2020/08/03_Windows编程_第12课_钩子(Hook)、服务(Server)

笔记本: Windows编程
创建时间: 2020/8/3 星期一 10:25
作者: ileemi

课前作业讲评

MFC dll 需要注意模块状态（模块状态不正确，程序可能出现异常）
切换模块状态API -- AfxGetStaticModuleState()
AFX_MANAGE_STATE(AfxGetStaticModuleState());

非模块对话框会用主线程的消息循环

使用SDK注册dll时最稳定的

FreeLibraryAndExitThread 卸载指定的代码不会返回，直接调用ExitThread退出。

钩子（Hook）

> 修改过程函数Windows有专业的API来进行监控。
> HOOK -- 钩子（好比高速上人工架设一个收费站（非法），达到自己的目的）

> SetWindowsHookEx：可以监控操作系统所有的消息，拦截各种消息，或者指定窗口的各种消息（仅仅是通知，对原来程序的消息不能影响），可以拦截自己的程序，也可以拦截别人的程序以及未创建产生的窗口的消息。

函数说明:
SetWindowsHookEx 函数将应用程序定义的钩子子程安装到钩子链中。您需要安装一个钩子子程来监视系统中某些类型的事件。这些事件要么与特定的线程相关联，要么与同一桌面中作为调用线程的所有线程相关联。

> 函数定义:

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

> 参数:
> 参数1：指定要安装的钩子子程的类型（拦截消息的时机）。WH_CALLWNDPROC 拦截所有消息。
> 参数2：指向钩子子程的指针（回调函数 CallWndProc ）。如果dwThreadId参数为0或者指定了一个由不同进程创建的线程的标识符，那么lpfn参数必须指向动态链接库(DLL)中的一个钩子子程。否则，lpfn可以指向与当前进程关联的代码中的一个钩子子程。
> 参数3：包含lpfn参数所指向的钩子子程的DLL句柄。如果dwThreadId参数指定了当前进程创建的线程，并且钩子子程在与当前进程关联的代码中，那么hMod参数必须设置为NULL。
> 参数4：指定要与钩子子程关联的线程的标识符。如果该参数为0，则该钩子子程与作为调用线程在同一桌面中运行的所有现有线程相关联。

> 消息是以线程为单位的，不同的线程有不同的消息队列。

安装钩子

API -- SetWindowsHookEx

卸载钩子

删除由 SetWindowsHookEx 函数安装在钩子链中的钩子子程:
API -- UnhookWindowsHookEx

代码示例:

```
//  保存创建钩子的返回值
HHOOK  m_hHook;

/*
CallWndProc钩子进程是一个与SetWindowsHookEx函数
一起使用的应用程序定义的或库定义的回调函数
*/
LRESULT  CALLBACK  CallWndProc(
    int  nCode,
    WPARAM  wParam,
    LPARAM  lParam
)
```

```
{
    //  指向CWPSTRUCT结构的指针，该结构包含有关消息的详细信息
    PCWPSTRUCT  pCWP  =  (PCWPSTRUCT)lParam;
    //  拦截WM_COMMAND消息
    if  (pCWP->message    ==  WM_COMMAND)
    {
        //SetWindowLong();
        //  哪个窗口句柄产生了哪些消息
        TRACE("[Test]  CallWndProc1  WM_COMMAND  hwnd：%p,  message：
%p",
        pCWP->hwnd,  pCWP->message);
    }

    //  拦截按键消息，  wParam  --  为虚拟码
    //TRACE("[Test]  WM_KEYDOWN  hwnd  =  %p,  KEY  =  %c",  pCWP-
>hwnd,  wParam);
    //switch  (pCWP->message)
    //{
    //case  WM_LBUTTONDOWN:
    //    TRACE("pCWP_WM_LBUTTONDOWN");
    //    break;
    //}

    //  将钩子信息传递给当前钩子链中的下一个钩子子程
    return  CallNextHookEx(m_hHook,  nCode,  wParam,  lParam);
}
//  拦截自己的程序
void  CHOOKDlg::OnBnClickedButton1()
{
    //  创建一个钩子
    m_hHook  =  ::SetWindowsHookEx(
        WH_CALLWNDPROC,  //  拦截所有消息
        //WH_KEYBOARD,    //  拦截程序中的按键消息
        CallWndProc,
        NULL,  //  应用程序实例句柄，  HOOK自己填写NULL即可
        ::GetCurrentThreadId()    //  获取主线程
        );

    //  判断钩子是否创建成功
    if  (m_hHook  !=  NULL)
    {
        AfxMessageBox("安装钩子1成功");
    }
    else
    {
        AfxMessageBox("安装钩子1失败");
    }
```

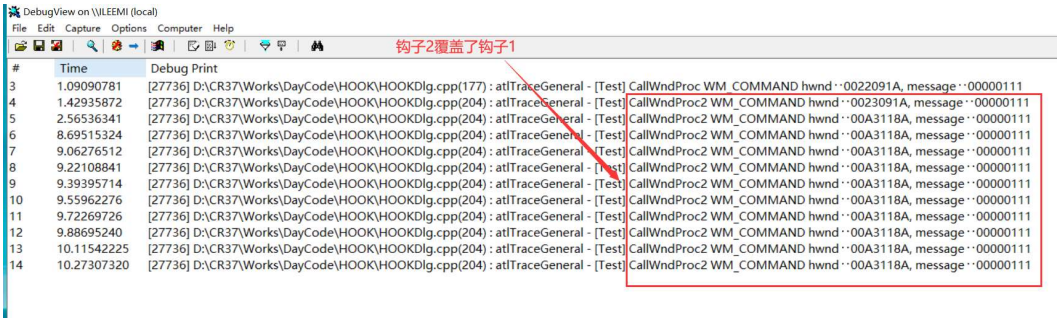
```
}

// 卸载钩子
void CHOOKDlg::OnBnClickedButton2()
{
    if (m_hHook != NULL || m_hHook2 != NULL)
    {
        // 检车钩子是否卸载成功
        if (UnhookWindowsHookEx(m_hHook))
        {
            AfxMessageBox("卸载钩子1成功");
        }
        else
        {
            AfxMessageBox("卸载钩子1失败");
        }
    }
}
```

用来监控进程中线程产生的消息，不能拦截对应的消息，如果非要拦截对应的消息，就需要修改过程函数。

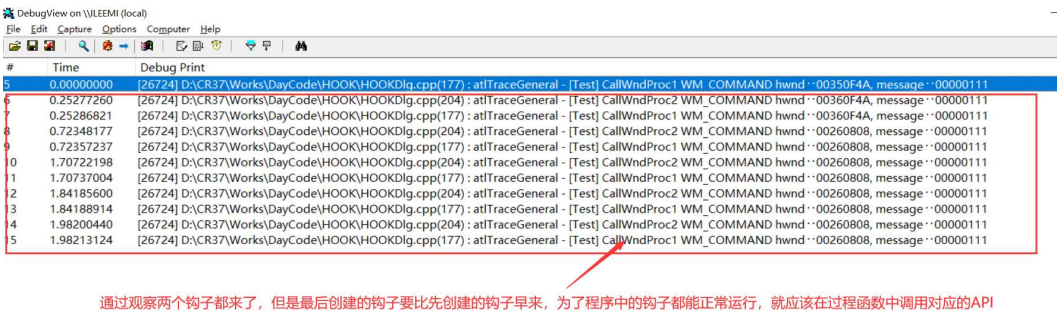
拦截消息

> 当在一个程序中创建多个钩子的时候，需要在钩子对应的过程函数中使用API -- CallNextHookEx 去调用另一个钩子
> CallNextHookEx -- 将钩子信息传递给当前钩子链中的下一个钩子子程。钩子子程可以在处理钩子信息之前或之后调用这个函数。

没使用 CallNextHookEx 前：



使用 CallNextHookEx 后：



> 通过观察两个钩子都来了，但是最后创建的钩子要比先创建的钩子早来，为了程序中的钩子都能正常运行，就应该在过程函数中调用 CallNextHookEx 函数来将钩子信息传递给当前钩子链中的下一个钩子子程。
> 当不使用 CallNextHookEx 函数，但是进程中的钩子又有多个的时候，有的钩子就不能正常运行。

代码示例：

```
LRESULT CALLBACK CallWndProc2(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
)
{
  // 指向CWPSTRUCT结构的指针，该结构包含有关消息的详细信息
  PCWPSTRUCT pCWP = (PCWPSTRUCT)lParam;
  // 拦截WM_COMMAND消息
  if (pCWP->message == WM_COMMAND)
  {
    // 哪个窗口句柄产生了哪些消息
    TRACE("[Test] CallWndProc2 WM_COMMAND hwnd: %p, message: %p", pCWP->hwnd, pCWP->message);
  }
  // 来将钩子信息传递给当前钩子链中的下一个钩子子程
  return CallNextHookEx(m_hHook, nCode, wParam, lParam);
}
```

钩子的恐怖之处在可以钩取别的进程的线程，也可以钩取整个操作系统的的所有进程。

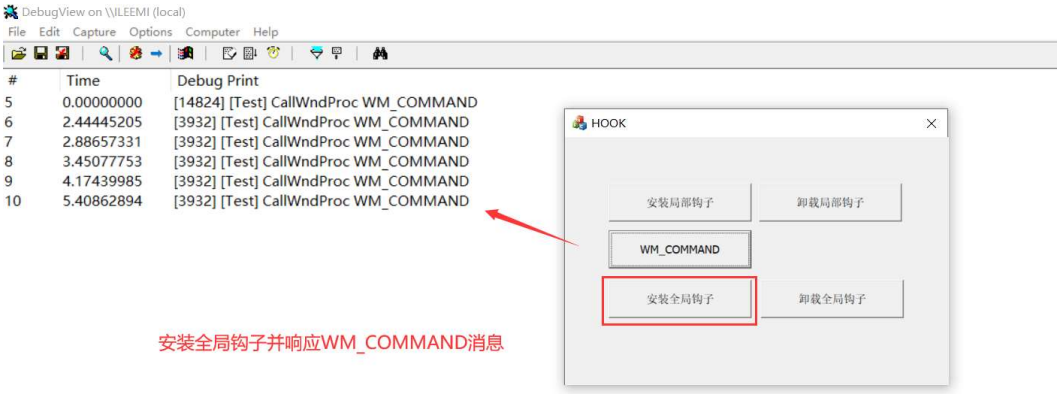
> 回调函数写在本地进程地址中（注册钩子的回调函数在自己的进程中），别的进程中钩子想要调用另一个进程中钩子的回调函数，就需要将回调函数写入到dll中，当需要钩取一个进程的时候，操作系统就会自动将这个dll注入到目标进程中，并且会调用这个回调函数，监控指定的进程，当目标进程创建的时候，系统就会将这个dll注入到目标进程中去。钩子自动注入，不在需要手动注入，一些有操作系统来完成。所有用钩子来完成dll注入就已经习以为常了。创建钩子API函数的第三个参数为Dll的模块基地址，参数2 函数指针就需要填写dll中的回调函数。

全局钩子，对整个操作系统的进程有效

> 安装全局钩子，需要提前写一个dll，函数内需要些dll的回调函数（比较麻烦）。
> 设置全局钩子，第三个参数需要写一个Dll的模块基地址（模块句柄），第二个参数需要写导出函数的地址（写导出函数的地址，就需要在dll中将这个函数进程导出）。为此，可以将 SetWindowsHookEx 放入到dll中，其参数2的过程函

> 数就不要导出，在dll中也不需要获取dll句柄。
> 在dll中封装一个函数导出，在dll中调用 SetWindowsHookEx API, 这样就不用考虑导出函数以及dll句柄的问题。

在dll给两个导出函数，分别用于创建钩子和卸载钩子，同时在dll中还需要添加一个导出函数。



钩取按键消息

WH_KEYBOARD_LL -- 安装一个钩子子程来监视低级键盘输入事件。有关更多信息，请参见LowLevelKeyboardProc钩子子程。参数3指向 KBDLLHOOKSTRUCT 结构的指针，其包含关于低级键盘输入事件的信息。

```
typedef struct tagKBDLLHOOKSTRUCT {
    DWORD     vkCode;
    DWORD     scanCode;
    DWORD     flags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
} KBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT;
```

代码示例：

```
// DII 文件
HHOOK g_hHook;        // 保存创建钩子的句柄
HMODULE g_hModule; // 保存模块句柄

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
  g_hModule = hModule;
    switch (ul_reason_for_call)
```

```
    {
    case DLL_PROCESS_ATTACH:
      ::MessageBox(NULL, "我来了", "Test", MB_OK);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}

LRESULT CALLBACK CallWndProc(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
)
{
  // 指向CWPSTRUCT结构的指针，该结构包含有关消息的详细信息
  PKBDLLHOOKSTRUCT pKBS = (PKBDLLHOOKSTRUCT)lParam;

  // 拦截键盘按键
  char szBuff[MAXBYTE];
  wsprintf(szBuff, "[Test] CallWndProc VK: %c", pKBS->vkCode);
  OutputDebugString(szBuff);
  // 拦截WM_COMMAND消息
  //if (pCWP->message == WM_COMMAND)
  //{
  //  // 哪个窗口句柄产生了哪些消息
  //  OutputDebugString("[Test] CallWndProc WM_COMMAND");
  //}
  // 来将钩子信息传递给当前钩子链中的下一个钩子子程
  return CallNextHookEx(g_hHook, nCode, wParam, lParam);
}

// 创建全局钩子 同时创建.def文件将这个函数进行导出
bool SetHook()
{
  // 拦截所有进程，线程
  g_hHook = SetWindowsHookEx(WH_KEYBOARD_LL, CallWndProc, g_hModule, 0);
    if  (g_hHook  !=  NULL)
    {
        return  true;
    }
    return  false;
}

// 卸载全局钩子
```

```
bool UnSetHook()
{
  return UnhookWindowsHookEx(g_hHook);
}
```

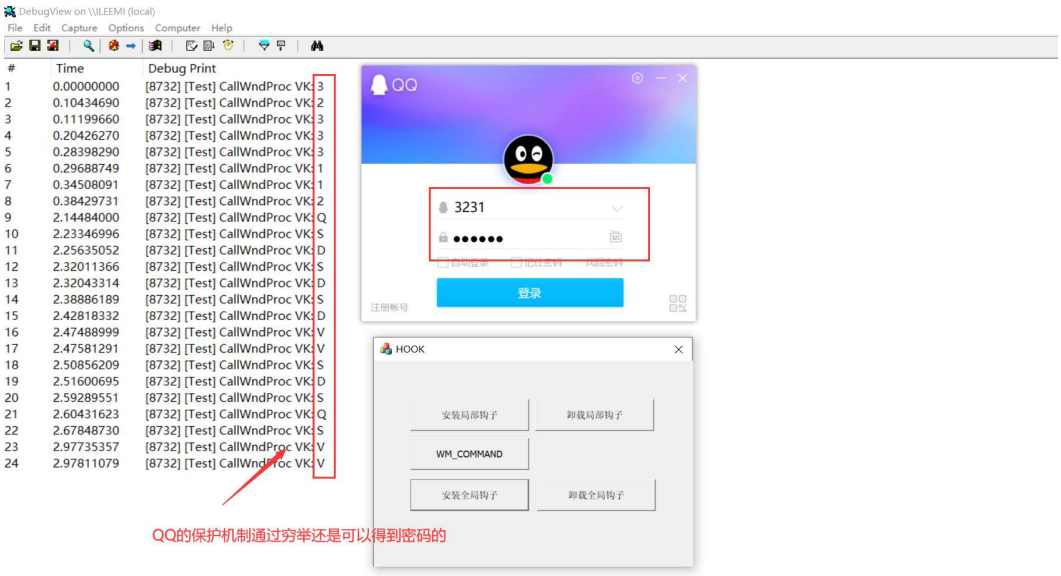
静态使用HookDll，代码示例：

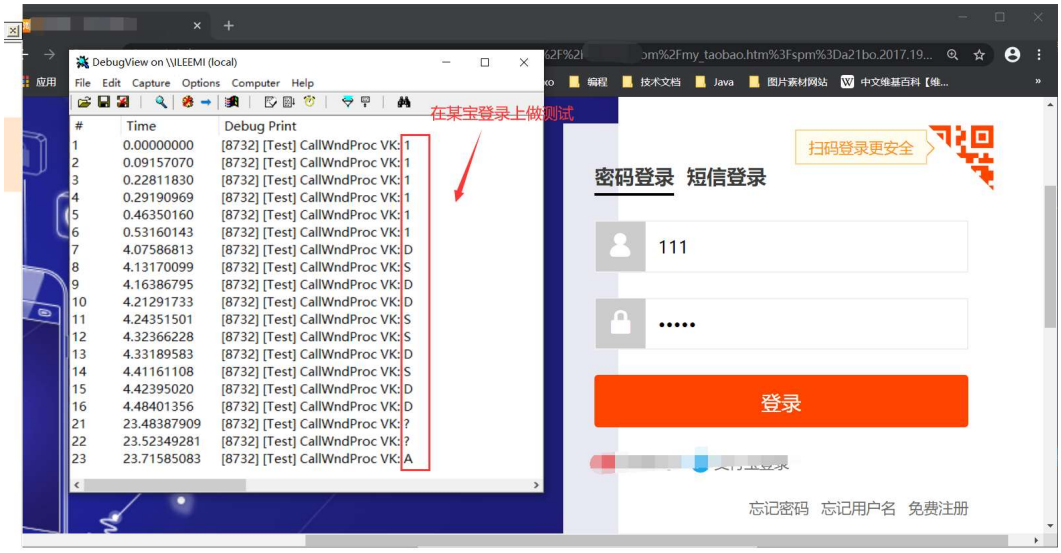
```
// 链接Dll
#pragma comment(lib, "debug/HookDll.lib")
bool SetHook();
bool UnSetHook();

HHOOK m_hHook;
/*
CallWndProc钩子子程是一个与SetWindowsHookEx函数
一起使用的应用程序定义的或库定义的回调函数
*/
LRESULT CALLBACK CallWndProc(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
)
{
  // 指向CWPSTRUCT结构的指针，该结构包含有关消息的详细信息
  PCWPSTRUCT pCWP = (PCWPSTRUCT)lParam;
  // 拦截WM_COMMAND消息
  if (pCWP->message  == WM_COMMAND)
  {
    //SetWindowLong();
    // 哪个窗口句柄产生了哪些消息
    TRACE("[Test] CallWndProc1 WM_COMMAND hwnd: %p, message: %p", pCWP-
>hwnd, pCWP->message);
  }

  // 拦截按键消息， wParam -- 为虚拟码
  //TRACE("[Test] WM_KEYDOWN hwnd = %p, KEY = %c", pCWP->hwnd, wParam);
  //switch (pCWP->message)
  //{
  //case WM_LBUTTONDOWN:
  //  TRACE("pCWP_WM_LBUTTONDOWN");
  //  break;
  //}
  // 来将钩子信息传递给当前钩子链中的下一个钩子子程
  return CallNextHookEx(m_hHook, nCode, wParam, lParam);
}

// 拦截自己的程序
```

```
void CHOOKDlg::OnBnClickedButton1()
{
  // 创建一个钩子
  m_hHook = ::SetWindowsHookEx(
    WH_CALLWNDPROC, // 拦截所有消息
    //WH_KEYBOARD,  // 拦截程序中的按键消息
    CallWndProc,
    NULL, // 应用程序实例句柄, HOOK自己填写NULL即可
    ::GetCurrentThreadId()  // 获取主线程
    );

  // 判断钩子是否创建成功
  if (m_hHook != NULL)
  {
    AfxMessageBox("安装钩子1成功");
  }
  else
  {
    AfxMessageBox("安装钩子1失败");
  }
}

// 卸载钩子
void CHOOKDlg::OnBnClickedButton2()
{
  if (m_hHook != NULL)
  {
    // 检车钩子是否卸载成功
    if (UnhookWindowsHookEx(m_hHook))
    {
      AfxMessageBox("卸载钩子成功");
    }
    else
    {
      AfxMessageBox("卸载钩子失败");
    }
  }
}

// 拦截WM_COMMAND消息
void CHOOKDlg::OnBnClickedButton3()
{
  // TODO: 在此添加控件通知处理程序代码
}

// 全局钩子，对整个操作系统的进程有效
void CHOOKDlg::OnBnClickedButton4()
```

```
{
  if (SetHook())
  {
    AfxMessageBox("安装全局钩子成功");
  }
  else
  {
    AfxMessageBox("安装全局钩子失败");
  }
}

// 卸载全局钩子
void CHOOKDlg::OnBnClickedButton5()
{
  if (UnSetHook())
  {
    AfxMessageBox("卸载全局钩子成功");
  }
  else
  {
    AfxMessageBox("卸载全局钩子失败");
  }
}

void  CHOOKDlg::OnDestroy()
{
    CDialogEx::OnDestroy();
    //  防止忘记卸载全局钩子
    if  (UnSetHook())
    {
        AfxMessageBox("卸载全局钩子成功");
    }
    else
    {
        AfxMessageBox("卸载全局钩子失败");
    }
}
```

32位64位的程序都可以监视其按键的输入值，同时键盘鼠标都可以下钩子。





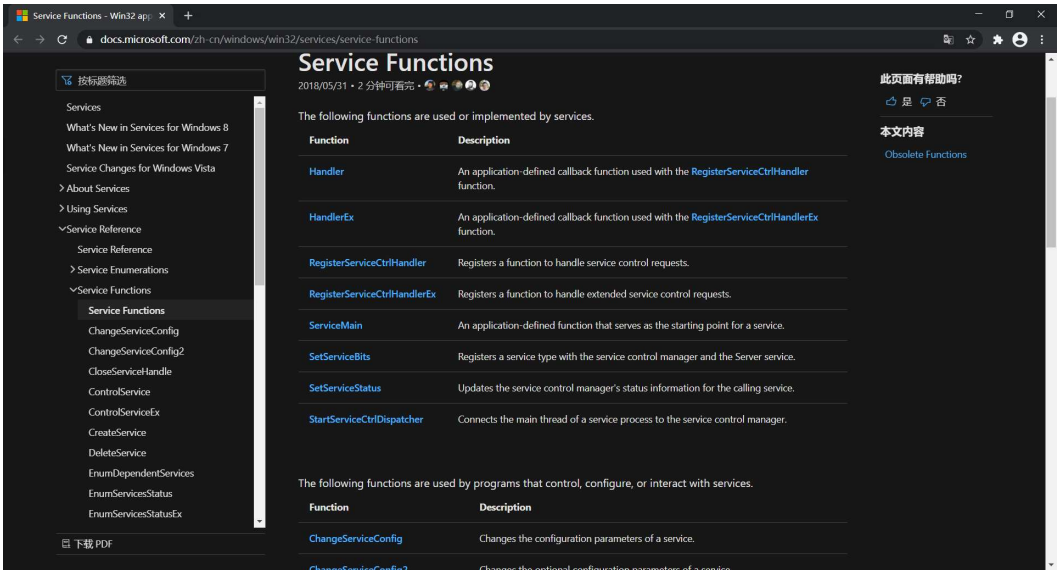
钩子的用途

钩子的用途，可以用来监控自己写的程序的一些消息等。

服务程序（Server）

服务程序（后台程序）：不需要进行交互的程序。

> 服务相关的API
>
> MSDN示例代码



编写服务程序的主要功能

服务程序的主要功能调用 StartServiceCtrlDispatcher 函数以连接到服务控制管理器（SCM）并启动控制调度程序线程。调度程序线程循环，等待对调度表中指定的服务的传入控制请求。当出现错误或进程中的所有服务已终止时，此线程将返回。当进程中的所有服务都终止时，SCM 将控制请求发送到调度程序线程，告诉它退出。然后，该线程从StartServiceCtrlDispatcher 调用返回，并且该过程可以终止。

启动程序之前需要先安装服务，服务程序并不是自动运行的， 安装服务后就可以return，返回，安装服务后面的代码由启动服务来负责运行。

创建的服务可以为其创建一个线程来负责干活，服务自带跨进程通讯，创建服务服务意外还可以给服务发消息（ControlService）。

可以将后台运行，不需要界面的程序做成服务。

服务的入口函数：ServiceMain
枚举服务：EnumServicesStatusExA