# 漫谈兼容内核之二十四:
# Windows 的结构化异常处理(一)

毛德操

结构化异常处理(Structured Exception Handling)，简称 SEH，是 Windows 操作系统的一个重要组成部分。

在 ReactOS 内核的源代码中，特别是在实现系统调用的代码中，读者已经看到很多类似于这样的代码：

```
if(MaximumSize != NULL && PreviousMode != KernelMode)
{
  _SEH_TRY
  {
    ProbeForRead(MaximumSize, sizeof(LARGE_INTEGER), sizeof(ULONG));
    /* make a copy on the stack */
    SafeMaximumSize = *MaximumSize;
    MaximumSize = &SafeMaximumSize;
  }
  _SEH_HANDLE
  {
    Status = _SEH_GetExceptionCode();
  }
  _SEH_END;

  if(!NT_SUCCESS(Status))
  {
    return Status;
  }
}
```

这段代码取自 NtCreateSection()，其参数之一是指针 MaximumSize。系统调用一般都是从用户空间调用的，因此 PreviousMode 一般不是 KernelMode。所以，只要指针 MaximumSize 不是 NULL，就要从它所指的地方从用户空间把数值复制到内核空间。那为什么不直接把它的数值作为参数传递，而要这样绕一下呢？这是因为它的类型为 LARGE_INTEGER，而作为参数传递的只能是 32 位(或以下)的普通整数。

然而，从用户空间复制数据到内核空间(或反过来)恰恰是容易出事的。这是因为，用户程序的质量相对而言是没有保证的，这个指针所指向的地址(所在的页面)也许根本就没有映射，或者也许不允许读，那样就会发生与页面映射和访问有关的异常(Exception)。

不过倒也并非只要发生异常就有问题，例如要是页面已经映射、也允许读，但是所在页面已经换出(Swap-Out)，那就会发生缺页异常；而缺页异常其实不是“异常”而是“正常”，内核从磁盘上换入(Swap-In)目标页面，就可以从异常处理程序返回、并继续运行了，就像

发生了一次中断一样。此时 CPU 将重新执行发生异常的指令，这一次一般就能正常完成了。所以，(物理上的)异常之是否真的(逻辑上)“异常”，还得看具体的原因。只要没有特别加以说明，本文中所讲的异常都是指真正意义上的异常。

对于异常的处理，内核一般会提供默认的方式，例如“杀掉”当前进程，让其一死百了，这样至少不会危害别的进程。但是如果具体的程序预期在某一段代码中有可能发生某几种特定的异常，并愿意为之提供解决、补救之道，那当然是更合理、更优雅的方式。举例言之，假如用户程序中有除法运算，CPU 在碰到除数为 0 的时候就会发生异常，此时默认的处理方式一般是中止该用户程序的运行，因为不知该怎样让它继续下去了。然而这可能发生在已经连续计算了几十个小时以后，离成功也许只有一步之遥了，让它就这样退出运行未免损失太大。如果程序的设计人员事先估计到有这样的可能，也许会选择在这种情况下弹出一个对话框，提示使用者改变几个参数，然后以新的条件继续运算；或者至少问一下用户，是否把发生问题时的“现场”信息通过邮件发送给程序的设计者。显然，这是更好的解决方案。问题在于如何来实现，如何为程序的设计者提供这样做的手段。

简而言之，就是要为程序的设计者提供一种手段，使得倘若在执行某一段代码的过程中发生了特定种类的异常就执行另一些指定的代码。事实上，这正是微软的“结构化异常处理(Structured Exception Handling)”、即 SEH 机制要解决的问题之一。后面读者将会看到，SEH 要解决两类问题，这是其中之一。

在上列的代码片断中，在_SEH_TRY{}里面是要加以“保护”的代码，即用户估计可能会在执行中发生异常的代码；而_SEH_HANDLE{}里面就是当发生异常时需要执行的代码；最后的_SEH_END 则说明与 SEH 有关的代码到此为止，从此以后的代码恢复常态。这样，如果在执行_SEH_TRY{}里面受保护代码的过程中发生了某些异常，CPU 就转入_SEH_HANDLE{}；而若顺利执行完_SEH_TRY{}里面的代码，那就跳过_SEH_HANDLE{}直接到达_SEH_END。

注意在_SEH_TRY{}里面可能会调用别的函数，被调用函数的代码虽然形式上不在_SEH_TRY{}里面，但是它的本次被调用执行却同样是在_SEH_TRY{}所指定的保护范围之内。在本文中，由_SEH_TRY{}所划定的范围称为一个“SEH 保护域”，也称“SEH 框架”，因为在执行这些代码时这表现为堆栈上的一个框架。所以在本文中“SEH 域”和“SEH 框架”是同义词。说是“保护域”，其实也可以说是“捕捉域”，就是说这是一个需要“捕捉”住异常的域(所以在 C++语言中用“catch”表示捕捉到异常之后要执行的代码)。注意 SEH 域和函数是互相独立的两个概念。同一个函数，这一次是从_SEH_TRY{}里面调用，它的执行就在 SEH 域中；下一次不是从_SEH_TRY{}里面调用，就不在这个 SEH 域中了。所以一个函数(的执行)是否在 SEH 域里面是个动态的概念。不过，SEH 域总是存在于某个函数的内部，而不可能游离在函数之外，就像 C 语句只能存在于函数之内一样。

在实际应用中，SEH 域还可以嵌套，就是在一个 SEH 域的内部又通过_SEH_TRY{}开辟了第二个 SEH 域。例如，前者也许是针对页面异常的 SEH 域，而在这里面又有一部分代码可能会引起“除数为 0”的异常，所以又得将其保护起来，形成一个嵌在外层保护域里面的内层保护域。

显然，多个 SEH 框架嵌套就形成了一个 SEH 框架栈。SEH 框架栈既可以只是实质的，也可以既是实质的、又是形式的。比方说，一个 SEH 域的内部调用了一个函数，而在这个函数中又有一个 SEH 域，那么这两个 SEH 域(框架)的嵌套是实质的，但却不是形式的，因为从代码上不能一目了然看出这样的嵌套关系，这种嵌套关系是运行起来才形成的。但是，如果在第一个 SEH 域的_SEH_TRY{}内部直接又有一个_SEH_TRY{}，那么这两个 SEH 域的嵌套关系就既是实质的、又是形式的。在本文中，前者所形成的 SEH 框架栈称为“实质”SEH 框架栈、或“全局”SEH 框架栈，后者所形成的则称为“形式”SEH 框架栈、或“局

部”SEH 框架栈。之所以如此，是因为 0.3.0 版 ReactOS 的代码中对于 SEH 机制的实现有了一些变动。不过，在 0.3.0 版 ReactOS 的代码中并未见到使用形式嵌套的 SEH 域。

回头看前面_SEH_TRY{}里面的代码。这里受保护的有三个语句，先看对 ProbeForRead()的调用。ProbeForRead()是个内核函数，这里也是在内核中调用，所以对这个函数的调用本身并没有问题。

```
VOID STDCALL
ProbeForRead (IN CONST VOID *Address, IN ULONG Length, IN ULONG Alignment)
{
    ASSERT(Alignment == 1 || Alignment == 2 || Alignment == 4 || Alignment == 8);

    if (Length == 0)
        return;

    if (((ULONG_PTR)Address & (Alignment - 1)) != 0)
    {
        ExRaiseStatus (STATUS_DATATYPE_MISALIGNMENT);
    }
    else if ((ULONG_PTR)Address + Length - 1 < (ULONG_PTR)Address ||
                (ULONG_PTR)Address + Length - 1 > (ULONG_PTR)MmUserProbeAddress)
    {
        ExRaiseStatus (STATUS_ACCESS_VIOLATION);
    }
}
```

其目的只是检查参数的合理性，而并不真的去访问用户空间。如果用户空间数据所在的地址不与给定数据类型(在这里是 ULONG)的边界对齐，或者所在的位置不对、长度不合理，那就要通过 ExRaiseStatus()以软件方法模拟异常。这是为什么呢？因为在正常的情况下这是不可能发生的，既然发生了就一定是出了问题，按理说最好是 CPU 在碰到这种情况时能引起一次异常，但是 386 结构的 CPU 不会(从 486 开始就会了，这就是 17 号异常“Alignment Check”)，所以就只好通过软件手段来模拟一次“软异常”。注意这里软异常的类型为 STATUS_DATATYPE_MISALIGNMENT 和 STATUS_ACCESS_VIOLATION，前者表示与数据类型的边界不对齐，后者表示越界访问，这相当于硬异常的异常号，但是丰富得多。

就前述的 SEH 域而言，由此而引起的效果与硬件异常相同，CPU 也会转入_SEH_HANDLE{}里面。

熟悉 C++的读者可能会联想到 throw 语句，实际上也确实是同一回事。

如果 ProbeForRead()没有检查出什么问题，前面的第二个语句是“SafeMaximumSize = *MaximumSize”，这是从用户空间读取数据写入系统空间。这里写入系统空间不会有问题，但是读用户空间可能会有问题，如果指针 MaximumSize 所指的页面无映射就会发生异常。所以要把它放在_SEH_TRY{}里面。

第三个语句是“MaximumSize = &SafeMaximumSize”，这是对指针 MaximumSize 进行赋值。作为调用参数，这个变量原先在用户空间堆栈上，CPU 因系统调用进入内核以后把它复制到了系统空间堆栈上。因而这个赋值操作应该不会引起异常，本可以放在外面，但是放在_SEH_TRY{}里面也无不可。所以，并非凡是放在_SEH_TRY{}里面的都必须是可能引

起异常的语句。对于不会引起异常的语句，放在_SEH_TRY{}里面或外面都是一样。

再看安排在发生异常时加以执行的代码、即_SEH_HANDLE{}里面的代码。在这里只有一个语句，就是对_SEH_GetExceptionCode()的调用。顾名思义，这就是获取具体异常的代码，例如 STATUS_DATATYPE_MISALIGNMENT、STATUS_ACCESS_VIOLATION 等等。然后将获取的代码赋值给变量 Status，这就完事了。再往下就是_SEH_END 及其后面的 if 语句了。当然，这里面也可以有不止一个、甚至很多的语句。

注意变量 Status 原本已经初始化成 STATUS_SUCCESS，而_SEH_TRY{}里面的代码都不会改变它的值；所以只要“!NT_SUCCESS(Status)”为真就一定已经发生过异常，因此这个系统调用就出错返回了，而且所返回的就是所发生异常的代码。而根据所返回的值判断本次系统调用是否成功，以及采取什么措施，那就是用户软件的事了。

这里还要说明一下，并不是所有的异常都会落入这_SEH_HANDLE{}里面。发生异常时，首先是由内核底层的异常处理程序“认领”和处理，例如缺页异常就会被其认领并处理，处理完就返回了。即使是不归其认领处理的异常，也还得看当时是否正在通过调试工具(debugger)调试程序，如果是就交由 debugger 处理。只有不受这二者拦截的异常才会落入_SEH_HANDLE{}。后面读者将看到，每个 SEH 域都可以通过一个“过滤函数”检查本次异常的类型，已决定是否认领。如果存在嵌套的 SEH 域，则首先要由嵌套在最内层(最底层)的 SEH 域先作过滤，决定不予认领才会交给上一层 SEH 域。所以，只有不被拦截、认领，并通过了层层过滤的异常才真正进入本 SEH 域的_SEH_HANDLE{}。

上面所引的是内核中的代码，用户空间的代码同样也可以利用 SEH 所提供的功能和机制，其实 C++语言中的 try{..}catch{…}最终也是利用 SEH 实现的。

那么，以_SEH_TRY{}、_SEH_HANDLE{}、以及_SEH_END 为程序设计手段的这种 SEH 机制具体是怎么实现的呢？这正是本文要加以介绍的内容。从现在起，凡是 ReacOS 的代码均引自其 0.3.0 版。

先大致介绍一下基本的原理。

从形式上看，由_SEH_TRY{}、_SEH_HANDLE{}、和_SEH_END 在程序结构上有点像是条件语句 if(){}else{}。可是，用条件语句是实现不了 SEH 的。这是因为：条件语句所判别的条件必须表现为一个布尔量，而对此布尔量的测试和相应的程序跳转只能发生在一个固定的点上，但是_SEH_TRY{}所要保护的却是一个“域”、一个范围。诚然，我们可以在程序中放上一个初值为 0 的全局量，比方说 excepted，如果发生异常就让底层的异常响应程序将此变量设置成 1。但是，总不能让_SEH_TRY{}里面的程序每执行完一条指令就来执行基于这个变量的条件语句(例如条件跳转)吧？怎么办呢，办法是预先设置好一个目标地址，只要发生了异常，就从底层的异常响应程序直接跳转到预设的目标地址。但是，这样的跳转必须发生在返回到因异常而被中断的程序中之前，因为一旦回到了被中断的程序，那里就没有实现此种跳转所需的代码了。从堆栈的角度看，这是要从内层的“异常框架”跳转到、而不是返回到外层的 SEH 框架中。此种垮框架的跳转称为“长程跳转(Long-Jump)”。C 语言程序库中有一对函数 setjmp()和 longjmp()，就是用来实现长程跳转的，前者用于设置长程跳转的目标地址，后者用于实际的跳转。

不过，这只是单个保护域的异常处理，还不能说是“结构化异常处理”。与单个保护域相连系的是单个目标地址、即单块_SEH_HANDLE{}代码；但是实际上可能发生的异常却是多样的，要在同一块_SEH_HANDLE{}代码中考虑应对所有不同原因的异常显然不现实。在编写一段需要受保护的代码时，程序员一般只能针对这段局部的代码作出估计，就其认为可能会发生的异常安排好应对措施。所以就有了让保护域嵌套的要求，保护域的嵌套使程序员得以将注意力集中在具体的局部，而又可以从全局上防止有些异常得不到合适的处理，这与

“结构化程序设计”在精神上是一致的。

另一方面，异常既可能发生于系统空间，也可能发生于用户空间，因此两个空间都需要有实现 SEH 域的手段。但是，即使是发生于用户空间的异常，首先进入的也是内核底层的异常响应程序，从而需要有个将异常提交给用户空间进行处理的手段。这样，从内核底层的异常响应/处理程序开始，根据具体情况进入系统空间或用户空间嵌套在最内层的保护域，再根据具体情况逐层上升到外层保护域，直至穷尽所有预设的保护措施，这就形成了一套完整的异常处理机制而成为一个体系，那才可以说是“结构化异常处理”。可以想像，这么一套机制的实现并非易事。

为实现结构化异常处理，Windows 在系统空间和用户空间都有一个后进先出的异常处理队列 ExceptionList。为简化叙述，这里先从概念上作一说明，实际的实现则还要复杂一点：每当程序进入一个 SEH 框架时，就把一个带有长程跳转目标地址的数据结构挂入相应空间的异常处理队列，成为其一个节点；在内核中就挂入系统空间的队列，在用户空间就挂入用户空间的队列。而当离开当前 SEH 框架时，则从队列中摘除这数据结构。由于是后进先出队列，所摘除的一定是最近挂入队列的数据结构。显然，队列中的每一个节点都代表着一个保护域。只要队列非空，CPU 就至少是在某个(最后进入的)保护域中运行。只要队列中的节点多于一个，后进节点所代表的保护域就一定是嵌套在先进入的保护域内部，而 CPU 则同时在多个保护域内部运行。所以异常处理队列本质上是一个堆栈，反映了保护域的层次关系。一般而言，当 CPU 运行于用户空间时，系统空间的异常处理队列应该是空的。

除长程跳转目标地址外，挂入 ExceptionList 的数据结构中还可以有两个函数指针。一个是“过滤(Filter)函数”的指针，这个函数判断所发生的异常是否就是本保护域所要保护、所要应对的那种异常，如果是才加以认领而执行本 SEH 域的长程跳转。另一个是“善后(final)函数”指针，善后函数的目的通常是释放动态获取的资源。

说到“善后函数”，这里有几个概念需要澄清一下。首先，前面讲到_SEH_TRY{}里面是要加以“保护”的代码，但是所谓保护并非让其不发生异常，而是说要为可能发生的异常准备好应对之道，就好像对于高空作业要在地面上铺设一张保护网、并准备好应急预案一样。根据具体的情况，应对之道可简可繁。前面代码中的应对之道就只是获取异常代码，然后使当前的系统调用夭折而返回，并把异常代码带回用户空间。而比较复杂的应对之道，则可能会试图消除发生异常的原因，例如对于因除数为 0 而引起的异常就有这样的可能。既然是应对之道，自然就带有“善后”的意思，可是这与“善后函数”不同。或许可以说，_SEH_HANDLE{}里面的代码所提供的应对之道是应用层面上的善后、是针对程序主流的善后，而“善后函数”所提供的是辅助性的、技术性的善后。在 SEH 域嵌套的情况下，这二者有很大的不同。例如，假定在针对页面异常的 SEH 域中嵌套了一个针对除数为 0 的异常，而实际发生的是页面异常，那么长程跳转的目标是上层 SEH 域的_SEH_HANDLE{}，以执行针对页面异常的应对之道，而针对除数为 0 的应对之道则得不到执行，因为后者所在的函数框架被长程跳转跨越了。可是，与后者相联系的善后函数却仍须执行，因为后者所在的那个函数可能已经动态分配了某些资源，而函数中本来用于释放这些资源的代码却被跳过了。

这样，简而言之，当发生异常时，异常响应程序就(按后进先出的次序)依次考察相应 ExceptionList 中的各个节点并执行其过滤函数(如果有的话)，如果过滤函数认为这就是本保护域所针对的异常、或默认为相符而无需过滤，就执行本保护域的长程跳转，进入本 SEH 域的_SEH_HANDLE{}里面的代码。而对于被跨越的各个内层 SEH 域，则执行其善后函数(如果有的话)。

应该说，这是设计得很好的一种方案。明白了基本的原理以后，下面就可以看具体的代码了。

在 ReactOS 的代码中，_SEH_TRY、_SEH_HANDLE、以及_SEH_END 都是宏定义。不过，在 ReactOS 的 0.3.0 版中，这些宏操作的定义有两套。其中之一依赖于较新版本的 C 编译对__try、__except、__finally 等较新语言成分的支持，由 C 编译在编译的时候自动生成相应的细节；另一种则不依赖于 C 编译对这些新语言成分的支持。这二者之间的关系有点像是高级语言与汇编语言之间的关系。对于深入理解 SEH 而言，后者反倒有助于读者更直观、更清晰地看到此项机制的原理和具体实现。反过来，搞明白了 SEH 机制在采用“朴素”C 编译工具时的实现，也就明白了在较新版本的 C 编译中__try、__except、__finally 这些新语言成分的原理。

先看_SEH_TRY 的定义：

```
#define _SEH_TRY \
{                                                                                \
  _SEH2_INIT_CONST int _SEH2TopTryLevel = (_SEHScopeKind != 0);                  \
  _SEHPortableFrame_t * const _SEH2CurPortableFrame = _SEHPortableFrame;         \
  {                                                                              \
    static const int _SEHScopeKind = 0;                                          \
    register int _SEH2State = 0;                                                 \
    register int _SEH2Handle = 0;                                                \
    _SEHFrame_t _SEH2Frame;                                                      \
    _SEHTryLevel_t _SEH2TryLevel;                                                \
    _SEHPortableFrame_t * const _SEHPortableFrame =                              \
          _SEH2TopTryLevel ? &_SEH2Frame.SEH_Header : _SEH2CurPortableFrame;     \
    (void)_SEHScopeKind;                                                         \
    (void)_SEHPortableFrame;                                                     \
    (void)_SEH2Handle;                                                           \
                                                                                 \
    for(;;)                                                                      \
    {                                                                            \
      if(_SEH2State)                                                             \
      {                                                                          \
        for(;;)                                                                  \
        {                                                                        \
          {
```

这里的变量_SEHScopeKind 有个很特别的作用，留待后面再作介绍。先看这里所涉及的几种数据结构，包括_SEHFrame_t、_SEHPortableFrame_t、和_SEHTryLevel_t。

实际上还有一种数据结构_SEHRegistration_t，是这里不能直接看到的，我们从这个数据结构开始：

```
typedef struct __SEHRegistration
{
  struct __SEHRegistration * SER_Prev;
  _SEHFrameHandler_t SER_Handler;
}_SEHRegistration_t;
```

这就是要挂入异常处理队列 ExceptionList 的数据结构，其中的指针 SER_Prev 用来构成后进先出的异常处理队列。而 SER_Handler 则是个函数指针，其类型定义如下：

```
typedef int (__cdecl * _SEHFrameHandler_t)
                    (struct _EXCEPTION_RECORD *, void *, struct _CONTEXT *, void *);
```

这个函数称为“框架处理函数”，对于一个具体节点的处理都是由这个函数实施的。如果节点所代表的只是单个 SEH 框架，那么这个函数要处理的就只是单个 SEH 框架，例如调用其过滤函数以确定是否认领，以及认领后调用其所有内层 SEH 域的善后函数以释放资源，执行本 SEH 域的长程跳转等等。而若节点所代表的是一个局部 SEH 框架栈，则节点内部还有一个局部的队列，这个函数就要有处理一个局部 SEH 框架栈的能力。在 0.3.0 版的 ReactOS 代码中，如前所述，ExceptionList 队列中的节点代表着一个局部 SEH 框架栈。而在以前的代码中则只代表单个 SEH 框架。读者以后将会看到，SEH 机制中还使用着别的框架处理函数。可见，“框架处理函数”指针的使用带来了实现上的灵活性。

不过_SEHRegistration_t 结构只是_SEHPortableFrame_t 结构内部的一个成分：

```
typedef struct __SEHPortableFrame
{
  _SEHRegistration_t   SPF_Registration;
  unsigned long   SPF_Code;
  _SEHHandler_t   SPF_Handler;
  _SEHPortableTryLevel_t   *SPF_TopTryLevel;
}_SEHPortableFrame_t;
```

其第一个成分 SPF_Registration 就是_SEHRegistration_t 数据结构。所以，获得了指向前者的指针，也就同时获得了指向其所在_SEHPortableFrame_t 数据结构的指针。

另一个成分 SPF_Code 是异常代码，其取值范围实际上是状态代码的一个子集。而状态代码的集合相当大，其完整的定义见于 ntstatus.h。例如 STATUS_SUCCESS 定义为 0，STATUS_GUARD_PAGE_VIOLATION 定义为 0x80000001，STATUS_UNSUCCESSFUL 定义为 0xC0000001，STATUS_ACCESS_VIOLATION 定义为 0xC0000005，等等。

_SEHPortableFrame_t 又是_SEHFrame_t 数据结构内部的一个成分：

```
typedef struct __SEHFrame
{
  _SEHPortableFrame_t   SEH_Header;
  void   *SEH_Locals;
}_SEHFrame_t;
```

可见，这实际上只是在_SEHPortableFrame_t 结构的基础上附加了一个指针 SEH_Locals，用来指向一个缓冲区。顾名思义，这个缓冲区用来传递一些与局部 SEH 框架栈有关的附加数据。所以，关键性的数据结构其实还是_SEHPortableFrame_t。

回到_SEHPortableFrame_t 数据结构，其中 SPF_Handler 又是个函数指针，其类型定义为：

```
typedef void    (__stdcall * _SEHHandler_t) (struct __SEHPortableTryLevel *);
```

这个指针所指向的函数，我们不妨称之为“实施函数”，因为长程跳转就是由这个函数实施的。但是，针对具体异常所实施的应对之道并不非得是长程跳转，也有可能是别的措施。为每个节点都配备一个实施函数，目的就在于为具体的实现提供灵活性，但是实际上都使用着同一个函数。

_SEHPortableFrame_t 结构中的 SPF_TopTryLevel 则是一个结构指针，指向一个 _SEHPortableTryLevel_t 数据结构的队列：

```
typedef struct __SEHPortableTryLevel
{
   struct __SEHPortableTryLevel * SPT_Next;
   const _SEHHandlers_t * SPT_Handlers;
}_SEHPortableTryLevel_t;
```

这里的第一个成分 SPT_Next 是结构指针，用来形成_SEHPortableTryLevel_t 结构的后进先出队列，这个队列构成一个局部 SEH 框架栈，而队列中的每个_SEHPortableTryLevel_t 结构则代表着具体的 SEH 框架。

第二个成分 SPT_Handlers 是指针，指向一个_SEHHandlers_t 数据结构。这是由两个函数指针构成的数据结构：

```
typedef struct __SEHHandlers
{
  _SEHFilter_t SH_Filter;
  _SEHFinally_t SH_Finally;
}_SEHHandlers_t;
```

这里 SH_Filter 和 SH_Finally 都是函数指针。前者用来指向一个“过滤函数”，后者用来指向一个“善后函数”。

也就是说，每个_SEHHandlers_t 结构、从而每个_SEHPortableTryLevel_t 结构，给定了一对过滤函数和善后函数。

同时，_SEHPortableTryLevel_t 数据结构又是_SEHTryLevel_t 结构中的一个成分：

```
typedef struct __SEHTryLevel
{
  _SEHPortableTryLevel_t    ST_Header;
  _SEHJmpBuf_t    ST_JmpBuf;
}_SEHTryLevel_t;
```

显然，这是在_SEHPortableTryLevel_t 数据结构的基础上加上了一个_SEHJmpBuf_t 数据结构，这就是用于长程跳转的。前面讲到了长程跳转的目标地址，那只是就概念上而言；实际需要的不仅仅是一个目标地址，而是一个包括各寄存器内容在内的“现场”映像，这是在设置长程跳转目标的时候保存下来的。

这样，每个_SEHTryLevel_t 数据结构实际上给定了一个三元组，即：过滤函数、善后函数、和长程跳转目标。

但是注意虽然每个_SEHTryLevel_t 结构各有自己的过滤函数、善后函数、和长程跳转目标，但是整个节点、即_SEHFrame_t 结构、却只有一个实施函数(见函数指针 SPF_Handler)。所以，同一个局部 SEH 框架栈中各 SEH 框架实施长程跳转的方式都是一样的。如果这方面不同，就不能合在一起。

现将这些数据结构的关系和作用总结如下：

1. ExceptionList 队列中各节点的数据结构是_SEHRegistration_t。
2. _SEHRegistration_t 的外层结构_SEHFrame_t 代表着一个形式嵌套的局部 SEH 框架栈，所以每个节点代表着一个局部 SEH 框架栈。
3. 每个节点有一个函数指针 SPF_Handler，提供一个实施函数。
4. _SEHFrame_t 的主体是_SEHPortableFrame_t。
5. _SEHPortableFrame_t 结构内部的指针 SPF_TopTryLevel 指向一个局部的_SEHPortableTryLevel_t 结构队列。
6. _SEHPortableTryLevel_t 的外层结构_SEHTryLevel_t 代表着一个具体的 SEH 框架，每个框架给定了一个包括过滤函数、善后函数、和长程跳转目标的三元组。

每个局部 SEH 框架栈的队列中可以有不止一个的_SEHPortableTryLevel_t，所以就可以有不止一个这样的三元组。

所以，ExceptionList 构成一个实质的、全局的 SEH 框架栈，队列中的每一个节点都是一个形式的、局部的 SEH 框架栈，但是各节点之间没有形式上的连系。全局 SEH 框架栈可以为空，即 ExceptionList 队列为空，表示没有设置任何的 SEH 域。但是局部 SEH 框架栈不能为空，一个局部 SEH 框架栈中至少有一个 SEH 框架，否则这个节点就不应该存在了。前面从概念上叙述时说“实际的实现则还要复杂一点”，就是因为这两个队列的划分。这种划分是 0.3.0 版中才有的，以前所有的 SEH 框架都直接在 ExceptionList 队列中，整个队列就是一个 SEH 框架栈，而没有实质与形式之分，不像现在这样分成两层。

回到_SEH_TRY 的代码，这里为_SEHFrame_t 数据结构_SEH2Frame 分配了空间，里面就包含着作为其结构成分的_SEHPortableFrame_t 数据结构。同样，为_SEHTryLevel_t 数据结构_SEH2TryLevel 分配了空间，里面就包含着作为其结构成分的_SEHPortableTryLevel_t 结构。其余的代码等一下与_SEH_HANDLE 和_SEH_END 合在一起看会更加清晰。

上面是_SEH_TRY 的定义，再看_SEH_HANDLE 的定义：

```
#define _SEH_HANDLE   \
    _SEH_EXCEPT(_SEH_STATIC_FILTER(_SEH_EXECUTE_HANDLER))
```

显然，_SEH_EXCEPT 本身也是一个宏操作，它的参数是另一个宏操作_SEH_STATIC_FILTER 的运算结果：

```
/* Declares a static filter */
#define _SEH_STATIC_FILTER(ACTION_)   ((_SEHFilter_t)((ACTION_) + 2))
```

其参数 ACTION_的取值范围为：

```
#define _SEH_CONTINUE_EXECUTION (-1)
#define _SEH_CONTINUE_SEARCH (0)
#define _SEH_EXECUTE_HANDLER (1)
```

所以，从效果上看，这只是把 ACTION_的取值范围-1 到+1 调整成了 1 到 3。可是为什么要作这样的调整呢？这是因为这个数值的类型_SEHFilter_t 实际上是个函数指针：

```
typedef long    \
(__stdcall *_SEHFilter_t)( struct _EXCEPTION_POINTERS *, struct __SEHPortableFrame *);
```

作为函数指针，数值 0 会引起歧义，因为空指针的值也是 0。所以对 ACTION_的值进行这样的调整是可以理解的。当然，这也并非唯一可行的做法，例如直接就把上述三个常数定义为 1、2、3 应该也无不可。可是函数指针怎么会有 1、2、3 这样的数值呢？其实这是对函数指针的变通使用。宏操作_SEH_EXCEPT()的参数可以是个真正的函数指针，那就是由程序员提供的过滤函数，但是也可以不提供特别的过滤函数而采用由 SEH 机制提供的三种处理方式之一，所以 1、2、3 起着类似于指令代码的作用，而这里选择的是_SEH_EXECUTE_HANDLER，实际上是 1，调整以后就成了 3。那么这三种处理方式到底是什么呢？这里简单提一下：

- _SEH_CONTINUE_EXECUTION 表示应该忽略本次异常，原来在干什么就继续干什么。或者本次异常已被认领并且解决，现在可以返回被中断的程序了。
- _SEH_CONTINUE_SEARCH 表示不予认领，应该继续考察队列中的下一个节点、即上一层 SEH 域。
- _SEH_EXECUTE_HANDLER 表示认领本次异常，应实施本 SEH 域的长程跳转。

如果提供了过滤函数，那么过滤函数的返回值也应该是这三者之一。

于是，在_SEH_HANDLE 中，宏操作_SEH_EXCEPT()的参数就是 3，而_SEH_EXCEPT()本身的定义则是：

```
#define _SEH_EXCEPT(FILTER_) \
        }                                                                   \
        break;                                                              \
      }                                                                     \
      _SEH2_ASSUME(_SEH2Handle == 0);                                        \
      break;                                                                \
    }                                                                       \
    else                                                                    \
    {                                                                       \
      _SEH_DECLARE_HANDLERS((FILTER_), 0);                                  \
      _SEH2TryLevel.ST_Header.SPT_Handlers = &_SEHHandlers;                 \
      if(_SEH2TopTryLevel)                                                  \
      {                                                                     \
        if(&_SEHLocals != _SEHDummyLocals)                                  \
          _SEH2Frame.SEH_Locals = &_SEHLocals;                              \
```

```
    _SEH2Frame.SEH_Header.SPF_Handler = _SEHCompilerSpecificHandler;          \
    _SEHEnterFrame(&_SEH2Frame.SEH_Header, &_SEH2TryLevel.ST_Header);     \
   }                                                                          \
   else                                                                       \
    _SEHEnterTry(&_SEH2TryLevel.ST_Header);                                   \
                                                                              \
   if((_SEH2Handle = _SEHSetJmp(_SEH2TryLevel.ST_JmpBuf)) == 0)              \
   {                                                                          \
    _SEH2_ASSUMING(++ _SEH2State);                                            \
    _SEH2_ASSUME(_SEH2State != 0);                                           \
    continue;                                                                 \
   }                                                                         \
   else                                                                       \
   {                                                                         \
    break;                                                                    \
   }                                                                         \
  }                                                                          \
  break;                                                                     \
 }                                                                           \
 _SEHLeave();                                                               \
 if(_SEH2Handle)                                                            \
 {
```

这里有几个重要的宏操作。首先是_SEH_DECLARE_HANDLERS：

```
# define _SEH_DECLARE_HANDLERS(FILTER_, FINALLY_) \
  _SEHHandlers_t _SEHHandlers = { (0), (0) };                                 \
  _SEHHandlers.SH_Filter = (FILTER_);                                          \
  _SEHHandlers.SH_Finally = (FINALLY_);
```

显然，这里创建了一个_SEHHandlers_t 数据结构，即_SEHHandlers，为其分配空间，并设置其过滤函数和善后函数指针，前者是数值 3，后者则为 0，即没有善后函数。

有了_SEHHandlers_t 数据结构，并将其地址设置到前面的_SEHPortableTryLevel_t 数据结构中以后，就是宏操作_SEHEnterFrame()或_SEHEnterTry()，分别定义为函数_SEHEnterFrame_f()和_SEHEnterTry_f()。具体取决于这个 SEH 域是否形式上嵌套在别的 SEH 域内部。如果形式上是独立的，那就是一个局部 SEH 框架栈的“顶层”，那是要挂到 ExceptionList 队列中的。而若是形式上嵌套在另一个 SEH 域的内部，则那个作为宿主的 SEH 框架必然已经在 ExceptionList 队列中，所以只是挂到那个节点的_SEHPortableTryLevel_t 结构队列中。

现在可以暂时中断一下，解释前面代码中_SEHScopeKind 的作用了。在文件 framebased.h 中，有这么一行代码：

```
static const int _SEHScopeKind = 1;
```

注意这是 static。需要用到 SEH 机制的 C 代码文件必须包含这个.h 文件，因为_SEH_TRY 的定义也在这个文件中。这样，就相当于具体的 C 代码文件中有了这么一个静态变量。

再看_SEH_TRY 代码的开头几行：

```
{                                                                                \
  _SEH2_INIT_CONST int _SEH2TopTryLevel = (_SEHScopeKind != 0);                  \
  _SEHPortableFrame_t * const _SEH2CurPortableFrame = _SEHPortableFrame;         \
  {                                                                              \
   static const int _SEHScopeKind = 0;                                          \
   . . . . . .
```

注意每个左花括号都代表着(堆栈上)一个新的框架。所以这里第一次出现的_SEHScopeKind 是对整个 C 代码文件中的这个静态变量的引用，因为在本框架中并没有定义这么一个变量，而这个变量的值是 1。这样，变量_SEH2TopTryLevel 就被赋值为 TRUE。而第二次出现_SEHScopeKind，则是在内层框架中定义了一个静态变量，并直接初始化为 0。现在设想在_SEH_TRY{}的内部又形式嵌套了一个_SEH_TRY{}。对于嵌在内层的_SEH_TRY{}，其第一次引用的_SEHScopeKind 是初始化为 0 的那个静态变量，而不是整个文件的那个同名静态变量，因为前者定义于最靠近引用处的那层框架中。于是，内层框架的_SEH2TopTryLevel 就被赋值为 FALSE 了。显然，只要是形式上嵌套在别的_SEH_TRY{}内部，其_SEH2TopTryLevel 就总是 FALSE。而若不是形式上嵌套在别的_SEH_TRY{}内部，例如在另一个函数中、甚至在另一个文件中，则其引用的_SEHScopeKind 就是整个文件的静态变量_SEH2TopTryLevel，所以其_SEH2TopTryLevel 为 TRUE。

而_SEH2TopTryLevel 的值，则被用来作为调用_SEHEnterFrame()或_SEHEnterTry()的依据，也就是进入 ExceptionList 还是进入其中当前节点的局部 SEH 框架队列的依据。所谓“TopTryLevel”，是指一个局部 SEH 框架栈的顶层。

我们在这里主要关心的是_SEHEnterFrame()、即_SEHEnterFrame_f()：

```
void _SEH_FASTCALL _SEHEnterFrame_f (_SEHPortableFrame_t * frame,
                                     _SEHPortableTryLevel_t * trylevel)
{
 /* ASSERT(frame); */
 /* ASSERT(trylevel); */
 frame->SPF_Registration.SER_Handler = _SEHFrameHandler;
 frame->SPF_Code = 0;
 frame->SPF_TopTryLevel = trylevel;
 trylevel->SPT_Next = NULL;
 _SEHRegisterFrame(&frame->SPF_Registration);
}
```

这个函数把给定的_SEHPortableFrame_t 数据结构通过其内部成分 SPF_Registration、即_SEHRegistration_t 数据结构、挂入系统空间的 ExceptionList，并设置好节点的框架处理函数指针 SER_Handler、使其指向_SEHFrameHandler()。这个函数的代码是与 SEH 域节点的数据结构和处理方式配套的，适用于所有通过_SEH_TRY、_SEH_HANDLE、和_SEH_END 设置的 SEH 域。

注意在此之前已把_SEHPortableFrame_t 结构中的函数指针 SPF_Handler 设置成指向_SEHCompilerSpecificHandler()，这就是默认的实施函数。

挂入异常处理队列是由_SEHRegisterFrame()完成的：

**__SEHRegisterFrame**:

```
mov ecx, [esp+4]
mov eax, [fs:0]
mov [ecx+0], eax
mov [fs:0], ecx
ret
```

Windows 内核对于段寄存器 FS 有特殊的设置和使用，当 CPU 运行于系统空间时就使 fs:0 指向当前 CPU 的 KPCR 数据结构。每个 CPU 都有一个 KPCR 数据结构，所以在多处理器系统中就有不止一个的 KPCR 数据结构，当运行于系统空间时每个 CPU 的 fs:0 都指向自己的 KPCR 数据结构。而 KPCR 结构的第一个成分是 KPCR_TIB 数据结构，KPCR_TIB 的第一个成分则是 VOID 指针 ExceptionList。不言而喻，这是一个由_SEHRegistration_t 数据结构链接而成的异常处理队列。

所谓“登记”，就是把一个_SEHRegistration_t 数据结构、即一个局部 SEH 框架栈、插入这个链的头部。而函数的调用参数就是一个_SEHRegistration_t 结构指针，结构中的第一个成分是指针 SER_Prev，所以[ecx+0]就是这个指针。注意这个指针的名称 SER_Prev 容易把人搞糊涂。从代码中看，通过参数传递下来的数据结构显然是插入了 ExceptionList 的头部，因为经过这些操作之后 fs:0 指向了新的数据结构。这说明，这个链表的本质是个堆栈，有着“后进先出”的性质。指针的名称 SER_Prev 提示我们：在队列中挂在后面的节点倒是代表着“Previous”即先前的 SEH 框架、实际上是上一层 SEH 框架。事实上，只有在 SEH 域形式上不嵌套而实质嵌套的条件下，这个队列中才会有不止一个的节点。

顺便还要提一下，在用户空间也有类似的队列。每个线程在用户空间都有个 TEB，TEB 数据结构中的第一个成分是 NT_TIB 数据结构，这里面的第一个成分即是指针 ExceptionList。而且，当 CPU 运行于用户空间时，fs:0 就是指向当前线程的 TEB，实际上也就是 ExceptionList。

熟悉设备驱动和中断处理的读者可能感觉到，这跟登记一个中断处理程序颇为相似。事实上也确实如此，只不过这是在为可能发生的异常、而不是中断、做好应对的准备。

但是至此还没有设置长程跳转的目标。这是由随后的_SEHSetJmp()完成的：

**_SEHSetJmp**:

```
__SEHSetJmp@4:
 ; jump buffer
 mov eax, [esp+4]

 ; program counter
 mov ecx, [esp+0]

 ; stack pointer
 lea edx, [esp+8]
```

```
 ; fill the jump buffer
 mov [eax+0], ebp
 mov [eax+4], edx
 mov [eax+8], ecx
 mov [eax+12], ebx
 mov [eax+16], esi
 mov [eax+20], edi

 xor eax, eax
 ret 4
```

调用这个函数时的实际参数是_SEH2TryLevel.ST_JmpBuf，这是一个数组、即“跳转缓冲区”的起始地址，这里让寄存器 EAX 指向这个数组。同时，又让 ECX 持有返回地址，而让 EDX 持有调用这个函数前夕的堆栈指针，然后将这二者连同 EBP、EBX、ESI、EDI 的内容都保存在跳转缓冲区中。这样，跳转缓冲区的内容就构成了调用_SEHSetJmp()前夕的(简化的)现场，其中的返回地址就是长程跳转的目标地址，所以跳转缓冲区的内容就代表着跳转目标。

值得注意的是，这个函数返回的值、即返回时 EAX 的内容一定是 0，所以对于返回值为 0 的判定必定为真。这决定了在前面代码中的 if 语句必然会进入其测试条件为真的部分。所以，这个 0 值实际上标志着一条路径，说明这是从执行_SEHSetJmp()而来，等一下我们就可以进一步看到它的意义。

这里顺便也看一下_SEHEnterTry_f()的代码。如前所述，其目的是为一个局部 SEH 框架栈添加一个 SEH 框架。ExceptionList 链表中的节点代表着一个局部 SEH 框架栈，节点内部都有一个_SEHPortableTryLevel_t 数据结构的链表，而链表中的每一个数据结构则代表着一个具体的 SEH 框架。

```
void _SEH_FASTCALL _SEHEnterTry_f(_SEHPortableTryLevel_t * trylevel)
{
   _SEHPortableFrame_t * frame;

   frame = _SEH_CONTAINING_RECORD (_SEHCurrentRegistration(),
                                   _SEHPortableFrame_t,    SPF_Registration);
   trylevel->SPT_Next = frame->SPF_TopTryLevel;
   frame->SPF_TopTryLevel = trylevel;
}
```

参数 trylevel 是个_SEHPortableTryLevel_t 结构指针，这就是需要插入队列的数据结构。

如果受保护代码的执行顺利完成、而并未发生异常，就要从其所在的局部 SEH 框架栈中撤销当前的框架，如果这是其中的最后一个 SEH 框架则还要从 ExceptionList 中摘除相应的节点，这是由宏操作_SEHLeave()、实际上是函数_SEHLeave_f()完成的：

```
void _SEH_FASTCALL _SEHLeave_f(void)
{
```

```
  _SEHPortableFrame_t * frame;
  _SEHPortableTryLevel_t * trylevel;

  frame = _SEH_CONTAINING_RECORD (_SEHCurrentRegistration(),
                                  _SEHPortableFrame_t, SPF_Registration);
  /* ASSERT(frame); */
  trylevel = frame->SPF_TopTryLevel;
  /* ASSERT(trylevel); */
  if(trylevel->SPT_Next)
    frame->SPF_TopTryLevel = trylevel->SPT_Next;
  else
    _SEHUnregisterFrame();
}
```

实际要摘除的总是最后一个 SEH 框架，这是指针 ExceptionList 所指节点中的最后一个_SEHPortableTryLevel_t 数据结构。函数_SEHCurrentRegistration()获取 ExceptionList 指针的内容。如果所指节点内部的队列中有不止一个的_SEHPortableTryLevel_t 数据结构，就只是从这队列中摘掉最后进入的数据结构。如果只剩下一个_SEHPortableTryLevel_t 数据结构，那就要通过_SEHUnregisterFrame()从 ExceptionList 中摘除整个节点、即撤消整个局部 SEH 框架栈的登记：

```
__SEHUnregisterFrame:
 mov ecx, [fs:0]
 mov ecx, [ecx+0]
 mov [fs:0], ecx
 ret
```

这几行代码就不需要解释了。但是另一个事却很值得一提，那就是，当从 ExceptionList 中摘除一个节点时，并不需要释放这个节点所占的空间。因为这些节点其实都在堆栈上，一旦其所在的框架因从函数调用返回或长程跳转而不复存在，这些数据结构所占的空间也就自然释放了。同样，从一个节点的 SPF_TopTryLevel 队列中摘除一个节点、即_SEHPortableTryLevel_t 数据结构的时候，也不需要释放。

最后是_SEH_END：

```
#define _SEH_END \
    }                                                                    \
  }                                                                      \
 }
```

于是，经过编译工具的替换处理以后，本文开头处那个 if 语句里面跟 SEH 有关的程序、即整个_SEH_TRY{} _SEH_HANDLE{} _SEH_END 的过程就变成了这样：

```
{                                                                        \
  _SEH2_INIT_CONST int _SEH2TopTryLevel = (_SEHScopeKind != 0);          \
```

```
_SEHPortableFrame_t * const _SEH2CurPortableFrame = _SEHPortableFrame;                \
{                                                                                     \
   static const int _SEHScopeKind = 0;                                                \
   register int _SEH2State = 0;                                                       \
   register int _SEH2Handle = 0;                                                      \
   _SEHFrame_t _SEH2Frame;                                                            \
   _SEHTryLevel_t _SEH2TryLevel;                                                      \
   _SEHPortableFrame_t * const _SEHPortableFrame =                                    \
         _SEH2TopTryLevel ? &_SEH2Frame.SEH_Header : _SEH2CurPortableFrame;           \
   (void)_SEHScopeKind;                                                               \
   (void)_SEHPortableFrame;                                                           \
   (void)_SEH2Handle;                                                                 \
                                                                                      \
   for(;;)                                                                            \
   {                                                                                  \
      if(_SEH2State)                                                                  \
      {                                                                               \
         for(;;)                                                                      \
         {                                                                            \
            {
               {
                  .ProbeForRead(MaximumSize, sizeof(LARGE_INTEGER), sizeof(ULONG));
                  SafeMaximumSize = *MaximumSize;
                  MaximumSize = &SafeMaximumSize;
               }
            }                                                                         \
            break;                                                                    \
         }                                                                            \
         _SEH2_ASSUME(_SEH2Handle == 0);                                              \
         break;                                                                       \
      }                                                                               \
      else                                                                            \
      {                                                                               \
         _SEH_DECLARE_HANDLERS((FILTER_), 0);                                         \
         _SEH2TryLevel.ST_Header.SPT_Handlers = &_SEHHandlers;                        \
         if(_SEH2TopTryLevel)                                                         \
         {                                                                            \
            if(&_SEHLocals != _SEHDummyLocals)                                        \
                  _SEH2Frame.SEH_Locals = &_SEHLocals;                                \
            _SEH2Frame.SEH_Header.SPF_Handler = _SEHCompilerSpecificHandler;          \
            _SEHEnterFrame(&_SEH2Frame.SEH_Header, &_SEH2TryLevel.ST_Header);\
         }                                                                            \
         else                                                                         \
            _SEHEnterTry(&_SEH2TryLevel.ST_Header);                                   \
```

```
                                                                    \
        if((_SEH2Handle = _SEHSetJmp(_SEH2TryLevel.ST_JmpBuf)) == 0)   \
        {                                                           \
          _SEH2_ASSUMING(++_SEH2State);                              \
          _SEH2_ASSUME(_SEH2State != 0);                             \
          continue;                                                  \
        }                                                           \
        else                                                        \
        {                                                           \
          break;                                                    \
        }                                                           \
      }                                                             \
      break;                                                        \
    }                                                               \
    _SEHLeave();                                                    \
    if(_SEH2Handle)                                                 \
    {
      Status = _SEH_GetExceptionCode();
    }                                                               \
  }                                                                 \
}
```

开始时_SEH2State 为 0，所以在进入外层 for 语句后的第一轮循环中执行的是 if 语句的 else 部分。在这里“登记”了本保护域的数据结构，并执行_SEHSetJmp()。

由于_SEHSetJmp()的返回值为 0，因而其所在 if 语句的判定条件得到满足，于是 _SEH2State 的值递增成 1，并执行 continue 语句而开始外层 for 语句的第二轮循环。注意 _SEHSetJmp()的返回值赋给了_SEH2Handle，所以_SEH2Handle 的值也是 0。

这一次 if(_SEH2State)的条件得到满足，所以就进入了内层的 for 循环，这里要执行的就是受保护的代码。虽说在形式上这是个无限循环，实际上里面的代码却只执行一次，因为后面马上就有个 break 语句。所以，这实际上与 do{}while(0)的效果是一样的。

我们先假定这里受保护的代码顺利得到执行、而并未发生异常。执行完这些代码以后，就因为 break 语句而跳出了内层的 for 循环。然后紧接着又是一个 break 语句，这次跳出的是外层的 for 循环。于是，就到了_SEHLeave()。

执行完_SEHLeave()以后，由于_SEH2Handle 的值是 0，最后这 if 语句里面的代码就不会得到执行。这样，从效果上看，就是保护域中的代码得到了正常执行。

那么，要是在执行保护域中的代码时发生了异常又会怎样呢？如果发生异常，内核中底层的异常响应程序会依次检查 ExceptionList 中的数据结构，具体的过程将在下一篇漫谈中介绍，如果某个节点中的数据结构表明这次异常正是它要保护的，就会通过_SEHLongJmp 执行一次长程跳转：

```
_SEHLongJmp:
__SEHLongJmp@8:
  ; return value
  mov eax, [esp+8]
```

```
; jump buffer
mov ecx, [esp+4]

; restore the saved context
mov ebp, [ecx+0]
mov esp, [ecx+4]
mov edx, [ecx+8]
mov ebx, [ecx+12]
mov esi, [ecx+16]
mov edi, [ecx+20]
jmp edx
```

这个函数有两个参数，第一个就是跳转缓冲区指针，第二个是数值 1。注意第二个参数被置入了 EAX，此后 EAX 的内容一直没有改变，一直到通过 jmp 指令实现长程跳转、即跨堆栈框架的跳转。所以，在执行长程跳转的时候 EAX 的内容为 1。由于相应的_SEHSetJmp()是在 if 语句中执行的，长程跳转的目标地址就是_SEHSetJmp()当时的返回地址，所以 jmp 指令以后就是 if 语句中检测该函数返回值的指令。这样，对于 if 语句而言，长程跳转的效果就好像是刚从_SEHSetJmp()返回一样，所不同的是此时的“返回值”是 1 而不是 0，从而将这两条路线区分开来。

于是，当发生异常而跳转到这里的时候就自然会进入它的 else 部分，并且_SEH2Handle 的值为 1，而 else 部分的 break 语句则使 CPU 跳出外层的 for 循环。然后先通过_SEHLeave()撤销当前 SEH 框架的登记，接着因为_SEH2Handle 为 1 而进入了这里的 if 语句里面，这就是_SEH_HANDLE{}里面的代码。在前面所引的例子中，这就是对_SEH_GetExceptionCode()的调用。

这样，一旦登记了一个 SEH 框架的异常处理，就好像为随后的危险动作布下了一个保护网，万一发生异常、摔下来也有个应对之道。而一旦 CPU 平安到达_SEHLeave()，撤销了登记，这保护网就撤掉了。

读者在下一篇漫谈中还会看到，在实施长程跳转的前夕，SEH 机制会摘除 ExceptionList 队列中(以及所在节点的局部队列中)目标节点之前的所有节点、并依次执行它们的善后函数。这个过程称为“展开(Unwinding)”。这些节点代表着堆栈上嵌套在目标 SEH 框架内部的各层 SEH 框架；而且这些节点(作为数据结构)本来就存在于堆栈上的这些框架中。一旦实施长程跳转，这些框架就不复存在、这些节点就失去意义了。“皮之不存毛将焉附”，当然应该把这些节点从链表中摘除。不过，只有在存在过滤函数的条件下，才可能会有“目标节点之前的”节点，否则目标节点必定是链表中的第一个节点，代表着最后进入的 SEH 框架。

前面所引的例子中没有提供过滤函数，这是因为宏操作_SEH_HANDLE 的定义为：

```
#define _SEH_HANDLE   \
    _SEH_EXCEPT(_SEH_STATIC_FILTER(_SEH_EXECUTE_HANDLER))
```

这里的“_SEH_STATIC_FILTER(_SEH_EXECUTE_HANDLER)”实际上可以是一个函数名，这就是过滤函数。所以，如果不用_SEH_HANDLE、而直接引用_SEH_EXCEPT()，那就可以提供过滤函数了。例如下面是取自 NtAddAtom()的一段代码：

```
NTSTATUS NTAPI
NtAddAtom(IN PWSTR AtomName, IN ULONG AtomNameLength, OUT PRTL_ATOM Atom)
{
    . . . . . .

    . . . . . .
    if (PreviousMode != KernelMode)
    {
        /* Enter SEH */
        _SEH_TRY
        {
            if (AtomName)
            {
                . . . . . .
                /* Probe the atom too */
                if (Atom) ProbeForWriteUshort(Atom);
            }
        }
        _SEH_EXCEPT(_SEH_ExSystemExceptionFilter)
        {
            Status = _SEH_GetExceptionCode();
        }
        _SEH_END;
    }
    else
    {
        . . . . . .
    }
    . . . . . .
    return Status;
}
```

这里把_SEH_TRY{} _SEH_HANDLE{} _SEH_END 这个“模板”改成_SEH_TRY{} _SEH_EXCEPT(){} _SEH_END，这就引入了过滤函数 SEH_ExSystemExceptionFilter()。

如前所述，过滤函数的返回值必须是下列的三者之一：

```
#define _SEH_CONTINUE_EXECUTION (-1)
#define _SEH_CONTINUE_SEARCH (0)
#define _SEH_EXECUTE_HANDLER (1)
```

所以，更确切地说，_SEH_EXCEPT()的参数是个整数，如果是函数名就是这个函数返回的值。前面宏操作_SEH_HANDLE 的定义中直接把_SEH_EXCEPT()的参数定义成(带偏置的)_SEH_EXECUTE_HANDLER，就表示不需要过滤函数，来者不拒，不管发生什么样的

异常都执行本保护域的长程跳转。

另一方面，前面宏操作_SEH_EXCEPT 的定义中引用了另一个宏操作_SEH_DECLARE_HANDLERS()，当时是这样引用的：

```
. . . . . .
_SEH_DECLARE_HANDLERS((FILTER_), 0);
. . . . . .
```

这个宏操作的定义实际上是这样：

```
# define _SEH_DECLARE_HANDLERS(FILTER_, FINALLY_) \
  _SEHHandlers_t _SEHHandlers = { (0), (0) };                                    \
  _SEHHandlers.SH_Filter = (FILTER_);                                              \
  _SEHHandlers.SH_Finally = (FINALLY_);
```

可见，之所以在前述两个“模板”中都没有使用善后函数，是因为固定把参数 FINALLY_设置成了 0。如果另外设计一个宏操作，传下善后函数的函数名，就可以在程序中给定善后函数了。不过，就内核而言善后函数并不是太重要，因为需要保护的大多是刚进入具体系统调用时的代码，此时一般还没有获取动态分配的资源。