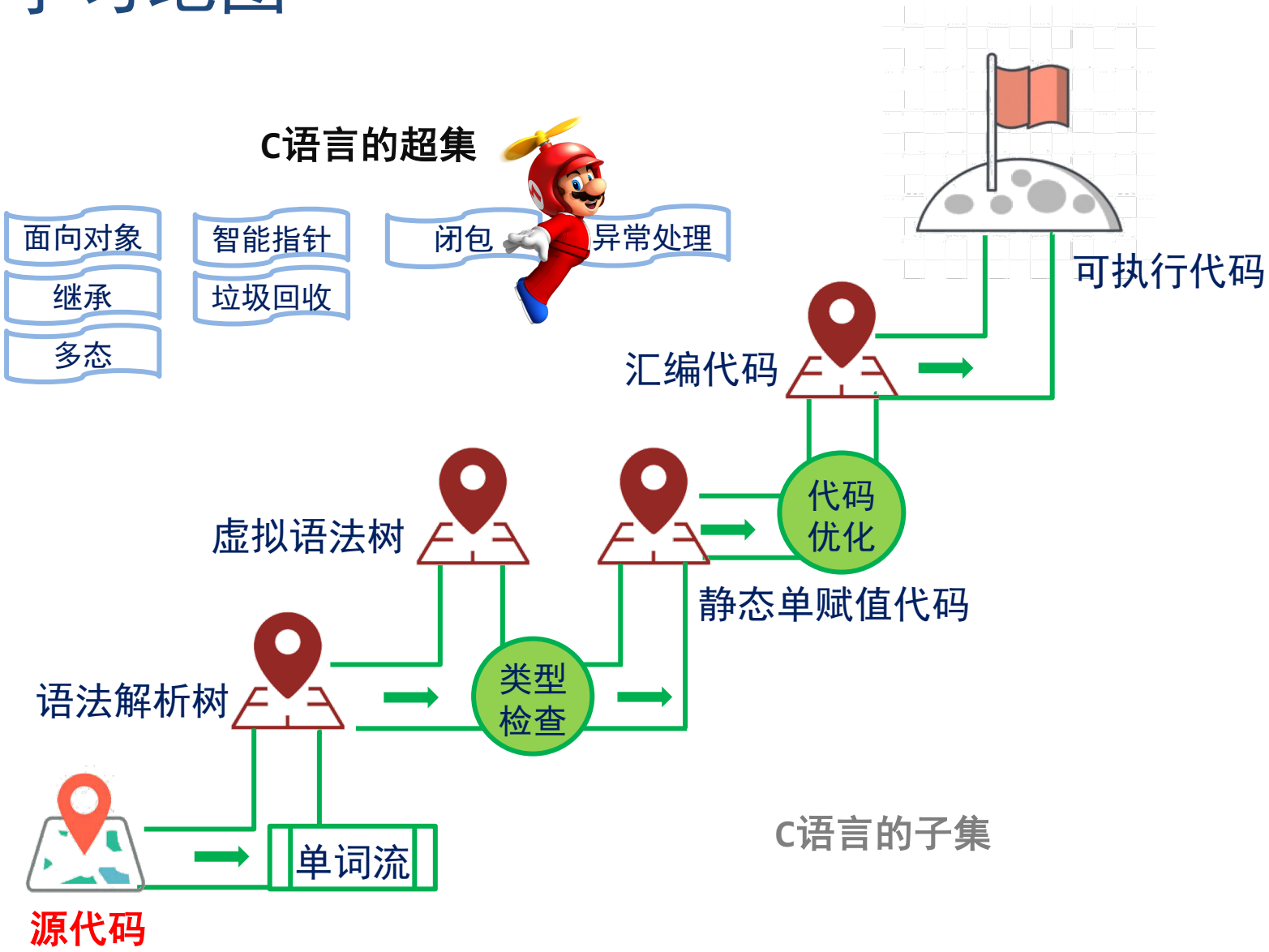
Lecture 9

异常处理

徐 辉

xuh@fudan.edu.cn

学习地图


C语言的超集
面向对象
继承
多态
智能指针
垃圾回收
闭包
异常处理
可执行代码
汇编代码
代码
优化
静态单赋值代码
虚拟语法树
类型
检查
语法解析树
单词流
源代码
C语言的子集

大纲

一、异常处理问题

二、栈展开

三、恢复地址

四、资源回收

一、异常处理问题

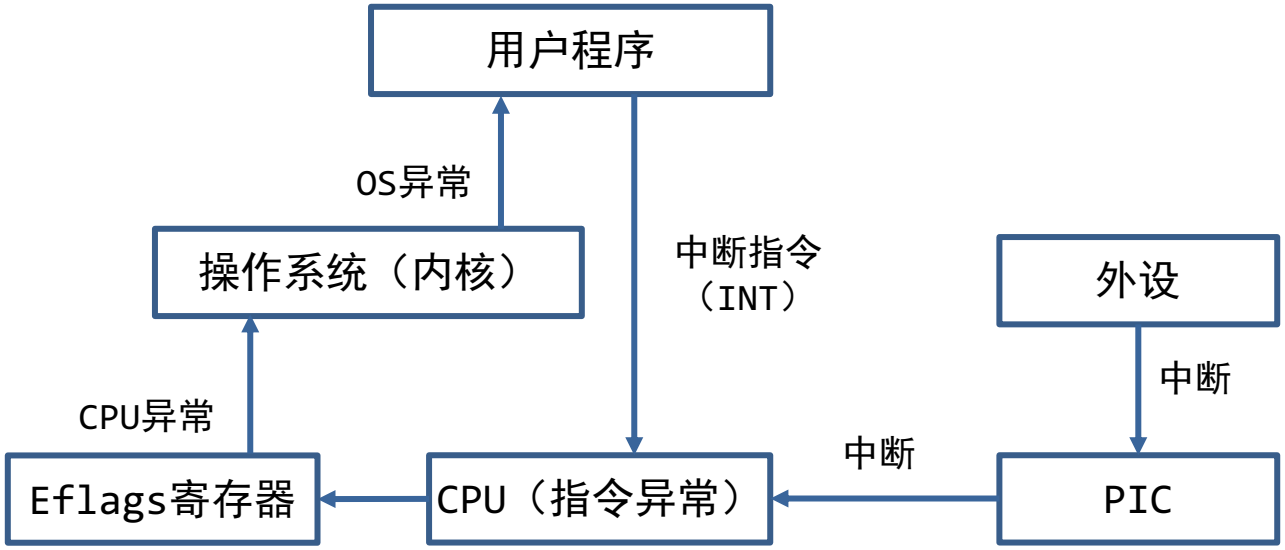
为什么需要异常处理？

- 程序运行期间可能遇到各种系统失效的情况；
- 继续运行程序会造成未知后果。
- Ariane 5火箭发射失败的例子：
  - 水平加速仪器读数异常，
  - 64bit浮点数转换为16bit整数，
  - 未在转换前作检查（性能考虑）。

Jacques-Louis Lions et al., Ariane 501 Inquiry Board report

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
    - 0x10 x87 Floating Point Exception

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

```
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

```
#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().
```

另外一种异常分类分法

- Abort：不能恢复的异常
- Fault：大概率可以恢复
- Trap：用户定义的异常，可以恢复
- Interrupt：中断，可以恢复

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

- 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：rbp、rsp、rbx、r12-r15

```
#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
```

问题

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

基于cleanup属性实现资源回收

```
void free_buffer(char **buffer){
  printf("Freeing buffer\n");
  free(*buffer);
}

void toy(){
  char *buf __attribute__
((__cleanup__(free_buffer))) = malloc(10);
  snprintf(buf, 10, "%s", "any chars");
  printf("Buffer: %s\n", buf);
}

int main(int argc, char **argv){
  toy();
  return 0;
}
```

```
#:./a.out
Buffer: any chars
Freeing buffer
```

```
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     eax, 14h
mov     edi, eax
call    _malloc
mov     rdi, offset aS
mov     [rbp+var_8], rax
mov     rsi, [rbp+var_8]
mov     al, 0
call    ___isoc99_scanf
mov     rdi, offset aBufferS
mov     rsi, [rbp+var_8]
mov     [rbp+var_C], eax
mov     al, 0
call    _printf
lea     rdi, [rbp+var_8]
mov     [rbp+var_10], eax
call    free_buffer
add     rsp, 10h
pop     rbp
retn
```

如果程序运行异常cleanup是否还有效?

```
void free_buffer(char **buffer){
  printf("Freeing buffer\n");
  free(*buffer);
}

void b(){
    printf("%s\n", 0x1111);
}

void a(){
  char *buf __attribute__ ((__cleanup__(free_buffer))) = malloc(10);
  snprintf(buf, 10, "%s", "any chars");
  printf("Buffer: %s\n", buf);
  b();
}

int  main(){
    a();
    return 0;
}
```

```
#:./a.out
Buffer: any chars
Segmentation fault (core dumped)
```

问题

- 如何使cleanup routine在异常处理时生效?

二、栈展开

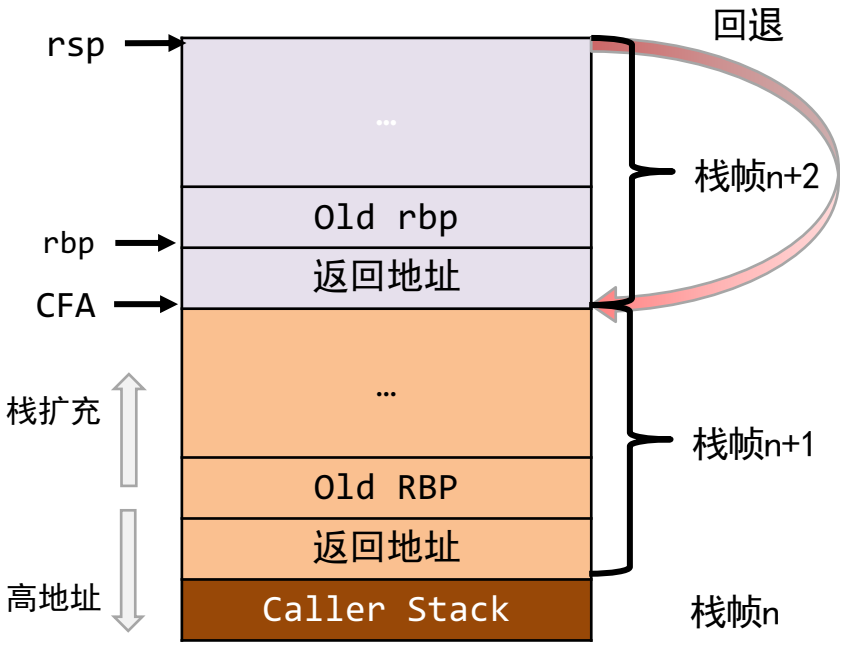
Stack unwinding

还原callee-saved寄存器

栈展开问题

```
int factorial(int n) {
   if(n == 0) {
      return 1;
   } else {
      return n * factorial(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
- CFA: canonical frame address
  - 栈帧的起始位置

编译时保存

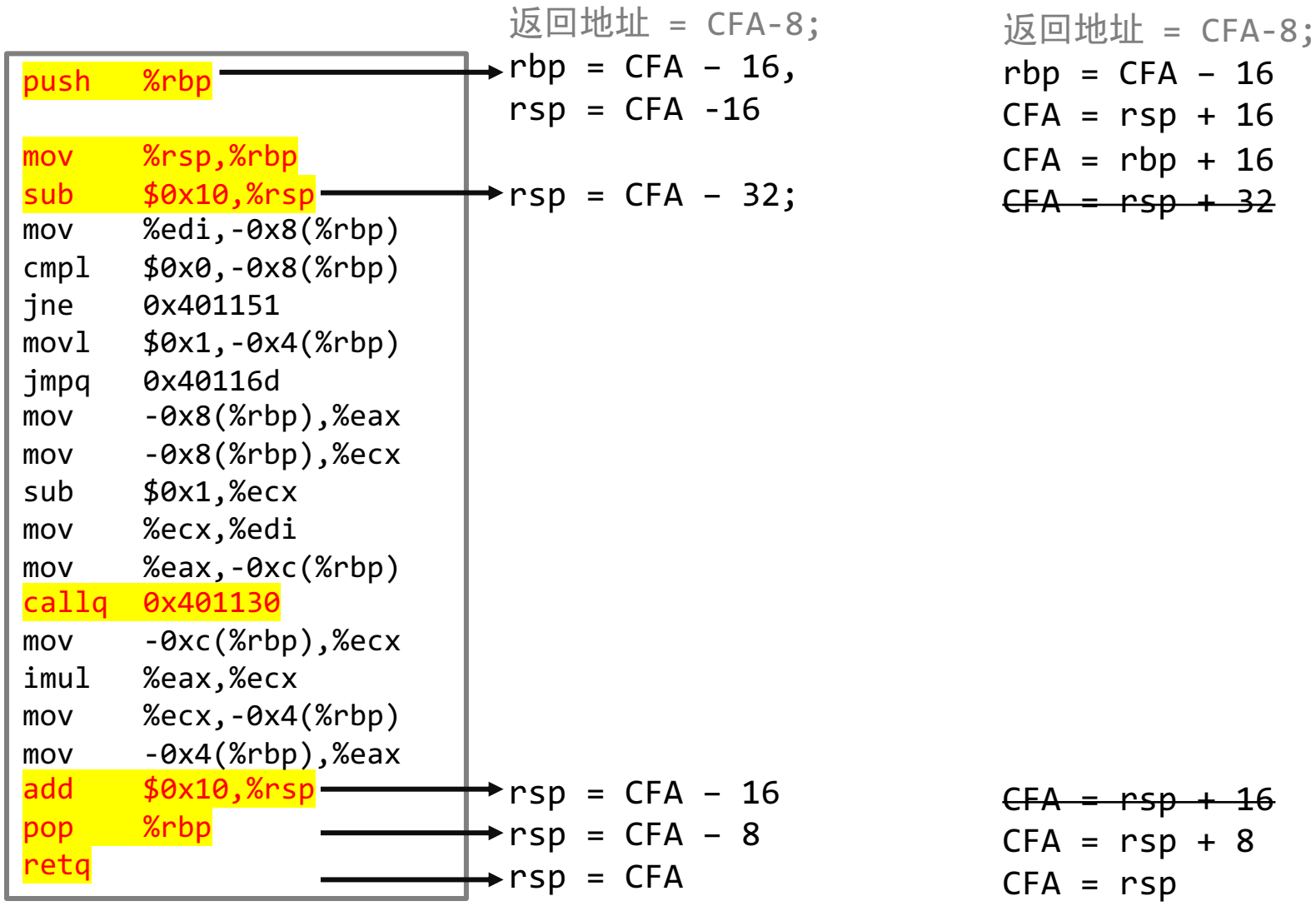
- 将异常处理所需数据提前保存在程序文件中
  - 遵循DWARF程序调试格式；
  - 不同于基于setjmp/longjmp的运行时方式。
- 通过ABI异常处理标准定义异常处理方式；
  - 根据异常位置确定恢复指令位置；
  - 退栈、恢复callee-saved寄存器。
- 无需在正常程序控制流中内联异常处理代码，开销低。

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

如何在编译时记录栈信息？

- 主要思路：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？
  1) 确定返回地址；
     - 有相对固定的保存位置
  2) 恢复callee-saved的寄存器。
     - 分析哪些指令会改变callee-saved寄存器？
       - 操作数涉及rbx/rbp/rsp/r12/r13/r14/r15
       - 改变栈帧的操作：push/pop

以栈帧基地址CFA为记录基准



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
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

练习

```
cab0: endbr64
cab4: push    %r13
cab6: mov     %rsi, %r13
cab9: mov     $0x2e, %esi
cabe: push    %r12
cac0: push    %rbp
cac1: mov     (%rdi), r12
cac4: mov     %r12, %rdi
cac7: call    4960
cacc: mov     0x0(%r13), %r13
cad0: mov     $0x2e, %esi
cad5: mov     %rax, %rbp
cad8: mov     %r13, %rdi
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: retn
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
```

LOC	CFA	rbp	r12	r13	ra
cab0	rsp+8	u	u	u	c-8
cab6	rsp+16	u	u	c-16	c-8
cac0	rsp+24	u	c-24	c-16	c-8
cac1	rsp+32	c-32	c-24	c-16	c-8
cb03	rsp+24	c-32	c-24	c-16	c-8
cb05	rsp+16	c-32	c-24	c-16	c-8
cb07	rsp+8	c-32	c-24	c-16	c-8
cb10	rsp+32	c-32	c-24	c-16	c-8
cb1d	rsp+24	c-32	c-24	c-16	c-8
cb25	rsp+16	c-32	c-24	c-16	c-8
cb27	rsp+8	c-32	c-24	c-16	c-8

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 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;
}
```

```
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]
./a.out(_start+0x2e) [0x4010ce]
```

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，只能层层展开。

三、恢复地址

基本概念

- Landing Pad：用于捕获异常和释放资源的用户代码。由Personality routine
- 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 $_ZTIPKc, %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

Call Site 1

Call Site 2

Call Site 3

```
.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:
```

```
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   _ZTIPKc                 # TypeInfo 1
```

GCC Except Table: a()函数

Call Site 1

Call Site 2

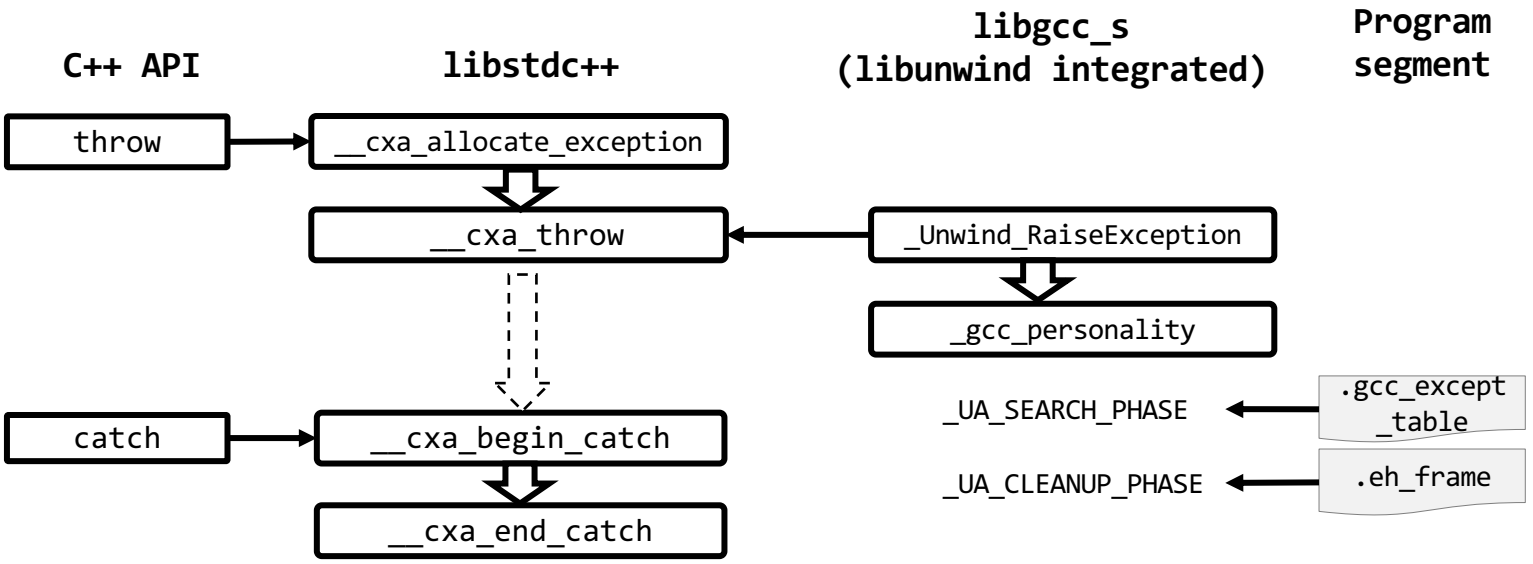
Call Site 3

Call Site 4

```
# %bb.0:pushq   %rbp
        movq    %rsp, %rbp
        subq    $48, %rsp
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %edi
.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)
.Ltmp3: movl    $_ZTIPKc, %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 <<
                                        #   Catch TypeInfo 1
        .byte   0                       #   No further actions
        .byte   2                       # >> Action Record 2 <<
                                        #   Catch TypeInfo 2
        .byte   125                     #   Continue to action 1
        .p2align        2
                                        # >> Catch TypeInfos <<
        .long   _ZTIi                   # TypeInfo 2
        .long   _ZTIPKc                 # TypeInfo 1
```

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)
```

四、资源回收

有哪些资源需要回收？

- 栈展开过程中：
  - 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

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
```

参考资料

- http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
- https://llvm.org/docs/ExceptionHandling.html
- https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- https://maskray.me/blog/2020-12-12-c++-exception-handling-abi