**使用 C 和 Win32 进行多线程编程**

|  |
| --- |
| Microsoft C/C++ 编译器 (MSVC) 提供对创建多线程应用程序的支持。 如果应用程序需要执行会导致用户界面无响应的高开销操作，请考虑使用多个线程。  借助 MSVC，有通过多种方式使用多线程进行编程：可以使用 C++/WinRT 和 Windows 运行时库、Microsoft 基础类 (MFC) 库、C++/CLI 和 .NET 运行时，或 C 运行时库和 Win32 API。 本文介绍 C 中的多线程。有关示例代码，请参阅 [C 中的示例多线程程序](https://learn.microsoft.com/zh-cn/cpp/parallel/sample-multithread-c-program?view=msvc-170)。 多线程程序 简单而言，线程是通过程序的执行路径。 它也是 Win32 调度的最小执行单位。 线程由堆栈、CPU 寄存器状态和系统调度程序执行列表中的条目组成。 每个线程共享进程的所有资源。  一个进程由一个或多个线程，以及内存中程序的代码、数据和其他资源组成。 典型的程序资源是打开的文件、信号灯和动态分配的内存。 当系统调度程序为某个程序的一个线程授予执行控制权时，该程序就会执行。 调度程序确定哪些线程应该运行以及何时运行。 较低优先级的线程可能必须等待较高优先级的线程完成其任务。 在多处理器计算机上，调度程序可将各个线程移到不同的处理器以平衡 CPU 负载。  进程中的每个线程都是独立运行的。 除非使它们彼此可见，否则线程将单独执行，并且不知道进程中的其他线程。 但是，共享公用资源的线程必须使用信号灯或其他进程间通信方法来协调其工作。 有关同步线程的详细信息，请参阅[编写多线程 Win32 程序](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-c-and-win32?view=msvc-170#writing-a-multithreaded-win32-program)。 多线程编程的库支持 所有版本的 CRT 现在都支持多线程，但某些功能的非锁定版本除外。 有关详细信息，请参阅[多线程库性能](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/multithreaded-libraries-performance?view=msvc-170)。 有关可与代码链接的 CRT 版本的信息，请参阅 [CRT 库功能](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/crt-library-features?view=msvc-170)。 多线程编程的包含文件 标准 CRT include 文件声明在库中实现的 C 运行时库函数。 如果编译器选项指定 [\_\_fastcall 或 \_\_vectorcall](https://learn.microsoft.com/zh-cn/cpp/build/reference/gd-gr-gv-gz-calling-convention?view=msvc-170) 调用约定，则编译器假定应使用寄存器调用约定来调用所有函数。 运行时库函数使用 C 调用约定，标准 include 文件中的声明告知编译器生成对这些函数的正确外部引用。 用于线程控制的 CRT 函数 所有 Win32 程序都至少有一个线程。 任何线程可以创建额外的线程。 线程可以快速完成其工作然后终止，或者可以在程序的生命周期内保持活动状态。  CRT 库为线程创建和终止提供以下函数：[\_beginthread、\_beginthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/beginthread-beginthreadex?view=msvc-170)、[\_endthread 和 \_endthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex?view=msvc-170)。  \_beginthread 和 \_beginthreadex 函数创建一个新线程并在操作成功时返回一个线程标识符。 如果线程完成执行，则它会自动终止。 或者，它可以通过调用 \_endthread 或 \_endthreadex 自行终止。  **备注**  如果从使用 libcmt.lib 生成的程序调用 C 运行时例程，则必须使用 \_beginthread 或 \_beginthreadex 函数启动线程。 不要使用 Win32 函数 ExitThread 和 CreateThread。 当多个线程阻塞以等待暂停的线程完成对 C 运行时数据结构的访问时，使用 SuspendThread 会导致死锁。 \_beginthread 和 \_beginthreadex 函数 \_beginthread 和 \_beginthreadex 函数创建一个新线程。 一个线程与进程中的其他线程共享该进程的代码和数据段，但具有自身独特的寄存器值、堆栈空间和当前指令地址。 系统将 CPU 时间分配到每个线程，以便进程中的所有线程可以并发执行。  \_beginthread 和 \_beginthreadex 类似于 Win32 API 中的 [CreateThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread) 函数，但存在以下差别：   * 它们初始化特定的 C 运行时库变量。 仅当你在线程中使用 C 运行时库时，这一点才很重要。 * CreateThread 帮助提供对安全属性的控制。 可以使用此函数来启动处于暂停状态的线程。   如果成功，\_beginthread 和 \_beginthreadex 将返回新线程的句柄；如果出错，则返回错误代码。 \_endthread 和 \_endthreadex 函数 [\_endthread](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex?view=msvc-170) 函数终止 \_beginthread 创建的线程（类似地，\_endthreadex 终止 \_beginthreadex 创建的线程）。 线程在完成时会自动终止。 \_endthread 和 \_endthreadex 可用于从线程内部进行条件终止。 例如，如果专用于通信处理的线程无法控制通信端口，则可能会退出。 编写多线程 Win32 程序 编写具有多个线程的程序时，必须协调它们的行为以及[程序资源的使用](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-c-and-win32?view=msvc-170#_core_sharing_common_resources_between_threads)。 此外，请确保每个线程接收[其自身的堆栈](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-c-and-win32?view=msvc-170#_core_thread_stacks)。 在线程之间共享公用资源 **备注**  有关从 MFC 角度展开的类似讨论，请参阅[**多线程：编程提示**](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170)和[**多线程：何时使用同步类**](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-when-to-use-the-synchronization-classes?view=msvc-170)。  每个线程有自身的堆栈和自身的 CPU 寄存器副本。 其他资源（例如文件、静态数据和堆内存）由进程中的所有线程共享。 使用这些公用资源的线程必须同步。 Win32 提供了多种同步资源的方式，包括信号灯、临界区、事件和互斥。  当多个线程访问静态数据时，程序必须为可能的资源冲突做好准备。 假设程序中的一个线程更新某个静态数据结构，该结构包含另一个线程显示的项的 x,y 坐标。 如果更新线程更改了 x 坐标，但在可以更改 y 坐标之前被抢占，则显示线程可能会调度在 y 坐标更新之前。 该项将显示在错误的位置。 要避免此问题，可以使用信号灯来控制对该结构的访问。  互斥（“互相排斥”的简写）是一种在彼此异步执行的线程或进程之间进行通信的方式。 这种通信可用于协调多个线程或进程的活动，协调方式通常是通过锁定和解锁资源来控制对共享资源的访问。 为了解决这种 x,y 坐标更新问题，更新线程在执行更新之前会设置一个互斥，指示数据结构已被使用。 在处理完这两个坐标后，它将清除互斥。 在更新显示画面之前，显示线程必须等待清除互斥。 这种等待互斥的过程通常称为互斥阻塞，因为进程已阻塞，在清除互斥之前无法继续。  [示例多线程 C 程序](https://learn.microsoft.com/zh-cn/cpp/parallel/sample-multithread-c-program?view=msvc-170)中所示的 Bounce.c 程序使用名为 ScreenMutex 的互斥来协调屏幕更新。 每当某个显示线程准备好写入屏幕时，它就会结合 ScreenMutex 的句柄和常量 INFINITE 来调用 WaitForSingleObject，以指示 WaitForSingleObject 调用应阻塞互斥而不是超时。如果 ScreenMutex 已清除，则 wait 函数将设置互斥，使其他线程不会干扰显示，并继续执行线程。 否则，在清除互斥之前，线程一直会阻塞。 当线程完成显示画面更新时，它会通过调用 ReleaseMutex 来释放互斥。  屏幕显示画面和静态数据不过是需要认真管理的资源中的两种。 例如，你的程序可能有多个线程访问同一个文件。 由于另一个线程可能已移动文件指针，因此每个线程必须在读取或写入之前重置文件指针。 此外，每个线程必须确保它在定位指针之后、访问文件之前的这段时间不会被抢占。 这些线程应使用信号灯来协调对文件的访问，方法是使用 WaitForSingleObject 和 ReleaseMutex 调用并将每个文件访问括住 |

# 实例1

|  |
| --- |
| HANDLE hIOMutex = CreateMutex (NULL, FALSE, NULL);  WaitForSingleObject( hIOMutex, INFINITE );  fseek( fp, desired\_position, 0L );  fwrite( data, sizeof( data ), 1, fp );  ReleaseMutex( hIOMutex); |

### 线程堆栈

应用程序的所有默认堆栈空间将分配到第一个执行线程，称为线程 1。 因此，必须指定要为程序所需的每个附加线程的单独堆栈分配多少内存。 如有必要，操作系统会为线程分配附加的堆栈空间，但你需要指定默认值。

\_beginthread 调用中的第一个参数是指向 BounceProc 函数的指针，该函数执行线程。 第二个参数指定线程的默认堆栈大小。 最后一个参数是传递给 BounceProc 的 ID 编号。 BounceProc 使用该 ID 编号来播种随机数生成器，并选择线程的颜色属性和显示字符。

调用 C 运行时库或 Win32 API 的线程必须为它们调用的库和 API 函数留出足够的堆栈空间。 C printf 函数需要 500 字节以上的堆栈空间，调用 Win32 API 例程时应有 2K 字节的可用堆栈空间。

由于每个线程具有自身的堆栈，因此你可以使用尽可能少的静态数据来避免潜在的数据项冲突。 将程序设计为对可专用于线程的所有数据使用自动堆栈变量。 Bounce.c 程序中的全局变量只有互斥，或者初始化后永不更改的变量。

Win32 还提供线程本地存储 (TLS) 来存储每个线程的数据。 有关详细信息，请参阅[线程本地存储 (TLS)](https://learn.microsoft.com/zh-cn/cpp/parallel/thread-local-storage-tls?view=msvc-170)。

## 避免与多线程程序有关的问题

在创建、链接或执行多线程 C 程序时你可能会遇到多种问题。 下表描述了一些较常见问题。 （有关从 MFC 角度展开的类似讨论，请参阅[多线程：编程提示](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170)。）

展开表

| **问题** | **可能的原因** |
| --- | --- |
| 出现一个消息框，指出程序导致了保护冲突。 | 有许多 Win32 编程错误会导致保护冲突。 保护冲突的一个常见原因是间接将数据分配到了 null 指针。 因为这会导致程序尝试访问不属于它的内存，因此会发生保护冲突。  若要检测保护冲突的原因，一种简单方法是使用调试信息编译程序，然后在 Visual Studio 环境中通过调试器运行该程序。 发生保护故障时，Windows 会将控制权转移给调试器，并且光标将定位在导致问题的行上。 |
| 程序生成大量的编译和链接错误。 | 可以通过将编译器的警告级别设置为其最高值之一并留意警告消息来消除许多潜在问题。 使用级别 3 或级别 4 警告级别选项，可以检测无意中进行了数据转换、缺少函数原型和使用了非 ANSI 功能的问题。 |

## 另请参阅

[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)  
[C 中的示例多线程程序](https://learn.microsoft.com/zh-cn/cpp/parallel/sample-multithread-c-program?view=msvc-170)  
[线程本地存储 (TLS)](https://learn.microsoft.com/zh-cn/cpp/parallel/thread-local-storage-tls?view=msvc-170)  
[利用 C++/WinRT 实现的并发和异步操作](https://learn.microsoft.com/zh-cn/windows/uwp/cpp-and-winrt-apis/concurrency)  
[使用 C++ 和 MFC 进行多线程编程](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-cpp-and-mfc?view=msvc-170)

# 多线程 C 程序示例

|  |
| --- |
| Bounce.c 是一个多线程程序示例，它在每次键入字母 a 或 A 时都会创建一个新线程。 每个线程在屏幕周围退回一个不同颜色的字母。 最多可以创建 32 个线程。 当键入 q 或 Q 时，程序正常终止。 编译并链接多线程程序 默认情况下，程序被编译为多线程。 在开发环境中编译和链接多线程程序 Bounce.c  1. 在“文件”菜单上，选择“新建”>“项目” 。 2. 在“创建新项目”对话框中，选择具有“C++”、“Windows”和“控制台”标记的“控制台应用”模板。 选择“下一步” 继续。 3. 在“配置新项目”对话框中，输入项目的名称，例如“Bounce”。 选择“创建”以继续。 4. 在“解决方案资源管理器”窗口中，打开项目下的“源文件”文件夹，并将源文件的名称更改为 .c 扩展名。 5. 在“编辑”窗口中，删除现有源代码并将其替换为示例代码。 6. 在 **“生成”** 菜单上，选择 **“生成解决方案”** 。 7. 按 F5 以在调试器中启动程序。  从命令行编译并链接多线程程序 Bounce.c  1. 打开 Visual Studio Tools 命令提示符。 这确保路径被设置为包含编译器。 2. 编译并链接程序：   Windows 命令提示符复制  cl bounce.c |

## 示例

若要在命令行上生成，请将此示例复制并保存到扩展名为 .c 的源文件中。 在 IDE 中，将模板创建的任何源代码替换为以下示例：

|  |
| --- |
| // sample\_multithread\_c\_program.c  // compile with: /c  //  // Bounce - Creates a new thread each time the letter 'a' is typed.  // Each thread bounces a character of a different color around  // the screen. All threads are terminated when the letter 'Q' is  // entered.  //  #include <windows.h>  #include <stdlib.h>  #include <string.h>  #include <stdio.h>  #include <conio.h>  #include <process.h>  #define MAX\_THREADS 32  // The function getrandom returns a random number between  // min and max, which must be in integer range.  #define getrandom( min, max ) (SHORT)((rand() % (int)(((max) + 1) - \  (min))) + (min))  int main(void); // Thread 1: main  void KbdFunc(void); // Keyboard input, thread dispatch  void BounceProc(void\* pMyID); // Threads 2 to n: display  void ClearScreen(void); // Screen clear  void ShutDown(void); // Program shutdown  void WriteTitle(int ThreadNum); // Display title bar information  HANDLE hConsoleOut; // Handle to the console  HANDLE hRunMutex; // "Keep Running" mutex  HANDLE hScreenMutex; // "Screen update" mutex  int ThreadNr = 0; // Number of threads started  CONSOLE\_SCREEN\_BUFFER\_INFO csbiInfo; // Console information  COORD consoleSize;  BOOL bTrails = FALSE;  HANDLE hThreads[MAX\_THREADS] = { NULL }; // Handles for created threads  int main(void) // Thread One  {  // Get display screen information & clear the screen.  hConsoleOut = GetStdHandle(STD\_OUTPUT\_HANDLE);  GetConsoleScreenBufferInfo(hConsoleOut, &csbiInfo);  consoleSize.X = csbiInfo.srWindow.Right;  consoleSize.Y = csbiInfo.srWindow.Bottom;  ClearScreen();  WriteTitle(0);  // Create the mutexes and reset thread count.  hScreenMutex = CreateMutexW(NULL, FALSE, NULL); // Cleared  hRunMutex = CreateMutexW(NULL, TRUE, NULL); // Set  // Start waiting for keyboard input to dispatch threads or exit.  KbdFunc();  // All threads done. Clean up handles.  if (hScreenMutex) CloseHandle(hScreenMutex);  if (hRunMutex) CloseHandle(hRunMutex);  if (hConsoleOut) CloseHandle(hConsoleOut);  }  void ShutDown(void) // Shut down threads  {  // Tell all threads to die  ReleaseMutex(hRunMutex);  while (ThreadNr > 0)  {  // Wait for each thread to complete  WaitForSingleObject(hThreads[--ThreadNr], INFINITE);  }  // Clean up display when done  WaitForSingleObject(hScreenMutex, INFINITE);  ClearScreen();  }  void KbdFunc(void) // Dispatch and count threads.  {  int KeyInfo;  do  {  KeyInfo = \_getch();  if (tolower(KeyInfo) == 'a' &&  ThreadNr < MAX\_THREADS)  {  ++ThreadNr;  hThreads[ThreadNr] =  (HANDLE)\_beginthread(BounceProc, 0, (void\*)(uintptr\_t)ThreadNr);  WriteTitle(ThreadNr);  }  if (tolower(KeyInfo) == 't')  {  bTrails = !bTrails;  }  } while (tolower(KeyInfo) != 'q');  ShutDown();  }  void BounceProc(void\* pMyID)  {  wchar\_t MyCell, OldCell;  WORD MyAttrib, OldAttrib = 0;  wchar\_t BlankCell = 0x20;  COORD Coords, Delta;  COORD Old = { 0,0 };  DWORD Dummy;  int MyID = (int)(uintptr\_t)pMyID;  // Generate update increments and initial  // display coordinates.  srand(MyID \* 3);  Coords.X = getrandom(0, consoleSize.X - 1);  Coords.Y = getrandom(0, consoleSize.Y - 1);  Delta.X = getrandom(-3, 3);  Delta.Y = getrandom(-3, 3);  // Set up character & generate color  // attribute from thread number.  if (MyID > 16)  MyCell = (wchar\_t)(0x60 + MyID - 16); // lower case  else  MyCell = (wchar\_t)(0x40 + MyID); // upper case  MyAttrib = MyID & 0x0f; // force black background  do  {  // Wait for display to be available, then lock it.  WaitForSingleObject(hScreenMutex, INFINITE);  if (!bTrails)  {  // If we still occupy the old screen position, blank it out.  ReadConsoleOutputCharacterW(hConsoleOut, &OldCell, 1,  Old, &Dummy);  ReadConsoleOutputAttribute(hConsoleOut, &OldAttrib, 1,  Old, &Dummy);  if ((OldCell == MyCell) && (OldAttrib == MyAttrib))  WriteConsoleOutputCharacterW(hConsoleOut, &BlankCell, 1,  Old, &Dummy);  }  // Draw new character, then clear screen lock  WriteConsoleOutputCharacterW(hConsoleOut, &MyCell, 1,  Coords, &Dummy);  WriteConsoleOutputAttribute(hConsoleOut, &MyAttrib, 1,  Coords, &Dummy);  ReleaseMutex(hScreenMutex);  // Increment the coordinates for next placement of the block.  Old.X = Coords.X;  Old.Y = Coords.Y;  Coords.X += Delta.X;  Coords.Y += Delta.Y;  // If we are about to go off the screen, reverse direction  if (Coords.X < 0 || Coords.X >= consoleSize.X)  {  Delta.X = -Delta.X;  Beep(400, 50);  }  if (Coords.Y < 0 || Coords.Y > consoleSize.Y)  {  Delta.Y = -Delta.Y;  Beep(600, 50);  }  }  // Repeat while RunMutex is still taken.  while (WaitForSingleObject(hRunMutex, 75L) == WAIT\_TIMEOUT);  }  void WriteTitle(int ThreadNum)  {  enum  {  sizeOfNThreadMsg = 120  };  wchar\_t NThreadMsg[sizeOfNThreadMsg] = { L"" };  swprintf\_s(NThreadMsg, sizeOfNThreadMsg,  L"Threads running: %02d. Press 'A' "  L"to start a thread, 'T' to toggle "  L"trails, 'Q' to quit.", ThreadNum);  SetConsoleTitleW(NThreadMsg);  }  void ClearScreen(void)  {  DWORD dummy = 0;  COORD Home = { 0, 0 };  FillConsoleOutputCharacterW(hConsoleOut, L' ',  consoleSize.X \* consoleSize.Y,  Home, &dummy);  } |

线程本地存储 (TLS) 是一种方法，给定的多线程进程中的每个线程可以使用这种方法分配用以存储线程特定的数据的位置。 通过 TLS API ([TlsAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-tlsalloc)) 支持动态绑定（运行时）线程特定数据。 Win32 和 Microsoft C++ 编译器现在除了支持现有的 API 实现外，还支持静态绑定（负载时）每线程数据。

# 线程本地存储 (TLS)

## TLS 的编译器实现

C++11：建议使用 **thread\_local** 存储类说明符指定对象和类成员的线程本地存储。 有关详细信息，请参阅[存储类 (C++)](https://learn.microsoft.com/zh-cn/cpp/cpp/storage-classes-cpp?view=msvc-170)。

MSVC 还提供特定于 Microsoft 的特性（[线程](https://learn.microsoft.com/zh-cn/cpp/cpp/thread?view=msvc-170)）作为扩展的存储类修饰符。 可以使用 **\_\_declspec** 关键字声明 **thread** 变量。 例如，以下代码声明了一个整数线程局部变量，并用一个值对其进行初始化：

C复制

\_\_declspec( thread ) int tls\_i = 1;

## 规则和限制

声明静态绑定的线程本地对象和变量时，必须遵守以下准则： 这些准则同时适用于 [thread](https://learn.microsoft.com/zh-cn/cpp/cpp/thread?view=msvc-170) 和 [thread\_local](https://learn.microsoft.com/zh-cn/cpp/cpp/storage-classes-cpp?view=msvc-170)：

* **thread** 特性仅可适用于类以及数据声明和定义。 它不能用于函数声明或定义。 例如，下面的代码生成一个编译器错误：

C复制

\_\_declspec( thread )void func(); // This will generate an error.

* **thread** 修饰符只能在具有 **static** 盘区的数据项上指定。 其中包括全局数据对象（**static** 和 **extern**），本地静态对象以及 C ++ 类的静态数据成员。 不能使用 **thread** 特性声明自动数据对象。 以下代码生成编译器错误：

C复制

void func1()

{

\_\_declspec( thread )int tls\_i; // This will generate an error.

}

int func2(\_\_declspec( thread )int tls\_i ) // This will generate an error.

{

return tls\_i;

}

* 线程本地对象的所有声明和定义必须全部指定 **thread** 特性。 例如，下面的代码将生成错误：

C复制

#define Thread \_\_declspec( thread )

extern int tls\_i; // This will generate an error, since the

int \_\_declspec( thread )tls\_i; // declaration and definition differ.

* **thread** 特性不能用作类型修饰符。 例如，下面的代码生成一个编译器错误：

C复制

char \_\_declspec( thread ) \*ch; // Error

* 由于允许执行使用 **thread** 特性的 C++ 对象的声明，因此以下两个示例在语义上是等效的：

C++复制

\_\_declspec( thread ) class B

{

// Code

} BObject; // OK--BObject is declared thread local.

class B

{

// Code

};

\_\_declspec( thread ) B BObject; // OK--BObject is declared thread local.

* 线程本地对象的地址不视为常数，并且涉及此类地址的任何表达式不会被视为常数表达式。 在标准 C 中，此操作的效果是禁止将线程本地变量的地址用作对象或指针的初始值设定项。 例如，C 编译器会将以下代码标记为错误：

C复制

\_\_declspec( thread ) int tls\_i;

int \*p = &tls\_i; //This will generate an error in C.

此限制不适用于 C++。 由于 C++ 允许所有对象的动态初始化，可以通过利用使用线程本地变量的地址的表达式将对象初始化。 就像完成线程本地对象的构造一样完成此操作。 例如，当编译成 C++ 源文件时，前面所示的代码不生成错误。 仅当从中获取地址的线程仍然存在时，线程本地变量的地址才有效。

* 标准 C 允许使用涉及引用自身的表达式进行对象或变量的初始化，但只适用于非静态范围的对象。 虽然 C++ 通常允许使用涉及引用自身的表达式进行此类的对象动态初始化，但是不允许对线程本地对象进行这种初始化。 例如：

C复制

\_\_declspec( thread )int tls\_i = tls\_i; // Error in C and C++

int j = j; // OK in C++, error in C

\_\_declspec( thread )int tls\_i = sizeof( tls\_i ) // Legal in C and C++

包含正在初始化的对象的 **sizeof** 表达式不表示对自身的引用，并在 C 和 C++ 中都启用。

因为将来可能要对线程本地存储功能进行增强，C++ 不允许对线程数据进行此类的动态初始化。

* 在 Windows Vista 之前的 Windows 操作系统上，\_\_declspec( thread ) 具有一些限制。 如果 DLL 将任何数据或对象声明为 \_\_declspec( thread )，可能会导致保护错误（如果动态加载）。 使用 [LoadLibrary](https://learn.microsoft.com/zh-cn/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryw) 加载 DLL 后，只要代码引用 \_\_declspec( thread ) 数据，就会导致系统故障。 由于线程的全局变量空间在运行时进行分配，此空间的大小基于对应用程序的要求加静态链接的所有 DLL 的要求的计算。 使用 LoadLibrary 时，不能扩展此空间以允许使用 \_\_declspec( thread ) 声明的线程本地变量。 如果该 DLL 可能使用 LoadLibrary 进行了加载，则使用 DLL 中的 TLS API（如 [TlsAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-tlsalloc)）来分配 TLS。

## 网址：[使用 C 和 Win32 进行多线程编程 | Microsoft Learn](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-c-and-win32?view=msvc-170)