堆和栈的区别  
一、预备知识—程序的内存分配  
一个由c/C++编译的程序占用的内存分为以下几个部分  
1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。  
2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。  
3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放   
4、文字常量区—常量字符串就是放在这里的。 程序结束后由系统释放  
5、程序代码区—存放函数体的二进制代码。  
二、例子程序   
这是一个前辈写的，非常详细   
//main.cpp   
int a = 0; 全局初始化区   
char \*p1; 全局未初始化区   
main()   
{   
int b; 栈   
char s[] = "abc"; 栈   
char \*p2; 栈   
char \*p3 = "123456"; 123456\0在常量区，p3在栈上。   
static int c =0； 全局（静态）初始化区   
p1 = (char \*)malloc(10);   
p2 = (char \*)malloc(20);   
分配得来得10和20字节的区域就在堆区。   
strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。   
}

二、堆和栈的理论知识   
2.1申请方式   
stack:   
由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间   
heap:   
需要程序员自己申请，并指明大小，在c中malloc函数   
如p1 = (char \*)malloc(10);   
在C++中用new运算符   
如p2 = (char \*)malloc(10);   
但是注意p1、p2本身是在栈中的。

2.2   
申请后系统的响应   
栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。   
堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，   
会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

2.3申请大小的限制   
栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。   
堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

2.4申请效率的比较：   
栈由系统自动分配，速度较快。但程序员是无法控制的。   
堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.   
另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活。

2.5堆和栈中的存储内容   
栈： 在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。   
当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。   
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

2.6存取效率的比较

char s1[] = "aaaaaaaaaaaaaaa";   
char \*s2 = "bbbbbbbbbbbbbbbbb";   
aaaaaaaaaaa是在运行时刻赋值的；   
而bbbbbbbbbbb是在编译时就确定的；   
但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。   
比如：   
#include   
void main()   
{   
char a = 1;   
char c[] = "1234567890";   
char \*p ="1234567890";   
a = c[1];   
a = p[1];   
return;   
}   
对应的汇编代码   
10: a = c[1];   
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]   
0040106A 88 4D FC mov byte ptr [ebp-4],cl   
11: a = p[1];   
0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]   
00401070 8A 42 01 mov al,byte ptr [edx+1]   
00401073 88 45 FC mov byte ptr [ebp-4],al   
第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到edx中，在根据edx读取字符，显然慢了。

2.7小结：   
堆和栈的区别可以用如下的比喻来看出：   
使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。   
使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。 

windows进程中的内存结构

在阅读本文之前，如果你连堆栈是什么多不知道的话，请先阅读文章后面的基础知识。

接触过编程的人都知道，高级语言都能通过变量名来访问内存中的数据。那么这些变量在内存中是如何存放的呢？程序又是如何使用这些变量的呢？下面就会对此进行深入的讨论。下文中的C语言代码如没有特别声明，默认都使用VC编译的release版。

首先，来了解一下 C 语言的变量是如何在内存分部的。C 语言有全局变量(Global)、本地变量(Local)，静态变量(Static)、寄存器变量(Regeister)。每种变量都有不同的分配方式。先来看下面这段代码：

#include <stdio.h>

int g1=0, g2=0, g3=0;

int main()   
{   
static int s1=0, s2=0, s3=0;   
int v1=0, v2=0, v3=0;

//打印出各个变量的内存地址

printf("0x%08x\n",&v1); //打印各本地变量的内存地址   
printf("0x%08x\n",&v2);   
printf("0x%08x\n\n",&v3);   
printf("0x%08x\n",&g1); //打印各全局变量的内存地址   
printf("0x%08x\n",&g2);   
printf("0x%08x\n\n",&g3);   
printf("0x%08x\n",&s1); //打印各静态变量的内存地址   
printf("0x%08x\n",&s2);   
printf("0x%08x\n\n",&s3);   
return 0;   
}

编译后的执行结果是：

0x0012ff78   
0x0012ff7c   
0x0012ff80

0x004068d0   
0x004068d4   
0x004068d8

0x004068dc   
0x004068e0   
0x004068e4

输出的结果就是变量的内存地址。其中v1,v2,v3是本地变量，g1,g2,g3是全局变量，s1,s2,s3是静态变量。你可以看到这些变量在内存是连续分布的，但是本地变量和全局变量分配的内存地址差了十万八千里，而全局变量和静态变量分配的内存是连续的。这是因为本地变量和全局/静态变量是分配在不同类型的内存区域中的结果。对于一个进程的内存空间而言，可以在逻辑上分成3个部份：代码区，静态数据区和动态数据区。动态数据区一般就是“堆栈”。“栈(stack)”和“堆(heap)”是两种不同的动态数据区，栈是一种线性结构，堆是一种链式结构。进程的每个线程都有私有的“栈”，所以每个线程虽然代码一样，但本地变量的数据都是互不干扰。一个堆栈可以通过“基地址”和“栈顶”地址来描述。全局变量和静态变量分配在静态数据区，本地变量分配在动态数据区，即堆栈中。程序通过堆栈的基地址和偏移量来访问本地变量。

├———————┤低端内存区域   
│ …… │   
├———————┤   
│ 动态数据区 │   
├———————┤   
│ …… │   
├———————┤   
│ 代码区 │   
├———————┤   
│ 静态数据区 │   
├———————┤   
│ …… │   
├———————┤高端内存区域

堆栈是一个先进后出的数据结构，栈顶地址总是小于等于栈的基地址。我们可以先了解一下函数调用的过程，以便对堆栈在程序中的作用有更深入的了解。不同的语言有不同的函数调用规定，这些因素有参数的压入规则和堆栈的平衡。windows API的调用规则和ANSI C的函数调用规则是不一样的，前者由被调函数调整堆栈，后者由调用者调整堆栈。两者通过“\_\_stdcall”和“\_\_cdecl”前缀区分。先看下面这段代码：

#include <stdio.h>

void \_\_stdcall func(int param1,int param2,int param3)   
{   
int var1=param1;   
int var2=param2;   
int var3=param3;   
printf("0x%08x\n",¶m1); //打印出各个变量的内存地址   
printf("0x%08x\n",¶m2);   
printf("0x%08x\n\n",¶m3);   
printf("0x%08x\n",&var1);   
printf("0x%08x\n",&var2);   
printf("0x%08x\n\n",&var3);   
return;   
}

int main()   
{   
func(1,2,3);   
return 0;   
}

编译后的执行结果是：

0x0012ff78   
0x0012ff7c   
0x0012ff80

0x0012ff68   
0x0012ff6c   
0x0012ff70

├———————┤<—函数执行时的栈顶（ESP）、低端内存区域   
│ …… │   
├———————┤   
│ var 1 │   
├———————┤   
│ var 2 │   
├———————┤   
│ var 3 │   
├———————┤   
│ RET │   
├———————┤<—“\_\_cdecl”函数返回后的栈顶（ESP）   
│ parameter 1 │   
├———————┤   
│ parameter 2 │   
├———————┤   
│ parameter 3 │   
├———————┤<—“\_\_stdcall”函数返回后的栈顶（ESP）   
│ …… │   
├———————┤<—栈底（基地址 EBP）、高端内存区域

上图就是函数调用过程中堆栈的样子了。首先，三个参数以从又到左的次序压入堆栈，先压“param3”，再压“param2”，最后压入“param1”；然后压入函数的返回地址(RET)，接着跳转到函数地址接着执行（这里要补充一点，介绍UNIX下的缓冲溢出原理的文章中都提到在压入RET后，继续压入当前EBP，然后用当前ESP代替EBP。然而，有一篇介绍windows下函数调用的文章中说，在windows下的函数调用也有这一步骤，但根据我的实际调试，并未发现这一步，这还可以从param3和var1之间只有4字节的间隙这点看出来）；第三步，将栈顶(ESP)减去一个数，为本地变量分配内存空间，上例中是减去12字节(ESP=ESP-3\*4，每个int变量占用4个字节)；接着就初始化本地变量的内存空间。由于“\_\_stdcall”调用由被调函数调整堆栈，所以在函数返回前要恢复堆栈，先回收本地变量占用的内存(ESP=ESP+3\*4)，然后取出返回地址，填入EIP寄存器，回收先前压入参数占用的内存(ESP=ESP+3\*4)，继续执行调用者的代码。参见下列汇编代码：

;--------------func 函数的汇编代码-------------------

:00401000 83EC0C sub esp, 0000000C //创建本地变量的内存空间   
:00401003 8B442410 mov eax, dword ptr [esp+10]   
:00401007 8B4C2414 mov ecx, dword ptr [esp+14]   
:0040100B 8B542418 mov edx, dword ptr [esp+18]   
:0040100F 89442400 mov dword ptr [esp], eax   
:00401013 8D442410 lea eax, dword ptr [esp+10]   
:00401017 894C2404 mov dword ptr [esp+04], ecx

……………………（省略若干代码）

:00401075 83C43C add esp, 0000003C ;恢复堆栈，回收本地变量的内存空间   
:00401078 C3 ret 000C ;函数返回，恢复参数占用的内存空间   
;如果是“\_\_cdecl”的话，这里是“ret”，堆栈将由调用者恢复

;-------------------函数结束-------------------------

;--------------主程序调用func函数的代码--------------

:00401080 6A03 push 00000003 //压入参数param3   
:00401082 6A02 push 00000002 //压入参数param2   
:00401084 6A01 push 00000001 //压入参数param1   
:00401086 E875FFFFFF call 00401000 //调用func函数   
;如果是“\_\_cdecl”的话，将在这里恢复堆栈，“add esp, 0000000C”

聪明的读者看到这里，差不多就明白缓冲溢出的原理了。先来看下面的代码：

#include <stdio.h>   
#include <string.h>

void \_\_stdcall func()   
{   
char lpBuff[8]="\0";   
strcat(lpBuff,"AAAAAAAAAAA");   
return;   
}

int main()   
{   
func();   
return 0;   
}

编译后执行一下回怎么样？哈，“"0x00414141"指令引用的"0x00000000"内存。该内存不能为"read"。”，“非法操作”喽！"41"就是"A"的16进制的ASCII码了，那明显就是strcat这句出的问题了。"lpBuff"的大小只有8字节，算进结尾的\0，那strcat最多只能写入7个"A"，但程序实际写入了11个"A"外加1个\0。再来看看上面那幅图，多出来的4个字节正好覆盖了RET的所在的内存空间，导致函数返回到一个错误的内存地址，执行了错误的指令。如果能精心构造这个字符串，使它分成三部分，前一部份仅仅是填充的无意义数据以达到溢出的目的，接着是一个覆盖RET的数据，紧接着是一段shellcode，那只要着个RET地址能指向这段shellcode的第一个指令，那函数返回时就能执行shellcode了。但是软件的不同版本和不同的运行环境都可能影响这段shellcode在内存中的位置，那么要构造这个RET是十分困难的。一般都在RET和shellcode之间填充大量的NOP指令，使得exploit有更强的通用性。

├———————┤<—低端内存区域   
│ …… │   
├———————┤<—由exploit填入数据的开始   
│ │   
│ buffer │<—填入无用的数据   
│ │   
├———————┤   
│ RET │<—指向shellcode，或NOP指令的范围   
├———————┤   
│ NOP │   
│ …… │<—填入的NOP指令，是RET可指向的范围   
│ NOP │   
├———————┤   
│ │   
│ shellcode │   
│ │   
├———————┤<—由exploit填入数据的结束   
│ …… │   
├———————┤<—高端内存区域

windows下的动态数据除了可存放在栈中，还可以存放在堆中。了解C++的朋友都知道，C++可以使用new关键字来动态分配内存。来看下面的C++代码：

#include <stdio.h>   
#include <iostream.h>   
#include <windows.h>

void func()   
{   
char \*buffer=new char[128];   
char bufflocal[128];   
static char buffstatic[128];   
printf("0x%08x\n",buffer); //打印堆中变量的内存地址   
printf("0x%08x\n",bufflocal); //打印本地变量的内存地址   
printf("0x%08x\n",buffstatic); //打印静态变量的内存地址   
}

void main()   
{   
func();   
return;   
}

程序执行结果为：

0x004107d0   
0x0012ff04   
0x004068c0

可以发现用new关键字分配的内存即不在栈中，也不在静态数据区。VC编译器是通过windows下的“堆(heap)”来实现new关键字的内存动态分配。在讲“堆”之前，先来了解一下和“堆”有关的几个API函数：

HeapAlloc 在堆中申请内存空间   
HeapCreate 创建一个新的堆对象   
HeapDestroy 销毁一个堆对象   
HeapFree 释放申请的内存   
HeapWalk 枚举堆对象的所有内存块   
GetProcessHeap 取得进程的默认堆对象   
GetProcessHeaps 取得进程所有的堆对象   
LocalAlloc   
GlobalAlloc

当进程初始化时，系统会自动为进程创建一个默认堆，这个堆默认所占内存的大小为1M。堆对象由系统进行管理，它在内存中以链式结构存在。通过下面的代码可以通过堆动态申请内存空间：

HANDLE hHeap=GetProcessHeap();   
char \*buff=HeapAlloc(hHeap,0,8);

其中hHeap是堆对象的句柄，buff是指向申请的内存空间的地址。那这个hHeap究竟是什么呢？它的值有什么意义吗？看看下面这段代码吧：

#pragma comment(linker,"/entry:main") //定义程序的入口   
#include <windows.h>

\_CRTIMP int (\_\_cdecl \*printf)(const char \*, ...); //定义STL函数printf   
/\*---------------------------------------------------------------------------   
写到这里，我们顺便来复习一下前面所讲的知识：   
(\*注)printf函数是C语言的标准函数库中函数，VC的标准函数库由msvcrt.dll模块实现。   
由函数定义可见，printf的参数个数是可变的，函数内部无法预先知道调用者压入的参数个数，函数只能通过分析第一个参数字符串的格式来获得压入参数的信息，由于这里参数的个数是动态的，所以必须由调用者来平衡堆栈，这里便使用了\_\_cdecl调用规则。BTW，Windows系统的API函数基本上是\_\_stdcall调用形式，只有一个API例外，那就是wsprintf，它使用\_\_cdecl调用规则，同printf函数一样，这是由于它的参数个数是可变的缘故。   
---------------------------------------------------------------------------\*/   
void main()   
{   
HANDLE hHeap=GetProcessHeap();   
char \*buff=HeapAlloc(hHeap,0,0x10);   
char \*buff2=HeapAlloc(hHeap,0,0x10);   
HMODULE hMsvcrt=LoadLibrary("msvcrt.dll");   
printf=(void \*)GetProcAddress(hMsvcrt,"printf");   
printf("0x%08x\n",hHeap);   
printf("0x%08x\n",buff);   
printf("0x%08x\n\n",buff2);   
}

执行结果为：

0x00130000   
0x00133100   
0x00133118

hHeap的值怎么和那个buff的值那么接近呢？其实hHeap这个句柄就是指向HEAP首部的地址。在进程的用户区存着一个叫PEB(进程环境块)的结构，这个结构中存放着一些有关进程的重要信息，其中在PEB首地址偏移0x18处存放的ProcessHeap就是进程默认堆的地址，而偏移0x90处存放了指向进程所有堆的地址列表的指针。windows有很多API都使用进程的默认堆来存放动态数据，如windows 2000下的所有ANSI版本的函数都是在默认堆中申请内存来转换ANSI字符串到Unicode字符串的。对一个堆的访问是顺序进行的，同一时刻只能有一个线程访问堆中的数据，当多个线程同时有访问要求时，只能排队等待，这样便造成程序执行效率下降。

最后来说说内存中的数据对齐。所位数据对齐，是指数据所在的内存地址必须是该数据长度的整数倍，DWORD数据的内存起始地址能被4除尽，WORD数据的内存起始地址能被2除尽，x86 CPU能直接访问对齐的数据，当他试图访问一个未对齐的数据时，会在内部进行一系列的调整，这些调整对于程序来说是透明的，但是会降低运行速度，所以编译器在编译程序时会尽量保证数据对齐。同样一段代码，我们来看看用VC、Dev-C++和lcc三个不同编译器编译出来的程序的执行结果：

#include <stdio.h>

int main()   
{   
int a;   
char b;   
int c;   
printf("0x%08x\n",&a);   
printf("0x%08x\n",&b);   
printf("0x%08x\n",&c);   
return 0;   
}

这是用VC编译后的执行结果：   
0x0012ff7c   
0x0012ff7b   
0x0012ff80   
变量在内存中的顺序：b(1字节)-a(4字节)-c(4字节)。

这是用Dev-C++编译后的执行结果：   
0x0022ff7c   
0x0022ff7b   
0x0022ff74   
变量在内存中的顺序：c(4字节)-中间相隔3字节-b(占1字节)-a(4字节)。

这是用lcc编译后的执行结果：   
0x0012ff6c   
0x0012ff6b   
0x0012ff64   
变量在内存中的顺序：同上。

三个编译器都做到了数据对齐，但是后两个编译器显然没VC“聪明”，让一个char占了4字节，浪费内存哦。

基础知识：   
堆栈是一种简单的数据结构，是一种只允许在其一端进行插入或删除的线性表。允许插入或删除操作的一端称为栈顶，另一端称为栈底，对堆栈的插入和删除操作被称为入栈和出栈。有一组CPU指令可以实现对进程的内存实现堆栈访问。其中，POP指令实现出栈操作，PUSH指令实现入栈操作。CPU的ESP寄存器存放当前线程的栈顶指针，EBP寄存器中保存当前线程的栈底指针。CPU的EIP寄存器存放下一个CPU指令存放的内存地址，当CPU执行完当前的指令后，从EIP寄存器中读取下一条指令的内存地址，然后继续执行。

参考：《Windows下的HEAP溢出及其利用》by: isno   
《windows核心编程》by: Jeffrey Richter 

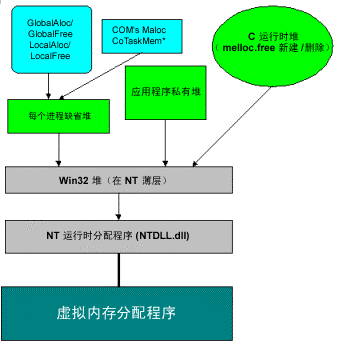
摘要： 讨论常见的堆性能问题以及如何防范它们。（共 9 页）

前言  
您是否是动态分配的 C/C++ 对象忠实且幸运的用户？您是否在模块间的往返通信中频繁地使用了“自动化”？您的程序是否因堆分配而运行起来很慢？不仅仅您遇到这样的问题。几乎所有项目迟早都会遇到堆问题。大家都想说，“我的代码真正好，只是堆太慢”。那只是部分正确。更深入理解堆及其用法、以及会发生什么问题，是很有用的。

什么是堆？  
（如果您已经知道什么是堆，可以跳到“什么是常见的堆性能问题？”部分）

在程序中，使用堆来动态分配和释放对象。在下列情况下，调用堆操作：

事先不知道程序所需对象的数量和大小。

对象太大而不适合堆栈分配程序。  
堆使用了在运行时分配给代码和堆栈的内存之外的部分内存。下图给出了堆分配程序的不同层。  
  


GlobalAlloc/GlobalFree：Microsoft Win32 堆调用，这些调用直接与每个进程的默认堆进行对话。

LocalAlloc/LocalFree：Win32 堆调用（为了与 Microsoft Windows NT 兼容），这些调用直接与每个进程的默认堆进行对话。

COM 的 IMalloc 分配程序（或 CoTaskMemAlloc / CoTaskMemFree）：函数使用每个进程的默认堆。自动化程序使用“组件对象模型 (COM)”的分配程序，而申请的程序使用每个进程堆。

C/C++ 运行时 (CRT) 分配程序：提供了 malloc() 和 free() 以及 new 和 delete 操作符。如 Microsoft Visual Basic 和 Java 等语言也提供了新的操作符并使用垃圾收集来代替堆。CRT 创建自己的私有堆，驻留在 Win32 堆的顶部。

Windows NT 中，Win32 堆是 Windows NT 运行时分配程序周围的薄层。所有 API 转发它们的请求给 NTDLL。

Windows NT 运行时分配程序提供 Windows NT 内的核心堆分配程序。它由具有 128 个大小从 8 到 1,024 字节的空闲列表的前端分配程序组成。后端分配程序使用虚拟内存来保留和提交页。

在图表的底部是“虚拟内存分配程序”，操作系统使用它来保留和提交页。所有分配程序使用虚拟内存进行数据的存取。

分配和释放块不就那么简单吗？为何花费这么长时间？

堆实现的注意事项  
传统上，操作系统和运行时库是与堆的实现共存的。在一个进程的开始，操作系统创建一个默认堆，叫做“进程堆”。如果没有其他堆可使用，则块的分配使用“进程堆”。语言运行时也能在进程内创建单独的堆。（例如，C 运行时创建它自己的堆。）除这些专用的堆外，应用程序或许多已载入的动态链接库 (DLL) 之一可以创建和使用单独的堆。Win32 提供一整套 API 来创建和使用私有堆。有关堆函数（英文）的详尽指导，请参见 MSDN。

当应用程序或 DLL 创建私有堆时，这些堆存在于进程空间，并且在进程内是可访问的。从给定堆分配的数据将在同一个堆上释放。（不能从一个堆分配而在另一个堆释放。）

在所有虚拟内存系统中，堆驻留在操作系统的“虚拟内存管理器”的顶部。语言运行时堆也驻留在虚拟内存顶部。某些情况下，这些堆是操作系统堆中的层，而语言运行时堆则通过大块的分配来执行自己的内存管理。不使用操作系统堆，而使用虚拟内存函数更利于堆的分配和块的使用。

典型的堆实现由前、后端分配程序组成。前端分配程序维持固定大小块的空闲列表。对于一次分配调用，堆尝试从前端列表找到一个自由块。如果失败，堆被迫从后端（保留和提交虚拟内存）分配一个大块来满足请求。通用的实现有每块分配的开销，这将耗费执行周期，也减少了可使用的存储空间。

Knowledge Base 文章 Q10758，“用 calloc() 和 malloc() 管理内存” （搜索文章编号）, 包含了有关这些主题的更多背景知识。另外，有关堆实现和设计的详细讨论也可在下列著作中找到：“Dynamic Storage Allocation: A Survey and Critical Review”，作者 Paul R. Wilson、Mark S. Johnstone、Michael Neely 和 David Boles；“International Workshop on Memory Management”, 作者 Kinross, Scotland, UK, 1995 年 9 月(<http://www.cs.utexas.edu/users/oops/papers.html>)（英文）。

Windows NT 的实现（Windows NT 版本 4.0 和更新版本） 使用了 127 个大小从 8 到 1,024 字节的 8 字节对齐块空闲列表和一个“大块”列表。“大块”列表（空闲列表[0]） 保存大于 1,024 字节的块。空闲列表容纳了用双向链表链接在一起的对象。默认情况下，“进程堆”执行收集操作。（收集是将相邻空闲块合并成一个大块的操作。）收集耗费了额外的周期，但减少了堆块的内部碎片。

单一全局锁保护堆，防止多线程式的使用。（请参见“Server Performance and Scalability Killers”中的第一个注意事项, George Reilly 所著，在 “MSDN Online Web Workshop”上（站点：<http://msdn.microsoft.com/workshop/server/iis/tencom.asp>（英文）。）单一全局锁本质上是用来保护堆数据结构，防止跨多线程的随机存取。若堆操作太频繁，单一全局锁会对性能有不利的影响。

什么是常见的堆性能问题？  
以下是您使用堆时会遇到的最常见问题：

分配操作造成的速度减慢。光分配就耗费很长时间。最可能导致运行速度减慢原因是空闲列表没有块，所以运行时分配程序代码会耗费周期寻找较大的空闲块，或从后端分配程序分配新块。

释放操作造成的速度减慢。释放操作耗费较多周期，主要是启用了收集操作。收集期间，每个释放操作“查找”它的相邻块，取出它们并构造成较大块，然后再把此较大块插入空闲列表。在查找期间，内存可能会随机碰到，从而导致高速缓存不能命中，性能降低。

堆竞争造成的速度减慢。当两个或多个线程同时访问数据，而且一个线程继续进行之前必须等待另一个线程完成时就发生竞争。竞争总是导致麻烦；这也是目前多处理器系统遇到的最大问题。当大量使用内存块的应用程序或 DLL 以多线程方式运行（或运行于多处理器系统上）时将导致速度减慢。单一锁定的使用—常用的解决方案—意味着使用堆的所有操作是序列化的。当等待锁定时序列化会引起线程切换上下文。可以想象交叉路口闪烁的红灯处走走停停导致的速度减慢。   
竞争通常会导致线程和进程的上下文切换。上下文切换的开销是很大的，但开销更大的是数据从处理器高速缓存中丢失，以及后来线程复活时的数据重建。

堆破坏造成的速度减慢。造成堆破坏的原因是应用程序对堆块的不正确使用。通常情形包括释放已释放的堆块或使用已释放的堆块，以及块的越界重写等明显问题。（破坏不在本文讨论范围之内。有关内存重写和泄漏等其他细节，请参见 Microsoft Visual C++(R) 调试文档 。）

频繁的分配和重分配造成的速度减慢。这是使用脚本语言时非常普遍的现象。如字符串被反复分配，随重分配增长和释放。不要这样做，如果可能，尽量分配大字符串和使用缓冲区。另一种方法就是尽量少用连接操作。  
竞争是在分配和释放操作中导致速度减慢的问题。理想情况下，希望使用没有竞争和快速分配/释放的堆。可惜，现在还没有这样的通用堆，也许将来会有。

在所有的服务器系统中（如 IIS、MSProxy、DatabaseStacks、网络服务器、 Exchange 和其他）, 堆锁定实在是个大瓶颈。处理器数越多，竞争就越会恶化。

尽量减少堆的使用  
现在您明白使用堆时存在的问题了，难道您不想拥有能解决这些问题的超级魔棒吗？我可希望有。但没有魔法能使堆运行加快—因此不要期望在产品出货之前的最后一星期能够大为改观。如果提前规划堆策略，情况将会大大好转。调整使用堆的方法，减少对堆的操作是提高性能的良方。

如何减少使用堆操作？通过利用数据结构内的位置可减少堆操作的次数。请考虑下列实例：

struct ObjectA {  
   // objectA 的数据   
}

struct ObjectB {  
   // objectB 的数据   
}

// 同时使用 objectA 和 objectB

//  
// 使用指针   
//  
struct ObjectB {  
   struct ObjectA \* pObjA;  
   // objectB 的数据   
}

//  
// 使用嵌入  
//  
struct ObjectB {  
   struct ObjectA pObjA;  
   // objectB 的数据   
}

//  
// 集合 – 在另一对象内使用 objectA 和 objectB  
//

struct ObjectX {  
   struct ObjectA  objA;  
   struct ObjectB  objB;  
}

避免使用指针关联两个数据结构。如果使用指针关联两个数据结构，前面实例中的对象 A 和 B 将被分别分配和释放。这会增加额外开销—我们要避免这种做法。

把带指针的子对象嵌入父对象。当对象中有指针时，则意味着对象中有动态元素（百分之八十）和没有引用的新位置。嵌入增加了位置从而减少了进一步分配/释放的需求。这将提高应用程序的性能。

合并小对象形成大对象（聚合）。聚合减少分配和释放的块的数量。如果有几个开发者，各自开发设计的不同部分，则最终会有许多小对象需要合并。集成的挑战就是要找到正确的聚合边界。

内联缓冲区能够满足百分之八十的需要（aka 80-20 规则）。个别情况下，需要内存缓冲区来保存字符串/二进制数据，但事先不知道总字节数。估计并内联一个大小能满足百分之八十需要的缓冲区。对剩余的百分之二十，可以分配一个新的缓冲区和指向这个缓冲区的指针。这样，就减少分配和释放调用并增加数据的位置空间，从根本上提高代码的性能。

在块中分配对象（块化）。块化是以组的方式一次分配多个对象的方法。如果对列表的项连续跟踪，例如对一个 {名称，值} 对的列表，有两种选择：选择一是为每一个“名称-值”对分配一个节点；选择二是分配一个能容纳（如五个）“名称-值”对的结构。例如，一般情况下，如果存储四对，就可减少节点的数量，如果需要额外的空间数量，则使用附加的链表指针。   
块化是友好的处理器高速缓存，特别是对于 L1-高速缓存，因为它提供了增加的位置 —不用说对于块分配，很多数据块会在同一个虚拟页中。

正确使用 \_amblksiz。C 运行时 (CRT) 有它的自定义前端分配程序，该分配程序从后端（Win32 堆）分配大小为 \_amblksiz 的块。将 \_amblksiz 设置为较高的值能潜在地减少对后端的调用次数。这只对广泛使用 CRT 的程序适用。  
使用上述技术将获得的好处会因对象类型、大小及工作量而有所不同。但总能在性能和可升缩性方面有所收获。另一方面，代码会有点特殊，但如果经过深思熟虑，代码还是很容易管理的。

其他提高性能的技术  
下面是一些提高速度的技术：

使用 Windows NT5 堆   
由于几个同事的努力和辛勤工作，1998 年初 Microsoft Windows(R) 2000 中有了几个重大改进：

改进了堆代码内的锁定。堆代码对每堆一个锁。全局锁保护堆数据结构，防止多线程式的使用。但不幸的是，在高通信量的情况下，堆仍受困于全局锁，导致高竞争和低性能。Windows 2000 中，锁内代码的临界区将竞争的可能性减到最小,从而提高了可伸缩性。

使用 “Lookaside”列表。堆数据结构对块的所有空闲项使用了大小在 8 到 1,024 字节（以 8-字节递增）的快速高速缓存。快速高速缓存最初保护在全局锁内。现在，使用 lookaside 列表来访问这些快速高速缓存空闲列表。这些列表不要求锁定，而是使用 64 位的互锁操作，因此提高了性能。

内部数据结构算法也得到改进。  
这些改进避免了对分配高速缓存的需求，但不排除其他的优化。使用 Windows NT5 堆评估您的代码；它对小于 1,024 字节 (1 KB) 的块（来自前端分配程序的块）是最佳的。GlobalAlloc() 和 LocalAlloc() 建立在同一堆上，是存取每个进程堆的通用机制。如果希望获得高的局部性能，则使用 Heap(R) API 来存取每个进程堆，或为分配操作创建自己的堆。如果需要对大块操作，也可以直接使用 VirtualAlloc() / VirtualFree() 操作。

上述改进已在 Windows 2000 beta 2 和 Windows NT 4.0 SP4 中使用。改进后，堆锁的竞争率显著降低。这使所有 Win32 堆的直接用户受益。CRT 堆建立于 Win32 堆的顶部，但它使用自己的小块堆，因而不能从 Windows NT 改进中受益。（Visual C++ 版本 6.0 也有改进的堆分配程序。）

使用分配高速缓存   
分配高速缓存允许高速缓存分配的块，以便将来重用。这能够减少对进程堆（或全局堆）的分配/释放调用的次数，也允许最大限度的重用曾经分配的块。另外，分配高速缓存允许收集统计信息,以便较好地理解对象在较高层次上的使用。

典型地，自定义堆分配程序在进程堆的顶部实现。自定义堆分配程序与系统堆的行为很相似。主要的差别是它在进程堆的顶部为分配的对象提供高速缓存。高速缓存设计成一套固定大小（如 32 字节、64 字节、128 字节等）。这一个很好的策略，但这种自定义堆分配程序丢失与分配和释放的对象相关的“语义信息”。

与自定义堆分配程序相反，“分配高速缓存”作为每类分配高速缓存来实现。除能够提供自定义堆分配程序的所有好处之外，它们还能够保留大量语义信息。每个分配高速缓存处理程序与一个目标二进制对象关联。它能够使用一套参数进行初始化，这些参数表示并发级别、对象大小和保持在空闲列表中的元素的数量等。分配高速缓存处理程序对象维持自己的私有空闲实体池（不超过指定的阀值）并使用私有保护锁。合在一起，分配高速缓存和私有锁减少了与主系统堆的通信量，因而提供了增加的并发、最大限度的重用和较高的可伸缩性。

需要使用清理程序来定期检查所有分配高速缓存处理程序的活动情况并回收未用的资源。如果发现没有活动，将释放分配对象的池，从而提高性能。

可以审核每个分配/释放活动。第一级信息包括对象、分配和释放调用的总数。通过查看它们的统计信息可以得出各个对象之间的语义关系。利用以上介绍的许多技术之一，这种关系可以用来减少内存分配。

分配高速缓存也起到了调试助手的作用，帮助您跟踪没有完全清除的对象数量。通过查看动态堆栈返回踪迹和除没有清除的对象之外的签名，甚至能够找到确切的失败的调用者。

MP 堆   
MP 堆是对多处理器友好的分布式分配的程序包，在 Win32 SDK（Windows NT 4.0 和更新版本）中可以得到。最初由 JVert 实现，此处堆抽象建立在 Win32 堆程序包的顶部。MP 堆创建多个 Win32 堆，并试图将分配调用分布到不同堆，以减少在所有单一锁上的竞争。

本程序包是好的步骤 —一种改进的 MP-友好的自定义堆分配程序。但是，它不提供语义信息和缺乏统计功能。通常将 MP 堆作为 SDK 库来使用。如果使用这个 SDK 创建可重用组件，您将大大受益。但是，如果在每个 DLL 中建立这个 SDK 库，将增加工作设置。

重新思考算法和数据结构   
要在多处理器机器上伸缩，则算法、实现、数据结构和硬件必须动态伸缩。请看最经常分配和释放的数据结构。试问，“我能用不同的数据结构完成此工作吗？”例如，如果在应用程序初始化时加载了只读项的列表，这个列表不必是线性链接的列表。如果是动态分配的数组就非常好。动态分配的数组将减少内存中的堆块和碎片，从而增强性能。

减少需要的小对象的数量减少堆分配程序的负载。例如，我们在服务器的关键处理路径上使用五个不同的对象，每个对象单独分配和释放。一起高速缓存这些对象，把堆调用从五个减少到一个，显著减少了堆的负载，特别当每秒钟处理 1,000 个以上的请求时。

如果大量使用“Automation”结构，请考虑从主线代码中删除“Automation BSTR”，或至少避免重复的 BSTR 操作。（BSTR 连接导致过多的重分配和分配/释放操作。）

摘要  
对所有平台往往都存在堆实现，因此有巨大的开销。每个单独代码都有特定的要求，但设计能采用本文讨论的基本理论来减少堆之间的相互作用。

评价您的代码中堆的使用。

改进您的代码，以使用较少的堆调用：分析关键路径和固定数据结构。

在实现自定义的包装程序之前使用量化堆调用成本的方法。

如果对性能不满意，请要求 OS 组改进堆。更多这类请求意味着对改进堆的更多关注。

要求 C 运行时组针对 OS 所提供的堆制作小巧的分配包装程序。随着 OS 堆的改进，C 运行时堆调用的成本将减小。

操作系统（Windows NT 家族）正在不断改进堆。请随时关注和利用这些改进。  
Murali Krishnan 是 Internet Information Server (IIS) 组的首席软件设计工程师。从 1.0 版本开始他就设计 IIS，并成功发行了 1.0 版本到 4.0 版本。Murali 组织并领导 IIS 性能组三年 (1995-1998), 从一开始就影响 IIS 性能。他拥有威斯康星州 Madison 大学的 M.S.和印度 Anna 大学的 B.S.。工作之外，他喜欢阅读、打排球和家庭烹饪。  
  
  
  
<http://community.csdn.net/Expert/FAQ/FAQ_Index.asp?id=172835>  
我在学习对象的生存方式的时候见到一种是在堆栈(stack)之中，如下    
CObject  object;    
还有一种是在堆(heap)中  如下    
CObject\*  pobject=new  CObject();    
   
请问    
（1）这两种方式有什么区别？    
（2）堆栈与堆有什么区别？？    
   
   
---------------------------------------------------------------    
   
1)  about  stack,  system  will  allocate  memory  to  the  instance  of  object  automatically,  and  to  the  
 heap,  you  must  allocate  memory  to  the  instance  of  object  with  new  or  malloc  manually.    
2)  when  function  ends,  system  will  automatically  free  the  memory  area  of  stack,  but  to  the   
heap,  you  must  free  the  memory  area  manually  with  free  or  delete,  else  it  will  result  in  memory  
leak.    
3)栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。    
4）堆上分配的内存可以有我们自己决定，使用非常灵活。

posted @ [2008-09-09 19:00](http://www.cppblog.com/road420/archive/2008/09/09/61402.html) 深邃者 阅读(135) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/09/09/61402.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=61402) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=61402)

**[Ansi、Unicode、UTF8字符串之间的转换和写入文本文件](http://www.cppblog.com/road420/archive/2008/09/04/60919.html)**

[Ansi、Unicode、UTF8字符串之间的转换和写入文本文件](http://www.cppblog.com/greatws/archive/2008/08/31/60546.html)

转载请注明出处<http://www.cppblog.com/greatws/archive/2008/08/31/60546.html>  
  
最近有人问我关于这个的问题，就此写一篇blog  
  
Ansi字符串我们最熟悉，英文占一个字节，汉字2个字节，以一个\0结尾，常用于txt文本文件  
Unicode字符串，每个字符(汉字、英文字母)都占2个字节，以2个连续的\0结尾，NT操作系统内核用的是这种字符串，常被定义为typedef unsigned short wchar\_t;所以我们有时常会见到什么char\*无法转换为unsigned short\*之类的错误，其实就是unicode  
UTF8是Unicode一种压缩形式，英文A在unicode中表示为0x0041，老外觉得这种存储方式太浪费，因为浪费了50%的空间，于是就把英文压缩成1个字节，成了utf8编码，但是汉字在utf8中占3个字节，显然用做中文不如ansi合算，这就是中国的网页用作ansi编码而老外的网页常用utf8的原因。  
UTF8在还游戏里运用的很广泛，比如WOW的lua脚本等  
  
下面来说一下转换，主要用代码来说明吧  
写文件我用了CFile类，其实用FILE\*之类的也是一样，写文件和字符串什么类别没有关系，硬件只关心数据和长度  
  
Ansi转Unicode  
介绍2种方法

http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CConvertDlg::OnBnClickedButtonAnsiToUnicode()  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    // ansi to unicode  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    char\* szAnsi = "abcd1234你我他";  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //预转换，得到所需空间的大小  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    int wcsLen = ::MultiByteToWideChar(CP\_ACP, NULL, szAnsi, strlen(szAnsi), NULL, 0);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //分配空间要给'\0'留个空间，MultiByteToWideChar不会给'\0'空间  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wchar\_t\* wszString = new wchar\_t[wcsLen + 1];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //转换  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MultiByteToWideChar(CP\_ACP, NULL, szAnsi, strlen(szAnsi), wszString, wcsLen);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //最后加上'\0'  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wszString[wcsLen] = '\0';  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //unicode版的MessageBox API  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MessageBoxW(GetSafeHwnd(), wszString, wszString, MB\_OK);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //接下来写入文本  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写文本文件，头2个字节0xfeff，低位0xff写在前  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CFile cFile;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Open(\_T("1.txt"), CFile::modeWrite | CFile::modeCreate);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //文件开头  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.SeekToBegin();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Write("\xff\xfe", 2);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写入内容  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Write(wszString, wcsLen \* sizeof(wchar\_t));  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Flush();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Close();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    delete[] wszString;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wszString =NULL;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //方法2  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //设置当前地域信息，不设置的话，使用这种方法，中文不会正确显示  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //需要#include<locale.h>  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    setlocale(LC\_CTYPE, "chs");   
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wchar\_t wcsStr[100];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //注意下面是大写S，在unicode中，代表后面是ansi字符串  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //swprintf是sprintf的unicode版本  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //格式的前面要加大写L，代表是unicode  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    swprintf(wcsStr, L"%S", szAnsi);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MessageBoxW(GetSafeHwnd(), wcsStr, wcsStr, MB\_OK);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

Unicode转Ansi  
也是2种方法

http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CConvertDlg::OnBnClickedButtonUnicodeToAnsi()  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    // unicode to ansi  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wchar\_t\* wszString = L"abcd1234你我他";  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //预转换，得到所需空间的大小，这次用的函数和上面名字相反  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    int ansiLen = ::WideCharToMultiByte(CP\_ACP, NULL, wszString, wcslen(wszString), NULL, 0, NULL, NULL);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //同上，分配空间要给'\0'留个空间  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    char\* szAnsi = new char[ansiLen + 1];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //转换  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //unicode版对应的strlen是wcslen  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::WideCharToMultiByte(CP\_ACP, NULL, wszString, wcslen(wszString), szAnsi, ansiLen, NULL, NULL);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //最后加上'\0'  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    szAnsi[ansiLen] = '\0';  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //Ansi版的MessageBox API  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MessageBoxA(GetSafeHwnd(), szAnsi, szAnsi, MB\_OK);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //接下来写入文本  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写文本文件，ANSI文件没有BOM  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CFile cFile;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Open(\_T("1.txt"), CFile::modeWrite | CFile::modeCreate);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //文件开头  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.SeekToBegin();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写入内容  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Write(szAnsi, ansiLen \* sizeof(char));  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Flush();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Close();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    delete[] szAnsi;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    szAnsi =NULL;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //方法2  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //和上面一样有另一种方法  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    setlocale(LC\_CTYPE, "chs");   
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    char szStr[100];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //注意下面是大写，在ansi中，代表后面是unicode字符串  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //sprintf  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    sprintf(szStr, "%S", wszString);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MessageBoxA(GetSafeHwnd(), szStr, szStr, MB\_OK);  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

Unicode转UTF8

http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CConvertDlg::OnBnClickedButtonUnicodeToU8()  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    // unicode to UTF8  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wchar\_t\* wszString = L"abcd1234你我他";  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //预转换，得到所需空间的大小，这次用的函数和上面名字相反  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    int u8Len = ::WideCharToMultiByte(CP\_UTF8, NULL, wszString, wcslen(wszString), NULL, 0, NULL, NULL);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //同上，分配空间要给'\0'留个空间  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //UTF8虽然是Unicode的压缩形式，但也是多字节字符串，所以可以以char的形式保存  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    char\* szU8 = new char[u8Len + 1];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //转换  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //unicode版对应的strlen是wcslen  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::WideCharToMultiByte(CP\_UTF8, NULL, wszString, wcslen(wszString), szU8, u8Len, NULL, NULL);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //最后加上'\0'  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    szU8[u8Len] = '\0';  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //MessageBox不支持UTF8,所以只能写文件  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //接下来写入文本  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写文本文件，UTF8的BOM是0xbfbbef  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    CFile cFile;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Open(\_T("1.txt"), CFile::modeWrite | CFile::modeCreate);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //文件开头  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.SeekToBegin();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写BOM，同样低位写在前  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Write("\xef\xbb\xbf", 3);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写入内容  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Write(szU8, u8Len \* sizeof(char));  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Flush();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    cFile.Close();  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    delete[] szU8;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    szU8 =NULL;  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}

UTF8转UNICODE

http://www.cppblog.com/Images/OutliningIndicators/None.gifvoid CConvertDlg::OnBnClickedButtonU8ToUnicode()  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockStart.gif{  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //UTF8 to Unicode  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //由于中文直接复制过来会成乱码，编译器有时会报错，故采用16进制形式  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    char\* szU8 = "abcd1234\xe4\xbd\xa0\xe6\x88\x91\xe4\xbb\x96\x00";  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //预转换，得到所需空间的大小  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    int wcsLen = ::MultiByteToWideChar(CP\_UTF8, NULL, szU8, strlen(szU8), NULL, 0);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //分配空间要给'\0'留个空间，MultiByteToWideChar不会给'\0'空间  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wchar\_t\* wszString = new wchar\_t[wcsLen + 1];  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //转换  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MultiByteToWideChar(CP\_UTF8, NULL, szU8, strlen(szU8), wszString, wcsLen);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //最后加上'\0'  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    wszString[wcsLen] = '\0';  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //unicode版的MessageBox API  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    ::MessageBoxW(GetSafeHwnd(), wszString, wszString, MB\_OK);  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif  
http://www.cppblog.com/Images/OutliningIndicators/InBlock.gif    //写文本同ansi to unicode  
http://www.cppblog.com/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
http://www.cppblog.com/Images/OutliningIndicators/None.gif

Ansi转换utf8和utf8转换Ansi就是上面2个的结合，把unicode作为中间量，进行2次转换即可

posted @ [2008-09-04 16:27](http://www.cppblog.com/road420/archive/2008/09/04/60919.html) 深邃者 阅读(371) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/09/04/60919.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=60919) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=60919)

[**读写ini文件**](http://www.cppblog.com/road420/archive/2008/09/02/60699.html)

|  |
| --- |
| ini文件(即Initialization file)，这种类型的文件中通常存放的是一个程序的初始化信息。ini文件由若干个节(Section)组成，每个Section由若干键(Key)组成，每个Key可以赋相应的值。读写ini文件实际上就是读写某个的Section中相应的Key的值，而这只要借助几个函数即可完成。 **一、向ini文件中写入信息的函数 1. 把信息写入系统的win.ini文件** BOOL WriteProfileString(      LPCTSTR lpAppName, // 节的名字，是一个以0结束的字符串      LPCTSTR lpKeyName, // 键的名字，是一个以0结束的字符串。若为NULL，则删除整个节      LPCTSTR lpString      // 键的值，是一个以0结束的字符串。若为NULL，则删除对应的键 )  **2. 把信息写入自己定义的.ini文件** BOOL WritePrivateProfileString(      LPCTSTR lpAppName,     // 同上      LPCTSTR lpKeyName,     // 同上      LPCTSTR lpString,      // 同上      LPCTSTR lpFileName     // 要写入的文件的文件名。若该ini文件与程序在同一个目录下，也可使用相对            //路径,否则需要给出绝度路径。 )  如： ::WriteProfileString("Test","id","xym");   //在win.ini中创建一个Test节，并在该节中创建一个键id,其值为xym  ::WritePrivateProfileString("Test","id","xym","d:\\vc\\Ex1\\ex1.ini"); //在Ex1目录下的ex1.ini中创建一个Test节，并在该节中创建一个键id,其值为xym  //若Ex1.ini文件与读写该文件的程序在同一个目录下，则上面语句也可写为： ::WritePrivateProfileString("Test","id","xym",".\\ex1.ini");  需要注意的是，C系列的语言中，转义字符'\\'表示反斜线'\'。另外，当使用相对路径时，\\前的.号不能丢掉了。  **二、从ini文件中读取数据的函数** **1、从系统的win.ini文件中读取信息 (1) 读取字符串** DWORD GetProfileString(      LPCTSTR lpAppName,           // 节名      LPCTSTR lpKeyName,           // 键名，读取该键的值      LPCTSTR lpDefault,           // 若指定的键不存在，该值作为读取的默认值      LPTSTR lpReturnedString,     // 一个指向缓冲区的指针，接收读取的字符串      DWORD nSize                  // 指定lpReturnedString指向的缓冲区的大小 )  如： CString str; ::GetProfileString("Test","id","Error",str.GetBuffer(20),20);  **(2) 读取整数** UINT GetProfileInt(      LPCTSTR lpAppName,     // 同上      LPCTSTR lpKeyName,     // 同上      INT nDefault           // 若指定的键名不存在，该值作为读取的默认值 )  如使用以下语句写入了年龄信息： ::WriteProfileString("Test","age","25");   //在win.ini中创建一个Test节，并在该节中创建一个键age,其值为25  则可用以下语句读取age键的值： int age; age=::GetProfileInt("Test","age",0);  **2、从自己的ini文件中读取信息** **(1) 读取字符串** DWORD GetPrivateProfileString(      LPCTSTR lpAppName,           // 同1(1)      LPCTSTR lpKeyName,           // 同1(1)      LPCTSTR lpDefault,           // 同1(1)      LPTSTR lpReturnedString,     // 同1(1)      DWORD nSize,                 // 同1(1)      LPCTSTR lpFileName           // 读取信息的文件名。若该ini文件与程序在同一个目录下，也可使用相                  //对路径,否则需要给出绝度路径。 )  如： CString str; ::GetPrivateProfileString("Test","id","Error",str.GetBuffer(20),20，".\\ex1.ini"); 或： ::GetPrivateProfileString("Test","id","Error",str.GetBuffer(20),20，"d:\\vc\\Ex1\\ex1.ini");  **(2) 读取整数**  UINT GetPrivateProfileInt(      LPCTSTR lpAppName,     // 同上      LPCTSTR lpKeyName,     // 同上      INT nDefault,          // 若指定的键名不存在，该值作为读取的默认值      LPCTSTR lpFileName     // 同上 )  如使用以下语句写入了年龄信息： ::WritePrivateProfileString("Test","age","25",".\\ex1.ini");   //在ex1.ini中创建一个Test节，并在该节中创建一个键age,其值为25  则可用以下语句读取age键的值： int age; age=::GetPrivateProfileInt("Test","age",0,".\\ex1.ini");  **三、 删除键值或节**        回顾一下WriteProfileString函数的说明 BOOL WriteProfileString(      LPCTSTR lpAppName, // 节的名字，是一个以0结束的字符串      LPCTSTR lpKeyName, // 键的名字，是一个以0结束的字符串。若为NULL，则删除整个节      LPCTSTR lpString      // 键的值，是一个以0结束的字符串。若为NULL，则删除对应的键 )        由此可见，要删除某个节，只需要将WriteProfileString第二个参数设为NULL即可。而要删除某个键，则只需要将该函数的第三个参数设为NULL即可。这是删除系统的win.ini中的节或键，类似的，要删除自己定义的ini文件中的节或键，也可做相同的操作。       如： ::WriteProfileString("Test",NULL,NULL);     //删除win.ini中的Test节 ::WriteProfileString("Test","id",NULL);     //删除win.ini中的id键  ::WritePrivateProfileString("Test",NULL,NULL,".\\ex1.ini");     //删除ex1.ini中的Test节 ::WritePrivateProfileString("Test","id",NULL,".\\ex1.ini");     //删除ex1.ini中的id键  **四、如何判断一个ini文件中有多少个节**       要判断一个ini文件中有多少个节，最简单的办法就是将所有的节名都找出来，然后统计节名的个数。而要将所有的节名找出来，使用GetPrivateProfileSectionNames函数就可以了，其原型如下： DWORD GetPrivateProfileSectionNames(      LPTSTR lpszReturnBuffer,     // 指向一个缓冲区，用来保存返回的所有节名      DWORD nSize,                 // 参数lpszReturnBuffer的大小      LPCTSTR lpFileName           // 文件名，若该ini文件与程序在同一个目录下，                                                 //也可使用相对路径,否则需要给出绝度路径 )  下面的是用来统计一个ini文件中共有多少个节的函数，当然，如果需要同时找到每个节中的各个键及其值，根据找到节名就可以很容易的得到了。  /\*统计共有多少个节 节名的分离方法：若chSectionNames数组的第一字符是'\0'字符，则表明 有0个节。否则，从chSectionNames数组的第一个字符开始，顺序往后找， 直到找到一个'\0'字符，若该字符的后继字符不是 '\0'字符，则表明前 面的字符组成一个节名。若连续找到两个'\0'字符，则统计结束\*/  int CTestDlg::CalcCount(void) { TCHAR      chSectionNames[2048]={0};      //所有节名组成的字符数组 char \* pSectionName; //保存找到的某个节名字符串的首地址 int i;      //i指向数组chSectionNames的某个位置，从0开始，顺序后移 int j=0;     //j用来保存下一个节名字符串的首地址相对于当前i的位置偏移量 int count=0;     //统计节的个数  //CString name; //char id[20]; ::GetPrivateProfileSectionNames(chSectionNames,2048,".\\ex1.ini");    for(i=0;i<2048;i++,j++) {      if(chSectionNames[0]=='\0')       break;      //如果第一个字符就是0，则说明ini中一个节也没有      if(chSectionNames[i]=='\0')      {       pSectionName=&chSectionNames[i-j]; //找到一个0，则说明从这个字符往前，减掉j个偏移量，            //就是一个节名的首地址        j=-1;        //找到一个节名后，j的值要还原，以统计下一个节名地址的偏移量            //赋成-1是因为节名字符串的最后一个字符0是终止符，不能作为节名             //的一部分       /\*::GetPrivateProfileString(pSectionName,"id","Error",id,20,".\\ex1.ini");       name.Format("%s",id);\*/          //在获取节名的时候可以获取该节中键的值，前提是我们知道该节中有哪些键。           AfxMessageBox(pSectionName);     //把找到的显示出来        if(chSectionNames[i+1]==0)       {         break;     //当两个相邻的字符都是0时，则所有的节名都已找到，循环终止       }      }       }  return count; } |

posted @ [2008-09-02 15:19](http://www.cppblog.com/road420/archive/2008/09/02/60699.html) 深邃者 阅读(213) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/09/02/60699.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=60699) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=60699)

[**Window API函数大全(2)**](http://www.cppblog.com/road420/archive/2008/08/28/60231.html)

10. API之硬件与系统函数  
ActivateKeyboardLayout 激活一个新的键盘布局。键盘布局定义了按键在一种物理性键盘上的位置与含义  
Beep 用于生成简单的声音  
CharToOem 将一个字串从ANSI字符集转换到OEM字符集  
ClipCursor 将指针限制到指定区域  
ConvertDefaultLocale 将一个特殊的地方标识符转换成真实的地方ID  
CreateCaret 根据指定的信息创建一个插入符（光标），并将它选定为指定窗口的默认插入符  
DestroyCaret 清除（破坏）一个插入符  
EnumCalendarInfo 枚举在指定“地方”环境中可用的日历信息  
EnumDateFormats 列举指定的“当地”设置中可用的长、短日期格式  
EnumSystemCodePages 枚举系统中已安装或支持的代码页  
EnumSystemLocales 枚举系统已经安装或提供支持的“地方”设置  
EnumTimeFormats 枚举一个指定的地方适用的时间格式  
ExitWindowsEx 退出windows，并用特定的选项重新启动  
ExpandEnvironmentStrings 扩充环境字串  
FreeEnvironmentStrings 翻译指定的环境字串块  
GetACP 判断目前正在生效的ANSI代码页  
GetAsyncKeyState 判断函数调用时指定虚拟键的状态  
GetCaretBlinkTime 判断插入符光标的闪烁频率  
GetCaretPos 判断插入符的当前位置  
GetClipCursor 取得一个矩形，用于描述目前为鼠标指针规定的剪切区域  
GetCommandLine 获得指向当前命令行缓冲区的一个指针  
GetComputerName 取得这台计算机的名称  
GetCPInfo 取得与指定代码页有关的信息  
GetCurrencyFormat 针对指定的“地方”设置，根据货币格式格式化一个数字  
GetCursor 获取目前选择的鼠标指针的句柄  
GetCursorPos 获取鼠标指针的当前位置  
GetDateFormat 针对指定的“当地”格式，对一个系统日期进行格式化  
GetDoubleClickTime 判断连续两次鼠标单击之间会被处理成双击事件的间隔时间  
GetEnvironmentStrings 为包含了当前环境字串设置的一个内存块分配和返回一个句柄  
GetEnvironmentVariable 取得一个环境变量的值  
GetInputState 判断是否存在任何待决（等待处理）的鼠标或键盘事件  
GetKBCodePage 由GetOEMCP取代，两者功能完全相同  
GetKeyboardLayout 取得一个句柄，描述指定应用程序的键盘布局  
GetKeyboardLayoutList 获得系统适用的所有键盘布局的一个列表  
GetKeyboardLayoutName 取得当前活动键盘布局的名称  
GetKeyboardState 取得键盘上每个虚拟键当前的状态  
GetKeyboardType 了解与正在使用的键盘有关的信息  
GetKeyNameText 在给出扫描码的前提下，判断键名  
GetKeyState 针对已处理过的按键，在最近一次输入信息时，判断指定虚拟键的状态  
GetLastError 针对之前调用的api函数，用这个函数取得扩展错误信息  
GetLocaleInfo 取得与指定“地方”有关的信息  
GetLocalTime 取得本地日期和时间  
GetNumberFormat 针对指定的“地方”，按特定的格式格式化一个数字  
GetOEMCP 判断在OEM和ANSI字符集间转换的windows代码页  
GetQueueStatus 判断应用程序消息队列中待决（等待处理）的消息类型  
GetSysColor 判断指定windows显示对象的颜色  
GetSystemDefaultLangID 取得系统的默认语言ID  
GetSystemDefaultLCID 取得当前的默认系统“地方”  
GetSystemInfo 取得与底层硬件平台有关的信息  
GetSystemMetrics 返回与windows环境有关的信息  
GetSystemPowerStatus 获得与当前系统电源状态有关的信息  
GetSystemTime 取得当前系统时间，这个时间采用的是“协同世界时间”（即UTC，也叫做GMT）格式  
GetSystemTimeAdjustment 使内部系统时钟与一个外部的时钟信号源同步  
GetThreadLocale 取得当前线程的地方ID  
GetTickCount 用于获取自windows启动以来经历的时间长度（毫秒）  
GetTimeFormat 针对当前指定的“地方”，按特定的格式格式化一个系统时间  
GetTimeZoneInformation 取得与系统时区设置有关的信息  
GetUserDefaultLangID 为当前用户取得默认语言ID  
GetUserDefaultLCID 取得当前用户的默认“地方”设置  
GetUserName 取得当前用户的名字  
GetVersion 判断当前运行的Windows和DOS版本  
GetVersionEx 取得与平台和操作系统有关的版本信息  
HideCaret 在指定的窗口隐藏插入符（光标）  
IsValidCodePage 判断一个代码页是否有效  
IsValidLocale 判断地方标识符是否有效  
keybd\_event 这个函数模拟了键盘行动  
LoadKeyboardLayout 载入一个键盘布局  
MapVirtualKey 根据指定的映射类型，执行不同的扫描码和字符转换  
MapVirtualKeyEx 根据指定的映射类型，执行不同的扫描码和字符转换  
MessageBeep 播放一个系统声音。系统声音的分配方案是在控制面板里决定的  
mouse\_event 模拟一次鼠标事件  
OemKeyScan 判断OEM字符集中的一个ASCII字符的扫描码和Shift键状态  
OemToChar 将OEM字符集的一个字串转换到ANSI字符集  
SetCaretBlinkTime 指定插入符（光标）的闪烁频率  
SetCaretPos 指定插入符的位置  
SetComputerName 设置新的计算机名  
SetCursor 将指定的鼠标指针设为当前指针  
SetCursorPos 设置指针的位置  
SetDoubleClickTime 设置连续两次鼠标单击之间能使系统认为是双击事件的间隔时间  
SetEnvironmentVariable 将一个环境变量设为指定的值  
SetKeyboardState 设置每个虚拟键当前在键盘上的状态  
SetLocaleInfo 改变用户“地方”设置信息  
SetLocalTime 设置当前地方时间  
SetSysColors 设置指定窗口显示对象的颜色  
SetSystemCursor 改变任何一个标准系统指针  
SetSystemTime 设置当前系统时间  
SetSystemTimeAdjustment 定时添加一个校准值使内部系统时钟与一个外部的时钟信号源同步  
SetThreadLocale 为当前线程设置地方  
SetTimeZoneInformation 设置系统时区信息  
ShowCaret 在指定的窗口里显示插入符（光标）  
ShowCursor 控制鼠标指针的可视性  
SwapMouseButton 决定是否互换鼠标左右键的功能  
SystemParametersInfo 获取和设置数量众多的windows系统参数  
SystemTimeToTzSpecificLocalTime 将系统时间转换成地方时间  
ToAscii 根据当前的扫描码和键盘信息，将一个虚拟键转换成ASCII字符  
ToUnicode 根据当前的扫描码和键盘信息，将一个虚拟键转换成Unicode字符  
UnloadKeyboardLayout 卸载指定的键盘布局  
VkKeyScan 针对Windows字符集中一个ASCII字符，判断虚拟键码和Shift键的状态  
  
11. API之进程和线程函数  
  
CancelWaitableTimer 这个函数用于取消一个可以等待下去的计时器操作  
CallNamedPipe 这个函数由一个希望通过管道通信的一个客户进程调用  
ConnectNamedPipe 指示一台服务器等待下去，直至客户机同一个命名管道连接  
CreateEvent 创建一个事件对象  
CreateMailslot 创建一个邮路。返回的句柄由邮路服务器使用（收件人）  
CreateMutex 创建一个互斥体（MUTEX）  
CreateNamedPipe 创建一个命名管道。返回的句柄由管道的服务器端使用  
CreatePipe 创建一个匿名管道  
CreateProcess 创建一个新进程（比如执行一个程序）  
CreateSemaphore 创建一个新的信号机  
CreateWaitableTimer 创建一个可等待的计时器对象  
DisconnectNamedPipe 断开一个客户与一个命名管道的连接  
DuplicateHandle 在指出一个现有系统对象当前句柄的情况下，为那个对象创建一个新句柄  
ExitProcess 中止一个进程  
FindCloseChangeNotification 关闭一个改动通知对象  
FindExecutable 查找与一个指定文件关联在一起的程序的文件名  
FindFirstChangeNotification 创建一个文件通知对象。该对象用于监视文件系统发生的变化  
FindNextChangeNotification 重设一个文件改变通知对象，令其继续监视下一次变化  
FreeLibrary 释放指定的动态链接库  
GetCurrentProcess 获取当前进程的一个伪句柄  
GetCurrentProcessId 获取当前进程一个唯一的标识符  
GetCurrentThread 获取当前线程的一个伪句柄  
GetCurrentThreadId 获取当前线程一个唯一的线程标识符  
GetExitCodeProces 获取一个已中断进程的退出代码  
GetExitCodeThread 获取一个已中止线程的退出代码  
GetHandleInformation 获取与一个系统对象句柄有关的信息  
GetMailslotInfo 获取与一个邮路有关的信息  
GetModuleFileName 获取一个已装载模板的完整路径名称  
GetModuleHandle 获取一个应用程序或动态链接库的模块句柄  
GetPriorityClass 获取特定进程的优先级别  
GetProcessShutdownParameters 调查系统关闭时一个指定的进程相对于其它进程的关闭早迟情况  
GetProcessTimes 获取与一个进程的经过时间有关的信息  
GetProcessWorkingSetSize 了解一个应用程序在运行过程中实际向它交付了多大容量的内存  
GetSartupInfo 获取一个进程的启动信息  
GetThreadPriority 获取特定线程的优先级别  
GetTheardTimes 获取与一个线程的经过时间有关的信息  
GetWindowThreadProcessId 获取与指定窗口关联在一起的一个进程和线程标识符  
LoadLibrary 载入指定的动态链接库，并将它映射到当前进程使用的地址空间  
LoadLibraryEx 装载指定的动态链接库，并为当前进程把它映射到地址空间  
LoadModule 载入一个Windows应用程序，并在指定的环境中运行  
MsgWaitForMultipleObjects 等侯单个对象或一系列对象发出信号。如返回条件已经满足，则立即返回  
SetPriorityClass 设置一个进程的优先级别  
SetProcessShutdownParameters 在系统关闭期间，为指定进程设置他相对于其它程序的关闭顺序  
SetProcessWorkingSetSize 设置操作系统实际划分给进程使用的内存容量  
SetThreadPriority 设定线程的优先级别  
ShellExecute 查找与指定文件关联在一起的程序的文件名  
TerminateProcess 结束一个进程  
WinExec 运行指定的程序  
  
12. API之控件与消息函数  
  
AdjustWindowRect 给定一种窗口样式，计算获得目标客户区矩形所需的窗口大小  
AnyPopup 判断屏幕上是否存在任何弹出式窗口  
ArrangeIconicWindows 排列一个父窗口的最小化子窗口  
AttachThreadInput 连接线程输入函数  
BeginDeferWindowPos 启动构建一系列新窗口位置的过程  
BringWindowToTop 将指定的窗口带至窗口列表顶部  
CascadeWindows 以层叠方式排列窗口  
ChildWindowFromPoint 返回父窗口中包含了指定点的第一个子窗口的句柄  
ClientToScreen 判断窗口内以客户区坐标表示的一个点的屏幕坐标  
CloseWindow 最小化指定的窗口  
CopyRect 矩形内容复制  
DeferWindowPos 该函数为特定的窗口指定一个新窗口位置  
DestroyWindow 清除指定的窗口以及它的所有子窗口  
DrawAnimatedRects 描绘一系列动态矩形  
EnableWindow 指定的窗口里允许或禁止所有鼠标及键盘输入  
EndDeferWindowPos 同时更新DeferWindowPos调用时指定的所有窗口的位置及状态  
EnumChildWindows 为指定的父窗口枚举子窗口  
EnumThreadWindows 枚举与指定任务相关的窗口  
EnumWindows 枚举窗口列表中的所有父窗口  
EqualRect 判断两个矩形结构是否相同  
FindWindow 寻找窗口列表中第一个符合指定条件的顶级窗口  
FindWindowEx 在窗口列表中寻找与指定条件相符的第一个子窗口  
FlashWindow 闪烁显示指定窗口  
GetActiveWindow 获得活动窗口的句柄  
GetCapture 获得一个窗口的句柄，这个窗口位于当前输入线程，且拥有鼠标捕获（鼠标活动由它接收）  
GetClassInfo 取得WNDCLASS结构（或WNDCLASSEX结构）的一个副本，结构中包含了与指定类有关的信息  
GetClassLong 取得窗口类的一个Long变量条目  
GetClassName 为指定的窗口取得类名  
GetClassWord 为窗口类取得一个整数变量  
GetClientRect 返回指定窗口客户区矩形的大小  
GetDesktopWindow 获得代表整个屏幕的一个窗口（桌面窗口）句柄  
GetFocus 获得拥有输入焦点的窗口的句柄  
GetForegroundWindow 获得前台窗口的句柄  
GetLastActivePopup 获得在一个给定父窗口中最近激活过的弹出式窗口的句柄  
GetParent 判断指定窗口的父窗口  
GetTopWindow 搜索内部窗口列表，寻找隶属于指定窗口的头一个窗口的句柄  
GetUpdateRect 获得一个矩形，它描叙了指定窗口中需要更新的那一部分  
GetWindow 获得一个窗口的句柄，该窗口与某源窗口有特定的关系  
GetWindowContextHelpId 取得与窗口关联在一起的帮助场景ID  
GetWindowLong 从指定窗口的结构中取得信息  
GetWindowPlacement 获得指定窗口的状态及位置信息  
GetWindowRect 获得整个窗口的范围矩形，窗口的边框、标题栏、滚动条及菜单等都在这个矩形内  
GetWindowText 取得一个窗体的标题（caption）文字，或者一个控件的内容  
GetWindowTextLength 调查窗口标题文字或控件内容的长短  
GetWindowWord 获得指定窗口结构的信息  
InflateRect 增大或减小一个矩形的大小  
IntersectRect 这个函数在lpDestRect里载入一个矩形，它是lpSrc1Rect与lpSrc2Rect两个矩形的交集  
InvalidateRect 屏蔽一个窗口客户区的全部或部分区域  
IsChild 判断一个窗口是否为另一窗口的子或隶属窗口  
IsIconic 判断窗口是否已最小化  
IsRectEmpty 判断一个矩形是否为空  
IsWindow 判断一个窗口句柄是否有效  
IsWindowEnabled 判断窗口是否处于活动状态  
IsWindowUnicode 判断一个窗口是否为Unicode窗口。这意味着窗口为所有基于文本的消息都接收Unicode文字  
IsWindowVisible 判断窗口是否可见  
IsZoomed 判断窗口是否最大化  
LockWindowUpdate 锁定指定窗口，禁止它更新  
MapWindowPoints 将一个窗口客户区坐标的点转换到另一窗口的客户区坐标系统  
MoveWindow 改变指定窗口的位置和大小  
OffsetRect 通过应用一个指定的偏移，从而让矩形移动起来  
OpenIcon 恢复一个最小化的程序，并将其激活  
PtInRect 判断指定的点是否位于矩形内部  
RedrawWindow 重画全部或部分窗口  
ReleaseCapture 为当前的应用程序释放鼠标捕获  
ScreenToClient 判断屏幕上一个指定点的客户区坐标  
ScrollWindow 滚动窗口客户区的全部或一部分  
ScrollWindowEx 根据附加的选项，滚动窗口客户区的全部或部分  
SetActiveWindow 激活指定的窗口  
SetCapture 将鼠标捕获设置到指定的窗口  
SetClassLong 为窗口类设置一个Long变量条目  
SetClassWord 为窗口类设置一个条目  
SetFocusAPI 将输入焦点设到指定的窗口。如有必要，会激活窗口  
SetForegroundWindow 将窗口设为系统的前台窗口  
SetParent 指定一个窗口的新父  
SetRect 设置指定矩形的内容  
SetRectEmpty 将矩形设为一个空矩形  
SetWindowContextHelpId 为指定的窗口设置帮助场景（上下文）ID  
SetWindowLong 在窗口结构中为指定的窗口设置信息  
SetWindowPlacement 设置窗口状态和位置信息  
SetWindowPos 为窗口指定一个新位置和状态  
SetWindowText 设置窗口的标题文字或控件的内容  
SetWindowWord 在窗口结构中为指定的窗口设置信息  
ShowOwnedPopups 显示或隐藏由指定窗口所有的全部弹出式窗口  
ShowWindow 控制窗口的可见性  
ShowWindowAsync 与ShowWindow相似  
SubtractRect 装载矩形lprcDst，它是在矩形lprcSrc1中减去lprcSrc2得到的结果  
TileWindows 以平铺顺序排列窗口  
UnionRect 装载一个lpDestRect目标矩形，它是lpSrc1Rect和lpSrc2Rect联合起来的结果  
UpdateWindow 强制立即更新窗口  
ValidateRect 校验窗口的全部或部分客户区  
WindowFromPoint 返回包含了指定点的窗口的句柄。忽略屏蔽、隐藏以及透明窗口

消息，就是指Windows发出的一个通知，告诉应用程序某个事情发生了。例如，单击鼠标、改变窗口尺寸、按下键盘上的一个键都会使Windows 发送一个消息给应用程序。消息本身是作为一个记录传递给应用程序的，这个记录中包含了消息的类型以及其他信息。例如，对于单击鼠标所产生的消息来说，这个 记录中包含了单击鼠标时的坐标。这个记录类型叫做TMsg，

它在Windows单元中是这样声明的：  
type  
TMsg = packed record  
hwnd: HWND; / /窗口句柄  
message: UINT; / /消息常量标识符  
wParam: WPARAM ; // 32位消息的特定附加信息  
lParam: LPARAM ; // 32位消息的特定附加信息  
time: DWORD; / /消息创建时的时间  
pt: TPoint; / /消息创建时的鼠标位置  
end;

posted @ [2008-08-28 10:30](http://www.cppblog.com/road420/archive/2008/08/28/60231.html) 深邃者 阅读(426) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/08/28/60231.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=60231) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=60231)

[**Window API函数大全(3)**](http://www.cppblog.com/road420/archive/2008/08/28/60230.html)

再补一点消息详解  
消息中有什么？  
是否觉得一个消息记录中的信息像希腊语一样？如果是这样，那么看一看下面的解释：  
hwnd 32位的窗口句柄。窗口可以是任何类型的屏幕对象，因为Win32能够维护大多数可视对象的句柄(窗口、对话框、按钮、编辑框等)。  
message 用于区别其他消息的常量值，这些常量可以是Windows单元中预定义的常量，也可以是自定义的常量。  
wParam 通常是一个与消息有关的常量值，也可能是窗口或控件的句柄。  
lParam 通常是一个指向内存中数据的指针。由于W P a r a m、l P a r a m和P o i n t e r都是3 2位的，  
因此，它们之间可以相互转换。

WM\_NULL = $0000;  
WM\_CREATE = $0001;  
应用程序创建一个窗口  
WM\_DESTROY = $0002;  
一个窗口被销毁  
WM\_MOVE = $0003;  
移动一个窗口  
WM\_SIZE = $0005;  
改变一个窗口的大小  
WM\_ACTIVATE = $0006;  
一个窗口被激活或失去激活状态；  
WM\_SETFOCUS = $0007;  
获得焦点后  
WM\_KILLFOCUS = $0008;  
失去焦点  
WM\_ENABLE = $000A;  
改变enable状态  
WM\_SETREDRAW = $000B;  
设置窗口是否能重画  
WM\_SETTEXT = $000C;  
应用程序发送此消息来设置一个窗口的文本  
WM\_GETTEXT = $000D;  
应用程序发送此消息来复制对应窗口的文本到缓冲区  
WM\_GETTEXTLENGTH = $000E;  
得到与一个窗口有关的文本的长度（不包含空字符）  
WM\_PAINT = $000F;  
要求一个窗口重画自己  
WM\_CLOSE = $0010;  
当一个窗口或应用程序要关闭时发送一个信号  
WM\_QUERYENDSESSION = $0011;  
当用户选择结束对话框或程序自己调用ExitWindows函数  
WM\_QUIT = $0012;  
用来结束程序运行或当程序调用postquitmessage函数  
WM\_QUERYOPEN = $0013;  
当用户窗口恢复以前的大小位置时，把此消息发送给某个图标  
WM\_ERASEBKGND = $0014;  
当窗口背景必须被擦除时（例在窗口改变大小时）  
WM\_SYSCOLORCHANGE = $0015;  
当系统颜色改变时，发送此消息给所有顶级窗口  
WM\_ENDSESSION = $0016;  
当系统进程发出WM\_QUERYENDSESSION消息后，此消息发送给应用程序，  
通知它对话是否结束  
WM\_SYSTEMERROR = $0017;  
WM\_SHOWWINDOW = $0018;  
当隐藏或显示窗口是发送此消息给这个窗口  
WM\_ACTIVATEAPP = $001C;  
发此消息给应用程序哪个窗口是激活的，哪个是非激活的；  
WM\_FONTCHANGE = $001D;  
当系统的字体资源库变化时发送此消息给所有顶级窗口  
WM\_TIMECHANGE = $001E;  
当系统的时间变化时发送此消息给所有顶级窗口  
WM\_CANCELMODE = $001F;  
发送此消息来取消某种正在进行的摸态（操作）  
WM\_SETCURSOR = $0020;  
如果鼠标引起光标在某个窗口中移动且鼠标输入没有被捕获时，就发消息给某个窗口  
WM\_MOUSEACTIVATE = $0021;  
当光标在某个非激活的窗口中而用户正按着鼠标的某个键发送此消息给当前窗口  
WM\_CHILDACTIVATE = $0022;  
发送此消息给MDI子窗口当用户点击此窗口的标题栏，或当窗口被激活，移动，改变大小  
WM\_QUEUESYNC = $0023;  
此消息由基于计算机的训练程序发送，通过WH\_JOURNALPALYBACK的hook程序  
分离出用户输入消息  
WM\_GETMINMAXINFO = $0024;  
此消息发送给窗口当它将要改变大小或位置；  
WM\_PAINTICON = $0026;  
发送给最小化窗口当它图标将要被重画  
WM\_ICONERASEBKGND = $0027;  
此消息发送给某个最小化窗口，仅当它在画图标前它的背景必须被重画  
WM\_NEXTDLGCTL = $0028;  
发送此消息给一个对话框程序去更改焦点位置  
WM\_SPOOLERSTATUS = $002A;  
每当打印管理列队增加或减少一条作业时发出此消息  
WM\_DRAWITEM = $002B;  
当button，combobox，listbox，menu的可视外观改变时发送  
此消息给这些空件的所有者  
WM\_MEASUREITEM = $002C;  
当button, combo box, list box, list view control, or menu item 被创建时  
发送此消息给控件的所有者  
WM\_DELETEITEM = $002D;  
当the list box 或 combo box 被销毁 或 当 某些项被删除通过LB\_DELETESTRING, LB\_RESETCONTENT, CB\_DELETESTRING, or CB\_RESETCONTENT 消息  
WM\_VKEYTOITEM = $002E;  
此消息有一个LBS\_WANTKEYBOARDINPUT风格的发出给它的所有者来响应WM\_KEYDOWN消息  
WM\_CHARTOITEM = $002F;  
此消息由一个LBS\_WANTKEYBOARDINPUT风格的列表框发送给他的所有者来响应WM\_CHAR消息  
WM\_SETFONT = $0030;  
当绘制文本时程序发送此消息得到控件要用的颜色  
WM\_GETFONT = $0031;  
应用程序发送此消息得到当前控件绘制文本的字体  
WM\_SETHOTKEY = $0032;  
应用程序发送此消息让一个窗口与一个热键相关连  
WM\_GETHOTKEY = $0033;  
应用程序发送此消息来判断热键与某个窗口是否有关联  
WM\_QUERYDRAGICON = $0037;  
此消息发送给最小化窗口，当此窗口将要被拖放而它的类中没有定义图标，应用程序能返回一个图标或光标的句柄，当用户拖放图标时系统显示这个图标或光标  
WM\_COMPAREITEM = $0039;  
发送此消息来判定combobox或listbox新增加的项的相对位置  
WM\_GETOBJECT = $003D;  
WM\_COMPACTING = $0041;  
显示内存已经很少了  
WM\_WINDOWPOSCHANGING = $0046;  
发送此消息给那个窗口的大小和位置将要被改变时，来调用setwindowpos函数或其它窗口管理函数  
WM\_WINDOWPOSCHANGED = $0047;  
发送此消息给那个窗口的大小和位置已经被改变时，来调用setwindowpos函数或其它窗口管理函数  
WM\_POWER = $0048;（适用于16位的windows）  
当系统将要进入暂停状态时发送此消息  
WM\_COPYDATA = $004A;  
当一个应用程序传递数据给另一个应用程序时发送此消息  
WM\_CANCELJOURNAL = $004B;  
当某个用户取消程序日志激活状态，提交此消息给程序  
WM\_NOTIFY = $004E;  
当某个控件的某个事件已经发生或这个控件需要得到一些信息时，发送此消息给它的父窗口  
WM\_INPUTLANGCHANGEREQUEST = $0050;  
当用户选择某种输入语言，或输入语言的热键改变  
WM\_INPUTLANGCHANGE = $0051;  
当平台现场已经被改变后发送此消息给受影响的最顶级窗口  
WM\_TCARD = $0052;  
当程序已经初始化windows帮助例程时发送此消息给应用程序  
WM\_HELP = $0053;  
此消息显示用户按下了F1，如果某个菜单是激活的，就发送此消息个此窗口关联的菜单，否则就  
发送给有焦点的窗口，如果当前都没有焦点，就把此消息发送给当前激活的窗口  
WM\_USERCHANGED = $0054;  
当用户已经登入或退出后发送此消息给所有的窗口，当用户登入或退出时系统更新用户的具体  
设置信息，在用户更新设置时系统马上发送此消息；  
WM\_NOTIFYFORMAT = $0055;  
公用控件，自定义控件和他们的父窗口通过此消息来判断控件是使用ANSI还是UNICODE结构  
在WM\_NOTIFY消息，使用此控件能使某个控件与它的父控件之间进行相互通信  
WM\_CONTEXTMENU = $007B;  
当用户某个窗口中点击了一下右键就发送此消息给这个窗口  
WM\_STYLECHANGING = $007C;  
当调用SETWINDOWLONG函数将要改变一个或多个 窗口的风格时发送此消息给那个窗口  
WM\_STYLECHANGED = $007D;  
当调用SETWINDOWLONG函数一个或多个 窗口的风格后发送此消息给那个窗口  
WM\_DISPLAYCHANGE = $007E;  
当显示器的分辨率改变后发送此消息给所有的窗口  
WM\_GETICON = $007F;  
此消息发送给某个窗口来返回与某个窗口有关连的大图标或小图标的句柄；  
WM\_SETICON = $0080;  
程序发送此消息让一个新的大图标或小图标与某个窗口关联；  
WM\_NCCREATE = $0081;  
当某个窗口第一次被创建时，此消息在WM\_CREATE消息发送前发送；  
WM\_NCDESTROY = $0082;  
此消息通知某个窗口，非客户区正在销毁  
WM\_NCCALCSIZE = $0083;  
当某个窗口的客户区域必须被核算时发送此消息  
WM\_NCHITTEST = $0084;//移动鼠标，按住或释放鼠标时发生  
WM\_NCPAINT = $0085;  
程序发送此消息给某个窗口当它（窗口）的框架必须被绘制时；  
WM\_NCACTIVATE = $0086;  
此消息发送给某个窗口 仅当它的非客户区需要被改变来显示是激活还是非激活状态；  
WM\_GETDLGCODE = $0087;  
发送此消息给某个与对话框程序关联的控件，widdows控制方位键和TAB键使输入进入此控件  
通过响应WM\_GETDLGCODE消息，应用程序可以把他当成一个特殊的输入控件并能处理它  
WM\_NCMOUSEMOVE = $00A0;  
当光标在一个窗口的非客户区内移动时发送此消息给这个窗口 //非客户区为：窗体的标题栏及窗  
的边框体  
WM\_NCLBUTTONDOWN = $00A1;  
当光标在一个窗口的非客户区同时按下鼠标左键时提交此消息  
WM\_NCLBUTTONUP = $00A2;  
当用户释放鼠标左键同时光标某个窗口在非客户区十发送此消息；  
WM\_NCLBUTTONDBLCLK = $00A3;  
当用户双击鼠标左键同时光标某个窗口在非客户区十发送此消息  
WM\_NCRBUTTONDOWN = $00A4;  
当用户按下鼠标右键同时光标又在窗口的非客户区时发送此消息  
WM\_NCRBUTTONUP = $00A5;  
当用户释放鼠标右键同时光标又在窗口的非客户区时发送此消息  
WM\_NCRBUTTONDBLCLK = $00A6;  
当用户双击鼠标右键同时光标某个窗口在非客户区十发送此消息  
WM\_NCMBUTTONDOWN = $00A7;  
当用户按下鼠标中键同时光标又在窗口的非客户区时发送此消息  
WM\_NCMBUTTONUP = $00A8;  
当用户释放鼠标中键同时光标又在窗口的非客户区时发送此消息  
WM\_NCMBUTTONDBLCLK = $00A9;  
当用户双击鼠标中键同时光标又在窗口的非客户区时发送此消息  
WM\_KEYFIRST = $0100;  
WM\_KEYDOWN = $0100;  
//按下一个键  
WM\_KEYUP = $0101;  
//释放一个键  
WM\_CHAR = $0102;  
//按下某键，并已发出WM\_KEYDOWN， WM\_KEYUP消息  
WM\_DEADCHAR = $0103;  
当用translatemessage函数翻译WM\_KEYUP消息时发送此消息给拥有焦点的窗口  
WM\_SYSKEYDOWN = $0104;  
当用户按住ALT键同时按下其它键时提交此消息给拥有焦点的窗口；  
WM\_SYSKEYUP = $0105;  
当用户释放一个键同时ALT 键还按着时提交此消息给拥有焦点的窗口  
WM\_SYSCHAR = $0106;  
当WM\_SYSKEYDOWN消息被TRANSLATEMESSAGE函数翻译后提交此消息给拥有焦点的窗口  
WM\_SYSDEADCHAR = $0107;  
当WM\_SYSKEYDOWN消息被TRANSLATEMESSAGE函数翻译后发送此消息给拥有焦点的窗口  
WM\_KEYLAST = $0108;  
WM\_INITDIALOG = $0110;  
在一个对话框程序被显示前发送此消息给它，通常用此消息初始化控件和执行其它任务  
WM\_COMMAND = $0111;  
当用户选择一条菜单命令项或当某个控件发送一条消息给它的父窗口，一个快捷键被翻译  
WM\_SYSCOMMAND = $0112;  
当用户选择窗口菜单的一条命令或当用户选择最大化或最小化时那个窗口会收到此消息  
WM\_TIMER = $0113; //发生了定时器事件  
WM\_HSCROLL = $0114;  
当一个窗口标准水平滚动条产生一个滚动事件时发送此消息给那个窗口，也发送给拥有它的控件  
WM\_VSCROLL = $0115;  
当一个窗口标准垂直滚动条产生一个滚动事件时发送此消息给那个窗口也，发送给拥有它的控件 WM\_INITMENU = $0116;  
当一个菜单将要被激活时发送此消息，它发生在用户菜单条中的某项或按下某个菜单键，它允许程序在显示前更改菜单  
WM\_INITMENUPOPUP = $0117;  
当一个下拉菜单或子菜单将要被激活时发送此消息，它允许程序在它显示前更改菜单，而不要改变全部  
WM\_MENUSELECT = $011F;  
当用户选择一条菜单项时发送此消息给菜单的所有者（一般是窗口）  
WM\_MENUCHAR = $0120;  
当菜单已被激活用户按下了某个键（不同于加速键），发送此消息给菜单的所有者；  
WM\_ENTERIDLE = $0121;  
当一个模态对话框或菜单进入空载状态时发送此消息给它的所有者，一个模态对话框或菜单进入空载状态就是在处理完一条或几条先前的消息后没有消息它的列队中等待  
WM\_MENURBUTTONUP = $0122;  
WM\_MENUDRAG = $0123;  
WM\_MENUGETOBJECT = $0124;  
WM\_UNINITMENUPOPUP = $0125;  
WM\_MENUCOMMAND = $0126;  
WM\_CHANGEUISTATE = $0127;  
WM\_UPDATEUISTATE = $0128;  
WM\_QUERYUISTATE = $0129;  
WM\_CTLCOLORMSGBOX = $0132;  
在windows绘制消息框前发送此消息给消息框的所有者窗口，通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置消息框的文本和背景颜色  
WM\_CTLCOLOREDIT = $0133;  
当一个编辑型控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置编辑框的文本和背景颜色  
WM\_CTLCOLORLISTBOX = $0134;  
当一个列表框控件将要被绘制前发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置列表框的文本和背景颜色  
WM\_CTLCOLORBTN = $0135;  
当一个按钮控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置按纽的文本和背景颜色  
WM\_CTLCOLORDLG = $0136;  
当一个对话框控件将要被绘制前发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置对话框的文本背景颜色  
WM\_CTLCOLORSCROLLBAR= $0137;  
当一个滚动条控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置滚动条的背景颜色  
WM\_CTLCOLORSTATIC = $0138;  
当一个静态控件将要被绘制时发送此消息给它的父窗口；通过响应这条消息，所有者窗口可以通过使用给定的相关显示设备的句柄来设置静态控件的文本和背景颜色  
WM\_MOUSEFIRST = $0200;  
WM\_MOUSEMOVE = $0200;  
// 移动鼠标  
WM\_LBUTTONDOWN = $0201;  
//按下鼠标左键  
WM\_LBUTTONUP = $0202;  
//释放鼠标左键  
WM\_LBUTTONDBLCLK = $0203;  
//双击鼠标左键  
WM\_RBUTTONDOWN = $0204;  
//按下鼠标右键  
WM\_RBUTTONUP = $0205;  
//释放鼠标右键  
WM\_RBUTTONDBLCLK = $0206;  
//双击鼠标右键  
WM\_MBUTTONDOWN = $0207;  
//按下鼠标中键  
WM\_MBUTTONUP = $0208;  
//释放鼠标中键  
WM\_MBUTTONDBLCLK = $0209;  
//双击鼠标中键  
WM\_MOUSEWHEEL = $020A;  
当鼠标轮子转动时发送此消息个当前有焦点的控件  
WM\_MOUSELAST = $020A;  
WM\_PARENTNOTIFY = $0210;  
当MDI子窗口被创建或被销毁，或用户按了一下鼠标键而光标在子窗口上时发送此消息给它的父窗口  
WM\_ENTERMENULOOP = $0211;  
发送此消息通知应用程序的主窗口that已经进入了菜单循环模式  
WM\_EXITMENULOOP = $0212;  
发送此消息通知应用程序的主窗口that已退出了菜单循环模式  
WM\_NEXTMENU = $0213;  
WM\_SIZING = 532;  
当用户正在调整窗口大小时发送此消息给窗口；通过此消息应用程序可以监视窗口大小和位置也可以修改他们  
WM\_CAPTURECHANGED = 533;  
发送此消息 给窗口当它失去捕获的鼠标时；  
WM\_MOVING = 534;  
当用户在移动窗口时发送此消息，通过此消息应用程序可以监视窗口大小和位置也可以修改他们；  
WM\_POWERBROADCAST = 536;  
此消息发送给应用程序来通知它有关电源管理事件；  
WM\_DEVICECHANGE = 537;  
当设备的硬件配置改变时发送此消息给应用程序或设备驱动程序  
WM\_IME\_STARTCOMPOSITION = $010D;  
WM\_IME\_ENDCOMPOSITION = $010E;  
WM\_IME\_COMPOSITION = $010F;  
WM\_IME\_KEYLAST = $010F;  
WM\_IME\_SETCONTEXT = $0281;  
WM\_IME\_NOTIFY = $0282;  
WM\_IME\_CONTROL = $0283;  
WM\_IME\_COMPOSITIONFULL = $0284;  
WM\_IME\_SELECT = $0285;  
WM\_IME\_CHAR = $0286;  
WM\_IME\_REQUEST = $0288;  
WM\_IME\_KEYDOWN = $0290;  
WM\_IME\_KEYUP = $0291;  
WM\_MDICREATE = $0220;  
应用程序发送此消息给多文档的客户窗口来创建一个MDI 子窗口  
WM\_MDIDESTROY = $0221;  
应用程序发送此消息给多文档的客户窗口来关闭一个MDI 子窗口  
WM\_MDIACTIVATE = $0222;  
应用程序发送此消息给多文档的客户窗口通知客户窗口激活另一个MDI子窗口，当客户窗口收到此消息后，它发出WM\_MDIACTIVE消息给MDI子窗口（未激活）激活它；  
WM\_MDIRESTORE = $0223;  
程序 发送此消息给MDI客户窗口让子窗口从最大最小化恢复到原来大小  
WM\_MDINEXT = $0224;  
程序 发送此消息给MDI客户窗口激活下一个或前一个窗口  
WM\_MDIMAXIMIZE = $0225;  
程序发送此消息给MDI客户窗口来最大化一个MDI子窗口；  
WM\_MDITILE = $0226;  
程序 发送此消息给MDI客户窗口以平铺方式重新排列所有MDI子窗口  
WM\_MDICASCADE = $0227;  
程序 发送此消息给MDI客户窗口以层叠方式重新排列所有MDI子窗口  
WM\_MDIICONARRANGE = $0228;  
程序 发送此消息给MDI客户窗口重新排列所有最小化的MDI子窗口  
WM\_MDIGETACTIVE = $0229;  
程序 发送此消息给MDI客户窗口来找到激活的子窗口的句柄  
WM\_MDISETMENU = $0230;  
程序 发送此消息给MDI客户窗口用MDI菜单代替子窗口的菜单  
WM\_ENTERSIZEMOVE = $0231;  
WM\_EXITSIZEMOVE = $0232;  
WM\_DROPFILES = $0233;  
WM\_MDIREFRESHMENU = $0234;  
WM\_MOUSEHOVER = $02A1;  
WM\_MOUSELEAVE = $02A3;  
WM\_CUT = $0300;  
程序发送此消息给一个编辑框或combobox来删除当前选择的文本  
WM\_COPY = $0301;  
程序发送此消息给一个编辑框或combobox来复制当前选择的文本到剪贴板  
WM\_PASTE = $0302;  
程序发送此消息给editcontrol或combobox从剪贴板中得到数据  
WM\_CLEAR = $0303;  
程序发送此消息给editcontrol或combobox清除当前选择的内容；  
WM\_UNDO = $0304;  
程序发送此消息给editcontrol或combobox撤消最后一次操作  
WM\_RENDERFORMAT = $0305；

WM\_RENDERALLFORMATS = $0306;  
WM\_DESTROYCLIPBOARD = $0307;  
当调用ENPTYCLIPBOARD函数时 发送此消息给剪贴板的所有者  
WM\_DRAWCLIPBOARD = $0308;  
当剪贴板的内容变化时发送此消息给剪贴板观察链的第一个窗口；它允许用剪贴板观察窗口来  
显示剪贴板的新内容；  
WM\_PAINTCLIPBOARD = $0309;  
当剪贴板包含CF\_OWNERDIPLAY格式的数据并且剪贴板观察窗口的客户区需要重画；  
WM\_VSCROLLCLIPBOARD = $030A;  
WM\_SIZECLIPBOARD = $030B;  
当剪贴板包含CF\_OWNERDIPLAY格式的数据并且剪贴板观察窗口的客户区域的大小已经改变是此消息通过剪贴板观察窗口发送给剪贴板的所有者；  
WM\_ASKCBFORMATNAME = $030C;  
通过剪贴板观察窗口发送此消息给剪贴板的所有者来请求一个CF\_OWNERDISPLAY格式的剪贴板的名字  
WM\_CHANGECBCHAIN = $030D;  
当一个窗口从剪贴板观察链中移去时发送此消息给剪贴板观察链的第一个窗口；  
WM\_HSCROLLCLIPBOARD = $030E;  
此消息通过一个剪贴板观察窗口发送给剪贴板的所有者 ；它发生在当剪贴板包含CFOWNERDISPALY格式的数据并且有个事件在剪贴板观察窗的水平滚动条上；所有者应滚动剪贴板图象并更新滚动条的值；  
WM\_QUERYNEWPALETTE = $030F;  
此消息发送给将要收到焦点的窗口，此消息能使窗口在收到焦点时同时有机会实现他的逻辑调色板  
WM\_PALETTEISCHANGING= $0310;  
当一个应用程序正要实现它的逻辑调色板时发此消息通知所有的应用程序  
WM\_PALETTECHANGED = $0311;  
此消息在一个拥有焦点的窗口实现它的逻辑调色板后发送此消息给所有顶级并重叠的窗口，以此来改变系统调色板  
WM\_HOTKEY = $0312;  
当用户按下由REGISTERHOTKEY函数注册的热键时提交此消息  
WM\_PRINT = 791;  
应用程序发送此消息仅当WINDOWS或其它应用程序发出一个请求要求绘制一个应用程序的一部分；  
WM\_PRINTCLIENT = 792;  
WM\_HANDHELDFIRST = 856;  
WM\_HANDHELDLAST = 863;  
WM\_PENWINFIRST = $0380;  
WM\_PENWINLAST = $038F;  
WM\_COALESCE\_FIRST = $0390;  
WM\_COALESCE\_LAST = $039F;  
WM\_DDE\_FIRST = $03E0;  
WM\_DDE\_INITIATE = WM\_DDE\_FIRST + 0;  
一个DDE客户程序提交此消息开始一个与服务器程序的会话来响应那个指定的程序和主题名；  
WM\_DDE\_TERMINATE = WM\_DDE\_FIRST + 1;  
一个DDE应用程序（无论是客户还是服务器）提交此消息来终止一个会话；  
WM\_DDE\_ADVISE = WM\_DDE\_FIRST + 2;  
一个DDE客户程序提交此消息给一个DDE服务程序来请求服务器每当数据项改变时更新它  
WM\_DDE\_UNADVISE = WM\_DDE\_FIRST + 3;  
一个DDE客户程序通过此消息通知一个DDE服务程序不更新指定的项或一个特殊的剪贴板格式的项  
WM\_DDE\_ACK = WM\_DDE\_FIRST + 4;  
此消息通知一个DDE（动态数据交换）程序已收到并正在处理WM\_DDE\_POKE, WM\_DDE\_EXECUTE, WM\_DDE\_DATA, WM\_DDE\_ADVISE, WM\_DDE\_UNADVISE, or WM\_DDE\_INITIAT消息  
WM\_DDE\_DATA = WM\_DDE\_FIRST + 5;  
一个DDE服务程序提交此消息给DDE客户程序来传递个一数据项给客户或通知客户的一条可用数据项  
WM\_DDE\_REQUEST = WM\_DDE\_FIRST + 6;  
一个DDE客户程序提交此消息给一个DDE服务程序来请求一个数据项的值；  
WM\_DDE\_POKE = WM\_DDE\_FIRST + 7;  
一个DDE客户程序提交此消息给一个DDE服务程序，客户使用此消息来请求服务器接收一个未经同意的数据项；服务器通过答复WM\_DDE\_ACK消息提示是否它接收这个数据项；  
WM\_DDE\_EXECUTE = WM\_DDE\_FIRST + 8;  
一个DDE客户程序提交此消息给一个DDE服务程序来发送一个字符串给服务器让它象串行命令一样被处理，服务器通过提交WM\_DDE\_ACK消息来作回应；  
WM\_DDE\_LAST = WM\_DDE\_FIRST + 8;  
WM\_APP = $8000;  
WM\_USER = $0400;  
此消息能帮助应用程序自定义私有消息；  
/////////////////////////////////////////////////////////////////////  
通 知消息(Notification message)是指这样一种消息，一个窗口内的子控件发生了一些事情，需要通知父窗口。通知消息只适用于标准的窗口控件如按钮、列表框、组合框、编辑 框，以及Windows 95公共控件如树状视图、列表视图等。例如，单击或双击一个控件、在控件中选择部分文本、操作控件的滚动条都会产生通知消息。  
按扭  
B N \_ C L I C K E D //用户单击了按钮  
B N \_ D I S A B L E //按钮被禁止  
B N \_ D O U B L E C L I C K E D //用户双击了按钮  
B N \_ H I L I T E //用户加亮了按钮  
B N \_ PA I N T按钮应当重画  
B N \_ U N H I L I T E加亮应当去掉  
组合框  
C B N \_ C L O S E U P组合框的列表框被关闭  
C B N \_ D B L C L K用户双击了一个字符串  
C B N \_ D R O P D O W N组合框的列表框被拉出  
C B N \_ E D I T C H A N G E用户修改了编辑框中的文本  
C B N \_ E D I T U P D AT E编辑框内的文本即将更新  
C B N \_ E R R S PA C E组合框内存不足  
C B N \_ K I L L F O C U S组合框失去输入焦点  
C B N \_ S E L C H A N G E在组合框中选择了一项  
C B N \_ S E L E N D C A N C E L用户的选择应当被取消  
C B N \_ S E L E N D O K用户的选择是合法的  
C B N \_ S E T F O C U S组合框获得输入焦点  
编辑框  
E N \_ C H A N G E编辑框中的文本己更新  
E N \_ E R R S PA C E编辑框内存不足  
E N \_ H S C R O L L用户点击了水平滚动条  
E N \_ K I L L F O C U S编辑框正在失去输入焦点  
E N \_ M A X T E X T插入的内容被截断  
E N \_ S E T F O C U S编辑框获得输入焦点  
E N \_ U P D AT E编辑框中的文本将要更新  
E N \_ V S C R O L L用户点击了垂直滚动条消息含义  
列表框  
L B N \_ D B L C L K用户双击了一项  
L B N \_ E R R S PA C E列表框内存不够  
L B N \_ K I L L F O C U S列表框正在失去输入焦点  
L B N \_ S E L C A N C E L选择被取消  
L B N \_ S E L C H A N G E选择了另一项  
L B N \_ S E T F O C U S列表框获得输入焦点

posted @ [2008-08-28 10:29](http://www.cppblog.com/road420/archive/2008/08/28/60230.html) 深邃者 阅读(172) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/08/28/60230.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=60230) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=60230)

[**Window API函数大全(一)**](http://www.cppblog.com/road420/archive/2008/08/28/60229.html)

1. API之网络函数

WNetAddConnection 创建同一个网络资源的永久性连接 WNetAddConnection2 创建同一个网络资源的连接 WNetAddConnection3 创建同一个网络资源的连接 WNetCancelConnection 结束一个网络连接 WNetCancelConnection2 结束一个网络连接 WNetCloseEnum 结束一次枚举操作 WNetConnectionDialog 启动一个标准对话框，以便建立同网络资源的连接 WNetDisconnectDialog 启动一个标准对话框，以便断开同网络资源的连接 WNetEnumResource 枚举网络资源 WNetGetConnection 获取本地或已连接的一个资源的网络名称 WNetGetLastError 获取网络错误的扩展错误信息 WNetGetUniversalName 获取网络中一个文件的远程名称以及/或者UNC（统一命名规范）名称 WNetGetUser 获取一个网络资源用以连接的名字 WNetOpenEnum 启动对网络资源进行枚举的过程

2. API之消息函数

BroadcastSystemMessage 将一条系统消息广播给系统中所有的顶级窗口 GetMessagePos 取得消息队列中上一条消息处理完毕时的鼠标指针屏幕位置 GetMessageTime 取得消息队列中上一条消息处理完毕时的时间 PostMessage 将一条消息投递到指定窗口的消息队列 PostThreadMessage 将一条消息投递给应用程序 RegisterWindowMessage 获取分配给一个字串标识符的消息编号 ReplyMessage 答复一个消息 SendMessage 调用一个窗口的窗口函数，将一条消息发给那个窗口 SendMessageCallback 将一条消息发给窗口 SendMessageTimeout 向窗口发送一条消息 SendNotifyMessage 向窗口发送一条消息

3. API之文件处理函数

CloseHandle 关闭一个内核对象。其中包括文件、文件映射、进程、线程、安全和同步对象等 CompareFileTime 对比两个文件的时间 CopyFile 复制文件 CreateDirectory 创建一个新目录 CreateFile 打开和创建文件、管道、邮槽、通信服务、设备以及控制台 CreateFileMapping 创建一个新的文件映射对象 DeleteFile 删除指定文件 DeviceIoControl 对设备执行指定的操作 DosDateTimeToFileTime 将DOS日期和时间值转换成一个 win32 FILETIME 值 FileTimeToDosDateTime 将一个 win32 FILETIME 值转换成DOS日期和时间值 FileTimeToLocalFileTime 将一个FILETIME结构转换成本地时间 FileTimeToSystemTime 根据一个FILETIME结构的内容，装载一个SYSTEMTIME结构 FindClose 关闭由FindFirstFile函数创建的一个搜索句柄 FindFirstFile 根据文件名查找文件 FindNextFile 根据调用FindFirstFile函数时指定的一个文件名查找下一个文件 FlushFileBuffers 针对指定的文件句柄，刷新内部文件缓冲区 FlushViewOfFile 将写入文件映射缓冲区的所有数据都刷新到磁盘 GetBinaryType 判断文件是否可以执行 GetCompressedFileSize 判断一个压缩文件在磁盘上实际占据的字节数 GetCurrentDirectory 在一个缓冲区中装载当前目录 GetDiskFreeSpace 获取与一个磁盘的组织有关的信息，以及了解剩余空间的容量 GetDiskFreeSpaceEx 获取与一个磁盘的组织以及剩余空间容量有关的信息 GetDriveType 判断一个磁盘驱动器的类型 GetExpandedName 取得一个压缩文件的全名 GetFileAttributes 判断指定文件的属性 GetFileInformationByHandle 这个函数提供了获取文件信息的一种机制 GetFileSize 判断文件长度 GetFileTime 取得指定文件的时间信息 GetFileType 在给出文件句柄的前提下，判断文件类型 GetFileVersionInfo 从支持版本标记的一个模块里获取文件版本信息 GetFileVersionInfoSize 针对包含了版本资源的一个文件，判断容纳文件版本信息需要一个多大的缓冲区 GetFullPathName 获取指定文件的完整路径名 GetLogicalDrives 判断系统中存在哪些逻辑驱动器字母 GetLogicalDriveStrings 获取一个字串，其中包含了当前所有逻辑驱动器的根驱动器路径 GetOverlappedResult 判断一个重叠操作当前的状态 GetPrivateProfileInt 为初始化文件（.ini文件）中指定的条目获取一个整数值 GetPrivateProfileSection 获取指定小节（在.ini文件中）所有项名和值的一个列表 GetPrivateProfileString 为初始化文件中指定的条目取得字串 GetProfileInt 取得win.ini初始化文件中指定条目的一个整数值 GetProfileSection 获取指定小节（在win.ini文件中）所有项名和值的一个列表 GetProfileString 为win.ini初始化文件中指定的条目取得字串 GetShortPathName 获取指定文件的短路径名 GetSystemDirectory 取得Windows系统目录（即System目录）的完整路径名 GetTempFileName 这个函数包含了一个临时文件的名字，它可由应用程序使用 GetTempPath 获取为临时文件指定的路径 GetVolumeInformation 获取与一个磁盘卷有关的信息 GetWindowsDirectory 获取Windows目录的完整路径名 hread 参考lread hwrite 参考lwrite函数 lclose 关闭指定的文件 lcreat 创建一个文件 llseek 设置文件中进行读写的当前位置 LockFile 锁定文件的某一部分，使其不与其他应用程序共享 LockFileEx 与LockFile相似，只是它提供了更多的功能 lopen 以二进制模式打开指定的文件 lread 将文件中的数据读入内存缓冲区 lwrite 将数据从内存缓冲区写入一个文件 LZClose 关闭由LZOpenFile 或 LZInit函数打开的一个文件 LZCopy 复制一个文件 LZInit 这个函数用于初始化内部缓冲区 LZOpenFile 该函数能执行大量不同的文件处理，而且兼容于压缩文件 LZRead 将数据从文件读入内存缓冲区 LZSeek 设置一个文件中进行读写的当前位置 MapViewOfFile 将一个文件映射对象映射到当前应用程序的地址空间 MoveFile 移动文件 OpenFile 这个函数能执行大量不同的文件操作 OpenFileMapping 打开一个现成的文件映射对象 QueryDosDevice 在Windows NT中，DOS设备名会映射成NT系统设备名。该函数可判断当前的设备映射情况 ReadFile 从文件中读出数据 ReadFileEx 与ReadFile相似，只是它只能用于异步读操作，并包含了一个完整的回调 RegCloseKey 关闭系统注册表中的一个项（或键） RegConnectRegistry 访问远程系统的部分注册表 RegCreateKey 在指定的项下创建或打开一个项 RegCreateKeyEx 在指定项下创建新项的更复杂的方式。在Win32环境中建议使用这个函数 RegDeleteKey 删除现有项下方一个指定的子项 RegDeleteValue 删除指定项下方的一个值 RegEnumKey 枚举指定项的子项。在Win32环境中应使用RegEnumKeyEx RegEnumKeyEx 枚举指定项下方的子项 RegEnumValue 枚举指定项的值 RegFlushKey 将对项和它的子项作出的改动实际写入磁盘 RegGetKeySecurity 获取与一个注册表项有关的安全信息 RegLoadKey 从以前用RegSaveKey函数创建的一个文件里装载注册表信息 RegNotifyChangeKeyValue 注册表项或它的任何一个子项发生变化时，用这个函数提供一种通知机制 RegOpenKey 打开一个现有的注册表项 RegOpenKeyEx 打开一个现有的项。在win32下推荐使用这个函数 RegQueryInfoKey 获取与一个项有关的信息 RegQueryValue 取得指定项或子项的默认（未命名）值 RegQueryValueEx 获取一个项的设置值 RegReplaceKey 用一个磁盘文件保存的信息替换注册表信息；并创建一个备份，在其中包含当前注册表信息 RegRestoreKey 从一个磁盘文件恢复注册表信息 RegSaveKey 将一个项以及它的所有子项都保存到一个磁盘文件 RegSetKeySecurity 设置指定项的安全特性 RegSetValue 设置指定项或子项的默认值 RegSetValueEx 设置指定项的值 RegUnLoadKey 卸载指定的项以及它的所有子项 RemoveDirectory 删除指定目录 SearchPath 查找指定文件 SetCurrentDirectory 设置当前目录 SetEndOfFile 针对一个打开的文件，将当前文件位置设为文件末尾 SetFileAttributes 设置文件属性 SetFilePointer 在一个文件中设置当前的读写位置 SetFileTime 设置文件的创建、访问及上次修改时间 SetHandleCount 这个函数不必在win32下使用；即使使用，也不会有任何效果 SetVolumeLabel 设置一个磁盘的卷标（Label） SystemTimeToFileTime 根据一个FILETIME结构的内容，载入一个SYSTEMTIME结构 UnlockFile 解除对一个文件的锁定 UnlockFileEx 解除对一个文件的锁定 UnmapViewOfFile 在当前应用程序的内存地址空间解除对一个文件映射对象的映射 VerFindFile 用这个函数决定一个文件应安装到哪里 VerInstallFile 用这个函数安装一个文件 VerLanguageName 这个函数能根据16位语言代码获取一种语言的名称 VerQueryValue 这个函数用于从版本资源中获取信息 WriteFile 将数据写入一个文件 WriteFileEx 与WriteFile类似，只是它只能用于异步写操作，并包括了一个完整的回调 WritePrivateProfileSection 为一个初始化文件（.ini）中指定的小节设置所有项名和值 WritePrivateProfileString 在初始化文件指定小节内设置一个字串 WriteProfileSection 为Win.ini初始化文件中一个指定的小节设置所有项名和值 WriteProfileString 在Win.ini初始化文件指定小节内设置一个字串

4. API之打印函数 AbortDoc 取消一份文档的打印 AbortPrinter 删除与一台打印机关联在一起的缓冲文件 AddForm 为打印机的表单列表添加一个新表单 AddJob 用于获取一个有效的路径名，以便用它为作业创建一个后台打印文件。它也会为作业分配一个作业编号 AddMonitor 为系统添加一个打印机监视器 AddPort 启动“添加端口”对话框，允许用户在系统可用端口列表中加入一个新端口 AddPrinter 在系统中添加一台新打印机 AddPrinterConnection 连接指定的打印机 AddPrinterDriver 为指定的系统添加一个打印驱动程序 AddPrintProcessor 为指定的系统添加一个打印处理器 AddPrintProvidor 为系统添加一个打印供应商 AdvancedDocumentProperties 启动打印机文档设置对话框 ClosePrinter 关闭一个打开的打印机对象 ConfigurePort 针对指定的端口，启动一个端口配置对话框 ConnectToPrinterDlg 启动连接打印机对话框，用它同访问网络的打印机连接 DeleteForm 从打印机可用表单列表中删除一个表单 DeleteMonitor 删除指定的打印监视器 DeletePort 启动“删除端口”对话框，允许用户从当前系统删除一个端口 DeletePrinter 将指定的打印机标志为从系统中删除 DeletePrinterConnection 删除与指定打印机的连接 DeletePrinterDriver 从系统删除一个打印机驱动程序 DeletePrintProcessor 从指定系统删除一个打印处理器 DeletePrintProvidor 从系统中删除一个打印供应商 DeviceCapabilities 利用这个函数可获得与一个设备的能力有关的信息 DocumentProperties 打印机配置控制函数 EndDocAPI 结束一个成功的打印作业 EndDocPrinter 在后台打印程序的级别指定一个文档的结束 EndPage 用这个函数完成一个页面的打印，并准备设备场景，以便打印下一个页 EndPagePrinter 指定一个页在打印作业中的结尾 EnumForms 枚举一台打印机可用的表单 EnumJobs 枚举打印队列中的作业 EnumMonitors 枚举可用的打印监视器 EnumPorts 枚举一个系统可用的端口 EnumPrinterDrivers 枚举指定系统中已安装的打印机驱动程序 EnumPrinters 枚举系统中安装的打印机 EnumPrintProcessorDatatypes 枚举由一个打印处理器支持的数据类型 EnumPrintProcessors 枚举系统中可用的打印处理器 Escape 设备控制函数 FindClosePrinterChangeNotification 关闭用FindFirstPrinterChangeNotification函数获取的一个打印机通告对象 FindFirstPrinterChangeNotification 创建一个新的改变通告对象，以便我们注意打印机状态的各种变化 FindNextPrinterChangeNotification 用这个函数判断触发一次打印机改变通告信号的原因 FreePrinterNotifyInfo 释放由FindNextPrinterChangeNotification函数分配的一个缓冲区 GetForm 取得与指定表单有关的信息 GetJob 获取与指定作业有关的信息 GetPrinter 取得与指定打印机有关的信息 GetPrinterData 为打印机设置注册表配置信息 GetPrinterDriver 针对指定的打印机，获取与打印机驱动程序有关的信息 GetPrinterDriverDirectory 判断指定系统中包含了打印机驱动程序的目录是什么 GetPrintProcessorDirectory 判断指定系统中包含了打印机处理器驱动程序及文件的目录 OpenPrinter 打开指定的打印机，并获取打印机的句柄 PrinterMessageBox 在拥有指定打印作业的系统上显示一个打印机出错消息框 PrinterProperties 启动打印机属性对话框，以便对打印机进行配置 ReadPrinter 从打印机读入数据 ResetDC 重设一个设备场景 ResetPrinter 改变指定打印机的默认数据类型及文档设置 ScheduleJob 提交一个要打印的作业 SetAbortProc 为Windows指定取消函数的地址 SetForm 为指定的表单设置信息 SetJob 对一个打印作业的状态进行控制 SetPrinter 对一台打印机的状态进行控制 SetPrinterData 设置打印机的注册表配置信息 StartDoc 开始一个打印作业 StartDocPrinter 在后台打印的级别启动一个新文档 StartPage 打印一个新页前要先调用这个函数 StartPagePrinter 在打印作业中指定一个新页的开始 WritePrinter 将发送目录中的数据写入打印机

5. API之文本和字体函数

AddFontResource 在Windows系统中添加一种字体资源 CreateFont 用指定的属性创建一种逻辑字体 CreateFontIndirect 用指定的属性创建一种逻辑字体 CreateScalableFontResource 为一种TureType字体创建一个资源文件，以便能用API函数AddFontResource将其加入Windows系统 DrawText 将文本描绘到指定的矩形中 DrawTextEx 与DrawText相似，只是加入了更多的功能 EnumFontFamilies 列举指定设备可用的字体 EnumFontFamiliesEx 列举指定设备可用的字体 EnumFonts 列举指定设备可用的字体 ExtTextOut 经过扩展的文本描绘函数。也请参考SetTextAlign函数 GetAspectRatioFilterEx 用SetMapperFlags要求Windows只选择与设备当前纵横比相符的光栅字体时，本函数可判断纵横比大小 GetCharABCWidths 判断TureType字体中一个或多个字符的A-B-C大小 GetCharABCWidthsFloat 查询一种字体中一个或多个字符的A-B-C尺寸 GetCharacterPlacement 该函数用于了解如何用一个给定的字符显示一个字串 GetCharWidth 调查字体中一个或多个字符的宽度 GetFontData 接收一种可缩放字体文件的数据 GetFontLanguageInfo 返回目前选入指定设备场景中的字体的信息 GetGlyphOutline 取得TureType字体中构成一个字符的曲线信息 GetKerningPairs 取得指定字体的字距信息 GetOutlineTextMetrics 接收与TureType字体内部特征有关的详细信息 GetRasterizerCaps 了解系统是否有能力支持可缩放的字体 GetTabbedTextExtent 判断一个字串占据的范围，同时考虑制表站扩充的因素 GetTextAlign 接收一个设备场景当前的文本对齐标志 GetTextCharacterExtra 判断额外字符间距的当前值 GetTextCharset 接收当前选入指定设备场景的字体的字符集标识符 GetTextCharsetInfo 获取与当前选定字体的字符集有关的详细信息 GetTextColor 判断当前字体颜色。通常也称为“前景色” GetTextExtentExPoint 判断要填入指定区域的字符数量。也用一个数组装载每个字符的范围信息 GetTextExtentPoint 判断一个字串的大小（范围） GetTextFace 获取一种字体的字样名 GetTextMetrics 获取与选入一种设备场景的物理字体有关的信息 GrayString 描绘一个以灰色显示的字串。通常由Windows用于标识禁止状态 PolyTextOut 描绘一系列字串 RemoveFontResource 从Windows系统中删除一种字体资源 SetMapperFlags Windows对字体进行映射时，可用该函数选择与目标设备的纵横比相符的光栅字体 SetTextAlign 设置文本对齐方式，并指定在文本输出过程中使用设备场景的当前位置 SetTextCharacterExtra 描绘文本的时候，指定要在字符间插入的额外间距 SetTextColor 设置当前文本颜色。这种颜色也称为“前景色” SetTextJustification 通过指定一个文本行应占据的额外空间，可用这个函数对文本进行两端对齐处理 TabbedTextOut 支持制表站的一个文本描绘函数 TextOut 文本绘图函数

6. API之菜单函数

AppendMenu 在指定的菜单里添加一个菜单项 CheckMenuItem 复选或撤消复选指定的菜单条目 CheckMenuRadioItem 指定一个菜单条目被复选成“单选”项目 CreateMenu 创建新菜单 CreatePopupMenu 创建一个空的弹出式菜单 DeleteMenu 删除指定的菜单条目 DestroyMenu 删除指定的菜单 DrawMenuBar 为指定的窗口重画菜单 EnableMenuItem 允许或禁止指定的菜单条目 GetMenu 取得窗口中一个菜单的句柄 GetMenuCheckMarkDimensions 返回一个菜单复选符的大小 GetMenuContextHelpId 取得一个菜单的帮助场景ID GetMenuDefaultItem 判断菜单中的哪个条目是默认条目 GetMenuItemCount 返回菜单中条目（菜单项）的数量 GetMenuItemID 返回位于菜单中指定位置处的条目的菜单ID GetMenuItemInfo 取得（接收）与一个菜单条目有关的特定信息 GetMenuItemRect 在一个矩形中装载指定菜单条目的屏幕坐标信息 GetMenuState 取得与指定菜单条目状态有关的信息 GetMenuString 取得指定菜单条目的字串 GetSubMenu 取得一个弹出式菜单的句柄，它位于菜单中指定的位置 GetSystemMenu 取得指定窗口的系统菜单的句柄 HiliteMenuItem 控制顶级菜单条目的加亮显示状态 InsertMenu 在菜单的指定位置处插入一个菜单条目，并根据需要将其他条目向下移动 InsertMenuItem 插入一个新菜单条目 IsMenu 判断指定的句柄是否为一个菜单的句柄 LoadMenu 从指定的模块或应用程序实例中载入一个菜单 LoadMenuIndirect 载入一个菜单 MenuItemFromPoint 判断哪个菜单条目包含了屏幕上一个指定的点 ModifyMenu 改变菜单条目 RemoveMenu 删除指定的菜单条目 SetMenu 设置窗口菜单 SetMenuContextHelpId 设置一个菜单的帮助场景ID SetMenuDefaultItem 将一个菜单条目设为默认条目 SetMenuItemBitmaps 设置一幅特定位图，令其在指定的菜单条目中使用，代替标准的复选符号（√） SetMenuItemInfo 为一个菜单条目设置指定的信息 TrackPopupMenu 在屏幕的任意地方显示一个弹出式菜单 TrackPopupMenuEx 与TrackPopupMenu相似，只是它提供了额外的功能

posted @ [2008-08-28 10:28](http://www.cppblog.com/road420/archive/2008/08/28/60229.html) 深邃者 阅读(241) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/08/28/60229.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=60229) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=60229)

[**windows消息说明**](http://www.cppblog.com/road420/archive/2008/07/22/56879.html)

**WM\_PAINT = $000F;  
要求一个窗口重画自己  
WM\_CLOSE = $0010;  
当一个窗口或应用程序要关闭时发送一个信号  
WM\_QUERYENDSESSION = $0011;  
当用户选择结束对话框或程序自己调用ExitWindows函数  
WM\_QUIT = $0012;  
用来结束程序运行或当程序调用postquitmessage函数  
WM\_QUERYOPEN = $0013;  
当用户窗口恢复以前的大小位置时，把此消息发送给某个图标  
WM\_ERASEBKGND = $0014;  
当窗口背景必须被擦除时（例在窗口改变大小时）  
WM\_SYSCOLORCHANGE = $0015;  
当系统颜色改变时，发送此消息给所有顶级窗口  
WM\_ENDSESSION = $0016;  
当系统进程发出WM\_QUERYENDSESSION消息后，此消息发送给应用程序，  
通知它对话是否结束  
WM\_SYSTEMERROR = $0017;  
WM\_SHOWWINDOW = $0018;  
当隐藏或显示窗口是发送此消息给这个窗口  
WM\_ACTIVATEAPP = $001C;  
发此消息给应用程序哪个窗口是激活的，哪个是非激活的；  
WM\_FONTCHANGE = $001D;  
当系统的字体资源库变化时发送此消息给所有顶级窗口  
WM\_TIMECHANGE = $001E;  
当系统的时间变化时发送此消息给所有顶级窗口  
WM\_CANCELMODE = $001F;  
发送此消息来取消某种正在进行的摸态（操作）  
WM\_SETCURSOR = $0020;  
如果鼠标引起光标在某个窗口中移动且鼠标输入没有被捕获时，就发消息给某个窗口  
WM\_MOUSEACTIVATE = $0021;  
当光标在某个非激活的窗口中而用户正按着鼠标的某个键发送此消息给当前窗口  
WM\_CHILDACTIVATE = $0022;  
发送此消息给MDI子窗口当用户点击此窗口的标题栏，或当窗口被激活，移动，改变大小**  
**WM\_QUEUESYNC = $0023;  
此消息由基于计算机的训练程序发送，通过WH\_JOURNALPALYBACK的hook程序  
分离出用户输入消息  
WM\_GETMINMAXINFO = $0024;  
此消息发送给窗口当它将要改变大小或位置；  
WM\_PAINTICON = $0026;  
发送给最小化窗口当它图标将要被重画  
WM\_ICONERASEBKGND = $0027;  
此消息发送给某个最小化窗口，仅当它在画图标前它的背景必须被重画  
WM\_NEXTDLGCTL = $0028;  
发送此消息给一个对话框程序去更改焦点位置  
WM\_SPOOLERSTATUS = $002A;  
每当打印管理列队增加或减少一条作业时发出此消息  
WM\_DRAWITEM = $002B;  
当button，combobox，listbox，menu的可视外观改变时发送  
此消息给这些空件的所有者**  
**WM\_MEASUREITEM = $002C;  
当button, combo box, list box, list view control, or menu item 被创建时  
发送此消息给控件的所有者  
WM\_DELETEITEM = $002D;  
当the list box 或 combo box 被销毁 或 当 某些项被删除通过LB\_DELETESTRING, LB\_RESETCONTENT, CB\_DELETESTRING, or CB\_RESETCONTENT 消息  
WM\_VKEYTOITEM = $002E;  
此消息有一个LBS\_WANTKEYBOARDINPUT风格的发出给它的所有者来响应WM\_KEYDOWN消息  
WM\_CHARTOITEM = $002F;  
此消息由一个LBS\_WANTKEYBOARDINPUT风格的列表框发送给他的所有者来响应WM\_CHAR消息**

posted @ [2008-07-22 18:16](http://www.cppblog.com/road420/archive/2008/07/22/56879.html) 深邃者 阅读(135) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/07/22/56879.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=56879) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=56879)

[**MFC共享DLL模块状态的切换AFX\_MODULE\_STATE**](http://www.cppblog.com/road420/archive/2008/07/22/56877.html)

在DLL中使用资源（一）  
  
yuwei - 转载 （2004-12-10 14:02:00）  .Net控件开发 ActiveX/COM开发 CLX/VCL开发 Java组件开发 VC/MFC 控件使用经验谈 DHtml/Script 控件开发基础 数据库 编程规范 单元测试 可重用软件技术理论 其 它  
   
   
   
     
  
在DLL中使用资源（一）  
现在最常看见的关于DLL的问题就是如何在DLL中使用对话框，这是一个很普遍的关于如何在DLL中使用资源的问题。这里我们从Win32 DLL和MFC DLL两个方面来分析并解决这个问题。  
  
1．Win32 DLL  
在Win32 DLL中使用对话框很简单，你只需要在你的DLL中添加对话框资源，而且可以在对话框上面设置你所需要的控件。然后使用DialogBox或者CreateDialog这两个函数（或相同作用的其它函数）来创建对话框，并定义你自己的对话框回调函数处理对话框收到的消息。下面通过一个具体实例来学习如何在Win32 DLL中使用对话框，可以按照以下步骤来完成这个例子：  
  
  
1）在VC菜单中File->New新建一个命名为UseDlg的Win32 Dynamic-Link Library工程，下一步选择A simple DLL project。  
  
  
2）在VC菜单中Insert->Resource添加一个ID为IDD\_DLG\_SHOW的Dialog资源，将此Dialog上的Cancel按钮去掉，仅保留OK按钮。再添加一个ID为IDD\_ABOUTBOX的对话框，其Caption为About。保存此资源，将资源文件命名为UseDlg.rc。并将resource.h和UseDlg.rc加入到工程里面。  
  
  
3）在UseDlg.app中包含resource.h，并添加如下代码：  
  
  
HINSTANCE hinst = NULL;  
  
HWND hwndDLG = NULL;  
  
  
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,  
  
WPARAM wParam, LPARAM lParam);  
  
BOOL CALLBACK AboutProc(HWND hDlg, UINT message,  
  
WPARAM wParam, LPARAM lParam);  
  
extern "C" \_\_declspec(dllexport) void ShowDlg();  
  
  
BOOL APIENTRY DllMain( HANDLE hModule,  
  
DWORD ul\_reason\_for\_call,  
  
LPVOID lpReserved  
  
)  
  
{  
  
switch(ul\_reason\_for\_call)  
  
{  
  
case DLL\_PROCESS\_ATTACH:  
  
hinst = (HINSTANCE)hModule;  
  
case DLL\_PROCESS\_DETACH:  
  
break;  
  
}  
  
return TRUE;  
  
}  
  
  
extern "C" \_\_declspec(dllexport) void ShowDlg()  
  
{  
  
hwndDLG = CreateDialog(hinst,MAKEINTRESOURCE(IDD\_DLG\_SHOW),  
  
NULL,(DLGPROC)DlgProc);  
  
ShowWindow(hwndDLG, SW\_SHOW);  
  
}  
  
  
BOOL CALLBACK DlgProc(HWND hDlg, UINT message,  
  
WPARAM wParam, LPARAM lParam)  
  
{  
  
switch(message)  
  
{  
  
case WM\_INITDIALOG:  
  
return TRUE;  
  
  
case WM\_COMMAND:  
  
if(LOWORD(wParam)==IDOK)  
  
DialogBox(hinst,MAKEINTRESOURCE(IDD\_ABOUTBOX),  
  
hDlg,(DLGPROC)AboutProc);  
  
return TRUE;  
  
case WM\_CLOSE:  
  
DestroyWindow(hDlg);  
  
hwndDLG = NULL;  
  
return TRUE;  
  
}  
  
return FALSE;  
  
}  
  
  
BOOL CALLBACK AboutProc(HWND hDlg, UINT message,  
  
WPARAM wParam, LPARAM lParam)  
  
{  
  
switch(message)  
  
{  
  
case WM\_CLOSE:  
  
EndDialog(hDlg,NULL);  
  
hwndDLG = NULL;  
  
return TRUE;  
  
}  
  
return FALSE;  
  
}  
  
  
4）编译生成UseDlg.dll和UseDlg.lib。  
  
  
接下来我们建立调用此DLL的应用程序，其步骤如下：  
  
  
  
1）在VC菜单中File->New新建一个命名为Use的MFC AppWizard(exe)工程，下一步选择Dialog Based之后点击Finish按钮。  
  
  
2）在主对话框上面添加一个按钮，之后双击此按钮，会弹出Add Member Function的对话框，直接点击OK进入void CUseDlg::OnButton1()函数。并在此函数内添加一个函数调用：ShowDlg();。  
  
  
3）紧跟在#include语句后面加上如下代码：  
  
  
extern "C" \_\_declspec(dllexport) void ShowDlg();  
  
#pragma comment(lib,"debug/UseDlg")  
  
  
4）将上面UseDlg工程中生成的UseDlg.dll和UseDlg.lib两个文件复制到Use工程的Debug目录内。  
  
  
5）编译生成Use.exe。  
  
  
运行Use.exe，点击Button1按钮，可以看到一个名称为Dialog的非模态对话框弹出。点击上面的按钮，可以弹出模态对话框About。运行成功。  
  
  
让我们来回顾一下在Win32 DLL中使用对话框的过程。  
  
  
在DLL中，我们定义了两个对话框资源：IDD\_DLG\_SHOW和IDD\_ABOUTBOX，并且导出了函数ShowDlg。在函数ShowDlg之中使用CreateDialog函数创建了非模态对话框IDD\_DLG\_SHOW，并指定了该对话框的回调函数DlgProc。在DlgProc之中处理了WM\_INITDIALOG、WM\_COMMAND和WM\_CLOSE消息，以响应用户对对话框所做的动作。在处理按钮动作的时候，使用DialogBox函数创建IDD\_ABOUTBOX这个模态对话框，指定其回调函数为AboutProc，并且在AboutProc中处理其相应消息。  
  
  
在EXE中，我们使用隐式链接的方法来调用DLL，并使用DLL中导出的ShowDlg函数来调用DLL中的对话框。  
  
  
在Win32 DLL中使用对话框就是这么简单，下面让我们来看一下在MFC DLL中如何使用对话框。  
  
2．MFC DLL  
在MFC DLL中使用对话框不像Win32 DLL中那么简单，主要是因为MFC程序中存在一个模块状态（Module State）的问题，也就是资源重复的问题。（此处的术语模块是指一个可执行程序，或指其操作不依赖于应用程序的其余部分但使用MFC运行库的共享副本的一个DLL（或一组DLL）。我们所创建的MFC DLL就是这种模块的一个典型实例。）  
  
  
在每个模块（EXE或DLL）中，都存在一种全局的状态数据，MFC依靠这种全局的状态数据来区分不同的模块，以执行正确的操作。这种数据包括：Windows实例句柄（用于加载资源），指向应用程序当前的CWinApp和CWinThread对象的指针，OLE模块引用计数，以及维护Windows对象句柄与相应的MFC对象实例之间连接的各种映射等。但当应用程序使用多个模块时，每个模块的状态数据不是应用程序范围的。相反，每个模块具有自已的MFC状态数据的私有副本。这种全局的状态数据就叫做MFC模块状态。  
  
  
模块的状态数据包含在结构中，并且总是可以通过指向该结构的指针使用。当代码在执行时进入了某一个模块时，只有此模块的状态为“当前”或“有效”状态时，MFC才能正确的区分此模块并执行正确的操作。  
  
  
例如，MFC应用程序可以使用下面代码从资源文件中加载字符串：  
  
  
CString str;  
  
str.LoadString(IDS\_MYSTRING);  
  
  
使用这种代码非常方便，但它掩盖了这样一个事实：即此程序中IDS\_MYSTRING可能不是唯一的标识符。一个程序可以加载多个DLL，某些DLL可能也用IDS\_MYSTRING标识符定义了一个资源。MFC怎样知道应该加载哪个资源呢？MFC使用当前模块状态查找资源句柄。如果当前模块不是我们要使用的正确模块，那么就会产生不正确的调用或者错误。  
  
  
  
按照MFC库的链接方法，一个MFC DLL有两种使用MFC库的方法：静态链接到MFC的DLL和动态链接到MFC的DLL。下面我们就按照这两种类型的MFC DLL来介绍如何切换当前模块状态以正确的在MFC DLL中使用资源。  
  
1、静态链接到MFC的DLL  
  
静态链接到MFC的规则DLL与MFC库静态链接，则此时MFC库不能共享，所以MFC总是使用它所链接的DLL的模块状态。这样也就不存在管理模块状态的问题。但使用这种方法的缺点是DLL程序将会变大，而且会在程序中留下重复代码。下面给出的例子验证了这一点。本例可以按照以下步骤来完成：  
  
  
1）在VC菜单中File->New新建一个命名为DLLStatic的MFC AppWizard的工程，下一步选择Regular DLL with MFC statically linked。  
  
  
2）在工程中添加一个对话框资源，其ID为：IDD\_ABOUTBOX。并在resource.h之中将IDD\_ABOUTBOX 的数值改为100。  
  
  
3）在DLLStatic.cpp中定义如下函数：  
  
  
void ShowDlg()  
  
{  
  
CDialog dlg(IDD\_ABOUTBOX);  
  
dlg.DoModal();  
  
}  
  
  
4）在DLLStatic.def文件中的EXPORTS语句中添加一行：ShowDlg，以导出ShowDlg函数。  
  
  
5）编译生成DLLStatic.dll和DLLStatic.lib。  
  
  
继续使用上一节中的Use工程，将前面生成的DLLStatic.dll和DLLStatic.lib两个文件复制到工程的Debug目录内，并将  
  
  
extern "C" \_\_declspec(dllexport) void ShowDlg();  
  
#pragma comment(lib,"debug/UseDlg")  
  
  
这两行改为：  
  
  
void ShowDlg();  
  
#pragma comment(lib,"debug/DLLStatic")  
  
  
编译并运行Use.exe。点击按钮，可以看到DLLStatic中的模态对话框弹出。  
  
  
本例中，可以注意到DLL中所定义的About对话框资源与EXE中所定义的About对话框资源ID完全相同，但是当我们点击Use.exe上面的按钮时，弹出的是DLL中的模态对话框。说明，当使用静态链接到MFC的规则DLL时，不存在管理模块状态的问题。  
  
   
2、动态链接到MFC的DLL  
  
在讨论关于动态链接到MFC的DLL的模块状态问题之前，先来看一个例子。本例可以通过如下步骤来完成：  
  
  
  
1）在VC菜单中File->New新建一个命名为DLLShared的MFC AppWizard的工程，下一步选择Regular DLL using shared MFC DLL。  
  
  
  
2）在工程中添加一个对话框资源，其ID为：IDD\_ABOUTBOX。并在resource.h之中将IDD\_ABOUTBOX 的数值改为100。  
  
  
  
3）在DLLShared.cpp中定义如下函数：  
  
  
  
void ShowDlg()  
  
{  
  
CDialog dlg(IDD\_ABOUTBOX);  
  
dlg.DoModal();  
  
}  
  
  
  
4）在DLLShared.def文件中的EXPORTS语句中添加一行：ShowDlg，以导出ShowDlg函数。  
  
  
  
5）编译生成DLLShared.dll和DLLShared.lib。  
  
  
  
继续使用上面的Use工程，将前面生成的DLLShared.dll和DLLShared.lib两个文件复制到工程的Debug目录内，并将  
  
  
  
extern "C" \_\_declspec(dllexport) void ShowDlg();  
  
#pragma comment(lib,"debug/DLLStatic")  
  
  
  
这两行改为：  
  
  
  
void ShowDlg();  
  
#pragma comment(lib,"debug/DLLShared")  
  
  
  
编译并运行Use.exe。点击按钮，这次你看到了什么？对，没错，这次弹出的是Use.exe的关于对话框。将上述例子的DLL类型换成MFC Extension DLL(using shared MFC DLL)也会出现相同的问题。  
  
  
  
为什么会出现上面的问题？这是因为在使用了MFC共享库的时候，默认情况下，MFC使用主应用程序的资源句柄来加载资源模板。虽然我们调用的是DLL中的函数来显示DLL中的对话框，并且对应的对话框模板是存储在DLL中的，但MFC仍旧在主应用程序也就是Use.exe中寻找相应的对话框模板。由于在DLL中所定义的对话框资源ID与主应用程序中所定义的关于对话框的资源ID相同，所以MFC就把主应用程序中的关于对话框显示了出来。如果二者不同，则MFC就认为DLL中所定义的对话框资源不存在，dlg.DoModal会返回0，也就是什么都不会显示。  
  
  
  
那么如何解决上述问题呢？解决办法就是在适当的时候进行模块状态切换，以保证具有当前状态的模块是我们所需要的模块从而使用正确的资源。MFC提供了下列函数和宏来完成这些工作：  
  
  
  
AfxGetStaticModuleState：这是一个函数，其函数原型为：  
  
  
  
AFX\_MODULE\_STATE\* AFXAPI AfxGetStaticModuleState( );  
  
  
  
此函数在堆栈上构造AFX\_MODULE\_STATE类的实例pModuleState并对其赋值后将其返回。在AFX\_MODULE\_STATE类的构造函数中，该类获取指向当前模块状态的指针并将其存储在成员变量中，然后将pModuleState设置为新的有效模块状态。在它的析构函数中，该类将存储在其成员变量中的指针还原为存贮的前一个模块状态。  
  
  
  
AFX\_MANAGE\_STATE：这是一个宏，其原型为：  
  
  
  
AFX\_MANAGE\_STATE( AFX\_MODULE\_STATE\* pModuleState )  
  
  
  
该宏用于将pModuleState（指向包含模块全局数据也就是模块状态的AFX\_MODULE\_STATE结构的指针）设置为当前的即时作用空间中（the remainder of the immediate containing scope）的有效模块状态。在离开包含该宏的作用空间时，前一个有效的模块状态自动还原。  
  
  
  
AfxGetResourceHandle：这个函数的原型为：  
  
  
  
HINSTANCE AfxGetResourceHandle( );  
  
  
  
该函数返回了一个保存了HINSTANCE类型的、应用程序默认所加载资源的模块的句柄。  
  
  
  
AfxSetResourceHandle：这个函数的原型为：  
  
  
  
void AfxSetResourceHandle( HINSTANCE hInstResource );  
  
  
  
该函数将hInstResource所代表的模块设置为具有当前状态的模块。  
  
  
  
通过使用上述四个函数或宏就可以正确的在动态链接到MFC的DLL中切换模块状态。接下来我们将通过修改上面出现问题的那个例子来介绍如何使用上述四个函数或宏。先来看看Regular DLL using shared MFC DLL类型：  
  
  
  
在上述例子的第三步的ShowDlg函数的第一条语句前加上如下语句（要确保该语句在函数实现的第一行）：  
  
  
  
AFX\_MANAGE\_STATE(AfxGetStaticModuleState());  
  
  
  
之后重新编译生成DLLShared.dll和DLLShared.lib，并将这两个文件重新拷贝到Use工程的Debug目录内。这次编译生成Use.exe并运行，点击按钮，可以看到弹出的时我们在DLL中所加入的那个对话框，而不再是Use.exe的关于对话框了。  
  
  
  
通过上面的讲解，相信你已经知道该语句的作用了。在函数ShowDlg的第一行加上这么一句后，每次调用DLL的应用程序使用该函数的时候，MFC库都会自动切换当前模块状态，这样就保证了资源读取的正确性。  
  
  
  
AFX\_MANAGE\_STATE(AfxGetStaticModuleState());是自动切换当前模块状态，也可以通过使用AfxGetResourceHandle和AfxSetResourceHandle来手动切换当前模块状态。具体使用方法如下：  
  
  
  
在上述例子的第三步的ShowDlg函数的第一条语句前加上如下语句（要确保该语句在函数实现的第一行）：  
  
  
  
HINSTANCE save\_hInstance = AfxGetResourceHandle();  
  
AfxSetResourceHandle(theApp.m\_hInstance);  
  
  
  
在调用对话框成功之后，也就是dlg.DoModal();之后，添加：  
  
  
  
AfxSetResourceHandle(save\_hInstance);  
  
  
  
这种方法在进入ShowDlg函数之后，通过AfxGetResourceHandle来获得并保存当前状态模块的句柄。然后获得DLL模块的句柄theApp.m\_hInstance（当然，也可以使用GetModuleHandle函数来获得DLL模块的句柄），并使用AfxSetResourceHandle函数来将其设置为当前状态状态。最后在调用对话框成功之后再用恢复AfxSetResourceHandle资源句柄，将当前模块状态恢复。  
  
  
  
这样做有些麻烦，但是有一点好处是可以在完成使用资源的任务之后就可以立即恢复资源句柄。而AFX\_MANAGE\_STATE(AfxGetStaticModuleState());的方法只能等函数的作用空间结束之后才恢复资源句柄。由于可执行文件必须重画工具条等原因，因此建议只要有可能就必须恢复资源句柄，否则可能会遇到许多问题。比如说，如果用户移动DLL的对话框，而此时资源句柄仍然为DLL的资源，那么程序就会崩溃。最好的恢复句柄的时机在对话框响应WM\_INITDIALOG消息的时候，因为这时对话框的模板等已经读出了。  
  
  
对于MFC Extension DLL(using shared MFC DLL)类型的MFC DLL，切换当前模块状态的方法与Regular DLL using shared MFC DLL类型的MFC DLL所使用的方法很相似，这里不再举例实现。二者不同的地方如下：  
  
  
  
在MFC扩展DLL中使用AFX\_MANAGE\_STATE(AfxGetStaticModuleState());时，会产生如下错误：  
  
  
  
mfcs42d.lib(dllmodul.obj) : error LNK2005: \_\_pRawDllMain already defined in dllextend.obj  
  
mfcs42d.lib(dllmodul.obj) : error LNK2005: \_DllMain@12 already defined in dllextend.obj  
  
mfcs42d.lib(dllmodul.obj) : error LNK2005: \_\_pRawDllMain already defined in dllextend.obj  
  
  
  
因此在MFC扩展DLL中需要将AFX\_MANAGE\_STATE(AfxGetStaticModuleState());换成AFX\_MANAGE\_STATE(AfxGetAppModuleState());才能正确切换当前模块状态。  
  
  
  
在MFC扩展DLL中使用AfxGetResourceHandle和AfxSetResourceHandle的方法与在Regular DLL using shared MFC DLL类型的MFC DLL中所使用的方法相同。并且，DLL模块的句柄可以通过MFC提供的DlgextentDLL这个结构的hModule成员来获得。即使用AfxSetResourceHandle(DlgextentDLL.hModule);语句。  
  
  
  
当然，对于动态链接到MFC的DLL，也可以在调用该DLL的MFC应用程序中使用AfxGetResourceHandle和AfxSetResourceHandle两个函数来切换当前状态模块。该DLL模块的句柄可以用GetModuleHandle函数来获得。在此不再赘述。

posted @ [2008-07-22 18:14](http://www.cppblog.com/road420/archive/2008/07/22/56877.html) 深邃者 阅读(320) | [评论 (0)](http://www.cppblog.com/road420/archive/2008/07/22/56877.html#FeedBack) | [编辑](http://www.cppblog.com/road420/admin/EditPosts.aspx?postid=56877) [收藏](http://www.cppblog.com/road420/AddToFavorite.aspx?id=56877)

[**让对话框对UPDATE\_COMMAND\_UI生效**](http://www.cppblog.com/road420/archive/2008/07/17/56413.html)

 问题：一般情况下我们用UPDATE\_COMMAND\_UI来修改菜单的状态(enable/disable, check/uncheck, change text)，但这个方法在一个基于对话框上的菜单却没有效果。  
void CTestDlg::OnUpdateFileExit(CCmdUI\* pCmdUI)  
{  
     pCmdUI->Enable(FALSE);  
     pCmdUI->SetCheck(TRUE);  
     pCmdUI->SetRadio(TRUE);  
     pCmdUI->SetText("Close");  
//以上方法在MDI、SDI程序中都能起作用，在对话框中却没有效果,根本没有调用这个函数。  
}  
   
原因分析：当显示一个下拉的菜单的时候，在显示菜单前会发送WM\_INITMENUPOPUP消息。而CFrameWnd::OnInitMenuPopup函数会刷新这个菜单项，同时如果有UPDATE\_COMMAND\_UI响应函数，则调用它。通过它来更新反应每个菜单的外观效果(enabled/disabled, checked/unchecked).  
在一个基于对话框的程序中，因为没有OnInitMenuPopup函数，所以不会调用UPDATE\_COMMAND\_UI响应函数，而是使用了CWnd类的默认处理, 这种处理没有调用UPDATE\_COMMAND\_UI响应函数。

解决方法如下：  
第一步：  
在对话框类的.cpp文件，添加一个ON\_WM\_INITMENUPOPUP入口到消息映射里面  
BEGIN\_MESSAGE\_MAP(CTestDlg, CDialog)  
//}}AFX\_MSG\_MAP  
ON\_WM\_INITMENUPOPUP()  
END\_MESSAGE\_MAP()  
第二步：  
在对话框类的.h文件添加消息函数声明。  
// Generated message map functions  
//{{AFX\_MSG(CDisableDlgMenuDlg)  
afx\_msg void OnInitMenuPopup(CMenu \*pPopupMenu, UINT nIndex,BOOL bSysMenu);  
//}}AFX\_MSG  
DECLARE\_MESSAGE\_MAP()  
第三步：  
在对话框类的.cpp文件添加如下函数代码(大部分代码取自WinFrm.cpp文件的函数CFrameWnd::OnInitMenuPopup):  
void C\*\*\*\*\*\*Dlg::OnInitMenuPopup(CMenu \*pPopupMenu, UINT nIndex,BOOL bSysMenu)  
{  
     ASSERT(pPopupMenu != NULL);  
    // Check the enabled state of various menu items.

    CCmdUI state;  
    state.m\_pMenu = pPopupMenu;  
    ASSERT(state.m\_pOther == NULL);  
    ASSERT(state.m\_pParentMenu == NULL);

    // Determine if menu is popup in top-level menu and set m\_pOther to  
    // it if so (m\_pParentMenu == NULL indicates that it is secondary popup).  
    HMENU hParentMenu;  
    if (AfxGetThreadState()->m\_hTrackingMenu == pPopupMenu->m\_hMenu)  
         state.m\_pParentMenu = pPopupMenu;??? // Parent == child for tracking popup.  
   else if ((hParentMenu = ::GetMenu(m\_hWnd)) != NULL)  
   {  
       CWnd\* pParent = this;  
       // Child windows don't have menus--need to go to the top!  
       if (pParent != NULL &&  
(hParentMenu = ::GetMenu(pParent->m\_hWnd)) != NULL)  
      {  
           int nIndexMax = ::GetMenuItemCount(hParentMenu);  
          for (int nIndex = 0; nIndex < nIndexMax; nIndex++)  
         {  
             if (::GetSubMenu(hParentMenu, nIndex) == pPopupMenu->m\_hMenu)  
             {  
                // When popup is found, m\_pParentMenu is containing menu.  
                state.m\_pParentMenu = CMenu::FromHandle(hParentMenu);  
                break;  
             }  
         }  
    }  
}

  state.m\_nIndexMax = pPopupMenu->GetMenuItemCount();  
  for (state.m\_nIndex = 0; state.m\_nIndex < state.m\_nIndexMax;  
  state.m\_nIndex++)  
  {  
      state.m\_nID = pPopupMenu->GetMenuItemID(state.m\_nIndex);  
      if (state.m\_nID == 0)  
          continue; // Menu separator or invalid cmd - ignore it.

      ASSERT(state.m\_pOther == NULL);  
      ASSERT(state.m\_pMenu != NULL);  
      if (state.m\_nID == (UINT)-1)  
      {  
          // Possibly a popup menu, route to first item of that popup.  
          state.m\_pSubMenu = pPopupMenu->GetSubMenu(state.m\_nIndex);  
          if (state.m\_pSubMenu == NULL ||  
 (state.m\_nID = state.m\_pSubMenu->GetMenuItemID(0)) == 0 ||  
state.m\_nID == (UINT)-1)  
       {  
              continue;    // First item of popup can't be routed to.  
       }  
       state.DoUpdate(this, TRUE);?? // Popups are never auto disabled.  
      }  
      else  
     {  
          // Normal menu item.  
          // Auto enable/disable if frame window has m\_bAutoMenuEnable  
          // set and command is \_not\_ a system command.  
         state.m\_pSubMenu = NULL;  
         state.DoUpdate(this, FALSE);  
      }

      // Adjust for menu deletions and additions.  
      UINT nCount = pPopupMenu->GetMenuItemCount();  
      if (nCount < state.m\_nIndexMax)  
        {  
                   state.m\_nIndex -= (state.m\_nIndexMax - nCount);  
                   while (state.m\_nIndex < nCount &&  
                  pPopupMenu->GetMenuItemID(state.m\_nIndex) == state.m\_nID)  
           {  
            state.m\_nIndex++;  
            }  
       }  
        state.m\_nIndexMax = nCount;  
   }  
}