COMP130014 编译

第十四讲：栈展开

徐 辉

xuh@fudan.edu.cn

大纲

一、异常调试问题

二、栈展开

三、语言级异常处理

一、异常处理问题

未定义行为

- 未对程序可能的行为做任何约束，编译器可以任意实现
  - 有符号整数运算溢出
  - 空指针
  - 悬空指针
  - 内存越界
  - 数据竞争
- 程序员保证代码不触发未定义行为
- 未定义行为会引发异常，导致程序异常终止
  - 如何调试程序错误?
  - 如何自动捕获和处理异常?

不同于未声明行为

- 语言标准中未明确具体的实现方法
- 编译器选择具体的实现方法，生成有意义的程序

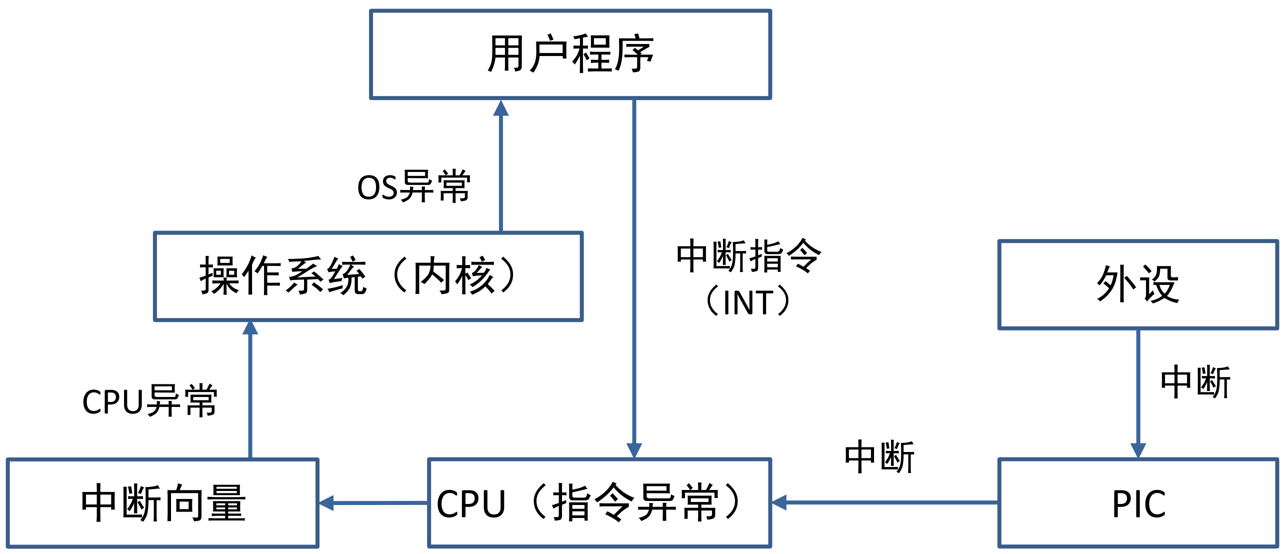
```
let a = f(x) + g(x);
let b = s(f(x), g(x));

let c = a + ++x++;
```

f(x)和g(x)的执行顺序？

x = ?, c = ?

异常来源

- CPU异常：CPU指令异常引发的中断（Interrupt）
- OS异常：OS抛出异常信号（signal）
- APP异常：用户在应用程序代码中自定义的异常



CPU异常

- CPU指令遇到除零、缺页等各种Fault
- 通过中断向量（interrupt vector）跳转到异常处理指令
  - 中断向量位于内存固定地址，记录不同异常对应的跳转地址
  - 以X86为例，用编号0x00-0x1F标记不同的CPU异常
    - 0x00 Division by zero
    - 0x01 Single-step interrupt (see trap flag)
    - 0x03 Breakpoint (INT 3)
    - 0x04 Overflow
    - 0x06 Invalid Opcode
    - 0x0B Segment not present
    - 0x0C Stack Segment Fault
    - 0x0D General Protection Fault
    - 0x0E Page Fault
    - ...

OS异常

- OS内核发给其它进程的IPC信号
- POSIX signals
  - SIGFPE：floating-point error，包括除零、溢出、下溢等。
  - SIGSEGV：segmentation fault，无效内存地址。
  - SIGBUS：bus error，如地址对齐问题
  - SIGILL：illegal instruction
  - SIGABRT：abort
  - SIGKILL：
  - …

获得函数调用栈：调试工具读取DWARF

```
void b(){ printf("%s\n", 0x1111); }
void a(){ b();}
int  main(){
    a();
    return 0;
}
```

```
#:./a.out
Segmentation fault: 11
#:.lldb a.out
(lldb) r
Process 37113 launched: '/Users/huixu/compiler/a.out' (arm64)
Process 37113 stopped
...
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1,
address=0x1110)
  * frame #0: 0x00000001902b9504 libsystem_platform.dylib`_platform_strlen + 4
    frame #1: 0x000000019011d770 libsystem_c.dylib`__vfprintf + 3580
    frame #2: 0x000000019012ca24 libsystem_c.dylib`vfprintf_l + 156
    frame #3: 0x0000000190147c3c libsystem_c.dylib`printf + 80
    frame #4: 0x0000000100003f4c a.out`b + 36
    frame #5: 0x0000000100003f64 a.out`a + 12
    frame #6: 0x0000000100003f88 a.out`main + 28
    frame #7: 0x000000018ff04274 dyld`start + 2840
```

应用程序异常

```
//C/C++代码
void b(int b) {
   cout << "Entering func b()..." << endl;
   if(b == 0) {throw "zero condition!";}
   cout << "Leaving func b()." << endl;
}

void a(int i) {
   cout << "Entering func a()..." << endl;
   b(i);
   cout << "Leaving func a()." << endl;
}

int main(int argc, char** argv) {
    int x = argv[1][0]-48;
    try {
       cout << "Entering block try..." << endl;
       a(x);
       cout << "Leaving block try." << endl;
    }catch (const char* msg) {
       cout << "Executing block catch." << endl;
    }
    cout << "Leaving func main()." << endl;
}
```

```
#:./a.out 1
Entering block try...
Entering func a()...
Entering func b()...
Leaving func b().
Leaving func a().
Leaving block try.
Leaving func main().
```

```
#:./a.out 0
Entering block try...
Entering func a()...
Entering func b()...
Executing block catch.
Leaving func main().
```

处理OS异常需要提前注册捕获

```
//C/C++代码
#include<iostream>
#include <signal.h>
using namespace std;

void handler(int signal) {
    throw "Div 0 is not allowed!!!";
}

int main(int argc, char** argv) {
    signal(SIGFPE, handler);
    int x = argv[1][0]-48;
    try{
        cout << "Entering block try..." << endl;
        x = 100/x;
        cout << "Leaving block try." << endl;
    }catch (const char* msg) {
        cout << msg << endl;
    }
   cout << "Leaving func main()." << endl;
}
```

不注册SIGFPE异常：

```
#:./a.out 0
Entering block try...
Floating point exception
(core dumped)
```

注册SIGFPE异常：

```
#:./a.out 0
Entering block try...
Div 0 is not allowed!!!
Leaving func main().
```

获得函数调用栈：注册异常并读取DWARF

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <execinfo.h>
#define BT_BUF_SIZE 100

void handler(int signal) {
    void *buffer[BT_BUF_SIZE];
    int nptrs = backtrace(buffer, BT_BUF_SIZE);
    printf("backtrace() returned %d addresses\n", nptrs);
    char **strings = backtrace_symbols(buffer, nptrs);
    for (int j = 0; j < nptrs; j++) printf("%s\n", strings[j]);
    free(strings);
    exit(EXIT_FAILURE);
}
void b(){ printf("%s\n", 0x1111); }
void a(){ b();}
int  main(){
    signal(SIGSEGV, handler);
    a();
    return 0;
}
```

异常处理需要处理的问题

- 指令跳转
  - 应该从哪个指令开始恢复程序运行？
  - 中断向量
- 寄存器恢复：
  - 栈基指针和栈顶指针应该指向哪里？
  - 其它寄存器内容应如何恢复？
- 资源回收：
  - 有堆内存需要释放？
  - 有哪些其它资源需要释放？

C标准库：setjmp/longjmp

```c
//C/C++代码
#include <stdio.h>
#include <setjmp.h>
static jmp_buf buf;
void second() {
    printf("enter second\n");
    longjmp(buf,1);
}
void first() {
    second();
    printf("exit first\n");
}
int main() {
    if (!setjmp(buf))
        first();
    else
        printf("exit main\n");
    return 0;
}
```

```
#:./a.out 0
enter second
exit main
```

- setjmp(env)：
  - 保存寄存器环境
  - 并设置为异常恢复点
  - 直接调用返回值为0
  - 通过longjmp调用返回值为value参数值
- longjmp(env,value)：
  - 跳转到异常恢复点
  - 还原所有callee-saved寄存器

思考

- 是否可以用setjmp/longjmp实现try-throw-catch?

二、栈展开

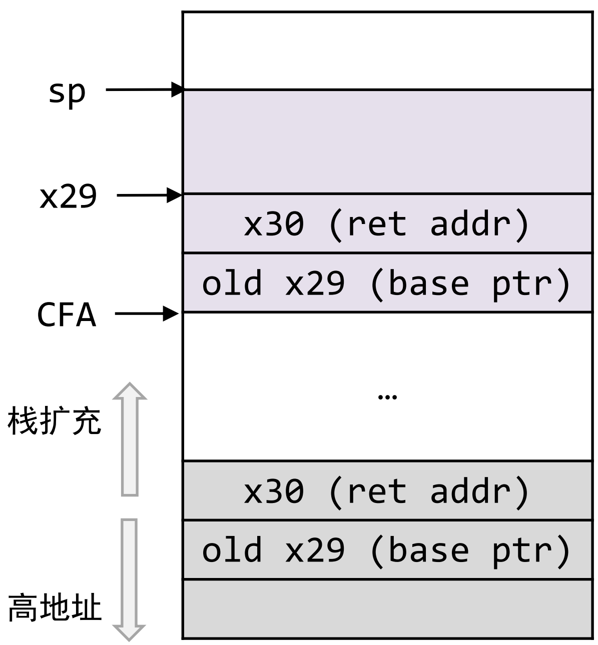
栈展开问题（Stack Unwinding）

- Callee-saved寄存器是保存在栈上的
- 程序返回上层函数时应还原寄存器状态
  - 正常返回 vs 异常退出

aarch64寄存器	调用规约	注释
X0-X7	参数1-8	
X0-X1	返回值	
X8	特殊用途：间接调用返回地址	
X9-X15	**临时寄存器**	**Caller-saved**
X16-X17	特殊用途：Intra-Procedure-Call	
X18	特殊用途：平台寄存器	
X19-X28	**普通寄存器**	**Callee-saved**
X29	栈帧基指针	
X30	返回地址	
SP	栈顶指针	Callee-saved

aarch64栈帧结构分析

```
fn fac(n:int) -> int {
   if(n == 0) {
      return 1;
   } else {
      ret n * fac(n-1);
   }
}
```



CFA：canonical frame address

```
_fac:
    sub sp, sp, 32
    stp x29, x30, [sp, 16]
    add x29, sp, 16
    .cfi_def_cfa x29, 16
    .cfi_offset x30, -8
    .cfi_offset x29, -16
    str w0, [sp, 8]
    ldr w8, [sp, 8]
    cbnz w8, LBB0_2
    b LBB0_1
LBB0_1:
    mov w8, 1
    str w8, [x29, -4]
    b LBB0_3
LBB0_2:
    ldr w8, [sp, 8]
    str w8, [sp, 4]
    ldr w8, [sp, 8]
    sub w0, w8, 1
    bl _fac
    ldr w8, [sp, 4]
    mul w8, w8, w0
    str w8, [x29, -4]
    b LBB0_3
LBB0_3:
    ldr w0, [x29, -4]
    ldp x29, x30, [sp, 16]
    add sp, sp, 32
    ret
```

编译时保存

- 将异常处理所需数据提前保存在程序文件中
  - 遵循DWARF程序调试格式
  - 不同于基于setjmp/longjmp的运行时方式
- 通过ABI异常处理标准定义异常处理方式
  - 根据异常位置确定恢复指令位置
  - 退栈、恢复callee-saved寄存器
- 无需在正常程序控制流中内联异常处理代码，开销低

http://itanium-cxx-abi.github.io/cxx-abi

如何在编译时记录栈信息？

- 主要目的：根据函数调用链层层回退
- 主要问题：指令异常时应如果恢复caller context？
  - 确定返回地址
  - 恢复所有callee-saved的寄存器：改变callee-saved寄存器的指令

以栈帧基地址CFA为记录基准

```
_fac:
    sub sp, sp, 32
    stp x29, x30, [sp, 16]
    add x29, sp, 16
    str w0, [sp, 8]
    ldr w8, [sp, 8]
    cbnz w8, LBB0_2
    b LBB0_1
LBB0_1:
    mov w8, 1
    str w8, [x29, -4]
    b LBB0_3
bl _facLBB0_2:
    ...
    str w8, [x29, -4]
    b LBB0_3
LBB0_3:
    ldr w0, [x29, -4]
    ldp x29, x30, [sp, 16]
    add sp, sp, 32
    ret
```

```
CFA = SP + 32
x30 = CFA – 8, x29 = CFA – 16
CFA = X29 + 16
```

```
.cfi_def_cfa x29, 16
.cfi_offset x30, -8
.cfi_offset x29, -16
```

可执行文件中的异常信息

```
#: otool -l ./a.out
Section
  sectname __unwind_info
   segname __TEXT
      addr 0x0000000100003fa8
      size 0x0000000000000058
    offset 16296
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x00000000
 reserved1 0
 reserved2 0
Load command 2
      cmd LC_SEGMENT_64
  cmdsize 152
  segname __DATA_CONST
   vmaddr 0x0000000100004000
   vmsize 0x0000000000004000
  fileoff 16384
 filesize 16384
  maxprot 0x00000003
 initprot 0x00000003
   nsects 1
    flags 0x10
```

Linux ELF文件可以使用pyreadelf工具查看

```
python3 pyelftools-master/scripts/readelf.py /bin/cat --debug-dump frames-interp
```

```
2690: endbr64
2694: push   %r15
2696: mov    %rsi,%rax
2699: push   %r14
269b: push   %r13
269d: push   %r12
269f: push   %rbp
26a0: push   %rbx
26a1: lea
0x4f94(%rip),%rbx
26a8: sub    $0x148,%rsp
26af: mov    %edi,0x2c(%rsp)
26b3: mov    (%rax),%rdi
…
27e7: sub    $0x8,%rsp
…
27fb: pushq  $0x0
…
2e96: pop    %rbx
2e97: pop    %rbp
2e98: pop    %r12
2e9a: pop    %r13
2e9c: pop    %r14
2e9e: pop    %r15
2ea0: retq
```

```
   LOC    CFA      rbx   rbp   r12   r13   r14   r15   ra
00002690 rsp+8     u     u     u     u     u     u     c-8
00002696 rsp+16    u     u     u     u     u     c-16  c-8
0000269b rsp+24    u     u     u     u     c-24  c-16  c-8
0000269d rsp+32    u     u     u     c-32  c-24  c-16  c-8
0000269f rsp+40    u     u     c-40  c-32  c-24  c-16  c-8
000026a0 rsp+48    u     c-48  c-40  c-32  c-24  c-16  c-8
000026a1 rsp+56    c-56  c-48  c-40  c-32  c-24  c-16  c-8
000026af rsp+384   c-56  c-48  c-40  c-32  c-24  c-16  c-8
000027eb rsp+392   c-56  c-48  c-40  c-32  c-24  c-16  c-8
000027fd rsp+400   c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002825 rsp+384   c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e96 rsp+56    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e97 rsp+48    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e98 rsp+40    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9a rsp+32    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9c rsp+24    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002e9e rsp+16    c-56  c-48  c-40  c-32  c-24  c-16  c-8
00002ea0 rsp+8     c-56  c-48  c-40  c-32  c-24  c-16  c-8
```

https://github.com/eliben/pyelftools

运行时和编译时方式栈帧还原方法对比

- 运行时：基于setjmp/longjmp的方式
  - 缺点：动态保存寄存器信息会带来一定的运行开销
  - 优点：栈帧还原速度快
- 编译时：基于DWARF的方式
  - 优点：无运行时开销
  - 缺点：增加ELF文件体积、栈帧还原速度慢

三、语言级异常处理

基本概念

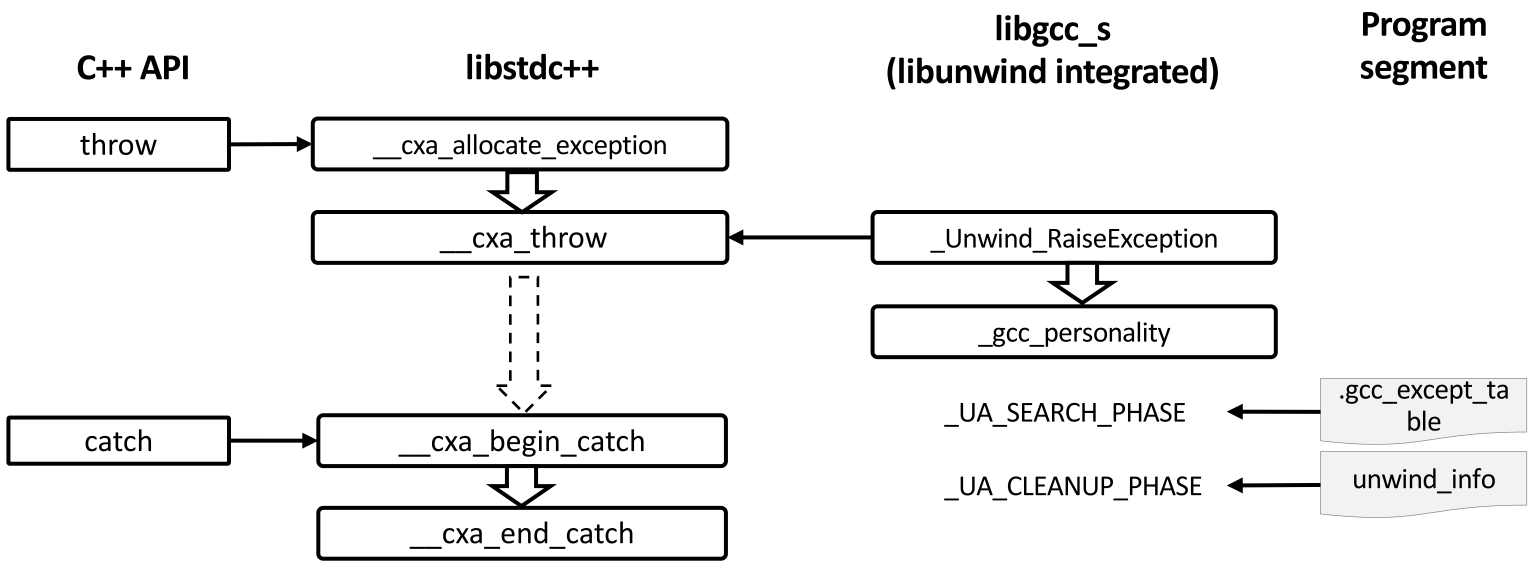
- Landing Pad：用于捕获异常和释放资源的用户代码
- Personality routine：实现landing pad的搜索和跳转
  - 由于不同的编程语言存在设计理念差异，ABI应支持个性化处理方法
  - 如c++的__gxx_personality_v0函数用于接收异常，包括异常类型、值、和指向gcc_exception_table的引用

应如何记录下列程序的异常登录点？

```
void handler(int signal) {
    throw "SIGFPE Received!!!";
}
void b(int b) {
    double y = b%b;
    if(b < 0) {throw -1;}
}
void a(int i) {
    try {
        b(i);
    } catch (const int msg) {          //catch 1
        cout << "Unsupported value:" << msg << endl;
    } catch (const char* msg) {        //catch 2
        cout << "Land in a: " << msg << endl;
        throw "a cannot handle!!!";
    }
}
int main(int argc, char** argv) {
    signal(SIGFPE, handler);
    int x;
    scanf("%d", &x);
    try {
        a(x);
    } catch (const char* msg) {        //catch 3
        cout << "Land in main: " << msg << endl;
    }
}
```

- 如果try b()失败：
  - landing pad为catch 1或catch 2
  - 如果catch1和catch2不匹配，则尝试catch 3
- 如果try a(x)失败：
  - landing pad为catch 3

C++异常处理流程



- throw
  - 调用__cxa_allocate_exception分配空间保存异常对象
  - __cxa_throw设置异常对象字段内容并跳转到_Unwind_RaiseException
  - _Unwind_RaiseException
    - 通过personality routines搜索匹配的try-catch
    - 进入cleanup阶段，进行栈展开，然后跳转到对应的catch块
- catch
  - 调用__cxa_begin_catch，执行catch code
  - __cxa_end_catch销毁exception object

抛出异常

```
void handler(int signal) {
    throw "SIGFPE Received!!!";
}
```

```
void b(int b) {
    double y = b%b;
    if(b < 0) {throw -1;}
}
```

```
stp x29, x30, [sp, #-16]!
mov x29, sp
mov w0, 4
bl ___cxa_allocate_exception
mov w8, #-1
str w8, [x0]
adrp x1, __ZTIi@GOTPAGE
ldr x1, [x1, __ZTIi@GOTPAGEOFF]
mov x2, 0
bl ___cxa_throw
```

```
stp     x29, x30, [sp, -16]!
mov     x29, sp
mov     w0, 4
bl      ___cxa_allocate_exception
mov     w8, -1
str     w8, [x0]
adrp    x1, __ZTIi@GOTPAGE
ldr     x1, [x1, __ZTIi@GOTPAGEOFF]
mov     x2, 0
bl      ___cxa_throw
```

捕获处理异常

```
void a(int i) {
    try{
        b(i);
    } catch (const int msg) {     //catch 1
        cout << "Unsupported value:" << msg << endl;
    } catch (const char* msg) {  //catch 2
        cout << "Land in a: " << msg << endl;
        throw "a cannot handle!!!";
    }
}
```

```
Lfunc_begin0:
    ...
Ltmp0:
    bl __Z1bi
Ltmp1:
    b       LBB2_1
LBB2_1:
    b       LBB2_8
LBB2_2:
Ltmp2:
    ...
    subs    w8, w8, 2
    b.ne    LBB2_9
    b       LBB2_4
LBB2_4:
    ldur    x0, [x29, -16]
    bl      ___cxa_begin_catch
    ...
```

GCC_except_table

```
Lcst_begin0:
        .uleb128 Ltmp0-Lfunc_begin0             ; >> Call Site 1 <<
        .uleb128 Ltmp1-Ltmp0                    ;   Call between Ltmp0 and Ltmp1
        .uleb128 Ltmp2-Lfunc_begin0             ;     jumps to Ltmp2
        .byte   3                               ;   On action: 2
        ...
```

Linux操作系统上的实验

```
# clang++ except_table.cpp
# ./a.out
0
Land in a: SIGFPE Received!!!
Land in main: a cannot handle!!!
# ./a.out
-1
Unsupported value:-1
# strip -R ".eh_frame" a.out
# ./a.out
0
terminate called after throwing an instance of 'char const*'
Aborted (core dumped)
# ./a.out
-1
terminate called after throwing an instance of 'int'
Aborted (core dumped)
# clang++ except_table.cpp
# strip -R ".gcc_except_table" a.out
# ./a.out
0
terminate called after throwing an instance of 'char const*'
Aborted (core dumped)
# ./a.out
-1
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

栈展开过程中需要回收的资源

- cleanup标注的对象
- 栈上的对象：
  - 栈展开时调用析构函数
- 堆上的对象：
  - 由于不确定是否存在其它引用，默认不应析构
  - unique_ptr可以析构
  - Rust所有权模型编译时静态分析是否能析构

分析：这段代码会输出什么？

```
void cleanA(char **buffer){ cout << "cleanup for A" << endl; free(*buffer); }
void cleanB(char **buffer){ cout << "cleanup for B" << endl; free(*buffer); }
class C {
    public:
        ~C(){ cout << "Destruct Obj C..." << endl; }
};
class B {
public:
    void doB(int b) {
        char *buf __attribute__ ((__cleanup__(cleanB))) = (char *) malloc(10);
        if(b == 0) { throw "error";}
        if(b < 0) { throw -1;}
    }
    ~B(){ cout << "Destruct B..."<< endl; }
};
class A {
private:
    B b;
public:
    void doA(int i) {
        char *buf __attribute__ ((__cleanup__(cleanA))) = (char *) malloc(10);
        C c;
        try{ b.doB(i); } catch (const int msg) {
            cout << "Land in doA: " << msg << endl;
        }
    }
    virtual ~A(){ cout << "Destruct A..."<< endl; }
};
int main(int argc, char** argv) {
    int x;
    scanf("%d", &x);
    A a;
    try{ a.doA(x); } catch (const char* msg) {
        cout << "Land in main: " << msg << endl;
    }
    cout << "Exit main" << endl;
}
```

```
#./a.out
0
cleanup for B
Destruct Obj C...
cleanup for A
Land in main: error
Exit main
Destruct A...
Destruct B...
#./a.out
-1
cleanup for B
Land in doA: -1
Destruct Obj C...
cleanup for A
Exit main
Destruct A...
Destruct B...
```

如果把a或c改为指针呢？ A* a = new A;

```
#./a.out
0
cleanup for B
cleanup for A
Land in main: error
Exit main
#./a.out
-1
cleanup for B
Land in doA: -1
cleanup for A
Exit main
```