

深入探索 Win32 结构化异常处理

在 Win32 操作系统提供的所有功能中，使用最广泛而又没有公开的恐怕要数结构化异常处理（Structured Exception Handling，SEH）了。当你考虑 Win32 结构化异常处理时，也许会想到 `__try`、`__finally` 和 `__except` 等术语。可能你在任何一本讲解 Win32 的好书上都能找到关于 SEH 较为详细的描述，甚至 Win32 SDK 文档也对使用 `__try`、`__finally` 和 `__except` 进行结构化异常处理进行了相当完整的描述。

既然已经有了这些文档，那为什么我还说 SEH 并未公开呢？本质上来说，Win32 结构化异常处理是操作系统提供的服务。你可能找到的所有关于 SEH 方面的文档都只是描述了某个特别的编译器的运行时库对操作系统实现的封装。关键字 `__try`、`__finally` 或者 `__except` 并没有什么神奇的。Microsoft 的操作系统和编译器开发小组定义了这些关键字和它们的作用。其它 C++ 编译器厂商完全按照它们的语义来就可以了。当编译器的 SEH 支持层把原始的操作系统 SEH 的复杂性封装起来的时候，它同时也把原始的操作系统 SEH 的细节隐藏了起来。

我曾经接到大量来自想自己实现编译器层面 SEH 的人发来的电子邮件，他们苦于找不到关于操作系统 SEH 实现方面的任何文档。按说，我应该能够告诉他们 Visual C++ 或 Borland C++ 的运行时库源代码就是他们想要的。但是不知出于什么原因，编译器层面的 SEH 看起来好像是个大秘密。无论是 Microsoft 还是 Borland 都没有提供他们的 SEH 支持层最底层的源代码。（现在 Microsoft 仍然没有提供这些源代码，它提供的是编译过的目标文件，而 Borland 则提供了相应的源代码。）

在本文中，我会剥掉结构化异常处理外面的包装直至其最基本的概念。在此过程中，我会把操作系统提供的支持与编译器通过代码生成和运行时库提供的支持分开来说。当我挖掘到关键的操作系统例程时，我使用的是运行于 Intel 处理器上的 Windows NT 4.0。但是我这里讲的大部分内容同样也适用于其它处理器。

我会避免涉及到真实的 C++ 异常处理，它使用的是 `catch()` 而不是 `__except`。从内部来讲，真实的 C++ 异常处理的实现与我这里要讲的非常相似。但是真实的 C++ 异常处理有一些其它的复杂问题，它会混淆我这里要讲的一些概念。

在挖掘组成 Win32 SEH 的晦涩的 .H 和 .INC 文件的过程中，我发现最好的信息来源之一是 IBM OS/2 头文件（特别是 BSEXCPT.H）。如果你涉足这方面已经有一段时间了，就不会感到太奇怪。这里描述的 SEH 机制是早在 Microsoft 还工作在 OS/2 上时就已经定义好的。由于这个原因，你会发现 Win32 下的 SEH 和 OS/2 下的 SEH 极其相似。（现在我们可能已经没有机会体验这一点了，OS/2 已经永远成为历史了。）

浅析 SEH

如果我把 SEH 的所有细节一股脑儿全倒给你，你可能无法接受。因此我先从小部分开始，然后层层深入。如果你以前从未接触过结构化异常处理，那正好，因为你头脑中没有一些自己设想的概念。如果你以前接触过 SEH，最好把头脑中有关 `__try`、`GetExceptionCode` 和 `EXCEPTION_EXECUTE_HANDLER` 之类的词统统忘掉，假设它对你来说是全新的。深呼吸。准备好了吗？让我们开始吧！

设想我告诉过你，当一个线程出现错误时，操作系统给你一个机会被告知这个错误。说得更明白一些就是，**当一个线程出现错误时，操作系统调用用户定义的一个回调函数**。这个回调函数可以做它想做的一切。例如它可以修复错误，或者它也可以播放一段音乐。无论回调函数做什么，它最后都要返回一个值来告诉系统下一步做什么。（这不是十分准确，但就此刻来说非常接近。）

当你的某一部分代码出错时，系统再回调你的其它代码，那么这个回调函数看起来是什么样子呢？换句话说，你想知道关于异常什么类型的信息呢？实际上这并不重要，因为 Win32 已经替你做了决定。异常回调函数看起来像下面这个样子：

EXCEPTION_DISPOSITION

```
__cdecl _except_handler( struct _EXCEPTION_RECORD *ExceptionRecord,  
                        void * EstablisherFrame,  
                        struct _CONTEXT *ContextRecord,  
                        void * DispatcherContext);
```

这个原型来自标准的 Win32 头文件 EXCPT.H，乍看起来有些费解。但如果你仔细看，它并不是很难理解。首先，忽略掉返回值的类型（EXCEPTION_DISPOSITION）。你得到的基本信息就是它是一个叫作 **_except_handler** 并且带有四个参数的函数。

这个函数的第一个参数是一个指向 **EXCEPTION_RECORD** 结构的指针。这个结构在 WINNT.H 中定义，如下所示：

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

这个结构中的 ExceptionCode 成员是赋予异常的代码。通过在 WINNT.H 中搜索以“STATUS_”开头的 #define 定义，你可以得到一个异常代码列表。例如所有人都非常熟悉的 STATUS_ACCESS_VIOLATION 的代码是 0xC0000005。一个更全面的异常代码列表可以在 Windows NT DDK 的 NTSTATUS.H 中找到。此结构的第四个成员是异常发生的地址。其它成员暂时可以忽略。

_except_handler 函数的第二个参数是一个指向 establisher 帧结构的指针。它是 SEH 中一个至关重要的参数，但是现在你可以忽略它。

_except_handler 回调函数的第三个参数是一个指向 **CONTEXT** 结构的指针。此结构在 WINNT.H 中定义，它代表某个特定线程的寄存器值。图 1 显示了 CONTEXT 结构的成员。当用于 SEH 时，CONTEXT 结构表示异常发生时寄存器的值。顺便说一下，这个 CONTEXT 结构就是 GetThreadContext 和 SetThreadContext 这两个 API 中使用的那个 CONTEXT 结构。

图 1 CONTEXT 结构

```
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;
DWORD EFlags;
DWORD Esp;
DWORD SegSs;
} CONTEXT;

```

`_except_handler` 回调函数的第四个参数被称为 `DispatcherContext`。它暂时也可以被忽略。

到现在为止，你头脑中已经有了一个当异常发生时会被操作系统调用的回调函数的模型了。这个回调函数带四个参数，其中三个指向其它结构。在这些结构中，一些域比较重要，其它的不那么重要。这里的关键是 **`_except_handler` 回调函数接收到操作系统传递过来的许多有价值的信息**，例如异常的类型和发生的地址。使用这些信息，异常回调函数就能决定下一步做什么。

对我来说，现在就写一个能够显示 `_except_handler` 作用的样例程序是再诱人不过的了。但是我们还缺少一些关键信息。特别是，当错误发生时操作系统是怎么知道到哪里去调用这个回调函数的呢？答案是还有一个称为 `EXCEPTION_REGISTRATION` 的结构。通篇你都会看到这个结构，所以不要跳过这一部分。我唯一能找到的 **`EXCEPTION_REGISTRATION`** 结构的正式定义是在 Visual C++ 运行时库源代码中的 `EXSUP.INC` 文件中：

```

_EXCEPTION_REGISTRATION struc
    prev          dd    ?
    handler        dd    ?
_EXCEPTION_REGISTRATION ends

```

这个结构在 `WINNT.H` 的 `NT_TIB` 结构的定义中被称为 `_EXCEPTION_REGISTRATION_RECORD`。唉，没有一个地方能够找到 `_EXCEPTION_REGISTRATION_RECORD` 的定义，所以我不得不使用 `EXSUP.INC` 中这个汇编语言的结构定义。这是我前面所说 `SEH` 未公开的一个证据。（读者可以使用内核调试器，如 `KD` 或 `SoftICE` 并加载调试符号来查看这个结构的定义。

下图是在 `KD` 中的结果：

```

lkd> dt _EXCEPTION_REGISTRATION_RECORD
+0x0000 Next          : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x0004 Handler        : Ptr32
lkd> _

```

下图是在 SoftICE 中的结果:

```

:types _EXCEPTION_REGISTRATION_RECORD
typedef struct _EXCEPTION_REGISTRATION_RECORD {
:
:   0x0      struct _EXCEPTION_REGISTRATION_RECORD * Next ;
:   0x4      enum _EXCEPTION_DISPOSITION stdcall Handler ( struct _EXCEPTION_RECORD **, void **, st
: } ;
:
:

```

译者注)

无论正在干什么, 现在让我们回到手头的问题上来。当异常发生时, 操作系统是如何知道到哪里去调用回调函数的呢? 实际上, `EXCEPTION_REGISTRATION` 结构由两个域组成, 第一个你现在可以忽略。第二个域 `handler`, 包含一个指向 `_except_handler` 回调函数的指针。这让你离答案更近一点, 但现在的问题是, 操作系统到哪里去找 `EXCEPTION_REGISTRATION` 结构呢?

要回答这个问题, 记住 **结构化异常处理是基于线程的** 这一点是非常有用的。也就是说, 每个线程有它自己的异常处理回调函数。在 1996 年五月的 *Under The Hood* 专栏中, 我介绍了一个关键的 Win32 数据结构——**线程信息块 (Thread Information/Environment Block, TIB 或 TEB)**。这个结构的某些域在 Windows NT、Windows 95、Win32s 和 OS/2 上是相同的。**TIB 的第一个 DWORD 是一个指向线程的 `EXCEPTION_REGISTRATION` 结构的指针。在基于 Intel 处理器的 Win32 平台上, FS 寄存器总是指向当前的 TIB。因此在 FS:[0] 处你可以找到一个指向 `EXCEPTION_REGISTRATION` 结构的指针。**

到现在为止, 我们已经有了足够的认识。当异常发生时, 系统查找出错线程的 TIB, 获取一个指向 `EXCEPTION_REGISTRATION` 结构的指针。在这个结构中有一个指向 `_except_handler` 回调函数的指针。现在操作系统已经知道了足够的信息去调用 `_except_handler` 函数, 如图 2 所示。

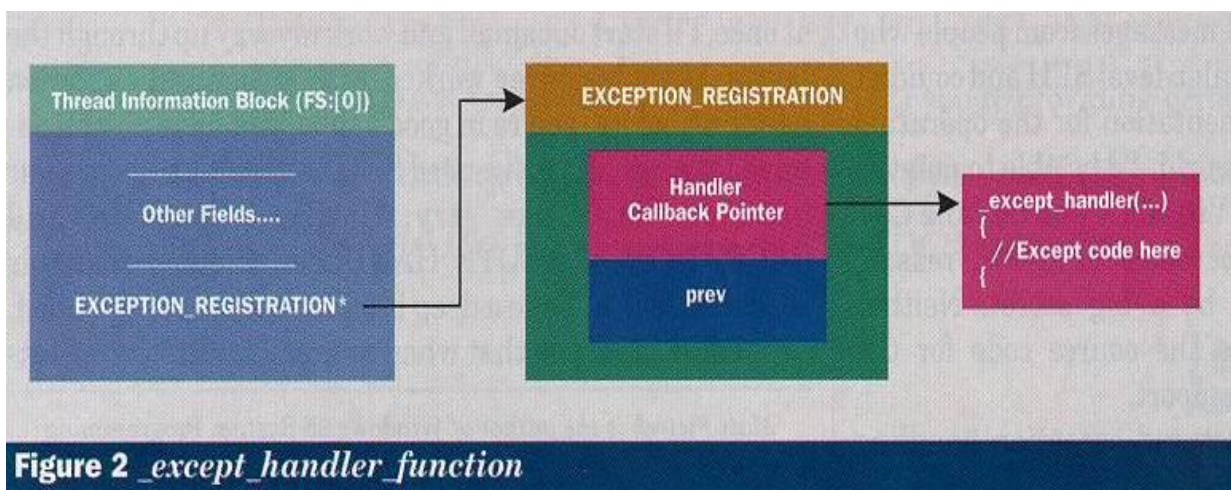


图 2 _except_handler 函数

把这些小块知识拼凑起来, 我写了一个小程序来演示上面这个对操作系统层面的结构化异常处理的简化描述, 如图 3 的 MYSEH.CPP 所示。它只有两个函数。 `main` 函数使用了三个内联汇编块。第一个内联汇编块通过两个 `PUSH` 指令 (“`PUSH handler`” 和 “`PUSH FS:[0]`”) 在堆栈上创建了一个 `EXCEPTION_REGISTRATION` 结构。 `PUSH FS:[0]` 这条指令保存了先前的 `FS:[0]` 中的值作为这个结构的一部分,

但这在此刻并不重要。重要的是现在堆栈上有一个 8 字节的 EXCEPTION_REGISTRATION 结构。紧接着的下一条指令（MOV FS:[0],ESP）使线程信息块中的第一个 DWORD 指向了新的 EXCEPTION_REGISTRATION 结构。（注意堆栈操作）

图 3 MYSEH.CPP

```
//=====
// MYSEH - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH.CPP
// 用命令行 CL MYSEH.CPP 编译
//=====

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

DWORD scratch;
EXCEPTION_DISPOSITION
__cdecl _except_handler( struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    unsigned i;
    // 指明是我们让流程转到我们的异常处理程序的
    printf( "Hello from an exception handler/n" );
    // 改变 CONTEXT 结构中 EAX 的值，以便它指向可以成功进写操作的位置
    ContextRecord->Eax = (DWORD)&scratch;
    // 告诉操作系统重新执行出错的指令
    return ExceptionContinueExecution;
}

int main()
{
    DWORD handler = (DWORD)_except_handler;
    __asm
    {
        // 创建 EXCEPTION_REGISTRATION 结构：
        push handler // handler 函数的地址
        push FS:[0] // 前一个 handler 函数的地址
        mov FS:[0],ESP // 安装新的 EXECEPTION_REGISTRATION 结构
    }
}
```

```

__asm
{
    mov eax,0 // 将 EAX 清零
    mov [eax], 1 // 写 EAX 指向的内存从而故意引发一个错误
}
printf( "After writing!/n" );
__asm
{
    // 移去我们的 EXCEPTION_REGISTRATION 结构
    mov eax,[ESP] // 获取前一个结构
    mov FS:[0], EAX // 安装前一个结构
    add esp, 8 // 将我们的 EXCEPTION_REGISTRATION 弹出堆栈
}
return 0;
}

```

如果你想知道我为什么把 `EXCEPTION_REGISTRATION` 结构创建在堆栈上而不是使用全局变量，我有一个很好的理由可以解释它。实际上，当你使用编译器的 `__try/__except` 语法结构时，编译器自己也把 `EXCEPTION_REGISTRATION` 结构创建在堆栈上。我只是简单地向你展示了如果使用 `__try/__except` 时编译器做法的简化版。

回到 `main` 函数，第二个 `__asm` 块通过先把 `EAX` 寄存器清零（`MOV EAX,0`）然后把此寄存器的值作为内存地址让下一条指令（`MOV [EAX],1`）向此地址写入数据而故意引发一个错误。最后的 `__asm` 块移除这个简单的异常处理程序：它首先恢复了 `FS:[0]` 中先前的内容，然后把 `EXCEPTION_REGISTRATION` 结构弹出堆栈（`ADD ESP,8`）。

现在假若你运行 `MYSEH.EXE`，就会看到整个过程。当 `MOV [EAX],1` 这条指令执行时，它引发一个访问违规。系统在 `FS:[0]` 处的 `TIB` 中查找，然后发现了一个指向 `EXCEPTION_REGISTRATION` 结构的指针。在 `MYSEH.CPP` 中，在这个结构中有一个指向 `_except_handler` 函数的指针。系统然后把所需的四个参数（我在前面已经说过）压入堆栈，接着调用 `_except_handler` 函数。

一旦进入 `_except_handler`，这段代码首先通过一个 `printf` 语句表明“哈!是我让它转到这里的!”。接着，`_except_handler` 修复了引发错误的问题——即 `EAX` 寄存器指向了一个不能写的内存地址（地址 0）。修复方法就是改变 `CONTEXT` 结构中的 `EAX` 的值使它指向一个允许写的位置。在这个简单的程序中，我专门为此设置了一个 `DWORD` 变量（`scratch`）。`_except_handler` 函数最后的动作是返回 `ExceptionContinueExecution` 这个值，它在 `EXCPT.H` 文件中定义。

当操作系统看到返回值为 `ExceptionContinueExecution` 时，它将其理解为你已经修复了问题，而引起错误的那条指令应该被重新执行。由于我的 `_except_handler` 函数已经让 `EAX` 寄存器指向一个合法的内存，`MOV [EAX],1` 指令再次执行，这次 `main` 函数一切正常。看，这也并不复杂，不是吗？

移向更深处

有了这个最简单的情景之后，让我们回去填补那些空白。虽然这个异常回调机制很好，但它并不是一个完美的解决方案。对于稍微复杂一些的应用程序来说，仅用一个函数就能处理程序中任何地方都可能发

生的异常是相当困难的。一个更实用的方案应该是有多个异常处理例程，每个例程针对程序中的一部分。实际上，操作系统提供的正是这个功能。

还记得系统用来查找异常回调函数的 `EXCEPTION_REGISTRATION` 结构吗？这个结构的第一个成员，称为 `prev`，前面我们暂时把它忽略了。它实际上是一个指向另外一个 `EXCEPTION_REGISTRATION` 结构的指针。这第二个 `EXCEPTION_REGISTRATION` 结构可以有一个完全不同的处理函数。它的 `prev` 域可以指向第三个 `EXCEPTION_REGISTRATION` 结构，依次类推。简单地说，就是有一个 `EXCEPTION_REGISTRATION` 结构链表。线程信息块的第一个 `DWORD`（在基于 Intel CPU 的机器上是 `FS:[0]`）指向这个链表的头部。

操作系统要这个 `EXCEPTION_REGISTRATION` 结构链表做什么呢？原来，当异常发生时，系统遍历这个链表以查找一个（其异常处理程序）同意处理这个异常的 `EXCEPTION_REGISTRATION` 结构。在 `MYSEH.CPP` 中，异常处理程序通过返回 `ExceptionContinueExecution` 表示它同意处理这个异常。异常回调函数也可以拒绝处理这个异常。在这种情况下，系统移向链表的下一个 `EXCEPTION_REGISTRATION` 结构并询问它的异常回调函数，看它是否同意处理这个异常。图 4 显示了这个过程。一旦系统找到一个处理这个异常的回调函数，它就停止遍历链表。

Initial State of Exception Handling Chain

Operating System's Course of Action

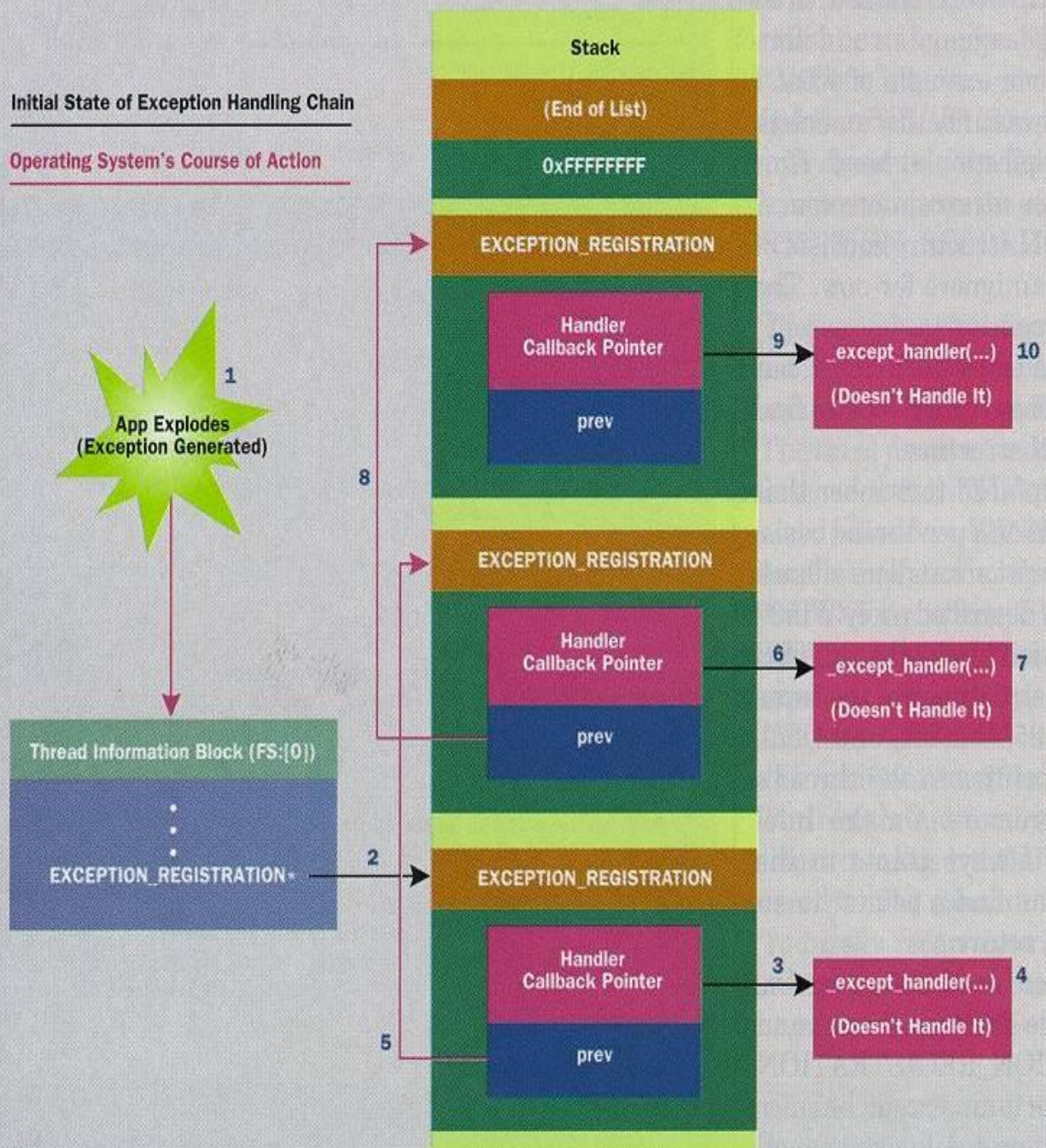


Figure 4 Finding a Structure to Handle the Exception

图 4 查找一个处理异常的 EXCEPTION_REGISTRATION 结构

图 5 的 MYSEH2.CPP 就是一个异常处理函数不处理某个异常的例子。为了使代码尽量简单，我使用了编译器层面的异常处理。main 函数只设置了一个 __try/__except 块。在 __try 块内部调用了 HomeGrownFrame 函数。这个函数与前面的 MYSEH 程序非常相似。它也是在堆栈上创建一个 EXCEPTION_REGISTRATION 结构，并且让 FS:[0]指向此结构。在建立了新的异常处理程序之后，这个函数通过向一个 NULL 指针所指向的内存处写入数据而故意引发一个错误：

```
*(PDWORD)0 = 0;
```

这个异常处理回调函数，同样被称为 _except_handler，却与前面的那个截然不同。它首先打印出 ExceptionRecord 结构中的异常代码和标志，这个结构的地址是作为一个指针参数被这个函数接收的。打印出异常标志的原因一会儿就清楚了。因为 _except_handler 函数并没有打算修复出错的代码，因此它返回 ExceptionContinueSearch。这导致操作系统继续在 EXCEPTION_REGISTRATION 结构链表中搜索下一个 EXCEPTION_REGISTRATION 结构。接下来安装的异常回调函数是针对 main 函数中的 __try/__except 块的。__except 块简单地打印出 “Caught the exception in main()”。此时我们只是简单地忽略这个异常来表明我们已经处理了它。

图 5 MYSEH2.CPP

```
//=====
// MYSEH2 - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH2.CPP
// 使用命令行 CL MYSEH2.CPP 编译
//=====

#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    printf( "Home Grown handler: Exception Code: %08X Exception Flags %X",
        ExceptionRecord->ExceptionCode, ExceptionRecord->ExceptionFlags );
    if ( ExceptionRecord->ExceptionFlags & 1 )
        printf( " EH_NONCONTINUABLE" );
    if ( ExceptionRecord->ExceptionFlags & 2 )
        printf( " EH_UNWINDING" );
    if ( ExceptionRecord->ExceptionFlags & 4 )
```

```

        printf( " EH_EXIT_UNWIND" );
    if ( ExceptionRecord->ExceptionFlags & 8 ) // 注意这个标志
        printf( " EH_STACK_INVALID" );
    if ( ExceptionRecord->ExceptionFlags & 0x10 ) // 注意这个标志
        printf( " EH_NESTED_CALL" );
    printf( "/n" );
    // 我们不想处理这个异常，让其它函数处理吧
    return ExceptionContinueSearch;
}

void HomeGrownFrame( void )
{
    DWORD handler = (DWORD)_except_handler;

    __asm
    {
        // 创建 EXCEPTION_REGISTRATION 结构:
        push handler    // handler 函数的地址
        push FS:[0]     // 前一个 handler 函数的地址
        mov FS:[0],ESP  // 安装新的 EXCEPTION_REGISTRATION 结构
    }

    *(PDWORD)0 = 0; // 写入地址 0，从而引发一个错误
    printf( "I should never get here!/n" );

    __asm
    {
        // 移去我们的 EXCEPTION_REGISTRATION 结构
        mov eax,[ESP]   // 获取前一个结构
        mov FS:[0], EAX // 安装前一个结构
        add esp, 8      // 把我们 EXCEPTION_REGISTRATION 结构弹出堆栈
    }
}

int main()
{
    __try
    {
        HomeGrownFrame();
    }
    __except( EXCEPTION_EXECUTE_HANDLER )
    {
        printf( "Caught the exception in main()/n" );
    }
}

```

```
}  
return 0;  
}
```

这里的关键是执行流程。当一个异常处理程序拒绝处理某个异常时，它实际上也就拒绝决定流程最终将从何处恢复。只有处理某个异常的异常处理程序才能决定待所有异常处理代码执行完毕之后流程将从何处恢复。这个规则的意义非常重大，虽然现在还不明显。

当使用结构化异常处理时，如果一个函数有一个异常处理程序但它却不处理某个异常，这个函数就有可能非正常退出。例如在 MYSEH2 中 HomeGrownFrame 函数就不处理异常。由于在链表中后面的某个异常处理程序（这里是 main 函数中的）处理了这个异常，因此出错指令后面的 printf 就永远不会执行。从某种程度上说，使用结构化异常处理与使用 setjmp 和 longjmp 运行时库函数有些类似。

如果你运行 MYSEH2，会发现其输出有些奇怪。看起来好像调用了两次 _except_handler 函数。根据你现有的知识，第一次调用当然可以完全理解。但是为什么会有第二次呢？

Home Grown handler: Exception Code: C0000005 Exception Flags 0

Home Grown handler: Exception Code: C0000027 Exception Flags 2 EH_UNWINDING

Caught the Exception in main()

比较一下以“Home Grown Handler”开头的两行，就会看出它们之间有明显的区别。第一次异常标志是 0，而第二次是 2。这把我们带入了展开（Unwinding）的世界中。实际上，当一个异常处理回调函数拒绝处理某个异常时，它会被再一次调用。但是这次回调并不是立即发生的。这有点复杂。我需要把异常发生时的情形好好梳理一下。

当异常发生时，系统遍历 EXCEPTION_REGISTRATION 结构链表，直到它找到一个处理这个异常的异常处理程序。一旦找到，系统就再次遍历这个链表，直到处理这个异常的结点为止。在这第二次遍历中，系统将再次调用每个异常处理函数。关键的区别是，在第二次调用中，异常标志被设置为 2。这个值被定义为 EH_UNWINDING。（EH_UNWINDING 的定义在 Visual C++ 运行时库源代码文件 EXCEPT.INC 中，但 Win32 SDK 中并没有与之等价的定义。）

EH_UNWINDING 表示什么意思呢？原来，当一个异常处理回调函数被第二次调用时（带 EH_UNWINDING 标志），操作系统给这个函数一个最后清理的机会。什么样的清理呢？一个绝好的例子是 C++ 类的析构函数。当一个函数的异常处理程序拒绝处理某个异常时，通常执行流程并不会正常地从那个函数退出。现在，想像一个定义了一个 C++ 类的实例作为局部变量的函数。C++ 规范规定析构函数必须被调用。这带 EH_UNWINDING 标志的第二次回调就给这个函数一个机会去做一些类似于调用析构函数和 __finally 块之类的清理工作。

在异常已经被处理完毕，并且所有前面的异常帧都已经被展开之后，流程从处理异常的那个回调函数决定的地方开始继续执行。一定要记住，仅仅把指令指针设置到所需的代码处就开始执行是不行的。流程恢复执行处的代码的堆栈指针和栈帧指针（在 Intel CPU 上是 ESP 和 EBP）也必须被恢复成它们在处理这个异常的函数的栈帧上的值。因此，这个处理异常的回调函数必须负责把堆栈指针和栈帧指针恢复成它们在包含处理这个异常的 SEH 代码的函数的堆栈上的值。

通常，展开操作导致堆栈上处理异常的帧以下的堆栈区域上的所有内容都被移除了，就好像我们从来没有调用过这些函数一样。展开的另外一个效果就是 EXCEPTION_REGISTRATION 结构链表上处理异常的那个结构之前的所有 EXCEPTION_REGISTRATION 结构都被移除了。这很好理解，因为这些

EXCEPTION_REGISTRATION 结构通常都被创建在堆栈上。在异常被处理后，堆栈指针和栈帧指针在内存中比那些从 EXCEPTION_REGISTRATION 结构链表上移除的 EXCEPTION_REGISTRATION 结构高。图 6 显示了我说的情况。

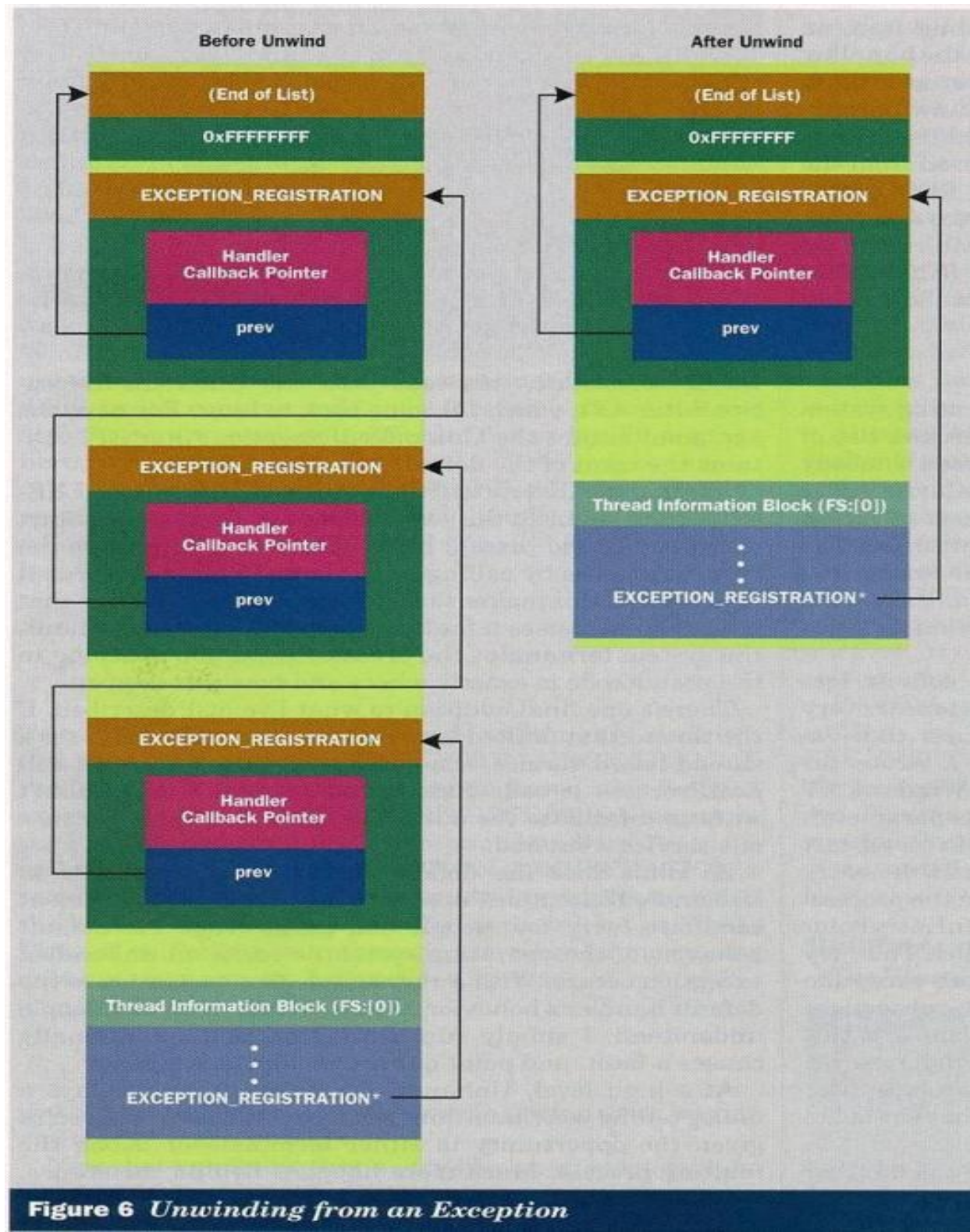


Figure 6 Unwinding from an Exception

图 6 从异常展开

救命！没有人处理它！

迄今为止，我实际上一直在假设操作系统总是能在 EXCEPTION_REGISTRATION 结构链表中找到一个异常处理程序。如果找不到怎么办呢？实际上，这几乎不可能发生。因为操作系统暗中已经为每个线程都提供了一个默认的异常处理程序。这个默认的异常处理程序总是链表的最后一个结点，并且它总是选择处理异常。它进行的操作与其它正常的异常处理回调函数有些不同，下面我会说明。

让我们来看一下系统是在什么时候插入了这个默认的、最后一个异常处理程序。很明显它需要在线程执行的早期，在任何用户代码开始执行之前。图 7 是我为 BaseProcessStart 函数写的伪代码，它是 Windows NT KERNEL32.DLL 的一个内部例程。这个函数带一个参数——线程入口点函数的地址。BaseProcessStart 运行在新进程的环境中，并且它调用这个进程的第一个线程的入口点函数。

图 7 BaseProcessStart 伪代码

```
BaseProcessStart( PVOID lpfnEntryPoint )
{
    DWORD retValue;
    DWORD currentESP;
    DWORD exceptionCode;
    currentESP = ESP;
    __try
    {
        NtSetInformationThread( GetCurrentThread(),
                                ThreadQuerySetWin32StartAddress,
                                &lpfnEntryPoint,
                                sizeof(lpfnEntryPoint) );
        retValue = lpfnEntryPoint();
        ExitThread( retValue );
    }
    __except( // 过滤器表达式代码
              exceptionCode = GetExceptionInformation(),
              UnhandledExceptionFilter( GetExceptionInformation() ) )
    {
        ESP = currentESP;
        if ( !_BaseRunningInServerProcess ) // 普通进程
            ExitProcess( exceptionCode );
        else // 服务
            ExitThread( exceptionCode );
    }
}
```

在上面的伪代码中，注意对 lpfnEntryPoint 的调用被一个 __try 和 __except 块封装着。就是这个 __try 块安装了默认的、异常处理程序链表上的最后一个异常处理程序。所有后来注册的异常处理程序都被安装在链表中这个结点的前面。如果 lpfnEntryPoint 函数返回，那么表明线程一直运行到完成并且没有引发异常。

这时 `BaseProcessStart` 调用 `ExitThread` 使线程退出。

如果线程引发了一个异常但是没有异常处理程序来处理它时怎么办呢？这时，执行流程转到 `__except` 关键字后面的括号中。在 `BaseProcessStart` 中，这段代码调用 `UnhandledExceptionFilter` 这个 API，后面我会讲到它。现在对于我们来说，重要的是 `UnhandledExceptionFilter` 这个 API 包含了默认的异常处理程序。

如果 `UnhandledExceptionFilter` 返回 `EXCEPTION_EXECUTE_HANDLER`，这时 `BaseProcessStart` 中的 `__except` 块开始执行。而 `__except` 块所做的只是调用 `ExitProcess` 函数去终止当前进程。稍微想一下你就会理解了。常识告诉我们，如果一个进程引发了一个错误而没有异常处理程序去处理它，这个进程就会被系统终止。你在伪代码中看到的正是这些。

对于上面所说的我还有一点要补充。如果引发错误的线程是作为服务来运行的，并且是基于线程的服务，那么 `__except` 块并不调用 `ExitProcess`，相反，它调用 `ExitThread`。不能仅仅因为一个服务出错就终止整个服务进程。

`UnhandledExceptionFilter` 中的默认异常处理程序都做了什么呢？当我在一个技术讲座上问起这个问题时，响应者寥寥无几。几乎没有人知道当未处理异常发生时，到底操作系统的默认行为是什么。简单地演示一下这个默认的行为也许会让很多人豁然开朗。我运行一个故意引发错误的程序，其结果如下（见图 8）。



图 8 未处理异常对话框

表面上看，`UnhandledExceptionFilter` 显示了一个对话框告诉你发生了一个错误。这时，你被给予了一个机会或者终止出错进程，或者调试它。但是幕后发生了许多事情，我会在文章最后详细讲述它。

正如我让你看到的那样，当异常发生时，用户写的代码可以（并且通常是这样）获得机会执行。同样，在展开操作期间，用户写的代码也可以执行。这个用户写的代码可能也有错误，并且可能引发另一个异常。由于这个原因，异常处理回调函数也可以返回另外两个值：`ExceptionNestedException` 和 `ExceptionCollidedUnwind`。很明显，它们很重要。但这是非常复杂的问题，我并不打算在这里涉及它们。要想理解其中的一些基本问题太困难了。

编译器层面的 SEH

虽然我在前面偶尔也使用了 `__try` 和 `__except`，但迄今为止几乎我写的所有内容都是关于操作系统方面对 SEH 的实现。然而看一下我那两个使用操作系统的原始 SEH 的小程序别扭的样子，编译器对这个功能进行封装实在是非常有必要的。现在让我们来看一下 Visual C++ 是如何在操作系统对 SEH 功能实现的基础上来创建它自己的结构化异常处理支持的。

在我们继续下去之前，记住其它编译器可以使用原始的系统 SEH 来做一些完全不同的事情这一点是非常重要的。并没有什么规定编译器必须实现 Win32 SDK 文档中描述的 `__try/__except` 模型。例如 Visual Basic 5.0 在它的运行时代码中使用了结构化异常处理，但是那里的数据结构和算法与我这里要讲的完全不同。

如果你把 Win32 SDK 文档中关于结构化异常处理方面的内容从头到尾读一遍，一定会遇到下面所谓的

“基于帧”的异常处理程序模型：

```
__try {  
    // 这里是被保护的代码  
}  
__except (过滤器表达式) {  
    // 这里是异常处理程序代码  
}
```

简单地说，在一个函数中，一个 `__try` 块中的所有代码就通过创建在这个函数的堆栈帧上的一个 `EXCEPTION_REGISTRATION` 结构来保护。在函数的入口处，这个新的 `EXCEPTION_REGISTRATION` 结构被放在异常处理程序链表的头部。在 `__try` 块结束后，相应的 `EXCEPTION_REGISTRATION` 结构从这个链表的头部被移除。正如我前面所说，异常处理程序链表的头部被保存在 `FS:[0]` 处。因此，如果你在调试器中单步跟踪时看到类似下面的指令时

```
MOV DWORD PTR FS:[00000000],ESP
```

或者

```
MOV DWORD PTR FS:[00000000],ECX
```

就能非常确定这段代码正在进入或退出一个 `__try/__except` 块。

既然一个 `__try` 块相当于堆栈上的一个 `EXCEPTION_REGISTRATION` 结构，那么 `EXCEPTION_REGISTRATION` 结构中的回调函数相当于什么呢？使用 Win32 的术语来说，异常处理回调函数相当于 [过滤器表达式 \(filter-expression\)](#) 代码。实际上，过滤器表达式就是 `__except` 关键字后面的小括号中的代码。就是这个过滤器表达式代码决定了后面的大括号中的代码是否执行。

由于过滤器表达式代码是你自己写的，你当然可以决定在你的代码中的某个地方是否处理某个特定的异常。它可以简单的只是一句“`EXCEPTION_EXECUTE_HANDLER`”，也可以先调用一个把 π 计算到 20,000,000 位的函数，然后再返回一个值来告诉操作系统下一步做什么。随你的便。关键是你的过滤器表达式代码必须是我前面讲的有效异常处理回调函数。

我刚才讲的虽然相当简单，但那只不过是隔着有色玻璃看世界罢了。实际它是非常复杂的。首先，你的过滤器表达式代码并不是被操作系统直接调用的。事实上，各个 `EXCEPTION_REGISTRATION` 结构的 `handler` 域都指向了同一个函数。这个函数在 Visual C++ 的运行时库中，它被称为 `__except_handler3`。正是这个 `__except_handler3` 调用了你的过滤器表达式代码，我一会儿再接着说它。

对我前面的简单描述需要修正的另一个地方是，并不是每次进入或退出一个 `__try` 块时就创建或撤销一个 `EXCEPTION_REGISTRATION` 结构。相反，在使用 SEH 的任何函数中只创建一个 `EXCEPTION_REGISTRATION` 结构。换句话说，你可以在一个函数中使用多个 `__try/__except` 块，但是在堆栈上只创建一个 `EXCEPTION_REGISTRATION` 结构。同样，你可以在一个函数中嵌套使用 `__try` 块，但 Visual C++ 仍旧只是创建一个 `EXCEPTION_REGISTRATION` 结构。

如果整个 EXE 或 DLL 只需要单个的异常处理程序 (`__except_handler3`)，同时，如果单个的 `EXCEPTION_REGISTRATION` 结构就能处理多个 `__try` 块的话，很明显，这里面还有很多东西我们不知道。这个技巧是通过一个通常情况下看不到的表中的数据来完成的。由于本文的目的就是要深入探索结构化异常处理，那就让我们来看一看这些数据结构吧。

扩展的异常处理帧

Visual C++ 的 SEH 实现并没有使用原始的 EXCEPTION_REGISTRATION 结构。它在这个结构的末尾添加了一些附加数据。这些附加数据正是允许单个函数（__except_handler3）处理所有异常并将执行流程传递到相应的过滤器表达式和__except 块的关键。我在 Visual C++运行时库源代码中的 EXSUP.INC 文件中找到了有关 Visual C++扩展的 EXCEPTION_REGISTRATION 结构格式的线索。在这个文件中，你会看到以下定义（已经被注释掉了）：

```
;struct _EXCEPTION_REGISTRATION{
;   struct _EXCEPTION_REGISTRATION *prev;
;   void (*handler)(   PEXCEPTION_RECORD,
;                       PEXCEPTION_REGISTRATION,
;                       PCONTEXT,
;                       PEXCEPTION_RECORD);
;   struct scopetable_entry *scopetable;
;   int trylevel;
;   int _ebp;
;   PEXCEPTION_POINTERS xpointers;
;};
```

在前面你已经见过前两个域：prev 和 handler。它们组成了基本的 EXCEPTION_REGISTRATION 结构。后面三个域：scopetable（作用域表）、trylevel 和 _ebp 是新增加的。scopetable 域指向一个 scopetable_entry 结构 数组，而 trylevel 域实际上是这个数组的索引。最后一个域 _ebp，是 EXCEPTION_REGISTRATION 结构创建之前栈帧指针（EBP）的值。

_ebp 域成为扩展的 EXCEPTION_REGISTRATION 结构的一部分并非偶然。它是通过 PUSH EBP 这条指令被包含进这个结构中的，而大多数函数开头都是这条指令（通常编译器并不为使用 FPO 优化的函数生成标准的堆栈帧，这样其第一条指令可能不是 PUSH EBP。但是如果使用了 SEH 的话，那么无论你是否使用了 FPO 优化，编译器一定生成标准的堆栈帧）。这条指令可以使 EXCEPTION_REGISTRATION 结构中所有其它的域都可以用一个相对于栈帧指针（EBP）的负偏移来访问。例如 trylevel 域在[EBP-04]处，scopetable 指针在[EBP-08]处，等等。（也就是说，这个结构是从[EBP-10H]处开始的。）

紧跟着扩展的 EXCEPTION_REGISTRATION 结构下面，Visual C++压入了另外两个值。紧跟着（即[EBP-14H]处）的一个 DWORD，是为一个指向 EXCEPTION_POINTERS 结构（一个标准的 Win32 结构）的指针所保留的空间。这个指针就是你调用 GetExceptionInformation 这个 API 时返回的指针。尽管 SDK 文档暗示 GetExceptionInformation 是一个标准的 Win32 API，但事实上它是一个编译器内联函数。当你调用这个函数时，Visual C++生成以下代码：

```
MOV EAX,DWORD PTR [EBP-14]
```

GetExceptionInformation 是一个编译器内联函数，与它相关的 GetExceptionCode 函数也是如此。此函数实际上只是返回 GetExceptionInformation 返回的数据结构（EXCEPTION_POINTERS）中的一个结构（EXCEPTION_RECORD）中的一个域（ExceptionCode）的值。当 Visual C++为 GetExceptionCode 函数生成下面的指令时，它到底是想干什么？我把这个问题留给读者。（现在就能理解为什么 SDK 文档提醒我们要注意这两个函数的使用范围了。）

```
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

现在回到扩展的 EXCEPTION_REGISTRATION 结构上来。在这个结构开始前的 8 个字节处 (即[EBP-18H]处), Visual C++保留了一个 DWORD 来保存所有 **prolog** 代码执行完毕之后的堆栈指针 (ESP) 的值 (实际生成的指令为 MOV DWORD PTR [EBP-18H],ESP)。这个 DWORD 中保存的值是函数执行时 ESP 寄存器的正常值 (除了在准备调用其它函数时把参数压入堆栈这个过程会改变 ESP 寄存器的值并在函数返回时恢复它的值外, 函数在执行过程中一般不改变 ESP 寄存器的值)。

看起来好像我一下子给你灌输了太多的信息, 这点我承认。在继续下去之前, 让我们先暂停, 来回顾一下 Visual C++为使用结构化异常处理的函数生成的标准异常堆栈帧, 它看起来像下面这个样子:

EBP-00 _ebp

EBP-04 trylevel

EBP-08 scopetable 数组指针

EBP-0C handler 函数地址

EBP-10 指向前一个 EXCEPTION_REGISTRATION 结构

EBP-14 GetExceptionInformation

EBP-18 栈帧中的标准 ESP

在操作系统看来, 只存在组成原始 EXCEPTION_REGISTRATION 结构的两个域: 即 **[EBP-10h] 处的 prev 指针**和**[EBP-0Ch]处的 handler 函数指针**。栈帧中的其它所有内容是针对于 Visual C++的。把这个 Visual C++生成的标准异常堆栈帧记到脑子里之后, 让我们来看一下真正实现编译器层面 SEH 的这个 Visual C++运行时库例程——__except_handler3。

__except_handler3 和 scopetable

我真的很希望让你看一看 Visual C++运行时库源代码, 让你自己好好研究一下__except_handler3 函数, 但是我办不到。因为 Microsoft 并没有提供。在这里你就将就着看一下我为__except_handler3 函数写的伪代码吧 (如图 9 所示)。

图 9 __except_handler3 函数的伪代码

```
int __except_handler3(
struct _EXCEPTION_RECORD * pExceptionRecord,
struct EXCEPTION_REGISTRATION * pRegistrationFrame,
struct _CONTEXT *pContextRecord,
void * pDispatcherContext )
{
    LONG filterFuncRet;
    LONG trylevel;
    EXCEPTION_POINTERS exceptPtrs;
    PSCOPETABLE pScopeTable;
    CLD // 将方向标志复位 (不测试任何条件!)
    // 如果没有设置 EXCEPTION_UNWINDING 标志或 EXCEPTION_EXIT_UNWIND 标志
    // 表明这是第一次调用这个处理程序 (也就是说, 并非处于异常展开阶段)
```

```

if ( ! (pExceptionRecord->ExceptionFlags
        & (EXCEPTION_UNWINDING | EXCEPTION_EXIT_UNWIND)) )
{
    // 在堆栈上创建一个 EXCEPTION_POINTERS 结构
    exceptPtrs.ExceptionRecord = pExceptionRecord;
    exceptPtrs.ContextRecord = pContextRecord;
    // 把前面定义的 EXCEPTION_POINTERS 结构的地址放在比
    // establiher 栈帧低 4 个字节的位置上。参考前面我讲
    // 的编译器为 GetExceptionInformation 生成的汇编代码 *(PDWORD)((PBYTE)pRegistrationFrame - 4)
=    &exceptPtrs;
    // 获取初始的“trylevel”值
    trylevel = pRegistrationFrame->trylevel;
    // 获取指向 scopetable 数组的指针
    scopeTable = pRegistrationFrame->scopetable;
    search_for_handler:
    if ( pRegistrationFrame->trylevel != TRYLEVEL_NONE )
    {
        if ( pRegistrationFrame->scopetable[trylevel].lpfnFilter )
        {
            PUSH EBP // 保存这个栈帧指针
            // !!! 非常重要!!! 切换回原来的 EBP。正是这个操作才使得
            // 栈帧上的所有局部变量能够在异常发生后仍然保持它的值不变。
            EBP = &pRegistrationFrame->_ebp;
            // 调用过滤器函数
            filterFuncRet = scopetable[trylevel].lpfnFilter();
            POP EBP // 恢复异常处理程序的栈帧指针
            if ( filterFuncRet != EXCEPTION_CONTINUE_SEARCH )
            {
                if ( filterFuncRet < 0 ) // EXCEPTION_CONTINUE_EXECUTION
                    return ExceptionContinueExecution ;
                // 如果能够执行到这里，说明返回值为 EXCEPTION_EXECUTE_HANDLER
                scopetable = pRegistrationFrame->scopetable;
                // 让操作系统清理已经注册的栈帧，这会使本函数被递归调用
                __global_unwind2( pRegistrationFrame );
                // 一旦执行到这里，除最后一个栈帧外，所有的栈帧已经
                // 被清理完毕，流程要从最后一个栈帧继续执行
                EBP = &pRegistrationFrame->_ebp;
                __local_unwind2( pRegistrationFrame, trylevel );
            }
        }
    }
}

```

```

        // NLG = "non-local-goto" (setjmp/longjmp stuff)
        __NLG_Notify( 1 ); // EAX = scopetable->lpfnHandler
        // 把当前的 trylevel 设置成当找到一个异常处理程序时
        // SCOPETABLE 中当前正在被使用的那一个元素的内容
        pRegistrationFrame->trylevel = scopetable->previousTryLevel;
        // 调用__except {}块，这个调用并不会返回
        pRegistrationFrame->scopetable[trylevel].lpfnHandler();
    }
}

scopeTable = pRegistrationFrame->scopetable;
trylevel = scopeTable->previousTryLevel;
goto search_for_handler;
}
else // trylevel == TRYLEVEL_NONE
{
    return ExceptionContinueSearch;
}
}
else // 设置了 EXCEPTION_UNWINDING 标志或 EXCEPTION_EXIT_UNWIND 标志
{
    PUSH EBP // 保存 EBP
    EBP = &pRegistrationFrame->_ebp; // 为调用__local_unwind2 设置 EBP
    __local_unwind2( pRegistrationFrame, TRYLEVEL_NONE )
    POP EBP // 恢复 EBP
    return ExceptionContinueSearch;
}
}

```

虽然__except_handler3 的代码看起来很多，但是记住一点：它只是一个我在文章开头讲过的异常处理回调函数。它同 MYSEH.EXE 和 MYSEH2.EXE 中的异常回调函数都带有同样的四个参数。__except_handler3 大体上可以由第一个 if 语句分为两部分。这是由于这个函数可以在两种情况下被调用，一次是正常调用，另一次是在展开阶段。其中大部分是在非展开阶段的回调。

__except_handler3 一开始就在堆栈上创建了一个 EXCEPTION_POINTERS 结构，并用它的两个参数来对这个结构进行初始化。我在伪代码中把这个结构称为 exceptPtrs，它的地址被放在[EBP-14h]处。你回忆一下前面我讲的编译器为 GetExceptionInformation 和 GetExceptionCode 函数生成的汇编代码就会意识到，这实际上初始化了这两个函数使用的指针。

接着，__except_handler3 从 EXCEPTION_REGISTRATION 帧中获取当前的 trylevel（在[EBP-04h]处）。trylevel 变量实际是 scopetable 数组的索引，而正是这个数组才使得一个函数中的多个__try 块和嵌套的__try 块能够仅使用一个 EXCEPTION_REGISTRATION 结构。每个 scopetable 元素结构如下：

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

SCOPETABLE 结构中的第二个成员和第三个成员比较容易理解。它们分别是过滤器表达式代码的地址和相应的__except 块的地址。但是 previousTryLevel 成员有点复杂。总之一句话，它用于嵌套的__try 块。这里的关键是 **函数中的每个__try 块都有一个相应的 SCOPETABLE 结构**。

正如我前面所说，当前的 trylevel 指定了要使用的 scopetable 数组的哪一个元素，最终也就是指定了过滤器表达式和__except 块的地址。现在想像一下两个__try 块嵌套的情形。如果内层__try 块的过滤器表达式不处理某个异常，那外层__try 块的过滤器表达式就必须处理它。那现在要问，__except_handler3 是如何知道 SCOPETABLE 数组的哪个元素相应于外层的__try 块的呢？答案是：外层__try 块的索引由 SCOPETABLE 结构的 previousTryLevel 域给出。利用这种机制，你可以嵌套任意层的__try 块。previousTryLevel 域就好像是一个函数中所有可能的异常处理程序构成的线性链表中的结点一样。如果 trylevel 的值为 0xFFFFFFFF（实际上就是-1，这个值在 EXSUP.INC 中被定义为 TRYLEVEL_NONE），标志着这个链表结束。

回到__except_handler3 的代码中。在获取了当前的 trylevel 之后，它就调用相应的 SCOPETABLE 结构中的过滤器表达式代码。如果过滤器表达式返回 EXCEPTION_CONTINUE_SEARCH，__exception_handler3 移向 SCOPETABLE 数组中的下一个元素，这个元素的索引由 previousTryLevel 域给出。如果遍历完整个线性链表（还记得吗？这个链表是由于在一个函数内部嵌套使用__try 块而形成的）都没有找到处理这个异常的代码，__except_handler3 返回 DISPOSITION_CONTINUE_SEARCH（原文如此，但根据__except_handler 函数的定义，这个返回值应该为 ExceptionContinueSearch。实际上这两个常量的值是一样的。我在伪代码中已经将其改正过来了），这导致系统移向下一个 EXCEPTION_REGISTRATION 帧（这个链表是由于函数嵌套调用而形成的）。

如果过滤器表达式返回 EXCEPTION_EXECUTE_HANDLER，这意味着异常应该由相应的__except 块处理。它同时也意味着所有前面的 EXCEPTION_REGISTRATION 帧都应该从链表中移除，并且相应的__except 块都应该被执行。第一个任务通过调用__global_unwind2 来完成的，后面我会讲到这个函数。跳过这中间的一些清理代码，流程离开__except_handler3 转向__except 块。令人奇怪的是，流程并不从__except 块中返回，虽然是 __except_handler3 使用 CALL 指令调用了它。

当前的 trylevel 值是如何被设置的呢？它实际上是由编译器隐含处理的。编译器非常机灵地修改这个扩展的 EXCEPTION_REGISTRATION 结构中的 trylevel 域的值（实际上是生成修改这个域的值的代码）。如果你检查编译器为使用 SEH 的函数生成的汇编代码，就会在不同的地方都看到修改这个位于[EBP-04h]处的 trylevel 域的值代码。

__except_handler3 是如何做到既通过 CALL 指令调用__except 块而又不让执行流程返回呢？由于 CALL 指令要向堆栈中压入了一个返回地址，你可以想象这有可能破坏堆栈。如果你检查一下编译器为__except 块生成的代码，你会发现它做的第一件事就是将 EXCEPTION_REGISTRATION 结构下面 8 个字节处（即[EBP-18H]处）的一个 DWORD 值加载到 ESP 寄存器中（实际代码为 MOV ESP,DWORD PTR [EBP-18H]），这个值是在函数的 prolog 代码中被保存在这个位置的（实际代码为 MOV DWORD PTR [EBP-18H],ESP）。

ShowSEHFrames 程序

如果你现在觉得已经被 EXCEPTION_REGISTRATION、scopetable、trylevel、过滤器表达式以及展开等等之类的词搞得晕头转向的话，那和我最初的感觉一样。但是编译器层面的结构化异常处理方面的知识并不适合一点一点的学。除非你从整体上理解它，否则有很多内容单独看并没有什么意义。当面对大堆的理论时，我最自然的做法就是写一些应用我学到的理论方面的程序。如果它能够按照预料的那样工作，我就知道我的理解（通常）是正确的。

图 10 是 ShowSEHFrame.EXE 的源代码。它使用 __try/__except 块设置了好几个 Visual C++ SEH 帧。然后它显示每一个帧以及 Visual C++ 为每个帧创建的 scopetable 的相关信息。这个程序本身并不生成也不依赖任何异常。相反，我使用了多个 __try 块以强制 Visual C++ 生成多个 EXCEPTION_REGISTRATION 帧以及相应的 scopetable。

图 10 ShowSEHFrames.CPP

```
//=====
// ShowSEHFrames - Matt Pietrek 1997
// Microsoft Systems Journal, February 1997
// FILE: ShowSEHFrames.CPP
// 使 用 命 令 行 CL ShowSehFrames.CPP 进 行 编 译
//=====
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#pragma hdrstop
//-----
// 本程序仅适用于 Visual C++，它使用的数据结构是特定于 Visual C++ 的
//-----
#ifndef _MSC_VER
#error Visual C++ Required (Visual C++ specific information is displayed)
#endif
//-----
// 结构定义
//-----

// 操作系统定义的基本异常帧
struct EXCEPTION_REGISTRATION
{
    EXCEPTION_REGISTRATION* prev;
    FARPROC handler;
};
// Visual C++ 扩展异常帧指向的数据结构
```

```

struct scopetable_entry
{
    DWORD previousTryLevel;
    FARPROC lpfnFilter;
    FARPROC lpfnHandler;
};

// Visual C++ 使用的扩展异常帧
struct VC_EXCEPTION_REGISTRATION : EXCEPTION_REGISTRATION
{
    scopetable_entry * scopetable;
    int trylevel;
    int _ebp;
};

//-----
// 原型声明
//-----
// __except_handler3 是 Visual C++运行时库函数，我们想打印出它的地址
// 但是它的原型并没有出现在任何头文件中，所以我们需要自己声明它。
extern "C" int __except_handler3(PEXCEPTION_RECORD,
                                EXCEPTION_REGISTRATION *,
                                PCONTEXT,
                                PEXCEPTION_RECORD);

//-----
// 代码
//-----
//
// 显示一个异常帧及其相应的 scopetable 的信息
//
void ShowSEHFrame( VC_EXCEPTION_REGISTRATION * pVCExcRec )
{
    printf( "Frame: %08X Handler: %08X Prev: %08X Scopetable: %08X/n",
           pVCExcRec, pVCExcRec->handler, pVCExcRec->prev,
           pVCExcRec->scopetable );
    scopetable_entry * pScopeTableEntry = pVCExcRec->scopetable;
    for ( unsigned i = 0; i <= pVCExcRec->trylevel; i++ )
    {
        printf( " scopetable[%u] PrevTryLevel: %08X "
               "filter: %08X __except: %08X/n", i,

```

```

        pScopeTableEntry->previousTryLevel,
        pScopeTableEntry->lpfnFilter,
        pScopeTableEntry->lpfnHandler );
        pScopeTableEntry++;
    }
    printf( "/n" );
}

//
// 遍历异常帧的链表，按顺序显示它们的信息
//
void WalkSEHFrames( void )
{
    VC_EXCEPTION_REGISTRATION * pVCExcRec;
    // 打印出__except_handler3 函数的位置
    printf( "_except_handler3 is at address: %08X/n", _except_handler3 );
    printf( "/n" );
    // 从 FS:[0]处获取指向链表头的指针
    __asm mov eax, FS:[0]
    __asm mov [pVCExcRec], EAX
    // 遍历异常帧的链表。0xFFFFFFFF 标志着链表的结尾
    while ( 0xFFFFFFFF != (unsigned)pVCExcRec )
    {
        ShowSEHFrame( pVCExcRec );
        pVCExcRec = (VC_EXCEPTION_REGISTRATION *) (pVCExcRec->prev);
    }
}

void Function1( void )
{
    // 嵌套 3 层__try 块以便强制为 scopetable 数组产生 3 个元素
    __try
    {
        __try
        {
            __try
            {
                WalkSEHFrames(); // 现在显示所有的异常帧的信息
            }
        }
    }
}

```

```

        } __except( EXCEPTION_CONTINUE_SEARCH )
        {}
    } __except( EXCEPTION_CONTINUE_SEARCH )
    {}
} __except( EXCEPTION_CONTINUE_SEARCH )
{}
}

int main()
{
    int i;
    // 使用两个 __try 块（并不嵌套），这导致为 scopetable 数组生成两个元素
    __try
    {
        i = 0x1234;
    } __except( EXCEPTION_CONTINUE_SEARCH )
    {
        i = 0x4321;
    }
    __try
    {
        Function1(); // 调用一个设置更多异常帧的函数
    } __except( EXCEPTION_EXECUTE_HANDLER )
    {
        // 应该永远不会执行到这里，因为我们并没有打算产生任何异常
        printf( "Caught Exception in main/n" );
    }
    return 0;
}


```

ShowSEHFrames 程序中比较重要的函数是 **WalkSEHFrames** 和 **ShowSEHFrame**。**WalkSEHFrames** 函数首先打印出 **__except_handler3** 的地址，打印它的原因很快就清楚了。接着，它从 **FS:[0]**处获取异常链表的头指针，然后遍历该链表。此链表中每个结点都是一个 **VC_EXCEPTION_REGISTRATION** 类型的结构，它是我自己定义的，用于描述 **Visual C++**的异常处理帧。对于这个链表中的每个结点，**WalkSEHFrames** 都把指向这个结点的指针传递给 **ShowSEHFrame** 函数。

ShowSEHFrame 函数一开始就打印出异常处理帧的地址、异常处理回调函数的地址、前一个异常处理帧的地址以及 **scopetable** 的地址。接着，对于每个 **scopetable** 数组中的元素，它都打印出其 **previousTryLevel**、过滤器表达式的地址以及相应的 **__except** 块的地址。我是如何知道 **scopetable** 数组中有多少个元素的呢？其实我并不知道。但是我假定 **VC_EXCEPTION_REGISTRATION** 结构中的当前 **trylevel** 域的值比 **scopetable** 数组

中的元素总数少 1。

图 11 是 ShowSEHFrames 的运行结果。首先检查以 “Frame:” 开头的每一行，你会发现它们显示的异常处理帧在堆栈上的地址呈递增趋势，并且在前三个帧中，它们的异常处理程序的地址是一样的（都是 004012A8）。再看输出的开始部分，你会发现这个 004012A8 不是别的，它正是 Visual C++ 运行时库函数 `__except_handler3` 的地址。这证明了我前面所说的单个回调函数处理所有异常这一点。



```
[e:\msj\seh\showsehframes.exe
__except_handler3 is at address: 004012A8

Frame: 0012FF40 Handler: 004012A8 Prev: 0012FF70 Scopetable: 00404000
  scopetable[0] PrevTryLevel: FFFFFFFF filter: 00401180 __except: 00401188
  scopetable[1] PrevTryLevel: 00000000 filter: 00401162 __except: 0040116A
  scopetable[2] PrevTryLevel: 00000001 filter: 00401144 __except: 0040114C

Frame: 0012FF70 Handler: 004012A8 Prev: 0012FFB0 Scopetable: 00404028
  scopetable[0] PrevTryLevel: FFFFFFFF filter: 004011E8 __except: 004011F0
  scopetable[1] PrevTryLevel: FFFFFFFF filter: 00401219 __except: 00401224

Frame: 0012FFB0 Handler: 004012A8 Prev: 0012FFE0 Scopetable: 00404040
  scopetable[0] PrevTryLevel: FFFFFFFF filter: 0040153F __except: 0040155A

Frame: 0012FFE0 Handler: 77F3AB6C Prev: FFFFFFFF Scopetable: 77F3C1B0
  scopetable[0] PrevTryLevel: FFFFFFFF filter: 77F1AFC4 __except: 77F1AFD7

[e:\msj\seh]
```

图 11 ShowSEHFrames 运行结果

你可能想知道为什么明明 ShowSEHFrames 程序只有两个函数使用 SEH，但是却有三个异常处理帧使用 `__except_handler3` 作为它们的异常回调函数。实际上第三个帧来自 Visual C++ 运行时库。Visual C++ 运行时库源代码中的 `CRT0.C` 文件清楚地表明了对 `main` 或 `WinMain` 的调用也被一个 `__try/__except` 块封装着。这个 `__try` 块的过滤器表达式代码可以在 `WINXFLTR.C` 文件中找到。

回到 ShowSEHFrames 程序，注意到最后一个帧的异常处理程序的地址是 77F3AB6C，这与其它三个不同。仔细观察一下，你会发现这个地址在 `KERNEL32.DLL` 中。这个特别的帧就是由 `KERNEL32.DLL` 中的 `BaseProcessStart` 函数安装的，这在前面我已经说过。

展开

在挖掘展开（Unwinding）的实现代码之前让我们先来搞清楚它的意思。我在前面已经讲过所有可能的异常处理程序是如何被组织在一个由线程信息块的第一个 `DWORD`（`FS:[0]`）所指向的链表中的。由于针对某个特定异常的处理程序可能不在这个链表的开头，因此就需要从链表中依次移除实际处理异常的那个异常处理程序之前的所有异常处理程序。

正如你在 Visual C++ 的 `__except_handler3` 函数中看到的那样，展开是由 `__global_unwind2` 这个运行时库（RTL）函数来完成的。这个函数只是对 `RtlUnwind` 这个未公开的 API 进行了非常简单的封装。（现在这个 API 已经被公开了，但给出的信息极其简单，详细信息可以参考最新的 Platform SDK 文档。）

```
__global_unwind2(void * pRegistFrame)
{
    _RtlUnwind( pRegistFrame, &__ret_label, 0, 0 );
    __ret_label:
}
```

```
}
```

虽然从技术上讲 RtlUnwind 是一个 KERNEL32 函数，但它只是转发到了 NTDLL.DLL 中的同名函数上。图 12 是我为此函数写的伪代码。

图 12 RtlUnwind 函数的伪代码

```
void _RtlUnwind( PEXCEPTION_REGISTRATION pRegistrationFrame,
                PVOID returnAddr, // 并未使用！（至少是在 i386 机器上）
                PEXCEPTION_RECORD pExcptRec,
                DWORD _eax_value)
{
    DWORD stackUserBase;
    DWORD stackUserTop;
    PEXCEPTION_RECORD pExcptRec;
    EXCEPTION_RECORD exceptRec;
    CONTEXT context;
    // 从 FS:[4]和 FS:[8]处获取堆栈的界限
    RtlpGetStackLimits( &stackUserBase, &stackUserTop );
    if ( 0 == pExcptRec ) // 正常情况
    {
        pExcptRec = &exceptRec;
        pExcptRec->ExceptionFlags = 0;
        pExcptRec->ExceptionCode = STATUS_UNWIND;
        pExcptRec->ExceptionRecord = 0;
        pExcptRec->ExceptionAddress = [ebp+4]; // RtlpGetReturnAddress() — 获取返回地址
        pExcptRec->ExceptionInformation[0] = 0;
    }
    if ( pRegistrationFrame )
        pExcptRec->ExceptionFlags |= EXCEPTION_UNWINDING;
    else // 这两个标志合起来被定义为 EXCEPTION_UNWIND_CONTEXT
        pExcptRec->ExceptionFlags |= (EXCEPTION_UNWINDING | EXCEPTION_EXIT_UNWIND);
    context.ContextFlags = ( CONTEXT_i486 | CONTEXT_CONTROL |
                           CONTEXT_INTEGER | CONTEXT_SEGMENTS);
    RtlpCaptureContext( &context );
    context.Esp += 0x10;
    context.Eax = _eax_value;
    PEXCEPTION_REGISTRATION pExcptRegHead;
    pExcptRegHead = RtlpGetRegistrationHead(); // 返回 FS:[0]的值
    // 开始遍历 EXCEPTION_REGISTRATION 结构链表
    while ( -1 != pExcptRegHead )
```

```

{
    EXCEPTION_RECORD excptRec2;
    if ( pExcptRegHead == pRegistrationFrame )
    {
        NtContinue( &context, 0 );
    }
else
{
    // 如果存在某个异常帧在堆栈上的位置比异常链表的头部还低
    // 说明一定出现了错误
    if ( pRegistrationFrame && (pRegistrationFrame <= pExcptRegHead) )
    {
        // 生成一个异常
        excptRec2.ExceptionRecord = pExcptRec;
        excptRec2.NumberParameters = 0;
        excptRec2.ExceptionCode = STATUS_INVALID_UNWIND_TARGET;
        excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
        RtlRaiseException( &excptRec2 );
    }
}

PVOID pStack = pExcptRegHead + 8; // 8 = sizeof(EXCEPTION_REGISTRATION)
// 确保 pExcptRegHead 在堆栈范围内，并且是 4 的倍数
if ( (stackUserBase <= pExcptRegHead )
    && (stackUserTop >= pStack )
    && (0 == (pExcptRegHead & 3)) )
{
    DWORD pNewRegistHead;
    DWORD retValue;
    retValue = RtlpExecuteHandlerForUnwind(pExcptRec, pExcptRegHead, &context,
        &pNewRegistHead, pExcptRegHead->handler );
    if ( retValue != DISPOSITION_CONTINUE_SEARCH )
    {
        if ( retValue != DISPOSITION_COLLIDED_UNWIND )
        {
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.NumberParameters = 0;
            excptRec2.ExceptionCode = STATUS_INVALID_DISPOSITION;
            excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

```

```

        RtlRaiseException( &excptRec2 );
    }
    else
        pExcptRegHead = pNewRegistHead;
    }
    PEXCEPTION_REGISTRATION pCurrExcptReg = pExcptRegHead;
    pExcptRegHead = pExcptRegHead->prev;
    RtlpUnlinkHandler( pCurrExcptReg );
}
else // 堆栈已经被破坏！生成一个异常
{
    excptRec2.ExceptionRecord = pExcptRec;
    excptRec2.NumberParameters = 0;
    excptRec2.ExceptionCode = STATUS_BAD_STACK;
    excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
    RtlRaiseException( &excptRec2 );
}
}

// 如果执行到这里，说明已经到了 EXCEPTION_REGISTRATION
// 结构链表的末尾，正常情况下不应该发生这种情况。
// （因为正常情况下异常应该被处理，这样就不会到链表末尾）
if ( -1 == pRegistrationFrame )
    NtContinue( &context, 0 );
else
    NtRaiseException( pExcptRec, &context, 0 );
}

```

RtlUnwind 函数的伪代码到这里就结束了，以下是它调用的几个函数的伪代码：

```

PEXCEPTION_REGISTRATION RtlpGetRegistrationHead( void )
{
    return FS:[0];
}

RtlpUnlinkHandler( PEXCEPTION_REGISTRATION pRegistrationFrame )
{
    FS:[0] = pRegistrationFrame->prev;
}

void RtlpCaptureContext( CONTEXT * pContext )
{

```

```

pContext->Eax = 0;
pContext->Ecx = 0;
pContext->Edx = 0;
pContext->Ebx = 0;
pContext->Esi = 0;
pContext->Edi = 0;
pContext->SegCs = CS;
pContext->SegDs = DS;
pContext->SegEs = ES;
pContext->SegFs = FS;
pContext->SegGs = GS;
pContext->SegSs = SS;
pContext->EFlags = flags; // 它对应的汇编代码为 __asm{ PUSHFD / pop [xxxxxxx] }
pContext->Eip = 此函数的调用者的调用者的返回地址 // 读者看一下这个函数的
pContext->Ebp = 此函数的调用者的调用者的 EBP // 汇编代码就会清楚这一点
pContext->Esp = pContext->Ebp + 8;
}

```

虽然 RtlUnwind 函数的规模看起来很大，但是如果你按一定方法把它分开，其实并不难理解。它首先从 FS:[4]和 FS:[8]处获取当前线程堆栈的界限。它们对于后面要进行的合法性检查非常重要，以确保所有将要被展开的异常帧都在堆栈范围内。

RtlUnwind 接着在堆栈上创建了一个空的 EXCEPTION_RECORD 结构并把 STATUS_UNWIND 赋给它的 ExceptionCode 域，同时把 EXCEPTION_UNWINDING 标志赋给它的 ExceptionFlags 域。指向这个结构的指针作为其中一个参数被传递给每个异常回调函数。然后，这个函数调用 RtlCaptureContext 函数来创建一个空的 CONTEXT 结构，这个结构也变成了在展开阶段调用每个异常回调函数时传递给它们的一个参数。

RtlUnwind 函数的其余部分遍历 EXCEPTION_REGISTRATION 结构链表。对于其中的每个帧，它都调用 RtlpExecuteHandlerForUnwind 函数，后面我会讲到这个函数。正是这个函数带 EXCEPTION_UNWINDING 标志调用了异常处理回调函数。每次回调之后，它调用 RtlpUnlinkHandler 移除相应的异常帧。

RtlUnwind 函数的第一个参数是一个帧的地址，当它遍历到这个帧时就停止展开异常帧。上面所说的这些代码之间还有一些安全性检查代码，它们用来确保不出问题。如果出现任何问题，RtlUnwind 就引发一个异常，指示出了什么问题，并且这个异常带有 EXCEPTION_NONCONTINUABLE 标志。当一个进程被设置了这个标志时，它就不允许再运行，必须终止。

未处理异常

在文章的前面，我并没有全面描述 UnhandledExceptionFilter 这个 API。通常情况下你并不直接调用它（尽管你可以这么做）。大多数情况下它都是由 KERNEL32 中进行默认异常处理的过滤器表达式代码调用。前面 BaseProcessStart 函数的伪代码已经表明了这一点。

图 13 是我为 UnhandledExceptionFilter 函数写的伪代码。这个 API 有点奇怪（至少在我看来是这样）。如果异常的类型是 EXCEPTION_ACCESS_VIOLATION，它就调用 _BasepCheckForReadOnlyResource。虽然我没有提供这个函数的伪代码，但可以简要描述一下。如果是因为要对 EXE 或 DLL 的资源节（.rsrc）进行写操

作而导致的异常，_BasepCurrentTopLevelFilter 就改变出错页面正常的只读属性，以便允许进行写操作。如果是这种特殊的情况，UnhandledExceptionFilter 返回 EXCEPTION_CONTINUE_EXECUTION，使系统重新执行出错指令。

图 13 UnhandledExceptionFilter 函数的伪代码

```
UnhandledExceptionFilter( STRUCT _EXCEPTION_POINTERS *pExceptionPtrs )
{
    PEXCEPTION_RECORD pExcptRec;

    DWORD currentESP;

    DWORD retValue;

    DWORD DEBUGPORT;

    DWORD dwTemp2;

    DWORD dwUseJustInTimeDebugger;

    CHAR szDbgCmdFmt[256]; // 从 AeDebug 这个注册表键值返回的字符串

    CHAR szDbgCmdLine[256]; // 实际的调试器命令行参数（已填入进程 ID 和事件 ID）

    STARTUPINFO startupinfo;

    PROCESS_INFORMATION pi;

    HARDERR_STRUCT harderr; // ???

    BOOL fAeDebugAuto;

    TIB * pTib; // 线程信息块

    pExcptRec = pExceptionPtrs->ExceptionRecord;
    if ( (pExcptRec->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
        && (pExcptRec->ExceptionInformation[0]) )
    {
        retValue=BasepCheckForReadOnlyResource(pExcptRec->ExceptionInformation[1]);
        if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
            return EXCEPTION_CONTINUE_EXECUTION;
    }
    // 查看这个进程是否运行于调试器下
    retValue = NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort,
        &debugPort, sizeof(debugPort), 0 );
```

```

if ( (retValue >= 0) && debugPort ) // 通知调试器
    return EXCEPTION_CONTINUE_SEARCH;
// 用户调用 SetUnhandledExceptionFilter 了吗?
// 如果调用了, 那现在就调用他安装的异常处理程序
if ( _BasepCurrentTopLevelFilter )
{
    retValue = _BasepCurrentTopLevelFilter( pExceptionPtrs );
    if ( EXCEPTION_EXECUTE_HANDLER == retValue )
        return EXCEPTION_EXECUTE_HANDLER;
    if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
        return EXCEPTION_CONTINUE_EXECUTION;
    // 只有返回值为 EXCEPTION_CONTINUE_SEARCH 时才会继续执行下去
}
// 调用过 SetErrorMode(SEM_NOGPFAULTERRORBOX)吗?
{
    harderr.elem0 = pExcptRec->ExceptionCode;
    harderr.elem1 = pExcptRec->ExceptionAddress;
    if ( EXCEPTION_IN_PAGE_ERROR == pExcptRec->ExceptionCode )
        harderr.elem2 = pExcptRec->ExceptionInformation[2];
    else
        harderr.elem2 = pExcptRec->ExceptionInformation[0];
    dwTemp2 = 1;
    fAeDebugAuto = FALSE;
    harderr.elem3 = pExcptRec->ExceptionInformation[1];
    pTib = FS:[18h];
    DWORD someVal = pTib->pProcess->OxC;
    if ( pTib->threadID != someVal )
    {
        __try
        {
            char szDbgCmdFmt[256];
            retValue = GetProfileStringA( "AeDebug", "Debugger", 0,
                szDbgCmdFmt, sizeof(szDbgCmdFmt)-1 );
        }
        if ( retValue )
            dwTemp2 = 2;
        char szAuto[8];
        retValue = GetProfileStringA( "AeDebug", "Auto", "0",
            szAuto, sizeof(szAuto)-1 );
    }
}

```

```

if ( retValue )
if ( 0 == strcmp( szAuto, "1" ) )
    if ( 2 == dwTemp2 )
        fAeDebugAuto = TRUE;
}
__except( EXCEPTION_EXECUTE_HANDLER )
{
    ESP = currentESP;
    dwTemp2 = 1;
    fAeDebugAuto = FALSE;
}
}
if ( FALSE == fAeDebugAuto )
{
    retValue=NtRaiseHardError(STATUS_UNHANDLED_EXCEPTION | 0x10000000,
        4, 0, &harderr,_BasepAlreadyHadHardError ? 1 : dwTemp2,
        &dwUseJustInTimeDebugger );
}
else
{
    dwUseJustInTimeDebugger = 3;
    retValue = 0;
}
if (retValue >= 0 && (dwUseJustInTimeDebugger == 3)
&& (!_BasepAlreadyHadHardError)&&(!_BaseRunningInServerProcess))
{
    _BasepAlreadyHadHardError = 1;
    SECURITY_ATTRIBUTES secAttr = { sizeof(secAttr), 0, TRUE };
    HANDLE hEvent = CreateEventA( &secAttr, TRUE, 0, 0 );
    memset( &startupinfo, 0, sizeof(startupinfo) );
    sprintf(szDbgCmdLine, szDbgCmdFmt, GetCurrentProcessId(), hEvent);
    startupinfo.cb = sizeof(startupinfo);
    startupinfo.lpDesktop = "Winsta0/Default"
    CsrIdentifyAlertableThread(); // ???
    retValue = CreateProcessA( 0,          // 应用程序名称
szDbgCmdLine, // 命令行
0, 0,          // 进程和线程安全属性
1,            // bInheritHandles

```

```

0, 0,      // 创建标志、环境
0,        // 当前目录
&statupinfo, // STARTUPINFO
&pi);     // PROCESS_INFORMATION
if ( retValue && hEvent )
{
    NtWaitForSingleObject( hEvent, 1, 0 );
    return EXCEPTION_CONTINUE_SEARCH;
}
}
if ( _BasepAlreadyHadHardError )
NtTerminateProcess(GetCurrentProcess(), pExcptRec->ExceptionCode);
}
return EXCEPTION_EXECUTE_HANDLER;
}

LPTOP_LEVEL_EXCEPTION_FILTER
SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter )
{
    // _BasepCurrentTopLevelFilter 是 KERNEL32.DLL 中的一个全局变量
    LPTOP_LEVEL_EXCEPTION_FILTER previous= _BasepCurrentTopLevelFilter;
    // 设置为新值
    _BasepCurrentTopLevelFilter = lpTopLevelExceptionFilter;
    return previous; // 返回以前的值
}

```

UnhandledExceptionFilter 接下来的任务是确定进程是否运行于 Win32 调试器下。也就是进程的创建标志中是否带有标志 **DEBUG_PROCESS** 或 **DEBUG_ONLY_THIS_PROCESS**。它使用 **NtQueryInformationProcess** 函数来确定进程是否正在被调试，我在本月的 **Under the Hood** 专栏中讲解了这个函数。如果正在被调试，**UnhandledExceptionFilter** 就返回 **EXCEPTION_CONTINUE_SEARCH**，这告诉系统去唤醒调试器并告诉它在被调试程序（debuggee）中产生了一个异常。

UnhandledExceptionFilter 接下来调用用户安装的未处理异常过滤器（如果存在的话）。通常情况下，用户并没有安装回调函数，但是用户可以调用 **SetUnhandledExceptionFilter** 这个 API 来安装。上面我也提供了这个 API 的伪代码。这个函数只是简单地用用户安装的回调函数的地址来替换一个全局变量，并返回替换前的值。

有了初步的准备之后，**UnhandledExceptionFilter** 就开始做它的主要工作：用一个时髦的应用程序错误对话框来通知你犯了低级的编程错误。有两种方法可以避免出现这个对话框。第一种方法是调用 **SetErrorMode** 函数并指定 **SEM_NOGPFAULTERRORBOX** 标志。另一种方法是将在 **AeDebug** 子键下的 **Auto** 的值

设为 1。此时 `UnhandledExceptionFilter` 跳过应用程序错误对话框直接启动 AeDebug 子键下的 Debugger 的值所指定的调试器。如果你熟悉“即时调试（Just In Time Debugging, JIT）”的话，这就是操作系统支持它的地方。接下来我会详细讲。

大多数情况下，上面的两个条件都为假。这样 `UnhandledExceptionFilter` 就调用 `NTDLL.DLL` 中的 `NtRaiseHardError` 函数。正是这个函数产生了应用程序错误对话框。这个对话框等待你单击“确定”按钮来终止进程，或者单击“取消”按钮来调试它。（单击“取消”按钮而不是“确定”按钮来加载调试器好像有点颠倒了，可能这只是我个人的感觉吧。）

如果你单击“确定”，`UnhandledExceptionFilter` 就返回 `EXCEPTION_EXECUTE_HANDLER`。调用 `UnhandledExceptionFilter` 的进程通常通过终止自身来作为响应（正像你在 `BaseProcessStart` 的伪代码中看到的那样）。这就产生了一个有趣的问题——大多数人都认为是系统终止了产生未处理异常的进程，而实际上更准确的说法应该是，系统进行了一些设置使得产生未处理异常的进程将自身终止掉了。

`UnhandledExceptionFilter` 执行时真正有意思的部分是当你单击应用程序错误对话框中的“取消”按钮，此时系统将调试器附加（attach）到出错进程上。这段代码首先调用 `CreateEvent` 来创建一个事件内核对象，调试器成功附加到出错进程之后会将此事件对象变成有信号状态。这个事件句柄以及出错进程的 ID 都被传到 `sprintf` 函数，由它将其格式化成一个命令行，用来启动调试器。一切就绪之后，`UnhandledExceptionFilter` 就调用 `CreateProcess` 来启动调试器。如果 `CreateProcess` 成功，它就调用 `NtWaitForSingleObject` 来等待前面创建的那个事件对象。此时这个调用被阻塞，直到调试器进程将此事件变成有信号状态，以表明它已经成功附加到出错进程上。`UnhandledExceptionFilter` 函数中还有一些其它的代码，我在这里只讲重要的。

进入地狱

如果你已经走了这么远，不把整个过程讲完对你有点不公平。我已经讲了当异常发生时操作系统是如何调用用户定义的回调函数的。我也讲了这些回调的内部情况，以及编译器是如何使用它们来实现 `__try` 和 `__except` 的。我甚至还讲了当某个异常没有被处理时所发生的情况以及系统所做的扫尾工作。剩下的就只有异常回调过程最初是从哪里开始的这个问题了。好吧，让我们深入系统内部来看一下结构化异常处理的开始阶段吧。

图 14 是我为 `KiUserExceptionDispatcher` 函数和一些相关函数写的伪代码。这个函数在 `NTDLL.DLL` 中，它是异常处理执行的起点。为了绝对准确起见，我必须指出：刚才说的并不是绝对准确。例如在 Intel 平台上，一个异常导致 CPU 将控制权转到 ring 0（0 特权级，即内核模式）的一个处理程序上。这个处理程序由中断描述符表（Interrupt Descriptor Table, IDT）中的一个元素定义，它是专门用来处理相应异常的。我跳过所有的内核模式代码，假设当异常发生时 CPU 直接将控制权转到了 `KiUserExceptionDispatcher` 函数。

图 14 KiUserExceptionDispatcher 的伪代码

```
KiUserExceptionDispatcher( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
    DWORD retValue;
    // 注意：如果异常被处理，那么 RtlDispatchException 函数就不会返回
    if ( RtlDispatchException( pExcerptRec, pContext ) )
        retValue = NtContinue( pContext, 0 );
    else
        retValue = NtRaiseException( pExcerptRec, pContext, 0 );
}
```

```

EXCEPTION_RECORD excptRec2;
excptRec2.ExceptionCode = retValue;
excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
excptRec2.ExceptionRecord = pExcptRec;
excptRec2.NumberParameters = 0;
    RtlRaiseException( &excptRec2 );
}

int RtlDispatchException( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
    DWORD stackUserBase;
    DWORD stackUserTop;
    PEXCEPTION_REGISTRATION pRegistrationFrame;
    DWORD hLog;
    // 从 FS:[4]和 FS:[8]处获取堆栈的界限
    RtlpGetStackLimits( &stackUserBase, &stackUserTop );
    pRegistrationFrame = RtlpGetRegistrationHead();
    while ( -1 != pRegistrationFrame )
    {
        PVOID justPastRegistrationFrame = &pRegistrationFrame + 8;
        if ( stackUserBase > justPastRegistrationFrame )
        {
            pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
            return DISPOSITION_DISMISS; // 0
        }
        if ( stackUserTop < justPastRegistrationFrame )
        {
            pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
            return DISPOSITION_DISMISS; // 0
        }
        if ( pRegistrationFrame & 3 ) // 确保堆栈按 DWORD 对齐
        {
            pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
            return DISPOSITION_DISMISS; // 0
        }
        if ( someProcessFlag )
        {
            hLog = RtlpLogExceptionHandler( pExcptRec, pContext, 0,

```



```

        pRegistrationFrame, 0x10 );
    }

    DWORD retValue, dispatcherContext;
    retValue= RtlpExecuteHandlerForException(pExcptRec, pRegistrationFrame,
        pContext, &dispatcherContext,

if ( someProcessFlag )
RtlpLogLastExceptionDisposition( hLog, retValue );
if ( 0 == pRegistrationFrame )
{
    pExcptRec->ExceptionFlags &= ~EH_NESTED_CALL; // 关闭标志
}
EXCEPTION_RECORD excptRec2;
DWORD yetAnotherValue = 0;
if ( DISPOSITION_DISMISS == retValue )
{
    if ( pExcptRec->ExceptionFlags & EH_NONCONTINUABLE )
    {
        excptRec2.ExceptionRecord = pExcptRec;
        excptRec2.ExceptionNumber = STATUS_NONCONTINUABLE_EXCEPTION;
        excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
        excptRec2.NumberParameters = 0;
        RtlRaiseException( &excptRec2 );
    }
    else
        return DISPOSITION_CONTINUE_SEARCH;
}
else if ( DISPOSITION_CONTINUE_SEARCH == retValue )
{}
else if ( DISPOSITION_NESTED_EXCEPTION == retValue )
{
    pExcptRec->ExceptionFlags |= EH_EXIT_UNWIND;
    if ( dispatcherContext > yetAnotherValue )
        yetAnotherValue = dispatcherContext;
}
else // DISPOSITION_COLLIDED_UNWIND
{
    excptRec2.ExceptionRecord = pExcptRec;

```

```

    excptRec2.ExceptionNumber = STATUS_INVALID_DISPOSITION;
    excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
    excptRec2.NumberParameters = 0;
    RtlRaiseException( &excptRec2 );
}

pRegistrationFrame = pRegistrationFrame->prev; // 转到前一个帧
}
return DISPOSITION_DISMISS;
}

_RtlpExecuteHandlerForException: // 处理异常（第一次）
MOV EDX,XXXXXXXX
JMP ExecuteHandler

RtlpExecuteHandlerForUnwind: // 处理展开（第二次）
    MOV EDX,XXXXXXXX

int ExecuteHandler( PEXCEPTION_RECORD pExcptRec,
PEXCEPTION_REGISTRATION pExcptReg,
CONTEXT * pContext,
PVOID pDispatcherContext,
FARPROC handler ) // 实际上是指向_except_handler()的指针
{
    // 安装一个 EXCEPTION_REGISTRATION 帧，EDX 指向相应的 handler 代码
    PUSH EDX
    PUSH FS:[0]
    MOV FS:[0],ESP
    // 调用异常处理回调函数
    EAX = handler( pExcptRec, pExcptReg, pContext, pDispatcherContext );
    // 移除 EXCEPTION_REGISTRATION 帧
    MOV ESP,DWORD PTR FS:[00000000]
    POP DWORD PTR FS:[00000000]
    return EAX;
}

_RtlpExecuteHandlerForException 使用的异常处理程序：
{
    // 如果设置了展开标志，返回 DISPOSITION_CONTINUE_SEARCH
    // 否则，给 pDispatcherContext 赋值并返回 DISPOSITION_NESTED_EXCEPTION
    return pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT ?
        DISPOSITION_CONTINUE_SEARCH : ( *pDispatcherContext =

```

```

        pRegistrationFrame->scopetable,
        DISPOSITION_NESTED_EXCEPTION );
}

_RtlpExecuteHandlerForUnwind 使用的异常处理程序:
{
    // 如果设置了展开标志, 返回 DISPOSITION_CONTINUE_SEARCH
    // 否则, 给 pDispatcherContext 赋值并返回 DISPOSITION_COLLIDED_UNWIND
return  pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT ?
        DISPOSITION_CONTINUE_SEARCH : ( *pDispatcherContext =
            pRegistrationFrame->scopetable,
            DISPOSITION_COLLIDED_UNWIND );
}

```

`KiUserExceptionDispatcher` 的核心是对 `RtlDispatchException` 的调用。这拉开了搜索已注册的异常处理程序的序幕。如果某个处理程序处理这个异常并继续执行, 那么对 `RtlDispatchException` 的调用就不会返回。如果它返回了, 只有两种可能: 或者调用了 `NtContinue` 以便让进程继续执行, 或者产生了新的异常。如果是这样, 那异常就不能再继续处理了, 必须终止进程。

现在把目光对准 `RtlDispatchException` 函数的代码, 这就是我通篇提到的遍历异常帧的代码。这个函数获取一个指向 `EXCEPTION_REGISTRATION` 结构链表的指针, 然后遍历此链表以寻找一个异常处理程序。由于堆栈可能已经被破坏了, 所以这个例程非常谨慎。在调用每个 `EXCEPTION_REGISTRATION` 结构中指定的异常处理程序之前, 它确保这个结构是按 `DWORD` 对齐的, 并且是在线程的堆栈之中, 同时在堆栈中比前一个 `EXCEPTION_REGISTRATION` 结构高。

`RtlDispatchException` 并不直接调用 `EXCEPTION_REGISTRATION` 结构中指定的异常处理程序。相反, 它调用 `RtlpExecuteHandlerForException` 来完成这个工作。根据 `RtlpExecuteHandlerForException` 的执行情况, `RtlDispatchException` 或者继续遍历异常帧, 或者引发另一个异常。这第二次的异常表明异常处理程序内部出现了错误, 这样就不能继续执行下去了。

`RtlpExecuteHandlerForException` 的代码与 `RtlpExecuteHandlerForUnwind` 的代码极其相似。你可能会回忆起来在前面讨论展开时我提到过它。这两个“函数”都只是简单地给 `EDX` 寄存器加载一个不同的值然后就调用 `ExecuteHandler` 函数。也就是说, `RtlpExecuteHandlerForException` 和 `RtlpExecuteHandlerForUnwind` 都是 `ExecuteHanlder` 这个公共函数的前端。

`ExecuteHandler` 查找 `EXCEPTION_REGISTRATION` 结构的 `handler` 域的值并调用它。令人奇怪的是, 对异常处理回调函数的调用本身也被一个结构化异常处理程序封装着。在 `SEH` 自身中使用 `SEH` 看起来有点奇怪, 但你思索一会儿就会理解其中的含义。如果在异常回调过程中引发了另外一个异常, 操作系统需要知道这个情况。根据异常发生在最初的回调阶段还是展开回调阶段, `ExecuteHandler` 或者返回 `DISPOSITION_NESTED_EXCEPTION`, 或者返回 `DISPOSITION_COLLIDED_UNWIND`。这两者都是“红色警报! 现在把一切都关掉!”类型的代码。

如果你像我一样, 那不仅理解所有与 `SEH` 有关的函数非常困难, 而且记住它们之间的调用关系也非常困难。为了帮助我自己记忆, 我画了一个调用关系图 (图 15)。

现在要问：在调用 `ExecuteHandler` 之前设置 `EDX` 寄存器的值有什么用呢？这非常简单。如果 `ExecuteHandler` 在调用用户安装的异常处理程序的过程中出现了什么错误，它就把 `EDX` 指向的代码作为原始的异常处理程序。它把 `EDX` 寄存器的值压入堆栈作为原始的 `EXCEPTION_REGISTRATION` 结构的 `handler` 域。这基本上与我在 `MYSEH` 和 `MYSEH2` 中对原始的结构化异常处理的使用情况一样。

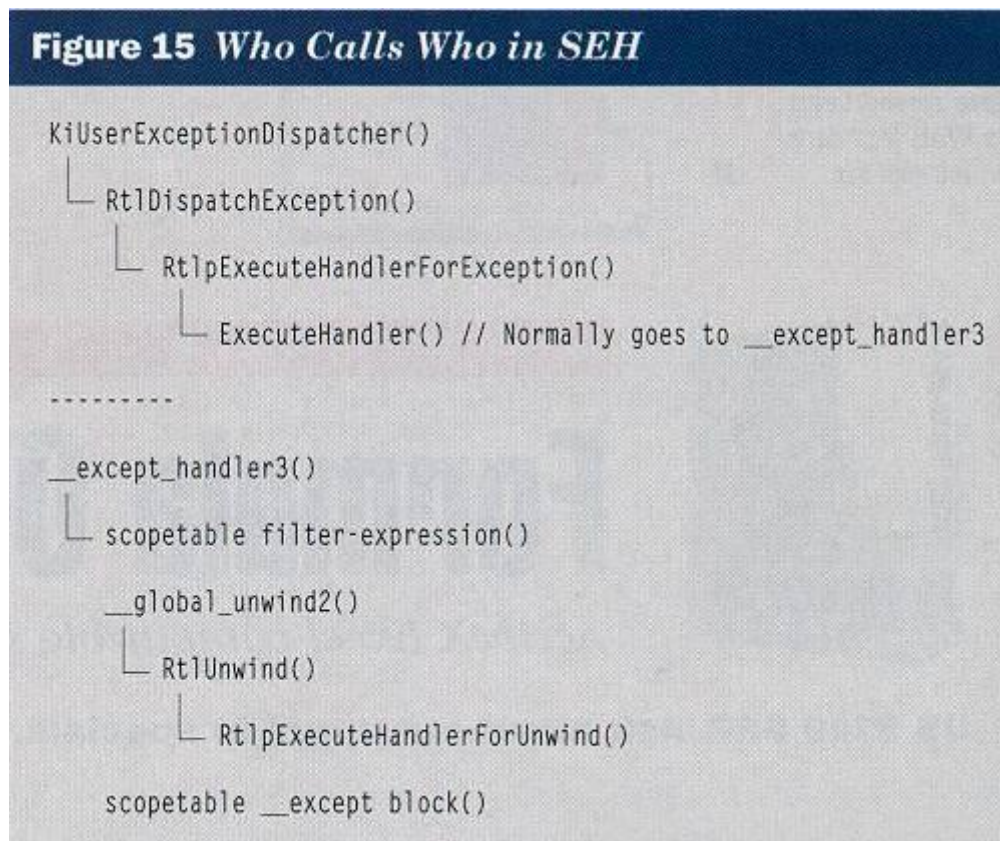


图 15 在 SEH 中是谁调用了谁

结论

结构化异常处理是 Win32 一个非常好的特性。多亏有了像 Visual C++ 之类的编译器的支持层对它的封装，一般的程序员才能付出比较小的学习代价就能利用 SEH 所提供的便利。但是在操作系统层面上，事情远比 Win32 文档说的复杂。

不幸的是，由于人人都认为系统层面的 SEH 是一个非常困难的问题，因此至今这方面的资料都不多。在本文中，我已经向你指出了系统层面的 SEH 就是围绕着简单的回调在打转。如果你理解了回调的本质，在此基础上分层理解，系统层面的结构化异常处理也不是那么难掌握。

附录：关于 prolog 和 epilog

美国英语中的“prolog”实际上就是“prologue”。从这个词的意思“序幕、序言”就能大致猜出它的作用。一个函数的 prolog 代码主要是为这个函数的执行做一些准备工作，例如设置堆栈帧、设置局部变量所使用的堆栈空间以及保存相关的寄存器等。标准的 prolog 代码开头一般为以下三条指令：

```
PUSH    EBP
MOV     EBP, ESP
SUB     ESP, XXX
```

上面的三条指令为使用 `EBP` 寄存器来访问函数的参数（正偏移）和局部变量（负偏移）做好了准备。

例如按照__stdcall 调用约定，调用者（caller）将被调函数（callee）的参数从右向左压入堆栈，然后用 CALL 指令调用这个函数。CALL 指令将返回地址压入堆栈，然后流程就转到了被调函数的 prolog 代码。此时[ESP]中是返回地址，[ESP+4]中是函数的第一个参数。本来可以就这样使用 ESP 寄存器来访问参数，但由于 PUSH 和 POP 指令会隐含修改 ESP 寄存器的值，这样同一个参数在不同时刻可能需要通过不同的指令形式来访问（例如，如果现在向堆栈中压入一个值的话，那访问第一个参数就需要使用[ESP+8]了）。为了解决这个问题，所以使用 EBP 寄存器。EBP 寄存器被称为栈帧（frame）指针，它正是用于此目的。当上述 prolog 指令中的前两条指令执行后，就可以使用 EBP 来访问参数了，并且在整个函数中都不会改变此寄存器的值。在前面的例子中， [EBP+8]处就是第一个参数的值，[EBP+0Ch]处是第二个参数的值，依次类推。

大多数 C/C++编译器都有“栈帧指针省略（ Frame-Pointer Omission ）” 这个选项（在 Microsoft C/C++编译器中为/Oy），它导致函数使用 ESP 来访问参数，从而可以空闲出一个寄存器（EBP）用于其它目的，并且由于不需要设置堆栈帧，从而会稍微提高运行速度。但是在某些情况下必须使用堆栈帧。作者在前面也提到过，Microsoft 已经在其 MSDN 文档中指明：结构化异常处理是 [基于帧](#) 的异常处理。也就是说，它必须使用堆栈帧。当你查看编译器为使用 SEH 的函数生成的汇编代码时就会清楚这一点。无论你是否使用 /Oy 选项，它都设置堆栈帧。

可能有的读者在调试应用程序时偶然进入到了系统 DLL（例如 NTDLL.DLL）中，但是却意外地发现许多函数的 prolog 代码的第一条指令并不是上面所说的“PUSH EBP”，而是一条“垃圾”指令——“MOV EDI, EDI”（这条指令占两个字节）。Microsoft C/C++编译器被称为优化编译器，它怎么可能生成这么一条除了占用空间之外别无它用的指令呢？实际上，如果你比较细心的话，会发现以这条指令开头的函数的前面有 5 条 NOP 指令（它们一共占 5 个字节），如下图所示。

001B:7C937A37	C9	LEAVE	
001B:7C937A38	C20C00	RET	000C
001B:7C937A3B	90	NOP	
001B:7C937A3C	90	NOP	
001B:7C937A3D	90	NOP	
001B:7C937A3E	90	NOP	
001B:7C937A3F	90	NOP	
_RtlUnwind			
001B:7C937A40	8BFF	MOV	EDI,EDI
001B:7C937A42	55	PUSH	EBP
001B:7C937A43	8BEC	MOV	EBP,ESP
001B:7C937A45	81EC7C030000	SUB	ESP,0000037C

考虑一下使用 JMP 指令进行近跳转和远跳转分别需要几个字节？他们正好分别是 2 个字节和 5 个字节！这难道是巧合？熟悉 API 拦截的读者可能已经猜到了，它们是供拦截 API 时使用的。实际上，这是 Microsoft 对系统打“热补丁”（Hot Patching）时拦截 API 用的。在打“热补丁”时，修补程序在 5 条 NOP 指令处写入一个远跳转指令，以跳转到被修补过的代码处。而“MOV EDI, EDI”处用一个近跳转指令覆盖，它跳转到 5 个 NOP 指令所在的位置。使用“MOV EDI, EDI”而不是直接使用两个 NOP 指令是出于性能考虑。

第三条指令用于为局部变量保留空间，其中的 xxx 就是需要保留的字节数。不使用局部变量的函数没有这条指令。另外，如果局部变量比较少的话——例如 2 个，为了性能考虑，编译器往往会使用类似于两条“PUSH ECX”这样的指令来为局部变量保留空间。这三条指令后面一般还有几条 PUSH 指令用于保存函数使用的寄存器（一般是 EBX、ESI 和 EDI）。

与 prolog 代码相对的就是 epilog 代码。与 prolog 类似，从它的意思“尾声、结尾”也能猜出它的作用。它主要做一些清理工作。标准的 epilog 代码如下：

```
MOV    ESP, EBP
POP     EBP
RET     XXX
```

这三条指令前面可能还有几条 POP 指令用于恢复在 **prolog** 代码中保存的寄存器（如果存在的话）。有了前面的分析，**epilog** 代码不言自明。需要说明的一点是，最后的 RET 指令用于返回调用者，并从堆栈中弹出无用信息，XXX 指定了弹出的字节数。它一般用于将参数弹出堆栈。因此从这个值就可以知道函数的参数个数（每个参数均为 4 字节）。

为了简化这种操作，Intel 引入了 ENTER 和 LEAVE 指令。其中 ENTER 相当于前面所说的 **prolog** 代码的前两条指令，而 LEAVE 相当于上面的 **epilog** 代码的前两条指令。但由于实现上 ENTER 指令比前面所说的两条指令执行速度慢，因此编译器都不使用这条指令。这样，你实际看到的情况就是：**prolog** 代码就是前面所说的那样，但 **epilog** 代码使用了 LEAVE 指令。