# 学习大纲

|  |
| --- |
|  |

## 使用 C++ 和 MFC 进行多线程编程

|  |
| --- |
| Microsoft 基础类 (MFC) 库为多线程应用程序提供支持。 本主题介绍了进程和线程以及 MFC 的多线程处理方法。  进程是应用程序的执行实例。 例如，当你双击记事本图标时，将启动运行记事本的进程。  线程是进程内的执行路径。 当你启动记事本时，操作系统将创建一个进程，并开始执行该进程的主线程。 当此线程终止时，进程也会终止。 此主线程以函数地址的形式由启动代码提供给操作系统。 通常，提供的是 main 或 WinMain 函数的地址。  如果需要，可以在应用程序中创建更多线程。 当你不想让用户等待后台或维护任务完成时，可能需要创建更多线程来处理这些任务。 MFC 应用程序中的所有线程都由 [CWinThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170) 对象表示。 在大多数情况下，你甚至不必显式创建这些对象；而是调用框架帮助程序函数 [AfxBeginThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxbeginthread)，该函数会为你创建 CWinThread 对象。  MFC 区分两种类型的线程：用户界面线程和工作线程。 用户界面线程通常用于处理用户输入并响应用户生成的事件和消息。 工作线程通常用于完成无需用户输入的任务，例如重新计算。 Win32 API 不区分线程类型；它只需要了解线程的起始地址，以便它可以开始执行线程。 MFC 通过为用户界面中的事件提供消息泵，专门处理用户界面线程。 CWinApp 是用户界面线程对象的一个示例，因为它派生自 CWinThread 并处理用户生成的事件和消息。  应特别注意多个线程可能需要访问同一对象的情况。 [多线程处理：编程提示](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170)中介绍了可用于解决这些情况下可能出现的问题的技术。 [多线程处理：如何使用同步类](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-how-to-use-the-synchronization-classes?view=msvc-170)中介绍了如何使用可用的类来同步从多个线程到单个对象的访问。  编写和调试多线程编程本质上是一项复杂而棘手的任务，因为必须确保对象不会被多个线程同时访问。 多线程主题并不教授多线程编程的基础知识，只介绍如何在多线程程序中使用 MFC。 Visual C++ 中包含的多线程 MFC 示例演示了 MFC 未包含的一些多线程添加功能和 Win32 API；但是，它们只是作为一个起点。  有关操作系统如何处理进程和线程的详细信息，请参阅 Windows SDK 中的[进程和线程](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/processes-and-threads) |

## 多线程处理：创建 MFC 用户界面线程

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 用户界面线程通常用于处理用户输入和响应用户事件，独立于执行应用程序其他部分的线程。 已为你创建并启动主应用程序线程（在 CWinApp 派生类中提供）。 本主题介绍创建其他用户界面线程所需的步骤。  创建用户界面线程时必须做的第一件事是从 [CWinThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170) 派生类。 必须使用 [DECLARE\_DYNCREATE](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/run-time-object-model-services?view=msvc-170#declare_dyncreate) 和 [IMPLEMENT\_DYNCREATE](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/run-time-object-model-services?view=msvc-170#implement_dyncreate) 宏声明和实现此类。 此类必须重写某些函数，也可以重写其他函数。 下表显示了这些函数及其应执行的操作。 创建用户界面线程时要重写的函数 展开表   | **功能** | **目的** | | --- | --- | | [ExitInstance](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#exitinstance) | 线程终止时执行清理。 通常会被重写。 | | [InitInstance](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#initinstance) | 执行线程实例初始化。 必须被重写。 | | [OnIdle](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#onidle) | 执行特定于线程的空闲时间处理。 通常不会被重写。 | | [PreTranslateMessage](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#pretranslatemessage) | 在将消息调度到 TranslateMessage 和 DispatchMessage 之前对其进行筛选。 通常不会被重写。 | | [ProcessWndProcException](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#processwndprocexception) | 截获由线程的消息和命令处理程序引发的未经处理的异常。 通常不会被重写。 | | [运行](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#run) | 控制线程的函数。 包含消息泵。 很少被重写。 |   MFC 通过参数重载提供两个版本的 AfxBeginThread：一个只能创建辅助线程，另一个既可创建用户界面线程也可创建辅助线程。 若要启动用户界面线程，请调用 [AfxBeginThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxbeginthread) 的第二个重载，并提供下列信息：   * 派生自 CWinThread 的类的 [RUNTIME\_CLASS](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/run-time-object-model-services?view=msvc-170#runtime_class)。 * （可选）所需优先级级别。 默认为正常优先级。 有关可用优先级级别的详细信息，请参阅 Windows SDK 中的 [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadpriority)。 * （可选）所需的线程堆栈大小。 默认为与创建线程时相同的堆栈大小。 * （可选）CREATE\_SUSPENDED，如果希望线程在挂起状态下创建的话。 默认为 0，否则正常启动线程。 * （可选）所需的安全属性。 默认为与父线程具有相同的访问使用情况。 有关此安全信息格式的详细信息，请参阅 Windows SDK 中的 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))。   AfxBeginThread 为你执行大部分工作。 它会创建类的新对象，使用你提供的信息初始化它，并调用 [CWinThread::CreateThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#createthread) 以开始执行线程。 在整个过程中都会进行检查，以确保在创建的任何部分失败时正确解除分配所有对象。 |

## 多线程处理：在 MFC 中创建工作线程

|  |
| --- |
| 工作线程通常用于处理后台任务，用户不必等待其继续使用应用程序。 重新计算和后台打印等任务是工作线程的好例子。 本主题详细介绍了创建工作线程所需的步骤。 主题包括：   * [启动线程](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-creating-worker-threads?view=msvc-170#_core_starting_the_thread) * [实现控制函数](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-creating-worker-threads?view=msvc-170#_core_implementing_the_controlling_function) * [示例](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-creating-worker-threads?view=msvc-170#_core_controlling_function_example)   创建工作线程相对简单。 只需完成两步就能使线程运行：实现控制函数并启动线程。 无需从 [CWinThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170) 获取类。 如果需要特殊版本的 CWinThread，则可以获取类，但对于大多数简单工作线程来说，这不是必需的。 无需修改即可使用 CWinThread。 启动线程 有两个重载版本的 AfxBeginThread：一个只能创建工作线程，一个既可创建用户界面线程也可创建工作线程。 若要使用第一个重载开始执行工作线程，请调用 [AfxBeginThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxbeginthread)，并提供以下信息：   * 控制函数的地址。 * 要传递给控制函数的参数。 * （可选）想要的线程优先级。 默认为正常优先级。 有关可用优先级级别的详细信息，请参阅 Windows SDK 中的 [SetThreadPriority](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadpriority)。 * （可选）所需的线程堆栈大小。 默认为与创建线程时相同的堆栈大小。 * （可选）CREATE\_SUSPENDED，如果希望线程在挂起状态下创建的话。 默认为 0，否则正常启动线程。 * （可选）所需的安全属性。 默认为与父线程具有相同的访问使用情况。 有关此安全信息格式的详细信息，请参阅 Windows SDK 中的 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))。   AfxBeginThread 为你创建和初始化 CWinThread 对象，启动对象，并返回其地址以便你稍后可以引用它。 在整个过程中都会进行检查，以确保在创建的任何部分失败时正确解除分配所有对象。 实现控制函数 控制函数定义线程。 输入此函数时，线程启动，退出时线程终止。 此函数应具有以下原型：  C++复制  UINT MyControllingFunction( LPVOID pParam );  该参数是单个值。 函数在此参数中接收的值是创建线程对象时传递给构造函数的值。 控制函数可以采用它选择的任何方式解释此值。 可将它视为标量值或指向包含多个参数的结构的指针，也可以忽略它。 如果参数引用结构，则结构不仅可用于将数据从调用方传递到线程，还可用于将数据从线程传回调用方。 如果使用此类结构将数据传回调用方，则线程需要在结果准备就绪时通知调用方。 有关从工作线程传到调用方的信息，请参阅[多线程处理：编程提示](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170)。  函数终止时，应返回指示终止原因的 UINT 值。 通常，此退出代码为 0 表示成功，其他值指示不同类型的错误。 这完全依赖于实现。 某些线程可以维护对象的使用数，并返回该对象的当前使用数。 若要查看应用程序如何检索此值，请参阅[多线程处理：终止线程](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-terminating-threads?view=msvc-170)。  使用 MFC 库编写的多线程程序可以执行的操作存在一些限制。 有关这些限制的说明以及有关使用线程的其他提示，请参阅[多线程处理：编程提示](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170)。 控制函数示例 以下示例演示如何定义控制函数并从程序的其他部分使用它。  C++复制  UINT MyThreadProc( LPVOID pParam )  {  CMyObject\* pObject = (CMyObject\*)pParam;  if (pObject == NULL ||  !pObject->IsKindOf(RUNTIME\_CLASS(CMyObject)))  return 1; // if pObject is not valid  // do something with 'pObject'  return 0; // thread completed successfully  }  // inside a different function in the program  .  .  .  pNewObject = new CMyObject;  AfxBeginThread(MyThreadProc, pNewObject);  .  . |

## 多线程处理：何时使用 MFC 同步类

|  |
| --- |
| MFC 提供的多线程类分为两类：同步对象（[CSyncObject](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/csyncobject-class?view=msvc-170)、[CSemaphore](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/csemaphore-class?view=msvc-170)、[CMutex](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cmutex-class?view=msvc-170)、[CCriticalSection](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/ccriticalsection-class?view=msvc-170) 和 [CEvent](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cevent-class?view=msvc-170)）和同步访问对象（[CMultiLock](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cmultilock-class?view=msvc-170) 和 [CSingleLock](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/csinglelock-class?view=msvc-170)）。  必须控制对资源的访问权限以确保资源的完整性时，使用同步类。 同步访问类用于获取对这些受控资源的访问权限。 本主题介绍何时使用各个类。  若要确定应使用的同步类，请询问以下系列问题：   1. 应用程序是否必须等待某些事情发生，然后才能访问资源（例如，必须先从通信端口接收数据，然后才能将其写入文件）？   如果是，请使用 CEvent。   1. 例如，同一应用程序内的多个线程是否可以同时访问此资源（例如，应用程序允许最多五个窗口查看同一文档）？   如果是，请使用 CSemaphore。   1. 是否可以多个应用程序使用此资源（例如，该资源位于 DLL 中）？   如果是，请使用 CMutex。  如果否，请使用 CCriticalSection。  CSyncObject 从不直接使用。 它是其他四个同步类的基类。 示例 1：使用三个同步类 以维护帐户链接列表的应用程序为例。 此应用程序允许在单独的窗口中检查最多 3 个帐户，但在任何特定时间都只能更新一个帐户。 更新帐户时，更新后的数据会通过网络发送到数据存档。  此示例应用程序使用所有三种类型的同步类。 它允许一次检查最多 3 个帐户，因此它使用 CSemaphore 来限制对 3 个视图对象的访问。 尝试查看第 4 个帐户时，应用程序会等到前 3 个窗口中某个窗口关闭，否则尝试失败。 更新帐户时，应用程序使用 CCriticalSection 来确保一次只更新一个帐户。 更新成功后，它会向 CEvent 发出信号，后者会释放一个等待事件接收信号的线程。 此线程将新数据发送到数据存档。 示例 2：使用同步访问类 选择要使用的同步访问类要更简单。 如果应用程序只涉及访问单个受控资源，请使用 CSingleLock。 如果需要访问任意一个受控资源，请使用 CMultiLock。 在示例 1 中，本可以使用 CSingleLock，因为在每种情况下，在任何特定时间只需要一个资源。  有关如何使用同步类的信息，请参阅[多线程处理：如何使用同步类](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-how-to-use-the-synchronization-classes?view=msvc-170)。 有关同步的信息，请参阅 Windows SDK 中的[同步](https://learn.microsoft.com/zh-cn/windows/win32/Sync/synchronization)。 有关 MFC 中的多线程处理支持的信息，请参阅[使用 C++ 和 MFC 进行多线程处理](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-cpp-and-mfc?view=msvc-170)。 |

## 多线程处理：如何使用 MFC 同步类

|  |
| --- |
| 编写多线程应用程序时，一个常见的问题是在线程之间同步资源访问。 让两个或更多个线程同时访问相同的数据可能会导致不可取和不可预测的结果。 例如，一个线程可能正在更新某个结构的内容，而另一个线程正在读取该结构的内容。 读取线程将接收的数据是未知的：旧数据、新写入的数据，或者可能是这两者的混合。 MFC 提供了许多同步和同步访问类来帮助解决此问题。 本主题介绍可用的类，并说明如何使用这些类在典型的多线程应用程序中创建线程安全类。  典型的多线程应用程序具有一个表示要在线程之间共享的资源的类。 设计良好的完全线程安全类不需要你调用任何同步函数。 所有内容都在内部处理到类，使你能够关注如何以最佳方式使用该类，而不是它可能会如何遭到损坏。 要创建完全线程安全的类，一个有效的方法是将同步类合并到资源类中。 将同步类合并到共享类的过程很简单。  以维护帐户链接列表的应用程序为例。 此应用程序允许在单独的窗口中检查最多 3 个帐户，但在任何特定时间都只能更新一个帐户。 更新帐户时，更新后的数据会通过网络发送到数据存档。  此示例应用程序使用所有三种类型的同步类。 它允许一次检查最多 3 个帐户，因此它使用 [CSemaphore](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/csemaphore-class?view=msvc-170) 来限制对 3 个视图对象的访问。 尝试查看第 4 个帐户时，应用程序会等到前 3 个窗口中某个窗口关闭，否则尝试失败。 更新帐户时，应用程序使用 [CCriticalSection](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/ccriticalsection-class?view=msvc-170) 来确保一次只更新一个帐户。 更新成功后，它会向 [CEvent](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cevent-class?view=msvc-170) 发出信号，后者会释放一个等待事件接收信号的线程。 此线程将新数据发送到数据存档。 设计线程安全类 要使类完全线程安全，请先将适当的同步类作为数据成员添加到共享类。 在先前的帐户管理示例中，CSemaphore 数据成员添加到视图类中，CCriticalSection 数据成员添加到链接列表类中，CEvent 数据成员添加到数据存储类中。  接下来，向修改类中的数据或访问受控资源的所有成员函数添加同步调用。 在每个函数中，应创建 [CSingleLock](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/csinglelock-class?view=msvc-170) 或 [CMultiLock](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cmultilock-class?view=msvc-170) 对象，并调用该对象的 Lock 函数。 当锁对象超出范围并被销毁时，对象的析构函数会调用 Unlock，从而释放资源。 当然，如果需要，你可直接调用 Unlock。  以这种方式设计线程安全类后，此类可以像非线程安全类一样轻松地在多线程应用程序中使用，但具有更高的安全级别。 将同步对象和同步访问对象封装到资源类中可提供完全线程安全编程的所有优势，但无需维护同步代码。  下面的代码示例使用数据成员 m\_CritSection（类型为 CCriticalSection，在共享资源类中声明）和对象 CSingleLock 来演示此方法。 通过使用 m\_CritSection 对象的地址创建 CSingleLock 对象，来尝试同步共享资源（派生自 CWinThread）。 尝试锁定资源，并在获得资源后对共享对象执行操作。 完成操作后，调用 Unlock 来解锁资源并。  C++复制  CSingleLock singleLock(&m\_CritSection);  singleLock.Lock();  // resource locked  //.usage of shared resource...  singleLock.Unlock();  **备注**  与其他 MFC 同步类不同，CCriticalSection 没有计时锁请求选项。 永远等不到线程变为空闲线程。  此方法的缺点是，该类的速度比没有添加同步对象的类稍微慢一些。 此外，如果存在有多个线程可能会删除该对象的情况，则合并方法可能并不总是有效。 如果无效，最好维护单独的同步对象。  若要了解如何确定在不同情况下要使用的同步类，请参阅[多线程处理：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)。 有关同步的详细信息，请参阅 Windows SDK 中的[同步](https://learn.microsoft.com/zh-cn/windows/win32/Sync/synchronization)。 若要详细了解 MFC 中的多线程处理支持，请参阅[使用 C++ 和 MFC 进行多线程处理](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-with-cpp-and-mfc?view=msvc-170)。 另请参阅 |

## 多线程处理：在 MFC 中终止线程

|  |
| --- |
| 两种正常情况导致线程终止：控制函数退出或线程不允许运行到完成。 如果字处理器使用线程进行后台打印，则打印成功完成后，控制函数将正常终止。 但如果用户需要取消打印，则必须提前终止后台打印线程。 本主题介绍如何实现每种情况，以及如何在线程终止后获取线程的退出代码。   * [普通线程终止](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-terminating-threads?view=msvc-170#_core_normal_thread_termination) * [线程提前终止](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-terminating-threads?view=msvc-170#_core_premature_thread_termination) * [检索线程的退出代码](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-terminating-threads?view=msvc-170#_core_retrieving_the_exit_code_of_a_thread)  普通线程终止 对于工作线程，普通线程终止很简单：退出控制函数并返回一个表示终止原因的值。 可以使用 [AfxEndThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxendthread) 函数或 **return** 语句。 通常，0 表示成功完成，但这由你决定。  对于用户界面线程，过程同样简单：从用户界面线程内部调用 Windows SDK 中的 [PostQuitMessage](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/nf-winuser-postquitmessage)。 PostQuitMessage 唯一采用的参数是线程的退出代码。 至于工作线程，0 通常表示成功完成。 线程提前终止 过早终止线程几乎一样简单：从线程内部调用 [AfxEndThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxendthread)。 将所需退出代码作为唯一参数传递。 这会停止执行线程，解除分配线程的堆栈，拆离附加到线程的所有 DLL，并从内存中删除线程对象。  AfxEndThread 必须从要终止的线程中调用。 如果要从另一个线程终止线程，则必须在两个线程之间设置通信方法。 检索线程的退出代码 若要获取工作线程或用户界面线程的退出代码，请调用 [GetExitCodeThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread) 函数。 有关此函数的信息，请参阅 Windows SDK。 此函数获取线程的句柄（存储在 CWinThread 对象的 m\_hThread 数据成员中）和一个 DWORD 的地址。  如果线程仍处于活动状态，GetExitCodeThread 会将 STILL\_ACTIVE 放置在提供的 DWORD 地址中；否则，退出代码将放置在此地址中。  检索 [CWinThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170) 对象的退出代码需要执行额外的步骤。 默认情况下，当 CWinThread 线程终止时，线程对象将删除。 这意味着将无法访问 m\_hThread 数据成员，因为 CWinThread 对象不再存在。 为避免这种情况，请执行下列操作之一：   * 将 m\_bAutoDelete 数据成员设置为 FALSE。 这样，CWinThread 对象就可以在线程终止后继续存在。 然后，可以在线程终止后访问 m\_hThread 数据成员。 但如果使用此方法，则需负责销毁 CWinThread 对象，因为框架不会自动删除该对象。 这是首选方法。 * 单独存储线程的句柄。 创建线程后，将其 m\_hThread 数据成员（使用 ::DuplicateHandle）复制到另一个变量并通过该变量访问它。 这样当终止发生时，对象会自动删除，而你仍然可以找出线程终止的原因。 请注意，复制句柄之前，线程不会终止。 最安全的方法是将 CREATE\_SUSPENDED 传递给 [AfxBeginThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/application-information-and-management?view=msvc-170#afxbeginthread)，存储句柄，然后通过调用 [ResumeThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170#resumethread) 继续线程。   利用任一方法都能确定 CWinThread 对象终止的原因。 另请参阅 [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) [\_endthread、\_endthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/endthread-endthreadex?view=msvc-170) [\_beginthread、\_beginthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/beginthread-beginthreadex?view=msvc-170) [ExitThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitthread) |

## 多线程处理：MFC 编程提示

|  |
| --- |
| 多线程应用程序比单线程应用程序的要求更严格，这是为了确保操作按预期顺序进行，并且多个线程访问的任何数据都不会损坏。 本主题介绍在使用 Microsoft 基础类 (MFC) 库进行多线程应用程序编程时避免潜在问题的方法。   * [从多个线程访问对象](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170#_core_accessing_objects_from_multiple_threads) * [从非 MFC 线程访问 MFC 对象](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170#_core_accessing_mfc_objects_from_non.2d.mfc_threads) * [Windows 句柄映射](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170#_core_windows_handle_maps) * [在线程之间通信](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-programming-tips?view=msvc-170#_core_communicating_between_threads)  从多个线程访问对象 MFC 对象本身不是线程安全的。 除非你使用 MFC 同步类和/或适当的 Win32 同步对象（如临界区），否则两个单独的线程无法操作同一对象。 有关临界区和其他相关对象的详细信息，请参阅 Windows SDK 中的[同步](https://learn.microsoft.com/zh-cn/windows/win32/Sync/synchronization)。  类库在内部使用临界区来保护全局数据结构，例如调试内存分配所使用的临界区。 从非 MFC 线程访问 MFC 对象 如果你有一个多线程应用程序，它创建线程的方式不是使用 [CWinThread](https://learn.microsoft.com/zh-cn/cpp/mfc/reference/cwinthread-class?view=msvc-170) 对象，则你无法从该线程访问其他 MFC 对象。 换句话说，若要从辅助线程访问任何 MFC 对象，则必须使用[多线程：创建用户界面线程](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-creating-user-interface-threads?view=msvc-170)或[多线程：创建工作线程](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-creating-worker-threads?view=msvc-170)中所述的方法之一创建该线程。 这些方法是唯一允许类库初始化那些处理多线程应用程序所需的内部变量的方法。 Windows 句柄映射 一般说来，线程只能访问其创建的 MFC 对象。 这是因为临时的和永久的 Windows 句柄映射保存在线程本地存储中，目的是阻止从多个线程同时进行的访问。 例如，工作线程无法在执行计算后调用文档的 UpdateAllViews 成员函数来修改包含有关新数据的视图的窗口。 这完全不起作用，因为从 CWnd 对象到 HWND 的映射是主线程的本地映射。 这意味着一个线程可能有从 Windows 句柄到 C++ 对象的映射，但另一个线程可能会将该句柄映射到不同的 C++ 对象。 在一个线程中所做的更改不会反映在另一个线程中。  有几种方法可以避免此问题。 第一种是将单个句柄（例如 HWND）而不是 C++ 对象传递给工作线程。 然后，工作线程通过调用相应的 FromHandle 成员函数将这些对象添加到其临时映射中。 还可以通过调用 Attach 将对象添加到线程的永久映射中，但只能在确保对象存在的时间超过线程存在的时间时执行此操作。  另一种方法是创建新的用户定义消息（这些消息对应于将由工作线程执行的不同任务），并使用 ::PostMessage 将这些消息发布到应用程序的主窗口。 这种通信方法类似于两个不同的应用程序进行会话，但两个线程都在相同的地址空间中执行。  有关句柄映射的详细信息，请参阅[技术说明 3](https://learn.microsoft.com/zh-cn/cpp/mfc/tn003-mapping-of-windows-handles-to-objects?view=msvc-170)。 有关线程本地存储的详细信息，请参阅 Windows SDK 中的[线程本地存储](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/thread-local-storage)和[使用线程本地存储](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/using-thread-local-storage)。 在线程之间通信 MFC 提供了许多类，这些类允许线程同步对对象的访问，以维护线程安全性。 [多线程处理：如何使用同步类](https://learn.microsoft.com/zh-cn/cpp/parallel/multithreading-how-to-use-the-synchronization-classes?view=msvc-170)和[多线程处理：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)中介绍了这些类的用法。 有关这些对象的详细信息，请参阅 Windows SDK 中的[同步](https://learn.microsoft.com/zh-cn/windows/win32/Sync/synchronization)。 |

## 多线程和区域设置

|  |
| --- |
| C 运行时库和 C++ 标准库都支持更改程序的区域设置。 本主题讨论了在多线程应用程序中使用这两个库的区域设置功能时出现的问题。 备注 使用 C 运行时库，可以使用 \_beginthread 和 \_beginthreadex 函数创建多线程应用程序。 本主题仅介绍使用这些函数创建的多线程应用程序。 有关详细信息，请参阅 [\_beginthread、\_beginthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/beginthread-beginthreadex?view=msvc-170)。  若要使用 C 运行时库更改区域设置，请使用 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 函数。 在早期版本的 Visual C++ 中，此函数始终会在整个应用程序中修改区域设置。 现在支持按每个线程设置区域设置。 这是使用 [\_configthreadlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/configthreadlocale?view=msvc-170) 函数来完成的。 若要指定 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 应仅更改当前线程中的区域设置，请在该线程中调用 \_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE)。 相反，调用 \_configthreadlocale(\_DISABLE\_PER\_THREAD\_LOCALE) 将导致该线程使用全局区域设置，并且该线程中对 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 的任何调用都将更改未显式启用每线程区域设置的所有线程中的区域设置。  若要使用 C++ 运行时库更改区域设置，请使用 [locale 类](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170)。 通过调用 [locale::global](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 方法，可在未显式启用每线程区域设置的每个线程中更改区域设置。 若要在单个线程或应用程序的一部分中更改区域设置，只需在该线程或部分代码中创建 locale 对象的实例。  **备注**  调用 [**locale::global**](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 会更改 C++ 标准库和 C 运行时库的区域设置。 但是，调用 [**setlocale**](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 仅更改 C 运行时库的区域设置；C++ 标准库不受影响。  以下示例演示了如何使用 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 函数、[locale 类](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170)和 [\_configthreadlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/configthreadlocale?view=msvc-170) 函数在多种不同方案中更改应用程序的区域设置。 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) [\_beginthread、\_beginthreadex](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/beginthread-beginthreadex?view=msvc-170) [\_configthreadlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/configthreadlocale?view=msvc-170) [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170)  [国际化](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/internationalization?view=msvc-170)  [区域设置](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/locale?view=msvc-170)  [<clocale>](https://learn.microsoft.com/zh-cn/cpp/standard-library/clocale?view=msvc-170)  [<区域设置>](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale?view=msvc-170)  [locale 类](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170) |

## 示例：在已启用每线程区域设置的情况下更改区域设置

在此示例中，主线程生成了两个子线程。 第一个线程（线程 A）通过调用 \_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE) 启用了每线程区域设置。 第二个线程（线程 B）以及主线程未启用每线程区域设置。 然后，线程 A 继续使用 C 运行时库的 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 函数更改区域设置。

由于线程 A 已启用每线程区域设置，因此只有线程 A 中的 C 运行时库函数开始使用“法语”区域设置。 线程 B 和主线程中的 C 运行时库函数继续使用“C”区域设置。 此外，由于 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 不会影响 C++ 标准库区域设置，因此 C++ 标准库对象将继续使用“C”区域设置。

C++复制

// multithread\_locale\_1.cpp

// compile with: /EHsc /MD

#include <clocale>

#include <cstdio>

#include <locale>

#include <process.h>

#include <windows.h>

#define NUM\_THREADS 2

using namespace std;

unsigned \_\_stdcall RunThreadA(void \*params);

unsigned \_\_stdcall RunThreadB(void \*params);

BOOL localeSet = FALSE;

HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()

{

HANDLE threads[NUM\_THREADS];

unsigned aID;

threads[0] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadA, NULL, 0, &aID);

unsigned bID;

threads[1] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadB, NULL, 0, &bID);

WaitForMultipleObjects(2, threads, TRUE, INFINITE);

printf\_s("[Thread main] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread main] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread main] locale::global is set to \"%s\"\n",

locale().name().c\_str());

CloseHandle(threads[0]);

CloseHandle(threads[1]);

CloseHandle(printMutex);

return 0;

}

unsigned \_\_stdcall RunThreadA(void \*params)

{

\_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE);

setlocale(LC\_ALL, "french");

localeSet = TRUE;

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread A] Per-thread locale is enabled.\n");

printf\_s("[Thread A] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread A] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

unsigned \_\_stdcall RunThreadB(void \*params)

{

while (!localeSet)

Sleep(100);

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread B] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread B] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread B] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

Output复制

[Thread A] Per-thread locale is enabled.

[Thread A] CRT locale is set to "French\_France.1252"

[Thread A] locale::global is set to "C"

[Thread B] Per-thread locale is NOT enabled.

[Thread B] CRT locale is set to "C"

[Thread B] locale::global is set to "C"

[Thread main] Per-thread locale is NOT enabled.

[Thread main] CRT locale is set to "C"

[Thread main] locale::global is set to "C"

## 示例：在未启用每线程区域设置的情况下更改区域设置

在此示例中，主线程生成了两个子线程。 第一个线程（线程 A）通过调用 \_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE) 启用了每线程区域设置。 第二个线程（线程 B）以及主线程未启用每线程区域设置。 然后，线程 B 继续使用 C 运行时库的 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 函数更改区域设置。

由于线程 B 没有启用每线程区域设置，因此线程 B 和主线程中的 C 运行时库函数开始使用“法语”区域设置。 线程 A 中的 C 运行时库函数继续使用“C”区域设置，因为线程 A 已启用每线程区域设置。 此外，由于 [setlocale](https://learn.microsoft.com/zh-cn/cpp/c-runtime-library/reference/setlocale-wsetlocale?view=msvc-170) 不会影响 C++ 标准库区域设置，因此 C++ 标准库对象将继续使用“C”区域设置。

C++复制

// multithread\_locale\_3.cpp

// compile with: /EHsc /MD

#include <clocale>

#include <cstdio>

#include <locale>

#include <process.h>

#include <windows.h>

#define NUM\_THREADS 2

using namespace std;

unsigned \_\_stdcall RunThreadA(void \*params);

unsigned \_\_stdcall RunThreadB(void \*params);

BOOL localeSet = FALSE;

BOOL configThreadLocaleCalled = FALSE;

HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()

{

HANDLE threads[NUM\_THREADS];

unsigned aID;

threads[0] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadA, NULL, 0, &aID);

unsigned bID;

threads[1] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadB, NULL, 0, &bID);

WaitForMultipleObjects(2, threads, TRUE, INFINITE);

printf\_s("[Thread main] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread main] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread main] locale::global is set to \"%s\"\n",

locale().name().c\_str());

CloseHandle(threads[0]);

CloseHandle(threads[1]);

CloseHandle(printMutex);

return 0;

}

unsigned \_\_stdcall RunThreadA(void \*params)

{

\_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE);

configThreadLocaleCalled = TRUE;

while (!localeSet)

Sleep(100);

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread A] Per-thread locale is enabled.\n");

printf\_s("[Thread A] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread A] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

unsigned \_\_stdcall RunThreadB(void \*params)

{

while (!configThreadLocaleCalled)

Sleep(100);

setlocale(LC\_ALL, "french");

localeSet = TRUE;

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread B] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread B] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread B] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

Output复制

[Thread B] Per-thread locale is NOT enabled.

[Thread B] CRT locale is set to "French\_France.1252"

[Thread B] locale::global is set to "C"

[Thread A] Per-thread locale is enabled.

[Thread A] CRT locale is set to "C"

[Thread A] locale::global is set to "C"

[Thread main] Per-thread locale is NOT enabled.

[Thread main] CRT locale is set to "French\_France.1252"

[Thread main] locale::global is set to "C"

## 示例：在已启用每线程区域设置的情况下更改全局区域设置

在此示例中，主线程生成了两个子线程。 第一个线程（线程 A）通过调用 \_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE) 启用了每线程区域设置。 第二个线程（线程 B）以及主线程未启用每线程区域设置。 然后，线程 A 继续使用 C++ 标准库的 [locale::global](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 方法更改区域设置。

由于线程 A 已启用每线程区域设置，因此只有线程 A 中的 C 运行时库函数开始使用“法语”区域设置。 线程 B 和主线程中的 C 运行时库函数继续使用“C”区域设置。 但是，由于 [locale::global](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 方法会“全局”更改区域设置，因此所有线程中的所有 C++ 标准库对象都开始使用“法语”区域设置。

C++复制

// multithread\_locale\_2.cpp

// compile with: /EHsc /MD

#include <clocale>

#include <cstdio>

#include <locale>

#include <process.h>

#include <windows.h>

#define NUM\_THREADS 2

using namespace std;

unsigned \_\_stdcall RunThreadA(void \*params);

unsigned \_\_stdcall RunThreadB(void \*params);

BOOL localeSet = FALSE;

HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()

{

HANDLE threads[NUM\_THREADS];

unsigned aID;

threads[0] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadA, NULL, 0, &aID);

unsigned bID;

threads[1] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadB, NULL, 0, &bID);

WaitForMultipleObjects(2, threads, TRUE, INFINITE);

printf\_s("[Thread main] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread main] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread main] locale::global is set to \"%s\"\n",

locale().name().c\_str());

CloseHandle(threads[0]);

CloseHandle(threads[1]);

CloseHandle(printMutex);

return 0;

}

unsigned \_\_stdcall RunThreadA(void \*params)

{

\_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE);

locale::global(locale("french"));

localeSet = TRUE;

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread A] Per-thread locale is enabled.\n");

printf\_s("[Thread A] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread A] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

unsigned \_\_stdcall RunThreadB(void \*params)

{

while (!localeSet)

Sleep(100);

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread B] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread B] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread B] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

Output复制

[Thread A] Per-thread locale is enabled.

[Thread A] CRT locale is set to "French\_France.1252"

[Thread A] locale::global is set to "French\_France.1252"

[Thread B] Per-thread locale is NOT enabled.

[Thread B] CRT locale is set to "C"

[Thread B] locale::global is set to "French\_France.1252"

[Thread main] Per-thread locale is NOT enabled.

[Thread main] CRT locale is set to "C"

[Thread main] locale::global is set to "French\_France.1252"

## 示例：在未启用每线程区域设置的情况下更改全局区域设置

在此示例中，主线程生成了两个子线程。 第一个线程（线程 A）通过调用 \_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE) 启用了每线程区域设置。 第二个线程（线程 B）以及主线程未启用每线程区域设置。 然后，线程 B 继续使用 C++ 标准库的 [locale::global](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 方法更改区域设置。

由于线程 B 没有启用每线程区域设置，因此线程 B 和主线程中的 C 运行时库函数开始使用“法语”区域设置。 线程 A 中的 C 运行时库函数继续使用“C”区域设置，因为线程 A 已启用每线程区域设置。 但是，由于 [locale::global](https://learn.microsoft.com/zh-cn/cpp/standard-library/locale-class?view=msvc-170#global) 方法会“全局”更改区域设置，因此所有线程中的所有 C++ 标准库对象都开始使用“法语”区域设置。

C++复制

// multithread\_locale\_4.cpp

// compile with: /EHsc /MD

#include <clocale>

#include <cstdio>

#include <locale>

#include <process.h>

#include <windows.h>

#define NUM\_THREADS 2

using namespace std;

unsigned \_\_stdcall RunThreadA(void \*params);

unsigned \_\_stdcall RunThreadB(void \*params);

BOOL localeSet = FALSE;

BOOL configThreadLocaleCalled = FALSE;

HANDLE printMutex = CreateMutex(NULL, FALSE, NULL);

int main()

{

HANDLE threads[NUM\_THREADS];

unsigned aID;

threads[0] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadA, NULL, 0, &aID);

unsigned bID;

threads[1] = (HANDLE)\_beginthreadex(

NULL, 0, RunThreadB, NULL, 0, &bID);

WaitForMultipleObjects(2, threads, TRUE, INFINITE);

printf\_s("[Thread main] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread main] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread main] locale::global is set to \"%s\"\n",

locale().name().c\_str());

CloseHandle(threads[0]);

CloseHandle(threads[1]);

CloseHandle(printMutex);

return 0;

}

unsigned \_\_stdcall RunThreadA(void \*params)

{

\_configthreadlocale(\_ENABLE\_PER\_THREAD\_LOCALE);

configThreadLocaleCalled = TRUE;

while (!localeSet)

Sleep(100);

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread A] Per-thread locale is enabled.\n");

printf\_s("[Thread A] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread A] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

unsigned \_\_stdcall RunThreadB(void \*params)

{

while (!configThreadLocaleCalled)

Sleep(100);

locale::global(locale("french"));

localeSet = TRUE;

WaitForSingleObject(printMutex, INFINITE);

printf\_s("[Thread B] Per-thread locale is NOT enabled.\n");

printf\_s("[Thread B] CRT locale is set to \"%s\"\n",

setlocale(LC\_ALL, NULL));

printf\_s("[Thread B] locale::global is set to \"%s\"\n\n",

locale().name().c\_str());

ReleaseMutex(printMutex);

return 1;

}

Output复制

[Thread B] Per-thread locale is NOT enabled.

[Thread B] CRT locale is set to "French\_France.1252"

[Thread B] locale::global is set to "French\_France.1252"

[Thread A] Per-thread locale is enabled.

[Thread A] CRT locale is set to "C"

[Thread A] locale::global is set to "French\_France.1252"

[Thread main] Per-thread locale is NOT enabled.

[Thread main] CRT locale is set to "French\_France.1252"

[Thread main] locale::global is set to "French\_France.1252"

# 演练

## 1新建一个文件夹Lesson55-MFC-control-thread，然后把上一节的项目拷贝粘贴到里面，修改一下项目名称：MFC-control-thread和项目文件夹名称已经解决方案名称，注意：源文件不修改没有问题，应用程序名称是根据项目名称来生成的。

|  |
| --- |
|  |

## 2.这一节我们来给线程添加控制功能，如暂停，停止等等,我们在第二个按钮下面添加一个暂停按钮和一个恢复按钮，ID方便是IDC\_BTN\_SUSPEND和IDC\_BTN\_RESUME然后再添加一个终止按钮，ID为IDC\_BTN\_END

|  |
| --- |
|  |

## 3.为了能够控制线程，我们需要把上一节的创建线程的代码修改一下，用一个变量把线程标识保存起来，需要先定义一个全局变量，然后用它来接收新创建的线程指针

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

### 效果：

|  |
| --- |
|  |

## 注意：其实是可以创建任意多个线程的，我们这里给他设置限制，规定只能创建一个线程。

## 4.我们来写线程暂停和恢复的代码，都是比较简单，直接调用这个线程对象的SuspandThread方法来暂停线程执行和调用ResumeThread方法来恢复线程的运行。

|  |
| --- |
| void CMFCThreadDlg::OnBnClickedBtnSuspend()  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  pthread1->SuspendThread();  }  }  void CMFCThreadDlg::OnBnClickedBtnResume()  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  pthread1->ResumeThread();  }  } |

## 5.然后我们来做终止线程的按钮点击事件函数，这个相对比较复杂，需要先定义一个全局布尔类型的变量来辅助实现，然后在对话框类的构造函数里面初始化为FALSE;

|  |
| --- |
|  |
|  |

## 6.其实mfc里面没有现成的停止线程方法，我们需要在我们的线程函数里面根据我们设置的标记变量来决定是否执行，我们需要修改一下线程函数的代码

|  |
| --- |
|  |

## 7.然后把每一个按钮的事件事件都需要修改一下

|  |
| --- |
| void CMFCThreadDlg::OnBnClickedButton2()  {  // TODO: 在此添加控件通知处理程序代码  //我们在这里限制只允许创建一个线程，不限制的话，它可以创建任意多个线程  if(pthread1 == NULL)  {  pthread1 = AfxBeginThread(ThreadProc,NULL);//mfc中创建线程  isRunning = TRUE;  }  else  {  AfxMessageBox(\_T("线程已经存在，这个应用程序不允许创建多个线程"));  }  }  void CMFCThreadDlg::OnBnClickedButton3()  {  // TODO: 在此添加控件通知处理程序代码  AfxBeginThread(ThreadProc2,NULL);//mfc中创建线程  }  void CMFCThreadDlg::OnBnClickedBtnSuspend()  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));    }  else  {  pthread1->SuspendThread();//  }  }  void CMFCThreadDlg::OnBnClickedBtnResume()  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  pthread1->ResumeThread();  }  }  void CMFCThreadDlg::OnBnClickedBtnEnd()  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  isRunning = FALSE;  AfxMessageBox(\_T("线程已经删除"));  //线程删除后把显示文本的内容清空  SetDlgItemText(IDC\_RESULT2,TEXT(""));  }  } |

### 程序有一个问题，就是如果启动一个线程，接着把它暂停，然后把他删除，就无法创建新线程，即使一个辅助线程都没有。

## 8.我们来解决一下，新建一个isPaused的全局布尔变量。默认在构造函数中设置为FALSE，按下暂停键设置为TRUE，按下创建和继续已及删除都设置为FALSE；在删除线程之前，需要判断一下线程是否是暂停了，如果是，需要把他恢复运行然后再删除。修改后的代码

|  |
| --- |
|  |
|  |
| void CMFCThreadDlg::OnBnClickedButton2() //创建线程  {  // TODO: 在此添加控件通知处理程序代码  //我们在这里限制只允许创建一个线程，不限制的话，它可以创建任意多个线程  if(pthread1 == NULL)  {  pthread1 = AfxBeginThread(ThreadProc,NULL);//mfc中创建线程  isRunning = TRUE;  isPaused =FALSE;  }  else  {  AfxMessageBox(\_T("线程已经启动，这个应用程序不允许创建多个线程"));  }  }  void CMFCThreadDlg::OnBnClickedBtnSuspend()//暂停线程  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));    }  else  {  //暂停的时候不能够把isRunning设置为False，因为这样子会删除线程  pthread1->SuspendThread();  isPaused = TRUE;  }  }  void CMFCThreadDlg::OnBnClickedBtnResume()//恢复线程的运行  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  pthread1->ResumeThread();  isPaused = FALSE;  }  }  void CMFCThreadDlg::OnBnClickedBtnEnd()//终止线程  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  //判断是否是暂停了，如果是先恢复运行，然后再删除，否则会有问题  if(isPaused) pthread1->ResumeThread();  isPaused = FALSE;  isRunning = FALSE;  AfxMessageBox(\_T("线程已经删除"));  //线程删除后把显示文本的内容清空  SetDlgItemText(IDC\_RESULT2,TEXT(""));  }  } |

# 小结：

## 1.MFC中创建线程使用AfxBeginThread函数，需要用一个CWinThread的指针变量来保存返回值，他指向一个线程

## 2.在MFC中暂停线程的执行需要使用CWinThread的指针变量->SuspendThread方法

## 3. .在MFC中恢复线程的执行需要使用CWinThread的指针变量->ResumeThread方法

## 4.MFC没有提供现成的终止线程的方法，需要我们使用一个布尔类型的变量作为运行标记，还需要一个布尔类型的变量作为暂停标记，在需要终止线程运行的地方把运行标记设置为FALSE.而且需要判断线程是否是暂停了，如果暂停了，不要直接终止，需要先把线程恢复运行，然后再终止线程。

# 这一节的学习到此为止，完整代码如下

## Lesson55-MFC-control-thread/MFC-control-thread/ MFCThreadDlg.h

|  |
| --- |
| // MFCThreadDlg.h : 头文件  //  #pragma once  #include "afxwin.h"  UINT ThreadProc(LPVOID lpvoid);//线程函数必须是全局函数，在类外面声明  UINT ThreadProc2(LPVOID lpvoid);//线程函数必须是全局函数，在类外面声明  // CMFCThreadDlg 对话框  class CMFCThreadDlg : public CDialogEx  {  // 构造  public:  CMFCThreadDlg(CWnd\* pParent = NULL); // 标准构造函数  // 对话框数据  enum { IDD = IDD\_MFCTHREAD\_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 OnBnClickedButton1();  afx\_msg void OnBnClickedButton2();  afx\_msg void OnBnClickedButton3();  afx\_msg void OnBnClickedBtnSuspend();  afx\_msg void OnBnClickedBtnResume();  afx\_msg void OnBnClickedBtnEnd();  }; |

## Lesson55-MFC-control-thread/MFC-control-thread/ MFCThreadDlg.cpp

|  |
| --- |
| // MFCThreadDlg.cpp : 实现文件  //  #include "stdafx.h"  #include "MFCThread.h"  #include "MFCThreadDlg.h"  #include "afxdialogex.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #endif  //定义保存线程的全局变量  CWinThread \* pthread1;  //保存线程运行状态的变量  BOOL isRunning;  BOOL isPaused;//暂停标记  //线程函数的定义，注意不是类的成员函数  UINT ThreadProc(LPVOID lpvoid)  {    for(int i=1;i<100;i++)  {  if(!isRunning) break;//用我们设置的标记来控制程序的执行，为假就跳出循环  //这里不是在类里面，只能够使用全局函数而不能使用类的成员函数  ::SetDlgItemInt(AfxGetApp()->m\_pMainWnd->m\_hWnd,IDC\_RESULT2,i,FALSE);  Sleep(1000);  }  //线程函数退出之前，需要把线程标识符设置为NULL，不会发生内存泄漏？！！  //这是因为，下面的代码都是根据线程标识符是否是NULL来决定是否继续操作的  //还有，当线程函数退出了，线程就会自动结束，这个可能和普通的情况有点不一样。  pthread1 = NULL;  isRunning = FALSE;//在退出之前保证标志位是FALSE  return 0;  }  UINT ThreadProc2(LPVOID lpvoid)  {    for(int i=1;i<100;i++)  {  //这里不是在类里面，只能够使用全局函数而不能使用类的成员函数  ::SetDlgItemInt(AfxGetApp()->m\_pMainWnd->m\_hWnd,IDC\_RESULT3,i+1,FALSE);  Sleep(400);  }  return 0;  }  // 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()  // CMFCThreadDlg 对话框  //构造函数  CMFCThreadDlg::CMFCThreadDlg(CWnd\* pParent /\*=NULL\*/)  : CDialogEx(CMFCThreadDlg::IDD, pParent)  {  m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);  pthread1 = NULL;//初始化线程指针变量  isRunning = FALSE;  isPaused = FALSE;  }  void CMFCThreadDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);    }  BEGIN\_MESSAGE\_MAP(CMFCThreadDlg, CDialogEx)  ON\_WM\_SYSCOMMAND()  ON\_WM\_PAINT()  ON\_WM\_QUERYDRAGICON()  ON\_BN\_CLICKED(IDOK, &CMFCThreadDlg::OnBnClickedOk)  ON\_BN\_CLICKED(IDC\_BUTTON1, &CMFCThreadDlg::OnBnClickedButton1)  ON\_BN\_CLICKED(IDC\_BUTTON2, &CMFCThreadDlg::OnBnClickedButton2)  ON\_BN\_CLICKED(IDC\_BUTTON3, &CMFCThreadDlg::OnBnClickedButton3)  ON\_BN\_CLICKED(IDC\_BTN\_SUSPEND, &CMFCThreadDlg::OnBnClickedBtnSuspend)  ON\_BN\_CLICKED(IDC\_BTN\_RESUME, &CMFCThreadDlg::OnBnClickedBtnResume)  ON\_BN\_CLICKED(IDC\_BTN\_END, &CMFCThreadDlg::OnBnClickedBtnEnd)  END\_MESSAGE\_MAP()  // CMFCThreadDlg 消息处理程序  BOOL CMFCThreadDlg::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 CMFCThreadDlg::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 CMFCThreadDlg::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 CMFCThreadDlg::OnQueryDragIcon()  {  return static\_cast<HCURSOR>(m\_hIcon);  }  void CMFCThreadDlg::OnBnClickedOk()//防止按下回车，程序退出  {  // 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))  {  CDialogEx::OnOK();  }  }  void CMFCThreadDlg::OnBnClickedButton1()  {  // TODO: 在此添加控件通知处理程序代码  //单线程代码，让我很重的话，主线程会阻塞  /\*for(int i=1;i<100;i++)  {  SetDlgItemInt(IDC\_RESULT1,i+1);  Sleep(1000);  }\*/  }  void CMFCThreadDlg::OnBnClickedButton2()//创建线程  {  // TODO: 在此添加控件通知处理程序代码  //我们在这里限制只允许创建一个线程，不限制的话，它可以创建任意多个线程  if(pthread1 == NULL)  {  pthread1 = AfxBeginThread(ThreadProc,NULL);//mfc中创建线程  isRunning = TRUE;  isPaused =FALSE;  }  else  {  AfxMessageBox(\_T("线程已经启动，这个应用程序不允许创建多个线程"));  }  }  void CMFCThreadDlg::OnBnClickedBtnSuspend()//暂停线程  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));    }  else  {  //暂停的时候不能够把isRunning设置为False，因为这样子会删除线程  pthread1->SuspendThread();  isPaused = TRUE;  }  }  void CMFCThreadDlg::OnBnClickedBtnResume()//恢复线程的运行  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  pthread1->ResumeThread();  isPaused = FALSE;  }  }  void CMFCThreadDlg::OnBnClickedBtnEnd()//终止线程  {  // TODO: 在此添加控件通知处理程序代码  //先判断线程是否存在，如果不存在不允许操作  if(pthread1 == NULL)  {  AfxMessageBox(\_T("线程不存在，请点击创建并启动按钮创建线程"));  }  else  {  //判断是否是暂停了，如果是先恢复运行，然后再删除，否则会有问题  if(isPaused) pthread1->ResumeThread();  isPaused = FALSE;  isRunning = FALSE;  AfxMessageBox(\_T("线程已经删除"));  //线程删除后把显示文本的内容清空  SetDlgItemText(IDC\_RESULT2,TEXT(""));  }  }  void CMFCThreadDlg::OnBnClickedButton3()  {  // TODO: 在此添加控件通知处理程序代码  AfxBeginThread(ThreadProc2,NULL);//mfc中创建线程  } |