

COMP130014.02 编译

第十五讲：异常处理

徐 辉

xuh@fudan.edu.cn



大纲

- ❖ 一、异常处理问题
- ❖ 二、栈展开
- ❖ 三、语言级异常处理

一、异常处理问题

未声明行为 (Unspecified Behavior)

- 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?

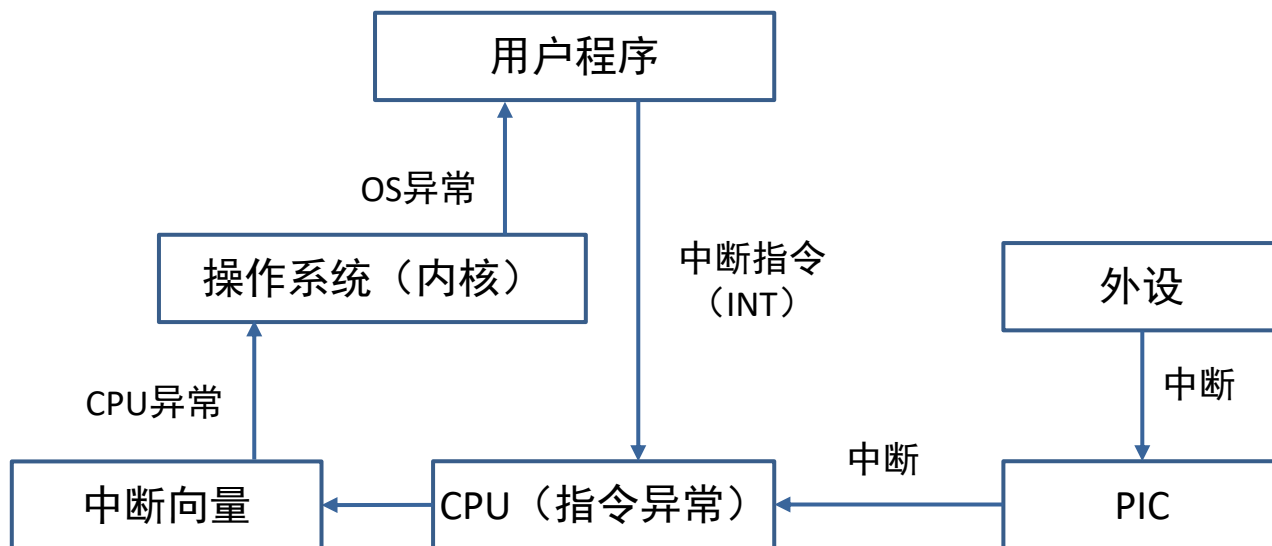
$x = ?$, $c = ?$

未定义行为（Undefined Behavior）

- 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

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:
 - ...

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().
```

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++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;
}
```

- 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

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

问题

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

二、栈展开

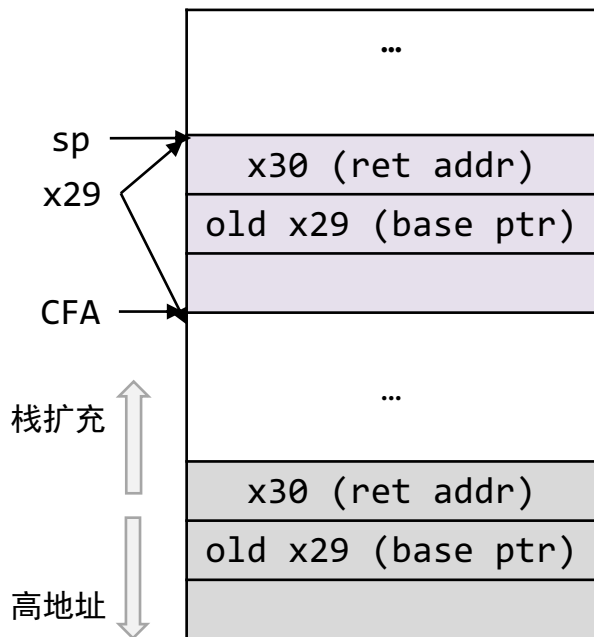
栈展开问题（Stack Unwinding）

- 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	Callee-saved
X30	返回地址	Caller-saved
SP	栈顶指针	Callee-saved

aarch64栈帧结构分析

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



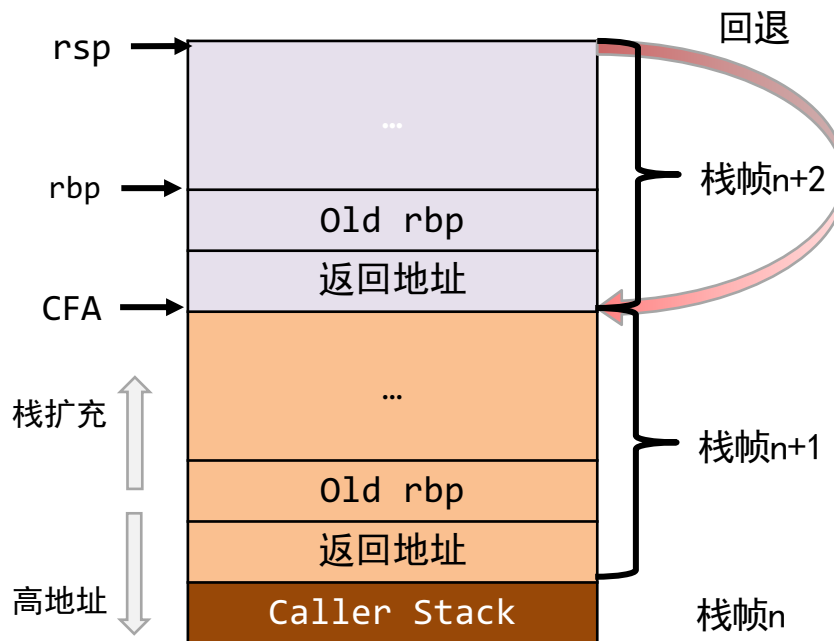
CFA: canonical frame address

```
stp      x29, x30, [sp, -32]!  
.cfi_def_cfa_offset 32  
.cfi_offset 29, -32  
.cfi_offset 30, -24  
mov      x29, sp  
str      w0, [sp, 28]  
ldr      w0, [sp, 28]  
cmp      w0, 0  
bne      .L2  
mov      w0, 1  
b        .L3  
  
.L2:  
ldr      w0, [sp, 28]  
sub      w0, w0, #1  
bl       fac  
mov      w1, w0  
ldr      w0, [sp, 28]  
mul      w0, w1, w0  
  
.L3:  
ldp      x29, x30, [sp], 32  
.cfi_restore 30  
.cfi_restore 29  
.cfi_def_cfa_offset 0  
ret
```


X86_64栈帧结构分析

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

```
0x401130: push    %rbp  
0x401131: mov     %rsp,%rbp  
0x401134: sub     $0x10,%rsp  
0x401138: mov     %edi,-0x8(%rbp)  
0x40113b: cmpl    $0x0,-0x8(%rbp)  
0x40113f: jne     0x401151  
0x401145: movl    $0x1,-0x4(%rbp)  
0x40114c: jmpq    0x40116d  
0x401151: mov     -0x8(%rbp),%eax  
0x401154: mov     -0x8(%rbp),%ecx  
0x401157: sub     $0x1,%ecx  
0x40115a: mov     %ecx,%edi  
0x40115c: mov     %eax,-0xc(%rbp)  
0x40115f: callq   0x401130  
0x401164: mov     -0xc(%rbp),%ecx  
0x401167: imul    %eax,%ecx  
0x40116a: mov     %ecx,-0x4(%rbp)  
0x40116d: mov     -0x4(%rbp),%eax  
0x401170: add     $0x10,%rsp  
0x401174: pop     %rbp  
0x401175: retq
```



- callee-saved寄存器用完必须还原
 - rbx/rbp/rsp/r12/r13/r14/r15

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

如何在编译时记录栈信息？

- 主要思路：根据函数调用链层层回退
- 主要问题：指令异常时应如何恢复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

```
push    %rbp      → 返回地址 = CFA-8;  
                      rbp = CFA - 16,  
                      rsp = CFA - 16  
  
mov     %rsp,%rbp  
sub     $0x10,%rsp →  rsp = CFA - 32;  
mov     %edi,-0x8(%rbp)  
cmpl    $0x0,-0x8(%rbp)  
jne     0x401151  
movl    $0x1,-0x4(%rbp)  
jmpq    0x40116d  
mov     -0x8(%rbp),%eax  
mov     -0x8(%rbp),%ecx  
sub     $0x1,%ecx  
mov     %ecx,%edi  
mov     %eax,-0xc(%rbp)  
callq   0x401130  
mov     -0xc(%rbp),%ecx  
imul    %eax,%ecx  
mov     %ecx,-0x4(%rbp)  
mov     -0x4(%rbp),%eax  
add     $0x10,%rsp →  rsp = CFA - 16  
pop     %rbp      →  rsp = CFA - 8  
retq    →  rsp = CFA
```

返回地址 = CFA-8;
rbp = CFA - 16
CFA = rsp + 16
CFA = rbp + 16
~~CFA = rsp + 32~~

~~CFA = rsp + 16~~
CFA = rsp + 8
CFA = rsp

Section Headers

```
#: readelf -S a.out
```

```
There are 31 section headers, starting at offset 0x1a30:
```

```
Section Headers:
```

[Nr]	Name	Type	Address	Offset	Size	EntSize	Flags	Link	Info	Align
[0]		NULL	..00000000	0000	0000	0000		0	0	0
[1]	.interp	PROGBITS	..004002a8	02a8	001c	0000	A	0	0	1
[2]	.note.gnu.build-id	NOTE	..004002c4	02c4	0024	0000	A	0	0	4
[3]	.note.ABI-tag	NOTE	..004002e8	02e8	0020	0000	A	0	0	4
[4]	.gnu.hash	GNU_HASH	..00400308	0308	001c	0000	A	5	0	8
[5]	.dynsym	DYNSYM	..00400328	0328	0060	0018	A	6	1	8
[6]	.dynstr	STRTAB	..00400388	0388	003f	0000	A	0	0	1
[7]	.gnu.version	VERSYM	..004003c8	03c8	0008	0002	A	5	0	2
[8]	.gnu.version_r	VERNEED	..004003d0	03d0	0020	0000	A	6	1	8
[9]	.rela.dyn	RELA	..004003f0	03f0	0030	0018	A	5	0	8
[10]	.rela.plt	RELA	..00400420	0420	0018	0018	AI	5	22	8
[11]	.init	PROGBITS	..00401000	1000	001b	0000	AX	0	0	4
[12]	.plt	PROGBITS	..00401020	1020	0020	0010	AX	0	0	16
[13]	.text	PROGBITS	..00401040	1040	01c5	0000	AX	0	0	16
[14]	.fini	PROGBITS	..00401208	1208	000d	0000	AX	0	0	4
[15]	.rodata	PROGBITS	..00402000	2000	0010	0000	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	..00402010	2010	003c	0000	A	0	0	4
[17]	.eh_frame	PROGBITS	..00402050	2050	00e8	0000	A	0	0	8
[18]	.init_array	INIT_ARRAY	..00403e10	2e10	0008	0008	WA	0	0	8
[19]	.fini_array	FINI_ARRAY	..00403e18	2e18	0008	0008	WA	0	0	8
[20]	.dynamic	DYNAMIC	..00403e20	2e20	01d0	0010	WA	6	0	8
[21]	.got	PROGBITS	..00403ff0	2ff0	0010	0008	WA	0	0	8
[22]	.got.plt	PROGBITS	..00404000	3000	0020	0008	WA	0	0	8
[23]	.data	PROGBITS	..00404020	3020	0010	0000	WA	0	0	8
[24]	.bss	NOBITS	..00404030	3030	0008	0000	WA	0	0	1
[25]	.comment	PROGBITS	..00000000	3030	0049	0001	MS	0	0	1
[26]	.symtab	SYMTAB	..00000000	3080	05b8	0018		27	43	8
[27]	.strtab	STRTAB	..00000000	3638	01ca	0000		0	0	1
[28]	.shstrtab	STRTAB	..00000000	3802	0103	0000				

动态链接符号

函数链接代码
代码段

只读数据

异常处理信息

全局偏移地址
(动态链接)

全局变量 (初始化)

全局变量 (未初始化)

使用pyreadelf工具查看

```
python3 pyelftools-master/scripts/readelf.py --debug-dump frames-interp a.out
```

```
0x401130: push    %rbp
0x401131: mov     %rsp,%rbp
0x401134: sub     $0x10,%rsp
0x401138: mov     %edi,-0x8(%rbp)
0x40113b: cmpl    $0x0,-0x8(%rbp)
0x40113f: jne     0x401151
0x401145: movl    $0x1,-0x4(%rbp)
0x40114c: jmpq    0x40116d
0x401151: mov     -0x8(%rbp),%eax
0x401154: mov     -0x8(%rbp),%ecx
0x401157: sub     $0x1,%ecx
0x40115a: mov     %ecx,%edi
0x40115c: mov     %eax,-0xc(%rbp)
0x40115f: callq   0x401130
0x401164: mov     -0xc(%rbp),%ecx
0x401167: imul    %eax,%ecx
0x40116a: mov     %ecx,-0x4(%rbp)
0x40116d: mov     -0x4(%rbp),%eax
0x401170: add     $0x10,%rsp
0x401174: pop     %rbp
0x401175: retq
```

LOC	CFA	rbp	ra
401130	rsp+8	u	c-8
401131	rsp+16	c-16	c-8
401134	rbp+16	c-16	c-8
401175	rsp+8	c-16	c-8

CFA是相对的，可根据运行时rsp计算

更多例子

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

2690: endbr64									
2694: push %r15	LOC	CFA	rbx	rbp	r12	r13	r14	r15	ra
2696: mov %rsi,%rax	00002690	rsp+8	u	u	u	u	u	u	c-8
2699: push %r14	00002696	rsp+16	u	u	u	u	u	c-16	c-8
269b: push %r13	0000269b	rsp+24	u	u	u	u	c-24	c-16	c-8
269d: push %r12	0000269d	rsp+32	u	u	u	c-32	c-24	c-16	c-8
269f: push %rbp	0000269f	rsp+40	u	u	c-40	c-32	c-24	c-16	c-8
26a0: push %rbx	000026a0	rsp+48	u	c-48	c-40	c-32	c-24	c-16	c-8
26a1: lea	000026a1	rsp+56	c-56	c-48	c-40	c-32	c-24	c-16	c-8
0x4f94(%rip),%rbx	000026af	rsp+384	c-56	c-48	c-40	c-32	c-24	c-16	c-8
26a8: sub \$0x148,%rsp	000027eb	rsp+392	c-56	c-48	c-40	c-32	c-24	c-16	c-8
26af: mov %edi,0x2c(%rsp)	000027fd	rsp+400	c-56	c-48	c-40	c-32	c-24	c-16	c-8
26b3: mov (%rax),%rdi	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
27e7: sub \$0x8,%rsp	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
27fb: pushq \$0x0	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
2e96: pop %rbx	00002e9e	rsp+16	c-56	c-48	c-40	c-32	c-24	c-16	c-8
2e97: pop %rbp	00002ea0	rsp+8	c-56	c-48	c-40	c-32	c-24	c-16	c-8
2e98: pop %r12									
2e9a: pop %r13									
2e9c: pop %r14									
2e9e: pop %r15									
2ea0: retq									

练习

cab0: endbr64					
cab4: push %r13	LOC	CFA	rbp	r12	r13 ra
cab6: mov %rsi, %r13	cab0	rsp+8	u	u	u c-8
cab9: mov \$0x2e, %esi	cab6	rsp+16	u	u	c-16 c-8
cabe: push %r12	cac0				
cac0: push %rbp	cac1				
cac1: mov (%rdi), r12	cb03				
cac4: mov %r12, %rdi	cb05				
cac7: call 4960	cb07				
cacc: mov 0x0(%r13), %r13	cb10				
cad0: mov \$0x2e, %esi	cb1d				
cad5: mov %rax, %rbp	cb25				
cad8: mov %r13, %rdi	cb27				
cadb: call 4960					
cae0: test %rax, %rax					
cae3: jz cb10					
cae5: mov %rax, %rsi					
cae8: test %rbp, %rbp					
caeb: lea 0xcd0c(%rip), %rax					
caf2: cmovz %rax, %rbp					
caf6: mov %rbp, %rdi					
caf9: call 4a80					
cafe: test %eax, %eax					
cb00: jz cb1c					
cb02: pop %rbp					
cb03: pop %r12					
cb05: pop %r13					
cb07: ret					
cb10: lea 0xcce7(%rip), %rsi					
cb17: test %rbp, %rbp					
cb1a: jnz caf6					
cb1c: pop %rbp					
cb1d: mov %r13, %rsi					
cb20: mov %r12, %rdi					
cb23: pop %r12					
cb25: pop %r13					
cb27: jmp 4a80					

基于DWARF获得函数调用栈

- Call stack是很多异常恢复的关键

```
void handler(int signal) {  
    void *buffer[BT_BUF_SIZE];  
    int npters = backtrace(buffer, BT_BUF_SIZE);  
    printf("backtrace() returned %d addresses\n", npters);  
    char **strings = backtrace_symbols(buffer, npters);  
    for (int j = 0; j < npters; 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;  
}
```

```
backtrace() returned 10 addresses  
./a.out(handler+0x22) [0x4011b2]  
/lib/x86_64-linux-gnu/libc.so.6(+0x46210) [0x7fbe5d2f9210]  
/lib/x86_64-linux-gnu/libc.so.6(+0x18b4e5) [0x7fbe5d43e4e5]  
/lib/x86_64-linux-gnu/libc.so.6(+0x7be95) [0x7fbe5d32ee95]  
/lib/x86_64-linux-gnu/libc.so.6(_IO_printf+0xaf) [0x7fbe5d317ebf]  
./a.out(b+0x1a) [0x4012aa]  
./a.out(a+0x9) [0x4012b9]  
./a.out(main+0x2c) [0x4012ec]  
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf3) [0x7fbe5d2da0b3]
```

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

抛出异常

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

```
pushq    %rbp  
movq     %rsp, %rbp  
subq     $16, %rsp  
movl     %edi, -4(%rbp)  
movl     $8, %edi  
callq    __cxa_allocate_exception  
movabsq  $_ZTIPLKc, %rcx  
xorl     %edx, %edx  
movabsq  $.L.str, %rsi  
movq     %rsi, (%rax)  
movq     %rax, %rdi  
movq     %rcx, %rsi  
callq    __cxa_throw
```

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

```
# %bb.0:pushq    %rbp  
        movq     %rsp, %rbp  
        subq     $16, %rsp  
        movl     %edi, -4(%rbp)  
        movl     -4(%rbp), %eax  
        cltd  
        idivl    -4(%rbp)  
        cvtsi2sd          %edx, %xmm0  
        movsd    %xmm0, -16(%rbp)  
        cmpl     $0, -4(%rbp)  
        jge     .LBB2_2  
# %bb.1:movl     $4, %edi  
        callq    __cxa_allocate_exception  
        movabsq  $_ZTIi, %rcx  
        xorl     %edx, %edx  
        movl     $-1, (%rax)  
        movq     %rax, %rdi  
        movq     %rcx, %rsi  
        callq    __cxa_throw  
.LBB2_2:addq     $16, %rsp  
        popq     %rbp  
        retq
```

GCC Except Table: main()函数

```

.Lfunc_begin1:
# %bb.0:
    pushq    %rbp
    movq     %rsp, %rbp
    subq     $64, %rsp
    movl     $0, -4(%rbp)
    movl     %edi, -8(%rbp)
    movq     %rsi, -16(%rbp)
    movl     $_Z7handleri, %esi
    movl     $8, %edi
    callq    signal
    movl     $.L.str.2, %edi
    xorl     %ecx, %ecx
    leaq     -20(%rbp), %rsi
    movq     %rax, -56(%rbp)
    movb     %cl, %al
    callq    __isoc99_scanf
    movl     -20(%rbp), %edi
.Ltmp10: movl     %eax, -60(%rbp)
    callq    _Z1ai
.Ltmp11: jmp     .LBB4_1
.LBB4_1: jmp     .LBB4_5
.LBB4_2:
.Ltmp12: movq     %rax, -32(%rbp)
    movl     %edx, -36(%rbp)
# %bb.3: movl     -36(%rbp), %eax
# %bb.4: movq     -32(%rbp), %rdi
    callq    __cxa_begin_catch
    movq     %rax, -48(%rbp)
    callq    __cxa_end_catch
.LBB4_5: movl     -4(%rbp), %eax
    addq     $64, %rsp
    popq     %rbp
    retq
.Lfunc_end4:

```

Call Site 1

Call Site 2

Call Site 3

```

GCC_except_table4:
.Lexception1:
    .byte    255                # @LPStart Encoding = omit
    .byte    3                  # @TType Encoding = udata4
    .uleb128 .Lttbase1-.Lttbaseref1
.Lttbaseref1:
    .byte    1                  # Call site Encoding = uleb128
    .uleb128 .Lcst_end1-.Lcst_begin1
.Lcst_begin1:
    .uleb128 .Lfunc_begin1-.Lfunc_begin1 # >> Call Site 1 <<
    .uleb128 .Ltmp10-.Lfunc_begin1      # Call between .Lfunc_begin1 and .Ltmp10
    .byte    0                          # has no landing pad
    .byte    0                          # On action: cleanup
    .uleb128 .Ltmp10-.Lfunc_begin1      # >> Call Site 2 <<
    .uleb128 .Ltmp11-.Ltmp10             # Call between .Ltmp10 and .Ltmp11
    .uleb128 .Ltmp12-.Lfunc_begin1      # jumps to .Ltmp12
    .byte    1                          # On action: 1
    .uleb128 .Ltmp11-.Lfunc_begin1      # >> Call Site 3 <<
    .uleb128 .Lfunc_end4-.Ltmp11         # Call between .Ltmp11 and .Lfunc_end4
    .byte    0                          # has no landing pad
    .byte    0                          # On action: cleanup
.Lcst_end1:
    .byte    1                      # >> Action Record 1 <<
                                         # Catch TypeInfo 1
    .byte    0                      # No further actions
    .p2align    2
                                         # >> Catch TypeInfos <<
    .long      _ZTIPLKc              # TypeInfo 1

```

GCC Except Table: a()函数

```

# %bb.0:pushq   %rbp
          movq   %rsp, %rbp
          subq   $48, %rsp
          movl   %edi, -4(%rbp)
          movl   -4(%rbp), %edi
Call Site 1 {
.Ltmp0: callq   _Z1bi
.Ltmp1: jmp     .LBB3_1
.LBB3_1: jmp     .LBB3_5
.LBB3_2:
.Ltmp2: movq   %rax, -16(%rbp)
          movl   %edx, -20(%rbp)
# %bb.3:movl   -20(%rbp), %eax
          movl   $2, %ecx
          cmpl   %ecx, %eax
          movl   %eax, -40(%rbp)
          jne    .LBB3_6
# %bb.4:movq   -16(%rbp), %rdi
          callq  __cxa_begin_catch
          movl   (%rax), %ecx
          movl   %ecx, -36(%rbp)
          callq  __cxa_end_catch
.LBB3_5: addq   $48, %rsp
          popq   %rbp
          retq
.LBB3_6: movl   $1, %eax
          movl   -40(%rbp), %ecx
          cmpl   %eax, %ecx
          jne    .LBB3_9
# %bb.7:movq   -16(%rbp), %rdi
          callq  __cxa_begin_catch
          movq   %rax, -32(%rbp)
          movl   $8, %edi
          callq  __cxa_allocate_exception
          movq   $.L.str.1, (%rax)
Call Site 3 {
.Ltmp3: movl   $_ZTIIPKc, %esi
          xorl   %ecx, %ecx
          movl   %ecx, %edx
          movq   %rax, %rdi
          callq  __cxa_throw
.Ltmp4: jmp     .LBB3_10
.LBB3_8:
.Ltmp5: movq   %rax, -16(%rbp)
          movl   %edx, -20(%rbp)
          callq  __cxa_end_catch
.LBB3_9: movq   -16(%rbp), %rdi
          callq  _Unwind_Resume

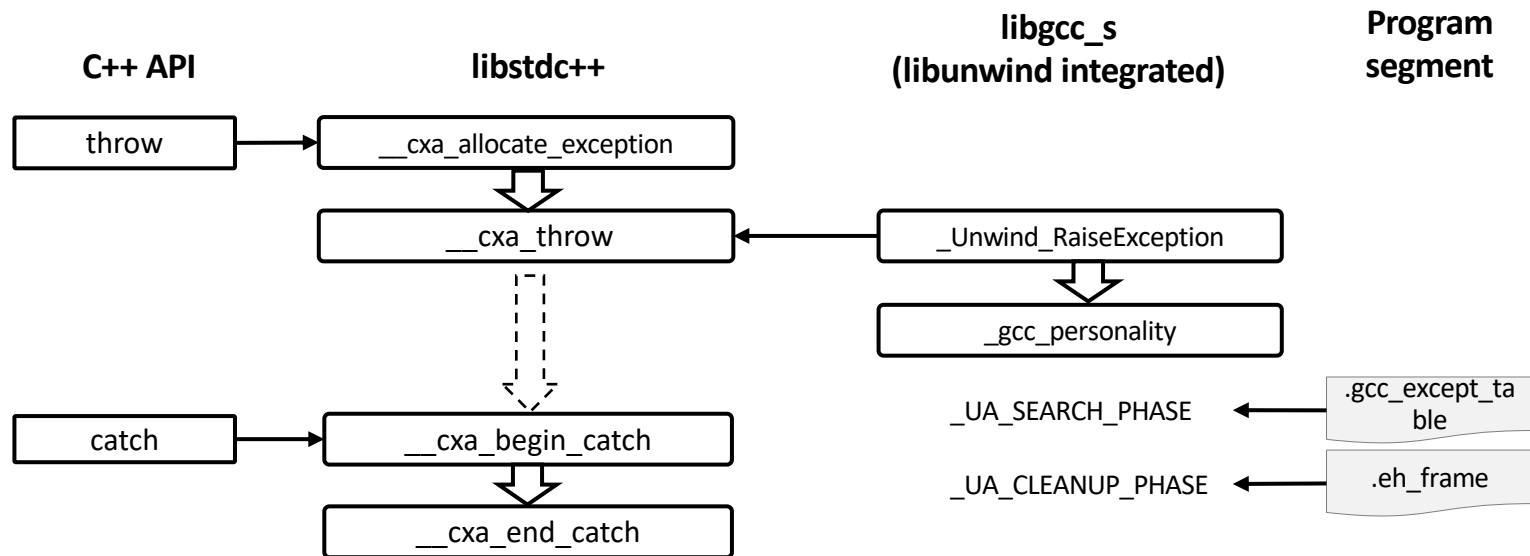
```

```

GCC_except_table3:
.Lexception0:
        .byte   255                # @LPStart Encoding = omit
        .byte   3                  # @TType Encoding = udata4
        .uleb128 .Lttbase0-.Lttbaseref0
.Lttbaseref0:
        .byte   1                  # Call site Encoding = uleb128
        .uleb128 .Lcst_end0-.Lcst_begin0
.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
        .uleb128 .Ltmp1-.Lfunc_begin0 # >> Call Site 2 <<
        .uleb128 .Ltmp3-.Ltmp1        # Call between .Ltmp1 and .Ltmp3
        .byte   0                      # has no landing pad
        .byte   0                      # On action: cleanup
        .uleb128 .Ltmp3-.Lfunc_begin0 # >> Call Site 3 <<
        .uleb128 .Ltmp4-.Ltmp3        # Call between .Ltmp3 and .Ltmp4
        .uleb128 .Ltmp5-.Lfunc_begin0 # jumps to .Ltmp5
        .byte   0                      # On action: cleanup
        .uleb128 .Ltmp4-.Lfunc_begin0 # >> Call Site 4 <<
        .uleb128 .Lfunc_end3-.Ltmp4   # Call between .Ltmp4 and .Lfunc_end3
        .byte   0                      # has no landing pad
        .byte   0                      # On action: cleanup
.Lcst_end0:
        .byte   1                    # >> Action Record 1 <<
        .byte   0                    # Catch TypeInfo 1
        .byte   2                    # No further actions
        .byte   2                    # >> Action Record 2 <<
        .byte   125                  # Catch TypeInfo 2
        .p2align 2                   # Continue to action 1
        .long    _ZTIi               # >> Catch TypeInfos <<
        .long    _ZTIIPKc            # TypeInfo 2

```


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

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：
 - stack unwinding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
```