

第2讲 编译实验工具简介

徐伟

0551-63607954, [xuwei hf@ustc.edu.cn](mailto:xuweihf@ustc.edu.cn)

中国科学技术大学 计算机科学与技术学院

目录



□ **GCC/G++ 编译C/C++ 程序**

□ **Makefile 基础**

□ **汇编语言基础**

□ **访问者模式**

□ **Git**

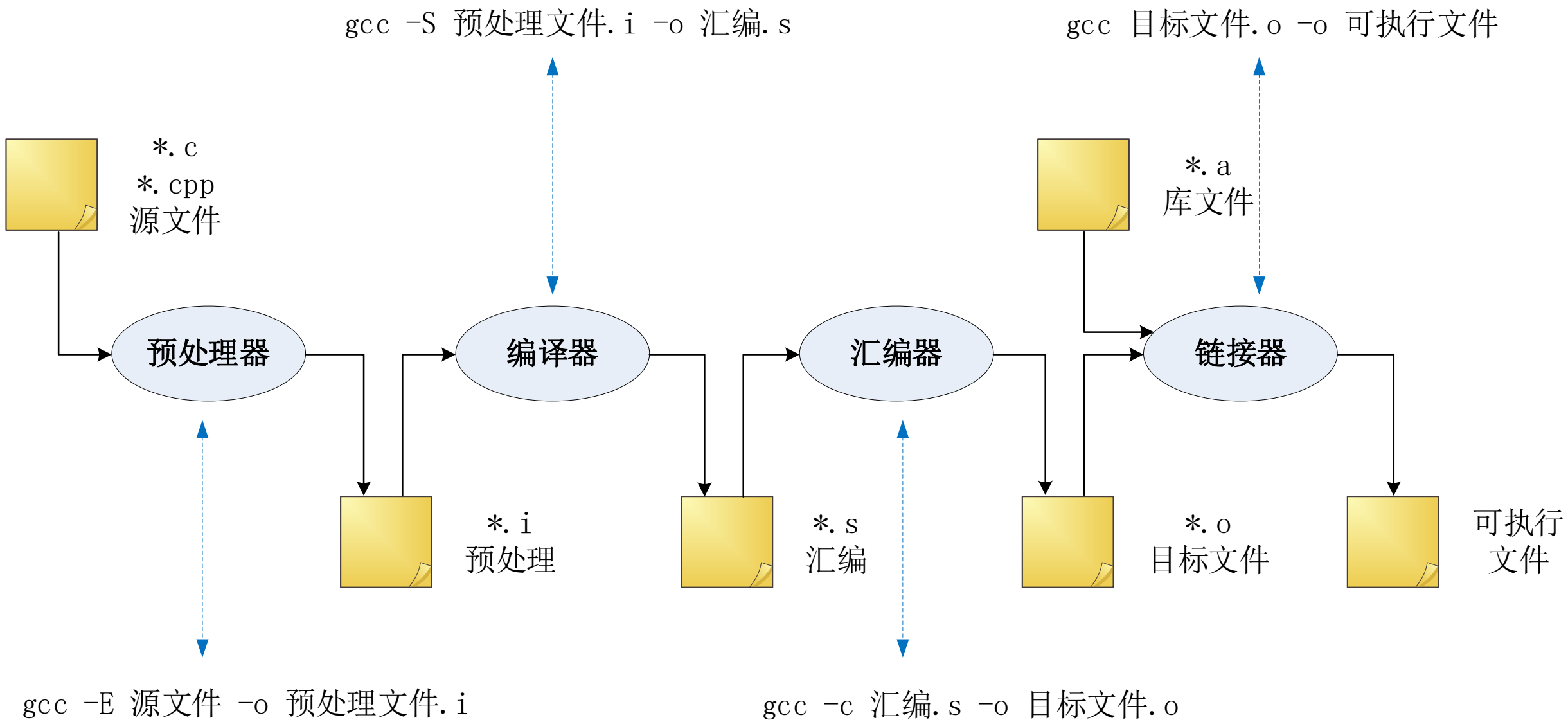
GCC/G++ 编译C/C++程序



gcc/g++选项	功能	目标文件常用后缀
-E (大写)	预处理指定的源文件，不进行编译。	.i
-S (大写)	编译指定的源文件，生成 汇编文件，但是不进行汇编。	.s
-c	编译、汇编指定的源文件，生成目标文件，但是不进行链接。	.o
-o	指定生成目标文件的文件名	
-m32	按照32位进行编译	
-m64	按照64位进行编译	

有关更多编译指令，可自行查看[GCC手册](#)

GCC/G++编译C/C++程序



□ 示例程序sample.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, world!\n");
6     return 0;
7 }
```

gcc -E sample.c -o sample.i

```
1 # 1 "sample.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "/usr/include/stdc-predef.h" 1 3 4
```

```
423 extern int fclose (FILE *__stream);
490 extern int fprintf (FILE *__restrict __stream,
491                    const char *__restrict __format, ...);
```

```
851 int main()
852 {
853     printf("Hello, world!\n");
854     return 0;
855 }
```

□ 示例程序sample.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, world!\n");
6     return 0;
7 }
```

gcc -S sample.i -o sample.s

```
1      .file   "sample.c"
2      .section      .rodata
3      .LC0:
4          .string "Hello, world!"
5      .text
6      .globl  main
7      .type   main, @function
8      main:
9      .LFB0:
10         pushq   %rbp
11         movq    %rsp, %rbp
12         movl    $.LC0, %edi
13         call    puts
14         movl    $0, %eax
15         popq    %rbp
16         ret
```

□ 示例程序sample.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello, world!\n");
6     return 0;
7 }
```

`gcc -c sample.s -o sample.o`

`gcc sample.o -o sample`

```
root@evassh-4250186:/data/workspace/myshixun# gcc -c sample.s -o sample.o
root@evassh-4250186:/data/workspace/myshixun# ls
platform-script sample.c sample.i sample.o sample.s secret
root@evassh-4250186:/data/workspace/myshixun# gcc sample.o -o sample
root@evassh-4250186:/data/workspace/myshixun# ls
platform-script sample sample.c sample.i sample.o sample.s secret
root@evassh-4250186:/data/workspace/myshixun# ./sample
Hello, world!
```

GCC/G++ 编译C/C++程序



gcc/g++选项	功能
-D	进行宏定义
-nostdinc	搜寻include 的文件路径中去掉标准的c语言头文件搜索路径
-I	指定头文件搜索路径
-nostdlib	在链接的时候不把系统相关的启动文件和系统相关的库连接进来。
-Wl	将后面的参数传递给链接器
-lc	链接libc库文件（libc库文件中有printf等函数）


□宏定义

test.c

```
1 #include <stdio.h>
2 #ifdef NEG
3 #define M -4
4 #else
5 #define M 4
6 #endif
7 int main()
8 {
9     printf("M=%d\n", M);
10    return 0;
11 }
```

gcc -E test.c -o test.i

```
1 ...
2 ...
3 ...
4 ...
5 ...
6 # 7 "test.c"
7 int main()
8 {
9     printf("Hello, world!\nM=%d\n", 4);
10    return 0;
11 }
```

A red arrow points from the right side of the slide towards the macro expansion in the preprocessor output, specifically highlighting the line where the macro M is replaced by the value 4.


□宏定义

test.c

```
1 #include <stdio.h>
2 #ifdef NEG
3 #define M -4
4 #else
5 #define M 4
6 #endif
7 int main()
8 {
9     printf("M=%d\n", M);
10    return 0;
11 }
```


gcc -E test.c -o test.i

```
1 ...
2 ...
3 ...
4 ...
5 ...
6 # 7 "test.c"
7 int main()
8 {
9     printf("Hello, world!\nM=%d\n", 4);
10    return 0;
11 }
```



gcc **-DNEG** -E test.c -o test.i

```
1 ...
2 ...
3 ...
4 ...
5 ...
6 # 7 "test.c"
7 int main()
8 {
9     printf("Hello, world!\nM=%d\n", -4);
10    return 0;
11 }
```



命令中增加了 **-DNEG** 相当于增加了一个名为NEG的宏，则在预处理阶段，根据代码将M define 为 -4。即将全部的M用-4来代替。

□ 指定头文件搜索路径

■ 查找默认头文件路径

❖ gcc -xc -v

```
#include "..." search starts here:
#include <...> search starts here:
/usr/lib/gcc/x86_64-linux-gnu/5/include
/usr/local/include
/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
End of search list.
```

■ 指定搜索路径

❖ gcc -E -nostdinc sample-io.c -o sample-io.i -I /usr/lib/gcc/x86_64-linux-gnu/5/include -I /usr/include

□指定库文件链接路径

■查找默认链接库路径

❖ gcc sample-io.c -o sample-io -Wl,--verbose

```
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crti.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crti.o
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o succeeded
/usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o
attempt to open /tmp/cc0pop5N.o succeeded
/tmp/cc0pop5N.o
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/libgcc.so failed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/libgcc.a succeeded
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/libgcc_s.so succeeded
-lgcc_s (/usr/lib/gcc/x86_64-linux-gnu/5/libgcc_s.so)
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/libc.so failed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/libc.a failed
attempt to open /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/libc.so succeeded
```

■指定链接路径

❖ gcc -nostdlib sample-io.c -o sample-io -Wl,/usr/lib/x86_64-linux-gnu/crt1.o,/usr/lib/x86_64-linux-gnu/crti.o,/usr/lib/x86_64-linux-gnu/crtn.o -lc

目录



□ GCC/G++ 编译C/C++ 程序

□ Makefile基础

□ 汇编语言基础

□ 访问者模式

□ Git

显示规则

- 目标文件：依赖文件
- 原则上第一个文件就是最终的文件
- 缩进必须TAB键

```
1 test:test.o      #目标文件test, 依赖文件test.o
2     gcc test.o -o test
3 test.o:test.s
4     gcc -c test.s -o test.o
5 test.s:test.i
6     gcc -S test.i -o test.s
7 test.i:test.c
8     gcc -E test.c -o test.i
```

假设单文件 test.c 编译

采用递归方式书写

使用make命令自动编译出
目标文件test

#表示注释

变量

- 变量一般定义在文件头部
- 使用\$(变量名)表示变量

符号	含义
=	赋值
+=	增加赋值
:=	常量赋值

```
1  TAR = test      #赋值
2  CC := gcc       #常量赋值
3
4  $(TAR):test.o   #常量使用
5      $(CC) test.o -o $(TAR)
6  test.o:test.s
7      $(CC) -c test.s -o test.o
8  test.s:test.i
9      $(CC) -S test.i -o test.s
10 test.i:test.c
11     $(CC) -E test.c -o test.i
```

□多文件编译

示例circle.c circle.h

cube.c cube.h

test.c编译

```
1 TAR = test
2 CC := gcc
3
4 $(TAR):circle.o cube.o test.o
5     $(CC) circle.o cube.o test.o -o $(TAR)
6 circle.o:circle.c
7     $(CC) -c circle.c -o test.o
8 cube.o:cube.c
9     $(CC) -c cube.c -o cube.o
10 test.o:test.c
11     $(CC) -c test.c -o test.o
12
13 .PHONY:      #伪目标
14 clean:      #make clean调用
15     rm -rf circle.o cube.o test.o test
```


□多文件编译

示例circle.c circle.h

cube.c cube.h

test.c编译

```
1 TAR = test
2 OBJ = circle.o cube.o test.o
3 CC := gcc
4
5 $(TAR):$(OBJ)
6     $(CC) $(OBJ) -o $(TAR)
7 %.o:%.c
8     $(CC) -c %.c -o %.o
9
10 .PHONY:      #伪目标
11 clean:       #make clean调用
12     rm -rf $(OBJ) $(TAR)
```

通配符 %.c %.o表示任意的.c或.o文件

*.c *.o表示所有的.c或.o文件

□通配符

通配符	含义
%	模式字符，用来通配任意个字符
\$@	目标文件名。多目标模式规则中，它代表的是触发规则被执行的文件名
\$\$	当目标文件是一个静态库文件时，代表静态库的一个成员名
\$<	第一个依赖的文件名
\$?	所有比目标文件更新的依赖文件列表
\$\$	代表的是所有依赖文件列表
\$\$	保留了依赖文件中重复出现的文件。主要用在程序链接时库的交叉引用场合

□通配符

```
1 TAR = test
2 OBJ = circle.o cube.o test.o
3 CC := gcc
4
5 $(TAR):$(OBJ)
6     $(CC) $(OBJ) -o $(TAR)
7 %.o:%.c
8     $(CC) -c %.c -o %.o
9
10 .PHONY:      #伪目标
11 clean:       #make clean调用
12     rm -rf $(OBJ) $(TAR)
```

```
1 TAR = test
2 OBJ = circle.o cube.o test.o
3 CC := gcc
4
5 $(TAR):$(OBJ)
6     $(CC) $^ -o $@
7 %.o:%.c
8     $(CC) -c $^ -o $@
9
10 .PHONY:      #伪目标
11 clean:       #make clean调用
12     rm -rf $(OBJ) $(TAR)
```

通配符 \$@ \$^

目录



□ GCC/G++ 编译C/C++ 程序

□ Makefile基础

□ 汇编语言基础

□ 访问者模式

□ Git

x86汇编指令 (AT&T语法)



□寄存器

寄存器	16位	32位	64位
累加寄存器	AX	EAX	RAX
基址寄存器	BX	EBX	RBX
计数寄存器	CX	ECX	RCX
数据寄存器	DX	EDX	RDY
堆栈基指针	BP	EBP	RBP
变址寄存器	SI	ESI	RSI
堆栈顶指针	SP	ESP	RSP
指令寄存器	IP	EIP	RIP

**64位汇编还有r8~r15
8个寄存器**

❑ X86架构Unix、Linux系统中更多采用AT&T格式

❑ AT&T格式，用\$前缀表示一个立即数

AT&T格式	Intel格式
<code>pushl \$1</code>	<code>pushl 1</code>

❑ AT&T格式，寄存器加%前缀

AT&T格式	Intel格式
<code>pushl %eax</code>	<code>pushl eax</code>

□ AT&T格式，目标操作数在源操作数右侧

AT&T格式	Intel格式
<code>movl \$1, %eax</code>	<code>mov eax, 1</code>

□ AT&T格式，操作数字长由操作符最后一个字母决定，后缀b/w/l/q分别表示8/16/32/64位

AT&T格式	Intel格式
<code>movb \$1, %al</code>	<code>mov al, byte ptr [bx]</code>
<code>movw \$1, %ax</code>	<code>mov ax, word ptr [bx]</code>

□ 程序调用框架介绍

```
1 int test(int a)
2 {
3     return a;
4 }
5 int main()
6 {
7     return test(1);
8 }
```

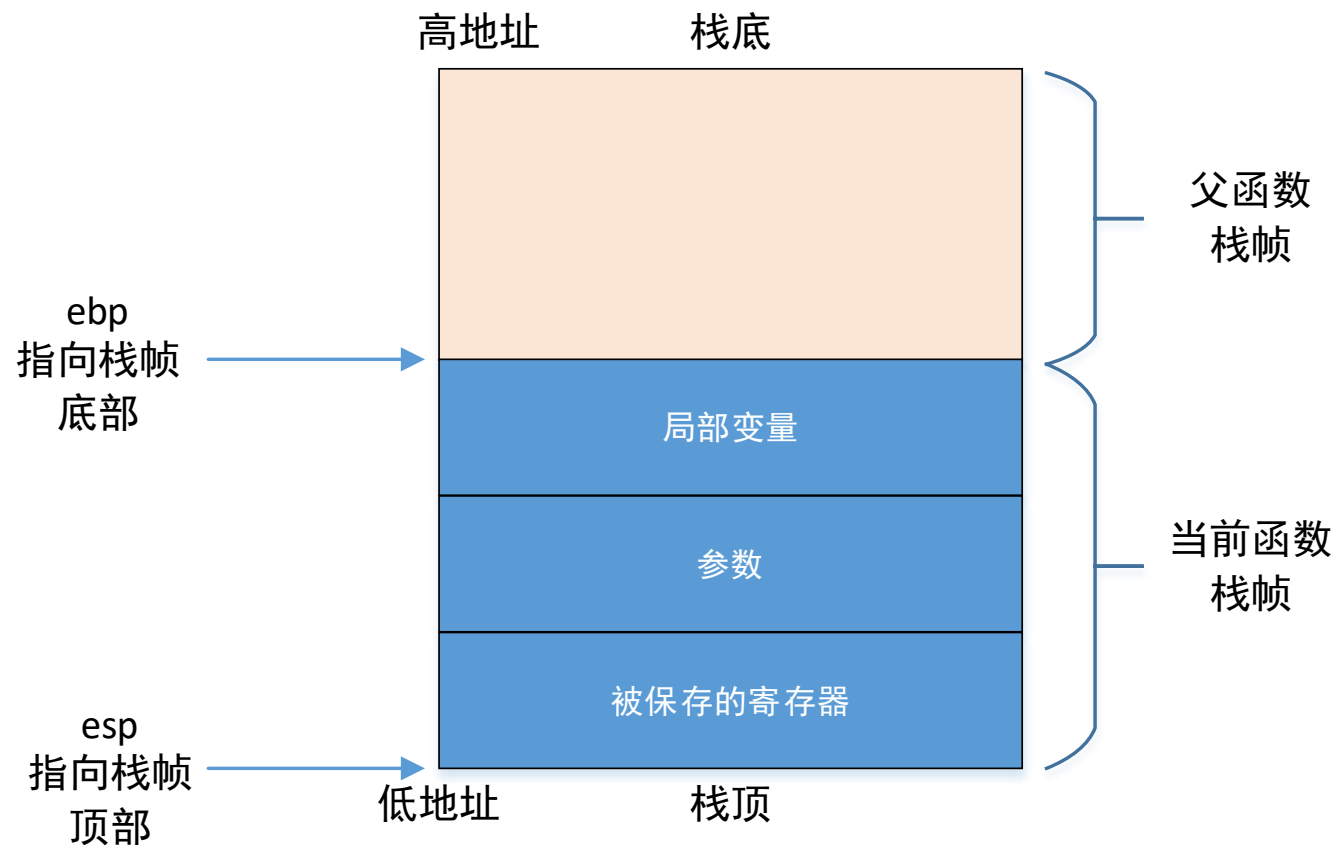
gcc -S -m32 test.c -o test.s

汇编核心代码:

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```


□ 程序调用

- 函数调用的内存申请和释放是通过对栈进行操作来完成的
- 被调用函数在栈内申请一块区域，这个区域被称为栈帧

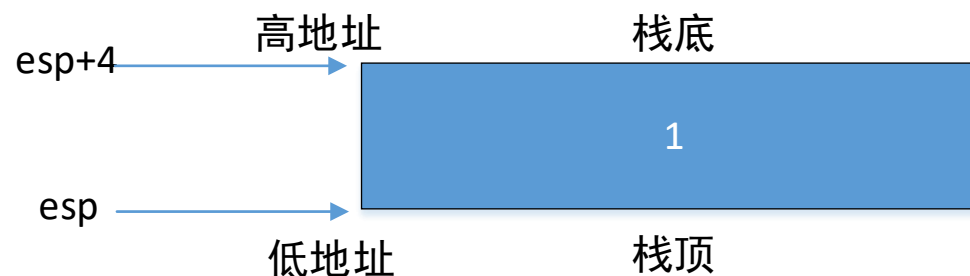


□ 程序调用

■ main函数调用test前

```
1 test:
2     pushl    %ebp
3     movl     %esp, %ebp
4     movl     8(%ebp), %eax
5     popl     %ebp
6     ret
7 main:
8     pushl    %ebp
9     movl     %esp, %ebp
10    pushl    $1
11    call     test
12    addl     $4, %esp
13    leave
14    ret
```

- 1、保存main父函数栈帧地址ebp
- 2、将当前栈顶作为main栈帧起始地址
- 3、压入调用函数实参



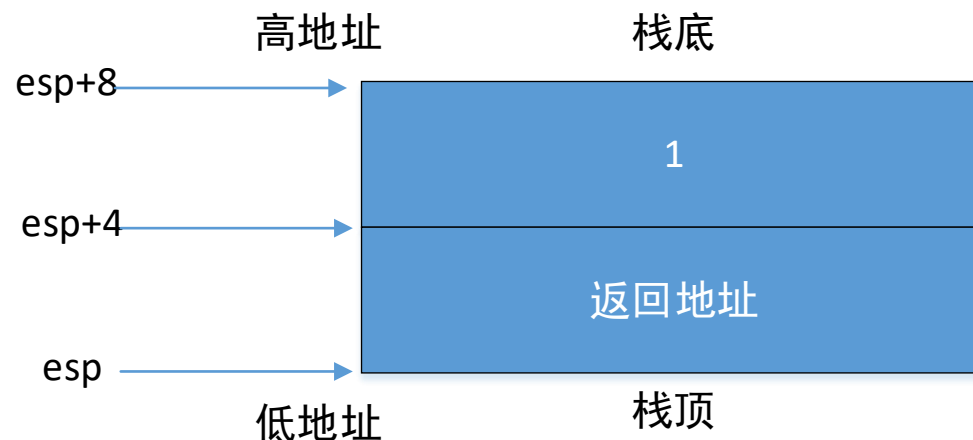
□ 程序调用

■ 执行call函数

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```

4、返回地址压栈

call指令先将当前EIP（返回地址）入栈，再执行jmp



□ 程序调用

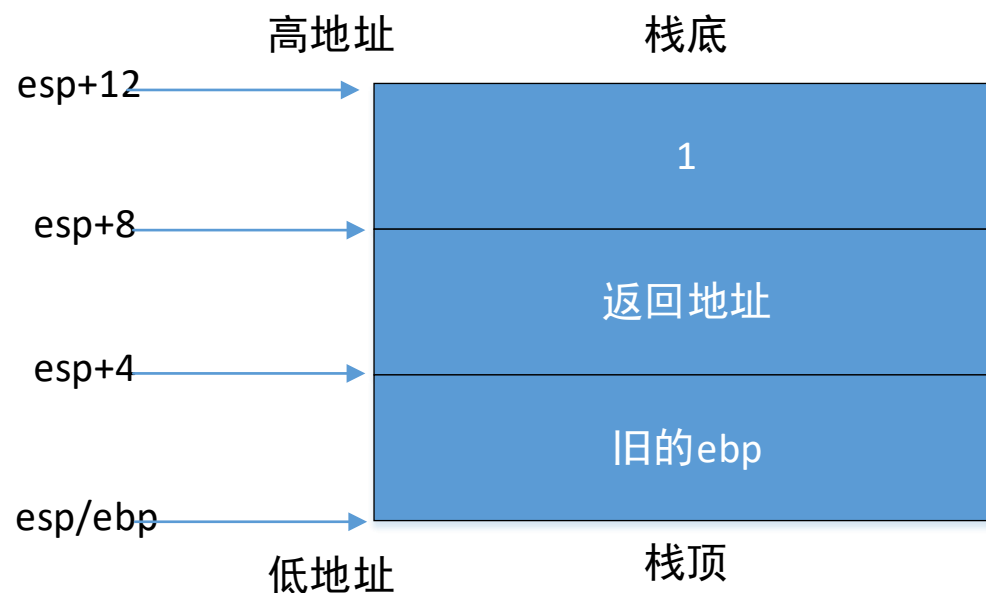
■ 执行test函数

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```

5、父函数ebp保存进栈

6、更新子函数ebp

7、读入参数

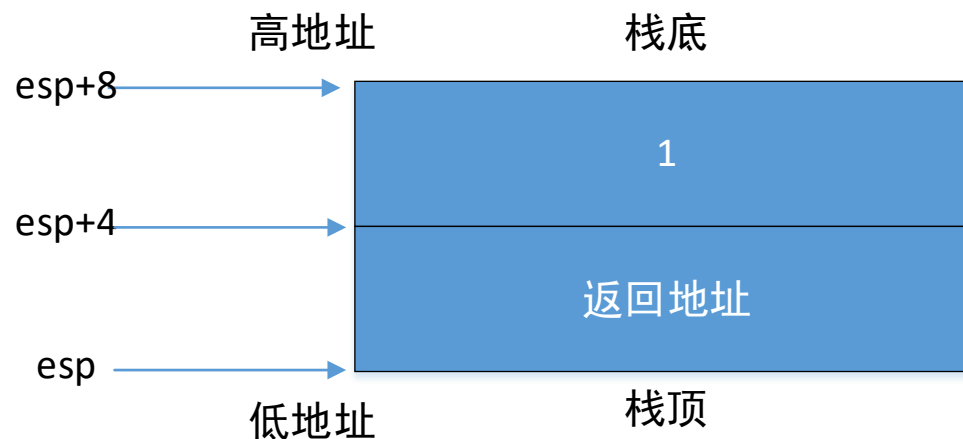


□ 程序调用

■ 恢复父函数ebp

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```

8、恢复父函数ebp

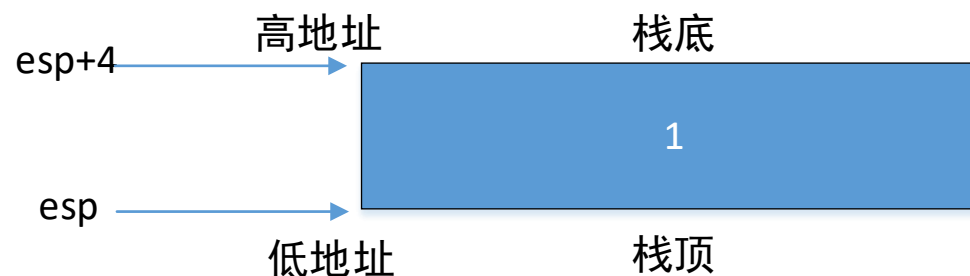


□ 程序调用

■ 执行test ret语句

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```

9、恢复旧的执行地址



□ 程序调用

■ 执行main函数leave

```
1 test:
2     pushl   %ebp
3     movl    %esp, %ebp
4     movl    8(%ebp), %eax
5     popl    %ebp
6     ret
7 main:
8     pushl   %ebp
9     movl    %esp, %ebp
10    pushl   $1
11    call    test
12    addl    $4, %esp
13    leave
14    ret
```

10、删除test函数实参

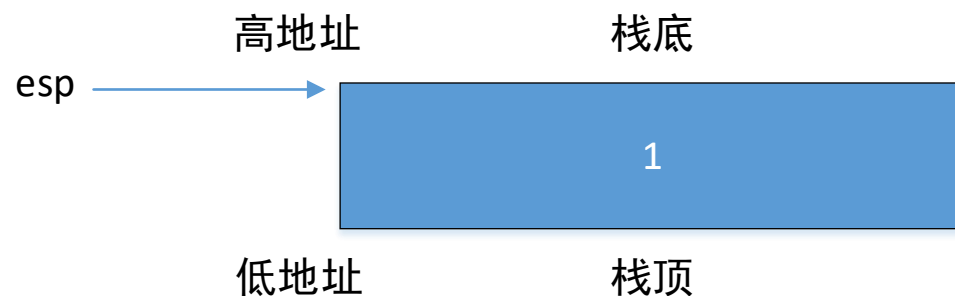
11、leave语句

恢复main父函数ebp

movl %ebp, %esp

pop %ebp

12、执行main父函数的返回地址



□ CFI调用框架

■ Call Frame Instructions

CFI 调用框架指令

■ 为实现堆栈回绕、异常处理

```
1      .file   "test.c"
2      .text
3      .globl  test
4      .type   test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19     .LFE0:
20         .size   test, .-test
21         .globl  main
22         .type   main, @function
```

实际32位汇编

□ CFI调用框架

■ Call Frame Instructions

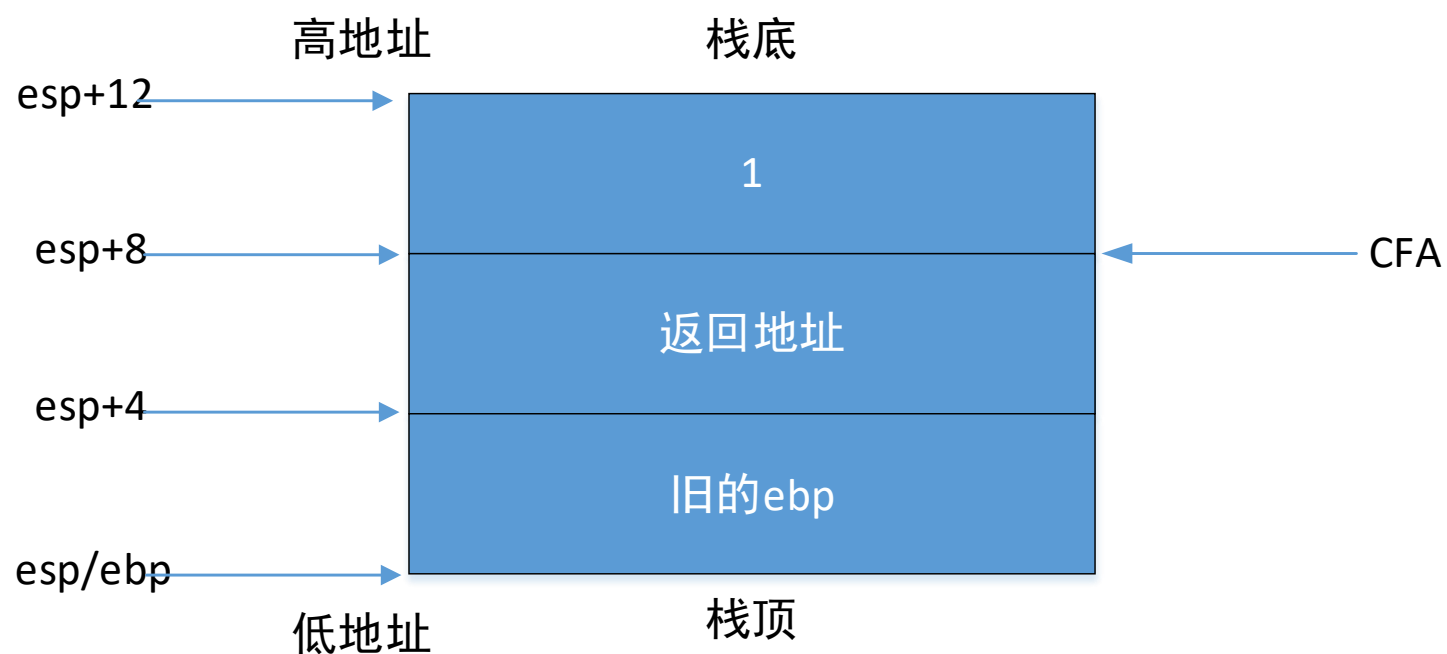
CFI 调用框架指令

■ 为实现堆栈回绕、异常处理

■ Canonical Frame Address

CFA, 标准框架地址

■ 在前一个调用框架中调用当前函数时的栈顶指针



□ CFI调用框架

CFI指令	功能
.cfi_startproc	函数开始
.cfi_endproc	函数结束

```
1      .file   "test.c"
2      .text
3      .globl  test
4      .type   test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19     .LFE0:
20         .size   test, .-test
21         .globl  main
22         .type   main, @function
```

CFI调用框架

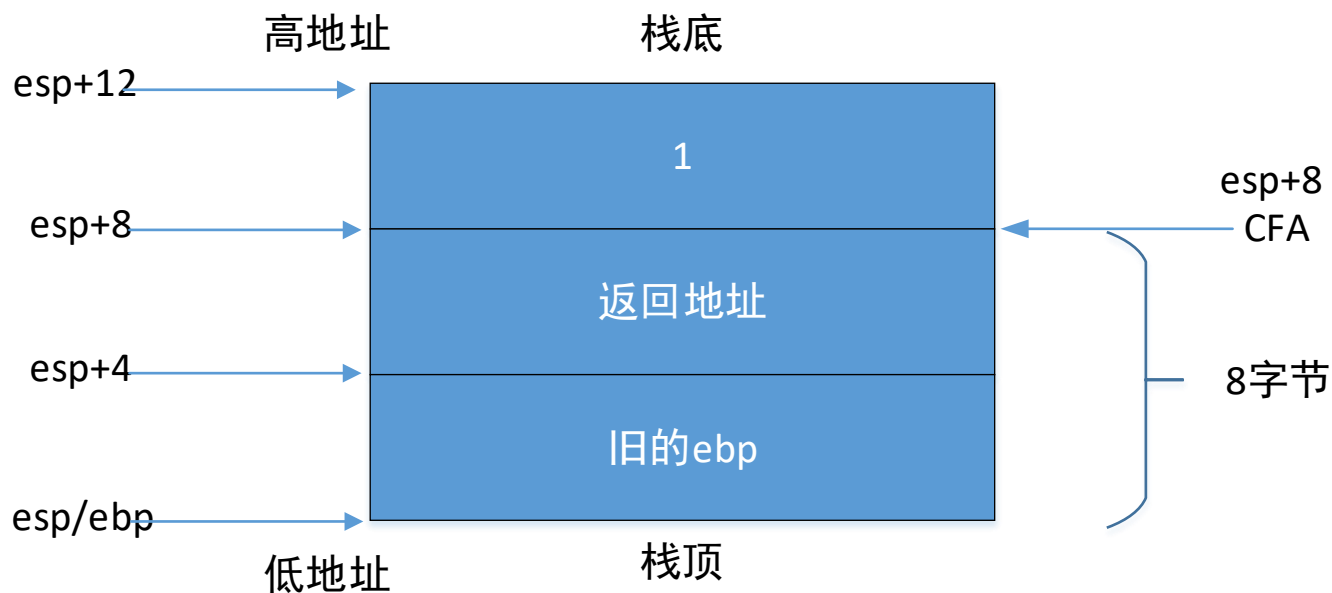
CFI指令	功能
.cfi_def_cfa_offset 8	此处距离CFA地址为8字节



```
1      .file    "test.c"
2      .text
3      .globl   test
4      .type    test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19     .LFE0:
20         .size    test, .-test
21         .globl   main
22         .type    main, @function
```

□ CFI调用框架

CFI指令	功能
.cfi_offset 5, -8	把5号寄存器原先值保存在距离CFA-8的位置



X86架构8个通用寄存器eax, ebx, ecx, edx, esp, **ebp**, esi, edi

```
1      .file    "test.c"
2      .text
3      .globl   test
4      .type    test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19  .LFE0:
20         .size    test, .-test
21         .globl   main
22         .type    main, @function
```

□ CFI调用框架

CFI指令	功能
.cfi_def_cfa_register 5	从此处, 试用5号寄存器(ebp)作为计算CFA的基址寄存器

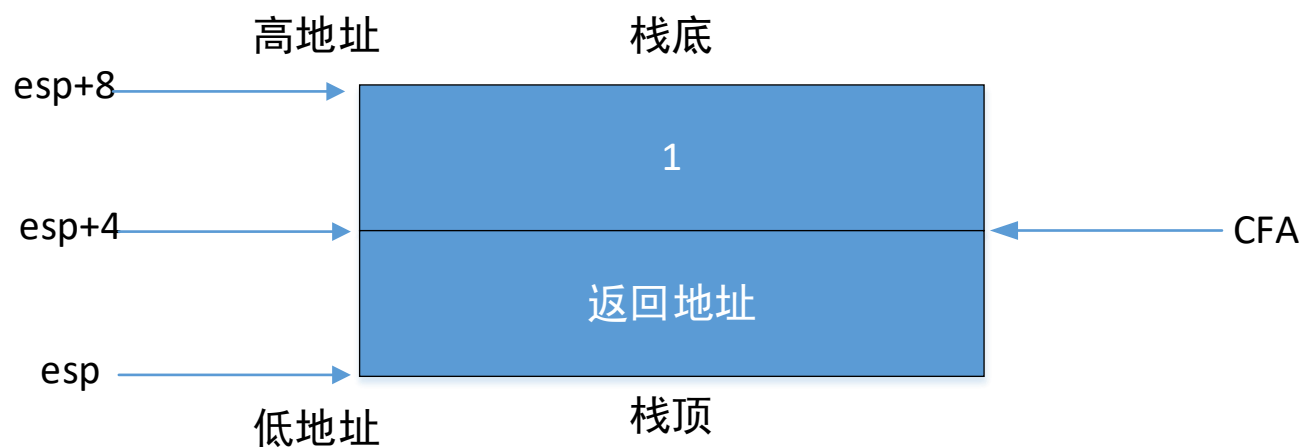


X86架构8个通用寄存器eax, ebx, ecx, edx, esp, **ebp**, esi, edi

```
1      .file    "test.c"
2      .text
3      .globl   test
4      .type    test, @function
5
6  test:
7      .LFB0:
8          .cfi_startproc
9          pushl   %ebp
10         .cfi_def_cfa_offset 8
11         .cfi_offset 5, -8
12         movl    %esp, %ebp
13         .cfi_def_cfa_register 5
14         movl    8(%ebp), %eax
15         popl    %ebp
16         .cfi_restore 5
17         .cfi_def_cfa 4, 4
18         ret
19     .LFE0:
20         .size    test, .-test
21         .globl   main
22         .type    main, @function
```

□ CFI调用框架

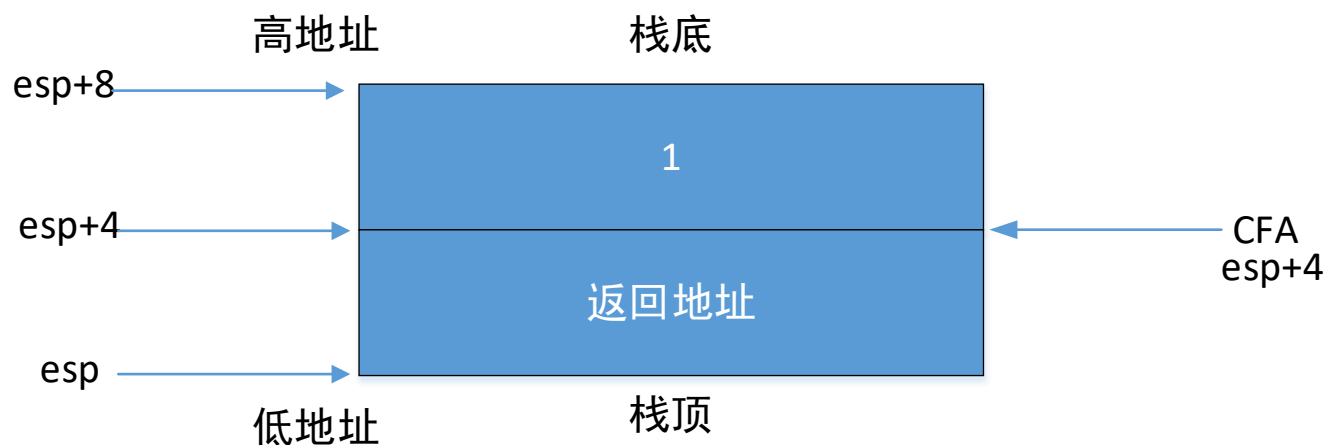
CFI指令	功能
.cfi_restore 5	5号寄存器(ebp)恢复函数开始的值



```
1      .file   "test.c"
2      .text
3      .globl  test
4      .type   test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19  .LFE0:
20         .size   test, .-test
21         .globl  main
22         .type   main, @function
```

□ CFI调用框架

CFI指令	功能
.cfi_def_cfa 4, 4	重新定义CFA, CFA为4号寄存器(esp)所指位置加4字节



```
1      .file   "test.c"
2      .text
3      .globl  test
4      .type   test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19  .LFE0:
20         .size   test, .-test
21         .globl  main
22         .type   main, @function
```

X86架构8个通用寄存器eax, ebx, ecx, edx, **esp**, ebp, esi, edi

```
1      .file   "test.c"
2      .text
3      .globl  test
4      .type   test, @function
5  test:
6      .LFB0:
7          .cfi_startproc
8          pushl   %ebp
9          .cfi_def_cfa_offset 8
10         .cfi_offset 5, -8
11         movl    %esp, %ebp
12         .cfi_def_cfa_register 5
13         movl    8(%ebp), %eax
14         popl    %ebp
15         .cfi_restore 5
16         .cfi_def_cfa 4, 4
17         ret
18         .cfi_endproc
19  .LFE0:
20         .size   test, .-test
21         .globl  main
22         .type   main, @function
```

```
23  main:
24      .LFB1:
25          .cfi_startproc
26          pushl   %ebp
27          .cfi_def_cfa_offset 8
28          .cfi_offset 5, -8
29          movl    %esp, %ebp
30          .cfi_def_cfa_register 5
31          pushl   $1
32          call    test
33          addl    $4, %esp
34          leave
35          .cfi_restore 5
36          .cfi_def_cfa 4, 4
37          ret
38          .cfi_endproc
39  .LFE1:
40         .size   main, .-main
41         .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609"
42         .section        .note.GNU-stack,"",@progbits
```


目录



□ GCC/G++ 编译C/C++ 程序

□ Makefile 基础

□ 汇编语言基础

□ 访问者模式

□ Git

□访问者模式定义

- 表示一个作用于某对象结构中的各个元素的操作。访问者模式可以在不改变各元素的类的前提下定义作用于这些元素的新操作
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

□访问者模式的优点

- 为操作存储不同类型元素的对象结构提供了一种解决方案
- 用户可以对不同类型的元素施加不同的操作

设计模式-访问者模式



访问者模式结构

■ Visitor

Declare a visit operation

■ ConcreteVisitor

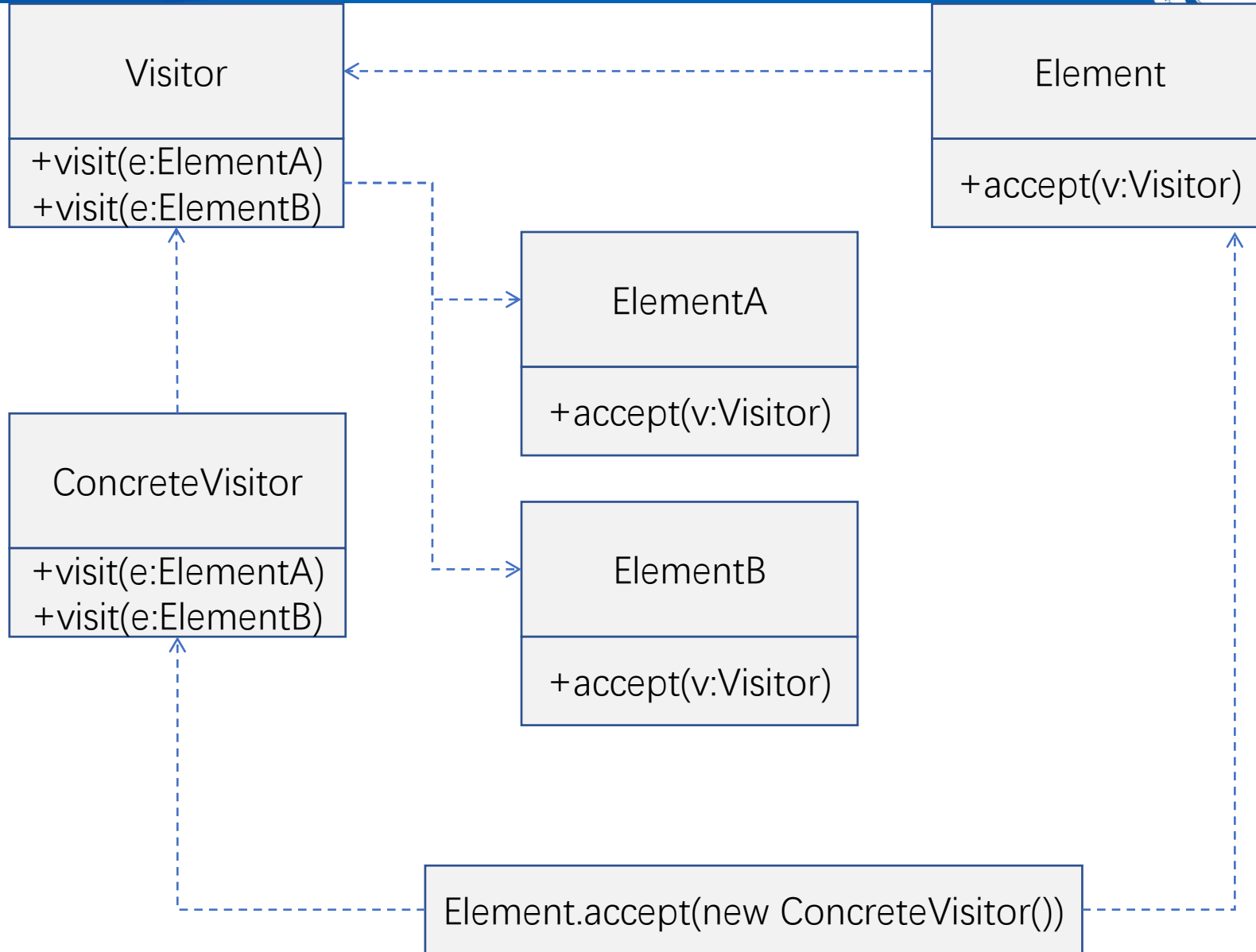
Implement each visit

■ Element

Define an Accept Operation

■ ConcreteElement

Implements Accept



设计模式-访问者模式



□访问者模式

- 被访问对象类都有accept方法用来接受访问者对象

```
ElementA.accept(Visitor V) { V.visit(this) }
```

- 访问者拥有visit方法，对访问到的对象结构中不同类型的元素作不同反应

```
ConcreteVisitor.visit(ElementA) {...}
```

```
ConcreteVisitor.visit(ElementB) {...}
```

实现对具体对象的操作

- 每一个元素accept方法，元素accept方法回调访问者visit
- 访问者得以处理对象结构的每一个元素
- 针对对象结构设计不同，访问者类来完成不同的操作

设计模式-访问者模式



□访问者模式示例

- 设计基类Exp和两个子类IntExp、AddExp；一个访问者类Visitor
- 通过访问者模式计算1到4之和

设计模式-访问者模式

```
1  /* Visitor.h */
2  class AddExp;
3  class IntExp;
4  class Visitor
5  {
6  public:
7      int result = 0;
8      virtual void visit(AddExp*);
9      virtual void visit(IntExp*);
10 };
```

Visitor.h

```
1  #include "Visitor.h"
2  class Exp{
3  public:
4      virtual void accept(Visitor&v) = 0;
5  };
6  class AddExp : public Exp{
7  public:
8      Exp* rhs;
9      Exp* lhs;
10     AddExp(Exp* lhs, Exp* rhs) : lhs(lhs), rhs(rhs) {}
11     virtual void accept(Visitor&v) override final;
12 };
13 class IntExp : public Exp{
14 public:
15     int value;
16     IntExp(int value) : value(value) {}
17     virtual void accept(Visitor&v) override final;
18 };
```

Exp.h

设计模式-访问者模式



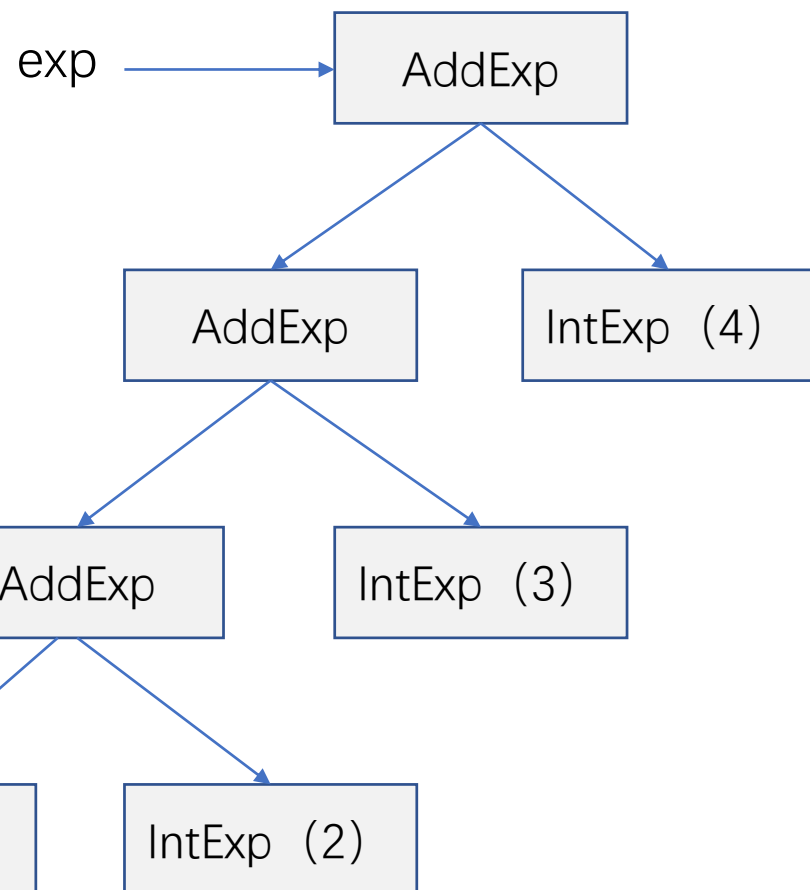
```
1  /* visitor.cpp */
2  #include "Exp.h"
3  void Visitor::visit(AddExp* add_exp){
4      add_exp->lhs->accept(*this);
5      add_exp->rhs->accept(*this);
6  }
7  void Visitor::visit(IntExp* int_exp){
8      result += int_exp->value;
9  }
10 void AddExp::accept(Visitor & v){
11     v.visit(this);
12 }
13 void IntExp::accept(Visitor & v){
14     v.visit(this);
15 }
```

Visitor.cpp

```
1  /* main.cpp */
2  #include<iostream>
3  #include<string>
4  #include "Exp.h"
5  using namespace std;
6  int main(){
7      Exp* exp = new IntExp(1);
8      for(int i = 2; i < 5; i++){
9          exp = new AddExp(exp, new IntExp(i));
10     }
11
12     Visitor CalSum;
13     exp->accept(CalSum);
14     cout << "Result is " << CalSum.result << endl;
15     return 0;
16 }
```

main.cpp

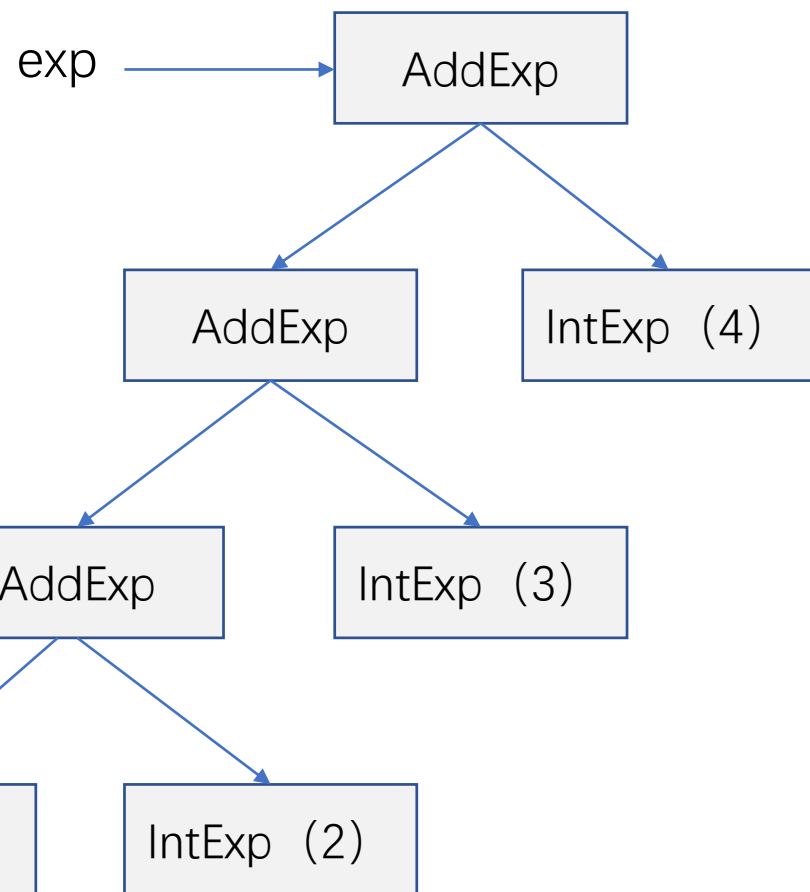
设计模式-访问者模式



```
1  /* main.cpp */
2  #include<iostream>
3  #include<string>
4  #include "Exp.h"
5  using namespace std;
6  int main(){
7      Exp* exp = new IntExp(1);
8      for(int i = 2; i < 5; i++){
9          exp = new AddExp(exp, new IntExp(i));
10     }
11
12     visitor CalSum;
13     exp->accept(CalSum);
14     cout << "Result is " << CalSum.result << endl;
15     return 0;
16 }
```

main.cpp

设计模式-访问者模式



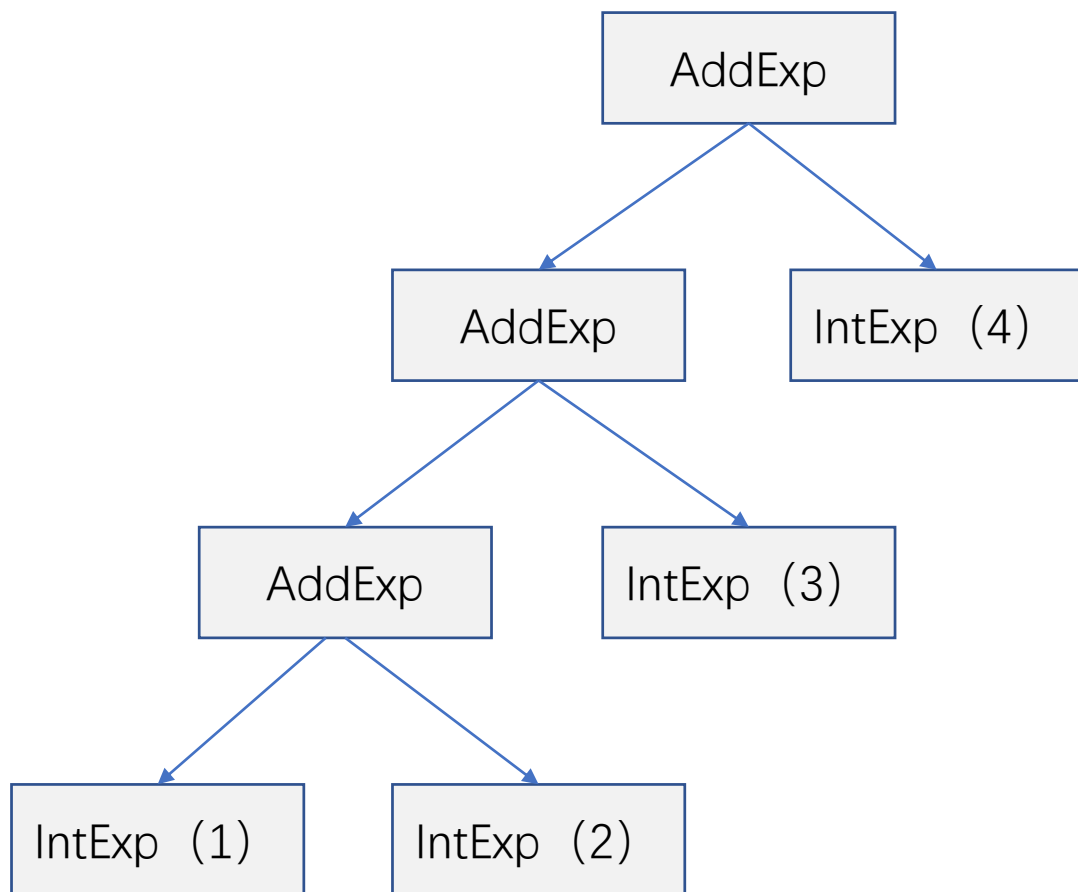
```
1  /* visitor.cpp */
2  #include "Exp.h"
3  void Visitor::visit(AddExp* add_exp){
4      add_exp->lhs->accept(*this);
5      add_exp->rhs->accept(*this);
6  }
7  void Visitor::visit(IntExp* int_exp){
8      result += int_exp->value;
9  }
10 void AddExp::accept(Visitor & v){
11     v.visit(this);
12 }
13 void IntExp::accept(Visitor & v){
14     v.visit(this);
15 }
```

Visitor.cpp

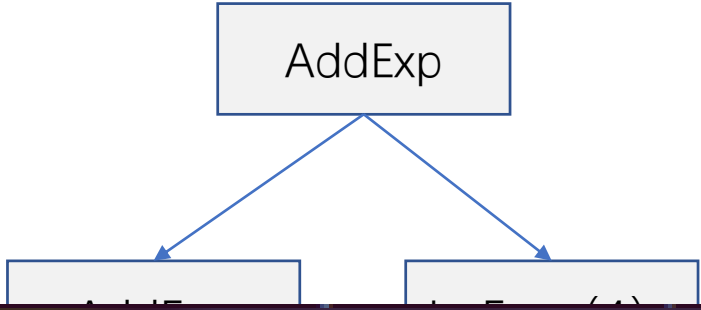
设计模式-访问者模式



访问者模式	示例
Visitor	Visitor
ConcreteVisitor	Visitor
Element	Exp
ConcreteElement	AddExp、IntExp



设计模式-访问者模式



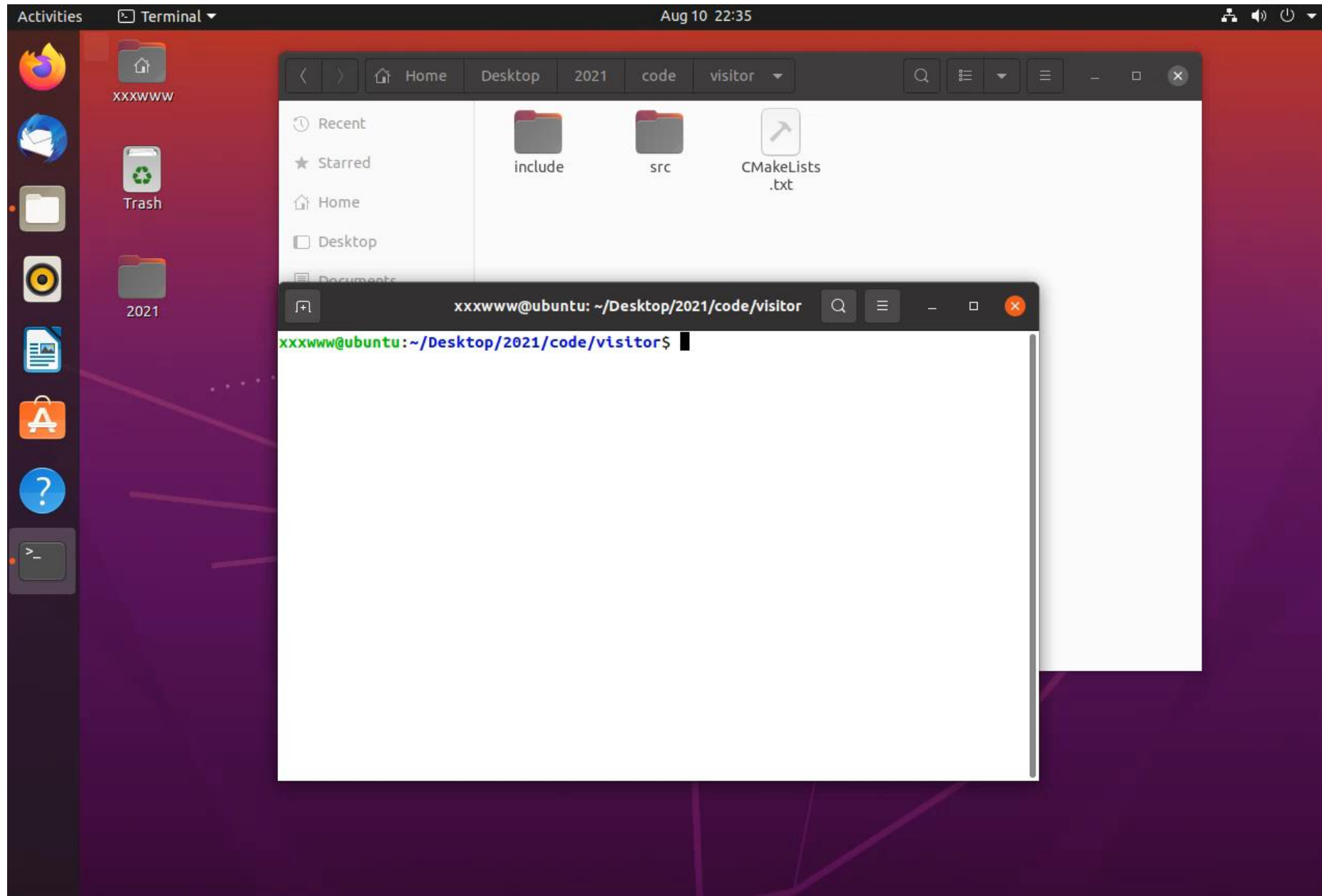
访问者模式	示例
Visitor	Visitor
ConcreteVisitor	Visitor
Element	Exp
ConcreteElement	AddExp、IntExp

```
/visitor/build$ make
Scanning dependencies of target example2
[ 33%] Building CXX object CMakeFiles/example2.dir/src/main.cpp.o
[ 66%] Building CXX object CMakeFiles/example2.dir/src/Visitor.cpp.o
[100%] Linking CXX executable example2
[100%] Built target example2

/visitor/build$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  example2  Makefile

/visitor/build$ ./example2
Result is 10
```

设计模式-访问者模式



目录



□ GCC/G++ 编译 C/C++ 程序

□ Makefile 基础

□ 汇编语言基础

□ 访问者模式

□ Git

- ❑ 将远端代码复制到本地

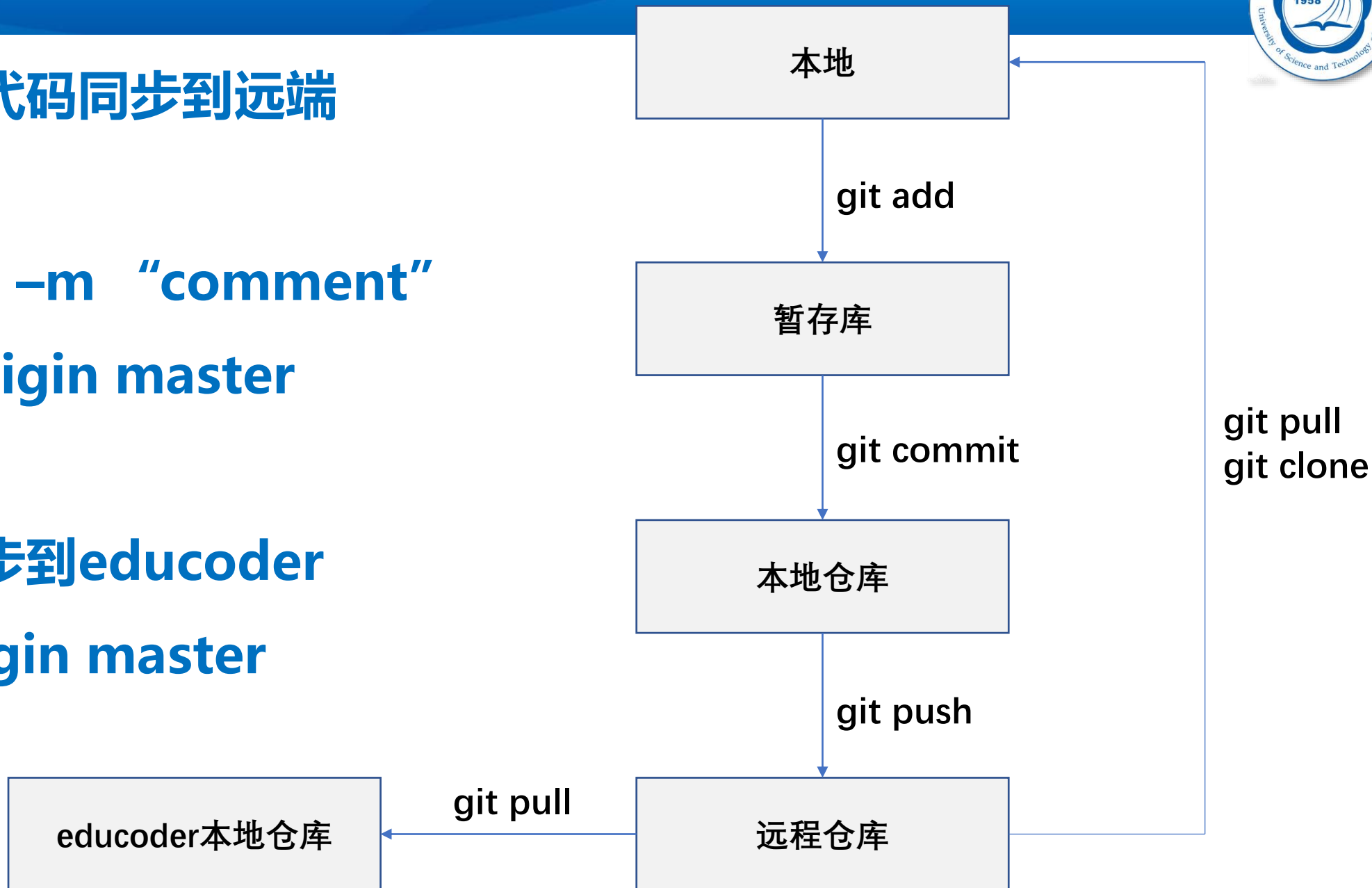
- ❑ `git clone https://XXXXX.git`

- ❑ 配置user信息

- ❑ `git config --global user.name "your name"`

- ❑ `git config --global user.email "your_email@domain.com"`

- ❑ 本地修改后代码同步到远端
- ❑ `git add .`
- ❑ `git commit -m "comment"`
- ❑ `git push origin master`
- ❑ 远端仓库同步到educoder
- ❑ `git pull origin master`





下期再见！

Thanks!