# 环境变量1，参考：[Windows Api学习指南学习笔记059-环境变量1](Windows%20Api学习指南学习笔记059-环境变量1.docx)

# 环境变量2，参考：[Windows Api学习指南学习笔记060-环境变量2](Windows%20Api学习指南学习笔记060-环境变量2.docx)

# 环境变量学习3.

# 学习大纲

|  |
| --- |
|  |
|  |

## STARTUPINFO结构体的用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指定创建时进程的窗口工作站、桌面、标准句柄和main窗口的外观。 语法 C++  typedef struct \_STARTUPINFOA {  DWORD cb;  LPSTR lpReserved;  LPSTR lpDesktop;  LPSTR lpTitle;  DWORD dwX;  DWORD dwY;  DWORD dwXSize;  DWORD dwYSize;  DWORD dwXCountChars;  DWORD dwYCountChars;  DWORD dwFillAttribute;  DWORD dwFlags;  WORD wShowWindow;  WORD cbReserved2;  LPBYTE lpReserved2;  HANDLE hStdInput;  HANDLE hStdOutput;  HANDLE hStdError;  } STARTUPINFOA, \*LPSTARTUPINFOA; 成员 cb  结构大小（以字节为单位）。  lpReserved  保留;必须为 NULL。  lpDesktop  桌面的名称，或此过程的桌面和窗口工作站的名称。 字符串中的反斜杠指示字符串同时包含桌面和窗口工作站名称。  有关详细信息，请参阅 [与桌面的线程连接](https://learn.microsoft.com/zh-cn/windows/desktop/winstation/thread-connection-to-a-desktop)。  lpTitle  对于控制台进程，这是在创建新的控制台窗口时显示在标题栏中的标题。 如果为 NULL，则可执行文件的名称将改为用作窗口标题。 对于不创建新控制台窗口的 GUI 或控制台进程，此参数必须为 NULL。  dwX  如果 **dwFlags** 指定STARTF\_USEPOSITION，则此成员是创建新窗口时窗口左上角的 x 偏移量（以像素为单位）。 否则，将忽略此成员。  偏移量从屏幕左上角开始。 对于 GUI 进程，如果 [createWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-createwindowa) 的 *x* 参数CW\_USEDEFAULT，则新进程首次调用 **CreateWindow** 以创建重叠窗口时使用指定位置。  dwY  如果 **dwFlags** 指定STARTF\_USEPOSITION，则此成员是创建新窗口时窗口左上角的 y 偏移量（以像素为单位）。 否则，将忽略此成员。  偏移量从屏幕左上角开始。 对于 GUI 进程，如果 [createWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-createwindowa) 的 *y* 参数CW\_USEDEFAULT，则新进程首次调用 **CreateWindow** 以创建重叠窗口时使用指定位置。  dwXSize  如果 **dwFlags** 指定STARTF\_USESIZE，则此成员是创建新窗口时窗口的宽度（以像素为单位）。 否则，将忽略此成员。  对于 GUI 进程，如果 [createWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-createwindowa) 的 *nWidth* 参数CW\_USEDEFAULT，则仅在新进程首次调用 **CreateWindow** 来创建重叠窗口时，才使用此方法。  dwYSize  如果 **dwFlags** 指定STARTF\_USESIZE，则此成员是创建新窗口时窗口的高度（以像素为单位）。 否则，将忽略此成员。  对于 GUI 进程，如果 [createWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-createwindowa) 的 *nHeight* 参数CW\_USEDEFAULT，则仅在新进程首次调用 **CreateWindow** 以创建重叠窗口时使用。  dwXCountChars  如果 **dwFlags** 指定STARTF\_USECOUNTCHARS，如果在控制台进程中创建新的控制台窗口，则此成员以字符列为单位指定屏幕缓冲区宽度。 否则，将忽略此成员。  dwYCountChars  如果 **dwFlags** 指定STARTF\_USECOUNTCHARS，如果在控制台进程中创建新的控制台窗口，则此成员以字符行为单位指定屏幕缓冲区高度。 否则，将忽略此成员。  dwFillAttribute  如果 **dwFlags** 指定STARTF\_USEFILLATTRIBUTE，则如果在控制台应用程序中创建新的控制台窗口，则此成员为初始文本和背景色。 否则，将忽略此成员。  此值可以是以下值的任意组合：FOREGROUND\_BLUE、FOREGROUND\_GREEN、FOREGROUND\_RED、FOREGROUND\_INTENSITY、BACKGROUND\_BLUE、BACKGROUND\_GREEN、BACKGROUND\_RED和BACKGROUND\_INTENSITY。 例如，以下值组合在白色背景上生成红色文本：  FOREGROUND\_RED| BACKGROUND\_RED| BACKGROUND\_GREEN| BACKGROUND\_BLUE  dwFlags  确定进程创建窗口时是否使用某些 **STARTUPINFO** 成员的位域。 此成员可以是以下一个或多个值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **STARTF\_FORCEONFEEDBACK**  0x00000040 | 指示在调用 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 后光标处于反馈模式两秒钟。 显示“在后台工作”光标 (查看鼠标控制面板实用工具) 中的“指针”选项卡。  如果在这两秒内进程进行第一次 GUI 调用，则系统会再给进程 5 秒。 如果在这五秒内进程显示一个窗口，则系统再给进程 5 秒以完成绘制窗口。  系统在第一次调用 [GetMessage](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/fax/-mfax-faxaccountincomingarchive-getmessage-vb) 后关闭反馈光标，无论进程是否正在绘制。 | | **STARTF\_FORCEOFFFEEDBACK**  0x00000080 | 指示在进程启动时强制关闭反馈光标。 将显示“普通选择”光标。 | | **STARTF\_PREVENTPINNING**  0x00002000 | 指示进程创建的任何窗口都不能固定在任务栏上。  此标志必须与 STARTF\_TITLEISAPPID 组合使用。 | | **STARTF\_RUNFULLSCREEN**  0x00000020 | 指示进程应在全屏模式下运行，而不是在窗口模式下运行。  此标志仅对 x86 计算机上运行的控制台应用程序有效。 | | **STARTF\_TITLEISAPPID**  0x00001000 | **lpTitle** 成员包含 AppUserModelID。 此标识符控制任务栏和**“开始”**菜单显示应用程序的方式，并使它能够与正确的快捷方式和跳转Lists相关联。 通常，应用程序将使用 [SetCurrentProcessExplicitAppUserModelID](https://learn.microsoft.com/zh-cn/windows/desktop/api/shobjidl_core/nf-shobjidl_core-getcurrentprocessexplicitappusermodelid) 和 **GetCurrentProcessExplicitAppUserModelID** 函数，而不是设置此标志。 有关详细信息，请参阅 [应用程序用户模型 ID](https://learn.microsoft.com/zh-cn/windows/desktop/shell/appids)。  如果使用STARTF\_PREVENTPINNING，则应用程序窗口不能固定在任务栏上。 应用程序使用与 AppUserModelID 相关的任何窗口属性仅覆盖该窗口的此设置。  此标志不能与STARTF\_TITLEISLINKNAME一起使用。 | | **STARTF\_TITLEISLINKNAME**  0x00000800 | **lpTitle** 成员包含用户为启动此过程而调用的快捷方式文件 (.lnk) 的路径。 这通常是在调用指向启动的应用程序的.lnk文件时由 shell 设置的。 大多数应用程序不需要设置此值。  此标志不能与STARTF\_TITLEISAPPID一起使用。 | | **STARTF\_UNTRUSTEDSOURCE**  0x00008000 | 命令行来自不受信任的源。 有关详细信息，请参阅“备注”。 | |  |  | | **STARTF\_USECOUNTCHARS**  0x00000008 | **dwXCountChars** 和 **dwYCountChars** 成员包含其他信息。 | | **STARTF\_USEFILLATTRIBUTE**  0x00000010 | **dwFillAttribute** 成员包含其他信息。 | | **STARTF\_USEHOTKEY**  0x00000200 | **hStdInput** 成员包含其他信息。  此标志不能与 **STARTF\_USESTDHANDLES**一起使用。 | | **STARTF\_USEPOSITION**  0x00000004 | **dwX** 和 **dwY** 成员包含其他信息。 | | **STARTF\_USESHOWWINDOW**  0x00000001 | **wShowWindow** 成员包含其他信息。 | | **STARTF\_USESIZE**  0x00000002 | **dwXSize** 和 **dwYSize** 成员包含其他信息。 | | **STARTF\_USESTDHANDLES**  0x00000100 | **hStdInput**、**hStdOutput** 和 **hStdError** 成员包含其他信息。  如果在调用其中一个进程创建函数时指定此标志，则句柄必须是可继承的，并且函数的 *bInheritHandles* 参数必须设置为 TRUE。 有关详细信息，请参阅 [处理继承](https://learn.microsoft.com/zh-cn/windows/desktop/SysInfo/handle-inheritance)。  如果在调用 [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow) 函数时指定此标志，则这些成员要么是在进程创建期间指定的句柄值，要么INVALID\_HANDLE\_VALUE。  当不再需要句柄时，必须使用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭这些句柄。  此标志不能与 **STARTF\_USEHOTKEY**一起使用。 |   wShowWindow  如果 **dwFlags** 指定STARTF\_USESHOWWINDOW，则此成员可以是可在 [ShowWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-showwindow) 函数的 *nCmdShow* 参数中指定的任何值，SW\_SHOWDEFAULT除外。 否则，将忽略此成员。  对于 GUI 进程，首次调用 [ShowWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-showwindow) 时，忽略其 *nCmdShow* 参数 **wShowWindow** 指定默认值。 在对 [ShowWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-showwindow) 的后续调用中，如果将 **ShowWindow** 的 *nCmdShow* 参数设置为 SW\_SHOWDEFAULT，则使用 **wShowWindow** 成员。  cbReserved2  保留供 C 运行时使用;必须为零。  lpReserved2  保留供 C 运行时使用;必须为 NULL。  hStdInput  如果 **dwFlags** 指定STARTF\_USESTDHANDLES，则此成员是进程的标准输入句柄。 如果未指定STARTF\_USESTDHANDLES，则标准输入的默认值为键盘缓冲区。  如果 **dwFlags** 指定STARTF\_USEHOTKEY，则此成员将指定一个热键值，该值作为[WM\_SETHOTKEY](https://learn.microsoft.com/zh-cn/windows/win32/inputdev/wm-sethotkey)消息的 *wParam* 参数发送到拥有进程的应用程序创建的第一个符合条件的顶级窗口。 如果窗口是使用WS\_POPUP窗口样式创建的，则它不符合条件，除非还设置了WS\_EX\_APPWINDOW扩展窗口样式。 有关详细信息，请参阅 [CreateWindowEx](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/nf-winuser-createwindowexa)。  否则，将忽略此成员。  hStdOutput  如果 **dwFlags** 指定STARTF\_USESTDHANDLES，则此成员是进程的标准输出句柄。 否则，将忽略此成员，标准输出的默认值为控制台窗口的缓冲区。  如果进程是从任务栏或跳转列表启动的，系统会将 **hStdOutput** 设置为包含用于启动进程的任务栏或跳转列表的监视器的句柄。 有关详细信息，请参阅备注。**Windows 7、Windows Server 2008 R2、Windows Vista、Windows Server 2008、Windows XP 和 Windows Server 2003：**Windows 8和Windows Server 2012中引入了此行为。  hStdError  如果 **dwFlags** 指定STARTF\_USESTDHANDLES，则此成员是进程的标准错误句柄。 否则，将忽略此成员，标准错误的默认值为控制台窗口的缓冲区。 注解 对于图形用户界面 (GUI) 进程，此信息会影响 [CreateWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-createwindowa) 函数创建并由 [ShowWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-showwindow) 函数显示的第一个窗口。 对于控制台进程，如果为进程创建了新的控制台，此信息将影响控制台窗口。 进程可以使用 [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow) 函数检索创建进程时指定的 **STARTUPINFO** 结构。  如果正在启动 GUI 进程，并且未指定STARTF\_FORCEONFEEDBACK或STARTF\_FORCEOFFFEEDBACK，则使用进程反馈游标。 GUI 进程是其子系统指定为“windows”的进程。  如果从任务栏或跳转列表启动某个进程，系统会设置 [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow) 以检索 **STARTUPINFO** 结构，并检查设置 **hStdOutput**。 如果是，请使用 [GetMonitorInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-getmonitorinfoa) 检查 **hStdOutput** 是否是 HMONITOR) (有效的监视器句柄。 然后，进程可以使用句柄定位其窗口。  如果指定 **了 STARTF\_UNTRUSTEDSOURCE** 标志，则应用程序应注意命令行不受信任。 如果设置了此标志，应用程序应禁用潜在的危险功能，例如宏、下载的内容和自动打印。 此标志是可选的，但建议调用 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 的应用程序在启动具有不受信任的命令行参数的程序时设置此标志， (如 Web 内容) 提供的命令行参数，以便新创建的进程可以应用适当的策略。  从 Windows Vista 开始支持**STARTF\_UNTRUSTEDSOURCE**标志，但在Windows 10 SDK 之前的 SDK 头文件中未定义它。 若要在Windows 10之前的版本中使用该标志，可以在程序中手动定义它。 示例 下面的代码示例演示如何使用 **StartUpInfoA**。  C++  #include <windows.h>  #include <stdio.h>  #include <tchar.h>  void \_tmain( int argc, TCHAR \*argv[] )  {  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory( &si, sizeof(si) );  si.cb = sizeof(si);  ZeroMemory( &pi, sizeof(pi) );  if( argc != 2 )  {  printf("Usage: %s [cmdline]\n", argv[0]);  return;  }  // Start the child process.  if( !CreateProcess( NULL, // No module name (use command line)  argv[1], // Command line  NULL, // Process handle not inheritable  NULL, // Thread handle not inheritable  FALSE, // Set handle inheritance to FALSE  0, // No creation flags  NULL, // Use parent's environment block  NULL, // Use parent's starting directory  &si, // Pointer to STARTUPINFO structure  &pi ) // Pointer to PROCESS\_INFORMATION structure  )  {  printf( "CreateProcess failed (%d).\n", GetLastError() );  return;  }  // Wait until child process exits.  WaitForSingleObject( pi.hProcess, INFINITE );  // Close process and thread handles.  CloseHandle( pi.hProcess );  CloseHandle( pi.hThread );  }  有关此示例的详细信息，请参阅 [创建进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-processes)。  **备注**  processthreadsapi.h 标头将 STARTUPINFO 定义为别名，该别名根据 UNICODE 预处理器常量的定义自动选择此函数的 ANSI 或 Unicode 版本。 将非特定编码别名与非非特定编码的代码混合使用可能会导致不匹配，从而导致编译或运行时错误。 有关详细信息，请参阅 [**函数原型的约定**](https://learn.microsoft.com/zh-cn/windows/win32/intl/conventions-for-function-prototypes)。 要求  | **要求** | **值** | | --- | --- | | **最低受支持的客户端** | Windows XP [仅限桌面应用] | | **最低受支持的服务器** | Windows Server 2003 [仅限桌面应用] | | **标头** | processthreadsapi.h (包括 Windows Server 2003、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) |  另请参阅 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)  [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera)  [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw)  [CreateProcessWithTokenW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithtokenw)  [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow) |

## PROCESS\_INFORMATION结构体的用法

|  |
| --- |
| 包含有关新创建的进程及其主线程的信息。 它与 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)、 [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera)、 [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw) 或 [CreateProcessWithTokenW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithtokenw) 函数一起使用。 语法 C++复制  typedef struct \_PROCESS\_INFORMATION {  HANDLE hProcess;  HANDLE hThread;  DWORD dwProcessId;  DWORD dwThreadId;  } PROCESS\_INFORMATION, \*PPROCESS\_INFORMATION, \*LPPROCESS\_INFORMATION; 成员 hProcess  新创建进程的句柄。 句柄用于指定对进程对象执行操作的所有函数中的进程。  hThread  新创建的进程的主线程的句柄。 句柄用于指定对线程对象执行操作的所有函数中的线程。  dwProcessId  一个可用于标识进程的 值。 值从创建进程时起有效，直到关闭进程的所有句柄并释放进程对象;此时，可以重用标识符。  dwThreadId  一个值，该值可用于标识线程。 值从创建线程时起有效，直到关闭线程的所有句柄并释放线程对象;此时，可以重用标识符。 注解 如果函数成功，请确保调用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数以在完成操作后关闭 **hProcess** 和 **hThread** 句柄。 否则，当子进程退出时，系统无法清理子进程的进程结构，因为父进程仍具有子进程的开放句柄。 但是，当父进程终止时，系统将关闭这些句柄，因此此时会清理与子进程对象相关的结构。 示例 有关示例，请参阅 [创建进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-processes)。 另请参阅 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)  [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera)  [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw)  [CreateProcessWithTokenW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithtokenw) |

## CreateProcess函数的用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 创建新进程及其主线程。 新进程在调用进程的安全上下文中运行。  如果调用进程正在模拟其他用户，则新进程将令牌用于调用进程，而不是模拟令牌。 若要在模拟令牌表示的用户的安全上下文中运行新进程，请使用 [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera) 或 [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw) 函数。 语法 C++复制  BOOL CreateProcessA(  [in, optional] LPCSTR lpApplicationName,  [in, out, optional] LPSTR lpCommandLine,  [in, optional] LPSECURITY\_ATTRIBUTES lpProcessAttributes,  [in, optional] LPSECURITY\_ATTRIBUTES lpThreadAttributes,  [in] BOOL bInheritHandles,  [in] DWORD dwCreationFlags,  [in, optional] LPVOID lpEnvironment,  [in, optional] LPCSTR lpCurrentDirectory,  [in] LPSTARTUPINFOA lpStartupInfo,  [out] LPPROCESS\_INFORMATION lpProcessInformation  ); 参数 [in, optional] lpApplicationName  要执行的模块的名称。 此模块可以是基于 Windows 的应用程序。 它可以是某种其他类型的模块 (例如 MS-DOS 或 OS/2) （如果本地计算机上提供了相应的子系统）。  字符串可以指定要执行的模块的完整路径和文件名，也可以指定部分名称。 对于部分名称，函数使用当前驱动器和当前目录来完成规范。 函数不会使用搜索路径。 此参数必须包含文件扩展名;不采用默认扩展名。  *lpApplicationName* 参数可以为 **NULL**。 在这种情况下，模块名称必须是 *lpCommandLine* 字符串中第一个空格分隔的标记。 如果使用包含空格的长文件名，请使用带引号的字符串来指示文件名结束和参数开始的位置;否则，文件名不明确。 例如，请考虑字符串“c：\program files\sub dir\program name”。 可以通过多种方式解释此字符串。 系统尝试按以下顺序解释可能性：   1. **c:\program.exe** 2. **c：\program files\sub.exe** 3. **c：\program files\sub dir\program.exe** 4. **c：\program files\sub dir\program name.exe**   如果可执行模块是 16 位应用程序，*lpApplicationName* 应为 *NULL，lpCommandLine* 指向的字符串应指定可执行模块及其参数。  若要运行批处理文件，必须启动命令解释器;将 *lpApplicationName* 设置为 cmd.exe 并将 *lpCommandLine* 设置为以下参数：/c 加上批处理文件的名称。  **重要**  MSRC 工程团队建议不要这样做。 有关更多详细信息，请参阅 [**MS14-019 – 通过.cmd或 .bat 文件修复二进制劫持**](https://msrc.microsoft.com/blog/2014/04/ms14-019-fixing-a-binary-hijacking-via-cmd-or-bat-file/) 。  [in, out, optional] lpCommandLine  要执行的命令行。  此字符串的最大长度为 32,767 个字符，包括 Unicode 终止 null 字符。 如果 *lpApplicationName* 为 **NULL**，则 *lpCommandLine* 的模块名称部分限制为 **MAX\_PATH** 个字符。  此函数的 Unicode 版本 **CreateProcessW** 可以修改此字符串的内容。 因此，此参数不能是指向只读内存 (的指针，例如 **const** 变量或文本字符串) 。 如果此参数是常量字符串，该函数可能会导致访问冲突。  *lpCommandLine* 参数可以为 NULL。 在这种情况下，函数使用 *lpApplicationName* 指向的字符串作为命令行。  如果 *lpApplicationName* 和 *lpCommandLine* 均为非 **NULL**，则 *lpApplicationName* 指向的以 null 结尾的字符串将指定要执行的模块， *lpCommandLine* 指向的以 null 结尾的字符串将指定命令行。 新进程可以使用 [GetCommandLine](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-getcommandlinea) 检索整个命令行。 用 C 编写的控制台进程可以使用 *argc* 和 *argv* 参数来分析命令行。 由于 argv[0] 是模块名称，因此 C 程序员通常将模块名称重复为命令行中的第一个标记。  如果 *lpApplicationName* 为 NULL，则命令行的第一个空格分隔标记将指定模块名称。 如果使用包含空格的长文件名，请使用带引号的字符串来指示文件名结束和参数开始的位置 (请参阅 *lpApplicationName* 参数) 的说明。 如果文件名不包含扩展名，则追加 .exe。 因此，如果文件扩展名.com，则此参数必须包含.com扩展名。 如果文件名以不带扩展名的句点 (.) 结尾，或者文件名包含路径，则不会追加 .exe。 如果文件名不包含目录路径，系统会按以下顺序搜索可执行文件：   1. 从中加载应用程序的目录。 2. 父进程的当前目录。 3. 32 位 Windows 系统目录。 使用 [GetSystemDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getsystemdirectorya) 函数获取此目录的路径。 4. 16 位 Windows 系统目录。 没有获取此目录的路径的函数，但会对其进行搜索。 此目录的名称为 System。 5. Windows 目录。 使用 [GetWindowsDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getwindowsdirectorya) 函数获取此目录的路径。 6. PATH 环境变量中列出的目录。 请注意，此函数不会搜索由 **应用路径** 注册表项指定的每个应用程序路径。 若要在搜索序列中包含此每个应用程序的路径，请使用 [ShellExecute](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/axe/shellexecute) 函数。   系统向命令行字符串添加一个终止 null 字符，以将文件名与参数分开。 这会将原始字符串划分为两个字符串以供内部处理。  [in, optional] lpProcessAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构确定返回的新进程对象的句柄是否可以由子进程继承。 如果 *lpProcessAttributes* 为 **NULL**，则不能继承句柄。  结构的 **lpSecurityDescriptor** 成员为新进程指定安全描述符。 如果 *lpProcessAttributes* 为 NULL 或 **lpSecurityDescriptor** 为 **NULL**，则进程将获取默认安全描述符。 进程的默认安全描述符中的 ACL 来自创建者的主令牌。**Windowsxp：**进程的默认安全描述符中的 ACL 来自创建者的主要令牌或模拟令牌。 此行为随 Windows XP SP2 和 Windows Server 2003 更改。  [in, optional] lpThreadAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构确定返回的新线程对象的句柄是否可以由子进程继承。 如果 *lpThreadAttributes* 为 NULL，则不能继承句柄。  结构的 **lpSecurityDescriptor** 成员指定主线程的安全描述符。 如果 *lpThreadAttributes* 为 NULL 或 **lpSecurityDescriptor** 为 NULL，则线程获取默认安全描述符。 线程的默认安全描述符中的 ACL 来自进程令牌。**Windowsxp：**线程的默认安全描述符中的 ACL 来自创建者的主令牌或模拟令牌。 此行为随 Windows XP SP2 和 Windows Server 2003 更改。  [in] bInheritHandles  如果此参数为 TRUE，则调用进程中的每个可继承句柄都由新进程继承。 如果参数为 FALSE，则不继承句柄。 请注意，继承的句柄与原始句柄具有相同的值和访问权限。 有关可继承句柄的其他讨论，请参阅备注。  **终端服务：**不能跨会话继承句柄。 此外，如果此参数为 TRUE，则必须在调用方所在的同一会话中创建进程。  **受保护的流程灯 (PPL) 进程：**当 PPL 进程创建非 PPL 进程时，将阻止泛型句柄继承，因为不允许将PROCESS\_DUP\_HANDLE从非 PPL 进程转换为 PPL 进程。 请参阅 [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-security-and-access-rights)  **Windows 7：**即使 参数为 FALSE，也会继承STD\_INPUT\_HANDLE、STD\_OUTPUT\_HANDLE和STD\_ERROR\_HANDLE。  [in] dwCreationFlags  控制优先级类和进程的创建的标志。 有关值的列表，请参阅 [进程创建标志](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-creation-flags)。  此参数还控制新进程的优先级类，该类用于确定进程线程的计划优先级。 有关值的列表，请参阅 [GetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getpriorityclass)。 如果未指定任何优先级类标志，则优先级类默认为 **NORMAL\_PRIORITY\_CLASS** ，除非创建过程的优先级类 **IDLE\_PRIORITY\_CLASS** 或 **BELOW\_NORMAL\_PRIORITY\_CLASS**。 在这种情况下，子进程接收调用进程的默认优先级类。  如果 dwCreationFlags 参数的值为 0：   * 进程同时继承调用方和父级控制台的错误模式。 * 假定新进程的环境块包含 ANSI 字符 (请参阅 lpEnvironment 参数以获取) 的其他信息。 * 基于 16 位 Windows 的应用程序 (VDM) 在共享虚拟 DOS 计算机中运行。   [in, optional] lpEnvironment  指向新进程的环境块的指针。 如果此参数为 **NULL**，则新进程使用调用进程的 环境。  环境块由以 null 结尾的字符串的以 null 结尾的块组成。 每个字符串采用以下格式：  *名字*=*value*\0  由于等号用作分隔符，因此不得在环境变量的名称中使用。  环境块可以包含 Unicode 或 ANSI 字符。 如果 *lpEnvironment* 指向的环境块包含 Unicode 字符，请确保 *dwCreationFlags* 包含 **CREATE\_UNICODE\_ENVIRONMENT**。  如果进程的环境块的总大小超过 32,767 个字符，则此函数的 **ANSI 版本 CreateProcessA** 将失败。  请注意，ANSI 环境块以两个零字节结尾：一个字节用于最后一个字符串，另一个字节用于终止该块。 Unicode 环境块以四个零字节结尾：两个字节表示最后一个字符串，另外两个字节终止该块。  [in, optional] lpCurrentDirectory  进程当前目录的完整路径。 字符串还可以指定 UNC 路径。  如果此参数为 **NULL**，则新进程将具有与调用进程相同的当前驱动器和目录。 (此功能主要用于需要启动应用程序并指定其初始驱动器和工作目录的 shell。)  [in] lpStartupInfo  指向 [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 或 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 结构的指针。  若要设置扩展属性，请使用 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 结构，并在 *dwCreationFlags* 参数中指定EXTENDED\_STARTUPINFO\_PRESENT。  当不再需要 [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 或 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 中的句柄时，必须使用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭它们。  **重要**  调用方负责确保 [**STARTUPINFO**](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 中的标准句柄字段包含有效的句柄值。 即使 **dwFlags** 成员指定了STARTF\_USESTDHANDLES，这些字段也会保持不变地复制到子进程 **，而无需验证**。 不正确的值可能会导致子进程出现错误或崩溃。 使用 [**应用程序验证程序**](https://learn.microsoft.com/zh-cn/windows-hardware/drivers/devtest/application-verifier) 运行时验证工具检测无效句柄。  [out] lpProcessInformation  指向接收有关新进程的标识信息的 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 结构的指针。  当不再需要 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 中的句柄时，必须使用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭它们。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 要获得更多的错误信息，请调用 GetLastError。  请注意， 函数在进程完成初始化之前返回 。 如果找不到所需的 DLL 或无法初始化，则进程将终止。 若要获取进程的终止状态，请调用 [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)。 注解 为进程分配了一个进程标识符。 标识符在进程终止之前有效。 它可用于标识进程，或在 [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess) 函数中指定以打开进程的句柄。 进程中的初始线程也分配有线程标识符。 可以在 [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread) 函数中指定它以打开线程的句柄。 标识符在线程终止之前有效，可用于唯一标识系统中的线程。 这些标识符在 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 结构中返回。  操作系统提供给进程的命令行中可执行文件的名称不一定与调用进程提供给 **CreateProcess** 函数的命令行中的名称相同。 操作系统可能会在未提供完全限定路径的可执行文件名称前面添加一个完全限定的路径。  调用线程可以使用 [WaitForInputIdle](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-waitforinputidle) 函数等待，直到新进程完成其初始化，并且正在等待没有挂起输入的用户输入。 这对于父进程和子进程之间的同步非常有用，因为 **CreateProcess** 返回时不会等待新进程完成其初始化。 例如，创建过程会在尝试查找与新进程关联的窗口之前使用 **WaitForInputIdle** 。  关闭进程的首选方法是使用 [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess) 函数，因为此函数会向附加到进程的所有 DLL 发送即将终止的通知。 关闭进程的其他方法不会通知附加的 DLL。 请注意，当线程调用 **ExitProcess** 时，进程的其他线程将终止，没有机会执行任何其他代码 (包括附加 DLL 的线程终止代码) 。 有关详细信息，请参阅 [终止进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/terminating-a-process)。  父进程可以在进程创建期间直接更改子进程的环境变量。 这是一个进程可以直接更改另一个进程的环境设置的唯一情况。 有关详细信息，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。  如果应用程序提供环境块，则系统驱动器的当前目录信息不会自动传播到新进程。 例如，有一个名为 =C： 的环境变量，其值为驱动器 C 上的当前目录。应用程序必须手动将当前目录信息传递给新进程。 为此，应用程序必须显式创建这些环境变量字符串， (按字母顺序对它们进行排序，因为系统使用排序的环境) ，并将其放入环境块中。 通常，由于环境块排序顺序，它们将位于环境块的前面。  获取驱动器 X 的当前目录信息的一种方法是进行以下调用： GetFullPathName("X:", ...)。 这避免了应用程序必须扫描环境块。 如果返回的完整路径为 X：，则无需将该值作为环境数据传递，因为根目录是新进程的驱动器 X 的默认当前目录。  使用指定**CREATE\_NEW\_PROCESS\_GROUP**创建进程时，将代表新进程隐式调用 [SetConsoleCtrlHandler](https://learn.microsoft.com/zh-cn/windows/console/setconsolectrlhandler) (**NULL，TRUE**) ;这意味着新进程已禁用 CTRL+C。 这允许 shell 自行处理 CTRL+C，并有选择地将该信号传递给子进程。 CTRL+BREAK 未禁用，可用于中断进程/进程组。  默认情况下，将 **TRUE** 作为 *bInheritHandles* 参数的值传递会导致新进程继承所有可继承句柄。 如果应用程序同时从多个线程创建进程，但希望每个进程继承不同的句柄，这可能会产生问题。 应用程序可以将 [UpdateProcThreadAttributeList](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-updateprocthreadattribute) 函数与 **PROC\_THREAD\_ATTRIBUTE\_HANDLE\_LIST** 参数一起使用，以提供要由特定进程继承的句柄列表。 安全备注 第一个参数 *lpApplicationName* 可以为 **NULL**，在这种情况下，可执行文件名称必须位于 *lpCommandLine* 指向的空格分隔字符串中。 如果可执行文件或路径名称中有空格，则存在运行其他可执行文件的风险，因为函数分析空格的方式。 下面的示例很危险，因为函数将尝试运行“Program.exe”（如果存在），而不是运行“MyApp.exe”。  syntax复制  LPTSTR szCmdline = \_tcsdup(TEXT("C:\\Program Files\\MyApp -L -S"));  CreateProcess(NULL, szCmdline, /\* ... \*/);  如果恶意用户在系统上创建名为“Program.exe”的应用程序，则使用 Program Files 目录错误地调用 **CreateProcess** 的任何程序都将运行此应用程序而不是预期的应用程序。  若要避免此问题，请不要为 *lpApplicationName* 传递 **NULL**。 如果确实为 *lpApplicationName* 传递 **NULL**，请在 *lpCommandLine* 中的可执行路径周围使用引号，如以下示例所示。  syntax复制  LPTSTR szCmdline[] = \_tcsdup(TEXT("\"C:\\Program Files\\MyApp\" -L -S"));  CreateProcess(NULL, szCmdline, /\*...\*/); 示例 有关示例，请参阅 [创建进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-processes)。  **备注**  processthreadsapi.h 标头将 CreateProcess 定义为别名，该别名根据 UNICODE 预处理器常量的定义自动选择此函数的 ANSI 或 Unicode 版本。 将非特定编码别名与非非特定编码的代码混合使用可能会导致不匹配，从而导致编译或运行时错误。 有关详细信息，请参阅 [**函数原型的约定**](https://learn.microsoft.com/zh-cn/windows/win32/intl/conventions-for-function-prototypes)。 要求  |  |  | | --- | --- | | **标头** | processthreadsapi.h (包括 Windows Server 2003、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle)  [ShellExecuteA](https://learn.microsoft.com/zh-cn/windows/win32/api/shellapi/nf-shellapi-shellexecutea)  [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera)  [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw)  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [GetCommandLine](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-getcommandlinea)  [GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/rrascfg/nf-rrascfg-ieapproviderconfig-initialize)  [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)  [GetFullPathName](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-getfullpathnamea)  [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow)  [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess)  [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/child-processes)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa)  [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa)  [SetErrorMode](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-seterrormode)  [TerminateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-terminateprocess)  [WaitForInputIdle](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-waitforinputidle) |

## TerminateProcess函数的用法，终止其他进程的运行

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 终止指定的进程及其所有线程。 语法 C++  BOOL TerminateProcess(  [in] HANDLE hProcess,  [in] UINT uExitCode  ); 参数 [in] hProcess  要终止的进程句柄。  句柄必须具有 **PROCESS\_TERMINATE** 访问权限。 有关详细信息，请参阅 [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/process-security-and-access-rights)。  [in] uExitCode  进程和线程将使用的退出代码，由于此调用而终止。 使用 [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess) 函数检索进程的退出值。 使用 [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread) 函数检索线程的退出值。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 要获得更多的错误信息，请调用 GetLastError。 注解 **TerminateProcess** 函数用于无条件地导致进程退出。 如果使用 **TerminateProcess** 而不是 [ExitProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess)，则动态链接库 (DLL) 维护的全局数据状态可能会受到威胁。  此函数停止进程内所有线程的执行，并请求取消所有挂起的 I/O。 在完成或取消所有挂起的 I/O 之前，终止的进程无法退出。 当进程终止时，其内核对象不会销毁，直到向进程开放句柄的所有进程释放这些句柄。  当进程终止自身时， **TerminateProcess** 将停止调用线程的执行，并且不返回。 否则， **TerminateProcess** 是异步的;它启动终止并立即返回。 如果需要确保进程已终止，请使用进程的句柄调用 [WaitForSingleObject](https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject) 函数。  进程无法阻止自身被终止。  进程终止后，使用打开进程句柄调用 **TerminateProcess** 失败 **，ERROR\_ACCESS\_DENIED** (5) 错误代码。 要求  |  |  | | --- | --- | | **标头** | processthreadsapi.h (包括 Windows Server 2003 上的 Windows.h、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [ExitProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)  [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread)  [OpenProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/process-and-thread-functions)  [进程](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/child-processes)  [终止进程](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/terminating-a-process)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

## SetEnvironmentVariable函数的用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 设置当前进程的指定环境变量的内容。 语法 C++复制  BOOL SetEnvironmentVariable(  [in] LPCTSTR lpName,  [in, optional] LPCTSTR lpValue  ); 参数 [in] lpName  环境变量的名称。 如果环境变量不存在且 *lpValue* 不为 NULL，则操作系统将创建该环境变量。  [in, optional] lpValue  环境变量的内容。 用户定义的环境变量的最大大小为 32,767 个字符。 有关详细信息，请参阅 [环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/environment-variables)。  **Windows Server 2003 和 Windows XP：**进程的环境块的总大小不能超过 32,767 个字符。  如果此参数为 NULL，则会从当前进程的环境中删除变量。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 要获得更多的错误信息，请调用 GetLastError。 注解 此函数对系统环境变量或其他进程的环境变量没有影响。 示例 有关示例，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。 要求  |  |  | | --- | --- | | **标头** | winbase.h (包括 Windows Server 2003 上的 Windows.h、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/environment-variables)  [GetEnvironmentVariable](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-getenvironmentvariable) |

## CFile 类，参考[CFile类文档](CFile类.docx)

# 演练

## 1.新建一个文件夹取名Lesson61-inherit-env-var，然后在里面创建一个基于MFC的应用程序parent-process,界面如下

|  |
| --- |
|  |
|  |
|  |

## 注意：为了防止我们的应用程序在用户按下回车就立马退出，我们需要添加一个PreTranslateMessage函数，在里面屏蔽回车键和esc键的响应。当然，也可以在IDOK按钮的响应代码里面用消息框的方法来实现

|  |
| --- |
|  |
|  |

## 2.然后给这两个编辑框绑定成员变量

|  |
| --- |
|  |

## 3.在按钮的点击事件中先检查编辑框的内容是否为空，如果是，提示用户输入内容，然后对于的编辑框会自动获得焦点

|  |
| --- |
|  |

### 效果

|  |
| --- |
|  |
|  |
|  |

## 4.然后我们在这个项目里面点击右击-》添加-》新建项目

|  |
| --- |
|  |

### 添加一个项目，叫做child-process，选择MFC基于对话框的应用程序，点击完成按钮

|  |
| --- |
|  |
|  |
|  |

### 然后我们把child-process的对话框标题改为：子进程应用程序，把不需要的控件删除，把确定按钮的文本改为退出，

|  |
| --- |
|  |

### 同样的，我们需要添加防止按下回车键退出程序的代码

|  |
| --- |
|  |

## 5.先把child-process设置为启动项目，然后运行，效果如图

|  |
| --- |
|  |
|  |
|  |

### 测试正常。

## 6.然后我们需要把parent-process项目设置为启动项目

|  |
| --- |
|  |

## 7.给子进程添加一个按钮，文本为获取环境变量，id为：IDC\_BTN\_GET\_VAR

|  |
| --- |
|  |

## 8.双击按钮给他添加一个点击事件处理函数

|  |
| --- |
|  |

## 9.回到父进程，在创建按钮点击事件函数里面设置环境变量然后启动子进程，我们先来实现启动子进程的代码

|  |
| --- |
| void CParentprocessDlg::OnBnClickedBtnCreate()  {  // TODO: 在此添加控件通知处理程序代码  //定义创建进程需要的结构体变量  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si,sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi,sizeof(pi));  //==============end===  CString strVar,strVal;  GetDlgItemText(IDC\_EDIT\_VAR,strVar);  GetDlgItemText(IDC\_EDIT\_VAL,strVal);  //检查两个编辑框是否有内容  if(strVar.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的名称"));  m\_editVar.SetFocus();  return;  }  else if( strVal.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的值"));  m\_editVal.SetFocus();  return;  }    //启动子进程  if(!CreateProcess(\_T("F:\\Projects\_f\\Learn-win-API\\Lesson61-inherit-env-var\\Parent-process\\Debug\\child-process.exe"),  NULL,NULL,NULL,FALSE,NULL,0,NULL,&si,&pi))  {  MessageBox(\_T("创建子进程失败"));  return;  }    } |

### 效果，点击按钮弹出子进程对话框

|  |
| --- |
|  |

## 10.其实我们需要在创建子进程之前设置好环境变量，我们之所以要先编写创建子进程的代码是为了保证子进程能够顺利启动，我们需要在创建子进程的代码之前添加设置环境变量的代码

|  |
| --- |
| void CParentprocessDlg::OnBnClickedBtnCreate()  {  // TODO: 在此添加控件通知处理程序代码  //定义创建进程需要的结构体变量  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si,sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi,sizeof(pi));  //==============end===  CString strVar,strVal;  GetDlgItemText(IDC\_EDIT\_VAR,strVar);  GetDlgItemText(IDC\_EDIT\_VAL,strVal);  //检查两个编辑框是否有内容  if(strVar.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的名称"));  m\_editVar.SetFocus();  return;  }  else if( strVal.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的值"));  m\_editVal.SetFocus();  return;  }  //设置环境变量  if(!SetEnvironmentVariable(strVar,strVal))  {  MessageBox(\_T("设置环境变量失败！！！"));  return;  }    //启动子进程  if(!CreateProcess(\_T("F:\\Projects\_f\\Learn-win-API\\Lesson61-inherit-env-var\\Parent-process\\Debug\\child-process.exe"),  NULL,NULL,NULL,FALSE,NULL,0,NULL,&si,&pi))  {  MessageBox(\_T("创建子进程失败"));  return;  }      } |

## 11.现在我们在子进程的按钮点击函数里面添加获取环境变量的值，我们这里比较简单，我们指定需要获取的环境变量名称

|  |
| --- |
|  |

## 12，其实我们可以把需要获取的环境变量写到一个文件中，然后子进程也从文件中读取环境变量名称然后获取他的值，我们在父进程设置环境变量后把环境变量的名字设置到一个叫做config.txt文件中

|  |
| --- |
| void CParentprocessDlg::OnBnClickedBtnCreate()  {  // TODO: 在此添加控件通知处理程序代码  //定义创建进程需要的结构体变量  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si,sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi,sizeof(pi));  //==============end===  CString strVar,strVal;  GetDlgItemText(IDC\_EDIT\_VAR,strVar);  GetDlgItemText(IDC\_EDIT\_VAL,strVal);  CHAR szName[256];  ZeroMemory(szName,256);  //检查两个编辑框是否有内容  if(strVar.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的名称"));  m\_editVar.SetFocus();  return;  }  else if( strVal.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的值"));  m\_editVal.SetFocus();  return;  }  //设置环境变量  if(!SetEnvironmentVariable(strVar,strVal))  {  MessageBox(\_T("设置环境变量失败！！！"));  return;  }  //把我们设置是环境变量保存到config.txt文件中  ::GetDlgItemTextA(this->m\_hWnd,IDC\_EDIT\_VAR,szName,256);  CFile file;  file.Open(\_T("config.txt"),CFile::modeCreate|CFile::modeWrite);  file.Write(szName,strlen(szName));  file.Close();  /\*FILE \*pf;  pf = fopen("config1.txt","w+");  fwrite(szName,1,strlen(szName),pf);  fclose(pf);\*/  //启动子进程  if(!CreateProcess(\_T("F:\\Projects\_f\\Learn-win-API\\Lesson61-inherit-env-var\\Parent-process\\Debug\\child-process.exe"),  NULL,NULL,NULL,FALSE,NULL,0,NULL,&si,&pi))  {  MessageBox(\_T("创建子进程失败"));  return;  }      } |

## 13，然后在子进程的按钮点击事件函数里面读取文件获取键然后获取他的值并且弹出消息框显示，注意读取文本文件最好使用CStdioFile，否则很有可能会出现乱码

|  |
| --- |
| //读取父进程设置的环境变量  void CchildprocessDlg::OnBnClickedBtnGetVar()  {  // TODO: 在此添加控件通知处理程序代码  //从文件中获取环境变量名称,最好使用CStidoFile类来操作  TCHAR szKName[256];  memset(szKName,0,256);  //CStdioFile file(\_T("config1.txt"),CFile::modeRead);  CStdioFile file(\_T("config.txt"),CFile::modeRead);  file.ReadString(szKName,256);  SetDlgItemText(IDC\_VAR,CString(szKName));  file.Close();  //-------------------  //根据环境变量名称获取值  TCHAR szVName[256];  memset(szVName,0,256);    if(0 == GetEnvironmentVariable(szKName,szVName,256))  {  MessageBox(\_T("获取环境变量失败"));  return;  }  CString str;  str.Format(\_T("环境变量的值：%s"),szVName);  MessageBox(str);  } |

### 效果

|  |
| --- |
|  |
|  |

### 这也是一种进程间通信的办法

# 这一节的学习到此为止，完整代码如下，主要是对话框类的代码，因为应用程序类基本上不用管。

## parent-processDlg.h

|  |
| --- |
| // Lesson58-create-processDlg.h : 头文件  //  #pragma once  #include "afxwin.h"  // CLesson58createprocessDlg 对话框  class CParentprocessDlg : public CDialogEx  {  // 构造  public:  CParentprocessDlg(CWnd\* pParent = NULL); // 标准构造函数  // 对话框数据  enum { IDD = IDD\_PARENTPROCESS\_DIALOG };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  HICON m\_hIcon;  // 生成的消息映射函数  virtual BOOL OnInitDialog();  afx\_msg void OnSysCommand(UINT nID, LPARAM lParam);  afx\_msg void OnPaint();  afx\_msg HCURSOR OnQueryDragIcon();  DECLARE\_MESSAGE\_MAP()  protected:  virtual BOOL PreTranslateMessage(MSG\* pMsg);//阻止按下回车结束程序  public:  afx\_msg void OnBnClickedBtnCreate();  // 和编辑框绑定的成员变量  afx\_msg void OnBnClickedExit();  // 输入环境变量名称的编辑框成员变量  CEdit m\_editVar;  // 输入环境变量值的编辑框成员变量  CEdit m\_editVal;  }; |

## parent-processDlg.cpp

|  |
| --- |
| // Lesson58-create-processDlg.cpp : 实现文件  //  #include "stdafx.h"  #include "parent-process.h"  #include "parent-processDlg.h"  #include "afxdialogex.h"  #include<stdio.h>  #ifdef \_DEBUG  #define new DEBUG\_NEW  #endif  // 用于应用程序“关于”菜单项的 CAboutDlg 对话框  class CAboutDlg : public CDialogEx  {  public:  CAboutDlg();  // 对话框数据  enum { IDD = IDD\_ABOUTBOX };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  DECLARE\_MESSAGE\_MAP()  };  CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)  {  }  void CAboutDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CAboutDlg, CDialogEx)  END\_MESSAGE\_MAP()  // CLesson58createprocessDlg 对话框  //对话框构造函数  CParentprocessDlg::CParentprocessDlg(CWnd\* pParent /\*=NULL\*/)  : CDialogEx(CParentprocessDlg::IDD, pParent)  {  m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);  }  void CParentprocessDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  DDX\_Control(pDX, IDC\_EDIT\_VAR, m\_editVar);  DDX\_Control(pDX, IDC\_EDIT\_VALUE, m\_editVal);  }  BEGIN\_MESSAGE\_MAP(CParentprocessDlg, CDialogEx)  ON\_WM\_SYSCOMMAND()  ON\_WM\_PAINT()  ON\_WM\_QUERYDRAGICON()  ON\_BN\_CLICKED(IDC\_BTN\_CREATE, &CParentprocessDlg::OnBnClickedBtnCreate)  ON\_BN\_CLICKED(IDC\_EXIT, &CParentprocessDlg::OnBnClickedExit)  END\_MESSAGE\_MAP()  // CLesson58createprocessDlg 消息处理程序  BOOL CParentprocessDlg::OnInitDialog()  {  CDialogEx::OnInitDialog();  // 将“关于...”菜单项添加到系统菜单中。  // IDM\_ABOUTBOX 必须在系统命令范围内。  ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);  ASSERT(IDM\_ABOUTBOX < 0xF000);  CMenu\* pSysMenu = GetSystemMenu(FALSE);  if (pSysMenu != NULL)  {  BOOL bNameValid;  CString strAboutMenu;  bNameValid = strAboutMenu.LoadString(IDS\_ABOUTBOX);  ASSERT(bNameValid);  if (!strAboutMenu.IsEmpty())  {  pSysMenu->AppendMenu(MF\_SEPARATOR);  pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  // 设置此对话框的图标。当应用程序主窗口不是对话框时，框架将自动  // 执行此操作  SetIcon(m\_hIcon, TRUE); // 设置大图标  SetIcon(m\_hIcon, FALSE); // 设置小图标  // TODO: 在此添加额外的初始化代码  return TRUE; // 除非将焦点设置到控件，否则返回 TRUE  }  void CParentprocessDlg::OnSysCommand(UINT nID, LPARAM lParam)  {  if ((nID & 0xFFF0) == IDM\_ABOUTBOX)  {  CAboutDlg dlgAbout;  dlgAbout.DoModal();  }  else  {  CDialogEx::OnSysCommand(nID, lParam);  }  }  // 如果向对话框添加最小化按钮，则需要下面的代码  // 来绘制该图标。对于使用文档/视图模型的 MFC 应用程序，  // 这将由框架自动完成。  void CParentprocessDlg::OnPaint()  {  if (IsIconic())  {  CPaintDC dc(this); // 用于绘制的设备上下文  SendMessage(WM\_ICONERASEBKGND, reinterpret\_cast<WPARAM>(dc.GetSafeHdc()), 0);  // 使图标在工作区矩形中居中  int cxIcon = GetSystemMetrics(SM\_CXICON);  int cyIcon = GetSystemMetrics(SM\_CYICON);  CRect rect;  GetClientRect(&rect);  int x = (rect.Width() - cxIcon + 1) / 2;  int y = (rect.Height() - cyIcon + 1) / 2;  // 绘制图标  dc.DrawIcon(x, y, m\_hIcon);  }  else  {  CDialogEx::OnPaint();  }  }  //当用户拖动最小化窗口时系统调用此函数取得光标  //显示。  HCURSOR CParentprocessDlg::OnQueryDragIcon()  {  return static\_cast<HCURSOR>(m\_hIcon);  }  BOOL CParentprocessDlg::PreTranslateMessage(MSG\* pMsg)  {  if (pMsg->message == WM\_KEYDOWN)  {  switch (pMsg->wParam)  {  case VK\_RETURN: //屏蔽回车键  return TRUE;  case VK\_ESCAPE: //屏蔽ESC键  return TRUE;    default:  break;  }  }  return CDialog::PreTranslateMessage(pMsg);  }  void CParentprocessDlg::OnBnClickedBtnCreate()  {  // TODO: 在此添加控件通知处理程序代码  //定义创建进程需要的结构体变量  STARTUPINFO si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si,sizeof(si));  si.cb = sizeof(si);  ZeroMemory(&pi,sizeof(pi));  //==============end===  CString strVar,strVal;  GetDlgItemText(IDC\_EDIT\_VAR,strVar);  GetDlgItemText(IDC\_EDIT\_VAL,strVal);  CHAR szName[256];  ZeroMemory(szName,256);  //检查两个编辑框是否有内容  if(strVar.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的名称"));  m\_editVar.SetFocus();  return;  }  else if( strVal.GetLength()==0)  {  MessageBox(\_T("请输入环境变量的值"));  m\_editVal.SetFocus();  return;  }  //设置环境变量  if(!SetEnvironmentVariable(strVar,strVal))  {  MessageBox(\_T("设置环境变量失败！！！"));  return;  }  //把我们设置是环境变量保存到config.txt文件中  ::GetDlgItemTextA(this->m\_hWnd,IDC\_EDIT\_VAR,szName,256);  CFile file;  file.Open(\_T("config.txt"),CFile::modeCreate|CFile::modeWrite);  file.Write(szName,strlen(szName));  file.Close();  /\*FILE \*pf;  pf = fopen("config1.txt","w+");  fwrite(szName,1,strlen(szName),pf);  fclose(pf);\*/  //启动子进程  if(!CreateProcess(\_T("F:\\Projects\_f\\Learn-win-API\\Lesson61-inherit-env-var\\Parent-process\\Debug\\child-process.exe"),  NULL,NULL,NULL,FALSE,NULL,0,NULL,&si,&pi))  {  MessageBox(\_T("创建子进程失败"));  return;  }      }  void CParentprocessDlg::OnBnClickedExit()  {  // TODO: 在此添加控件通知处理程序代码  if(IDOK==::MessageBox(this->m\_hWnd,\_T("退出程序?"),\_T("退出确认"),MB\_OKCANCEL))  {  //ExitProcess(0); //window退出方法  PostQuitMessage(0);//window退出方法2  //exit(0);//c语言的退出方法  }  } |

## child-processDlg.h

|  |
| --- |
| // child-processDlg.h : 头文件  //  #pragma once  // CchildprocessDlg 对话框  class CchildprocessDlg : public CDialogEx  {  // 构造  public:  CchildprocessDlg(CWnd\* pParent = NULL); // 标准构造函数  // 对话框数据  enum { IDD = IDD\_CHILDPROCESS\_DIALOG };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  HICON m\_hIcon;  // 生成的消息映射函数  virtual BOOL OnInitDialog();  afx\_msg void OnSysCommand(UINT nID, LPARAM lParam);  afx\_msg void OnPaint();  afx\_msg HCURSOR OnQueryDragIcon();  DECLARE\_MESSAGE\_MAP()  public:  afx\_msg void OnBnClickedOk();  afx\_msg void OnBnClickedBtnGetVar();  }; |

## child-processDlg.cpp

|  |
| --- |
| // child-processDlg.cpp : 实现文件  //  #include "stdafx.h"  #include "child-process.h"  #include "child-processDlg.h"  #include "afxdialogex.h"  #include<stdio.h>  #ifdef \_DEBUG  #define new DEBUG\_NEW  #endif  // 用于应用程序“关于”菜单项的 CAboutDlg 对话框  class CAboutDlg : public CDialogEx  {  public:  CAboutDlg();  // 对话框数据  enum { IDD = IDD\_ABOUTBOX };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  DECLARE\_MESSAGE\_MAP()  };  CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)  {  }  void CAboutDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CAboutDlg, CDialogEx)  END\_MESSAGE\_MAP()  // CchildprocessDlg 对话框  CchildprocessDlg::CchildprocessDlg(CWnd\* pParent /\*=NULL\*/)  : CDialogEx(CchildprocessDlg::IDD, pParent)  {  m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);  }  void CchildprocessDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CchildprocessDlg, CDialogEx)  ON\_WM\_SYSCOMMAND()  ON\_WM\_PAINT()  ON\_WM\_QUERYDRAGICON()  ON\_BN\_CLICKED(IDOK, &CchildprocessDlg::OnBnClickedOk)  ON\_BN\_CLICKED(IDC\_BTN\_GET\_VAR, &CchildprocessDlg::OnBnClickedBtnGetVar)  END\_MESSAGE\_MAP()  // CchildprocessDlg 消息处理程序  BOOL CchildprocessDlg::OnInitDialog()  {  CDialogEx::OnInitDialog();  // 将“关于...”菜单项添加到系统菜单中。  // IDM\_ABOUTBOX 必须在系统命令范围内。  ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);  ASSERT(IDM\_ABOUTBOX < 0xF000);  CMenu\* pSysMenu = GetSystemMenu(FALSE);  if (pSysMenu != NULL)  {  BOOL bNameValid;  CString strAboutMenu;  bNameValid = strAboutMenu.LoadString(IDS\_ABOUTBOX);  ASSERT(bNameValid);  if (!strAboutMenu.IsEmpty())  {  pSysMenu->AppendMenu(MF\_SEPARATOR);  pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  // 设置此对话框的图标。当应用程序主窗口不是对话框时，框架将自动  // 执行此操作  SetIcon(m\_hIcon, TRUE); // 设置大图标  SetIcon(m\_hIcon, FALSE); // 设置小图标  // TODO: 在此添加额外的初始化代码  return TRUE; // 除非将焦点设置到控件，否则返回 TRUE  }  void CchildprocessDlg::OnSysCommand(UINT nID, LPARAM lParam)  {  if ((nID & 0xFFF0) == IDM\_ABOUTBOX)  {  CAboutDlg dlgAbout;  dlgAbout.DoModal();  }  else  {  CDialogEx::OnSysCommand(nID, lParam);  }  }  // 如果向对话框添加最小化按钮，则需要下面的代码  // 来绘制该图标。对于使用文档/视图模型的 MFC 应用程序，  // 这将由框架自动完成。  void CchildprocessDlg::OnPaint()  {  if (IsIconic())  {  CPaintDC dc(this); // 用于绘制的设备上下文  SendMessage(WM\_ICONERASEBKGND, reinterpret\_cast<WPARAM>(dc.GetSafeHdc()), 0);  // 使图标在工作区矩形中居中  int cxIcon = GetSystemMetrics(SM\_CXICON);  int cyIcon = GetSystemMetrics(SM\_CYICON);  CRect rect;  GetClientRect(&rect);  int x = (rect.Width() - cxIcon + 1) / 2;  int y = (rect.Height() - cyIcon + 1) / 2;  // 绘制图标  dc.DrawIcon(x, y, m\_hIcon);  }  else  {  CDialogEx::OnPaint();  }  }  //当用户拖动最小化窗口时系统调用此函数取得光标  //显示。  HCURSOR CchildprocessDlg::OnQueryDragIcon()  {  return static\_cast<HCURSOR>(m\_hIcon);  }  void CchildprocessDlg::OnBnClickedOk()  {  // TODO: 在此添加控件通知处理程序代码  //防止程序响应回车键退出  if(IDOK==::MessageBox(this->m\_hWnd,\_T("退出程序?"),\_T("退出确认"),MB\_OKCANCEL))  {  CDialogEx::OnOK();  }  }  //读取父进程设置的环境变量  void CchildprocessDlg::OnBnClickedBtnGetVar()  {  // TODO: 在此添加控件通知处理程序代码  //从文件中获取环境变量名称,最好使用CStidoFile类来操作  TCHAR szKName[256];  memset(szKName,0,256);  //CStdioFile file(\_T("config1.txt"),CFile::modeRead);  CStdioFile file(\_T("config.txt"),CFile::modeRead);  file.ReadString(szKName,256);  SetDlgItemText(IDC\_VAR,CString(szKName));  file.Close();  //-------------------  //根据环境变量名称获取值  TCHAR szVName[256];  memset(szVName,0,256);    if(0 == GetEnvironmentVariable(szKName,szVName,256))  {  MessageBox(\_T("获取环境变量失败"));  return;  }  CString str;  str.Format(\_T("环境变量的值：%s"),szVName);  MessageBox(str);  } |

# 扩展：我们这个程序只能传递一个环境变量，我们可以使用一个列表框或者组合框先收集所有的环境变量然后再发送给子进程