# 计算机科学与工程学院

# 课程设计报告

题目全称：反汇编分析 Windows 异常处理机制

课程名称：计算机操作系统 、任课教师：蒲晓蓉

指导老师：李林 职称：

| 序号 | 学生姓名 | 学号 | 班号 | 成绩 |
|---|---|---|---|---|
| 1 | 党标 | 2010063040035 | 2010063040 | |
| 2 | 胥钰淋 | 2010063040021 | 2010063040 | |
| 3 | 魏一驹 | 2010063040012 | 2010063040 | |
| 4 | 叶文婷 | 2010063040003 | 2010063040 | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| | | | | |

（注：学生姓名填写按学生对该课程设计的贡献及工作量由高到底排列，分数按排名依次递减。序号排位为“ 1”的学生成绩最高，排位为“ 10”的学生成绩最低。）

指导老师评语：

签字：

# 本小组成员任务分工情况

| 序号 | 姓名 | 学号 | 任务分工 | 完成情况 |
|---|---|---|---|---|
| 1 | 党标 | 2010063040035 | 编译器级 SHE研究 | |
| 2 | 胥钰淋 | 2010063040021 | 系统级 SHE研究 | |
| 3 | 魏一驹 | 2010063040012 | 终止处理研究 | |
| 4 | 叶文婷 | 2010063040003 | 文档撰写 | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |

# 摘 要

结构化异常处理（ Structured Exception Handling ）是 Windows为应用程序提供的一种系统服务。几乎所有介绍 Windows高级编程的书籍都会详细的介绍结构化异常处理的作用和使用方法。但是，很少有书籍详细的介绍结构化异常处理的实现机制。而微软官方也没有提供结构化异常处理的源代码及机制介绍。所以，大部分的程序开发者对结构化异常处理是知其然不知其所以然。本文，将利用反汇编的方法一层一层地剖开结构化异常处理的内部实现机制。

**关键词：** 结构化异常处理，反汇编， Window 核心

# ABSTRACT

At its heart, Win32 Structured Exception Handling is an operating system-provided service. All the docs you're likely to find about SEH describe one particular compiler's runtime library wrapping around the operating system implementation. However, few docs explain the internal of SEH. In this paper, I'll strip SEH to its most fundamental concepts.

**Keywords:** SEH, Win32, Disassembly

# 目录

# 第一章 绪论

## 1.1 选题背景及意义

程序在运行的过程中难免会发生一些不可预知的特殊情况，例如，除数为零、访问了一个不存在的内存空间等等。在计算机中，我们把这种特殊情况称为异常。那么当程序运行过程中产生异常，操作系统应该如何处理它呢？当然，最简单的方式是直接终止该进程。但是，在某些稳定性要求较高的系统中，因为一个异常就贸然终止进程所带来的损失是巨大的。例如，一个证券交易系统中，如果仅因为一个异常而终止整个系统，那么大量用户将会因此而得不到服务甚至遭受经济损失。

试想，如果程序发生异常时，操作系统能够按照程序员预先设定好的方式处理异常，例如，程序员可以设定为对异常进行修复并继续执行、跳过产生异常的指令继续执行或者直接退出等等。那么贸然终止进程所带来的损失便可以得到缓解。程序开发者也就从这个令人头疼的异常处理中解脱了出来，从而提高的程序的开发效率。

微软公司的 Windows操作系统就为应用程序提供了这样一直系统服务，即结构化异常处理（ Structured Exception Handling ），下文中我们将简称为 SEH。一般情况下，程序开发者不直接使用系统 SEH，而是通过编译器的帮助来使用 SEH。编译器通过运行时库对操作系统的 SEH进行了封装。开发者只需在程序中使用编译器定义的 _try, _finally和 _except关键字，程序即可在编译器的帮助下得到系统 SEH的支持。

尽管我们可以毫不费力地找到介绍 SEH用法的文档，我们却很难找到介绍 SEH实现细节的文档。一方面，由于微软的 Windows并非一个开源的操作系统，所以，微软官方并没有提供有关 SEH实现细节的源代码。另一方面，主流的编译器也都选择了不公开其对 SEH支持的核心层源代码。这些原因导致 SEH成了 Windows服务中应用最广泛，但是却又最缺乏文档的技术。

本文，我们将利用反汇编技术一层一层的剖析 SEH实现机制。从而让 SHE不再神秘。

## 1.2 主要内容与章节安排

接下来，第二章将介绍本次课程设计的研究环境和目标；第三章将介绍 SHE中终止处理的使用方法及并分析实现机制；第四章则会介绍系统级的SHE及其主要数据结构；第五章将详细阐述编译器级的 SHE数据结构及其实现机制。在最后一章，我们将对本文所做的工作做出总结，并对以后的研究做出展望。

# 第二章 研究环境与目标

## 2.1 研究环境

| | |
|---|---|
| 系统环境 | Microsoft Windows 7 旗舰版 |
| 编译环境 | Microsoft Visual Studio 2010 |
| 调试环境 | Microsoft Visual Studio 2010 |
| 绘图工具 | 亿图图示专家 2012 |

## 2.2 研究目标

1. 学会 SHE 的基本使用方法
2. 理清系统级和编译器级 SHE 实现机制
3. 画出 SHE 异常处理链栈帧结构图
4. 分析并列和嵌套的异常处理块
5. 分析线程入口函数

# 第三章 终止处理机制分析

## 3.1 终止处理程序

终止处理程序是 SHE 体系的一个重要组成部分。终止处理程序能确保调用和执行一个代码块，对应与具体的实现，终止处理的使用方法如下：

| 终止处理的使用方法 |
| --- |
| __try {<br>// 受保护的代码<br>}<br>__finally {<br>// 终止处理程序<br>} |

try 块可能会因为 return， goto ，异常等非自然退出，也可能会因为成功执行而自然退出。但不论 try 块是如何退出的， finally块的内容都会被执行。当 try 块中有 return， goto ，异常等语句试图过早的退出 try 块时，编译程序会增加一些额外的代码以保证finally块中的代码首先被执行。这种情况下，引发 finally块的执行的过程即是一个局部展开的过程。当 try 块中代码自然退出时，程序会通过一条 call 指令转去执行 finally块。除此之外，另一种能引起 finally块执行的是全局展开。全局展开和异常处理程序紧密相关，我们将在下文中详细介绍。

## 3.2 终止处理的实现机制

上节中，我们了解了终止处理的使用方法和其适用的几种情况。本节中我们将详细解释编译器如何保证终止处理程序总是被执行。

首先，我们定义下面两个函数：

示例 1：定义函数 Funcenstein1( )，并在函数中使用 SHE 的终止处理服务。try 块中不使用显示的非正常退出。

示例 2：定义函数 Funcenstein2( )，并在函数中使用 SHE 的终止处理服务。但在 try 块中使用 goto 语句显示的非正常退出。

代码如下：

| 示例 1 | 示例 2 |
| --- | --- |
| int Funcenstein1( ){<br>int i; | int Funcenstein2( ){<br>int i; |

```
__try{
    i = 5;
}
__finally{
    i = 0;
}
return i;
}
```

```
__try{
    i = 5;
    goto NEXT;
}
__finally{
    i = 0;
}
NEXT:
  return i;
}
```

调试运行示例 1 中的代码，在 try 块结束的位置下断点，并转到反汇编窗口。结果如图 3-1 所示。

```
    43:         i = 5;
0117143C C7 45 E0 05 00 00 00 mov         dword ptr [ebp-20h],5
    44:     }
    45:     __finally{
01171443 C7 45 FC FE FF FF FF mov         dword ptr [ebp-4],0FFFFFFFEh
0117144A E8 02 00 00 00       call        $LN5 (1171451h)
0117144F EB 08                jmp         $LN8 (1171459h)
    46:         i = 0;
01171451 C7 45 E0 00 00 00 00 mov         dword ptr [ebp-20h],0
$LN6:
01171458 C3                   ret
    47:     }
    48:     return i;
01171459 8B 45 E0             mov         eax,dword ptr [ebp-20h]
    49: }
```

图 3-1 正常退出 try 块的情况

从图 3-1 中我们可以看出，当正常退出 try 块时，编译器将finally块中的代码视为一个终止处理程序，通过增加call指令执行终止处理程序。

调试运行示例 2 中的代码，在 goto 语句所在位置下断点，并转到反汇编窗口。结果如图 3-2 所示。

```
    53:     __try{
00231505 C7 45 FC 00 00 00 00 mov         dword ptr [ebp-4],0
    54:         i = 5;
0023150C C7 45 E0 05 00 00 00 mov         dword ptr [ebp-20h],5
    55:         goto NEXT;
00231513 6A FE                push        0FFFFFFFEh
00231515 8D 45 F0             lea         eax,[ebp-10h]
00231518 50                   push        eax
00231519 68 00 70 23 00       push        offset ___security_cookie (237000h)
0023151E E8 64 FB FF FF       call        @ILT+130(__local_unwind4) (231087h)
00231523 83 C4 0C             add         esp,0Ch
00231526 EB 16                jmp         NEXT (23153Eh)
    56:     }
    57:     __finally{
```

图 3-2 非正常退出 try 块的情况

如图 3-2 所示，当程序非正常退出 try 块时，编译器会在程序非正常退出 try 块之前增加一段代码。该代码调用了函数 __local_unwind4。该函数即是完成局部展开功能的函数。局部展开函数中将直接或间接调用finally块中的终止处理程序从而保证终止处理程序被执行。下面我们给出 __local_unwind4的汇编代码及注释。

_LOCAL_UNWIND4 汇编代码 ：

```
_local_unwind4:
 push       ebx
push       esi
push       edi
 mov       edx,dword ptr [esp+10h]   ; edx = __security_cookie
 mov       eax,dword ptr [esp+14h]   ; eax = pExceptionRegistartion
 mov       ecx,dword ptr [esp+18h]   ; ecx = ulUntilTryLevel
 push       ebp
push       edx                ; __security_cookie
 push       eax                ; pExceptionRegistartion
push       ecx                ; ulUntilTryLevel
push       ecx
 push       offset _unwind_handler4 (0FFAD660h) ; 处理展开中可能产生的异常
push       dword ptr fs:[0]                ;指向上一个异常帧
 mov       eax,dword ptr [___security_cookie (0FFF04BCh)]
 xor       eax,esp                    ;加密 __security_cookie
 mov       dword ptr [esp+8],eax     ; [esp+8] 被赋予加密后的 __security_cookie
 mov       dword ptr fs:[0],esp      ;加入异常链
_lu_top:
 mov       eax,dword ptr [esp+30h]  ; eax = pExceptionRegistartion
 mov       ebx,dword ptr [eax+8]    ; ebx = pExceptionRegistration->scopetable
 mov       ecx,dword ptr [esp+2Ch]  ; ecx = &__security_cookie
 xor       ebx,dword ptr [ecx]      ; 解密 scopetable
 mov       esi,dword ptr [eax+0Ch]  ; esi = pExceptionRegistration->trylevel
cmp        esi,0FFFFFFFEh
je         _lu_done (0FFAD652h) ;若 trylevel 等于 TRYLEVEL_INVALID ，结束

 mov       edx,dword ptr [esp+34h]  ; edx = ulUntilTryLevel
 cmp       edx,0FFFFFFFEh ; 比较指定的结束点是否等于 TRYLEVEL_INVALID
 je        _lu_top+22h (0FFAD624h) ; 若相等，跳过下一比较。

 cmp       esi,edx ; 比较当前 trylevel 和 ulUntilTryLevel，后者为指定的结束点
 jbe       _lu_done (0FFAD652h) ; 已经遍历到指定 scopetable_entry 了

 lea       esi,[esi+esi*2]
lea        ebx,[ebx+esi*4+10h]  ; ebx= scopetable[trylevel]
```

```
 mov       ecx,dword ptr [ebx]   ;ecx = scopetable[trylevel]-> previousTryLevel
 mov       dword ptr [eax+0Ch],ecx ; trylevel = ecx
cmp        dword ptr [ebx+4],0  ; 判断   lpfnFilter 是否等于   NULL
 jne       _lu_top (0FFAD602h)   若不等于，跳回 _lu_top. 继续遍历

 push      101h      ; 若   lpfnFilter 等于   NULL 则说明，此处对于   finally块
 mov       eax,dword ptr [ebx+8]
call       _NLG_Notify (0FFB0D15h)  ; 通知程序下面会执行终止处理程序

mov        ecx,1
 mov       eax,dword ptr [ebx+8]       ;eax =  相应终止处理程序的首地址
 call      _NLG_Call (0FFB0D34h)   ; 调用相应终止处理程序
 jmp       _lu_top (0FFAD602h)       ; 返回顶部，继续遍历
_lu_done:
pop        dword ptr fs:[0]           ;遍历结束，扫尾工作
add        esp,18h
pop        edi
pop        esi
pop        ebx
 ret
```

通过图 3-2 和反汇编分析局部展开函数的汇编代码，我们得知，局部展开函数共有三个参数。第一个参数指定 __security_cookie 的偏移量，第二个参数指定对应的异常帧首地址，第三个参数指定了遍历操作的终止层。局部展开函数会自内向外的遍历各层的 try 块，直到参数三指定的 try 块，如果其有对于的 finally块，它就会调用 _NLG_Call 函数间接的去执行相应的 finally块中的代码。其遍历流程如图 3-3 所示。

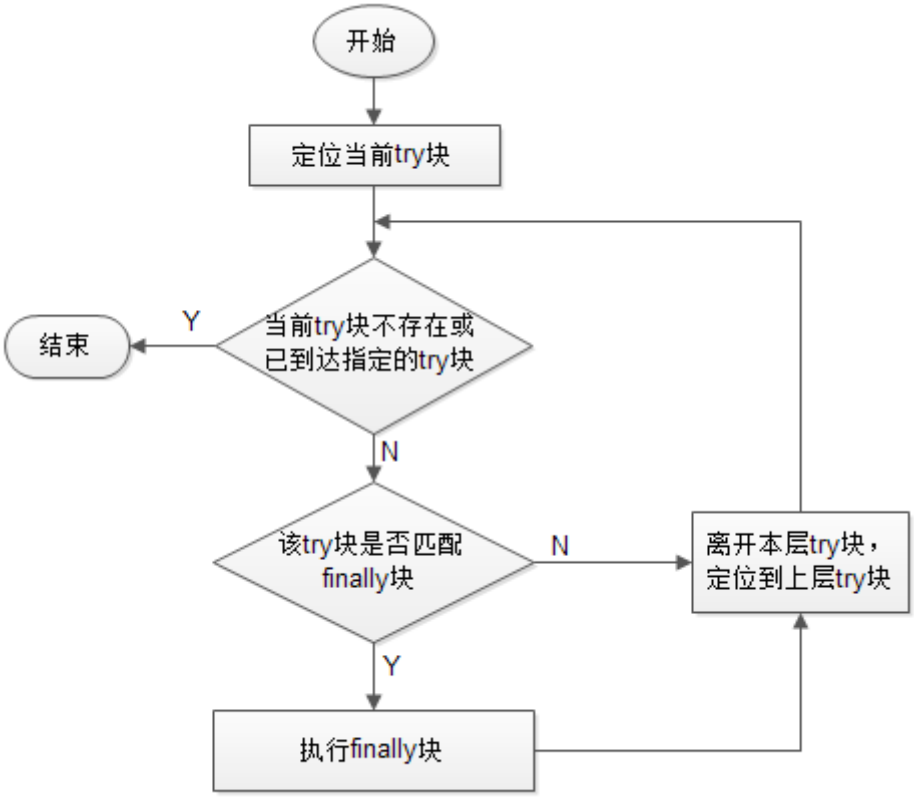


图 3-3 局部展开流程图

//这里可以加入局部展开的伪代码

至此，我们已经基本弄明白了什么是局部展开和局部展开的编译器级实现机制。在分析局部展开函数的时候我们引用了一些名词，如异常帧。也引用了一些结构，如 ExceptionRegistration 。那么这些新概念代表什么呢？实际上，这些概念来源于 SHE 中的异常处理功能。在下面的章节中，我们将逐步介绍这些结构体的定义和作用，并逐步展开异常处理程序的实现机制。

# 第四章 系统级异常处理的机制分析

## 4.1 系统级异常处理的概念

在第一章中，我们提到， SHE是 Windows操作系统为应用程序提供的一种系统服务。实际上， Windows 提供的异常处理机制实际上只是一个简单的框架，一般情况下开发人员都不会直接用到。开发人员通常所用的异常处理都是编译器在系统提供的异常处理机制上进行了扩展的版本。这里先不说扩展版，本章我们主要介绍系统级的异常处理。

具体地讲，操作系统提供了一种“触发异常 -解决异常”的异常管理机制。但是操作系统本身并不提供任何解决异常的方案。具体的异常处理程序由程序开发人员编写。当一个线程触发了一个异常，操作系统捕获异常，并从该线程对应的 SHE链表中找到可以处理这个异常的异常处理函数，然后执行该异常处理函数。

## 4.2 系统级异常处理的机制分析

### 4.2.1 异常处理链表

从系统级的异常处理流程可以看出，理解异常处理的关键是理解异常处理链表。那么异常处理链表的结构是怎样的呢？

SHE链表结点为结构体 _EXCEPTION_REGISTRATION_RECORD ，其定义如下：

_EXCEPTION_REGISTRATION_RECORD 结构体的定义

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
        struct _EXCEPTION_REGISTRATION_RECORD *  Next;
        PEXCEPTION_ROUTINE                       Handler;
}       EXCEPTION_REGISTRATION_RECORD            ,          *
PEXCEPTION_REGISTRATION_RECORD;
```

结构体内各个成员含义如下：

1. Next 成员指向下一个EXCEPTION_REGISTRATION_RECORD 。于是便构成了SHE链表。
2. Handler成员指向相应的异常处理函数。

有了链表节点的结构我们就可以分析出链表的组织方式了。 SHE链结构如图 4-1所示。

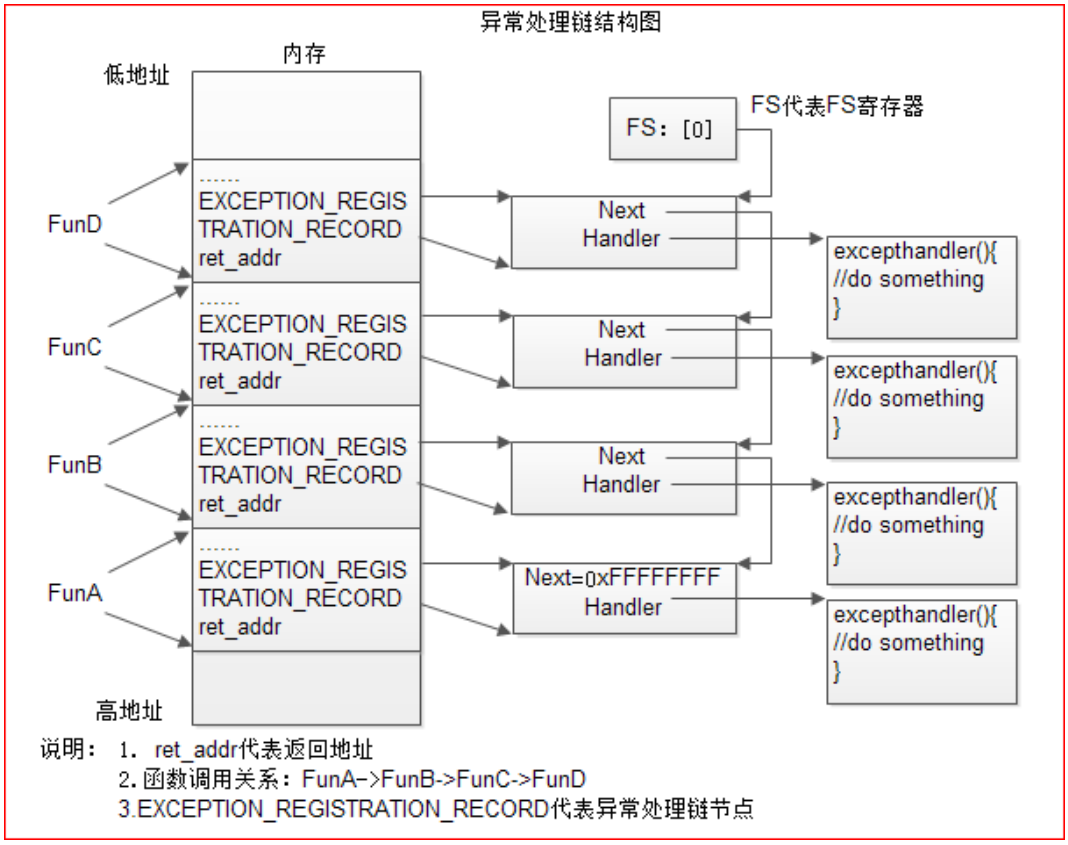


图 4-1 异常处理链结构图

如图 4-1 中所示， FS 寄存器指向的线程信息块 TIB，而 TIB 的前四个字节即是指向异常处理链的指针。所以，通过 FS:[0] 即可访问到异常处理链。

当一个异常发生，操作系统在异常触发到异常处理函数执行期间做了什么动作呢？为了了解这个流程，我们做了一个实验：我们构造了一个内存访问违规的硬件异常，在异常处理过滤器中下断点，调试运行。程序被断下后，查看程序的调用栈，如图 4-2 所示。

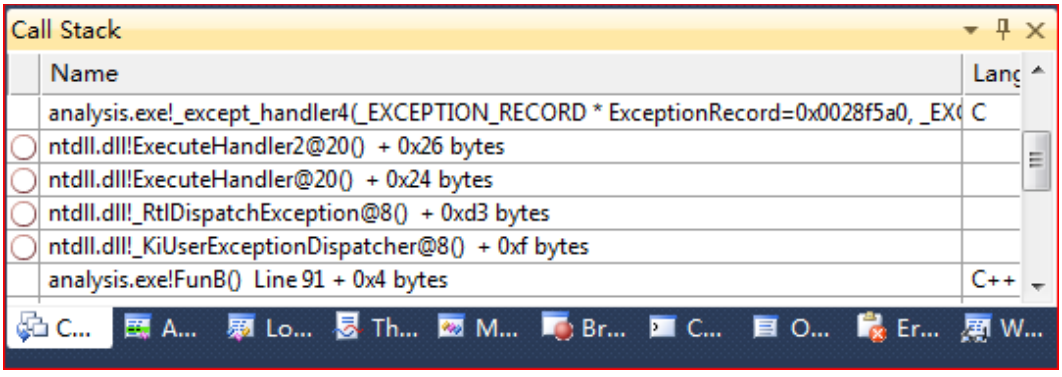


图 4-2 异常处理过程中的调用栈

从图 4-2 中，我们可以看出：当我们在函数 analysis.exe!FunB() 中的异常被 CPU 检测到后，操作系统的一系列函数将被调用，顺序如图 4-3 所示。

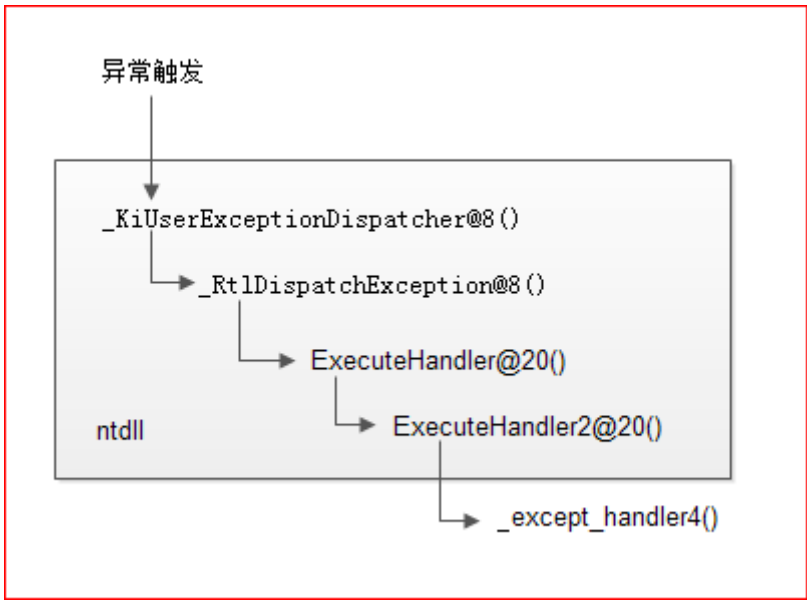


图 4-3 操作系统调用异常处理函数的流程

从图 4-3所示的流程，我们可以看出，当一个异常被触发，操作系统会执行一系列的异常分发函数，并调用异常处理函数。从函数 _KiUserExceptionDispatcher@8() 的命名中我们可以得知，这个函数属于用户级。那么，或许还存在对应的系统级函数。从异常发生到调用该用户级函数的过程中一定也发生了模式的切换。但是，这些已经超出了我们的研究范围，我们暂且不去管它。

### 4.2.2 异常处理回调函数

我们前面提到，操作系统的 SHE只是一个框架，它不会对异常给出任何解决方案，只会把异常抛给用户进程，让用户自行处理。通过上面的介绍，我们已经了解操作系统调用用户的异常处理函数的整个流程。但是，用户的异常处理函数如何知道用什么方案去处理操作系统抛给它的异常呢？所以操作系统需要告诉异常处理函数，当前发生了什么样的异常。此信息通过结构体参数传递给异常处理函数。

下面我们给出异常处理函数的原型和相关结构体的定义：

异常回调函数的原型（该原型在 VC/include/excpt.h 中有声明）

```
EXCEPTION_DISPOSITION __cdecl _except_handler (
    _In_ struct _EXCEPTION_RECORD *_ExceptionRecord,
    _In_ void * _EstablisherFrame,
    _Inout_ struct _CONTEXT *_ContextRecord,
    _Inout_ void * _DispatcherContext
);
```

第一个参数是指向 EXCEPTION_RECORD 结构指针，该结构在 WINNT.H中定义（成员含义查阅 MSDN）：

_EXCEPTION_RECORD结构体的定义

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    ULONG_PTR
    ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

说明：

1.ExceptionCode 是有操作系统提供给异常的一个数，代表异常发生的原因。例如 STATUS_ACCESS_VIOLATION 的编码为 0xC0000005。

2.ExceptionAddress 表示异常发生的地址。

3. 其它成员我们暂时忽略

第二个参数是指向异常帧结构的指针，是一个很重要的参数。在操作系统级，其异常帧结构即是我们上面介绍的 EXCEPTION_REGISTRATION_RECORD 。然而在编译器对该异常帧进行了扩展。扩展部分内容我们会在下一章中详细介绍。

第三个参数是CONTEXT结构的指针， CONTEXT结构在 WINNT.H中定义，它表示特定线程异常发生时寄存器的值。CONTEXT结构如下：

CONTEXT结构体的定义

```
typedef struct _CONTEXT {
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
    DWORD   Eax;
    DWORD   Ebp;
    DWORD   Eip;
    DWORD   SegCs;          // MUST BE SANITIZED
```

```
    DWORD   EFlags;          // MUST BE SANITIZED
    DWORD   Esp;
    DWORD   SegSs;
    BYTE    ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT ,*PCONTEXT;
```

第四个参数是 _DispatcherContext，用于存放下一个异常帧，全局展开在执行当前异常帧局部展开中发生嵌套展开时使用。暂时忽略。

当一个异常产生，操作系统首先根据异常信息填写 _EXCEPTION_RECORD、CONTEXT 等结构。然后调用一系列的异常分发函数、异常执行函数，最后调用用户自定义的异常处理函数，并把和该异常有关的 _EXCEPTION_RECORD、CONTEXT 等结构以参数的形式传递给异常处理函数。

# 第五章 编译器级的异常处理机制

## 5.1 编译器级的异常处理的基本用法

操作系统级的 SHE 可以很大程度上提供用户程序的健壮性和容错性。但是，如上文所讲 SHE 是基于异常帧的，构造异常帧显然是一个机械而又繁琐的过程。因此，大部分情况下，程序员并不会直接使用操作系统级 SHE，而是通过编译器来使用。编译器对系统级的 SHE 进行了封装。程序开发人员只需简单的使用 __try/__except关键字即可为程序添加异常处理功能。

编译器的 __try/__except的使用方法如下：

示例5-1 异常处理的使用方法

```
__try {
  // 受保护的代码
}
__except( 过滤表达式 ) {
  // 异常处理程序代码
}
```

编译器会根据用户的提供的过滤表达式和 __except 块中的异常处理代码生成一个异常处理回调函数。经分析，编译器将这个函数命名为 _except_handler4() 。

回想我们在第三章中提到的终止处理 __try/__finally 关键字。事实上，从编译器级的 SHE 来看， __try/__finally 和 __try/_except 的实现机制是一致的。坦白的说，我们可以将 __finally 块看成是总是执行的 __except 块。

语法规定，一个__try块后可以跟一个 __except 或者 __finally ，不能同时跟两个关键字。但是 try 块可以嵌套使用。为了说明这个问题，我们给出一个错误用法，和一个正确用法：

示例 5-2 异常处理的错误用法

```
__try {
      // 受保护的代码
}
__except( 过滤表达式 ) {
      // 异常处理程序代码
}
__finally{
      //终止处理程序代码
}
```

示例 5-3 异常处理的嵌套用法

```
__try {
    __try{
        __try{
        // 受保护的代码
        }
        __except( 过滤表达式 2){
        // 异常处理程序代码 2
        }
    }
    __except( 过滤表达式 1) {
      // 异常处理程序代码 1
    }
}
  __finally{
      // 终止处理程序代码
}
```

示例 5-3 中，我们共嵌套了三层，其中有两层 __except 块。也就是说，我们想要在这里构造两个异常处理回调函数。回想图 4-1 所示的异常链结构，我们发现，一个异常帧（一个函数只能对应个异常帧）中只有一个异常处理回调函数的指针。那么我们如何让它指向两个回调函数呢？事实上，一个指针是不能指向两个或更多个函数的。这就需要编译器为我们做些事情了。

## 5.2 编译器级的异常处理机制分析

### 5.2.1 扩展异常帧

编译器级的 SHE 将我们通过 __except 块构造的一段或多段异常处理程序代码封装成了一个异常处理回调函数，即 _except_handler4()，然后让异常帧中的函数指针指向了这个函数。显然，单单利用操作系统 SHE 中的数据结构不足以完成这项任务。

事实上，编译器级的 SHE 对操作系统支持的异常帧结构进行了扩展。扩展后的结构如下：

扩展后的异常帧结构 _EXCEPTION_REGISTRATION 的定义

```
typedef struct _EXCEPTION_REGISTRATION {
    struct _EXCEPTION_REGISTRATION *   prev;
     PEXCEPTION_ROUTINE            handler;
    struct scopetable_entry *          scopetable;
     int                          trylevel;
     int                          _ebp;
} EXCEPTION_REGISTRATION，* PEXCEPTION_REGISTRATION;
```

前两个域构成了操作系统级的异常帧结构 EXCEPTION_REGISTRATION_RECORD。第三个域指向一个 struct scopetable_entry 类型的数组。其实数组的每一个元素对于一个 try 块。第四个域 trylevel 是一个整数，编译器按照 try 块的出现顺序对 try 块进行编号。而 trylevel 记录这个编号，表明当前指令处在哪个 try 块中。执行过程中，随着 CPU 执行不同 try 块中的代码，该值会不断改变。另外，由于 scopetable 数组中元素和 try 块一一对应，所以 trylevel 的数值可以用来索引当前指令所对应的 scopetable 元素。第四个成员 _ebp 是通过进入函数后的第一条指令 push ebp 指令压入栈的。

除此之外，编译器还在栈中保留了两个 DWORD，分别在 ebp-14h和 ebp-18h处，ebp-18h处的 DWORD是用来保存的是所有 prolog 工作完成后的 esp 的正常值。 ebp-14h 处的 DWORD 为了支持内联函数 GetExceptionInformation 和 GetExceptionCode， ebp-14h处的 DWORD将保存一个指向 EXCEPTION_POINTERS 结构的指针。实际上，这个指针就是内联函数 GetExceptionInformation 的返回值。而 EXCEPTION_POINTERS 就是一个指向 EXCEPTION_RECORD 的指针。当你调用内联函数 GetExceptionInformation 时，编译器生成如下代码：

```
GetExceptionInformation 的反汇编代码
MOV  EAX, DWORD PTR [EBP-14] ; 执行完毕， EAX 指向EXCEPTION_POINTERS结构
```

内联函数 GetExceptionCode 类似。它的返回值是 GetExceptionInformation 返回的 EXCEPTION_POINTERS 所指向的 EXCEPTION_RECORD 结构中 ExceptionCode 成员。当你使用 GetExceptionCode 函数时，编译器生成如下代码：

```
GetExceptionCode 的反汇编代码
MOV EAX,DWORD PTR [EBP-14] ; 执行完毕，EAX 指向 EXCEPTION_POINTERS 结构
MOV EAX,DWORD PTR [EAX] ; 执行完毕，EAX 指向 EXCEPTION_RECORD 结构
MOV EAX,DWORD PTR [EAX] ; 执行完毕，EAX 中是 ExceptionCode 的值
```

经过了上面的扩展后，我们发现异常处理的帧结构变成了如图5-1所示。

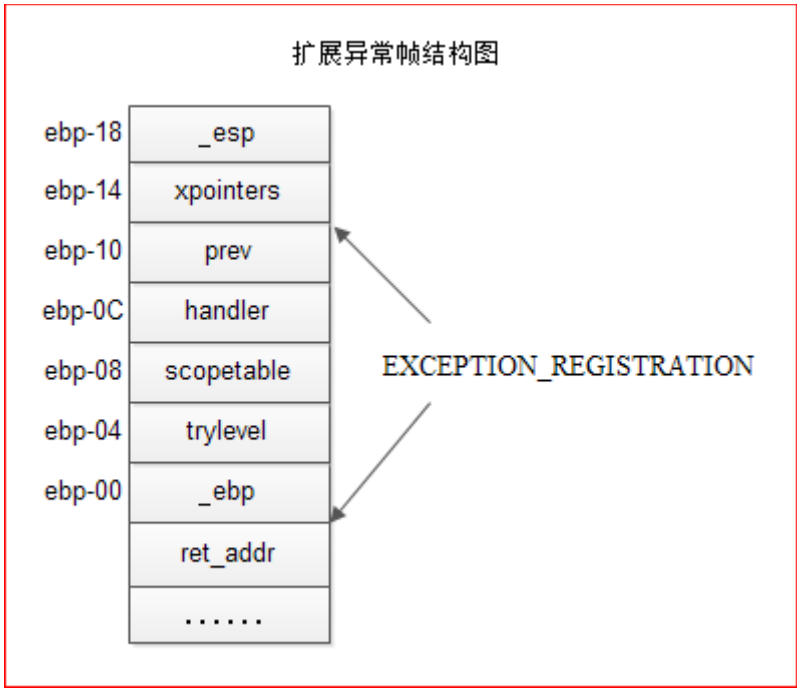


图 5-1 扩展异常帧结构图

### 5.2.2 Scopetable 的结构及作用

编译器能够将多个 try 块异常处理代码封装到一个 _except_handler4(),其利用的关键数据结构就是扩展异常帧中的 scopetable 数组。数组中的每一个元素对应一个 try 块。下面我们给出该数组元素的结构声明：

scopetable_entry 的结构定义

```
typedef struct socpetable_entry
{
DWORD previousTryLevel;
DWORD lpfnFilter;
DWORD lpfnHandler;
} SCOPETABLE, *PSCOPETABLE;
```

该结构中的第一个域 previousTryLevel 反应了 try 块的嵌套情况，它保存了当前块的外层块所对应的编号。若此块已经是最外层，则该值为 TRYLEVEL_INVALID(-2) 。第二域是一个指向过滤表达式的指针。若该块为 __finally 块，则值为 NULL 。第三个域是一个指向 handler 函数的指针，该函数的内容即是我们在 __except 块中所写的代码。

下面我们通过实例来说明 trylevel 和 scoptable 的作用。实例代码如图 5-2 所示。

```
void
FunTestTrylevel(){
    __try{
        __try{
        int i = 0;
        }
        __except(EXCEPTION_CONTINUE_SEARCH){

        }
    }
    __except(EXCEPTION_EXECUTE_HANDLER){
    }
    __try{
    }
    __finally{
    }
}
```

图 5-2 测试用代码

调试运行，在 i=0 处下断点。进入反汇编窗口。首先让我们来看下进入 try 块之前的指令。如图 5-3 所示。

```
   112: void
   113: FunTestTrylevel(){
012037A0  push        ebp
012037A1  mov         ebp,esp
012037A3  push        0FFFFFFFEh  ; trylevel初始化为-2
012037A5  push        offset ___rtc_tzz+104h (1206B30h)
012037AA  push        offset @ILT+115(__except_handler4)
012037AF  mov         eax,dword ptr fs:[00000000h]
012037B5  push        eax
012037B6  add         esp,0FFFFFF2Ch
012037BC  push        ebx
012037BD  push        esi
012037BE  push        edi
012037BF  lea         edi,[ebp-0E4h]
012037C5  mov         ecx,33h
012037CA  mov         eax,0CCCCCCCCh
012037CF  rep stos    dword ptr es:[edi]
012037D1  mov         eax,dword ptr [___security_cookie
012037D6  xor         dword ptr [ebp-8],eax
012037D9  xor         eax,ebp
012037DB  push        eax
012037DC  lea         eax,[ebp-10h]
012037DF  mov         dword ptr fs:[00000000h],eax
012037E5  mov         dword ptr [ebp-18h],esp
```

初始化异常帧

将异常帧加入异常链

图 5-3 异常帧的初始化

从图 5-3 中我们可以看出初始化异常帧的时候， trylevel 被初始化为 -2 ，

表示当前不在任何块中。

接下来我们分析进入 try 块的过程。如图5-4所示。

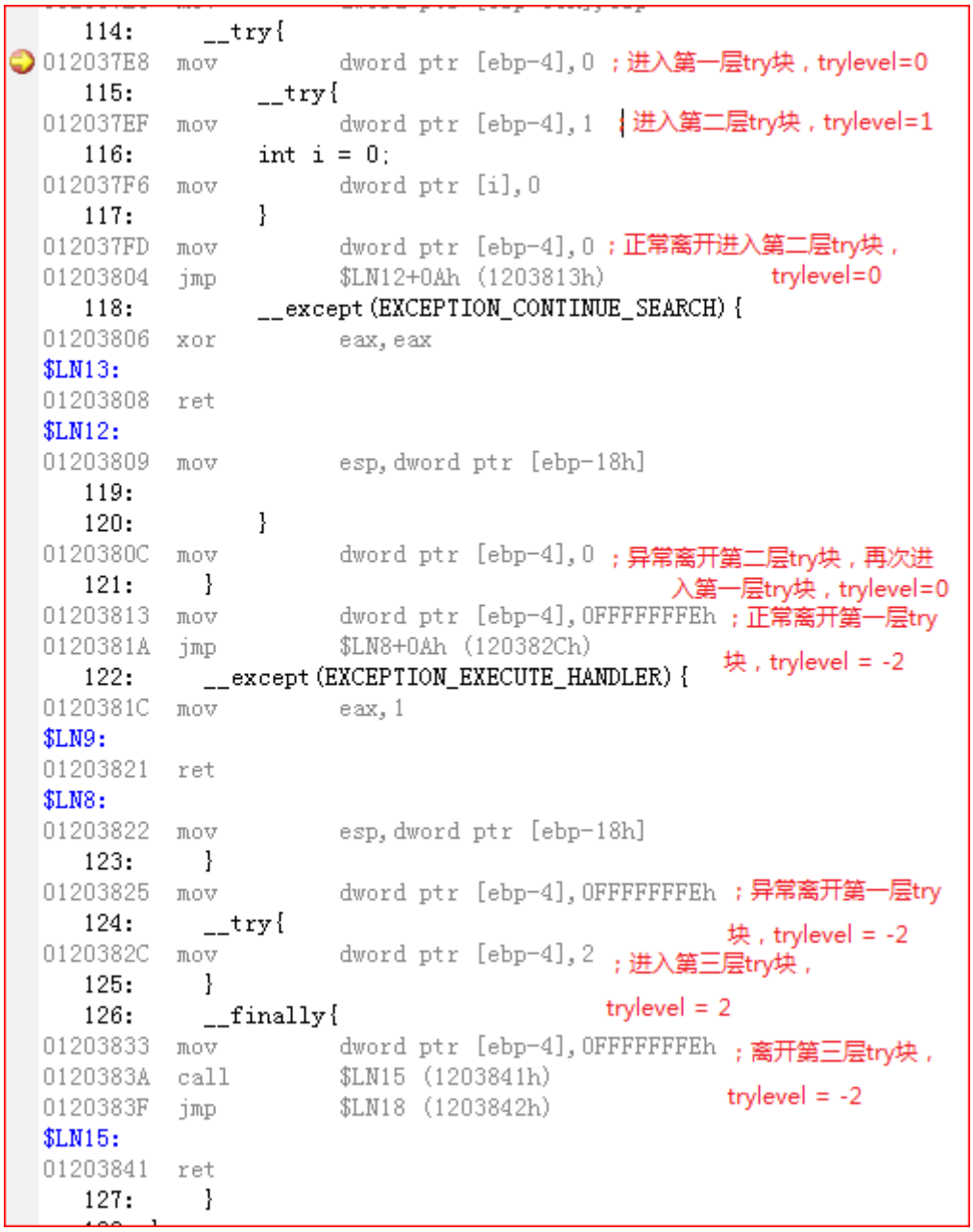


图 5-4 try 块随代码的变化情况

从图 5-4，我们可以清晰的看出，编译器将 try 块按照出现的顺序编号。起始号码为 0。程序执行过程中， trylevel 保存当前所在块的编码。不在任何块中，则编码为 -2 。

下面我们看一下 scopetable 中的内容。从图 5-3 中找到 scopetable 的地址 1206b30h，用内存窗体查看，结果如图 5-5 所示。

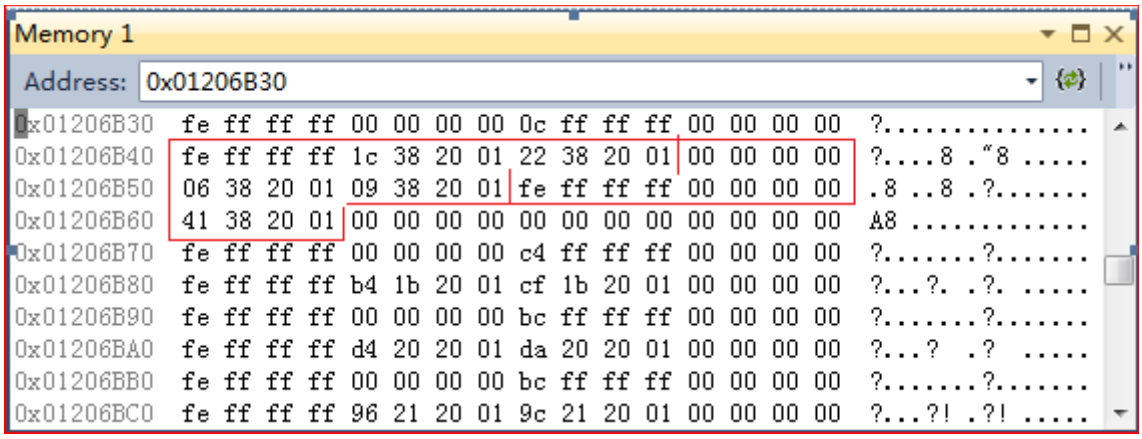


图 5-5 用内存窗体查看 scopetable 内容

从图 5-5 我们可以看出，实际上异常帧中的 scopetable 指针并不直接指向 scopetable 数组。

它们之间有 16 个字节的间隙。查阅资料，得知这可能是留给后面做安全验证用的。这里我们暂且忽略。查看 scopetable 数组的内容，第一个元素对应第一个 try 块其成员分别是 previousTrylevel=-2，表示外层没有 try 块。lpfnFilter=0x0120381c，表示该块对应的过滤表达式的首地址是 0x0120381c。lpfnHandler=0x01203822，表示该块对应的异常处理代码的首地址是 0x01203822。第二个元素对应第二个 try 块其成员分别是 previousTrylevel=0，表示外层是 0 号 try 块。lpfnFilter=0x01203806，表示该块对应的过滤表达式的首地址是 0x01203806。lpfnHandler=0x01203809，表示该块对应的异常处理代码的首地址是 0x01203809。第三个元素对应第三个 try 块其成员分别是 previousTrylevel=-2，表示外层没有 try 块。lpfnFilter=0x000000，表示该块对应 __finally 块。lpfnHandler=0x01203841，表示该块对应的终止处理代码的首地址是 0x01203809。

至此，我们了解到，编译器级的 SHE 通过在异常帧中扩展了 trylevel 和 scopetable 来打的定位和区别 try 块的目的。从而能够将多个 try 块封装到一个 _except_handler4 函数中。那么 _except_handler4 是如何决定何时使用哪块异常处理代码呢？

### 5.2.3 _except_handler4 的实现机制

下面我们给出 _except_handler4 的反汇编代码，如图 5-6 所示。

从图 5-6 可以看出，_except_handler4 实际上只是对 _except_handler4_common 进行了调用。从传的参数情况推测，_except_handler4_common 增加了安全验证的异常处理回调函数。限于文章篇幅，这里不再给出 _except_handler4_common 的汇编代码，仅给出通过汇编代码还原出的处理流程图，由于安全验证部分对我们来说不那么重要，因此这里我们省略对安全验证的分析。流程图如图 5-7 所示。

```
_except_handler4:
01101A90  mov         edi,edi
01101A92  push        ebp
01101A93  mov         ebp,esp
01101A95  mov         eax,dword ptr [DispatcherContext]
01101A98  push        eax
01101A99  mov         ecx,dword ptr [ContextRecord]
01101A9C  push        ecx
01101A9D  mov         edx,dword ptr [EstablisherFrame]
01101AA0  push        edx
01101AA1  mov         eax,dword ptr [ExceptionRecord]
01101AA4  push        eax
01101AA5  push        offset @ILT+15(@__security_check_cookie@4)
01101AAA  push        offset ___security_cookie (1107000h)
01101AAF  call        @ILT+420(__except_handler4_common) (11011A9
01101AB4  add         esp,18h
01101AB7  pop         ebp
01101AB8  ret
```

图 5-6 _except_handler4 的反汇编代码

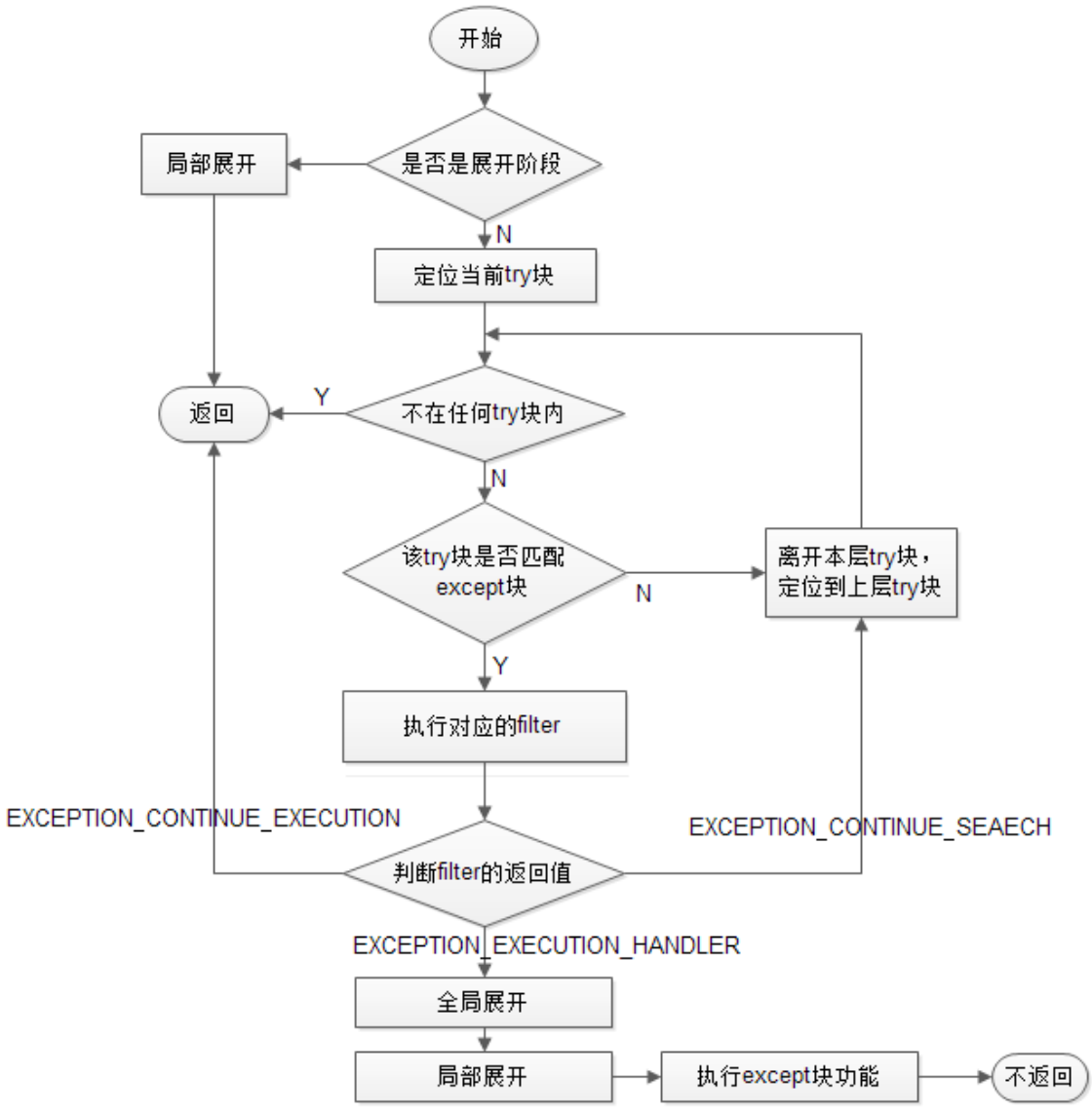


图 5-7 _except_handler4_common 的处理流程图

结合图 5-7，我们可以知道，当 _except_handler4_common 被调用，它会首先根据调用者传递的参数判断当前是处于展开阶段还是异常处理阶段。若是处在异常处理阶段，则它会依次执行 scopetable 中不为空的过滤表达式。若过滤表达式返回 EXCEPTION_CONTINUE_EXECUTION ，则退出异常处理函数，用户程序重新执行产生异常的指令。若返回值为 EXCEPTION_CONTINUE_SEARCH ，则继续搜索下一个 scopetable 中的元素。若返回值为 EXCEPTION_EXECUTION_HANDLER ，则表明该块对应的异常处理代码愿意处理该异常。但是在该异常处理前要先全局展开和局部展开，以保证内层的终止处理代码执行。当执行完 except 块中的代码后，不返回。顺序执行 except 块后的指令。

我们注意到，当过滤表达式的值返回 EXCEPTION_EXECUTION_HANDLER 时，会发生全局展开。全局展开通过再次遍历异常链表，依次执行 except 块内层的 __finally ，以保证终止处理代码永远被执行。由于全局展开的过程较为复杂，这里不在进一步深入。

### 5.2.4 顶级异常处理

我们在调程序的时候经常会看到这样的现象：我们的程序产生异常，而我们并没有对这个异常做任何处理。这个时候我们会看到如图 5-8 所示的对话框。

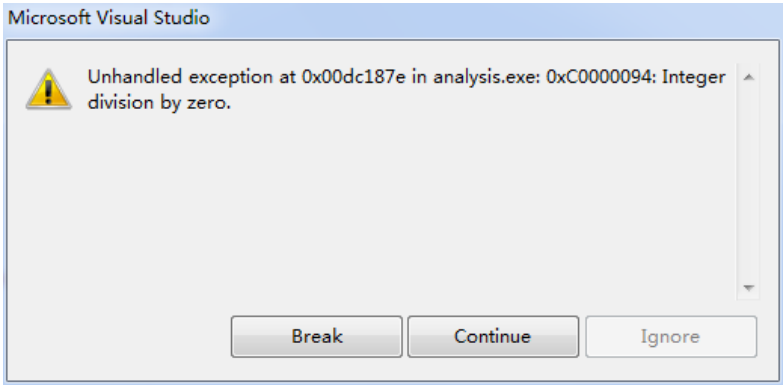


图 5-9 未异常提示

显然，这个提示不是我们添加的。它会在用户程序没有办法处理异常的时候发生。是异常的最后一道防线。我们叫它顶级异常处理。那操作系统是什么时候把这个段代码插进去的呢。我们调试运行一个程序，然后查看调用栈，如图 5-10 所示。

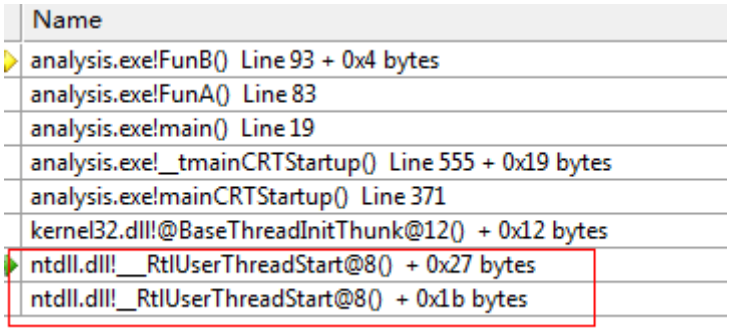


图 5-10 函数的调用栈

我们看到，我们编写的程序总是以 ntdll!__RtUserTheadStart 函数开始，那么是不是在这里插入的顶级异常处理代码呢？让我们来看下它的汇编代码吧。如图 5-11 所示。

```
___RtlUserThreadStart@8:
77A53754  push        14h
77A53756  push        77A41208h
77A5375B  call        __SEH_prolog4 (77A42B9Ch)
77A53760  and         dword ptr [ebp-4],0
77A53764  mov         eax,dword ptr [_Kernel32ThreadInitThunkFunction
77A53769  push        dword ptr [ebp+0Ch]
77A5376C  test        eax,eax
77A5376E  je          ___RtlUserThreadStart@8+25h (779F5E77h)
77A53774  mov         edx,dword ptr [ebp+8]
77A53777  xor         ecx,ecx
77A53779  call        eax
77A5377B  mov         dword ptr [ebp-4],0FFFFFFFEh
77A53782  call        __SEH_epilog4 (77A42BE1h)
77A53787  ret         8
77A5378A  call        _LdrpImageHasTls@0 (77A48976h)
77A5378F  test        eax,eax
77A53791  jne         _LdrpInitializeThread@4+186h (77A04CC8h)
77A53797  mov         dword ptr [ebp-4],0FFFFFFFEh
77A5379E  call        _LdrpInitializeThread@4+1EBh (77A53862h)
77A537A3  call        __SEH_epilog4 (77A42BE1h)
77A537A8  ret         4
```

图 5-11 __RtUserTheadStart 汇编代码

正如我们所猜测的那样。 __RtUserTheadStart 函数在刚开始的时候调用了 __SEH_prolog4 函数。而 __SEH_prolog4 的功能即是在栈中插入了顶级异常处理的异常帧。下面我们给出 __RtUserTheadStart 的伪代码。

__RtUserTheadStart 的伪代码

```
__RtUserTheadStart( PVOID lpfnEntryPoint )
{
      _try
      {
          NtSetInformationThread(
                        GetCurrentThread(),
                         ThreadQuerySetWin32StartAddress,
                         &lpfnEntryPoint,
                       sizeof(lpfnEntryPoint) );
        retValue = lpfnEntryPoint();
        ExitThread( retValue );
    }
    _except(EXCEPTION_EXECUTION_HANDLER)
      {
        if ( !_BaseRunningInServerProcess )
           ExitProcess( exceptionCode );
```

```
        else
                ExitThread( exceptionCode );
      }
}
```

从 __RtUserTheadStart 的伪代码中我们可以看出，该函数完成的功能即是将用户代码封装到一个异常处理块中。而这个异常处理块就是我们所说的顶级异常处理。

# 第六章 总结与展望

本文，我们先是以提高供程序健壮性和容错性的需求为主线讲述了结构化异常处理的作用和意义。从第三章开始，我们介绍了结构化异常理知识体系中最简单的一个概念，终止处理。并通过反汇编的方式给出了终止处理的实现机制。进一步深入，我们在第四章详细的介绍了操作系统提供的结构化异常处理的框架，数据结构，还绘制了异常处理链的结构图。第五章，也是最重要的一章，我们介绍了 MSVS 是如何支持操作系统的结构化异常处理。并通过构建示例，利用反汇编的方法实证了扩展异常帧中的某些数据结构的作用。除此之外，我们还分析了编译器的异常处理回调函数，并绘制了它的流程图。文章的最后，我们又提到了顶级异常处理，并解释了它的实现机制。

虽然，本文涉及了操作系统级的异常处理和编译器级的异常处理。但是，我们所介绍的知识点和深度还远远不够。你可能注意到，文中我们多次用到了“忽略”一次。因为，结构化异常处理是一个相当复杂的知识体系，为了高效快速的的分析 SHE ，我们只能暂时忽略那些复杂或者繁琐的知识点。在后续的研究中，我们将重新拾起那些被我们“忽略”的知识点。从细节入手，深入理解结构化异常处理体系。

# 参考文献

[1] Jeffrey Richter. Windows 核心编程（第五版）. 北京：清华大学出版社，2008.9
[2] Matt Pietrek. A Crash Course on the Depths of Win32? Structured Exception Handling.
[3] boxcounter.SHE 分析笔记（x86 篇）. http://www.boxcounter.com/，2011.10
[4] Matt Pietrek. Win32 结构化异常处理（SEH）探秘. http://www.vckbase.com/，2009.2
[5] yuzl. 深入研究 Win32 结构化异常处理（总结）. CSDN: yuzl 的专栏，2010.4