# 参考文档：

## 1.关于匿名管道参考：[匿名管道操作](匿名管道操作.docx)

## 2.关于命名管道参考：[命名管道操作](命名管道操作.docx)

# 学习大纲

|  |
| --- |
|  |

# 管道也是一种进程间通信的共享内存。

# Api用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 创建新进程及其主线程。 新进程在调用进程的安全上下文中运行。  如果调用进程正在模拟其他用户，则新进程将令牌用于调用进程，而不是模拟令牌。 若要在模拟令牌表示的用户的安全上下文中运行新进程，请使用 [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera) 或 [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw) 函数。 语法 C++复制  BOOL CreateProcessA(  [in, optional] LPCSTR lpApplicationName,  [in, out, optional] LPSTR lpCommandLine,  [in, optional] LPSECURITY\_ATTRIBUTES lpProcessAttributes,  [in, optional] LPSECURITY\_ATTRIBUTES lpThreadAttributes,  [in] BOOL bInheritHandles,  [in] DWORD dwCreationFlags,  [in, optional] LPVOID lpEnvironment,  [in, optional] LPCSTR lpCurrentDirectory,  [in] LPSTARTUPINFOA lpStartupInfo,  [out] LPPROCESS\_INFORMATION lpProcessInformation  ); 参数 [in, optional] lpApplicationName  要执行的模块的名称。 此模块可以是基于 Windows 的应用程序。 它可以是某种其他类型的模块 (例如 MS-DOS 或 OS/2) （如果本地计算机上提供了相应的子系统）。  字符串可以指定要执行的模块的完整路径和文件名，也可以指定部分名称。 对于部分名称，函数使用当前驱动器和当前目录来完成规范。 函数不会使用搜索路径。 此参数必须包含文件扩展名;不采用默认扩展名。  *lpApplicationName* 参数可以为 **NULL**。 在这种情况下，模块名称必须是 *lpCommandLine* 字符串中第一个空格分隔的标记。 如果使用包含空格的长文件名，请使用带引号的字符串来指示文件名结束和参数开始的位置;否则，文件名不明确。 例如，请考虑字符串“c：\program files\sub dir\program name”。 可以通过多种方式解释此字符串。 系统尝试按以下顺序解释可能性：   1. **c:\program.exe** 2. **c：\program files\sub.exe** 3. **c：\program files\sub dir\program.exe** 4. **c：\program files\sub dir\program name.exe**   如果可执行模块是 16 位应用程序，*lpApplicationName* 应为 *NULL，lpCommandLine* 指向的字符串应指定可执行模块及其参数。  若要运行批处理文件，必须启动命令解释器;将 *lpApplicationName* 设置为 cmd.exe 并将 *lpCommandLine* 设置为以下参数：/c 加上批处理文件的名称。  **重要**  MSRC 工程团队建议不要这样做。 有关更多详细信息，请参阅 [**MS14-019 – 通过.cmd或 .bat 文件修复二进制劫持**](https://msrc.microsoft.com/blog/2014/04/ms14-019-fixing-a-binary-hijacking-via-cmd-or-bat-file/) 。  [in, out, optional] lpCommandLine  要执行的命令行。  此字符串的最大长度为 32,767 个字符，包括 Unicode 终止 null 字符。 如果 *lpApplicationName* 为 **NULL**，则 *lpCommandLine* 的模块名称部分限制为 **MAX\_PATH** 个字符。  此函数的 Unicode 版本 **CreateProcessW** 可以修改此字符串的内容。 因此，此参数不能是指向只读内存 (的指针，例如 **const** 变量或文本字符串) 。 如果此参数是常量字符串，该函数可能会导致访问冲突。  *lpCommandLine* 参数可以为 NULL。 在这种情况下，函数使用 *lpApplicationName* 指向的字符串作为命令行。  如果 *lpApplicationName* 和 *lpCommandLine* 均为非 **NULL**，则 *lpApplicationName* 指向的以 null 结尾的字符串将指定要执行的模块， *lpCommandLine* 指向的以 null 结尾的字符串将指定命令行。 新进程可以使用 [GetCommandLine](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-getcommandlinea) 检索整个命令行。 用 C 编写的控制台进程可以使用 *argc* 和 *argv* 参数来分析命令行。 由于 argv[0] 是模块名称，因此 C 程序员通常将模块名称重复为命令行中的第一个标记。  如果 *lpApplicationName* 为 NULL，则命令行的第一个空格分隔标记将指定模块名称。 如果使用包含空格的长文件名，请使用带引号的字符串来指示文件名结束和参数开始的位置 (请参阅 *lpApplicationName* 参数) 的说明。 如果文件名不包含扩展名，则追加 .exe。 因此，如果文件扩展名.com，则此参数必须包含.com扩展名。 如果文件名以不带扩展名的句点 (.) 结尾，或者文件名包含路径，则不会追加 .exe。 如果文件名不包含目录路径，系统会按以下顺序搜索可执行文件：   1. 从中加载应用程序的目录。 2. 父进程的当前目录。 3. 32 位 Windows 系统目录。 使用 [GetSystemDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getsystemdirectorya) 函数获取此目录的路径。 4. 16 位 Windows 系统目录。 没有获取此目录的路径的函数，但会对其进行搜索。 此目录的名称为 System。 5. Windows 目录。 使用 [GetWindowsDirectory](https://learn.microsoft.com/zh-cn/windows/desktop/api/sysinfoapi/nf-sysinfoapi-getwindowsdirectorya) 函数获取此目录的路径。 6. PATH 环境变量中列出的目录。 请注意，此函数不会搜索由 **应用路径** 注册表项指定的每个应用程序路径。 若要在搜索序列中包含此每个应用程序的路径，请使用 [ShellExecute](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/axe/shellexecute) 函数。   系统向命令行字符串添加一个终止 null 字符，以将文件名与参数分开。 这会将原始字符串划分为两个字符串以供内部处理。  [in, optional] lpProcessAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构确定返回的新进程对象的句柄是否可以由子进程继承。 如果 *lpProcessAttributes* 为 **NULL**，则不能继承句柄。  结构的 **lpSecurityDescriptor** 成员为新进程指定安全描述符。 如果 *lpProcessAttributes* 为 NULL 或 **lpSecurityDescriptor** 为 **NULL**，则进程将获取默认安全描述符。 进程的默认安全描述符中的 ACL 来自创建者的主令牌。**Windowsxp：**进程的默认安全描述符中的 ACL 来自创建者的主要令牌或模拟令牌。 此行为随 Windows XP SP2 和 Windows Server 2003 更改。  [in, optional] lpThreadAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构确定返回的新线程对象的句柄是否可以由子进程继承。 如果 *lpThreadAttributes* 为 NULL，则不能继承句柄。  结构的 **lpSecurityDescriptor** 成员指定主线程的安全描述符。 如果 *lpThreadAttributes* 为 NULL 或 **lpSecurityDescriptor** 为 NULL，则线程获取默认安全描述符。 线程的默认安全描述符中的 ACL 来自进程令牌。**Windowsxp：**线程的默认安全描述符中的 ACL 来自创建者的主令牌或模拟令牌。 此行为随 Windows XP SP2 和 Windows Server 2003 更改。  [in] bInheritHandles  如果此参数为 TRUE，则调用进程中的每个可继承句柄都由新进程继承。 如果参数为 FALSE，则不继承句柄。 请注意，继承的句柄与原始句柄具有相同的值和访问权限。 有关可继承句柄的其他讨论，请参阅备注。  **终端服务：**不能跨会话继承句柄。 此外，如果此参数为 TRUE，则必须在调用方所在的同一会话中创建进程。  **受保护的流程灯 (PPL) 进程：**当 PPL 进程创建非 PPL 进程时，将阻止泛型句柄继承，因为不允许将PROCESS\_DUP\_HANDLE从非 PPL 进程转换为 PPL 进程。 请参阅 [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-security-and-access-rights)  **Windows 7：**即使 参数为 FALSE，也会继承STD\_INPUT\_HANDLE、STD\_OUTPUT\_HANDLE和STD\_ERROR\_HANDLE。  [in] dwCreationFlags  控制优先级类和进程的创建的标志。 有关值的列表，请参阅 [进程创建标志](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-creation-flags)。   |  |  | | --- | --- | | **CREATE\_BREAKAWAY\_FROM\_JOB**  0x01000000 | 与作业关联的进程的子进程不与作业关联。 如果调用进程未与作业关联，则此常量不起作用。 如果调用进程与作业相关联，则作业必须设置 **JOB\_OBJECT\_LIMIT\_BREAKAWAY\_OK** 限制。 | | **CREATE\_DEFAULT\_ERROR\_MODE**  0x04000000 | 新进程不会继承调用进程的错误模式。 相反，新进程会获取默认错误模式。 此功能对于在运行时禁用了硬错误的多线程 shell 应用程序特别有用。 默认行为是让新进程继承调用方的错误模式。 设置此标志会更改该默认行为。 | | **CREATE\_NEW\_CONSOLE**  0x00000010 | 新进程具有新的控制台，而不是继承其父级的控制台 (默认) 。 有关详细信息，请参阅 [创建控制台](https://learn.microsoft.com/zh-cn/windows/console/creation-of-a-console)。 此标志不能与 **DETACHED\_PROCESS** 一起使用。 | | **CREATE\_NEW\_PROCESS\_GROUP**  0x00000200 | 新进程是新进程组的根进程。 进程组包括此根进程的所有子代进程。 新进程组的进程标识符与 lpProcessInformation 参数中返回的进程标识符相同。 [**GenerateConsoleCtrlEvent**](https://learn.microsoft.com/zh-cn/windows/console/generateconsolectrlevent) 函数使用进程组来向一组控制台进程发送 CTRL+BREAK 信号。 如果指定了此标志，将为新进程组中的所有进程禁用 CTRL+C 信号。 如果使用 **CREATE\_NEW\_CONSOLE** 指定，则忽略此标志。 | | **CREATE\_NO\_WINDOW**  0x08000000 | 进程是在没有控制台窗口的情况下运行的控制台应用程序。 因此，未设置应用程序的控制台句柄。 如果应用程序不是控制台应用程序，或者与 **CREATE\_NEW\_CONSOLE** 或 **DETACHED\_PROCESS**一起使用，则忽略此标志。 | | **CREATE\_PROTECTED\_PROCESS**  0x00040000 | 进程将作为受保护的进程运行。 系统限制对受保护进程和受保护进程的线程的访问。 有关进程如何与受保护进程交互的详细信息，请参阅 [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/win32/procthread/process-security-and-access-rights)。 若要激活受保护的进程，二进制文件必须具有特殊签名。 此签名由 Microsoft 提供，但当前不适用于非 Microsoft 二进制文件。 目前有四个受保护的进程：媒体基础、音频引擎、Windows 错误报告和系统。 加载到这些二进制文件中的组件也必须签名。 多媒体公司可以利用前两个受保护的流程。 有关详细信息，请参阅 [受保护的媒体路径概述](https://learn.microsoft.com/zh-cn/windows/win32/medfound/protected-media-path)。 **Windows Server 2003 和 Windows XP：** 不支持此值。 | | **CREATE\_PRESERVE\_CODE\_AUTHZ\_LEVEL**  0x02000000 | 允许调用方执行子进程，该子进程绕过通常自动应用于进程的进程限制。 | | **CREATE\_SECURE\_PROCESS**  0x00400000 | 此标志允许启动在Virtualization-Based安全环境中运行的安全进程。 | | **CREATE\_SEPARATE\_WOW\_VDM**  0x00000800 | 仅当启动基于 Windows 的 16 位应用程序时，此标志才有效。 如果设置，新进程将在专用虚拟 DOS 计算机 (VDM) 中运行。 默认情况下，所有基于 Windows 的 16 位应用程序在单个共享 VDM 中作为线程运行。 单独运行的优点是崩溃仅终止单个 VDM;在其他 VM 中运行的任何其他程序继续正常运行。 此外，在单独的 VM 中运行的基于 Windows 的 16 位应用程序具有单独的输入队列。 这意味着，如果一个应用程序暂时停止响应，则单独的 VM 中的应用程序将继续接收输入。 单独运行的缺点是这样做所需的内存要多得多。 仅当用户请求 16 位应用程序应在其自己的 VDM 中运行时才应使用此标志。 | | **CREATE\_SHARED\_WOW\_VDM**  0x00001000 | 仅当启动基于 Windows 的 16 位应用程序时，标志才有效。 如果 WIN.INI 的 Windows 部分中的 DefaultSeparateVDM 开关为 **TRUE**，则此标志将替代该开关。 新进程在共享的虚拟 DOS 计算机中运行。 | | **CREATE\_SUSPENDED**  0x00000004 | 新进程的主线程处于挂起状态创建，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。 | | **CREATE\_UNICODE\_ENVIRONMENT**  0x00000400 | 如果设置了此标志，则 lpEnvironment 指向的环境块使用 Unicode 字符。 否则，环境块使用 ANSI 字符。 | | **DEBUG\_ONLY\_THIS\_PROCESS**  0x00000002 | 调用线程启动并调试新进程。 它可以使用 [**WaitForDebugEvent**](https://learn.microsoft.com/zh-cn/windows/win32/api/debugapi/nf-debugapi-waitfordebugevent) 函数接收所有相关的调试事件。 | | **DEBUG\_PROCESS**  0x00000001 | 调用线程启动并调试新进程以及新进程创建的所有子进程。 它可以使用 [**WaitForDebugEvent**](https://learn.microsoft.com/zh-cn/windows/win32/api/debugapi/nf-debugapi-waitfordebugevent) 函数接收所有相关的调试事件。 使用 **DEBUG\_PROCESS** 的进程将成为调试链的根。 这会一直持续到使用 **DEBUG\_PROCESS** 创建链中的另一个进程。 如果此标志与 **DEBUG\_ONLY\_THIS\_PROCESS** 结合使用，则调用方仅调试新进程，而不调试任何子进程。 | | **DETACHED\_PROCESS**  0x00000008 | 对于控制台进程，新进程不会继承其父级的控制台 (默认) 。 新进程可以在以后调用 [**AllocConsole**](https://learn.microsoft.com/zh-cn/windows/console/allocconsole) 函数来创建控制台。 有关详细信息，请参阅 [创建控制台](https://learn.microsoft.com/zh-cn/windows/console/creation-of-a-console)。 此值不能与 **CREATE\_NEW\_CONSOLE** 一起使用。 | | **EXTENDED\_STARTUPINFO\_PRESENT**  0x00080000 | 使用扩展启动信息创建进程; lpStartupInfo 参数指定 [**STARTUPINFOEX**](https://learn.microsoft.com/zh-cn/windows/desktop/api/WinBase/ns-winbase-startupinfoexa) 结构。 **Windows Server 2003 和 Windows XP：** 不支持此值。 | | **INHERIT\_PARENT\_AFFINITY**  0x00010000 | 进程继承其父级的相关性。 如果父进程在多个 [处理器组中](https://learn.microsoft.com/zh-cn/windows/win32/procthread/processor-groups)具有线程，则新进程将继承父进程正在使用的任意组的组相对相关性。 **Windows Server 2008、Windows Vista、Windows Server 2003 和 Windows XP：** 不支持此值。 |   此参数还控制新进程的优先级类，该类用于确定进程线程的计划优先级。 有关值的列表，请参阅 [GetPriorityClass](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getpriorityclass)。 如果未指定任何优先级类标志，则优先级类默认为 **NORMAL\_PRIORITY\_CLASS** ，除非创建过程的优先级类 **IDLE\_PRIORITY\_CLASS** 或 **BELOW\_NORMAL\_PRIORITY\_CLASS**。 在这种情况下，子进程接收调用进程的默认优先级类。  如果 dwCreationFlags 参数的值为 0：   * 进程同时继承调用方和父级控制台的错误模式。 * 假定新进程的环境块包含 ANSI 字符 (请参阅 lpEnvironment 参数以获取) 的其他信息。 * 基于 16 位 Windows 的应用程序 (VDM) 在共享虚拟 DOS 计算机中运行。   [in, optional] lpEnvironment  指向新进程的环境块的指针。 如果此参数为 **NULL**，则新进程使用调用进程的 环境。  环境块由以 null 结尾的字符串的以 null 结尾的块组成。 每个字符串采用以下格式：  *名字*=*value*\0  由于等号用作分隔符，因此不得在环境变量的名称中使用。  环境块可以包含 Unicode 或 ANSI 字符。 如果 *lpEnvironment* 指向的环境块包含 Unicode 字符，请确保 *dwCreationFlags* 包含 **CREATE\_UNICODE\_ENVIRONMENT**。  如果进程的环境块的总大小超过 32,767 个字符，则此函数的 **ANSI 版本 CreateProcessA** 将失败。  请注意，ANSI 环境块以两个零字节结尾：一个字节用于最后一个字符串，另一个字节用于终止该块。 Unicode 环境块以四个零字节结尾：两个字节表示最后一个字符串，另外两个字节终止该块。  [in, optional] lpCurrentDirectory  进程当前目录的完整路径。 字符串还可以指定 UNC 路径。  如果此参数为 **NULL**，则新进程将具有与调用进程相同的当前驱动器和目录。 (此功能主要用于需要启动应用程序并指定其初始驱动器和工作目录的 shell。)  [in] lpStartupInfo  指向 [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 或 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 结构的指针。  若要设置扩展属性，请使用 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 结构，并在 *dwCreationFlags* 参数中指定EXTENDED\_STARTUPINFO\_PRESENT。  当不再需要 [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 或 [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa) 中的句柄时，必须使用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭它们。  **重要**  调用方负责确保 [**STARTUPINFO**](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 中的标准句柄字段包含有效的句柄值。 即使 **dwFlags** 成员指定了STARTF\_USESTDHANDLES，这些字段也会保持不变地复制到子进程 **，而无需验证**。 不正确的值可能会导致子进程出现错误或崩溃。 使用 [**应用程序验证程序**](https://learn.microsoft.com/zh-cn/windows-hardware/drivers/devtest/application-verifier) 运行时验证工具检测无效句柄。  [out] lpProcessInformation  指向接收有关新进程的标识信息的 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 结构的指针。  当不再需要 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 中的句柄时，必须使用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 关闭它们。 返回值 如果该函数成功，则返回值为非零值。  如果函数失败，则返回值为零。 要获得更多的错误信息，请调用 GetLastError。  请注意， 函数在进程完成初始化之前返回 。 如果找不到所需的 DLL 或无法初始化，则进程将终止。 若要获取进程的终止状态，请调用 [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)。 注解 为进程分配了一个进程标识符。 标识符在进程终止之前有效。 它可用于标识进程，或在 [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess) 函数中指定以打开进程的句柄。 进程中的初始线程也分配有线程标识符。 可以在 [OpenThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthread) 函数中指定它以打开线程的句柄。 标识符在线程终止之前有效，可用于唯一标识系统中的线程。 这些标识符在 [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information) 结构中返回。  操作系统提供给进程的命令行中可执行文件的名称不一定与调用进程提供给 **CreateProcess** 函数的命令行中的名称相同。 操作系统可能会在未提供完全限定路径的可执行文件名称前面添加一个完全限定的路径。  调用线程可以使用 [WaitForInputIdle](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-waitforinputidle) 函数等待，直到新进程完成其初始化，并且正在等待没有挂起输入的用户输入。 这对于父进程和子进程之间的同步非常有用，因为 **CreateProcess** 返回时不会等待新进程完成其初始化。 例如，创建过程会在尝试查找与新进程关联的窗口之前使用 **WaitForInputIdle** 。  关闭进程的首选方法是使用 [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess) 函数，因为此函数会向附加到进程的所有 DLL 发送即将终止的通知。 关闭进程的其他方法不会通知附加的 DLL。 请注意，当线程调用 **ExitProcess** 时，进程的其他线程将终止，没有机会执行任何其他代码 (包括附加 DLL 的线程终止代码) 。 有关详细信息，请参阅 [终止进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/terminating-a-process)。  父进程可以在进程创建期间直接更改子进程的环境变量。 这是一个进程可以直接更改另一个进程的环境设置的唯一情况。 有关详细信息，请参阅 [更改环境变量](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/changing-environment-variables)。  如果应用程序提供环境块，则系统驱动器的当前目录信息不会自动传播到新进程。 例如，有一个名为 =C： 的环境变量，其值为驱动器 C 上的当前目录。应用程序必须手动将当前目录信息传递给新进程。 为此，应用程序必须显式创建这些环境变量字符串， (按字母顺序对它们进行排序，因为系统使用排序的环境) ，并将其放入环境块中。 通常，由于环境块排序顺序，它们将位于环境块的前面。  获取驱动器 X 的当前目录信息的一种方法是进行以下调用： GetFullPathName("X:", ...)。 这避免了应用程序必须扫描环境块。 如果返回的完整路径为 X：，则无需将该值作为环境数据传递，因为根目录是新进程的驱动器 X 的默认当前目录。  使用指定**CREATE\_NEW\_PROCESS\_GROUP**创建进程时，将代表新进程隐式调用 [SetConsoleCtrlHandler](https://learn.microsoft.com/zh-cn/windows/console/setconsolectrlhandler) (**NULL，TRUE**) ;这意味着新进程已禁用 CTRL+C。 这允许 shell 自行处理 CTRL+C，并有选择地将该信号传递给子进程。 CTRL+BREAK 未禁用，可用于中断进程/进程组。  默认情况下，将 **TRUE** 作为 *bInheritHandles* 参数的值传递会导致新进程继承所有可继承句柄。 如果应用程序同时从多个线程创建进程，但希望每个进程继承不同的句柄，这可能会产生问题。 应用程序可以将 [UpdateProcThreadAttributeList](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-updateprocthreadattribute) 函数与 **PROC\_THREAD\_ATTRIBUTE\_HANDLE\_LIST** 参数一起使用，以提供要由特定进程继承的句柄列表。 安全备注 第一个参数 *lpApplicationName* 可以为 **NULL**，在这种情况下，可执行文件名称必须位于 *lpCommandLine* 指向的空格分隔字符串中。 如果可执行文件或路径名称中有空格，则存在运行其他可执行文件的风险，因为函数分析空格的方式。 下面的示例很危险，因为函数将尝试运行“Program.exe”（如果存在），而不是运行“MyApp.exe”。  syntax复制  LPTSTR szCmdline = \_tcsdup(TEXT("C:\\Program Files\\MyApp -L -S"));  CreateProcess(NULL, szCmdline, /\* ... \*/);  如果恶意用户在系统上创建名为“Program.exe”的应用程序，则使用 Program Files 目录错误地调用 **CreateProcess** 的任何程序都将运行此应用程序而不是预期的应用程序。  若要避免此问题，请不要为 *lpApplicationName* 传递 **NULL**。 如果确实为 *lpApplicationName* 传递 **NULL**，请在 *lpCommandLine* 中的可执行路径周围使用引号，如以下示例所示。  syntax复制  LPTSTR szCmdline[] = \_tcsdup(TEXT("\"C:\\Program Files\\MyApp\" -L -S"));  CreateProcess(NULL, szCmdline, /\*...\*/); 示例 有关示例，请参阅 [创建进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-processes)。  **备注**  processthreadsapi.h 标头将 CreateProcess 定义为别名，该别名根据 UNICODE 预处理器常量的定义自动选择此函数的 ANSI 或 Unicode 版本。 将非特定编码别名与非非特定编码的代码混合使用可能会导致不匹配，从而导致编译或运行时错误。 有关详细信息，请参阅 [**函数原型的约定**](https://learn.microsoft.com/zh-cn/windows/win32/intl/conventions-for-function-prototypes)。 要求  |  |  | | --- | --- | | **标头** | processthreadsapi.h (包括 Windows Server 2003、Windows Vista、Windows 7、Windows Server 2008 Windows Server 2008 R2) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle)  [ShellExecuteA](https://learn.microsoft.com/zh-cn/windows/win32/api/shellapi/nf-shellapi-shellexecutea)  [CreateProcessAsUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasusera)  [CreateProcessWithLogonW](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createprocesswithlogonw)  [ExitProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-exitprocess)  [GetCommandLine](https://learn.microsoft.com/zh-cn/windows/desktop/api/processenv/nf-processenv-getcommandlinea)  [GetEnvironmentStrings](https://learn.microsoft.com/zh-cn/windows/desktop/api/rrascfg/nf-rrascfg-ieapproviderconfig-initialize)  [GetExitCodeProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess)  [GetFullPathName](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-getfullpathnamea)  [GetStartupInfo](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getstartupinfow)  [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess)  [PROCESS\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-process_information)  [进程和线程函数](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-and-thread-functions)  [进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/child-processes)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [STARTUPINFO](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/ns-processthreadsapi-startupinfoa)  [STARTUPINFOEX](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/ns-winbase-startupinfoexa)  [SetErrorMode](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-seterrormode)  [TerminateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-terminateprocess)  [WaitForInputIdle](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-waitforinputidle) |

# 演练

## 今天这个项目比较特别，是一个MFC对话框程序和一个命令行程序用管道进行通信，是一个单管道的，使用匿名管道

|  |
| --- |
|  |

### 匿名管道只能用于父子进程之间的通信

## 1.新建一个MFC对话框应用程序，取名Lesson70-pipe-comm，把默认的控件删除，然后重写PreTranslateMessage函数

|  |
| --- |
|  |
|  |

## 2.1.对话框给布局如图

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

## 2.2.为了编程方便，我们给上面的小编辑框绑定一个成员变量，使他能够获得输入焦点

|  |
| --- |
|  |
|  |

## 3.由于cmd窗口是使用ansi的，所以我们需要打开项目的属性，把字符集改为多字节字符集

|  |
| --- |
|  |

## 4.双击执行按钮，进入点击事件函数，我们先输入代码

|  |
| --- |
| void CLesson70pipecommDlg::OnBnClickedBtnRun()  {  // TODO: 在此添加控件通知处理程序代码  BOOL bCreate;  STARTUPINFO si={0};  si.cb = sizeof(STARTUPINFO);  si.wShowWindow = TRUE;  si.dwFlags = STARTF\_USESHOWWINDOW;  PROCESS\_INFORMATION pi = {0};  TCHAR szCmd[MAX\_PATH\*2] = \_T("cmd");  bCreate = CreateProcess(NULL,szCmd,NULL,NULL,TRUE,CREATE\_NEW\_CONSOLE,NULL,NULL,&si,&pi);  CloseHandle(pi.hProcess);//因为是命令行窗口，不需要句柄  CloseHandle(pi.hThread);//因为是命令行窗口，不需要句柄  if(!bCreate)  {  MessageBox(\_T("创建进程失败"));  return;  }  } |

### 然后编译运行程序，点击创建按钮效果如下

|  |
| --- |
|  |

## 5.我们来添加创建管道的代码

|  |
| --- |
| void CLesson70pipecommDlg::OnBnClickedBtnRun()  {  // TODO: 在此添加控件通知处理程序代码  HANDLE hRead,hWrite;//管道的读写句柄  SECURITY\_ATTRIBUTES sa={0};//管道的安全属性.  sa.nLength = sizeof(SECURITY\_ATTRIBUTES);  sa.bInheritHandle = TRUE;  sa.lpSecurityDescriptor = NULL;  BOOL bCreate;  //创建管道  bCreate = CreatePipe(&hRead,&hWrite,&sa,0);//最后一个参数为0表示我们要使用默认的大小  if(!bCreate)  {  MessageBox(\_T("创建管道失败"));  return;  }  STARTUPINFO si={0};  si.cb = sizeof(STARTUPINFO);  si.wShowWindow = TRUE;  si.dwFlags = STARTF\_USESHOWWINDOW;  PROCESS\_INFORMATION pi = {0};  TCHAR szCmd[MAX\_PATH\*2] = \_T("cmd");  bCreate = CreateProcess(NULL,szCmd,NULL,NULL,TRUE,CREATE\_NEW\_CONSOLE,NULL,NULL,&si,&pi);  CloseHandle(pi.hProcess);//因为是命令行窗口，不需要句柄  CloseHandle(pi.hThread);//因为是命令行窗口，不需要句柄  if(!bCreate)  {  MessageBox(\_T("创建进程失败"));  return;  }  } |

### 编译运行程序，没有弹出创建管道失败消息，说明创建成功

|  |
| --- |
|  |

## 6.然后我们来给命令窗口设输入句柄和输出句柄，把命令窗口的输出接到管道的写入端，然后需要隐藏命令窗口，并且需要修改命令参数，不能用cmd了,按钮点击的完整代码如下

|  |
| --- |
| void CLesson70pipecommDlg::OnBnClickedBtnRun()  {  // TODO: 在此添加控件通知处理程序代码  CString strCmd;//保存命令参数  GetDlgItemText(IDC\_EDIT\_CMD,strCmd);  if(strCmd.GetLength()==0)  {  MessageBox(\_T("请输入命令行参数,前面需要留有空格"));  m\_edit\_cmd.SetFocus();  return;  }  HANDLE hRead,hWrite;//管道的读写句柄  SECURITY\_ATTRIBUTES sa={0};//管道的安全属性.  sa.nLength = sizeof(SECURITY\_ATTRIBUTES);  sa.bInheritHandle = TRUE;  sa.lpSecurityDescriptor = NULL;  BOOL bCreate;  //创建管道  bCreate = CreatePipe(&hRead,&hWrite,&sa,0);//最后一个参数为0表示我们要使用默认的大小  if(!bCreate)  {  MessageBox(\_T("创建管道失败"));  return;  }  STARTUPINFO si={0};//可以在这里添加启动进程的输入  si.cb = sizeof(STARTUPINFO);  si.hStdInput = GetStdHandle(STD\_INPUT\_HANDLE);//设置进程的输入  si.hStdOutput = si.hStdError = hWrite;//设置进程的输出到管道的写入端  //si.wShowWindow = TRUE;  si.wShowWindow = SW\_HIDE; //隐藏窗口  //si.dwFlags = STARTF\_USESHOWWINDOW;  si.dwFlags = STARTF\_USESHOWWINDOW|STARTF\_USESTDHANDLES;//添加使用标准句柄功能  PROCESS\_INFORMATION pi = {0};  //TCHAR szCmd[MAX\_PATH\*2] = \_T("cmd");  TCHAR szCmd[MAX\_PATH\*2] = {0};  //从第一个编辑框里面获取命令行参数  GetDlgItemText(IDC\_EDIT\_CMD,szCmd,MAX\_PATH\*2);  //MessageBox(szCmd);  //lstrcpy(szCmd,strCmd);  bCreate = CreateProcess(NULL,szCmd,NULL,NULL,TRUE,0,NULL,NULL,&si,&pi);  //bCreate = CreateProcess(NULL,szCmd,NULL,NULL,TRUE,CREATE\_NEW\_CONSOLE,NULL,NULL,&si,&pi);//因为上面已经隐藏了所以这里要改为0  CloseHandle(pi.hProcess);//因为是命令行窗口，不需要句柄  CloseHandle(pi.hThread);//因为是命令行窗口，不需要句柄  if(!bCreate)  {  CloseHandle(hRead);//创建进程失败需要关闭管道句柄  CloseHandle(hWrite);//创建进程失败需要关闭管道句柄  MessageBox(\_T("创建进程失败"));  return;  }  else  {  CString str;  //读取数据之前,需要先关闭管道的写端  CloseHandle(hWrite);  TCHAR szBuffer[4096];    DWORD dwRead;//读取到的字节数  while(TRUE)  {  memset(szBuffer,0,4096);  if(!ReadFile(hRead,szBuffer,4096,&dwRead,NULL))//可以像读取文件一样读取管道  {  break;//读取完毕就退出循环  }  str +=szBuffer;    }  SetDlgItemText(IDC\_EDIT\_RESULT,str);  CloseHandle(hRead);//读取完成需要关闭管道读取句柄    }  } |

## 测试了一下,貌似对很多命令都创建进程失败,只有ipconfig和ping 127.0.0.1和net user成功

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

# 这一节的学习到此为止,完整代码如下

## Lesson70-pipe-commDlg.h

|  |
| --- |
| // Lesson70-pipe-commDlg.h : 头文件  //  #pragma once  #include "afxwin.h"  // CLesson70pipecommDlg 对话框  class CLesson70pipecommDlg : public CDialogEx  {  // 构造  public:  CLesson70pipecommDlg(CWnd\* pParent = NULL); // 标准构造函数  // 对话框数据  enum { IDD = IDD\_LESSON70PIPECOMM\_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:  virtual BOOL PreTranslateMessage(MSG\* pMsg);  afx\_msg void OnBnClickedBtnRun();  // 绑定小编辑框的成员变量  CEdit m\_edit\_cmd;  }; |

## Lesson70-pipe-commDlg.cpp

|  |
| --- |
| // Lesson70-pipe-commDlg.cpp : 实现文件  //  #include "stdafx.h"  #include "Lesson70-pipe-comm.h"  #include "Lesson70-pipe-commDlg.h"  #include "afxdialogex.h"  #ifdef \_DEBUG  #define new DEBUG\_NEW  #endif  // 用于应用程序“关于”菜单项的 CAboutDlg 对话框  class CAboutDlg : public CDialogEx  {  public:  CAboutDlg();  // 对话框数据  enum { IDD = IDD\_ABOUTBOX };  protected:  virtual void DoDataExchange(CDataExchange\* pDX); // DDX/DDV 支持  // 实现  protected:  DECLARE\_MESSAGE\_MAP()  };  CAboutDlg::CAboutDlg() : CDialogEx(CAboutDlg::IDD)  {  }  void CAboutDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  }  BEGIN\_MESSAGE\_MAP(CAboutDlg, CDialogEx)  END\_MESSAGE\_MAP()  // CLesson70pipecommDlg 对话框  CLesson70pipecommDlg::CLesson70pipecommDlg(CWnd\* pParent /\*=NULL\*/)  : CDialogEx(CLesson70pipecommDlg::IDD, pParent)  {  m\_hIcon = AfxGetApp()->LoadIcon(IDR\_MAINFRAME);  }  void CLesson70pipecommDlg::DoDataExchange(CDataExchange\* pDX)  {  CDialogEx::DoDataExchange(pDX);  DDX\_Control(pDX, IDC\_EDIT\_CMD, m\_edit\_cmd);  }  BEGIN\_MESSAGE\_MAP(CLesson70pipecommDlg, CDialogEx)  ON\_WM\_SYSCOMMAND()  ON\_WM\_PAINT()  ON\_WM\_QUERYDRAGICON()  ON\_BN\_CLICKED(IDC\_BTN\_RUN, &CLesson70pipecommDlg::OnBnClickedBtnRun)  END\_MESSAGE\_MAP()  // CLesson70pipecommDlg 消息处理程序  BOOL CLesson70pipecommDlg::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 CLesson70pipecommDlg::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 CLesson70pipecommDlg::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 CLesson70pipecommDlg::OnQueryDragIcon()  {  return static\_cast<HCURSOR>(m\_hIcon);  }  BOOL CLesson70pipecommDlg::PreTranslateMessage(MSG\* pMsg)  {  // TODO: 在此添加专用代码和/或调用基类  if(pMsg->message==WM\_KEYDOWN)  {  switch(pMsg->wParam)  {  case VK\_RETURN:  return TRUE;  case VK\_ESCAPE:  return TRUE;  }  }  return CDialogEx::PreTranslateMessage(pMsg);  }  void CLesson70pipecommDlg::OnBnClickedBtnRun()  {  // TODO: 在此添加控件通知处理程序代码  CString strCmd;//保存命令参数  GetDlgItemText(IDC\_EDIT\_CMD,strCmd);  if(strCmd.GetLength()==0)  {  MessageBox(\_T("请输入命令行参数,前面需要留有空格"));  m\_edit\_cmd.SetFocus();  return;  }  HANDLE hRead,hWrite;//管道的读写句柄  SECURITY\_ATTRIBUTES sa={0};//管道的安全属性.  sa.nLength = sizeof(SECURITY\_ATTRIBUTES);  sa.bInheritHandle = TRUE;  sa.lpSecurityDescriptor = NULL;  BOOL bCreate;  //创建管道  bCreate = CreatePipe(&hRead,&hWrite,&sa,0);//最后一个参数为0表示我们要使用默认的大小  if(!bCreate)  {  MessageBox(\_T("创建管道失败"));  return;  }  STARTUPINFO si={0};//可以在这里添加启动进程的输入  si.cb = sizeof(STARTUPINFO);  si.hStdInput = GetStdHandle(STD\_INPUT\_HANDLE);//设置进程的输入  si.hStdOutput = si.hStdError = hWrite;//设置进程的输出到管道的写入端  //si.wShowWindow = TRUE;  si.wShowWindow = SW\_HIDE; //隐藏窗口  //si.dwFlags = STARTF\_USESHOWWINDOW;  si.dwFlags = STARTF\_USESHOWWINDOW|STARTF\_USESTDHANDLES;//添加使用标准句柄功能  PROCESS\_INFORMATION pi = {0};  //TCHAR szCmd[MAX\_PATH\*2] = \_T("cmd");  TCHAR szCmd[MAX\_PATH\*2] = {0};  //从第一个编辑框里面获取命令行参数  GetDlgItemText(IDC\_EDIT\_CMD,szCmd,MAX\_PATH\*2);  //MessageBox(szCmd);  //lstrcpy(szCmd,strCmd);  bCreate = CreateProcess(NULL,szCmd,NULL,NULL,TRUE,0,NULL,NULL,&si,&pi);  CloseHandle(pi.hProcess);//因为是命令行窗口，不需要句柄  CloseHandle(pi.hThread);//因为是命令行窗口，不需要句柄  if(!bCreate)  {  CloseHandle(hRead);//创建进程失败需要关闭管道句柄  CloseHandle(hWrite);//创建进程失败需要关闭管道句柄  MessageBox(\_T("创建进程失败"));  return;  }  else  {  CString str;  //读取数据之前,需要先关闭管道的写端  CloseHandle(hWrite);  TCHAR szBuffer[4096];    DWORD dwRead;//读取到的字节数  while(TRUE)  {  memset(szBuffer,0,4096);  if(!ReadFile(hRead,szBuffer,4096,&dwRead,NULL))//可以像读取文件一样读取管道  {  break;//读取完毕就退出循环  }  str +=szBuffer;    }  SetDlgItemText(IDC\_EDIT\_RESULT,str);  CloseHandle(hRead);//读取完成需要关闭管道读取句柄    }  } |

# 扩展1,这个程序可以改进一下,不需要每一次点击按钮都创建一个新进程,此时需要使用双管道.

# 扩展2,可以学习一下双向匿名管道和命名管道的使用