Windows 程序设计

fred@real.net.cn

课件 14 异常与程序调试

基本目标: 了解 SEH 异常，掌握利用 MAP 文件调试异常的技术

结构化异常处理

如果一个优美的编程环境,总是有足够的内存,没有人会传递给你一个无效的指针,需要的文件总是存在.利用 SHE,你可以完全不用考虑代码里是不是有错误，这样就把主要的工作同错误处理分离开来。我们可以利用 SHE，使得我们自己的程序更加健壮，但是 SEH 机制也是可以被破坏的。

使用 SEH 的负担主要由编译程序来负担，而不是操作系统来承担。当异常块出现时，编译程序需要生成特殊的代码。编译程序负责准备堆栈的结构和其他的内部信息，供操作系统使用和参考。

SEH 主要包括两个功能：结束处理（termination handing）和异常处理(Exception handing)。

一个结束处理程序能够确保取调用和执行一个代码块（结束处理程序），而不管另外一块代码（保护体是如何退出的）。结束处理的格式如下

```
__try
{
//保护体
}
__finally
{
//结束处理程序
}
```

一个利用结束处理的例子：

```
long dofunc()
{
    long nRet = 0;
    __try
    {
        printf("in guard body\n");
        nRet = 3;
        return nRet;
    }
    __finally
    {
        printf("in termination handler\n");
        nRet = 4;
    }
    printf("At last of dofunc\n");
    nRet = 9;
```

```
    return nRet;
}
```

函数执行后的输出结果是：

```
in guard body
in termination handler
return value:3
```

当编译器检查源代码时，它看到在 try 块中有 return 语句，这样编译程序就生成代码将返回值保存在一个编译程序建立的一个临时变量中。编译程序然后再生成代码来执行 finally 块中包含的指令，这成为局部展开。在 Finally 块中的指令执行之后，编译程序临时变量的值被取出并从函数返回（这也是对返回值的修改没有起作用的原因）。

在编写代码时，应该避免引起结束处理程序的 try 块中的过早退出，因为这会影响程序的性能。__leave 关键字有助于编写避免引起局部展开的代码。当控制流自然地离开 try 块并进入 finally 块时，开销是最小的。

一个使用__leave 关键字的例子：

```
long dofunc3()
{
    long nRet = 0;
    __try
    {
        printf("in guard body\n");
        __leave;
        nRet = 3;
        return nRet;
    }
    __finally
    {
        printf("in termination handler\n");
        nRet = 4;
    }
    printf("At last of dofunc\n");
    nRet = 9;
    return nRet;
}
```

```
in guard body
in termination handler
At last of dofunc
return value:9
```

尽管结束处理程序可以捕捉 try 块过早退出的大多数情况，比如使用 break、continue、return、goto、longjump 退出代码块。但是当线程或者进程退出的时候它不能引起__finally 块中的代码执行。

在 finally 中，可以利用 AbnormalTermination()来判定 try 块中的代码是否是过早退出。如果控制流是离开 try 块并自然地进入 finally 块（使用__leave 离开 try 块是自然的离开），函数的返回值是 FALSE，如果控制流是非正常地退出－－通常是由于 goto、return、break、continue 语句引起的局部展开，或者由于内存访问违法等异常引起的全局展开，函数的返回

值是 TRUE。

使用结束处理程序的理由：

- 简化错误处理，因为所有的清理工作都在一个位置并且保证执行
- 提高程序的可读性
- 使得代码更加容易维护
- 如果使用得当，具有最小的系统开销

异常处理程序和软件异常

对于存取一个无效的内存地址或者用 0 来除一个数值这类错误是经常发生的，这虽然不是我们希望的。（一个软件的崩溃并没有导致世界发生混乱，这是隔离的好处，也是提供高质量软件的重要方法与技术，CPU 负责捕获无效内存访问和用 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，异常由软件系统进行处理，或者由作为软件系统一部分的人来进行处理）。CPU 引发的异常，就是硬件异常，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。

异常处理程序的语法如下：

```
void DoFunc()
{
    __try
    {
        //异常保护体
    }
    __except(EXCEPTION_EXECUTE_HANDLER)//异常过滤器
    {
        //异常处理程序
    }
}
```

异常过滤器的值只能是下面三个值之一:

```
EXCEPTION_EXECUTE_HANDLER
EXCEPTION_CONTINUE_EXECUTION
EXCEPTION_CONTINUE_SEARCH
```

为了加深对异常运作机制的理解,下面给出一个例子来说明异常的运作原理:

```
DWORD ExceptionFilter1()
{
    printf("In Exception Filter1\n");
    return EXCEPTION_CONTINUE_SEARCH;
}
void func1()
{
    __try
    {
        __try
        {
            char* p = 0;
```

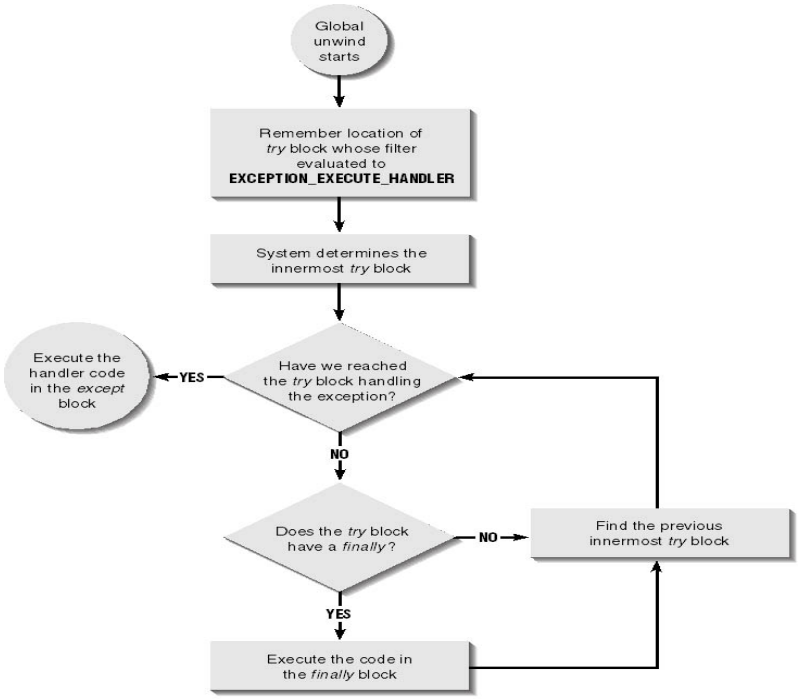
```
            *p = 0;
        }
        __except(ExceptionFilter1())
        {
            printf("In Exception1 handler\n");
        }
    }
    __finally
    {
        printf("In Func1 finally handler\n");
    }
}

DWORD ExceptionFilter2()
{
    printf("In Exception Filter2\n");
    return EXCEPTION_EXECUTE_HANDLER;
}
void func2()
{
    __try
    {
        func1();
    }
    __except(ExceptionFilter2())
    {
        printf("In Exception2 handler\n");
    }
}
int main(int argc, char* argv[])
{
    func2();
    return 0;
}
```

运行输出的结果是:

```
In Exception Filter1
In Exception Filter2
In Func1 finally handler
In Exception2 handler
```

上面程序运作的解释： 当一个异常过滤器的值为 EXCEPTION_EXECUTE_HANDLER,系统必须执行一个全局展开。全局展开的流程图如下图所示。




```
DWORD ExceptionFilter1()
{
    printf("In Exception Filter1\n");
    return EXCEPTION_CONTINUE_SEARCH;
}
void func1()
{
    __try
    {
//进行一些处理
        __try
        {
//在这里发生一个异常
            char* p = 0;
            *p = 0;
        }
        __except(ExceptionFilter1())
        {
            printf("In Exception1 handler\n");
        }
    }
    __finally
    {
//进行全局展开会执行这里
        printf("In Func1 finally handler\n");
    }
```

```
}

DWORD ExceptionFilter2()
{
    printf("In Exception Filter2\n");
    return EXCEPTION_EXECUTE_HANDLER;
}
void func2()
{
//(1)首先执行这里
    __try
    {
//(2)在这里调用另外一个函数
        func1();
//由于函数内部会发生异常,并且向外传递,这里的代码不会被执行到
    }
    __except(ExceptionFilter2())
    {
//在进行全局展开以后,会执行这里的异常处理程序
        printf("In Exception2 handler\n");
    }
}
int main(int argc, char* argv[])
{
    func2();
    return 0;
}
```

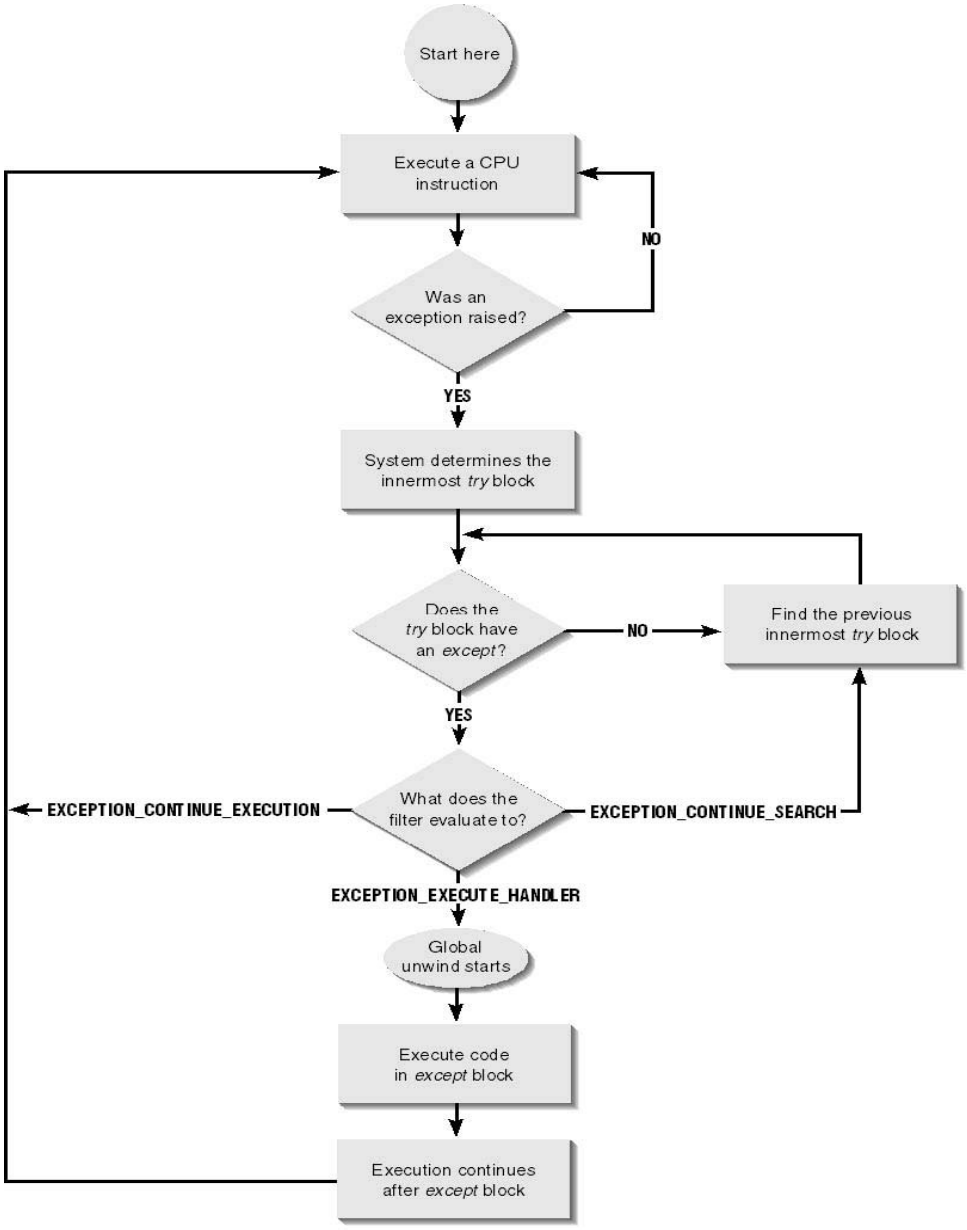
在进行全局展开的时候,首先进行异常过滤表达式的计算,当系统看到一个EXCEPTION_EXECUTE_HANDLER 的表达式的时候,系统就开始一个全局展开。SHE 比较难于理解，因为在代码执行的过程中与系统的交互太多。当一个过滤器返回EXCEPTION_EXECUTE_HANDLER 时，过滤器是在告诉系统，线程的指令指针应该指向except 块中的代码。但是目前这个指针可能不在那里，这样每当线程需要从一个 try－finally 块离开时，必须保证执行 finally 块中的代码。在发生异常的时候，全局展开就是保证这条规则的机制。

在异常过滤表达式中可以调用函数 GetExceptionCode（）函数得到目前发生异常的种类：

- 与内存相关的异常（EXCEPTION_ACCESS_VIOLATION,EXCEPTION_GUARD_PAGE,EXCEPTION_STACK_OVERFLOW,EXCEPTION_ILLEGAL_INSTRUCTION）
- 与异常相关的异常()
- 与调试相关的异常(EXCEPTION_BREAKPOINT,EXCEPTION_SINGLE_STEP,EXCEPTION_INVALID_HANDLE)
- 与整数相关的异常(EXCEPTION_INT_DIVIDE_BY_ZERO)

- 与浮点数相关的异常(EXCEPTION_FLT_DIVIDE_BY_ZERO)

在 Windows 操作系统中，下图描述了一个异常的发生过程：



当一个异常发生的时候，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：为了得到这些结构的信息，我们可以使用函数 GetExceptionInformation(),需要注意的是，该函数只能在异常过滤表达式中进行调用。一旦控制被转移到异常处理程序，栈中的数据就被删除。如果需要在异常处理程序中使用，可以将他们进行保存到我们建立的变量中。

C＋＋异常与结构化异常

SHE 是可以用于任何编程语言的操作系统设施，C＋＋异常处理只能用来编写 C＋＋程序。如果利用 C＋＋语言编写程序，应该使用 C＋＋的异常处理。原因是在异常发生的时刻，可以保证 C＋＋对象被析构。但是对于访问违法这类错误，C＋＋异常处理是否可以进行捕捉是需要查看系统的异常模式。

C＋＋异常是不可以被修复进行重试的，也就是说，C＋＋异常的过滤表达式不可能返回 EXCEPTION_CONTINUE_EXECUTION，使得 CPU 重试失败的指令。在正常的情况下，

C＋＋异常处理不能使得程序从硬件异常中进行恢复。

如果在发生硬件异常的时候，C＋＋异常处理能够区分不同的异常类型就比较好。VC 提供了一种机制，可以将结构化异常转化为 C＋＋异常。

同步异常与异步异常模型

在使用 C++异常处理时,必须理解异步异常和同步异常处理之间的差别。它们之间的差别在于：编译器为程序生成异常处理代码的方法，取决于编译器所假定的引发异常的方法。

在异步 C＋＋异常处理中，编译器假定每条指令都可以生成一个异常，并且代码必须已经准备好处理任何地方出现的异常。VC 5 默认的异常处理模型是异步异常处理。

同步异常处理是编译器期望程序员使用一个显式的 throw 语句来引发异常。如果没有 throw 语句，那么则不会生成异常处理代码。

编译器默认的/GX 开关映射到/Ehsc(同步异常)，因此若要强行打开异步异常，需要显式地使用开关/Eha(异步异常)。如果使用异步异常，但是又想一些不会产生异常地函数生成紧凑的代码，可以使用__declspec(nothrow)来声明或者定义这些函数。

异常模型对于代码地生成影响很大，下面用一个函数为例来说明：

```
class CObj
{
public:
    CObj();
    ~CObj();
    void DoFunc();
};
CObj::CObj()
{
    printf("CObj\n");
}
CObj::~CObj()
{
    printf("~CObj\n");
}
void CObj::DoFunc()
{
    printf("Do Something\n");
}

void TestFunc()
{
    CObj oo;
    oo.DoFunc();
}

//异步异常生成地代码
234:    void TestFunc()
235:    {
```

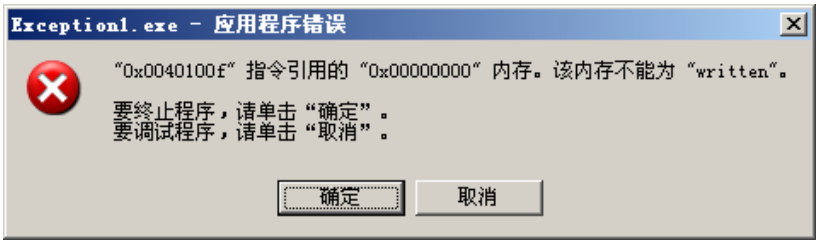
```
00401550     push            0FFh
00401552     push            offset $L17305 (00407d48)
00401557     mov              eax,fs:[00000000]
0040155D     push            eax
0040155E     mov              dword ptr fs:[0],esp
00401565     push            ecx
236:         CObj oo;
00401566     lea             ecx,[esp]
0040156A     call            CObj::CObj (00401510)
0040156F     mov              dword ptr [esp+0Ch],0
237:         oo.DoFunc();
00401577     lea             ecx,[esp]
0040157B     call            CObj::DoFunc (00401540)
238:    }
00401580     mov              dword ptr [esp+0Ch],0FFFFFFFFh
00401588     lea             ecx,[esp]
0040158C     call            CObj::~CObj (00401530)
00401591     mov              ecx,dword ptr [esp+4]
00401595     mov              dword ptr fs:[0],ecx
0040159C     add              esp,10h
0040159F     ret
```

//同步异常生成地代码

```
234:    void TestFunc()
235:    {
00401550     push            ecx
236:         CObj oo;
00401551     lea             ecx,[esp]
00401555     call            CObj::CObj (00401510)
237:         oo.DoFunc();
0040155A     lea             ecx,[esp]
0040155E     call            CObj::DoFunc (00401540)
238:    }
00401563     lea             ecx,[esp]
00401567     call            CObj::~CObj (00401530)
0040156C     pop              ecx
0040156D     ret
```

开发环境中异常的运作

在我们的程序中，如果异常没有被我们处理，那么就出现了“未处理异常”。这时，系统会调用一个默认的由系统提供的一个异常过滤器处理函数：UnHandledExceptionFilter（），这个函数负责显示一个消息框，指出存在一个未处理的异常，可以让用户结束或者调试这个进程。在 Windows2000 操作系统中，这个消息框可能如图所示：

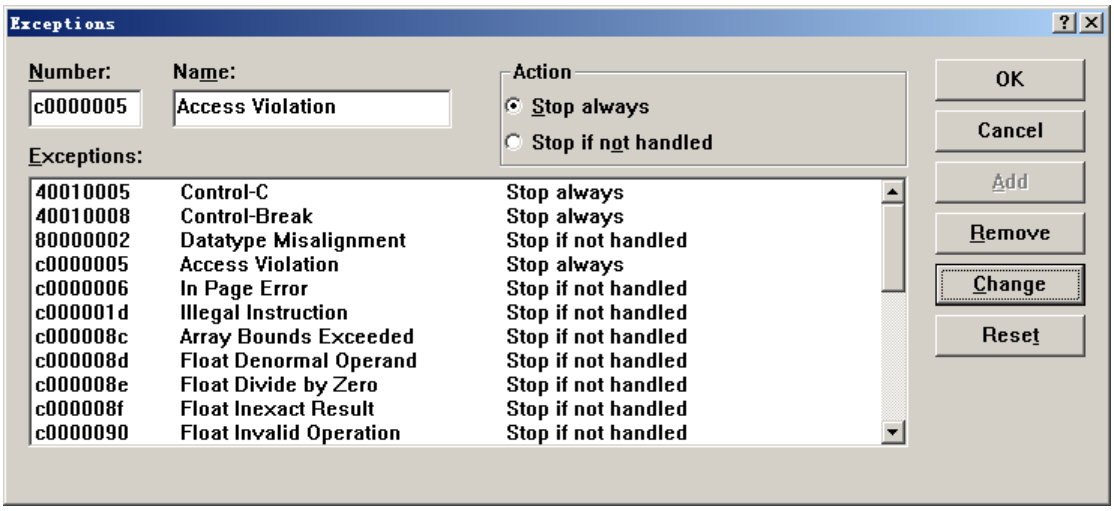


如果单击取消按钮，就可以将调试程序动态地挂接在运行中的进程上，这是 Windiws 提供的最好的特性之一。这种随时可以将调试程序连接到任何进程的能力成为即时调试(Just-in-time Debugging)。进行即时调试时，系统在下面地方找到调试程序：

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

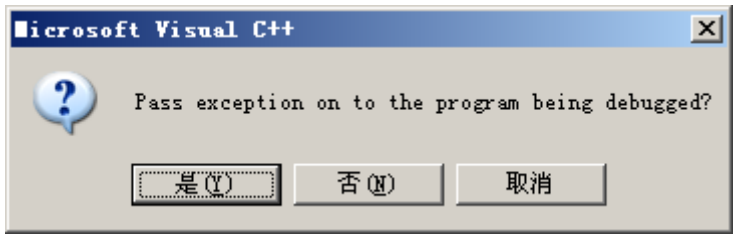
比如在你的安装了 VC 的机器上，Debugger 的值可能是："C:\Program Files2003\vc\Common\MSDev98\Bin\msdev.exe" -p %ld -e %ld。

Microsoft Visual C++调试程序支持调试异常。当一个进程的线程引起一个异常，操作系统立即通知调试程序。这个通知被称作最初通知（First chance notification）。正常情况下，调试程序要响应最初机会通知，告诉线程去搜索异常过滤器。如果所有的异常过滤器都返回 EXCEPTION_CONTINUE_SEARCH，操作系统再次通知调试程序，称为最后机会通知(last chance notification)。



在上面的窗口中，选择访问违法（Access Violation）异常，将它的动作改变成 Stop Always。这样，当被调试的程序中的一个线程引起一个访问违法时，调试程序会接到最初的机会通知。显示如下的对话框：

在这里，线程还没有机会去搜索异常过滤器，现在我们可以在代码中设置断点，检查变量，或者检查线程的调用堆栈。现在还没有异常过滤器执行，只是异常发生了而已。（这是一个及其好的调试方法，在一个异常发生的时刻可以检查进程的状态，引起异常的代码，然后可以解决异常）如果我们现在再继续运行，会出现下面的对话框：



当点击《取消》按钮，返回到调试程序。

点击《否》按钮是告诉被调试的线程去重试失败的 CPU 指令。

对于大多数的异常，重试失败的指令只会再一次引起异常，没有什么用途。但是对于一个由函数 RaiseException 引起的异常，重试是让线程继续执行，好像没有发生异常一样。按这种方式执行，对于调试 C＋＋程序特别有用：相当于一个从未执行的 throw 语句。

单击《是》按钮会使被调试的程序的线程去搜索异常过滤器。如果找到返回值为 EXCEPTION_EXECUTE_HANDLER 或者 EXCEPTION_CONTINUE_EXECUTE 一个异常过滤器,则程序继续运行。如果所有的过滤器都返回 EXCEPTION_CONTINUE_SEARCH,调试程序收到一个最后的机会通知，显示下面的对话框：

在这一步，必须要调试这个程序或者结束这个程序。

事后调试的基本知识

事后调试的两个最基本的目标：

- 发现程序是在哪里崩溃的；（简单地说，就是要找到程序中导致崩溃的指令地址，在程序对应的 MAP 文件里，你能找到错误出现的源码文件名称以及行号）
- 找出导致程序崩溃的原因；（要找到崩溃的原因，你可以通过线程的堆栈跟踪找到出错的函数是在哪里被调用的，通过堆栈内容还可以知道相关变量的值）

调试 Windows 程序需要了解的一个最基本的知识是：Windows 是如何处理错误的。由于 Windows API 函数从来都不抛出异常，所以几乎所有的 Windows 错误都是通过函数返回值进行处理的（由少数几个函数允许用户通过传递适当的标记值使得函数可以抛出异常，比如 HeapCreate、HeapAlloc、HeapReAlloc、InitializeCriticalSection 等）。

Windows API 的错误处理过程可以分为两步：

首先，API 函数返回一个特殊的值，指明有个错误的发生；然后，用户可以通过调用 API 函数 GetLastError 得到这个特殊的错误码。最新的错误码是基于每个线程来设置的，它存储在线程的局部存储空间中,所以一个线程不会破坏另外一个线程的错误码。如果你愿意，你也可以在自己实现的函数中实现对 GetLastError 的支持，方法是：调用 SetLastError API 函数设置错误码，然后返回一个特殊的值来说明某个错误的发生。

错误码的位域有固定的映射格式，在 Winerror.h 中有详细的说明。Windows 的异常代码和 COM 的 HRESULT 码都有同样的映射格式。

Bits30－31：严重性 0（成功）1（供参考）2（警告）3（错误）

Bit29：Microsoft/客户 0（Microsoft 定义的代码） 1（客户定义的代码）

Bit28：保留，必须是 0

Bits16－27：设备代码，有 Microsoft 定义

Bits0－15：异常代码，有 Microsoft 或者客户定义

如果创建自己的错误代码，必须使得第 29 位为 1，这样可以确保我们创建的错误代码与 Microsoft 公司目前或者将来定义的错误代码不会发生冲突。一般我们定义的都是错误,所以我们定义的错误码应该是:0xE000XXXX 的形状。

在 Visual C＋＋中，可以通过在 Watch 窗口中输入@ERR 来监视 GetLastError 函数的返回值。ERR 是调试器用来显示最新错误码的一个虚拟寄存器。你还可以使用@ERR,hr 的格式来将错误码转换成文本格式。如果你希望在程序中将错误码转换成错误文本，可以使用 FormatMessage API 函数。

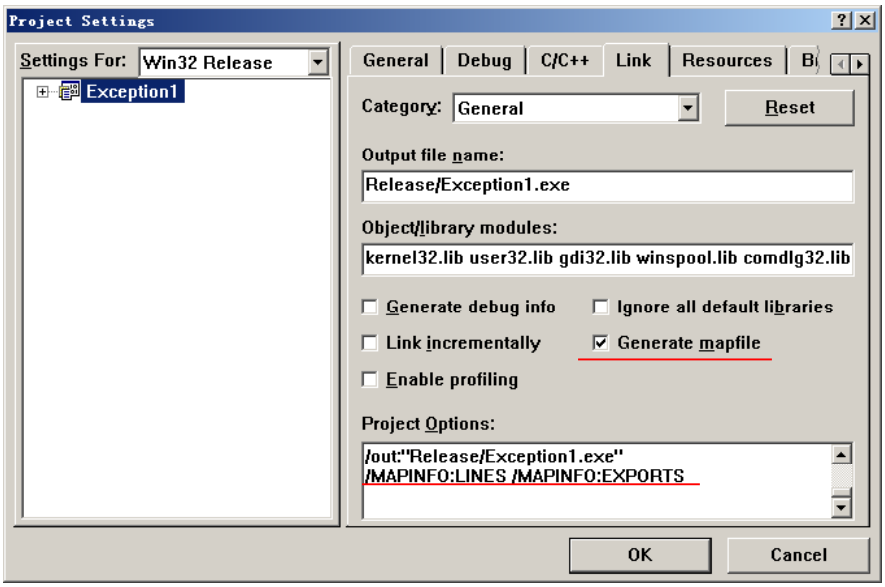
异常调试

Windows 程序的崩溃是由一个没有被处理的异常引起的。实际上，只要你处理了所有的异常，不破坏 Windows，不破坏堆栈，不在一个异常被处理的时候将异常抛出，你可以在 Windows 程序中做任何事情。下面介绍一种利用映射文件进行调试的技术。

映射文件包含模块的最佳装载地址以及程序代码地址和源程序行号的映射。映射文件记录的信息比较原始，但是它是可读性非常好的文本文件，并且不依赖任何版本的 Visual C++。下面结合一个例子进行说明：

1. 为了创建映射文件，需要对 Visual C＋＋工程中进行相应的设置。

a)



b) /MAPINFO:LINES 的作用是输出代码地址和源代码行号的映射作为默认设置。

c) /MAPINFO:EXPORTS 的作用是得到引出序号(ordinal)

2. 查看映射文件：打开与模块匹配的映射文件，顶端是模块的名称和模块的创建时间。接下来是最佳装载地址，这是映射文件假定的装载地址的虚拟地址。如果在装载的时刻发生了虚拟地址空间冲突，那么就会发生模块重新定位的现象。

3. 接下来是找到与崩溃地址匹配的最好的函数。匹配的最好的函数是在崩溃地址上或者在更低的地址上。

4. 最后一步是找到匹配得最好的源文件行号。映射文件是按照相对虚拟地址列举行号，所以我们需要做一些工作。我们需要将崩溃的地址减去实际的基地址和 PE 文件头（0x1000）,利用剩下的值来查找最好的匹配。

一个具体的例子，程序的代码如下：

```
1 // UseMapfile.cpp : Defines the entry point for the console application.
2 //
```

```
3
4 #include "stdafx.h"
5
6 int Div(int i,int j)
7 {
8     return i / j; //注意由于这里 j 会被传递 0，所以会发生除零异常
9 }
10 int Caller()
11{
12    int i = 100;
13    int j = 0;
14
15    int k = Div(i,j);
16    return k;
17}
18 int main(int argc, char* argv[])
19 {
20    printf("Value:%d\n",Caller());
21    return 0;
22}
```

（1）首先利用利用 Debug Configration 的模式进行编译连接，注意设定生成 MAP 文件的选项，运行生成的程序，由于发生被 0 除的未处理异常，系统弹出下面的对话框：

从上面的异常对话框我们可以知道发生异常的指令的位置在：0x0040104c

（2）下面是程序的映射文件（注意：为了节约空间，其中有所省略）

UseMapfile

Timestamp is 41f5fc30 (Tue Jan 25 15:58:40 2005)

Preferred load address is 00400000

Start	Length	Name	Class
0001:00000000	00010930H	.text	CODE
0001:00010930	00010000H	.textbss	CODE
0002:00000000	000013caH	.rdata	DATA
0002:000013ca	00000000H	.edata	DATA
0003:00000000	00000104H	.CRT$XCA	DATA
0003:00000104	00000104H	.CRT$XCZ	DATA

```
 0003:00000208 00000104H .CRT$XIA                 DATA
 0003:0000030c 00000109H .CRT$XIC                 DATA
 0003:00000418 00000104H .CRT$XIZ                 DATA
 0003:0000051c 00000104H .CRT$XPA                 DATA
 0003:00000620 00000104H .CRT$XPX                 DATA
 0003:00000724 00000104H .CRT$XPZ                 DATA
 0003:00000828 00000104H .CRT$XTA                 DATA
 0003:0000092c 00000104H .CRT$XTZ                 DATA
 0003:00000a30 00003216H .data                    DATA
 0003:00003c48 000019b4H .bss                     DATA
 0004:00000000 00000014H .idata$2                 DATA
 0004:00000014 00000014H .idata$3                 DATA
 0004:00000028 00000120H .idata$4                 DATA
 0004:00000148 00000120H .idata$5                 DATA
 0004:00000268 000004f4H .idata$6                 DATA

  Address         Publics by Value              Rva+Base       Lib:Object

 0001:00000030       ?Div@@YAHHH@Z                 00401030 f   UseMapfile.obj
 0001:00000060       ?Caller@@YAHXZ                00401060 f   UseMapfile.obj
 0001:000000c0       _main                         004010c0 f   UseMapfile.obj
 0001:00000110       __chkesp                      00401110 f   LIBCD:chkesp.obj
 0001:00000150       _printf                       00401150 f   LIBCD:printf.obj
 0001:000001d0       _mainCRTStartup               004011d0 f   LIBCD:crt0.obj
 0001:00000300       __amsg_exit                   00401300 f   LIBCD:crt0.obj
 0001:00000360       __CrtDbgBreak                 00401360 f   LIBCD:dbgrpt.obj
 0001:00000370       __CrtSetReportMode            00401370 f   LIBCD:dbgrpt.obj
 0001:000003d0       __CrtSetReportFile            004013d0 f   LIBCD:dbgrpt.obj
 0001:00000450       __CrtSetReportHook            00401450 f   LIBCD:dbgrpt.obj
 0001:00000470       __CrtDbgReport                00401470 f   LIBCD:dbgrpt.obj
……
Line numbers for .\Debug\UseMapfile.obj(C:\0\UseMapfile\UseMapfile.cpp) segment .text

       7 0001:00000030       8 0001:00000048       9 0001:0000004f       11 0001:00000060
      12 0001:00000078      13 0001:0000007f      15 0001:00000086       16 0001:00000099
      17 0001:0000009c      19 0001:000000c0      20 0001:000000d8       21 0001:000000eb
      22 0001:000000ed

Line numbers for C:\Program Files2003\vc\VC98\LIB\LIBCD.lib(_freebuf.c) segment .text

      47 0001:0000c5d0      48 0001:0000c5d6      50 0001:0000c600       52 0001:0000c61d
      56 0001:0000c62e      66 0001:0000c640      67 0001:0000c653       69 0001:0000c65d
……
```

现在我们知道在地址 0x0040104c 处发生了崩溃，检查 UseMapfile.map 文件，可以看到

最佳装载地址是在 0x00400000，没有发生装载地址冲突。

（3）要找最好的匹配函数，可以通过崩溃地址在 Rva+Base 列找到函数

```
0001:00000030       ?Div@@YAHHH@Z                  00401030 f     UseMapfile.obj
 0001:00000060       ?Caller@@YAHXZ                 00401060 f     UseMapfile.obj
 0001:000000c0       _main                          004010c0 f     UseMapfile.obj
 0001:00000110       __chkesp                       00401110 f     LIBCD:chkesp.obj
```

（4）为了找到匹配的最好的源代码行号，我们需要进行如下的计算：

0x0040104c - 0x00400000 – 0x1000 = 0x4c

在行号段进行查找，我们得到是在第 8 行附近，经过确认，确实如此。

```
Line numbers for .\Debug\UseMapfile.obj(C:\0\UseMapfile\UseMapfile.cpp) segment .text

   7 0001:00000030       8 0001:00000048       9 0001:0000004f      11 0001:00000060
  12 0001:00000078      13 0001:0000007f      15 0001:00000086      16 0001:00000099
  17 0001:0000009c      19 0001:000000c0      20 0001:000000d8      21 0001:000000eb
  22 0001:000000ed
```

利用 MAP 文件可以进行任何崩溃的调试，包括常见的访问违法错误都可以进行处理，关键是要设定好编译选项，保证代码与 MAP 的版本对应。这样一旦知道了在源代码出错地地方，我们就可以进行进一步地深入分析，找到问题的真正原因。