Welcome to the Windows Console documentation!

In the sections on the left of this page, you'll find information about the concepts, APIs and related functions, structures, etc. through which you can programmatically control and interact with the Windows Console.

# 欢迎使用Windows控制台文档！

在此页面左侧的部分中，您将找到有关概念，API和相关功能，结构等的信息，您可以通过这些信息以编程方式控制Windows控制台并与之交互。

About Character Mode Applications

Character mode (or "command-line") applications:

[Optionally] Read data from standard input (stdin)

Do "work"

[Optionally] Write data to standard output (stdout) or standard error (stderr)

Character mode applications communicate with the end-user through a "console" (or "terminal") application. A console converts user input from keyboard, mouse, touch-screen, pen, etc., and send it to a character mode application's stdin. A console may also display a character mode application's text output on the user's screen.

In Windows, the console is built-in and provides a rich API through which character mode applications can interact with the user.

Consoles

Input and Output Methods

Console Code Pages

Console Control Handlers

Console Aliases

Console Buffer Security and Access Rights

Console Application Issues

## 关于字符模式应用

字符模式（或“命令行”）应用程序：

[可选]从标准输入（stdin）读取数据

做工作”

[可选]将数据写入标准输出（stdout）或标准错误（stderr）

字符模式应用程序通过“控制台”（或“终端”）应用程序与最终用户通信。 控制台转换来自键盘，鼠标，触摸屏，笔等的用户输入，并将其发送到字符模式应用程序的stdin。 控制台还可以在用户的屏幕上显示字符模式应用程序的文本输出。

在Windows中，控制台是内置的，并提供了丰富的API，字符模式应用程序可通过该API与用户进行交互。

游戏机

输入和输出方法

控制台代码页

控制台控制处理程序

控制台别名

控制台缓冲区安全性和访问权限

控制台应用程序问题

Input and Output Methods

There are two different approaches to console I/O, the choice of which depends on how much flexibility and control an application needs. The high-level approach enables simple character stream I/O, but it limits access to a console's input and screen buffers. The low-level approach requires that developers write more code and choose among a greater range of functions, but it also gives an application more flexibility.

An application can use the file I/O functions, ReadFile and WriteFile, and the console functions, ReadConsole and WriteConsole, for high-level I/O that provides indirect access to a console's input and screen buffers. The high-level input functions filter and process the data in a console's input buffer to return input as a stream of characters, discarding mouse and buffer-resizing input. Similarly, the high-level output functions write a stream of characters that are displayed at the current cursor location in a screen buffer. An application controls the way these functions work by setting a console's I/O modes.

The low-level I/O functions provide direct access to a console's input and screen buffers, enabling an application to access mouse and buffer-resizing input events and extended information for keyboard events. Low-level output functions enable an application to read from or write to a specified number of consecutive character cells in a screen buffer, or to read or write to rectangular blocks of character cells at a specified location in a screen buffer. A console's input modes affect low-level input by enabling the application to determine whether mouse and buffer-resizing events are placed in the input buffer. A console's output modes have no effect on low-level output.

The high-level and low-level I/O methods are not mutually exclusive, and an application can use any combination of these functions. Typically, however, an application uses one approach or the other exclusively.

The following topics describe the console modes and the high-level and low-level I/O functions.

Console Modes

High-Level Console I/O

High-Level Console Modes

High-Level Console Input and Output Functions

Low-Level Console I/O

Low-Level Console Modes

Low-Level Console Input Functions

Low-Level Console Output Functions

## 输入和输出方法

控制台I / O有两种不同的方法，其选择取决于应用程序需要多少灵活性和控制力。高级方法支持简单的字符流I / O，但它限制了对控制台输入和屏幕缓冲区的访问。低级方法要求开发人员编写更多代码并在更大范围的功能中进行选择，但同时也为应用程序提供了更大的灵活性。

应用程序可以将文件I / O功能ReadFile和WriteFile以及控制台功能ReadConsole和WriteConsole用于高级别I / O，以提供对控制台输入和屏幕缓冲区的间接访问。高级输入功能过滤并处理控制台输入缓冲区中的数据，以字符流的形式返回输入，丢弃鼠标和调整缓冲区大小的输入。同样，高级输出函数将字符流写入屏幕缓冲区中当前光标位置处显示的字符流。应用程序通过设置控制台的I / O模式来控制这些功能的工作方式。

低级I / O功能提供对控制台输入和屏幕缓冲区的直接访问，使应用程序可以访问鼠标和缓冲区调整大小的输入事件以及键盘事件的扩展信息。低级输出功能使应用程序可以读取或写入屏幕缓冲区中指定数量的连续字符单元，或者读取或写入屏幕缓冲区中指定位置处的字符单元的矩形块。控制台的输入模式通过使应用程序确定是否将鼠标和缓冲区调整大小事件放置在输入缓冲区中，从而影响低级输入。控制台的输出模式对低电平输出没有影响。

高级I / O方法和低级I / O方法不是互斥的，并且应用程序可以使用这些功能的任意组合。但是，通常，应用程序仅使用一种方法或另一种方法。

以下主题描述了控制台模式以及高级和低级I / O功能。

控制台模式

高级控制台I / O

高级控制台模式

高级控制台输入和输出功能

低层控制台I / O

低层控制台模式

底层控制台输入功能

底层控制台输出功能

Console Code Pages

A code page is a mapping of 256 character codes to individual characters. Different code pages include different special characters, typically customized for a language or a group of languages.

Associated with each console are two code pages: one for input and one for output. A console uses its input code page to translate keyboard input into the corresponding character value. It uses its output code page to translate the character values written by the various output functions into the images displayed in the console window. An application can use the SetConsoleCP and GetConsoleCP functions to set and retrieve a console's input code pages and the SetConsoleOutputCP and GetConsoleOutputCP functions to set and retrieve its output code pages.

The identifiers of the code pages available on the local computer are stored in the registry under the following key.

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage

For information about using the registry functions to determine the available code pages, see Registry.

## 控制台代码页

代码页是256个字符代码到各个字符的映射。不同的代码页包含不同的特殊字符，通常针对一种语言或一组语言进行自定义。

每个控制台都有两个代码页：一个用于输入，一个用于输出。控制台使用其输入代码页将键盘输入转换为相应的字符值。它使用其输出代码页将各种输出功能写入的字符值转换为控制台窗口中显示的图像。应用程序可以使用SetConsoleCP和GetConsoleCP函数来设置和检索控制台的输入代码页，而可以使用SetConsoleOutputCP和GetConsoleOutputCP函数来设置和检索其输出代码页。

本地计算机上可用的代码页的标识符存储在注册表中的以下键下。

HKEY\_LOCAL\_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ Nls \ CodePage

有关使用注册表功能确定可用代码页的信息，请参见注册表。

Console Control Handlers

Each console process has its own list of control handler functions that are called by the system when the process receives a CTRL+C, CTRL+BREAK, or CTRL+CLOSE signal. Initially, the list of control handlers for each process contains only a default handler function that calls the ExitProcess function. A console process can add or remove additional HandlerRoutine functions by calling the SetConsoleCtrlHandler function. This function does not affect the lists of control handlers for other processes. When a console process receives any of the control signals, it calls the handler functions on a last-registered, first-called basis until one of the handlers returns TRUE. If none of the handlers returns TRUE, the default handler is called.

The function's dwCtrlType parameter identifies which control signal was received, and the return value indicates whether the signal was handled.

For an example of a control handler function, see Registering a Control Handler Function.

## 控制台控制处理程序

每个控制台进程都有其自己的控制处理程序功能列表，当进程接收到CTRL + C，CTRL + BREAK或CTRL + CLOSE信号时，系统会调用这些列表。最初，每个进程的控制处理程序列表仅包含一个默认处理程序函数，该函数调用ExitProcess函数。控制台进程可以通过调用SetConsoleCtrlHandler函数来添加或删除其他HandlerRoutine函数。此功能不会影响其他进程的控制处理程序列表。当控制台进程接收到任何控制信号时，它将在最后注册的优先级基础上调用处理程序函数，直到其中一个处理程序返回TRUE。如果没有任何处理程序返回TRUE，则调用默认处理程序。

函数的dwCtrlType参数标识接收到哪个控制信号，并且返回值指示是否处理了该信号。

有关控制处理程序功能的示例，请参见注册控制处理程序功能。

Console Aliases

Console aliases are used to map source strings to target strings. For example, you can define a console alias that maps "test" to "cd \a\_very\_long\_path\test". When you type "test" at the command line, the console subsystem expands the alias and executes the specified cd command.

To define a console alias, use Doskey.exe to create a macro, or use the AddConsoleAlias function. The following example uses Doskey.exe:

doskey test=cd \a\_very\_long\_path\test

The following call to AddConsoleAlias creates the same console alias:

AddConsoleAlias( TEXT("test"),

TEXT("cd \\<a\_very\_long\_path>\\test"),

TEXT("cmd.exe"));

To add parameters to a console alias macro using Doskey.exe, use the batch parameters $1 through $9. For more information on the special codes that can be used in Doskey macro definitions, see the command-line help for Doskey.exe or Doskey on TechNet.

All instances of an executable file running in the same console window share any defined console aliases. Multiple instances of the same executable file running in different console windows do not share console aliases. Different executable files running in the same console window do not share console aliases.

To retrieve the target string for a specified source string and executable file, use the GetConsoleAlias function. To retrieve all aliases for a specified executable file, use the GetConsoleAliases function. To retrieve the names of all aliases for which console aliases have been defined, use the GetConsoleAliasExes function.

## 控制台别名

控制台别名用于将源字符串映射到目标字符串。例如，您可以定义一个控制台别名，将“ test”映射到“ cd \ a\_very\_long\_path \ test”。在命令行上键入“ test”时，控制台子系统将扩展别名并执行指定的cd命令。

若要定义控制台别名，请使用Doskey.exe创建宏，或使用AddConsoleAlias函数。下面的示例使用Doskey.exe：

doskey test = cd \ a\_very\_long\_path \ test

对AddConsoleAlias的以下调用将创建相同的控制台别名：

AddConsoleAlias（TEXT（“ test”），

TEXT（“ cd \\ <a\_very\_long\_path> \\ test”），

TEXT（“ cmd.exe”））;

若要使用Doskey.exe将参数添加到控制台别名宏，请使用批处理参数$ 1到$ 9。有关可在Doskey宏定义中使用的特殊代码的更多信息，请参见TechNet上Doskey.exe或Doskey的命令行帮助。

在同一控制台窗口中运行的可执行文件的所有实例都共享任何已定义的控制台别名。在不同的控制台窗口中运行的同一可执行文件的多个实例不共享控制台别名。在同一控制台窗口中运行的不同可执行文件不共享控制台别名。

要检索指定源字符串和可执行文件的目标字符串，请使用GetConsoleAlias函数。要检索指定可执行文件的所有别名，请使用GetConsoleAliases函数。要检索已为其定义了控制台别名的所有别名的名称，请使用GetConsoleAliasExes函数。

Console Buffer Security and Access Rights

The Windows security model enables you to control access to console input buffers and console screen buffers. For more information about security, see Access-Control Model.

You can specify a security descriptor for the console input and console screen buffers when you call the CreateFile or CreateConsoleScreenBuffer function. If you specify NULL, the object gets a default security descriptor. The ACLs in the default security descriptor for a console buffer come from the primary or impersonation token of the creator.

The handles returned by CreateFile, CreateConsoleScreenBuffer, and GetStdHandle have the GENERIC\_READ and GENERIC\_WRITE access rights.

The valid access rights include the GENERIC\_READ and GENERIC\_WRITE generic access rights.

Value Meaning

GENERIC\_READ (0x80000000L) Requests read access to the console screen buffer, enabling the process to read data from the buffer.

GENERIC\_WRITE (0x40000000L) Requests write access to the console screen buffer, enabling the process to write data to the buffer.

## 控制台缓冲区安全性和访问权限

Windows安全模型使您可以控制对控制台输入缓冲区和控制台屏幕缓冲区的访问。有关安全性的更多信息，请参见访问控制模型。

调用CreateFile或CreateConsoleScreenBuffer函数时，可以为控制台输入和控制台屏幕缓冲区指定安全描述符。如果指定NULL，则对象将获取默认的安全描述符。控制台缓冲区的默认安全描述符中的ACL来自创建者的主令牌或模拟令牌。

由CreateFile，CreateConsoleScreenBuffer和GetStdHandle返回的句柄具有GENERIC\_READ和GENERIC\_WRITE访问权限。

有效的访问权限包括GENERIC\_READ和GENERIC\_WRITE通用访问权限。

值含义

GENERIC\_READ（0x80000000L）请求对控制台屏幕缓冲区的读取访问权，使进程能够从缓冲区中读取数据。

GENERIC\_WRITE（0x40000000L）请求对控制台屏幕缓冲区的写访问，从而使进程能够将数据写入缓冲区。

Console Application Issues

The 8-bit console functions use the OEM code page. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by the other functions and vice versa. For example, if FindFirstFileA returns a string that contains certain extended ANSI characters, WriteConsoleA will not display the string properly.

The best long-term solution for a console application is to use Unicode. Barring that solution, a console application should use the SetFileApisToOEM function. That function changes relevant file functions so that they produce OEM character set strings rather than ANSI character set strings.

The following are file functions:

CopyFile GetFileAttributes LoadLibrary

CreateDirectory GetFullPathName LoadLibraryEx

CreateFile GetModuleFileName MoveFile

CreateProcess GetModuleHandle MoveFileEx

DeleteFile GetSystemDirectory OpenFile

FindFirstFile GetTempFileName RemoveDirectory

FindNextFile GetTempPath SearchPath

GetCurrentDirectory GetVolumeInformation SetCurrentDirectory

GetDiskFreeSpace GetWindowsDirectory SetFileAttributes

GetDriveType

When dealing with command lines, a console application should obtain the command line in Unicode form and convert it to OEM form, using the relevant character-to-OEM functions. Note, also, that argv uses the ANSI character set.

## 控制台应用程序问题

8位控制台功能使用OEM代码页。默认情况下，所有其他功能都使用ANSI代码页。这意味着控制台功能返回的字符串可能无法被其他功能正确处理，反之亦然。例如，如果FindFirstFileA返回包含某些扩展ANSI字符的字符串，则WriteConsoleA将无法正确显示该字符串。

控制台应用程序的最佳长期解决方案是使用Unicode。除该解决方案外，控制台应用程序应使用SetFileApisToOEM函数。该函数更改了相关的文件函数，以便它们生成OEM字符集字符串，而不是ANSI字符集字符串。

以下是文件功能：

CopyFile GetFileAttributes LoadLibrary

CreateDirectory GetFullPathName LoadLibraryEx

CreateFile GetModuleFileName MoveFile

CreateProcess GetModuleHandle MoveFileEx

DeleteFile GetSystemDirectory OpenFile

FindFirstFile GetTempFileName RemoveDirectory

FindNextFile GetTempPath SearchPath

GetCurrentDirectory GetVolumeInformation SetCurrentDirectory

GetDiskFreeSpace GetWindowsDirectory SetFileAttributes

GetDriveType

处理命令行时，控制台应用程序应使用Unicode字符相关功能，以Unicode格式获取命令行并将其转换为OEM格式。还要注意，argv使用ANSI字符集。

Consoles

A console (or "terminal) is an application that provides I/O to character-mode applications. This processor-independent mechanism makes it easy to port existing character-mode applications or to create new character-mode tools and applications.

A console consists of an input buffer and one or more screen buffers. The input buffer contains a queue of input records, each of which contains information about an input event. The input queue always includes key-press and key-release events. It can also include mouse events (pointer movements and button presses and releases) and events during which user actions affect the size of the active screen buffer. A screen buffer is a two-dimensional array of character and color data for output in a console window. Any number of processes can share a console.

Creation of a Console

Attaching to a Console

Closing a Console

Console Handles

Console Input Buffer

Console Screen Buffers

Window and Screen Buffer Size

Console Selection

Scrolling the Screen Buffer

# 控制台

控制台（或“终端”）是为字符模式应用程序提供I / O的应用程序，这种独立于处理器的机制使移植现有字符模式应用程序或创建新的字符模式工具和应用程序变得容易。

控制台由一个输入缓冲区和一个或多个屏幕缓冲区组成。输入缓冲区包含输入记录队列，每个记录包含有关输入事件的信息。输入队列始终包含按键和按键释放事件。它还可以包括鼠标事件（指针移动以及按钮的按下和释放）以及在此期间用户操作影响活动屏幕缓冲区大小的事件。屏幕缓冲区是字符和颜色数据的二维数组，用于在控制台窗口中输出。任意数量的进程可以共享一个控制台。

创建一个控制台

附加到控制台

关闭控制台

控制台句柄

控制台输入缓冲区

控制台屏幕缓冲区

窗口和屏幕缓冲区大小

控制台选择

滚动屏幕缓冲区

Creation of a Console

The system creates a new console when it starts a console process, a character-mode process whose entry point is the main function. For example, the system creates a new console when it starts the command processor. When the command processor starts a new console process, the user can specify whether the system creates a new console for the new process or whether it inherits the command processor's console.

A process can create a console by using one of the following methods:

A GUI or console process can use the CreateProcess function with CREATE\_NEW\_CONSOLE to create a console process with a new console. (By default, a console process inherits its parent's console, and there is no guarantee that input is received by the process for which it was intended.)

A graphical user interface (GUI) or console process that is not currently attached to a console can use the AllocConsole function to create a new console. (GUI processes are not attached to a console when they are created. Console processes are not attached to a console if they are created using CreateProcess with DETACHED\_PROCESS.)

Typically, a process uses AllocConsole to create a console when an error occurs requiring interaction with the user. For example, a GUI process can create a console when an error occurs that prevents it from using its normal graphical interface, or a console process that does not normally interact with the user can create a console to display an error.

A process can also create a console by specifying the CREATE\_NEW\_CONSOLE flag in a call to CreateProcess. This method creates a new console that is accessible to the child process but not to the parent process. Separate consoles enable both parent and child processes to interact with the user without conflict. If this flag is not specified when a console process is created, both processes are attached to the same console, and there is no guarantee that the correct process will receive the input intended for it. Applications can prevent confusion by creating child processes that do not inherit handles of the input buffer, or by enabling only one child process at a time to inherit an input buffer handle while preventing the parent process from reading console input until the child has finished.

Creating a new console results in a new console window, as well as separate I/O screen buffers. The process associated with the new console uses the GetStdHandle function to get the handles of the new console's input and screen buffers. These handles enable the process to access the console.

When a process uses CreateProcess, it can specify a STARTUPINFO structure, whose members control the characteristics of the first new console (if any) created for the child process. The STARTUPINFO structure specified in the call to CreateProcess affects a console created if the CREATE\_NEW\_CONSOLE flag is specified. It also affects a console created if the child process subsequently uses AllocConsole. The following console characteristics can be specified:

Size of the new console window, in character cells

Location of the new console window, in screen pixel coordinates

Size of the new console's screen buffer, in character cells

Text and background color attributes of the new console's screen buffer

Display name for the title bar of the new console's window

The system uses default values if the STARTUPINFO values are not specified. A child process can use the GetStartupInfo function to determine the values in its STARTUPINFO structure.

A process cannot change the location of its console window on the screen, but the following console functions are available to set or retrieve the other properties specified in the STARTUPINFO structure.

Function Description

GetConsoleScreenBufferInfo Retrieves the window size, screen buffer size, and color attributes.

SetConsoleWindowInfo Changes the size of the console window.

SetConsoleScreenBufferSize Changes the size of the console screen buffer.

SetConsoleTextAttribute Sets the color attributes.

SetConsoleTitle Sets the console window title.

GetConsoleTitle Retrieves the console window title.

A process can use the FreeConsole function to detach itself from an inherited console or from a console created by AllocConsole.

## 创建一个控制台

系统在启动控制台进程时创建一个新的控制台，该控制台是字符模式进程，其入口点是主要功能。例如，系统在启动命令处理器时创建一个新的控制台。当命令处理器启动新的控制台进程时，用户可以指定系统是为新进程创建新的控制台还是继承该命令处理器的控制台。

进程可以使用以下方法之一创建控制台：

GUI或控制台进程可以将CreateProcess函数与CREATE\_NEW\_CONSOLE一起使用，以使用新控制台创建控制台进程。 （默认情况下，控制台进程会继承其父级的控制台，并且不能保证预期的进程会收到输入。）

当前未连接到控制台的图形用户界面（GUI）或控制台进程可以使用AllocConsole函数创建新的控制台。 （GUI进程在创建时未附加到控制台。如果控制台进程是使用带有DETACHED\_PROCESS的CreateProcess创建的，则它们不会附加到控制台。）

通常，当发生需要与用户交互的错误时，进程将使用AllocConsole创建控制台。例如，GUI进程可以在发生错误而无法使用其常规图形界面时创建控制台，或者通常不与用户交互的控制台进程可以创建控制台以显示错误。

进程还可以通过在对CreateProcess的调用中指定CREATE\_NEW\_CONSOLE标志来创建控制台。此方法创建一个新控制台，子进程可以访问它，但父进程不能访问。独立的控制台使父进程和子进程都可以与用户进行交互，而不会发生冲突。如果在创建控制台进程时未指定此标志，则两个进程都将附加到同一控制台，并且不能保证正确的进程将收到为其指定的输入。应用程序可以通过创建不继承输入缓冲区的句柄的子进程，或者通过一次仅启用一个子进程来继承输入缓冲区的句柄，同时防止父进程读取控制台输入直到子进程完成来避免混淆。

创建一个新的控制台会在一个新的控制台窗口中产生一个单独的I / O屏幕缓冲区。与新控制台关联的进程使用GetStdHandle函数来获取新控制台的输入和屏幕缓冲区的句柄。这些句柄使进程可以访问控制台。

当一个进程使用CreateProcess时，它可以指定一个STARTUPINFO结构，该结构的成员控制为子进程创建的第一个新控制台（如果有）的特性。如果指定了CREATE\_NEW\_CONSOLE标志，则对CreateProcess的调用中指定的STARTUPINFO结构会影响创建的控制台。如果子进程随后使用AllocConsole，它也会影响创建的控制台。可以指定以下控制台特征：

新控制台窗口的大小，以字符单元格为单位

新控制台窗口的位置，以屏幕像素坐标表示

新控制台屏幕缓冲区的大小，以字符单元格为单位

新控制台屏幕缓冲区的文本和背景颜色属性

新控制台窗口标题栏的显示名称

如果未指定STARTUPINFO值，则系统使用默认值。子进程可以使用GetStartupInfo函数来确定其STARTUPINFO结构中的值。

进程无法在屏幕上更改其控制台窗口的位置，但是以下控制台功能可用于设置或检索STARTUPINFO结构中指定的其他属性。

功能说明

GetConsoleScreenBufferInfo检索窗口大小，屏幕缓冲区大小和颜色属性。

SetConsoleWindowInfo更改控制台窗口的大小。

SetConsoleScreenBufferSize更改控制台屏幕缓冲区的大小。

SetConsoleTextAttribute设置颜色属性。

SetConsoleTitle设置控制台窗口标题。

GetConsoleTitle检索控制台窗口标题。

进程可以使用FreeConsole函数将自己与继承的控制台或由AllocConsole创建的控制台分离。

Attaching to a Console

A process can use the AttachConsole function to attach to a console. A process can be attached to one console.

A console can have many processes attached to it. To retrieve a list of the processes attached to a console, call the GetConsoleProcessList function.

## 附加到控制台

进程可以使用AttachConsole函数来附加到控制台。 进程可以附加到一个控制台。

控制台可以附加许多进程。 要检索附加到控制台的进程的列表，请调用GetConsoleProcessList函数。

Closing a Console

A process can use the FreeConsole function to detach itself from its console. If other processes share the console, the console is not destroyed, but the process that called FreeConsole cannot refer to it. After calling FreeConsole, the process can use AllocConsole to create a new console or AttachConsole to attach to another console.

A console is closed when the last process attached to it terminates or calls FreeConsole.

## 关闭控制台

进程可以使用FreeConsole函数将自身与其控制台分离。 如果其他进程共享该控制台，则不会破坏该控制台，但是调用FreeConsole的进程无法引用该控制台。 调用FreeConsole后，该过程可以使用AllocConsole创建新控制台，或使用AttachConsole附加到另一个控制台。

当控制台上附加的最后一个进程终止或调用FreeConsole时，它将关闭。

Console Handles

A console process uses handles to access the input and screen buffers of its console. A process can use the GetStdHandle, CreateFile, or CreateConsoleScreenBuffer function to open one of these handles.

The GetStdHandle function provides a mechanism for retrieving the standard input (STDIN), standard output (STDOUT), and standard error (STDERR) handles associated with a process. During console creation, the system creates these handles. Initially, STDIN is a handle to the console's input buffer, and STDOUT and STDERR are handles of the console's active screen buffer. However, the SetStdHandle function can redirect the standard handles by changing the handle associated with STDIN, STDOUT, or STDERR. Because the parent's standard handles are inherited by any child process, subsequent calls to GetStdHandle return the redirected handle. A handle returned by GetStdHandle may, therefore, refer to something other than console I/O. For example, before creating a child process, a parent process can use SetStdHandle to set a pipe handle to be the STDIN handle that is inherited by the child process. When the child process calls GetStdHandle, it gets the pipe handle. This means that the parent process can control the standard handles of the child process. The handles returned by GetStdHandle have GENERIC\_READ | GENERIC\_WRITE access unless SetStdHandle has been used to set the standard handle to have lesser access.

The value of the handles returned by GetStdHandle are not 0, 1, and 2, so the standard predefined stream constants in Stdio.h (STDIN, STDOUT, and STDERR) cannot be used in functions that require a console handle.

The CreateFile function enables a process to get a handle to its console's input buffer and active screen buffer, even if STDIN and STDOUT have been redirected. To open a handle to a console's input buffer, specify the CONIN$ value in a call to CreateFile. Specify the CONOUT$ value in a call to CreateFile to open a handle to a console's active screen buffer. CreateFile enables you to specify the read/write access of the handle that it returns.

The CreateConsoleScreenBuffer function creates a new screen buffer and returns a handle. This handle can be used in any function that accepts a handle to console output. The new screen buffer is not active until its handle is specified in a call to the SetConsoleActiveScreenBuffer function. Note that changing the active screen buffer does not affect the handle returned by GetStdHandle. Similarly, using SetStdHandle to change the STDOUT handle does not affect the active screen buffer.

Console handles returned by CreateFile and CreateConsoleScreenBuffer can be used in any of the console functions that require a handle to a console's input buffer or of a console screen buffer. Handles returned by GetStdHandle can be used by the console functions if they have not been redirected to refer to something other than console I/O. If a standard handle has been redirected to refer to a file or a pipe, however, the handle can only be used by the ReadFile and WriteFile functions.

A process can use the DuplicateHandle function to create a duplicate console handle that has different access or inheritability from the original handle. Note, however, that a process can create a duplicate console handle only for its own use. This differs from other handle types (such as file, pipe, or mutex objects), for which DuplicateHandle can create a duplicate that is valid for a different process.

To close a console handle, a process can use the CloseHandle function.

## 控制台句柄

控制台进程使用句柄访问其控制台的输入和屏幕缓冲区。进程可以使用GetStdHandle，CreateFile或CreateConsoleScreenBuffer函数打开这些句柄之一。

GetStdHandle函数提供了一种用于检索与流程关联的标准输入（STDIN），标准输出（STDOUT）和标准错误（STDERR）句柄的机制。在创建控制台期间，系统将创建这些句柄。最初，STDIN是控制台输入缓冲区的句柄，而STDOUT和STDERR是控制台活动屏幕缓冲区的句柄。但是，SetStdHandle函数可以通过更改与STDIN，STDOUT或STDERR相关联的句柄来重定向标准句柄。由于父级的标准句柄由任何子进程继承，因此对GetStdHandle的后续调用将返回重定向的句柄。因此，GetStdHandle返回的句柄可能引用的不是控制台I / O。例如，在创建子进程之前，父进程可以使用SetStdHandle将管道句柄设置为由子进程继承的STDIN句柄。当子进程调用GetStdHandle时，它将获取管道句柄。这意味着父进程可以控制子进程的标准句柄。 GetStdHandle返回的句柄具有GENERIC\_READ |除非已使用SetStdHandle将标准句柄设置为具有较小的访问权限，否则将进行GENERIC\_WRITE访问。

GetStdHandle返回的句柄的值不是0、1和2，因此Stdio.h中的标准预定义流常量（STDIN，STDOUT和STDERR）不能在需要控制台句柄的函数中使用。

即使已重定向STDIN和STDOUT，CreateFile函数也可使进程获取其控制台的输入缓冲区和活动屏幕缓冲区的句柄。要打开控制台输入缓冲区的句柄，请在对CreateFile的调用中指定CONIN $值。在对CreateFile的调用中指定CONOUT $值，以打开控制台活动屏幕缓冲区的句柄。 CreateFile使您可以指定它返回的句柄的读/写访问权限。

CreateConsoleScreenBuffer函数创建一个新的屏幕缓冲区并返回一个句柄。此句柄可用于任何接受控制台输出句柄的函数中。直到在对SetConsoleActiveScreenBuffer函数的调用中指定了它的句柄之后，新的屏幕缓冲区才处于活动状态。请注意，更改活动屏幕缓冲区不会影响GetStdHandle返回的句柄。同样，使用SetStdHandle更改STDOUT句柄也不会影响活动屏幕缓冲区。

由CreateFile和CreateConsoleScreenBuffer返回的控制台句柄可用于需要控制台输入缓冲区或控制台屏幕缓冲区的任何控制台功能。如果尚未将GetStdHandle返回的句柄重定向到控制台I / O以外的其他函数，则它们可以被控制台函数使用。但是，如果已将标准句柄重定向为引用文件或管道，则该句柄只能由ReadFile和WriteFile函数使用。

进程可以使用DuplicateHandle函数创建与原始句柄具有不同访问或继承性的重复控制台句柄。但是请注意，进程只能为自己使用而创建重复的控制台句柄。这与其他句柄类型（例如文件，管道或互斥对象）不同，DuplicateHandle可以为其创建对不同进程有效的副本。

要关闭控制台句柄，进程可以使用CloseHandle函数。

Console Input Buffer

Each console has an input buffer that contains a queue of input event records. When a console's window has the keyboard focus, a console formats each input event (such as a single keystroke, a movement of the mouse, or a mouse-button click) as an input record that it places in the console's input buffer.

Applications can access a console's input buffer indirectly by using the high-level console I/O functions, or directly by using the low-level console input functions. The high-level input functions filter and process the data in the input buffer, returning only a stream of input characters. The low-level input functions enable applications to read input records directly from a console's input buffer, or to place input records into the input buffer. To open a handle to a console's input buffer, specify the CONIN$ value in a call to the CreateFile function.

An input record is a structure containing information about the type of event that occurred (keyboard, mouse, window resizing, focus, or menu event) as well as specific details about the event. The EventType member in an INPUT\_RECORD structure indicates which type of event is contained in the record.

Focus and menu events are placed in a console's input buffer for internal use by the system and should be ignored by applications.

Keyboard Events

Keyboard events are generated when any key is pressed or released; this includes control keys. However, the ALT key has special meaning to the system when pressed and released without being combined with another character, and it is not passed through to the application. Also, the CTRL+C key combination is not passed through if the input handle is in processed mode.

If the input event is a keystroke, the Event member in INPUT\_RECORD is a KEY\_EVENT\_RECORD structure containing the following information:

A Boolean value indicating whether the key was pressed or released.

A repeat count that can be greater than one when a key is held down.

The virtual-key code, identifying the given key in a device-independent manner.

The virtual-scan code, indicating the device-dependent value generated by the keyboard hardware.

The translated Unicode™ or ANSI character.

A flag variable indicating the state of the control keys (the ALT, CTRL, SHIFT, NUM LOCK, SCROLL LOCK, and CAPS LOCK keys) and indicating whether an enhanced key was pressed. Enhanced keys for the IBM® 101-key and 102-key keyboards are the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad and the divide (/) and ENTER keys in the numeric keypad.

Mouse Events

Mouse events are generated whenever the user moves the mouse or presses or releases one of the mouse buttons. Mouse events are placed in the input buffer only if the following conditions are met:

The console input mode is set to ENABLE\_MOUSE\_INPUT (the default mode).

The console window has the keyboard focus.

The mouse pointer is within the borders of the console's window.

If the input event is a mouse event, the Event member in INPUT\_RECORD is a MOUSE\_EVENT\_RECORD structure containing the following information:

The coordinates of the mouse pointer in terms of the character-cell row and column in the console screen buffer's coordinate system.

A flag variable indicating the state of the mouse buttons.

A flag variable indicating the state of the control keys (ALT, CTRL, SHIFT, NUM LOCK, SCROLL LOCK, and CAPS LOCK) and indicating whether an enhanced key was pressed. Enhanced keys for the IBM 101-key and 102-key keyboards are the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad and the divide (/) and ENTER keys in the numeric keypad.

A flag variable indicating whether the event was a normal button-press or button-release event, a mouse movement event, or the second click of a double-click event.

Note The mouse position coordinates are in terms of the console screen buffer, not the console window. The screen buffer may have been scrolled with respect to the window, so the upper left corner of the window is not necessarily the (0,0) coordinate of the console screen buffer. To determine the coordinates of the mouse relative to the coordinate system of the window, subtract the window origin coordinates from the mouse position coordinates. Use the GetConsoleScreenBufferInfo function to determine the window origin coordinates.

The dwButtonState member of the MOUSE\_EVENT\_RECORD structure has a bit corresponding to each mouse button. The bit is 1 if the button is down and 0 if the button is up. A button-release event is detected by a 0 value for the dwEventFlags member of MOUSE\_EVENT\_RECORD and a change in a button's bit from 1 to 0. The GetNumberOfConsoleMouseButtons function retrieves the number of buttons on the mouse.

Buffer-Resizing Events

A console window's menu enables the user to change the size of the active screen buffer; this change generates a buffer-resizing event. Buffer-resizing events are placed in the input buffer if the console's input mode is set to ENABLE\_WINDOW\_INPUT (that is, the default mode is disabled).

If the input event is a buffer-resizing event, the Event member of INPUT\_RECORD is a WINDOW\_BUFFER\_SIZE\_RECORD structure containing the new size of the console screen buffer, expressed in character-cell columns and rows.

If the user reduces the size of the console screen buffer, any data in the discarded portion of the buffer is lost.

Changes to the console screen buffer size as a result of application calls to the SetConsoleScreenBufferSize function are not generated as buffer-resizing events.

## 控制台输入缓冲区

每个控制台都有一个输入缓冲区，其中包含输入事件记录队列。当控制台的窗口具有键盘焦点时，控制台会将每个输入事件（例如单个击键，鼠标的移动或鼠标按钮的单击）格式化为输入记录，并将其放置在控制台的输入缓冲区中。

应用程序可以使用高级控制台I / O功能间接访问控制台的输入缓冲区，也可以使用低级控制台输入功能直接访问控制台的输入缓冲区。高级输入函数过滤和处理输入缓冲区中的数据，仅返回输入字符流。低级输入功能使应用程序可以直接从控制台的输入缓冲区读取输入记录，或将输入记录放入输入缓冲区。要打开控制台输入缓冲区的句柄，请在对CreateFile函数的调用中指定CONIN $值。

输入记录是一种结构，其中包含有关发生的事件类型（键盘，鼠标，窗口大小调整，焦点或菜单事件）的信息，以及有关该事件的特定详细信息。 INPUT\_RECORD结构中的EventType成员指示记录中包含哪种类型的事件。

焦点和菜单事件放置在控制台的输入缓冲区中，供系统内部使用，应由应用程序忽略。

键盘事件

当按下或释放任何键时，会生成键盘事件。这包括控制键。但是，ALT键在不与其他字符组合的情况下按下并释放时，对系统具有特殊意义，并且不会传递给应用程序。另外，如果输入句柄处于处理模式，则不会传递CTRL + C组合键。

如果输入事件是按键，则INPUT\_RECORD中的Event成员是KEY\_EVENT\_RECORD结构，其中包含以下信息：

一个布尔值，指示是按下键还是释放键。

按住某个键时的重复计数可以大于一个。

虚拟密钥代码，以与设备无关的方式标识给定密钥。

虚拟扫描代码，指示由键盘硬件生成的与设备有关的值。

转换后的Unicode™或ANSI字符。

一个标志变量，指示控制键（ALT，CTRL，SHIFT，NUM LOCK，SCROLL LOCK和CAPS LOCK键）的状态，并指示是否按下了增强键。 IBM®101键和102键键盘的增强键是INS，DEL，HOME，END，PAGE UP，PAGE DOWN和数字小键盘左侧簇中的箭头键，以及除号（/）和数字键盘中的ENTER键。

鼠标事件

每当用户移动鼠标或按下或释放鼠标按钮之一时，都会生成鼠标事件。仅当满足以下条件时，鼠标事件才会放置在输入缓冲区中：

控制台输入模式设置为ENABLE\_MOUSE\_INPUT（默认模式）。

控制台窗口具有键盘焦点。

鼠标指针在控制台窗口的边界内。

如果输入事件是鼠标事件，则INPUT\_RECORD中的Event成员是MOUSE\_EVENT\_RECORD结构，其中包含以下信息：

鼠标指针在控制台屏幕缓冲区的坐标系中根据字符单元格的行和列的坐标。

一个标志变量，指示鼠标按钮的状态。

一个标志变量，指示控制键的状态（ALT，CTRL，SHIFT，NUM LOCK，SCROLL LOCK和CAPS LOCK），并指示是否按下了增强键。 IBM 101键和102键键盘的增强键是INS，DEL，HOME，END，PAGE UP，PAGE DOWN和数字小键盘左侧以及除（/）和ENTER键的群集中的箭头键数字键盘中的键。

一个标志变量，指示该事件是正常的按钮按下事件还是按钮释放事件，鼠标移动事件还是双击事件的第二次单击。

注意鼠标位置的坐标取决于控制台屏幕缓冲区，而不是控制台窗口。屏幕缓冲区可能已相对于窗口滚动，因此窗口的左上角不一定是控制台屏幕缓冲区的（0,0）坐标。要确定鼠标相对于窗口坐标系的坐标，请从鼠标位置坐标中减去窗口原点坐标。使用GetConsoleScreenBufferInfo函数确定窗口原点坐标。

MOUSE\_EVENT\_RECORD结构的dwButtonState成员具有与每个鼠标按钮相对应的位。如果按钮按下，则位为1；如果按钮按下，则位为0。 MOUSE\_EVENT\_RECORD的dwEventFlags成员的值为0且按钮的位从1变为0时检测到按钮释放事件。GetNumberOfConsoleMouseButtons函数检索鼠标上的按钮数。

调整缓冲区大小的事件

控制台窗口的菜单使用户可以更改活动屏幕缓冲区的大小。 此更改将生成缓冲区调整大小事件。 如果控制台的输入模式设置为ENABLE\_WINDOW\_INPUT（即默认模式为禁用），则将缓冲区大小调整事件放置在输入缓冲区中。

如果输入事件是缓冲区调整大小的事件，则INPUT\_RECORD的事件成员是WINDOW\_BUFFER\_SIZE\_RECORD结构，其中包含控制台屏幕缓冲区的新大小，以字符单元格的列和行表示。

如果用户减小了控制台屏幕缓冲区的大小，则该缓冲区的废弃部分中的所有数据都会丢失。

由于应用程序调用SetConsoleScreenBufferSize函数而导致控制台屏幕缓冲区大小的更改不会作为缓冲区调整大小事件生成。

Console Screen Buffers

A screen buffer is a two-dimensional array of character and color data for output in a console window. A console can have multiple screen buffers. The active screen buffer is the one that is displayed on the screen.

The system creates a screen buffer whenever it creates a new console. To open a handle to a console's active screen buffer, specify the CONOUT$ value in a call to the CreateFile function. A process can use the CreateConsoleScreenBuffer function to create additional screen buffers for its console. A new screen buffer is not active until its handle is specified in a call to the SetConsoleActiveScreenBuffer function. However, screen buffers can be accessed for reading and writing whether they are active or inactive.

Each screen buffer has its own two-dimensional array of character information records. The data for each character is stored in a CHAR\_INFO structure that specifies the Unicode or ANSI character and the foreground and background colors in which that character is displayed.

A number of properties associated with a screen buffer can be set independently for each screen buffer. This means that changing the active screen buffer can have a dramatic effect on the appearance of the console window. The properties associated with a screen buffer include:

Screen buffer size, in character rows and columns.

Text attributes (foreground and background colors for displaying text to be written by the WriteFile or WriteConsole function).

Window size and location (the rectangular region of the console screen buffer that is displayed in the console window).

Cursor position, appearance, and visibility.

Output modes (ENABLE\_PROCESSED\_OUTPUT and ENABLE\_WRAP\_AT\_EOL\_OUTPUT). For more information about console output modes, see High-Level Console Modes.

When a screen buffer is created, it contains blanks. Its cursor is visible and positioned at the buffer's origin (0,0), and the window is positioned with its upper left corner at the buffer's origin. The size of the console screen buffer, the window size, the text attributes, and the appearance of the cursor are determined by the user or by the system defaults. To retrieve the current values of the various properties associated with the console screen buffer, use the GetConsoleScreenBufferInfo, GetConsoleCursorInfo, and GetConsoleMode functions.

Applications that change any of the console screen buffer properties should either create their own screen buffer or save the state of the inherited screen buffer during startup and restore it at exit.

Cursor Appearance and Position

A screen buffer's cursor can be visible or hidden. When it is visible, its appearance can vary, ranging from completely filling a character cell to appearing as a horizontal line at the bottom of the cell. To retrieve information about the appearance and visibility of the cursor, use the GetConsoleCursorInfo function. This function reports whether the cursor is visible and describes the appearance of the cursor as the percentage of a character cell that it fills. To set the appearance and visibility of the cursor, use the SetConsoleCursorInfo function.

Characters written by the high-level console I/O functions are written at the current cursor location, advancing the cursor to the next location. To determine the current cursor position in the coordinate system of a screen buffer, use GetConsoleScreenBufferInfo. You can use SetConsoleCursorPosition to set the cursor position and, thereby, control the placement of text that is written or echoed by the high-level I/O functions. If you move the cursor, text at the new cursor location is overwritten.

The position, appearance, and visibility of the cursor are set independently for each screen buffer.

Character Attributes

Character attributes can be divided into two classes: color and DBCS. The following attributes are defined in the Wincon.h header file.

Attribute Meaning

FOREGROUND\_BLUE Text color contains blue.

FOREGROUND\_GREEN Text color contains green.

FOREGROUND\_RED Text color contains red.

FOREGROUND\_INTENSITY Text color is intensified.

BACKGROUND\_BLUE Background color contains blue.

BACKGROUND\_GREEN Background color contains green.

BACKGROUND\_RED Background color contains red.

BACKGROUND\_INTENSITY Background color is intensified.

COMMON\_LVB\_LEADING\_BYTE Leading byte.

COMMON\_LVB\_TRAILING\_BYTE Trailing byte.

COMMON\_LVB\_GRID\_HORIZONTAL Top horizontal.

COMMON\_LVB\_GRID\_LVERTICAL Left vertical.

COMMON\_LVB\_GRID\_RVERTICAL Right vertical.

COMMON\_LVB\_REVERSE\_VIDEO Reverse foreground and background attributes.

COMMON\_LVB\_UNDERSCORE Underscore.

The foreground attributes specify the text color. The background attributes specify the color used to fill the cell's background. The other attributes are used with DBCS.

An application can combine the foreground and background constants to achieve different colors. For example, the following combination results in bright cyan text on a blue background.

FOREGROUND\_BLUE | FOREGROUND\_GREEN | FOREGROUND\_INTENSITY | BACKGROUND\_BLUE

If no background constant is specified, the background is black, and if no foreground constant is specified, the text is black. For example, the following combination produces black text on a white background.

BACKGROUND\_BLUE | BACKGROUND\_GREEN | BACKGROUND\_RED

Each screen buffer character cell stores the color attributes for the colors used in drawing the foreground (text) and background of that cell. An application can set the color data for each character cell individually, storing the data in the Attributes member of the CHAR\_INFO structure for each cell. The current text attributes of each screen buffer are used for characters subsequently written or echoed by the high-level functions.

An application can use GetConsoleScreenBufferInfo to determine the current text attributes of a screen buffer and the SetConsoleTextAttribute function to set the character attributes. Changing a screen buffer's attributes does not affect the display of characters previously written. These text attributes do not affect characters written by the low-level console I/O functions (such as the WriteConsoleOutput or WriteConsoleOutputCharacter function), which either explicitly specify the attributes for each cell that is written or leave the attributes unchanged.

Font Attributes

The GetCurrentConsoleFont function retrieves information about the current console font. The information stored in the CONSOLE\_FONT\_INFO structure includes the width and height of each character in the font.

The GetConsoleFontSize function retrieves the size of the font used by the specified console screen buffer.

## 控制台屏幕缓冲区

屏幕缓冲区是字符和颜色数据的二维数组，用于在控制台窗口中输出。一个控制台可以具有多个屏幕缓冲区。活动屏幕缓冲区是屏幕上显示的缓冲区。

每当创建新控制台时，系统都会创建一个屏幕缓冲区。要打开控制台活动屏幕缓冲区的句柄，请在对CreateFile函数的调用中指定CONOUT $值。进程可以使用CreateConsoleScreenBuffer函数为其控制台创建其他屏幕缓冲区。直到在对SetConsoleActiveScreenBuffer函数的调用中指定了它的句柄之后，新的屏幕缓冲区才处于活动状态。但是，可以访问屏幕缓冲区以读取和写入它们是否处于活动状态。

每个屏幕缓冲区都有自己的字符信息记录的二维数组。每个字符的数据都存储在CHAR\_INFO结构中，该结构指定Unicode或ANSI字符以及显示该字符的前景色和背景色。

可以为每个屏幕缓冲区独立设置与屏幕缓冲区关联的许多属性。这意味着更改活动屏幕缓冲区可能会对控制台窗口的外观产生重大影响。与屏幕缓冲区关联的属性包括：

屏幕缓冲区大小，以字符行和列表示。

文本属性（用于显示要由WriteFile或WriteConsole函数编写的文本的前景色和背景色）。

窗口的大小和位置（控制台窗口中显示的控制台屏幕缓冲区的矩形区域）。

光标位置，外观和可见性。

输出模式（ENABLE\_PROCESSED\_OUTPUT和ENABLE\_WRAP\_AT\_EOL\_OUTPUT）。有关控制台输出模式的更多信息，请参见高级控制台模式。

创建屏幕缓冲区时，它包含空白。它的光标可见，并位于缓冲区的原点（0,0），并且窗口的左上角位于缓冲区的原点。控制台屏幕缓冲区的大小，窗口大小，文本属性和光标的外观由用户或系统默认值确定。若要检索与控制台屏幕缓冲区关联的各种属性的当前值，请使用GetConsoleScreenBufferInfo，GetConsoleCursorInfo和GetConsoleMode函数。

更改任何控制台屏幕缓冲区属性的应用程序应该创建自己的屏幕缓冲区，或者在启动期间保存继承的屏幕缓冲区的状态，然后在退出时将其还原。

光标外观和位置

屏幕缓冲区的光标可以显示或隐藏。当它可见时，其外观可以变化，从完全填充字符单元格到在单元格底部以水平线显示。若要检索有关光标的外观和可见性的信息，请使用GetConsoleCursorInfo函数。此函数报告光标是否可见，并将光标的外观描述为它填充的字符单元格的百分比。若要设置光标的外观和可见性，请使用SetConsoleCursorInfo函数。

由高级控制台I / O功能写入的字符被写入当前光标位置，从而将光标前进到下一个位置。若要确定当前光标在屏幕缓冲区坐标系中的位置，请使用GetConsoleScreenBufferInfo。您可以使用SetConsoleCursorPosition设置光标位置，从而控制由高级I / O函数写入或回显的文本的位置。如果移动光标，新光标位置的文本将被覆盖。

光标的位置，外观和可见性是针对每个屏幕缓冲区独立设置的。

角色属性

字符属性可以分为两类：颜色和DBCS。 Wincon.h头文件中定义了以下属性。

属性含义

FOREGROUND\_BLUE文本颜色包含蓝色。

FOREGROUND\_GREEN文本颜色包含绿色。

FOREGROUND\_RED文本颜色包含红色。

FOREGROUND\_INTENSITY文本颜色增强。

Background\_BLUE背景色包含蓝色。

Background\_GREEN背景色包含绿色。

Background\_RED背景色包含红色。

Background\_INTENSITY背景色被增强。

COMMON\_LVB\_LEADING\_BYTE前导字节。

COMMON\_LVB\_TRAILING\_BYTE尾随字节。

COMMON\_LVB\_GRID\_HORIZONTAL顶部水平。

COMMON\_LVB\_GRID\_LVERTICAL左垂直。

COMMON\_LVB\_GRID\_RVERTICAL右垂直。

COMMON\_LVB\_REVERSE\_VIDEO反转前景和背景属性。

COMMON\_LVB\_UNDERSCORE下划线。

前景色属性指定文本颜色。 background属性指定用于填充单元格背景的颜色。其他属性与DBCS一起使用。

应用程序可以组合前景常数和背景常数以实现不同的颜色。例如，以下组合会在蓝色背景上产生明亮的青色文本。

FOREGROUND\_BLUE | FOREGROUND\_GREEN | FOREGROUND\_INTENSITY |背景\_蓝色

如果未指定背景常量，则背景为黑色，如果未指定前景常量，则文本为黑色。例如，以下组合在白色背景上产生黑色文本。

背景|蓝色| Background\_GREEN |背景\_红色

每个屏幕缓冲区字符单元格都存储用于绘制该单元格的前景色（文本）和背景的颜色的颜色属性。应用程序可以分别为每个字符单元设置颜色数据，然后将数据存储在每个单元的CHAR\_INFO结构的Attributes成员中。每个屏幕缓冲区的当前文本属性用于随后由高级功能写入或回显的字符。

应用程序可以使用GetConsoleScreenBufferInfo来确定屏幕缓冲区的当前文本属性，并可以使用SetConsoleTextAttribute函数来设置字符属性。更改屏幕缓冲区的属性不会影响先前写入的字符的显示。这些文本属性不会影响由低级别控制台I / O函数（例如WriteConsoleOutput或WriteConsoleOutputCharacter函数）编写的字符，后者会显式指定要写入的每个单元的属性，或者保持属性不变。

字体属性

GetCurrentConsoleFont函数检索有关当前控制台字体的信息。存储在CONSOLE\_FONT\_INFO结构中的信息包括字体中每个字符的宽度和高度。

GetConsoleFontSize函数检索指定的控制台屏幕缓冲区使用的字体的大小。

Window and Screen Buffer Size

The size of a screen buffer is expressed in terms of a coordinate grid based on character cells. The width is the number of character cells in each row, and the height is the number of rows. Associated with each screen buffer is a window that determines the size and location of the rectangular portion of the console screen buffer displayed in the console window. A screen buffer's window is defined by specifying the character-cell coordinates of the upper left and lower right cells of the window's rectangle.

A screen buffer can be any size, limited only by available memory. The dimensions of a screen buffer's window cannot exceed the corresponding dimensions of either the console screen buffer or the maximum window that can fit on the screen based on the current font size (controlled exclusively by the user).

The GetConsoleScreenBufferInfo function returns the following information about a screen buffer and its window:

The current size of the console screen buffer

The current location of the window

The maximum size of the window given the current screen buffer size, the current font size, and the screen size

The GetLargestConsoleWindowSize function returns the maximum size of a console's window based on the current font and screen sizes. This size differs from the maximum window size returned by GetConsoleScreenBufferInfo in that the console screen buffer size is ignored.

To change a screen buffer's size, use the SetConsoleScreenBufferSize function. This function fails if either dimension of the specified size is less than the corresponding dimension of the console's window.

To change the size or location of a screen buffer's window, use the SetConsoleWindowInfo function. This function fails if the specified window-corner coordinates exceed the limits of the console screen buffer or the screen. Changing the window size of the active screen buffer changes the size of the console window displayed on the screen.

A process can change its console's input mode to enable window input so that the process is able to receive input when the user changes the console screen buffer size. If an application enables window input, it can use GetConsoleScreenBufferInfo to retrieve window and screen buffer size at startup. This information can then be used to determine the way data is displayed in the window. If the user changes the console screen buffer size, the application can respond by changing the way data is displayed. For example, an application can adjust the way text wraps at the end of the line if the number of characters per row changes. If an application does not enable window input, it must either use the inherited window and screen buffer sizes, or set them to the desired size during startup and restore the inherited sizes at exit. For additional information about window input mode, see Low-Level Console Modes.

## 窗口和屏幕缓冲区大小

屏幕缓冲区的大小以基于字符单元的坐标网格表示。宽度是每行字符单元的数量，高度是行的数量。与每个屏幕缓冲区相关联的是一个窗口，该窗口确定控制台窗口中显示的控制台屏幕缓冲区的矩形部分的大小和位置。通过指定窗口矩形的左上和右下单元格的字符单元格坐标来定义屏幕缓冲区的窗口。

屏幕缓冲区可以是任何大小，仅受可用内存限制。屏幕缓冲区的窗口尺寸不能超过控制台屏幕缓冲区的相应尺寸，也不能超过基于当前字体大小（由用户完全控制）可以在屏幕上容纳的最大窗口的相应尺寸。

GetConsoleScreenBufferInfo函数返回有关屏幕缓冲区及其窗口的以下信息：

控制台屏幕缓冲区的当前大小

窗口的当前位置

给定当前屏幕缓冲区大小，当前字体大小和屏幕大小的窗口的最大大小

GetLargestConsoleWindowSize函数根据当前字体和屏幕尺寸返回控制台窗口的最大尺寸。此大小不同于GetConsoleScreenBufferInfo返回的最大窗口大小，因为控制台屏幕缓冲区大小被忽略。

若要更改屏幕缓冲区的大小，请使用SetConsoleScreenBufferSize函数。如果指定大小的任一尺寸小于控制台窗口的相应尺寸，则此功能将失败。

若要更改屏幕缓冲区窗口的大小或位置，请使用SetConsoleWindowInfo函数。如果指定的窗口角坐标超过了控制台屏幕缓冲区或屏幕的限制，则此功能将失败。更改活动屏幕缓冲区的窗口大小会更改屏幕上显示的控制台窗口的大小。

进程可以更改其控制台的输入模式以启用窗口输入，以便当用户更改控制台屏幕缓冲区大小时，该进程能够接收输入。如果应用程序启用了窗口输入，则可以在启动时使用GetConsoleScreenBufferInfo检索窗口和屏幕缓冲区的大小。然后，可以使用此信息来确定数据在窗口中的显示方式。如果用户更改了控制台屏幕缓冲区的大小，则应用程序可以通过更改数据的显示方式进行响应。例如，如果每行的字符数发生变化，则应用程序可以调整文本在行尾的换行方式。如果应用程序不启用窗口输入，则它必须使用继承的窗口和屏幕缓冲区大小，或者在启动过程中将它们设置为所需的大小，并在退出时恢复继承的大小。有关窗口输入模式的其他信息，请参见低级控制台模式。

Console Selection

An accessibility application needs information about the user's selection in the console. To retrieve the current console selection, call the GetConsoleSelectionInfo function. The CONSOLE\_SELECTION\_INFO structure contains information about the selection, such as the anchor, coordinates, and status.

## 控制台选择

可访问性应用程序需要有关控制台中用户选择的信息。 要检索当前的控制台选择，请调用GetConsoleSelectionInfo函数。 CONSOLE\_SELECTION\_INFO结构包含有关选择的信息，例如锚点，坐标和状态。

Pseudoconsoles

A pseudoconsole is a device type that allows applications to become the host for character-mode applications.

This is in contrast to a typical console session where the operating system will create a hosting window on behalf of the character-mode application to handle graphical output and user input.

With a pseudoconsole, the hosting window is not created. The application that makes the pseudoconsole must become responsible for displaying the graphical output and collecting user input. Alternatively, the information can be relayed further to another application responsible for these activities at a later point in the chain.

This functionality is designed for third-party "terminal window" applications to exist on the platform or for redirection of character-mode activities to a remote "terminal window" session on another machine or even on another platform.

Note that the underlying console session will still be created on behalf of the application requesting the pseudoconsole. All the rules of console sessions still apply including the ability for multiple client character-mode applications to connect to the session.

To provide maximum compatibility with the existing world of pseudoterminal functionality, the information provided over the pseudoconsole channel will always be encoded in UTF-8. This does not affect the codepage or encoding of the client applications that are attached. Translation will happen inside the pseudoconsole system as necessary.

An example for getting started can be found at Creating a Pseudoconsole Session.

Some additional background information on pseudoconsoles can be found at the announcement blog post: Windows Command-Line: Introducing the Windows Pseudo Console (ConPTY).

## 伪控制台

伪控制台是一种设备类型，它允许应用程序成为字符模式应用程序的主机。

这与典型的控制台会话相反，在典型的控制台会话中，操作系统将代表字符模式应用程序创建一个托管窗口以处理图形输出和用户输入。

使用伪控制台，不会创建托管窗口。制作伪控制台的应用程序必须负责显示图形输出并收集用户输入。可替代地，信息可以在链的稍后点进一步中继到负责这些活动的另一个应用程序。

此功能旨在用于平台上存在的第三方“终端窗口”应用程序，或用于将字符模式活动重定向到另一台机器甚至另一台平台上的远程“终端窗口”会话。

请注意，仍将代表请求伪控制台的应用程序创建基础控制台会话。控制台会话的所有规则仍然适用，包括多个客户端字符模式应用程序可以连接到会话的功能。

为了与现有的伪终端功能世界提供最大的兼容性，伪控制台通道上提供的信息将始终以UTF-8编码。这不会影响附加的客户端应用程序的代码页或编码。翻译将根据需要在伪控制台系统内部进行。

在创建伪控制台会话中可以找到一个入门示例。

有关伪控制台的其他一些背景信息，可以在公告博客中找到：Windows命令行：Windows Pseudo Console（ConPTY）简介。

Console Reference

The following sections describe the Console API:

Console Functions

Console Structures

Console WinEvents

# 控制台开发人员指南和API参考

以下各节介绍了控制台API：

控制台函数

控制台结构

控制台WinEvents

Using the Console

The following examples demonstrate how to use the console functions:

Using the high-level input and output functions

Reading and writing blocks of characters and attributes

Reading input buffer events

Clearing the screen

Scrolling a screen buffer's window

Scrolling a screen buffer's contents

Registering a control handler function

## 使用控制台

下面的示例演示如何使用控制台功能：

使用高级输入和输出功能

读写字符和属性块

读取输入缓冲区事件

清除画面

滚动屏幕缓冲区的窗口

滚动屏幕缓冲区的内容

注册控制处理程序功能

High-Level Console Input and Output Functions

The ReadFile and WriteFile functions, or the ReadConsole and WriteConsole functions, enable an application to read console input and write console output as a stream of characters. ReadConsole and WriteConsole behave exactly like ReadFile and WriteFile except that they can be used either as wide-character functions (in which text arguments must use Unicode) or as ANSI functions (in which text arguments must use characters from the Windows character set). Applications that need to maintain a single set of sources to support either Unicode or the ANSI character set should use ReadConsole and WriteConsole.

ReadConsole and WriteConsole can only be used with console handles; ReadFile and WriteFile can be used with other handles (such as files or pipes). ReadConsole and WriteConsole fail if used with a standard handle that has been redirected and is no longer a console handle.

To get keyboard input, a process can use ReadFile or ReadConsole with a handle to the console's input buffer, or it can use ReadFile to read input from a file or a pipe if STDIN has been redirected. These functions only return keyboard events that can be translated into ANSI characters (or Unicode characters in the case of ReadConsole). The input that can be returned includes control key combinations. The functions do not return keyboard events involving the function keys or arrow keys. Input events generated by mouse, window, focus, or menu input are discarded.

If line input mode is enabled (the default mode), ReadFile and ReadConsole do not return to the calling application until the ENTER key is pressed. If line input mode is disabled, the functions do not return until at least one character is available. In either mode, all available characters are read until either no more keys are available or the specified number of characters has been read. Unread characters are buffered until the next read operation. The functions report the total number of characters actually read. If echo input mode is enabled, characters read by these functions are written to the active screen buffer at the current cursor position.

A process can use WriteFile or WriteConsole to write to either an active or inactive screen buffer, or it can use WriteFile to write to a file or a pipe if STDOUT has been redirected. Processed output mode and wrap at EOL output mode control the way characters are written or echoed to a screen buffer.

Characters written by WriteFile or WriteConsole, or echoed by ReadFile or ReadConsole, are inserted in a screen buffer at the current cursor position. As each character is written, the cursor position advances to the next character cell; however, the behavior at the end of a row depends on the console screen buffer's wrap at EOL output mode. An application can use the GetConsoleScreenBufferInfo function to determine the current cursor position and the SetConsoleCursorPosition function to set the cursor position.

For an example that uses the high-level console I/O functions, see Using the High-Level Input and Output Functions.

### 高级控制台输入和输出功能

ReadFile和WriteFile函数或ReadConsole和WriteConsole函数使应用程序能够读取控制台输入并将控制台输出作为字符流写入。 ReadConsole和WriteConsole的行为与ReadFile和WriteFile完全相同，不同之处在于它们既可以用作宽字符函数（其中文本参数必须使用Unicode），也可以用作ANSI函数（其中文本参数必须使用Windows字符集中的字符）。需要维护一组源以支持Unicode或ANSI字符集的应用程序应使用ReadConsole和WriteConsole。

ReadConsole和WriteConsole只能与控制台句柄一起使用。 ReadFile和WriteFile可以与其他句柄（例如文件或管道）一起使用。如果将ReadConsole和WriteConsole与已重定向的标准句柄一起使用，并且不再是控制台句柄，则该操作将失败。

要获取键盘输入，进程可以将ReadFile或ReadConsole与控制台输入缓冲区的句柄一起使用，如果STDIN已重定向，则可以使用ReadFile从文件或管道中读取输入。这些函数仅返回可以转换为ANSI字符（对于ReadConsole则为Unicode字符）的键盘事件。可以返回的输入包括控制键组合。这些功能不返回涉及功能键或箭头键的键盘事件。由鼠标，窗口，焦点或菜单输入生成的输入事件将被丢弃。

如果启用了线路输入模式（默认模式），则在按ENTER键之前，ReadFile和ReadConsole不会返回到调用应用程序。如果禁用了行输入模式，则只有在至少一个字符可用之前，这些功能才会返回。在任何一种模式下，都将读取所有可用字符，直到不再有可用的键或已读取指定数量的字符为止。未读的字符被缓冲，直到下一次读操作为止。这些功能报告实际读取的字符总数。如果启用了回声输入模式，则将通过这些功能读取的字符写入当前光标位置的活动屏幕缓冲区。

进程可以使用WriteFile或WriteConsole写入活动或不活动的屏幕缓冲区，或者如果STDOUT已重定向，则可以使用WriteFile写入文件或管道。处理后的输出模式和EOL输出模式的换行控制字符写入或回显到屏幕缓冲区的方式。

由WriteFile或WriteConsole编写的字符，或由ReadFile或ReadConsole回显的字符，被插入到当前光标位置的屏幕缓冲区中。写入每个字符时，光标位置前进到下一个字符单元；但是，行尾的行为取决于控制台屏幕缓冲区在EOL输出模式下的换行。应用程序可以使用GetConsoleScreenBufferInfo函数来确定当前光标位置，并可以使用SetConsoleCursorPosition函数来设置光标位置。

有关使用高级控制台I / O功能的示例，请参见使用高级输入和输出功能。

Using the High-Level Input and Output Functions

The following example uses the high-level console I/O functions for console I/O. For more information about the high-level console I/O functions, see High-Level Console I/O.

The example assumes that the default I/O modes are in effect initially for the first calls to the ReadFile and WriteFile functions. Then the input mode is changed to turn offline input mode and echo input mode for the second calls to ReadFile and WriteFile. The SetConsoleTextAttribute function is used to set the colors in which subsequently written text will be displayed. Before exiting, the program restores the original console input mode and color attributes.

The example's NewLine function is used when line input mode is disabled. It handles carriage returns by moving the cursor position to the first cell of the next row. If the cursor is already in the last row of the console screen buffer, the contents of the console screen buffer are scrolled up one line.

### 使用高级输入和输出功能

以下示例将高级控制台I / O功能用于控制台I / O。有关高级控制台I / O功能的更多信息，请参阅高级控制台I / O。

该示例假定默认的I / O模式最初在首次调用ReadFile和WriteFile函数时生效。然后，将输入模式更改为脱机输入模式和第二次调用ReadFile和WriteFile的echo输入模式。 SetConsoleTextAttribute函数用于设置颜色，随后的书写文本将以这些颜色显示。在退出之前，该程序将还原原始的控制台输入模式和颜色属性。

禁用线路输入模式时，将使用示例的NewLine函数。它通过将光标位置移动到下一行的第一个单元格来处理回车。如果光标已经在控制台屏幕缓冲区的最后一行中，那么控制台屏幕缓冲区的内容将向上滚动一行。

#include <windows.h>

void NewLine(void);

void ScrollScreenBuffer(HANDLE, INT);

HANDLE hStdout, hStdin;

CONSOLE\_SCREEN\_BUFFER\_INFO csbiInfo;

int main(void)

{

LPSTR lpszPrompt1 = "Type a line and press Enter, or q to quit: ";

LPSTR lpszPrompt2 = "Type any key, or q to quit: ";

CHAR chBuffer[256];

DWORD cRead, cWritten, fdwMode, fdwOldMode;

WORD wOldColorAttrs;

// Get handles to STDIN and STDOUT.

hStdin = GetStdHandle(STD\_INPUT\_HANDLE);

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (hStdin == INVALID\_HANDLE\_VALUE ||

hStdout == INVALID\_HANDLE\_VALUE)

{

MessageBox(NULL, TEXT("GetStdHandle"), TEXT("Console Error"),

MB\_OK);

return 1;

}

// Save the current text colors.

if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))

{

MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),

TEXT("Console Error"), MB\_OK);

return 1;

}

wOldColorAttrs = csbiInfo.wAttributes;

// Set the text attributes to draw red text on black background.

if (! SetConsoleTextAttribute(hStdout, FOREGROUND\_RED |

FOREGROUND\_INTENSITY))

{

MessageBox(NULL, TEXT("SetConsoleTextAttribute"),

TEXT("Console Error"), MB\_OK);

return 1;

}

// Write to STDOUT and read from STDIN by using the default

// modes. Input is echoed automatically, and ReadFile

// does not return until a carriage return is typed.

//

// The default input modes are line, processed, and echo.

// The default output modes are processed and wrap at EOL.

while (1)

{

if (! WriteFile(

hStdout, // output handle

lpszPrompt1, // prompt string

lstrlenA(lpszPrompt1), // string length

&cWritten, // bytes written

NULL) ) // not overlapped

{

MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),

MB\_OK);

return 1;

}

if (! ReadFile(

hStdin, // input handle

chBuffer, // buffer to read into

255, // size of buffer

&cRead, // actual bytes read

NULL) ) // not overlapped

break;

if (chBuffer[0] == 'q') break;

}

// Turn off the line input and echo input modes

if (! GetConsoleMode(hStdin, &fdwOldMode))

{

MessageBox(NULL, TEXT("GetConsoleMode"), TEXT("Console Error"),

MB\_OK);

return 1;

}

fdwMode = fdwOldMode &

~(ENABLE\_LINE\_INPUT | ENABLE\_ECHO\_INPUT);

if (! SetConsoleMode(hStdin, fdwMode))

{

MessageBox(NULL, TEXT("SetConsoleMode"), TEXT("Console Error"),

MB\_OK);

return 1;

}

// ReadFile returns when any input is available.

// WriteFile is used to echo input.

NewLine();

while (1)

{

if (! WriteFile(

hStdout, // output handle

lpszPrompt2, // prompt string

lstrlenA(lpszPrompt2), // string length

&cWritten, // bytes written

NULL) ) // not overlapped

{

MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),

MB\_OK);

return 1;

}

if (! ReadFile(hStdin, chBuffer, 1, &cRead, NULL))

break;

if (chBuffer[0] == '\r')

NewLine();

else if (! WriteFile(hStdout, chBuffer, cRead,

&cWritten, NULL)) break;

else

NewLine();

if (chBuffer[0] == 'q') break;

}

// Restore the original console mode.

SetConsoleMode(hStdin, fdwOldMode);

// Restore the original text colors.

SetConsoleTextAttribute(hStdout, wOldColorAttrs);

return 0;

}

// The NewLine function handles carriage returns when the processed

// input mode is disabled. It gets the current cursor position

// and resets it to the first cell of the next row.

void NewLine(void)

{

if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))

{

MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),

TEXT("Console Error"), MB\_OK);

return;

}

csbiInfo.dwCursorPosition.X = 0;

// If it is the last line in the screen buffer, scroll

// the buffer up.

if ((csbiInfo.dwSize.Y-1) == csbiInfo.dwCursorPosition.Y)

{

ScrollScreenBuffer(hStdout, 1);

}

// Otherwise, advance the cursor to the next line.

else csbiInfo.dwCursorPosition.Y += 1;

if (! SetConsoleCursorPosition(hStdout,

csbiInfo.dwCursorPosition))

{

MessageBox(NULL, TEXT("SetConsoleCursorPosition"),

TEXT("Console Error"), MB\_OK);

return;

}

}

void ScrollScreenBuffer(HANDLE h, INT x)

{

SMALL\_RECT srctScrollRect, srctClipRect;

CHAR\_INFO chiFill;

COORD coordDest;

srctScrollRect.Left = 0;

srctScrollRect.Top = 1;

srctScrollRect.Right = csbiInfo.dwSize.X - (SHORT)x;

srctScrollRect.Bottom = csbiInfo.dwSize.Y - (SHORT)x;

// The destination for the scroll rectangle is one row up.

coordDest.X = 0;

coordDest.Y = 0;

// The clipping rectangle is the same as the scrolling rectangle.

// The destination row is left unchanged.

srctClipRect = srctScrollRect;

// Set the fill character and attributes.

chiFill.Attributes = FOREGROUND\_RED|FOREGROUND\_INTENSITY;

chiFill.Char.AsciiChar = (char)' ';

// Scroll up one line.

ScrollConsoleScreenBuffer(

h, // screen buffer handle

&srctScrollRect, // scrolling rectangle

&srctClipRect, // clipping rectangle

coordDest, // top left destination cell

&chiFill); // fill character and color

}

High-Level Console Modes

The behavior of the high-level console functions is affected by the console input and output modes. All of the following console input modes are enabled for a console's input buffer when a console is created:

Line input mode

Processed input mode

Echo input mode

Both of the following console output modes are enabled for a console screen buffer when it is created:

Processed output mode

Wrapping at EOL output mode

All three input modes, along with processed output mode, are designed to work together. It is best to either enable or disable all of these modes as a group. When all are enabled, the application is said to be in "cooked" mode, which means that most of the processing is handled for the application. When all are disabled, the application is in "raw" mode, which means that input is unfiltered and any processing is left to the application.

An application can use the GetConsoleMode function to determine the current mode of a console's input buffer or screen buffer. You can enable or disable any of these modes by using the following values in the SetConsoleMode function. Note that setting the output mode of one screen buffer does not affect the output mode of other screen buffers.

Mode Description

ENABLE\_PROCESSED\_INPUT Used with a console input handle to cause the system to process any system editing or control key input rather than returning it as input in the read operation's buffer. If line input is also enabled, backspaces and carriage returns are handled correctly. A backspace causes the cursor to move back one space without affecting the character at the cursor position. A carriage return is converted to carriage return – line feed character combination. If echo input mode is enabled and the output should reflect system editing, processed output must be enabled for the active screen buffer. If processed input is enabled, the CTRL+C key combination is passed on to the appropriate handler regardless of whether line input is enabled. For more information about control handlers, see Console Control Handlers.

ENABLE\_LINE\_INPUT Used with a console input handle to cause the ReadFile and ReadConsole functions to return when the ENTER key is pressed. If line input mode is disabled, the functions return when one or more characters are available in the input buffer.

ENABLE\_ECHO\_INPUT Used with a console input handle to cause keyboard input read by the ReadFile or ReadConsole function to be echoed to the active screen buffer. Characters are echoed only if the process that calls ReadFile or ReadConsole has an open handle to the active screen buffer. Echo mode cannot be enabled unless line input is also enabled. The output mode of the active screen buffer affects the way echoed input is displayed.

ENABLE\_PROCESSED\_OUTPUT Used with a console screen buffer handle to cause the system to perform the appropriate action for ANSI control characters that are written to a screen buffer. The backspace, tab, bell, carriage return, and line feed characters are processed. A tab character moves the cursor to the next tab stop, which occurs every eight characters. A bell character sounds a short tone.

ENABLE\_WRAP\_AT\_EOL\_OUTPUT Used with a console screen buffer handle to cause the current output position (cursor position) to move to the first column in the next row (line) when the end of the current row is reached. If the bottom of the window region is reached, the window origin is moved down one row. This movement has the effect of scrolling the contents of the window up one row. If the bottom of the console screen buffer is reached, the contents of the console screen buffer are scrolled up one row, and the top row of the console screen buffer is discarded.

If this mode is disabled, the last character in the row is overwritten with any subsequent characters.

### 高级控制台模式

控制台输入和输出模式会影响高级控制台功能的行为。创建控制台时，将为控制台的输入缓冲区启用以下所有控制台输入模式：

线路输入方式

处理输入模式

回声输入模式

创建控制台屏幕缓冲区时，以下两种控制台输出模式均已启用：

处理输出模式

在EOL输出模式下包装

所有这三种输入模式以及已处理的输出模式均设计为可协同工作。最好同时启用或禁用所有这些模式。启用所有功能后，该应用程序被称为处于“煮熟”模式，这意味着大部分处理是为该应用程序处理的。如果全部禁用，则应用程序处于“原始”模式，这意味着输入未过滤，并且任何处理都留给了应用程序。

应用程序可以使用GetConsoleMode函数来确定控制台的输入缓冲区或屏幕缓冲区的当前模式。您可以通过使用SetConsoleMode函数中的以下值来启用或禁用任何这些模式。请注意，设置一个屏幕缓冲区的输出模式不会影响其他屏幕缓冲区的输出模式。

模式说明

ENABLE\_PROCESSED\_INPUT与控制台输入句柄一起使用，以使系统处理任何系统编辑或控制键输入，而不是将其作为读取操作缓冲区中的输入返回。如果还启用了行输入，则正确处理了退格和回车符。退格键使光标向后移动一个空格，而不会影响光标位置的字符。回车符转换为回车符–换行符组合。如果启用了回声输入模式，并且输出应反映系统编辑，则必须为活动屏幕缓冲区启用处理后的输出。如果启用了已处理的输入，则无论是否启用了行输入，CTRL + C组合键都将传递到适当的处理程序。有关控制处理程序的更多信息，请参见控制台控制处理程序。

ENABLE\_LINE\_INPUT与控制台输入句柄一起使用，以在按ENTER键时返回ReadFile和ReadConsole函数。如果禁用了行输入模式，则当输入缓冲区中有一个或多个字符可用时，函数将返回。

ENABLE\_ECHO\_INPUT与控制台输入句柄一起使用，以使ReadFile或ReadConsole函数读取的键盘输入回显到活动屏幕缓冲区。仅当调用ReadFile或ReadConsole的进程具有活动屏幕缓冲区的打开句柄时，才会回显字符。除非还启用了线路输入，否则无法启用回声模式。活动屏幕缓冲区的输出模式会影响回显输入的显示方式。

ENABLE\_PROCESSED\_OUTPUT与控制台屏幕缓冲区句柄一起使用，以使系统对写入屏幕缓冲区的ANSI控制字符执行适当的操作。退格键，制表符，铃声，回车符和换行符将被处理。制表符将光标移动到下一个制表符停止位置，该位置每8个字符出现一次。铃声字符听起来很短。

ENABLE\_WRAP\_AT\_EOL\_OUTPUT与控制台屏幕缓冲区句柄一起使用，当到达当前行的末尾时，导致当前输出位置（光标位置）移至下一行（行）的第一列。如果到达窗口区域的底部，则将窗口原点向下移动一行。此移动具有将窗口的内容向上滚动一行的效果。如果到达控制台屏幕缓冲区的底部，则将控制台屏幕缓冲区的内容向上滚动一行，并丢弃控制台屏幕缓冲区的第一行。

如果禁用此模式，则该行中的最后一个字符将被所有后续字符覆盖。

High-Level Console I/O

The high-level I/O functions provide a simple way to read a stream of characters from console input or to write a stream of characters to console output. A high-level read operation gets input characters from a console's input buffer and stores them in a specified buffer. A high-level write operation takes characters from a specified buffer and writes them to a screen buffer at the current cursor location, advancing the cursor as each character is written.

High-level I/O gives you a choice between the ReadFile and WriteFile functions and the ReadConsole and WriteConsole functions. They are identical, except for two important differences. The console functions support the use of either Unicode characters or the ANSI character set; the file I/O functions do not support Unicode. Also, the file I/O functions can be used to access files, pipes, and serial communications devices; the console functions can only be used with console handles. This distinction is important if an application relies on standard handles that may have been redirected.

When using either set of high-level functions, an application can control the text and background colors used to display characters subsequently written to a screen buffer. An application can also use the console modes that affect high-level console I/O to enable or disable the following properties:

Echoing of keyboard input to the active screen buffer

Line input, in which a read operation does not return until the ENTER key is pressed

Automatic processing of keyboard input to handle carriage returns, CTRL+C, and other input details

Automatic processing of output to handle line wrapping, carriage returns, backspaces, and other output details

For more information, see the following topics:

Console Modes

High-Level Console Modes

High-Level Console Input and Output Functions

### 高级控制台I / O

高级I / O功能提供了一种从控制台输入读取字符流或将字符流写入控制台输出的简单方法。高级读取操作从控制台的输入缓冲区中获取输入字符，并将它们存储在指定的缓冲区中。高级写操作从指定的缓冲区获取字符，并将其写入当前光标位置的屏幕缓冲区，从而在写入每个字符时使光标前进。

高级I / O使您可以在ReadFile和WriteFile函数以及ReadConsole和WriteConsole函数之间进行选择。它们是相同的，除了两个重要的区别。控制台功能支持使用Unicode字符或ANSI字符集。文件I / O功能不支持Unicode。此外，文件I / O功能可用于访问文件，管道和串行通信设备。控制台功能只能与控制台句柄一起使用。如果应用程序依赖于可能已重定向的标准句柄，则此区别很重要。

使用这两组高级功能时，应用程序可以控制文本和背景颜色，这些颜色用于显示随后写入屏幕缓冲区的字符。应用程序还可以使用影响高级控制台I / O的控制台模式来启用或禁用以下属性：

将键盘输入回显到活动屏幕缓冲区

行输入，在此输入中，直到按ENTER键才返回读操作

自动处理键盘输入以处理回车符，CTRL + C和其他输入详细信息

自动处理输出以处理换行，回车，退格和其他输出详细信息

有关更多信息，请参见以下主题：

控制台模式

高级控制台模式

高级控制台输入和输出功能

Low-Level Console Input Functions

A low-level console input functions buffer contains input records that can include information about keyboard, mouse, buffer-resizing, focus, and menu events. The low-level functions provide direct access to the input buffer, unlike the high-level functions that filter and process the input buffer's data, discarding all but keyboard input.

There are five low-level functions for accessing a console's input buffer:

ReadConsoleInput

PeekConsoleInput

GetNumberOfConsoleInputEvents

WriteConsoleInput

FlushConsoleInputBuffer

The ReadConsoleInput, PeekConsoleInput, and WriteConsoleInput functions use the INPUT\_RECORD structure to read from or write to an input buffer.

Following are descriptions of the low-level console input functions.

Function Description

ReadConsoleInput Reads and removes input records from an input buffer. The function does not return until at least one record is available to be read. Then all available records are transferred to the buffer of the calling process until either no more records are available or the specified number of records has been read. Unread records remain in the input buffer for the next read operation. The function reports the total number of records that have been read. For an example that uses ReadConsoleInput, see Reading Input Buffer Events.

PeekConsoleInput Reads without removing the pending input records in an input buffer. All available records up to the specified number are copied into the buffer of the calling process. If no records are available, the function returns immediately. The function reports the total number of records that have been read.

GetNumberOfConsoleInputEvents Determines the number of unread input records in an input buffer.

WriteConsoleInput Places input records into the input buffer behind any pending records in the buffer. The input buffer grows dynamically, if necessary, to hold as many records as are written. To use this function, the specified input buffer handle must have the GENERIC\_READ access right.

FlushConsoleInputBuffer Discards all unread events in the input buffer. To use this function, the specified input buffer handle must have the GENERIC\_READ access right.

A thread of an application's process can perform a wait operation to wait for input to be available in an input buffer. To initiate a wait operation, specify a handle to the input buffer in a call to any of the wait functions. These functions can return when the state of one or more objects is signaled. The state of a console input handle becomes signaled when there are unread records in its input buffer. The state is reset to nonsignaled when the input buffer becomes empty. If there is no input available, the calling thread enters an efficient wait state, consuming very little processor time while waiting for the conditions of the wait operation to be satisfied.

### 底层控制台输入功能

低级控制台输入功能缓冲区包含输入记录，这些记录可以包含有关键盘，鼠标，缓冲区大小调整，焦点和菜单事件的信息。与过滤和处理输入缓冲区的数据，丢弃除键盘输入之外的所有功能的高级功能不同，这些低级功能可直接访问输入缓冲区。

有五个用于访问控制台输入缓冲区的低级功能：

ReadConsoleInput

PeekConsoleInput

GetNumberOfConsoleInputEvents

WriteConsoleInput

FlushConsoleInputBuffer

ReadConsoleInput，PeekConsoleInput和WriteConsoleInput函数使用INPUT\_RECORD结构来读取或写入输入缓冲区。

以下是对低级别控制台输入功能的描述。

功能说明

ReadConsoleInput从输入缓冲区读取和删除输入记录。直到至少有一条记录可供读取，该函数才会返回。然后，将所有可用记录转移到调用进程的缓冲区中，直到不再有可用记录或已读取指定数量的记录为止。未读的记录保留在输入缓冲区中，以进行下一个读操作。该功能报告已读取的记录总数。有关使用ReadConsoleInput的示例，请参见读取输入缓冲区事件。

PeekConsoleInput读取而不删除输入缓冲区中的未决输入记录。直到指定号码的所有可用记录都被复制到调用进程的缓冲区中。如果没有可用的记录，该函数将立即返回。该功能报告已读取的记录总数。

GetNumberOfConsoleInputEvents确定输入缓冲区中未读的输入记录数。

WriteConsoleInput将输入记录放入缓冲区中任何未决记录后面的输入缓冲区中。如果需要，输入缓冲区将动态增长，以保存与写入的一样多的记录。要使用此功能，指定的输入缓冲区句柄必须具有GENERIC\_READ访问权限。

FlushConsoleInputBuffer丢弃输入缓冲区中的所有未读事件。要使用此功能，指定的输入缓冲区句柄必须具有GENERIC\_READ访问权限。

应用程序进程的线程可以执行等待操作，以等待输入在输入缓冲区中可用。要启动等待操作，请在对任何等待函数的调用中指定输入缓冲区的句柄。当发出一个或多个对象的状态信号时，这些函数可以返回。当控制台输入句柄的输入缓冲区中有未读的记录时，将发出信号。当输入缓冲区为空时，状态重置为无信号。如果没有可用的输入，则调用线程将进入高效的等待状态，在等待满足等待操作的条件时消耗很少的处理器时间。

Low-Level Console Output Functions

The low-level console output functions provide direct access to the character cells of a screen buffer. One set of functions reads from or writes to consecutive cells beginning at any location in the console screen buffer. Another set of functions reads from or writes to rectangular blocks of cells.

The following functions read from or write to a specified number of consecutive character cells in a screen buffer, beginning with a specified cell.

Function Description

ReadConsoleOutputCharacter Copies a string of Unicode or ANSI characters from a screen buffer.

WriteConsoleOutputCharacter Writes a string of Unicode or ANSI characters to a screen buffer.

ReadConsoleOutputAttribute Copies a string of text and background color attributes from a screen buffer.

WriteConsoleOutputAttribute Writes a string of text and background color attributes to a screen buffer.

FillConsoleOutputCharacter Writes a single Unicode or ANSI character to a specified number of consecutive cells in a screen buffer.

FillConsoleOutputAttribute Writes a text and background color attribute combination to a specified number of consecutive cells in a screen buffer.

For all of these functions, when the last cell of a row is encountered, reading or writing wraps around to the first cell of the next row. When the end of the last row of the console screen buffer is encountered, the write functions discard all unwritten characters or attributes, and the read functions report the number of characters or attributes actually written.

The following functions read from or write to rectangular blocks of character cells at a specified location in a screen buffer.

Function Description

ReadConsoleOutput Copies character and color data from a specified block of screen buffer cells into a given block in a destination buffer.

WriteConsoleOutput Writes character and color data to a specified block of screen buffer cells from a given block in a source buffer.

These functions treat screen buffers and source or destination buffers as two-dimensional arrays of CHAR\_INFO structures (containing character and color attribute data for each cell). The functions specify the width and height, in character cells, of the source or destination buffer, and the pointer to the buffer is treated as a pointer to the origin cell (0,0) of the two-dimensional array. The functions use a SMALL\_RECT structure to specify which rectangle to access in the console screen buffer, and the coordinates of the upper left cell in the source or destination buffer determine the location of the corresponding rectangle in that buffer.

These functions automatically clip the specified screen buffer rectangle to fit within the boundaries of the console screen buffer. For example, if the rectangle specifies lower right coordinates that are (column 100, row 50) and the console screen buffer is only 80 columns wide, the coordinates are clipped so that they are (column 79, row 50). Similarly, this adjusted rectangle is again clipped to fit within the boundaries of the source or destination buffer. The screen buffer coordinates of the actual rectangle that was read from or written to are specified. For an example that uses these functions, see Reading and Writing Blocks of Characters and Attributes.

The illustration shows a ReadConsoleOutput operation where clipping occurs when the block is read from the console screen buffer, and again when the block is copied into the destination buffer. The function reports the actual screen buffer rectangle that it copied from.

screen buffer window with destination buffer

### 底层控制台输出功能

低级控制台输出功能提供对屏幕缓冲区的字符单元格的直接访问。一组函数从控制台屏幕缓冲区中的任何位置开始读取或写入连续的单元格。另一组功能读取或写入矩形单元格块。

以下功能从指定单元开始，从屏幕缓冲区中读取或写入指定数量的连续字符单元。

功能说明

ReadConsoleOutputCharacter从屏幕缓冲区复制Unicode或ANSI字符串。

WriteConsoleOutputCharacter将Unicode或ANSI字符的字符串写入屏幕缓冲区。

ReadConsoleOutputAttribute从屏幕缓冲区复制文本和背景颜色属性的字符串。

WriteConsoleOutputAttribute将文本和背景颜色属性字符串写入屏幕缓冲区。

FillConsoleOutputCharacter将单个Unicode或ANSI字符写入屏幕缓冲区中指定数量的连续单元格中。

FillConsoleOutputAttribute将文本和背景颜色属性组合写入屏幕缓冲区中指定数量的连续单元格。

对于所有这些功能，当遇到一行的最后一个单元格时，读取或写入将换行到下一行的第一个单元格。当遇到控制台屏幕缓冲区的最后一行的末尾时，写入函数将丢弃所有未写入的字符或属性，而读取函数将报告实际写入的字符或属性的数量。

以下功能在屏幕缓冲区中指定位置处读取或写入字符单元的矩形块。

功能说明

ReadConsoleOutput将字符和颜色数据从屏幕缓冲单元的指定块复制到目标缓冲区中的给定块。

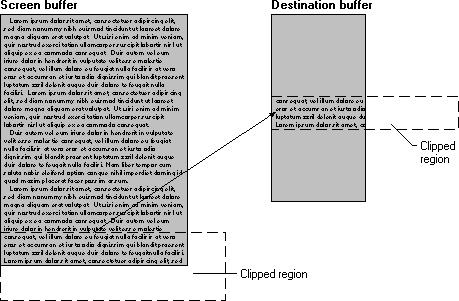
WriteConsoleOutput将字符和颜色数据从源缓冲区中的给定块写入屏幕缓冲区单元的指定块。

这些函数将屏幕缓冲区和源缓冲区或目标缓冲区视为CHAR\_INFO结构的二维数组（包含每个单元格的字符和颜色属性数据）。这些函数在字符单元格中指定源或目标缓冲区的宽度和高度，并且指向该缓冲区的指针被视为指向二维数组的原始单元格（0,0）的指针。这些函数使用SMALL\_RECT结构指定要在控制台屏幕缓冲区中访问的矩形，并且源缓冲区或目标缓冲区中左上单元格的坐标确定该缓冲区中相应矩形的位置。

这些功能会自动剪切指定的屏幕缓冲区矩形以适合控制台屏幕缓冲区的边界。例如，如果矩形指定的右下角坐标为（第100列，第50行），并且控制台屏幕缓冲区只有80列宽，则将坐标剪裁为它们（第79列，第50行）。类似地，再次调整此调整后的矩形以适合源缓冲区或目标缓冲区的边界。指定从中读取或写入的实际矩形的屏幕缓冲区坐标。有关使用这些功能的示例，请参见读取和写入字符和属性块。

该图显示了ReadConsoleOutput操作，当从控制台屏幕缓冲区中读取该块时，以及当将该块复制到目标缓冲区中时，都会发生剪辑。该函数报告从其复制的实际屏幕缓冲区矩形。

具有目标缓冲区的屏幕缓冲区窗口



Low-Level Console I/O

The low-level console I/O functions expand an application's control over console I/O by enabling direct access to a console's input and screen buffers. These functions enable an application to perform the following tasks:

Receive input about mouse and buffer-resizing events

Receive extended information about keyboard input events

Write input records to the input buffer

Read input records without removing them from the input buffer

Determine the number of pending events in the input buffer

Flush the input buffer

Read and write strings of Unicode or ANSI characters at a specified location in a screen buffer

Read and write strings of text and background color attributes at a specified screen buffer location

Read and write rectangular blocks of character and color data at a specified screen buffer location

Write a single Unicode or ANSI character, or a text and background color attribute combination, to a specified number of consecutive cells beginning at a specified screen buffer location

For more information, see the following topics:

Console Modes

Low-Level Console Modes

Low-Level Console Input Functions

Low-Level Console Output Functions

### 低层控制台I / O

通过允许直接访问控制台的输入和屏幕缓冲区，低级控制台I / O功能扩展了应用程序对控制台I / O的控制。这些功能使应用程序可以执行以下任务：

接收有关鼠标和缓冲区大小调整事件的输入

接收有关键盘输入事件的扩展信息

将输入记录写入输入缓冲区

读取输入记录而不将其从输入缓冲区中删除

确定输入缓冲区中的未决事件数

刷新输入缓冲区

在屏幕缓冲区的指定位置读取和写入Unicode或ANSI字符的字符串

在指定的屏幕缓冲区位置读写文本和背景颜色属性的字符串

在指定的屏幕缓冲区位置读取和写入矩形的字符和颜色数据块

从指定的屏幕缓冲区位置开始，向指定数量的连续单元格写入单个Unicode或ANSI字符，或文本和背景颜色属性组合

有关更多信息，请参见以下主题：

控制台模式

低层控制台模式

底层控制台输入功能

底层控制台输出功能

Low-Level Console Modes

The types of input events reported in a console's input buffer depend on the console's mouse and window input modes. The console's processed input mode determines how the system handles the CTRL+C key combination. To set or retrieve the state of a console's input modes, an application can specify a console input buffer handle in a call to the SetConsoleMode or GetConsoleMode function. The following modes are used with console input handles.

Mode Description

ENABLE\_MOUSE\_INPUT Controls whether mouse events are reported in the input buffer. By default, mouse input is enabled and window input is disabled. Changing either of these modes affects only input that occurs after the mode is set; pending mouse or window events in the input buffer are not flushed. The mouse pointer is displayed regardless of the mouse mode.

ENABLE\_WINDOW\_INPUT Controls whether buffer-resizing events are reported in the input buffer. By default, mouse input is enabled and window input is disabled. Changing either of these modes affects only input that occurs after the mode is set; pending mouse or window events in the input buffer are not flushed. The mouse pointer is displayed regardless of the mouse mode.

ENABLE\_PROCESSED\_INPUT Controls the processing of input for applications using the high-level console I/O functions. However, if processed input mode is enabled, the CTRL+C key combination is not reported in the console's input buffer. Instead, it is passed on to the appropriate control handler function. For more information about control handlers, see Console Control Handlers.

The output modes of a screen buffer do not affect the behavior of the low-level output functions.

### 低层控制台模式

控制台的输入缓冲区中报告的输入事件的类型取决于控制台的鼠标和窗口输入模式。控制台的处理后输入模式决定了系统如何处理CTRL + C组合键。要设置或检索控制台输入模式的状态，应用程序可以在对SetConsoleMode或GetConsoleMode函数的调用中指定控制台输入缓冲区句柄。控制台输入句柄使用以下模式。

模式说明

ENABLE\_MOUSE\_INPUT控制是否在输入缓冲区中报告鼠标事件。默认情况下，启用鼠标输入，并禁用窗口输入。更改这两种模式中的任何一种仅影响设置模式后发生的输入；否则，将不影响输入。输入缓冲区中的未决鼠标或窗口事件不会刷新。无论使用哪种鼠标模式，都会显示鼠标指针。

ENABLE\_WINDOW\_INPUT控制是否在输入缓冲区中报告缓冲区调整大小事件。默认情况下，启用鼠标输入，并禁用窗口输入。更改这两种模式中的任何一种仅影响设置模式后发生的输入；否则，将不影响输入。输入缓冲区中的未决鼠标或窗口事件不会刷新。无论使用哪种鼠标模式，都会显示鼠标指针。

ENABLE\_PROCESSED\_INPUT使用高级控制台I / O功能控制应用程序的输入处理。但是，如果启用了已处理输入模式，则在控制台的输入缓冲区中不会报告CTRL + C组合键。而是将其传递给适当的控制处理函数。有关控制处理程序的更多信息，请参见控制台控制处理程序。

屏幕缓冲区的输出模式不会影响低电平输出功能的行为。

Reading and Writing Blocks of Characters and Attributes

The ReadConsoleOutput function copies a rectangular block of character and color attribute data from a console screen buffer into a destination buffer. The function treats the destination buffer as a two-dimensional array of CHAR\_INFO structures. Similarly, the WriteConsoleOutput function copies a rectangular block of character and color attribute data from a source buffer to a console screen buffer. For more information about reading from or writing to rectangular blocks of screen buffer cells, see Input and Output Methods.

The following example uses the CreateConsoleScreenBuffer function to create a new screen buffer. After the SetConsoleActiveScreenBuffer function makes this the active screen buffer, a block of characters and color attributes is copied from the top two rows of the STDOUT screen buffer into a temporary buffer. The data is then copied from the temporary buffer into the new active screen buffer. When the application is finished using the new screen buffer, it calls SetConsoleActiveScreenBuffer to restore the original STDOUT screen buffer.

### 字符和属性的读写块

ReadConsoleOutput函数将字符和颜色属性数据的矩形块从控制台屏幕缓冲区复制到目标缓冲区。该函数将目标缓冲区视为CHAR\_INFO结构的二维数组。

同样，WriteConsoleOutput函数将字符和颜色属性数据的矩形块从源缓冲区复制到控制台屏幕缓冲区。有关读取或写入屏幕缓冲单元的矩形块的更多信息，请参见输入和输出方法。

下面的示例使用CreateConsoleScreenBuffer函数创建一个新的屏幕缓冲区。在SetConsoleActiveScreenBuffer函数将此设置为活动屏幕缓冲区之后，会将字符和颜色属性块从STDOUT屏幕缓冲区的前两行复制到一个临时缓冲区中。然后将数据从临时缓冲区复制到新的活动屏幕缓冲区。应用程序使用新的屏幕缓冲区完成后，它将调用SetConsoleActiveScreenBuffer还原原始的STDOUT屏幕缓冲区。

#include <windows.h>

#include <stdio.h>

int main(void)

{

HANDLE hStdout, hNewScreenBuffer;

SMALL\_RECT srctReadRect;

SMALL\_RECT srctWriteRect;

CHAR\_INFO chiBuffer[160]; // [2][80];

COORD coordBufSize;

COORD coordBufCoord;

BOOL fSuccess;

// Get a handle to the STDOUT screen buffer to copy from and

// create a new screen buffer to copy to.

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

hNewScreenBuffer = CreateConsoleScreenBuffer(

GENERIC\_READ | // read/write access

GENERIC\_WRITE,

FILE\_SHARE\_READ |

FILE\_SHARE\_WRITE, // shared

NULL, // default security attributes

CONSOLE\_TEXTMODE\_BUFFER, // must be TEXTMODE

NULL); // reserved; must be NULL

if (hStdout == INVALID\_HANDLE\_VALUE ||

hNewScreenBuffer == INVALID\_HANDLE\_VALUE)

{

printf("CreateConsoleScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

// Make the new screen buffer the active screen buffer.

if (! SetConsoleActiveScreenBuffer(hNewScreenBuffer) )

{

printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

// Set the source rectangle.

srctReadRect.Top = 0; // top left: row 0, col 0

srctReadRect.Left = 0;

srctReadRect.Bottom = 1; // bot. right: row 1, col 79

srctReadRect.Right = 79;

// The temporary buffer size is 2 rows x 80 columns.

coordBufSize.Y = 2;

coordBufSize.X = 80;

// The top left destination cell of the temporary buffer is

// row 0, col 0.

coordBufCoord.X = 0;

coordBufCoord.Y = 0;

// Copy the block from the screen buffer to the temp. buffer.

fSuccess = ReadConsoleOutput(

hStdout, // screen buffer to read from

chiBuffer, // buffer to copy into

coordBufSize, // col-row size of chiBuffer

coordBufCoord, // top left dest. cell in chiBuffer

&srctReadRect); // screen buffer source rectangle

if (! fSuccess)

{

printf("ReadConsoleOutput failed - (%d)\n", GetLastError());

return 1;

}

// Set the destination rectangle.

srctWriteRect.Top = 10; // top lt: row 10, col 0

srctWriteRect.Left = 0;

srctWriteRect.Bottom = 11; // bot. rt: row 11, col 79

srctWriteRect.Right = 79;

// Copy from the temporary buffer to the new screen buffer.

fSuccess = WriteConsoleOutput(

hNewScreenBuffer, // screen buffer to write to

chiBuffer, // buffer to copy from

coordBufSize, // col-row size of chiBuffer

coordBufCoord, // top left src cell in chiBuffer

&srctWriteRect); // dest. screen buffer rectangle

if (! fSuccess)

{

printf("WriteConsoleOutput failed - (%d)\n", GetLastError());

return 1;

}

Sleep(5000);

// Restore the original active screen buffer.

if (! SetConsoleActiveScreenBuffer(hStdout))

{

printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

return 0;

}

Reading Input Buffer Events

The ReadConsoleInput function can be used to directly access a console's input buffer. When a console is created, mouse input is enabled and window input is disabled. To ensure that the process receives all types of events, this example uses the SetConsoleMode function to enable window and mouse input. Then it goes into a loop that reads and handles 100 console input events. For example, the message "Keyboard event" is displayed when the user presses a key and the message "Mouse event" is displayed when the user interacts with the mouse.

### 读取输入缓冲区事件

ReadConsoleInput函数可用于直接访问控制台的输入缓冲区。 创建控制台后，将启用鼠标输入，并禁用窗口输入。 为了确保该进程接收所有类型的事件，此示例使用SetConsoleMode函数启用窗口和鼠标输入。 然后进入读取和处理100个控制台输入事件的循环。 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”。

#include <windows.h>

#include <stdio.h>

HANDLE hStdin;

DWORD fdwSaveOldMode;

VOID ErrorExit(LPSTR);

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);

}

Clearing the Screen

There are two ways to clear the screen in a console application.

Example 1

The first method is to use the C run-time system function. The system function invokes the cls command provided by the command interpreter to clear the screen.

Example 2

The second method is to write a function to programmatically clear the screen using the FillConsoleOutputCharacter and FillConsoleOutputAttribute functions. The following sample code demonstrates this technique.

### 清除屏幕

有两种方法可以清除控制台应用程序中的屏幕。

例子1

第一种方法是使用C运行时系统功能。 系统功能调用命令解释器提供的cls命令以清除屏幕。

#include <stdlib.h>

int main( void )

{

system("cls");

return 0;

}

例子2

第二种方法是编写一个函数，以使用FillConsoleOutputCharacter和FillConsoleOutputAttribute函数以编程方式清除屏幕。 下面的示例代码演示了此技术。

#include <windows.h>

void cls( HANDLE hConsole )

{

COORD coordScreen = { 0, 0 }; // home for the cursor

DWORD cCharsWritten;

CONSOLE\_SCREEN\_BUFFER\_INFO csbi;

DWORD dwConSize;

// Get the number of character cells in the current buffer.

if( !GetConsoleScreenBufferInfo( hConsole, &csbi ))

{

return;

}

dwConSize = csbi.dwSize.X \* csbi.dwSize.Y;

// Fill the entire screen with blanks.

if( !FillConsoleOutputCharacter( hConsole, // Handle to console screen buffer

(TCHAR) ' ', // Character to write to the buffer

dwConSize, // Number of cells to write

coordScreen, // Coordinates of first cell

&cCharsWritten ))// Receive number of characters written

{

return;

}

// Get the current text attribute.

if( !GetConsoleScreenBufferInfo( hConsole, &csbi ))

{

return;

}

// Set the buffer's attributes accordingly.

if( !FillConsoleOutputAttribute( hConsole, // Handle to console screen buffer

csbi.wAttributes, // Character attributes to use

dwConSize, // Number of cells to set attribute

coordScreen, // Coordinates of first cell

&cCharsWritten )) // Receive number of characters written

{

return;

}

// Put the cursor at its home coordinates.

SetConsoleCursorPosition( hConsole, coordScreen );

}

int main( void )

{

HANDLE hStdout;

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

cls(hStdout);

return 0;

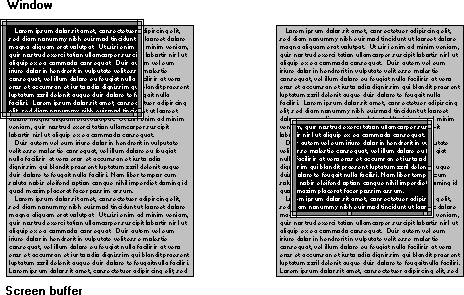
}

Scrolling the Screen Buffer

The console window displays a portion of the active screen buffer. Each screen buffer maintains its own current window rectangle that specifies the coordinates of the upper left and lower right character cells to be displayed in the console window. To determine the current window rectangle of a screen buffer, use [**GetConsoleScreenBufferInfo**](https://docs.microsoft.com/zh-cn/windows/console/getconsolescreenbufferinfo). When a screen buffer is created, the upper left corner of its window is at the upper left corner of the console screen buffer at (0,0).

The window rectangle can change to display different parts of the console screen buffer. The window rectangle of a screen buffer can change in the following situations:

* When [**SetConsoleWindowInfo**](https://docs.microsoft.com/zh-cn/windows/console/setconsolewindowinfo) is called to specify a new window rectangle, it scrolls the view of the console screen buffer by changing the position of the window rectangle without changing the size of the window. For examples of scrolling the window's contents, see [Scrolling a Screen Buffer's Window](https://docs.microsoft.com/zh-cn/windows/console/scrolling-a-screen-buffer-s-window).



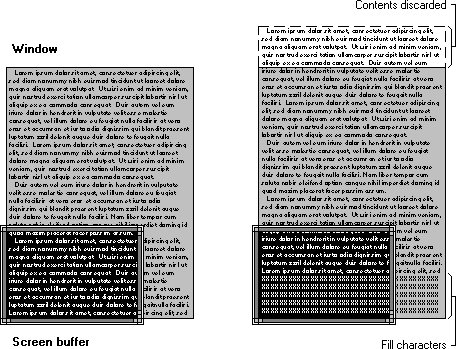
* When using the [**WriteFile**](https://msdn.microsoft.com/library/windows/desktop/aa365747) function to write to a screen buffer with wrap at end-of-line (EOL) output mode enabled, the window rectangle shifts automatically, so the cursor is always displayed.
* When the [**SetConsoleCursorPosition**](https://docs.microsoft.com/zh-cn/windows/console/setconsolecursorposition) function specifies a new cursor position that is outside the boundaries of the current window rectangle, the window rectangle shifts automatically to display the cursor.
* When the user changes the size of the console window or uses the window's scroll bars, the window rectangle of the active screen buffer can change. This change is not reported as a window resizing event in the input buffer.

In each of these situations, the window rectangle shifts to display a different part of the console screen buffer, but the contents of the console screen buffer remain in the same position. The following situations can cause the console screen buffer's contents to shift:

* When the [**ScrollConsoleScreenBuffer**](https://docs.microsoft.com/zh-cn/windows/console/scrollconsolescreenbuffer) function is called, a rectangular block is copied from one part of a screen buffer to another.
* When using [**WriteFile**](https://msdn.microsoft.com/library/windows/desktop/aa365747) to write to a screen buffer with wrap at EOL output mode enabled, the console screen buffer's contents scroll automatically when the end of the console screen buffer is encountered. This scrolling discards the top row of the console screen buffer.

[**ScrollConsoleScreenBuffer**](https://docs.microsoft.com/zh-cn/windows/console/scrollconsolescreenbuffer) specifies the console screen buffer rectangle that is moved and the new upper left coordinates to which the rectangle is copied. This function can scroll a portion or the entire contents of the console screen buffer.

The illustration shows a [**ScrollConsoleScreenBuffer**](https://docs.microsoft.com/zh-cn/windows/console/scrollconsolescreenbuffer) operation that scrolls the entire contents of the console screen buffer up by several rows. The contents of the top rows are discarded, and the bottom rows are filled with a specified character and color.



The effects of [**ScrollConsoleScreenBuffer**](https://docs.microsoft.com/zh-cn/windows/console/scrollconsolescreenbuffer) can be limited by specifying an optional clipping rectangle so that the contents of the console screen buffer outside the clipping rectangle are unchanged. The effect of clipping is to create a subwindow (the clipping rectangle) whose contents are scrolled without affecting the rest of the console screen buffer. For an example that uses **ScrollConsoleScreenBuffer**, see [Scrolling a Screen Buffer's Contents](https://docs.microsoft.com/zh-cn/windows/console/scrolling-a-screen-buffer-s-contents).

### 滚动屏幕缓冲区

控制台窗口显示活动屏幕缓冲区的一部分。每个屏幕缓冲区都维护自己的当前窗口矩形，该矩形指定要在控制台窗口中显示的左上和右下字符单元格的坐标。若要确定屏幕缓冲区的当前窗口矩形，请使用GetConsoleScreenBufferInfo。创建屏幕缓冲区时，其窗口的左上角位于控制台屏幕缓冲区的左上角（0,0）。

窗口矩形可以更改以显示控制台屏幕缓冲区的不同部分。在以下情况下，屏幕缓冲区的窗口矩形可以更改：

调用SetConsoleWindowInfo以指定新的窗口矩形时，它将通过更改窗口矩形的位置而不更改窗口大小来滚动控制台屏幕缓冲区的视图。有关滚动窗口内容的示例，请参见滚动屏幕缓冲区的窗口。

屏幕缓冲区窗口

当使用WriteFile函数在启用了行尾换行（EOL）输出模式的情况下写入屏幕缓冲区时，窗口矩形会自动移动，因此始终显示光标。

当SetConsoleCursorPosition函数指定一个新的光标位置超出当前窗口矩形的边界时，窗口矩形会自动移动以显示光标。

当用户更改控制台窗口的大小或使用窗口的滚动条时，活动屏幕缓冲区的窗口矩形可以更改。在输入缓冲区中，此更改未报告为窗口大小调整事件。

在每种情况下，窗口矩形都会移动以显示控制台屏幕缓冲区的不同部分，但控制台屏幕缓冲区的内容将保持在同一位置。以下情况会导致控制台屏幕缓冲区的内容移位：

调用ScrollConsoleScreenBuffer函数时，会将矩形块从屏幕缓冲区的一部分复制到另一部分。

当使用WriteFile在启用了EOL输出模式的自动换行的情况下写入屏幕缓冲区时，遇到控制台屏幕缓冲区的末尾时，控制台屏幕缓冲区的内容会自动滚动。此滚动将放弃控制台屏幕缓冲区的第一行。

ScrollConsoleScreenBuffer指定要移动的控制台屏幕缓冲区矩形以及将矩形复制到的新的左上坐标。此功能可以滚动控制台屏幕缓冲区的一部分或全部内容。

该图显示了ScrollConsoleScreenBuffer操作，该操作将控制台屏幕缓冲区的全部内容向上滚动几行。最上面几行的内容将被丢弃，最下面几行将使用指定的字符和颜色填充。

屏幕缓冲区窗口

可以通过指定一个可选的剪切矩形来限制ScrollConsoleScreenBuffer的效果，以使剪切矩形外部的控制台屏幕缓冲区的内容保持不变。剪辑的作用是创建一个子窗口（剪辑矩形），其内容可以滚动而不影响控制台屏幕缓冲区的其余部分。有关使用ScrollConsoleScreenBuffer的示例，请参见滚动屏幕缓冲区的内容。

Scrolling a Screen Buffer's Contents

The ScrollConsoleScreenBuffer function moves a block of character cells from one part of a screen buffer to another part of the same screen buffer. The function specifies the upper left and lower right cells of the source rectangle to be moved and the destination coordinates of the new location for the upper left cell. The character and color data in the source cells is moved to the new location, and any cells left empty by the move are filled in with a specified character and color. If a clipping rectangle is specified, the cells outside of it are left unchanged.

ScrollConsoleScreenBuffer can be used to delete a line by specifying coordinates of the first cell in the line as the destination coordinates and specifying a scrolling rectangle that includes all the rows below the line.

The following example shows the use of a clipping rectangle to scroll only the bottom 15 rows of the console screen buffer. The rows in the specified rectangle are scrolled up one line at a time, and the top row of the block is discarded. The contents of the console screen buffer outside the clipping rectangle are left unchanged.

### 滚动屏幕缓冲区的内容

ScrollConsoleScreenBuffer函数将字符单元块从屏幕缓冲区的一部分移到同一屏幕缓冲区的另一部分。该函数指定要移动的源矩形的左上和右下单元格，以及左上单元格新位置的目标坐标。源单元格中的字符和颜色数据将移动到新位置，并且任何通过移动保留为空的单元格都将使用指定的字符和颜色进行填充。如果指定了裁剪矩形，则其外部的单元格将保持不变。

通过将行中第一个单元格的坐标指定为目标坐标，并指定一个包括该行以下所有行的滚动矩形，可以使用ScrollConsoleScreenBuffer删除行。

以下示例显示了使用剪切矩形仅滚动控制台屏幕缓冲区的底部15行。将指定矩形中的行一次向上滚动一行，并丢弃该块的第一行。剪切矩形外部的控制台屏幕缓冲区的内容保持不变。

#include <windows.h>

#include <stdio.h>

int main( void )

{

HANDLE hStdout;

CONSOLE\_SCREEN\_BUFFER\_INFO csbiInfo;

SMALL\_RECT srctScrollRect, srctClipRect;

CHAR\_INFO chiFill;

COORD coordDest;

int i;

printf("\nPrinting 20 lines for reference. ");

printf("Notice that line 6 is discarded during scrolling.\n");

for(i=0; i<=20; i++)

printf("%d\n", i);

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (hStdout == INVALID\_HANDLE\_VALUE)

{

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

return 1;

}

// Get the screen buffer size.

if (!GetConsoleScreenBufferInfo(hStdout, &csbiInfo))

{

printf("GetConsoleScreenBufferInfo failed %d\n", GetLastError());

return 1;

}

// The scrolling rectangle is the bottom 15 rows of the

// screen buffer.

srctScrollRect.Top = csbiInfo.dwSize.Y - 16;

srctScrollRect.Bottom = csbiInfo.dwSize.Y - 1;

srctScrollRect.Left = 0;

srctScrollRect.Right = csbiInfo.dwSize.X - 1;

// The destination for the scroll rectangle is one row up.

coordDest.X = 0;

coordDest.Y = csbiInfo.dwSize.Y - 17;

// The clipping rectangle is the same as the scrolling rectangle.

// The destination row is left unchanged.

srctClipRect = srctScrollRect;

// Fill the bottom row with green blanks.

chiFill.Attributes = BACKGROUND\_GREEN | FOREGROUND\_RED;

chiFill.Char.AsciiChar = (char)' ';

// Scroll up one line.

if(!ScrollConsoleScreenBuffer(

hStdout, // screen buffer handle

&srctScrollRect, // scrolling rectangle

&srctClipRect, // clipping rectangle

coordDest, // top left destination cell

&chiFill)) // fill character and color

{

printf("ScrollConsoleScreenBuffer failed %d\n", GetLastError());

return 1;

}

return 0;

}

Scrolling a Screen Buffer's Window

The SetConsoleWindowInfo function can be used to scroll the contents of a screen buffer in the console window. This function can also change the window size. The function can either specify the new upper left and lower right corners of the console screen buffer's window as absolute screen buffer coordinates or specify the changes from the current window coordinates. The function fails if the specified window coordinates are outside the boundaries of the console screen buffer.

The following example scrolls the view of the console screen buffer up by modifying the window coordinates returned by the GetConsoleScreenBufferInfo function. The ScrollByAbsoluteCoord function demonstrates how to specify absolute coordinates, while the ScrollByRelativeCoord function demonstrates how to specify relative coordinates.

### 滚动屏幕缓冲区的窗口

SetConsoleWindowInfo函数可用于在控制台窗口中滚动屏幕缓冲区的内容。 此功能还可以更改窗口大小。 该函数可以将控制台屏幕缓冲区窗口的新左上角和右下角指定为绝对屏幕缓冲区坐标，或指定从当前窗口坐标开始的更改。 如果指定的窗口坐标超出控制台屏幕缓冲区的边界，则该函数将失败。

下面的示例通过修改GetConsoleScreenBufferInfo函数返回的窗口坐标来向上滚动控制台屏幕缓冲区的视图。 ScrollByAbsoluteCoord函数演示如何指定绝对坐标，而ScrollByRelativeCoord函数演示如何指定相对坐标。

#include <windows.h>

#include <stdio.h>

#include <conio.h>

HANDLE hStdout;

int ScrollByAbsoluteCoord(int iRows)

{

CONSOLE\_SCREEN\_BUFFER\_INFO csbiInfo;

SMALL\_RECT srctWindow;

// Get the current screen buffer size and window position.

if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))

{

printf("GetConsoleScreenBufferInfo (%d)\n", GetLastError());

return 0;

}

// Set srctWindow to the current window size and location.

srctWindow = csbiInfo.srWindow;

// Check whether the window is too close to the screen buffer top

if ( srctWindow.Top >= iRows )

{

srctWindow.Top -= (SHORT)iRows; // move top up

srctWindow.Bottom -= (SHORT)iRows; // move bottom up

if (! SetConsoleWindowInfo(

hStdout, // screen buffer handle

TRUE, // absolute coordinates

&srctWindow)) // specifies new location

{

printf("SetConsoleWindowInfo (%d)\n", GetLastError());

return 0;

}

return iRows;

}

else

{

printf("\nCannot scroll; the window is too close to the top.\n");

return 0;

}

}

int ScrollByRelativeCoord(int iRows)

{

CONSOLE\_SCREEN\_BUFFER\_INFO csbiInfo;

SMALL\_RECT srctWindow;

// Get the current screen buffer window position.

if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))

{

printf("GetConsoleScreenBufferInfo (%d)\n", GetLastError());

return 0;

}

// Check whether the window is too close to the screen buffer top

if (csbiInfo.srWindow.Top >= iRows)

{

srctWindow.Top =- (SHORT)iRows; // move top up

srctWindow.Bottom =- (SHORT)iRows; // move bottom up

srctWindow.Left = 0; // no change

srctWindow.Right = 0; // no change

if (! SetConsoleWindowInfo(

hStdout, // screen buffer handle

FALSE, // relative coordinates

&srctWindow)) // specifies new location

{

printf("SetConsoleWindowInfo (%d)\n", GetLastError());

return 0;

}

return iRows;

}

else

{

printf("\nCannot scroll; the window is too close to the top.\n");

return 0;

}

}

int main( void )

{

int i;

printf("\nPrinting twenty lines, then scrolling up five lines.\n");

printf("Press any key to scroll up ten lines; ");

printf("then press another key to stop the demo.\n");

for(i=0; i<=20; i++)

printf("%d\n", i);

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

if(ScrollByAbsoluteCoord(5))

\_getch();

else return 0;

if(ScrollByRelativeCoord(10))

\_getch();

else return 0;

}

CTRL+C and CTRL+BREAK Signals

The CTRL+C and CTRL+BREAK key combinations receive special handling by console processes. By default, when a console window has the keyboard focus, CTRL+C or CTRL+BREAK is treated as a signal (SIGINT or SIGBREAK) and not as keyboard input. By default, these signals are passed to all console processes that are attached to the console. (Detached processes are not affected.) The system creates a new thread in each client process to handle the event. The thread raises an exception if the process is being debugged. The debugger can handle the exception or continue with the exception unhandled.

CTRL+BREAK is always treated as a signal, but an application can change the default CTRL+C behavior in two ways that prevent the handler functions from being called:

The SetConsoleMode function can disable the ENABLE\_PROCESSED\_INPUT input mode for a console's input buffer, so CTRL+C is reported as keyboard input rather than as a signal.

When SetConsoleCtrlHandler is called with NULL and TRUE values for its parameters, the calling process ignores CTRL+C signals. Normal CTRL+C processing is restored by calling SetConsoleCtrlHandler with NULL and FALSE values. This attribute of ignoring or not ignoring CTRL+C signals is inherited by child processes, but it can be enabled or disabled by any process without affecting existing processes.

### CTRL + C和CTRL + BREAK信号

CTRL + C和CTRL + BREAK组合键可通过控制台进程进行特殊处理。默认情况下，当控制台窗口具有键盘焦点时，CTRL + C或CTRL + BREAK被视为信号（SIGINT或SIGBREAK），而不是键盘输入。默认情况下，这些信号会传递到连接到控制台的所有控制台进程。 （独立的进程不受影响。）系统在每个客户端进程中创建一个新线程来处理事件。如果正在调试进程，线程将引发异常。调试器可以处理异常或继续处理未处理的异常。

CTRL + BREAK始终被视为信号，但是应用程序可以通过两种方式来更改默认的CTRL + C行为，以防止调用处理程序函数：

函数可以为控制台的输入缓冲区禁用ENABLE\_PROCESSED\_INPUT输入模式，因此CTRL + C被报告为键盘输入而不是信号。

当使用参数的NULL和TRUE值调用SetConsoleCtrlHandler时，调用过程将忽略CTRL + C信号。通过使用NULL和FALSE值调用SetConsoleCtrlHandler，可以恢复正常的CTRL + C处理。忽略或不忽略CTRL + C信号的属性由子进程继承，但是任何进程都可以启用或禁用它，而不会影响现有进程。

CTRL+CLOSE Signal

The system generates a CTRL+CLOSE signal when the user closes a console. All processes attached to the console receive the signal, giving each process an opportunity to clean up before termination. When a process receives this signal, the handler function can take one of the following actions after performing any cleanup operations:

Call ExitProcess to terminate the process.

Return FALSE. If none of the registered handler functions returns TRUE, the default handler terminates the process.

Return TRUE. In this case, no other handler functions are called and the process terminates.

### CTRL + CLOSE信号

当用户关闭控制台时，系统会生成CTRL + CLOSE信号。 控制台上连接的所有进程均会接收信号，从而使每个进程都有机会在终止之前进行清理。 当进程收到此信号时，处理程序函数在执行任何清除操作后可以采取以下操作之一：

调用ExitProcess终止该过程。

返回FALSE。 如果所有注册的处理程序函数均未返回TRUE，则默认处理程序将终止该过程。

返回TRUE。 在这种情况下，不会调用其他处理程序函数，并且该过程终止。

Registering a Control Handler Function

This is an example of the SetConsoleCtrlHandler function that is used to install a control handler.

When a CTRL+C signal is received, the control handler returns TRUE, indicating that it has handled the signal. Doing this prevents other control handlers from being called.

When a CTRL\_CLOSE\_EVENT signal is received, the control handler returns TRUE and the process terminates.

When a CTRL\_BREAK\_EVENT, CTRL\_LOGOFF\_EVENT, or CTRL\_SHUTDOWN\_EVENT signal is received, the control handler returns FALSE. Doing this causes the signal to be passed to the next control handler function. If no other control handlers have been registered or none of the registered handlers returns TRUE, the default handler will be used, resulting in the process being terminated.

### 注册控制处理程序功能

这是SetConsoleCtrlHandler函数的示例，该函数用于安装控件处理程序。

收到CTRL + C信号后，控制处理程序将返回TRUE，表示已处理该信号。 这样做可以防止其他控件处理程序被调用。

当收到CTRL\_CLOSE\_EVENT信号时，控制处理程序返回TRUE，并且过程终止。

当收到CTRL\_BREAK\_EVENT，CTRL\_LOGOFF\_EVENT或CTRL\_SHUTDOWN\_EVENT信号时，控制处理程序将返回FALSE。 这样做会使信号传递到下一个控制处理程序函数。 如果没有其他控制处理程序被注册，或者没有一个注册处理程序返回TRUE，则将使用默认处理程序，从而导致进程终止。

// CtrlHandler.cpp : This file contains the 'main' function. Program execution begins and ends there.

//

#include "pch.h"

#include <windows.h>

#include <stdio.h>

BOOL WINAPI CtrlHandler(DWORD fdwCtrlType)

{

switch (fdwCtrlType)

{

// Handle the CTRL-C signal.

case CTRL\_C\_EVENT:

printf("Ctrl-C event\n\n");

Beep(750, 300);

return TRUE;

// CTRL-CLOSE: confirm that the user wants to exit.

case CTRL\_CLOSE\_EVENT:

Beep(600, 200);

printf("Ctrl-Close event\n\n");

return TRUE;

// Pass other signals to the next handler.

case CTRL\_BREAK\_EVENT:

Beep(900, 200);

printf("Ctrl-Break event\n\n");

return FALSE;

case CTRL\_LOGOFF\_EVENT:

Beep(1000, 200);

printf("Ctrl-Logoff event\n\n");

return FALSE;

case CTRL\_SHUTDOWN\_EVENT:

Beep(750, 500);

printf("Ctrl-Shutdown event\n\n");

return FALSE;

default:

return FALSE;

}

}

int main(void)

{

if (SetConsoleCtrlHandler(CtrlHandler, TRUE))

{

printf("\nThe Control Handler is installed.\n");

printf("\n -- Now try pressing Ctrl+C or Ctrl+Break, or");

printf("\n try logging off or closing the console...\n");

printf("\n(...waiting in a loop for events...)\n\n");

while (1) {}

}

else

{

printf("\nERROR: Could not set control handler");

return 1;

}

return 0;

}

Console Virtual Terminal Sequences

Virtual terminal sequences are control character sequences that can control cursor movement, color/font mode, and other operations when written to the output stream. Sequences may also be received on the input stream in response to an output stream query information sequence or as an encoding of user input when the appropriate mode is set.

You can use GetConsoleMode and SetConsoleMode flags to configure this behavior. A sample of the suggested way to enable virtual terminal behaviors is included at the end of this document.

The behavior of the following sequences is based on the VT100 and derived terminal emulator technologies, most specifically the xterm terminal emulator. More information about terminal sequences can be found at http://vt100.net and at http://invisible-island.net/xterm/ctlseqs/ctlseqs.html.

Output Sequences

The following terminal sequences are intercepted by the console host when written into the output stream, if the ENABLE\_VIRTUAL\_TERMINAL\_PROCESSING flag is set on the screen buffer handle using the SetConsoleMode flag. Note that the DISABLE\_NEWLINE\_AUTO\_RETURN flag may also be useful in emulating the cursor positioning and scrolling behavior of other terminal emulators in relation to characters written to the final column in any row.

Simple Cursor Positioning

In all of the following descriptions, ESC is always the hexadecimal value 0x1B. No spaces are to be included in terminal sequences. For an example of how these sequences are used in practice, please see the example at the end of this topic.

The following table describes simple escape sequences with a single action command directly after the ESC character. These sequences have no parameters and take effect immediately.

All commands in this table are generally equivalent to calling the SetConsoleCursorPosition console API to place the cursor.

Cursor movement will be bounded by the current viewport into the buffer. Scrolling (if available) will not occur.

### 控制台虚拟终端序列

虚拟终端序列是控制字符序列，可以在写入输出流时控制光标移动，颜色/字体模式和其他操作。当设置了适当的模式时，还可以响应于输出流查询信息序列或者在用户输入的编码的情况下，在输入流上接收序列。

您可以使用GetConsoleMode和SetConsoleMode标志来配置此行为。本文档末尾包含了建议的启用虚拟终端行为的方法的示例。

以下序列的行为基于VT100和派生的终端仿真器技术，尤其是xterm终端仿真器。有关终端序列的更多信息，请参见http://vt100.net和<http://invisible-island.net/xterm/ctlseqs/ctlseqs.html>。

输出顺序

如果使用SetConsoleMode标志在屏幕缓冲区句柄上设置了ENABLE\_VIRTUAL\_TERMINAL\_PROCESSING标志，则在写入输出流时，控制台主机将拦截以下终端序列。请注意，DISABLE\_NEWLINE\_AUTO\_RETURN标志在模拟其他终端仿真器的光标位置和滚动行为方面也可能很有用，该行为与写入任何行的最后一列的字符有关。

简单的光标定位

在以下所有描述中，ESC始终是十六进制值0x1B。终端序列中不得包含空格。有关如何在实践中使用这些序列的示例，请参见本主题末尾的示例。

下表描述了直接在ESC字符后使用单个动作命令的简单转义序列。这些序列没有参数，并且立即生效。

该表中的所有命令通常等效于调用SetConsoleCursorPosition控制台API来放置光标。

光标移动将由当前视口限制到缓冲区中。滚动（如果可用）将不会发生。

Creating a Pseudoconsole session

The Windows Pseudoconsole, sometimes also referred to as pseudo console, ConPTY, or the Windows PTY, is a mechanism designed for creating an external host for character-mode subsystem activities that replace the user interactivity portion of the default console host window.

Hosting a pseudoconsole session is a bit different than a traditional console session. Traditional console sessions automatically start when the operating system recognizes that a character-mode application is about to run. In contrast, a pseudoconsole session and the communication channels need to be created by the hosting application prior to creating the process with the child character-mode application to be hosted. The child process will still be created using the CreateProcess function, but with some additional information that will direct the operating system to establish the appropriate environment.

You can find additional background information about this system on the initial announcement blog post.

Complete examples of using the Pseudoconsole are available on our GitHub repository Microsoft/console in the samples directory.

Preparing the communication channels

The first step is to create a pair of synchronous communication channels that will be provided during creation of the pseudoconsole session for bidirectional communication with the hosted application. These channels are processed by the pseudoconsole system using ReadFile and WriteFile with synchronous I/O. Any file or I/O device handle like a file, file stream, physical disk, volume, tape drive, socket, communications resource, mailslot, or pipe is acceptable as long as it does not require an OVERLAPPED structure to be processed for asynchronous communication.

NOTE:

To prevent race conditions and deadlocks, we highly recommend that each of the communication channels is serviced on a separate thread that maintains its own client buffer state and messaging queue inside your application. Servicing all of the pseudoconsole activities on the same thread may result in a deadlock where one of the communications buffers is filled and waiting for your action while you attempt to dispatch a blocking request on another channel.

Creating the Pseudoconsole

With the communications channels that have been established, identify the "read" end of the input channel and the "write" end of the output channel. This pair of handles is provided on calling CreatePseudoConsole to create the object.

When creating, a size representing the X and Y dimensions (in count of characters) is also provided. This is the dimensions that will apply to the display surface for the final (terminal) presentation window. The values are used to create an in-memory buffer inside the pseudoconsole system.

The buffer size provide answers to client character-mode applications that probe for information using the client-side console functions like GetConsoleScreenBufferInfoEx and dictates the layout and positioning of text when clients use functions like WriteConsoleOutput.

Finally, a flags field is provided on creation of a pseudoconsole to perform special functionality. By default, set this to 0 to have no special functionality.

At this time, only one special flag is available to request the inheritence of the cursor position from a console session already attached to the caller of the pseudoconsole API. This is intended for use in more advanced scenarios where a hosting application that is preparing a pseudoconsole session is itself also a client character-mode application of a another console environment.

A sample snippet is provided below utilizing CreatePipe to establish a pair of communication channels and create the pseudoconsole.

### 创建一个伪控制台会话

Windows Pseudoconsole（有时也称为伪控制台，ConPTY或Windows PTY）是一种机制，旨在为字符模式子系统活动创建外部主机，以替换默认控制台主机窗口的用户交互部分。

托管伪控制台会话与传统的控制台会话有些不同。当操作系统识别出字符模式应用程序即将运行时，传统的控制台会话将自动启动。相反，在使用要托管的子字符模式应用程序创建进程之前，托管应用程序需要创建伪控制台会话和通信通道。仍将使用CreateProcess函数创建子进程，但是带有一些其他信息，这些信息将指导操作系统建立适当的环境。

您可以在初始公告博客文章中找到有关此系统的其他背景信息。

在我们的GitHub存储库Microsoft / console中的samples目录中提供了使用Pseudoconsole的完整示例。

准备沟通渠道

第一步是创建一对同步通信通道，该通道将在创建伪控制台会话期间提供，用于与托管应用程序进行双向通信。这些通道由伪控制台系统使用带有同步I / O的ReadFile和WriteFile进行处理。任何文件或I / O设备句柄（例如文件，文件流，物理磁盘，卷，磁带驱动器，套接字，通信资源，邮筒或管道）都可以接受，只要它不需要处理OVERLAPPED结构即可进行异步通信。

注意：

为了防止出现竞争情况和死锁，我们强烈建议在单独的线程中为每个通信通道提供服务，这些线程在应用程序内部维护其自己的客户端缓冲区状态和消息传递队列。在同一线程上服务所有伪控制台活动可能会导致死锁，在该死锁中，通信缓冲区之一已满，并在您尝试在另一个通道上分派阻塞请求时等待您的操作。

创建伪控制台

使用已建立的通信通道，标识输入通道的“读取”端和输出通道的“写入”端。在调用CreatePseudoConsole创建对象时提供了这对句柄。

创建时，还提供了代表X和Y尺寸（以字符数为单位）的尺寸。这是将应用于最终（终端）展示窗口的显示表面的尺寸。这些值用于在伪控制台系统内部创建内存中的缓冲区。

缓冲区大小为客户端字符模式应用程序提供了答案，客户端字符模式应用程序使用客户端控制台功能（如GetConsoleScreenBufferInfoEx）探测信息，并在客户端使用WriteConsoleOutput等功能时规定文本的布局和位置。

最后，在创建伪控制台时会提供标志字段以执行特殊功能。默认情况下，将此设置为0不会具有特殊功能。

此时，只有一个特殊标志可用于从已经附加到伪控制台API调用程序的控制台会话中请求继承光标位置。它旨在用于更高级的场景中，其中准备伪控制台会话的宿主应用程序本身也是另一个控制台环境的客户端字符模式应用程序。

下面提供了一个示例片段，利用CreatePipe建立了一对通信通道并创建了伪控制台。

HRESULT SetUpPseudoConsole(COORD size)

{

HRESULT hr = S\_OK;

// Create communication channels

// - Close these after CreateProcess of child application with pseudoconsole object.

HANDLE inputReadSide, outputWriteSide;

// - Hold onto these and use them for communication with the child through the pseudoconsole.

HANDLE outputReadSide, inputWriteSide;

if (!CreatePipe(&inputReadSide, &inputWriteSide, NULL, 0))

{

return HRESULT\_FROM\_WIN32(GetLastError());

}

if (!CreatePipe(&outputReadSide, &outputWriteSide, NULL, 0))

{

return HRESULT\_FROM\_WIN32(GetLastError());

}

HPCON hPC;

hr = CreatePseudoConsole(size, inputReadSide, outputWriteSide, 0, &hPC);

if (FAILED(hr))

{

return hr;

}

// ...

}

注意：

该代码段不完整，仅用于演示此特定呼叫。您将需要适当地管理HANDLE的寿命。未能正确管理HANDLE的生存期可能会导致死锁，特别是在同步I / O调用中。

创建用于附加到伪控制台的客户端字符模式应用程序的CreateProcess调用完成后，应将创建过程中给定的句柄从该过程中释放出来。当伪控制台会话关闭其句柄副本时，这将减少基础设备对象上的引用计数，并允许I / O操作正确检测断开的通道。

为创建子进程做准备

下一阶段是准备STARTUPINFOEX结构，该结构将在启动子进程时传达伪控制台信息。

该结构包含提供复杂启动信息的能力，包括用于进程和线程创建的属性。

以两次调用的方式使用InitializeProcThreadAttributeList，首先计算保存列表所需的字节数，分配请求的内存，然后再次调用以提供不透明的内存指针以将其设置为属性列表。

接下来，调用UpdateProcThreadAttribute并传递带有标志PROC\_THREAD\_ATTRIBUTE\_PSEUDOCONSOLE，伪控制台句柄和伪控制台句柄大小的初始化属性列表。

HRESULT PrepareStartupInformation(HPCON hpc, STARTUPINFOEX\* psi)

{

// Prepare Startup Information structure

STARTUPINFOEX si;

ZeroMemory(&si, sizeof(si));

si.StartupInfo.cb = sizeof(STARTUPINFOEX);

// Discover the size required for the list

size\_t bytesRequired;

InitializeProcThreadAttributeList(NULL, 1, 0, &bytesRequired);

// Allocate memory to represent the list

si.lpAttributeList = (PPROC\_THREAD\_ATTRIBUTE\_LIST)HeapAlloc(GetProcessHeap(), 0, bytesRequired);

if (!si.lpAttributeList)

{

return E\_OUTOFMEMORY;

}

// Initialize the list memory location

if (!InitializeProcThreadAttributeList(si.lpAttributeList, 1, 0, &bytesRequired))

{

HeapFree(GetProcessHeap(), 0, si.lpAttributeList);

return HRESULT\_FROM\_WIN32(GetLastError());

}

// Set the pseudoconsole information into the list

if (!UpdateProcThreadAttribute(attributeList,

0,

PROC\_THREAD\_ATTRIBUTE\_PSEUDOCONSOLE,

hpc,

sizeof(hpc),

NULL,

NULL))

{

HeapFree(GetProcessHeap(), 0, si.lpAttributeList);

return HRESULT\_FROM\_WIN32(GetLastError());

}

\*psi = si;

return S\_OK;

}

创建托管流程

接下来，调用CreateProcess，将STARTUPINFOEX结构与可执行文件的路径以及任何其他配置信息（如果适用）一起传递。 在调用以提醒系统伪控制台引用包含在扩展信息中时，设置EXTENDED\_STARTUPINFO\_PRESENT标志很重要。

HRESULT SetUpPseudoConsole(COORD size)

{

// ...

PCWSTR childApplication = L"C:\\windows\\system32\\cmd.exe";

// Create mutable text string for CreateProcessW command line string.

const size\_t bytesRequired = wcslen(childApplication) + 1; // +1 null terminator

PWSTR cmdLineMutable = (PWSTR)HeapAlloc(GetProcessHeap(), 0, bytesRequired);

if (!cmdLineMutable)

{

return E\_OUTOFMEMORY;

}

wcscpy\_s(cmdLineMutable, bytesRequired, childApplication);

PROCESS\_INFORMATION pi;

ZeroMemory(&pi, sizeof(pi));

// Call CreateProcess

if (!CreateProcessW(NULL,

cmdLineMutable,

NULL,

NULL,

FALSE,

EXTENDED\_STARTUPINFO\_PRESENT,

NULL,

NULL,

&siEx.StartupInfo,

&pi))

{

HeapFree(GetProcessHeap(), 0, cmdLineMutable);

return HRESULT\_FROM\_WIN32(GetLastError());

}

// ...

}

注意：

在托管进程仍在启动和连接时关闭伪控制台会话，可能导致客户端应用程序显示错误对话框。如果为宿主进程赋予了无效的伪控制台启动句柄，则会显示相同的错误对话框。对于托管的进程初始化代码，这两种情况是相同的。发生故障时，来自托管客户端应用程序的弹出对话框将读取0xc0000142，其中包含本地化消息，详细说明了初始化失败。

与伪控制台会话进行通信

成功创建过程后，托管应用程序可以使用输入管道的写入端将用户交互信息发送到伪控制台中，并使用输出管道的读取端从伪控制台接收图形表示信息。

完全由托管应用程序决定如何处理进一步的活动。托管应用程序可以在另一个线程中启动一个窗口，以收集用户交互输入并将其序列化到伪控制台和托管字符模式应用程序的输入管道的写入端。可以启动另一个线程以耗尽伪控制台的输出管道的读取端，解码文本和虚拟终端序列信息，并将其呈现给屏幕。

线程还可以用于将信息从伪控制台通道中继到另一个通道或设备（包括网络），以将信息远程传输到另一个进程或机器，并避免对该信息进行任何本地转码。

调整伪控制台的大小

在整个运行过程中，可能存在由于用户交互或从另一个显示/交互设备带外接收到的请求而需要更改缓冲区大小的情况。

这可以通过ResizePseudoConsole函数完成，该函数以字符数指定缓冲区的高度和宽度。

// Theoretical event handler function with theoretical

// event that has associated display properties

// on Source property.

void OnWindowResize(Event e)

{

// Retrieve width and height dimensions of display in

// characters using theoretical height/width functions

// that can retrieve the properties from the display

// attached to the event.

COORD size;

size.X = GetViewWidth(e.Source);

size.Y = GetViewHeight(e.Source);

// Call pseudoconsole API to inform buffer dimension update

ResizePseudoConsole(m\_hpc, size);

}

结束伪控制台会话

要结束会话，请使用原始伪控制台创建中的句柄调用ClosePseudoConsole函数。关闭会话时，将终止所有附加的客户端字符模式应用程序，例如CreateProcess调用中的应用程序。如果原始子级是创建其他进程的shell型应用程序，则树中所有相关的附加进程也将终止。

注意：

如果以单线程同步方式使用伪控制台，则关闭会话会有一些副作用，这些副作用可能导致死锁。关闭伪控制台会话的操作可能会向hOutput发出最终帧更新，该更新应从通信通道缓冲区中删除。此外，如果在创建伪控制台时选择了PSEUDOCONSOLE\_INHERIT\_CURSOR，则尝试关闭伪控制台而不响应游标继承查询消息（在hOutput上接收并通过hInput答复）可能会导致另一个死锁情况。建议伪控制台的通信通道在单独的线程上进行服务，并保持耗尽和处理状态，直到客户端应用程序退出或调用ClosePseudoConsole函数完成拆解活动而自行破坏为止。

Console Functions

The following functions are used to access a console.

Function Description

AddConsoleAlias Defines a console alias for the specified executable.

AllocConsole Allocates a new console for the calling process.

AttachConsole Attaches the calling process to the console of the specified process.

ClosePseudoConsole Closes a pseudoconsole from the given handle.

CreatePseudoConsole Allocates a new pseudoconsole for the calling process.

CreateConsoleScreenBuffer Creates a console screen buffer.

FillConsoleOutputAttribute Sets the text and background color attributes for a specified number of character cells.

FillConsoleOutputCharacter Writes a character to the console screen buffer a specified number of times.

FlushConsoleInputBuffer Flushes the console input buffer.

FreeConsole Detaches the calling process from its console.

GenerateConsoleCtrlEvent Sends a specified signal to a console process group that shares the console associated with the calling process.

GetConsoleAlias Retrieves the specified alias for the specified executable.

GetConsoleAliases Retrieves all defined console aliases for the specified executable.

GetConsoleAliasesLength Returns the size, in bytes, of the buffer needed to store all of the console aliases for the specified executable.

GetConsoleAliasExes Retrieves the names of all executables with console aliases defined.

GetConsoleAliasExesLength Returns the size, in bytes, of the buffer needed to store the names of all executables that have console aliases defined.

GetConsoleCP Retrieves the input code page used by the console associated with the calling process.

GetConsoleCursorInfo Retrieves information about the size and visibility of the cursor for the specified console screen buffer.

GetConsoleDisplayMode Retrieves the display mode of the current console.

GetConsoleFontSize Retrieves the size of the font used by the specified console screen buffer.

GetConsoleHistoryInfo Retrieves the history settings for the calling process's console.

GetConsoleMode Retrieves the current input mode of a console's input buffer or the current output mode of a console screen buffer.

GetConsoleOriginalTitle Retrieves the original title for the current console window.

GetConsoleOutputCP Retrieves the output code page used by the console associated with the calling process.

GetConsoleProcessList Retrieves a list of the processes attached to the current console.

GetConsoleScreenBufferInfo Retrieves information about the specified console screen buffer.

GetConsoleScreenBufferInfoEx Retrieves extended information about the specified console screen buffer.

GetConsoleSelectionInfo Retrieves information about the current console selection.

GetConsoleTitle Retrieves the title for the current console window.

GetConsoleWindow Retrieves the window handle used by the console associated with the calling process.

GetCurrentConsoleFont Retrieves information about the current console font.

GetCurrentConsoleFontEx Retrieves extended information about the current console font.

GetLargestConsoleWindowSize Retrieves the size of the largest possible console window.

GetNumberOfConsoleInputEvents Retrieves the number of unread input records in the console's input buffer.

GetNumberOfConsoleMouseButtons Retrieves the number of buttons on the mouse used by the current console.

GetStdHandle Retrieves a handle for the standard input, standard output, or standard error device.

HandlerRoutine An application-defined function used with the SetConsoleCtrlHandler function.

PeekConsoleInput Reads data from the specified console input buffer without removing it from the buffer.

ReadConsole Reads character input from the console input buffer and removes it from the buffer.

ReadConsoleInput Reads data from a console input buffer and removes it from the buffer.

ReadConsoleOutput Reads character and color attribute data from a rectangular block of character cells in a console screen buffer.

ReadConsoleOutputAttribute Copies a specified number of foreground and background color attributes from consecutive cells of a console screen buffer.

ReadConsoleOutputCharacter Copies a number of characters from consecutive cells of a console screen buffer.

ResizePseudoConsole Resizes the internal buffers for a pseudoconsole to the given size.

ScrollConsoleScreenBuffer Moves a block of data in a screen buffer.

SetConsoleActiveScreenBuffer Sets the specified screen buffer to be the currently displayed console screen buffer.

SetConsoleCP Sets the input code page used by the console associated with the calling process.

SetConsoleCtrlHandler Adds or removes an application-defined HandlerRoutine from the list of handler functions for the calling process.

SetConsoleCursorInfo Sets the size and visibility of the cursor for the specified console screen buffer.

SetConsoleCursorPosition Sets the cursor position in the specified console screen buffer.

SetConsoleDisplayMode Sets the display mode of the specified console screen buffer.

SetConsoleHistoryInfo Sets the history settings for the calling process's console.

SetConsoleMode Sets the input mode of a console's input buffer or the output mode of a console screen buffer.

SetConsoleOutputCP Sets the output code page used by the console associated with the calling process.

SetConsoleScreenBufferInfoEx Sets extended information about the specified console screen buffer.

SetConsoleScreenBufferSize Changes the size of the specified console screen buffer.

SetConsoleTextAttribute Sets the foreground (text) and background color attributes of characters written to the console screen buffer.

SetConsoleTitle Sets the title for the current console window.

SetConsoleWindowInfo Sets the current size and position of a console screen buffer's window.

SetCurrentConsoleFontEx Sets extended information about the current console font.

SetStdHandle Sets the handle for the standard input, standard output, or standard error device.

WriteConsole Writes a character string to a console screen buffer beginning at the current cursor location.

WriteConsoleInput Writes data directly to the console input buffer.

WriteConsoleOutput Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer.

WriteConsoleOutputAttribute Copies a number of foreground and background color attributes to consecutive cells of a console screen buffer.

WriteConsoleOutputCharacter Copies a number of characters to consecutive cells of a console screen buffer.

## 控制台函数

以下功能用于访问控制台。

功能说明

AddConsoleAlias 为指定的可执行文件定义控制台别名。

AllocConsole为调用过程分配一个新的控制台。

AttachConsole将调用进程附加到指定进程的控制台。

ClosePseudoConsole从给定的句柄关闭伪控制台。

CreatePseudoConsole为调用过程分配一个新的伪控制台。

CreateConsoleScreenBuffer创建控制台屏幕缓冲区。

FillConsoleOutputAttribute为指定数量的字符单元格设置文本和背景颜色属性。

FillConsoleOutputCharacter将字符写入控制台屏幕缓冲区指定的次数。

FlushConsoleInputBuffer刷新控制台输入缓冲区。

FreeConsole从其控制台分离调用过程。

GenerateConsoleCtrlEvent将指定信号发送到共享与调用进程关联的控制台的控制台进程组。

GetConsoleAlias检索指定可执行文件的指定别名。

GetConsoleAliases检索指定可执行文件的所有定义的控制台别名。

GetConsoleAliasesLength返回为指定的可执行文件存储所有控制台别名所需的缓冲区大小（以字节为单位）。

GetConsoleAliasExes检索定义了控制台别名的所有可执行文件的名称。

GetConsoleAliasExesLength返回缓冲区的大小（以字节为单位），该缓冲区用于存储定义了控制台别名的所有可执行文件的名称。

GetConsoleCP检索与调用进程关联的控制台使用的输入代码页。

GetConsoleCursorInfo检索有关指定控制台屏幕缓冲区的光标的大小和可见性的信息。

GetConsoleDisplayMode检索当前控制台的显示模式。

GetConsoleFontSize检索指定控制台屏幕缓冲区使用的字体的大小。

GetConsoleHistoryInfo检索调用进程的控制台的历史记录设置。

GetConsoleMode检索控制台输入缓冲区的当前输入模式或控制台屏幕缓冲区的当前输出模式。

GetConsoleOriginalTitle检索当前控制台窗口的原始标题。

GetConsoleOutputCP检索与调用进程关联的控制台使用的输出代码页。

GetConsoleProcessList检索附加到当前控制台的进程的列表。

GetConsoleScreenBufferInfo检索有关指定控制台屏幕缓冲区的信息。

GetConsoleScreenBufferInfoEx检索有关指定控制台屏幕缓冲区的扩展信息。

GetConsoleSelectionInfo检索有关当前控制台选择的信息。

GetConsoleTitle检索当前控制台窗口的标题。

GetConsoleWindow检索与调用进程关联的控制台使用的窗口句柄。

GetCurrentConsoleFont检索有关当前控制台字体的信息。

GetCurrentConsoleFontEx检索有关当前控制台字体的扩展信息。

GetLargestConsoleWindowSize检索最大可能的控制台窗口的大小。

GetNumberOfConsoleInputEvents检索控制台输入缓冲区中未读输入记录的数量。

GetNumberOfConsoleMouseButtons检索当前控制台使用的鼠标按钮数量。

GetStdHandle检索标准输入，标准输出或标准错误设备的句柄。

HandlerRoutine与SetConsoleCtrlHandler函数一起使用的应用程序定义的函数。

PeekConsoleInput从指定的控制台输入缓冲区读取数据，而不将其从缓冲区中删除。

ReadConsole从控制台输入缓冲区读取字符输入，并将其从缓冲区中删除。

ReadConsoleInput从控制台输入缓冲区读取数据，并将其从缓冲区中删除。

ReadConsoleOutput从控制台屏幕缓冲区中字符单元的矩形块中读取字符和颜色属性数据。

ReadConsoleOutputAttribute从控制台屏幕缓冲区的连续单元中复制指定数量的前景色和背景色属性。

ReadConsoleOutputCharacter从控制台屏幕缓冲区的连续单元中复制许多字符。

ResizePseudoConsole将伪控制台的内部缓冲区调整为给定大小。

ScrollConsoleScreenBuffer在屏幕缓冲区中移动数据块。

SetConsoleActiveScreenBuffer将指定的屏幕缓冲区设置为当前显示的控制台屏幕缓冲区。

SetConsoleCP设置与调用进程关联的控制台使用的输入代码页。

SetConsoleCtrlHandler从调用过程的处理程序函数列表中添加或删除应用程序定义的HandlerRoutine。

SetConsoleCursorInfo设置指定控制台屏幕缓冲区的光标的大小和可见性。

SetConsoleCursorPosition设置光标在指定控制台屏幕缓冲区中的位置。

SetConsoleDisplayMode设置指定的控制台屏幕缓冲区的显示模式。

SetConsoleHistoryInfo设置调用进程的控制台的历史记录设置。

SetConsoleMode设置控制台的输入缓冲区的输入模式或控制台屏幕缓冲区的输出模式。

SetConsoleOutputCP设置与调用进程关联的控制台使用的输出代码页。

SetConsoleScreenBufferInfoEx设置有关指定控制台屏幕缓冲区的扩展信息。

SetConsoleScreenBufferSize更改指定的控制台屏幕缓冲区的大小。

SetConsoleTextAttribute设置写入控制台屏幕缓冲区的字符的前景色（文本）和背景颜色属性。

SetConsoleTitle设置当前控制台窗口的标题。

SetConsoleWindowInfo设置控制台屏幕缓冲区的窗口的当前大小和位置。

SetCurrentConsoleFontEx设置有关当前控制台字体的扩展信息。

SetStdHandle设置标准输入，标准输出或标准错误设备的句柄。

WriteConsole从当前光标位置开始，将字符串写入控制台屏幕缓冲区。

WriteConsoleInput直接将数据写入控制台输入缓冲区。

WriteConsoleOutput将字符和颜色属性数据写入控制台屏幕缓冲区中字符单元格的指定矩形块中。

WriteConsoleOutputAttribute将许多前景和背景颜色属性复制到控制台屏幕缓冲区的连续单元格中。

WriteConsoleOutputCharacter将许多字符复制到控制台屏幕缓冲区的连续单元格中。

Console Structures

The following structures are used to access a console.

CHAR\_INFO CONSOLE\_CURSOR\_INFO CONSOLE\_FONT\_INFO CONSOLE\_FONT\_INFOEX CONSOLE\_HISTORY\_INFO CONSOLE\_READCONSOLE\_CONTROL CONSOLE\_SCREEN\_BUFFER\_INFO CONSOLE\_SCREEN\_BUFFER\_INFOEX CONSOLE\_SELECTION\_INFO COORD FOCUS\_EVENT\_RECORD INPUT\_RECORD KEY\_EVENT\_RECORD MENU\_EVENT\_RECORD MOUSE\_EVENT\_RECORD SMALL\_RECT WINDOW\_BUFFER\_SIZE\_RECORD

## 控制台结构

以下结构用于访问控制台。

CHAR\_INFO CONSOLE\_CURSOR\_INFO CONSOLE\_FONT\_INFO CONSOLE\_FONT\_INFOEX CONSOLE\_HISTORY\_INFO CONSOLE\_READCONSOLE\_CONTROL CONSOLE\_SCREEN\_BUFFER\_INFO CONSOLE\_SCREEN\_BUFFER\_INFOEX CONSOLE\_SELECTION\_INFO COORD FOCUS\_EVENT\_RECORD INPUT\_RECORD KEY\_EVENT\_RECORD MENU\_EVENT\_RECORD MOUSE\_EVENT\_RECORD SMALL\_RECT WINDOW\_BUFFER\_SIZE\_RECORD

Console WinEvents

The following event constants are used in the event parameter of the WinEventProc callback function. For more information, see WinEvents.

Constant/value Description

EVENT\_CONSOLE\_CARET 0x4001

The console caret has moved. The idObject parameter is one or more of the following values: CONSOLE\_CARET\_SELECTION or CONSOLE\_CARET\_VISIBLE.

The idChild parameter is a COORD structure that specifies the cursor's current position.

EVENT\_CONSOLE\_END\_APPLICATION 0x4007

A console process has exited. The idObject parameter contains the process identifier of the terminated process.

EVENT\_CONSOLE\_LAYOUT 0x4005

The console layout has changed.

EVENT\_CONSOLE\_START\_APPLICATION 0x4006

A new console process has started. The idObject parameter contains the process identifier of the newly created process. If the application is a 16-bit application, the idChild parameter is CONSOLE\_APPLICATION\_16BIT and idObject is the process identifier of the NTVDM session associated with the console.

EVENT\_CONSOLE\_UPDATE\_REGION 0x4002

More than one character has changed. The idObject parameter is a COORD structure that specifies the start of the changed region. The idChild parameter is a COORD structure that specifies the end of the changed region.

EVENT\_CONSOLE\_UPDATE\_SCROLL 0x4004

The console has scrolled. The idObject parameter is the horizontal distance the console has scrolled. The idChild parameter is the vertical distance the console has scrolled.

EVENT\_CONSOLE\_UPDATE\_SIMPLE 0x4003

A single character has changed. The idObject parameter is a COORD structure that specifies the character that has changed. The idChild parameter specifies the character in the low word and the character attributes in the high word.

Requirements

Minimum supported client

Windows 2000 Professional [desktop apps only]

Minimum supported server

Windows 2000 Server [desktop apps only]

Header

Winuser.h

## 控制台WinEvents

WinEventProc回调函数的event参数中使用以下事件常量。有关更多信息，请参见WinEvents。

常数/值说明

EVENT\_CONSOLE\_CARET 0x4001

控制台插入符号已移动。 idObject参数是以下值之一或多个：CONSOLE\_CARET\_SELECTION或CONSOLE\_CARET\_VISIBLE。

idChild参数是一个COORD结构，用于指定光标的当前位置。

EVENT\_CONSOLE\_END\_APPLICATION 0x4007

控制台进程已退出。 idObject参数包含终止进程的进程标识符。

EVENT\_CONSOLE\_LAYOUT 0x4005

控制台布局已更改。

EVENT\_CONSOLE\_START\_APPLICATION 0x4006

新的控制台过程已开始。 idObject参数包含新创建的进程的进程标识符。如果应用程序是16位应用程序，则idChild参数为CONSOLE\_APPLICATION\_16BIT，idObject是与控制台关联的NTVDM会话的进程标识符。

EVENT\_CONSOLE\_UPDATE\_REGION 0x4002

更改了多个字符。 idObject参数是一个COORD结构，用于指定更改区域的开始。 idChild参数是一个COORD结构，用于指定更改区域的结尾。

EVENT\_CONSOLE\_UPDATE\_SCROLL 0x4004

控制台已滚动。 idObject参数是控制台滚动的水平距离。 idChild参数是控制台滚动的垂直距离。

EVENT\_CONSOLE\_UPDATE\_SIMPLE 0x4003

单个字符已更改。 idObject参数是一个COORD结构，用于指定已更改的字符。 idChild参数指定低位单词中的字符和高位单词中的字符属性。

要求

最低支持的客户

Windows 2000 Professional [仅桌面应用程序]

最低支持的服务器

Windows 2000 Server [仅桌面应用程序]

标头

Winuser.h

# CreateProcessW函数

2018/12/05

12分钟阅读时长

创建一个新进程及其主线程。新进程在调用进程的安全上下文中运行。

如果呼叫过程正在模拟另一个用户，则新过程将令牌用于呼叫过程，而不是模拟令牌。若要在模拟令牌表示的用户的安全上下文中运行新进程，请使用CreateProcessAsUser或CreateProcessWithLogonW函数。

句法

C ++

BOOL CreateProcessW（

LPCWSTR lpApplicationName，

LPWSTR lpCommandLine，

LPSECURITY\_ATTRIBUTES lpProcessAttributes，

LPSECURITY\_ATTRIBUTES lpThreadAttributes，

BOOL bInheritHandles，

DWORD dwCreationFlags，

LPVOID lpEnvironment，

LPCWSTR lpCurrentDirectory，

LPSTARTUPINFOW lpStartupInfo，

LPPROCESS\_INFORMATION lpProcessInformation

）;

参量

lpApplicationName

要执行的模块的名称。该模块可以是基于Windows的应用程序。如果适当的子系统在本地计算机上可用，则它可以是某种其他类型的模块（例如，MS-DOS或OS / 2）。

该字符串可以指定要执行的模块的完整路径和文件名，也可以指定部分名称。对于部分名称，该函数使用当前驱动器和当前目录来完成规范。该功能将不使用搜索路径。此参数必须包含文件扩展名；不假定默认扩展名。

lpApplicationName参数可以为NULL。在这种情况下，模块名称必须是lpCommandLine字符串中第一个由空格分隔的标记。如果您使用的长文件名包含空格，请使用带引号的字符串指示文件名的结尾和参数的开头；否则，文件名不明确。例如，考虑字符串“ c：\ program files \ sub dir \ program name”。该字符串可以通过多种方式解释。系统尝试按以下顺序解释可能性：

c：\ program.exe c：\ program files \ sub.exe c：\ program files \ sub dir \ program.exe c：\ program files \ sub dir \ program name.exe如果可执行模块是16位应用程序，lpApplicationName应该为NULL，并且lpCommandLine指向的字符串应指定可执行模块及其参数。

要运行批处理文件，必须启动命令解释器。将lpApplicationName设置为cmd.exe，并将lpCommandLine设置为以下参数：/ c加上批处理文件的名称。

lpCommandLine

要执行的命令行。

此字符串的最大长度为32,768个字符，包括Unicode终止的空字符。如果lpApplicationName为NULL，则lpCommandLine的模块名称部分限制为MAX\_PATH字符。

此函数的Unicode版本CreateProcessW可以修改此字符串的内容。因此，此参数不能是指向只读存储器的指针（例如const变量或文字字符串）。如果此参数是常量字符串，则该函数可能会导致访问冲突。

lpCommandLine参数可以为NULL。在这种情况下，该函数将lpApplicationName指向的字符串用作命令行。

如果lpApplicationName和lpCommandLine均为非NULL，则lpApplicationName指向的以Null结尾的字符串指定要执行的模块，而lpCommandLine指向的以Null结尾的字符串指定命令行。新进程可以使用GetCommandLine检索整个命令行。用C编写的控制台进程可以使用argc和argv参数来解析命令行。因为argv [0]是模块名称，所以C程序员通常将模块名称作为命令行中的第一个标记重复。

如果lpApplicationName为NULL，则命令行的第一个用空格分隔的标记指定模块名称。如果使用的长文件名包含空格，请使用带引号的字符串指示文件名的结尾和参数的开头（请参阅lpApplicationName参数的说明）。如果文件名不包含扩展名，则将附加.exe。因此，如果文件扩展名是.com，则此参数必须包含.com扩展名。如果文件名以不带扩展名的句点（。）结尾，或者文件名包含路径，则不会附加.exe。如果文件名不包含目录路径，则系统按照以下顺序搜索可执行文件：

**WriteConsoleOutput function**

* 07/12/2018
* 3 minutes to read

Writes character and color attribute data to a specified rectangular block of character cells in a console screen buffer. The data to be written is taken from a correspondingly sized rectangular block at a specified location in the source buffer.

**Syntax**

C

BOOL WINAPI WriteConsoleOutput(

\_In\_          HANDLE      hConsoleOutput,

\_In\_    const CHAR\_INFO   \*lpBuffer,

\_In\_          COORD       dwBufferSize,

\_In\_          COORD       dwBufferCoord,

\_Inout\_       PSMALL\_RECT lpWriteRegion

);

**Parameters**

*hConsoleOutput* [in]  
A handle to the console screen buffer. The handle must have the **GENERIC\_WRITE** access right. For more information, see [Console Buffer Security and Access Rights](https://docs.microsoft.com/en-us/windows/console/console-buffer-security-and-access-rights).

*lpBuffer* [in]  
The data to be written to the console screen buffer. This pointer is treated as the origin of a two-dimensional array of [**CHAR\_INFO**](https://docs.microsoft.com/en-us/windows/console/char-info-str) structures whose size is specified by the *dwBufferSize* parameter.

*dwBufferSize* [in]  
The size of the buffer pointed to by the *lpBuffer* parameter, in character cells. The **X** member of the [**COORD**](https://docs.microsoft.com/en-us/windows/console/coord-str) structure is the number of columns; the **Y** member is the number of rows.

*dwBufferCoord* [in]  
The coordinates of the upper-left cell in the buffer pointed to by the *lpBuffer* parameter. The **X** member of the [**COORD**](https://docs.microsoft.com/en-us/windows/console/coord-str) structure is the column, and the **Y** member is the row.

*lpWriteRegion* [in, out]  
A pointer to a [**SMALL\_RECT**](https://docs.microsoft.com/en-us/windows/console/small-rect-str) structure. On input, the structure members specify the upper-left and lower-right coordinates of the console screen buffer rectangle to write to. On output, the structure members specify the actual rectangle that was used.

**Return value**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [**GetLastError**](https://msdn.microsoft.com/library/windows/desktop/ms679360).

**Remarks**

**WriteConsoleOutput** treats the source buffer and the destination screen buffer as two-dimensional arrays (columns and rows of character cells). The rectangle pointed to by the *lpWriteRegion* parameter specifies the size and location of the block to be written to in the console screen buffer. A rectangle of the same size is located with its upper-left cell at the coordinates of the *dwBufferCoord* parameter in the *lpBuffer* array. Data from the cells that are in the intersection of this rectangle and the source buffer rectangle (whose dimensions are specified by the *dwBufferSize* parameter) is written to the destination rectangle.

Cells in the destination rectangle whose corresponding source location are outside the boundaries of the source buffer rectangle are left unaffected by the write operation. In other words, these are the cells for which no data is available to be written.

Before **WriteConsoleOutput** returns, it sets the members of *lpWriteRegion* to the actual screen buffer rectangle affected by the write operation. This rectangle reflects the cells in the destination rectangle for which there existed a corresponding cell in the source buffer, because **WriteConsoleOutput** clips the dimensions of the destination rectangle to the boundaries of the console screen buffer.

If the rectangle specified by *lpWriteRegion* lies completely outside the boundaries of the console screen buffer, or if the corresponding rectangle is positioned completely outside the boundaries of the source buffer, no data is written. In this case, the function returns with the members of the structure pointed to by the *lpWriteRegion* parameter set such that the **Right** member is less than the **Left**, or the **Bottom** member is less than the **Top**. To determine the size of the console screen buffer, use the [**GetConsoleScreenBufferInfo**](https://docs.microsoft.com/en-us/windows/console/getconsolescreenbufferinfo) function.

**WriteConsoleOutput** has no effect on the cursor position.

This function uses either Unicode characters or 8-bit characters from the console's current code page. The console's code page defaults initially to the system's OEM code page. To change the console's code page, use the [**SetConsoleCP**](https://docs.microsoft.com/en-us/windows/console/setconsolecp) or [**SetConsoleOutputCP**](https://docs.microsoft.com/en-us/windows/console/setconsoleoutputcp) functions, or use the **chcp** or **mode con cp select=** commands.

**Examples**

For an example, see [Reading and Writing Blocks of Characters and Attributes](https://docs.microsoft.com/en-us/windows/console/reading-and-writing-blocks-of-characters-and-attributes).

**Requirements**

|  |  |
| --- | --- |
| Minimum supported client | Windows 2000 Professional [desktop apps only] |
| Minimum supported server | Windows 2000 Server [desktop apps only] |
| Header | ConsoleApi2.h (via Wincon.h, include Windows.h) |
| Library | Kernel32.lib |
| DLL | Kernel32.dll |
| Unicode and ANSI names | **WriteConsoleOutputW** (Unicode) and **WriteConsoleOutputA** (ANSI) |
|  |  |
|  |  |
|  |  |
|  |  |

**See also**

[Console Functions](https://docs.microsoft.com/en-us/windows/console/console-functions)

[**CHAR\_INFO**](https://docs.microsoft.com/en-us/windows/console/char-info-str)

[**COORD**](https://docs.microsoft.com/en-us/windows/console/coord-str)

[**GetConsoleScreenBufferInfo**](https://docs.microsoft.com/en-us/windows/console/getconsolescreenbufferinfo)

[Low-Level Console Output Functions](https://docs.microsoft.com/en-us/windows/console/low-level-console-output-functions)

[**ReadConsoleOutput**](https://docs.microsoft.com/en-us/windows/console/readconsoleoutput)

[**ReadConsoleOutputAttribute**](https://docs.microsoft.com/en-us/windows/console/readconsoleoutputattribute)

[**ReadConsoleOutputCharacter**](https://docs.microsoft.com/en-us/windows/console/readconsoleoutputcharacter)

[**SetConsoleCP**](https://docs.microsoft.com/en-us/windows/console/setconsolecp)

[**SetConsoleOutputCP**](https://docs.microsoft.com/en-us/windows/console/setconsoleoutputcp)

[**SMALL\_RECT**](https://docs.microsoft.com/en-us/windows/console/small-rect-str)

[**WriteConsoleOutputAttribute**](https://docs.microsoft.com/en-us/windows/console/writeconsoleoutputattribute)

[**WriteConsoleOutputCharacter**](https://docs.microsoft.com/en-us/windows/console/writeconsoleoutputcharacter)

# WriteConsoleOutput函数

2018/07/12

3分钟阅读

将字符和颜色属性数据写入控制台屏幕缓冲区中字符单元格的指定矩形块中。要写入的数据取自源缓冲区中指定位置的相应大小的矩形块。

句法

C

布尔WINAPI WriteConsoleOutput（

\_输入\_ hConsoleOutput，

\_In\_ const CHAR\_INFO \* lpBuffer，

\_In\_ COORD dwBufferSize，

\_In\_ COORD dwBufferCoord，

\_Inout\_ PSMALL\_RECT lpWriteRegion

）;

参量

hConsoleOutput [输入]

控制台屏幕缓冲区的句柄。句柄必须具有GENERIC\_WRITE访问权限。有关更多信息，请参见控制台缓冲区安全性和访问权限。

lpBuffer [输入]

要写入控制台屏幕缓冲区的数据。该指针被视为CHAR\_INFO结构的二维数组的原点，该数组的大小由dwBufferSize参数指定。

dwBufferSize [输入]

lpBuffer参数指向的缓冲区大小，以字符单元格为单位。 COORD结构的X成员是列数； Y成员是行数。

dwBufferCoord [输入]

lpBuffer参数指向的缓冲区中左上角单元的坐标。 COORD结构的X成员是列，Y成员是行。

lpWriteRegion [输入，输出]

指向SMALL\_RECT结构的指针。在输入时，结构成员指定要写入的控制台屏幕缓冲区矩形的左上角和右下角坐标。在输出时，结构成员指定所使用的实际矩形。

返回值

如果函数成功，则返回值为非零。

如果函数失败，则返回值为零。若要获取扩展的错误信息，请调用GetLastError。

备注

WriteConsoleOutput将源缓冲区和目标屏幕缓冲区视为二维数组（字符单元格的行和行）。 lpWriteRegion参数指向的矩形指定要在控制台屏幕缓冲区中写入的块的大小和位置。大小相同的矩形及其左上单元位于lpBuffer数组中dwBufferCoord参数的坐标处。来自此矩形和源缓冲区矩形（其尺寸由dwBufferSize参数指定）相交的单元格中的数据将写入目标矩形。

目标矩形中其相应源位置在源缓冲区矩形的边界之外的单元不受写操作的影响。换句话说，这些是没有可用数据写入的单元。

在WriteConsoleOutput返回之前，它将lpWriteRegion的成员设置为受写操作影响的实际屏幕缓冲区矩形。此矩形反映了目标矩形中的单元格，源缓冲区中存在与其对应的单元格，因为WriteConsoleOutput将目标矩形的尺寸裁剪到控制台屏幕缓冲区的边界。

如果lpWriteRegion指定的矩形完全位于控制台屏幕缓冲区的边界之外，或者如果相应的矩形完全位于源缓冲区的边界之外，则不会写入任何数据。在这种情况下，该函数将返回lpWriteRegion参数集所指向的结构的成员，以使Right成员小于Left，或者Bottom成员小于Top。若要确定控制台屏幕缓冲区的大小，请使用GetConsoleScreenBufferInfo函数。

WriteConsoleOutput对光标位置没有影响。

此功能使用控制台当前代码页中的Unicode字符或8位字符。控制台的代码页最初默认为系统的OEM代码页。要更改控制台的代码页，请使用SetConsoleCP或SetConsoleOutputCP函数，或使用chcp或mode con cp select =命令。

例子

有关示例，请参见读取和写入字符和属性块。

要求

最低支持的客户

Windows 2000 Professional [仅桌面应用程序]

最低支持的服务器

Windows 2000 Server [仅桌面应用程序]

标头

ConsoleApi2.h（通过Wincon.h，包括Windows.h）

图书馆

内核32.lib

动态链接库

内核32.dll

Unicode和ANSI名称

WriteConsoleOutputW（Unicode）和WriteConsoleOutputA（ANSI）

也可以看看

控制台功能

CHAR\_INFO

协调

GetConsoleScreenBufferInfo

底层控制台输出功能

ReadConsoleOutput

ReadConsoleOutputAttribute

ReadConsoleOutputCharacter

SetConsoleCP

SetConsoleOutputCP

SMALL\_RECT

WriteConsoleOutputAttribute

WriteConsoleOutputCharacter

# 欢迎使用Windows终端，控制台和命令行存储库

该存储库包含以下源代码：

Windows终端

Windows控制台主机（conhost.exe）

两个项目之间共享的组件

色彩工具

显示如何使用Windows控制台API的示例项目

相关存储库包括：

控制台API文档

Cascadia代码字体

安装并运行Windows Terminal

👉注意：Windows Terminal需要Windows 10 1903（内部版本18362）或更高版本

从此存储库手动安装构建

对于无法从Microsoft Store安装终端的用户，可以从该存储库的“发行”页面手动下载终端版本。

⚠注意：如果您手动安装Terminal：

确保安装Desktop Bridge VC ++ v14可再发行组件包，否则终端可能无法安装和/或运行，并可能在启动时崩溃

发行新版本时，Terminal不会自动更新，因此您需要定期安装最新的Terminal发行版才能获得所有最新的修复和改进！

通过Chocolatey安装（非官方）

Chocolatey用户可以通过安装microsoft-windows-terminal软件包来下载并安装最新的Terminal版本：

choco安装microsoft-windows-terminal

要使用Chocolatey升级Windows Terminal，请运行以下命令：

choco升级microsoft-windows-terminal

如果您在安装/升级软件包时遇到任何问题，请转到Windows Terminal软件包页面，并按照Chocolatey整理流程进行操作

项目建设状态

项目建设状态

终端构建状态

色彩工具

Windows Terminal v1.0路线图

此处介绍了Windows Terminal v1.0的交付计划，并将在项目进行时进行更新。

终端和控制台概述

在深入研究代码之前，请花几分钟查看以下概述：

## Windows终端

Windows Terminal是一个新的，现代的，功能丰富的，高效的终端应用程序，适用于命令行用户。它包含Windows命令行社区最常请求的许多功能，包括对选项卡，富文本，全球化，可配置性，主题和样式等的支持。

终端还需要满足我们的目标和措施，以确保它保持快速和高效，并且不会消耗大量内存或电量。

Windows控制台主机

Windows控制台主机conhost.exe是Windows的原始命令行用户体验。它还托管Windows的命令行基础结构和Windows控制台API服务器，输入引擎，渲染引擎，用户首选项等。此存储库中的控制台主机代码是构建Windows本身conhost.exe的实际源。

自2014年获得Windows命令行所有权以来，该团队向控制台添加了一些新功能，包括背景透明性，基于行的选择，对ANSI /虚拟终端序列的支持，24位颜色，伪控制台（“ ConPTY” ）， 和更多。

但是，由于Windows控制台的主要目标是保持向后兼容性，因此我们无法添加社区（和团队）最近几年一直想要的许多功能，包括选项卡，unicode文本和表情符号。

这些限制导致我们创建了新的Windows Terminal。

您可以在命令行团队博客中的此系列博客文章中，全面了解命令行的发展，特别是Windows命令行。

共享组件

在检修Windows Console的同时，我们对代码库进行了相当大的现代化，将逻辑实体干净地划分为模块和类，引入了一些关键的可扩展性点，用更安全，更高效的STL容器替换了一些旧的本地集合和容器，并使代码更简单，使用Microsoft的Windows实施库-WIL更安全。

这项大修导致Console的几个关键组件可在Windows的任何终端实现中重复使用。这些组件包括新的基于DirectWrite的文本布局和呈现引擎，能够存储UTF-16和UTF-8的文本缓冲区，VT解析器/发射器等。

创建新的Windows终端

当我们开始计划新的Windows Terminal应用程序时，我们探索并评估了几种方法和技术堆栈。我们最终决定，通过继续对C ++代码库进行投资，可以最好地实现我们的目标，这将使我们能够在现有控制台和新终端中重用上述多个现代化组件。此外，我们意识到这将使我们能够将终端的大部分核心本身构建为可重用的UI控件，其他人可以将其合并到自己的应用程序中。

这项工作的结果包含在此存储库中，并作为Windows终端应用程序提供，您可以从Microsoft Store中下载，也可以直接从此存储库的版本中下载。

资源资源

有关Windows终端的更多信息，您可能会发现其中一些有用和有趣的资源：

命令行博客

命令行背景知识博客系列

Windows终端启动：终端“嘶嘶声视频”

Windows Terminal启动：Build 2019会议

以电台形式运行：Show 645-与Richard Turner一起使用的Windows终端

Azure Devops播客：第54集-Windows终端上DevOps上的Kayla肉桂和Rich Turner

Microsoft Ignite 2019会议：现代Windows命令行：Windows终端-BRK3321

常问问题

我构建并运行了新的终端，但它看起来就像旧的控制台

原因：您在Visual Studio中启动了不正确的解决方案。

解决方案：确保在Visual Studio中构建和部署CascadiaPackage项目。

⚠注意：OpenConsole.exe只是本地构建的conhost.exe，它是承载Windows命令行基础结构的经典Windows控制台。 Windows终端使用OpenConsole连接到命令行应用程序并与之通信（通过ConPty）。

文献资料

所有项目文档都位于./doc文件夹中。如果您想为文档做出贡献，请提交请求请求。

贡献

我们很高兴与您（我们的社区）共同构建和增强Windows Terminal！

在开始使用功能/修复之前，请阅读并遵循我们的《贡献者指南》，以帮助避免任何浪费或重复的工作。

与团队沟通

与团队沟通的最简单方法是通过GitHub问题。

请提出新的问题，功能要求和建议，但是在创建新问题之前，请先搜索类似的已打开/已关闭的现有问题。

如果您想提出一个您认为尚不存在问题的问题，请通过Twitter与我们联系：

程序经理Kayla Cinnamon：@cinnamon\_msft

Rich Turner，项目经理：@richturn\_ms

Dustin Howett，工程主管：@dhowett

Michael Niksa，高级开发人员：@michaelniksa

Mike Griese，开发人员：@zadjii

Carlos Zamora，开发人员：@cazamor\_msft

Leon Liang，开发人员：@leonmsft

开发人员指南

先决条件

您必须运行Windows 1903（内部版本== 10.0.18362.0）或更高版本才能运行Windows终端

您必须在Windows设置应用程序中启用开发人员模式才能本地安装和运行Windows终端

您必须安装Windows 10 1903 SDK

您必须至少安装VS 2019

您必须通过VS Installer安装以下工作负载。注意：在VS 2019中打开解决方案将提示您自动安装缺少的组件：

使用C ++进行桌面开发

通用Windows平台开发

以下各个组成部分

C ++（v142）通用Windows平台工具

建立代码

该存储库使用git子模块为其某些依赖项。为确保子模块已还原或更新，请确保在构建之前运行以下命令：

git子模块更新--init --recursive

OpenConsole.sln可以使用/ tools目录中的一组便捷脚本和工具从Visual Studio或命令行中构建：

在PowerShell中构建

导入模块。\ tools \ OpenConsole.psm1

设置MsBuildDevEnvironment

调用OpenConsoleBuild

在Cmd中建立

。\ tools \ razzle.cmd

bcz

运行与调试

要在VS中调试Windows终端，请右键单击CascadiaPackage（在解决方案资源管理器中），然后转到属性。在“调试”菜单中，将“应用程序过程”和“背景任务过程”更改为“仅限本机”。

然后，您应该可以通过按F5键来构建和调试Terminal项目。

👉您将无法通过运行WindowsTerminal.exe直接启动终端。有关原因的更多详细信息，请参见＃926，＃4043

编码指导

请阅读以下有关我们编码实践的简短文档。

👉如果您发现这些文档中缺少某些内容，请随时在存储库中的任何位置为我们的任何文档文件做出贡献（或编写一些新文件！）

这是一项正在进行的工作，因为我们了解了为成为我们的项目的有效贡献者而需要提供的人员。

编码风格

代码组织

旧版代码库中的异常

在WIL中与Windows交互的有用的智能指针和宏

行为守则

该项目采用了Microsoft开源行为准则。有关更多信息，请参见《行为准则》常见问题解答，或与opencode@microsoft.com联系，并提出其他任何问题或意见。