# 创建窗口全过程

## 创建窗口

#include <windows.h>

//

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

{

switch (message) {

case WM\_DESTROY:

PostQuitMessage(0);

break;

}

return DefWindowProc(hWnd, message, wParam, lParam);

}

int CALLBACK WinMain(HINSTANCE hInstance,

HINSTANCE hPrevInstance,

LPSTR lpCmdLine,

int nCmdShow)

{

WNDCLASS wc = { 0 };

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hbrBackground = (HBRUSH)(COLOR\_WINDOW + 1);

wc.hCursor = NULL;

wc.hIcon = NULL;

wc.hInstance = hInstance;

wc.lpfnWndProc = WndProc;

wc.lpszClassName = "Main";

wc.lpszMenuName = nullptr;

wc.style = CS\_HREDRAW | CS\_VREDRAW;

RegisterClass(&wc);

HWND hWnd = CreateWindowW(L"Main", L"window", WS\_OVERLAPPEDWINDOW,

100, 100, 500, 500, nullptr, nullptr, hInstance, nullptr);

ShowWindow(hWnd, SW\_SHOW);

UpdateWindow(hWnd);

MSG nMsg = { 0 };

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

{

TranslateMessage(&nMsg);

DispatchMessage(&nMsg); //派发消息

}

return 0;

}

## 细节

ATOM RegisterClass(

CONST WNDCLASS \*IpWndClass//窗囗类的数据

）；注册成功后，返回一个数字标识。

typedef struct \_WNDCLASS{

UINT style;//窗囗类的风格

WNDPROC IpfnWndProc；//窗口处理函数

int cbCIsExtra；//窗囗类的附加数据buff的大小

int cbWndExtra；//窗囗的附加数据buff的大小

HINSTANCE hInstance;/当前模块的实例句柄

HICON hIcon;//窗口图标句柄

HCURSOR hCursor;//鼠标的句柄

HBRUSH hbrBackground;//绘制窗口背景的画刷句柄

LPCTSTR lpszMenuName;/窗口菜单的资源ID字符串

LPCTSTR IpszClassName;/窗囗类的名称

}WNDCLASS,\*PWNDCLASS;

CreateWindow/CreateWindowEx)

HWND CreateWindowEx(

DWORD dwExStyle,//窗囗的扩展风格 CreateWindowEx比CreateWindow多此参数

LPCTSTR IpClassName,//已经注册的窗囗类名称

LPCTSTR IpWindowName,//窗口标题栏的名字

DWORD dwStyle,//窗囗的基本风格

int x,/窗囗左上角水平坐标位置

int y,//窗口左上角垂直坐标位置

int nWidth,//窗囗的宽度

int nHeight,/窗囗的高度

HWND hWndParent,//窗口的父窗口句柄

HMENU hMenu,//窗口菜单句柄

HINSTANCE hInstance,//应用程序实例句柄

LPVOID IpParam//窗囗创建时附加参数

）；创建成功返回窗口句

## 创建子窗口时

要设置父窗口句柄·

创建风格要增加WS\_CHILD | WS\_VISIBLE

## 消息组成（windows平台下）

窗口句柄

消息ID

消息的两个参数（两个附带信息）

消息产生的时间

消息产生时的鼠标位置

### 没写笔记引以为戒！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！

## 字体的使用

1创建字体

HFONT CreateFont(

int nHeight,//字体高度

int nWidth,//字体宽度

int nEscapement,//字符串倾斜角度

int nOrientation,//字符旋转角度 （Z轴）

int fnWeight,//字体的粗细

DWORD fdwltalic,//斜体

DWORD fdwUnderline,//字符下划线

DWORD fdwStrikeOut,//删除线

DWORD fdwCharSet,//字符集

DWORD fdwOutputPrecision,//输出精度

DWORD fdwClipPrecision,//剪切精度

DWORD fdwQuality,/输出质量

DWORD fdwPitchAndFamily,/匹配字体

LPCTSTR IpszFace//字体名称 ）；

## 对话框

非模式对话框：

创建对话框

HWND CreateDialog(

HINSTANCE hInstance,//应用程序实例句柄

LPCTSTR IpTemplate,//模板资源

IDHWND hWndParent,//父窗口

DLGPROC IpDialogFunc/自定义函数)

非阻塞函数，创建成功返回窗口句柄，需要使用ShowWindow函数显示,

对话框的关闭

关闭时使用DestroyWindow销毁窗口，不能使用EndDialog关闭对话框。

模式对话框：

创建对话框

DialogBox(

HINSTANCE hInstance,//应用程序实例句柄

LPCTSTR IpTemplate,//模板资源

IDHWND hWndParent,//父窗口

DLGPROC IpDialogFunc/自定义函数)

阻塞函数，需EndDialog关闭对话框，返回值为EndDialog的参数2，不能使用DestroyWindow销毁窗口

# 静动态库

## 动态库

动态库特点·动态库特点

1）运行时独立存在

2）源码不会链接到执行程序

3）使用时加载（使用动态库必须使动态库执行）

与静态库的比较：

1）由于静态库是将代码嵌入到使用程序中，多个程序使用时，会有多份代码，所以代码体积会增大。动态库的代码只需要存在一份，其他程序通过函数地址使用，所以代码体积小。

2）静态库发生变化后，新的代码需要重新链接嵌入到执行程序中。动态库发生变化后，如果库中函数的定义（或地址）未变化，其他使用DLL的程序不需重新链接。

动态库的创建·

创建动态库项目

添加库程序

库程序导出-提供给使用者库中的函数等信息。

1）声明导出：使用\_declspec(dllexport)导出函数注意：动态库编译链接后，也会有LIB文件，是作为动态库函数映射使用，与静态库不完全相同。(此方法编译修改函数名)

2）模块定义文件.def例如：(此方法编译不修改函数名)

LIBRARY DLLFunc//库

EXPORTS //库导出表

DLL\_Mul @1//导出的函数

动态库的使用：

隐式链接（操作系统负责使动态库执行）

1）头文件和函数原型可以在函数原型的声明前，增加\_declspec(dllimport)

2）导入动态库的Lib文件

3）在程序中使用函数

4）隐式链接的情况，dll文件可以存放的路径：

（1）与执行文件中同一个目录下

（2)当前工作目录

（3）Windows目录

（4）Windows/System32目录

（5)Windows/System

（6）环境变量PATH指定目录

显式链接（程序员自己负责使动态库执行）

1）定义函数指针类型 typedef

2）加载动态库HMODULE LoadLibrary(LPCTSTR IpFileName//动态库文件名或全路径）；返回DLL的实例句柄（HINSTANCE)

3）获取函数地址FARPROC GetProcAddress(HMODULE hModule,//DLL句柄LPCSTR IpProcName //函数名称）；成功返回函数地址

4）使用函数

5）卸载动态库BOOL FreeLibrary(HMODULE hModule //DLL的实例句柄

动态库中封装类 ·

在类名称前增加\_declspec(dllexport)定义，例如：

class\_declspec(dllexport)CMath{

...

}

通常使用预编译开关切换类的导入导出定义，例如：

#ifdef DLLCLASS\_EXPORTS

#define EXT\_CLASS \_declspec(dllexport)//DLL ori

#else

#define EXT\_CLASS \_declspec(dllimport)/使用者

#endif

class EXT\_CLASS CMath{

...

}

# 多线程

## 线程基础

Windows线程是可以执行的代码的实例。系统是以线程为单位调度程序。一个程序当中可以有多个线程，实现多任务的处理。

Windows线程的特点：

1）线程都具有1个ID

2）每个线程都具有自己的内存栈3)同一进程中的线程使用同一个地址空间。

线程的调度：

将CPU的执行时间划分成时间片，依次根据时间片执行不同的线程。

线程轮询：线程A->线程B->线程A..…

创建线程

创建线程 HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES IpThreadAttributes,//安全属性 （废弃）

SIZE\_T dwStackSize,//线程栈的大小

LPTHREAD\_START\_ROUTINE IpStartAddress,//线程处理函数的函数地址

LPVOID IpParameter,/传递给线程处理函数的参数

DWORD dwCreationFlags,//线程的创建方式，0立即启动、CREATE\_SUSPENDED 手动

LPDWORD IpThreadld ∥创建成功，返回线程的ID ）；

创建成功，返回线程句柄

定义线程处理函数

DWORD WINAPI ThreadProc(

LPVOID IpParameter//创建线程时，传递给线程的参数

）；

销毁线程 ·

挂起

DWORD SuspendThread(

HANDLE hThread //handle to thread

）； ·

唤醒

DWORD ResumeThread(

HANDLE hThread //handle to thread

);

结束指定线程

BOOL Terminate Thread(

HANDLE hThread,// handle to thread

DWORD dwExitCode // exit code ·

)

结束函数所在的线程 (函数指ExitThread)

VOID ExitThread(

DWORD dwExitCode //exit code for this thread

）；

线程相关操作 ·

获取当前线程的ID

GetCurrentThreadld ·

获取当前线程的句柄

GetCurrentThread ·

等候单个句柄有信号 （只能用于有信号句柄）

VOID WaitForSingleObject(

HANDLE handle,//句柄BUFF的地址

DWORD dwMilliseconds //等候时间 (INFINITE为无线时间)

）；

同时等候多个句柄有信号

DWORD WaitForMultipleObjects(

DWORD nCount,/∥句柄数量

CONST HANDLE \*IpHandles,//句柄BUFF的地址

BOOL bWaitAll,//等候方式

DWORD dwMilliseconds //等候时间INFINITE

）；

bWaitAll-等候方式

TRUE-表示所有句柄都有信号，才结束等候

FASLE-表示句柄中只要有1个有信号，就结束等候。

## 原子锁·

相关问题

多个线程对同一个数据进行原子操作，会产生结果丢失。比如执行++运算时。

·错误代码分析：

当线程A执行g\_value++时，如果线程切换时间正好是在线程A将值保存到g\_value之前，线程B继续执行g value++，那么当线程A再次被切换回来之后，会将原来线程A保存的值保存到g\_value上，线程B进行的加法操作被覆盖。

·使用原子锁函数

InterlockedIncrement

InterlockedDecrement

InterlockedCompareExchange

InterlockedExchange

......

原子锁的实现：

直接对数据所在的内存操作，并且在任何一个瞬间只能有一个线程访问。

原子锁优缺点：

优：保证数据准确、所有锁里面数度最快的

缺：只能对运算符有用、一个函数只对应一个运算操作、

## 互斥·

相关的问题

多线程下代码或资源的共享使用。

互斥的使用

1创建互斥HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES IpMutexAttributes,//安全属性

BOOL bInitialOwner,//初始的拥有者

TRUE/FALSELPCTSTR IpName /命名

）；创建成功返回互斥句柄

2等候互斥 WaitFor..…互斥的等候遵循谁先等候谁先获取。

3释放互斥 BOOL ReleaseMutex(

HANDLE hMutex //handle to mutex

)

4关闭互斥句柄 CloseHandle

\*互斥里的函数同一时间只能被一个线程使用、互斥句柄为信号句柄（工作时无信号，Relese释放后有信号）

## 事件·

相关问题

程序之间的通知的问题。

事件的使用

1创建事件HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES IpEventAttributes,//安全属性

BOOL bManualReset,/事件重置（复位）方式，TRUE手动，FALSE自动

BOOL bInitialState,//事件初始状态，TRUE有信号

LPCTSTR IpName//事件命名

）；创建成功返回事件句柄

2等候事件

， WaitForSingleObject/WaitForMultipleObjects

3触发事件（将事件设置成有信号状态）

BOOL SetEvent( HANDLE hEvent //handle to event ）；

4复位事件（将事件设置成无信号状态）

BOOL ResetEvent( HANDLE hEvent // handle to event ）；

5关闭事件

CloseHandle

小心事件的死锁。

## 信号量

相关的问题

类似于事件，解决通知的相关问题。但提供一个计数器，可以设置次数。

信号量的使用

1创建信号量

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES IpSemaphoreAttributes,//安全属性

LONG IInitialCount,//初始化信号量数量

LONG IMaximumCount,/信号量的最大值

LPCTSTR IpName /命名

）；创建成功返回信号量句柄

2等候信号量

WaitFor..每等候通过一次，信号量的信号减1，直到为0阻塞

3给信号量指定计数值

BOOL ReleaseSemaphore(

HANDLE hSemaphore,//信号量句柄

LONG IReleaseCount,//释放数量（即重新设置初始化信号量数量，不能超过最大值）

LPLONG IpPreviousCount/释放前原来信号量的数量，可以为NULL）；

4关闭句柄 CloseHandle