链滴

Win32 汇编学习 (3)：简单的窗口

作者：Akkuman

原文链接：https://ld246.com/article/1518196064845

来源网站：链滴

这次我们将写一个 Windows 程序，它会在桌面显示一个标准的窗口，以此根据代码来学习如何创建个简单的窗口。

<!--more-->

理论：

Windows 程序中，在写图形用户界面时需要调用大量的标准 Windows Gui 函数。其实这对用户和序员来说都有好处，对于用户，面对的是同一套标准的窗口，对这些窗口的操作都是一样的，所以使不同的应用程序时无须重新学习操作。对程序员来说，这些 Gui 源代码都是经过了微软的严格测试，时拿来就可以用的。当然至于具体地写程序对于程序员来说还是有难度的。为了创建基于窗口的应用序，必须严格遵守规范。做到这一点并不难，只要用模块化或面向对象的编程方法即可。

下面我就列出在桌面显示一个窗口的几个步骤：

1. 得到您应用程序的句柄(必需)；
2. 得到命令行参数(如果您想从命令行得到参数，可选)；
3. 注册窗口类(必需，除非您使用 Windows 预定义的窗口类，如 MessageBox 或 dialog box；
4. 产生窗口(必需)；
5. 在桌面显示窗口(必需，除非您不想立即显示它)；
6. 刷新窗口客户区；
7. 进入无限的获取窗口消息的循环；
8. 如果有消息到达，由负责该窗口的窗口回调函数处理；
9. 如果用户关闭窗口，进行退出处理。

相对于单用户的 DOS 下的编程来说，Windows 下的程序框架结构是相当复杂的。但是 Windows 和 DOS 在系统架构上是截然不同的。Windows 是一个多任务的操作系统，故系统中同时有多个应用程彼此协同运行。这就要求 Windows 程序员必须严格遵守编程规范，并养成良好的编程风格。

内容：

下面是我们简单的窗口程序的源代码。在进入复杂的代码前，指出几点要点：

● 您应当把程序中要用到的所有常量和结构体的声明放到一个头文件中，并且在源程序的开始处包含个头文件。这么做将会节省您大量的时间，也免得一次又一次的敲键盘。目前，我所使用的是masm32 com提供的。您也可以定义您自己的常量和结构体，但最好把它们放到独立的头文件中

● 用 includelib 指令，包含您的程序要引用的库文件，譬如：若您的程序要调用 "MessageBox"，就应当在源文件中加入如下一行： includelib user32.lib 这条语句告诉 MASM 您的程序将要用到一引入库。如果您不止引用一个库，只要简单地加入 includelib 语句，不要担心链接器如何处理这么多库，只要在链接时用链接开关 /LIBPATH 指明库所在的路径即可。

● 在其它地方运用头文件中定义函数原型，常数和结构体时，要严格保持和头文件中的定义一致，包大小写。在查询函数定义时，这将节约您大量的时间；

● 在编译，链接时用makefile文件，免去重复敲键。

```
.386
.model flat,stdcall
option casemap:none
```

```
include windows.inc
include user32.inc
includelib user32.lib           ; calls to functions in user32.lib and kernel32.lib
include kernel32.inc
includelib kernel32.lib

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.DATA                   ; initialized data
ClassName db "SimpleWinClass",0       ; the name of our window class
AppName db "Our First Window",0       ; the name of our window

.DATA?              ; Uninitialized data
hInstance HINSTANCE ?       ; Instance handle of our program
CommandLine LPSTR ?
.CODE               ; Here begins our code
start:
invoke GetModuleHandle, NULL            ; get the instance handle of our program.
                                        ; Under Win32, hmodule==hinstance mov hInsta
ce,eax
mov hInstance,eax
invoke GetCommandLine                   ; get the command line. You don't have to call this fu
ction IF
                                        ; your program doesn't process the command line

mov CommandLine,eax
invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT        ; call the main funct
on
invoke ExitProcess, eax                 ; quit our program. The exit code is returned in eax fr
m WinMain.

WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX                                 ; create local variables on stack
    LOCAL msg:MSG
    LOCAL hwnd:HWND

    mov   wc.cbSize,SIZEOF WNDCLASSEX                   ; fill values in members of wc
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,NULL
    push  hInstance
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_WINDOW+1
    mov   wc.lpszMenuName,NULL
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc                     ; register our window class
    invoke CreateWindowEx,NULL,\
               ADDR ClassName,\
```

```
                ADDR AppName,\
                WS_OVERLAPPEDWINDOW,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                CW_USEDEFAULT,\
                NULL,\
                NULL,\
                hInst,\
                NULL
    mov   hwnd,eax
    invoke ShowWindow, hwnd,CmdShow               ; display our window on desktop
    invoke UpdateWindow, hwnd                       ; refresh the client area

    .WHILE TRUE                                     ; Enter message loop
                invoke GetMessage, ADDR msg,NULL,0,0
                .BREAK .IF (!eax)
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
    .ENDW
    mov     eax,msg.wParam                          ; return exit code in eax
    ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY                    ; if the user closes our window
        invoke PostQuitMessage,NULL         ; quit our application
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam    ; Default message processing
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp

end start
```

分析：

看到一个简单的 Windows 程序有这么多行，您是不是有点想死? 但是您必须要知道的是上面的大多代码都是模板而已，模板的意思即是指这些代码对差不多所有标准 Windows 程序来说都是相同的。写 Windows 程序时您可以把这些代码拷来拷去，当然把这些重复的代码写到一个库中也挺好。其实正要写的代码集中在 WinMain 中。这和一些 C 编译器一样，无须要关心其它杂务，集中精力于 Win ain 函数。**唯一不同的是 C 编译器要求您的源代码有必须有一个函数叫 WinMain。否则 C 无法知道哪个函数和有关的前后代码链接。相对C，汇编语言提供了较大的灵活性，它不强行要求一个叫 Win ain 的函数。**

做好心理准备，下面我们开始分析代码。

```
.386
.model flat，stdcall
option casemap：none

WinMain proto：DWORD，：DWORD，：DWORD，：DWORD
```

```
include windows.inc
include user32.inc
include kernel32.inc
includelib user32.lib
includelib kernel32.lib
```

您可以把前三行看成是"必须"的.

.386告诉MASN我们要用80386指令集。

. model flat，stdcall告诉MASM 我们用的内存寻址模式，此处也可以加入stdcall告诉MASM我们所的参数传递约定。

接下来是函数 WinMain 的原型声明，因为我们稍后要用到该函数，故必须先声明。我们必须包含 wi dow.inc 文件，因为其中包含大量要用到的常量和结构的定义，该文件是一个文本文件，您可以用任 文本编辑器打开并且查看它

我们的程序调用 user32.dll (譬如：CreateWindowEx， RegisterWindowClassEx) 和 kernel32.dll (xitProcess)中的函数，所以必须链接这两个库。接下来我如果问：您需要把什么库链入您的程序呢？ 案是：先查到您要调用的函数在什么库中，然后包含进来。譬如：若您要调用的函数在 gdi32.dll 中 您就要包含gdi32.inc头文件。和 MASM 相比，TASM 则要简单得多，您只要引入一个库，即：impo t32.lib。<但 Tasm5 麻烦的是 windows.inc 非常的不全面，而且如果在 Windows.inc 中包含全部的 API 定义会内存不够，所以每次你得把用到的 API 定义拷贝出来>

```
.DATA

ClassName db "SimpleWinClass"，0
AppName db "Our First Window"，0

.DATA?

hInstance HINSTANCE ?
CommandLine LPSTR ?
```

接下来是DATA"分段"。 在 .DATA 中我们定义了两个以 NULL 结尾的字符串 (ASCIIZ)：其中 ClassN me 是 Windows 类名，AppName 是我们窗口的名字。这两个变量都是初始化了的。未进行初始化 两个变量放在 .DATA? "分段"中，其中 hInstance 代表应用程序的句柄，CommandLine 保存从命令 传入的参数。HINSTACE 和 LPSTR 是两个数据类型名，它们在头文件中定义，可以看做是 DWORD 别名，之所以要这么重新定仅是为了易记。您可以查看 windows.inc 文件，在 .DATA? 中的变量都 未经初始化的，这也就是说在程序刚启动时它们的值是什么无关紧要，只不过占有了一块内存，以后 以再利用而已。

```
.CODE
start:
invoke GetModuleHandle， NULL
mov hInstance，eax
invoke GetCommandLine
mov CommandLine，eax
invoke WinMain， hInstance，NULL，CommandLine， SW_SHOWDEFAULT
invoke ExitProcess，eax
.....
end start
```

.CODE "分段"包含了您应用程序的所有代码，这些代码必须都在 .code 和 end 之间。至于 label 的

名只要遵从 Windows 规范而且保证唯一则具体叫什么倒是无所谓。我们程序的第一条语句是调用 GetModuleHandle 去查找我们应用程序的句柄。在Win32下，应用程序的句柄和模块的句柄是一样的您可以把实例句柄看成是您的应用程序的 ID 号。我们在调用几个函数是都把它作为参数来进行传递所以在一开始便得到并保存它就可以省许多的事。

特别注意：WIN32下的实例句柄实际上是您应用程序在内存中的线性地址。

WIN32 中函数的函数如果有返回值，那它是通过 eax 寄存器来传递的。其他的值可以通过传递进的参数地址进行返回。一个 WIN32 函数被调用时总会保存好段寄存器和 ebx，edi，esi和ebp 寄器，而 ecx和edx 中的值总是不定的，不能在返回时应用。特别注意：从 Windows API 函数中返回，eax，ecx，edx 中的值和调用前不一定相同。当函数返回时，返回值放在eax中。如果您应用程序的函数提供给 Windows 调用时，也必须遵守这一点，即在函数入口处保存段寄存器和 ebx，esp，es，edi 的值并在函数返回时恢复。如果不这样一来的话，您的应用程序很快会崩溃。从您的程序中提给 Windows 调用的函数大体上有两种：Windows 窗口过程和 Callback 函数。

如果您的应用程序不处理命令行那么就无须调用 GetCommandLine，这里只是告诉您如果要调用应怎么做。

下面则是调用WinMain了。该函数共有4个参数：应用程序的实例句柄，该应用程序的前一实例句柄命令行参数串指针和窗口如何显示。Win32 没有前一实例句柄的概念，所以第二个参数总为0。之所保留它是为了和 Win16 兼容的考虑，在 Win16下，如果 hPrevInst 是 NULL，则该函数是第一次运。特别注意：您不用必须声明一个名为 WinMain 函数，事实上在这方面您可以完全作主，您甚至无有一个和 WinMain 等同的函数。您只要把 WinMain 中的代码拷到GetCommandLine 之后，其所现的功能完全相同。在 WinMain 返回时，把返回码放到 eax 中。然后在应用程序结束时通过 ExitProess 函数把该返回码传递给 Windows 。

```
WinMain proc Inst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
```

上面是WinMain的定义。注意跟在 proc 指令后的parameter：type形式的参数，它们是由调用者传 WinMain 的，我们引用是直接用参数名即可。至于压栈和退栈时的平衡堆栈工作由 MASM 在编译加入相关的前序和后序汇编指令来进行。 LOCAL wc：WNDCLASSEX LOCAL msg：MSG LOCAL hnd：HWND LOCAL 伪指令为局部变量在栈中分配内存空间，所有的 LOCAL 指令必须紧跟在 PROC 之后。LOCAL 后跟声明的变量，其形式是 变量名:变量类型。譬如 LOCAL wc：WNDCLASSEX 即是诉 MASM 为名字叫 wc 的局部边量在栈中分配长度为 WNDCLASSEX 结构体长度的内存空间，然后们在用该局部变量是无须考虑堆栈的问题，考虑到 DOS 下的汇编，这不能不说是一种恩赐。不过这要求这样声明的局部变量在函数结束时释放栈空间，(也即不能在函数体外被引用)，另一个缺点是您不能初始化您的局部变量，不得不在稍后另外再对其赋值。

```
mov wc.cbSize，SIZEOF WNDCLASSEX
mov wc.style， CS_HREDRAW or CS_VREDRAW
mov wc.lpfnWndProc， OFFSET WndProc
mov wc.cbClsExtra，NULL
mov wc.cbWndExtra，NULL
push hInstance
pop wc.hInstance
mov wc.hbrBackground，COLOR_WINDOW+1
mov wc.lpszMenuName，NULL
mov wc.lpszClassName，OFFSET ClassName
invoke LoadIcon，NULL，IDI_APPLICATION
mov wc.hIcon，eax
mov wc.hIconSm，eax
invoke LoadCursor，NULL，IDC_ARROW
mov wc.hCursor，eax invoke
RegisterClassEx， addr w
```

上面几行从概念上说确实是非常地简单。只要几行指令就可以实现。其中的主要概念就是窗口类（window class），一个窗口类就是一个有关窗口的规范，这个规范定义了几个主要的窗口的元素，如：图、光标、背景色、和负责处理该窗口的函数。您产生一个窗口时就必须要有这样的一个窗口类。如果要产生不止一个同种类型的窗口时，最好的方法就是把这个窗口类存储起来，这种方法可以节约许多内存空间。也许今天您不会太感觉到，可是想想以前 PC 大多数只有 1M 内存时，这么做是非常有必的。如果您要定义自己的创建窗口类就必须：在一个 WINDCLASS 或 WINDOWCLASSEXE 结构体指明您窗口的组成元素，然后调用 RegisterClass 或 RegisterClassEx ，再根据该窗口类产生窗口。不同特色的窗口必须定义不同的窗口类。 WINDOWS有几个预定义的窗口类，譬如：按钮、编辑框。要产生该种风格的窗口无须预先再定义窗口类了，只要包预定义类的类名作为参数调用 CreateWindowEx 即可。

WNDCLASSEX 中最重要的成员莫过于lpfnWndProc了。前缀 lpfn 表示该成员是一个指向函数的长针。在 Win32中由于内存模式是 FLAT 型，所以没有 near 或 far 的区别。每一个窗口类必须有一个口过程，当 Windows 把属于特定窗口的消息发送给该窗口时，该窗口的窗口类负责处理所有的消息如键盘消息或鼠标消息。由于窗口过程差不多智能地处理了所有的窗口消息循环，所以您只要在其中入消息处理过程即可。下面我将要讲解 WNDCLASSEX 的每一个成员

```
WNDCLASSEX STRUCT DWORD
  cbSize          DWORD     ?
  style           DWORD     ?
  lpfnWndProc     DWORD     ?
  cbClsExtra      DWORD     ?
  cbWndExtra      DWORD     ?
  hInstance       DWORD     ?
  hIcon           DWORD     ?
  hCursor         DWORD     ?
  hbrBackground   DWORD     ?
  lpszMenuName    DWORD     ?
  lpszClassName   DWORD     ?
  hIconSm         DWORD     ?
WNDCLASSEX ENDS
```

- cbSize：WNDCLASSEX 的大小。我们可以用sizeof（WNDCLASSEX）来获得准确的值。
- style：从这个窗口类派生的窗口具有的风格。您可以用“or”操作符来把几个风格或到一起。
- lpfnWndProc：窗口处理函数的指针。
- cbClsExtra：指定紧跟在窗口类结构后的附加字节数。
- cbWndExtra：指定紧跟在窗口事例后的附加字节数。如果一个应用程序在资源中用CLASS伪指令册一个对话框类时，则必须把这个成员设成DLGWINDOWEXTRA。
- hInstance：本模块的事例句柄。
- hIcon：图标的句柄。
- hCursor：光标的句柄。
- hbrBackground：背景画刷的句柄。
- lpszMenuName：指向菜单的指针。
- lpszClassName：指向类名称的指针。
- hIconSm：和窗口类关联的小图标。如果该值为NULL。则把hCursor中的图标转换成大小合适的图标。

```
invoke CreateWindowEx,  NULL, \
ADDR ClassName, \
```

```
ADDR AppName，\
WS_OVERLAPPEDWINDOW，\
CW_USEDEFAULT，\
CW_USEDEFAULT，\
CW_USEDEFAULT，\
CW_USEDEFAULT，\
NULL，\
NULL，\
hInst，\
NULL
```

注册窗口类后，我们将调用CreateWindowEx来产生实际的窗口。请注意该函数有12个参数。

```
CreateWindowExA proto dwExStyle：DWORD，\
lpClassName：DWORD，\
lpWindowName：DWORD，\
dwStyle：DWORD，\
X：DWORD，\
Y：DWORD，\
nWidth：DWORD，\
nHeight：DWORD，\
hWndParent：DWORD ，\
hMenu：DWORD，\
hInstance：DWORD，\
lpParam：DWORD
```

我们来仔细看一看这些的参数：

● dwExStyle：附加的窗口风格。相对于旧的CreateWindow这是一个新的参数。在9X/NT中您可以用新的窗口风格。您可以在Style中指定一般的窗口风格，但是一些特殊的窗口风格，如顶层窗口则必在此参数中指定。如果您不想指定任何特别的风格，则把此参数设为NULL。

● lpClassName： （必须）。ASCIIZ形式的窗口类名称的地址。可以是您自定义的类，也可以是预定的类名。像上面所说，每一个应用程序必须有一个窗口类。

● lpWindowName：ASCIIZ形式的窗口名称的地址。该名称会显示在标题条上。如果该参数空白，标题条上什么都没有。

● dwStyle：窗口的风格。在此您可以指定窗口的外观。可以指定该参数为零，但那样该窗口就没有统菜单，也没有最大化和最小化按钮，也没有关闭按钮，那样您不得不按Alt+F4 来关闭它。最为普的窗口类风格是 WS_OVERLAPPEDWINDOW。 一种窗口风格是一种按位的掩码，这样您可以用or您希望的窗口风格或起来。像 WS_OVERLAPPEDWINDOW 就是由几种最为普遍的风格or起来的。

● X，Y： 指定窗口左上角的以像素为单位的屏幕坐标位置。缺省地可指定为 CW_USEDEFAULT，这 Windows 会自动为窗口指定最合适的位置。

● nWidth，nHeight： 以像素为单位的窗口大小。缺省地可指定为 CW_USEDEFAULT，这样 Windows 会自动为窗口指定最合适的大小。

● hWndParent： 父窗口的句柄（如果有的话）。这个参数告诉 Windows 这是一个子窗口和他的父口是谁。这和 MDI（多文档结构）不同，此处的子窗口并不会局限在父窗口的客户区内。他只是用来诉 Windows 各个窗口之间的父子关系，以便在父窗口销毁是一同把其子窗口销毁。在我们的例子程中因为只有一个窗口，故把该参数设为 NULL。

● hMenu： WINDOWS菜单的句柄。如果只用系统菜单则指定该参数为NULL。回头看一看WNDCLSSEX 结构中的 lpszMenuName 参数，它也指定一个菜单，这是一个缺省菜单，任何从该窗口类派的窗口若想用其他的菜单需在该参数中重新指定。其实该参数有双重意义：一方面若这是一个自定义口时该参数代表菜单句柄，另一方面，若这是一个预定义窗口时，该参数代表是该窗口的 ID 号。Win

ows 是根据lpClassName 参数来区分是自定义窗口还是预定义窗口的。

- hInstance： 产生该窗口的应用程序的实例句柄。
- lpParam： （可选）指向欲传给窗口的结构体数据类型参数的指针。如在MDI中在产生窗口时传递LIENTCREATESTRUCT 结构的参数。一般情况下，该值总为零，这表示没有参数传递给窗口。可以过GetWindowLong 函数检索该值。

```
mov hwnd，eax
invoke ShowWindow， hwnd，CmdShow
invoke UpdateWindow， hwnd
```

调用CreateWindowEx成功后，窗口句柄在eax中。我们必须保存该值以备后用。我们刚刚产生的窗不会自动显示，所以必须调用 ShowWindow 来按照我们希望的方式来显示该窗口。接下来调用 Upd teWindow 来更新客户区。

```
.WHILE TRUE
invoke GetMessage， ADDR msg，NULL，0，0
.BREAK .IF (!eax)
invoke TranslateMessage， ADDR msg
invoke DispatchMessage， ADDR msg
.ENDW
```

这时候我们的窗口已显示在屏幕上了。但是它还不能从外界接收消息。所以我们必须给它提供相关的息。我们是通过一个**消息循环**来完成该项工作的。每一个模块仅有一个消息循环，我们不断地调用 Ge Message 从 Windows 中获得消息。GetMessage 传递一个 MSG 结构体给 Windows ，然后 Wind ws 在该函数中填充有关的消息，一直到 Windows 找到并填充好消息后 GetMessage 才会返回。在段时间内系统控制权可能会转移给其他的应用程序。这样就构成了Windows 下的多任务结构。如果 G tMessage 接收到 WM_QUIT 消息后就会返回 FALSE，使循环结束并退出应用程序。TranslateMess ge 函数是一个是实用函数，它从键盘接受原始按键消息，然后解释成 WM_CHAR，再把 WM_CHAR 放入消息队列，由于经过解释后的消息中含有按键的 ASCII 码，这比原始的扫描码好理解得多。**如果的应用程序不处理按键消息的话，可以不调用该函数。**DispatchMessage 会把消息发送给负责该窗过程的函数。

```
mov eax，msg.wParam
ret
WinMain endp
```

如果消息循环结束了，退出码存放在 MSG 中的 wParam中，您可以通过把它放到 eax 寄存器中传给 Windows，目前 Windows 没有利用到这个结束码，但我们最好还是遵从 Windows 规范已防意外。

```
WndProc proc hWnd：HWND， uMsg：UINT， wParam：WPARAM， lParam：LPARAM
```

是我们的窗口处理函数。您可以随便给该函数命名。其中第一个参数 hWnd 是接收消息的窗口的句 。uMsg 是接收的消息。注意 uMsg 不是一个 MSG 结构，其实上只是一个 DWORD 类型数。Wind ws 定义了成百上千个消息，大多数您的应用程序不会处理到。当有该窗口的消息发生时，Windows 发送一个相关消息给该窗口。其窗口过程处理函数会智能的处理这些消息。wParam 和 lParam 只是加参数，以方便传递更多的和该消息有关的数据。

```
.IF uMsg==WM_DESTROY
invoke PostQuitMessage，NULL
.ELSE
invoke DefWindowProc，hWnd，uMsg，wParam，lParam
ret
.ENDIF
xor eax，eax
```

```
ret
WndProc endp
```

上面可以说是关键部分。这也是我们写 Windows 程序时需要改写的主要部分。此处您的程序检查 Windows 传递过来的消息，如果是我们感兴趣的消息则加以处理，处理完后，在 eax 寄存器中传递 0 否则必须调用 DefWindowProc，把该窗口过程接收到的参数传递给缺省的窗口处理函数。所有消中您必须处理的是 WM_DESTROY，当您的应用程序结束时 Windows 把这个消息传递进来，您的应用程序接收到该消息时它已经在屏幕上消失了，这仅是通知您的应用程序窗口已销毁，您必须己准备返回 Windows 。在此消息中您可以做一些清理工作，但无法阻止退出应用程序。如果您要在口销毁前做一些额外工作，可以处理 WM_CLOSE 消息。在处理完清理工作后，您必须调用 PostQuiMessage，该函数会把 WM_QUIT 消息传回您的应用程序，而该消息会使得 GetMessage 返回，并 eax 寄存器中放入 0，然后会结束消息循环并退回 WINDOWS。您可以在您的程序中调用 DestroyWndow 函数，它会发送一个 WM_DESTROY 消息给您自己的应用程序，从而迫使它退出。