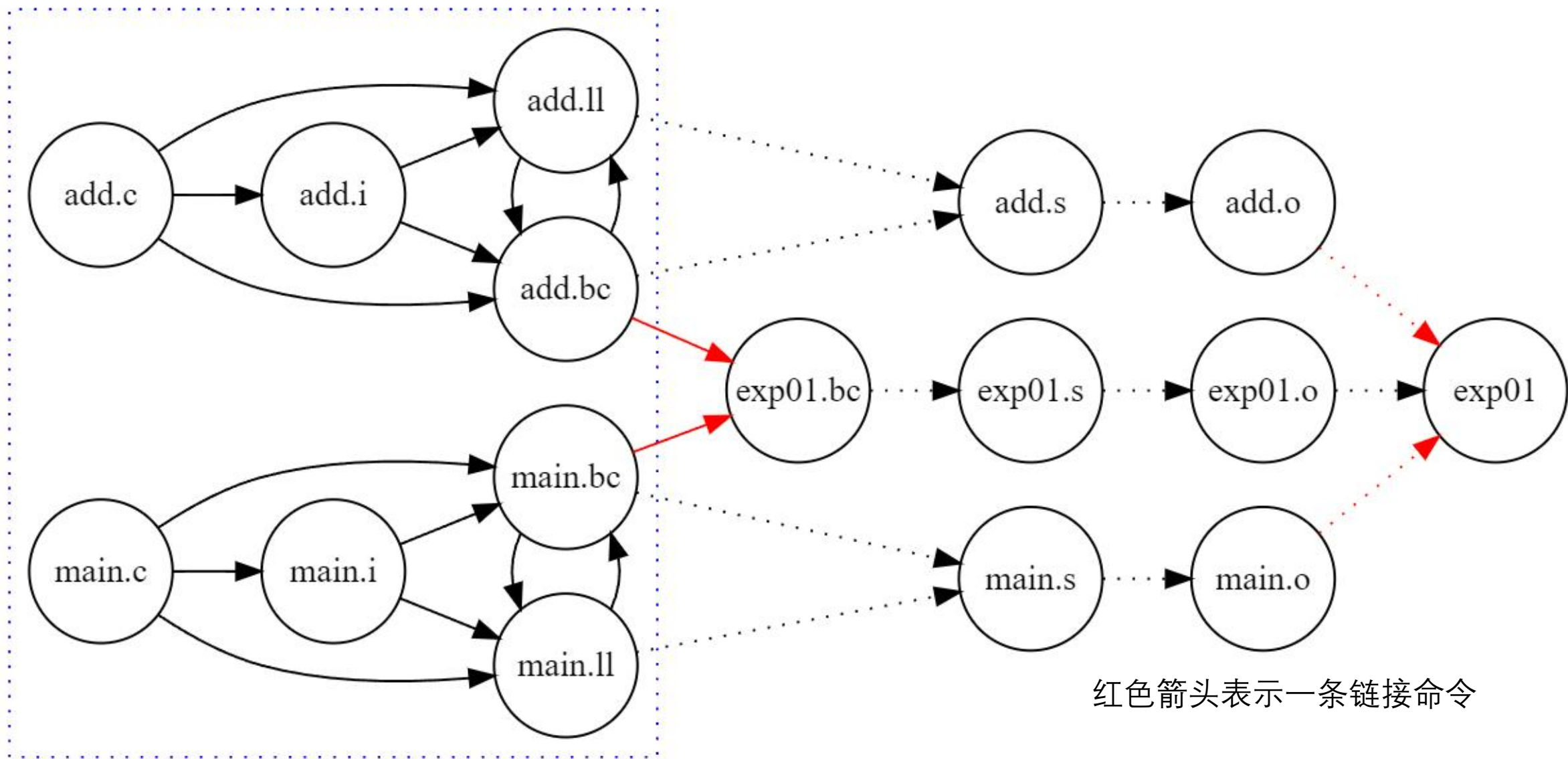
可执行程序

objdump -dS

预处理器等  
类似gcc, 后  
面不再详述



# clang编译可执行程序的文件变换



# clang工作流程-详细-预处理器(Preprocessor)

- 预处理器，主要是处理宏定义和文件包含等信息，例如#include所包含文件拷贝到C文件中，#define宏展开；预处理宏#ifdef等启用。
- 采用clang -E选项可对C程序预处理
- 具体命令  
clang -E -o main.i main.c  
clang -E -o add.i add.c



# clang工作流程-详细-编译器(Compiler)

- 主要功能是对源程序进行翻译，目标是汇编语言程序。其翻译主要包含把**字符流**(文本文件)形成记号 (Token) 流或符号流、检查是否满足C语言的语法要求、进行语义转换、代码优化、生成目标汇编代码等
- 采用clang -S选项可对C程序编译。可追加选项-O n进行代码优化
- 具体命令

```
clang -S -c -o main.s main.c
```

```
clang -S -c -o add.s add.c
```

```
clang -S -o main.s main.i
```

```
clang -S -o add.s add.i
```

# clang工作流程-详细-输出记号流

- 主要功能是对源程序进行词法分析，输出所识别出的所有Token，也就是语法产生式的终结符。
- 采用-fsyntax-only -Xclang -dump-tokens选项
- 具体命令

`clang -fsyntax-only -Xclang -dump-tokens main.c`

`clang -fsyntax-only -Xclang -dump-tokens add.c`

# clang工作流程-详细-输出记号流

```
1  #include "add.h"
2
3  int add(int a, int b)
4  {
5  |... return a + b;
6  }
```



```
int add(int a, int b);
```

```
int add(int a, int b)
{
|... return a + b;
}
```



int 'int'	[StartOfLine] Loc=<./add.h:4:1>
identifier 'add'	[LeadingSpace] Loc=<./add.h:4:5>
l_paren '('	Loc=<./add.h:4:8>
int 'int'	Loc=<./add.h:4:9>
identifier 'a'	[LeadingSpace] Loc=<./add.h:4:13>
comma ','	Loc=<./add.h:4:14>
int 'int'	[LeadingSpace] Loc=<./add.h:4:16>
identifier 'b'	[LeadingSpace] Loc=<./add.h:4:20>
r_paren ')'	Loc=<./add.h:4:21>
semi ';'	Loc=<./add.h:4:22>
int 'int'	[StartOfLine] Loc=<add.c:3:1>
identifier 'add'	[LeadingSpace] Loc=<add.c:3:5>
l_paren '('	Loc=<add.c:3:8>
int 'int'	Loc=<add.c:3:9>
identifier 'a'	[LeadingSpace] Loc=<add.c:3:13>
comma ','	Loc=<add.c:3:14>
int 'int'	[LeadingSpace] Loc=<add.c:3:16>

# clang工作流程-详细-输出抽象语法树

- 主要功能是对源程序进行语法分析，可产生出抽象语法树。
- 采用-fsyntax-only -Xclang -ast-dump选项
- 具体命令

`clang -fsyntax-only -Xclang -ast-dump main.c`

`clang -fsyntax-only -Xclang -ast-dump add.c`

# clang工作流程-详细-输出记号流

```
1  #include "add.h"
2
3  int add(int a, int b)
4  {
5      ... return a + b;
6  }
```



预处理

```
int add(int a, int b);
```

```
int add(int a, int b)
{
    ... return a + b;
}
```



```
TranslationUnitDecl 0x1cc38a8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x1cc40d0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
|  |-BuiltinType 0x1cc3e70 '__int128'
|  |-TypedefDecl 0x1cc4140 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
|  |  |-BuiltinType 0x1cc3e90 'unsigned __int128'
|  |-TypedefDecl 0x1cc4448 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
|  |  |-RecordType 0x1cc4220 'struct __NSConstantString_tag'
|  |    |-Record 0x1cc4198 '__NSConstantString_tag'
|  |-TypedefDecl 0x1cc44e0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
|  |  |-PointerType 0x1cc44a0 'char *'
|  |    |-BuiltinType 0x1cc3950 'char'
|  |-TypedefDecl 0x1cc47d8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag[1]'
|  |  |-ConstantArrayType 0x1cc4780 'struct __va_list_tag[1]' 1
|  |    |-RecordType 0x1cc45c0 'struct __va_list_tag'
|  |      |-Record 0x1cc4538 '__va_list_tag'
|  |-FunctionDecl 0x1d1a500 <./add.h:4:1, col:21> col:5 add 'int (int, int)'
|  |  |-ParmVarDecl 0x1d1a3a0 <col:9, col:13> col:13 a 'int'
|  |  |-ParmVarDecl 0x1d1a420 <col:16, col:20> col:20 b 'int'
|  |-FunctionDecl 0x1d1a730 prev 0x1d1a500 <add.c:3:1, line:6:1> line:3:5 add 'int (int, int)'
|  |  |-ParmVarDecl 0x1d1a610 <col:9, col:13> col:13 used a 'int'
|  |  |-ParmVarDecl 0x1d1a690 <col:16, col:20> col:20 used b 'int'
|  |-CompoundStmt 0x1d1a880 <line:4:1, line:6:1>
|    |-ReturnStmt 0x1d1a870 <line:5:5, col:16>
|      |-BinaryOperator 0x1d1a850 <col:12, col:16> 'int' '+'
|        |-ImplicitCastExpr 0x1d1a820 <col:12> 'int' <LValueToRValue>
|          |-DeclRefExpr 0x1d1a7e0 <col:12> 'int' lvalue ParmVar 0x1d1a610 'a' 'int'
|            |-ImplicitCastExpr 0x1d1a838 <col:16> 'int' <LValueToRValue>
|              |-DeclRefExpr 0x1d1a800 <col:16> 'int' lvalue ParmVar 0x1d1a690 'b' 'int'
```

# clang工作流程-详细-生成LLVM IR

- 主要功能是对源程序翻译生成LLVM IR
- 采用-emit-llvm选项, -S指定生成文本的, 不指定生成二进制的

- 具体命令

```
clang -c -S -emit-llvm -o main.ll main.c
```

```
clang -c -S -emit-llvm -o add.ll add.c
```


```
clang -c -emit-llvm -o main.bc main.c
```

```
clang -c -emit-llvm -o add.bc add.c
```



# clang工作流程-详细-生成LLVM IR

```
1  #include "add.h"
2
3  int add(int a, int b)
4  {
5      ... return a + b;
6  }
    
```



```
6  ; Function Attrs: noinline nounwind optnone uwtable
7  define dso_local i32 @add(i32 noundef %0, i32 noundef %1) #0 {
8      %3 = alloca i32, align 4
9      %4 = alloca i32, align 4
10     store i32 %0, i32* %3, align 4
11     store i32 %1, i32* %4, align 4
12     %5 = load i32, i32* %3, align 4
13     %6 = load i32, i32* %4, align 4
14     %7 = add nsw i32 %5, %6
15     ret i32 %7
16 }
```

IR定义了一个函数，函数名称为@add，形参为两个i32类型的%0和%1。原来的函数名add变成了@add，符号@开头的符号代表符号是全局唯一的。

%0和%1代表函数调用时传递的实参值，%3代表C语言函数中的形参a，%4代表形参b，默认通过alloca指令在栈中分配空间，之后通过第10和11行的store指令实现把实参的值赋值给形参a(%3)和b(%4)，从而实现函数的参数传递。

第12行到第14行实现表达式a+b的求值，先从栈中内存取值，然后相加赋值给变量%7。

第15行实现把a+b的值%7返回。

# clang工作流程-详细-LLVM IR转换

- 主要功能是实现LLVM IR的二进制与文本格式的转换
- llvm-dis实现二进制到文本, llvm-as实现文本到二进制

- 具体命令

```
llvm-as -o add.bc add.ll
```

```
llvm-as -o main.bc main.ll
```

```
llvm-dis -o main.ll main.bc
```

```
llvm-dis -o add.ll add.bc
```

```
clang -c -emit-llvm -o main.bc main.ll
```

```
clang -c -emit-llvm -o add.bc add.ll
```



# clang工作流程-详细-LLVM IR链接与运行

- 主要功能是实现LLVM IR的链接与JIT运行
- llvm-link实现多个IR的链接, lli实现IR的JIT运行
- 具体命令  
llvm-link -o exp01.bc add.bc main.bc  
lli exp01.bc

# clang工作流程-详细-生成汇编代码

- 主要功能是对源程序进行翻译生成汇编代码。Llc命令通过指定目标可生成不同架构的汇编，默认本地架构。

- 具体命令

```
clang -S -c -o main.s main.c
```

```
clang -S -c -o add.s add.c
```

```
clang -S -c -o main.s main.ll
```

```
clang -S -c -o add.s add.ll
```

```
clang -S -c -o main.s main.bc
```

```
clang -S -c -o add.s add.bc
```

```
llc -filetype=asm --relocation-model=pic -o main.s main.ll
```

```
llc -filetype=asm --relocation-model=pic -o add.s add.ll
```

```
llc -filetype=asm --relocation-model=pic -o main.s main.bc
```

```
llc -filetype=asm --relocation-model=pic -o add.s add.bc
```

# clang工作流程-详细-生成目标文件

- 主要功能是对源程序进行翻译生成目标文件(二进制格式), 可通过objdump等命令查看

- 具体命令

clang -c -o main.o main.c

clang -c -o add.o add.c

clang -c -o main.o main.ll

clang -c -o add.o add.ll

clang -c -o main.o main.bc

clang -c -o add.o add.bc

# clang工作流程-详细-生成可执行程序

- 主要功能是对源程序或目标文件进行编译链接生成可执行程序（二进制），可通过objdump等命令查看

- 具体命令

```
clang -o exp01 main.c add.c
```

```
clang -o exp01 main.ll add.ll
```

```
clang -o exp01 main.bc add.bc
```

```
clang -o exp01 main.s add.s
```

```
clang -o exp01 main.o add.o
```