第24章　异常处理程序和软件异常

异常是我们不希望有的事件。在编写程序的时候，程序员不会想去存取一个无效的内存地址或用0来除一个数值。不过，这样的错误还是常常会发生的。 CPU负责捕捉无效内存访问和用0除一个数值这种错误，并相应引发一个异常作为对这些错误的反应。 CPU引发的异常，就是所谓的硬件异常（hardware exception）。在本章的后面，我们还会看到操作系统和应用程序也可以引发相应的异常，称为软件异常（software exception）。

当出现一个硬件或软件异常时，操作系统向应用程序提供机会来考察是什么类型的异常被引发，并能够让应用程序自己来处理异常。下面就是异常处理程序的文法：

```
__try {
   // Guarded body
   .
   .
   .
}
__except (exception filter) {
   // Exception handler
   .
   .
   .
}
```

注意--except关键字。每当你建立一个try块，它必须跟随一个finally块或一个except块。一个try 块之后不能既有finally块又有except块。但可以在try-except块中嵌套try-finally块，反过来也可以。

24.1　通过例子理解异常过滤器和异常处理程序

与结束处理程序（前一章讨论过）不同，异常过滤器（ exception filter）和异常处理程序是通过操作系统直接执行的，编译程序在计算异常过滤器表达式和执行异常处理程序方面不做什么事。下面几节的内容举例说明try-except块的正常执行，解释操作系统如何以及为什么计算异常过滤器，并给出操作系统执行异常处理程序中代码的环境。

24.1.1　Funcmeister1

这里是一个try-exception块的更具体的例子。

```
DWORD Funcmeister1() {
   DWORD dwTemp;

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Perform some operation.
      dwTemp = 0;
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Handle an exception; this never executes.
```

```
      :
      }

      // 3. Continue processing.
      return(dwTemp);
   }
```

在Funcmeister1的try块中，只是把一个0赋给dwTemp变量。这个操作决不会造成异常的引发，所以except块中的代码永远不会执行。注意这与 try-finally行为的不同。在dwTemp被设置成0之后，下一个要执行的指令是return语句。

尽管在结束处理程序的try块中使用return、goto、continue和break语句遭到强烈地反对，但在异常处理程序的try块中使用这些语句不会产生速度和代码规模方面的不良影响。这样的语句出现在与except块相结合的try块中不会引起局部展开的系统开销。

24.1.2 Funcmeister2

让我们修改这个函数，看会发生什么事情：

```
DWORD Funcmeister2() {
   DWORD dwTemp = 0;

   // 1. Do any processing here.
   :
   __try {
      // 2. Perform some operation(s).
      dwTemp = 5 / dwTemp;      // Generates an exception
      dwTemp += 10;             // Never executes
   }
   __except ( /* 3. Evaluate filter. */ EXCEPTION_EXECUTE_HANDLER) {
      // 4. Handle an exception.

      MessageBeep(0);
      :
    }

   // 5. Continue processing.
   return(dwTemp);
}
```

Funcmeister2中，try块中有一个指令试图以0来除5。CPU将捕捉这个事件，并引发一个硬件异常。当引发了这个异常时，系统将定位到except块的开头，并计算异常过滤器表达式的值，过滤器表达式的结果值只能是下面三个标识符之一，这些标识符定义在 Windows的Excpt.h文件中（见表24-1）。

表24-1 标识符及其定义

标 识 符	定 义 为
EXCEPTION_EXECUTE_HANDLER	1
EXCEPTION_CONTINUE_SEARCH	0
EXCEPTION_CONTINUE_EXECUTION	-1

下面几节将讨论这些标识符如何改变线程的执行。在阅读这些内容时可参阅图 24-1，该图概括了系统如何处理一个异常的情况。

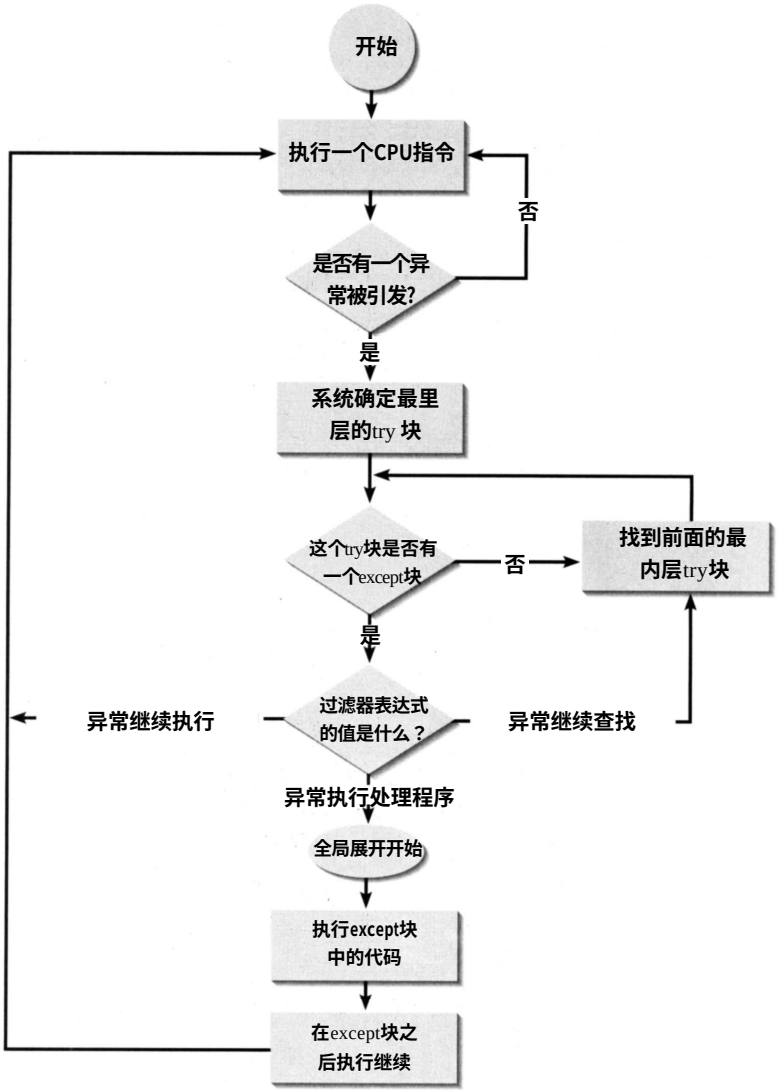

图24-1 系统如何处理一个异常

24.2 EXCEPTION_EXECUTE_HANDLER

在Funcmeister2中，异常过滤器表达式的值是 EXCEPTION_EXECUTE_HANDLER。这个值的意思是要告诉系统：“我认出了这个异常。即，我感觉这个异常可能在某个时候发生，我已编写了代码来处理这个问题，现在我想执行这个代码。”在这个时候，系统执行一个全局展开（本章后面将讨论），然后执行向except块中代码（异常处理程序代码）的跳转。在 except块中代码执行完之后，系统考虑这个要被处理的异常并允许应用程序继续执行。这种机制使Windows应用程序可以抓住错误并处理错误，再使程序继续运行，不需要用户知道错误的发生。

但是，当except块执行后，代码将从何处恢复执行？稍加思索，我们就可以想到几种可能性。

第一种可能性是从产生异常的CPU指令之后恢复执行。在Funcmeister2中执行将从对dwTemp加10的指令开始恢复。这看起来像是合理的做法，但实际上，很多程序的编写方式使得当前面的指令出错时，后续的指令不能够继续成功地执行。

在Funcmeister2中，代码可以继续正常执行，但是，Funcmeister2已不是正常的情况。代码应该尽可能地结构化，这样，在产生异常的指令之后的CPU指令有望获得有效的返回值。例如，可能有一个指令分配内存，后面一系列指令要执行对该内存的操作。如果内存不能够被分配，则所有后续的指令都将失败，上面这个程序重复地产生异常。

这里是另外一个例子，说明为什么在一个失败的CPU指令之后，执行不能够继续。我们用下面的程序行来替代Funcmeister2中产生异常的C语句：

```
malloc(5 / dwTemp);
```

对上面的程序行，编译程序产生CPU指令来执行除法，将结果压入栈中，并调用malloc函数。如果除法失败，代码就不能继续执行。系统必须向栈中压东西，否则，栈就被破坏了。

所幸的是，微软没有让系统从产生异常的指令之后恢复指令的执行。这种决策使我们免于面对上面的问题。

第二种可能性是从产生异常的指令恢复执行。这是很有意思的可能性。如果在except块中有这样的语句会怎么样呢：

```
dwTemp = 2;
```

在except块中有了这个赋值语句，可以从产生异常的指令恢复执行。这一次，将用2来除5，执行将继续，不会产生其他的异常。可以做些修改，让系统重新执行产生异常的指令。你会发现这种方法将导致某些微妙的行为。我们将在“EXCEPTION_ CONTINUE_EXECUTION”一节中讨论这种技术。

第三种可能性是从except块之后的第一条指令开始恢复执行。这实际是当异常过滤器表达式的值为EXCEPTION_EXECUTE_HANDLER时所发生的事。在except块中的代码结束执行后，控制从except块之后的第一条指令恢复。

24.2.1 一些有用的例子

假如要实现一个完全强壮的应用程序，该程序需要每周7天，每天24小时运行。在今天的世界里，软件变得这么复杂，有那么多的变量和因子来影响程序的性能，笔者认为如果不用SEH，要实现完全强壮的应用程序简直是不可能的。我们先来看一个样板程序，即C的运行时函数strcpy：

```
char* strcpy(
   char* strDestination,
   const char* strSource);
```

这是一个相当简单的函数，它怎么会引起一个进程结束呢？如果调用者对这些参数中的某一个传递NULL（或任何无效的地址），strcpy就引起一个存取异常，并且导致整个进程结束。

使用SEH，就可以建立一个完全强壮的strcpy函数：

```
char* RobustStrCpy(char* strDestination, const char* strSource) {

   __try {
      strcpy(strDestination, strSource);
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Nothing to do here
   }
```

```
   return(strDestination);
}
```

这个函数所做的一切就是将对strcpy的调用置于一个结构化的异常处理框架中。如果strcpy执行成功，函数就返回。如果strcpy引起一个存取异常，异常过滤器返回EXCEPTION_EXECUTE_HANDLER，导致该线程执行异常处理程序代码。在这个函数中，处理程序代码什么也不做，RobustStrCpy只是返回到它的调用者，根本不会造成进程结束。

我们再看另外一个例子。这个函数返回一个字符串里的以空格分界的符号个数：

```
int RobustHowManyToken(const char* str) {

   int nHowManyTokens = -1;  // -1 indicates failure
   char* strTemp = NULL;     // Assume failure

   __try {

      // Allocate a temporary buffer
      strTemp = (char*) malloc(strlen(str) + 1);

      // Copy the original string to the temporary buffer
      strcpy(strTemp, str);

      // Get the first token
      char* pszToken = strtok(strTemp, " ");

      // Iterate through all the tokens
      for (; pszToken != NULL; pszToken = strtok(NULL, " "))
         nHowManyTokens++;

      nHowManyTokens++;       // Add 1 since we started at -1
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      // Nothing to do here
   }
   // Free the temporary buffer (guaranteed)
   free(strTemp);

   return(nHowManyTokens);
}
```

这个函数分配一个临时缓冲区并将一个字符串复制到里面。然后该函数用C运行时函数strtok来获取字符串的符号。临时缓冲区是必要的，因strtok要修改它所操作的串。感谢有了SEH，这个非常简单的函数就处理了所有的可能性。我们来看在几个不同的情况下函数是如何执行的。

首先，如果调用者向函数传递了NULL（或任何无效的内存地址），nHowManyTokens被初始化成-1。在try块中对strlen的调用会引起存取异常。异常过滤器获得控制并将控制转移给except块，except块什么也不做。在except块之后，调用free来释放临时内存块。但是，这个内存从未分配，所以结束调用free，向它传递NULL作为参数。ANSI C明确说明用NULL作为参数调用free是合法的。这时free什么也不做，这并不是错误。最后，函数返回-1，指出失败。注意进程并没有结束。

其次，调用者可能向函数传递了一个有效的地址，但对malloc的调用（在try块中）可能失败并返回NULL。这将导致对strcpy的调用引起一个存取异常。同样，异常过滤器被调用，except块执行（什么也不做），free被调用，传递给它NULL（什么也不做），返回-1，告诉调用程序该函数失败。注意进程也没有结束。

最后，假定调用者向函数传递了一个有效的地址，并且对malloc的调用也成功了。这种情况下，其余的代码也会成功地在nHowManyTokens变量中计算符号的数量。在try块的结尾，异

常过滤器不会被求值，except块中代码不会被执行，临时内存缓冲区将被释放，并向调用者返回nHowManyTokens。

使用SEH会感觉很好。RobustHowManyToken函数说明了如何在不使用try-finally的情况下保证释放资源。在异常处理程序之后的代码也都能保证被执行（假定函数没有从try块中返回——应避免的事情）。

我们再看一个特别有用的SEH例子。这里的函数重复一个内存块：

```
PBYTE RobustMemDup(PBYTE pbSrc, size_t cb) {

   PBYTE pbDup = NULL;       // Assume failure

   __try {

      // Allocate a buffer for the duplicate memory block
      pbDup = (PBYTE) malloc(cb);

      memcpy(pbDup, pbSrc, cb);
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      free(pbDup);
      pbDup = NULL;
   }

   return(pbDup);
}
```

这个函数分配一个内存缓冲区，并从源块向目的块复制字节。然后函数将复制的内存缓冲区的地址返回给调用程序（如果函数失败则返回NULL）。希望调用程序在不需要缓冲区时释放它。这是在except块中实际有代码的第一个例子。我们看一看这个函数在不同条件下是如何执行的。

- 如果调用程序对pbSrc参数传递了一个无效地址，或者如果对malloc的调用失败（返回NULL），memcpy将引起一个存取异常。该存取异常执行过滤器，将控制转移到except块。在except块内，内存缓冲区被释放，pbDup被设置成NULL以便调用程序能够知道函数失败。这里，注意ANSI C允许对free传递NULL。
- 如果调用程序给函数传递一个有效地址，并且如果对malloc的调用成功，则新分配内存块的地址返回给调用程序。

24.2.2 全局展开

当一个异常过滤器的值为EXCEPTION_EXECUTE_HANDLER时，系统必须执行一个全局展开(global unwind)。这个全局展开使所有那些在处理异常的y_except块之后开始执行但未完成的try-finally块恢复执行。图24-2是描述系统如何执行全局展开的流程图，在解释后面的例子时，请参阅这个图。

```
void FuncOStimpy1() {

   // 1. Do any processing here.
   .
   .
   .
   __try {
      // 2. Call another function.
      FuncORen1();

      // Code here never executes.
   }

   __except ( /* 6. Evaluate filter. */ EXCEPTION_EXECUTE_HANDLER) {
      // 8. After the unwind, the exception handler executes.
```

```
      MessageBox(…);
   }

   // 9. Exception handled--continue execution.
   :
   :
}
void FuncORen1() {
   DWORD dwTemp = 0;

   // 3. Do any processing here.
   :
   :
   __try {
      // 4. Request permission to access protected data.
      WaitForSingleObject(g_hSem, INFINITE);

      // 5. Modify the data.
      //    An exception is generated here.
      g_dwProtectedData = 5 / dwTemp;
   }
   __finally {
      // 7. Global unwind occurs because filter evaluated
      //    to EXCEPTION_EXECUTE_HANDLER.

      // Allow others to use protected data.
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // Continue processing--never executes.
   :
   :
}
```

函数FuncOStimpy1和FuncOren1结合起来可以解释SEH最令人疑惑的方面。程序中注释的标号给出了执行的次序，我们现在开始做一些分析。

FuncOStimpy1开始执行，进入它的try块并调用FuncORen1。FuncORen1开始执行，进入它的try块并等待获得信标。当它得到信标，FuncORen1试图改变全局数据变量g_dwProtectedData。但由于除以0而产生一个异常。系统因此取得控制，开始搜索一个与 except块相配的try块。因为FuncORen1中的try与同一个finally块相配，所以系统再上溯寻找另外的try块。这里，系统在FuncOStimpy1中找到一个try块，并且发现这个try块与一个except块相配。

系统现在计算与FuncOStimpy1中except块相联的异常过滤器的值，并等待返回值。当系统看到返回值是EXCEPTION_EXECUTE_HANDLER的，系统就在FuncORen1的finally块中开始一个全局展开。注意这个展开是在系统执行 FuncOStimpy1的except块中的代码之前发生的。对于一个全局展开，系统回到所有未完成的 try块的结尾，查找与 finally块相配的try块。在这里，系统发现的finally块是FuncORen1中所包含的finally块。

当系统执行 FuncORen1的finally块中的代码时，就可以清楚地看到 SEH的作用了。FuncORen1释放信标，使另一个线程恢复执行。如果这个 finally块中不包含ReleaseSemaphore的调用，则信标不会被释放。

在finally块中包含的代码执行完之后，系统继续上溯，查找需要执行的未完成 finally块。在这个例子中已经没有这样的 finally块了。系统到达要处理异常的try-except块就停止上溯。这时，全局展开结束，系统可以执行except块中所包含的代码。

结构化异常处理就是这样工作的。 SEH比较难于理解，是因为在代码的执行当中与系统牵

扯太多。程序代码不再是从头到尾执行，系统使代码段按照它的规定次序执行。这种执行次序虽然复杂，但可以预料。按图24-1和图24-2的流程图去做，就可以有把握地使用SEH。

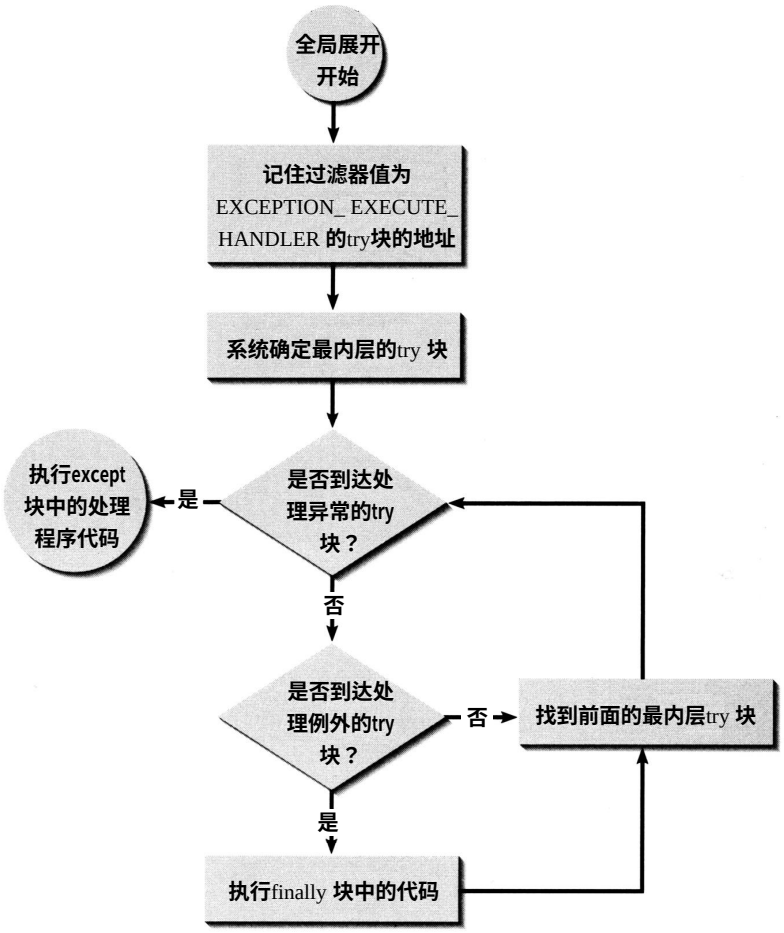

图24-2 系统如何执行一个全局展开

为了更好地理解这个执行次序，我们再从不同的角度来看发生的事情。当一个过滤器返回EXCEPTION_EXECUTE_HANDLER时，过滤器是在告诉系统，线程的指令指针应该指向except块中的代码。但这个指令指针在FuncORen1的try块里。回忆一下第23章，每当一个线程要从一个try-finally块离开时，必须保证执行finally块中的代码。在发生异常时，全局展开就是保证这条规则的机制。

24.2.3 暂停全局展开

通过在finally块里放入一个return语句，可以阻止系统去完成一个全局展开。请看下面的代码：

```
void FuncMonkey() {
   __try {
      FuncFish();
   }
   __except (EXCEPTION_EXECUTE_HANDLER) {
      MessageBeep(0);
   }
   MessageBox(…);
```

```
}

void FuncFish() {
   FuncPheasant();
   MessageBox(…);
}

void FuncPheasant() {

   __try {
      strcpy(NULL, NULL);
   }

   __finally {
      return;
   }
}
```

在FuncPheasant的try块中，当调用strcpy函数时，会引发一个内存存取异常。当异常发生时，系统开始查看是否有一个过滤器可以处理这个异常。系统会发现在 FuncMonkey中的异常过滤器是处理这个异常的，并且系统开始一个全局展开。

全局展开启动，先执行FuncPheasant的finally块中的代码。这个代码块包含一个return语句。这个return语句使系统停止做展开，FuncPheasant将实际返回到FuncFish。然后FuncFish又返回到函数FuncMonkey。FuncMonkey中的代码继续执行，调用MessageBox。

注意FuncMonkey的异常块中的代码从不会执行对 MessageBeep的调用。FuncPheasant的finally块中的return语句使系统完全停止了展开，继续执行，就像什么也没有发生。

微软专门设计SEH按这种方式工作。程序员有可能希望使展开停止，让代码继续执行下去。这种方法为程序员提供了一种手段。原则上，应该小心避免在 finally块中安排return语句。

24.3 EXCEPTION_CONTINUE_EXECUTION

我们再仔细考察一下异常过滤器，看它是如何计算出定义在 Excpt.h中的三个异常标识符之一的。在“ Funcmeister2 ”一节中，为简单起见，在过滤器里直接硬编码了标识符EXCEPTION_EXECUTE_HANDLER，但实际上可以让过滤器调用一个函数确定应该返回哪一个标识符。这里是另外一个例子。

```
char g_szBuffer[100];

void FunclinRoosevelt1() {
   int x = 0;
   char *pchBuffer = NULL;

   __try {
      *pchBuffer = 'J';
      x = 5 / x;
   }
   __except (OilFilter1(&pchBuffer)) {
      MessageBox(NULL, "An exception occurred", NULL, MB_OK);
   }
   MessageBox(NULL, "Function completed", NULL, MB_OK);
}
```

```
LONG OilFilter1(char **ppchBuffer) {
   if (*ppchBuffer == NULL) {
      *ppchBuffer = g_szBuffer;
      return(EXCEPTION_CONTINUE_EXECUTION);
   }
   return(EXCEPTION_EXECUTE_HANDLER);
}
```

这里，首先遇到的问题是在我们试图向pchBuffer所指向的缓冲区中放入一个字母‘J’时发生的。因为这里没有初始化pchBuffer，使它指向全局缓冲区g_szBuffur。pchBuffer实际指向NULL。CPU将产生一个异常，并计算与异常发生的try块相关联的except块的异常过滤器。在except块中，对OilFilter函数传递了pchBuffer变量的地址。

当OilFilter获得控制时，它要查看*ppchBuffer是不是NULL，如果是，把它设置成指向全局缓冲区g_szBuffer。然后这个过滤器返回EXCEPTION_CONTINUE_EXECUTION。当系统看到过滤器的值是EXCEPTION_CONTINUE_EXECUTION时，系统跳回到产生异常的指令，试图再执行一次。这一次，指令将执行成功，字母‘J’将放在g_szBuffer的第一个字节。

随着代码继续执行，我们又在try块中碰到除以0的问题。系统又要计算过滤器的值。这一次，OilFilter看到*ppchBuffer不是NULL，就返回EXCEPTION_EXECUTE_HANDLER，这是告诉系统去执行except块中的代码。这会显示一个消息框，用文本串报告发生了异常。

如你所见，在异常过滤器中可以做很多的事情。当然过滤器必须返回三个异常标识符之一，但可以执行任何其他你想执行的任务。

使用带警告的EXCEPTION_CONTINUE_EXECUTION

我们已经知道，修改FunclinRoosevelt1函数中的问题可使系统继续执行，也可能不执行，这要取决于程序的目标CPU，取决于编译程序为C／C++语句生成的指令，取决于编译程序的选项设置。

一个编译程序对下面的C/C++语句可能生成两条机器指令：

```
*pchBuffer = 'J';
```

机器指令可能是这个样子：

```
MOV EAX, [pchBuffer]   // Move the address into a register
MOV [EAX], 'J'         // Move 'J' into the address
```

第二条指令产生异常。异常过滤器可以捕获这个异常，修改pchBuffer的值，并告诉系统重新执行第二条CPU指令。但问题是，寄存器的值可能不改变，不能反映装入到pchBuffer的新值，这样重新执行CPU指令又产生另一个异常。这就发生了死循环。

如果编译程序优化了代码，继续执行可能顺利；如果编译程序没有优化代码，继续执行就可能失败。这可能是个非常难修复的bug，需要检查源代码生成的汇编语言程序，确定程序出了什么错。这个例子的寓意就是在异常过滤器返回EXCEPTION_CONTINUE_EXECUTION时，要特别地小心。

有一种情况可保证EXCEPTION_CONTINUE_EXECUTION每次总能成功：当离散地向一个保留区域提交存储区时。在第15章讨论过如何保存一个大的地址空间，并向这个地址空间离散地提交存储区。VMAlloc示例程序说明了这个例子。编写VMAlloc程序的一种更好的办法是必要时使用SEH提交存储区，而不是每次调用Virtual Alloc函数。

在第16章，我们讨论了线程栈。特别是，我们讲解了系统如何为线程的栈保留一个1MB的地址空间范围，以及在线程需要内存区时，系统如何自动向栈提交新的内存区。为此，系统在

内部建立了一个SEH框架。当一个线程试图去存取并不存在的栈存储区时，就产生一个异常。系统的异常过滤器可以确定这个异常是源于试图存取栈的保留地址空间。异常过滤器调用VirtualAlloc向线程的栈提交更多的存储区，然后过滤器返回 EXCEPTION_CONTINUE_EXECUTION。这时，试图存取栈存储区的CPU指令可以成功执行，线程可以继续运行。

将虚拟内存技术同结构化异常处理结合起来，可以编写一些执行非常快，非常高效的程序。下一章的SpreadSheet示例程序将说明如何使用SEH有效地实现内存管理。这个代码执行得非常快。

24.4 EXCEPTION_CONTINUE_SEARCH

迄今为止我们看到的例子都很平常。通过增加一个函数调用，让我们来看看其他方面的问题：

```
char g_szBuffer[100];

void FunclinRoosevelt2() {
   char *pchBuffer = NULL;

   __try {
      FuncAtude2(pchBuffer);
   }
   __except (OilFilter2(&pchBuffer)) {
      MessageBox(…);
   }
}

void FuncAtude2(char *sz) {
   *sz = 0;
}

LONG OilFilter2 (char **ppchBuffer) {
   if (*ppchBuffer == NULL) {
      *ppchBuffer = g_szBuffer;
      return(EXCEPTION_CONTINUE_EXECUTION);
   }
   return(EXCEPTION_EXECUTE_HANDLER);
}
```

当FunclinRoosevelt2执行时，它调用FuncAtude2并传递参数NULL。当FuncAtude2执行时，引发了一个异常。同前面的例子一样，系统计算与最近执行的 try块相关联的异常过滤器的值。在这个例子中，FunclinRoosevelt2中的try块是最近执行的try块，所以系统调用OilFilter2函数来求异常过滤器的值——尽管这个异常是在FunclinAtude2函数中产生的。

现在我们让问题变得更复杂一点，在程序中再增加一个try_except块。

```
char g_szBuffer[100];

void FunclinRoosevelt3() {

   char *pchBuffer = NULL;

   __try {
      FuncAtude3(pchBuffer);
   }
```

```
    __except (OilFilter3(&pchBuffer)) {
       MessageBox(…);
    }
}
void FuncAtude3(char *sz) {
    __try {
       *sz = 0;
    }
    __except (EXCEPTION_CONTINUE_SEARCH) {
       // This never executes.
       .
       .
       .
    }
}

LONG OilFilter3(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
       *ppchBuffer = g_szBuffer;
       return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

现在，当FuncAtude 3试图向地址NULL里存放0时，会引发一个异常。但这时将执行FuncAtude3的异常过滤器。FuncAtude3的异常过滤器很简单，只是取值EXCEPTION_CONTINUE_SEARCH。这个标识符是告诉系统去查找前面与一个except块相匹配的try块，并调用这个try块的异常处理器。

因为FuncAtude3的过滤器的值为EXCEPTION_CONTINUE_SEARCH，系统将查找前面的try块（在FunclinRoosevelt3里），并计算其异常过滤器的值，这里异常过滤器是OilFilter3。OilFilter3看到pchBuffer是NULL，将pchBuffer设定为指向全局缓冲区，然后告诉系统恢复执行产生异常的指令。这将使FuncAtude3的try块中的代码执行，但不幸的是，FuncAtude3的局部变量sz没有变化，恢复执行失败的指令只是产生另一个异常。这样，又造成死循环。

前面说过，系统要查找最近执行的与except块相匹配的try块，并计算它的过滤器值。这就是说，系统在查找过程当中，将略过那些与finally块相匹配而不是与except块相匹配的try块。这样做的理由很明显：finally块没有异常过滤器，系统没有什么要计算的。如果前面例子中FuncAtude3包含一个finally块而不是except块，系统将在一开始就通过FunclinRoosevelt3的OilFilter3计算异常过滤器的值。

第25章提供有关EXCEPTION_CONTINUE_SEARCH的更多信息。

24.5 GetExceptionCode

一个异常过滤器在确定要返回什么值之前，必须分析具体情况。例如，异常处理程序可能知道发生了除以0引起的异常时该怎么做，但是不知道该如何处理一个内存存取异常。异常过滤器负责检查实际情况并返回适当的值。

下面的代码举例说明了一种方法，指出所发生异常的类别：

```
__try {
    x = 0;
    y = 4 / x;
}
```

```
__except ((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO) ?
   EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
   // Handle divide by zero exception.
}
```

内部函数GetExceptionCode返回一个值，该值指出所发生异常的种类：

```
DWORD GetExceptionCode();
```

下面列出所有预定义的异常和相应的含意，这些内容取自 Platform SDK文档。这些异常标识符可以在WinBase.h文件中找到。我们对这些异常做了分类。

1. 与内存有关的异常

- EXCEPTION_ACCESS_VIOLATION。线程试图对一个虚地址进行读或写，但没有做适当的存取。这是最常见的异常。
- EXCEPTION_DATATYPE_MISALIGNMENT。线程试图读或写不支持对齐（alignment）的硬件上的未对齐的数据。例如，16位数值必须对齐在2字节边界上，32位数值要对齐在4字节边界上。
- EXCEPTION_ARRAY_BOUNDS_EXCEEDED。线程试图存取一个越界的数组元素，相应的硬件支持边界检查。
- EXCEPTION_IN_PAGE_ERROR。由于文件系统或一个设备启动程序返回一个读错误，造成不能满足要求的页故障。
- EXCEPTION_GUARD_PAGE。一个线程试图存取一个带有PAGE_GUARD保护属性的内存页。该页是可存取的，并引起一个EXCEPTION_GUARD_PAGE异常。
- EXCEPTION_STACK_OVERFLOW。线程用完了分配给它的所有栈空间。
- EXCEPTION_ILLEGAL_INSTRUCTION。线程执行了一个无效的指令。这个异常由特定的CPU结构来定义；在不同的CPU上，执行一个无效指令可引起一个陷井错误。
- EXCEPTION_PRIV_INSTRUCTION。线程执行一个指令，其操作在当前机器模式中不允许。

2. 与异常相关的异常

- EXCEPTION_INVALID_DISPOSITION。一个异常过滤器返回一值，这个值不是EXCEPTION_EXECUTE_HANDLER、EXCEPTION_CONTINUE_SEARCH、EXCEPTION_CONTINUE_EXECUTION三者之一。
- EXCEPTION_NONCONTINUABLE_EXCEPTION。一个异常过滤器对一个不能继续的异常返回EXCEPTION_CONTINUE_EXECUTION。

3. 与调试有关的异常

- EXCEPTION_BREAKPOINT。遇到一个断点。
- EXCEPTION_SINGLE_STEP。一个跟踪陷井或其他单步指令机制告知一个指令已执行完毕。
- EXCEPTION_INVALID_HANDLE。向一个函数传递了一个无效句柄。

4. 与整数有关的异常

- EXCEPTION_INT_DIVIDE_BY_ZERO。线程试图用整数0来除一个整数
- EXCEPTION_INT_OVERFLOW。一个整数操作的结果超过了整数值规定的范围。

5. 与浮点数有关的异常

- EXCEPTION_FLT_DENORMAL_OPERAND。浮点操作中的一个操作数不正常。不正常的值是一个太小的值，不能表示标准的浮点值。
- EXCEPTION_FLT_DIVIDE_BY_ZERO。线程试图用浮点数0来除一个浮点。

- EXCEPTION_FLT_INEXACT_RESULT。浮点操作的结果不能精确表示成十进制小数。
- EXCEPTION_FLT_INVALID_OPERATION。表示任何没有在此列出的其他浮点数异常。
- EXCEPTION_FLT_OVERFLOW。浮点操作的结果超过了允许的值。
- EXCEPTION_FLT_STACK_CHECK。由于浮点操作造成栈溢出或下溢。
- EXCEPTION_FLT_UNDERFLOW。浮点操作的结果小于允许的值。

内部函数GetExceptionCode只能在一个过滤器中调用（--except之后的括号里），或在一个异常处理程序中被调用。下面的代码是合法的：

```
__try {
   y = 0;
   x = 4 / y;
}

__except (
   ((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ||
    (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)) ?
   EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {

   switch (GetExceptionCode()) {
      case EXCEPTION_ACCESS_VIOLATION:
         // Handle the access violation.
         .
         .
         .
         break;
      case EXCEPTION_INT_DIVIDE_BY_ZERO:
         // Handle the integer divide by 0.
         .
         .
         .
         break;
   }
}
```

但是，不能在一个异常过滤器函数里面调用 GetExceptionCode。编译程序会捕捉这样的错误。当编译下面的代码时，将产生编译错。

```
__try {
   y = 0;
   x = 4 / y;
}

__except (CoffeeFilter()) {

   // Handle the exception.
   .
   .
   .
}

LONG CoffeeFilter (void) {
   // Compilation error: illegal call to GetExceptionCode.
   return((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

可以按下面的形式改写代码：

```
__try {
```

```
   y = 0;
   x = 4 / y;
}

__except (CoffeeFilter(GetExceptionCode())) {

   // Handle the exception.
   .
   .
   .
}

LONG CoffeeFilter (DWORD dwExceptionCode) {
   return((dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) ?
      EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

异常代码遵循在WinError.h文件中定义的有关错误代码的规则。每个DWORD被划分如表24-2所示。

表24-2 一个错误代码的构成

位	31-30	29	28	27-16	15-0
内容	严重性系数	微软/客户	保留	设备代码	异常代码
意义	0=成功 1=信息 2=警告 3=错误	0=微软定义的代码 1=客户定义的代码	必须为0	微软定义(见表24-3)	微软／客户定义

目前，微软定义了下面一些设备代码(见表24-3)。

表24-3 设备代码及其值

设备代码	值	设备代码	值
FACILITY_NULL	0	FACILITY_CONTROL	10
FACILITY_RPC	1	FACILITY_CERT	11
FACILITY_DISPATCH	2	FACILITY_INTERNET	12
FACILITY_STORAGE	3	FACILITY_MEDIASERVER	13
FACILITY_ITF	4	FACILITY_MSMQ	14
FACILITY_WIN32	7	FACILITY_SETUPAPI	15
FACILITY_WINDOWS	8	FACILITY_SCARD	16
FACILITY_SECURITY	9	FACILITY_COMPLUS	17

我们将EXCEPTION_ACCESS_VIOLATION异常代码拆开来，看各位(bit)都是什么。在WinBase.h中找到EXCEPTION_ACCESS_VIOLATION，它的值为0xC0000005：

C 0 0 0 0 0 0 5 （十六进制）

1100 0000 0000 0000 0000 0000 0000 0101 （二进制）

第30位和第31位都是1，表示存取异常是一个错误（线程不能继续运行）。第29位是0，表示Microsoft已经定义了这个代码。第28位是0，留待后用。第16位至27位是0，代表FACILITY_NULL（存取异常可发生在系统中任何地方，不是使用特定设备才发生的异常）。第0位到第15位包含一个数5，表示微软将存取异常这种异常的代码定义成5。

24.6 GetExceptionInformation

当一个异常发生时，操作系统要向引起异常的线程的栈里压入三个结构，这三个结构是：

EXCEPTION_RECORD结构、CONTEXT结构和EXCEPTION_POINTERS结构。

EXCEPTION_RECORD结构包含有关已发生异常的独立于CPU的信息，CONTEXT结构包含已发生异常的依赖于CPU的信息。EXCEPTION_POINTERS结构只有两个数据成员，二者都是指针，分别指向被压入栈的EXCEPTION_RECORD和CONTEXT结构：

```
typedef struct _EXCEPTION_POINTERS {
   PEXCEPTION_RECORD ExceptionRecord;
   PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

为了取得这些信息并在你自己的程序中使用这些信息，需要调用 GetExceptionInformation 函数：

```
PEXCEPTION_POINTERS GetExceptionInformation();
```

这个内部函数返回一个指向EXCEPTION_POINTERS结构的指针。

关于GetExceptionInformation函数，要记住的最重要事情是它只能在异常过滤器中调用，因为仅仅在处理异常过滤器时， CONTEXT、EXCEPTION_RECORD和EXCEPTION_POINTERS才是有效的。一旦控制被转移到异常处理程序，栈中的数据就被删除。

如果需要在你的异常处理程序块里面存取这些异常信息（虽然很少有必要这样做），必须将EXCEPTION_POINTERS结构所指向的CONTEXT数据结构和／或EXCEPTION_RECORD数据结构保存在你所建立的一个或多个变量里。下面的代码说明了如何保存 EXCEPTION_RECORD和CONTEXT数据结构：

```
void FuncSkunk() {
   // Declare variables that we can use to save the exception
   // record and the context if an exception should occur.
   EXCEPTION_RECORD SavedExceptRec;
   CONTEXT SavedContext;
   :
   __try {
      :
   }

   __except (
      SavedExceptRec =
         *(GetExceptionInformation())->ExceptionRecord,
      SavedContext =
         *(GetExceptionInformation())->ContextRecord,
      EXCEPTION_EXECUTE_HANDLER) {

      // We can use the SavedExceptRec and SavedContext
      // variables inside the handler code block.
      switch (SavedExceptRec.ExceptionCode) {
         :
      }
   }
   :
}
```

注意在异常过滤器中C语言逗号（,）操作符的使用。许多程序员不习惯使用这个操作符。它告诉编译程序从左到右执行以逗号分隔的各表达式。当所有的表达式被求值之后，返回最后的（或最右的）表达式的结果。

在FuncSkunk中，左边的表达式将执行，将栈中的 EXCEPTION_RECORD结构保存在Saved ExceptRec局部变量里。这个表达式的结果是SavedExceptRec的值。这个结果被丢弃，再计算右边下一个表达式。第二个表达式将栈中的 CONTEXT结构保存在SavedContext局部变量里。第二个表达式的结果是 SavedContext，同样当计算第三个表达式时丢弃第二个表达式的结果。第三个表达式很简单，只是一个数值EXCEPTION_EXECUTE_HANDLER。这个最右边的表达式的结果就是整个由逗号分隔的表达式组的结果。

由于异常过滤器的值是 EXCEPTION_EXECUTE_HANDLER，except块中的代码要执行。这时，已被初始过的 SavedExceptRec和SavedContext变量可以在 except块中使用。要记住，SavedExceptRec和SavedContext变量要在try块之外说明，这一点很重要。

我们都可以猜到， EXCEPTION_POINTERS结构的 ExceptionRecord成员指向EXCEPTION_RECORD结构：

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

EXCEPTION_RECORD结构包含有关最近发生的异常的详细信息，这些信息独立于 CPU：

- ExceptionCode包含异常的代码。这同内部函数GetExceptionCode返回的信息是一样的。
- ExceptionFlags包含有关异常的标志。当前只有两个值，分别是 0（指出一个可以继续的异常）和EXCEPTION_NONCONTINUABLE（指出一个不可继续的异常）。在一个不可继续的异常之后，若要继续执行，会引发一个 EXCEPTION_NONCONTINUABLE_EXCEPTION异常。
- ExceptionRecord指向另一个未处理异常的EXCEPTION_RECORD结构。在处理一个异常的时候，有可能引发另外一个异常。例如，异常过滤器中的代码就可能用零来除一个数。当嵌套异常发生时，可将异常记录链接起来，以提供另外的信息。如果在处理一个异常过滤器的过程当中又产生一个异常，就发生了嵌套异常。如果没有未处理异常，这个成员就包含一个NULL。
- ExceptionAddress指出产生异常的CPU指令的地址。
- NumberParameters规定了与异常相联系的参数数量（0到15）。这是在ExceptionInformation数组中定义的元素数量。对几乎所有的异常来说，这个值都是零。
- ExceptionInformation规定一个附加参数的数组，用来描述异常。对大多数异常来说，数组元素是未定义的。

EXCEPTION_RECORD结构的最后两个成员，NumberParameters和ExceptionInformation向异常过滤器提供一些有关异常的附加信息。目前只有一种类型的异常提供附加信息，就是EXCEPTION_ACCESS_VIOLATION。所有其他可能的异常都将 NumberParameters设置成零。我们可以检验ExceptionInformation的数组成员来查看关于所产生异常的附加信息。

对于一个EXCEPTION_ACCESS_VIOLATION异常来说，ExceptionInformation[0]包含一个

标志，指出引发这个存取异常的操作的类型。如果这个值是 0，表示线程试图要读不可访问的数据。如果这个值是1，表示线程要写不可访问的数据。ExceptionInformation[1] 指出不可访问数据的地址。

通过使用这些成员，我们可以构造异常过滤器，提供大量有关程序的信息。例如，可以这样编写异常过滤器：

```
__try {
   .
   .
   .
}
__except (ExpFltr(GetExceptionInformation()->ExceptionRecord)) {
   .
   .
   .
}

LONG ExpFltr (PEXCEPTION_RECORD pER) {
   char szBuf[300], *p;
   DWORD dwExceptionCode = pER->ExceptionCode;

   sprintf(szBuf, "Code = %x, Address = %p",
      dwExceptionCode, pER->ExceptionAddress);
   // Find the end of the string.
   p = strchr(szBuf, 0);

   // I used a switch statement in case Microsoft adds
   // information for other exception codes in the future.
   switch (dwExceptionCode) {
      case EXCEPTION_ACCESS_VIOLATION:
         sprintf(p, "Attempt to %s data at address %p",
            pER->ExceptionInformation[0] ? "write" : "read",
            pER->ExceptionInformation[1]);
         break;

      default:
         break;
   }

   MessageBox(NULL, szBuf, "Exception", MB_OK | MB_ICONEXCLAMATION);

   return(EXCEPTION_CONTINUE_SEARCH);
}
```

EXCEPTION_POINTERS结构的ContextRecord成员指向一个CONTEXT结构（第7章讨论过）。这个结构是依赖于平台的，也就是说，对于不同的CPU平台，这个结构的内容也不一样。

本质上，对CPU上每一个可用的寄存器，这个结构相应地包含一个成员。当一个异常被引发时，可以通过检验这个结构的成员找到更多的信息。遗憾的是，为了得到这种可能的好处，要求程序员编写依赖于平台的代码，以确认程序所运行的机器，使用适当的 CONTEXT结构。最好的办法是在代码中安排一个 #ifdefs指令。Windows支持的不同CPU的CONTEXT结构定义在WinNT.h文件中。

24.7 软件异常

迄今为止，我们一直在讨论硬件异常，也就是 CPU捕获一个事件并引发一个异常。在代码

中也可以强制引发一个异常。这也是一个函数向它的调用者报告失败的一种方法。传统上，失败的函数要返回一些特殊的值来指出失败。函数的调用者应该检查这些特殊值并采取一种替代的动作。通常，这个调用者要清除所做的事情并将它自己的失败代码返回给它的调用者。这种错误代码的逐层传递会使源程序的代码变得非常难于编写和维护。

另外一种方法是让函数在失败时引发异常。用这种方法，代码更容易编写和维护，而且也执行得更好，因为通常不需要执行那些错误测试代码。实际上，仅当发生失败时也就是发生异常时才执行错误测试代码。

但令人遗憾的是，许多开发人员不习惯于在错误处理中使用异常。这有两方面的原因。第一个原因是多数开发人员不熟悉SEH。即使有一个程序员熟悉它，但其他程序员可能不熟悉它。如果一个程序员编写了一个引发异常的函数，但其他程序员并不编写SEH框架来捕获这个异常，那么进程就会被操作系统结束。

开发人员不使用SEH的第二个原因是它不能移植到其他操作系统。许多公司的产品要面向多种操作系统，因此希望有单一的源代码作为产品的基础，这是可以理解的。 SEH是专门针对Windows的技术。

本段讨论通过异常返回错误有关的内容。首先，让我们看一看 Windows Heap函数，例如HeapCreate、heapAlloc等。回顾第18章的内容，我们知道这些函数向开发人员提供一种选择。通常当某个堆（heap）函数失败，它会返回 NULL来指出失败。然而可以对这些堆函数传递HEAP_GENERATE_EXCEPTIONS标志。如果使用这个标志并且函数失败，函数不会返回NULL，而是由函数引发一个 STATUS_NO_MEMORY软件异常，程序代码的其他部分可以用SEH框架来捕获这个异常。

如果想利用这个异常，可以编写你的 try块，好像内存分配总能成功。如果内存分配失败，可以利用except块来处理这个异常，或通过匹配 try块与finally块，清除函数所做的事。这非常方便。

程序捕获软件异常采取的方法与捕获硬件异常完全相同。也就是说，前一章介绍的内容可以同样适用于软件异常。

本节重讨论如何让你自己的函数引发软件异常，作为指出失败的方法。实际上，可以用类似于微软实现堆函数的方法来实现你的函数：让函数的调用者传递一个标志，告诉函数如何指出失败。

引发一个软件异常很容易，只需要调用RaiseException函数：

```
VOID RaiseException(
   DWORD dwExceptionCode,
   DWORD dwExceptionFlags,
   DWORD nNumberOfArguments,
   CONST ULONG_PTR *pArguments);
```

第一个参数 dwExceptionCode是标识所引发异常的值。 HeapAlloc函数对这个参数设定STATUS_NO_MEMORY。如果程序员要定义自己的异常标识符，应该遵循标准 Windows错误代码的格式，像WinError.h文件中定义的那样。参阅表24-1。

如果要建立你自己的异常代码，要填充DWORD的4个部分：

- 第31位和第30位包含严重性系数(severity)。
- 第29位是1（0表示微软建立的异常，如HeapAlloc的STATUS_NO_MEMORY）。
- 第28位是0。
- 第27位到16位是某个微软定义的设备代码。

• 第15到0位是一个任意值，用来标识引起异常的程序段。

RaiseException的第二个参数 dwExceptionFlags，必须是 0或EXCEPTION_NONCONTINUABLE。本质上，这个标志是用来规定异常过滤器返回 EXCEPTION_CONTINUE _EXECUTION来响应所引发的异常是否合法。如果没有向 RaiseException传递EXCEPTION_ NONCONTINUABLE参数值，则过滤器可以返回 EXCEPTION_CONTINUE_EXECUTION。正常情况下，这将导致线程重新执行引发软件异常的同一 CPU指令。但微软已做了一些动作，所以在调用RaiseException函数之后，执行会继续进行。

如果你向RaiseException传递了EXCEPTION_NONCONTINUABLE标志，你就是在告诉系统，你引发异常的类型是不能被继续执行的。这个标志在操作系统内部被用来传达致命（不可恢复）的错误信息。另外，当 HeapAlloc引发STATUS_NO_MEMORY软件异常时，它使用EXCEPTION_NONCONTINUABLE标志来告诉系统，这个异常不能被继续。意思就是没有办法强制分配内存并继续运行。

如果一个过滤器忽略EXCEPTION_NONCONTINUABLE并返回EXCEPTION_CONTINUE_EXECUTION，系统会引发新的异常：EXCEPTION_NONCONTINUABLE_EXCEPTION。

当程序在处理一个异常的时候，有可能又引发另一个异常。比如说，一个无效的内存存取有可能发生在一个 finally块、一个异常过滤器、或一个异常处理程序里。当发生这种情况时，系统压栈异常。回忆一下GetExceptionInformation函数。这个函数返回EXCEPTION_ POINTERS结构的地址。EXCEPTION_POINTERS的ExceptionRecord成员指向一个EXCEPTION_ RECORD结构，这个结构包含另一个 ExceptionRecord成员。这个成员是一个指向另外的 EXCEPTION_RECORD的指针，而这个结构包含有关以前引发异常的信息。

通常系统一次只处理一个异常，并且ExceptionRecord成员为NULL。然而如果处理一个异常的过程中又引发另一个异常，第一个EXCEPTION_RECORD结构包含有关最近引发异常的信息，并且这个EXCEPTION_RECORD结构的ExceptionRecord成员指向以前发生的异常的EXCEPTION_RECORD结构。如果增加的异常没有完全处理，可以继续搜索这个 EXCEPTION_RECORD结构的链表，来确定如何处理异常。

RaiseException的第三个参数nNumberOfArguments和第四个参数pArguments，用来传递有关所引发异常的附加信息。通常，不需要附加的参数，只需对 pArguments参数传递NULL，这种情况下， RaiseException函数忽略 nNumberOfArguments参数。如果需要传递附加参数，nNumberOfArguments参数必须规定由 pArguments参数所指向的ULONG_PTR数组中的元素数目。这个数目不能超过EXCEPTION_MAXIMUM_PARAMETERS，EXCEPTION_ MAXIMUM_PARAMETERS 在WinNT.h中定义成15。

在处理这个异常期间，可使异常过滤器参照 EXCEPTION_RECORD结构中的 NumberParameters和ExceptionInformation成员来检查nNumberOfArguments和pArguments参数中的信息。

你可能由于某种原因想在自己的程序中产生自己的软件异常。例如，你可能想向系统的事件日志发送通知消息。每当程序中的一个函数发现某种问题，你可以调用 RaiseException并让某些异常处理程序上溯调用树查看特定的异常，或者将异常写到日志里或弹出一个消息框。你还可能想建立软件异常来传达程序内部致使错误的信息。