# 学习大纲

|  |
| --- |
|  |

## CreateThread函数用法，window下面的创建线程api

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 创建在调用进程的虚拟地址空间内执行的线程。  若要创建在另一个进程的虚拟地址空间中运行的线程，请使用 [CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread) 函数。 语法 C++复制  HANDLE CreateThread(  [in, optional] LPSECURITY\_ATTRIBUTES lpThreadAttributes,  [in] SIZE\_T dwStackSize,  [in] LPTHREAD\_START\_ROUTINE lpStartAddress,  [in, optional] \_\_drv\_aliasesMem LPVOID lpParameter,  [in] DWORD dwCreationFlags,  [out, optional] LPDWORD lpThreadId  ); 参数 [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，该结构确定是否可由子进程继承返回的句柄。 如果 *lpThreadAttributes* 为 NULL，则无法继承句柄。  结构的 **lpSecurityDescriptor** 成员指定新线程的安全描述符。 如果 *lpThreadAttributes* 为 NULL，则线程将获取默认的安全描述符。 线程的默认安全描述符中的 ACL 来自创建者的主令牌。  [in] dwStackSize  堆栈的初始大小（以字节为单位）。 系统将此值舍入到最近的页面。 如果此参数为零，则新线程将使用可执行文件的默认大小。 有关详细信息，请参阅 [线程堆栈大小](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-stack-size)。  [in] lpStartAddress  指向要由线程执行的应用程序定义函数的指针。 此指针表示线程的起始地址。 有关线程函数的详细信息，请参阅 [ThreadProc](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/ms686736(v=vs.85))。  [in, optional] lpParameter  指向要传递给线程的变量的指针。  [in] dwCreationFlags  控制线程创建的标志。  展开表   |  |  | | --- | --- | | **值** | **含义** | | 0 | 线程在创建后立即运行。 | | **CREATE\_SUSPENDED**  0x00000004 | 线程以挂起状态创建，在调用 [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread) 函数之前不会运行。 | | **STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION**  0x00010000 | *dwStackSize* 参数指定堆栈的初始保留大小。 如果未指定此标志， *dwStackSize* 将指定提交大小。 |   [out, optional] lpThreadId  指向接收线程标识符的变量的指针。 如果此参数为 **NULL**，则不返回线程标识符。 返回值 如果函数成功，则返回值是新线程的句柄。  如果函数失败，则返回值为 NULL。 要获得更多的错误信息，请调用 GetLastError。  请注意，即使 *lpStartAddress* 指向数据、代码或不可访问，**CreateThread** 也可能会成功。 如果启动地址在线程运行时无效，则会发生异常，并且线程将终止。 由于启动地址无效而导致的线程终止作为线程进程的错误退出进行处理。 此行为类似于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 的异步性质，即即使它引用无效或缺少动态链接库 (dll) ，也会创建进程。 注解 进程可以创建的线程数受可用虚拟内存的限制。 默认情况下，每个线程都有一 MB 的堆栈空间。 因此，如果没有 /3GB boot.ini 选项，则无法在 32 位系统上创建 2,048 个或更多线程。 如果减小默认堆栈大小，可以创建更多线程。 但是，如果为每个处理器创建一个线程并生成应用程序维护上下文信息的请求队列，则应用程序的性能会更好。 线程会在处理下一个队列中的请求之前处理队列中的所有请求。  使用 **THREAD\_ALL\_ACCESS** 访问权限创建新线程句柄。 如果在创建线程时未提供安全描述符，则会使用正在创建线程的进程的主标记为新线程构造默认安全描述符。 当调用方尝试使用 [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread) 函数访问线程时，将针对此安全描述符评估调用方的有效令牌，以授予或拒绝访问权限。  调用 [GetCurrentThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentthread) 函数时，新创建的线程对自身具有完全访问权限。  **Windows Server 2003：**线程对自身的访问权限是通过根据为线程构造的默认安全描述符评估在其中创建线程的进程的主令牌来计算的。 如果在远程进程中创建线程，则使用远程进程的主标记。 因此，在调用 [GetCurrentThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentthread) 时，新创建的线程可能会降低对自身的访问权限。 某些访问权限（包括 **THREAD\_SET\_THREAD\_TOKEN** 和 **THREAD\_GET\_CONTEXT** ）可能不存在，从而导致意外失败。 因此，不建议在模拟其他用户时创建线程。  如果线程是在可运行状态下创建的， (也就是说，如果未) 使用 **CREATE\_SUSPENDED** 标志，则线程可以在 **CreateThread** 返回之前开始运行，尤其是在调用方收到所创建线程的句柄和标识符之前。  线程执行从 *lpStartAddress* 参数指定的函数开始。 如果此函数返回，则 **DWORD** 返回值用于在 [对 ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread) 函数的隐式调用中终止线程。 使用 [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodethread) 函数获取线程的返回值。  线程的创建优先级为 **THREAD\_PRIORITY\_NORMAL**。 使用 [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority) 和 [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority) 函数获取和设置线程的优先级值。  当线程终止时，线程对象将达到信号状态，满足正在等待该对象的任何线程。  线程对象将保留在系统中，直到线程终止，并且通过调用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭了该线程的所有句柄。  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)、[ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread)、**CreateThread**、[CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread) 函数以及由于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)) 调用而 (启动的进程在进程中相互序列化。 一次只能在地址空间中发生其中一个事件。 这意味着存在以下限制：   * 在进程启动和 DLL 初始化例程期间，可以创建新线程，但在为进程完成 DLL 初始化之前，它们不会开始执行。 * 进程中一次只能有一个线程在 DLL 初始化或分离例程中。 * 在 DLL 初始化或分离例程中没有线程之前，[ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess) 不会完成。   在可执行文件中调用 C 运行时库 (CRT) 的线程应使用 [\_beginthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/beginthread-beginthreadex) 和 [\_endthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex) 函数进行线程管理，而不是 **CreateThread** 和 [ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread);这需要使用 CRT 的多线程版本。 如果使用 **CreateThread** 创建的线程调用 CRT，则 CRT 可能会在内存不足的情况下终止进程。  **Windows Phone 8.1：**Windows Phone 8.1 及更高版本上的 Windows Phone 应用商店应用支持此函数。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 8.1、Windows Server 2012 R2 及更高版本的 Windows 应用商店应用支持此函数。 示例 有关示例，请参阅 [创建线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-threads)。 要求 展开表   | **要求** | **值** | | --- | --- | | **最低受支持的客户端** | Windows XP [桌面应用 | UWP 应用] | | **最低受支持的服务器** | Windows Server 2003 [桌面应用 | UWP 应用] | | **目标平台** | Windows | | **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;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle)  [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)  [CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread)  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread)  [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodethread)  [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority)  [SuspendThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-suspendthread)  [ThreadProc](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/ms686736(v=vs.85))  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## \_beginethread函数用法，c语言的创建线程api，程序员比较喜欢这个，因为它比较简便

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 创建线程。 语法， C++复制  uintptr\_t \_beginthread( // NATIVE CODE  void( \_\_cdecl \*start\_address )( void \* ),  unsigned stack\_size,  void \*arglist  );  uintptr\_t \_beginthread( // MANAGED CODE  void( \_\_clrcall \*start\_address )( void \* ),  unsigned stack\_size,  void \*arglist  );  uintptr\_t \_beginthreadex( // NATIVE CODE  void \*security,  unsigned stack\_size,  unsigned ( \_\_stdcall \*start\_address )( void \* ),  void \*arglist,  unsigned initflag,  unsigned \*thrdaddr  );  uintptr\_t \_beginthreadex( // MANAGED CODE  void \*security,  unsigned stack\_size,  unsigned ( \_\_clrcall \*start\_address )( void \* ),  void \*arglist,  unsigned initflag,  unsigned \*thrdaddr  ); 参数 *start\_address* 启动开始执行新线程的例程的地址。 对于 **\_beginthread**，调用约定为 [\_\_cdecl](https://learn.microsoft.com/zh-cn/cpp/cpp/cdecl?view=msvc-170)（本机代码）或 [\_\_clrcall](https://learn.microsoft.com/zh-cn/cpp/cpp/clrcall?view=msvc-170)（托管代码）。 对于 **\_beginthreadex**，调用约定为 [\_\_stdcall](https://learn.microsoft.com/zh-cn/cpp/cpp/stdcall?view=msvc-170)（本机代码）或 [\_\_clrcall](https://learn.microsoft.com/zh-cn/cpp/cpp/clrcall?view=msvc-170)（托管代码）。  *stack\_size* 新线程的堆栈大小或 0。  *arglist* 要传递到新线程的参数列表或 NULL。  *Security* 指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，此结构确定返回的句柄是否由子进程继承。 如果 *Security* 为 NULL，则不能继承句柄。  *initflag* 控制新线程的初始状态的标志。 将 *initflag* 设置为 0 以立即运行，或设置为 CREATE\_SUSPENDED 以在挂起状态下创建线程；使用 [ResumeThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread) 来执行此线程。 将 *initflag* 设置为 STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION 标志以将 *stack\_size* 用作堆栈的初始保留大小（以字节计）；如果未指定此标志，则 *stack\_size* 将指定提交大小。  *thrdaddr* 指向接收线程标识符的 32 位变量。 如果是 NULL，则不使用它。 返回值 如果成功，则这些函数中的每一个都会返回一个句柄到新创建的线程；但是，如果新创建的线程退出过快，则 **\_beginthread** 可能不会返回有效句柄。 （请参见“备注”部分中的讨论。）发生错误时，**\_beginthread** 返回 -1L，并在线程过多的情况下将 errno 设置为 EAGAIN；如果参数无效或堆栈大小错误，则设置为 EINVAL；如果资源（如内存）不足，则设置为 EACCES。 发生错误时， **\_beginthreadex** 返回 0 并设置 errno 和 **\_doserrno** 。  如果 *start\_address* 为 NULL，则会调用无效的参数处理程序，如[参数验证](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/parameter-validation?view=msvc-170)中所述。 如果允许执行继续，则这些功能将 errno 设置为 EINVAL 并返回 -1。  有关这些和其他的返回代码的详细信息，请参阅 [errno、\_doserrno、\_sys\_errlist 和 \_sys\_nerr](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/errno-doserrno-sys-errlist-and-sys-nerr?view=msvc-170)。  有关 **uintptr\_t** 的详细信息，请参阅[标准类型](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/standard-types?view=msvc-170)。 注解 **\_beginthread** 函数创建一个在 *start\_address*处开始执行例程的线程。 *start\_address* 处的例程必须使用 **\_\_cdecl** （对于本机代码）或 **\_\_clrcall** （对于托管代码）调用约定，且应没有任何返回值。 当线程从该例程返回时，它会自动终止。 有关线程的详细信息，请参阅[针对旧代码的多线程支持 (Visual C++)](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-support-for-older-code-visual-cpp?view=msvc-170)。  与 **\_beginthread** 相比，**\_beginthreadex** 更类似于 Win32 [CreateThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread) API。 **\_beginthreadex** 与 **\_beginthread** 的差别体现在以下方面：   * **\_beginthreadex** 还有三个参数：*initflag*、*Security*、**threadaddr**。 新线程可通过指定的 security 创建为挂起状态，并且可使用线程标识符 *thrdaddr*进行访问。 * *start\_address* 处传递给 **\_beginthreadex** 的例程必须使用 **\_\_stdcall** （对于本机代码）或 **\_\_clrcall** （对于托管代码）调用约定，并且必须返回线程退出代码。 * 如果失败，**\_beginthreadex** 会返回 0，而不是 -1L。 * 使用 **\_beginthreadex** 创建的线程已通过对 [\_endthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex?view=msvc-170) 的调用终止。   与 **\_beginthreadex** 相比， **\_beginthread** 让你可以在更大程度上控制如何创建线程。 **\_endthreadex** 函数也更为灵活。 例如，通过 **\_beginthreadex**，你可以使用安全信息、设置线程的初始状态（运行或挂起）并获取新创建线程的线程标识符。 你还可以将 **\_beginthreadex** 返回的线程句柄与同步 API 结合使用，但无法通过 **\_beginthread** 完成此操作。  使用 **\_beginthreadex** 比 **\_beginthread** 更安全。 如果由 **\_beginthread** 生成的线程很快退出，则返回到 **\_beginthread** 调用方的句柄可能无效或指向另一个线程。 但是，由 **\_beginthreadex** 返回的句柄必须由 **\_beginthreadex** 的调用方关闭，因此如果 **\_beginthreadex** 未返回任何错误，则可以保证其为有效句柄。  可以显式调用 [\_endthread](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex?view=msvc-170) 或 **\_endthreadex** 终止线程；但是，当线程从作为参数传递的例程中返回时，会自动调用 **\_endthread** 或 **\_endthreadex**。 通过对 **\_endthread** 或 **\_endthreadex** 的调用来终止线程有助于确保正确恢复为线程分配的资源。  **\_endthread** 会自动关闭线程句柄，而 **\_endthreadex** 则不会。 因此，当你使用 **\_beginthread** 和 **\_endthread** 时，不要通过调用 Win32 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/win32/api/handleapi/nf-handleapi-closehandle) API 来显式关闭线程句柄。 该行为与 Win32 [ExitThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitthread) API 不同。  **备注**  对于与 Libcmt.lib 链接的可执行文件，请不要调用 Win32 ExitThread API，这样就不会阻止运行时系统回收已分配的资源。 **\_endthread** 和 **\_endthreadex** 回收分配的线程资源，然后调用 ExitThread。  当调用了 **\_beginthread** 或 **\_beginthreadex** 中的任一个时，操作系统将处理堆栈的分配；你不必将该线程堆栈的地址传递给这两个函数中的任何一个。 此外， *stack\_size* 参数还可为 0，在这种情况下，操作系统使用的值与为主线程指定的堆栈相同。  *arglist* 是传递到新创建的线程的参数。 它通常是数据项的地址，例如字符串。 *arglist* 在不需要时可以为 NULL，但必须对 **\_beginthread** 和 **\_beginthreadex** 赋值才能传递到新线程。 如果任何线程调用 [abort](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/abort?view=msvc-170)、 **exit**、 **\_exit**或 ExitProcess，所有线程都会终止。  新线程的区域设置通过使用每进程全局当前区域设置信息来初始化。 如果通过对 [\_configthreadlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/configthreadlocale?view=msvc-170) 的调用（全局或仅针对新线程）启用了每线程区域设置，则线程可以通过调用 **setlocale** 或 **\_wsetlocale** 独立从其他线程更改区域设置。 没有设置每线程区域设置标志的线程可能会影响所有其他线程（包括也未设置每线程区域设置标志的线程，以及所有新创建的线程）中的区域设置信息。 有关详细信息，请参阅 [Locale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/locale?view=msvc-170)。  对于 **/clr** 代码，**\_beginthread** 和 **\_beginthreadex** 都有两个重载。 一个采用本机调用约定函数指针，另一个采用 **\_\_clrcall** 函数指针。 第一个重载不是应用程序域安全的且永远不会是。 如果要编写 **/clr** 代码，则必须确保新线程在访问受管理资源之前进入正确的应用程序域。 例如，可以使用 [call\_in\_appdomain](https://learn.microsoft.com/zh-cn/cpp/dotnet/call-in-appdomain-function?view=msvc-170) 来完成此操作。 第二个重载是应用程序安全域；新创建的线程总是在 **\_beginthread** 或 **\_beginthreadex**调用方的应用程序域中结束。  默认情况下，此函数的全局状态范围限定为应用程序。 若要更改此行为，请参阅 [CRT 中的全局状态](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/global-state?view=msvc-170)。 要求 展开表   | **例程** | **必需的标头** | | --- | --- | | **\_beginthread** | <process.h> | | **\_beginthreadex** | <process.h> |   有关兼容性的详细信息，请参阅 [兼容性](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/compatibility?view=msvc-170)。 库 仅限 [C 运行库](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-library-features?view=msvc-170) 的多线程版本。  若要使用 **\_beginthread** 或 **\_beginthreadex**，应用程序必须与一个多线程 C 运行库链接。 示例 下面的示例使用 **\_beginthread** 和 **\_endthread**。  C复制  // crt\_BEGTHRD.C  // compile with: /MT /D "\_X86\_" /c  // processor: x86  #include <windows.h>  #include <process.h> /\* \_beginthread, \_endthread \*/  #include <stddef.h>  #include <stdlib.h>  #include <conio.h>  void Bounce( void \* );  void CheckKey( void \* );  // GetRandom returns a random integer between min and max.  #define GetRandom( min, max ) ((rand() % (int)(((max) + 1) - (min))) + (min))  // GetGlyph returns a printable ASCII character value  #define GetGlyph( val ) ((char)((val + 32) % 93 + 33))  BOOL repeat = TRUE; // Global repeat flag  HANDLE hStdOut; // Handle for console window  CONSOLE\_SCREEN\_BUFFER\_INFO csbi; // Console information structure  int main()  {  int param = 0;  int \* pparam = &param;  // Get display screen's text row and column information.  hStdOut = GetStdHandle( STD\_OUTPUT\_HANDLE );  GetConsoleScreenBufferInfo( hStdOut, &csbi );  // Launch CheckKey thread to check for terminating keystroke.  \_beginthread( CheckKey, 0, NULL );  // Loop until CheckKey terminates program or 1000 threads created.  while( repeat && param < 1000 )  {  // launch another character thread.  \_beginthread( Bounce, 0, (void \*) pparam );  // increment the thread parameter  param++;  // Wait one second between loops.  Sleep( 1000L );  }  }  // CheckKey - Thread to wait for a keystroke, then clear repeat flag.  void CheckKey( void \* ignored )  {  \_getch();  repeat = 0; // \_endthread implied  }  // Bounce - Thread to create and control a colored letter that moves  // around on the screen.  //  // Params: parg - the value to create the character from  void Bounce( void \* parg )  {  char blankcell = 0x20;  CHAR\_INFO ci;  COORD oldcoord, cellsize, origin;  DWORD result;  SMALL\_RECT region;  cellsize.X = cellsize.Y = 1;  origin.X = origin.Y = 0;  // Generate location, letter and color attribute from thread argument.  srand( \_threadid );  oldcoord.X = region.Left = region.Right =  GetRandom(csbi.srWindow.Left, csbi.srWindow.Right - 1);  oldcoord.Y = region.Top = region.Bottom =  GetRandom(csbi.srWindow.Top, csbi.srWindow.Bottom - 1);  ci.Char.AsciiChar = GetGlyph(\*((int \*)parg));  ci.Attributes = GetRandom(1, 15);  while (repeat)  {  // Pause between loops.  Sleep( 100L );  // Blank out our old position on the screen, and draw new letter.  WriteConsoleOutputCharacterA(hStdOut, &blankcell, 1, oldcoord, &result);  WriteConsoleOutputA(hStdOut, &ci, cellsize, origin, &region);  // Increment the coordinate for next placement of the block.  oldcoord.X = region.Left;  oldcoord.Y = region.Top;  region.Left = region.Right += GetRandom(-1, 1);  region.Top = region.Bottom += GetRandom(-1, 1);  // Correct placement (and beep) if about to go off the screen.  if (region.Left < csbi.srWindow.Left)  region.Left = region.Right = csbi.srWindow.Left + 1;  else if (region.Right >= csbi.srWindow.Right)  region.Left = region.Right = csbi.srWindow.Right - 2;  else if (region.Top < csbi.srWindow.Top)  region.Top = region.Bottom = csbi.srWindow.Top + 1;  else if (region.Bottom >= csbi.srWindow.Bottom)  region.Top = region.Bottom = csbi.srWindow.Bottom - 2;  // If not at a screen border, continue, otherwise beep.  else  continue;  Beep((ci.Char.AsciiChar - 'A') \* 100, 175);  }  // \_endthread given to terminate  \_endthread();  }  按任意键结束示例应用程序。  下面的代码示例演示如何使用由具有同步 API [WaitForSingleObject](https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject) 的 **\_beginthreadex** 返回的线程句柄。 主线程需等待第二个线程终止才能继续。 当第二个线程调用 **\_endthreadex** 时，它会导致其线程对象进入信号状态，这会允许主线程继续运行。 它不能通过 **\_beginthread** 和 **\_endthread** 来完成，因为 **\_endthread** 会调用 CloseHandle，这会在线程对象可以设为信号状态之前销毁它。  C++复制  // crt\_begthrdex.cpp  // compile with: /MT  #include <windows.h>  #include <stdio.h>  #include <process.h>  unsigned Counter;  unsigned \_\_stdcall SecondThreadFunc( void\* pArguments )  {  printf( "In second thread...\n" );  while ( Counter < 1000000 )  Counter++;  \_endthreadex( 0 );  return 0;  }  int main()  {  HANDLE hThread;  unsigned threadID;  printf( "Creating second thread...\n" );  // Create the second thread.  hThread = (HANDLE)\_beginthreadex( NULL, 0, &SecondThreadFunc, NULL, 0, &threadID );  // Wait until second thread terminates. If you comment out the line  // below, Counter will not be correct because the thread has not  // terminated, and Counter most likely has not been incremented to  // 1000000 yet.  WaitForSingleObject( hThread, INFINITE );  printf( "Counter should be 1000000; it is-> %d\n", Counter );  // Destroy the thread object.  CloseHandle( hThread );  }  Output复制  Creating second thread...  In second thread...  Counter should be 1000000; it is-> 1000000 另请参阅  * [进程和环境控制](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/process-and-environment-control?view=msvc-170) * [abort](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/abort?view=msvc-170) * [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread) |

## CreateRemoteThread函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 创建在另一个进程的虚拟地址空间中运行的线程。  使用 [CreateRemoteThreadEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethreadex) 函数创建在另一个进程的虚拟地址空间中运行的线程，并选择性地指定扩展属性。 语法 C++复制  HANDLE CreateRemoteThread(  [in] HANDLE hProcess,  [in] LPSECURITY\_ATTRIBUTES lpThreadAttributes,  [in] SIZE\_T dwStackSize,  [in] LPTHREAD\_START\_ROUTINE lpStartAddress,  [in] LPVOID lpParameter,  [in] DWORD dwCreationFlags,  [out] LPDWORD lpThreadId  ); 参数 [in] hProcess  要在其中创建线程的进程句柄。 句柄必须具有 **PROCESS\_CREATE\_THREAD**、 **PROCESS\_QUERY\_INFORMATION**、 **PROCESS\_VM\_OPERATION**、 **PROCESS\_VM\_WRITE**和 **PROCESS\_VM\_READ** 访问权限，并且在某些平台上没有这些权限可能会失败。 有关详细信息，请参阅 [进程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-security-and-access-rights)。  [in] 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，该结构指定新线程的安全描述符，并确定子进程是否可以继承返回的句柄。 如果 *lpThreadAttributes* 为 NULL，则线程将获取默认安全描述符，并且无法继承句柄。 访问控制在来自创建者主令牌的线程的默认安全描述符中列出 (ACL) 。  **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] dwStackSize  堆栈的初始大小（以字节为单位）。 系统将此值舍入到最近的页。 如果此参数为 0 (零) ，则新线程将使用可执行文件的默认大小。 有关详细信息，请参阅 [线程堆栈大小](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-stack-size)。  [in] lpStartAddress  指向 **LPTHREAD\_START\_ROUTINE要由** 线程执行的应用程序定义函数的指针，表示远程进程中线程的起始地址。 函数必须存在于远程进程中。 有关详细信息，请参阅 [ThreadProc](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/ms686736(v=vs.85))。  [in] lpParameter  指向要传递给线程函数的变量的指针。  [in] dwCreationFlags  控制线程创建的标志。  展开表   |  |  | | --- | --- | | **值** | **含义** | | 0 | 线程在创建后立即运行。 | | **CREATE\_SUSPENDED**  0x00000004 | 线程以挂起状态创建，在调用 [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread) 函数之前不会运行。 | | **STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION**  0x00010000 | *dwStackSize* 参数指定堆栈的初始保留大小。 如果未指定此标志， *dwStackSize* 将指定提交大小。 |   [out] lpThreadId  指向接收线程标识符的变量的指针。  如果此参数为 **NULL**，则不返回线程标识符。 返回值 如果函数成功，则返回值是新线程的句柄。  如果函数失败，则返回值为 NULL。 要获得更多的错误信息，请调用 GetLastError。  请注意，即使 *lpStartAddress* 指向数据、代码或不可访问，**CreateRemoteThread** 也可能成功。 如果启动地址在线程运行时无效，则会发生异常，并且线程终止。 由于启动地址无效而导致的线程终止作为线程进程的错误退出进行处理。 此行为类似于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 的异步性质，即即使进程引用无效或缺少动态链接库，也会创建 (DLL) 。 注解 **CreateRemoteThread** 函数会导致新的执行线程在指定进程的地址空间中开始。 线程有权访问进程打开的所有对象。  在Windows 8之前，终端服务根据设计隔离每个终端会话。 因此，如果目标进程与调用进程位于不同的会话中， **CreateRemoteThread** 将失败。  创建新线程句柄时具有对新线程的完全访问权限。 如果未提供安全描述符，则句柄可用于任何需要线程对象句柄的函数中。 提供安全描述符时，在授予访问权限之前，对句柄的所有后续使用执行访问检查。 如果访问检查拒绝访问，则请求进程无法使用句柄获取对线程的访问权限。  如果线程是在可运行状态下创建的， (也就是说，如果未) 使用 **CREATE\_SUSPENDED** 标志，则线程可以在 [CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread) 返回之前开始运行，尤其是在调用方收到所创建线程的句柄和标识符之前。  创建的线程优先级为 **THREAD\_PRIORITY\_NORMAL**。 使用 [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority) 和 [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority) 函数获取和设置线程的优先级值。  当线程终止时，线程对象会获得一个信号状态，该状态满足正在等待该对象的线程。  线程对象将保留在系统中，直到线程终止，并且通过调用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭其所有句柄。  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)、[ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread)、[CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread)、**CreateRemoteThread** 函数以及由于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 调用 () 而启动的进程在进程中相互序列化。 一次只有其中一个事件在地址空间中发生。 这意味着存在以下限制：   * 在进程启动和 DLL 初始化例程期间，可以创建新线程，但在为进程完成 DLL 初始化之前，这些线程不会开始执行。 * 一个进程中一个线程一次只能位于 DLL 初始化或分离例程中。 * [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess) 在所有线程完成其 DLL 初始化或分离例程后返回。   此函数的常见用途是将线程注入正在调试以发出中断的进程。 但是，不建议使用此用法，因为额外的线程会让调试应用程序的人员感到困惑，并且使用此技术有几个副作用：   * 它将单线程应用程序转换为多线程应用程序。 * 它会更改进程的计时和内存布局。 * 它会导致调用进程中每个 DLL 的入口点。   此函数的另一个常见用途是将线程注入进程以查询堆或其他进程信息。 这可能会导致上一段落中提到的相同副作用。 此外，如果线程尝试获取另一个线程正在使用的锁的所有权，应用程序可能会死锁。 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)  [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)  [CreateRemoteThreadEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethreadex)  [CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread)  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread)  [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority)  [ThreadProc](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/ms686736(v=vs.85))  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## ExitThread函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 结束调用线程。 语法 C++  void ExitThread(  [in] DWORD dwExitCode  ); parameters [in] dwExitCode  线程的退出代码。 返回值 无 备注 **ExitThread** 是在 C 代码中退出线程的首选方法。 但是，在 C++ 代码中，线程先退出，然后才能调用任何析构函数或执行任何其他自动清理。 因此，在 C++ 代码中，应从线程函数返回 。  当显式 (或通过从线程过程) 返回来调用此函数时，将解除分配当前线程的堆栈，取消由未与完成端口关联的线程启动的所有挂起 I/O，并且线程将终止。 调用所有附加动态链接库的入口点函数 (DLL) ，其值指示线程正在与 DLL 分离。  如果调用此函数时线程是进程中的最后一个线程，则线程的进程也会终止。  线程对象的状态将发出信号，释放已等待线程终止的任何其他线程。 线程的终止状态从 STILL\_ACTIVE 更改为 *dwExitCode* 参数的值。  终止线程不一定从操作系统中删除线程对象。 关闭线程的最后一个句柄时，将删除线程对象。  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)、**ExitThread**、[CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread)、[CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread) 函数以及由于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 调用 () 而启动的进程在进程中相互序列化。 一次只能在地址空间中发生其中一个事件。 这意味着存在以下限制：   * 在进程启动和 DLL 初始化例程期间，可以创建新线程，但在为进程完成 DLL 初始化之前，这些线程不会开始执行。 * 一个进程中一个线程一次只能位于 DLL 初始化或分离例程中。 * [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess) 在 DLL 初始化或分离例程中没有线程之前不会返回。   链接到静态 C 运行时库 (CRT) 的可执行文件中的线程应使用 **\_beginthread** 和 **\_endthread** 进行线程管理，而不是 **CreateThread** 和 **ExitThread**。 否则，当线程调用 **ExitThread** 时，会导致小内存泄漏。 另一个解决方法是将可执行文件链接到 DLL 中的 CRT，而不是静态 CRT。 请注意，仅当 DLL 链接到静态 CRT 且线程调用 [DisableThreadLibraryCalls](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-disablethreadlibrarycalls) 函数时，才会从 DLL 发生此内存泄漏。 否则，从链接到静态 CRT 的 DLL 中的线程调用 **CreateThread** 和 **ExitThread** 是安全的。  使用 [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodethread) 函数检索线程的退出代码。  **Windows Phone 8.1：**Windows Phone 8.1 及更高版本上的 Windows Phone 应用商店应用支持此函数。  **Windows 8.1**和 **Windows Server 2012 R2**：Windows 8.1、Windows Server 2012 R2 及更高版本的 Windows 应用商店应用支持此函数。 示例 有关示例，请参阅 [使用事件对象](https://learn.microsoft.com/zh-cn/windows/desktop/Sync/using-event-objects)。 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;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  请参阅 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)  [CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread)  [CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread)  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [FreeLibraryAndExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/libloaderapi/nf-libloaderapi-freelibraryandexitthread)  [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodethread)  [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [TerminateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-terminatethread)  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## GetExitCodeThread函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 检索指定线程的终止状态。 语法 C++  BOOL GetExitCodeThread(  [in] HANDLE hThread,  [out] LPDWORD lpExitCode  ); 参数 [in] hThread  线程的句柄。  句柄必须具有 **THREAD\_QUERY\_INFORMATION** 或 **THREAD\_QUERY\_LIMITED\_INFORMATION** 访问权限。 有关详细信息，请参阅 [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights)。  **Windows Server 2003 和 Windows XP：**句柄必须具有 **THREAD\_QUERY\_INFORMATION** 访问权限。  [out] lpExitCode  指向用于接收线程终止状态的变量的指针。 有关详细信息，请参阅“备注”。 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 此函数立即返回。 如果指定的线程未终止且函数成功，则返回的状态 **为STILL\_ACTIVE**。 如果线程已终止且函数成功，则返回的状态为以下值之一：   * [ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread) 或 [TerminateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-terminatethread) 函数中指定的退出值。 * 线程函数的返回值。 * 线程进程的退出值。   **重要GetExitCodeThread** 函数仅在线程终止后返回由应用程序定义的有效错误代码。 因此，应用程序不应使用 **STILL\_ACTIVE** (259) 作为错误代码。 如果线程返回 **STILL\_ACTIVE** (259) 作为错误代码，则测试此值的应用程序可能会将其解释为表示线程仍在运行，并在线程终止后继续测试线程完成情况，这可能会使应用程序进入无限循环。 为避免此问题，调用方应仅在确认线程退出后调用 **GetExitCodeThread** 函数。 使用等待持续时间为零的 [**WaitForSingleObject**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject) 函数来确定线程是否已退出。    **Windows Phone 8.1：**Windows Phone 8.1 及更高版本上的 Windows Phone 应用商店应用支持此函数。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 8.1、Windows Server 2012 R2 及更高版本的 Windows 应用商店应用支持此函数。 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;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [ExitThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitthread)  [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)  [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [TerminateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-terminatethread)  [终止线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/terminating-a-thread) |

## GetThreadPriority函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 检索指定线程的优先级值。 此值与线程进程的优先级类一起确定线程的基本优先级级别。 语法 C++复制  int GetThreadPriority(  [in] HANDLE hThread  ); 参数 [in] hThread  线程的句柄。  句柄必须具有 **THREAD\_QUERY\_INFORMATION** 或 **THREAD\_QUERY\_LIMITED\_INFORMATION** 访问权限。 有关详细信息，请参阅 [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights)。  **Windows Server 2003：**句柄必须具有 **THREAD\_QUERY\_INFORMATION** 访问权限。 返回值 如果函数成功，则返回值为线程的优先级。  如果函数失败，则 **返回值THREAD\_PRIORITY\_ERROR\_RETURN**。 要获得更多的错误信息，请调用 GetLastError。  **Windows Phone 8.1：**此函数将始终返回**THREAD\_PRIORITY\_NORMAL**。  线程的优先级是以下值之一。  展开表   |  |  | | --- | --- | | **返回代码/值** | **说明** | | **THREAD\_PRIORITY\_ABOVE\_NORMAL**  1 | 优先级高于优先级类 1 磅。 | | **THREAD\_PRIORITY\_BELOW\_NORMAL**  -1 | 优先级低于优先级类 1 磅。 | | **THREAD\_PRIORITY\_HIGHEST**  2 | 优先级高于优先级类 2 磅。 | | **THREAD\_PRIORITY\_IDLE**  -15 | **IDLE\_PRIORITY\_CLASS**、**BELOW\_NORMAL\_PRIORITY\_CLASS、NORMAL\_PRIORITY\_CLASS、ABOVE\_NORMAL\_PRIORITY\_CLASS**或HIGH\_PRIORITY\_CLASS进程的基本优先级**为** 1，REALTIME\_PRIORITY\_CLASS进程的基本优先级为 16。 | | **THREAD\_PRIORITY\_LOWEST**  -2 | 优先级低于优先级类 2 磅。 | | **THREAD\_PRIORITY\_NORMAL**  0 | 优先级类的正常优先级。 | | **THREAD\_PRIORITY\_TIME\_CRITICAL**  15 | 对于**IDLE\_PRIORITY\_CLASS、BELOW\_NORMAL\_PRIORITY\_CLASS**、**NORMAL\_PRIORITY\_CLASS**、**ABOVE\_NORMAL\_PRIORITY\_CLASS**或HIGH\_PRIORITY\_CLASS进程，基本优先级级别**为** 15，REALTIME\_PRIORITY\_CLASS进程的基本优先级级别为 31。 |     如果线程具有 **REALTIME\_PRIORITY\_CLASS** 基类，此函数还可以返回以下值之一：-7、-6、-5、-4、-3、3、4、5 或 6。 有关详细信息，请参阅 [计划优先级](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/scheduling-priorities)。 注解 每个线程都有一个基本优先级级别，由线程的优先级值及其进程的优先级类确定。 操作系统使用所有可执行线程的基优先级级别来确定哪个线程获取下一段 CPU 时间。 线程在每个优先级级别以轮循机制方式进行计划，只有在较高级别没有可执行线程时，才会在较低级别调度线程。  有关显示优先级类和线程优先级值的每种组合的基本优先级的表，请参阅 [SetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setpriorityclass) 函数。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 应用商店应用支持此函数。  **Windows Phone 8.1：**Windows Phone 应用商店应用可以调用此函数，但它不起作用。 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;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [GetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getpriorityclass)  [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [计划优先级](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/scheduling-priorities)  [SetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setpriorityclass)  [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadpriority)  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## ResumeThread函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 递减线程的挂起计数。 当暂停计数减为零时，将恢复线程的执行。 语法 C++  DWORD ResumeThread(  [in] HANDLE hThread  ); 参数 [in] hThread  要重启的线程的句柄。  此句柄必须具有THREAD\_SUSPEND\_RESUME访问权限。 有关详细信息，请参阅 [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights)。 返回值 如果函数成功，则返回值是线程的上一个挂起计数。  如果函数失败，则返回值 (DWORD) -1。 要获得更多的错误信息，请调用 GetLastError。 注解 **ResumeThread** 函数检查主题线程的挂起计数。 如果暂停计数为零，则线程当前未挂起。 否则，主题线程的挂起计数将递减。 如果生成的值为零，则继续执行主题线程。  如果返回值为零，则表示指定的线程未挂起。 如果返回值为 1，则表示指定的线程已暂停，但已重新启动。 如果返回值大于 1，则指定的线程仍挂起。  请注意，报告调试事件时，报告进程中的所有线程都会被冻结。 调试器应使用 [SuspendThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-suspendthread) 和 **ResumeThread** 函数来限制可在进程中执行的线程集。 通过暂停进程中的所有线程（报告调试事件的线程除外），可以“单步执行”单个线程。 如果其他线程被挂起，则继续操作不会释放这些线程。  **Windows Phone 8.1：**Windows Phone 8.1 及更高版本上的 Windows Phone 应用商店应用支持此函数。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 8.1、Windows Server 2012 R2 及更高版本的 Windows 应用商店应用支持此函数。 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;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [SuspendThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-suspendthread)  [挂起线程执行](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/suspending-thread-execution)  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## SetThreadPriority函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 设置指定线程的优先级值。 此值与线程进程的优先级类一起确定线程的基本优先级级别。 语法 C++复制  BOOL SetThreadPriority(  [in] HANDLE hThread,  [in] int nPriority  ); 参数 [in] hThread  要设置其优先级值的线程的句柄。  句柄必须具有 **THREAD\_SET\_INFORMATION** 或 **THREAD\_SET\_LIMITED\_INFORMATION** 访问权限。 有关详细信息，请参阅 [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights)。**Windows Server 2003：**句柄必须具有 **THREAD\_SET\_INFORMATION** 访问权限。  [in] nPriority  线程的优先级值。 此参数的取值可为下列值之一：  展开表   |  |  | | --- | --- | | **优先级** | **含义** | | **THREAD\_MODE\_BACKGROUND\_BEGIN**  0x00010000 | 开始后台处理模式。 系统降低了线程的资源计划优先级，以便它可以执行后台工作，而不会显著影响前台的活动。  仅当 *hThread* 是当前线程的句柄时，才能指定此值。 如果线程已处于后台处理模式，则函数将失败。  **Windows Server 2003：**不支持此值。 | | **THREAD\_MODE\_BACKGROUND\_END**  0x00020000 | 结束后台处理模式。 系统还原线程的资源计划优先级，就像线程进入后台处理模式之前一样。  仅当 *hThread* 是当前线程的句柄时，才能指定此值。 如果线程不处于后台处理模式，函数将失败。  **Windows Server 2003：**不支持此值。 | | **THREAD\_PRIORITY\_ABOVE\_NORMAL**  1 | 优先级高于优先级类 1 磅。 | | **THREAD\_PRIORITY\_BELOW\_NORMAL**  -1 | 优先级低于优先级类 1 磅。 | | **THREAD\_PRIORITY\_HIGHEST**  2 | 优先级高于优先级类 2 磅。 | | **THREAD\_PRIORITY\_IDLE**  -15 | **IDLE\_PRIORITY\_CLASS**、**BELOW\_NORMAL\_PRIORITY\_CLASS、NORMAL\_PRIORITY\_CLASS、ABOVE\_NORMAL\_PRIORITY\_CLASS**或HIGH\_PRIORITY\_CLASS进程的基本优先级**为** 1，REALTIME\_PRIORITY\_CLASS进程的基本优先级为 16。 | | **THREAD\_PRIORITY\_LOWEST**  -2 | 优先级低于优先级类 2 磅。 | | **THREAD\_PRIORITY\_NORMAL**  0 | 优先级类的正常优先级。 | | **THREAD\_PRIORITY\_TIME\_CRITICAL**  15 | **IDLE\_PRIORITY\_CLASS**、**BELOW\_NORMAL\_PRIORITY\_CLASS、NORMAL\_PRIORITY\_CLASS**、**ABOVE\_NORMAL\_PRIORITY\_CLASS**或HIGH\_PRIORITY\_CLASS进程的基本优先级**为** 15，REALTIME\_PRIORITY\_CLASS进程的基本优先级**为** 31。 |     如果线程具有 **REALTIME\_PRIORITY\_CLASS** 基类，则此参数也可以是 -7、-6、-5、-4、-3、3、4、5 或 6。 有关详细信息，请参阅 [计划优先级](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/scheduling-priorities)。 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。  **Windows Phone 8.1：**Windows Phone应用商店应用可以调用此函数，但它不起作用。 函数将返回指示成功的非零值。 注解 每个线程都有由线程的优先级值及其进程的优先级类确定的基本优先级。 系统使用所有可执行线程的基本优先级级别来确定哪个线程获取下一段 CPU 时间。 线程在每个优先级级别以轮循机制方式进行计划，并且仅当较高级别没有可执行线程时，才会在较低级别调度线程。  **SetThreadPriority** 函数允许设置线程相对于其进程的优先级类的基本优先级。 例如，在调用 **IDLE\_PRIORITY\_CLASS** 进程的线程的 **SetThreadPriority** 中指定 **THREAD\_PRIORITY\_HIGHEST** 会将线程的基本优先级级别设置为 6。 有关显示优先级类和线程优先级值的每种组合的基本优先级的表，请参阅 [计划优先级](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/scheduling-priorities)。  对于 **IDLE\_PRIORITY\_CLASS**、 **BELOW\_NORMAL\_PRIORITY\_CLASS**、 **NORMAL\_PRIORITY\_CLASS**、 **ABOVE\_NORMAL\_PRIORITY\_CLASS**和 **HIGH\_PRIORITY\_CLASS** 进程，当发生对线程很重要的事件时，系统会动态提升线程的基本优先级。 **REALTIME\_PRIORITY\_CLASS** 进程不会收到动态提升。  所有线程最初从 **THREAD\_PRIORITY\_NORMAL**开始。 使用 [GetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getpriorityclass) 和 [SetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setpriorityclass) 函数获取和设置进程的优先级类。 使用 [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority) 函数获取线程的优先级值。  使用流程的优先级类来区分时间关键型应用程序与具有正常或低于正常计划要求的应用程序。 使用线程优先级值来区分进程任务的相对优先级。 例如，处理窗口输入的线程的优先级可能高于对 CPU 执行密集计算的线程。  处理优先级时，请非常小心，确保高优先级线程不会占用所有可用的 CPU 时间。 基本优先级高于 11 的线程会干扰操作系统的正常运行。 使用 **REALTIME\_PRIORITY\_CLASS** 可能会导致磁盘缓存无法刷新、鼠标停止响应等。  **THREAD\_PRIORITY\_\*** 值会影响线程的 CPU 计划优先级。 对于执行后台工作（如文件 I/O、网络 I/O 或数据处理）的线程，调整 CPU 计划优先级是不够的;即使空闲 CPU 优先级线程在使用磁盘和内存时也很容易干扰系统的响应能力。 执行后台工作的线程应使用 **THREAD\_MODE\_BACKGROUND\_BEGIN** 和 **THREAD\_MODE\_BACKGROUND\_END** 值来调整其资源计划优先级;与用户交互的线程不应使用 **THREAD\_MODE\_BACKGROUND\_BEGIN**。  当线程处于后台处理模式时，它应尽量减少与进程中的其他线程共享资源（例如关键部分、堆和句柄），否则可能会发生优先级反转。 如果有线程以高优先级执行，则后台处理模式下的线程可能不会立即计划，但永远不会耗尽。  **Windows Server 2008 和 Windows Vista：**当系统启动时， **SetThreadPriority** 函数返回成功返回值，但不会更改从系统启动文件夹启动或 **HKEY\_LOCAL\_MACHINE**\**SOFTWARE**\**Microsoft**\**Windows**\**CurrentVersion**\**Run** 注册表项中列出的应用程序的线程优先级。 这些应用程序在短时间内以降低优先级运行 (大约 60 秒) ，以使系统在启动期间对用户操作做出更快速的响应。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 应用商店应用支持此函数。  **Windows Phone 8.1：**Windows Phone 应用商店应用可以调用此函数，但它不起作用。 示例 以下示例演示如何使用线程后台模式。  C++复制  #include <windows.h>  #include <tchar.h>  int main( void )  {  DWORD dwError, dwThreadPri;  if(!SetThreadPriority(GetCurrentThread(), THREAD\_MODE\_BACKGROUND\_BEGIN))  {  dwError = GetLastError();  if( ERROR\_THREAD\_MODE\_ALREADY\_BACKGROUND == dwError)  \_tprintf(TEXT("Already in background mode\n"));  else \_tprintf(TEXT("Failed to enter background mode (%d)\n"), dwError);  goto Cleanup;  }  // Display thread priority  dwThreadPri = GetThreadPriority(GetCurrentThread());  \_tprintf(TEXT("Current thread priority is 0x%x\n"), dwThreadPri);  //  // Perform background work  //  ;  if(!SetThreadPriority(GetCurrentThread(), THREAD\_MODE\_BACKGROUND\_END))  {  \_tprintf(TEXT("Failed to end background mode (%d)\n"), GetLastError());  }  Cleanup:  // Clean up  ;  return 0;  } 要求  |  |  | | --- | --- | | **标头** | processthreadsapi.h (包括 Windows Vista 上的 Windows.h、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [GetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getpriorityclass)  [GetThreadPriority](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getthreadpriority)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [计划优先级](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/scheduling-priorities)  [SetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setpriorityclass)  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## SuspendThread函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 挂起指定的线程。  64 位应用程序可以使用 [Wow64SuspendThread 函数挂起 WOW64](https://learn.microsoft.com/zh-cn/windows/win32/api/wow64apiset/nf-wow64apiset-wow64suspendthread) 线程。 语法 C++  DWORD SuspendThread(  [in] HANDLE hThread  ); 参数 [in] hThread  要挂起的线程的句柄。  句柄必须具有 **THREAD\_SUSPEND\_RESUME** 访问权限。 有关详细信息，请参阅 [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights)。 返回值 如果函数成功，则返回值为线程的上一个挂起计数;否则为 (DWORD) -1。 若要获取扩展的错误信息，请使用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数。 注解 如果函数成功，则挂起指定线程的执行，线程的挂起计数将递增。 挂起线程会导致线程停止执行用户模式 (应用程序) 代码。  此函数主要用于调试器。 它不用于线程同步。 如果调用线程尝试获取暂停线程拥有的同步对象，在拥有同步对象的线程（如互斥体或关键部分）上调用 **SuspendThread** 可能会导致死锁。 为避免这种情况，应用程序中不是调试器的线程应向另一个线程发出信号，以暂停自身。 目标线程必须设计为watch此信号并做出适当的响应。  每个线程都有一个暂停计数 (，其最大值为 **MAXIMUM\_SUSPEND\_COUNT**) 。 如果挂起计数大于零，则暂停线程;否则，线程不会挂起，并且符合执行条件。 调用 **SuspendThread** 会导致目标线程的挂起计数递增。 尝试递增超过最大暂停计数会导致错误而不递增计数。  [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread) 函数会减少挂起线程的挂起计数。  **Windows Phone 8.1：**Windows Phone 8.1 及更高版本上的 Windows Phone 应用商店应用支持此函数。  **Windows 8.1**和**Windows Server 2012 R2**：Windows 8.1、Windows Server 2012 R2 及更高版本的 Windows 应用商店应用支持此函数。 要求  |  |  | | --- | --- | | **标头** | processthreadsapi.h (包括 Windows Vista 上的 Windows.h、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib;Windows Phone 8.1 上的 WindowsPhoneCore.lib | | **DLL** | Kernel32.dll;Windows Phone 8.1 上的 KernelBase.dll |  另请参阅 [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [ResumeThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-resumethread)  [挂起线程执行](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/suspending-thread-execution)  [线程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/multiple-threads) |

## ThreadProc函数

|  |
| --- |
| 创建线程 [**CreateThread**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread) 函数为进程创建新线程。 创建线程必须指定新线程要执行的代码的起始地址。 通常，起始地址是在程序代码 (中定义的函数的名称，有关详细信息，请参阅 [ThreadProc](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/ms686736(v=vs.85))) 。 此函数采用单个参数并返回 **DWORD** 值。 一个进程可以有多个线程同时执行同一函数。  下面是一个简单的示例，演示如何创建执行本地定义的函数 MyThreadFunction的新线程。  调用线程使用 [**WaitForMultipleObjects**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects) 函数一直存在，直到所有工作线程都终止。 调用线程在等待时阻止;若要继续处理，调用线程将使用 [**WaitForSingleObject**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject) 并等待每个工作线程发出其等待对象的信号。 请注意，如果要在终止工作线程之前关闭该工作线程的句柄，则不会终止工作线程。 但是，句柄将不能用于后续函数调用。  C++  #include <windows.h>  #include <tchar.h>  #include <strsafe.h>  #define MAX\_THREADS 3  #define BUF\_SIZE 255  DWORD WINAPI MyThreadFunction( LPVOID lpParam );  void ErrorHandler(LPCTSTR lpszFunction);  // Sample custom data structure for threads to use.  // This is passed by void pointer so it can be any data type  // that can be passed using a single void pointer (LPVOID).  typedef struct MyData {  int val1;  int val2;  } MYDATA, \*PMYDATA;  int \_tmain()  {  PMYDATA pDataArray[MAX\_THREADS];  DWORD dwThreadIdArray[MAX\_THREADS];  HANDLE hThreadArray[MAX\_THREADS];  // Create MAX\_THREADS worker threads.  for( int i=0; i<MAX\_THREADS; i++ )  {  // Allocate memory for thread data.  pDataArray[i] = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP\_ZERO\_MEMORY,  sizeof(MYDATA));  if( pDataArray[i] == NULL )  {  // If the array allocation fails, the system is out of memory  // so there is no point in trying to print an error message.  // Just terminate execution.  ExitProcess(2);  }  // Generate unique data for each thread to work with.  pDataArray[i]->val1 = i;  pDataArray[i]->val2 = i+100;  // Create the thread to begin execution on its own.  hThreadArray[i] = CreateThread(  NULL, // default security attributes  0, // use default stack size  MyThreadFunction, // thread function name  pDataArray[i], // argument to thread function  0, // use default creation flags  &dwThreadIdArray[i]); // returns the thread identifier  // Check the return value for success.  // If CreateThread fails, terminate execution.  // This will automatically clean up threads and memory.  if (hThreadArray[i] == NULL)  {  ErrorHandler(TEXT("CreateThread"));  ExitProcess(3);  }  } // End of main thread creation loop.  // Wait until all threads have terminated.  WaitForMultipleObjects(MAX\_THREADS, hThreadArray, TRUE, INFINITE);  // Close all thread handles and free memory allocations.  for(int i=0; i<MAX\_THREADS; i++)  {  CloseHandle(hThreadArray[i]);  if(pDataArray[i] != NULL)  {  HeapFree(GetProcessHeap(), 0, pDataArray[i]);  pDataArray[i] = NULL; // Ensure address is not reused.  }  }  return 0;  }  DWORD WINAPI MyThreadFunction( LPVOID lpParam )  {  HANDLE hStdout;  PMYDATA pDataArray;  TCHAR msgBuf[BUF\_SIZE];  size\_t cchStringSize;  DWORD dwChars;  // Make sure there is a console to receive output results.  hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);  if( hStdout == INVALID\_HANDLE\_VALUE )  return 1;  // Cast the parameter to the correct data type.  // The pointer is known to be valid because  // it was checked for NULL before the thread was created.    pDataArray = (PMYDATA)lpParam;  // Print the parameter values using thread-safe functions.  StringCchPrintf(msgBuf, BUF\_SIZE, TEXT("Parameters = %d, %d\n"),  pDataArray->val1, pDataArray->val2);  StringCchLength(msgBuf, BUF\_SIZE, &cchStringSize);  WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize, &dwChars, NULL);  return 0;  }  void ErrorHandler(LPCTSTR lpszFunction)  {  // Retrieve the system error message for the last-error code.  LPVOID lpMsgBuf;  LPVOID lpDisplayBuf;  DWORD dw = GetLastError();  FormatMessage(  FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |  FORMAT\_MESSAGE\_FROM\_SYSTEM |  FORMAT\_MESSAGE\_IGNORE\_INSERTS,  NULL,  dw,  MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),  (LPTSTR) &lpMsgBuf,  0, NULL );  // Display the error message.  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM\_ZEROINIT,  (lstrlen((LPCTSTR) lpMsgBuf) + lstrlen((LPCTSTR) lpszFunction) + 40) \* sizeof(TCHAR));  StringCchPrintf((LPTSTR)lpDisplayBuf,  LocalSize(lpDisplayBuf) / sizeof(TCHAR),  TEXT("%s failed with error %d: %s"),  lpszFunction, dw, lpMsgBuf);  MessageBox(NULL, (LPCTSTR) lpDisplayBuf, TEXT("Error"), MB\_OK);  // Free error-handling buffer allocations.  LocalFree(lpMsgBuf);  LocalFree(lpDisplayBuf);  }  函数 MyThreadFunction 避免使用 C 运行时库 (CRT) ，因为它的许多函数不是线程安全的，尤其是在不使用多线程 CRT 时。 如果要在函数中使用 ThreadProc CRT，请改用 **\_beginthreadex** 函数。  如果创建线程在新线程之前退出，则传递本地变量的地址是有风险的，因为指针将变为无效。 而是将指针传递到动态分配的内存，或者使创建线程等待新线程终止。 还可以使用全局变量将数据从创建线程传递到新线程。 使用全局变量时，通常需要同步多个线程的访问。 有关同步的详细信息，请参阅 [同步多线程的执行](https://learn.microsoft.com/zh-cn/windows/win32/procthread/synchronizing-execution-of-multiple-threads)。  创建线程可以使用 [**CreateThread**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread) 参数来指定以下内容：   * 新线程句柄的安全属性。 这些安全属性包括一个继承标志，用于确定是否可由子进程继承句柄。 安全属性还包括安全描述符，系统使用该描述符在授予访问权限之前对线程句柄的所有后续使用执行访问检查。 * 新线程的初始堆栈大小。 线程的堆栈在进程的内存空间中自动分配;系统根据需要增加堆栈，并在线程终止时释放堆栈。 有关详细信息，请参阅 [线程堆栈大小](https://learn.microsoft.com/zh-cn/windows/win32/procthread/thread-stack-size)。 * 一个创建标志，可用于创建处于挂起状态的线程。 挂起后，线程在调用 [**ResumeThread**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread) 函数之前不会运行。   还可以通过调用 [**CreateRemoteThread**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createremotethread) 函数来创建线程。 调试器进程使用此函数来创建在所调试进程的地址空间中运行的线程。 |

## SECURITY\_ATTRIBUTES结构体

|  |  |  |
| --- | --- | --- |
| **SECURITY\_ATTRIBUTES** 结构包含对象的[安全描述符](https://learn.microsoft.com/zh-cn/windows/win32/api/winnt/ns-winnt-security_descriptor)，并指定通过指定此结构检索的句柄是否可继承。 此结构为由各种函数（如 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-createfilew)、 [**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe)、 [**CreateProcess**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa)、 [**RegCreateKeyEx**](https://learn.microsoft.com/zh-cn/windows/win32/api/winreg/nf-winreg-regcreatekeyexw) 或 [**RegSaveKeyEx**](https://learn.microsoft.com/zh-cn/windows/win32/api/winreg/nf-winreg-regsavekeyexa)）创建的对象提供安全设置。 语法 C++复制  typedef struct \_SECURITY\_ATTRIBUTES {  DWORD nLength;  LPVOID lpSecurityDescriptor;  BOOL bInheritHandle;  } SECURITY\_ATTRIBUTES, \*PSECURITY\_ATTRIBUTES, \*LPSECURITY\_ATTRIBUTES; 成员 nLength  此结构的大小（以字节为单位）。 将此值设置为 **SECURITY\_ATTRIBUTES** 结构的大小。  lpSecurityDescriptor  指向 [**SECURITY\_DESCRIPTOR**](https://learn.microsoft.com/zh-cn/windows/win32/api/winnt/ns-winnt-security_descriptor) 结构的指针，该结构控制对 对象的访问。 如果此成员的值为 **NULL**，则为对象分配与调用进程的 [访问令牌](https://learn.microsoft.com/zh-cn/windows/win32/secauthz/access-tokens) 关联的默认安全描述符。 这与通过分配 **NULL**[自由访问控制列表](https://learn.microsoft.com/zh-cn/windows/win32/secauthz/dacls-and-aces) (DACL) 向所有人授予访问权限不同。 默认情况下，进程的访问令牌中的默认 DACL 仅允许访问令牌表示的用户。  有关创建安全描述符的信息，请参阅 [创建安全描述符](https://learn.microsoft.com/zh-cn/windows/win32/secauthz/creating-a-security-descriptor-for-a-new-object-in-c--)。  bInheritHandle  一个布尔值，指定是否在创建新进程时继承返回的句柄。 如果此成员为 **TRUE**，则新进程将继承句柄。 要求  |  |  | | --- | --- | | **标头** | wtypesbase.h | |

## 一个线程，其实就是一个函数，也就是ThreadProc类型的函数，也就是把一个函数作为一个线程来运行。

# 演练

## 1.新建一个win32项目，取名Lesson53-multithread

|  |
| --- |
|  |

## 2.找到WndProc函数,添加对WM\_CREATE消息的响应处理代码，在里面创建一个新线程，这里使用\_beginthread函数来创建线程，需要包含头文件<process.h>,还要声明并且定义一个线程执行函数ThreadProc。我们在WndProc函数外面定义这个函数。我们需要创建2个线程，因此需要定义两个线程函数

|  |
| --- |
|  |

## 3.然后我们在WM\_CRATE消息处理代码里面创建一个线程

|  |
| --- |
|  |

## 4.然后我们添加第一个线程函数的代码，注意需要把InitInstance函数里面的窗口句柄改为全局变量。还需要响应WM\_SIZE消息来保存客户区的宽度和高度

|  |
| --- |
|  |
|  |
|  |
| VOID MyThreadProc1(PVOID pvoid)  {  //随机绘制矩形  HBRUSH hbrush,hbrushOrg;  HDC hdc;  hdc = GetDC(hWnd);  int l,r,t,b;//矩形的左上角，右下角坐标  int iRed,iGreen,iBlue;  srand(GetTickCount());//设置随机数种子  while(TRUE)  {  if(cxClient !=0 && cyClient!=0)  {  //随机生成左上角，右下角坐标,  l = rand()%cxClient;  t = rand()%cyClient;  r = rand()%cxClient;  b = rand()%cyClient;  //随机生成红绿蓝颜色分量  iRed = rand()%255;  iGreen = rand()%255;  iBlue= rand()%255;  hbrush = CreateSolidBrush(RGB(iRed,iGreen,iBlue));  hbrushOrg = (HBRUSH)SelectObject(hdc,hbrush);  //因为坐标是随机生成的有可能r比l小，我们取小的坐标作为左上角，同理取大的坐标作为右下角  Rectangle(hdc,min(l,r),min(t,b),max(l,r),max(t,b));  SelectObject(hdc,hbrushOrg);  DeleteObject(hbrush);  Sleep(300);//太快了，让他睡眠一会，如300毫秒  }  }  ReleaseDC(hWnd,hdc);  } |

## 5.然后我们可以把这个函数的内容复制粘贴到另外一个线程函数中，把Rectangle函数改为Ellipse函数，这样子，第二个线程函数就做好了

|  |
| --- |
| VOID MyThreadProc2(PVOID pvoid)  {  //随机绘制椭圆  HBRUSH hbrush,hbrushOrg;  HDC hdc;  hdc = GetDC(hWnd);  int l,r,t,b;//矩形的左上角，右下角坐标  int iRed,iGreen,iBlue;  srand(GetTickCount());//设置随机数种子  while(TRUE)  {  if(cxClient !=0 || cyClient!=0)  {  //随机生成左上角，右下角坐标,  l = rand()%cxClient;  t = rand()%cyClient;  r = rand()%cxClient;  b = rand()%cyClient;  //随机生成红绿蓝颜色分量  iRed = rand()%255;  iGreen = rand()%255;  iBlue= rand()%255;  hbrush = CreateSolidBrush(RGB(iRed,iGreen,iBlue));  hbrushOrg = (HBRUSH)SelectObject(hdc,hbrush);  //因为坐标是随机生成的有可能r比l小，我们取小的坐标作为左上角，同理取大的坐标作为右下角  Ellipse(hdc,min(l,r),min(t,b),max(l,r),max(t,b));  SelectObject(hdc,hbrushOrg);  DeleteObject(hbrush);  Sleep(300);//太快了，让他睡眠一会，如300毫秒  }  }  ReleaseDC(hWnd,hdc);  } |

## 6.然后我们创建第二个线程

|  |
| --- |
|  |

### 这样，一个多线程的程序就做好了。

## 扩展，我们创建2个写日志的函数，每一个函数把日志写到各自的log文件中

|  |
| --- |
| VOID LoggingThreadProc1(PVOID pvoid)  {  HANDLE file1;  TCHAR szMsg[MAX\_PATH];  DWORD szWritten;  memset(szMsg,0,MAX\_PATH);  time\_t curTime;  curTime = time(0);  struct tm\* ptm;  ptm = gmtime(&curTime);  wsprintf(szMsg,TEXT("[%d-%d-%d %#02d:%#02d:%#02d]Thread1 working...\r\n"),1900+ptm->tm\_year,1+ptm->tm\_mon,ptm->tm\_mday,  ptm->tm\_hour-3,ptm->tm\_min,ptm->tm\_sec);  file1 = CreateFile(TEXT("log1.txt"),GENERIC\_WRITE,FILE\_SHARE\_READ,NULL,CREATE\_ALWAYS,FILE\_ATTRIBUTE\_NORMAL,NULL);  if(INVALID\_HANDLE\_VALUE == file1)  {  MessageBox(hWnd,\_T("创建文件失败"),\_T("错误"),0);  return;  }  while(TRUE)  {  if(!WriteFile(file1,szMsg,MAX\_PATH,&szWritten,NULL))  {  MessageBox(hWnd,\_T("写入文件失败"),\_T("错误"),0);  CloseHandle(file1);  break;  }  Sleep(1000);  }  CloseHandle(file1);  } |
| VOID LoggingThreadProc2(PVOID pvoid)  {  HANDLE file1;  TCHAR szMsg[MAX\_PATH];  DWORD szWritten;  memset(szMsg,0,MAX\_PATH);  time\_t curTime;  curTime = time(0);  struct tm\* ptm;  ptm = gmtime(&curTime);  wsprintf(szMsg,TEXT("[%d-%d-%d %#02d:%#02d:%#02d]Thread2 working...\r\n"),1900+ptm->tm\_year,1+ptm->tm\_mon,ptm->tm\_mday,  ptm->tm\_hour-3,ptm->tm\_min,ptm->tm\_sec);  file1 = CreateFile(TEXT("log2.txt"),GENERIC\_WRITE,FILE\_SHARE\_READ,NULL,CREATE\_ALWAYS,FILE\_ATTRIBUTE\_NORMAL,NULL);  if(INVALID\_HANDLE\_VALUE == file1)  {  MessageBox(hWnd,\_T("创建文件失败"),\_T("错误"),0);  return;  }  while(TRUE)  {  if(!WriteFile(file1,szMsg,MAX\_PATH,&szWritten,NULL))  {  MessageBox(hWnd,\_T("写入文件失败"),\_T("错误"),0);  CloseHandle(file1);  break;  }  Sleep(2000);  }  CloseHandle(file1);  } |

## 也是可以正常工作的，但是为了有点视觉效果，我们还是改为绘图线程

# 这一节的学习到此为止，完整代码如下

## Lesson53-multithread.cpp

|  |
| --- |
| // Lesson53-multithread.cpp : 定义应用程序的入口点。  //  #include "stdafx.h"  #include "Lesson53-multithread.h"  #include <process.h>  #include<time.h>  #include<stdio.h>  #define MAX\_LOADSTRING 100  // 全局变量:  HINSTANCE hInst; // 当前实例  TCHAR szTitle[MAX\_LOADSTRING]; // 标题栏文本  TCHAR szWindowClass[MAX\_LOADSTRING]; // 主窗口类名  HWND hWnd;//全局窗口句柄  int cxClient,cyClient;//保存客户区宽度和高度的全局变量  // 此代码模块中包含的函数的前向声明:  ATOM MyRegisterClass(HINSTANCE hInstance);  BOOL InitInstance(HINSTANCE, int);  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  INT\_PTR CALLBACK About(HWND, UINT, WPARAM, LPARAM);  int APIENTRY \_tWinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,  LPTSTR lpCmdLine,  int nCmdShow)  {  UNREFERENCED\_PARAMETER(hPrevInstance);  UNREFERENCED\_PARAMETER(lpCmdLine);  // TODO: 在此放置代码。  MSG msg;  HACCEL hAccelTable;  // 初始化全局字符串  LoadString(hInstance, IDS\_APP\_TITLE, szTitle, MAX\_LOADSTRING);  LoadString(hInstance, IDC\_LESSON53MULTITHREAD, szWindowClass, MAX\_LOADSTRING);  MyRegisterClass(hInstance);  // 执行应用程序初始化:  if (!InitInstance (hInstance, nCmdShow))  {  return FALSE;  }  hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC\_LESSON53MULTITHREAD));  // 主消息循环:  while (GetMessage(&msg, NULL, 0, 0))  {  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))  {  TranslateMessage(&msg);  DispatchMessage(&msg);  }  }  return (int) msg.wParam;  }  //  // 函数: MyRegisterClass()  //  // 目的: 注册窗口类。  //  // 注释:  //  // 仅当希望  // 此代码与添加到 Windows 95 中的“RegisterClassEx”  // 函数之前的 Win32 系统兼容时，才需要此函数及其用法。调用此函数十分重要，  // 这样应用程序就可以获得关联的  // “格式正确的”小图标。  //  ATOM MyRegisterClass(HINSTANCE hInstance)  {  WNDCLASSEX wcex;  wcex.cbSize = sizeof(WNDCLASSEX);  wcex.style = CS\_HREDRAW | CS\_VREDRAW;  wcex.lpfnWndProc = WndProc;  wcex.cbClsExtra = 0;  wcex.cbWndExtra = 0;  wcex.hInstance = hInstance;  wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI\_LESSON53MULTITHREAD));  wcex.hCursor = LoadCursor(NULL, IDC\_ARROW);  wcex.hbrBackground = (HBRUSH)(COLOR\_WINDOW+1);  wcex.lpszMenuName = MAKEINTRESOURCE(IDC\_LESSON53MULTITHREAD);  wcex.lpszClassName = szWindowClass;  wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI\_SMALL));  return RegisterClassEx(&wcex);  }  //  // 函数: InitInstance(HINSTANCE, int)  //  // 目的: 保存实例句柄并创建主窗口  //  // 注释:  //  // 在此函数中，我们在全局变量中保存实例句柄并  // 创建和显示主程序窗口。  //  BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)  {  //HWND hWnd; //在外面设置为全局变量  hInst = hInstance; // 将实例句柄存储在全局变量中  hWnd = CreateWindow(szWindowClass, szTitle, WS\_OVERLAPPEDWINDOW,  CW\_USEDEFAULT, 0, CW\_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);  if (!hWnd)  {  return FALSE;  }  ShowWindow(hWnd, nCmdShow);  UpdateWindow(hWnd);  return TRUE;  }  //定义线程函数  VOID MyThreadProc1(PVOID pvoid)  {  //随机绘制矩形  HBRUSH hbrush,hbrushOrg;  HDC hdc;  hdc = GetDC(hWnd);  int l,r,t,b;//矩形的左上角，右下角坐标  int iRed,iGreen,iBlue;  srand(GetTickCount());//设置随机数种子  while(TRUE)  {  if(cxClient !=0 || cyClient!=0)  {  //随机生成左上角，右下角坐标,  l = rand()%cxClient;  t = rand()%cyClient;  r = rand()%cxClient;  b = rand()%cyClient;  //随机生成红绿蓝颜色分量  iRed = rand()%255;  iGreen = rand()%255;  iBlue= rand()%255;  hbrush = CreateSolidBrush(RGB(iRed,iGreen,iBlue));  hbrushOrg = (HBRUSH)SelectObject(hdc,hbrush);  //因为坐标是随机生成的有可能r比l小，我们取小的坐标作为左上角，同理取大的坐标作为右下角  Rectangle(hdc,min(l,r),min(t,b),max(l,r),max(t,b));  SelectObject(hdc,hbrushOrg);  DeleteObject(hbrush);  Sleep(500);//太快了，让他睡眠一会，如300毫秒  }  }  ReleaseDC(hWnd,hdc);  }  VOID MyThreadProc2(PVOID pvoid)  {  //随机绘制椭圆  HBRUSH hbrush,hbrushOrg;  HDC hdc;  hdc = GetDC(hWnd);  int l,r,t,b;//矩形的左上角，右下角坐标  int iRed,iGreen,iBlue;  srand(GetTickCount());//设置随机数种子  while(TRUE)  {  if(cxClient !=0 || cyClient!=0)  {  //随机生成左上角，右下角坐标,  l = rand()%cxClient;  t = rand()%cyClient;  r = rand()%cxClient;  b = rand()%cyClient;  //随机生成红绿蓝颜色分量  iRed = rand()%255;  iGreen = rand()%255;  iBlue= rand()%255;  hbrush = CreateSolidBrush(RGB(iRed,iGreen,iBlue));  hbrushOrg = (HBRUSH)SelectObject(hdc,hbrush);  //因为坐标是随机生成的有可能r比l小，我们取小的坐标作为左上角，同理取大的坐标作为右下角  Ellipse(hdc,min(l,r),min(t,b),max(l,r),max(t,b));  SelectObject(hdc,hbrushOrg);  DeleteObject(hbrush);  Sleep(300);//太快了，让他睡眠一会，如300毫秒  }  }  ReleaseDC(hWnd,hdc);  }  VOID LoggingThreadProc1(PVOID pvoid)  {  HANDLE file1;  TCHAR szMsg[MAX\_PATH];  DWORD szWritten;  memset(szMsg,0,MAX\_PATH);  time\_t curTime;  curTime = time(0);  struct tm\* ptm;  ptm = gmtime(&curTime);  wsprintf(szMsg,TEXT("[%d-%d-%d %#02d:%#02d:%#02d]Thread1 working...\r\n"),1900+ptm->tm\_year,1+ptm->tm\_mon,ptm->tm\_mday,  ptm->tm\_hour-3,ptm->tm\_min,ptm->tm\_sec);  file1 = CreateFile(TEXT("log1.txt"),GENERIC\_WRITE,FILE\_SHARE\_READ,NULL,CREATE\_ALWAYS,FILE\_ATTRIBUTE\_NORMAL,NULL);  if(INVALID\_HANDLE\_VALUE == file1)  {  MessageBox(hWnd,\_T("创建文件失败"),\_T("错误"),0);  return;  }  while(TRUE)  {  if(!WriteFile(file1,szMsg,MAX\_PATH,&szWritten,NULL))  {  MessageBox(hWnd,\_T("写入文件失败"),\_T("错误"),0);  CloseHandle(file1);  break;  }  Sleep(1000);  }  CloseHandle(file1);  }  VOID LoggingThreadProc2(PVOID pvoid)  {  HANDLE file1;  TCHAR szMsg[MAX\_PATH];  DWORD szWritten;  memset(szMsg,0,MAX\_PATH);  time\_t curTime;  curTime = time(0);  struct tm\* ptm;  ptm = gmtime(&curTime);  wsprintf(szMsg,TEXT("[%d-%d-%d %#02d:%#02d:%#02d]Thread2 working...\r\n"),1900+ptm->tm\_year,1+ptm->tm\_mon,ptm->tm\_mday,  ptm->tm\_hour-3,ptm->tm\_min,ptm->tm\_sec);  file1 = CreateFile(TEXT("log2.txt"),GENERIC\_WRITE,FILE\_SHARE\_READ,NULL,CREATE\_ALWAYS,FILE\_ATTRIBUTE\_NORMAL,NULL);  if(INVALID\_HANDLE\_VALUE == file1)  {  MessageBox(hWnd,\_T("创建文件失败"),\_T("错误"),0);  return;  }  while(TRUE)  {  if(!WriteFile(file1,szMsg,MAX\_PATH,&szWritten,NULL))  {  MessageBox(hWnd,\_T("写入文件失败"),\_T("错误"),0);  CloseHandle(file1);  break;  }  Sleep(2000);  }  CloseHandle(file1);  }  //  // 函数: WndProc(HWND, UINT, WPARAM, LPARAM)  //  // 目的: 处理主窗口的消息。  //  // WM\_COMMAND - 处理应用程序菜单  // WM\_PAINT - 绘制主窗口  // WM\_DESTROY - 发送退出消息并返回  //  //  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  {  int wmId, wmEvent;  PAINTSTRUCT ps;  HDC hdc;    switch (message)  {  case WM\_CREATE:  //创建新线程。  //第二个参数为0表示由windows来分配，第3个参数其实是线程函数的参数，我们的线程函数没有参数，所以这里是NULL  \_beginthread(MyThreadProc1,0,NULL);  \_beginthread(MyThreadProc2,0,NULL);  /\*\_beginthread(LoggingThreadProc1,0,NULL);  \_beginthread(LoggingThreadProc2,0,NULL);\*/  break;  case WM\_SIZE:  cxClient = LOWORD(lParam);  cyClient = HIWORD(lParam);  break;  case WM\_COMMAND:  wmId = LOWORD(wParam);  wmEvent = HIWORD(wParam);  // 分析菜单选择:  switch (wmId)  {  case IDM\_ABOUT:  DialogBox(hInst, MAKEINTRESOURCE(IDD\_ABOUTBOX), hWnd, About);  break;  case IDM\_EXIT:  DestroyWindow(hWnd);  break;  default:  return DefWindowProc(hWnd, message, wParam, lParam);  }  break;  case WM\_PAINT:  hdc = BeginPaint(hWnd, &ps);  // TODO: 在此添加任意绘图代码...  EndPaint(hWnd, &ps);  break;  case WM\_DESTROY:  PostQuitMessage(0);  break;  default:  return DefWindowProc(hWnd, message, wParam, lParam);  }  return 0;  }  // “关于”框的消息处理程序。  INT\_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)  {  UNREFERENCED\_PARAMETER(lParam);  switch (message)  {  case WM\_INITDIALOG:  return (INT\_PTR)TRUE;  case WM\_COMMAND:  if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)  {  EndDialog(hDlg, LOWORD(wParam));  return (INT\_PTR)TRUE;  }  break;  }  return (INT\_PTR)FALSE;  } |