Windows 异常处理机制

异常产生后，首先是要记录异常信息(异常的类型、异常发生的位置等)，然后要寻找异常的处理函数，我们称为异常的分发,最后找到异常处理函数并调用，我们称为异常处理。

异常记录->异常的分发->异常的处理

异常的分类

```
1、CPU产生的异常
a = 1;
b = 0;
c = a/b;
2、软件模拟产生的异常
throw 1;
```

CPU异常的产生

```
1、CPU指令检测到异常（例如：除0）
2、查询IDT表，执行中断处理函数
3、调用CommonDispatchException
    该函数构造了一个_EXCEPTION_RECORD结构体并赋值
    typedef struct _EXCEPTION_RECORD {
    DWORD                    ExceptionCode;     //异常代码        ----------》重点
    DWORD                    ExceptionFlags;    //异常状态
    struct _EXCEPTION_RECORD  *ExceptionRecord; //下一个异常
    PVOID                    ExceptionAddress;  //异常发生的地址   ----------》重点
    DWORD                    NumberParameters;  //附加参数个数
    ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
//附加参数指针
    } EXCEPTION_RECORD, *PEXCEPTION_RECORD;
4、KiDispatchException（分发异常：目的是找到异常的处理函数）
```

异常代码	值	来源
EXCEPTION_ACCESS_VIOLATION	0xC0000005L	非法访问（访问违例）
EXCEPTION_DATATYPE_MISALIGNMENT	0x80000002L	CPU 的对齐检查异常，#AC（17）
EXCEPTION_BREAKPOINT	0x80000003L	CPU 的断点异常，#BP（3）
EXCEPTION_SINGLE_STEP	0x80000004L	CPU 的调试异常，#DB（1）
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	0xC000008CL	CPU 的数组越界异常，#BR（5）
EXCEPTION_FLT_DIVIDE_BY_ZERO	0xC000008EL	CPU 的协处理器异常，#NM（7）
EXCEPTION_FLT_INEXACT_RESULT	0xC000008FL	CPU 的协处理器异常，#NM（7）
EXCEPTION_FLT_INVALID_OPERATION	0xC0000090L	CPU 的协处理器异常，#NM（7）
EXCEPTION_FLT_OVERFLOW	0xC0000091L	CPU 的协处理器异常，#NM（7）
EXCEPTION_FLT_STACK_CHECK	0xC0000092L	CPU 的协处理器异常，#NM（7）
EXCEPTION_FLT_UNDERFLOW	0xC0000093L	CPU 的协处理器异常，#NM（7）
EXCEPTION_INT_DIVIDE_BY_ZERO	0xC0000094L	CPU 的除零异常，#DE（0）
EXCEPTION_INT_OVERFLOW	0xC0000095L	CPU 的溢出异常，#OF（4）
EXCEPTION_PRIV_INSTRUCTION	0xC0000096L	CPU 的一般保护异常，#GP（13）
EXCEPTION_IN_PAGE_ERROR	0xC0000006L	CPU 的页错误异常，#PF（14）
EXCEPTION_ILLEGAL_INSTRUCTION	0xC000001DL	CPU 的无效指令异常，#UD（6）

软件模拟产生的异常

1、CxxThrowException
2、(KERNEL32.DLL)RaiseException
(DWORD dwExceptionCode, DWORD dwExceptionFlags, DWORD nNumberOfArguments,const ULONG_PTR*IpArguments)
    1）填充ExceptionRecord结构体

```
    typedef struct _EXCEPTION_RECORD {
    DWORD                    ExceptionCode;     //异常代码 软件模拟异常该值固定
    DWORD                    ExceptionFlags;    //异常状态
    struct _EXCEPTION_RECORD  *ExceptionRecord; //下一个异常
    PVOID                    ExceptionAddress;  //异常发生的地址 软件模拟异常该值不是真正异常的地址 存放的是RtlRaiseException函数的地址
    DWORD                    NumberParameters;  //附加参数个数
    ULONG_PTR                ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS]; //附加参数指针
    } EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

    2)调用NTDLL.DLL!RtlRaiseException()
3、NTDLL.DLL!RtlRaiseException()
4、NT!NtRaiseException
5、NT!KiRaiseException
    1）_EXCEPTION_RECORD.ExceptionCode最高位清零 用于区分CPU异常。
    2）调用KiDispatchException开始分发异常

用户层异常与内核层异常

异常可以发生在用户空间，也可以发生在内核空间。
无论是CPU异常还是模拟异常，是用户层异常还是内核异常，都要通过KiDispatchException函数进行分发。

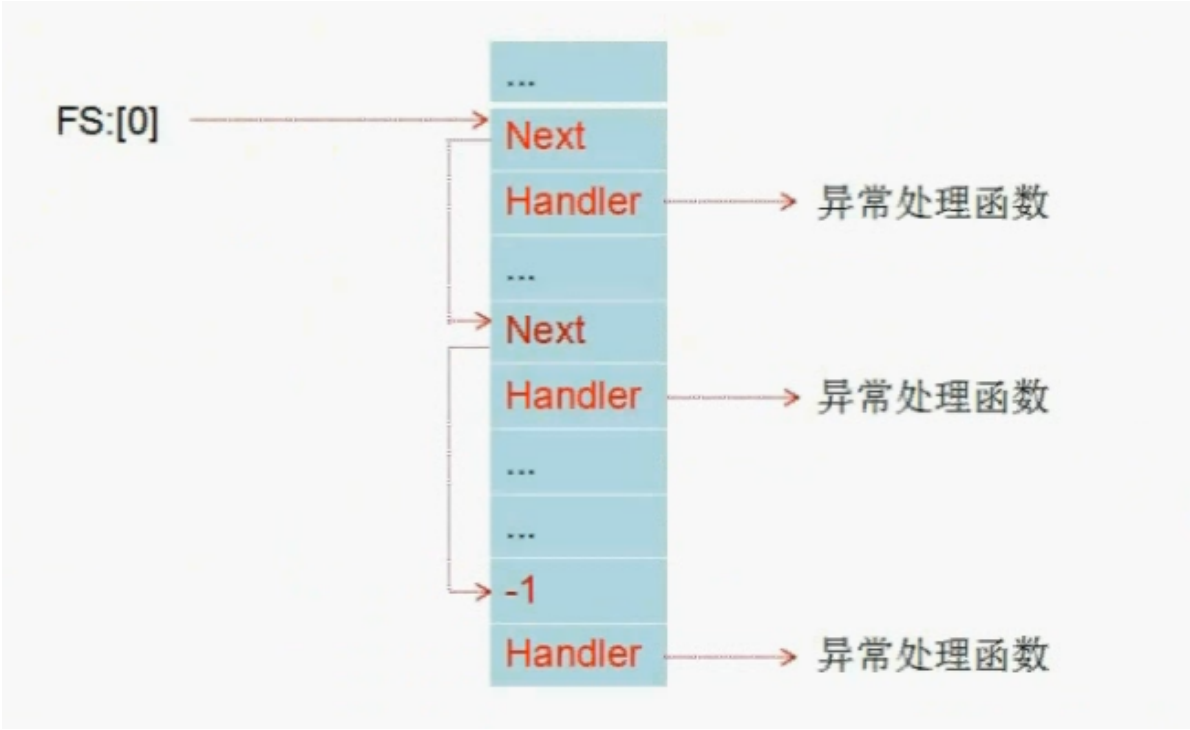
内核层异常的分发

内核异常处理流程

```
KiDispatchException(
    PEXCEPTION_RECORD ExceptionRecord,    //指向ExceptionRecord指针
    PKEXCEPTION_FRAME ExceptionFrame,     //对x86 为NULL
    PKTRAP_FRAME TrapFrame,               //陷阱框架指针
    KPROCESSOR_MODE PreviousMode,         //模式
    BOOLEAN FirstChance                   //是否为进行的第一次调用
)
1)_KeContextFromKframes 将 Trap_frame 备份到 context 为返回3环做准备
2)判断先前模式 0是内核调用 1是用户层调
3)是否是第一次机会
4)是否有内核调试器
5)如果没有或者内核调试器不处理
6)调用RtIDispatchException函数调用处理异常
    typedef struct _EXCEPTION_REGISTRATION_RECORD{
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
    }EXCEPTION_REGISTRATION_RECORD;
    RtlDispatchException的作用就是:
        遍历异常链表，调用异常处理函数，如果异常被正确处理了，该函数返回1.
        如果当前异常处理函数不能处理该异常，那么调用下一个，以此类推。
        如果到最好也没有人处理这个异常，返回0。
7)如果返回FALSE也就是0
```

8)再次判断是否有内核调试器有调用没有直接蓝屏

RtlDispatchException函数执行过程



用户层异常的分发

异常如果发生在内核层，处理起来比较简单，因为异常处理函数也在0环,不用切换堆栈，但是如果异常发生在3环，就意味着必须要切换堆栈，回到3环执行处理函数。
切换堆栈的处理方式与用户APC的执行过程几乎是一样的，惟一的区别就是执行用户APC时返回3环后执行的函数是KiUserApcDispatcher，而异常处理时返回3环后执行的函数是KiUserExceptionDispatcher.

用户异常处理流程

```
KiDispatchException(
    PEXCEPTION_RECORD ExceptionRecord,    //指向ExceptionRecord指针
    PKEXCEPTION_FRAME ExceptionFrame,     //对x86 为NULL
    PKTRAP_FRAME TrapFrame,               //陷阱框架指针
    KPROCESSOR_MODE PreviousMode,         //模式
    BOOLEAN FirstChance                   //是否为进行的第一次调用
)
1)_KeContextFromkKframes 将 Trap_frame 备份到 context 为返回3环做准备
2)判断先前模式 0是内核调用 1是用户层调
3)是否是第一次机会
4)是否有内核调试器
5)发送给3环调试
6)如果3环调试器没有处理这个异常修正EIP为KiUserExceptionDispatcher
7) KiDispatchException函数执行结束:CPU异常与模拟异常返回地点不同
    CPU异常:CPU检测到异常->查IDT执行处理函数->CommonDispatchException-
>KiDispatchException 通过IRETD返回3环
    模拟异常:CxxThrowException->RaiseException-> RtlRaiseException
NT!NtRaiseException->NT!KiRaiseException->KiDispatchException 通过系统调用返回3环
8)无论通过那种方式，但线程再次回到3环时，将执行KiU serExceptionDispatcher函数
```

VEH(向量化异常处理)

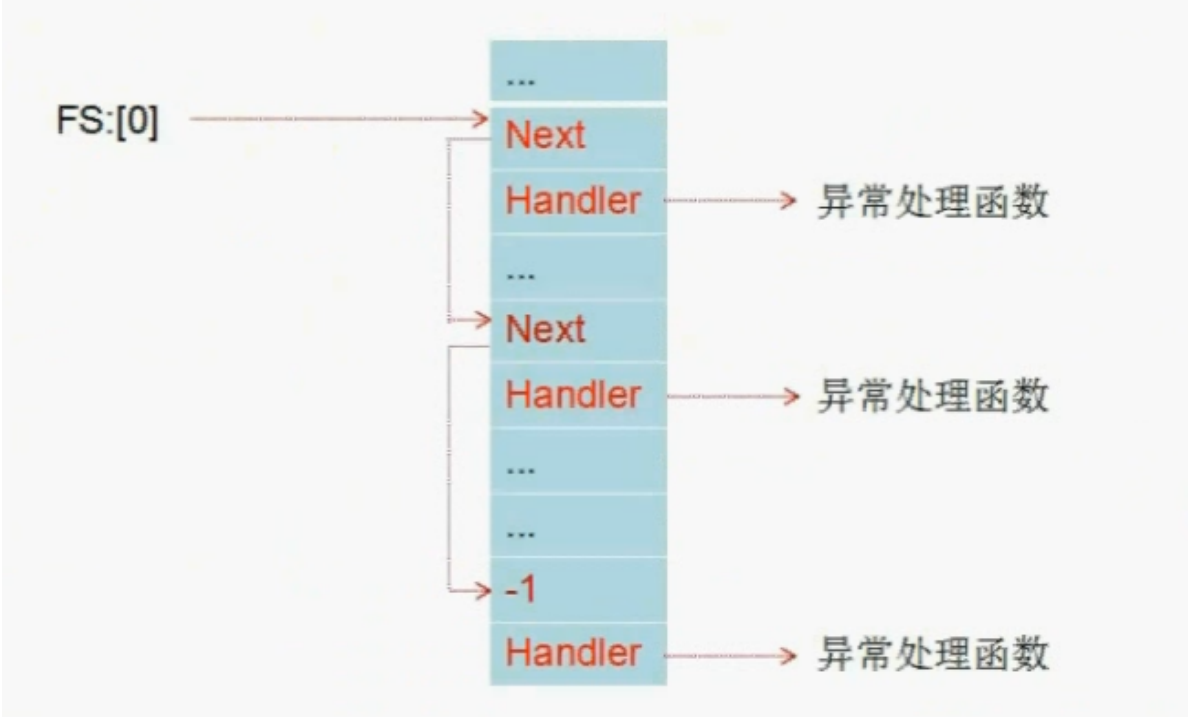
当用户异常产生后，内核函数KiDispatchException并不是像处理内核异常那样在0环直接进行处理，而是修正3环EIP为KiUserExceptionDispatcher函数后就结束了。
KiUserExceptionDispatcher
1)调用RtlDispatchException查找并执行异常处理函数
2)如果RtIDispatchException返回真,调用ZwContinue再次进入0环，但线程再次返回3环时,会从修正后的位置开始执行。
3)如果RtIDispatchException返回假,调用ZwRaiseException进行第二轮异常分发
RtlDispatchException函数分析
1)先查找VEH链表(全局链表)，如果有则调用
2)再查找SEH链表(局部链表，在堆栈中)，如果有则调用
注意:与内核调用时的区别!

具体处理流程

1)CPU捕获异常信息
2)通过KiDispatchException进行分发(EIP=KiUserExceptionDispatcher)
3)KiUserExceptionDispatcher调用RtIDispatchException
4)RtIDispatchException查找VEH处理函数链表并调用相关处理函数
5)代码返回到KiUserExceptionDispatcher
6)调用ZwContinue再次进入0环(ZwContinue调用NtContinue,主要作用就是恢复_TRAP _FRAME然后通过_KiServiceExit返回到3环).
7)线程再次返回3环后，从修正后的位置开始执行

SEH(结构化异常处理)

1)FS:[0]指向SEH链表的第一个成员
2)SEH的异常处理函数必须在当前线程的堆栈中
3)只有当VEH中的异常处理函数不存在或者不处理才会到SEH链表中查找



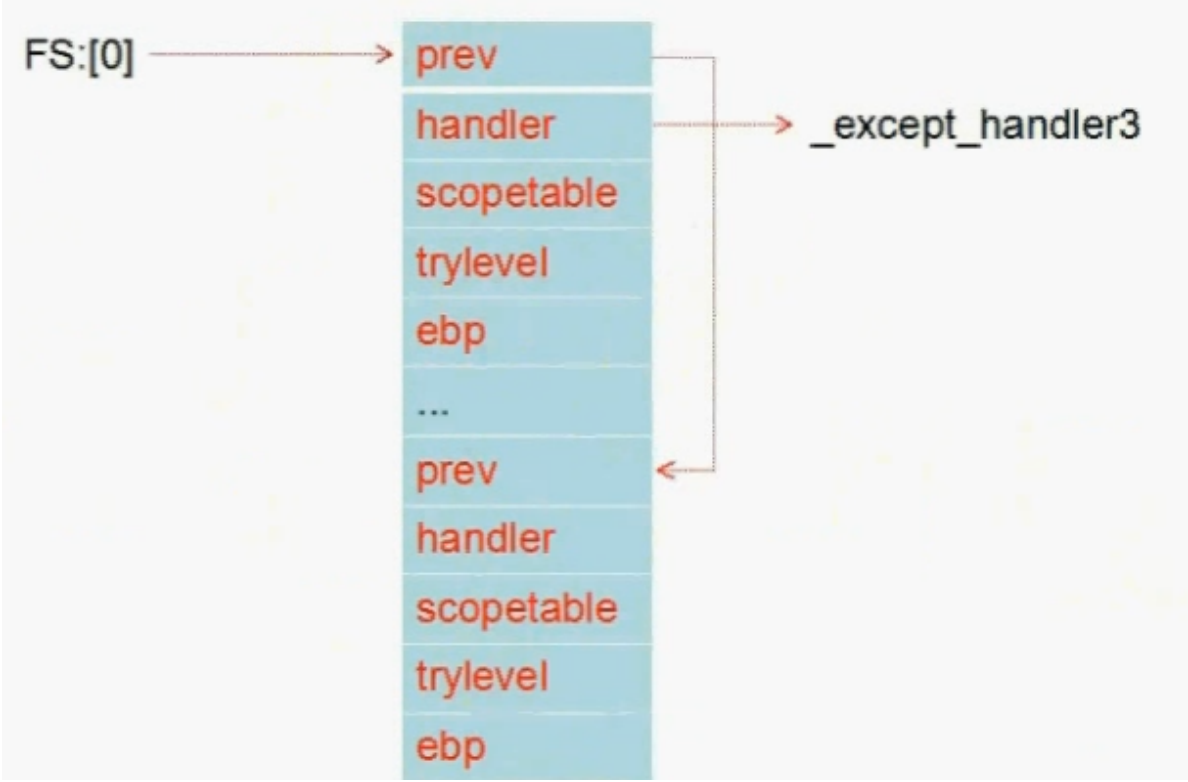
需要SEH处理异常时

```
1)挂入链表:
asm{
        mov eax,FS:[0]
        mov temp,eax
        lea ecx,myException
        mov FS:[0],ecx
    }
myException.prev =(MyException*)temp;
myException.handle = (DWORD)&MyEexception_handler;

EXCEPTION_DISPOSITION_cdecl MyEexception_handler(
    struct_EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext)
{
if( ExceptionRecord->ExceptionCode == 0xc0000094)                2)异常过滤
{
    ContextRecord->Eip = ContextRecord->Eip+2;                   3)异常处理
    //ContextRecord->Ecx =1;
    return ExceptionContinueExecution;
}
return ExceptionContinueSearch;

windows平台支持
try                          1)挂入链表
{
}
except(过滤表达式)            2)异常过滤
{
异常处理程序                  3)异常处理程序
}

过滤表达式
1)EXCEPTION_EXECUTE_HANDLER(1)          执行except代码
2)EXCEPTION_CONTINUE_SEARCH(0)          寻找下一个异常处理函数
3)EXCEPTION_CONTINUE_EXECUTION(-1)      返回出错位置重新执行
```

_try _except实现细节

```
_try _except自动挂入链表 _except handler3
每个使用_try _except的函数，不管其内部嵌套或反复使用多少_try _except,都只注册一遍，即只将一个_EXCEPTION_REGISTRATION_RECORD挂入当前线程的异常链表中
(对于递归函数，每一次调用都会创建一个_EXCEPTION_REGISTRATION_RECORD，并挂入线程的异常链表中)。
原本的结构体
    typedef struct _EXCEPTION_REGISTRATION_RECORD {
        struct _EXCEPTION_REGISTRATION_RECORD*Next;
        PEXCEPTION_ROUTINE Handler;
    }EXCEPTION_REGISTRATION_RECORD;
为了实现_try _except重复使用 将原本的结构体进行了拓展
    struct _EXCEPTION_REGISTRATION{
        struct _EXCEPTION_REGISTRATION*prev;
        void (*handler)(PEXCEPTION_RECORD,
PEXCEPTION_REGISTRATION,PCONTEXT,PEXCEPTION_RECORD);
```

```
        struct scopetable_entry *scopetable;
        int trylevel; //异常发生在哪个try里
        int _ebp; //保存栈底
    };
scopetable成员分析
struct scopetable_entry
{
    DWORD   previousTryLevel        //上一个try{}结构编号
    PDWRD   lpfnFilter              //过滤函数的起始地址
    PDWRD   lpfnHandler             //异常处理程序的地址
}

try{}except{}
try{
    try{}
    except{}
}
except{}
scopetable[0].previousTryLevel = -1;
scopetable[0].lpfnFilter = 过滤函数1;
scopetable[0].lpfnHandler = 异常处理函数1;

scopetable[1].previousTryLevel = -1;
scopetable[1].lpfnFilter = 过滤函数2;
scopetable[1].lpfnHandler = 异常处理函数2;

scopetable[2].previousTryLevel = 1; //它有上一个 被嵌套所以是1
scopetable[2].lpfnFilter = 过滤函数3;
scopetable[2].lpfnHandler = 异常处理函数3;
```



```
_except_handler3执行过程
1)CPU检测到异常查中断表执行处理函数CommonDispatchException KiDispatchException
KiUserExceptionDispatcher RtlDispatchException VEH SEH
2)执行_except_handler3函数
    <1>根据trylevel选择scopetable数组
    <2>调用scopetable数组中对应的IpfnFilter函数
        1)EXCEPTION_EXECUTE_HANDLER(1)执行except代码
        2) EXCEPTION_CONTINUE_SEARCH(0)寻找下一个
        3) EXCEPTION_CONTINUE_EXECUTION(-1)重新执行
```

_try _finally

```
try
{
//可能出错的代码
}
finally
{
//一定要执行的代码
}
无论try中有
    continue
    break
    return
finally一定会运行
如果 try中使用continue、break、return提前流出 会在try中调用局部展开函数调用finally代码
全局展开就是从该try处逐个局部展开
```

未处理异常/顶层处理函数

```
1、CPU检测到异常查IDT表执行中断处理程序 CommonDispatchException(CxxThrowException
RaiseException RtlRaiseException() NtRaiseException KiRaiseException) //记录异常
信息
2、KiDispatchException //在ring 0 调用该内核函数 该函数会对异常进行分发 看看是ring 3的异
常 还是ring 0的异常 如果是ring 0的异常，会将eip指向KiUserExceptionDispatcher这个ring 3
的函数
3、KiUserExceptionDispatcher //调用RtlDispatchException函数
4、RtlDispatchException //查找异常处理函数在哪里 首先去VEH查找
5、VEH //查找异常处理函数，没有就去SEH查
6、SEH //SEH和线程相关，存在于一个堆栈的链表
```

程序有最后一个防线

例如在程序在main函数执行之前是由kernel32.dll中mainCRTStartup()调用的，编译器都添加异常处理程序 相当于try except

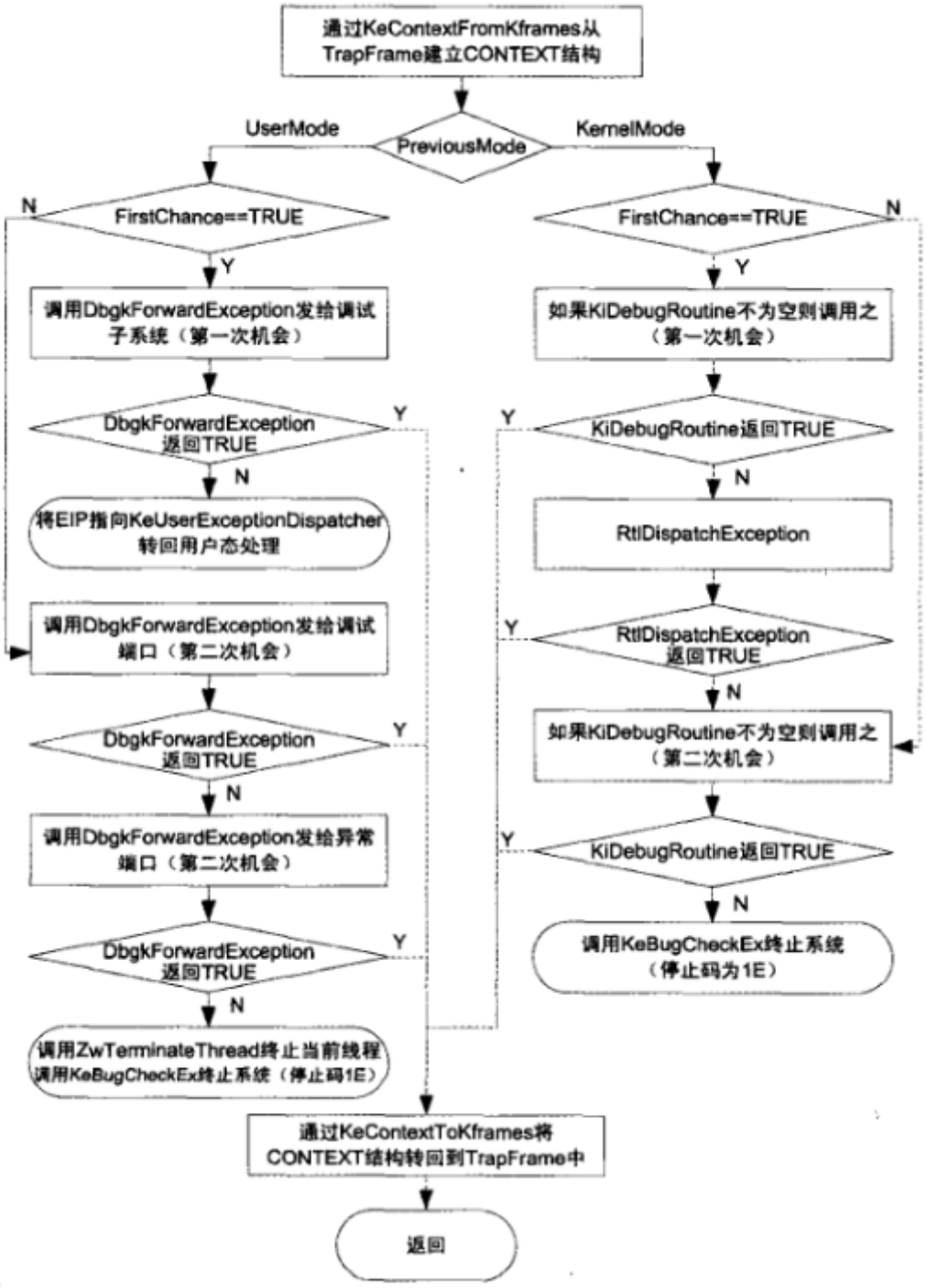
```
_try
{

}
_except(UnhandledExceptionFilter(GetExceptionInformation()){
    //终止线程
    //终止进程
}
```

只有程序被调试时，才会存在未处理异常

```
UnhandledExceptionFilter执行流程:
1)通过NtQueryInformationProcess查询当前进程是否正在被调试，如果是，返回
EXCEPTION_CONTINUE_SEARCH，此时会进入第二轮分发
2)如果没有被调试:
    查询是否通过SetUnhandledExceptionFilter注册处理函数如果有就调用
    如果没有通过SetUnhandledExceptionFilter注册处理函数弹出窗口让用户选择终止程序还是启动即时调试器
    如果用户没有启用即时调试器,那么该函数返回EXCEPTION_EXECUTE_HANDLER
```

KiUserExceptionDispatcher函数分析

```
1)调用RtlDispatchException查找并执行异常处理函数
2)如果RtIDispatchException返回真,调用ZwContinue再次进入0环，但线程再次返回3环时,会从修正后的位置开始执行。
3)如果RtIDispatchException返回假,调用ZwRaiseException进行第二轮异常分发
```


通过KeContextFromKframes从TrapFrame建立CONTEXT结构
PreviousMode
UserMode
KernelMode
FirstChance==TRUE
N
Y
调用DbgkForwardException发给调试子系统（第一次机会）
DbgkForwardException返回TRUE
将EIP指向KeUserExceptionDispatcher转回用户态处理
调用DbgkForwardException发给调试端口（第二次机会）
DbgkForwardException返回TRUE
调用DbgkForwardException发给异常端口（第二次机会）
DbgkForwardException返回TRUE
调用ZwTerminateThread终止当前线程
调用KeBugCheckEx终止系统（停止码1E）
如果KiDebugRoutine不为空则调用之（第一次机会）
KiDebugRoutine返回TRUE
RtlDispatchException
RtlDispatchException返回TRUE
如果KiDebugRoutine不为空则调用之（第二次机会）
KiDebugRoutine返回TRUE
调用KeBugCheckEx终止系统（停止码为1E）
通过KeContextToKframes将CONTEXT结构转回到TrapFrame中
返回