重要知识一

2008-03-29 14:05

一、 前言

所谓游戏外挂，其实是一种游戏外辅程序，它可以协助玩家自动产生游戏动作、修改游戏网络数据包以及修改游戏内存数据等，以实现玩家用最少的时间和金钱去完成功力升级和过关斩将。虽然，现在对游戏外挂程序的“合法”身份众说纷纭，在这里我不想对此发表任何个人意见，让时间去说明一切吧。

不管游戏外挂程序是不是“合法”身份，但是它却是具有一定的技术含量的，在这些小小程序中使用了许多高端技术，如拦截 Sock 技术、拦截 API 技术、模拟键盘与鼠标技术、直接修改程序内存技术等等。本文将对常见的游戏外挂中使用的技术进行全面剖析。

二、认识外挂

游戏外挂的历史可以追溯到单机版游戏时代，只不过当时它使用了另一个更通俗易懂的名字??游戏修改器。它可以在游戏中追踪锁定游戏主人公的各项能力数值。这样玩家在游戏中可以达到主角不掉血、不耗费魔法、不消耗金钱等目的。这样降低了游戏的难度，使得玩家更容易通关。

随着网络游戏的时代的来临，游戏外挂在原有的功能之上进行了新的发展，它变得更加多种多样，功能更加强大，操作更加简单，以至有些游戏的外挂已经成为一个体系，比如《石器时代》，外挂品种达到了几十种，自动战斗、自动行走、自动练级、自动补血、加速、不遇敌、原地遇敌、快速增加经验值、按键精灵……几乎无所不包。

游戏外挂的设计主要是针对于某个游戏开发的，我们可以根据它针对的游戏的类型可大致可将外挂分为两种大类。

一类是将游戏中大量繁琐和无聊的攻击动作使用外挂自动完成，以帮助玩家轻松搞定攻击对象并可以快速的增加玩家的经验值。比如在《龙族》中有一种工作的设定，玩家的工作等级越高，就可以驾驭越好的装备。但是增加工作等级却不是一件有趣的事情，毋宁说是重复枯燥的机械劳动。如果你想做法师用的杖，首先需要做基本工作--?砍树。砍树的方法很简单，在一棵大树前不停的点鼠标就可以了，每 10000 的经验升一级。这就意味着玩家要在大树前不停的点击鼠标，这种无聊的事情通过"按键精灵"就可以解决。外挂的"按键精灵"功能可以让玩家摆脱无趣的点击鼠标的工作。

另一类是由外挂程序产生欺骗性的网络游戏封包，并将这些封包发送到网络游戏服务器，利用这些虚假信息欺骗服务器进行游戏数值的修改，达到修改角色能力数值的目的。这类外挂程序针对性很强，一般在设计时都是针对某个游戏某个版本来做的，因为每个网络游戏服务器与客户端交流的数据包各不相同，外挂程序必须要对欺骗的网络游戏服务器的数据包进行分析，才能产生服务器识别的数据包。这类外挂程序也是当前最流利的一类游戏外挂程序。

另外，现在很多外挂程序功能强大，不仅实现了自动动作代理和封包功能，而且还提

供了对网络游戏的客户端程序的数据进行修改，以达到欺骗网络游戏服务器的目的。我相信，随着网络游戏商家的反外挂技术的进展，游戏外挂将会产生更多更优秀的技术，让我们期待着看场技术大战吧.....

三、外挂技术综述

可以将开发游戏外挂程序的过程大体上划分为两个部分：

前期部分工作是对外挂的主体游戏进行分析，不同类型的外挂分析主体游戏的内容也不相同。如外挂为上述谈到的外挂类型中的第一类时，其分析过程常是针对游戏的场景中的攻击对象的位置和分布情况进行分析，以实现外挂自动进行攻击以及位置移动。如外挂为外挂类型中的第二类时，其分析过程常是针对游戏服务器与客户端之间通讯包数据的结构、内容以及加密算法的分析。因网络游戏公司一般都不会公布其游戏产品的通讯包数据的结构、内容和加密算法的信息，所以对于开发第二类外挂成功的关键在于是否能正确分析游戏包数据的结构、内容以及加密算法，虽然可以使用一些工具辅助分析，但是这还是一种坚苦而复杂的工作。

后期部分工作主要是根据前期对游戏的分析结果，使用大量的程序开发技术编写外挂程序以实现对游戏的控制或修改。如外挂程序为第一类外挂时，通常会使用到鼠标模拟技术来实现游戏角色的自动位置移动，使用键盘模拟技术来实现游戏角色的自动攻击。如外挂程序为第二类外挂时，通常会使用到挡截 Sock 和挡截 API 函数技术，以挡截游戏服务器传来的网络数据包并将数据包修改后封包后传给游戏服务器。另外，还有许多外挂使用对游戏客户端程序内存数据修改技术以及游戏加速技术。

本文主要是针对开发游戏外挂程序后期使用的程序开发技术进行探讨，重点介绍的如下几种在游戏外挂中常使用的程序开发技术：

- 动作模拟技术：主要包括键盘模拟技术和鼠标模拟技术。
- 封包技术：主要包括挡截 Sock 技术和挡截 API 技术。

四、动作模拟技术

我们在前面介绍过，几乎所有的游戏都有大量繁琐和无聊的攻击动作以增加玩家的功力，还有那些数不完的迷宫，这些好像已经成为了角色游戏的代名词。现在，外挂可以帮助玩家从这些繁琐而无聊的工作中摆脱出来，专注于游戏情节的进展。外挂程序为了实现自动角色位置移动和自动攻击等功能，需要使用到键盘模拟技术和鼠标模拟技术。下面我们将重点介绍这些技术并编写一个简单的实例帮助读者理解动作模拟技术的实现过程。

１． 鼠标模拟技术

几乎所有的游戏中都使用了鼠标来改变角色的位置和方向，玩家仅用一个小小的鼠标，就可以使角色畅游天下。那么，我们如何实现在没有玩家的参与下角色也可以自动行走呢。其实实现这个并不难，仅仅几个 Windows API 函数就可以搞定，让我们先来认识认识这些 API 函数。

(1) 模拟鼠标动作 API 函数 mouse_event，它可以实现模拟鼠标按下和放开等动作。

```
VOID mouse_event(
    DWORD dwFlags, // 鼠标动作标识。
    DWORD dx, // 鼠标水平方向位置。
    DWORD dy, // 鼠标垂直方向位置。
    DWORD dwData, // 鼠标轮子转动的数量。
    DWORD dwExtraInfo // 一个关联鼠标动作辅加信息。
);
```

其中，dwFlags 表示了各种各样的鼠标动作和点击活动，它的常用取值如下：

MOUSEEVENTF_MOVE　表示模拟鼠标移动事件。

MOUSEEVENTF_LEFTDOWN 表示模拟按下鼠标左键。

MOUSEEVENTF_LEFTUP 表示模拟放开鼠标左键。

MOUSEEVENTF_RIGHTDOWN 表示模拟按下鼠标右键。

MOUSEEVENTF_RIGHTUP 表示模拟放开鼠标右键。

MOUSEEVENTF_MIDDLEDOWN 表示模拟按下鼠标中键。

MOUSEEVENTF_MIDDLEUP 表示模拟放开鼠标中键。

(2)、设置和获取当前鼠标位置的 API 函数。获取当前鼠标位置使用 GetCursorPos() 函数，设置当前鼠标位置使用 SetCursorPos()函数。

```
BOOL GetCursorPos(
  LPPOINT  lpPoint // 返回鼠标的当前位置。
);
BOOL SetCursorPos(
int X, // 鼠标的水平方向位置。
    int Y //鼠标的垂直方向位置。
);
```

通常游戏角色的行走都是通过鼠标移动至目的地，然后按一下鼠标的按钮就搞定了。下面我们使用上面介绍的 API 函数来模拟角色行走过程。

```
CPoint oldPoint,newPoint;
GetCursorPos(&oldPoint); //保存当前鼠标位置。
newPoint.x = oldPoint.x+40;
newPoint.y = oldPoint.y+10;
SetCursorPos(newPoint.x,newPoint.y); //设置目的地位置。
```

```
mouse_event(MOUSEEVENTF_RIGHTDOWN,0,0,0,0);//模拟按下鼠标右键。
mouse_event(MOUSEEVENTF_RIGHTUP,0,0,0,0);//模拟放开鼠标右键。
```

2. 键盘模拟技术

在很多游戏中，不仅提供了鼠标的操作，而且还提供了键盘的操作，在对攻击对象进行攻击时还可以使用快捷键。为了使这些攻击过程能够自动进行，外挂程序需要使用键盘模拟技术。像鼠标模拟技术一样，Windows API 也提供了一系列 API 函数来完成对键盘动作的模拟。

模拟键盘动作 API 函数 keydb_event，它可以模拟对键盘上的某个或某些键进行按下或放开的动作。

```
VOID keybd_event(
    BYTE bVk, // 虚拟键值。
    BYTE bScan, // 硬件扫描码。
    DWORD dwFlags, // 动作标识。
    DWORD dwExtraInfo // 与键盘动作关联的辅加信息。
);
```

其中，bVk 表示虚拟键值，其实它是一个 BYTE 类型值的宏，其取值范围为 1-254。有关虚拟键值表请在 MSDN 上使用关键字“Virtual-Key Codes”查找相关资料。bScan 表示当键盘上某键被按下和放开时，键盘系统硬件产生的扫描码，我们可以 MapVirtualKey()函数在虚拟键值与扫描码之间进行转换。dwFlags 表示各种各样的键盘动作，它有两种取值：KEYEVENTF_EXTENDEDKEY 和 KEYEVENTF_KEYUP。

下面我们使用一段代码实现在游戏中按下 Shift+R 快捷键对攻击对象进行攻击。

```
keybd_event(VK_CONTROL,MapVirtualKey(VK_CONTROL,0),0,0); //按下 CTRL 键。
keybd_event(0x52,MapVirtualKey(0x52,0),0,0);//键下 R 键。
keybd_event(0x52,MapVirtualKey(0x52,0), KEYEVENTF_KEYUP,0);//放开 R 键。
keybd_event(VK_CONTROL,MapVirtualKey(VK_CONTROL,0),
KEYEVENTF_KEYUP,0);//放开 CTRL 键。
```

3. 激活外挂

上面介绍的鼠标和键盘模拟技术实现了对游戏角色的动作部分的模拟，但要想外挂能工作于游戏之上，还需要将其与游戏的场景窗口联系起来或者使用一个激活键，就象按键精灵的那个激活键一样。我们可以用 GetWindow 函数来枚举窗口，也可以用 Findwindow 函数来查找特定的窗口。另外还有一个 FindWindowEx 函数可以找到窗口的子窗口，当游戏切换场景的时候我们可以用 FindWindowEx 来确定一些当前窗口的特征，从而判断是否还在这个场景，方法很多了，比如可以 GetWindowInfo 来确定一些东西，比如当查找不到某个按钮的时候就说明游戏场景已经切换了等等办法。当使用激活键进行关联，需要使用 Hook 技术开发一个全局键盘钩子，在这里就不具体介绍全局钩子的开发过程了，在后面的实例中我们将会使用到全局钩子，到时将学习到全局钩子的相关知识。

4． 实例实现

通过上面的学习，我们已经基本具备了编写动作式游戏外挂的能力了。下面我们将创建一个画笔程序外挂，它实现自动移动画笔字光标的位置并写下一个红色的“R”字。以这个实例为基础，加入相应的游戏动作规则，就可以实现一个完整的游戏外挂。这里作者不想使用某个游戏作为例子来开发外挂（因没有游戏商家的授权啊！），如读者感兴趣的话可以找一个游戏试试，最好仅做测试技术用。

首先，我们需要编写一个全局钩子，使用它来激活外挂，激活键为 F10。创建全局钩子步骤如下：

(1)．选择 MFC AppWizard(DLL)创建项目 ActiveKey，并选择 MFC Extension DLL（共享 MFC 拷贝）类型。

(2).插入新文件 ActiveKey.h，在其中输入如下代码：

```
#ifndef _KEYDLL_H
#define _KEYDLL_H

class AFX_EXT_CLASS CKeyHookublic CObject
{
   public:
CKeyHook();
~CKeyHook();
HHOOK Start();   //安装钩子
BOOL Stop(); //卸载钩子
};
#endif
```

(3).在 ActiveKey.cpp 文件中加入声明 " #include ActiveKey.h " 。

(4).在 ActiveKey.cpp 文件中加入共享数据段，代码如下：

```
//Shared data section
#pragma data_seg("sharedata"
HHOOK glhHook=NULL; //钩子句柄。
HINSTANCE glhInstance=NULL; //DLL 实例句柄。
#pragma data_seg()
```

(5).在 ActiveKey.def 文件中设置共享数据段属性，代码如下：

```
SETCTIONS
shareddata READ WRITE SHARED
```

(6).在 ActiveKey.cpp 文件中加入 CkeyHook 类的实现代码和钩子函数代码：

```
//键盘钩子处理函数。
extern "C" LRESULT WINAPI KeyboardProc(int nCode,WPARAM
wParam,LPARAM lParam)
{
if( nCode >= 0
{
if( wParam == 0X79 //当按下 F10 键时，激活外挂。
  {
    //外挂实现代码。
CPoint newPoint,oldPoint;
     GetCursorPos(&oldPoint);
     newPoint.x = oldPoint.x+40;
     newPoint.y = oldPoint.y+10;
     SetCursorPos(newPoint.x,newPoint.y);
     mouse_event(MOUSEEVENTF_LEFTDOWN,0,0,0,0);//模拟按下鼠标左键。
    mouse_event(MOUSEEVENTF_LEFTUP,0,0,0,0);//模拟放开鼠标左键。
    keybd_event(VK_SHIFT,MapVirtualKey(VK_SHIFT,0),0,0); //按下 SHIFT 键。
    keybd_event(0x52,MapVirtualKey(0x52,0),0,0);//按下 R 键。
    keybd_event(0x52,MapVirtualKey(0x52,0),KEYEVENTF_KEYUP,0);//放开 R 键。
    keybd_event(VK_SHIFT,MapVirtualKey(VK_SHIFT,0),KEYEVENTF_KEYUP,0);//放
开 SHIFT 键。
            SetCursorPos(oldPoint.x,oldPoint.y);
  }
}
return CallNextHookEx(glhHook,nCode,wParam,lParam);
}

CKeyHook::CKeyHook(){}
CKeyHook::~CKeyHook()
{
if( glhHook
Stop();
}
//安装全局钩子。
HHOOK CKeyHook::Start()
{
glhHook = SetWindowsHookEx(WH_KEYBOARD,KeyboardProc,glhInstance,0);//设置键
盘钩子。
return glhHook;
}
//卸载全局钩子。
BOOL CKeyHook::Stop()
{
BOOL bResult = TRUE;
```

```
   if( glhHook
        bResult = UnhookWindowsHookEx(glhHook);//卸载键盘钩子。
        return bResult;
        }
```

(7).修改 DllMain 函数，代码如下：

```
        extern "C" int APIENTRY
        DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
        {
//如果使用 lpReserved 参数则删除下面这行
UNREFERENCED_PARAMETER(lpReserved);

if (dwReason == DLL_PROCESS_ATTACH)
{
      TRACE0("NOtePadHOOK.DLL Initializing!\n";
       //扩展 DLL 仅初始化一次
      if (!AfxInitExtensionModule(ActiveKeyDLL, hInstance))
return 0;
      new CDynLinkLibrary(ActiveKeyDLL);
                  //把 DLL 加入动态 MFC 类库中
      glhInstance = hInstance;
      //插入保存 DLL 实例句柄
}
else if (dwReason == DLL_PROCESS_DETACH)
{
      TRACE0("NotePadHOOK.DLL Terminating!\n";
      //终止这个链接库前调用它
      AfxTermExtensionModule(ActiveKeyDLL);
}
return 1;
        }
```

(8).编译项目 ActiveKey，生成 ActiveKey.DLL 和 ActiveKey.lib。

接着，我们还需要创建一个外壳程序将全局钩子安装了 Windows 系统中，这个外壳程序编写步骤如下：

(1).创建一个对话框模式的应用程序，项目名为 Simulate。

(2).在主对话框中加入一个按钮，使用 ClassWizard 为其创建 CLICK 事件。

(3).将 ActiveKey 项目 Debug 目录下的 ActiveKey.DLL 和 ActiveKey.lib 拷贝到 Simulate 项目目录下。

(4).从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象

/库模块”中输入 ActiveKey.lib。

(5).将 ActiveKey 项目中的 ActiveKey.h 头文件加入到 Simulate 项目中，并在 Stdafx.h 中加入#include ActiveKey.h。

(6).在按钮单击事件函数输入如下代码：

```
void CSimulateDlg::OnButton1()
{
// TODO: Add your control notification handler code here
if( !bSetup
{
m_hook.Start();//激活全局钩子。
}
else
{
m_hook.Stop();//撤消全局钩子。
}
bSetup = !bSetup;

}
```

(7).编译项目，并运行程序，单击按钮激活外挂。

(8).启动画笔程序，选择文本工具并将笔的颜色设置为红色，将鼠标放在任意位置后，按 F10 键，画笔程序自动移动鼠标并写下一个红色的大写 R。图一展示了按 F10 键前的画笔程序的状态，图二展示了按 F10 键后的画笔程序的状态。

利用 HOOK 拦截封包原理

截获 API 是个很有用的东西，比如你想分析一下别人的程序是怎样工作的。这里我介绍一下一种我自己试验通过的方法。
首先，我们必须设法把自己的代码放到目标程序的进程空间里去。Windows Hook 可以帮我们实现这一点。SetWindowsHookEx 的声明如下：

```
HHOOK SetWindowsHookEx(
int idHook, // hook type
HOOKPROC lpfn, // hook procedure
HINSTANCE hMod, // handle to application instance
DWORD dwThreadId // thread identifier
);
```

具体的参数含义可以翻阅 msdn，没有 msdn 可谓寸步难行。
这里 Hook 本身的功能并不重要，我们使用它的目的仅仅只是为了能够让 Windows 把我们的代码植入别的进程里去。hook Type 我们任选一种即可，只要保证是目标程序肯定会调用到就行，这里我用的是 WH_CALLWNDPROC。lpfn 和 hMod 分别指向我们的钩子代码

及其所在的 dll，dwThreadId 设为 0，表示对所有系统内的线程都挂上这样一个 hook，这样我们才能把代码放到别的进程里去。

之后，我们的代码就已经进入了系统内的所有进程空间了。必须注意的是，我们只需要截获我们所关心的目标程序的调用，因此还必须区分一下进程号。我们自己的钩子函数中，第一次运行将进行最重要的 API 重定向的工作。也就是通过将所需要截获的 API 的开头几个字节改为一个跳转指令，使其跳转到我们的 API 中来。这是最关键的部分。这里我想截三个调用，ws2_32.dll 中的 send 和 recv、user32.dll 中的 GetMessageA。

```
DWORD dwCurrentPID = 0;
HHOOK hOldHook = NULL;
DWORD pSend = 0;
DWORD pRecv = 0;
GETMESSAGE pGetMessage = NULL;

BYTE btNewBytes[8] = { 0x0B8, 0x0, 0x0, 0x40, 0x0, 0x0FF, 0x0E0, 0 };
DWORD dwOldBytes[3][2];

HANDLE hDebug = INVALID_HANDLE_value;

LRESULT CALLBACK CallWndProc( int nCode, WPARAM wParam, LPARAM lParam
{
DWORD dwSize;
DWORD dwPIDWatched;
HMODULE hLib;

if( dwCurrentPID == 0
{
dwCurrentPID = GetCurrentProcessId();
HWND hwndMainHook;
hwndMainHook = ::FindWindow( 0, "MainHook" ;
dwPIDWatched = ::SendMessage( hwndMainHook, (WM_USER+100), 0, 0 ;
hOldHook = (HHOOK)::SendMessage( hwndMainHook, (WM_USER+101), 0, 0 ;

if( dwCurrentPID == dwPIDWatched
{
hLib = LoadLibrary( "ws2_32.dll" ;
pSend = (DWORD)GetProcAddress( hLib, "send" ;
pRecv = (DWORD)GetProcAddress( hLib, "recv" ;

::ReadProcessMemory( INVALID_HANDLE_value, (void *)pSend, (void *)dwOldBytes[0],
sizeof(DWORD)*2, &dwSize ;
*(DWORD *)( btNewBytes + 1 = (DWORD)new_send;
::WriteProcessMemory( INVALID_HANDLE_value, (void *)pSend, (void *)btNewBytes,
sizeof(DWORD)*2, &dwSize ;
```

```
::ReadProcessMemory( INVALID_HANDLE_value, (void *)pRecv, (void *)dwOldBytes[1],
sizeof(DWORD)*2, &dwSize ;
*(DWORD *)( btNewBytes + 1 = (DWORD)new_recv;
::WriteProcessMemory( INVALID_HANDLE_value, (void *)pRecv, (void *)btNewBytes,
sizeof(DWORD)*2, &dwSize ;

hLib = LoadLibrary( "user32.dll" ;
pGetMessage = (GETMESSAGE)GetProcAddress( hLib, "GetMessageA" ;
::ReadProcessMemory( INVALID_HANDLE_value, (void *)pGetMessage, (void
*)dwOldBytes[2], sizeof(DWORD)*2, &dwSize ;
*(DWORD *)( btNewBytes + 1 = (DWORD)new_GetMessage;
::WriteProcessMemory( INVALID_HANDLE_value, (void *)pGetMessage, (void
*)btNewBytes, sizeof(DWORD)*2, &dwSize ;

hDebug = ::CreateFile( "C:\\Trace.log", GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0 ;
}
}

if( hOldHook != NULL
{
return CallNextHookEx( hOldHook, nCode, wParam, lParam ;
}

return 0;
}
```

上面的钩子函数，只有第一次运行时有用，就是把三个函数的首 8 字节修改一下（实际上只需要 7 个）。btNewBytes 中的指令实际就是

```
mov eax, 0x400000
jmp eax
```

这里的 0x400000 就是新的函数的地址，比如 new_recv/new_send/new_GetMessage，此时，偷梁换柱已经完成。再看看我们的函数中都干了些什么。以 GetMessageA 为例：

```
BOOL _stdcall new_GetMessage( LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin,
UINT wMsgFilterMax
{
DWORD dwSize;
char szTemp[256];
BOOL r = false;

//Watch here before it's executed.
sprintf( szTemp, "Before GetMessage : HWND 0x%8.8X, msgMin 0x%8.8X, msgMax
0x%8.8x \r\n", hWnd, wMsgFilterMin, wMsgFilterMax ;
::WriteFile( hDebug, szTemp, strlen(szTemp), &dwSize, 0 ;
```

```
//Watch over

// restore it at first
::WriteProcessMemory( INVALID_HANDLE_value, (void *)pGetMessage, (void
*)dwOldBytes[2], sizeof(DWORD)*2, &dwSize ;

// execute it
r = pGetMessage( lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax ;

// hook it again
*(DWORD *)( btNewBytes + 1 = (DWORD)new_GetMessage;
::WriteProcessMemory( INVALID_HANDLE_value, (void *)pGetMessage, (void
*)btNewBytes, sizeof(DWORD)*2, &dwSize ;

//Watch here after it's executed
sprintf( szTemp, "Result of GetMessage is %d.\r\n", r ;
::WriteFile( hDebug, szTemp, strlen( szTemp , &dwSize, 0 ;
if( r
{
sprintf( szTemp, "Msg : HWND 0x%8.8X, MSG 0x%8.8x, wParam 0x%8.8X, lParam
0x%8.8X\r\nTime 0x%8.8X, X %d, Y %d\r\n",
lpMsg->hwnd, lpMsg->message,
lpMsg->wParam, lpMsg->lParam, lpMsg->time,
lpMsg->pt.x, lpMsg->pt.y ;
::WriteFile( hDebug, szTemp, strlen( szTemp , &dwSize, 0 ;
}
strcpy( szTemp, "\r\n" ;
::WriteFile( hDebug, szTemp, strlen( szTemp , &dwSize, 0 ;

//Watch over

return r;
}
```

先将截获下来的参数，写入到一个 log 文件中，以便分析。然后恢复原先保留下来的 GetMessageA 的首 8 字节，然后执行真正的 GetMessageA 调用，完毕后再将执行结果也写入 log 文件，然后将 GetMessageA 的执行结果返回给调用者。
整个截获的过程就是这样。你可以把其中的写 log 部分改成你自己想要的操作。这里有个不足的地方是，截获动作是不能够并发进行的，如果目标进程是多线程的，就会有问题。解决办法是，可以在每次 new_GetMessage 中加入一个 CriticalSection 的锁和解锁，以使调用变为串行进行，但这个我没有试验过。

WriteProecssMemory 在 win9x 下不能对 2g 以上的空间写操作，NT 内核的可以；WSOCKET32.DLL 等不是在 2g 以上，可以对它们用 WriteProecssMemory

重要知识二

2008-03-29 14:06

五、封包技术　　通过对动作模拟技术的介绍，我们对游戏外挂有了一定程度上的认识，也学会了使用动作模拟技术来实现简单的动作模拟型游戏外挂的制作。这种动作模拟型游戏外挂有一定的局限性，它仅仅只能解决使用计算机代替人力完成那么有规律、繁琐而无聊的游戏动作。但是，随着网络游戏的盛行和复杂度的增加，很多游戏要求将客户端动作信息及时反馈回服务器，通过服务器对这些动作信息进行有效认证后，再向客户端发送下一步游戏动作信息，这样动作模拟技术将失去原有的效应。为了更好地“外挂”这些游戏，游戏外挂程序也进行了升级换代，它们将以前针对游戏用户界面层的模拟推进到数据通讯层，通过封包技术在客户端挡截游戏服务器发送来的游戏控制数据包，分析数据包并修改数据包；同时还需按照游戏数据包结构创建数据包，再模拟客户端发送给游戏服务器，这个过程其实就是一个封包的过程。　　封包的技术是实现第二类游戏外挂的最核心的技术。封包技术涉及的知识很广泛，实现方法也很多，如挡截 WinSock、挡截 API 函数、挡截消息、VxD 驱动程序等。在此我们也不可能在此文中将所有的封包技术都进行详细介绍，故选择两种在游戏外挂程序中最常用的两种方法：挡截 WinSock 和挡截 API 函数。　　1．挡截 WinSock　　众所周知，Winsock 是 Windows 网络编程接口，它工作于 Windows 应用层，它提供与底层传输协议无关的高层数据传输编程接口。在 Windows 系统中，使用 WinSock 接口为应用程序提供基于 TCP/IP 协议的网络访问服务，这些服务是由 Wsock32.DLL 动态链接库提供的函数库来完成的。　　由上说明可知，任何 Windows 基于 TCP/IP 的应用程序都必须通过 WinSock 接口访问网络，当然网络游戏程序也不例外。由此我们可以想象一下，如果我们可以控制 WinSock 接口的话，那么控制游戏客户端程序与服务器之间的数据包也将易如反掌。按着这个思路，下面的工作就是如何完成控制 WinSock 接口了。由上面的介绍可知，WinSock 接口其实是由一个动态链接库提供的一系列函数，由这些函数实现对网络的访问。有了这层的认识，问题就好办多了，我们可以制作一个类似的动态链接库来代替原 WinSock 接口库，在其中实现 WinSock32.dll 中实现的所有函数，并保证所有函数的参数个数和顺序、返回值类型都应与原库相同。在这个自制作的动态库中，可以对我们感兴趣的函数（如发送、接收等函数）进行挡截，放入外挂控制代码，最后还继续调用原 WinSock 库中提供的相应功能函数，这样就可以实现对网络数据包的挡截、修改和发送等封包功能。　　下面重点介绍创建挡截 WinSock 外挂程序的基本步骤：　　(1) 创建 DLL 项目，选择 Win32 Dynamic-Link Library，再选择 An empty DLL project。　　(2) 新建文件 wsock32.h，按如下步骤输入代码：　　① 加入相关变量声明：　　HMODULE hModule=NULL; //模块句柄　　char buffer[1000]; //缓冲区　　FARPROC proc; //函数入口指针　　② 定义指向原 WinSock 库中的所有函数地址的指针变量，因 WinSock 库共提供 70 多个函数，限于篇幅，在此就只选择几个常用的函数列出，有关这些库函数的说明可参考 MSDN 相关内容。　　//定义指向原 WinSock 库函数地址的指针变量。　　SOCKET (__stdcall *socket1)(int ,int,int);//创建 Sock 函数。　　int　(__stdcall *WSAStartup1)(WORD,LPWSADATA);//初始化 WinSock 库函数。　　int　(__stdcall *WSACleanup1)();//清除 WinSock 库函数。　　int (__stdcall *recv1)(SOCKET ,char FAR * ,int ,int ;//接收数据函数。　　int (__stdcall *send1)(SOCKET ,const char * ,int ,int);//发送数据函数。　　int (__stdcall *connect1)(SOCKET,const struct sockaddr *,int);//创建连接函数。　　int (__stdcall *bind1)(SOCKET ,const struct sockaddr *,int ;//绑定函数。　　.....其它函数地址指针的定义略。　　(3) 新建 wsock32.cpp 文件，按如下步骤输入代码：　　① 加入相关头文件声明：　　#include "windows.h"　　#include "stdio.h"　　#include "wsock32.h"　　② 添加 DllMain 函数，在此函数中首先需要加载原 WinSock 库，并获取此库中所有函数的地址。代码如下：　　BOOL WINAPI DllMain (HANDLE hInst,ULONG ul_reason_for_call,LPVOID lpReserved)

{ if(hModule==NULL){ //加载原 WinSock 库，原 WinSock 库已复制为 wsock32.001。 hModule=LoadLibrary("wsock32.001"; } else return 1;// 获取原 WinSock 库中的所有函数的地址并保存，下面仅列出部分代码。 if(hModule!=NULL){ //获取原 WinSock 库初始化函数的地址，并保存到 WSAStartup1 中。proc=GetProcAddress(hModule,"WSAStartup"; WSAStartup1=(int (_stdcall *)(WORD,LPWSADATA))proc; //获取原 WinSock 库消除函数的地址，并保存到 WSACleanup1 中。 proc=GetProcAddress(hModule i,"WSACleanup"; WSACleanup1=(int (_stdcall *)())proc; //获取原创建 Sock 函数的地址，并保存到 socket1 中。 proc=GetProcAddress(hModule,"socket"; socket1=(SOCKET (_stdcall *)(int ,int,int))proc; //获取原创建连接函数的地址，并保存到 connect1 中。 proc=GetProcAddress(hModule,"connect"; connect1=(int (_stdcall *)(SOCKET ,const struct sockaddr *,int)proc; //获取原发送函数的地址，并保存到 send1 中。 proc=GetProcAddress(hModule,"send"; send1=(int (_stdcall *)(SOCKET ,const char * ,int ,int)proc; //获取原接收函数的地址，并保存到 recv1 中。 proc=GetProcAddress(hModule,"recv"; recv1=(int (_stdcall *)(SOCKET ,char FAR * ,int ,int)proc; .....其它获取函数地址代码略。 } else return 0; return 1;} ③ 定义库输出函数，在此可以对我们感兴趣的函数中添加外挂控制代码，在所有的输出函数的最后一步都调用原 WinSock 库的同名函数。部分输出函数定义代码如下：//库输出函数定义。//WinSock 初始化函数。 int PASCAL FAR WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData) { //调用原 WinSock 库初始化函数 return WSAStartup1(wVersionRequired,lpWSAData); } //WinSock 结束清除函数。 int PASCAL FAR WSACleanup(void) { return WSACleanup1(); //调用原 WinSock 库结束清除函数。 } //创建 Socket 函数。 SOCKET PASCAL FAR socket (int af, int type, int protocol) { //调用原 WinSock 库创建 Socket 函数。 return socket1(af,type,protocol); } //发送数据包函数 int PASCAL FAR send(SOCKET s,const char * buf,int len,int flags) { //在此可以对发送的缓冲 buf 的内容进行修改，以实现欺骗服务器。 外挂代码..... //调用原 WinSock 库发送数据包函数。 return send1(s,buf,len,flags); }//接收数据包函数。 int PASCAL FAR recv(SOCKET s, char FAR * buf, int len, int flags) { //在此可以挡截到服务器端发送到客户端的数据包，先将其保存到 buffer 中。 strcpy(buffer,buf); //对 buffer 数据包数据进行分析后，对其按照玩家的指令进行相关修改。 外挂代码..... //最后调用原 WinSock 中的接收数据包函数。 return recv1(s, buffer, len, flags); } .....其它函数定义代码略。 (4)、新建 wsock32.def 配置文件，在其中加入所有库输出函数的声明，部分声明代码如下： LIBRARY "wsock32" EXPORTS WSAStartup @1 WSACleanup @2 recv @3 send @4 socket @5 bind @6 closesocket @7 connect @8 .....其它输出函数声明代码略。 (5)、从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象/库模块”中输入 Ws2_32.lib。 (6)、编译项目，产生 wsock32.dll 库文件。 (7)、将系统目录下原 wsock32.dll 库文件拷贝到被外挂程序的目录下，并将其改名为 wsock.001；再将上面产生的 wsock32.dll 文件同样拷贝到被外挂程序的目录下。重新启动游戏程序，此时游戏程序将先加载我们自己制作的 wsock32.dll 文件，再通过该库文件间接调用原 WinSock 接口函数来实现访问网络。上面我们仅仅介绍了挡载 WinSock 的实现过程，至于如何加入外挂控制代码，还需要外挂开发人员对游戏数据包结构、内容、加密算法等方面的仔细分析（这个过程将是一个艰辛的过

程），再生成外挂控制代码。关于数据包分析方法和技巧，不是本文讲解的范围，如您感兴趣可以到网上查查相关资料。　　2.挡截 API　　挡截 API 技术与挡截 WinSock 技术在原理上很相似，但是前者比后者提供了更强大的功能。挡截 WinSock 仅只能挡截 WinSock 接口函数，而挡截 API 可以实现对应用程序调用的包括 WinSock API 函数在内的所有 API 函数的挡截。如果您的外挂程序仅打算对 WinSock 的函数进行挡截的话，您可以只选择使用上小节介绍的挡截 WinSock 技术。随着大量外挂程序在功能上的扩展，它们不仅仅只提供对数据包的挡截，而且还对游戏程序中使用的 Windows API 或其它 DLL 库函数的挡截，以使外挂的功能更加强大。例如，可以通过挡截相关 API 函数以实现对非中文游戏的汉化功能，有了这个利器，可以使您的外挂程序无所不能了。　　挡截 API 技术的原理核心也是使用我们自己的函数来替换掉 Windows 或其它 DLL 库提供的函数，有点同挡截 WinSock 原理相似吧。但是，其实现过程却比挡截 WinSock 要复杂的多，如像实现挡截 Winsock 过程一样，将应用程序调用的所有的库文件都写一个模拟库有点不大可能，就只说 Windows API 就有上千个，还有很多库提供的函数结构并未公开，所以写一个模拟库代替的方式不大现实，故我们必须另谋良方。　　挡截 API 的最终目标是使用自定义的函数代替原函数。那么，我们首先应该知道应用程序何时、何地、用何种方式调用原函数。接下来，需要将应用程序中调用该原函数的指令代码进行修改，使它将调用函数的指针指向我们自己定义的函数地址。这样，外挂程序才能完全控制应用程序调用的 API 函数，至于在其中如何加入外挂代码，就应需求而异了。最后还有一个重要的问题要解决，如何将我们自定义的用来代替原 API 函数的函数代码注入被外挂游戏程序进行地址空间中，因在 Windows 系统中应用程序仅只能访问到本进程地址空间内的代码和数据。　　综上所述，要实现挡截 API 函数，至少需要解决如下三个问题：　　● 如何定位游戏程序中调用 API 函数指令代码？　　● 如何修改游戏程序中调用 API 函数指令代码？　　● 如何将外挂代码（自定义的替换函数代码）注入到游戏程序进程地址空间？　　下面我们逐一介绍这几个问题的解决方法：　　(1) 、定位调用 API 函数指令代码　　我们知道，在汇编语言中使用 CALL 指令来调用函数或过程的，它是通过指令参数中的函数地址而定位到相应的函数代码的。那么，我们如果能寻找到程序代码中所有调用被挡截的 API 函数的 CALL 指令的话，就可以将该指令中的函数地址参数修改为替代函数的地址。虽然这是一个可行的方案，但是实现起来会很繁琐，也不稳健。庆幸的是，Windows 系统中所使用的可执行文件（PE 格式）采用了输入地址表机制，将所有在程序调用的 API 函数的地址信息存放在输入地址表中，而在程序代码 CALL 指令中使用的地址不是 API 函数的地址，而是输入地址表中该 API 函数的地址项，如想使程序代码中调用的 API 函数被代替掉，只用将输入地址表中该 API 函数的地址项内容修改即可。具体理解输入地址表运行机制，还需要了解一下 PE 格式文件结构，其中图三列出了 PE 格式文件的大致结构。图三：PE 格式大致结构图　　PE 格式文件一开始是一段 DOS 程序，当你的程序在不支持 Windows 的环境中运行时，它就会显示“This Program cannot be run in DOS mode”这样的警告语句，接着这个 DOS 文件头，就开始真正的 PE 文件内容了。首先是一段称为“IMAGE_NT_HEADER”的数据，其中是许多关于整个 PE 文件的消息，在这段数据的尾端是一个称为 Data Directory 的数据表，通过它能快速定位一些 PE 文件中段（section）的地址。在这段数据之后，则是一个“IMAGE_SECTION_HEADER”的列表，其中的每一项都详细描述了后面一个段的相关信息。接着它就是 PE 文件中最主要的段数据了，执行代码、数据和资源等等信息就分别存放在这些段中。　　在所有的这些段里，有一个被称为“.idata”的段（输入数据段）值得我们去注意，该段中包含着一些被称为输入地址表（IAT，Import Address Table）的数据列表。每个用隐式方式加载的 API 所在的 DLL 都有一个 IAT 与之对应，同时一个 API 的地址也与 IAT 中一项相对应。当一个应用程序加载到内存中后，针对每一个 API 函数调用，相应的产生如下的汇编指令：　　JMP DWORD PTR [XXXXXXXX]　　或　　CALL DWORD PTR [XXXXXXXX]　　其中，[XXXXXXXX]表示指向了输入地址表中一个项，其内容是一个 DWORD，而正是这个 DWORD 才是 API 函数在内存中的真正地

址。因此我们要想拦截一个 API 的调用，只要简单的把那个 DWORD 改为我们自己的函数的地址。 (2) 、修改调用 API 函数代码 从上面对 PE 文件格式的分析可知，修改调用 API 函数代码其实是修改被调用 API 函数在输入地址表中 IAT 项内容。由于 Windows 系统对应用程序指令代码地址空间的严密保护机制，使得修改程序指令代码非常困难，以至于许多高手为之编写 VxD 进入 Ring0。在这里，我为大家介绍一种较为方便的方法修改进程内存，它仅需要调用几个 Windows 核心 API 函数，下面我首先来学会一下这几个 API 函数：DWORD VirtualQuery(LPCVOID lpAddress, // address of region PMEMORY_BASIC_INFORMATION lpBuffer, // information buffer DWORD dwLength // size of buffer); 该函数用于查询关于本进程内虚拟地址页的信息。其中，lpAddress 表示被查询页的区域地址；lpBuffer 表示用于保存查询页信息的缓冲；dwLength 表示缓冲区大小。返回值为实际缓冲大小。 BOOL VirtualProtect(LPVOID lpAddress, // region of committed pages SIZE_T dwSize, // size of the region DWORD flNewProtect, // desired access protection PDWORD lpflOldProtect // old protection); 该函数用于改变本进程内虚拟地址页的保护属性。其中，lpAddress 表示被改变保护属性页区域地址；dwSize 表示页区域大小；flNewProtect 表示新的保护属性，可取值为 PAGE_READONLY、PAGE_READWRITE、PAGE_EXECUTE 等；lpflOldProtect 表示用于保存改变前的保护属性。如果函数调用成功返回“T”，否则返回“F”。 有了这两个 API 函数，我们就可以随心所欲的修改进程内存了。首先，调用 VirtualQuery()函数查询被修改内存的页信息，再根据此信息调用 VirtualProtect()函数改变这些页的保护属性为 PAGE_READWRITE，有了这个权限您就可以任意修改进程内存数据了。下面一段代码演示了如何将进程虚拟地址为 0x0040106c 处的字节清零。 BYTE* pData = 0x0040106c; MEMORY_BASIC_INFORMATION mbi_thunk; //查询页信息。 VirtualQuery(pData, &mbi_thunk, sizeof(MEMORY_BASIC_INFORMATION)); //改变页保护属性为读写。 VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize, PAGE_READWRITE, &mbi_thunk.Protect); //清零。 *pData = 0x00; //恢复页的原保护属性。 DWORD dwOldProtect; VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize, mbi_thunk.Protect, &dwOldProtect); (3)、注入外挂代码进入被挂游戏进程中 完成了定位和修改程序中调用 API 函数代码后，我们就可以随意设计自定义的 API 函数的替代函数了。做完这一切后，还需要将这些代码注入到被外挂游戏程序进程内存空间中，不然游戏进程根本不会访问到替代函数代码。注入方法有很多，如利用全局钩子注入、利用注册表注入挡截 User32 库中的 API 函数、利用 CreateRemoteThread 注入（仅限于 NT/2000）、利用 BHO 注入等。因为我们在动作模拟技术一节已经接触过全局钩子，我相信聪明的读者已经完全掌握了全局钩子的制作过程，所以我们在后面的实例中，将继续利用这个全局钩子。至于其它几种注入方法，如果感兴趣可参阅 MSDN 有关内容。 有了以上理论基础，我们下面就开始制作一个挡截 MessageBoxA 和 recv 函数的实例，在开发游戏外挂程序 时，可以此实例为框架，加入相应的替代函数和处理代码即可。此实例的开发过程如下： (1) 打开前面创建的 ActiveKey 项目。 (2) 在 ActiveKey.h 文件中加入 HOOKAPI 结构，此结构用来存储被挡截 API 函数名称、原 API 函数地址和替代函数地址。 typedef struct tag_HOOKAPI { LPCSTR szFunc;//被 HOOK 的 API 函数名称。 PROC pNewProc;//替代函数地址。 PROC pOldProc;//原 API 函数地址。 }HOOKAPI, *LPHOOKAPI; (3) 打开 ActiveKey.cpp 文件，首先加入一个函数，用于定位输入库在输入数据段中的 IAT 地址。代码如下： extern "C" __declspec(dllexport)PIMAGE_IMPORT_DESCRIPTOR LocationIAT(HMODULE hModule, LPCSTR szImportMod) //其中，hModule 为进程模块句柄；szImportMod

为输入库名称。 { //检查是否为 DOS 程序，如是返回 NULL，因 DOS 程序没有 IAT。 PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER) hModule; if(pDOSHeader->e_magic != IMAGE_DOS_SIGNATURE) return NULL; //检查是否为 NT 标志，否则返回 NULL。 PIMAGE_NT_HEADERS pNTHeader = (PIMAGE_NT_HEADERS)((DWORD)pDOSHeader+ (DWORD)(pDOSHeader->e_lfanew)); if(pNTHeader->Signature != IMAGE_NT_SIGNATURE) return NULL; //没有 IAT 表则返回 NULL。 if(pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress == 0) return NULL; //定位第一个 IAT 位置。 PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD)pDOSHeader + (DWORD)(pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress)); //根据输入库名称循环检查所有的 IAT，如匹配则返回该 IAT 地址，否则检测下一个 IAT。 while (pImportDesc->Name) { //获取该 IAT 描述的输入库名称。 PSTR szCurrMod = (PSTR)((DWORD)pDOSHeader + (DWORD)(pImportDesc->Name)); if (stricmp(szCurrMod, szImportMod) == 0) break; pImportDesc++; } if(pImportDesc->Name == NULL) return NULL; return pImportDesc; } 再加入一个函数，用来定位被挡截 API 函数的 IAT 项并修改其内容为替代函数地址。代码如下： extern "C" __declspec(dllexport) HookAPIByName(HMODULE hModule, LPCSTR szImportMod, LPHOOKAPI pHookApi) //其中，hModule 为进程模块句柄；szImportMod 为输入库名称；pHookAPI 为 HOOKAPI 结构指针。 { //定位 szImportMod 输入库在输入数据段中的 IAT 地址。 PIMAGE_IMPORT_DESCRIPTOR pImportDesc = LocationIAT(hModule, szImportMod); if (pImportDesc == NULL) return FALSE; //第一个 Thunk 地址。 PIMAGE_THUNK_DATA pOrigThunk = (PIMAGE_THUNK_DATA)((DWORD)hModule + (DWORD)(pImportDesc->OriginalFirstThunk)); //第一个 IAT 项的 Thunk 地址。 PIMAGE_THUNK_DATA pRealThunk = (PIMAGE_THUNK_DATA)((DWORD)hModule + (DWORD)(pImportDesc->FirstThunk)); //循环查找被截 API 函数的 IAT 项，并使用替代函数地址修改其值。 while(pOrigThunk->u1.Function) { //检测此 Thunk 是否为 IAT 项。if((pOrigThunk->u1.Ordinal & IMAGE_ORDINAL_FLAG) != IMAGE_ORDINAL_FLAG) { //获取此 IAT 项所描述的函数名称。 PIMAGE_IMPORT_BY_NAME pByName =(PIMAGE_IMPORT_BY_NAME)((DWORD)hModule+(DWORD)(pOrigThunk->u1.AddressOfData)); if(pByName->Name[0] == '\0') return FALSE; //检测是否为挡截函数。if(strcmpi(pHookApi->szFunc, (char*)pByName->Name) == 0) { MEMORY_BASIC_INFORMATION mbi_thunk; //查询修改页的信息。 VirtualQuery(pRealThunk, &mbi_thunk, sizeof(MEMORY_BASIC_INFORMATION)); //改变修改页保护属性为 PAGE_READWRITE。 VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize, PAGE_READWRITE, &mbi_thunk.Protect); //保存原来的 API 函数地址。 if(pHookApi->pOldProc == NULL) pHookApi->pOldProc = (PROC)pRealThunk->u1.Function; //修改 API 函数 IAT 项内容为替代函数地址。pRealThunk->u1.Function = (PDWORD)pHookApi->pNewProc; //恢复修改页保护属性。DWORD dwOldProtect; VirtualProtect(mbi_thunk.BaseAddress, mbi_thunk.RegionSize, mbi_thunk.Protect, &dwOldProtect); } } pOrigThunk++; pRealThunk++; }

SetLastError(ERROR_SUCCESS); //设置错误为 ERROR_SUCCESS，表示成功。 return TRUE; } (4) 定义替代函数，此实例中只给 MessageBoxA 和 recv 两个 API 进行挡截。代码如下： static int WINAPI MessageBoxA1 (HWND hWnd , LPCTSTR lpText, LPCTSTR lpCaption, UINT uType) { //过滤掉原 MessageBoxA 的正文和标题内容，只显示如下内容。return MessageBox(hWnd, "Hook API OK!", "Hook API", uType); } static int WINAPI recv1(SOCKET s, char FAR *buf, int len, int flags { //此处可以挡截游戏服务器发送来的网络数据包，可以加入分析和处理数据代码。 return recv(s,buf,len,flags); } (5) 在 KeyboardProc 函数中加入激活挡截 API 代码，在 if(wParam == 0X79 语句中后面加入如下 else if 语句： ..... //当激活 F11 键时，启动挡截 API 函数功能。 else if(wParam == 0x7A { HOOKAPI api[2];api[0].szFunc ="MessageBoxA";//设置被挡截函数的名称。api[0].pNewProc = (PROC)MessageBoxA1;//设置替代函数的地址。 api[1].szFunc ="recv";//设置被挡截函数的名称。api[1].pNewProc = (PROC)recv1; //设置替代函数的地址。//设置挡截 User32.dll 库中的 MessageBoxA 函数。 HookAPIByName(GetModuleHandle(NULL),"User32.dll",&api[0]);//设置挡截 Wsock32.dll 库中的 recv 函数。 HookAPIByName(GetModuleHandle(NULL),"Wsock32.dll",&api[1]); } ..... (6) 在 ActiveKey.cpp 中加入头文件声明 "#include "wsock32.h"。 从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象/库模块”中输入 Ws2_32..lib。 (7) 重新编译 ActiveKey 项目，产生 ActiveKey.dll 文件，将其拷贝到 Simulate.exe 目录下。运行 Simulate.exe 并启动全局钩子。激活任意应用程序，按 F11 键后，运行此程序中可能调用 MessageBoxA 函数的操作，看看信息框是不是有所变化。同样，如此程序正在接收网络数据包，就可以实现封包功能了。 六、结束语 除了以上介绍的几种游戏外挂程序常用的技术以外，在一些外挂程序中还使用了游戏数据修改技术、游戏加速技术等。在这篇文章里，就不逐一介绍了。

C#.NET 中的类型转换

2008-04-08 20:18

C# 出来也有些日子了，最近由于编程的需要，对 C# 的类型转换做了一些研究，其内容涉及 C# 的装箱/拆箱/别名、数值类型间相互转换、字符的 ASCII 码和 Unicode 码、数值字符串和数值之间的转换、字符串和字符数组/字节数组之间的转换、各种数值类型和字节数组之间的转换、十六进制数输出以及日期型数据的一些转换处理，在这里与大家分享——

1. 装箱、拆箱还是别名

许多 C#.NET 的书上都有介绍 int -> Int32 是一个装箱的过程，反之则是拆箱的过程。许多其它变量类型也是如此，如：short <-> Int16，long <-> Int64 等。对于一般的程序员来说，大可不必去了解这一过程，因为这些装箱和拆箱的动作都是可以自动完成的，不需要写代码进行干预。但是我们需要记住这些类型之间的关系，所以，我们使用“别名”来记忆它们之间的关系。

C# 是全面向对象的语言，比 Java 的面向对象都还彻底——它把简单数据类型通过默认的装箱动作封装成了类。Int32、Int16、Int64 等就是相应的类名，而那些我们熟悉的、简单易记的名称，如 int、short、long 等，我们就可以把它称作是 Int32、Int16、Int64 等类型的别名。

那么除了这三种类型之外，还有哪些类有“别名”呢？常用的有如下一些：

bool -> System.Boolean (布尔型，其值为 true 或者 false)
char -> System.Char (字符型，占有两个字节，表示 1 个 Unicode 字符)
byte -> System.Byte (字节型，占 1 字节，表示 8 位正整数，范围 0 ~ 255)
sbyte -> System.SByte (带符号字节型，占 1 字节，表示 8 位整数，范围 -128 ~ 127)
ushort -> System.UInt16 (无符号短整型，占 2 字节，表示 16 位正整数，范围 0 ~ 65,535)
uint -> System.UInt32 (无符号整型，占 4 字节，表示 32 位正整数，范围 0 ~ 4,294,967,295)
ulong -> System.UInt64 (无符号长整型，占 8 字节，表示 64 位正整数，范围 0 ~ 大约 10 的 20 次方)
short -> System.Int16 (短整型，占 2 字节，表示 16 位整数，范围 -32,768 ~ 32,767)
int -> System.Int32 (整型，占 4 字节，表示 32 位整数，范围 -2,147,483,648 到 2,147,483,647)
long -> System.Int64 (长整型，占 8 字节，表示 64 位整数，范围大约 -(10 的 19) 次方 到 10 的 19 次方)
float -> System.Single (单精度浮点型，占 4 个字节)
double -> System.Double (双精度浮点型，占 8 个字节)

我们可以用下列代码做一个实验：

```
private void TestAlias() {    // this.textBox1 是一个文本框，类型为 System.Windows.Forms.TextBox    // 设计中已经将其 Multiline 属性设置为 true    byte a = 1; char b = 'a'; short c = 1;    int d = 2; long e = 3; uint f = 4; bool g = true;    this.textBox1.Text = "";
this.textBox1.AppendText("byte -> " + a.GetType().FullName + "\n");
this.textBox1.AppendText("char -> " + b.GetType().FullName + "\n");
this.textBox1.AppendText("short -> " + c.GetType().FullName + "\n");
this.textBox1.AppendText("int -> " + d.GetType().FullName + "\n");
this.textBox1.AppendText("long -> " + e.GetType().FullName + "\n");
this.textBox1.AppendText("uint -> " + f.GetType().FullName + "\n");
this.textBox1.AppendText("bool -> " + g.GetType().FullName + "\n");}
```

在窗体中新建一个按钮，并在它的单击事件中调用该 TestAlias() 函数，我们将看到运行结果如下：

byte -> System.Byte
char -> System.Char
short -> System.Int16
int -> System.Int32
long -> System.Int64
uint -> System.UInt32
bool -> System.Boolean

这足以说明各别名对应的类！

2. 数值类型之间的相互转换

这里所说的数值类型包括 byte, short, int, long, fload, double 等，根据这个排列顺序，各种类型的值依次可以向后自动进行转换。举个例来说，把一个 short 型的数据赋值给一个 int 型的变量，short 值会自动行转换成 int 型值，再赋给 int 型变量。如下例：

```
private void TestBasic() {    byte a = 1; short b = a; int c = b;
long d = c; float e = d; double f = e;    this.textBox1.Text = "";
this.textBox1.AppendText("byte a = " + a.ToString() + "\n");
this.textBox1.AppendText("short b = " + b.ToString() + "\n");
this.textBox1.AppendText("int c = " + c.ToString() + "\n");
this.textBox1.AppendText("long d = " + d.ToString() + "\n");
this.textBox1.AppendText("float e = " + e.ToString() + "\n");
this.textBox1.AppendText("double f = " + f.ToString() + "\n");}
```

译顺利通过，运行结果是各变量的值均为 1；当然，它们的类型分别还是 System.Byte 型……System.Double 型。现在我们来试试，如果把赋值的顺序反过来会怎么样呢？在 TestBasic() 函数中追加如下语句：

```
int g = 1;
short h = g;
this.textBox1.AppendText("h = " + h.ToString() + "\n");
```

结果编译报错：
G:\Projects\Visual C#\Convert\Form1.cs(118): 无法将类型“int”隐式转换为“short”
其中，Form1.cs 的 118 行即 short h = g 所在行。

这个时候，如果我们坚持要进行转换，就应该使用强制类型转换，这在 C 语言中常有提及，就是使用“(类型名) 变量名”形式的语句来对数据进行强制转换。如上例修改如下：

```
short g = 1;
byte h = (byte) g; // 将 short 型的 g 的值强制转换成 short 型后再赋给变量 h
this.textBox1.AppendText("h = " + h.ToString() + "\n");
```

编译通过，运行结果输出了 h = 1，转换成功。
但是，如果我们使用强制转换，就不得不再考虑一个问题：short 型的范围是 -32768 ~ 23767，而 byte 型的范围是 0 ~ 255，那么，如果变量 g 的大小超过了 byte 型的范围又会出现什么样的情况呢？我们不妨再一次改写代码，将值改为 265，比 255 大 10

```
short g = 265; //265 = 255 + 10
byte h = (byte) g;
this.textBox1.AppendText("h = " + h.ToString() + "\n");
```

编译没有出错，运行结果却不是 h = 265，而是 h = 9。

因此，我们在进行转换的时候，应当注意被转换的数据不能超出目标类型的范围。这不仅体现在多字节数据类型(相对，如上例的 short) 转换为少字节类型(相对，如上例的 byte) 时，也体现在字节数相同的有符号类型和无符号类型之间，如将 byte 的 129 转换为 sbyte 就会溢出。这方面的例子大同小异，就不详细说明了。

3. 字符的 ASCII 码和 Unicode 码

很多时候我们需要得到一个英文字符的 ASCII 码，或者一个汉字字符的 Unicode 码，或者从相关的编码查询它是哪一个字符的编码。很多人，尤其是从 VB 程序序转过来学 C# 的人，会报怨 C# 里为什么没有提供现成的函数来做这个事情——因为在 VB 中有 Asc() 函数和 Chr() 函数用于这类转换。
但是如果你学过 C，你就会清楚，我们只需要将英文字符型数据强制转换成合适的数值型数据，就可以得到相应的 ASCII 码；反之，如果将一个合适的数值型数据强制转换成字符型数据，就可以得到相应的字符。
C# 中字符的范围扩大了，不仅包含了单字节字符，也可以包含双字节字符，如中文字符等。而在字符和编码之间的转换，则仍延用了 C 语言的做法——强制转换。不妨看看下面的例子

```
private void TestChar() {    char ch = 'a'; short ii = 65;
this.textBox1.Text = "";    this.textBox1.AppendText("The ASCII code
of \'" + ch + "\' is: " + (short) ch + "\n");
this.textBox1.AppendText("ASCII is " + ii.ToString() + ", the char
is: " + (char) ii + "\n");    char cn = '中'; short uc = 22478;
this.textBox1.AppendText("The Unicode of \'" + cn + "\' is: " +
(short) cn + "\n");    this.textBox1.AppendText("Unicode is " +
uc.ToString() + ", the char is: " + (char) uc + "\n");}
```

它的运行结果是

```
The ASCII code of 'a' is: 97
ASCII is 65, the char is: A
The Unicode of '中' is: 20013
Unicode is 22478, the char is: 城
```

从这个例子中，我们便能非常清楚的了解——通过强制转换，可以得以字符的编码，或者得到编码表示的字符。如果你需要的不是 short 型的编码，请参考第 1 条进行转换，即可得到 int 等类型的编码值。

4. 数值字符串和数值之间的转换

首先，我们得搞明白，什么是数值字符串。我们知道，在 C# 中，字符串是用一对双引号包含的若干字符来表示的，如 "123"。而 "123" 又相对特殊，因为组成该字符串的字符都是数字，这样的字符串，就是数值字符串。在我们的眼中，这即是一串字符，也是一个数，但计算机却只认为它是一个字符串，不是数。因此，我们在某些时候，比如输入数值的时候，把字符串转换成数值；而在另一些时候，我们需要相反的转换。
将数值转换成字符串非常简单，因为每一个类都有一个 void ToString() 方法。所有数

值型的 void ToString() 方法都能将数据转换为数值字符串。如 123.ToSting() 就将得到字符串 "123"。
那么反过来，将数值型字符串转换成数值又该怎么办呢？我们仔细查找一下，会发现 short, int, float 等数值类型均有一个 static Parse() 函数。这个函数就是用来将字符串转换为相应数值的。我们以一个 float 类型的转换为例： float f = float.Parse("543.21"); 其结果 f 的值为 543.21F。当然，其它的数值类型也可以使用同样的方法进行转换，下面的例子可以更明确的说明转换的方法：

```
private void TestStringValue() {    float f = 54.321F;    string str = "123";    this.textBox1.Text = "";    this.textBox1.AppendText("f = " + f.ToString() + "\n");    if (int.Parse(str) == 123) {        this.textBox1.AppendText("str convert to int successfully.");    } else {        this.textBox1.AppendText("str convert to int failed.");    }}
```

运行结果：

f = 54.321
str convert to int successfully.

5. 字符串和字符数组之间的转换

字符串类 System.String 提供了一个 void ToCharArray() 方法，该方法可以实现字符串到字符数组的转换。如下例：

```
private void TestStringChars() {    string str = "mytest";    char[] chars = str.ToCharArray();    this.textBox1.Text = "";    this.textBox1.AppendText("Length of \"mytest\" is " + str.Length + "\n");    this.textBox1.AppendText("Length of char array is " + chars.Length + "\n");    this.textBox1.AppendText("char[2] = " + chars[2] + "\n");}
```

例中以对转换转换到的字符数组长度和它的一个元素进行了测试，结果如下：

Length of "mytest" is 6
Length of char array is 6
char[2] = t

可以看出，结果完全正确，这说明转换成功。那么反过来，要把字符数组转换成字符串又该如何呢？
我们可以使用 System.String 类的构造函数来解决这个问题。System.String 类有两个构造函数是通过字符数组来构造的，即 String(char[]) 和 String[char[], int, int)。后者之所以多两个参数，是因为可以指定用字符数组中的哪一部分来构造字符串。而前者则是用字符数组的全部元素来构造字符串。我们以前者为例，在 TestStringChars() 函数中输入如下语句：

```
char[] tcs = {'t', 'e', 's', 't', ' ', 'm', 'e'};
string tstr = new String(tcs);
this.textBox1.AppendText("tstr = \"" + tstr + "\"\n");
```

运行结果输入 tstr = "test me"，测试说明转换成功。

实际上，我们在很多时候需要把字符串转换成字符数组只是为了得到该字符串中的某个字符。如果只是为了这个目的，那大可不必兴师动众的去进行转换，我们只需要使用 System.String 的 [] 运算符就可以达到目的。请看下例，再在 TestStringChars() 函数中加入如如下语名：

```
char ch = tstr[3];
this.textBox1.AppendText("\"" + tstr + "\"[3] = " + ch.ToString());
```

正确的输出是 "test me"[3] = t，经测试，输出正确。

6. 字符串和字节数组之间的转换

如果还想从 System.String 类中找到方法进行字符串和字节数组之间的转换，恐怕你会失望了。为了进行这样的转换，我们不得不借助另一个类：System.Text.Encoding。该类提供了 bye[] GetBytes(string) 方法将字符串转换成字节数组，还提供了 string GetString(byte[]) 方法将字节数组转换成字符串。

System.Text.Encoding 类似乎没有可用的构造函数，但我们可以找到几个默认的 Encoding，即 Encoding.Default(获取系统的当前 ANSI 代码页的编码)、Encoding.ASCII(获取 7 位 ASCII 字符集的编码)、Encoding.Unicode(获取采用 Little-Endian 字节顺序的 Unicode 格式的编码)、Encoding.UTF7(获取 UTF-7 格式的编码)、Encoding.UTF8(获取 UTF-8 格式的编码) 等。这里主要说说 Encoding.Default 和 Encoding.Unicode 用于转换的区别。

在字符串转换到字节数组的过程中，Encoding.Default 会将每个单字节字符，如半角英文，转换成 1 个字节，而把每个双字节字符，如汉字，转换成 2 个字节。而 Encoding.Unicode 则会将它们都转换成两个字节。我们可以通过下列简单的了解一下转换的方法，以及使用 Encoding.Default 和 Encodeing.Unicode 的区别：

```
private void TestStringBytes() {    string s = "C#语言";    byte[] b1 = System.Text.Encoding.Default.GetBytes(s);    byte[] b2 = System.Text.Encoding.Unicode.GetBytes(s);    string t1 = "", t2 = "";
foreach (byte b in b1) {        t1 += b.ToString("") + " ";    }
foreach (byte b in b2) {        t2 += b.ToString("") + " ";    }
this.textBox1.Text = "";    this.textBox1.AppendText("b1.Length = " + b1.Length + "\n");    this.textBox1.AppendText(t1 + "\n");
this.textBox1.AppendText("b2.Length = " + b2.Length + "\n");
this.textBox1.AppendText(t2 + "\n");}
```

运行结果如下，不说详述，相信大家已经明白了。

```
b1.Length = 6
67 35 211 239 209 212
```

```
b2.Length = 8
67 0 35 0 237 139 0 138
```

将字节数组转换成字符串，使用 Encoding 类的 string GetString(byte[]) 或 string GetString(byte[], int, int) 方法，具体使用何种 Encoding 还是由编码决定。在 TestStringBytes() 函数中添加如下语句作为实例：

```
byte[] bs = {97, 98, 99, 100, 101, 102};
string ss = System.Text.Encoding.ASCII.GetString(bs);
this.textBox1.AppendText("The string is: " + ss + "\n");
```

运行结果为：The string is: abcdef

7. 各种数值类型和字节数组之间的转换

在第 1 条中我们可以查到各种数值型需要使用多少字节的空间来保存数据。将某种数值类型的数据转换成字节数组的时候，得到的一定是相应大小的字节数组；同样，需要把字节数组转换成数值类型，也需要这个字节数组大于相应数值类型的字节数。

现在介绍此类转换的主角：System.BitConverter。该类提供了 byte[] GetBytes(...) 方法将各种数值类型转换成字节数组，也提供了 ToInt32、ToInt16、ToInt64、ToUInt32、ToSignle、ToBoolean 等方法将字节数组转换成相应的数值类型。

由于这类转换通常只是在需要进行较细微的编码/解码操作时才会用到，所以这里就不详细叙述了，仅把 System.BitConverter 类介绍给大家。

8. 转换成十六进制

任何数据在计算机内部都是以二进制保存的，所以进制与数据的存储无关，只与输入输出有关。所以，对于进制转换，我们只关心字符串中的结果。

在上面的第 4 条中提到了 ToString() 方法可以将数值转换成字符串，不过在字符串中，结果是以十进制显示的。现在我们带给它加一些参数，就可以将其转换成十六进制——使用 ToString(string) 方法。

这里需要一个 string 类型的参数，这就是格式说明符。十六进制的格式说明符是 "x" 或者 "X"，使用这两种格式说明符的区别主要在于 A-F 六个数字："x" 代表 a-f 使用小写字母表示，而 "X" 而表示 A-F 使用大字字母表示。如下例：

```
private void TestHex() {    int a = 188;    this.textBox1.Text = "";
this.textBox1.AppendText("a(10) = " + a.ToString() + "\n");
this.textBox1.AppendText("a(16) = " + a.ToString("x") + "\n");
this.textBox1.AppendText("a(16) = " + a.ToString("X") + "\n");}
```

运行结果如下：

```
a(10) = 188
a(16) = bc
```

a(16) = BC

这时候，我们可能有另一种需求，即为了显示结果的整齐，我们需要控制十六进制表示的长度，如果长度不够，用前导的 0 填补。解决这个问题，我们只需要在格式说明符“x”或者“X”后写上表示长度的数字就行了。比如，要限制在 4 个字符的长度，可以写成“X4”。在上例中追加一句：

```
this.textBox1.AppendText("a(16) = " + a.ToString("X4") + "\n");
```

其结果将输出 a(16) = 00BC。

现在，我们还要说一说如何将一个表示十六进制数的字符串转换成整型。这一转换，同样需要借助于 Parse() 方法。这里，我需要 Parse(string, System.Globalization.NumberStyles) 方法。第一个参数是表示十六进制数的字符串，如“AB”、“20”(表示十进制的 32) 等。第二个参数 System.Globalization.NumberStyles 是一个枚举类型，用来表示十六进制的枚举值是 HexNumber。因此，如果我们要将“AB”转换成整型，就应该这样写：int b = int.Parse("AB", System.Globalization.NumberStyles.HexNumber)，最后得到的 b 的值是 171。

9. 日期型数据和长整型数据之间的转换

为什么要将日期型数据转换为长整型数据呢？原因很多，但就我个人来说，经常将它用于数据库的日期存储。由于各种数据库对日期型的定义和处理是不一样的，各种语言对日期型数据的定义的处理也各不相同，因为，我宁愿将日期型数据转换成长整型再保存到数据库中。虽然也可以使用字符串来保存，但使用字符串也会涉及到许多问题，如区域等问题，而且，它需要比保存长整型数据更多的空间。

日期型数据，在 C# 中的参与运算的时候，应该也是转换为长整型数据来运算的。它的长整型值是自 0001 年 1 月 1 日午夜 12:00 以来所经过时间以 100 毫微秒为间隔表示时的数字。这个数在 C# 的 DateTime 中被称为 Ticks(刻度)。DateTime 类型有一个名为 Ticks 的长整型只读属性，就保存着这个值。如此，要从一个 DataTime 型数据得到 long 型值就非常简单了，只需要读出 DataTime 对象的 Ticks 值即可，如：

```
long longDate = DateTime.Now.Ticks;
```

DateTime 的构造函数中也提供了相应的，从长整型数据构造 DateTime 型数据的函数：DateTime(long)。如：

```
DateTime theDate = new DateTime(longDate);
```

但这样对于很多 VB6 程序员来说，是给他们出了一道难题，因为 VB6 中的日期型数据内部是以 Double 型表示的，将其转换为长整型后得到的仅仅是日期，而没有时间。如何协调这两种日期类型呢？

System.DateTime 提供了 double ToOADate() 和 static DateTime FromOADate(double) 两个函数来解决这个问题。前者将当前对象按原来的 double 值输出，后者则从一个 double 值获得一个 System.DateTime 对象。举例如下：

```
private void TestDateTimeLong() {    double doubleDate =
```

```
DateTime.Now.ToOADate();     DateTime theDate =
DateTime.FromOADate(doubleDate);     this.textBox1.Text = "";
this.textBox1.AppendText("Double value of now: " +
doubleDate.ToString() + "\n");     this.textBox1.AppendText("DateTime
from double value: " + theDate.ToString() + "\n");}
```

运行结果：

Double value of now: 37494.661541713
DateTime from double value: 2002-8-26 15:52:37

10. 格式化日期型数据

编程的过程中，通常需要将日期型数据按照一定的格式输出，当然，输出结果肯定是字符串。为此，我们需要使用 System.DateTime 类的 ToString() 方法，并为其指定格式字符串。

MSDN 中，System.Globalization.DateTimeFormatInfo 类的概述里对模式字符串有非常详细的说明，因此，这里我只对常用的一些格式进行说明，首先请看下表：

d	月中的某一天	一位数的日期没有前导零
dd	月中的某一天	一位数的日期有一个前导零
ddd	周中某天的缩写名称	在 AbbreviatedDayNames 中定义
dddd	周中某天的完整名称	在 DayNames 中定义
M	月份数字	一位数的月份没有前导零
MM	月份数字	一位数的月份有一个前导零
MMM	月份的缩写名称	在 AbbreviatedMonthNames 中定义
MMMM	月份的完整名称	在 MonthNames 中定义
y	不包含纪元的年份	如果不包含纪元的年份小于 10，则显示不具有前导零的年份
yy	不包含纪元的年份	如果不包含纪元的年份小于 10，则显示具有前导零的年份
yyyy	包括纪元的四位数的年份	
h	12 小时制的小时	一位数的小时数没有前导零
hh	12 小时制的小时	一位数的小时数有前导零
H	24 小时制的小时	一位数的小时数没有前导零
HH	24 小时制的小时	一位数的小时数有前导零
m	分钟	一位数的分钟数没有前导零
mm	分钟	一位数的分钟数有一个前导零
s	秒	一位数的秒数没有前导零
ss	秒	一位数的秒数有一个前导零

为了便于大家的理解，不妨试试下面的程序：

```
private void TestDateTimeToString() {     DateTime now = DateTime.Now;
string format;     this.textBox1.Text = "";     format = "yyyy-MM-dd
```

```
HH:mm:ss″;     this.textBox1.AppendText(format + ″: ″ +
now.ToString(format) + ″\n″);     format = ″yy 年 M 日 d 日″;
this.textBox1.AppendText(format + ″: ″ + now.ToString(format) +
″\n″);}
```

这段程序将输出结果：

```
yyyy-MM-dd HH:mm:ss: 2002-08-26 17:03:04
yy 年 M 日 d 日: 02 年 8 日 26 日
```

这时候，又出现一个问题，如果要输出的文本信息中包含格式字符怎么办？如

```
format = "year: yyyy, month: MM, day: dd";
this.textBox1.AppendText(now.ToString(format) + "\n");
```

将输出：

```
2ear: 2002, 4on 下 5: 08, 26a2: 26
```

这并不是我想要的结果，怎么办呢？有办法——

```
format = "\"year\": yyyy, \'month\': MM, \'day\': dd";
this.textBox1.AppendText(now.ToString(format) + "\n");
```

看，这次运行结果对了：

```
year: 2002, month: 08, day: 26
```

可以看出，只需要使用单引号或者双引号将文本信息括起来就好。

如果文本信息中包含双引号或者单引号又怎么办呢？这个问题，请读者们动动脑筋吧！

c#字符型数字转换为整数得方法比较

2008-03-25 14:57

c#字符型数字转换为整数得方法比较

首先，我要指出的是，在 C# 中，int 其实就是 System.Int32，即都是 32 位的。

其次，(int) 和 Convert.ToInt32 是两个不同的概念，前者是类型转换，而后者则是内容转换，它们并不总是等效的。我们很清楚 C# 提供类型检查，你不能把一个 string 强制转换成 int，隐式转换就更加不可能，例如如下的代码就行不通了：

```
string text = "1412";
int id = (int)text;
```

因为 string 和 int 是两个完全不同并且互不兼容的类型。说到这里，你可能会问什么才算是兼容的呢？其实，能够使用 (int) 进行强类型转换的只能是数值类型了，例如 long、short、double 等，不过进行这种转

换时你需要考虑精度问题。

然而，我们很清楚上面的代码中 text 实际上储存的是一个数值，我们希望把这个数值提取出来并以 int 的形式储存起来以便日后的运算使用，那么你就需要进行内容转换了。内容转换也叫内容解释，我们把上面的代码稍稍修改就可以达到目的了：

```
string text = "1412";
int id = Convert.ToInt32(text);
```

除此之外，你还可以使用 Int32.Parse 和 Int32.TryParse 来进行解释。

另外，你发现 Convert.ToInt32 有很多重载版本，例如 Convert.ToInt32(double value);，当我们用这个版本来把一个 double 转换成 int 时，ToInt32 会检查被转换的数值是否能够用 int 表示，即是否会发生“越界”，如果是就会抛出 OverflowException，否则就会为你转换，但使用 (int) 进行强制转换，如果被转换的数值大于 Int32.MaxValue，那么你将得到一个错误的结果，例如下面的代码：

```
double d = Int32.MaxValue + 0.1412;
int i = (int)d;
```

不过无论你进行什么数值转换，精度问题都是必须考虑的。

用 Visual C#调用 Windows API 函数

2008-01-15 15:45

Api 函数是构筑 Windws 应用程序的基石，每一种 Windows 应用程序开发工具，它提供的底层函数都间接或直接地调用了 Windows API 函数，同时为了实现功能扩展，一般也都提供了调用 WindowsAPI 函数的接口， 也就是说具备调用动态连接库的能力。Visual C#和其它开发工具一样也能够调用动态链接库的 API 函数。.NET 框架本身提供了这样一种服务,允许受管辖的代码调用动态链接库中实现的非受管辖函数,包括操作系统提供的 Windows API 函数。它能够定位和调用输出函数,根据需要，组织其各个参数(整型、字符串类型、数组、和结构等等)跨越互操作边界。

下面以 C#为例简单介绍调用 API 的基本过程：

动态链接库函数的声明

动态链接库函数使用前必须声明，相对于 VB,C#函数声明显得更加罗嗦，前者通过 Api Viewer 粘贴以后，可以直接使用，而后者则需要对参数作些额外的变化工作。

动态链接库函数声明部分一般由下列两部分组成，一是函数名或索引号，二是动态链接库的文件名。

譬如，你想调用 User32.DLL 中的 MessageBox 函数，我们必须指明函数的名字 MessageBoxA 或 MessageBoxW，以及库名字 User32.dll,我们知道 Win32 API 对每一个涉及字符串和字符的函数一般都存在两个版本，单字节字符的 ANSI 版本和双字节字符的 UNICODE 版本。

下面是一个调用 API 函数的例子：

```
[DllImport("KERNEL32.DLL", EntryPoint="MoveFileW", SetLastError=true,
```

CharSet=CharSet.Unicode, ExactSpelling=true,

CallingConvention=CallingConvention.StdCall)]

public static extern bool MoveFile(String src, String dst);

其中入口点 EntryPoint 标识函数在动态链接库的入口位置，在一个受管辖的工程中，目标函数的原始名字和序号入口点不仅标识一个跨越互操作界限的函数。而且，你还可以把这个入口点映射为一个不同的名字，也就是对函数进行重命名。重命名可以给调用函数带来种种便利，通过重命名，一方面我们不用为函数的大小写伤透脑筋，同时它也可以保证与已有的命名规则保持一致，允许带有不同参数类型的函数共存，更重要的是它简化了对 ANSI 和 Unicode 版本的调用。CharSet 用于标识函数调用所采用的是 Unicode 或是 ANSI 版本，ExactSpelling＝false 将告诉编译器,让编译器决定使用 Unicode 或者是 Ansi 版本。其它的参数请参考 MSDN 在线帮助.

在 C#中，你可以在 EntryPoint 域通过名字和序号声明一个动态链接库函数，如果在方法定义中使用的函数名与 DLL 入口点相同，你不需要在 EntryPoint 域显示声明函数。否则，你必须使用下列属性格式指示一个名字和序号。

[DllImport("dllname", EntryPoint="Functionname")]

[DllImport("dllname", EntryPoint="#123")]

值得注意的是，你必须在数字序号前加“＃”

下面是一个用 MsgBox 替换 MessageBox 名字的例子：

[C#]

using System.Runtime.InteropServices;

public class Win32 {

[DllImport("user32.dll", EntryPoint="MessageBox")]

public static extern int MsgBox(int hWnd, String text, String caption, uint type);

}

许多受管辖的动态链接库函数期望你能够传递一个复杂的参数类型给函数，譬如一个用户定义的结构类型成员或者受管辖代码定义的一个类成员，这时你必须提供额外的信息格式化这个类型，以保持参数原有的布局和对齐。

C#提供了一个 StructLayoutAttribute 类，通过它你可以定义自己的格式化类型，在受管辖代码中，格式化类型是一个用 StructLayoutAttribute 说明的结构或类成员，通过它能够保证其内部成员预期的布局信息。布局的选项共有三种：

布局选项

描述

LayoutKind.Automatic

为了提高效率允许运行态对类型成员重新排序。

注意：永远不要使用这个选项来调用不受管辖的动态链接库函数。

LayoutKind.Explicit

对每个域按照 FieldOffset 属性对类型成员排序

LayoutKind.Sequential

对出现在受管辖类型定义地方的不受管辖内存中的类型成员进行排序。

传递结构成员

下面的例子说明如何在受管辖代码中定义一个点和矩形类型，并作为一个参数传递给 User32.dll 库中的 PtInRect 函数，

函数的不受管辖原型声明如下：

BOOL PtInRect(const RECT *lprc, POINT pt);

注意你必须通过引用传递 Rect 结构参数，因为函数需要一个 Rect 的结构指针。

[C#]

```
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct Point {
public int x;
public int y;
}

[StructLayout(LayoutKind.Explicit]
public struct Rect {
[FieldOffset(0)] public int left;
[FieldOffset(4)] public int top;
[FieldOffset(8)] public int right;
[FieldOffset(12)] public int bottom;
}

class Win32API {
[DllImport("User32.dll")]
public static extern Bool PtInRect(ref Rect r, Point p);
}
```

类似你可以调用 GetSystemInfo 函数获得系统信息：

```
? using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential)]
public struct SYSTEM_INFO {
public uint dwOemId;
public uint dwPageSize;
public uint lpMinimumApplicationAddress;
public uint lpMaximumApplicationAddress;
```

```
public uint dwActiveProcessorMask;
public uint dwNumberOfProcessors;
public uint dwProcessorType;
public uint dwAllocationGranularity;
public uint dwProcessorLevel;
public uint dwProcessorRevision;
}
```

```
[DllImport("kernel32")]
static extern void GetSystemInfo(ref SYSTEM_INFO pSI);

SYSTEM_INFO pSI = new SYSTEM_INFO();
GetSystemInfo(ref pSI);
```

类成员的传递

同样只要类具有一个固定的类成员布局，你也可以传递一个类成员给一个不受管辖的动态链接库函数，下面的例子主要说明如何传递一个 sequential 顺序定义的 MySystemTime 类给 User32.dll 的 GetSystem Time 函数, 函数用 C/C++调用规范如下:

```
void GetSystemTime(SYSTEMTIME* SystemTime);
```

不像传值类型,类总是通过引用传递参数.

```
[C#]
[StructLayout(LayoutKind.Sequential)]
public class MySystemTime {
public ushort wYear;
public ushort wMonth;
public ushort wDayOfWeek;
public ushort wDay;
public ushort wHour;
public ushort wMinute;
public ushort wSecond;
public ushort wMilliseconds;
}
class Win32API {
[DllImport("User32.dll")]
public static extern void GetSystemTime(MySystemTime st);
```

}

回调函数的传递:

从受管辖的代码中调用大多数动态链接库函数,你只需创建一个受管辖的函数定义，然后调用它即可,这个过程非常直接。

如果一个动态链接库函数需要一个函数指针作为参数，你还需要做以下几步：

首先，你必须参考有关这个函数的文档，确定这个函数是否需要一个回调；第二，你必须在受管辖代码中创建一个回调函数；最后，你可以把指向这个函数的指针作为一个参数创递给 DLL 函数,.

回调函数及其实现:

回调函数经常用在任务需要重复执行的场合,譬如用于枚举函数,譬如 Win32 API 中的 EnumFontFamilies (字体枚举), EnumPrinters(打印机), EnumWindows (窗口枚举)函数. 下面以窗口枚举为例,谈谈如何通过调用 EnumWindow 函数遍历系统中存在的所有窗口

分下面几个步骤:

1. 在实现调用前先参考函数的声明

BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARMAM lParam)

显然这个函数需要一个回调函数地址作为参数.

2. 创建一个受管辖的回调函数,这个例子声明为代表类型(delegate),也就是我们所说的回调,它带有两个参数 hwnd 和 lparam,第一个参数是一个窗口句柄，第二个参数由应用程序定义，两个参数均为整形。

当这个回调函数返回一个非零值时，标示执行成功，零则暗示失败，这个例子总是返回 True 值，以便持续枚举。

3. 最后创建以代表对象(delegate)，并把它作为一个参数传递给 EnumWindows 函数，平台会自动地 把代表转化成函数能够识别的回调格式。

[C#]

```
using System;
using System.Runtime.InteropServices;

public delegate bool CallBack(int hwnd, int lParam);

public class EnumReportApp {

[DllImport("user32")]
public static extern int EnumWindows(CallBack x, int y);

public static void Main()
{
CallBack myCallBack = new CallBack(EnumReportApp.Report);
EnumWindows(myCallBack, 0);
```

```
}

public static bool Report(int hwnd, int lParam) {
Console.Write("窗口句柄为");
Console.WriteLine(hwnd);
return true;
}
}
```

指针类型参数传递：

在 Windows API 函数调用时，大部分函数采用指针传递参数，对一个结构变量指针，我们除了使用上面的类和结构方法传递参数之外，我们有时还可以采用数组传递参数。

下面这个函数通过调用 GetUserName 获得用户名

```
BOOL GetUserName(
LPTSTR lpBuffer, // 用户名缓冲区
LPDWORD nSize // 存放缓冲区大小的地址指针
);

[DllImport("Advapi32.dll",
EntryPoint="GetComputerName",
ExactSpelling=false,
SetLastError=true)]
static extern bool GetComputerName (
[MarshalAs(UnmanagedType.LPArray)] byte[] lpBuffer,
 [MarshalAs(UnmanagedType.LPArray)] Int32[] nSize );
```

这个函数接受两个参数，char * 和 int *,因为你必须分配一个字符串缓冲区以接受字符串指针，你可以使用 String 类代替这个参数类型，当然你还可以声明一个字节数组传递 ANSI 字符串，同样你也可以声明一个只有一个元素的长整型数组，使用数组名作为第二个参数。上面的函数可以调用如下：

```
byte[] str=new byte[20];
Int32[] len=new Int32[1];
len[0]=20;
GetComputerName (str,len);
MessageBox.Show(System.Text.Encoding.ASCII.GetString(str));
```

最后需要提醒的是，每一种方法使用前必须在文件头加上：

```
using System.Runtime.InteropServices;
```

c#做外挂

2008-01-15 14:28

做外挂我也是现学的。可以说写的这个教程是现学现卖，希望对用 C#的外挂爱好者能有点帮助。
本教程中有一些以“废话”字样标注的内容，赶时间的可以直接越过。

第一课：C#使用 WINDOW API 和对内存的操作。
这一课是些简单的东西，了解的可以直接越过。考虑到大多数使用 c#的人都是做网站的，可能没有机会接触这些，所以我在这里做一下粗略的介绍。

step 1:认识 WINAPI

windows 系统里提供了很多的函数，我们如果做外挂的话，就需要用到其中的函数(以下简称 API)。(废话：这些 API 被封装在系统路径下的 DLL 文件里。事实上，我们不用关心它在哪，我们只要知道怎么用就可以了，)用起来很简单，格式如下：

```
public partial class Form1 : Form
{
[DllImport("kernel32.dll")]                    \
public static extern int ReadProcessMemory(    |
int hProcess,                                  |
int lpBaseAddress,                             |
int[] lpBuffer,                                  >代码段 1
int nSize,                                     |
int lpNumberOfBytesWritten                     |
);                                              /
...
public Form1()
{
InitializeComponent();
ReadProcessMemory(processhandle,... >代码 2
...
}
...
}
```

代码段 1 就是引用 api 的代码。我们引用的函数，是做外挂时最常用的函数，从它的名字就可以看的出来它的作用---读取进程内存。(废话：从代码里，我们很容易看的出来，这个函数被封装在了 kernel32.dll 这个文件里。)引用之后，我们就可以在自己的代码中使用这个函数了(如代码 2)。
(废话：WINDOWS 还提供很多的 API，如果你有兴趣了解的话，可以到网上搜 WINAPI 手册。想深入了解的话，可以看 MSDN。)
step 2:读写内存
下面我来说一下，如何使用上一步引用的那个 API 读取游戏的数据。先来看看参数：

```
public static extern int ReadProcessMemory(
int hProcess, //进程，如果你是做外挂的话，它代表你要挂的那个游戏。
int lpBaseAddress, //你要读取的内存地址
```

int[] lpBuffer, //从上面那个参数地址里读出来的东西(调用这个函数的就是为了它) 不管这个参数是什么类型，它应该是一个数组，否则读不出东西来
int nSize, //长度，上一个参数，类型是 int，那个长度应该用 4
int lpNumberOfBytesWritten //用 0 就行了，想知道它是干嘛的，自己去 MSND 吧
关于第一个参数 hProcess 如何获取，我过会再说。假设它已经搞定了，那么这个函数，我们需要关心的只有 lpBaseAddress 和 lpBuffer，既读的地址，和读出来的值。(废话：对了，这个函数貌似还有个返回值，我们这里用不到它。如果你有兴趣了解，MSDN)读出来的值 out int lpBuffer 我们在引用 API 的时候声明为 int 型了，但是，我们要从内存里读的值不一定总是 int。我们可以多次引用这个 API，第 3 个参数分别用不同的类型。
下面，我们结合实际，来写一段读取诛仙人物 HP 的代码。首先，我们需要知道人物 HP 的地址，(废话：如何知道这个地址，用 CE 还是 IE，你自己搞定吧。)我是用 IE 在这里 http://www.ghoffice.com/bbs/read.php?tid-35908-fpage-2.html 找到的，它这里是这样写的：
人物基址:[[&H12F830]+&H28]=base
生命:[base+&H254]
(注：&H 表示 16 进制，在 C#里我们用 0x 表示)
一对[]表示读一次地址。也就是说 123 表示值 123，而[123]就表示从地址 123 读出来的值。几对[]，就要用几次 ReadProcessMemory，我们来写下代码：

```
int[] Base=new int[1];
int[] hp=new int[1];
ReadProcessMemory(process, 0x12F830, Base;, 4, 0);//相当于 Base=[&H12F830]
ReadProcessMemory(process, Base+0x28, Base;, 4, 0);//相当于 Base=[Base+&H28]
//读出了人物基址 base
ReadProcessMemory(process, Base+0x254, hp;, 4, 0);//相当于 hp=[base+&H254]
//读出了 hp
```

怎么样，很简单吧。
我们读 HP 只用了 3 行 ReadProcessMemory。有的时候，读某个值可能需要很多对[]，就要写 N 行 ReadProcessMemory，这样写起来就很麻烦，看起来也很晕。下面我们来写个函数，让读内存的过程看起来和[]表示法差不多。

```
//为了看起来好看，函数的名字最好短些，所以我们用 r，表示 read
public static int r(int add)
{
int[] r=new int[1];
try
{
ReadProcessMemory(process, add, r, 4, 0);
return r[0];
}
catch (Exception ex)
{
return -1;
}
}
```

这个函数很简单，不用我多说了吧。
有了这个函数，上面的读取 HP 的代码，我们就可以写成这样了：

```
int Base;
```

```
int hp;
Base=r(r(0x12F830)+0x28);
//读出了人物基址 base
hp=r(base+&H254);
//读出了 hp
```
看起来清晰多了吧。

下面我来说下读取字符串，首先引用 API：
```
[DllImport("kernel32.dll")]
public static extern int ReadProcessMemory(
int hProcess,
int lpBaseAddress,
byte[] lpBuffer,
int nSize,
int lpNumberOfBytesRead
);
```
然后和上面一样，写一个读字符串的方法。
```
public static string rString(IntPtr process, uint add)
{
string[] r;
string temp;
byte[] b = new byte[256];
try
{
API.ReadProcessMemory(process, (IntPtr)add, b, 256, (IntPtr)0);
//读出的 byte[]要按 Unicode 编码为字符串
temp = System.Text.Encoding.Unicode.GetString(b);
//截取第一段字符串
r = temp.Split('\0');
return r[0];
}
catch (Exception ex)
{
return "error";
}
}
```
这个函数和上面那个函数差不多，多的东西注释里已经写了，也很简单，不必我废话了。
下面，我们来读人物的名字。还是刚才那个帖子里得到的，人物名字偏移如下：
人物角色名:[[base+3a4]+0]
代码如下：
```
string name;
name=rString(r(basse + 0x3a4)+0x0);//+0x0 可以去掉
```
读其他类型的数据和读 INT 的雷同，我就不废话了，大家自己搞定吧。

现在万事俱备，就差这个 process 了，下面我来说下，如果获得游戏的进程句柄(废话:进程句柄:一个用来表示某进程的整形值。推广到一般，**句柄，就是表示某**的整形值)。分

两步，第一步：
System.Diagnostics.Process[] GamesProcess
= System.Diagnostics.Process.GetProcessesByName("elementclient");
这一步用的是.NET 本身的方法，System.Diagnostics.Process 是.NET 里的进程类，GetProcessesByName 静态方法是通过进程的名字获得进程数组。这行语句执行之后，所有游戏进程就放在了 GamesProcess 里面。如果你想做多开挂的话，可以通过数组 GamesProcess 的下标，来确定你要挂的游戏。
第二步：
int ProcessID=GamesProcess[0].Id;
int process = OpenProcess(0x1F0FFF, 0, ProcessID);
第 1 行是获得进程 ID，就是任务管理器里看到的 PID。第 2 行就是获得进程句柄。OpenProcess 也是一个系统 API，也是在 kernel32.dll 里。他的 3 个参数和返回值都声明为 INT 就 OK 了。如何引用请看 step 1。大家应该可以看出来怎么用，第 3 个参数是进程 ID，返回的就是进程句柄(废话：1,2 参数做何用，想知道的自己看 MSDN。懒人直接用示例里的参数就行了。以后此类废话不再多说了)。

看到这里，大家可以试着写一个读取人物资料的小东西试试了。当然，前提是你要知道资料的地址。
写内存：
(废话：修改游戏数据，对于写现在的网游外挂来说，意义不是很大。因为重要数据的处理都是在服务端进行的，改了也没用。人们使用写内存，通常是改游戏的代码，以实现一些特殊功能，比如诛仙里的穿墙，无限跳等。要想知道如何改，需要反汇编分析经验。就不是本菜鸟能及的了，呵呵)
WriteProcessMemory(process, (IntPtr)add, bytes, (UInt32)bytes.Length, 0);
写进程内存函数。这个 API 的各参数和 ReadProcessMemory 是一一对应的。大家自己声明，用用看吧。喜欢的话，也可以向上面一样自己写个函数，以简化写内存的代码。在下一课，我们要用这个函数来向游戏里写代码。

下一课将是些更有趣的东西。我们要通过外挂让游戏执行一些操作。敬请期待吧，呵呵。

第 2 课 C#注入
这一课其实也很简单，只不过知道的人不多而已。

step 3:注入

注入没什么复杂的，它是一个很简单的过程。用语言描述就一句话：在别的程序里写入你的代码，并执行。
(废话：传说注入分 3 种，我说的这种属于哪个呢？我懒的去想，能用就行了，呵呵。)
实现起来也很简单，就几行代码：

Copy code

```
byte[] bytes={0xC3};//我们要写入的代码
int addr = VirtualAllocEx(process, 0, bytes.Length, 0x1000, 0x40);//一，申请空间
WriteProcessMemory(process, addr, bytes, bytes.Length, 0);//二，把代码写进去
int hThread = CreateRemoteThread(process, 0, 0, addr, 0, 0, threadId);//三，执行写入的
```

代码
WaitForSingleObject(hThread, 0xFFFFFFFF);//等待线程结束
VirtualFreeEx(process, addr, 0, 0x8000); //四，释放申请的空间
CloseHandle(hThread);//五，关闭线程句饼

仔细看一下这几行代码，你会发现非常简单，几乎不需要我多说什么。这几个豆耐特里豆不出来的函数，都是 API。根据上面的使用方法，引用一下，就可以用了。你能看懂的那几个参数和返回值，就是需要你关心的。不知道的的参数都不用理会，直接用上面的值就行了。还有疑问的话，可以参考 WINAPI 手册。值得注意的地方是，第四步释放申请，如果你看了 API 手册，会发现第三个参数是大小，但如果你用 bytes.Length 的话就错了，一定要用 0。
(废话：如果你不知道怎么根据上面的使用方法引用 API，我就简单说两句。以第二行为例，我们看到 VirtualAllocEx 的返回值和 5 个参数都是 int 行，那么这样声明就行：
[DllImport("Kernel32.dll")]//没有特殊说明的话，API 一般都是从这个 DLL 引用的
public static extern int VirtualAllocEx(
int p1,
int p2,
int p3,
int p4,
int p5
);
大家可以看出来，要申明一个 API 只要知道各参数和返回值的类型，以及 DLL 文件名就可以了。喜欢的话，你可以把参数的名字起的有意义些。)

简简单单几行代码就实现了注入，是不是没你想像的复杂？呵呵。
现在的一个问题就是，代码从何而来？
大家可以使用我的工具将你找到的 CALL 转换为机器码。(废话:这个工具的原理，就是调用 MASM 编译，所以任何你在 MASM 里能使用的语法和指令(限函数内)，都可以在这里用，当然，语法和 MASM 里的语法规则是一样的。使用的方法在附件里有详细的说明,我就不在这里浪费篇章了。)
工具转换得到的结果是型如 60b8d0305a00ffd08b561c83c40461c3 的字符串,大家可以用下面的方法把它转换为 byte[]

Copy code

```
public static byte[] getBytes(string HEX)
{
byte[] bytes = new byte[HEX.Length / 2];
for (int i = 0; i < bytes.Length; i++)
{
bytes[i] = Convert.ToByte(Int32.Parse(HEX.Substring(i * 2 , 2),

System.Globalization.NumberStyles.AllowHexSpecifier));
}
return bytes;
}
```

OK,到这里,大家可以着手试着用外挂调用一下游戏里的攻击 CALL 了.(如果你不会找 CALL,你可以试着在此论坛里找找)

下一节里,我会以技能 CALL 举例说明如何使用参数.
上一课里有些错误,我已经更正并用红字标注了,如果有人因为这些错误在实践时受挫,我表示道歉,呵呵。
过些天我可能会写个简单的 DEMO,大家敬请期待吧.

待续....

[转]汇编语言的准备知识--给初次接触汇编者

2008-01-15 15:29

转]汇编语言的准备知识--给初次接触汇编者
在接触到游戏修改后发现需要很多的汇编知识，于是找汇编基础知识恶补，到网上搜索到一篇不错的文章，给各位想我一样的初学者一起学习！

教程: 汇编语言的准备知识--给初次接触汇编者(1)

汇编语言和 CPU 以及内存,端口等硬件知识是连在一起的. 这也是为什么汇编语言没有通用性的原因. 下面简单讲讲基本知识(针对 INTEL x86 及其兼容机)

==========

x86 汇编语言的指令,其操作对象是 CPU 上的寄存器,系统内存,或者立即数. 有些指令表面上没有操作数, 或者看上去缺少操作数, 其实该指令有内定的操作对象, 比如 push 指令, 一定是对 SS:ESP 指定的内存操作, 而 cdq 的操作对象一定是 eax / edx.

在汇编语言中,寄存器用名字来访问. CPU 寄存器有好几类, 分别有不同的用处:

1. 通用寄存器:
EAX,EBX,ECX,EDX,ESI,EDI,EBP,ESP(这个虽然通用,但很少被用做除了堆栈指针外的用途)

这些 32 位可以被用作多种用途,但每一个都有"专长". EAX 是"累加器"(accumulator), 它是很多加法乘法指令的缺省寄存器. EBX 是"基地址"(base)寄存器, 在内存寻址时存放基地址. ECX 是计数器(counter), 是重复(REP)前缀指令和 LOOP 指令的内定计数器. EDX 是...(忘了..哈哈)但它总是被用来放整数除法产生的余数. 这 4 个寄存器的低 16 位可以被单独访问,分别用 AX,BX,CX 和 DX. AX 又可以单独访问低 8 位(AL)和高 8 位(AH), BX,CX,DX 也类似. 函数的返回值经常被放在 EAX 中.

ESI/EDI 分别叫做"源/目标索引寄存器"(source/destination index),因为在很多字符串操作指令中, DS:ESI 指向源串,而 ES:EDI 指向目标串.

EBP 是"基址指针"(BASE POINTER), 它最经常被用作高级语言函数调用的"框架指针"(frame pointer). 在破解的时候,经常可以看见一个标准的函数起始代码:

```
push ebp ;保存当前 ebp
mov ebp,esp ;EBP 设为当前堆栈指针
sub esp, xxx ;预留 xxx 字节给函数临时变量.
...
```

这样一来,EBP 构成了该函数的一个框架, 在 EBP 上方分别是原来的 EBP, 返回地址和参数. EBP 下方则是临时变量. 函数返回时作 mov esp,ebp/pop ebp/ret 即可.

ESP 专门用作堆栈指针.

2. 段寄存器:

CS(Code Segment，代码段) 指定当前执行的代码段. EIP (Instruction pointer, 指令指针)则指向该段中一个具体的指令. CS:EIP 指向哪个指令, CPU 就执行它. 一般只能用 jmp, ret, jnz, call 等指令来改变程序流程,而不能直接对它们赋值.

DS(DATA SEGMENT, 数据段) 指定一个数据段. 注意:在当前的计算机系统中, 代码和数据没有本质差别, 都是一串二进制数, 区别只在于你如何用它. 例如, CS 制定的段总是被用作代码, 一般不能通过 CS 指定的地址去修改该段. 然而,你可以为同一个段申请一个数据段描述符"别名"而通过 DS 来访问/修改. 自修改代码的程序常如此做.

ES,FS,GS 是辅助的段寄存器, 指定附加的数据段.

SS(STACK SEGMENT)指定当前堆栈段. ESP 则指出该段中当前的堆栈顶. 所有 push/pop 系列指令都只对 SS:ESP 指出的地址进行操作.

3. 标志寄存器(EFLAGS):

该寄存器有 32 位,组合了各个系统标志. EFLAGS 一般不作为整体访问, 而只对单一的标志位感兴趣. 常用的标志有:

进位标志 C(CARRY), 在加法产生进位或减法有借位时置 1, 否则为 0.
零标志 Z(ZERO), 若运算结果为 0 则置 1, 否则为 0
符号位 S(SIGN), 若运算结果的最高位置 1, 则该位也置 1.
溢出标志 O(OVERFLOW), 若(带符号)运算结果超出可表示范围, 则置 1.

JXX 系列指令就是根据这些标志来决定是否要跳转, 从而实现条件分枝. 要注意,很多 JXX 指令是等价的, 对应相同的机器码. 例如, JE 和 JZ 是一样的,都是当 Z=1 是跳转. 只有 JMP 是无条件跳转. JXX 指令分为两组, 分别用于无符号操作和带符号操作. JXX 后面的"XX" 有如下字母:

无符号操作: 带符号操作:
A = "ABOVE", 表示"高于" G = "GREATER", 表示"大于"
B = "BELOW", 表示"低于" L = "LESS", 表示"小于"
C = "CARRY", 表示"进位"或"借位" O = "OVERFLOW", 表示"溢出"
S = "SIGN", 表示"负"
通用符号:
E = "EQUAL" 表示"等于", 等价于 Z (ZERO)
N = "NOT" 表示"非", 即标志没有置位. 如 JNZ "如果 Z 没有置位则跳转"
Z = "ZERO", 与 E 同.

如果仔细想一想,就会发现 JA = JNBE, JAE = JNB, JBE = JNA, JG = JNLE, JGE= JNL, JL= JNGE, ...

4. 端口

端口是直接和外部设备通讯的地方。外设接入系统后，系统就会把外设的数据接口映射到特定的端口地址空间，这样，从该端口读入数据就是从外设读入数据，而向外设写入数据就是向端口写入数据。当然这一切都必须遵循外设的工作方式。端口的地址空间与内存地址空间无关，系统总共提供对 64K 个 8 位端口的访问，编号 0-65535. 相邻的 8 位端口可以组成成一个 16 位端口，相邻的 16 位端口可以组成一个 32 位端口。端口输入输出由指令 IN,OUT,INS 和 OUTS 实现，具体可参考汇编语言书籍。

[转]汇编语言的准备知识--给初次接触汇编者 2

2008-01-15 15:29

汇编指令的操作数可以是内存中的数据， 如何让程序从内存中正确取得所需要的数据就是对内存的寻址。

INTEL 的 CPU 可以工作在两种寻址模式:实模式和保护模式。 前者已经过时，就不讲了， WINDOWS 现在是 32 位保护模式的系统， PE 文件就基本是运行在一个 32 位线性地址空间， 所以这里就只介绍 32 位线性空间的寻址方式。

其实线性地址的概念是很直观的， 就想象一系列字节排成一长队，第一个字节编号为 0， 第二个编号位 1， 。。。。 一直到 4294967295(十六进制 FFFFFFFF，这是 32 位二进制数所能表达的最大值了)。 这已经有 4GB 的容量! 足够容纳一个程序所有的代码和数据。 当然， 这并不表示你的机器有那么多内存。 物理内存的管理和分配是很复杂的内容， 初学者不必在意， 总之， 从程序本身的角度看， 就好象是在那么大的内存中。

在 INTEL 系统中， 内存地址总是由"段选择符:有效地址"的方式给出。段选择符(SELECTOR)存放在某一个段寄存器中， 有效地址则可由不同的方式给出。 段选择符通过检索段描述符确定段的起始地址， 长度(又称段限制)， 粒度， 存取权限， 访问性质等。 先不用深究这些， 只要知道段选择符可以确定段的性质就行了。 一旦由选择符确定了段， 有效地址相对于段的基地址开始算。 比如由选择符 1A7 选择的数据段， 其基地址是 400000， 把 1A7 装入 DS 中， 就确定使用该数据段。 DS:0 就指向线性地址 400000。 DS:1F5278 就指向线性地址 5E5278。 我们在一般情况下， 看不到也不需要看到段的起始地址， 只需要关心在该段中的有效地址就行了。 在 32 位系统中， 有效地址也是由 32 位数字表示， 就是说， 只要有一个段就足以涵盖 4GB 线性地址空间， 为什么还要有不同的段选择符呢? 正如前面所说的， 这是为了对数据进行不同性质的访问。 非法的访问将产生异常中断， 而这正是保护模式的核心内容， 是构造优先级和多任务系统的基础。 这里有涉及到很多深层的东西， 初学者先可不必理会。

有效地址的计算方式是: 基址 间址*比例因子 偏移量。 这些量都是指段内的相对于段起始地址的量度， 和段的起始地址没有关系。 比如， 基址=100000， 间址=400， 比例因子=4， 偏移量=20000， 则有效地址为:

100000 400*4 20000=100000 1000 20000=121000。 对应的线性地址是 400000 121000=521000。 (注意， 都是十六进制数)。

基址可以放在任何 32 位通用寄存器中， 间址也可以放在除 ESP 外的任何一个通用寄存器中。 比例因子可以是 1， 2， 4 或 8。 偏移量是立即数。 如: [EBP EDX*8 200]就是一个有效的有效地址表达式。 当然， 多数情况下用不着这么复杂， 间址，比例因子和偏移量不一定要出现。

内存的基本单位是字节(BYTE)。 每个字节是 8 个二进制位， 所以每个字节能表示的最大的数是 11111111， 即十进制的 255。 一般来说， 用十六进制比较方便， 因为每 4 个二进制位刚好等于 1 个十六进制位， 11111111b = 0xFF。 内存中的字节是连续存放的， 两个字节构成一个字(WORD)， 两个字构成一个双字(DWORD)。 在 INTEL 架构中，采用 small endian 格式， 即在内存中，高位字节在低位字节后面。 举例说明:十六进制数

803E7D0C， 每两位是一个字节， 在内存中的形式是: 0C 7D 3E 80。 在 32 位寄存器中则是正常形式，如在 EAX 就是 803E7D0C。 当我们的形式地址指向这个数的时候，实际上是指向第一个字节，即 0C。 我们可以指定访问长度是字节， 字或者双字。 假设 DS:[EDX]指向第一个字节 0C:

```
mov AL， byte ptr DS:[EDX] ;把字节 0C 存入 AL
mov AX， word ptr DS:[EDX] ;把字 7D0C 存入 AX
mov EAX， dword ptr DS:[EDX] ;把双字 803E7D0C 存入 EAX
```

在段的属性中，有一个就是缺省访问宽度。如果缺省访问宽度为双字(在 32 位系统中经常如此)，那么要进行字节或字的访问，就必须用 byte/word ptr 显式地指明。

缺省段选择：如果指令中只有作为段内偏移的有效地址，而没有指明在哪一个段里的时候，有如下规则：

如果用 ebp 和 esp 作为基址或间址，则认为是在 SS 确定的段中；
其他情况，都认为是在 DS 确定的段中。

如果想打破这个规则，就必须使用段超越前缀。举例如下：

```
mov eax， dword ptr [edx] ;缺省使用 DS，把 DS:[EDX]指向的双字送入 eax
mov ebx， dword ptr ES:[EDX] ;使用 ES:段超越前缀，把 ES:[EDX]指向的双字送入 ebx
```

堆栈：

堆栈是一种数据结构，严格地应该叫做“栈”。“堆”是另一种类似但不同的结构。SS 和 ESP 是 INTEL 对栈这种数据结构的硬件支持。push/pop 指令是专门针对栈结构的特定操作。SS 指定一个段为栈段，ESP 则指出当前的栈顶。push xxx 指令作如下操作：

把 ESP 的值减去 4；
把 xxx 存入 SS:[ESP]指向的内存单元。

这样，esp 的值减小了 4，并且 SS:[ESP]指向新压入的 xxx。 所以栈是“倒着长”的，从高地址向低地址方向扩展。pop yyy 指令做相反的操作，把 SS:[ESP]指向的双字送到 yyy 指定的寄存器或内存单元，然后把 esp 的值加上 4。这时，认为该值已被弹出，不再在栈上了，因为它虽然还暂时存在在原来的栈顶位置，但下一个 push 操作就会把它覆盖。因此，在栈段中地址低于 esp 的内存单元中的数据均被认为是未定义的。

最后，有一个要注意的事实是，汇编语言是面向机器的，指令和机器码基本上是一一对应的，所以它们的实现取决于硬件。有些看似合理的指令实际上是不存在的，比如:

```
mov DS:[edx]， ds:[ecx] ;内存单元之间不能直接传送
mov DS， 1A7 ;段寄存器不能直接由立即数赋值
mov EIP， 3D4E7 ;不能对指令指针直接操作。
```

[转]汇编语言的准备知识--给初次接触汇编者 3

2008-01-15 15:30

“汇编语言”作为一门语言，对应于高级语言的编译器，我们需要一个“汇编器”来把汇编语言原文件汇编成机器可执行的代码。高级的汇编器如 MASM, TASM 等等为我们写汇编程序提供了很多类似于高级语言的特征，比如结构化、抽象等。在这样的环境中编写的汇编程序，有很大一部分是面向汇编器的伪指令，已经类同于高级语言。现在的汇编环境已经如此高级，即使全部用汇编语言来编写 windows 的应用程序也是可行的，但这不是汇编语言的长处。汇编语言的长处在于编写高效且需要对机器硬件精确控制的程序。而且我想这里的人学习汇编的目的多半是为了在破解时看懂反汇编代码，很少有人真的要拿汇编语言编程序吧？（汗.....）

好了，言归正传。大多数汇编语言书都是面向汇编语言编程的，我的帖是面向机器和反汇编的，希望能起到相辅相成的作用。有了前面两篇的基础，汇编语言书上对大多数指令的介绍应该能够看懂、理解了。这里再讲一讲一些常见而操作比较复杂的指令。我这里讲的都是机器的硬指令，不针对任何汇编器。

无条件转移指令 jmp:

这种跳转指令有三种方式：短(short)，近(near)和远(far)。短是指要跳至的目标地址与当前地址前后相差不超过 128 字节。近是指跳转的目标地址与当前地址在用一个段内，即 CS 的值不变，只改变 EIP 的值。远指跳到另一个代码段去执行，CS/EIP 都要改变。短和近在编码上有所不同，在汇编指令中一般很少显式指定，只要写 jmp 目标地址，几乎任何汇编器都会根据目标地址的距离采用适当的编码。远转移在 32 位系统中很少见到，原因前面已经讲过，由于有足够的线性空间，一个程序很少需要两个代码段，就连用到的系统模块也被映射到同一个地址空间。

jmp 的操作数自然是目标地址，这个指令支持直接寻址和间接寻址。间接寻址又可分为寄存器间接寻址和内存间接寻址。举例如下(32 位系统):

```
jmp 8E347D60 ;直接寻址段内跳转
jmp EBX ;寄存器间接寻址：只能段内跳转
jmp dword ptr [EBX] ;内存间接寻址，段内跳转
jmp dword ptr [00903DEC] ;同上
jmp fward ptr [00903DF0] ;内存间接寻址，段间跳转
```

解释：

在 32 位系统中，完整目标地址由 16 位段选择子和 32 位偏移量组成。因为寄存器的宽度是 32 位，因此寄存器间接寻址只能给出 32 位偏移量，所以只能是段内近转移。在内存间接寻址时，指令后面是方括号内的有效地址，在这个地址上存放跳转的目标地址。比如，在[00903DEC]处有如下数据：7C 82 59 00 A7 01 85 65 9F 01

内存字节是连续存放的，如何确定取多少作为目标地址呢？dword ptr 指明该有效地址指明的是双字，所以取

0059827C 作段内跳转。反之，fward ptr 指明后面的有效地址是指向 48 位完全地址，所以取 19F:658501A7 做远跳转。

注意：在保护模式下，如果段间转移涉及优先级的变化，则有一系列复杂的保护检查，现在可不加理会。将来等各位功力提升以后可以自己去学习。

条件转移指令 jxx:只能作段内转移，且只支持直接寻址。

==========

调用指令 CALL:

Call 的寻址方式与 jmp 基本相同，但为了从子程序返回，该指令在跳转以前会把紧接着它的下一条指令的地址压进堆栈。如果是段内调用（目标地址是 32 位偏移量），则压入的也只是一个偏移量。如果是段间调用（目标地址是 48 位全地址），则也压入下一条指令的完全地址。同样，如果段间转移涉及优先级的变化，则有一系列复杂的保护检查。

与之对应 retn/retf 指令则从子程序返回。它从堆栈上取得返回地址（是 call 指令压进去的）并跳到该地址执行。retn 取 32 位偏移量作段内返回，retf 取 48 位全地址作段间返回。retn/f 还可以跟一个立即数作为操作数，该数实际上是从堆栈上传给子程序的参数的个数（以字计）返回后自动把堆栈指针 esp 加上指定的数*2，从而丢弃堆栈中的参数。这里具体的细节留待下一篇讲述。

虽然 call 和 ret 设计为一起工作，但它们之间没有必然的联系。就是说，如果你直接用 push 指令向堆栈中压入一个数，然后执行 ret，他同样会把你压入的数作为返回地址，而跳到那里去执行。这种非正常的流程转移可以被用作反跟踪手段。

==========

中断指令 INT n

在保护模式下，这个指令必定会被操作系统截获。在一般的 PE 程序中，这个指令已经不太见到了，而在 DOS 时代，中断是调用操作系统和 BIOS 的重要途径。现在的程序可以文质彬彬地用名字来调用 windows 功能，如 call user32!getwindowtexta。从程序角度看，INT 指令把当前的标志寄存器先压入堆栈，然后把下一条指令的完全地址也压入堆栈，最后根据操作数 n 来检索“中断描述符表”，试图转移到相应的中断服务程序去执行。通常，中断服务程序都是操作系统的核心代码，必然会涉及到优先级转换和保护性检查、堆栈切换等等，细节可以看一些高级的教程。

与之相应的中断返回指令 IRET 做相反的操作。它从堆栈上取得返回地址，并用来设置 CS:EIP,然后从堆栈中弹出标志寄存器。注意，堆栈上的标志寄存器值可能已经被中断服务程序所改变，通常是进位标志 C, 用来表示功能是否正常完成。同样的，IRET 也不一定非要和 INT 指令对应，你可以自己在堆栈上压入标志和地址，然后执行 IRET 来实现流程转移。实际上，多任务操作系统常用此伎俩来实现任务转换。

广义的中断是一个很大的话题，有兴趣可以去查阅系统设计的书籍。

==========

装入全指针指令 LDS,LES,LFS,LGS,LSS

这些指令有两个操作数。第一个是一个通用寄存器，第二个操作数是一个有效地址。指令从该地址取得 48 位全指针，将选择符装入相应的段寄存器，而将 32 位偏移量装入指定的通用寄存器。注意在内存中，指针的存放形式总是 32 位偏移量在前面，16 位选择符在后面。装入指针以后，就可以用 DS:[ESI]这样的形式来访问指针指向的数据了。

==========

字符串操作指令

这里包括 CMPS,SCAS,LODS,STOS,MOVS,INS 和 OUTS 等。这些指令有一个共同的特点，就是没有显式的操作数，而由硬件规定使用 DS:[ESI]指向源字符串，用 ES:[EDI]指向目的字符串，用 AL/AX/EAX 做暂存。这是硬件规定的，所以在使用这些指令之前一定要设好相应的指针。

这里每一个指令都有 3 种宽度形式，如 CMPSB(字节比较)、CMPSW(字比较)、CMPSD(双字比较)等。

CMPSB:比较源字符串和目标字符串的第一个字符。若相等则 Z 标志置 1。若不等则 Z 标志置 0。指令执行完后，ESI 和 EDI 都自动加 1，指向源/目标串的下一个字符。如果用 CMPSW,则比较一个字，ESI/EDI 自动加 2 以指向下一个字。

如果用 CMPSD,则比较一个双字，ESI/EDI 自动加 4 以指向下一个双字。（在这一点上这些指令都一样，不再赘述）

SCAB/W/D 把 AL/AX/EAX 中的数值与目标串中的一个字符/字/双字比较。

LODSB/W/D 把源字符串中的一个字符/字/双字送入 AL/AX/EAX

STOSB/W/D 把 AL/AX/EAX 中的直送入目标字符串中

MOVSB/W/D 把源字符串中的字符/字/双字复制到目标字符串

INSB/W/D 从指定的端口读入字符/字/双字到目标字符串中，端口号码由 DX 寄存器指定。

OUTSB/W/D 把源字符串中的字符/字/双字送到指定的端口，端口号码由 DX 寄存器指定。

串操作指令经常和重复前缀 REP 和循环指令 LOOP 结合使用以完成对整个字符串的操作。而 REP 前缀和 LOOP 指令都有硬件规定用 ECX 做循环计数器。举例：

```
LDS ESI,SRC_STR_PTR
LES EDI,DST_STR_PTR
MOV ECX,200
REP MOVSD
```

上面的代码从 SRC_STR 拷贝 200 个双字到 DST_STR. 细节是：REP 前缀先检查 ECX 是否为 0，若否则执行一次 MOVSD,ECX 自动减 1，然后执行第二轮检查、执行..... 直到发现 ECX=0 便不再执行 MOVSD,结束重复而执行下面的指令。

```
LDS ESI,SRC_STR_PTR
MOV ECX,100
LOOP1:
```

LODSW
.... (deal with value in AX)

LOOP LOOP1
.....

从 SRC_STR 处理 100 个字。同样，LOOP 指令先判断 ECX 是否为零，来决定是否循环。每循环一轮 ECX 自动减 1。

REP 和 LOOP 都可以加上条件，变成 REPZ/REPNZ 和 LOOPZ/LOOPNZ. 这是除了 ECX 外，还用检查零标志 Z. REPZ 和 LOOPZ 在 Z 为 1 时继续循环，否则退出循环，即使 ECX 不为 0。REPNZ/LOOPNZ 则相反。

[转]汇编语言的准备知识--给初次接触汇编者 4

2008-01-15 15:30

高级语言程序的汇编解析

在高级语言中，如 C 和 PASCAL 等等，我们不再直接对硬件资源进行操作，而是面向于问题的解决，这主要体现在数据抽象化和程序的结构化。例如我们用变量名来存取数据，而不再关心这个数据究竟在内存的什么地方。这样，对硬件资源的使用方式完全交给了编译器去处理。不过，一些基本的规则还是存在的，而且大多数编译器都遵循一些规范，这使得我们在阅读反汇编代码的时候日子好过一点。这里主要讲讲汇编代码中一些和高级语言对应的地方。

1. 普通变量。通常声明的变量是存放在内存中的。编译器把变量名和一个内存地址联系起来（这里要注意的是，所谓的“确定的地址”是对编译器而言在编译阶段算出的一个临时的地址。在连接成可执行文件并加载到内存中执行的时候要进行重定位等一系列调整，才生成一个实时的内存地址，不过这并不影响程序的逻辑，所以先不必太在意这些细节，只要知道所有的函数名字和变量名字都对应一个内存的地址就行了），所以变量名在汇编代码中就表现为一个有效地址，就是放在方括号中的操作数。例如，在 C 文件中声明：

int my_age;

这个整型的变量就存在一个特定的内存位置。语句 my_age= 32; 在反汇编代码中可能表现为：

mov word ptr [007E85DA], 20

所以在方括号中的有效地址对应的是变量名。又如：

char my_name[11] = "lianzi2000";

这样的说明也确定了一个地址，对应于 my_name. 假设地址是 007E85DC,则内存中 [007E85DC]='l',[007E85DD]='i', etc. 对 my_name 的访问也就是对这地址处的数据访问。

指针变量其本身也同样对应一个地址,因为它本身也是一个变量。如：

char *your_name;

这时也确定变量"your_name"对应一个内存地址，假设为 007E85F0. 语句 your_name=my_name;很可能表现为：

mov [007E85F0], 007E85DC ;your_name 的内容是 my_name 的地址。

2. 寄存器变量

在 C 和 C 中允许说明寄存器变量。register int i; 指明 i 是寄存器存放的整型变量。通常，编译器都把寄存器变量放在 esi 和 edi 中。寄存器是在 cpu 内部的结构，对它的访问要比内存快得多，所以把频繁使用的变量放在寄存器中可以提高程序执行速度。

3. 数组

不管是多少维的数组，在内存中总是把所有的元素都连续存放，所以在内存中总是一维的。例如，int i_array[2][3]; 在内存确定了一个地址，从该地址开始的 12 个字节用来存贮该数组的元素。所以变量名 i_array 对应着该数组的起始地址，也即是指向数组的第一个元素。存放的顺序一般是 i_array[0][0],[0][1],[0][2],[1][0],[1][1],[1][2] 即最右边的下标变化最快。当需要访问某个元素时，程序就会从多维索引值换算成一维索引，如访问 i_array[1][1],换算成内存中的一维索引值就是 1*3 1=4.这种换算可能在编译的时候就可以确定，也可能要到运行时才可以确定。无论如何，如果我们把 i_array 对应的地址装入一个通用寄存器作为基址，则对数组元素的访问就是一个计算有效地址的问题：

; i_array[1][1]=0x16

lea ebx,xxxxxxxx ;i_array 对应的地址装入 ebx
mov edx,04 ;访问 i_array[1][1]，编译时就已经确定
mov word ptr [ebx edx*2], 16 ;

当然，取决于不同的编译器和程序上下文，具体实现可能不同，但这种基本的形式是确定的。从这里也可以看到比例因子的作用（还记得比例因子的取值为 1，2，4 或 8 吗？），因为在目前的系统中简单变量总是占据 1,2,4 或者 8 个字节的长度，所以比例因子的存在为在内存中的查表操作提供了极大方便。

4. 结构和对象

结构和对象的成员在内存中也都连续存放，但有时为了在字边界或双字边界对齐，可能有些微调整，所以要确定对象的大小应该用 sizeof 操作符而不应该把成员的大小相加来计算。当我们声明一个结构变量或初始化一个对象时，这个结构变量和对象的名字也对应一个内存地址。举例说明：

struct tag_info_struct
{

```
int age;
int sex;
float height;
float weight;
} marry;
```

变量 marry 就对应一个内存地址。在这个地址开始，有足够多的字节(sizeof(marry))容纳所有的成员。每一个成员则对应一个相对于这个地址的偏移量。这里假设此结构中所有的成员都连续存放，则 age 的相对地址为 0，sex 为 2, height 为 4,weight 为 8。

```
; marry.sex=0;

lea ebx,xxxxxxxx ;marry 对应的内存地址
mov word ptr [ebx 2], 0
.....
```

对象的情况基本相同。注意成员函数具体的实现在代码段中，在对象中存放的是一个指向该函数的指针。

5. 函数调用

一个函数在被定义时，也确定一个内存地址对应于函数名字。如：

```
long comb(int m, int n)
{
long temp;
.....

return temp;
}
```

这样，函数 comb 就对应一个内存地址。对它的调用表现为：

CALL xxxxxxxx ;comb 对应的地址。这个函数需要两个整型参数，就通过堆栈来传递：

```
；lresult=comb(2,3);

push 3
push 2
call xxxxxxxx
mov dword ptr [yyyyyyyy], eax ;yyyyyyyy 是长整型变量 lresult 的地址
```

这里请注意两点。第一，在 C 语言中，参数的压栈顺序是和参数顺序相反的，即后面的参数先压栈，所以先执行 push 3. 第二，在我们讨论的 32 位系统中，如果不指明参数类

型，缺省的情况就是压入 32 位双字。因此，两个 push 指令总共压入了两个双字，即 8 个字节的数据。然后执行 call 指令。call 指令又把返回地址，即下一条指令(mov dword ptr....)的 32 位地址压入，然后跳转到 xxxxxxxx 去执行。

在 comb 子程序入口处(xxxxxxxx)，堆栈的状态是这样的：

03000000 （请回忆 small endian 格式）
02000000
yyyyyyyy <--ESP 指向返回地址

前面讲过，子程序的标准起始代码是这样的：

push ebp ;保存原先的 ebp
mov ebp, esp;建立框架指针
sub esp, XXX;给临时变量预留空间
.....

执行 push ebp 之后，堆栈如下：

03000000
02000000
yyyyyyyy
old ebp <---- esp 指向原来的 ebp

执行 mov ebp,esp 之后，ebp 和 esp 都指向原来的 ebp. 然后 sub esp, xxx 给临时变量留空间。这里，只有一个临时变量 temp,是一个长整数，需要 4 个字节，所以 xxx=4。这样就建立了这个子程序的框架：

03000000
02000000
yyyyyyyy
old ebp <---- 当前 ebp 指向这里
temp

所以子程序可以用[ebp 8]取得第一参数(m),用[ebp C]来取得第二参数(n)，以此类推。临时变量则都在 ebp 下面，如这里的 temp 就对应于[ebp-4].

子程序执行到最后，要返回 temp 的值：

mov eax,[ebp-04]
然后执行相反的操作以撤销框架：

mov esp,ebp ;这时 esp 和 ebp 都指向 old ebp,临时变量已经被撤销
pop ebp ;撤销框架指针，恢复原 ebp.

这是 esp 指向返回地址。紧接的 retn 指令返回主程序：

retn 4

该指令从堆栈弹出返回地址装入 EIP,从而返回到主程序去执行 call 后面的指令。同时调整 esp(esp=esp 4*2),从而撤销参数，使堆栈恢复到调用子程序以前的状态，这就是堆栈的平衡。调用子程序前后总是应该维持堆栈的平衡。从这里也可以看到，临时变量 temp 已经随着子程序的返回而消失，所以试图返回一个指向临时变量的指针是非法的。

为了更好地支持高级语言，INTEL 还提供了指令 Enter 和 Leave 来自动完成框架的建立和撤销。Enter 接受两个操作数，第一个指明给临时变量预留的字节数，第二个是子程序嵌套调用层数，一般都为 0。enter xxx,0 相当于:

```
push ebp
mov ebp,esp
sub esp,xxx
```

leave 则相当于：

```
mov esp,ebp
pop ebp
```