漫谈兼容内核之二十五：
Windows 的结构化异常处理(二)

毛德操

在前一篇漫谈中，读者看到了宏操作 SEH_TRY{} SEH_HANDLE{} SEH_END 怎样为一个 SEH 保护域的代码执行做好了发生异常的准备。

现在，假定受保护的代码在执行中真的发生了异常。例如在执行语句“SafeMaximumSize = *MaximumSize”的时候发生了异常，原因是参数 MaximumSize 所指向的页面无映射。

对于内核的底层，异常和中断基本上是同一回事。CPU 会根据引起异常的原因跳转到不同的地址，就好像向量中断一样。例如因除数为 0 而引起的异常就跳转到_KiTrap0：

```
_KiTrap0:
    /* Push error code */
    push 0

    /* Enter trap */
    TRAP_PROLOG(0)

    /* Call the C exception handler */
    push 0
    push ebp
    call _KiTrapHandler
    add esp, 8

    /* Check for v86 recovery */
    cmp eax, 1

    /* Return to caller */
    jne _Kei386EoiHelper@0
    jmp _KiV86Complete
```

像_KiTrap0 这样的程序入口有 20 个，分别为_KiTrap0 至_KiTrap19，这就像中断机制有许多中断向量、从而有许多中断响应程序入口一样。其中特别值得一提的是：

- 0 号异常为除法运算出错，特别是除数为 0。
- 3 号异常用于通过自陷指令“INT 3”实现的程序断点。
- 6 号异常为遇到非法指令。
- 9 号异常为浮点指令异常。
- 13 号异常为总保护(General Protection)，例如企图在用户空间执行特权指令等等。
- 14 号异常为存储页面异常。
- 16 号异常为浮点运算异常。

异常(Exception)可以分成三大类。第一类是因执行指令失败而引起的，例如存储页面的映射和访问权限引起的异常、还有因为除数为 0 而引起的异常，就都属于这一类，这是最大

的一类。这一类异常有个共同的特点，就是发生异常时自动压入堆栈的是失败了的那条指令的地址，而不是它的下一条指令的地址，其用意是返回以后可以重新执行一遍这条指令。第二类是因为执行自陷指令、例如“INT 3”、而引起的。这一类异常的返回地址是自陷指令的下一条指令所在的地址。第三类是已经无法恢复的严重出错，所以称为“Abort”。例如 8 号异常为“双重出错(Double Fault)”，就属于这一类。

注意在_KiTrap0 下面有个注释“/* Push error code */”，并有一条指令“push 0”。这是因为 CPU 对于有些异常会自动产生一个出错代码。一般而言，CPU 在发生异常时自动压入堆栈的数据依次为 EFLAGS、CS、EIP、和出错代码 ERRORCODE，如果异常发生在用户空间则在 EFLAGS 之前还有用户空间的 SS 和 ESP。但是 CPU 对于 0 号异常以及别的一些异常是不产生出错代码的。这样，对于 0 号等等异常，堆栈上就少了一项数据，使得堆栈上“异常框架”的大小也不一样了。为此，这里就补上一条指令“push 0”，在堆栈上占住一个位置，使所有的异常都有相同大小的框架。相比之下，例如_KiTrap13 和_KiTrap14 下面就没有这条指令。

所有这些异常响应程序都要执行一个宏操作 TRAP_PROLOG()，其参数是异常的编号，对于_KiTrap0 这就是 0。

```
#define TRAP_PROLOG(Label) \
    /* Just to be safe, clear out the HIWORD, since it's reserved */ \
    mov word ptr [esp+2], 0; \
\
    /* Save the non-volatiles */ \
    push ebp; \
    push ebx; \
    push esi; \
    push edi; \
\
    /* Save FS and set it to PCR */ \
    push fs; \
    mov ebx, KGDT_R0_PCR; \
    mov fs, bx; \
\
    /* Save exception list and bogus previous mode */ \
    push fs:[KPCR_EXCEPTION_LIST]; \
    push -1; \
\
    /* Save volatiles and segment registers */ \
    push eax; \
    push ecx; \
    push edx; \
    push ds; \
    push es; \
    push gs; \
\
    /* Set the R3 data segment */ \
```

```
    mov ax, KGDT_R3_DATA + RPL_MASK; \
\
    /* Skip debug registers and debug stuff */ \
    sub esp, 0x30; \
\
    /* Load the segment registers */ \
    mov ds, ax; \
    mov es, ax; \
\
    /* Set up frame */ \
    mov ebp, esp; \
\
    . . . . . .
\
    /* Get current thread */ \
    mov ecx, [fs:KPCR_CURRENT_THREAD]; \
    cld; \
\
    /* Flush DR7 */ \
    and dword ptr [ebp+KTRAP_FRAME_DR7], 0; \
\
    . . . . . .
\
    /* Set the Trap Frame Debug Header */ \
    SET_TF_DEBUG_HEADER
```

由 CPU 自动(或由前面的 push 指令)压入堆栈的出错代码是 32 位的，但是实际上只用其低 16 位，而高 16 位是保留不用的，所以这个宏操作一开头就将其高 16 位写成 0。

下面就是保存现场并形成“异常框架”、或曰“陷阱框架(Trap Frame)”。这些代码类似于 Linux 中的 SAVE_ALL，但略为复杂一些。

这里与段寄存器 FS 有关的几条指令值得特别注意。首先是把用户空间的 fs 保存在堆栈上，然后把 fs 设置成 KGDT_R0_PCR、即 0x30，这是因为当 CPU 运行于内核中时 fs 应选择其 KPCR 数据结构所在的段，从而使 fs:0 指向这 KPCR 结构的起点，这是 Windows 的一个约定。数值 0x30 表示所选择的是 GDT(而不是 LDT)，访问权限 RPL 为 0、即系统态，而用于 GDT 的下标则为 6。当然，GDT 中下标为 6 的表项保证了使 fs:0 指向 KPCR 结构的起点。于是，fs:[KPCR_EXCEPTION_LIST]就是这个 KPCR 结构中成分 Tib.ExceptionList 的值，即指向异常处理链表的指针。这是因为常数 KPCR_EXCEPTION_LIST 定义为 0，而 KPCR 结构中的第一个成分 Tib 是 KPCR_TIB 数据结构，ExceptionList 则是 KPCR_TIB 结构中的第一个字段。显然，之所以要把这个指针压入堆栈，是因为在异常处理的过程中可能改变这个指针。

然后把-1 压入堆栈，代码中的注释说这是“bogus previous mode”、即虚构的“先前模式”。发生异常时在内核堆栈上形成的框架相当于一个 KTRAP_FRAME 数据结构：

```
typedef struct _KTRAP_FRAME
```

```
{
    ULONG DbgEbp;
    . . . . . .
    ULONG TempSegCs;
    ULONG TempEsp;
    ULONG Dr0;
    . . . . . .
    ULONG Dr7;
    ULONG SegGs;
    . . . . . .
    ULONG PreviousPreviousMode;
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    ULONG SegFs;
    . . . . . .
    ULONG SegCs;
    ULONG EFlags;
    ULONG HardwareEsp;
    ULONG HardwareSegSs;
    . . . . . .
} KTRAP_FRAME, *PKTRAP_FRAME;
```

这里压入堆栈的-1 所占的位置就是结构中的 PreviousPreviousMode。真正的先前模式、即发生异常时的 CPU 模式，是可以从 SegCs 字段中获取的。这是因为段寄存器 CS 的最低两位构成 CPL 位段，而内核运行于 0 环、用户态运行于 3 环，所以只要最低位为非 0 就是运行于用户模式。而 PreviousPreviousMode，则是先前的先前模式。

在代码中，有关先前模式的定义是这样：

```
#define KernelMode                  0x0
#define UserMode                    0x1
#define MODE_MASK                   0x0001
```

既然是-1、即 0xffffffff，其最低位当然是 1。这样，如果从异常框架的这个字段获取先前模式，那么所得到的将是 UserMode。

后面的代码、特别是宏操作 SET_TF_DEBUG_HEADER，大多与 Debug 有关，这里就不多说了：

```
.macro SET_TF_DEBUG_HEADER
    /* Get the Debug Trap Frame EBP/EIP */
    mov ebx, [ebp+KTRAP_FRAME_EBP]
    mov edi, [ebp+KTRAP_FRAME_EIP]

    /* Write the debug data */
    mov [ebp+KTRAP_FRAME_DEBUGPOINTER], edx
    mov dword ptr [ebp+KTRAP_FRAME_DEBUGARGMARK], 0xBADB0D00
```

```
    mov [ebp+KTRAP_FRAME_DEBUGEBP], ebx
    mov [ebp+KTRAP_FRAME_DEBUGEIP], edi
.endm
```

回到_KiTrap0 的代码。构建好异常框架以后，就是对 KiTrapHandler()的调用，这就是公共的异常响应/处理程序的入口。这里的两个调用参数一个是异常号 0，另一个是指向异常框架的指针。完成了异常处理以后，如果返回的话，就根据 CPU 的工作模式分别转入 Kei386EoiHelper()或 KiV86Complete()，前者处理常规的 EOI、即中断返回，后者处理 VM86 模式下的中断返回。

如前所述，类似于_KiTrap0 的异常响应入口有 20 个之多，我们当然没有必要每个都看；但是_KiTrap14 却是要看一下的，因为 14 号异常是页面异常，它的处理是特殊的。

```
_KiTrap14:
    /* Enter trap */
    TRAP_PROLOG(14)

    /* Call the C exception handler */
    push 14
    push ebp
    call _KiPageFaultHandler
    add esp, 8

    /* Check for v86 recovery */
    cmp eax, 1

    /* Return to caller */
    jne _Kei386EoiHelper@0
    jmp _KiV86Complete
```

与_KiTrap0 的代码相比，这里就没有开头处的 push 指令了，因为对于 14 号异常 CPU 会自动压入出错代码。但是，更重要的不同之处在于，这里调用的是专门用于页面异常处理的 KiPageFaultHandler()、而不是公共的 KiTrapHandler()。这是因为，页面异常并不一定是不正常的，有些页面“异常”实际上是“正常”，例如缺页异常就是这样。所以，对页面异常的处理不同于其它异常。

[_KiTrap14() > KiPageFaultHandler()]

```
ULONG KiPageFaultHandler(PKTRAP_FRAME Tf, ULONG ExceptionNr)
{
    ULONG_PTR cr2;
    NTSTATUS Status;
    KPROCESSOR_MODE Mode;
```

```
    ASSERT(ExceptionNr == 14);
    /* Store the exception number in an unused field in the trap frame. */
    Tf->DbgArgMark = 14;
    /* get the faulting address */
    cr2 = Ke386GetCr2();
    Tf->DbgArgPointer = cr2;

    /* it's safe to enable interrupts after cr2 has been saved */
    if (Tf->EFlags & (X86_EFLAGS_VM|X86_EFLAGS_IF))
    {
        Ke386EnableInterrupts();
    }

    . . . . . .

    Mode = Tf->ErrCode & 0x4 ? UserMode : KernelMode;

    /* handle the fault */
    if (Tf->ErrCode & 0x1)
    {
        Status = MmAccessFault(Mode, cr2, FALSE);
    }
    else
    {
        Status = MmNotPresentFault(Mode, cr2, FALSE);
    }

    . . . . . .

    if (NT_SUCCESS(Status))
    {
        return 0;
    }

    /* Handle user exceptions differently */
    if (Mode == KernelMode)
    {
        return(KiKernelTrapHandler(Tf, 14, (PVOID)cr2));
    }
    else
    {
        return(KiUserTrapHandler(Tf, 14, (PVOID)cr2));
    }
}
```

参数 Tf 是个 KTRAP_FRAME 结构指针，指向堆栈上的异常框架。

发生 14 号异常时，控制寄存器 CR2 含有导致访问失败的 32 位线性目标地址，而失败的指令所在的地址则在堆栈上的异常框架中。

从代码中可以看到，对于因访问内存失败而引起的 14 号异常，首先是试图通过 MmAccessFault()或 MmNotPresentFault()解决，如果成功就直接返回了，不成功才继续往下执行 KiKernelTrapHandler()或 KiUserTrapHandler()。这就是内核底层对于异常的拦截和过滤，只有在由内核底层拦截或处理的范围之外的异常，才会在通过了这一层过滤之后到达由 SEH 机制安排的异常处理。那么哪一些访问内存失败是由内核底层处理的呢？例如目标页面不在内存而需要换入，再如对 COW 页面、即“写入时复制(Copy On Write)”页面的处理。在我们这个情景中，需要对付的是一般的页面无映射或访问权限不符，这就不是内核底层本身所能妥善处理的了，所以要根据发生异常时 CPU 所运行的空间而交给 KiUserTrapHandler()或 KiKernelTrapHandler()去处理。

还要注意，对于因访问内存失败而引起的 14 号异常，此时已经通过 Ke386EnableInterrupts()打开了中断。异常和中断的区别之一就是：CPU 在发生某些异常时并不自动关中断，在发生中断时则总是自动关中断。而具体到 14 号异常，却是自动关中断的。可是对于 14 号异常的处理可能是个时间很长的过程，而且后面我们将看到有可能就不返回了，所以这里要通过 Ke386EnableInterrupts()先将中断打开。不过这只能安排在读取了控制寄存器 CR2 的内容之后，否则就有可能丢失导致访问失败的地址。

相比之下，别的异常都不存在需要先由内核中的某些底层函数加以拦截、过滤的问题，所以都使用公共的函数 KiTrapHandler()。具体的代码这里就不看了，只是说明一点，这个函数最后同样也是根据 CPU 在发生异常时所运行的空间而调用 KiKernelTrapHandler()或者 KiUserTrapHandler()。

在我们现在这个情景中，异常发生于系统空间，所以是执行 KiKernelTrapHandler()：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler()]

```
ULONG
KiKernelTrapHandler(PKTRAP_FRAME Tf, ULONG ExceptionNr, PVOID Cr2)
{
  EXCEPTION_RECORD Er;

  Er.ExceptionFlags = 0;
  Er.ExceptionRecord = NULL;
  Er.ExceptionAddress = (PVOID)Tf->Eip;

  if (ExceptionNr == 14)
  {
      Er.ExceptionCode = STATUS_ACCESS_VIOLATION;
      Er.NumberParameters = 2;
      Er.ExceptionInformation[0] = Tf->ErrCode & 0x1;
      Er.ExceptionInformation[1] = (ULONG)Cr2;
  }
  else
```

```
  {
      if (ExceptionNr < ARRAY_SIZE(ExceptionToNtStatus))
      {
        Er.ExceptionCode = ExceptionToNtStatus[ExceptionNr];
      }
      else
      {
        Er.ExceptionCode = STATUS_ACCESS_VIOLATION;
      }
      Er.NumberParameters = 0;
  }
  /* FIXME: Which exceptions are noncontinuable? */
  Er.ExceptionFlags = 0;
  KiDispatchException(&Er, NULL, Tf, KernelMode, TRUE);
  return(0);
}
```

这个函数是由所有异常共用的，从这里的条件语句也可以看出 14 号异常的特殊性。

显然，这里的核心是对于 KiDispatchException()的调用，其余都是在为此而准备一个“异常纪录块”Er、即 EXCEPTION_RECORD 数据结构，其类型定义如下：

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

每一次异常都要有个异常纪录块，用来记录有关本次异常的信息，作为 SEH 机制处理本次异常的依据之一。注意这个异常纪录块存在于堆栈上 KiKernelTrapHandler()的函数调用框架里，所以是与这个函数调用框架共存亡的。

先简单介绍一下这数据结构中的几个字段。ExceptionCode 是异常代码，表明本次异常的性质或类型，对于页面访问异常是 STATUS_ACCESS_VIOLATION。而 ExceptionFlags 则是一些状态标志位，用来记录处理过程中的一些状态，注意这里设置的是全 0。换言之，凡是来自 KiKernelTrapHandler()的“硬异常”，其状态标志位为全 0。第三个字段 ExceptionRecord 是个指针，用来指向另一个异常纪录块。如果对于一次异常的处理本身又引起了“软异常”，也就是说有了异常的嵌套(不是 SEH 域的嵌套)、而且后一次异常为软件模拟的“软异常”，那么后一次异常的纪录块就通过这个指针指向前一次异常的纪录块。而对于来自 KiKernelTrapHandler()的“硬异常”，则这个指针为 NULL。至于 ExceptionAddress，则是引起异常的指令所在的地址、也即本次异常的返回地址。

此外，对于不同原因的异常，有关本次异常的其它信息在种类上和数量上都有不同，所以后面还有个数组 ExceptionInformation[]，而 NumberParameters 则说明该数组中有效数据的

个数。

对于 14 号异常的纪录块，在数组 ExceptionInformation[]里面提供了两项数据。第一项是出错代码中的最低位，这是一个标志位，称为 P 标志位，为 1 表示访问权限不符，为 0 表示缺页。前面分头调用 MmAccessFault()或 MmNotPresentFault()，所依据的也就是这一位。第二项来自控制寄存器 CR2，是导致访问失败的 32 位线性地址。显然，对于 14 号异常，程序既然能执行到这儿，就说明这并非“正常”、“良性”的页面异常。

还要注意，这个函数处理的是发生于系统空间的异常。对于 14 号异常，这是指发生异常时 CPU 正在执行系统空间的代码，而并不是指当时所访问的目标页面在系统空间。事实上，在我们这个情景中，所访问的页面恰恰是在用户空间。

另一方面，既然是运行于系统空间，在发生本次异常之前就已经在使用当前线程的系统空间堆栈。我们有必要搞清楚至此为止对系统空间堆栈的使用。

堆栈是从上向下伸展的，所以最上面是 CPU 从用户空间进入内核时的陷阱框架、即当时所保存的现场、或者说用户空间上下文。在这个框架下面是若干个函数调用框架，具体的个数取决于进入内核后函数调用的层次(为简单起见，我们不考虑发生于中断处理里面的异常，也暂不考虑发生于异常处理里面的嵌套异常)。所以，在典型的情况下，本次异常就发生在上述的最后一个、即最下方的函数调用框架中。由于发生了异常，在这个函数调用框架的下方就有了一个异常框架，并且在此以下又有了若干函数调用框架，我们目前就在调用 KiKernelTrapHandler()这个函数所形成的框架中。在常规的情况下，随着异常处理的进行，由于逐层的函数调用和返回，这个堆栈会动态地向下伸展和向上收缩。最后，当 CPU 从异常返回的时侯，异常框架就从堆栈上消失(丢弃)了，于是就回到了当初发生异常时所在的那个框架，这必定是当时堆栈最下方的那个框架，也即紧挨在异常框架上方的那个框架。

而对于通过 SEH 机制处理的异常，情况就不同了。此时的处理流程不再总是逐层返回，而是可能从异常框架下面的某个函数调用框架一下子“长程跳转”到异常框架上面的某个框架、即当初执行某一次_SEHSetJmp()时所在的那个框架，一下子把它下面包括异常框架在内的所有框架都废弃了。而前面通过_SEHSetJmp()设置的数据结构，以及异常处理链表等等，则都为此作好了准备。所谓“长程跳转(Long Jump)”，就是跨函数调用框架的跳转；如果是发生在同一个框架内部那就是普通跳转了。

在 SEH 框架嵌套的条件下，长程跳转还可能跨越若干 SEH 框架、即设置了其它 SEH 域的函数调用框架。例如，在上一篇漫谈所述的例子中，实际的异常可能发生在内层针对除数为 0 的 SEH 域中(同时也是在外层针对页面异常的 SEH 域中)；但是发生异常的原因却是访问内存、所以要长程跳转到当初设置了外层 SEH 域的函数中；此时就要跨越内层的 SEH 域。在堆栈上，这就体现为跨越并废弃更多的函数调用框架。

不过这是后话，现在还没有到可以作长程跳转的时候，我们继续往下看对于 KiDispatchException()的调用，注意这里的调用参数 KernelMode 是个常数、而不是变量。

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()]

```
VOID
NTAPI
KiDispatchException(PEXCEPTION_RECORD ExceptionRecord,
                    PKEXCEPTION_FRAME ExceptionFrame, PKTRAP_FRAME TrapFrame,
                    KPROCESSOR_MODE PreviousMode, BOOLEAN FirstChance)
{
  CONTEXT Context;
```

```
. . . . . .
BOOLEAN UserDispatch = FALSE;

/* Increase number of Exception Dispatches */
KeGetCurrentPrcb()->KeExceptionDispatchCount++;

/* Set the context flags */
Context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS;

/* Check if User Mode */
if (PreviousMode == UserMode)
{
   /* Add the FPU Flag */
   Context.ContextFlags |= CONTEXT_FLOATING_POINT;
   if (KeI386FxsrPresent) Context.ContextFlags |= CONTEXT_EXTENDED_REGISTERS;
}

/* Get a Context */
KeTrapFrameToContext(TrapFrame, ExceptionFrame, &Context);

/* Handle kernel-mode first, it's simpler */
if (PreviousMode == KernelMode)
{
   /* Check if this is a first-chance exception */
   if (FirstChance == TRUE)
   {
         /* Break into the debugger for the first time */
         Action = KdpEnterDebuggerException(ExceptionRecord, PreviousMode,
                                            &Context, TrapFrame, TRUE, TRUE);
         /* If the debugger said continue, then continue */
         if (Action == kdContinue) goto Handled;

         /* If the Debugger couldn't handle it, dispatch the exception */
         if (RtlDispatchException(ExceptionRecord, &Context))
         {
               /* It was handled by an exception handler, continue */
               goto Handled;
         }
   }

   /* This is a second-chance exception, only for the debugger */
   Action = KdpEnterDebuggerException(ExceptionRecord, PreviousMode,
                                      &Context, TrapFrame, FALSE, FALSE);
   /* If the debugger said continue, then continue */
```

```
    if (Action == kdContinue) goto Handled;

    /* Third strike; you're out */
    KEBUGCHECKWITHTF(KMODE_EXCEPTION_NOT_HANDLED,
                     ExceptionRecord->ExceptionCode,
                     (ULONG_PTR)ExceptionRecord->ExceptionAddress,
                     ExceptionRecord->ExceptionInformation[0],
                     ExceptionRecord->ExceptionInformation[1],
                     TrapFrame);
  }
  else
  {
        ......
  }

Handled:
  /* Convert the context back into Trap/Exception Frames */
  KeContextToTrapFrame(&Context,
                       NULL,
                       TrapFrame,
                       Context.ContextFlags,
                       PreviousMode);
  return;
}
```

先看调用参数。异常纪录块 ExceptionRecord 就是前面准备好的；指针 ExceptionFrame 是 NULL，这是个 KEXCEPTION_FRAME 结构指针，但是从代码中看这只是为 PowerPC 芯片而设的，用于保存一些附加寄存器的内容，386 架构的处理器芯片无此要求；而陷阱框架指针 Tf 指向堆栈上因异常而形成的框架，我们在前面倒是称之为“异常框架”。此外，PreviousMode 为 KernelMode；而 FirstChance 为 TRUE，表示即将进行的是第一次努力。

这个函数的代码中有很大一部分是用于用户空间异常的，因为 KiUserTrapHandler()最后也是调用这个函数。但是我们现在要集中关注系统空间异常，所以把那些代码删去了。

除前面准备好的异常纪录块外，对于异常的处理还需要用到陷阱框架中的许多信息，以及别的信息、例如有关浮点处理器的现场信息，所以在进入实质性的异常处理前要通过 KeTrapFrameToContext()将这些信息整理、收集到一个上下文数据结构中。完成了处理之后，如果要从 KiDispatchException()返回的话，则反过来通过 KeContextToTrapFrame()更新有关的原始信息，因为在异常处理的过程中可能会有所改变。

有些异常可能发生在程序调试的过程中，如果是这样就要由调试程序先进行处理，有的是由调试程序直接采取措施，有的是由调试人员决定采取什么措施。例如，由于程序断点所引起的异常(其实是自陷)，就只能由调试程序和调试人员采取措施。所以，只要是处于 Debug 状态，异常处理的第一步就是交由调试程序(debugger)处理。当然，对内核的调试不像对应用软件那么简单，但是道理是一样的。首先，所用的内核映像必须是可调试的版本，在编译、连接时就加上了调试可选项。另一方面，还得有个调试工具，那一般就是 kd、即“内核调试器(Kernel Debugger)”。这两个条件是缺一不可的。

通过 KdpEnterDebuggerException()把本次异常提交调试程序处理的结果有两种：

- 内核并不处在被调试的状态，或调试程序(以及调试人员)不能解决问题，本次异常需要由 SEH 机制加以处理，此时返回常数 kdHandleException。
- 调试程序已经解决了问题，可以继续运行、而不需要 SEH 处理的介入，此时返回常数 kdContinue。

如果返回的是 kdContinue，就跳转到程序标签 Handled 下面。此时数据结构 Context 中的内容可能已有改变，所以通过 KeContextToTrapFrame()对异常框架作相应修改，然后返回，本次异常就这样对付过去了。对于 Context 结构内容的改变，不妨设想一下，调试人员可能会选择让程序的执行跳转到另一个点上，或者修改某一个寄存器的内容，这时候就要修改上下文中的返回地址或寄存器映像。

从总体上说，对发生于系统空间的异常，内核分三步采取措施：

1. 第一步、即“FirstChance”，是先把问题提交给调试程序，如不能解决(返回的不是 kdContinue)就调用 RtlDispatchException()进行实质性的 SEH 处理。在实际运行中，处于调试状态的时候总是少数，所以在绝大部分的情况下第一步的核心就是 SEH 的处理。SEH 机制对异常的处理有三种可能的结果：
   - 如果被某个 SEH 框架所认领，并实施长程跳转，程序就不返回了，而此刻所在的函数调用框架也随同别的一些框架一起被跨越和废弃。
   - 被某个 SEH 框架所认领，但是认为可以不作长程跳转而继续运行(例如只需要执行一下善后函数就行)。这样，程序就会从 RtlDispatchException()返回，并且返回的值是 TRUE。此时问题已经解决，所以也是通过程序标签 Handled 下面的代码返回。
   - 所有的 SEH 框架都拒绝认领，这意味着处理本次异常的第一步努力已经失败。
2. 要是第一步失败，就进行第二步，再次通过 KdpEnterDebuggerException()把问题提交给调试程序。注意此时的最后两个调用参数都变成了 FALSE，而第一次时都是 TRUE。这两个参数，一个是 FirstChance，其意义自明；第二个是 Gdb，表示需要取得别的调试支持。如果这一次取得了成功、问题解决了，那么返回值是 kdContinue。否则就要采取第三步措施了。
3. 第三步，实际上此时已经无计可施，宏操作 KEBUGCHECKWITHTF()的作用是显示出错信息并将出错的现场“转储(Dump)”到文件中以便事后分析，然后就使 CPU 进入停机状态。

对于发生于系统空间的异常，这三步措施、或者说三次努力，都是在 KiDispatchException()内部完成的，如果调用参数 FirstChance 为 1 就一气呵成，但是在调用这个函数时也可以使 FirstChance 为 0 而跳过第一步。

而对于发生于用户空间的异常，则对于 ExceptionList 的扫描处理是在用户空间进行的，并且在用户空间也有一个类似于 KiDispatchException()；而有的措施却又需要通过系统空间才能进行，所以不一定能在同一个函数中一气呵成。

显然，SEH 处理的核心就是对 ExceptionList 的扫描处理，这是由 RtlDispatchException()完成的，事实上绝大多数的异常都可以通过这个函数得到妥善的处理。

下面就是由 RtlDispatchException()实现的实质性的 SEH 处理了。这种处理在有些文献中称为“基于框架(Frame-Based)”的异常处理，其基础当然就是 ExceptionList。

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException()]

```
BOOLEAN
NTAPI
RtlDispatchException(IN PEXCEPTION_RECORD ExceptionRecord,
                     IN PCONTEXT Context)
{
   PEXCEPTION_REGISTRATION_RECORD RegistrationFrame, NestedFrame = NULL;
   . . . . . .

   /* Get the current stack limits and registration frame */
   RtlpGetStackLimits(&StackLow, &StackHigh);
   RegistrationFrame = RtlpGetExceptionList();
   DPRINT("RegistrationFrame is 0x%p\n", RegistrationFrame);

   /* Now loop every frame */
   while (RegistrationFrame != EXCEPTION_CHAIN_END)
   {
      /* Find out where it ends */
      RegistrationFrameEnd = (ULONG_PTR)RegistrationFrame + sizeof(*RegistrationFrame);

      /* Make sure the registration frame is located within the stack */
      if ((RegistrationFrameEnd > StackHigh) ||
              ((ULONG_PTR)RegistrationFrame < StackLow) ||
              ((ULONG_PTR)RegistrationFrame & 0x3))
      {
          . . . . . .
          continue;
          . . . . . .
      }

      . . . . . .

      /* Call the handler */
      DPRINT("Executing handler: %p\n", RegistrationFrame->Handler);
      ReturnValue = RtlpExecuteHandlerForException(ExceptionRecord,
                                                   RegistrationFrame, Context, &DispatcherContext,
                                                   RegistrationFrame->Handler);
      DPRINT("Handler returned: %p\n", (PVOID)ReturnValue);

      /* Check if this is a nested frame */
      if (RegistrationFrame == NestedFrame)
      {
              /* Mask out the flag and the nested frame */
              ExceptionRecord->ExceptionFlags &= ~EXCEPTION_NESTED_CALL;
              NestedFrame = NULL;
```

```
}

/* Handle the dispositions */
if (ReturnValue == ExceptionContinueExecution)
{
        /* Check if it was non-continuable */
        if (ExceptionRecord->ExceptionFlags & EXCEPTION_NONCONTINUABLE)
        {
            /* Set up the exception record */
            ExceptionRecord2.ExceptionRecord = ExceptionRecord;
            ExceptionRecord2.ExceptionCode =
                                    STATUS_NONCONTINUABLE_EXCEPTION;
            ExceptionRecord2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
            ExceptionRecord2.NumberParameters = 0;

            /* Raise the exception */
            DPRINT("Non-continuable\n");
            RtlRaiseException(&ExceptionRecord2);
        }
        else
        {
            /* Return to caller */
            return TRUE;
        }
}
else if (ReturnValue == ExceptionNestedException)
{
        /* Turn the nested flag on */
        ExceptionRecord->ExceptionFlags |= EXCEPTION_NESTED_CALL;

        /* Update the current nested frame */
        if (NestedFrame < DispatcherContext) NestedFrame = DispatcherContext;
}
else if (ReturnValue == ExceptionContinueSearch)
{
        /* Do nothing */
}
else
{
        /* Set up the exception record */
        ExceptionRecord2.ExceptionRecord = ExceptionRecord;
        ExceptionRecord2.ExceptionCode = STATUS_INVALID_DISPOSITION;
        ExceptionRecord2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
        ExceptionRecord2.NumberParameters = 0;
```

```
                /* Raise the exception */
                RtlRaiseException(&ExceptionRecord2);
        }

        /* Go to the next frame */
        RegistrationFrame = RegistrationFrame->Next;
    }

    /* Unhandled, return false */
    DPRINT("FALSE\n");
    return FALSE;
}
```

SEH 的基础就是异常处理链表，正是这个队列中的节点及其内容使长程跳转成为可能。所以这里一开始就通过 RtlpGetExceptionList()找到异常处理链表，这当然就是当前 CPU 的 KPCR 结构中的指针 ExceptionList。这个指针指向链表中的第一个节点，其数据结构是 EXCEPTION_REGISTRATION_RECORD。这种数据结构的定义如下：

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

这和上一篇漫谈中所说的_SEHRegistration_t 结构实质上是同一回事，只是结构和成分的名称不同。这里的 Handler 是个函数指针，其类型为 PEXCEPTION_ROUTINE，也跟_SEHRegistration_t 结构中的_SEHFrameHandler_t 相同，因而这里见到的函数指针 handler 就是前面设置的函数指针 SER_Handler。之所以如此，笔者猜想，是因为有关的代码重用于用户空间的异常处理，因为历史的原因而使用了不同的名称。注意_SEHRegistration_t 数据结构是另一种数据结构_SEHPortableFrame_t 内部的第一个成分，所以获得了一个_SEHRegistration_t 结构指针，也就获得了指向其所在_SEHPortableFrame_t 结构的指针。

上一篇漫谈中还曾讲到，异常处理链表中的每一个数据结构都在堆栈上，都是相应函数调用框架中的局部量。所以这里还通过 RtlpGetStackLimits()获取当前线程的(系统空间)堆栈位置、即其 StackLow 和 StackHigh 两个端点；下面在处理队列中的数据结构时先加以比对，以确认其位置上的合理性。不过也有个例外，就是倘若异常发生于 DPC 处理(Windows 的 DPC 相当于 Linux 的 bh、或“软中断”)的过程中，那么由于 DPC 处理时使用的是一个独立的堆栈，所以就要对 StackLow 和 StackHigh 作出相应的调整，然后重新加以比对。当然，如果比对的结果确实不符，那就无法继续下去了。

然后就通过一个 while 循环来依次搜寻处理 ExceptionList 链表中的每一个节点，由 RtlpExecuteHandlerForException()加以尝试。链表中的每个节点都代表着一个局部 SHE 框架栈。前一篇漫谈中讲过，局部 SHE 框架栈就是一组不仅实质上嵌套、而且(代码)形式上也嵌套的 SEH 域所形成的框架。但是，事实上在现有的 ReactOS 代码中还没有见到使用形式嵌套的 SEH 域，所以 ExceptionList 中的每个节点实质上都只代表着单个的 SEH 域。因此，

为叙述上的方便，下面只要不至于引起误解或与代码冲突，就假定 ExceptionList 中的每个节点都只代表着一个 SEH 框架(而不是栈)。

由于异常处理链表是个后进先出的队列，里面的第一个节点(数据结构)代表着最近进入的 SEH 框架。如果链表中有不止一个的节点，就说明有了 SEH 框架嵌套。在嵌套的情况下，队列中的第一个节点代表着最底层的那个保护域，如果这个节点(执行过滤函数以后)拒绝认领本次异常，就说明并非这个 SEH 域所针对的异常，那就应该往上跑一层，看是否为上一层 SEH 域所针对的异常。所以顺着异常处理链表考察下一个节点就是在逐层往上跑，直至有某个节点认领本次异常为止。一个节点认领了实际发生的异常以后，一般就会执行预先规定的长程跳转，直接就跑到了那个框架中的 if 语句里面，而眼下这个函数的调用框架也就因为长程跳转而被丢弃了。不过，后面读者将看到，在长程跳转之前还有个“展开(Unwinding)”的过程，那就是调用所有被跨越 SEH 框架的善后函数。

所以，在正常条件下这个 while 循环注定是短命的。只有在队列中的每个 SHE 框架都拒绝认领所发生的异常、或者在处理中出错的情况下，这个 while 循环才可能以穷尽整个队列而告终。

如果 RtlpExecuteHandlerForException()返回，那么其返回值有这么几种可能：

```
#define ExceptionContinueExecution   0
#define ExceptionContinueSearch      1
#define ExceptionNestedException     2
#define ExceptionCollidedUnwind      3
```

这个返回值实际上是对下一步应该如何进行的指示。所以此后的程序大体上相当于一个以此为条件的 case 语句。

- ExceptionContinueExecution 表示认领了、但是不作长程跳转。这说明或者是问题已经解决，或者是可以忽略本次异常的发生，总之是可以继续执行原来的程序了。但是这里还有个条件，就是被异常所中断的程序还能够继续执行才行。这时候就要看异常纪录块 ExceptionRecord 中的 EXCEPTION_NONCONTINUABLE 标志位。如果为 1 就表示无法继续执行，所以通过 RtlRaiseException()引起一次类型为 STATUS_NONCONTINUABLE_EXCEPTION 的“软异常”。而若可以继续，则直接结束循环而返回，并最终从本次异常返回(此时本次异常的框架还在堆栈上)。注意此时函数返回的是 TRUE，表示问题已经解决，而若返回 FALSE 则表示失败。
- ExceptionContinueSearch 表示不予认领，应该继续考察队列里面的下一个节点、即上升到更高一层的 SEH 框架。这是之所以需要 ExceptionList、以及这里的 while 循环的原因。此时在本轮循环中不需要再做什么，前进到 ExceptionList 队列中的下一个节点就是了。
- ExceptionNestedException 则表示 RtlpExecuteHandlerForException()发现本次异常是一次嵌套异常，即发生于异常处理过程中的新的异常。下面读者将看到，为了捕捉嵌套异常，在每考察/处理一个 ExceptionList 中的某个节点时先要在 ExceptionList 链表的头部插入一个临时的“保护节点”，处理完目标节点后再将保护节点删去。这样，当嵌套异常发生时，原来对 ExceptionList 的扫描/处理被中断、而先要处理这新的异常，并因此而再次进入这里的 while 循环。显然此时最先受到考察的是那个临时的保护节点，而 ExceptionNestedException 就是在这个时候返回的，同时还通过参数 DispatcherContext 返回一个指针，指向该临时节点所保护的目标节点、即原来正在处理中的那个节点。在这样的情况下，代码中使局部量 NestedFrame 指向

所保护的目标节点，并使异常纪录块中的 EXCEPTION_NESTED_CALL 标志位设成 1，然后就继续在 ExceptionList 中往前搜索。一直到过了所保护的节点以后，才把这标志位以及 NestedFrame 清 0。这样，只要异常纪录块中的这个标志位为 1，就表示目前处于在处理过程中发生了嵌套异常的那个 SEH 框架内部。注意对嵌套异常的处理可能在 ExceptionList 中跑得更远，也即属于更高层的 SEH 框架。由于嵌套的深度有可能大于 1，实际的代码比之上述还要更复杂一些。

- 如果除此之外出现了别的返回值，包括 ExceptionCollidedUnwind，那就是发生了严重的错误。就其本质而言，这样的错误也相当于异常，ExceptionList 中理应也有相应的节点为此作好了准备，只是 CPU 的硬件不会因此而发生硬异常。所以，就通过 RtlRaiseException()引起一次类型为 STATUS_INVALID_DISPOSITION 的软异常。这当然是一次嵌套异常，因为原来的异常框架还在。

关于通过 RtlRaiseException()引起软异常的问题，下一篇漫谈中还要详细介绍，这里先简单提一下。所谓软异常，就是以函数调用的手段来模拟一次异常。在典型的情况下，每个 SEH 域都有一个过滤函数，过滤函数检查异常纪录块的类型以确定是否应该认领和处理本次异常。以上列的第一种情况为例，假定原来是访问内存失败而引起的 14 号异常，类型是 STATUS_ACCESS_VIOLATION，这已经得到了处理，结论是继续执行；然而状态标志位又表明无法继续。这样，问题已不再是原来访问内存失败的问题，而变成了类型为 STATUS_NONCONTINUABLE_EXCEPTION 的另一个问题，这就可能需要由针对此种异常的另一个 SEH 域来处理了，因而以此类型模拟一次异常，使 SEH 机制再次搜索 ExceptionList。这就是发起软异常的意图。

最后，如果 while()循环结束，那就说明异常处理链表中所有的节点都不是为本类异常准备的，也即事先并没有估计到本类异常的发生、也没有为此作出安排，所以就返回 FALSE，让上一层 KiDispatchException()采取其第二步措施。

显然，关键在于 RtlpExecuteHandlerForException()，就是对 ExceptionList 中具体节点的考察和处理。先看其调用界面：

```
EXCEPTION_DISPOSITION
RtlpExecuteHandlerForException(PEXCEPTION_RECORD ExceptionRecord,
                PEXCEPTION_REGISTRATION RegistrationFrame, PCONTEXT Context,
                PVOID DispatcherContext, PEXCEPTION_HANDLER ExceptionHandler);
```

其中第一个参数指向异常纪录块，这是关于本次(实际发生的)异常的信息；第二个参数指向 ExceptionList 队列中的一个节点，这是关于当前所考察 SEH 域的信息；第三个参数指向本次异常发生时的上下文数据结构。第四个参数 DispatcherContext 是个指针，仅对用于嵌套异常的临时保护节点有效。最后一个参数是函数指针，指向由当前节点提供的框架处理函数。对于普通的节点这就是_SEHFrameHandler()，是在_SEHEnterFrame_f()中设置好的。

再看具体的实现，这是一段汇编代码。

```
[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
  > RtlpDispatchException() > RtlpExecuteHandlerForException()]

_RtlpExecuteHandlerForException@20:
    /* Copy the routine in EDX */
    mov edx, offset _RtlpExceptionProtector
```

```
    /* Jump to common routine */
    jmp _RtlpExecuteHandler@20
```

首先让寄存器 EDX 指向一个函数 RtlpExceptionProtector()，其作用下面就会看到。然后跳转到标签_RtlpExecuteHandler 下面。事实上，_RtlpExecuteHandler 下面这一段代码是由 RtlpExecuteHandlerForException()与另一个函数 RtlpExecuteHandlerForUnwind()共用的，所不同的只是置入 EDX 的函数指针。

```
_RtlpExecuteHandlerForUnwind@20:
    /* Copy the routine in EDX */
    mov edx, offset _RtlpExceptionProtector
    /* Run the common routine */
_RtlpExecuteHandler@20:
    /* Save non-volatile */
    push ebx
    push esi
    push edi
    /* Clear registers */
    xor eax, eax
    xor ebx, ebx
    xor esi, esi
    xor edi, edi

    /* Call the 2nd-stage executer */
    push [esp+0x20]
    push [esp+0x20]
    push [esp+0x20]
    push [esp+0x20]
    push [esp+0x20]
    call _RtlpExecuteHandler2@20

    /* Restore non-volatile */
    pop edi
    pop esi
    pop ebx
    ret 0x14
```

按理说，在 RtlpExecuteHandlerForUnwind()里面置入 EDX 的指针应该指向 RtlpUnwindProtector()、而不是像在 RtlpExecuteHandlerForException()里面那样指向 RtlpExceptionProtector()。所以这很可能是个错误。毕竟 0.3.0 版的 ReactOS 还很新，里面有些错误也不奇怪。而在 0.2.6 版的代码中则确实指向 RtlpUnwindProtector()，那应该是正确的。

显然，具体的处理是由_RtlpExecuteHandler2()实现的，这里只是为这个函数的调用进行事先的准备和事后的恢复。

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()]

_RtlpExecuteHandler2@20:

```
    /* Set up stack frame */
    push ebp
    mov ebp, esp
    /* Save the Frame */
    push [ebp+0xC]                          /* 指向原节点、这是要保护的目标节点 */
    /* Push handler address */
    push edx                                /* 成为新节点中的Handler指针，指向保护函数 */

    /* Push the exception list */
    push [fs:TEB_EXCEPTION_LIST]            /* 成为新节点中的 Next 指针 */
    /* Link us to it */
    mov [fs:TEB_EXCEPTION_LIST], esp        /* 让 ExceptionList 指向新节点 */

    /* Call the handler */
    push [ebp+0x14]
    push [ebp+0x10]
    push [ebp+0xC]
    push [ebp+8]
    mov ecx, [ebp+0x18]                     /* 参数 ExceptionHandler */
    call ecx                                /* 调用 ExceptionHandler，4 个调用参数 */

    /* Unlink us */
    mov esp, [fs:TEB_EXCEPTION_LIST]
    /* Restore it */
    pop [fs:TEB_EXCEPTION_LIST]             /* 新节点已从 ExceptionList 中摘除 */

    /* Undo stack frame and return */
    mov esp, ebp                            /* 新节点不复存在于堆栈上 */
    pop ebp
    ret 0x14
```

这里的常数 TEB_EXCEPTION_LIST 定义为 0，所以“fs:TEB_EXCEPTION_LIST”就是“fs:0”，即指向 KPCR 结构中的第一个字段，即 ExceptionList。但是在这里引用常数 TEB_EXCEPTION_LIST 有些误导，因为现在这是在系统空间、而不是在用户空间。究其原因，则是这个函数同样也用于用户空间的异常处理(代码重用)，而 TEB 的第一个字段确实同样也是 ExceptionList。

注意这里的 call 指令所引用的是 ECX、而不是 EDX。ECX 的内容来自堆栈上的调用参数，就是前面的最后一个参数 ExceptionHandler。对于普通的节点(下面就要讲到不普通的节点)，这实际上是_SEHFrameHandler()。

这段代码有点奥妙。注意这里先把作为参数的指针 RegistrationFrame 压入堆栈，再把寄

存器 EDX 的内容、即指向_RtlpExceptionProtector()或_RtlpUnwindProtector()的函数指针压入堆栈，然后又把[fs:0]、即指针 ExceptionList 的内容压入堆栈，再把当前的堆栈指针写入 ExceptionList。这样一来，ExceptionList 所指处的内容是一个_SEHRegistration_t 结构指针，这个指针的上方是一个函数指针(再上方是指向目标节点的指针)。我们可以把这两个指针看成一个_SEHPortableFrame_t 数据结构，而这里对堆栈和[fs:0]的操作，则实际上是把又一个节点插入了异常处理队列的头部(逻辑上则是尾部)。但是，与这个队列中原有的节点相比，这个新的节点又有所不同。原来的节点虽然同为_SEHRegistration_t 数据结构，却都是_SEHPortableFrame_t 数据结构中的一个成分；而现在挂入队列的节点却是孤立地存在(外加一个指针)。但是这并不成为问题，因为现在这个函数指针所指向的函数也不一样。毕竟，是这个函数决定了怎样去访问和使用有关的数据结构，只要二者配套就行。为区别于目标节点中的框架处理函数，我们称现在这个函数为“保护函数(protector)”。与此相应，我们不妨称这样的节点为“保护节点”，而称原来的节点为“普通节点”。

保护节点的存在是暂时的，从保护函数返回时下面的两条指令就把这个新的节点删除了，于是 ExceptionList 又恢复了原状。由于节点存在于堆栈上，有关的队列操作就很干净利索。后面读者就会看到，保护节点不会执行长程跳转，所以一定会返回。

那么为什么要在 ExceptionList 中插入一个保护节点呢？这是因为，下面就要对一个普通节点执行_SEHFrameHandler()了，执行这个函数的过程本身(例如对过滤函数的调用)有可能会引起新的异常，所以也应该把它保护起来、为可能发生的异常作好准备。而相应的保护函数，如果异常果真发生的话，则是_RtlpExceptionProtector()。其实这个函数并不真的起什么保护作用，其目的只是用来表明发生了嵌套异常。

在异常处理的过程中又发生新的异常，这就是嵌套异常。但是读者要注意嵌套异常和嵌套 SEH 域的区别，不要混淆。

万一真的发生了嵌套异常，则 CPU 照样还是通过上述的路线经 RtlDispatchException()进入 RtlpExecuteHandlerForException()，并且有了一个新的异常纪录块。但是实际的调用参数却不同了：首先异常纪录块是新的，并且此时的 RegistrationFrame 指向临时的保护节点，而保护节点的处理函数 RegistrationFrame->handler 则指向 RtlpExceptionProtector()、而不再是_SEHFrameHandler()。在深入到普通节点的处理函数之前，我们不妨先看一下 RtlpExceptionProtector()的代码：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> RtlpExceptionProtector()]

_RtlpExceptionProtector:

```
    /* Assume we'll continue */
    mov eax, ExceptionContinueSearch
    /* Put the exception record in ECX and check the Flags */
    mov ecx, [esp+4]                    /*  指向异常纪录块  */
    test dword ptr [ecx+EXCEPTION_RECORD_EXCEPTION_FLAGS],
                                                    EXCEPTION_UNWIND
    jnz return

    /* Save the frame in ECX and Context in EDX */
    mov ecx, [esp+8]                    /*  使 ECX 指向当前节点，这是一个保护节点  */
```

```
    mov edx, [esp+16]                /* 使 EDX 指向第四个参数 DispatcherContext */
    /* Get the nested frame */
    mov eax, [ecx+8]                 /* 当前节点中的第三个指针，指向所保护的节点 */
    /* Set it as the dispatcher context */
    mov [edx], eax                   /* 通过第四个参数返回指向所保护节点的指针 */
    /* Return nested exception */
    mov eax, ExceptionNestedException
return:
    ret 16
```

这里的常数 EXCEPTION_UNWIND 定义为

(EXCEPTION_UNWINDING + EXCEPTION_EXIT_UNWIND)。

所以，代码中的 test 指令所测试的是两个标志位，只要其中有任何一个为 1 就执行转移指令，从而返回常数 ExceptionContinueSearch。否则就返回 ExceptionNestedException，并将第二个参数、即指针 RegistrationFrame、复制到第四个参数 DispatcherContext 所指的地方。

前面 RtlDispatchException()中的 while 循环显然在扫描 ExceptionList，这是为了替新发生的异常寻找能够认领、处理本次异常的节点。凡是新发生的异常，其异常纪录块中的上述两个标志位都是 0，所以 RtlpExceptionProtector()会返回 ExceptionNestedException，这就表明发生了嵌套异常。那么在什么时候会返回 ExceptionContinueSearch 呢？后面读者将看到，在执行长程跳转之前，还有个“展开”的过程，在此过程中又要扫描 ExceptionList 中位于目标节点之前的那些节点，以调用它们的善后函数。到那时候，就至少要把异常纪录块中的标志位 EXCEPTION_UNWINDING 设成 1，于是这个函数就返回 ExceptionContinueSearch 了，因为保护节点本身是不具备认领和处理一次异常的能力的。

所以，真正解决问题还得靠代表着 SEH 框架的普通节点，而普通节点的处理函数是 _SEHFrameHandler()：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException() > RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2() > _SEHFrameHandler()]

```
static int __cdecl
_SEHFrameHandler(struct _EXCEPTION_RECORD * ExceptionRecord,
        void * EstablisherFrame, struct _CONTEXT * ContextRecord, void * DispatcherContext)
{
  _SEHPortableFrame_t * frame;

  _SEHCleanHandlerEnvironment();
  frame = EstablisherFrame;

  if(ExceptionRecord->ExceptionFlags & (4 | 2))   /* Unwinding */
    _SEHLocalUnwind(frame, NULL);
  else   /* Handling */
  {
    int ret;
```

```
_SEHPortableTryLevel_t * trylevel;

if(ExceptionRecord->ExceptionCode)
   frame->SPF_Code = ExceptionRecord->ExceptionCode;
else
   frame->SPF_Code = 0xC0000001;

for (trylevel = frame->SPF_TopTryLevel; trylevel != NULL; trylevel = trylevel->SPT_Next)
{
   _SEHFilter_t pfnFilter = trylevel->SPT_Handlers->SH_Filter;

   switch((UINT_PTR)pfnFilter)
   {
      case (UINT_PTR)_SEH_STATIC_FILTER(_SEH_EXECUTE_HANDLER):
      case (UINT_PTR)_SEH_STATIC_FILTER(_SEH_CONTINUE_SEARCH):
      case (UINT_PTR)_SEH_STATIC_FILTER(_SEH_CONTINUE_EXECUTION):
      {
          ret = (int)((UINT_PTR)pfnFilter) - 2;
          break;
      }

      default:
      {
          if(trylevel->SPT_Handlers->SH_Filter)
          {
             EXCEPTION_POINTERS ep;
             ep.ExceptionRecord = ExceptionRecord;
             ep.ContextRecord = ContextRecord;
             ret = pfnFilter(&ep, frame);
          }
          else
             ret = _SEH_CONTINUE_SEARCH;
          break;
      }
   }

   if(ret < 0)    /* _SEH_CONTINUE_EXECUTION */
      return ExceptionContinueExecution;
   else if(ret > 0)    /* _SEH_EXECUTE_HANDLER */
      _SEHCallHandler(frame, trylevel);
   else    /* _SEH_CONTINUE_SEARCH */
      continue;
}    /* end for */
/* FALLTHROUGH */
```

```
    }    /* end if */

    return ExceptionContinueSearch;
}
```

这个函数实际上把本来是两个函数的代码合在了一起，成为一个 if 语句，这也跟在异常处理的整个过程中要先后两次扫描 ExceptionList 有关。如果异常纪录块中的标志位 EXCEPTION_UNWINDING 或 EXCEPTION_EXIT_UNWIND 为 1，就说明这个 SEH 框架已经在“展开”的过程中，这是为展开而调用本节点的这个函数，所以调用_SEHLocalUnwind()。否则便是在搜寻能够认领和处理本次异常的 SEH 框架的过程中，是从 RtlpDispatchException()中经由 RtlpExecuteHandlerForException()调用下来的。所处的阶段不同，调用的路线也就不同，这里的操作也就随之而不同。此刻我们是在后一种情景中。

所谓搜寻 SEH 框架，就是依次执行节点中各局部 SEH 框架的过滤函数，看看是否应该执行这个 SEH 框架的长程跳转，如果是就加以实施，不是就加以拒绝。如果节点中的所有 SEH 框架都拒绝，就退回 RtlpDispatchException()继续考察链表中的下一个节点，如果不再有下一个节点，搜寻就失败了。

过滤函数根据什么信息确定是否应该执行实施函数进行长程跳转呢？我们在前面看到，作为参数传下来的异常纪录块中有异常号码、发生异常的指令所在地址等等信息。如果程序员的用意是让一个 SEH 域只针对某种特定种类的异常，就可以为此提供一个过滤函数(见代码中和数据结构中的函数指针 frame->SPF_Handlers->SH_Filter)，在这个函数中检查、比对异常纪录块中的有关信息，并返回代表着三种行为之一的常数，例如_SEH_EXECUTE_HANDLER。所以，过滤函数就好像是“频道选择器”一样。另一方面，如果程序员并不提供特定的过滤函数，那就可以直接指定为三种行为之一。如果既无特定的过滤函数，所指定的又非三种默认行为之一，那就默认为_SEH_CONTINUE_SEARCH，即跳过本节点、继续搜寻。

在我们这个情景中，当时指定的是_SEH_EXECUTE_HANDLER。从代码中可见，这意味着认领并执行_SEHCallHandler()，最后则实施长程跳转而不再返回。而其余两种选择，则 ExceptionContinueSearch 表示拒绝认领，ExceptionContinueExecution 表示认领但无需处理。

既然决定认领并处理本次异常，下面就是具体的处理了。如前所述，具体的处理包括展开和长程跳转两个步骤，我们顺着执行的流程往下看_SEHCallHandler()。

```
[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> _SEHFrameHandler() > _SEHCallHandler()]

static void __cdecl
_SEHCallHandler(_SEHPortableFrame_t * frame, _SEHPortableTryLevel_t * trylevel)
{
  _SEHGlobalUnwind(frame);
  _SEHLocalUnwind(frame, trylevel);
  frame->SPF_Handler(trylevel);
}
```

参数 frame 指向目标节点的_SEHPortableFrame_t 数据结构，而 trylevel 指向节点中代表着具体 SEH 框架的_SEHPortableTryLevel_t 数据结构。

所谓展开，就是要逐层调用长程跳转中所跨越的那些 SEH 框架的善后函数，目的主要是防止资源泄漏。而所跨越的那些 SEH 框架，则包括：

- ExceptionList 中位于目标节点之前的所有(普通)节点中的 SEH 框架。
- 目标节点中位于目标 SEH 框架之前的所有 SEH 框架。

前者的展开由_SEHGlobalUnwind()完成，后者的展开由_SEHLocalUnwind()完成。

之所以有 SEH 框架会被跨越，必定是因为那些 SEH 框架拒绝认领和处理本次异常，这是它们的过滤函数在起作用。尽管在我们这个情景中并没有过滤函数，因而不可能跨越 SEH 框架，也就谈不上展开的问题，但我们还是应该看一下这是怎么实现的。

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException() > RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2() > _SEHFrameHandler() > _SEHCallHandler() > _SEHGlobalUnwind()]

__SEHGlobalUnwind:

```
  extern __SEHRtlUnwind

; RtlUnwind clobbers all the "don't clobber" registers, so we save them
  push ebx
  mov ebx, [esp+8]
  push esi
  push edi

  push dword 0x0                        ; ReturnValue
  push dword 0x0                        ; ExceptionRecord
  push dword .RestoreRegisters          ; TargetIp
  push ebx                              ; TargetFrame
  call [__SEHRtlUnwind]

.RestoreRegisters:
  pop edi
  pop esi
  pop ebx
  ret
```

代码中的_SEHRtlUnwind 是一个函数指针，实际的处理是通过这个指针所指向的函数完成的。调用时有 4 个参数，其中 TargetFrame 就是从 SEHCallHandler()传下来的指针，指向 ExceptionList 中的一个节点；第二个参数 TargetIp 来自 RestoreRegisters，这就是 call 指令下面的标签，实际上就是这标签所在处的地址。第三个参数、即指针 ExceptionRecord 为 0；第四个参数 ReturnValue 也是 0。

函数指针_SEHRtlUnwind 则固定设置成指向 RtlUnwind()

```
void const * _SEHRtlUnwind = RtlUnwind;
```

所以 SEHGlobalUnwind()只是个包装、过渡，目的在于保存和恢复几个寄存器的内容，并为 RtlUnwind()的调用补充提供几个参数，实际的操作由 RtlUnwind()完成：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException() > RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2() > _SEHFrameHandler() > _SEHCallHandler() > _SEHGlobalUnwind() > RtlUnwind()]

```
VOID NTAPI
RtlUnwind(PVOID RegistrationFrame OPTIONAL, PVOID ReturnAddress OPTIONAL,
          PEXCEPTION_RECORD ExceptionRecord OPTIONAL, PVOID EaxValue)
{
    . . . . . .
EXCEPTION_RECORD ExceptionRecord2, ExceptionRecord3;
CONTEXT LocalContext;
    . . . . . .

    /* Get the current stack limits */
    RtlpGetStackLimits(&StackLow, &StackHigh);
    /* Check if we don't have an exception record */
    if (!ExceptionRecord)
    {
        /* Overwrite the argument */
        ExceptionRecord = &ExceptionRecord3;

        /* Setup a local one */
        ExceptionRecord3.ExceptionFlags = 0;
        ExceptionRecord3.ExceptionCode = STATUS_UNWIND;
        ExceptionRecord3.ExceptionRecord = NULL;
        ExceptionRecord3.ExceptionAddress = RtlpGetExceptionAddress();
        ExceptionRecord3.NumberParameters = 0;
    }

    /* Check if we have a frame */
    if (RegistrationFrame)
    {
        /* Set it as unwinding */
        ExceptionRecord->ExceptionFlags |= EXCEPTION_UNWINDING;
    }
    else
    {
        /* Set the Exit Unwind flag as well */
        ExceptionRecord->ExceptionFlags |= (EXCEPTION_UNWINDING |
                                            EXCEPTION_EXIT_UNWIND);
```

```
}

/* Now capture the context */
Context = &LocalContext;
LocalContext.ContextFlags = CONTEXT_INTEGER |
                                    CONTEXT_CONTROL | CONTEXT_SEGMENTS;
RtlpCaptureContext(Context);

/* Pop the current arguments off */
Context->Esp += sizeof(RegistrationFrame) + sizeof(ReturnAddress) +
                        sizeof(ExceptionRecord) + sizeof(ReturnValue);

/* Set the new value for EAX */
Context->Eax = (ULONG)EaxValue;

/* Get the current frame */
RegistrationFrame2 = RtlpGetExceptionList();

/* Now loop every frame */
while (RegistrationFrame2 != EXCEPTION_CHAIN_END)
{
      DPRINT("RegistrationFrame is 0x%p\n", RegistrationFrame2);

      /* If this is the target */
      if (RegistrationFrame2 == RegistrationFrame)
      {
            /* Continue execution */
            ZwContinue(Context, FALSE);
      }

      /* Check if the frame is too low */
      if ((RegistrationFrame) && ((ULONG_PTR)RegistrationFrame <
                                          (ULONG_PTR)RegistrationFrame2))
      {
            /* Create an invalid unwind exception */
            ExceptionRecord2.ExceptionCode = STATUS_INVALID_UNWIND_TARGET;
            ExceptionRecord2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
            ExceptionRecord2.ExceptionRecord = ExceptionRecord;
            ExceptionRecord2.NumberParameters = 0;

            /* Raise the exception */
            DPRINT1("Frame is invalid\n");
            RtlRaiseException(&ExceptionRecord2);
      }
```

```
/* Find out where it ends */
RegistrationFrameEnd = (ULONG_PTR)RegistrationFrame2 +
                                                    sizeof(*RegistrationFrame2);

/* Make sure the registration frame is located within the stack */
if ((RegistrationFrameEnd > StackHigh) ||
    ((ULONG_PTR)RegistrationFrame < StackLow) ||
    ((ULONG_PTR)RegistrationFrame & 0x3))
{
    . . . . . .
}
else
{
    /* Call the handler */
    DPRINT("Executing unwind handler: %p\n", RegistrationFrame2->Handler);
    ReturnValue = RtlpExecuteHandlerForUnwind(ExceptionRecord,
                                  RegistrationFrame2, Context, &DispatcherContext,
                                  RegistrationFrame2->Handler);
    DPRINT("Handler returned: %p\n", (PVOID)ReturnValue);

    /* Handle the dispositions */
    if (ReturnValue == ExceptionContinueSearch)
    {
        /* Do nothing */
    }
    else if (ReturnValue == ExceptionCollidedUnwind)
    {
        /* Get the previous frame */
        RegistrationFrame2 = DispatcherContext;
    }
    else
    {
        /* Set up the exception record */
        ExceptionRecord2.ExceptionRecord = ExceptionRecord;
        ExceptionRecord2.ExceptionCode = STATUS_INVALID_DISPOSITION;
        ExceptionRecord2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
        ExceptionRecord2.NumberParameters = 0;

        /* Raise the exception */
        RtlRaiseException(&ExceptionRecord2);
    }

    /* Go to the next frame */
```

```
                OldFrame = RegistrationFrame2;
                RegistrationFrame2 = RegistrationFrame2->Next;

                /* Remove this handler */
                if (RegistrationFrame2 != RegistrationFrame)
                {
                    RtlpSetExceptionList(OldFrame);
                }
            }
        }   /* end while */

        /* Check if we reached the end */
        if (RegistrationFrame == EXCEPTION_CHAIN_END)
        {
            /* Unwind completed, so we don't exit */
            ZwContinue(Context, FALSE);
        }
        else
        {
            /* This is an exit_unwind or the frame wasn't present in the list */
            ZwRaiseException(ExceptionRecord, Context, FALSE);
        }
}
```

先看调用参数。参数 RegistrationFrame 指向需要“展开”的 SEH 框架所在的节点，目的是要调用 ExceptionList 中此前所有节点中的所有 SEH 框架的善后函数。参数 ReturnAddress 是这个函数的返回地址，就是前面_SEHGlobalUnwind()代码中标签.RestoreRegisters 所在的地址。

参数 ExceptionRecord 指向本次异常的异常纪录块，但是从前面_SEHGlobalUnwind()传下来的实参是 0，所以这里另外创建了一个异常纪录块，这个纪录块也在堆栈上。所以，现在指针 ExceptionRecord 所指向的已经不是原先从 KiDispatchException()一路走来的那个异常纪录块了。纪录块中的 EXCEPTION_UNWINDING 标志位被设成了 1，表示已经进入了“展开”阶段。

至于 EaxValue，则其类型虽然是 PVOID，实际上却只是一个无符号整数，从前面_SEHGlobalUnwind()传下来的实参也是 0。

这个函数的代码比较长，但是执行的主线其实很简单：通过 while 循环对于 ExceptionList 链表中位于目标节点之前的各个节点执行 RtlpExecuteHandlerForUnwind()，以释放沿途所有 SEH 域所在函数调用框架的有关资源，因为这些框架即将因长程跳转而被跨越和丢弃。

在 while 循环开始之前还有一项准备工作，就是创建了一个上下文数据结构 LocalContext，并使指针 Context 指向这个数据结构。上下文的内容通过 RtlpCaptureContext()获取，但是经过了一些调整。所以，展开阶段使用的异常纪录块和上下文都与搜寻阶段不同。

这里通过 RtlpCaptureContext()获取的基本上是上一层函数调用框架的上下文，即_SEHGlobalUnwind()内部、调用 RtlUnwind()前夕的现场，但是把所有通用寄存器的映像都清成了 0。而这里所作的调整，则是对堆栈指针 ESP 映像的修改。本来 Context->Esp 指向当

前函数返回地址所在的位置，现在则使其跳过 4 个调用参数所在的单元，把它拨回到调用 RtlUnwind()前夕的位置上。之所以需要这个上下文数据结构，主要是为了后面执行 ZwContinue()的需要。

下面就是 while 循环了。从程序上看，这个 while 循环与搜寻阶段 RtlDispatchException()里面的那个 while 循环其实很相像。

如前所述，所谓展开的过程局限于目标节点之前，目标节点本身是不在其内的。所以，如果循环到了目标节点本身，这循环就应该结束了，所以此时通过 ZwContinue()回到由 Context 所描述的上下文中，实际上就是上一层函数调用框架中、即_SEHGlobalUnwind()内部。可是为什么不直接用 retern 语句返回呢？这可能是为了把一些通用寄存器的内容清 0，而正常的函数返回则只改变 EAX 的值。

与在 RtlDispatchException()中一样，这里也有对于节点位置合理性的检查。ExceptionList 中各个节点的数据结构都是某个函数的局部量，都存在于堆栈上，后进入 ExceptionList 的节点显然应该有更低的地址，因为堆栈是向下伸展的。如果颠倒了过来，那当然是出了问题，所以要通过 RtlRaiseException()启动一次类型为 STATUS_INVALID_UNWIND_TARGET 的软异常。另一方面，异常也有可能是发生在 DPC 处理的过程中，而 DPC 处理使用的是一个独立的堆栈。但是，如果一个节点的数据结构既不在当前线程的堆栈上，也不在 DPC 堆栈上，那就又有问题了。

排除了这些因素，下面的 else 部分就是对需要在长程跳转中跨越的节点的处理。首先就是 RtlpExecuteHandlerForUnwind()。读者已经在前面看到过这个函数的代码。

就如在搜寻阶段一样，在处理一个具体节点的展开之前，也要先在 ExceptionList 的前面插入一个临时的保护节点，处理完以后再将其删除。不过，这次要保护的是发生于展开过程中的嵌套异常，所以保护函数是 RtlpUnwindProtector()。同样，展开阶段的临时保护节点也针对着具体的普通节点。当 CPU 从 RtlpExecuteHandlerForUnwind()返回时，临时的保护节点已经被删除。

从代码中看，RtlpExecuteHandlerForUnwind()的返回值应该是 ExceptionContinueSearch 或 ExceptionCollidedUnwind 这二者之一，否则便是错误：

- 如果返回值是 ExceptionContinueSearch，就表示对该节点的善后处理已经完成，因此通过 RtlpSetExceptionList()从 ExceptionList 中摘除这个节点。注意节点的数据结构只是存在于堆栈上，所以不需要释放。
- 从代码中看，并猜测其意图：如果返回值是 ExceptionCollidedUnwind，就表示企图展开的节点是个针对展开过程的保护节点。此时 RtlpExecuteHandlerForUnwind()通过调用参数 DispatcherContext 返回一个指针，指向该节点所保护的、正在展开的那个节点。笔者对于为何此时会返回 ExceptionCollidedUnwind 感到困惑，下面还要讲到这个问题。
- 否则就是出了问题，所以通过 RtlRaiseException()启动一次软异常，类型为 STATUS_INVALID_DISPOSITION。

这里还要说明一下，在展开阶段的 while 循环中必然会碰上针对搜寻阶段的保护节点，因为搜寻阶段对目标节点的处理此时尚未结束，针对目标节点的保护节点还在 ExceptionList 中。从前面 RtlpExceptionProtector()的代码中可以看出，此时返回的是 ExceptionContinueSearch，读者不妨回过去结合 RtlDispatchException()的代码看一下。

顺便再看一下 RtlpUnwindProtector()的代码：

_RtlpUnwindProtector:

```
 /* Assume we'll continue */
```

```
 mov eax, ExceptionContinueSearch

 /* Put the exception record in ECX and check the Flags */
 mov ecx, [esp+4]                          /*  使 ECX 指向异常纪录块  */
 test dword ptr [ecx+EXCEPTION_RECORD_EXCEPTION_FLAGS], EXCEPTION_UNWIND
 jnz .return

 /* Save the frame in ECX and Context in EDX */
 mov ecx, [esp+8]                          /*  使 ECX 指向当前节点  */
 mov edx, [esp+16]                         /*  使 EDX 指向参数 DispatcherContext */
 /* Get the nested frame */
 mov eax, [ecx+8]                          /*  使 EAX 指向所保护的节点  */
 /* Set it as the dispatcher context */
 mov [edx], eax                            /*  返回指向所保护节点的指针  */
 /* Return collided unwind */
 mov eax, ExceptionCollidedUnwind
.return:
 ret 16
```

可见，如果是在展开的过程中碰上了展开阶段的保护节点，由于表示展开过程的两个标志位至少有一个是 1，所以就直接返回 ExceptionContinueSearch，因为这个节点没有善后函数可调用。反之，如果这两个标志位都是 0，就返回 ExceptionCollidedUnwind。

在 ReactOS 的代码中，可能返回 ExceptionCollidedUnwind 的只有这么一次，所以只有在遇上展开过程保护节点时才有可能。而在展开的过程中又遇上展开过程保护节点，则只能是因为展开某个节点的过程引起了嵌套的异常，这嵌套的异常又被 ExceptionList 中的某个节点认领和处理，并因此而启动了一个新的、嵌套的展开过程，这才会遇上前一次展开过程中插入队列的保护节点。

可是什么情况下异常纪录块中的这两个标志位都是 0 呢？按理说只可能是在从 RtlDispatchException()的 while 循环中通过 RtlpExecuteHandlerForException()处理这个节点时才有可能，因为在 RtlUnwind()的 while 循环开始之前至少已把这两个标志位之一设成了 1。如果是这样，则针对返回 ExceptionCollidedUnwind 所作的处理不应该在 RtlUnwind()中，而应该在 RtlDispatchException()中。这个问题令笔者困惑，只能暂且存疑。

对于普通节点，RtlpExecuteHandlerForUnwind()实际调用的仍是_SEHFrameHandler()，其调用路线为：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> _SEHFrameHandler() > _SEHCallHandler() > _SEHGlobalUnwind() > **RtlUnwind()**
> **RtlpExecuteHandlerForUnwind()** > **_RtlpExecuteHandler2()**> **_SEHFrameHandler()**]

但是，这次进入_SEHFrameHandler()的条件不同了，异常纪录块中与展开有关的两个标志位至少有一位为 1，所以这次走的是 if 语句中满足测试条件的那一部分，即对_SEHLocalUnwind()的调用。顾名思义，这是对一个节点、即一个局部 SEH 框架栈的展开：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> _SEHFrameHandler() > _SEHCallHandler() > _SEHGlobalUnwind() > RtlUnwind()
> RtlpExecuteHandlerForUnwind() > _RtlpExecuteHandler2()> _SEHFrameHandler()
> _SEHLocalUnwind()]

```
static void __stdcall
_SEHLocalUnwind (_SEHPortableFrame_t * frame, _SEHPortableTryLevel_t * dsttrylevel)
{
   _SEHPortableTryLevel_t * trylevel;

   for (trylevel = frame->SPF_TopTryLevel; trylevel != dsttrylevel;
                                                trylevel = trylevel->SPT_Next)
   {
      _SEHFinally_t pfnFinally;

      /* ASSERT(trylevel); */
      pfnFinally = trylevel->SPT_Handlers->SH_Finally;
      if(pfnFinally)
         pfnFinally(frame);
   }
}
```

这就很简单了，在参数 frame 所指向的节点内部，通过一个 for 循环扫描其 SEH 框架链表，如果一个 SEH 框架提供了善后函数，就执行这个函数。注意在_SEHFrameHandler()中调用这个函数时的实参 dsttrylevel 为 NULL，表示目标 SEH 框架不在这个节点中，所以 for 循环穷尽这个节点中的所有 SEH 域。而_SEHGlobalUnwind()，即全局的展开，则就是通过在 RtlUnwind()中循环地(间接)调用_SEHLocalUnwind()而完成的。

回到 RtlUnwind()的代码中，循环结束的条件只有两个，要么就是在 ExceptionList 中遇到了目标节点而通过 ZwContinue()返回，要么就是穷尽了整个 ExceptionList，使之成为空队列，但是却仍未遇到目标节点。所以，如果程序执行到了 while 循环的下面，唯一合理的解释就是本来就不存在目标节点，即 RegistrationFrame 为相当于空指针的 EXCEPTION_CHAIN_END，此时也通过 ZwContinue()返回到_SEHGlobalUnwind()。要不然就是出了问题，所以由 ZwRaiseException()启动一次类型为 STATUS_UNWIND 的软异常。

再回到_SEHCallHandler()的代码中，下面又是_SEHLocalUnwind()。但是这一次处理的是目标 SEH 域所在的节点，而实参 dsttrylevel 就指向代表着目标 SEH 域的数据结构，所以 for 循环在碰上这个数据结构是就停下来了。

不过，在我们现在的这个情景中，由于实际上并未提供善后函数，所以所谓“展开”实际上并没有做什么事。

完成了展开以后，就是通过实施函数进行长程跳转了。这个函数指针已在宏操作_SEH_TRY 中设置成指向_SEHCompilerSpecificHandler()，下面我们看它的代码：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()

> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> _SEHFrameHandler() > _SEHCallHandler() > _SEHCompilerSpecificHandler()]

```
static __declspec(noreturn)
__inline void __stdcall _SEHCompilerSpecificHandler(_SEHPortableTryLevel_t * trylevel)
{
 _SEHTryLevel_t * mytrylevel;
 mytrylevel = _SEH_CONTAINING_RECORD(trylevel, _SEHTryLevel_t, ST_Header);
 _SEHLongJmp(mytrylevel->ST_JmpBuf, 1);
}
```

函数定义的前缀__declspec(noreturn)说明这个函数的执行是不返回的。

参数 trylevel 指向一个_SEHPortableTryLevel_t 结构，这是_SEHTryLevel_t 结构中的一个成分，因此可以推算出其所在_SEHTryLevel_t 结构的地址。获得了指向其外层_SEHTryLevel_t 结构的指针以后，当初通过_SEHSetJmp()设置的地址和上下文就在这个数据结构的 ST_JmpBuf 中，现在就用来作为调用_SEHLongJmp()的第一个参数。第二个参数是 1，这就是要在跳转到目标 SEH 域的 if 语句中去时放在寄存器 Eax 里面的数值，正是这个非 0 数值使那里的 if 语句转入其 else 部分。_SEHLongJmp()的代码已在上一篇漫谈中看过，这里就不重复了。

如果不考虑嵌套异常和软异常，在一般“正常”的条件下，因长程跳转而被垮越、丢弃的函数调用框架一共有多少呢？这包括两个部分。

第一部分是异常发生之后所形成、SEHLongJmp()之上的所有框架，这从下面的函数调用路径可以看出：

[_KiTrap14() > KiPageFaultHandler() > KiKernelTrapHandler() > KiDispatchException()
> RtlDispatchException() > RtlpExecuteHandlerForException() > _RtlpExecuteHandler2()
> _SEHFrameHandler() > _SEHCallHandler() > _SEHCompilerSpecificHandler()
> SEHLongJmp()]

其中_KiTrap14()的框架实际上就是异常框架，其余都是函数调用框架。

第二部分取决于异常发生时所处的函数调用框架和 SEH 域、以及 SEH 域嵌套的情况。如上所述，长程跳转的目标是相应_SEHSetJmp()所在的函数框架，如果受保护的代码中又有函数调用，而异常又发生在被逐层调用的函数中，则所有这些函数调用框架都会被跳过。另一方面，发生异常前最后进入的 SEH 域未必就是目标 SEH 域，因为它的过滤函数可能会把问题推给上一层 SEH 域。这样，就又要上溯到上一层 SEH 域中_SEHSetJmp()所在的函数框架。余类推。

总而言之，凡是堆栈上目标 SEH 域中_SEHSetJmp()所在函数框架以下的框架全部都会被跳过。对于 ExceptionList 的处理正是为了实现这一点。

至于在我们这个情景中，则当初设置的过滤条件是_SEH_EXECUTE_HANDLER，这意味着没有过滤函数，而直接执行处理函数。这样，实际上就总是执行 ExceptionList 中第一个节点的处理函数，换言之总是长程跳转到发生异常前最后进入的 SEH 域的_SEH_HANDLE{…}部分。

回到最初_SEH_TRY{…} _SEH_HANDLE{…} _SEH_END 的代码中，看那里的 if 语句可知：

- 如果在执行受保护代码的过程中并未发生异常，则在执行完这些代码以后会执行_SEHLeave()。
- 如果发生了异常，并因而发生了长程跳转，则首先就执行_SEHLeave()。

总之，在这两种情况下都会执行_SEHLeave()，而_SEHLeave()实际上就是_SEHUnregisterFrame()，就是从 ExceptionList 队列中摘除代表着本 SEH 域的节点，这当然是最后进入 ExceptionList 的节点。而此前的节点，则都已在展开的过程中被摘除了。