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

# 多线程管道服务器

以下示例是多线程管道服务器。 它有一个main线程，其中包含一个循环，用于创建管道实例并等待管道客户端连接。 当管道客户端连接时，管道服务器会创建一个线程来为该客户端提供服务，然后继续在main线程中执行循环。 管道客户端可以在调用 [**CreateNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-createnamedpipea) 和 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 函数之间的间隔内成功连接到管道实例。 如果发生这种情况， **ConnectNamedPipe** 返回零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 返回ERROR\_PIPE\_CONNECTED。

为服务每个管道实例而创建的线程从管道读取请求，并将答复写入管道，直到管道客户端关闭其句柄。 发生这种情况时，线程将刷新管道、断开连接、关闭其管道句柄并终止。 main线程将运行，直到发生错误或进程结束。

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);

}

# 使用重叠 I/O 的命名管道服务器

下面是单线程管道服务器的示例，该服务器使用重叠操作为多个管道客户端的同时连接提供服务。 管道服务器创建固定数量的管道实例。 每个管道实例都可以连接到单独的管道客户端。 当管道客户端使用完其管道实例后，服务器会断开与客户端的连接，并重复使用该管道实例连接到新客户端。 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。

在管道实例上的每个 [**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) 操作中，[**将 OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构指定为参数。 尽管该示例演示了对不同管道实例的同步操作，但它通过使用 **OVERLAPPED** 结构中的 事件对象来避免对单个管道实例执行同时操作。 由于同一事件对象用于每个实例的读取、写入和连接操作，因此无法知道哪个操作的完成导致事件设置为使用同一管道实例进行同时操作的信号状态。

每个管道实例的事件句柄存储在传递给 [**WaitForMultipleObjects**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects) 函数的数组中。 此函数等待某个事件发出信号，并返回导致等待操作完成的事件的数组索引。 本主题中的示例使用此数组索引检索包含管道实例信息的结构。 服务器使用 结构的 **fPendingIO** 成员来跟踪实例上的最新 I/O 操作是否处于挂起状态，这需要调用 [**GetOverlappedResult**](https://learn.microsoft.com/zh-cn/windows/desktop/api/ioapiset/nf-ioapiset-getoverlappedresult) 函数。 服务器使用 结构的 **dwState** 成员来确定必须为管道实例执行的下一个操作。

重叠 [**的 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) 操作可以在函数返回时完成。 否则，如果操作处于挂起状态，则指定 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构中的事件对象在函数返回之前将设置为非对齐状态。 挂起的操作完成后，系统会将事件对象的状态设置为已发出信号。 如果操作在函数返回之前完成，则事件对象的状态不会更改。

由于该示例使用手动重置事件对象，因此 [**WaitForMultipleObjects**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects) 函数不会将事件对象的状态更改为非签名。 这一点很重要，因为该示例依赖于保持信号状态的事件对象，除非存在挂起的操作。

如果在 [**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/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 返回时操作已完成，则函数的返回值指示结果。 对于读取和写入操作，还会返回传输的字节数。 如果操作仍处于挂起状态， **ReadFile**、 **WriteFile** 或 **ConnectNamedPipe** 函数将返回零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_IO\_PENDING。 在这种情况下，使用 [**GetOverlappedResult**](https://learn.microsoft.com/zh-cn/windows/desktop/api/ioapiset/nf-ioapiset-getoverlappedresult) 函数在操作完成后检索结果。 **GetOverlappedResult** 仅返回挂起操作的结果。 它不会报告在返回重叠的 **ReadFile**、 **WriteFile** 或 **ConnectNamedPipe** 函数之前完成的操作的结果。

在与客户端断开连接之前，必须等待指示客户端已完成的信号。 (刷新文件缓冲区会破坏重叠 I/O 的目的，因为刷新操作会在等待客户端清空管道时阻止服务器线程的执行。) 在此示例中，信号是在管道客户端关闭其句柄后尝试从管道读取时生成的错误。

C++

#include <windows.h>

#include <stdio.h>

#include <tchar.h>

#include <strsafe.h>

#define CONNECTING\_STATE 0

#define READING\_STATE 1

#define WRITING\_STATE 2

#define INSTANCES 4

#define PIPE\_TIMEOUT 5000

#define BUFSIZE 4096

typedef struct

{

OVERLAPPED oOverlap;

HANDLE hPipeInst;

TCHAR chRequest[BUFSIZE];

DWORD cbRead;

TCHAR chReply[BUFSIZE];

DWORD cbToWrite;

DWORD dwState;

BOOL fPendingIO;

} PIPEINST, \*LPPIPEINST;

VOID DisconnectAndReconnect(DWORD);

BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);

VOID GetAnswerToRequest(LPPIPEINST);

PIPEINST Pipe[INSTANCES];

HANDLE hEvents[INSTANCES];

int \_tmain(VOID)

{

DWORD i, dwWait, cbRet, dwErr;

BOOL fSuccess;

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

// The initial loop creates several instances of a named pipe

// along with an event object for each instance. An

// overlapped ConnectNamedPipe operation is started for

// each instance.

for (i = 0; i < INSTANCES; i++)

{

// Create an event object for this instance.

hEvents[i] = CreateEvent(

NULL, // default security attribute

TRUE, // manual-reset event

TRUE, // initial state = signaled

NULL); // unnamed event object

if (hEvents[i] == NULL)

{

printf("CreateEvent failed with %d.\n", GetLastError());

return 0;

}

Pipe[i].oOverlap.hEvent = hEvents[i];

Pipe[i].oOverlap.Offset = 0;

Pipe[i].oOverlap.OffsetHigh = 0;

Pipe[i].hPipeInst = 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

INSTANCES, // number of instances

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

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

PIPE\_TIMEOUT, // client time-out

NULL); // default security attributes

if (Pipe[i].hPipeInst == INVALID\_HANDLE\_VALUE)

{

printf("CreateNamedPipe failed with %d.\n", GetLastError());

return 0;

}

// Call the subroutine to connect to the new client

Pipe[i].fPendingIO = ConnectToNewClient(

Pipe[i].hPipeInst,

&Pipe[i].oOverlap);

Pipe[i].dwState = Pipe[i].fPendingIO ?

CONNECTING\_STATE : // still connecting

READING\_STATE; // ready to read

}

while (1)

{

// Wait for the event object to be signaled, indicating

// completion of an overlapped read, write, or

// connect operation.

dwWait = WaitForMultipleObjects(

INSTANCES, // number of event objects

hEvents, // array of event objects

FALSE, // does not wait for all

INFINITE); // waits indefinitely

// dwWait shows which pipe completed the operation.

i = dwWait - WAIT\_OBJECT\_0; // determines which pipe

if (i < 0 || i > (INSTANCES - 1))

{

printf("Index out of range.\n");

return 0;

}

// Get the result if the operation was pending.

if (Pipe[i].fPendingIO)

{

fSuccess = GetOverlappedResult(

Pipe[i].hPipeInst, // handle to pipe

&Pipe[i].oOverlap, // OVERLAPPED structure

&cbRet, // bytes transferred

FALSE); // do not wait

switch (Pipe[i].dwState)

{

// Pending connect operation

case CONNECTING\_STATE:

if (! fSuccess)

{

printf("Error %d.\n", GetLastError());

return 0;

}

Pipe[i].dwState = READING\_STATE;

break;

// Pending read operation

case READING\_STATE:

if (! fSuccess || cbRet == 0)

{

DisconnectAndReconnect(i);

continue;

}

Pipe[i].cbRead = cbRet;

Pipe[i].dwState = WRITING\_STATE;

break;

// Pending write operation

case WRITING\_STATE:

if (! fSuccess || cbRet != Pipe[i].cbToWrite)

{

DisconnectAndReconnect(i);

continue;

}

Pipe[i].dwState = READING\_STATE;

break;

default:

{

printf("Invalid pipe state.\n");

return 0;

}

}

}

// The pipe state determines which operation to do next.

switch (Pipe[i].dwState)

{

// READING\_STATE:

// The pipe instance is connected to the client

// and is ready to read a request from the client.

case READING\_STATE:

fSuccess = ReadFile(

Pipe[i].hPipeInst,

Pipe[i].chRequest,

BUFSIZE\*sizeof(TCHAR),

&Pipe[i].cbRead,

&Pipe[i].oOverlap);

// The read operation completed successfully.

if (fSuccess && Pipe[i].cbRead != 0)

{

Pipe[i].fPendingIO = FALSE;

Pipe[i].dwState = WRITING\_STATE;

continue;

}

// The read operation is still pending.

dwErr = GetLastError();

if (! fSuccess && (dwErr == ERROR\_IO\_PENDING))

{

Pipe[i].fPendingIO = TRUE;

continue;

}

// An error occurred; disconnect from the client.

DisconnectAndReconnect(i);

break;

// WRITING\_STATE:

// The request was successfully read from the client.

// Get the reply data and write it to the client.

case WRITING\_STATE:

GetAnswerToRequest(&Pipe[i]);

fSuccess = WriteFile(

Pipe[i].hPipeInst,

Pipe[i].chReply,

Pipe[i].cbToWrite,

&cbRet,

&Pipe[i].oOverlap);

// The write operation completed successfully.

if (fSuccess && cbRet == Pipe[i].cbToWrite)

{

Pipe[i].fPendingIO = FALSE;

Pipe[i].dwState = READING\_STATE;

continue;

}

// The write operation is still pending.

dwErr = GetLastError();

if (! fSuccess && (dwErr == ERROR\_IO\_PENDING))

{

Pipe[i].fPendingIO = TRUE;

continue;

}

// An error occurred; disconnect from the client.

DisconnectAndReconnect(i);

break;

default:

{

printf("Invalid pipe state.\n");

return 0;

}

}

}

return 0;

}

// DisconnectAndReconnect(DWORD)

// This function is called when an error occurs or when the client

// closes its handle to the pipe. Disconnect from this client, then

// call ConnectNamedPipe to wait for another client to connect.

VOID DisconnectAndReconnect(DWORD i)

{

// Disconnect the pipe instance.

if (! DisconnectNamedPipe(Pipe[i].hPipeInst) )

{

printf("DisconnectNamedPipe failed with %d.\n", GetLastError());

}

// Call a subroutine to connect to the new client.

Pipe[i].fPendingIO = ConnectToNewClient(

Pipe[i].hPipeInst,

&Pipe[i].oOverlap);

Pipe[i].dwState = Pipe[i].fPendingIO ?

CONNECTING\_STATE : // still connecting

READING\_STATE; // ready to read

}

// ConnectToNewClient(HANDLE, LPOVERLAPPED)

// This function is called to start an overlapped connect operation.

// It returns TRUE if an operation is pending or FALSE if the

// connection has been completed.

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);

}

# 使用完成例程的命名管道服务器

以下示例是创建消息类型管道并使用重叠操作的单线程管道服务器。 它使用扩展函数 [**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) 通过完成例程执行重叠 I/O，该例程在操作完成后排队等待执行。 管道服务器使用 [**WaitForSingleObjectEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobjectex) 函数，该函数执行可警报的等待操作，该操作在完成例程准备好执行时返回。 当向事件对象发出信号时，wait 函数也会返回 ，在此示例中指示重叠的 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 操作已完成 (新客户端已连接) 。 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。

最初，管道服务器创建管道的单个实例，并启动重叠的 [**ConnectNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-connectnamedpipe) 操作。 当客户端连接时，服务器分配一个结构来为该管道实例提供存储，然后调用 [**ReadFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfileex) 函数以启动一系列 I/O 操作来处理与客户端的通信。 每个操作指定一个完成例程，该例程在序列中执行下一个操作。 当客户端断开连接且管道实例关闭时，序列终止。 为新客户端启动操作序列后，服务器将创建另一个管道实例，并等待下一个客户端连接。

[**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) 函数的参数指定完成例程和指向 [**OVERLAPPED**](https://learn.microsoft.com/zh-cn/windows/desktop/api/minwinbase/ns-minwinbase-overlapped) 结构的指针。 此指针将传递到其 lpOverLap 参数中的完成例程。 由于 **OVERLAPPED** 结构指向为每个管道实例分配的结构中的第一个成员，因此完成例程可以使用其 lpOverLap 参数访问管道实例的结构。

C++复制

#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);

}

# 命名管道客户端

命名管道客户端使用 [**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。 如果管道存在，但其所有实例都忙， **则 CreateFile** 返回 **INVALID\_HANDLE\_VALUE** ， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 函数返回ERROR\_PIPE\_BUSY。 发生这种情况时，命名管道客户端使用 [**WaitNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-waitnamedpipea) 函数等待命名管道的实例变为可用。

如果指定的访问与服务器创建管道时指定的 (双工、出站或入站) 的访问不兼容， [**则 CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数将失败。 对于双工管道，客户端可以指定读取、写入或读/写访问权限;对于出站管道 (只写服务器) ，客户端必须指定只读访问权限;对于入站管道 (只读服务器) ，客户端必须指定只写访问。

[**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 返回的句柄默认为字节读取模式、阻止等待模式、禁用重叠模式和禁用写通模式。 管道客户端可以使用 **CreateFile** 通过指定FILE\_FLAG\_OVERLAPPED来启用重叠模式，或者通过指定FILE\_FLAG\_WRITE\_THROUGH来启用写通模式。 客户端可以使用 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数通过指定PIPE\_NOWAIT来启用非阻止模式，或者通过指定PIPE\_READMODE\_MESSAGE来启用消息读取模式。

以下示例演示了一个管道客户端，该客户端打开命名管道，将管道句柄设置为消息读取模式，使用 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 函数向服务器发送请求，并使用 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 函数读取服务器的回复。 此管道客户端可以与本主题底部列出的任何消息类型服务器一起使用。 但是，对于字节类型服务器，此管道客户端在调用 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 以更改为消息读取模式时失败。 由于客户端在消息读取模式下从管道读取，因此 **ReadFile** 操作可以在读取部分消息后返回零。 当消息大于读取缓冲区时，就会发生这种情况。 在这种情况下， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 返回ERROR\_MORE\_DATA，客户端可以使用对 **ReadFile** 的其他调用来读取消息的其余部分。

此管道客户端可以与“另请参阅”下列出的任何管道服务器一起使用。

C++复制

#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;

}

# 命名管道上的事务

命名管道事务是客户端/服务器通信，它将写入操作和读取操作合并到单个网络操作中。 事务只能在双工消息类型管道上使用。 事务可提高客户端与远程服务器之间的网络通信性能。 进程可以使用 [**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 和 [**CallNamedPipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-callnamedpipea) 函数来执行命名管道事务。

管道客户端最常使用 [**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 函数将请求消息写入命名管道服务器并读取服务器的响应消息。 管道客户端必须指定GENERIC\_READ |GENERIC\_WRITE通过调用 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数打开其管道句柄时进行访问。 然后，管道客户端通过调用 [**SetNamedPipeHandleState**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-setnamedpipehandlestate) 函数将管道句柄设置为消息读取模式。 如果在对 **TransactNamedPipe** 的调用中指定的读取缓冲区不够大，无法容纳服务器写入的整个消息，则函数返回零， [**GetLastError**](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 返回ERROR\_MORE\_DATA。 客户端可以通过调用 [**ReadFile、ReadFileEx**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 或 [**PeekNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-peeknamedpipe) 函数来读取消息的其余部分。

[**TransactNamedPipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe) 通常由管道客户端调用，但也可以由管道服务器使用。

以下示例演示使用 [**TransactNamedPipe 的**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-transactnamedpipe)管道客户端。 此管道客户端可以与“另请参阅”下列出的任何管道服务器一起使用。

C++复制

#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。

C++复制

#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;

}