**匿名管道操作**

[**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数创建匿名管道并返回两个句柄：管道的读取句柄和管道的写入句柄。 读取句柄对管道具有只读访问权限，写入句柄对管道具有只读访问权限。 若要使用管道进行通信，管道服务器必须将管道句柄传递给另一个进程。 通常，这是通过继承完成的;也就是说，进程允许子进程继承句柄。 进程还可以使用 [**DuplicateHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-duplicatehandle) 函数复制管道句柄，并使用某种形式的进程间通信（例如 DDE 或共享内存）将其发送到不相关的进程。

管道服务器可以将读取句柄或写入句柄发送到管道客户端，具体取决于客户端是应使用匿名管道发送信息还是接收信息。 若要从管道读取数据，请在调用 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 函数时使用管道的读取句柄。 当另一个进程写入管道时， **ReadFile** 调用将返回。 如果管道的所有写入句柄都已关闭，或者在读取操作完成之前发生错误， **ReadFile** 调用也可以返回 。

若要写入管道，请在调用 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 函数时使用管道的写入句柄。 **WriteFile** 调用在将指定字节数写入管道或发生错误之前不会返回。 如果管道缓冲区已满，并且有更多字节要写入， **则 WriteFile** 不会返回，直到另一个进程从管道读取，从而提供更多的缓冲区空间。 管道服务器在调用 [**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe) 时指定管道的缓冲区大小。

匿名管道不支持异步 (重叠) 读取和写入操作。 这意味着不能将 [**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) 函数用于匿名管道。 此外，当这些函数与匿名管道一起使用时，[**将忽略 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) 的 *lpOverlapped* 参数。

匿名管道存在，直到所有管道句柄（读取和写入）都已关闭。 进程可以使用 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数关闭其管道句柄。 当进程终止时，所有管道句柄也会关闭。

匿名管道是使用具有唯一名称的命名管道实现的。 因此，通常可以将匿名管道的句柄传递给需要命名管道句柄的函数。

 创建匿名管道，并将句柄返回到管道的读取和写入端。

## CreatePipe函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 语法 C++复制  BOOL CreatePipe(  [out] PHANDLE hReadPipe,  [out] PHANDLE hWritePipe,  [in, optional] LPSECURITY\_ATTRIBUTES lpPipeAttributes,  [in] DWORD nSize  ); 参数 [out] hReadPipe  指向接收管道的读取句柄的变量的指针。  [out] hWritePipe  指向接收管道的写入句柄的变量的指针。  [in, optional] lpPipeAttributes  指向 [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构的指针，该结构确定返回的句柄是否可以由子进程继承。 如果 *lpPipeAttributes* 为 **NULL**，则不能继承句柄。  结构的 **lpSecurityDescriptor** 成员指定新管道的安全描述符。 如果 *lpPipeAttributes* 为 **NULL**，则管道获取默认安全描述符。 管道的默认安全描述符中的 ACL 来自创建者的主要令牌或模拟令牌。  [in] nSize  管道的缓冲区大小（以字节为单位）。 大小只是一个建议：系统使用 值来计算适当的缓冲机制。 如果此参数为零，则系统使用默认缓冲区大小。 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 **CreatePipe** 创建管道，并将指定的管道大小分配给存储缓冲区。 **CreatePipe** 还会创建句柄，进程在后续调用 [ReadFile 和 WriteFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile) 函数时使用该句柄从缓冲区读取和写入缓冲区。  为了从管道读取数据，进程在调用 [ReadFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile) 函数时使用读取句柄。 当以下情况之一为 true 时，**ReadFile** 将返回：在管道的写入端完成写入操作、已读取请求的字节数或发生错误。  当进程使用 [WriteFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-writefile) 写入匿名管道时，写入操作在写入所有字节之前不会完成。 如果在写入所有字节之前管道缓冲区已满，则在另一个进程或线程使用 [ReadFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile) 使更多缓冲区可用之前**，WriteFile** 不会返回。  匿名管道是使用具有唯一名称的命名管道实现的。 因此，通常可以将匿名管道的句柄传递给需要命名管道句柄的函数。  如果 **CreatePipe** 失败，则输出参数的内容不确定。 在此事件中，不应对其内容做出假设。  若要释放管道使用的资源，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/win32/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，管道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，始终会删除管道实例。 示例 有关示例，请参阅 [使用重定向的输入和输出创建子进程](https://learn.microsoft.com/zh-cn/windows/win32/ProcThread/creating-a-child-process-with-redirected-input-and-output)。 要求 展开表   | **要求** | **值** | | --- | --- | | **最低受支持的客户端** | Windows 2000 专业版 [桌面应用 |UWP 应用] | | **最低受支持的服务器** | Windows 2000 Server [桌面应用 |UWP 应用] | | **目标平台** | Windows | | **标头** | namedpipeapi.h | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [管道函数](https://learn.microsoft.com/zh-cn/windows/win32/ipc/pipe-functions)  [管道概述](https://learn.microsoft.com/zh-cn/windows/win32/ipc/pipes)  [ReadFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile)  [SECURITY\_ATTRIBUTES](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))  [WriteFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-writefile) |

## 实例

### 父进程的代码。 它采用单个命令行参数：文本文件的名称。

|  |
| --- |
| #include <windows.h>  #include <tchar.h>  #include <stdio.h>  #include <strsafe.h>  #define BUFSIZE 4096    HANDLE g\_hChildStd\_IN\_Rd = NULL;  HANDLE g\_hChildStd\_IN\_Wr = NULL;  HANDLE g\_hChildStd\_OUT\_Rd = NULL;  HANDLE g\_hChildStd\_OUT\_Wr = NULL;  HANDLE g\_hInputFile = NULL;    void CreateChildProcess(void);  void WriteToPipe(void);  void ReadFromPipe(void);  void ErrorExit(PCTSTR);    int \_tmain(int argc, TCHAR \*argv[])  {  SECURITY\_ATTRIBUTES saAttr;    printf("\n->Start of parent execution.\n");  // Set the bInheritHandle flag so pipe handles are inherited.    saAttr.nLength = sizeof(SECURITY\_ATTRIBUTES);  saAttr.bInheritHandle = TRUE;  saAttr.lpSecurityDescriptor = NULL;  // Create a pipe for the child process's STDOUT.    if ( ! CreatePipe(&g\_hChildStd\_OUT\_Rd, &g\_hChildStd\_OUT\_Wr, &saAttr, 0) )  ErrorExit(TEXT("StdoutRd CreatePipe"));  // Ensure the read handle to the pipe for STDOUT is not inherited.  if ( ! SetHandleInformation(g\_hChildStd\_OUT\_Rd, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdout SetHandleInformation"));  // Create a pipe for the child process's STDIN.    if (! CreatePipe(&g\_hChildStd\_IN\_Rd, &g\_hChildStd\_IN\_Wr, &saAttr, 0))  ErrorExit(TEXT("Stdin CreatePipe"));  // Ensure the write handle to the pipe for STDIN is not inherited.    if ( ! SetHandleInformation(g\_hChildStd\_IN\_Wr, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdin SetHandleInformation"));    // Create the child process.    CreateChildProcess();  // Get a handle to an input file for the parent.  // This example assumes a plain text file and uses string output to verify data flow.    if (argc == 1)  ErrorExit(TEXT("Please specify an input file.\n"));  g\_hInputFile = CreateFile(  argv[1],  GENERIC\_READ,  0,  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_READONLY,  NULL);  if ( g\_hInputFile == INVALID\_HANDLE\_VALUE )  ErrorExit(TEXT("CreateFile"));    // Write to the pipe that is the standard input for a child process.  // Data is written to the pipe's buffers, so it is not necessary to wait  // until the child process is running before writing data.    WriteToPipe();  printf( "\n->Contents of %S written to child STDIN pipe.\n", argv[1]);    // Read from pipe that is the standard output for child process.    printf( "\n->Contents of child process STDOUT:\n\n");  ReadFromPipe();  printf("\n->End of parent execution.\n");  // The remaining open handles are cleaned up when this process terminates.  // To avoid resource leaks in a larger application, close handles explicitly.  return 0;  }    void CreateChildProcess()  // Create a child process that uses the previously created pipes for STDIN and STDOUT.  {  TCHAR szCmdline[]=TEXT("child");  PROCESS\_INFORMATION piProcInfo;  STARTUPINFO siStartInfo;  BOOL bSuccess = FALSE;    // Set up members of the PROCESS\_INFORMATION structure.    ZeroMemory( &piProcInfo, sizeof(PROCESS\_INFORMATION) );    // Set up members of the STARTUPINFO structure.  // This structure specifies the STDIN and STDOUT handles for redirection.    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );  siStartInfo.cb = sizeof(STARTUPINFO);  siStartInfo.hStdError = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdOutput = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdInput = g\_hChildStd\_IN\_Rd;  siStartInfo.dwFlags |= STARTF\_USESTDHANDLES;    // Create the child process.    bSuccess = CreateProcess(NULL,  szCmdline, // command line  NULL, // process security attributes  NULL, // primary thread security attributes  TRUE, // handles are inherited  0, // creation flags  NULL, // use parent's environment  NULL, // use parent's current directory  &siStartInfo, // STARTUPINFO pointer  &piProcInfo); // receives PROCESS\_INFORMATION    // If an error occurs, exit the application.  if ( ! bSuccess )  ErrorExit(TEXT("CreateProcess"));  else  {  // Close handles to the child process and its primary thread.  // Some applications might keep these handles to monitor the status  // of the child process, for example.  CloseHandle(piProcInfo.hProcess);  CloseHandle(piProcInfo.hThread);    // Close handles to the stdin and stdout pipes no longer needed by the child process.  // If they are not explicitly closed, there is no way to recognize that the child process has ended.    CloseHandle(g\_hChildStd\_OUT\_Wr);  CloseHandle(g\_hChildStd\_IN\_Rd);  }  }    void WriteToPipe(void)  // Read from a file and write its contents to the pipe for the child's STDIN.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;    for (;;)  {  bSuccess = ReadFile(g\_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);  if ( ! bSuccess || dwRead == 0 ) break;    bSuccess = WriteFile(g\_hChildStd\_IN\_Wr, chBuf, dwRead, &dwWritten, NULL);  if ( ! bSuccess ) break;  }    // Close the pipe handle so the child process stops reading.    if ( ! CloseHandle(g\_hChildStd\_IN\_Wr) )  ErrorExit(TEXT("StdInWr CloseHandle"));  }    void ReadFromPipe(void)  // Read output from the child process's pipe for STDOUT  // and write to the parent process's pipe for STDOUT.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;  HANDLE hParentStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);  for (;;)  {  bSuccess = ReadFile( g\_hChildStd\_OUT\_Rd, chBuf, BUFSIZE, &dwRead, NULL);  if( ! bSuccess || dwRead == 0 ) break;  bSuccess = WriteFile(hParentStdOut, chBuf,  dwRead, &dwWritten, NULL);  if (! bSuccess ) break;  }  }    void ErrorExit(PCTSTR lpszFunction)  // Format a readable error message, display a message box,  // and exit from the application.  {  LPVOID lpMsgBuf;  LPVOID lpDisplayBuf;  DWORD dw = GetLastError();  FormatMessage(  FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |  FORMAT\_MESSAGE\_FROM\_SYSTEM |  FORMAT\_MESSAGE\_IGNORE\_INSERTS,  NULL,  dw,  MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),  (LPTSTR) &lpMsgBuf,  0, NULL );  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM\_ZEROINIT,  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)\*sizeof(TCHAR));  StringCchPrintf((LPTSTR)lpDisplayBuf,  LocalSize(lpDisplayBuf) / sizeof(TCHAR),  TEXT("%s failed with error %d: %s"),  lpszFunction, dw, lpMsgBuf);  MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB\_OK);  LocalFree(lpMsgBuf);  LocalFree(lpDisplayBuf);  ExitProcess(1);  } |

### 子进程的代码。 它使用 STDIN 和 STDOUT 继承的句柄来访问父级创建的管道。 父进程从其输入文件读取信息并将信息写入管道。 子级使用 STDIN 通过管道接收文本，并使用 STDOUT 写入管道。 父级从管道的读取端读取，并将信息显示到其 STDOUT。

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #define BUFSIZE 4096    int main(void)  {  CHAR chBuf[BUFSIZE];  DWORD dwRead, dwWritten;  HANDLE hStdin, hStdout;  BOOL bSuccess;    hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);  hStdin = GetStdHandle(STD\_INPUT\_HANDLE);  if (  (hStdout == INVALID\_HANDLE\_VALUE) ||  (hStdin == INVALID\_HANDLE\_VALUE)  )  ExitProcess(1);    // Send something to this process's stdout using printf.  printf("\n \*\* This is a message from the child process. \*\* \n");  // This simple algorithm uses the existence of the pipes to control execution.  // It relies on the pipe buffers to ensure that no data is lost.  // Larger applications would use more advanced process control.  for (;;)  {  // Read from standard input and stop on error or no data.  bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);    if (! bSuccess || dwRead == 0)  break;    // Write to standard output and stop on error.  bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);    if (! bSuccess)  break;  }  return 0;  } |

## DuplicateHandle 函数用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 复制对象句柄。 语法 C++复制  BOOL DuplicateHandle(  [in] HANDLE hSourceProcessHandle,  [in] HANDLE hSourceHandle,  [in] HANDLE hTargetProcessHandle,  [out] LPHANDLE lpTargetHandle,  [in] DWORD dwDesiredAccess,  [in] BOOL bInheritHandle,  [in] DWORD dwOptions  ); 参数 [in] hSourceProcessHandle  包含要复制的句柄的进程句柄。  句柄必须具有PROCESS\_DUP\_HANDLE访问权限。 有关详细信息，请参阅 [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-security-and-access-rights)。  [in] hSourceHandle  要复制的句柄。 这是在源进程的上下文中有效的开放对象句柄。 有关可以复制其句柄的对象列表，请参阅以下“备注”部分。  [in] hTargetProcessHandle  要接收重复句柄的进程句柄。 句柄必须具有PROCESS\_DUP\_HANDLE访问权限。  此参数是可选的，如果“选项”中设置了**DUPLICATE\_CLOSE\_SOURCE**标志，则可以将此参数指定为 NULL。  [out] lpTargetHandle  指向接收重复句柄的变量的指针。 此句柄值在目标进程的上下文中有效。  如果 *hSourceHandle* 是 [GetCurrentProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess) 或 [GetCurrentThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentthread) 返回的伪句柄， **则 DuplicateHandle** 将分别将其转换为进程或线程的实际句柄。  如果 *lpTargetHandle* 为 **NULL**，则该函数将复制句柄，但不向调用方返回重复句柄值。 此行为仅用于向后兼容此函数的早期版本。 不应使用此功能，因为目标进程终止之前会丢失系统资源。  如果 hTargetProcessHandle 为 **NULL**，则忽略此参数。  [in] dwDesiredAccess  为新句柄请求的访问权限。 有关可为每个对象类型指定的标志，请参阅以下“备注”部分。  如果 *dwOptions* 参数指定DUPLICATE\_SAME\_ACCESS标志，则忽略此参数。 否则，可以指定的标志取决于要复制其句柄的对象的类型。  如果 hTargetProcessHandle 为 **NULL**，则忽略此参数。  [in] bInheritHandle  一个指示句柄是否可继承的变量。 如果 **为 TRUE**，则重复句柄可由目标进程创建的新进程继承。 如果 **为 FALSE**，则无法继承新句柄。  如果 hTargetProcessHandle 为 **NULL**，则忽略此参数。  [in] dwOptions  可选操作。 此参数可以是零，也可以是以下值的任意组合。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **DUPLICATE\_CLOSE\_SOURCE**  0x00000001 | 关闭源句柄。 无论返回什么错误状态，都会发生这种情况。 | | **DUPLICATE\_SAME\_ACCESS**  0x00000002 | 忽略 *dwDesiredAccess* 参数。 重复句柄具有与源句柄相同的访问权限。 |  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 重复句柄引用与原始句柄相同的对象。 因此，对 对象的任何更改都通过两个句柄反映。 例如，如果复制文件句柄，则两个句柄的当前文件位置始终相同。 对于具有不同文件位置的文件句柄，请使用 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数创建共享同一文件访问权限的文件句柄。  源进程或目标进程 (或源进程和目标进程) ，都可以调用 **DuplicateHandle**。 例如，进程可以使用 **DuplicateHandle** 创建可继承句柄的不可继承副本，或具有与原始句柄不同的访问权限的句柄。  源进程使用 [GetCurrentProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess) 函数获取自身的句柄。 此句柄是伪句柄，但 **DuplicateHandle** 将其转换为实际进程句柄。 若要获取目标进程句柄，可能需要使用某种形式的进程间通信 (例如，命名管道或共享内存) 将进程标识符传达给源进程。 源进程可以在 [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess) 函数中使用此标识符来获取目标进程的句柄。  如果调用 **DuplicateHandle** 的进程不是目标进程，则源进程必须使用进程间通信将重复句柄的值传递给目标进程。  **DuplicateHandle** 可用于复制 32 位进程和 64 位进程之间的句柄。 生成的句柄大小适当，以在目标进程中工作。 有关详细信息，请参阅 [进程互操作性](https://learn.microsoft.com/zh-cn/windows/desktop/WinProg64/process-interoperability)。  **DuplicateHandle** 可以将句柄复制到以下类型的对象。  展开表   |  |  | | --- | --- | | **Object** | **描述** | | 访问令牌 | 句柄由 [CreateRestrictedToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-createrestrictedtoken)、 [DuplicateToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-duplicatetoken)、 [DuplicateTokenEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/securitybaseapi/nf-securitybaseapi-duplicatetokenex)、 [OpenProcessToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocesstoken) 或 [OpenThreadToken](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openthreadtoken) 函数返回。 | | 更改通知 | 句柄由 [FindFirstChangeNotification](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-findfirstchangenotificationa) 函数返回。 | | 通信设备 | 句柄由 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数返回。 | | 控制台输入 | 指定 CONIN$ 时， [由 CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数返回句柄;指定STD\_INPUT\_HANDLE时， [由 GetStdHandle](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) 函数返回句柄。 可以复制控制台句柄，以便仅在同一进程中使用。 | | 控制台屏幕缓冲区 | 指定 CONOUT$ 时， [由 CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数返回句柄;指定STD\_OUTPUT\_HANDLE时， [由 GetStdHandle](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) 函数返回句柄。 可以复制控制台句柄，以便仅在同一进程中使用。 | | 桌面 | 句柄由 [GetThreadDesktop](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-getthreaddesktop) 函数返回。 | | 事件 | 句柄由 [CreateEvent](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-createeventa) 或 [OpenEvent](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-openeventa) 函数返回。 | | 文件 | 句柄由 [CreateFile](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-createfilea) 函数返回。 | | 文件映射 | 句柄由 [CreateFileMapping](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createfilemappinga) 函数返回。 | | 作业 | 句柄由 [CreateJobObject](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createjobobjecta) 函数返回。 | | Mailslot | 句柄由 [CreateMailslot](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createmailslota) 函数返回。 | | Mutex | 句柄由 [CreateMutex](https://learn.microsoft.com/zh-cn/windows/desktop/api/synchapi/nf-synchapi-createmutexa) 或 [OpenMutex] ( 返回。/synchapi/nf-synchapi-openmutexw.md) 函数。 | | 管道 | 命名管道句柄由 [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。 [CreatePipe](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数返回匿名管道句柄。 | | 进程 | 句柄由 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)、 [GetCurrentProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess) 或 [OpenProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-openprocess) 函数返回。 | | 注册表项 | 句柄由 [RegCreateKey、RegCreateKeyEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/winreg/nf-winreg-regcreatekeya)、[RegOpenKey](https://learn.microsoft.com/zh-cn/windows/desktop/api/winreg/nf-winreg-regopenkeya) 或 [RegOpenKeyEx](https://learn.microsoft.com/zh-cn/windows/desktop/api/winreg/nf-winreg-regopenkeyexa) 函数返回。 请注意， [RegConnectRegistry](https://learn.microsoft.com/zh-cn/windows/desktop/api/winreg/nf-winreg-regconnectregistrya) 函数返回的注册表项句柄不能用于对 **DuplicateHandle 的**调用。 | | Semaphore | 句柄由 [CreateSemaphore](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-createsemaphorea) 或 [OpenSemaphore](https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-opensemaphorew) 函数返回。 | | 线程 | 句柄由 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa)、 [CreateThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createthread)、 [CreateRemoteThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread) 或 [GetCurrentThread](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentthread) 函数返回 | | Timer | 句柄由 [CreateWaitableTimerW](https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-createwaitabletimerw) 或 [OpenWaitableTimerW](https://learn.microsoft.com/zh-cn/windows/win32/api/synchapi/nf-synchapi-openwaitabletimerw) 函数返回。 | | 事务 | 句柄由 [CreateTransaction](https://learn.microsoft.com/zh-cn/windows/desktop/api/ktmw32/nf-ktmw32-createtransaction) 函数返回。 | | 窗口工作站 | 句柄由 [GetProcessWindowStation](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/nf-winuser-getprocesswindowstation) 函数返回。 |     不应使用 **DuplicateHandle** 将句柄复制到以下对象：   * I/O 完成端口。 不会返回错误，但不能使用重复句柄。 * 套接字。 不会返回错误，但 Winsock 可能无法在目标进程中识别重复句柄。 此外，使用 **DuplicateHandle** 会干扰基础对象上的内部引用计数。 若要复制套接字句柄，请使用 [WSADuplicateSocket](https://learn.microsoft.com/zh-cn/windows/desktop/api/winsock2/nf-winsock2-wsaduplicatesocketa) 函数。 * 除 [GetCurrentProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentprocess) 或 [GetCurrentThread](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-getcurrentthread) 函数返回的伪句柄之外。   *dwDesiredAccess* 参数指定新句柄的访问权限。 所有对象都支持 [标准访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/standard-access-rights)。 对象可能还支持其他访问权限，具体取决于对象类型。 有关详细信息，请参阅下列主题：   * [桌面安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/winstation/desktop-security-and-access-rights) * [文件安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/FileIO/file-security-and-access-rights) * [文件映射安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/Memory/file-mapping-security-and-access-rights) * [作业对象安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/job-object-security-and-access-rights) * [进程安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/process-security-and-access-rights) * [注册表项安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/SysInfo/registry-key-security-and-access-rights) * [同步对象安全性和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/Sync/synchronization-object-security-and-access-rights) * [线程安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/thread-security-and-access-rights) * [窗口工作站安全和访问权限](https://learn.microsoft.com/zh-cn/windows/desktop/winstation/window-station-security-and-access-rights)   在某些情况下，新句柄可能具有比原始句柄更多的访问权限。 但是，在其他情况下， **DuplicateHandle** 无法创建具有比原始权限更多的句柄。 例如，使用GENERIC\_READ访问权限创建的文件句柄不能重复，因此它同时具有GENERIC\_READ和GENERIC\_WRITE访问权限。  通常，当该进程使用句柄完成时，目标进程会关闭重复的句柄。 若要从源进程关闭重复的句柄，请使用以下参数调用 **DuplicateHandle** ：   * 将 *hSourceProcessHandle* 设置为创建句柄的 **DuplicateHandle** 调用的目标进程。 * 将 *hSourceHandle* 设置为要关闭的重复句柄。 * 将 *hTargetProcessHandle* 设置为 **NULL**。 * 将 *dwOptions* 设置为 DUPLICATE\_CLOSE\_SOURCE。  要求  |  |  | | --- | --- | | **标头** | handleapi.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)  [句柄继承](https://learn.microsoft.com/zh-cn/windows/desktop/SysInfo/handle-inheritance)  [句柄和对象函数](https://learn.microsoft.com/zh-cn/windows/desktop/SysInfo/handle-and-object-functions) |

## 实例

以下示例创建互斥体，将句柄复制到互斥体，并将其传递给另一个线程。 复制句柄可确保增加引用计数，以便在两个线程关闭句柄之前不会销毁互斥对象。

|  |
| --- |
| #include <windows.h>  DWORD CALLBACK ThreadProc(PVOID pvParam);  int main()  {  HANDLE hMutex = CreateMutex(NULL, FALSE, NULL);  HANDLE hMutexDup, hThread;  DWORD dwThreadId;  DuplicateHandle(GetCurrentProcess(),  hMutex,  GetCurrentProcess(),  &hMutexDup,  0,  FALSE,  DUPLICATE\_SAME\_ACCESS);  hThread = CreateThread(NULL, 0, ThreadProc,  (LPVOID) hMutexDup, 0, &dwThreadId);  // Perform work here, closing the handle when finished with the  // mutex. If the reference count is zero, the object is destroyed.  CloseHandle(hMutex);  // Wait for the worker thread to terminate and clean up.  WaitForSingleObject(hThread, INFINITE);  CloseHandle(hThread);  return 0;  }  DWORD CALLBACK ThreadProc(PVOID pvParam)  {  HANDLE hMutex = (HANDLE)pvParam;  // Perform work here, closing the handle when finished with the  // mutex. If the reference count is zero, the object is destroyed.  CloseHandle(hMutex);  return 0;  } |

**管道句柄继承**

* 项目
* 2023/06/13
* 6 个参与者

反馈

管道服务器控制是否可以通过以下方式继承其句柄：

* [**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数接收[**SECURITY\_ATTRIBUTES**](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85))结构。 如果管道服务器将此结构的 **bInheritHandle** 成员设置为 **TRUE**，则可以继承 **CreatePipe** 创建的句柄。
* 管道服务器可以使用 [**DuplicateHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-duplicatehandle) 函数更改管道句柄的继承。 管道服务器可以创建可继承管道句柄的不可继承副本或不可继承管道句柄的可继承副本。
* [**CreateProcess**](https://learn.microsoft.com/zh-cn/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) 函数使管道服务器能够指定子进程是继承其可继承的所有句柄还是不继承任何句柄。

当子进程继承管道句柄时，系统将允许该进程访问管道。 但是，父进程必须将句柄值传达给子进程。 父进程通常通过将标准输出句柄重定向到子进程来执行此操作，如以下步骤所示：

1. 调用 [**GetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) 函数以获取当前标准输出句柄;保存此句柄，以便在创建子进程后还原原始标准输出句柄。
2. 调用 [**SetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/setstdhandle) 函数，将标准输出句柄设置为管道的写入句柄。 现在，父进程可以创建子进程。
3. 调用 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 函数以关闭管道的写入句柄。 子进程继承写入句柄后，父进程不再需要其副本。
4. 调用 [**SetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/setstdhandle) 以还原原始标准输出句柄。

子进程使用 [**GetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) 函数获取其标准输出句柄，该句柄现在是管道写入端的句柄。 然后，子进程使用 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-writefile) 函数将其输出发送到管道。 当子项完成管道处理后，它应通过调用 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-closehandle) 或通过终止（自动关闭句柄）来关闭管道句柄。

父进程使用 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 函数从管道接收输入。 数据作为字节流写入匿名管道。 这意味着，从管道读取的父进程无法区分在单独的写入操作中写入的字节，除非父进程和子进程都使用协议来指示写入操作的结束位置。 关闭管道的所有写入句柄时， **ReadFile** 函数将返回零。 在调用 **ReadFile** 之前，父进程必须将其句柄关闭到管道的写入端。 如果未完成此操作， **ReadFile** 操作将无法返回零，因为父进程具有指向管道写入端的打开句柄。

重定向标准输入句柄的过程与重定向标准输出句柄的过程类似，只不过管道的读取句柄用作子级的标准输入句柄。 在这种情况下，父进程必须确保子进程不会继承管道的写入句柄。 如果未完成此操作，则子进程执行的 [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/desktop/api/fileapi/nf-fileapi-readfile) 操作无法返回零，因为子进程具有指向管道写入端的打开句柄。

有关使用匿名管道重定向子进程的标准句柄的示例程序，请参阅 [使用重定向的输入和输出创建子进程](https://learn.microsoft.com/zh-cn/windows/desktop/ProcThread/creating-a-child-process-with-redirected-input-and-output)。

## GetStdHandle 函数语法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 语法 C  HANDLE WINAPI GetStdHandle(  \_In\_ DWORD nStdHandle  ); 参数 nStdHandle [in] 标准设备。 此参数的取值可为下列值之一。  展开表   | **值** | **含义** | | --- | --- | | **STD\_INPUT\_HANDLE**((DWORD)-10) | 标准输入设备。 最初，这是输入缓冲区 CONIN$ 的控制台。 | | **STD\_OUTPUT\_HANDLE**((DWORD)-11) | 标准输出设备。 最初，这是活动控制台屏幕缓冲区 CONOUT$。 | | **STD\_ERROR\_HANDLE**((DWORD)-12) | 标准错误设备。 最初，这是活动控制台屏幕缓冲区 CONOUT$。 |   **备注**  这些常量的值都是无符号数，但是在头文件中被定义为从有符号数的强制转换，并利用 C 编译器将它们滚动到刚好低于最大 32 位值。 在以不分析标头并重新定义常量的语言与这些句柄交互时，请注意此约束。 例如，((DWORD)-10) 实际上是无符号数 4294967286。 返回值 如果该函数成功，则返回值为指定设备的句柄，或为由先前对 [SetStdHandle](https://learn.microsoft.com/zh-cn/windows/console/setstdhandle) 的调用设置的重定向句柄。 除非应用程序已使用 SetStdHandle 来设置具有较少访问权限的标准句柄，否则该句柄具有 GENERIC\_READ 和 GENERIC\_WRITE 访问权限。  **提示**  完成后，无需使用 [**CloseHandle**](https://learn.microsoft.com/zh-cn/windows/win32/api/handleapi/nf-handleapi-closehandle) 释放此句柄。 有关详细信息，请参阅[**注解**](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle#handle-disposal)。  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**](https://learn.microsoft.com/zh-cn/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror)。  如果应用程序没有关联的标准句柄（例如在交互式桌面上运行的服务），并且尚未重定向这些句柄，则返回值为 NULL。 备注 GetStdHandle 返回的句柄可供需要在控制台中进行读取或写入的应用程序使用。 创建控制台时，标准输入句柄是控制台输入缓冲区的句柄，而标准输出和标准错误句柄则是控制台的活动屏幕缓冲区的句柄。 这些句柄可供 [ReadFile](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile) 和 [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-writefile) 函数使用，也可供访问控制台输入缓冲区或屏幕缓冲区的任何控制台函数（例如 [ReadConsoleInput](https://learn.microsoft.com/zh-cn/windows/console/readconsoleinput)、[WriteConsole](https://learn.microsoft.com/zh-cn/windows/console/writeconsole) 或 [GetConsoleScreenBufferInfo](https://learn.microsoft.com/zh-cn/windows/console/getconsolescreenbufferinfo) 函数）使用。  可通过调用 [SetStdHandle](https://learn.microsoft.com/zh-cn/windows/console/setstdhandle) 来重定向进程的标准句柄，在这种情况下，GetStdHandle 会返回重定向的句柄。 如果已重定向标准句柄，则可在对 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-createfilea) 函数的调用中指定 CONIN$ 值，以获取控制台输入缓冲区的句柄。 同样，你也可指定 CONOUT$ 值来获取控制台的活动屏幕缓冲区的句柄。  以 Main 方法为入口点的进程的标准句柄取决于生成应用程序时传递给链接器的 [/SUBSYSTEM](https://learn.microsoft.com/zh-cn/cpp/build/reference/subsystem-specify-subsystem) 标记的配置。 指定 /SUBSYSTEM:CONSOLE 请求操作系统使用控制台会话在启动时填充句柄，前提是父对象尚未通过继承填充标准句柄表。 相反，/SUBSYSTEM:WINDOWS 意味着应用程序不需要控制台，并且可能不会使用标准句柄。 有关句柄继承的详细信息，请参阅 [STARTF\_USESTDHANDLES](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 的文档。  某些应用程序在其声明的子系统的边界外运行；例如，/SUBSYSTEM:WINDOWS 应用程序可能会检查/使用用于日志记录或调试的标准句柄，但使用图形用户界面正常运行。 这些应用程序需要仔细探查标准句柄的状态，并在需要时使用 [AttachConsole](https://learn.microsoft.com/zh-cn/windows/console/attachconsole)、[AllocConsole](https://learn.microsoft.com/zh-cn/windows/console/allocconsole) 和 [FreeConsole](https://learn.microsoft.com/zh-cn/windows/console/freeconsole) 添加/删除控制台。  某些应用程序可能还会改变其对继承句柄类型的行为。 可使用 [GetFileType](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-getfiletype) 消除控制台、管道和文件等之间各类型的歧义。 句柄释放 处理完从 GetStdHandle 检索到的句柄时，不需要执行 [CloseHandle](https://learn.microsoft.com/zh-cn/windows/win32/api/handleapi/nf-handleapi-closehandle)。 返回的值只是进程表中存储的值的副本。 进程本身通常被视为这些句柄及其生存期的所有者。 每个句柄在创建时都放置在表中，具体取决于 [CreateProcess](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw) 调用的继承和启动具体信息，并且将在进程销毁时释放。  对于有意尝试替换这些句柄或阻止进程的其他部分使用它们的应用程序来说，手动操作这些句柄的生存期可能是可取的。 由于可通过运行代码缓存 HANDLE，因此该代码不一定会拾取通过 SetStdHandle 所做的更改。 通过 CloseHandle 显式关闭句柄将在进程范围内关闭该句柄，并且任何缓存引用的下一次使用都将遇到错误。  在进程表中替换标准句柄的指导原则是使用 GetStdHandle 从表中获取现有的 HANDLE，使用 SetStdHandle 将新的 HANDLE 放入使用 CreateFile（或类似函数）打开的文件中，然后关闭检索到的句柄。  GetStdHandle 或 SetStdHandle 函数都不验证作为句柄存储在进程表中的值。 在实际读/写操作（如 ReadFile 或 WriteFile）时执行验证。 附加/分离行为 附加到新控制台时，除非在进程创建期间指定了 STARTF\_USESTDHANDLES，否则标准句柄始终会替换为控制台句柄。  如果标准句柄的现有值为 NULL，或者标准句柄的现有值与控制台句柄 pseudohandle 类似，则该句柄会被替换为控制台句柄。  当父对象使用 CREATE\_NEW\_CONSOLE 和 STARTF\_USESTDHANDLES 创建控制台进程时，除非标准句柄的现有值为 NULL 或为 控制台句柄 pseudohandle，否则不会替换标准句柄。  **备注**  控制台进程启动时必须已填充标准句柄，否则它们将使用新控制台的适当句柄自动进行填充。 图形用户界面 (GUI) 应用程序可在没有标准句柄的情况下启动，它们不会自动进行填充。 示例 有关示例，请参阅[读取输入缓冲区事件](https://learn.microsoft.com/zh-cn/windows/console/reading-input-buffer-events)。 要求  |  |  | | --- | --- | | Header | ProcessEnv.h (via Winbase.h, include Windows.h) | | 库 | Kernel32.lib | | DLL | Kernel32.dll |  另请参阅 [控制台函数](https://learn.microsoft.com/zh-cn/windows/console/console-functions)  [控制台句柄](https://learn.microsoft.com/zh-cn/windows/console/console-handles)  [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-createfilea)  [**GetConsoleScreenBufferInfo**](https://learn.microsoft.com/zh-cn/windows/console/getconsolescreenbufferinfo)  [**ReadConsoleInput**](https://learn.microsoft.com/zh-cn/windows/console/readconsoleinput)  [**ReadFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-readfile)  [**SetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/setstdhandle)  [**WriteConsole**](https://learn.microsoft.com/zh-cn/windows/console/writeconsole)  [**WriteFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-writefile) |

## 实例

|  |
| --- |
| **读取输入缓冲区事件**  [**ReadConsoleInput**](https://learn.microsoft.com/zh-cn/windows/console/readconsoleinput) 函数可用于直接访问控制台的输入缓冲区。 在创建控制台时，将启用鼠标输入并禁用窗口输入。 为了确保进程接收所有类型的事件，此示例使用 [**SetConsoleMode**](https://learn.microsoft.com/zh-cn/windows/console/setconsolemode) 函数来启用窗口和鼠标输入。 然后，它会进入一个循环，读取和处理 100 个控制台输入事件。 例如，当用户按下键时显示消息“键盘事件”，而当用户与鼠标交互时显示消息“鼠标事件”。  C  #include <windows.h>  #include <stdio.h>  HANDLE hStdin;  DWORD fdwSaveOldMode;  VOID ErrorExit(LPCSTR);  VOID KeyEventProc(KEY\_EVENT\_RECORD);  VOID MouseEventProc(MOUSE\_EVENT\_RECORD);  VOID ResizeEventProc(WINDOW\_BUFFER\_SIZE\_RECORD);  int main(VOID)  {  DWORD cNumRead, fdwMode, i;  INPUT\_RECORD irInBuf[128];  int counter=0;  // Get the standard input handle.  hStdin = GetStdHandle(STD\_INPUT\_HANDLE);  if (hStdin == INVALID\_HANDLE\_VALUE)  ErrorExit("GetStdHandle");  // Save the current input mode, to be restored on exit.  if (! GetConsoleMode(hStdin, &fdwSaveOldMode) )  ErrorExit("GetConsoleMode");  // Enable the window and mouse input events.  fdwMode = ENABLE\_WINDOW\_INPUT | ENABLE\_MOUSE\_INPUT;  if (! SetConsoleMode(hStdin, fdwMode) )  ErrorExit("SetConsoleMode");  // Loop to read and handle the next 100 input events.  while (counter++ <= 100)  {  // Wait for the events.  if (! ReadConsoleInput(  hStdin, // input buffer handle  irInBuf, // buffer to read into  128, // size of read buffer  &cNumRead) ) // number of records read  ErrorExit("ReadConsoleInput");  // Dispatch the events to the appropriate handler.  for (i = 0; i < cNumRead; i++)  {  switch(irInBuf[i].EventType)  {  case KEY\_EVENT: // keyboard input  KeyEventProc(irInBuf[i].Event.KeyEvent);  break;  case MOUSE\_EVENT: // mouse input  MouseEventProc(irInBuf[i].Event.MouseEvent);  break;  case WINDOW\_BUFFER\_SIZE\_EVENT: // scrn buf. resizing  ResizeEventProc( irInBuf[i].Event.WindowBufferSizeEvent );  break;  case FOCUS\_EVENT: // disregard focus events  case MENU\_EVENT: // disregard menu events  break;  default:  ErrorExit("Unknown event type");  break;  }  }  }  // Restore input mode on exit.  SetConsoleMode(hStdin, fdwSaveOldMode);  return 0;  }  VOID ErrorExit (LPSTR lpszMessage)  {  fprintf(stderr, "%s\n", lpszMessage);  // Restore input mode on exit.  SetConsoleMode(hStdin, fdwSaveOldMode);  ExitProcess(0);  }  VOID KeyEventProc(KEY\_EVENT\_RECORD ker)  {  printf("Key event: ");  if(ker.bKeyDown)  printf("key pressed\n");  else printf("key released\n");  }  VOID MouseEventProc(MOUSE\_EVENT\_RECORD mer)  {  #ifndef MOUSE\_HWHEELED  #define MOUSE\_HWHEELED 0x0008  #endif  printf("Mouse event: ");  switch(mer.dwEventFlags)  {  case 0:  if(mer.dwButtonState == FROM\_LEFT\_1ST\_BUTTON\_PRESSED)  {  printf("left button press \n");  }  else if(mer.dwButtonState == RIGHTMOST\_BUTTON\_PRESSED)  {  printf("right button press \n");  }  else  {  printf("button press\n");  }  break;  case DOUBLE\_CLICK:  printf("double click\n");  break;  case MOUSE\_HWHEELED:  printf("horizontal mouse wheel\n");  break;  case MOUSE\_MOVED:  printf("mouse moved\n");  break;  case MOUSE\_WHEELED:  printf("vertical mouse wheel\n");  break;  default:  printf("unknown\n");  break;  }  }  VOID ResizeEventProc(WINDOW\_BUFFER\_SIZE\_RECORD wbsr)  {  printf("Resize event\n");  printf("Console screen buffer is %d columns by %d rows.\n", wbsr.dwSize.X, wbsr.dwSize.Y);  } |

## SetStdHandle 函数语法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 设置用于指定的标准设备的句柄（标准输入、标准输出或标准错误）。 语法 C++复制  BOOL WINAPI SetStdHandle(  \_In\_ DWORD  nStdHandle,  \_In\_ HANDLE hHandle  ); 参数 nStdHandle [in] 要为其设置句柄的标准设备。 此参数的取值可为下列值之一。  展开表   | **值** | **含义** | | --- | --- | | **STD\_INPUT\_HANDLE**((DWORD)-10) | 标准输入设备。 最初，这是输入缓冲区 CONIN$ 的控制台。 | | **STD\_OUTPUT\_HANDLE**((DWORD)-11) | 标准输出设备。 最初，这是活动控制台屏幕缓冲区 CONOUT$。 | | **STD\_ERROR\_HANDLE**((DWORD)-12) | 标准错误设备。 最初，这是活动控制台屏幕缓冲区 CONOUT$。 |   **备注**  这些常量的值都是无符号数，但是在头文件中被定义为从有符号数的强制转换，并利用 C 编译器将它们滚动到刚好低于最大 32 位值。 在以不分析标头并重新定义常量的语言与这些句柄交互时，请注意此约束。 例如，((DWORD)-10) 实际上是无符号数 4294967286。  hHandle [in] 标准设备的句柄。 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**](https://learn.microsoft.com/zh-cn/windows/win32/api/errhandlingapi/nf-errhandlingapi-getlasterror)。 备注 进程的标准句柄可通过调用 **SetStdHandle** 来重定向，在这种情况下 [**GetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) 将返回重定向的句柄。 如果已重定向标准句柄，则可在对 [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-createfilea) 函数的调用中指定 CONIN$ 值，以获取控制台输入缓冲区的句柄。 同样，你也可指定 CONOUT$ 值来获取控制台的活动屏幕缓冲区的句柄。 示例 有关示例，请参阅[使用重定向输入和输出创建子进程](https://learn.microsoft.com/zh-cn/windows/win32/procthread/creating-a-child-process-with-redirected-input-and-output)。 要求  |  |  | | --- | --- | | Header | ProcessEnv.h (via Winbase.h, include Windows.h) | | 库 | Kernel32.lib | | DLL | Kernel32.dll |  另请参阅 [控制台函数](https://learn.microsoft.com/zh-cn/windows/console/console-functions)  [控制台句柄](https://learn.microsoft.com/zh-cn/windows/console/console-handles)  [**CreateFile**](https://learn.microsoft.com/zh-cn/windows/win32/api/fileapi/nf-fileapi-createfilea)  [**GetStdHandle**](https://learn.microsoft.com/zh-cn/windows/console/getstdhandle) |

## 实例

|  |
| --- |
| **使用重定向输入和输出创建子进程**  本主题中的示例演示了如何使用 [**CreateProcess**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa) 函数从控制台进程创建子进程。 它还演示了一种使用匿名管道重定向子进程的标准输入和输出句柄的技术。 请注意，命名管道还可用于重定向进程 I/O。  [**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/desktop/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数使用 [**SECURITY\_ATTRIBUTES**](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构创建两个管道的读取和写入端的可继承句柄。 一个管道的读取端充当子进程的标准输入，另一个管道的写入端是子进程的标准输出。 这些管道句柄在 [**STARTUPINFO**](https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/ns-processthreadsapi-startupinfoa) 结构中指定，这使得它们成为子进程继承的标准句柄。  父进程使用这两个管道的对端写入子进程的输入并从子进程的输出中读取。 如 [**SECURITY\_ATTRIBUTES**](https://learn.microsoft.com/zh-cn/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) 结构中指定的，这些句柄也是可继承的。 但是，不能继承这些句柄。 因此，在创建子进程之前，父进程使用 [**SetHandleInformation**](https://learn.microsoft.com/zh-cn/windows/desktop/api/handleapi/nf-handleapi-sethandleinformation) 函数来确保无法继承子进程标准输入的写入句柄和子进程标准输出的读取句柄。 有关详细信息，请参阅 [管道](https://learn.microsoft.com/zh-cn/windows/desktop/ipc/pipes)。  下面是父进程的代码。 它采用单个命令行参数：文本文件的名称。  C++  #include <windows.h>  #include <tchar.h>  #include <stdio.h>  #include <strsafe.h>  #define BUFSIZE 4096    HANDLE g\_hChildStd\_IN\_Rd = NULL;  HANDLE g\_hChildStd\_IN\_Wr = NULL;  HANDLE g\_hChildStd\_OUT\_Rd = NULL;  HANDLE g\_hChildStd\_OUT\_Wr = NULL;  HANDLE g\_hInputFile = NULL;    void CreateChildProcess(void);  void WriteToPipe(void);  void ReadFromPipe(void);  void ErrorExit(PCTSTR);    int \_tmain(int argc, TCHAR \*argv[])  {  SECURITY\_ATTRIBUTES saAttr;    printf("\n->Start of parent execution.\n");  // Set the bInheritHandle flag so pipe handles are inherited.    saAttr.nLength = sizeof(SECURITY\_ATTRIBUTES);  saAttr.bInheritHandle = TRUE;  saAttr.lpSecurityDescriptor = NULL;  // Create a pipe for the child process's STDOUT.    if ( ! CreatePipe(&g\_hChildStd\_OUT\_Rd, &g\_hChildStd\_OUT\_Wr, &saAttr, 0) )  ErrorExit(TEXT("StdoutRd CreatePipe"));  // Ensure the read handle to the pipe for STDOUT is not inherited.  if ( ! SetHandleInformation(g\_hChildStd\_OUT\_Rd, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdout SetHandleInformation"));  // Create a pipe for the child process's STDIN.    if (! CreatePipe(&g\_hChildStd\_IN\_Rd, &g\_hChildStd\_IN\_Wr, &saAttr, 0))  ErrorExit(TEXT("Stdin CreatePipe"));  // Ensure the write handle to the pipe for STDIN is not inherited.    if ( ! SetHandleInformation(g\_hChildStd\_IN\_Wr, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdin SetHandleInformation"));    // Create the child process.    CreateChildProcess();  // Get a handle to an input file for the parent.  // This example assumes a plain text file and uses string output to verify data flow.    if (argc == 1)  ErrorExit(TEXT("Please specify an input file.\n"));  g\_hInputFile = CreateFile(  argv[1],  GENERIC\_READ,  0,  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_READONLY,  NULL);  if ( g\_hInputFile == INVALID\_HANDLE\_VALUE )  ErrorExit(TEXT("CreateFile"));    // Write to the pipe that is the standard input for a child process.  // Data is written to the pipe's buffers, so it is not necessary to wait  // until the child process is running before writing data.    WriteToPipe();  printf( "\n->Contents of %S written to child STDIN pipe.\n", argv[1]);    // Read from pipe that is the standard output for child process.    printf( "\n->Contents of child process STDOUT:\n\n");  ReadFromPipe();  printf("\n->End of parent execution.\n");  // The remaining open handles are cleaned up when this process terminates.  // To avoid resource leaks in a larger application, close handles explicitly.  return 0;  }    void CreateChildProcess()  // Create a child process that uses the previously created pipes for STDIN and STDOUT.  {  TCHAR szCmdline[]=TEXT("child");  PROCESS\_INFORMATION piProcInfo;  STARTUPINFO siStartInfo;  BOOL bSuccess = FALSE;    // Set up members of the PROCESS\_INFORMATION structure.    ZeroMemory( &piProcInfo, sizeof(PROCESS\_INFORMATION) );    // Set up members of the STARTUPINFO structure.  // This structure specifies the STDIN and STDOUT handles for redirection.    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );  siStartInfo.cb = sizeof(STARTUPINFO);  siStartInfo.hStdError = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdOutput = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdInput = g\_hChildStd\_IN\_Rd;  siStartInfo.dwFlags |= STARTF\_USESTDHANDLES;    // Create the child process.    bSuccess = CreateProcess(NULL,  szCmdline, // command line  NULL, // process security attributes  NULL, // primary thread security attributes  TRUE, // handles are inherited  0, // creation flags  NULL, // use parent's environment  NULL, // use parent's current directory  &siStartInfo, // STARTUPINFO pointer  &piProcInfo); // receives PROCESS\_INFORMATION    // If an error occurs, exit the application.  if ( ! bSuccess )  ErrorExit(TEXT("CreateProcess"));  else  {  // Close handles to the child process and its primary thread.  // Some applications might keep these handles to monitor the status  // of the child process, for example.  CloseHandle(piProcInfo.hProcess);  CloseHandle(piProcInfo.hThread);    // Close handles to the stdin and stdout pipes no longer needed by the child process.  // If they are not explicitly closed, there is no way to recognize that the child process has ended.    CloseHandle(g\_hChildStd\_OUT\_Wr);  CloseHandle(g\_hChildStd\_IN\_Rd);  }  }    void WriteToPipe(void)  // Read from a file and write its contents to the pipe for the child's STDIN.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;    for (;;)  {  bSuccess = ReadFile(g\_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);  if ( ! bSuccess || dwRead == 0 ) break;    bSuccess = WriteFile(g\_hChildStd\_IN\_Wr, chBuf, dwRead, &dwWritten, NULL);  if ( ! bSuccess ) break;  }    // Close the pipe handle so the child process stops reading.    if ( ! CloseHandle(g\_hChildStd\_IN\_Wr) )  ErrorExit(TEXT("StdInWr CloseHandle"));  }    void ReadFromPipe(void)  // Read output from the child process's pipe for STDOUT  // and write to the parent process's pipe for STDOUT.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;  HANDLE hParentStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);  for (;;)  {  bSuccess = ReadFile( g\_hChildStd\_OUT\_Rd, chBuf, BUFSIZE, &dwRead, NULL);  if( ! bSuccess || dwRead == 0 ) break;  bSuccess = WriteFile(hParentStdOut, chBuf,  dwRead, &dwWritten, NULL);  if (! bSuccess ) break;  }  }    void ErrorExit(PCTSTR lpszFunction)  // Format a readable error message, display a message box,  // and exit from the application.  {  LPVOID lpMsgBuf;  LPVOID lpDisplayBuf;  DWORD dw = GetLastError();  FormatMessage(  FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |  FORMAT\_MESSAGE\_FROM\_SYSTEM |  FORMAT\_MESSAGE\_IGNORE\_INSERTS,  NULL,  dw,  MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),  (LPTSTR) &lpMsgBuf,  0, NULL );  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM\_ZEROINIT,  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)\*sizeof(TCHAR));  StringCchPrintf((LPTSTR)lpDisplayBuf,  LocalSize(lpDisplayBuf) / sizeof(TCHAR),  TEXT("%s failed with error %d: %s"),  lpszFunction, dw, lpMsgBuf);  MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB\_OK);  LocalFree(lpMsgBuf);  LocalFree(lpDisplayBuf);  ExitProcess(1);  }  下面是子进程的代码。 它使用 STDIN 和 STDOUT 继承的句柄来访问父级创建的管道。 父进程从其输入文件读取信息并将信息写入管道。 子级使用 STDIN 通过管道接收文本，并使用 STDOUT 写入管道。 父级从管道的读取端读取，并将信息显示到其 STDOUT。  C++复制  #include <windows.h>  #include <stdio.h>  #define BUFSIZE 4096    int main(void)  {  CHAR chBuf[BUFSIZE];  DWORD dwRead, dwWritten;  HANDLE hStdin, hStdout;  BOOL bSuccess;    hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);  hStdin = GetStdHandle(STD\_INPUT\_HANDLE);  if (  (hStdout == INVALID\_HANDLE\_VALUE) ||  (hStdin == INVALID\_HANDLE\_VALUE)  )  ExitProcess(1);    // Send something to this process's stdout using printf.  printf("\n \*\* This is a message from the child process. \*\* \n");  // This simple algorithm uses the existence of the pipes to control execution.  // It relies on the pipe buffers to ensure that no data is lost.  // Larger applications would use more advanced process control.  for (;;)  {  // Read from standard input and stop on error or no data.  bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);    if (! bSuccess || dwRead == 0)  break;    // Write to standard output and stop on error.  bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);    if (! bSuccess)  break;  }  return 0;  } |

**匿名管道安全性和访问权限**

Windows 安全性使你能够控制对匿名管道的访问。 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 [**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数时，可以为管道指定[安全描述符](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-descriptors)。 安全描述符控制对管道读取和写入端的访问。 如果指定 **NULL**，管道将获取默认安全描述符。 管道的默认安全描述符中的 ACL 来自创建者的主要令牌或模拟令牌。

若要检索管道的安全描述符，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。

[**CreatePipe**](https://learn.microsoft.com/zh-cn/windows/win32/api/namedpipeapi/nf-namedpipeapi-createpipe) 函数向匿名管道返回两个句柄：具有GENERIC\_READ和 SYNCHRONIZE 访问权限的读取句柄;以及具有GENERIC\_WRITE和 SYNCHRONIZE 访问权限的写入句柄。 GENERIC\_READ和GENERIC\_WRITE访问使用与命名管道相同的访问权限映射。

匿名管道GENERIC\_READ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的 DACL 的权限。

匿名管道的GENERIC\_WRITE访问权限结合了将数据写入管道、向其追加数据、写入管道属性、写入扩展属性和读取管道 DACL 的权限。

## GetSecurityInfo 函数语法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **GetSecurityInfo** 函数检索句柄指定的对象的[安全描述符](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/s-gly)的副本。 语法 C++  DWORD GetSecurityInfo(  [in] HANDLE handle,  [in] SE\_OBJECT\_TYPE ObjectType,  [in] SECURITY\_INFORMATION SecurityInfo,  [out, optional] PSID \*ppsidOwner,  [out, optional] PSID \*ppsidGroup,  [out, optional] PACL \*ppDacl,  [out, optional] PACL \*ppSacl,  [out, optional] PSECURITY\_DESCRIPTOR \*ppSecurityDescriptor  ); 参数 [in] handle  要从中检索安全信息的 对象的句柄。  [in] ObjectType  [SE\_OBJECT\_TYPE](https://learn.microsoft.com/zh-cn/windows/desktop/api/accctrl/ne-accctrl-se_object_type) 枚举值，该值指示对象的类型。  [in] SecurityInfo  一组位标志，指示要检索的安全信息的类型。 此参数可以是 [SECURITY\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-information) 位标志的组合。  [out, optional] ppsidOwner  指向变量的指针，该变量接收指向 *ppSecurityDescriptor* 中返回的安全描述符中所有者 SID 的指针。 仅当设置OWNER\_SECURITY\_INFORMATION标志时，返回的指针才有效。 如果不需要所有者 SID，此参数可以为 **NULL** 。  [out, optional] ppsidGroup  指向变量的指针，该变量接收指向返回 [的安全描述符](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/s-gly)中主组 SID 的指针。 仅当设置了GROUP\_SECURITY\_INFORMATION标志时，返回的指针才有效。 如果不需要组 SID，此参数可以为 **NULL** 。  [out, optional] ppDacl  指向变量的指针，该变量接收指向返回的安全描述符中 DACL 的指针。 仅当设置了DACL\_SECURITY\_INFORMATION标志时，返回的指针才有效。 如果不需要 DACL，此参数可以为 **NULL** 。  [out, optional] ppSacl  指向变量的指针，该变量接收指向返回的安全描述符中 SACL 的指针。 仅当设置SACL\_SECURITY\_INFORMATION标志时，返回的指针才有效。 如果不需要 SACL，此参数可以为 **NULL** 。  [out, optional] ppSecurityDescriptor  指向变量的指针，该变量接收指向对象的安全描述符的指针。 使用完指针后，通过调用 [LocalFree](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-localfree) 函数释放返回的缓冲区。  如果*任何 ppsidOwner、ppsidGroup*、*ppDacl* 或 *ppSacl* 参数不为 **NULL**，则此参数是必需的。 返回值 如果函数成功，则返回值为 ERROR\_SUCCESS。  如果函数失败，则返回值为 WinError.h 中定义的非零错误代码。 注解 如果 *ppsidOwner*、 *ppsidGroup*、 *ppDacl* 和 *ppSacl* 参数为非 **NULL**，并且 *SecurityInfo* 参数指定从对象中检索它们，则这些参数将指向 *ppSecurityDescriptor* 中返回的安全描述符中的相应参数。  若要从对象的安全描述符读取所有者、组或 DACL，必须在打开句柄时授予调用进程READ\_CONTROL访问权限。 若要获取READ\_CONTROL访问权限，调用方必须是对象的所有者，或者对象的 DACL 必须授予访问权限。  若要从安全描述符读取 SACL，必须在打开句柄时授予调用进程ACCESS\_SYSTEM\_SECURITY访问权限。 获取此访问权限的正确方法是在调用方当前令牌中启用SE\_SECURITY\_NAME特权，打开ACCESS\_SYSTEM\_SECURITY访问的句柄，然后禁用该权限。 有关启用特权的安全隐患的信息，请参阅 [使用特殊特权运行](https://learn.microsoft.com/zh-cn/windows/desktop/SecBP/running-with-special-privileges)。  可以将 **GetSecurityInfo** 函数与以下类型的对象一起使用：   * NTFS 文件系统上的本地或远程文件或目录 * Named pipes * 本地或远程打印机 * 本地或远程 Windows 服务 * 网络共享 * 注册表项 * 信号灯、事件、互斥体和可等待计时器 * 进程、线程、作业和文件映射对象 * 交互式服务窗口工作站和桌面 * 目录服务对象   此函数不处理争用条件。 如果线程在另一个线程更改对象的安全描述符的大致时间调用此函数，则此函数可能会失败。 示例 有关使用此函数的示例，请参阅 [查找文件对象的所有者](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/finding-the-owner-of-a-file-object-in-c--)。 要求  |  |  | | --- | --- | | **标头** | aclapi.h | | **Library** | Advapi32.lib | | **DLL** | Advapi32.dll |  另请参阅 [ACL](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-acl)  [访问控制概述](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-control)  [基本访问控制函数](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/authorization-functions)  [GetNamedSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-getnamedsecurityinfoa)  [LocalFree](https://learn.microsoft.com/zh-cn/windows/desktop/api/winbase/nf-winbase-localfree)  [特权常量](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/authorization-constants)  [SECURITY\_DESCRIPTOR](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-security_descriptor)  [SECURITY\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-information)  [SE\_OBJECT\_TYPE](https://learn.microsoft.com/zh-cn/windows/desktop/api/accctrl/ne-accctrl-se_object_type)  [SID](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-sid)  [SetNamedSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-setnamedsecurityinfoa)  [SetSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-setsecurityinfo) |

## 实例

|  |
| --- |
| **在 C++ 中查找文件对象的所有者**  以下示例使用 [**GetSecurityInfo**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Aclapi/nf-aclapi-getsecurityinfo) 和 [**LookupAccountSid**](https://learn.microsoft.com/zh-cn/windows/desktop/api/Winbase/nf-winbase-lookupaccountsida) 函数查找并打印文件所有者的名称。 文件存在于本地服务器上的当前工作目录中。  C++  #include <stdio.h>  #include <windows.h>  #include <tchar.h>  #include "accctrl.h"  #include "aclapi.h"  #pragma comment(lib, "advapi32.lib")  int main(void)  {  DWORD dwRtnCode = 0;  PSID pSidOwner = NULL;  BOOL bRtnBool = TRUE;  LPTSTR AcctName = NULL;  LPTSTR DomainName = NULL;  DWORD dwAcctName = 1, dwDomainName = 1;  SID\_NAME\_USE eUse = SidTypeUnknown;  HANDLE hFile;  PSECURITY\_DESCRIPTOR pSD = NULL;  // Get the handle of the file object.  hFile = CreateFile(  TEXT("myfile.txt"),  GENERIC\_READ,  FILE\_SHARE\_READ,  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  // Check GetLastError for CreateFile error code.  if (hFile == INVALID\_HANDLE\_VALUE) {  DWORD dwErrorCode = 0;  dwErrorCode = GetLastError();  \_tprintf(TEXT("CreateFile error = %d\n"), dwErrorCode);  return -1;  }  // Get the owner SID of the file.  dwRtnCode = GetSecurityInfo(  hFile,  SE\_FILE\_OBJECT,  OWNER\_SECURITY\_INFORMATION,  &pSidOwner,  NULL,  NULL,  NULL,  &pSD);  // Check GetLastError for GetSecurityInfo error condition.  if (dwRtnCode != ERROR\_SUCCESS) {  DWORD dwErrorCode = 0;  dwErrorCode = GetLastError();  \_tprintf(TEXT("GetSecurityInfo error = %d\n"), dwErrorCode);  return -1;  }  // First call to LookupAccountSid to get the buffer sizes.  bRtnBool = LookupAccountSid(  NULL, // local computer  pSidOwner,  AcctName,  (LPDWORD)&dwAcctName,  DomainName,  (LPDWORD)&dwDomainName,  &eUse);  // Reallocate memory for the buffers.  AcctName = (LPTSTR)GlobalAlloc(  GMEM\_FIXED,  dwAcctName \* sizeof(wchar\_t));  // Check GetLastError for GlobalAlloc error condition.  if (AcctName == NULL) {  DWORD dwErrorCode = 0;  dwErrorCode = GetLastError();  \_tprintf(TEXT("GlobalAlloc error = %d\n"), dwErrorCode);  return -1;  }  DomainName = (LPTSTR)GlobalAlloc(  GMEM\_FIXED,  dwDomainName \* sizeof(wchar\_t));  // Check GetLastError for GlobalAlloc error condition.  if (DomainName == NULL) {  DWORD dwErrorCode = 0;  dwErrorCode = GetLastError();  \_tprintf(TEXT("GlobalAlloc error = %d\n"), dwErrorCode);  return -1;  }  // Second call to LookupAccountSid to get the account name.  bRtnBool = LookupAccountSid(  NULL, // name of local or remote computer  pSidOwner, // security identifier  AcctName, // account name buffer  (LPDWORD)&dwAcctName, // size of account name buffer  DomainName, // domain name  (LPDWORD)&dwDomainName, // size of domain name buffer  &eUse); // SID type  // Check GetLastError for LookupAccountSid error condition.  if (bRtnBool == FALSE) {  DWORD dwErrorCode = 0;  dwErrorCode = GetLastError();  if (dwErrorCode == ERROR\_NONE\_MAPPED)  \_tprintf(TEXT  ("Account owner not found for specified SID.\n"));  else  \_tprintf(TEXT("Error in LookupAccountSid.\n"));  return -1;  } else if (bRtnBool == TRUE)  // Print the account name.  \_tprintf(TEXT("Account owner = %s\n"), AcctName);  return 0;  } |

## SetSecurityInfo 函数语法

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SetSecurityInfo** 函数在指定对象[的安全描述符](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/s-gly)中设置指定的安全信息。 调用方通过句柄标识对象。  若要设置对象的 SACL，调用方必须启用 **SE\_SECURITY\_NAME** 特权。 语法 C++  DWORD SetSecurityInfo(  [in] HANDLE handle,  [in] SE\_OBJECT\_TYPE ObjectType,  [in] SECURITY\_INFORMATION SecurityInfo,  [in, optional] PSID psidOwner,  [in, optional] PSID psidGroup,  [in, optional] PACL pDacl,  [in, optional] PACL pSacl  ); 参数 [in] handle  要为其设置安全信息的 对象的句柄。  [in] ObjectType  [SE\_OBJECT\_TYPE](https://learn.microsoft.com/zh-cn/windows/desktop/api/accctrl/ne-accctrl-se_object_type) 枚举的成员，指示*句柄*参数标识的对象类型。  [in] SecurityInfo  一组位标志，指示要设置的安全信息的类型。 此参数可以是 [SECURITY\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-information) 位标志的组合。  [in, optional] psidOwner  指向标识对象所有者的 SID 的指针。 SID 必须是可以分配为安全描述符的所有者 SID 的 SID。 *SecurityInfo* 参数必须包含 OWNER\_SECURITY\_INFORMATION 标志。 如果不设置所有者 SID，此参数可以为 **NULL** 。  [in, optional] psidGroup  指向标识对象的主要组的 SID 的指针。 *SecurityInfo* 参数必须包含 GROUP\_SECURITY\_INFORMATION 标志。 如果未设置主组 SID，此参数可以为 **NULL** 。  [in, optional] pDacl  指向 对象的新 DACL 的指针。 除非 *SecurityInfo* 参数的值包含 **DACL\_SECURITY\_INFORMATION** 标志，否则将忽略此参数。 如果 *SecurityInfo* 参数的值包含 **DACL\_SECURITY\_INFORMATION** 标志，并且此参数的值设置为 **NULL**，则向所有人授予对对象的完全访问权限。 有关 **null** DACL 的信息，请参阅 [创建 DACL](https://learn.microsoft.com/zh-cn/windows/desktop/SecBP/creating-a-dacl)。  [in, optional] pSacl  指向 对象的新 SACL 的指针。 *SecurityInfo* 参数必须包含以下任何标志：SACL\_SECURITY\_INFORMATION、LABEL\_SECURITY\_INFORMATION、ATTRIBUTE\_SECURITY\_INFORMATION、SCOPE\_SECURITY\_INFORMATION或BACKUP\_SECURITY\_INFORMATION。 如果设置SACL\_SECURITY\_INFORMATION或SCOPE\_SECURITY\_INFORMATION，则调用方必须启用SE\_SECURITY\_NAME特权。 如果未设置 SACL，此参数可以为 **NULL** 。 返回值 如果函数成功，该函数将返回ERROR\_SUCCESS。  如果函数失败，它将返回 WinError.h 中定义的非零错误代码。 注解 如果要 (DACL) 或[系统访问控制列表](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/s-gly) (SACL) 对象设置任意访问控制列表，则系统会根据 [ACE 继承规则](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/ace-inheritance-rules)自动将 (ACE) 的任何可继承[访问控制条目](https://learn.microsoft.com/zh-cn/windows/desktop/SecGloss/a-gly)传播到现有子对象。  可以将 **SetSecurityInfo** 函数用于以下类型的对象：   * NTFS 上的本地或远程文件或目录 * Named pipes * 本地或远程打印机 * 本地或远程 Windows 服务 * 网络共享 * 注册表项 * 信号灯、事件、互斥体和可等待计时器 * 进程、线程、作业和文件映射对象 * 窗口工作站和桌面 * 目录服务对象   **SetSecurityInfo** 函数不会根据首选顺序重新排序允许访问或拒绝访问的 ACE。 将可继承的 ACE 传播到现有子对象时， **SetSecurityInfo** 按顺序将继承的 ACE 置于子对象的 DACL 中的所有非继承 ACE 之后。  **注意** 如果对对象的子级的共享访问权限不可用，则此函数不会将未受保护的 ACE 传播给子级。 例如，如果目录以独占访问权限打开，则当目录的安全性发生更改时，操作系统不会将未受保护的 ACE 传播到该目录的子目录或文件。    **警告** 如果打开提供的 *句柄* 时 [**ACCESS\_MASK**](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-mask) 值为 **MAXIMUM\_ALLOWED**，则 **SetSecurityInfo** 函数不会将 ACE 传播给子级。   要求  |  |  | | --- | --- | | **标头** | aclapi.h | | **Library** | Advapi32.lib | | **DLL** | Advapi32.dll |  另请参阅 [ACL](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-acl)  [访问控制](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/access-control)  [基本访问控制函数](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/authorization-functions)  [GetNamedSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-getnamedsecurityinfoa)  [GetSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-getsecurityinfo)  [SECURITY\_DESCRIPTOR](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-security_descriptor)  [SECURITY\_INFORMATION](https://learn.microsoft.com/zh-cn/windows/desktop/SecAuthZ/security-information)  [SE\_OBJECT\_TYPE](https://learn.microsoft.com/zh-cn/windows/desktop/api/accctrl/ne-accctrl-se_object_type)  [SID](https://learn.microsoft.com/zh-cn/windows/desktop/api/winnt/ns-winnt-sid)  [SetNamedSecurityInfo](https://learn.microsoft.com/zh-cn/windows/desktop/api/aclapi/nf-aclapi-setnamedsecurityinfoa) |