80386 三种工作模式。事实，保护，虚拟模式。保护模式是80386的主要工作模式。（保护主要针对存储器的保护）

Windows多任务的实现。

80386多任务操作系统的支持性主要体现在两个方面：

1. 在硬件上为任务之间的切换提供了良好的条件；
2. 它实现了多任务隔离，多任务隔离可以使每个任务都有独立的地址空间，就像每个任务都独享一个cpu一样。

在windows下任务被进程取代。进程是正在运行程序的实例。但是占有cpu时间片执行指令的并不是进程，而是线程，线程是进程内代码的执行单元。

Windows为每个进程分配4GB的地址空间主要依靠cpu的支持。Cpu在保护模式下支持虚拟存储。他可以帮助操作系统将磁盘空间当做内存空间来使用。在磁盘上使用这一机制的文件称作为页文件，它包含了对所有进程都有效的虚拟内存。

各进程的地址空间分为 用户空间和系统空间。

内核模式和用户模式

80386处理器定义了4种（0~3）特权级别，或者成为环，其中0级是最高级（特权级）别，3级是最低级（用户级）。

为了阻止应用程序访问或者修改关键的系统数据（即2GB系统空间内的数据）windows提供了两种访问模式：内核模式和用户模式，它们分别使用了处理器中的0和3这两个特权级别。用户程序的代码在用户模式下运行，系统程序（如系统服务程序和硬件驱动）的代码在内核模式下运行。

虽然系统中的每个进程都有自己的4GB私有地址空间。但是内核模式下的系统和设备驱动程序公用一块虚拟地址空间。

内核对象是系统提供的用户模式下代码与内核模式下代码进行交互的基本接口。 一个内核对象是一块内核分配的内存，它只能被运行在内核模式下的代码访问。内核对象记录的数据在整个系统中只有一份，所以它们称为系统资源。

PcGuard

1. 动画窗口的实现 新键类 class COpenScreenWnd : public CWnd

CDC m\_MemDC; // 创建内存DC

BITMAP m\_bm; // 创建位图结构变量

CBitmap m\_bitmap; // 创建位图对象

CBitmap \*old\_bitmap; // 创建位图对象指针

void CreatScreenWnd(); // 创建窗口

afx\_msg void OnPaint(); // 重载此函数

实现部分：

在构造函数中

m\_bitmap.LoadBitmap(IDB\_BITMAP1);// 拷贝资源位图

m\_bitmap.GetBitmap(&m\_bm);// 得到位图结构中的大小信息

void COpenScreenWnd::CreatScreenWnd()中创建窗口

//建立大小与位图大小相同的窗口

CreateEx(0,AfxRegisterWndClass(0,AfxGetApp()->LoadStandardCursor(IDC\_ARROW)),

"ImageScreen",

WS\_POPUP,

0,

0,

m\_bm.bmWidth,

m\_bm.bmHeight,

NULL,

NULL,

NULL );

在void COpenScreenWnd::OnPaint()中绘制动画

m\_MemDC.CreateCompatibleDC(NULL);// 建立一个和dc兼容的内存DC放置位图

old\_bitmap=m\_MemDC.SelectObject(&m\_bitmap);// 将创建的位图选入内存DC

int i,j,stepx,stepy,dispnum,x,y;

int bit\_Arry[20][20]; // 数组记录已显示过的数据组

for ( i=0; i<20; i++ )

for ( j=0; j<20; j++ )

bit\_Arry[i][j]=0;

stepx=m\_bm.bmWidth/20;

stepy=m\_bm.bmHeight/20;

srand( (unsigned)time( NULL ) );

dispnum=0;

//记录已显示过的数据组的个数

while(true)

{

x=rand() % 20;

y=rand() % 20;

if ( bit\_Arry[x][y] ) //如果为1，则已经显示了，跳出循环。

continue;

bit\_Arry[x][y]=1; //显示，设置为1

//将源矩形中的位图拷贝到目标矩形中，如果有必要，可以扩展或压缩该位图//使其与目标矩形尺寸吻合

// strechblt输出的图总是完整的，而且充满你指定的目标DC区域，而bitblt则可能输出的图是不完整的，也可能无法充满目标dc制定区域。

dc.StretchBlt( x\*stepx, y\*stepy, //目标设备逻辑横、纵坐标

stepx,stepy, //显示位图的像素宽、高度

&m\_MemDC, // 位图内存设备对象

x\*stepx, y\*stepy, // 位图的起始横、纵坐标

stepx,stepy, // 位图的像素宽、高度

SRCCOPY); // 将源矩形区域直接拷贝到目标矩形区域

dispnum++;

if ( dispnum >=400 ) // 判断显示完背景图吗

break; // 跳出循环

Sleep(10);

}// while

Sleep(200);// 暂停 200毫秒

m\_MemDC.SelectObject(old\_bitmap); // 还原DC

1. 超链接的实现

在类中添加变量

//用于保存超链接静态文本框的屏幕坐标

RECT m\_pRectLink;

并且添加 鼠标事件

ON\_WM\_MOUSEMOVE()

void CPcGuardDlg::OnMouseMove(UINT nFlags, CPoint point)

添加 ON\_WM\_LBUTTONDOWN()

// 鼠标左单击消息

afx\_msg void OnLButtonDown(UINT nFlags, CPoint point);

分别在实现中添加代码如下：

OnMouseMove(UINT nFlags, CPoint point)

//下面设置鼠标在静态文本区时，将光标设成小手状

if (point.x > m\_pRectLink.left && point.x < m\_pRectLink.right && point.y > m\_pRectLink.top && point.y < m\_pRectLink.bottom )

//此处添加判断坐标算法

{

HCURSOR hCursor;

hCursor = AfxGetApp()->LoadStandardCursor(IDC\_HAND);

//将鼠标设为小手状

SetCursor(hCursor);

}

OnLButtonDown(UINT nFlags, CPoint point)

//此处添加判断坐标算法

if (point.x > m\_pRectLink.left && point.x < m\_pRectLink.right && point.y > m\_pRectLink.top && point.y < m\_pRectLink.bottom)

{

//鼠标左键按下

if (nFlags == MK\_LBUTTON)

{

//为改善鼠标效果，此处加入以上变换鼠标形状的代码

ShellExecute(0, NULL, "http://blog.csdn.net/gfsfg8545", NULL,NULL, SW\_NORMAL);

}

}

FileMonitor

三种方法：

**调用Windows的api函数ReadDirectoryChanges监视指定目录中文件（推荐使用），这个使用异步IO：**

**或者用SHChangeNotifyRegister和SHChangeNotifyDeregister来注册事件SHChangeNotify：**

**或者调用API函数－FindFirstChangeNotification 、FindCloseChangeNotification、FindNextChangeNotification 和WaitForSingleObject ：**

1. //创建文件，打开目录，得到目录句柄 （在监视线程中打开）

CreateFile

定义 FILE\_NOTIFY\_INFORMATION

ReadDirectoryChangesW ()在这里设置监控属性和接受监控结果 W代表它是Unicode版本

在对监控的文件夹同时删除、移入多个文件时，这个函数不能返回所有删除或移入的文件的信息，比如，对于监控的文件夹，同时 选中多个文件，十个以上，执行删除，时这个函数返回的变更数据就不完整了(注：我设置的接收数据缓冲区大小为4KB,一般的情况是够用了，不至于缓冲区太小而不能容纳文件变更数据)

写入配置信息

::WritePrivateProfileString("Selections","Add",set.m\_bAddNew? "1":"0",".\\Config.ini");

读取配置信息

GetPrivateProfileInt("Selections","Add",0,".\\Config.ini")

OnButtonStart() 开始按钮

m\_hThread=::CreateThread(NULL, 0, ThreadMonitorProc, this, 0, &ThreadID); //创建监视线程

OnButtonStop() 停止监视

// 终止线程

::TerminateThread(m\_hThread, 0);

// 关闭目录句柄

CloseHandle(m\_hDir);

// 处理右键消息

void CFileMonitorDlg::OnRclickList(NMHDR\* pNMHDR, LRESULT\* pResult)

{

NM\_LISTVIEW\* pNMListView = (NM\_LISTVIEW\*)pNMHDR;

if(pNMListView->iItem != -1)

{

DWORD dwPos = GetMessagePos(); //获得消息坐标

CPoint point( LOWORD(dwPos), HIWORD(dwPos) );

CMenu menu;

VERIFY( menu.LoadMenu(IDR\_MENU1));

CMenu\* popup = menu.GetSubMenu(0);

ASSERT( popup != NULL );

popup->TrackPopupMenu(TPM\_LEFTALIGN | TPM\_RIGHTBUTTON, point.x, point.y, this);

}

\*pResult = 0;

}

转到相应文件方法：

Explorer /select,

WinExec(str,SW\_SHOWNORMAL);//显示文件的位置

ProcessManage

#include <Tlhelp32.h>//系统快照

//提升权限

bool CProcessManageDlg::EnablePrivilege()

{

// 提升进程特权,以便访问系统进程

BOOL status = false;

HANDLE hToken;

// 打开一个进程的访问令牌

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

{

// 取得特权名称为"SetPrivilege"的LUID

LUID uID;

::LookupPrivilegeValue(NULL, SE\_DEBUG\_NAME, &uID);

// 调整特权级别

TOKEN\_PRIVILEGES tp;

tp.PrivilegeCount = 1;

tp.Privileges[0].Luid = uID;

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

::AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);

if (ERROR\_SUCCESS == GetLastError())

status =true;

else

status =false;

// 关闭访问令牌句柄

::CloseHandle(hToken);

}

return status;

}

//列举进程 PROCESSENTRY32 结构将获得进程的信息（很重要）

int CProcessManageDlg::EnumProcess(void)

{

int i=0;

int j=0;

HANDLE hProcessSnap;//快照句柄

CString tem,title;

//The SHGetFileInfo API provides an easy way to get attributes for a file given a pathname

SHFILEINFO info;

//定义文件系统时间(创建,退出,内核,用户)

FILETIME ftCreation,

ftExit,

ftKernel,

ftUser;

//定义时间结构 COleDateTime使用的位数是双浮点的两倍(更精确)

COleDateTime timeCreation,

timeCurrent;

COleDateTimeSpan timeDiff;

m\_ListProcess.DeleteAllItems();// 清除列表中的所有项

PROCESSENTRY32 pe32={0};// 定义PROCESSENTRY32 结构

pe32.dwSize = sizeof( PROCESSENTRY32 ); // 设置PROCESSENTRY32结构的大小

hProcessSnap = CreateToolhelp32Snapshot( TH32CS\_SNAPPROCESS, 0 );// 创建进程快照

if( hProcessSnap == INVALID\_HANDLE\_VALUE )

{

return 0;

}

// 首次调用

if(!Process32First( hProcessSnap, &pe32 ) )

{

CloseHandle( hProcessSnap );

return 0;

}

// 遍历进程快照，轮流显示每个进程的信息

while( Process32Next( hProcessSnap, &pe32 ))

{

// 获取文件图标

SHGetFileInfo(GetProcessPath(pe32.th32ProcessID),0,&info,sizeof(&info),

SHGFI\_ICON | SHGFI\_SYSICONINDEX|SHGFI\_SMALLICON);

j=m\_ImageList.Add(info.hIcon);// 添加程序图标

m\_ListProcess.InsertItem(i,pe32.szExeFile,j); // 进程名

tem.Format("%d",pe32.th32ProcessID);

m\_ListProcess.SetItemText(i,1,tem); // PID

title=pe32.szExeFile;

// 判断是否为 smss.exe进程

//smss.exe(Session Manager Subsystem)，该进程为会话管理子系统用以初始化系统变量

if(!strcmp(title,"smss.exe"))

{

m\_ListProcess.SetItemText(i,4,"C:\\WINDOWS\\system32\\smss.exe");

}

//是否为系统进程

else if(!strcmp(pe32.szExeFile,"system")||!strcmp(pe32.szExeFile,"[System Process]"))

{

m\_ListProcess.SetItemText(i,0,"Idle");

m\_ListProcess.SetItemText(i,4,"NT OS Kernel");// NT 核心进程

}

else if(!strcmp(pe32.szExeFile,"system")||!strcmp(pe32.szExeFile,"System"))

{

m\_ListProcess.SetItemText(i,4,"NT OS Kernel");

}

else

{

m\_ListProcess.SetItemText(i,4,GetProcessPath(pe32.th32ProcessID));//进程路径

}

HANDLE hProcess;// 进程句柄

timeCurrent = COleDateTime::GetCurrentTime();// 获得当前时间

hProcess = OpenProcess(PROCESS\_ALL\_ACCESS | PROCESS\_VM\_READ | PROCESS\_QUERY\_INFORMATION,

FALSE,pe32.th32ProcessID);// 打开此进程

if(NULL != hProcess)

{

GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser);

timeCreation = ftCreation;

timeDiff = timeCurrent - timeCreation;

tem.Format("%u小时:%u分:%u秒",timeDiff.GetHours(), timeDiff.GetMinutes(),

timeDiff.GetSeconds());// 获取进程运行时间

m\_ListProcess.SetItemText(i,2,tem);// 设置进程运行时间

//使用API函数GetProcessMemoryInfo来获取内存的使用情况

//ppsmemCounters是返回内存使用情况的结构

PROCESS\_MEMORY\_COUNTERS pmc;

pmc.cb = sizeof(PROCESS\_MEMORY\_COUNTERS);

if(::GetProcessMemoryInfo(hProcess, &pmc,sizeof(pmc)))

{

tem.Format(\_T("%.2f M"),(float)(pmc.WorkingSetSize/1024)/1024);

m\_ListProcess.SetItemText(i,3,tem);

}

CloseHandle(hProcess);// 关闭此进程句柄

}

m\_vProcessId[i]=pe32.th32ProcessID; //存放ID信息

m\_ListProcess.SetItemData(i,i);//索引,为排序做准备

i++;

}

CString str1;

str1.Format("%d",i);// 进程数量

SetDlgItemText(IDC\_STATIC\_PRO\_NUM,str1);

CloseHandle( hProcessSnap );//关闭句柄

UpdateData(FALSE);

return 0;

}

//列举进程模块信息 主要用头文件:   Psapi.h. 中的EnumProcessModules 函数

BOOL CProcessManageDlg::GetDllInfo(DWORD dwPID)

{

// 防止闪屏

m\_ListDll.SetRedraw(FALSE);

// 模块句柄数组

HMODULE hMods[1024];

HANDLE hProcess;

DWORD cbHmodule;// 所含模块数

unsigned int i;

m\_ListDll.DeleteAllItems();

char szModName[MAX\_PATH];

// 打开目标进程，取得进程句柄

hProcess = OpenProcess( PROCESS\_QUERY\_INFORMATION |PROCESS\_VM\_READ,FALSE, dwPID);

// 枚举 进程模块

if( hProcess && EnumProcessModules(hProcess, hMods, sizeof(hMods), &cbHmodule))

{

for ( i = 0; i <=(cbHmodule / sizeof(HMODULE)); i++ )

{

// 得到模块的完整路径名

if ( GetModuleFileNameEx( hProcess, hMods[i], szModName,

sizeof(szModName)))

{

//模块名，修改时间

CString ModName, FTime;

// 文件的属性

CFileStatus status;

ModName = szModName;

ModName.Replace("\\??\\","");

ModName.Replace("\\SystemRoot","C:\\WINDOWS");

m\_ListDll.InsertItem(0xffff,ModName);

//CString 转 char\*

char\* szFileName=(LPTSTR)(LPCTSTR)ModName;

// 获得文件版本信息 Returns size of version info in bytes

DWORD dwSize =::GetFileVersionInfoSize(szFileName,NULL);

LPVOID pBlock = malloc(dwSize);

// 把版本信息 写到缓冲区 Read version info into buffer

if(::GetFileVersionInfo(szFileName,0,dwSize,pBlock))

{

char\* pVerValue = NULL;

UINT nSize = 0;

//这是得到语言

VerQueryValue(pBlock,\_T("\\VarFileInfo\\Translation"),

(LPVOID\*)&pVerValue,&nSize);

CString strSubBlock,strTranslation,strTemp;

//语言转化 080404b0为中文，040904E4为英文

strTemp.Format("000%x",\*((unsigned short int \*)pVerValue));

strTranslation = strTemp.Right(4);

strTemp.Format("000%x",\*((unsigned short int \*)&pVerValue[2]));

strTranslation += strTemp.Right(4);

//文件描述 FileDescription

strSubBlock.Format("\\StringFileInfo\\%s\\FileDescription",strTranslation);

VerQueryValue(pBlock,(LPTSTR)(LPCTSTR)strSubBlock,(LPVOID\*)&pVerValue,&nSize);

if((NULL!=pVerValue)&&(0!=nSize))// 判断是否得到查询结果

{

strTemp.Format("%s",pVerValue);

m\_ListDll.SetItemText(i, 1, strTemp);

}

else

{

m\_ListDll.SetItemText(i, 1, "");

}

}

delete pBlock; // 释放申请的内存

// 用于查看文件信息的函数

m\_ListDll.SetItemData(i,i);

}

}

int j = 0;

j= m\_ListDll.GetItemCount();

m\_numOfDll.Format("%d",j-1);

CloseHandle(hProcess);

//更新内容

m\_ListDll.SetRedraw(TRUE);

m\_ListDll.Invalidate();

m\_ListDll.UpdateWindow();

return TRUE;

}

else

{

CloseHandle( hProcess );

return FALSE;

}

}

// 暂停进程 (注意实际是暂停线程)

void CProcessManageDlg::OnSuspend()

{

// TODO: Add your command handler code here

CString str;

int nItem = m\_ListProcess.GetNextItem( -1, LVNI\_ALL | LVNI\_SELECTED);

str = m\_ListProcess.GetItemText(nItem, 1);// 进程ID

unsigned int Pid;

THREADENTRY32 th32;

th32.dwSize=sizeof(th32);

Pid = (DWORD)atoi(str); // CString 转化为DWORD

HANDLE hThreadSnap=::CreateToolhelp32Snapshot(TH32CS\_SNAPTHREAD,0); //创建进程快照

if(hThreadSnap==INVALID\_HANDLE\_VALUE)

{

AfxMessageBox("CreateToolhelp32Snapshot调用失败");

}

BOOL bRet = ::Thread32First(hThreadSnap,&th32);

while(bRet)

{

if(th32.th32OwnerProcessID == Pid)

{ //HANDLE hThread = OpenThread(THREAD\_SET\_CONTEXT, FALSE, \_\_dwMainThreadId);

// 由于vc6.0（低版本中的Winbase.h 没有此函数，因此

HANDLE handle=lpfnOpenThread(THREAD\_ALL\_ACCESS,FALSE,th32.th32ThreadID); // 打开线程内核对象

if(!(::SuspendThread(handle))) //暂停线程

{

}

CloseHandle(handle);// 关闭内核对象

}

bRet=::Thread32Next(hThreadSnap,&th32);

}

::CloseHandle(hThreadSnap);// 关闭内核对象

}

// 百度一下该进程

\_T("www.baidu.com/s?wd=");

ShellExecuteEx

// 复制进程路径

void CProcessManageDlg::OnCopy()

{

//获得当前选中项

int nItem = m\_ListProcess.GetNextItem( -1, LVNI\_ALL | LVNI\_SELECTED);

if(OpenClipboard())//打开剪贴板

{

CString str;

HANDLE hClip;

char \*pBuf;

EmptyClipboard();//清空剪贴板

str=m\_ListProcess.GetItemText(nItem,4);//获得文本

hClip=::GlobalAlloc(GMEM\_MOVEABLE,str.GetLength()+1);//从堆中分配内存

pBuf=(char\*)GlobalLock(hClip);//对内存加锁

strcpy(pBuf,str);//拷贝到分配的内存中

GlobalUnlock(hClip);//解锁

SetClipboardData(CF\_TEXT,hClip);

CloseClipboard();//关闭剪贴板

}

}

进程间通信（它在底层实际上是通过文件映射来实现的。它的缺点是灵活性不高，并且它只能用于Windows平台的单机环境下）

COPYDATASTRUCT data;//定义数据结构

ProcessMsg ProMsg ;//此处不能利用指针(切忌)即 不能动态分配内存存储数据

ProMsg.PID=(DWORD)atoi(strPid);

char \* pStr = (LPTSTR)(LPCTSTR)StrProName;//转化成char\*

strncpy(ProMsg.ProcessName, pStr, sizeof(ProMsg.ProcessName));//拷贝到目标

data.cbData = sizeof(ProcessMsg);

data.lpData = (void\*)&ProMsg;

data.dwData = TYPE\_PROCESSMANAGE\_MSG;

::SendMessage(hWnd,WM\_COPYDATA,(WPARAM)hWnd,(LPARAM)&data);//发送！

RegMonitor

// 拦截WIN32 API的原理

// Detours定义了三个概念：

//(1) Target函数：要拦截的函数，通常为Windows的API。

//(2) Trampoline函数：Target函数的部分复制品。因为Detours将会改写Target函数，所以先把Target函数的前5个字节复制保存好，一方面仍然保存Target函数的过程调用语义，另一方面便于以后的恢复。

//(3) Detour 函数：用来替代Target函数的函数。

//Detours在Target函数的开头加入JMP Address\_of\_ Detour\_ Function指令（共5个字节）把对Target函数 的调用引导到自己的Detour函数， 把Target函数的开头的5个字节加上JMP Address\_of\_ Target \_ Function+ 5共10个字节作为Trampoline函数。

/\*DETOUR\_TRAMPOLINE(trampoline\_prototype, target\_name);

功能：该宏把名为target\_name 的Target函数生成Trampoline函数，以后调用 trampoline\_prototype在语义上等于调用Target函数\*/

DETOUR\_TRAMPOLINE(LONG WINAPI Real\_RegCreateKeyExA(HKEY, LPCSTR, DWORD, LPSTR, DWORD, REGSAM, LPSECURITY\_ATTRIBUTES, PHKEY,

LPDWORD), RegCreateKeyExA);

// Replace\_ 替代真实的 API Real\_ 即Trampoline函数

// RegDeleteKeyW替换

LONG WINAPI Replace\_RegDeleteKeyW(HKEY hKey,LPCWSTR lpSubKey )

{

LONG nRet;

\_\_try

{

nRet = Real\_RegDeleteKeyW(hKey, lpSubKey);

if(!HOOK\_RegDeleteKey)

return nRet;

if(GetCurrentProcessId() == ExplorerPID)

return nRet;

if(GetCurrentProcessId() == RegMonPID)

return nRet;

if((PID == 0) || ((GetCurrentProcessId() == PID) && (PID>0)) )

{

if(lpSubKey == NULL)

return nRet;

COPYDATASTRUCT data;

PARAMS params;

memset(&params,0,sizeof(PARAMS));

params.PID = GetCurrentProcessId();

params.hKey = hKey;

memcpy(params.buf1, lpSubKey, wcslen(lpSubKey)\*2);

data.cbData = sizeof(PARAMS);

data.lpData = (void \*)&params;

data.dwData = TYPE\_RegDeleteKeyW;

SendMessage(g\_hWnd, WM\_COPYDATA, (WPARAM)g\_hWnd, (LPARAM)&data);

}

}\_\_finally

{

};

return nRet;

}

// 拦截函数

void Intercept()

{

// 使用DetourFunctionWithTrampoline函数来截获目标函数。这个函数有两个参数：trampoline函数以

// 及截获函数的指针。因为目标函数已经被加到trampoline函数中，所有不需要在参数中特别指定。

DetourFunctionWithTrampoline((PBYTE)Real\_RegCreateKeyExA, (PBYTE)Replace\_RegCreateKeyExA);

// 卸载拦截

void UnIntercept()

{

// 调用DetourRemove来去掉对一个目标函数的截获

DetourRemove((PBYTE)Real\_RegCreateKeyExA,(PBYTE)Replace\_RegCreateKeyExA);

// 允许Vista/Win7下,不同权限进程间(高向低不需要，低向高需要)发送消息：WM\_COPYDATA

// 这一段很重要不然的话 编译出的程序不能以管理员身份运行(应用程序在Windows 7上运行时，因为UIPI机制，

// 这种消息传递被阻断了) ChangeWindowMessageFilter

// 利用这个函数，我们可以添加或者删除能够通过特权等级隔离的Windows消息。这就像拥有较高特权等级的进程,

// 设置了一个过滤器,允许通过的Windows消息都被添加到这个过滤器的白名单,只有在这个白名单上的消息才允许传

// 递进来

typedef BOOL (WINAPI FAR \*ChangeWindowMessageFilter\_PROC)(UINT,DWORD);

ChangeWindowMessageFilter\_PROC pfnChangeWindowMessageFilter;

pfnChangeWindowMessageFilter = (ChangeWindowMessageFilter\_PROC)::GetProcAddress\

( (m\_HmoduleUser32=::GetModuleHandle(\_T("USER32.dll") ) ),"ChangeWindowMessageFilter");

if (pfnChangeWindowMessageFilter)

{

// 设置了一个过滤器 WM\_COPYDATA

pfnChangeWindowMessageFilter(WM\_COPYDATA, 1);

}

?

void CRegMonitorDlg::QueryValueExA\_Msg(PARAMS\* params,int index)

注册表相关

·hKey为主键值，可以取下面的一些数值：HKEY\_CLASSES\_ROOT、HKEY\_CURRENT\_CONFIG、　　HKEY\_CURRENT\_USER、HKEY\_LOCAL\_MACHINE、HKEY\_USER、HKEY\_PERFORMANCE\_DATA(WINNT操作系统)、HKEY\_DYN\_DATA（WIN9X操作系统）；

参数lpType为一个指向数据类型的指针，数据类型为下列类型之一：REG\_BINARY 二进制数据、REG\_DWORD 32位整数、REG\_DWORD\_LITTLE\_ENDIAN little－endian格式的数据，例如0X12345678以（0X78 0X56 0X34 0X12）方式保存、REG\_DWORD\_BIG\_ENDIAN big－endian格式的数据，例如0X12345678以（0X12 0X34 0X56 0X78）方式保存、REG\_EXPAND\_SZ 一个包含未扩展环境变量的字符串、REG\_LINK 一个Unicode类型的链接、REG\_MULIT\_SZ 以两个零结尾的字符串、REG\_NONE 无类型数值、REG\_RESOURCE\_LIST 设备驱动资源列表、REG\_SZ 一个以零结尾的字符串根据函数使用的字符集类型的不同而设置为Unicode或ANSI类型的字符串；

**现有注册表数据类型**

|  |  |  |  |
| --- | --- | --- | --- |
| **类 型** | **类型索引** | **大 小** | **说 明** |
| REG\_BINARY | 3 | 0~多个字节 | 可以包含任何数据的二进制对象颜色描述 |
| REG\_COLOR\_RGB | \* | 4字节 | 颜色描述 |
| REG\_DWORD | 4 | 4字节 | DWORD值 |
| REG\_DWORD\_BIG\_ENDIAN | 5 | 4字节 | 一个DWORD值的逆序存储形式 |
| REG\_DWORD\_LITTLE\_ENDIAN | 4 | 4字节 | DWORD值 |
| REG\_EXPAND\_SZ | 2 | 0~多个字节 | 包含环境变量占位符的字符串 |
| REG\_FILE\_NAME | \* | 0~多个字节 | 文件名 |
| REG\_FILE\_TIME | \* | 未知 | 文件时间 |
| REG\_FULL\_RESOURCE\_DESCRIPTOR | 9 | 未知 | 硬件资源列表 |
| REG\_LINK | 6 | 0~多个字节 | 命名一个符号链接(symbolic link)的Unicode串 |
| REG\_MULTI\_SZ | 7 | 0~多个字节 | 以null字符分隔的字符串集合，集合中的最后一个字符串以两个null字符结尾 |
| REG\_NONE | 0 | 未知 | REG\_NONE类型的对象无须分类，它与REG\_UNKNOWN类型不同 |
| REG\_RESOURCE\_LIST | 8 | 未知 | 设备使用的资源列表 |
| REG\_RESOURCE\_REQUIREMENTS\_ LIST | 10 | 未知 | 驱动程序要求的资源列表 |
| REG\_SZ | 1 | 0~多个字节 | 以一个null字符线的字符串 |
| REG\_UNKNOWN | （未定义） | 未知 | 因为数据类型索引无效而无法确定的对象类型 |

viusKiller

// 病毒查杀对话框 运行流程是：选择扫描目标——开始扫描硬盘——按照特征码识

// 别出病毒文件——查找病毒进程——结束病毒进程——列出病毒文件——用户选择

// 是否删除可疑文件启动设置——其余清理。

//创建各线程开始进行查找

DWORD WINAPI MainThreadProc(LPVOID lpParam)

int CKillerDlg::ScanFile(char \*FileDir,char \*FileType)

DWORD WINAPI VirusCompareProc(LPVOID lpParam);

bool CKillerDlg::IsVirus(CString strFilePath)

void CKillerDlg::ScanProcess(CString ModuleName)