# [命名管道 - Win32 apps | Microsoft Learn](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipes)

# 命名管道

命名管道是用于管道服务器与一个或多个管道客户端之间通信的命名、单向或双工管道。 命名管道的所有实例共享相同的管道名称，但每个实例都有自己的缓冲区和句柄，并为客户端/服务器通信提供单独的管道。 通过使用实例，多个管道客户端可以同时使用同一个命名管道。

任何进程都可以访问命名管道，受到安全检查，使命名管道成为相关或不相关进程之间通信的简单形式。

任何进程都可以充当服务器和客户端，使对等通信成为可能。 在此处使用，术语管道服务器是指创建命名管道的进程，术语管道客户端是指连接到命名管道实例的进程。 用于实例化命名管道的服务器端函数是 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea)。 用于接受连接的服务器端函数是 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe)。 客户端进程使用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数连接到命名管道。

命名管道可用于在同一台计算机上的进程之间或网络上不同计算机上的进程之间提供通信。 如果服务器服务正在运行，则所有命名管道都可以远程访问。 如果打算仅在本地使用命名管道，请拒绝对 NT AUTHORITY\NETWORK 的访问或切换到本地 RPC。

# 管道名称

每个命名管道都有一个唯一的名称，用于将其与系统命名对象列表中的其他命名管道区分开来。 管道服务器在调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数以创建命名管道的一个或多个实例时指定管道的名称。 管道客户端在调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数以连接到命名管道实例时指定管道名称。

在 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea)、[**WaitNamedPipe 或 CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-waitnamedpipea) 函数中指定管道的名称时，请使用以下形式：

\\ ServerName\pipe\PipeName

其中 ，ServerName 是远程计算机的名称或句点，以指定本地计算机。 PipeName 指定的管道名称字符串可以包含反斜杠以外的任何字符，包括数字和特殊字符。 整个管道名称字符串的长度最多为 256 个字符。 管道名称不区分大小写。

管道服务器不能在另一台计算机上创建管道，因此 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 必须为服务器名称使用句点，如以下示例所示。

\\.\pipe\PipeName

管道服务器可以向其管道客户端提供管道名称，以便它们可以连接到管道。 管道客户端从某些永久性源（例如注册表项、文件或其他应用程序）中发现管道名称。 否则，客户端必须在编译时知道管道名称。

## [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 创建命名管道的实例，并返回后续管道操作的句柄。 命名管道服务器进程使用此函数创建特定命名管道的第一个实例并建立其基本属性，或创建现有命名管道的新实例。 语法 C++复制  HANDLE CreateNamedPipeA(  [in] LPCSTR lpName,  [in] DWORD dwOpenMode,  [in] DWORD dwPipeMode,  [in] DWORD nMaxInstances,  [in] DWORD nOutBufferSize,  [in] DWORD nInBufferSize,  [in] DWORD nDefaultTimeOut,  [in, optional] LPSECURITY\_ATTRIBUTES lpSecurityAttributes  ); 参数 [in] lpName  唯一的管道名称。 此字符串必须具有以下形式：  \\.\pipe\*pipename*  名称的 pipename 部分可以包含反斜杠以外的任何字符，包括数字和特殊字符。 整个管道名称字符串的长度最多为 256 个字符。 管道名称不区分大小写。  [in] dwOpenMode  打开模式。  如果 *dwOpenMode* 指定除 0 以外的任何内容或下表中列出的标志，则函数将失败。  此参数必须指定以下管道访问模式之一。 必须为管道的每个实例指定相同的模式。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **PIPE\_ACCESS\_DUPLEX**  0x00000003 | 管道是双向的;服务器和客户端进程都可以从管道读取和写入管道。 此模式为服务器提供 **对管道的等效GENERIC\_READ** 和 **GENERIC\_WRITE** 访问权限。 当客户端使用 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数连接到管道时，可以指定**GENERIC\_READ**或**GENERIC\_WRITE**，或者同时指定这两者。 | | **PIPE\_ACCESS\_INBOUND**  0x00000001 | 管道中的数据流仅从客户端流向服务器。 此模式为服务器提供对管道 **的等效GENERIC\_READ** 访问权限。 连接到管道时，客户端必须指定 **GENERIC\_WRITE** 访问。 如果客户端必须通过调用 [GetNamedPipeInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-getnamedpipeinfo) 或 [GetNamedPipeHandleState](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-getnamedpipehandlestatea) 函数来读取管道设置，则客户端必须在连接到管道时指定 **GENERIC\_WRITE** 并 **FILE\_READ\_ATTRIBUTES** 访问。 | | **PIPE\_ACCESS\_OUTBOUND**  0x00000002 | 管道中的数据流仅从服务器流向客户端。 此模式为服务器提供等效于 **GENERIC\_WRITE** 访问管道的权限。 连接到管道时，客户端必须指定 **GENERIC\_READ** 访问。 如果客户端必须通过调用 [SetNamedPipeHandleState](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数更改管道设置，则客户端必须在连接到管道时指定 **GENERIC\_READ** 和 **FILE\_WRITE\_ATTRIBUTES** 访问。 |     此参数还可以包括以下一个或多个标志，这些标志启用写通模式和重叠模式。 对于同一管道的不同实例，这些模式可能不同。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **FILE\_FLAG\_FIRST\_PIPE\_INSTANCE**  0x00080000 | 如果尝试使用此标志创建管道的多个实例，则创建第一个实例会成功，但创建下一个实例会失败并 **ERROR\_ACCESS\_DENIED**。 | | **FILE\_FLAG\_WRITE\_THROUGH**  0x80000000 | 已启用写通模式。 此模式仅影响对字节类型管道的写入操作，并且仅当客户端和服务器进程位于不同计算机上时才影响写入操作。 如果启用此模式，则写入命名管道的函数不会返回，直到写入的数据通过网络传输并且位于远程计算机上的管道缓冲区中。 如果未启用此模式，系统会通过缓冲数据，直到最小字节数累积或最长时间结束，来提高网络操作的效率。 | | **FILE\_FLAG\_OVERLAPPED**  0x40000000 | 已启用重叠模式。 如果启用此模式，执行读取、写入和连接操作（可能需要很长时间才能完成）的函数会立即返回。 此模式使启动操作的线程能够在后台执行耗时操作时执行其他操作。 例如，在重叠模式下，线程可以处理同时输入和输出 (多个管道实例上的 I/O) 操作，或在同一管道句柄上同时执行读取和写入操作。 如果未启用重叠模式，则在操作完成之前，对管道句柄执行读取、写入和连接操作的函数不会返回。 [ReadFileEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfileex) 和 [WriteFileEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefileex) 函数只能与重叠模式下的管道句柄一起使用。 [ReadFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、[WriteFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile)、[ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 和 [TransactNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 函数可以同步执行，也可以作为重叠操作执行。 |     此参数可以包括以下安全访问模式的任意组合。 对于同一管道的不同实例，这些模式可能不同。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **WRITE\_DAC**  0x00040000L | 调用方将具有对命名管道的任意访问控制列表 (ACL) 的写入访问权限。 | | **WRITE\_OWNER**  0x00080000L | 调用方将具有对命名管道所有者的写入访问权限。 | | **ACCESS\_SYSTEM\_SECURITY**  0x01000000L | 调用方将具有对命名管道的 SACL 的写入访问权限。 有关详细信息，请参阅[访问控制Lists (ACL)](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-control-lists)和 [SACL 访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/sacl-access-right)。 |   [in] dwPipeMode  管道模式。  如果 *dwPipeMode* 指定除 0 以外的任何内容或下表中列出的标志，则函数将失败。  可以指定以下类型模式之一。 必须为管道的每个实例指定相同的类型模式。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **PIPE\_TYPE\_BYTE**  0x00000000 | 数据作为字节流写入管道。 此模式不能与PIPE\_READMODE\_MESSAGE一起使用。 管道不区分在不同写入操作期间写入的字节。 | | **PIPE\_TYPE\_MESSAGE**  0x00000004 | 数据作为消息流写入管道。 管道将每次写入操作期间写入的字节视为消息单元。 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数在未完全读取消息时返回**ERROR\_MORE\_DATA**。 此模式可与 **PIPE\_READMODE\_MESSAGE** 或 **PIPE\_READMODE\_BYTE** 一起使用。 |     可以指定以下读取模式之一。 同一管道的不同实例可以指定不同的读取模式。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **PIPE\_READMODE\_BYTE**  0x00000000 | 数据作为字节流从管道中读取。 此模式可用于 **PIPE\_TYPE\_MESSAGE** 或 **PIPE\_TYPE\_BYTE**。 | | **PIPE\_READMODE\_MESSAGE**  0x00000002 | 数据作为消息流从管道中读取。 仅当还指定 **了PIPE\_TYPE\_MESSAGE** 时，才能使用此模式。 |     可以指定以下等待模式之一。 同一管道的不同实例可以指定不同的等待模式。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **PIPE\_WAIT**  0x00000000 | 阻止模式已启用。 在 [ReadFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、 [WriteFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 或 [ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数中指定管道句柄时，在有要读取的数据、写入所有数据或连接客户端之前，操作不会完成。 在某些情况下，使用此模式可能意味着无限期等待客户端进程执行操作。 | | **PIPE\_NOWAIT**  0x00000001 | 启用了非阻止模式。 在此模式下， [ReadFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、 [WriteFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 和 [ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 始终立即返回。  请注意，支持非阻止模式以实现与 Microsoft LAN Manager 版本 2.0 的兼容性，并且不应用于通过命名管道实现异步 I/O。 有关异步管道 I/O 的详细信息，请参阅 [同步和重叠输入和输出](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/synchronous-and-overlapped-input-and-output)。 |     可以指定以下远程客户端模式之一。 同一管道的不同实例可以指定不同的远程客户端模式。  展开表   |  |  | | --- | --- | | **模型** | **含义** | | **PIPE\_ACCEPT\_REMOTE\_CLIENTS**  0x00000000 | 可以接受来自远程客户端的Connections，并对照管道的安全描述符进行检查。 | | **PIPE\_REJECT\_REMOTE\_CLIENTS**  0x00000008 | 来自远程客户端的Connections将自动被拒绝。 |   [in] nMaxInstances  可为此管道创建的最大实例数。 管道的第一个实例可以指定此值;必须为管道的其他实例指定相同的数字。 可接受的值范围为 1 到 **PIPE\_UNLIMITED\_INSTANCES** (255) 。  如果 **此参数PIPE\_UNLIMITED\_INSTANCES**，则可以创建的管道实例数仅受系统资源的可用性限制。 如果 *nMaxInstances* 大于 **PIPE\_UNLIMITED\_INSTANCES**，则返回值为 INVALID\_HANDLE\_VALUE，GetLastError 返回**ERROR\_INVALID\_PARAMETER**。  [in] nOutBufferSize  要为输出缓冲区保留的字节数。 有关调整命名管道缓冲区大小的讨论，请参阅以下“备注”部分。  [in] nInBufferSize  要为输入缓冲区保留的字节数。 有关调整命名管道缓冲区大小的讨论，请参阅以下“备注”部分。  [in] nDefaultTimeOut  如果 [WaitNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-waitnamedpipea) 函数指定NMPWAIT\_USE\_DEFAULT\_WAIT，则为默认超时值 **（以毫秒为单位**）。 命名管道的每个实例必须指定相同的值。  如果值为零，则默认超时为 50 毫秒。  [in, optional] lpSecurityAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构指定新的命名管道的安全描述符，并确定子进程是否可以继承返回的句柄。 如果 *lpSecurityAttributes* 为 **NULL**，则命名管道将获取默认安全描述符，并且无法继承句柄。 命名管道的默认安全描述符中的 ACL 授予 LocalSystem 帐户、管理员和创建者所有者的完全控制权。 他们还向 Everyone 组的成员和匿名帐户授予读取访问权限。 返回值 如果函数成功，则返回值是命名管道实例的服务器端的句柄。  如果函数失败，则返回值为 INVALID\_HANDLE\_VALUE。 要获得更多的错误信息，请调用 GetLastError。 注解 若要使用 **CreateNamedPipe** 创建命名管道的实例，用户必须具有对命名管道对象的 **FILE\_CREATE\_PIPE\_INSTANCE** 访问权限。 如果创建新的命名管道，则安全属性参数 (ACL) 的访问控制列表定义命名管道的任意访问控制。  命名管道的所有实例必须指定相同的管道类型 (字节类型或消息类型) 、管道访问 (双工、入站或出站) 、实例计数和超时值。 如果使用不同的值，此函数将失败， [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 将返回 **ERROR\_ACCESS\_DENIED**。  客户端进程使用 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [CallNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-callnamedpipea) 函数连接到命名管道。 命名管道的客户端以字节模式启动，即使服务器端处于消息模式也是如此。 为了避免在接收数据时出现问题，请将客户端设置为消息模式。 若要更改管道的模式，管道客户端必须打开具有 **GENERIC\_READ** 的只读管道，并 **FILE\_WRITE\_ATTRIBUTES** 访问权限。  在管道客户端启动之前，管道服务器不应执行阻止读取操作。 否则，可能会出现争用情况。 当初始化代码（如 C 运行时）需要锁定和检查继承的句柄时，通常会发生这种情况。  每次创建命名管道时，系统都会使用非分页池（内核使用的物理内存）创建入站和/或出站缓冲区。 可以创建的管道实例 (以及对象（如线程和进程) ）的数量受可用非分页池的限制。 每个读取或写入请求都需要缓冲区中用于读取或写入数据的空间，以及用于内部数据结构的额外空间。  输入和输出缓冲区大小为公告。 为命名管道的每一端保留的实际缓冲区大小为系统默认值、系统最小值或最大值，或向上舍入到下一个分配边界的指定大小。 指定的缓冲区大小应足够小，以便进程不会耗尽非分页池，但要足够大，足以容纳典型请求。  每当发生管道写入操作时，系统首先会尝试根据管道写入配额为内存充电。 如果剩余的管道写入配额足以满足请求，则写入操作将立即完成。 如果剩余的管道写入配额太小而无法满足请求，系统将尝试使用为进程保留的非分页池来扩展缓冲区以容纳数据。 写入操作将阻塞，直到从管道读取数据，以便释放额外的缓冲区配额。 因此，如果指定的缓冲区大小太小，系统将根据需要增大缓冲区，但缺点是操作会阻塞。 如果操作重叠，则会阻止系统线程;否则，应用程序线程将被阻止。  若要释放命名管道使用的资源，应用程序应始终在不再需要句柄时关闭句柄，这可以通过调用 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数或在与实例句柄关联的进程结束时完成。 请注意，命名管道的实例可能有多个与之关联的句柄。 关闭命名管道实例的最后一个句柄时，始终会删除命名管道的实例。  **Windows 10版本 1709：**管道仅在应用容器中受支持;即，从一个 UWP 进程到另一个属于同一应用的 UWP 进程。 此外，命名管道必须使用语法 \\.\pipe\LOCAL\ 来表示管道名称。 示例 有关示例，请参阅 [多线程管道服务器](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/multithreaded-pipe-server)。 要求  |  |  | | --- | --- | | **标头** | winbase.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe)  [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea)  [管道函数](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipe-functions)  [管道概述](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipes)  [ReadFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)  [ReadFileEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfileex)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [TransactNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe)  [WaitNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-waitnamedpipea)  [WriteFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile)  [WriteFileEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefileex) |

## 实例

此管道服务器可与命名管道客户端中所述的 [管道客户端](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipe-client)一起使用。

C++

#include <windows.h>

#include <stdio.h>

#include <tchar.h>

#include <strsafe.h>

#define BUFSIZE 512

DWORD WINAPI InstanceThread(LPVOID);

VOID GetAnswerToRequest(LPTSTR, LPTSTR, LPDWORD);

int \_tmain(VOID)

{

BOOL fConnected = FALSE;

DWORD dwThreadId = 0;

HANDLE hPipe = INVALID\_HANDLE\_VALUE, hThread = NULL;

LPCTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");

// The main loop creates an instance of the named pipe and

// then waits for a client to connect to it. When the client

// connects, a thread is created to handle communications

// with that client, and this loop is free to wait for the

// next client connect request. It is an infinite loop.

for (;;)

{

\_tprintf( TEXT("\nPipe Server: Main thread awaiting client connection on %s\n"), lpszPipename);

hPipe = CreateNamedPipe(

lpszPipename, // pipe name

PIPE\_ACCESS\_DUPLEX, // read/write access

PIPE\_TYPE\_MESSAGE | // message type pipe

PIPE\_READMODE\_MESSAGE | // message-read mode

PIPE\_WAIT, // blocking mode

PIPE\_UNLIMITED\_INSTANCES, // max. instances

BUFSIZE, // output buffer size

BUFSIZE, // input buffer size

0, // client time-out

NULL); // default security attribute

if (hPipe == INVALID\_HANDLE\_VALUE)

{

\_tprintf(TEXT("CreateNamedPipe failed, GLE=%d.\n"), GetLastError());

return -1;

}

// Wait for the client to connect; if it succeeds,

// the function returns a nonzero value. If the function

// returns zero, GetLastError returns ERROR\_PIPE\_CONNECTED.

fConnected = ConnectNamedPipe(hPipe, NULL) ?

TRUE : (GetLastError() == ERROR\_PIPE\_CONNECTED);

if (fConnected)

{

printf("Client connected, creating a processing thread.\n");

// Create a thread for this client.

hThread = CreateThread(

NULL, // no security attribute

0, // default stack size

InstanceThread, // thread proc

(LPVOID) hPipe, // thread parameter

0, // not suspended

&dwThreadId); // returns thread ID

if (hThread == NULL)

{

\_tprintf(TEXT("CreateThread failed, GLE=%d.\n"), GetLastError());

return -1;

}

else CloseHandle(hThread);

}

else

// The client could not connect, so close the pipe.

CloseHandle(hPipe);

}

return 0;

}

DWORD WINAPI InstanceThread(LPVOID lpvParam)

// This routine is a thread processing function to read from and reply to a client

// via the open pipe connection passed from the main loop. Note this allows

// the main loop to continue executing, potentially creating more threads of

// of this procedure to run concurrently, depending on the number of incoming

// client connections.

{

HANDLE hHeap = GetProcessHeap();

TCHAR\* pchRequest = (TCHAR\*)HeapAlloc(hHeap, 0, BUFSIZE\*sizeof(TCHAR));

TCHAR\* pchReply = (TCHAR\*)HeapAlloc(hHeap, 0, BUFSIZE\*sizeof(TCHAR));

DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;

BOOL fSuccess = FALSE;

HANDLE hPipe = NULL;

// Do some extra error checking since the app will keep running even if this

// thread fails.

if (lpvParam == NULL)

{

printf( "\nERROR - Pipe Server Failure:\n");

printf( " InstanceThread got an unexpected NULL value in lpvParam.\n");

printf( " InstanceThread exitting.\n");

if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);

if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);

return (DWORD)-1;

}

if (pchRequest == NULL)

{

printf( "\nERROR - Pipe Server Failure:\n");

printf( " InstanceThread got an unexpected NULL heap allocation.\n");

printf( " InstanceThread exitting.\n");

if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);

return (DWORD)-1;

}

if (pchReply == NULL)

{

printf( "\nERROR - Pipe Server Failure:\n");

printf( " InstanceThread got an unexpected NULL heap allocation.\n");

printf( " InstanceThread exitting.\n");

if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);

return (DWORD)-1;

}

// Print verbose messages. In production code, this should be for debugging only.

printf("InstanceThread created, receiving and processing messages.\n");

// The thread's parameter is a handle to a pipe object instance.

hPipe = (HANDLE) lpvParam;

// Loop until done reading

while (1)

{

// Read client requests from the pipe. This simplistic code only allows messages

// up to BUFSIZE characters in length.

fSuccess = ReadFile(

hPipe, // handle to pipe

pchRequest, // buffer to receive data

BUFSIZE\*sizeof(TCHAR), // size of buffer

&cbBytesRead, // number of bytes read

NULL); // not overlapped I/O

if (!fSuccess || cbBytesRead == 0)

{

if (GetLastError() == ERROR\_BROKEN\_PIPE)

{

\_tprintf(TEXT("InstanceThread: client disconnected.\n"));

}

else

{

\_tprintf(TEXT("InstanceThread ReadFile failed, GLE=%d.\n"), GetLastError());

}

break;

}

// Process the incoming message.

GetAnswerToRequest(pchRequest, pchReply, &cbReplyBytes);

// Write the reply to the pipe.

fSuccess = WriteFile(

hPipe, // handle to pipe

pchReply, // buffer to write from

cbReplyBytes, // number of bytes to write

&cbWritten, // number of bytes written

NULL); // not overlapped I/O

if (!fSuccess || cbReplyBytes != cbWritten)

{

\_tprintf(TEXT("InstanceThread WriteFile failed, GLE=%d.\n"), GetLastError());

break;

}

}

// Flush the pipe to allow the client to read the pipe's contents

// before disconnecting. Then disconnect the pipe, and close the

// handle to this pipe instance.

FlushFileBuffers(hPipe);

DisconnectNamedPipe(hPipe);

CloseHandle(hPipe);

HeapFree(hHeap, 0, pchRequest);

HeapFree(hHeap, 0, pchReply);

printf("InstanceThread exiting.\n");

return 1;

}

VOID GetAnswerToRequest( LPTSTR pchRequest,

LPTSTR pchReply,

LPDWORD pchBytes )

// This routine is a simple function to print the client request to the console

// and populate the reply buffer with a default data string. This is where you

// would put the actual client request processing code that runs in the context

// of an instance thread. Keep in mind the main thread will continue to wait for

// and receive other client connections while the instance thread is working.

{

\_tprintf( TEXT("Client Request String:\"%s\"\n"), pchRequest );

// Check the outgoing message to make sure it's not too long for the buffer.

if (FAILED(StringCchCopy( pchReply, BUFSIZE, TEXT("default answer from server") )))

{

\*pchBytes = 0;

pchReply[0] = 0;

printf("StringCchCopy failed, no outgoing message.\n");

return;

}

\*pchBytes = (lstrlen(pchReply)+1)\*sizeof(TCHAR);

}

## [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 连接到消息类型管道 (，并等待管道实例不可用) ，写入管道并从管道读取，然后关闭管道。 语法 C++复制  BOOL CallNamedPipeA(  [in] LPCSTR lpNamedPipeName,  [in] LPVOID lpInBuffer,  [in] DWORD nInBufferSize,  [out] LPVOID lpOutBuffer,  [in] DWORD nOutBufferSize,  [out] LPDWORD lpBytesRead,  [in] DWORD nTimeOut  ); 参数 [in] lpNamedPipeName  管道名称。  [in] lpInBuffer  要写入管道的数据。  [in] nInBufferSize  写入缓冲区的大小（以字节为单位）。  [out] lpOutBuffer  指向接收从管道读取的数据的缓冲区的指针。  [in] nOutBufferSize  读取缓冲区的大小（以字节为单位）。  [out] lpBytesRead  指向变量的指针，该变量接收从管道读取的字节数。  [in] nTimeOut  等待命名管道可用的毫秒数。 除了数值之外，还可以指定以下特殊值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **NMPWAIT\_NOWAIT**  0x00000001 | 不等待命名管道。 如果命名管道不可用，函数将返回错误。 | | **NMPWAIT\_WAIT\_FOREVER**  0xffffffff | 无限期等待。 | | **NMPWAIT\_USE\_DEFAULT\_WAIT**  0x00000000 | 使用对 [CreateNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createnamedpipea) 函数的调用中指定的默认超时。 |  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。  如果服务器进程写入管道的消息长于 *nOutBufferSize*，**则 CallNamedPipe** 返回 [FALSE，GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 返回ERROR\_MORE\_DATA。 消息的其余部分将被丢弃，因为 **CallNamedPipe** 在返回之前会关闭管道的句柄。 注解 如果 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 无法立即) 、**TransactNamedPipe** 和 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数打开管道，则调用 [CallNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 等效于调用 **CreateFile** (或 WaitNamedPipe。 使用访问标志GENERIC\_READ调用 **CreateFile** |GENERIC\_WRITE，以及 **FALSE** 的继承句柄标志。  如果管道是字节类型管道，**则 CallNamedPipe** 将失败。  **Windows 10版本 1709：**管道仅在应用容器中受支持;即，从一个 UWP 进程到属于同一应用的另一个 UWP 进程。 此外，命名管道必须为管道名称使用 语法 \\.\pipe\LOCAL\ 。 示例 有关示例，请参阅 [命名管道上的事务](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/transactions-on-named-pipes)。 要求  |  |  | | --- | --- | | **标头** | winbase.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle)  [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea)  [CreateNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createnamedpipea)  [管道函数](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipe-functions)  [管道概述](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipes)  [TransactNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe)  [WaitNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-waitnamedpipea) |

## 实例:

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <conio.h>  #include <tchar.h>  #define BUFSIZE 512    int \_tmain(int argc, TCHAR \*argv[])  {  HANDLE hPipe;  LPTSTR lpszWrite = TEXT("Default message from client");  TCHAR chReadBuf[BUFSIZE];  BOOL fSuccess;  DWORD cbRead, dwMode;  LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");  if( argc > 1)  {  lpszWrite = argv[1];  }    // Try to open a named pipe; wait for it, if necessary.  while (1)  {  hPipe = CreateFile(  lpszPipename, // pipe name  GENERIC\_READ | // read and write access  GENERIC\_WRITE,  0, // no sharing  NULL, // default security attributes  OPEN\_EXISTING, // opens existing pipe  0, // default attributes  NULL); // no template file    // Break if the pipe handle is valid.  if (hPipe != INVALID\_HANDLE\_VALUE)  break;    // Exit if an error other than ERROR\_PIPE\_BUSY occurs.  if (GetLastError() != ERROR\_PIPE\_BUSY)  {  printf("Could not open pipe\n");  return 0;  }    // All pipe instances are busy, so wait for 20 seconds.  if (! WaitNamedPipe(lpszPipename, 20000) )  {  printf("Could not open pipe\n");  return 0;  }  }    // The pipe connected; change to message-read mode.  dwMode = PIPE\_READMODE\_MESSAGE;  fSuccess = SetNamedPipeHandleState(  hPipe, // pipe handle  &dwMode, // new pipe mode  NULL, // don't set maximum bytes  NULL); // don't set maximum time  if (!fSuccess)  {  printf("SetNamedPipeHandleState failed.\n");  return 0;  }    // Send a message to the pipe server and read the response.  fSuccess = TransactNamedPipe(  hPipe, // pipe handle  lpszWrite, // message to server  (lstrlen(lpszWrite)+1)\*sizeof(TCHAR), // message length  chReadBuf, // buffer to receive reply  BUFSIZE\*sizeof(TCHAR), // size of read buffer  &cbRead, // bytes read  NULL); // not overlapped  if (!fSuccess && (GetLastError() != ERROR\_MORE\_DATA))  {  printf("TransactNamedPipe failed.\n");  return 0;  }    while(1)  {  \_tprintf(TEXT("%s\n"), chReadBuf);  // Break if TransactNamedPipe or ReadFile is successful  if(fSuccess)  break;  // Read from the pipe if there is more data in the message.  fSuccess = ReadFile(  hPipe, // pipe handle  chReadBuf, // buffer to receive reply  BUFSIZE\*sizeof(TCHAR), // size of buffer  &cbRead, // number of bytes read  NULL); // not overlapped  // Exit if an error other than ERROR\_MORE\_DATA occurs.  if( !fSuccess && (GetLastError() != ERROR\_MORE\_DATA))  break;  else \_tprintf( TEXT("%s\n"), chReadBuf);  }  \_getch();  CloseHandle(hPipe);    return 0;  } |

管道客户端使用 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 将 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea)、 [**WaitNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-waitnamedpipea) (（如有必要）) 、 [**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 和 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数调用合并到单个调用中。 由于管道句柄在函数返回之前关闭，因此如果消息大于读取缓冲区的指定大小，则消息中的任何其他字节都将丢失。 以下示例是上一个改写为使用 **CallNamedPipe** 的示例。

## 实例2

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <conio.h>  #include <tchar.h>  #define BUFSIZE 512    int \_tmain(int argc, TCHAR \*argv[])  {  LPTSTR lpszWrite = TEXT("Default message from client");  TCHAR chReadBuf[BUFSIZE];  BOOL fSuccess;  DWORD cbRead;  LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");  if( argc > 1)  {  lpszWrite = argv[1];  }    fSuccess = CallNamedPipe(  lpszPipename, // pipe name  lpszWrite, // message to server  (lstrlen(lpszWrite)+1)\*sizeof(TCHAR), // message length  chReadBuf, // buffer to receive reply  BUFSIZE\*sizeof(TCHAR), // size of read buffer  &cbRead, // number of bytes read  20000); // waits for 20 seconds    if (fSuccess || GetLastError() == ERROR\_MORE\_DATA)  {  \_tprintf( TEXT("%s\n"), chReadBuf );    // The pipe is closed; no more data can be read.    if (! fSuccess)  {  printf("\nExtra data in message was lost\n");  }  }    \_getch();  return 0;  } |

## WaitNamedPipe函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 等待到超时间隔过去或指定命名管道的实例可用于连接 (即管道的服务器进程对管道) 具有挂起的 [ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 操作。 语法 C++复制  BOOL WaitNamedPipeA(  [in] LPCSTR lpNamedPipeName,  [in] DWORD nTimeOut  ); 参数 [in] lpNamedPipeName  命名管道的名称。 字符串必须包含执行服务器进程的计算机的名称。 如果管道是本地的，则 *服务器名称* 可以使用句点。 使用以下管道名称格式：  \\*servername*\pipe\*pipename*  [in] nTimeOut  函数等待命名管道实例可用的毫秒数。 可以使用以下值之一，而不是指定毫秒数。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **NMPWAIT\_USE\_DEFAULT\_WAIT**  0x00000000 | 超时间隔是服务器进程在 [CreateNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createnamedpipea) 函数中指定的默认值。 | | **NMPWAIT\_WAIT\_FOREVER**  0xffffffff | 在命名管道的实例可用之前，函数不会返回 。 |  返回值 如果在超时间隔之前管道实例可用，则返回值为非零值。  如果在超时间隔之前管道实例不可用，则返回值为零。 要获得更多的错误信息，请调用 GetLastError。 注解 如果不存在指定命名管道的实例，则无论超时值如何， **WaitNamedPipe** 函数都会立即返回。  如果超时间隔过期， **WaitNamedPipe** 函数将失败， **并ERROR\_SEM\_TIMEOUT**错误。  如果函数成功，则进程应使用 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数打开命名管道的句柄。 返回值 **TRUE** 指示至少有一个管道实例可用。 后续对管道的 **CreateFile** 调用可能会失败，因为实例已被服务器关闭或由另一个客户端打开。  **Windows 10版本 1709：**管道仅在应用容器中受支持;即，从一个 UWP 进程到另一个属于同一应用的 UWP 进程。 此外，命名管道必须使用语法 \\.\pipe\LOCAL\ 来表示管道名称。 示例 有关示例，请参阅 [命名管道客户端](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/named-pipe-client)。 要求  |  |  | | --- | --- | | **标头** | winbase.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [CallNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-callnamedpipea)  [ConnectNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe)  [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea)  [CreateNamedPipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createnamedpipea)  [管道函数](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipe-functions)  [管道概述](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipes) |

## 实例

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <conio.h>  #include <tchar.h>  #define BUFSIZE 512    int \_tmain(int argc, TCHAR \*argv[])  {  HANDLE hPipe;  LPTSTR lpvMessage=TEXT("Default message from client.");  TCHAR chBuf[BUFSIZE];  BOOL fSuccess = FALSE;  DWORD cbRead, cbToWrite, cbWritten, dwMode;  LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");  if( argc > 1 )  lpvMessage = argv[1];    // Try to open a named pipe; wait for it, if necessary.    while (1)  {  hPipe = CreateFile(  lpszPipename, // pipe name  GENERIC\_READ | // read and write access  GENERIC\_WRITE,  0, // no sharing  NULL, // default security attributes  OPEN\_EXISTING, // opens existing pipe  0, // default attributes  NULL); // no template file    // Break if the pipe handle is valid.    if (hPipe != INVALID\_HANDLE\_VALUE)  break;    // Exit if an error other than ERROR\_PIPE\_BUSY occurs.    if (GetLastError() != ERROR\_PIPE\_BUSY)  {  \_tprintf( TEXT("Could not open pipe. GLE=%d\n"), GetLastError() );  return -1;  }    // All pipe instances are busy, so wait for 20 seconds.    if ( ! WaitNamedPipe(lpszPipename, 20000))  {  printf("Could not open pipe: 20 second wait timed out.");  return -1;  }  }    // The pipe connected; change to message-read mode.    dwMode = PIPE\_READMODE\_MESSAGE;  fSuccess = SetNamedPipeHandleState(  hPipe, // pipe handle  &dwMode, // new pipe mode  NULL, // don't set maximum bytes  NULL); // don't set maximum time  if ( ! fSuccess)  {  \_tprintf( TEXT("SetNamedPipeHandleState failed. GLE=%d\n"), GetLastError() );  return -1;  }    // Send a message to the pipe server.    cbToWrite = (lstrlen(lpvMessage)+1)\*sizeof(TCHAR);  \_tprintf( TEXT("Sending %d byte message: \"%s\"\n"), cbToWrite, lpvMessage);  fSuccess = WriteFile(  hPipe, // pipe handle  lpvMessage, // message  cbToWrite, // message length  &cbWritten, // bytes written  NULL); // not overlapped  if ( ! fSuccess)  {  \_tprintf( TEXT("WriteFile to pipe failed. GLE=%d\n"), GetLastError() );  return -1;  }  printf("\nMessage sent to server, receiving reply as follows:\n");    do  {  // Read from the pipe.    fSuccess = ReadFile(  hPipe, // pipe handle  chBuf, // buffer to receive reply  BUFSIZE\*sizeof(TCHAR), // size of buffer  &cbRead, // number of bytes read  NULL); // not overlapped    if ( ! fSuccess && GetLastError() != ERROR\_MORE\_DATA )  break;    \_tprintf( TEXT("\"%s\"\n"), chBuf );  } while ( ! fSuccess); // repeat loop if ERROR\_MORE\_DATA  if ( ! fSuccess)  {  \_tprintf( TEXT("ReadFile from pipe failed. GLE=%d\n"), GetLastError() );  return -1;  }  printf("\n<End of message, press ENTER to terminate connection and exit>");  \_getch();    CloseHandle(hPipe);    return 0;  } |

# 命名管道打开模式

管道服务器在 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数的 dwOpenMode 参数中指定管道访问、重叠和写通模式。 管道客户端可以使用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数为其管道句柄指定这些打开模式。

## 访问模式

设置管道访问模式等效于指定与管道服务器的句柄关联的读取或写入访问权限。 下表显示了可使用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 指定的每个访问模式的等效泛型访问权限。

展开表

| **访问模式** | **等效的泛型访问权限** |
| --- | --- |
| PIPE\_ACCESS\_INBOUND | GENERIC\_READ |
| PIPE\_ACCESS\_OUTBOUND | GENERIC\_WRITE |
| PIPE\_ACCESS\_DUPLEX | GENERIC\_READ |GENERIC\_WRITE |

如果管道服务器使用 PIPE\_ACCESS\_INBOUND 创建管道，则管道服务器为只读管道，管道客户端为只读。 如果管道服务器使用PIPE\_ACCESS\_OUTBOUND创建管道，则管道服务器为只读管道，管道客户端为只读。 使用 PIPE\_ACCESS\_DUPLEX 创建的管道对于管道服务器和管道客户端都是可读/写的。

使用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 连接到命名管道的管道客户端必须在 dwDesiredAccess 参数中指定与管道服务器指定的访问模式兼容的访问权限。 例如，客户端必须指定GENERIC\_READ访问权限才能打开管道服务器使用 PIPE\_ACCESS\_OUTBOUND 创建的管道的句柄。 对于管道的所有实例，访问模式必须相同。

若要读取管道属性（如读取模式或阻止模式），管道句柄必须具有FILE\_READ\_ATTRIBUTES访问权限;若要写入管道属性，管道句柄必须具有FILE\_WRITE\_ATTRIBUTES访问权限。 这些访问权限可以与适用于管道的通用访问权限组合使用：GENERIC\_READ具有只读管道的FILE\_WRITE\_ATTRIBUTES，或对只读管道使用FILE\_READ\_ATTRIBUTES GENERIC\_WRITE。 以这种方式限制访问权限可为管道提供更好的安全性。

## 重叠模式

在重叠模式下，执行长时间读取、写入和连接操作的函数可以立即返回。 这使线程能够在后台执行耗时操作时执行其他操作。 若要指定重叠模式，请使用 FILE\_FLAG\_OVERLAPPED 标志。 有关详细信息，请参阅 [同步和重叠输入和输出](https://learn.microsoft.com/zh-cn/windows/win32/ipc/synchronous-and-overlapped-input-and-output)。

[**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数允许管道客户端使用 dwFlagsAndAttributes 参数为其管道句柄设置重叠模式 (FILE\_FLAG\_OVERLAPPED) 。

## Write-Through模式

使用 FILE\_FLAG\_WRITE\_THROUGH 指定写通模式。 此模式仅影响对不同计算机上的管道客户端和管道服务器之间的字节类型管道的写入操作。 在写通模式下，写入命名管道的函数在数据通过网络传输并进入远程计算机上的管道缓冲区之前不会返回。 对于每个写入操作需要同步的应用程序，写通模式非常有用。

如果未启用写通模式，系统会通过缓冲数据来提高网络操作的效率，直到累积最小字节数或最长时间段已过。 缓冲使系统能够将多个写入操作合并到单个网络传输中。 这意味着，在系统将数据放入出站缓冲区之后，但在系统通过网络传输数据之前，写入操作可以成功完成。

[**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数允许管道客户端使用 dwFlagsAndAttributes 参数为其管道句柄设置写通模式 (FILE\_FLAG\_WRITE\_THROUGH) 。 创建管道句柄后，无法更改管道句柄的写通模式。 对于同一管道实例的服务器和客户端句柄，写通模式可能不同。

管道客户端可以使用 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数来控制禁用写通模式的管道的字节数和传输前的超时期限。 对于只读管道，必须使用GENERIC\_READ打开管道句柄，并FILE\_WRITE\_ATTRIBUTES访问权限。

# 命名管道类型、读取和等待模式

管道服务器在 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数的 dwPipeMode 参数中指定管道类型模式、读取模式和等待模式。 管道客户端可以使用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数为其管道句柄指定这些管道模式。

## 类型模式

管道的类型模式确定如何将数据写入命名管道。 可以通过命名管道以字节流或消息流的形式传输数据。 在调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 创建命名管道实例时，管道服务器指定管道类型。 对于管道的所有实例，类型模式必须相同。

若要创建字节类型管道，请指定PIPE\_TYPE\_BYTE或使用默认值。 数据作为字节流写入管道，系统不会区分以不同写入操作写入的字节。

若要创建消息类型管道，请指定PIPE\_TYPE\_MESSAGE。 系统会将每个写入操作中写入管道的字节视为消息单元。 系统始终对消息类型管道执行写入操作，就像启用了直通写模式一样。

## 读取模式

管道的读取模式决定了如何从命名管道读取数据。 管道服务器在调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 时指定管道句柄的初始读取模式。 可以在字节读取模式或消息读取模式下读取数据。 字节类型管道的句柄只能处于字节读取模式。 消息类型管道的句柄可以处于字节读取或消息读取模式。 对于消息类型管道，对于同一管道实例的服务器句柄和客户端句柄，读取模式可能不同。

若要在字节读取模式下创建管道句柄，请指定PIPE\_READMODE\_BYTE或使用默认值。 数据作为字节流从管道中读取。 读取管道中的所有可用字节或读取指定数量的字节时，读取操作将成功完成。

若要在消息读取模式下创建管道句柄，请指定PIPE\_READMODE\_MESSAGE。 数据作为消息流从管道中读取。 仅当读取整个消息时，读取操作才会成功完成。 如果要读取的指定字节数小于下一条消息的大小，则函数在返回零之前读取尽可能多的消息， ([**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_MORE\_DATA) 。 可以使用另一个读取操作读取消息的其余部分。

对于管道客户端， [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 返回的管道句柄最初始终处于字节读取模式。 管道客户端和管道服务器都可以使用 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数更改管道句柄的读取模式。 管道句柄必须具有FILE\_WRITE\_ATTRIBUTES访问权限。

## 等待模式

管道句柄的等待模式确定 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数如何处理冗长操作。 在阻塞等待模式下，函数无限期地等待管道另一端的进程完成操作。 在非阻止等待模式下，函数在需要无限期等待的情况下立即返回。

当管道为空时， [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 操作受管道句柄的等待模式的影响。 使用阻塞等待句柄时，在线程写入管道另一端的数据可用之前，操作不会成功完成。 使用非阻止等待句柄，函数立即返回零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_NO\_DATA。

当管道的缓冲区中空间不足时， [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 操作受管道句柄的等待模式的影响。 使用阻塞等待句柄时，在从管道另一端读取的线程在缓冲区中创建足够的空间之前，写入操作无法成功。 使用非阻止等待句柄时，写入操作将立即返回非零值，无需为消息类型管道) 写入任何字节 (，也不会写入缓冲区保留的字节数 (字节类型管道) 。

当没有客户端连接或等待连接到管道实例时， [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 操作受管道句柄的等待模式的影响。 使用阻塞等待句柄时，在管道客户端通过调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数连接到管道实例之前，连接操作不会成功。 使用非阻止等待句柄时，连接操作将立即返回零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_PIPE\_LISTENING。

默认情况下， [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 或 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数返回的所有命名管道句柄都是在启用阻止等待模式的情况下创建的。 若要在非阻止等待模式下创建管道，管道服务器在调用 **CreateNamedPipe** 时指定PIPE\_NOWAIT。

管道客户端和管道服务器都可以通过在对 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数的调用中指定PIPE\_WAIT或PIPE\_NOWAIT来更改管道句柄的等待模式。

**备注**

支持非阻止等待模式，以便与 Microsoft LAN Manager 版本 2.0 兼容。 不应使用此模式来实现具有命名管道的 I/O) (重叠的输入和输出。 应改用重叠的 I/O，因为它允许在函数返回后在后台运行耗时的操作。 有关重叠 I/O 的详细信息，请参阅 [**同步和重叠输入和输出**](https://learn.microsoft.com/zh-cn/windows/win32/ipc/synchronous-and-overlapped-input-and-output)。

# 命名管道实例

最简单的管道服务器创建管道的单个实例，连接到单个客户端，与客户端通信，断开与客户端的连接，关闭管道句柄，然后终止。 但是，管道服务器与多个管道客户端通信更为常见。 管道服务器可以使用单个管道实例通过按顺序连接到每个客户端并断开与多个管道客户端的连接，但性能会很差。 管道服务器必须创建多个管道实例，才能有效同时处理多个客户端。

为多个管道实例提供服务有三种基本策略。

* 为每个管道实例创建单独的线程。 有关多线程管道服务器的示例，请参阅 [多线程管道服务器](https://learn.microsoft.com/zh-cn/windows/win32/ipc/multithreaded-pipe-server)。
* 通过在 [**ReadFile、WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数中指定 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构，使用重叠操作。 有关示例，请参阅 [使用重叠 I/O 的命名管道服务器](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipe-server-using-overlapped-i-o)。
* 通过使用 [**ReadFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfileex) 和 [**WriteFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefileex) 函数来使用重叠操作，这些函数指定要在操作完成时执行的完成例程。 有关示例，请参阅 [使用完成例程的命名管道服务器](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipe-server-using-completion-routines)。

多线程管道服务器最容易写入，因为每个实例的线程处理单个管道客户端的通信。 系统根据需要为每个线程分配处理器时间。 但每个线程都使用系统资源，这是处理大量客户端的管道服务器的缺点。

使用单线程服务器可以更轻松地协调影响多个客户端的操作，并且更容易保护共享资源免受多个客户端同时访问。 单线程服务器的挑战在于，它需要协调重叠的操作，以分配处理器时间来处理客户端的并发需求。

# 命名管道操作

管道服务器首次调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数时，它使用 nMaxInstances 参数指定可以同时存在的管道实例的最大数目。 服务器可以重复调用 **CreateNamedPipe** 以创建管道的其他实例，只要它不超过最大实例数。 如果函数成功，则每次调用都会返回一个指向命名管道实例的服务器端的句柄。

管道服务器创建管道实例后，管道客户端可以通过调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数连接到该实例。 如果管道实例可用， **CreateFile** 会将句柄返回到管道实例的客户端。 如果没有可用的管道实例，管道客户端可以使用 [**WaitNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-waitnamedpipea) 函数等待管道变得可用。

管道服务器可以通过调用 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数来确定管道客户端何时连接到管道实例。 如果管道句柄处于阻止等待模式，则在连接客户端之前， **ConnectNamedPipe** 不会返回。

除了 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 之外，管道客户端和服务器还可以调用多个函数之一，以便从命名管道读取和写入。 这些函数的行为取决于管道的类型以及指定的管道句柄的生效模式，如下所示：

* [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 和 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 函数可与字节类型管道或消息类型管道一起使用。
* 如果为重叠操作打开了管道句柄，[**ReadFileEx 和 WriteFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefileex) 函数可以与字节类型管道或消息类型管道一起使用。
* [**PeekNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-peeknamedpipe) 函数可用于读取，而无需删除字节类型管道或消息类型管道的内容。 **PeekNamedPipe** 还可以返回有关管道实例的其他信息。
* 如果将调用进程的管道句柄设置为消息读取模式，则 [**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 函数可与消息类型的双工管道一起使用。 函数在单个操作中写入请求消息并读取回复消息，从而提高网络性能。

在管道客户端启动之前，管道服务器不应执行阻止读取操作。 否则，可能会出现争用情况。 当初始化代码（如 C 运行时库的初始化代码）需要锁定和检查继承的句柄时，通常会发生这种情况。

当客户端和服务器使用完管道实例时，服务器应首先调用 [**FlushFileBuffers**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-flushfilebuffers) 函数，以确保客户端读取写入管道的所有字节或消息。 在客户端从管道读取所有数据之前，**FlushFileBuffers** 不会返回。 然后，服务器调用 [**DisconnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-disconnectnamedpipe) 函数以关闭与管道客户端的连接。 此函数会使客户端的句柄无效（如果尚未关闭）。 管道中的任何未读数据将被丢弃。 客户端断开连接后，服务器调用 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数以关闭其管道实例的句柄。 或者，服务器可以使用 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 使新客户端能够连接到管道的此实例。

进程可以通过调用 [**GetNamedPipeInfo**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-getnamedpipeinfo) 函数来检索有关命名管道的信息，该函数返回管道的类型、输入和输出缓冲区的大小以及可以创建的管道实例的最大数目。 [**GetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-getnamedpipehandlestatea) 函数报告管道句柄的读取和等待模式、管道实例的当前数目，以及通过网络通信的管道的其他信息。 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数设置管道句柄的读取模式和等待模式。 对于与远程服务器通信的管道客户端，函数还控制要收集的最大字节数或传输消息之前的最大等待时间， (假设客户端的句柄未在启用) 的写通模式的情况下打开。

# 同步和重叠管道 I/O

[**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、[**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile)、[**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数可以在管道上同步或异步执行输入和输出操作。 当函数同步运行时，它不会返回，直到它正在执行的操作完成。 这意味着，调用线程的执行可能会无限期地被阻止，同时等待耗时的操作完成。 当函数异步运行时，即使操作尚未完成，它也会立即返回。 这样，当调用线程可以自由执行其他任务时，就可以在后台执行耗时的操作。

使用异步 I/O 使管道服务器能够使用执行以下步骤的循环：

1. 在对 wait 函数的调用中指定多个事件对象，并等待其中一个对象设置为信号状态。
2. 使用 wait 函数的返回值确定已完成的重叠操作。
3. 执行清理已完成的操作所需的任务，并为该管道句柄启动下一个操作。 这可能需要为同一管道句柄启动另一个重叠操作。

重叠操作使一个管道可以同时读取和写入数据，使单个线程可以在多个管道句柄上同时执行 I/O 操作。 这使单线程管道服务器能够有效地处理与多个管道客户端的通信。 有关示例，请参阅 [使用重叠 I/O 的命名管道服务器](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipe-server-using-overlapped-i-o)。

要使管道服务器使用同步操作与多个客户端通信，它必须为每个管道客户端创建单独的线程，以便一个或多个线程可以在其他线程等待时运行。 有关使用同步操作的多线程管道服务器的示例，请参阅 [多线程管道服务器](https://learn.microsoft.com/zh-cn/windows/win32/ipc/multithreaded-pipe-server)。

## 启用异步操作

仅当为指定的管道句柄启用重叠模式并指定指向 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构的有效指针时，才能异步执行 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、[**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile)、[**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数。 如果 **OVERLAPPED** 指针为 **NULL**，则函数返回值可能会错误地指示操作已完成。 因此，强烈建议使用 FILE\_FLAG\_OVERLAPPED 创建句柄，并且需要异步行为，应始终指定有效的 **OVERLAPPED** 结构。

指定的 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构的 **hEvent** 成员必须包含手动重置事件对象的句柄。 这是 [**CreateEvent**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-createeventa) 函数创建的同步对象。 启动重叠操作的线程使用事件对象来确定操作何时完成。 在同一句柄上执行同步操作时，不应使用管道句柄进行同步，因为无法知道哪个操作的完成导致了管道句柄的信号。 在同一管道句柄上同时执行操作的唯一可靠方法是对每个操作使用单独的 **OVERLAPPED** 结构及其自己的事件对象。 有关事件对象的详细信息，请参阅 [同步](https://learn.microsoft.com/zh-cn/windows/desktop/Sync/synchronization)。

此外，可以使用 [**GetQueuedCompletionStatus 或 GetQueuedCompletionStatusEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/ioapiset/nf-ioapiset-getqueuedcompletionstatus) 函数在重叠操作完成时收到通知。 在这种情况下，无需在 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构中分配手动重置事件，并且完成操作针对管道句柄的方式与异步读取或写入操作相同。 有关详细信息，请参阅 [I/O 完成端口](https://learn.microsoft.com/zh-cn/windows/desktop/FileIO/i-o-completion-ports)。

异步执行 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile)、 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile)、 [**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 操作时，会发生以下情况之一：

* 如果在函数返回时操作已完成，则返回值指示操作的成功或失败。 如果发生错误，则返回值为零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_IO\_PENDING以外的内容。
* 如果函数返回时操作尚未完成，则返回值为零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 返回ERROR\_IO\_PENDING。 在这种情况下，调用线程必须等待操作完成。 然后，调用线程必须调用 [**GetOverlappedResult**](https://learn.microsoft.com/zh-cn/windows/desktop/api/ioapiset/nf-ioapiset-getoverlappedresult) 函数来确定结果。

## 使用完成例程

[**ReadFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfileex) 和 [**WriteFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefileex) 函数提供另一种形式的重叠 I/O。 与重叠的 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 和 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 函数（它们使用事件对象发出完成信号）不同，扩展函数指定 完成例程。 完成例程是在读取或写入操作完成后排队等待执行的函数。 在调用 **ReadFileEx** 和 **WriteFileEx** 的线程通过调用 fAlertable 参数设置为 **TRUE** 的可[警报等待函数](https://learn.microsoft.com/zh-cn/windows/desktop/Sync/wait-functions)之一来启动可警报等待操作之前，不会执行完成例程。 在可发出警报的等待操作中，当 **ReadFileEx** 或 **WriteFileEx** 完成例程排队等待执行时，函数也会返回 。 管道服务器可以使用扩展函数对连接到它的每个客户端执行一系列读取和写入操作。 序列中的每个读取或写入操作指定一个完成例程，每个完成例程启动序列中的下一步。 有关示例，请参阅 [命名管道服务器使用完成例程](https://learn.microsoft.com/zh-cn/windows/win32/ipc/named-pipe-server-using-completion-routines)。

## 实例

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <tchar.h>  #include <strsafe.h>  #define PIPE\_TIMEOUT 5000  #define BUFSIZE 4096    typedef struct  {  OVERLAPPED oOverlap;  HANDLE hPipeInst;  TCHAR chRequest[BUFSIZE];  DWORD cbRead;  TCHAR chReply[BUFSIZE];  DWORD cbToWrite;  } PIPEINST, \*LPPIPEINST;    VOID DisconnectAndClose(LPPIPEINST);  BOOL CreateAndConnectInstance(LPOVERLAPPED);  BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);  VOID GetAnswerToRequest(LPPIPEINST);  VOID WINAPI CompletedWriteRoutine(DWORD, DWORD, LPOVERLAPPED);  VOID WINAPI CompletedReadRoutine(DWORD, DWORD, LPOVERLAPPED);    HANDLE hPipe;    int \_tmain(VOID)  {  HANDLE hConnectEvent;  OVERLAPPED oConnect;  LPPIPEINST lpPipeInst;  DWORD dwWait, cbRet;  BOOL fSuccess, fPendingIO;    // Create one event object for the connect operation.    hConnectEvent = CreateEvent(  NULL, // default security attribute  TRUE, // manual reset event  TRUE, // initial state = signaled  NULL); // unnamed event object  if (hConnectEvent == NULL)  {  printf("CreateEvent failed with %d.\n", GetLastError());  return 0;  }    oConnect.hEvent = hConnectEvent;    // Call a subroutine to create one instance, and wait for  // the client to connect.    fPendingIO = CreateAndConnectInstance(&oConnect);    while (1)  {  // Wait for a client to connect, or for a read or write  // operation to be completed, which causes a completion  // routine to be queued for execution.    dwWait = WaitForSingleObjectEx(  hConnectEvent, // event object to wait for  INFINITE, // waits indefinitely  TRUE); // alertable wait enabled    switch (dwWait)  {  // The wait conditions are satisfied by a completed connect  // operation.  case 0:  // If an operation is pending, get the result of the  // connect operation.    if (fPendingIO)  {  fSuccess = GetOverlappedResult(  hPipe, // pipe handle  &oConnect, // OVERLAPPED structure  &cbRet, // bytes transferred  FALSE); // does not wait  if (!fSuccess)  {  printf("ConnectNamedPipe (%d)\n", GetLastError());  return 0;  }  }    // Allocate storage for this instance.    lpPipeInst = (LPPIPEINST) GlobalAlloc(  GPTR, sizeof(PIPEINST));  if (lpPipeInst == NULL)  {  printf("GlobalAlloc failed (%d)\n", GetLastError());  return 0;  }    lpPipeInst->hPipeInst = hPipe;    // Start the read operation for this client.  // Note that this same routine is later used as a  // completion routine after a write operation.    lpPipeInst->cbToWrite = 0;  CompletedWriteRoutine(0, 0, (LPOVERLAPPED) lpPipeInst);    // Create new pipe instance for the next client.    fPendingIO = CreateAndConnectInstance(  &oConnect);  break;    // The wait is satisfied by a completed read or write  // operation. This allows the system to execute the  // completion routine.    case WAIT\_IO\_COMPLETION:  break;    // An error occurred in the wait function.    default:  {  printf("WaitForSingleObjectEx (%d)\n", GetLastError());  return 0;  }  }  }  return 0;  }    // CompletedWriteRoutine(DWORD, DWORD, LPOVERLAPPED)  // This routine is called as a completion routine after writing to  // the pipe, or when a new client has connected to a pipe instance.  // It starts another read operation.    VOID WINAPI CompletedWriteRoutine(DWORD dwErr, DWORD cbWritten,  LPOVERLAPPED lpOverLap)  {  LPPIPEINST lpPipeInst;  BOOL fRead = FALSE;    // lpOverlap points to storage for this instance.    lpPipeInst = (LPPIPEINST) lpOverLap;    // The write operation has finished, so read the next request (if  // there is no error).    if ((dwErr == 0) && (cbWritten == lpPipeInst->cbToWrite))  fRead = ReadFileEx(  lpPipeInst->hPipeInst,  lpPipeInst->chRequest,  BUFSIZE\*sizeof(TCHAR),  (LPOVERLAPPED) lpPipeInst,  (LPOVERLAPPED\_COMPLETION\_ROUTINE) CompletedReadRoutine);    // Disconnect if an error occurred.    if (! fRead)  DisconnectAndClose(lpPipeInst);  }    // CompletedReadRoutine(DWORD, DWORD, LPOVERLAPPED)  // This routine is called as an I/O completion routine after reading  // a request from the client. It gets data and writes it to the pipe.    VOID WINAPI CompletedReadRoutine(DWORD dwErr, DWORD cbBytesRead,  LPOVERLAPPED lpOverLap)  {  LPPIPEINST lpPipeInst;  BOOL fWrite = FALSE;    // lpOverlap points to storage for this instance.    lpPipeInst = (LPPIPEINST) lpOverLap;    // The read operation has finished, so write a response (if no  // error occurred).    if ((dwErr == 0) && (cbBytesRead != 0))  {  GetAnswerToRequest(lpPipeInst);    fWrite = WriteFileEx(  lpPipeInst->hPipeInst,  lpPipeInst->chReply,  lpPipeInst->cbToWrite,  (LPOVERLAPPED) lpPipeInst,  (LPOVERLAPPED\_COMPLETION\_ROUTINE) CompletedWriteRoutine);  }    // Disconnect if an error occurred.    if (! fWrite)  DisconnectAndClose(lpPipeInst);  }    // DisconnectAndClose(LPPIPEINST)  // This routine is called when an error occurs or the client closes  // its handle to the pipe.    VOID DisconnectAndClose(LPPIPEINST lpPipeInst)  {  // Disconnect the pipe instance.    if (! DisconnectNamedPipe(lpPipeInst->hPipeInst) )  {  printf("DisconnectNamedPipe failed with %d.\n", GetLastError());  }    // Close the handle to the pipe instance.    CloseHandle(lpPipeInst->hPipeInst);    // Release the storage for the pipe instance.    if (lpPipeInst != NULL)  GlobalFree(lpPipeInst);  }    // CreateAndConnectInstance(LPOVERLAPPED)  // This function creates a pipe instance and connects to the client.  // It returns TRUE if the connect operation is pending, and FALSE if  // the connection has been completed.    BOOL CreateAndConnectInstance(LPOVERLAPPED lpoOverlap)  {  LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");    hPipe = CreateNamedPipe(  lpszPipename, // pipe name  PIPE\_ACCESS\_DUPLEX | // read/write access  FILE\_FLAG\_OVERLAPPED, // overlapped mode  PIPE\_TYPE\_MESSAGE | // message-type pipe  PIPE\_READMODE\_MESSAGE | // message read mode  PIPE\_WAIT, // blocking mode  PIPE\_UNLIMITED\_INSTANCES, // unlimited instances  BUFSIZE\*sizeof(TCHAR), // output buffer size  BUFSIZE\*sizeof(TCHAR), // input buffer size  PIPE\_TIMEOUT, // client time-out  NULL); // default security attributes  if (hPipe == INVALID\_HANDLE\_VALUE)  {  printf("CreateNamedPipe failed with %d.\n", GetLastError());  return 0;  }    // Call a subroutine to connect to the new client.    return ConnectToNewClient(hPipe, lpoOverlap);  }  BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)  {  BOOL fConnected, fPendingIO = FALSE;    // Start an overlapped connection for this pipe instance.  fConnected = ConnectNamedPipe(hPipe, lpo);    // Overlapped ConnectNamedPipe should return zero.  if (fConnected)  {  printf("ConnectNamedPipe failed with %d.\n", GetLastError());  return 0;  }    switch (GetLastError())  {  // The overlapped connection in progress.  case ERROR\_IO\_PENDING:  fPendingIO = TRUE;  break;    // Client is already connected, so signal an event.    case ERROR\_PIPE\_CONNECTED:  if (SetEvent(lpo->hEvent))  break;    // If an error occurs during the connect operation...  default:  {  printf("ConnectNamedPipe failed with %d.\n", GetLastError());  return 0;  }  }  return fPendingIO;  }  VOID GetAnswerToRequest(LPPIPEINST pipe)  {  \_tprintf( TEXT("[%d] %s\n"), pipe->hPipeInst, pipe->chRequest);  StringCchCopy( pipe->chReply, BUFSIZE, TEXT("Default answer from server") );  pipe->cbToWrite = (lstrlen(pipe->chReply)+1)\*sizeof(TCHAR);  } |

# 命名管道安全性和访问权限

Windows 安全性使你能够控制对命名管道的访问。 有关安全性的详细信息，请参阅 [访问控制模型](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-control-model)。

调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数时，可以为命名管道指定[安全描述符](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-descriptors)。 安全描述符控制对命名管道客户端和服务器端的访问。 如果指定 **NULL**，则命名管道将获取默认的安全描述符。 命名管道的默认安全描述符中的 ACL 授予 LocalSystem 帐户、管理员和创建者所有者的完全控制权。 他们还向 Everyone 组的成员和匿名帐户授予读取访问权限。

若要检索命名管道的安全描述符，请调用 [**GetSecurityInfo**](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-getsecurityinfo) 函数。 若要更改命名管道的安全描述符，请调用 [**SetSecurityInfo**](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-setsecurityinfo) 函数。

当线程调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 以打开现有命名管道服务器端的句柄时，系统会在返回句柄之前执行访问检查。 访问检查将线程的访问令牌和请求[的访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-rights-and-access-masks)与命名管道的安全描述符中的 DACL 进行比较。 除了请求的访问权限外，DACL 还必须允许调用线程FILE\_CREATE\_PIPE\_INSTANCE访问命名管道。

同样，当客户端调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 或 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数以连接到命名管道的客户端端时，系统会在授予对客户端的访问权限之前执行访问检查。

[**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 函数返回的句柄始终具有 SYNCHRONIZE 访问权限。 它还具有GENERIC\_READ和/或GENERIC\_WRITE，具体取决于管道的打开模式。 下面是每个打开模式的访问权限。

展开表

| **打开模式** | **自动删除** |
| --- | --- |
| PIPE\_ACCESS\_DUPLEX (0x00000003) | FILE\_GENERIC\_READ、FILE\_GENERIC\_WRITE 和 SYNCHRONIZE |
| PIPE\_ACCESS\_INBOUND (0x00000001) | FILE\_GENERIC\_READ和 SYNCHRONIZE |
| PIPE\_ACCESS\_OUTBOUND (0x00000002) | FILE\_GENERIC\_WRITE和 SYNCHRONIZE |

FILE\_GENERIC\_READ命名管道的访问权限结合了从管道读取数据、读取管道属性、读取扩展属性和读取管道 DACL 的权限。

FILE\_GENERIC\_WRITE命名管道的访问权限结合了将数据写入管道、向管道追加数据、写入管道属性、写入扩展属性和读取管道 DACL 的权限。 由于 FILE\_APPEND\_DATA 和 FILE\_CREATE\_PIPE\_INSTANCE 具有相同的定义，因此 FILE\_GENERIC\_WRITE 允许创建管道的权限。 若要避免此问题，请使用个人权限，而不是使用FILE\_GENERIC\_WRITE。

如果要读取或写入对象的 SACL，可以请求ACCESS\_SYSTEM\_SECURITY对命名管道对象的访问权限。 有关详细信息，请参阅 [访问控制列表 (ACL)](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-control-lists)和 [SACL 访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/sacl-access-right)。

若要防止远程用户或其他终端服务会话上的用户访问命名管道，请使用管道 DACL 上的登录 SID。 登录 SID 也用于运行方式登录;它是用于保护每个会话对象命名空间的 SID。 有关详细信息，请参阅 [获取 C++ 中的登录 SID](https://learn.microsoft.com/zh-cn/previous-versions/aa446670(v=vs.85))。

## 实例

|  |
| --- |
| #include <windows.h>  #pragma comment(lib, "advapi32.lib")  BOOL GetLogonSID (HANDLE hToken, PSID \*ppsid)  {  BOOL bSuccess = FALSE;  DWORD dwIndex;  DWORD dwLength = 0;  PTOKEN\_GROUPS ptg = NULL;  // Verify the parameter passed in is not NULL.  if (NULL == ppsid)  goto Cleanup;  // Get required buffer size and allocate the TOKEN\_GROUPS buffer.  if (!GetTokenInformation(  hToken, // handle to the access token  TokenGroups, // get information about the token's groups  (LPVOID) ptg, // pointer to TOKEN\_GROUPS buffer  0, // size of buffer  &dwLength // receives required buffer size  ))  {  if (GetLastError() != ERROR\_INSUFFICIENT\_BUFFER)  goto Cleanup;  ptg = (PTOKEN\_GROUPS)HeapAlloc(GetProcessHeap(),  HEAP\_ZERO\_MEMORY, dwLength);  if (ptg == NULL)  goto Cleanup;  }  // Get the token group information from the access token.  if (!GetTokenInformation(  hToken, // handle to the access token  TokenGroups, // get information about the token's groups  (LPVOID) ptg, // pointer to TOKEN\_GROUPS buffer  dwLength, // size of buffer  &dwLength // receives required buffer size  ))  {  goto Cleanup;  }  // Loop through the groups to find the logon SID.  for (dwIndex = 0; dwIndex < ptg->GroupCount; dwIndex++)  if ((ptg->Groups[dwIndex].Attributes & SE\_GROUP\_LOGON\_ID)  == SE\_GROUP\_LOGON\_ID)  {  // Found the logon SID; make a copy of it.  dwLength = GetLengthSid(ptg->Groups[dwIndex].Sid);  \*ppsid = (PSID) HeapAlloc(GetProcessHeap(),  HEAP\_ZERO\_MEMORY, dwLength);  if (\*ppsid == NULL)  goto Cleanup;  if (!CopySid(dwLength, \*ppsid, ptg->Groups[dwIndex].Sid))  {  HeapFree(GetProcessHeap(), 0, (LPVOID)\*ppsid);  goto Cleanup;  }  break;  }  bSuccess = TRUE;  Cleanup:  // Free the buffer for the token groups.  if (ptg != NULL)  HeapFree(GetProcessHeap(), 0, (LPVOID)ptg);  return bSuccess;  }  The following function frees the buffer allocated by the GetLogonSID example function.  c++复制  #include <windows.h>  #pragma comment(lib, "advapi32.lib")  VOID FreeLogonSID (PSID \*ppsid)  {  HeapFree(GetProcessHeap(), 0, (LPVOID)\*ppsid);  } |

# 模拟命名管道客户端

模拟 是线程在与拥有线程的进程不同的安全上下文中执行的能力。 模拟使服务器线程能够在客户端的安全上下文限制内代表客户端执行操作。 客户端通常具有较低级别的访问权限。 有关详细信息，请参阅 [模拟](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/client-impersonation)。

命名管道服务器线程可以调用 [**ImpersonateNamedPipeClient**](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient) 函数，以假定连接到管道客户端的用户的访问令牌。 例如，命名管道服务器可以提供对管道服务器具有特权访问权限的数据库或文件系统的访问权限。 当管道客户端向服务器发送请求时，服务器将模拟客户端并尝试访问受保护的数据库。 然后，系统会根据客户端的安全级别授予或拒绝服务器的访问权限。 服务器完成后，它使用 [**RevertToSelf**](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-reverttoself) 函数还原其原始安全令牌。

[模拟级别](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/impersonation-levels)确定服务器在模拟客户端时可以执行的操作。 默认情况下，服务器在 SecurityImpersonation 模拟级别模拟。 但是，当客户端调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数以打开管道客户端端的句柄时，客户端可以使用 SECURITY\_SQOS\_PRESENT 标志来控制服务器的模拟级别。

## [**ImpersonateNamedPipeClient**](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient) 函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ImpersonateNamedPipeClient** 函数模拟命名管道客户端应用程序。 语法 C++复制  BOOL ImpersonateNamedPipeClient(  [in] HANDLE hNamedPipe  ); 参数 [in] hNamedPipe  命名管道的句柄。 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 **ImpersonateNamedPipeClient** 函数允许命名管道的服务器端模拟客户端。 调用此函数时，命名管道文件系统将更改调用 [进程的](https://learn.microsoft.com/zh-cn/windows/win32/SecGloss/p-gly) 线程，以开始模拟从管道中读取的最后一条消息 [的安全上下文](https://learn.microsoft.com/zh-cn/windows/win32/SecGloss/s-gly) 。 只有管道的服务器端可以调用此函数。  模拟完成后，服务器可以调用 [RevertToSelf](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-reverttoself) 函数。  **重要** 如果 **ImpersonateNamedPipeClient** 函数失败，则不会模拟客户端，并且所有后续客户端请求都在调用函数的进程的安全上下文中进行。 如果调用进程作为特权帐户运行，它可以执行不允许客户端执行的操作。 为了避免安全风险，调用进程应始终检查返回值。 如果返回值指示函数调用失败，则不应执行任何客户端请求。    如果存在以下任一情况，则所有模拟函数（包括 **ImpersonateNamedPipeClient** ）都允许请求的模拟：   * 请求的令牌模拟级别小于 **SecurityImpersonation**，例如 **SecurityIdentification** 或 **SecurityAnonymous**。 * 调用方具有 **SeImpersonatePrivilege** 特权。 * 调用方登录会话中的进程 (或另一个进程) 通过 [LogonUser](https://learn.microsoft.com/zh-cn/windows/win32/api/winbase/nf-winbase-logonusera) 或 [LsaLogonUser](https://learn.microsoft.com/zh-cn/windows/win32/api/ntsecapi/nf-ntsecapi-lsalogonuser) 函数使用显式凭据创建令牌。 * 经过身份验证的标识与调用方相同。   **具有 SP1 及更早版本的 Windows XP： 不支持 SeImpersonatePrivilege** 特权。 示例 有关使用此函数的示例，请参阅 [使用 ACL 验证客户端访问](https://learn.microsoft.com/zh-cn/windows/win32/SecAuthZ/verifying-client-access-with-acls-in-c--)。 要求  |  |  | | --- | --- | | **标头** | namedpipeapi.h | | **Library** | Advapi32.lib | | **DLL** | Advapi32.dll |  另请参阅 [授权函数](https://learn.microsoft.com/zh-cn/windows/win32/secauthz/authorization-functions)  [客户端/服务器访问控制概述](https://learn.microsoft.com/zh-cn/windows/win32/SecAuthZ/client-server-access-control)  [DdeImpersonateClient](https://learn.microsoft.com/zh-cn/windows/win32/api/ddeml/nf-ddeml-ddeimpersonateclient)  [DuplicateToken](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-duplicatetoken)  [RevertToSelf](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-reverttoself) |

### 实例: [使用 ACL 验证客户端访问](https://learn.microsoft.com/zh-cn/windows/win32/SecAuthZ/verifying-client-access-with-acls-in-c--)

以下示例演示服务器如何检查[*安全描述符*](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/s-gly)允许客户端的访问权限。 该示例使用 [**ImpersonateNamedPipeClient**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient) 函数;但是，它将使用任何其他模拟函数以相同的方式工作。 模拟客户端后，该示例调用 [**OpenThreadToken**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthreadtoken) 函数以获取 [*模拟令牌*](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/i-gly)。 然后，它调用 [**MapGenericMask**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-mapgenericmask) 函数，根据 [**GENERIC\_MAPPING**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winnt/ns-winnt-generic_mapping) 结构中指定的映射，将任何泛型访问权限转换为相应的特定和标准权限。

[**AccessCheck**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-accesscheck) 函数根据安全描述符 DACL 中客户端允许的权限检查请求的访问权限。 若要检查访问权限并在安全事件日志中生成条目，请使用 [**AccessCheckAndAuditAlarm**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-accesscheckandauditalarma) 函数。

|  |
| --- |
| #include <windows.h>  #pragma comment(lib, "advapi32.lib")  BOOL ImpersonateAndCheckAccess(  HANDLE hNamedPipe, // handle of pipe to impersonate  PSECURITY\_DESCRIPTOR pSD, // security descriptor to check  DWORD dwAccessDesired, // access rights to check  PGENERIC\_MAPPING pGeneric, // generic mapping for object  PDWORD pdwAccessAllowed // returns allowed access rights  )  {  HANDLE hToken;  PRIVILEGE\_SET PrivilegeSet;  DWORD dwPrivSetSize = sizeof( PRIVILEGE\_SET );  BOOL fAccessGranted=FALSE;  // Impersonate the client.  if (! ImpersonateNamedPipeClient(hNamedPipe) )  return FALSE;  // Get an impersonation token with the client's security context.  if (! OpenThreadToken( GetCurrentThread(), TOKEN\_ALL\_ACCESS,  TRUE, &hToken ))  {  goto Cleanup;  }  // Use the GENERIC\_MAPPING structure to convert any  // generic access rights to object-specific access rights.  MapGenericMask( &dwAccessDesired, pGeneric );  // Check the client's access rights.  if( !AccessCheck(  pSD, // security descriptor to check  hToken, // impersonation token  dwAccessDesired, // requested access rights  pGeneric, // pointer to GENERIC\_MAPPING  &PrivilegeSet, // receives privileges used in check  &dwPrivSetSize, // size of PrivilegeSet buffer  pdwAccessAllowed, // receives mask of allowed access rights  &fAccessGranted )) // receives results of access check  {  goto Cleanup;  }  Cleanup:  RevertToSelf();  if (hToken != INVALID\_HANDLE\_VALUE)  CloseHandle(hToken);  return fAccessGranted;  } |

## [**RevertToSelf**](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-reverttoself) 函数用法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **RevertToSelf** 函数终止客户端应用程序的模拟。 语法 C++复制  BOOL RevertToSelf(); 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 使用 [DdeImpersonateClient、ImpersonateDdeClientWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/ddeml/nf-ddeml-ddeimpersonateclient)、ImpersonateLoggedOnUser、[ImpersonateNamedPipeClient](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient)、[ImpersonateSelf](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-impersonateself)、[ImpersonateAnonymousToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-impersonateanonymoustoken) 或 [SetThreadToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadtoken) 函数完成开始的任何模拟后，进程应调用 **RevertToSelf** 函数。  使用 [RpcImpersonateClient](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcimpersonateclient) 函数模拟客户端的 RPC 服务器必须调用 [RpcRevertToSelf](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcreverttoself) 或 [RpcRevertToSelfEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcreverttoselfex) 以结束模拟。  如果 **RevertToSelf** 失败，应用程序将继续在客户端的上下文中运行，这是不合适的。 如果 **RevertToSelf** 失败，应关闭进程。 示例 有关使用此函数的示例，请参阅 [使用 ACL 验证客户端访问](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/verifying-client-access-with-acls-in-c--)。 要求  |  |  | | --- | --- | | **标头** | securitybaseapi.h (包括 Windows.h) | | **Library** | Advapi32.lib | | **DLL** | Advapi32.dll |  另请参阅 [客户端/服务器访问控制函数](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/authorization-functions)  [客户端/服务器访问控制概述](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/client-server-access-control)  [DdeImpersonateClient](https://learn.microsoft.com/zh-cn/windows/desktop/api/ddeml/nf-ddeml-ddeimpersonateclient)  [ImpersonateAnonymousToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-impersonateanonymoustoken)  [ImpersonateDdeClientWindow](https://learn.microsoft.com/zh-cn/windows/desktop/api/dde/nf-dde-impersonateddeclientwindow)  [ImpersonateLoggedOnUser](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-impersonateloggedonuser)  [ImpersonateNamedPipeClient](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-impersonatenamedpipeclient)  [ImpersonateSelf](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-impersonateself)  [RpcImpersonateClient](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcimpersonateclient)  [RpcRevertToSelf](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcreverttoself)  [RpcRevertToSelfEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/rpcdce/nf-rpcdce-rpcreverttoselfex)  [SetThreadToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-setthreadtoken) |

# 授权) (模拟级别

[**SECURITY\_IMPERSONATION\_LEVEL**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winnt/ne-winnt-security_impersonation_level)枚举定义了四个模拟级别，用于确定服务器可以在客户端上下文中执行的操作。

展开表

| **模拟级别** | **说明** |
| --- | --- |
| SecurityAnonymous | 服务器无法模拟或标识客户端。 |
| SecurityIdentification | 服务器可以获取客户端的标识和特权，但不能模拟客户端。 |
| SecurityImpersonation | 服务器可以在本地系统上模拟客户端的安全上下文。 |
| SecurityDelegation | 服务器可以在远程系统上模拟客户端的安全上下文。 |

命名管道、RPC 或 DDE 连接的客户端可以控制模拟级别。 例如，命名管道客户端可以调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数以打开命名管道的句柄并指定服务器的模拟级别。

当命名管道、RPC 或 DDE 连接是远程连接时，将忽略传递给 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 以设置模拟级别的标志。 在这种情况下，客户端的模拟级别由服务器启用的模拟级别确定，该级别由目录服务中的服务器帐户上的标志设置。 例如，如果为服务器启用了委派，则客户端的模拟级别也将设置为委派，即使传递给 **CreateFile** 的标志指定了标识模拟级别。

DDE 客户端使用具有 [**SECURITY\_QUALITY\_OF\_SERVICE**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winnt/ns-winnt-security_quality_of_service) 结构的 [**DdeSetQualityOfService**](https://learn.microsoft.com/zh-cn/windows/win32/api/dde/nf-dde-ddesetqualityofservice) 函数来指定模拟级别。 SecurityImpersonation 级别是命名管道、RPC 和 DDE 服务器的默认值。 [**ImpersonateSelf**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-impersonateself)、[**DuplicateToken**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-duplicatetoken) 和 [**DuplicateTokenEx**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-duplicatetokenex) 函数允许调用方指定模拟级别。 使用 [**GetTokenInformation**](https://learn.microsoft.com/zh-cn/windows/win32/api/securitybaseapi/nf-securitybaseapi-gettokeninformation) 函数检索 [访问令牌](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/a-gly)的模拟级别。

在 SecurityImpersonation 级别，线程的大多数操作发生在线程[模拟令牌](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/i-gly)的安全上下文中，而不是发生在拥有线程[的进程的主令牌](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/p-gly)中。 例如，如果模拟线程打开[安全对象](https://learn.microsoft.com/zh-cn/windows/win32/secauthz/securable-objects)，则系统会使用模拟令牌来检查线程的访问。 同样，如果模拟线程创建新对象（例如通过调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数），则新对象的所有者是客户端 [访问令牌](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/a-gly)的默认所有者。

但是，在以下情况下，系统使用进程的主令牌，而不是调用线程的模拟令牌：

* 如果模拟线程调用 [**CreateProcess**](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 函数，则新进程始终继承进程的主标记。
* 对于需要 SE\_TCB\_NAME 特权的函数（例如 [**LogonUser**](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-logonusera) 函数），系统始终检查进程主令牌中的特权。
* 对于需要SE\_AUDIT\_NAME特权的函数（例如 [**ObjectOpenAuditAlarm**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-objectopenauditalarma) 函数），系统始终检查进程主令牌中的特权。
* 在调用 [**OpenThreadToken**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthreadtoken) 函数时，线程可以指定函数是使用模拟令牌还是主令牌来确定是否授予请求的访问权限。