

2020/07/24_Windows编程_第6课_进程间的操作

笔记本: Windows编程
创建时间: 2020/7/24 星期五 10:04
作者: ileemi
标签: 进程间的操作

- [进程间的操作](#)
 - [进程间的句柄](#)
 - [句柄继承](#)
 - [文件句柄继承](#)
 - [拷贝句柄](#)
 - [进程间的操作](#)
 - [进程内存的读取与写入](#)
 - [随机基址, 固定基址下的数据访问](#)
- [飞机游戏实战](#)

模块大小写十六进制

窗口类型也分 Unicode 和 Ansic

获取窗口过程函数 -- GetWindowLong(GetWindowLongA/GetWindowLongW) -
-GetClassLong 这两个API分版本 (32/64 (ptr))

进程间的操作

通过一个进程操控另外一个进程，在操作系统中的每个进程都是独立的。

进程间的句柄

窗口句柄是个例外，所有进程都可以操作，也可以跨进程操作，窗口句柄是全局的，窗口句柄一般不需要打开。而有些句柄是需要打开的。

由于每个进程的句柄是不共享的，操作系统在设计的时候，为每个进程都有一个句柄表。

当需要将一个进程和句柄共享给另外一个进程的时候，有两种方法：

1. 通过继承（将父进程的句柄继承给父进程的子进程）
2. 通过拷贝的方式（将一个进程的句柄拷贝到另一个进程）

句柄继承

子程序继承父进程，并使用父进程的句柄（父进程的句柄以命令参数传递给子进程）

父进程代码示例：

```
#include <iostream>
#include <Windows.h>

int main()
{
    // 进程间的操作
    // 获取进程的句柄(并设置为可以继承)
    HANDLE hNotepad = OpenProcess(PROCESS_ALL_ACCESS, TRUE, 16080);
    printf("hNotepad = %p\r\n", hNotepad);

    STARTUPINFO si = { 0 };
    si.cb = sizeof(STARTUPINFO);
    PROCESS_INFORMATION pi;

    // 继承句柄需要定义结构体 — SECURITY_ATTRIBUTES
    SECURITY_ATTRIBUTES sa = { 0 };
    sa.bInheritHandle = TRUE;
    sa.nLength = sizeof(sa);

    // 将进程句柄做为命令行参数进行传递给子进程使用
    char szBuff[MAXBYTE];
    sprintf_s(szBuff, " %p", hNotepad);

    char szProcessPath[] =
        "D:\\CR37\\Works\\DayCode\\ChildProcess\\Debug\\ChildProcess.exe";
    // 创建进程 (同时允许子进程继承使用父进程的进程句柄)
    if (!CreateProcess(szProcessPath, szBuff, &sa, NULL, TRUE, 0, NULL,
        NULL, &si, &pi))
    {
        printf("CreateProcess failed.");
    }

    system("pause");
    // 通过进程句柄结束进程
    // TerminateProcess(hNotepad, 0);
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    CloseHandle(hNotepad);
    return 0;
}
```

子进程代码示例：

```

#include <iostream>
#include <Windows.h>

int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        return 0;
    }

    // 将父进程传递的命令行参数进行格式化
    HANDLE hNotepad = (HANDLE)strtoul(argv[1], NULL, 16);
    printf("Child hNotepad = %p\r\n", hNotepad);

    system("pause");
    // 终止记事本的进程
    TerminateProcess(hNotepad, 0);
    return 0;
}

```

运行父进程后，可以通过子进程关闭记事本。

文件句柄继承

`CreateFile`

参数4 是一个结构体 "SECURITY_ATTRIBUTES", 在使用 CreateFile 需要继承的时候就需要使用参数4这个结构体。

窗口句柄不需要继承，C库不存在句柄。

拷贝句柄

`DuplicateToKey` -- 创建了一个新的访问令牌，该令牌重复了一个已经存在的令牌

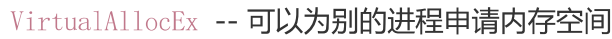
`DuplicateHandle` -- (拷贝句柄到指定的进程)函数复制一个对象句柄。重复句柄引用与原始句柄相同的对象。因此，对对象的任何更改都通过这两个句柄反映。例如，对于两个句柄来说，文件句柄的当前文件标记总是相同的（**比继承句柄好用，不限制进程间的父子关系，不需要其它进程同意**）。

进程间的操作

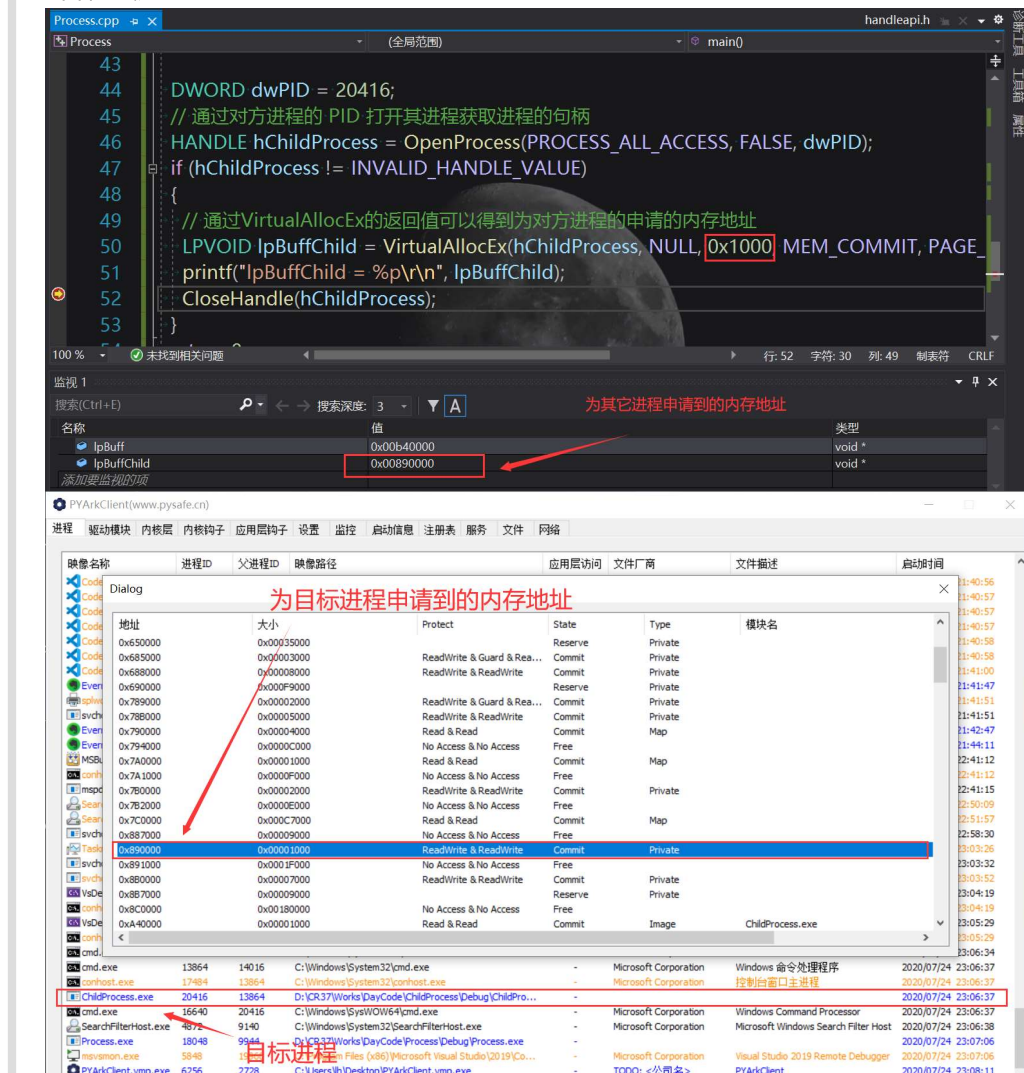
获取当前进程的ID -- `GetCurrentProcessID`

内存保护属性：可读，可写，可执行

返回一个申请到的缓冲区的首地址：



操作示例：



VirtualFreeEx -- 可以释放、解压指定进程的虚拟地址空间中的内存区域

```
44 DWORD dwPID = 20416;
45 // 通过对方进程的 PID 打开其进程获取进程的句柄
46 HANDLE hChildProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
47 if (hChildProcess != INVALID_HANDLE_VALUE)
48 {
49     // 通过VirtualAllocEx的返回值可以得到为对方进程的申请的内存地址
50     LPVOID lpBuffChild = VirtualAllocEx(hChildProcess, NULL, 0x1000, MEM_COMMIT, PAGE_READWRITE);
51     printf("申请到的内存地址为: %p\r\n", lpBuffChild);
52     if (VirtualFreeEx(hChildProcess, lpBuffChild, 0, MEM_RELEASE))
53     {
54         printf("Free Ok\r\n");
55     }
56     else
57     {
```

Dialog 在没有VirtualFreeEx之前，申请到的内存地址状态

地址	大小	Protect	State	Type	模块名
0x7B0000	0x00002000	ReadWrite	Commit	Private	
0x7B2000	0x0000E000	No Access	Free		
0x7C0000	0x000C7000	Read	Commit	Map	
0x887000	0x00009000	No Access	Free		
0x890000	0x00001000	ReadWrite	Commit	Private	
0x891000	0x0000F000	No Access	Free		
0x8A0000	0x00001000	ReadWrite	Commit	Private	
0x8A1000	0x0000F000	No Access	Free		
0x8B0000	0x00007000	ReadWrite	Commit	Private	
0x8B7000	0x00009000	ReadWrite	Reserve	Private	
0x8C0000	0x00001000	ReadWrite	Commit	Private	
0x8C1000	0x0017F000	No Access	Free		
0xA40000	0x00001000	Read	Commit	Image	ChildProcess.exe
0xA41000	0x00001000	WriteCopyExecute	Commit	Image	ChildProcess.exe
0xA51000	0x00006000	ReadExecute	Commit	Image	ChildProcess.exe
0xA57000	0x00003000	Read	Commit	Image	ChildProcess.exe
0xA5A000	0x00001000	ReadWrite	Commit	Image	ChildProcess.exe
0xA5B000	0x00001000	Read	Commit	Image	ChildProcess.exe
0xA5C000	0x00001000	WriteCopy	Commit	Image	ChildProcess.exe
0xA5D000	0x00003000	Read	Commit	Image	ChildProcess.exe

Dialog 在VirtualFreeEx之后，申请到的内存地址状态已被释放

地址	大小	Protect	State	Type	模块名
0x8B0000	0x00007000	ReadWrite	Commit	Private	
0x8B7000	0x00009000	ReadWrite	Reserve	Private	
0x8C0000	0x00180000	No Access	Free		
0xA40000	0x00001000	Read	Commit	Image	ChildProcess.exe
0xA41000	0x00001000	WriteCopyExecute	Commit	Image	ChildProcess.exe
0xA51000	0x00006000	ReadExecute	Commit	Image	ChildProcess.exe
0xA57000	0x00003000	Read	Commit	Image	ChildProcess.exe

示例代码：

```
#include <iostream>
#include <Windows.h>

int main()
{
    // 指定申请的内存位置只限于低2G 参数1为NULL，
    // 申请内存的位置有操作系统自动找位置

    LPVOID lpBuff = VirtualAlloc((LPVOID)NULL, 0x1000, MEM_COMMIT,
    PAGE_READWRITE);

    DWORD dwPID = 20416;
    // 通过对方进程的 PID 打开其进程获取进程的句柄
    HANDLE hChildProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
    if (hChildProcess != INVALID_HANDLE_VALUE)
    {
        // 通过VirtualAllocEx的返回值可以得到为对方进程的申请的内存地址
        LPVOID lpBuffChild = VirtualAllocEx(hChildProcess, NULL, 0x1000,
        MEM_COMMIT, PAGE_READWRITE);
        printf("申请到的内存地址为: %p\r\n", lpBuffChild);
        if (VirtualFreeEx(hChildProcess, lpBuffChild, 0, MEM_RELEASE))
        {
```

```

        printf("Free Ok\r\n");
    }
    else
    {
        printf("Free Error\r\n");
    }

    CloseHandle(hChildProcess);
}

return 0;
}

```

VirtualProtectEx -- 修改进程（目标进程）保护属性（将目标进程原来内存的可读可写属性修改为只读等）

进程内存的读取与写入

为申请到的内存可以写入数据，去读进程内存，写入数据到进程内存中。

ReadProcessMemory -- 从指定进程中的内存区域读取数据。要读取的整个区域必须是可访问的，否则操作将失败。

WriteProcessMemory -- 将数据写入指定进程中的内存区域。要写入的整个区域必须是可访问的，否则操作将失败。

Platform SDK: Debugging and Error Handling

Debugging Functions

The following functions are used with debugging.

Function	Description
ContinueDebugEvent	Enables a debugger to continue a thread that previously reported a debugging event.
DebugActiveProcess	Enables a debugger to attach to an active process and debug it.
DebugActiveProcessStop	Stops the debugger from debugging the specified process.
DebugBreak	Causes a breakpoint exception to occur in the current process.
DebugBreakProcess	Causes a breakpoint exception to occur in the specified process.
DebugSetProcessKillOnExit	Sets the action to be performed when the debugging thread exits.
FatalExit	Transfers execution control to the debugger.
FlushInstructionCache	Flushes the instruction cache for the specified process.
GetThreadContext	Retrieves the context of the specified thread.
GetThreadSelectorEntry	Retrieves a descriptor table entry for the specified selector and thread.
IsDebuggerPresent	Determines whether the calling process is running under the context of a debugger.
OutputDebugString	Sends a string to the debugger for display.
ReadProcessMemory	Reads data from an area of memory in a specified process.
SetThreadContext	Sets the context for the specified thread.
WaitForDebugEvent	Waits for a debugging event to occur in a process being debugged.
WriteProcessMemory	Writes data to an area of memory in a specified process.

通过 **ReadProcessMemory** 和 **WriteProcessMemory** 在知道目标进程重要模块内存地址的情况下，可以操控其内存。

读取目标进程中的局部变量，并尝试修改，操作结果如下：

```

Microsoft Visual Studio 调试控制台
申请到的内存地址为: lpBuffChild = 00640000
读取成功: ChildnProcNum = 10, ProcdwBytes = 4
写入成功: ChildnProcNum = 999, ProcdwBytes = 4

D:\CR37\Works\第二阶段\Windows编程\Codes\20200724 - 进程间的操作_修改游戏数据\Process\Debug\Process.exe (进程
16832) 已退出, 代码为 0。
按任意键关闭此窗口。 . . .

```




代码示例:

```
#include <iostream>
#include <Windows.h>

int main()
{
    // 指定申请的内存位置只限于低2G 参数1为NULL,
    // 申请内存的位置有操作系统自动找位置
    LPVOID lpBuff = VirtualAlloc((LPVOID)NULL, 0x1000, MEM_COMMIT,
    PAGE_READWRITE);

    // 存储目标进程的 PID
    DWORD dwPID = 476;

    // 通过对方进程的 PID 打开其进程获取进程的句柄
    HANDLE hChildProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
    if (hChildProcess != INVALID_HANDLE_VALUE)
    {
        // 通过VirtualAllocEx的返回值可以得到为对方进程的申请的内存地址
        LPVOID lpBuffChild = VirtualAllocEx(hChildProcess, NULL, 0x1000,
        MEM_COMMIT, PAGE_READWRITE);
        printf("申请到的内存地址为: lpBuffChild = %p\r\n", lpBuffChild);

        // 读取进程的内存
        int nNum = 0;
        SIZE_T dwBytes; // 保存读取到的字节数
        if (ReadProcessMemory(hChildProcess, (LPVOID)0x0053FA7C,
            &nNum, sizeof(nNum), &dwBytes))
        {
            printf("读取成功: ChildnProcNum = %d, ProcdwBytes = %d\r\n", nNum,
            dwBytes);
        }
        else
        {
            printf("ReadProcessMemory Error");
        }
        nNum = 999;
        if (WriteProcessMemory(hChildProcess, (LPVOID)0x0053FA7C,
            &nNum, sizeof(nNum), &dwBytes))
        {
            printf("写入成功: ChildnProcNum = %d, ProcdwBytes = %d\r\n", nNum,
            dwBytes);
        }
    }
}
```

```

else
{
    printf("WriteProcessMemory Error");
}
CloseHandle(hChildProcess);
}
return 0;
}

```

```

#include <iostream>
#include <Windows.h>

int main(int argc, char* argv[])
{
    int nNum = 10;
    printf("ChildProcess PID: %d nNum = %p\r\n",
        GetCurrentProcessId(), &nNum);

    system("pause");
    printf("nNum = %d\r\n", nNum);
    system("pause");
    return 0;
}

```

随机基址，固定基址下的数据访问

变量的分类：

1. 局部变量
2. 全局变量
3. 堆变量

高版本系统下有随即基址（为了保证软件的安全），全局变量的地址是固定的

不关闭随机基址：

```

&g_nNum = 00C8A000, ChildProcess_nNum = 666,
&ChildProcess_nNum = 00DCF8D4, &pHeap = 01314FD8
&g_nNum = 00C8A000, ChildProcess_nNum = 666,
&ChildProcess_nNum = 004FF9AC, &pHeap = 007C5078
&g_nNum = 00C8A000, ChildProcess_nNum = 666,
&ChildProcess_nNum = 012FFE7C, &pHeap = 01655078

```

不关闭随机基址, 全局变量的地址固定，局部变量的地址是不固定的，堆地址也是不固定的

关闭随机基址:

```
&g_nNum = 0041A000, ChildProcess_nNum = 666,  
&ChildProcess_nNum = 0019FECC, &pHeap = 007B4FD8  
&g_nNum = 0041A000, ChildProcess_nNum = 666,  
&ChildProcess_nNum = 0019FECC, &pHeap = 00645898  
&g_nNum = 0041A000, ChildProcess_nNum = 666,  
&ChildProcess_nNum = 0019FECC, &pHeap = 00565078
```

关闭随机基址后，全局变量的地址固定，局部地址是固定的，不会变化的原因是，局部变量在main函数内（申请局部变量的时机是确定的），当局部变量在其它函数内部且不确定函数何时调用的时候局部变量的地址就不确定。堆地址在关闭随机地址以及不关闭随机基址都会变化。

随机基址的方式：将主模块当dll使用（dll加载到内存的地址也是不固定的），主程序.exe的地址是固定，因为它是第一个被加载的模块，在内存中的位置是固定的。

开启随机基址后，地址会一直变化，但是在文件中的偏移是固定的，可以使用主模块的基址 + 模块在内存中的偏移（BASE+OFFSET）

开启随机基址后，main函数的地址也是不固定的：

```
&main = 003A12C6  
&main = 006312C6  
&main = 005912C6
```

获取模块的地址(不固定): `GetModuleHandle(NULL)`

模块在内存中的偏移(不会变)，比如main函数在内存中的偏移，可以通过main函数在内存中地址减去模块的地址

```
int offset = (int)main - (int)GetModuleHandle(NULL);
```

全局变量的在内存中偏移也可以这样算：

```
(int)&g_nNum - (int)GetModuleHandle(NULL)
```

第一次测试：

```
&main = 006C12C6  
&pHandle = 006B0000, mainoffset = 000112C6  
&g_nNum = 006CA000  
&pHandle = 006B0000, g_nNumoffset = 0001A000
```

第二次测试：

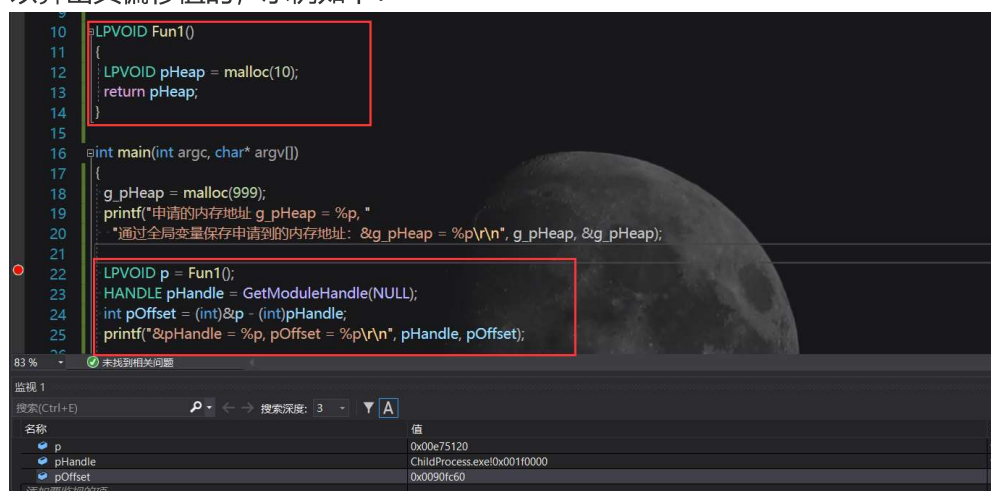
```
&main = 00E712C6  
&pHandle = 00E60000, mainoffset = 000112C6  
&g_nNum = 00E7A000  
&pHandle = 00E60000, g_nNumoffset = 0001A000
```

当程序的核心数据在堆中的时候：

通过堆申请到的内存会保存到一个变量（可以是全局的，也可以是局部的）中：

1. 通过获取全局变量的地址，取出全局变量地址上的地址，就可以访问到通过堆申请到的内存：
申请的内存地址 `g_pHeap = 01738FA8`, 通过全局变量保存申请到的内存地址：
`&g_pHeap = 00AAA14C`

2. 通过局部变量存储堆申请到的内存（局部变量的定义不确定），考虑到重要数据是长期使用的，推断不会保存到临时函数中，猜测会保存到一个可以长期使用的函数中，但是还会有一个局部变量去使用这个函数的返回值，局部变量我们是可计算出其偏移值的，示例如下：



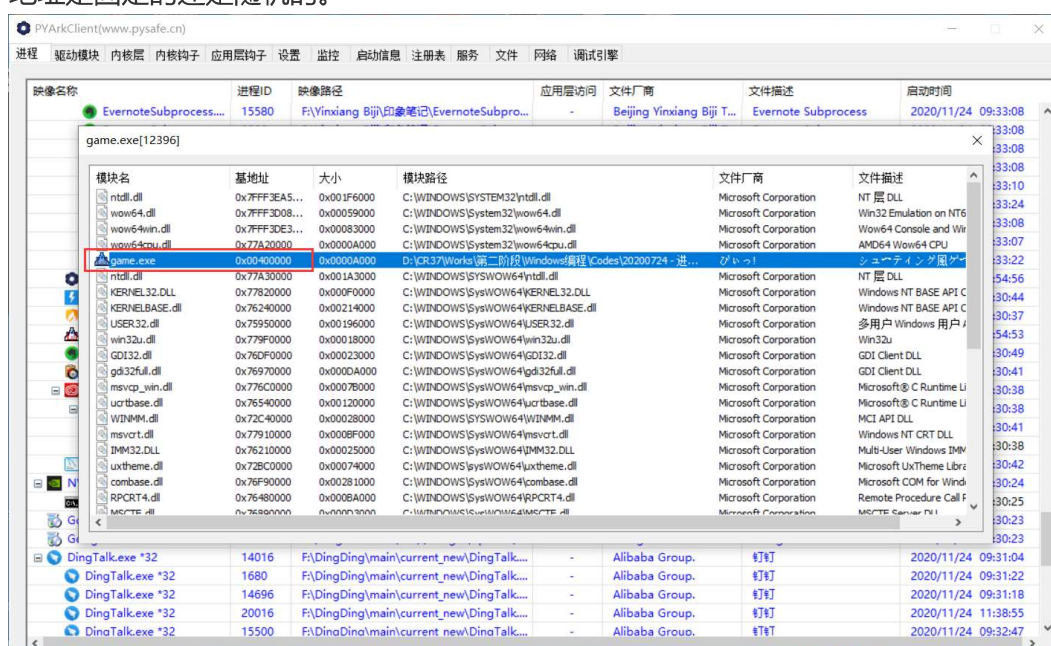
```
10 LPVOID Fun1()
11 {
12     LPVOID pHeap = malloc(10);
13     return pHeap;
14 }
15
16 int main(int argc, char* argv[])
17 {
18     g_pHeap = malloc(999);
19     printf("申请的内存地址 g_pHeap = %p, "
20           "通过全局变量保存申请到的内存地址: &g_pHeap = %p\r\n", g_pHeap, &g_pHeap);
21
22     LPVOID p = Fun1();
23     HANDLE pHandle = GetModuleHandle(NULL);
24     int pOffset = (int)&p - (int)pHandle;
25     printf("&pHandle = %p, pOffset = %p\r\n", pHandle, pOffset);
26 }
```

名称	值	类型
p	0x00e75120	void*
pHandle	ChildProcess.exe!0x001f0000	void*
pOffset	0x0090fc60	int

修改游戏相关的内存属性就是通过上面的方式，计算出对应模块的内存地址。

飞机游戏实战

修改游戏数据前，需要对游戏进行分析，使用工具只看每次打开的游戏进程主模块的地址是固定的还是随机的。



1. 使用 Spy++ 以及 FindWindow 分别获取这个游戏的窗口类名及窗口句柄。
2. 通过窗口句柄使用 GetWindowThreadProcessID 获取游戏的进程 PID。
3. 通过进程 PID 使用 OpenProcess 打开这个进程（做检查）。

绘制矩形范围：Rectangle

CreateRectRgn -- 创建一个矩形区域

PtInRegion -- 确定指定的点是否在指定的区域内

CreatePen -- 创建具有指定样式、宽度和颜色的逻辑钢笔。该笔随后可被选中进入关联设备并用于绘制直线和曲线

GetStockObject -- 检索库存钢笔、画笔、字体或调色板中的一个句柄

SelectObject -- 将一个对象选择到指定的设备上下文(DC)中

代码示例:

```
// AirPlaneGame.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 <Windows.h>  
#include <time.h>
```

```
#define SPUERMAN_ADDR (0x00403616)    // 无敌地址  
#define PLAN_X_ADDR (0x00406D6C)     // 飞机的X轴坐标地址  
#define PLAN_Y_ADDR (0x00406D70)     // 飞机的Y轴坐标地址  
#define CURRENT_BULLET_NUM_ADDR (0x00406DA8) // 当前子弹的地址  
#define BULLET_ARY_ADDR (0x00406E10)  // 子弹数组地址
```

```
HDC hdc;  
int nWndWidth;  
int nWndHeight;
```

```
bool IsCanMove(HANDLE hGameProcess, int x, int y)  
{
```

```
    DWORD dwBytes;
```

```
// 创建一个矩形区域
```

```
    HRGN hrgn = CreateRectRgn(x - 16, nWndHeight - y - 32, x + 32,  
nWndHeight - y + 16);
```

```
    Rectangle(hdc, x - 16, nWndHeight - y - 32, x + 32, nWndHeight - y +  
16);
```

```
    int nCurrentBulletNum = 0;
```

```
    ReadProcessMemory(hGameProcess, (LPVOID)CURRENT_BULLET_NUM_ADDR,  
        &nCurrentBulletNum, sizeof(nCurrentBulletNum), &dwBytes);
```

```
    for (size_t i = 0; i < nCurrentBulletNum; i++)  
    {
```

```
        int nCurrentBulletX = 0;
```

```
        int nCurrentBulletY = 0;
```

```
// 分别读取子弹的X轴、Y轴坐标
```

```
        ReadProcessMemory(hGameProcess,  
            (LPVOID)(BULLET_ARY_ADDR + i * 15 + 0),
```

```
            &nCurrentBulletX,
```

```
            sizeof(nCurrentBulletX), &dwBytes);
```

```
        ReadProcessMemory(hGameProcess,
```

```

(LPVOID) (BULLET_ARY_ADDR + i * 15 + 4),
&nCurrentBulletY,
sizeof(nCurrentBulletY), &dwBytes);

nCurrentBulletX = (nCurrentBulletX >> 6) - 4;
nCurrentBulletY = nWndHeight - (nCurrentBulletY >> 6) - 4;

// 读取每一个子弹的坐标位置
printf("nCurrentBullet = %d, nCurrentBulletX = %d, nCurrentBulletY = %d\r\n",
    i + 1, nCurrentBulletX, nCurrentBulletY);

// 判断子弹的坐标是否在矩形范围内
if (PtInRegion(hrgn, nCurrentBulletX, nCurrentBulletY))
{
    return false;
}
return true;
}

int main()
{
    // 通过窗口类名获取窗口句柄
    HWND hWnd = FindWindow("wcTKKN", NULL);

    hdc = ::GetDC(hWnd);
    HPEN hpen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
    SelectObject(hdc, hpen);
    SelectObject(hdc, GetStockObject(NULL_BRUSH));

    // 通过窗口句柄获取游戏窗口的大小
    RECT rect;
    GetClientRect(hWnd, &rect);
    nWndWidth = rect.right - rect.left; // 游戏窗口的宽度
    nWndHeight = rect.bottom - rect.top; // 游戏窗口的高度

    // 通过窗口句柄获取进程的句柄
    DWORD dwPID;
    DWORD dwResult;
    dwResult = GetWindowThreadProcessId(hWnd, &dwPID);
    if (dwResult == NULL)
    {
        printf("GetWindowThreadProcessId Error!\r\n");
    }
}

```

```

// 通过进程ID打开进程，获取进程的句柄
HANDLE hGameProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
if (hGameProcess == NULL)
{
    printf("OpenProcess Error!");
}

// 读取进程内存
char szBuff[0x1000]; // 存储读取到的数据
DWORD dwBytes;       // 存储读取到数据的字节数

// 读取进程指定位置上的数据
ReadProcessMemory(hGameProcess, (LPCVOID)SPUERMAN_ADDR, szBuff,
sizeof(szBuff), &dwBytes);
printf("%02x\r\n", *(unsigned char*)szBuff);

// 修改指定位置上的数据
szBuff[0] = 0xeb;
WriteProcessMemory(hGameProcess, (LPVOID)SPUERMAN_ADDR, szBuff,
sizeof(szBuff), &dwBytes);

// 移动飞机的坐标
int nPlanX = 0;
int nPlanY = 0;
printf("游戏窗口的宽度为: %d\r\n", nWndWidth);
printf("游戏窗口的高度为: %d\r\n", nWndHeight);
//WriteProcessMemory(hGameProcess, (LPVOID)PLAN_X_ADDR, &nPlanX,
sizeof(nPlanX), &dwBytes);
//WriteProcessMemory(hGameProcess, (LPVOID)PLAN_Y_ADDR, &nPlanY,
sizeof(nPlanY), &dwBytes);

// 读取当前游戏中子弹的数量
int nCurrentBulletNum = 1;
//WriteProcessMemory(hGameProcess, (LPVOID)CURRENT_BULLET_NUM_ADDR,
// &nCurrentBulletNum, sizeof(nCurrentBulletNum), &dwBytes);

ReadProcessMemory(hGameProcess,
(LPVOID)CURRENT_BULLET_NUM_ADDR,
&nCurrentBulletNum,
sizeof(nCurrentBulletNum), &dwBytes);

printf("当前子弹数量为: %d\r\n", nCurrentBulletNum);

int x = 0;
int y = 0;

srand((unsigned)time(NULL));

```

```
while (true)
{
    if (nWndWidth == 0 || nWndHeight == 0)
    {
        return 0;
    }
    x = rand() % nWndWidth;
    y = rand() % nWndHeight;

    if (IsCanMove(hGameProcess, x, y))
    {
        // 更改飞机的位置
        WriteProcessMemory(hGameProcess, (LPVOID)PLAN_X_ADDR, &x,
sizeof(x), &dwBytes);
        WriteProcessMemory(hGameProcess, (LPVOID)PLAN_Y_ADDR, &y,
sizeof(y), &dwBytes);
    }
    Sleep(100);
}

// 关闭进程句柄
CloseHandle(hGameProcess);
return 0;
}
```