2020/07/28_Windows编程_第8课_进程间的通讯_管道

笔记本:	Windows编程
创建时间:	2020/7/28 星期二 10:58
作者:	ileemi
标签:	管道

进程间的通讯——管道

管道(Pipe)

一般不需要发送消息，创建管道（服务器Server）,使用管道（客户端Client）操作和文件操作类似（管道两侧一个发送数据一个读取数据）,但是不会将数据写入到文件。

管道的使用方式有三种：

1. 命名管道
2. 匿名管道 -- 类似内存映射 （需要继承句柄有父子继承关系或者拷贝句柄）
3. 重定位（匿名管道）-- 通过管道去控制一个软件的输入和输出

命名管道

本质上管道内存数据也是通过共享内存实现的，其跨进程通讯要比其它方法简单，数据格式比较灵活。不需要提前通知。使用起来较为方便。

> API:
> `CreateNamedPipe` -- 创建命名管道的实例，并返回后续管道操作的句柄。
> `ConnectNamedPipe` -- 允许指定管道服务器进程等待客户端进程连接到指定管道的实例。客户端进程通过调用 `CreateFile` 或 `CallNamedPipe` 函数进行连接。
> `WaitNamedPipe` -- 一直等待，直到超时时间结束，或者指定的命名管道的实例可用来连接(也就是说，管道的服务器进程在管道上有一个挂起的 `ConnectNamedPipe` 操作)。

管道操作和文件操作没有太多的区别，但是管道不会写文件。使用管道时，一端发送数据，另一端接收数据，不能同时发送数据（会导致管道堵塞）。

服务器一般接收数据（读取数据），客户端发送数据。使用管道发送数据不需要进行通知，管道中有数据就接收，没有就不进行接收。

服务器

代码示例：

```
// Server.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
#include <iostream>
#include <stdio.h>
#include <Windows.h>

// 为管道定义一个名字，需要转义字符
#define PIPE_NAME "\\\\.\\pipe\\Test "

// 服务器，客户端 1. 命名管道 2. 匿名管道 3. 重定位（匿名管道）
int main()
{
    // 创建命名管道
    HANDLE hPipe = CreateNamedPipe(
        PIPE_NAME,     // 命名管道名
        PIPE_ACCESS_DUPLEX,  // 命名管道打开模式
        PIPE_TYPE_BYTE,          // 管道模式
        1,           // 命名管道的最大连接数量
        1024,     // 命名管道最大读取字节数
        1024,     // 命名管道最大写入字节数
        1000,     // 命名管道连接检查时间
        NULL       // 安全属性
  );
    // 检查命名管道是否创建成功
    if (hPipe == INVALID_HANDLE_VALUE)
    {
        printf("CreateNamedPipe Error! \r\n");
        return 0;
    }

    printf("CreateNamedPipe OK! \r\n");

    // 连接命名管道 -- 等待连接
    printf("Wait Connect...\r\n");
    if (!ConnectNamedPipe(hPipe, NULL))
    {
        // 管道连接失败
        printf("ConnectNamedPipe Error! \r\n");
        return 0;
```

```
    }
    printf("Connect  OK! \r\n");

    //  管道连接成功，开始接收客户端发送过来的数据

    //  存储从管道中读取到客户端发送过来的数据
    CHAR  chRequest[1024];
    DWORD  cbBytesRead  =  0;

    //  保存向客户端回复的数据
    CHAR  chBuff[1024];
    DWORD  cbWritten  =  0;
    while  (true)
    {
        if  (!ReadFile(hPipe,  chRequest,  sizeof(chRequest),  &cbBytesRead,

        {
            printf("ReadFile  End\r\n");
            break;
        }
      //  将读取到的数据进行显示
        chRequest[cbBytesRead]  =  '\0';
        printf("客户端：写入的数据：  %s,  数据的大
小：  %d\r\n",  chRequest,  cbBytesRead);
        //  向客户端回复读取数据成功
        printf("CMD:");
        scanf("%s",  chBuff);
        if  (!WriteFile(hPipe,  chBuff,  strlen(chBuff),  &cbWritten,  NULL)

        {
            break;
        }
    }
    return  0;
}
```

客户端

等待指定的命名管道

> API:
> WaitNamedPipe 、CreateFile

代码示例：

```cpp
// Client.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
// 客户端
#include <iostream>
#include <Windows.h>

// 为管道定义一个名字，需要转义字符
#define PIPE_NAME "\\\\.\\pipe\\Test "

int main()
{
    // 连接管道（连接服务器），等待时间为10秒
    if (WaitNamedPipe(PIPE_NAME, 10000))
    {
        printf("Could not open pipe\r\n");
    }

    // 打开指定的管道文件
    HANDLE hPipe = CreateFile(
        PIPE_NAME,    // pipe name
        GENERIC_READ | GENERIC_WRITE, // read and write access
        0,         // no sharing
        NULL, // no security attributes
        OPEN_EXISTING,    // opens existing pipe
        0,         // default attributes
        NULL    // no template file
    );
    if (hPipe == INVALID_HANDLE_VALUE)
    {
        printf("CreateFile Error!\r\n");
        return 0;
    }

    // 保存向服务器发送数据
    CHAR chBuff[1024];
    DWORD cbWritten = 0;

    // 保存从命名管道中读取到的数据
    CHAR chRequest[1024];
    DWORD cbBytesRead = 0;
    while (true)
    {
        printf("CMD:");
        scanf("%s", chBuff);
        if (!WriteFile(hPipe, chBuff, strlen(chBuff), &cbWritten, NULL)
```

```
            {
                break;
            }

            //  显示发送的数据内容
            printf("发送数据为：%s,  发送数据的字节数：
%d\r\n",  chBuff,  cbWritten);

            //  同步读取  --  阻塞，异步读  --  非阻塞
            if  (!ReadFile(hPipe,  chRequest,  sizeof(chRequest),  &cbBytesRead,

            {
                printf("ReadFile  End\r\n");
                break;
            }

            //  将读取到的数据显示出来
            chRequest[cbBytesRead]  =  '\0';
            printf("服务器：返回的数据：  %s\n",  chRequest);
        }
        return  0;
}
```

重定位输入输出缓冲区（匿名管道）

> 创建进程句柄，创建管道的时候，允许子进程继承父进程，创建进程的时候允许子进程去继承。

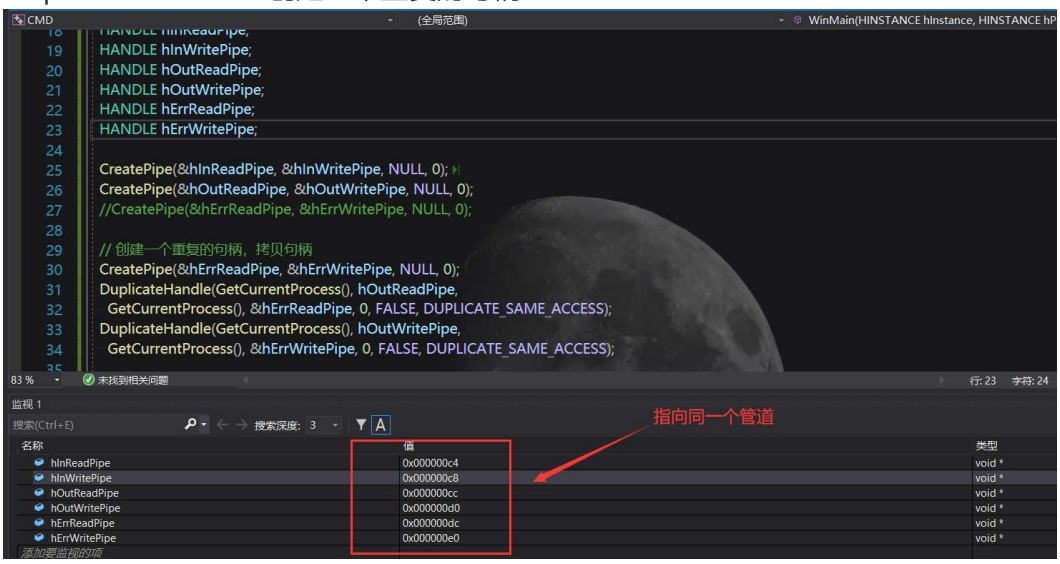
只有自己创建的程序才能够重定位该程序的输入输出缓冲区（需要有父子关系）。

STARTUPINFO 结构体最后的三个参数分别为标准输入的句柄，标准输出的句柄，标准错误的句柄：

```
typedef struct _STARTUPINFO {
    DWORD   cb;
    LPTSTR  lpReserved;
    LPTSTR  lpDesktop;
    LPTSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

通常情况下标准错误设备和标准输出设置是同一种设备。
Windows 操作系统允许将一个标准设备绑定到一个句柄上，可以创建一个管道，让标准输入设置去管道中读取数据。用管道取代输入输出缓冲区。

GetStdHandle -- 为标准输入、标准输出或标准错误设备检索句柄
CreatePipe -- 创建一个匿名管道，并将句柄返回到管道的读端和写端
DuplicateHandle -- 创建一个重复的句柄



用的不多，但是病毒最喜欢用这种手法。

代码示例：

```
// 窗口版
#include <iostream>
#include <Windows.h>
#include <strsafe.h>
#define BUFSIZE 4096*2

void WriteToPipe(void);
void ReadFromPipe(void);
void ErrorExit(PTSTR lpszFunction);

// 创建管道，分别绑定标准输入和标准输出以及标准错误
HANDLE g_hInReadPipe = NULL;
HANDLE g_hInWritePipe = NULL;
HANDLE g_hOutReadPipe = NULL;
HANDLE g_hOutWritePipe = NULL;
HANDLE g_hInputFile = NULL;

char g_szReadData[BUFSIZE] = { 0 };
char g_szWriteData[BUFSIZE] = { 0 };
bool g_bFlag = false;

int  main()
{
  // 打开一个已存在的进程
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  // 创建进程句柄
```

```
SECURITY_ATTRIBUTES saAttr;
saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;

// 创建管道的时候，允许子进程继承父进程
if (!CreatePipe(&g_hInReadPipe, &g_hInWritePipe, &saAttr, 0))
{
  puts("StdoutRd CreatePipe");
}
if (!CreatePipe(&g_hOutReadPipe, &g_hOutWritePipe, &saAttr, 0))
{
  puts("StdoutRd CreatePipe");
}


// 设置打开进程的窗口属性
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
// 设置的属性需要打开对应的标志、
// 系统通过该标志确定结构体的成员的有效性
si.dwFlags = STARTF_USEPOSITION |
  STARTF_USESIZE |
  STARTF_USESHOWWINDOW |
  STARTF_USESTDHANDLES;
si.dwX = 50; // 创建或者打开进程窗口的左上角X轴坐标
si.dwY = 50; // 创建或者打开进程窗口的左上角Y轴坐标
si.dwXSize = 200; // 创建或者打开进程窗口的宽度
si.dwYSize = 100; // 创建或者打开进程窗口的高度
si.wShowWindow = SW_SHOWNORMAL; // 窗口显示的方式

// 标准输入/输出/错误句柄
// Windows 操作系统允许将一个标准设备绑定到一个句柄上
// 可以创建一个管道，让标准输入设置去管道中读取数据
si.hStdInput = g_hInReadPipe;  // 重定位
si.hStdOutput = g_hOutWritePipe;
si.hStdError = g_hOutWritePipe;

ZeroMemory(&pi, sizeof(pi));

// Start the child process.
if (!CreateProcess("C:\\Windows\\System32\\cmd.exe",
  NULL,
  NULL,
  NULL,
  TRUE,  //允许继承
  0,
```

```
        NULL,
        NULL,
        &si, &pi))
    {
        puts("CreateProcess failed.");
    }
    puts("CreateProcess OK.");

    // 从缓冲区中读入数据
    ReadFromPipe();
    ReadFromPipe();

    Sleep(100);

    while (true)
    {
        // 输入数据到缓冲区
        WriteToPipe();
        Sleep(1000);

        // 从缓冲区中读入数据
        ReadFromPipe();
    }

    // 关闭句柄
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    CloseHandle(g_hOutWritePipe);
    CloseHandle(g_hInReadPipe);

    return 0;
}

// 往管道写入数据
void WriteToPipe(void)
{
    DWORD dwWritten = 0;
    CHAR chBuff[BUFSIZE] = { 0 };
    BOOL bSuccess = FALSE;
    printf("cmd:");
    fgets(chBuff, sizeof(chBuff), stdin);
    fflush(stdin);
    if (strcmp(g_szWriteData, chBuff) == 0)
    {
        g_bFlag = true;
        return;
```

```
    }
    bSuccess = WriteFile(g_hInWritePipe, chBuff, strlen(chBuff),
&dwWritten, NULL);
    memset(g_szWriteData, 0, BUFSIZE);
    memcpy(g_szWriteData, chBuff, strlen(chBuff));
    g_bFlag = false;
    if (!bSuccess || dwWritten == 0)
    {
      puts("数据写入失败");
      return;
    }
}

// 往管道读取数据
void ReadFromPipe(void)
{
    DWORD dwRead;
    CHAR chBuff[BUFSIZE] = { 0 };
    BOOL bSuccess = FALSE;
    HANDLE hParentStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    if (g_bFlag)
    {
      memcpy(chBuff, g_szReadData, strlen(g_szReadData));
      puts(chBuff);
      return;
    }
    bSuccess = ReadFile(g_hOutReadPipe, chBuff, BUFSIZE, &dwRead, NULL);
    memcpy(g_szReadData, chBuff, strlen(chBuff));
    if (!bSuccess || dwRead == 0)
    {
      return;
    }
    chBuff[dwRead] = '\0';
    puts(chBuff);
}

// 输出错误信息
void ErrorExit(PTSTR lpszFunction)
{
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

    FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_FROM_SYSTEM |
      FORMAT_MESSAGE_IGNORE_INSERTS,
```

```
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&lpMsgBuf,
        0, NULL);

    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR)lpMsgBuf) + lstrlen((LPCTSTR)lpszFunction) + 40) *
sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(TCHAR),
        TEXT("%s failed with error %d: %s"),
        lpszFunction, dw, lpMsgBuf);
    puts((char*)lpDisplayBuf);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
    ExitProcess(1);
}
```

> 微软示例