# 第一节课，参考：[Windows Api学习指南学习笔记059-环境变量1](Windows%20Api学习指南学习笔记059-环境变量1.docx)

# 我们来接着学习，

## HeapAlloc函数的用法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 从堆中分配内存块。 分配的内存不可移动。 语法 C++  DECLSPEC\_ALLOCATOR LPVOID HeapAlloc(  [in] HANDLE hHeap,  [in] DWORD dwFlags,  [in] SIZE\_T dwBytes  ); 参数 [in] hHeap  要从中分配内存的堆的句柄。 此句柄由 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 或 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回。  [in] dwFlags  堆分配选项。 指定这些值中的任何一个都将替代使用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 创建堆时指定的相应值。 此参数可使用以下一个或多个值。  展开表   |  |  | | --- | --- | | **值** | **含义** | | **HEAP\_GENERATE\_EXCEPTIONS**  0x00000004 | 系统将引发异常以指示函数失败（例如内存不足情况），而不是返回 **NULL**。  若要确保为此函数的所有调用生成异常，请在调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 时指定**HEAP\_GENERATE\_EXCEPTIONS**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_GENERATE\_EXCEPTIONS** 。 | | **HEAP\_NO\_SERIALIZE**  0x00000001 | 序列化访问将不用于此分配。  有关详细信息，请参阅“备注”。  若要确保禁用对此函数的所有调用的序列化访问，请在调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 中指定**HEAP\_NO\_SERIALIZE**。 在这种情况下，无需在此函数调用中额外指定 **HEAP\_NO\_SERIALIZE** 。  访问进程的默认堆时，不应指定此值。 系统可能会在应用程序的进程中创建其他线程，例如同时访问进程的默认堆的 CTRL+C 处理程序。 | | **HEAP\_ZERO\_MEMORY**  0x00000008 | 分配的内存将初始化为零。 否则，内存不会初始化为零。 |   [in] dwBytes  要分配的字节数。  如果 *hHeap* 参数指定的堆是“不可增长的”堆， *则 dwBytes* 必须小于 0x7FFF8。 可以通过使用非零值调用 [HeapCreate](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapcreate) 函数来创建不可增长的堆。 返回值 如果函数成功，则返回值是指向已分配内存块的指针。  如果函数失败并且您尚未指定 **HEAP\_GENERATE\_EXCEPTIONS**，则返回值为 **NULL**。  如果函数失败并且已指定 **HEAP\_GENERATE\_EXCEPTIONS**，则函数可能会生成下表中列出的任一异常。 特定例外取决于堆损坏的性质。 有关详细信息，请参阅 [GetExceptionCode](https://learn.microsoft.com/zh-cn/windows/desktop/Debug/getexceptioncode)。  展开表   |  |  | | --- | --- | | **异常代码** | **说明** | | **STATUS\_NO\_MEMORY** | 由于缺少可用内存或堆损坏，分配尝试失败。 | | **STATUS\_ACCESS\_VIOLATION** | 由于堆损坏或函数参数不正确，分配尝试失败。 |   如果函数失败，则它不会调用 [SetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-setlasterror)。 应用程序无法调用 [GetLastError](https://learn.microsoft.com/zh-cn/windows/desktop/api/errhandlingapi/nf-errhandlingapi-getlasterror) 以获取扩展错误信息。 注解 如果 **HeapAlloc** 函数成功，它将分配至少请求的内存量。  若要从进程的默认堆分配内存，请将 **HeapAlloc** 与 [GetProcessHeap](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-getprocessheap) 函数返回的句柄一起使用。  若要释放 **HeapAlloc** 分配的内存块，请使用 [HeapFree](https://learn.microsoft.com/zh-cn/windows/desktop/api/heapapi/nf-heapapi-heapfree) 函数。  **HeapAlloc** 分配的内存不可移动。 **HeapAlloc** 返回的地址在释放或重新分配内存块之前有效;内存块不需要锁定。 由于系统无法压缩专用堆，因此它可能会碎片化。  **HeapAlloc** 返回的内存对齐方式在 WinNT.h 中**MEMORY\_ALLOCATION\_ALIGNMENT**：  C++复制  #if defined(\_WIN64) || defined(\_M\_ALPHA)  #define MEMORY\_ALLOCATION\_ALIGNMENT 16  #else  #define MEMORY\_ALLOCATION\_ALIGNMENT 8  #endif  以各种分配大小分配大量内存的应用程序可以使用 [低碎片堆](https://learn.microsoft.com/zh-cn/windows/desktop/Memory/low-fragmentation-heap) 来减少堆碎片。  当两个或多个线程尝试从同一堆同时分配或释放块时，序列化可确保相互排斥。 序列化的性能成本很小，但每当多个线程从同一个堆分配和释放内存时，必须使用它。 设置 **HEAP\_NO\_SERIALIZE** 值可消除堆上的相互排斥。 如果不进行序列化，使用同一堆句柄的两个或多个线程可能会尝试同时分配或释放内存，这可能会导致堆损坏。 因此，只能在以下情况下安全地使用 **HEAP\_NO\_SERIALIZE** 值：   * 进程只有一个线程。 * 进程有多个线程，但只有一个线程调用特定堆的堆函数。 * 进程具有多个线程，应用程序提供自己的机制，用于对特定堆进行相互排斥。  示例 有关示例，请参阅 [AWE 示例](https://learn.microsoft.com/zh-cn/windows/win32/Memory/awe-example)。 要求  |  |  | | --- | --- | | **标头** | heapapi.h (包括 Windows.h) | | **Library** | Kernel32.lib | | **DLL** | Kernel32.dll |  另请参阅 [堆函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/heap-functions)  [HeapFree](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heapfree)  [HeapReAlloc](https://learn.microsoft.com/zh-cn/windows/win32/api/heapapi/nf-heapapi-heaprealloc)  [内存管理函数](https://learn.microsoft.com/zh-cn/windows/win32/Memory/memory-management-functions)  [VBS enclave 中可用的 Vertdll API](https://learn.microsoft.com/zh-cn/windows/win32/trusted-execution/enclaves-available-in-vertdll) |

# AWE 示例

* 项目
* 2024/08/16
* 3 个参与者

反馈

以下示例程序演示了 [**地址窗口化扩展**](https://learn.microsoft.com/zh-cn/windows/win32/Memory/address-windowing-extensions)。

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  #include <tchar.h>  #define MEMORY\_REQUESTED 1024\*1024 // request a megabyte  BOOL  LoggedSetLockPagesPrivilege ( HANDLE hProcess,  BOOL bEnable);  void \_cdecl main()  {  BOOL bResult; // generic Boolean value  ULONG\_PTR NumberOfPages; // number of pages to request  ULONG\_PTR NumberOfPagesInitial; // initial number of pages requested  ULONG\_PTR \*aPFNs; // page info; holds opaque data  PVOID lpMemReserved; // AWE window  SYSTEM\_INFO sSysInfo; // useful system information  int PFNArraySize; // memory to request for PFN array  GetSystemInfo(&sSysInfo); // fill the system information structure  \_tprintf(\_T("This computer has page size %d.\n"), sSysInfo.dwPageSize);  // Calculate the number of pages of memory to request.  NumberOfPages = MEMORY\_REQUESTED/sSysInfo.dwPageSize;  \_tprintf (\_T("Requesting %d pages of memory.\n"), NumberOfPages);  // Calculate the size of the user PFN array.  PFNArraySize = NumberOfPages \* sizeof (ULONG\_PTR);  \_tprintf (\_T("Requesting a PFN array of %d bytes.\n"), PFNArraySize);  aPFNs = (ULONG\_PTR \*) HeapAlloc(GetProcessHeap(), 0, PFNArraySize);  if (aPFNs == NULL)  {  \_tprintf (\_T("Failed to allocate on heap.\n"));  return;  }  // Enable the privilege.  if( ! LoggedSetLockPagesPrivilege( GetCurrentProcess(), TRUE ) )  {  return;  }  // Allocate the physical memory.  NumberOfPagesInitial = NumberOfPages;  bResult = AllocateUserPhysicalPages( GetCurrentProcess(),  &NumberOfPages,  aPFNs );    if( bResult != TRUE )  {  \_tprintf(\_T("Cannot allocate physical pages (%u)\n"), GetLastError() );  return;  }  if( NumberOfPagesInitial != NumberOfPages )  {  \_tprintf(\_T("Allocated only %p pages.\n"), NumberOfPages );  return;  }  // Reserve the virtual memory.    lpMemReserved = VirtualAlloc( NULL,  MEMORY\_REQUESTED,  MEM\_RESERVE | MEM\_PHYSICAL,  PAGE\_READWRITE );  if( lpMemReserved == NULL )  {  \_tprintf(\_T("Cannot reserve memory.\n"));  return;  }  // Map the physical memory into the window.    bResult = MapUserPhysicalPages( lpMemReserved,  NumberOfPages,  aPFNs );  if( bResult != TRUE )  {  \_tprintf(\_T("MapUserPhysicalPages failed (%u)\n"), GetLastError() );  return;  }  // unmap    bResult = MapUserPhysicalPages( lpMemReserved,  NumberOfPages,  NULL );  if( bResult != TRUE )  {  \_tprintf(\_T("MapUserPhysicalPages failed (%u)\n"), GetLastError() );  return;  }  // Free the physical pages.  bResult = FreeUserPhysicalPages( GetCurrentProcess(),  &NumberOfPages,  aPFNs );  if( bResult != TRUE )  {  \_tprintf(\_T("Cannot free physical pages, error %u.\n"), GetLastError());  return;  }  // Free virtual memory.  bResult = VirtualFree( lpMemReserved,  0,  MEM\_RELEASE );  // Release the aPFNs array.  bResult = HeapFree(GetProcessHeap(), 0, aPFNs);  if( bResult != TRUE )  {  \_tprintf(\_T("Call to HeapFree has failed (%u)\n"), GetLastError() );  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  LoggedSetLockPagesPrivilege: a function to obtain or  release the privilege of locking physical pages.  Inputs:  HANDLE hProcess: Handle for the process for which the  privilege is needed  BOOL bEnable: Enable (TRUE) or disable?  Return value: TRUE indicates success, FALSE failure.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  BOOL  LoggedSetLockPagesPrivilege ( HANDLE hProcess,  BOOL bEnable)  {  struct {  DWORD Count;  LUID\_AND\_ATTRIBUTES Privilege [1];  } Info;  HANDLE Token;  BOOL Result;  // Open the token.  Result = OpenProcessToken ( hProcess,  TOKEN\_ADJUST\_PRIVILEGES,  & Token);  if( Result != TRUE )  {  \_tprintf( \_T("Cannot open process token.\n") );  return FALSE;  }  // Enable or disable?  Info.Count = 1;  if( bEnable )  {  Info.Privilege[0].Attributes = SE\_PRIVILEGE\_ENABLED;  }  else  {  Info.Privilege[0].Attributes = 0;  }  // Get the LUID.  Result = LookupPrivilegeValue ( NULL,  SE\_LOCK\_MEMORY\_NAME,  &(Info.Privilege[0].Luid));  if( Result != TRUE )  {  \_tprintf( \_T("Cannot get privilege for %s.\n"), SE\_LOCK\_MEMORY\_NAME );  return FALSE;  }  // Adjust the privilege.  Result = AdjustTokenPrivileges ( Token, FALSE,  (PTOKEN\_PRIVILEGES) &Info,  0, NULL, NULL);  // Check the result.  if( Result != TRUE )  {  \_tprintf (\_T("Cannot adjust token privileges (%u)\n"), GetLastError() );  return FALSE;  }  else  {  if( GetLastError() != ERROR\_SUCCESS )  {  \_tprintf (\_T("Cannot enable the SE\_LOCK\_MEMORY\_NAME privilege; "));  \_tprintf (\_T("please check the local policy.\n"));  return FALSE;  }  }  CloseHandle( Token );  return TRUE;  } |

# 更多的堆函数，请参考：[Window堆函数](Window堆函数.docx)

# 演练

## 0.虽然这个项目好像没有什么实际用途，但是它可以学习一些内存函数的写法，我们还是练一练。

## 1.新建一个文件夹Lesson60-append-var-value把上一节的项目拷贝过来改名environment-variable2

|  |
| --- |
|  |

## 2.我们在这里添加一个append选项，在这个选项的代码里面读取Path系统环境变量的值然后添加我们的新内容再设置回去，不过只在程序运行期间有效，我们只是学习一下堆函数的用法

|  |
| --- |
|  |

## 3.我们来编写这个追加函数的代码

|  |
| --- |
| DWORD AppendEnvVarValue(LPTSTR szName,LPTSTR szNewValue)  {  DWORD dwReturn,dwErr;  DWORD dwNewValSize;  PVOID szVal;  dwNewValSize = lstrlen(szNewValue)+1;  //动态分配内存用来保存读取到的变量值  szVal = HeapAlloc(GetProcessHeap(),0,BUFSIZE);  if(NULL == szVal)  {  printf("分配内存错误...\n");  return FALSE;  }  //先获取原来的值  dwReturn = GetEnvironmentVariable(szName,(LPTSTR)szVal,BUFSIZE);  if(dwReturn == 0)  {  dwErr = GetLastError();  if(ERROR\_ENVVAR\_NOT\_FOUND == dwErr)  {  printf("找不到环境变量\n");  }  else  {  printf("读取环境变量错误...\n");  }  HeapFree(GetProcessHeap(),0,szVal);//有错误就需要释放动态分配的内存  return FALSE;  }  else if(dwReturn > BUFSIZE)//如果动态分配的内存不够大  {  //需要重新根据我们需要的大小来重新分配内存  szVal = HeapReAlloc(GetProcessHeap(),0,szVal,dwReturn + dwNewValSize);  //重新分配内存后，需要再读取一次，因为上一次失败了嘛  dwReturn = GetEnvironmentVariable(szName,(LPTSTR)szVal,dwReturn);//注意此时的内存需要用大的那个而不是我们定义那个  //再检查一次  if(!dwReturn)  {  printf("读取环境变量错误...\n");  //读取失败也需要释放动态分配的内存  HeapFree(GetProcessHeap(),0,szVal);  return FALSE;  }  }  //读成功了，我们来添加新值  lstrcat((LPTSTR)szVal,TEXT(";"));//需要先加一个分号  lstrcat((LPTSTR)szVal,szNewValue);  //把新值重新设置回去，注意只在程序运行期间有效  if(!SetEnvironmentVariable(szName,(LPTSTR)szVal))  {  printf("设置环境变量错误...\n");  //出错了有需要释放动态分配的内存  HeapFree(GetProcessHeap(),0,szVal);  return FALSE;  }  //显示所以环境变量  showAllEnvVariables();  HeapFree(GetProcessHeap(),0,szVal);//释放我们动态分配的内存  return TRUE;  } |

## 4.重新生成项目，然后来测试一下

|  |
| --- |
|  |
|  |

### 可以看到，追加成功

# 这一节课的学习到此为止，完整代码如下

## append-env-var-value.cpp

|  |
| --- |
| #include<Windows.h>  #include<stdio.h>  #define BUFSIZE 4096  DWORD AppendEnvVarValue(LPTSTR szName,LPTSTR szValue);//函数声明  DWORD SetProcVariable(char \*name,char \*value);  void DeleteOneVariable(char \*key);  void showAllEnvVariables()  {  //获取应该完整的环境变量字符串包含所有的环境变量  PVOID data = GetEnvironmentStrings();  LPTSTR szEnv;  for(szEnv=(LPTSTR)data;\*szEnv;)  {  printf("环境变量字符串：%s\n",szEnv);  while(\*szEnv++);  }  //显示完毕后，需要释放环境变量字符串  FreeEnvironmentStrings(LPCH(data));  }  void GetOneVariableAndShow(char\* key,char\* val ,int size)  {    if(0 == GetEnvironmentVariable(key,val,size))  {  printf("获取变量失败\n");  return ;  }  else  {  printf("%s:%s\n",key,val);    }    }  void GetOneVariableDeleteOrNot(char\* key,char\* val ,int size)  {    if(0 == GetEnvironmentVariable(key,val,size))  {  printf("获取变量失败\n");  return ;  }  else  {  printf("%s:%s\n",key,val);  //使用了一次getchar()后，在使用第二次之前，需要先调用一次getchar让他先读取车，然后再次调用getchar函数并且接收结果  //这里输入没有调用，但是外部已经调用了一次，所以这一次需要清理  getchar();  DeleteOneVariable(key);  }    }  void DeleteOneVariable(char \*key)  {  printf("需要删除这个环境变量吗？(y/n): ");  if('y'==getchar())  {  printf("deleting\n");  SetProcVariable(key,NULL); //设置为NULL就是删除  printf("环境变量已经被删除\n");  showAllEnvVariables();  }  else  {  printf("\n\n");  showAllEnvVariables();  }  }    DWORD SetProcVariable(char \*name,char \*value)  {  if(!SetEnvironmentVariable(name,value))  {  printf("设置环境变量失败：%d\n",GetLastError());  return -1;  }  return 0;  }  int main(int argc,char \*argv[])  {    if(0==lstrcmp("all",argv[1])&& argc==2)//这里all表示显示所有  {  showAllEnvVariables();  }  else if(0==lstrcmp("add",argv[1])&& argc==4)//这里add表示新增，所以需要  {  SetProcVariable(argv[2],argv[3]);  //因为整个进程环境变量在程序结束时就会消失，  //所以我们需要在创建它以后马上调用查看所以进程环境变量的函数来查看  showAllEnvVariables();  printf("需要查找这个环境变量吗？(y/n): ");  if('y'==getchar())  {  printf("getting\n");  char val[256];  memset(val,0,256);  GetOneVariableDeleteOrNot(argv[2],val,256);//获取后可以在里面调用删除函数  }  }  else if(0==lstrcmp("set",argv[1])&& argc==4)//这里set表示修改，所以需要  {  SetProcVariable(argv[2],argv[3]);  }  else if(0==lstrcmp("get",argv[1])&& argc==3)//这里set表示修改，所以需要  {  printf("getting\n");  char val[256];  memset(val,0,256);  GetOneVariableAndShow(argv[2],val,256);//获取后可以在里面调用删除函数  }  else if(0==lstrcmp("append",argv[1])&& argc==4)//这里set表示修改，所以需要  {  printf("追加前\n");  showAllEnvVariables();  if(!AppendEnvVarValue(argv[2],argv[3]))  {  printf("追加环境变量：%s的值错误\n",argv[3]);  return 2;  }  printf("追加后\n");  showAllEnvVariables();  }  system("pause");  return 0;  }  DWORD AppendEnvVarValue(LPTSTR szName,LPTSTR szNewValue)  {  DWORD dwReturn,dwErr;  DWORD dwNewValSize;  PVOID szVal;  dwNewValSize = lstrlen(szNewValue)+1;  //动态分配内存用来保存读取到的变量值  szVal = HeapAlloc(GetProcessHeap(),0,BUFSIZE);  if(NULL == szVal)  {  printf("分配内存错误...\n");  return FALSE;  }  //先获取原来的值  dwReturn = GetEnvironmentVariable(szName,(LPTSTR)szVal,BUFSIZE);  if(dwReturn == 0)  {  dwErr = GetLastError();  if(ERROR\_ENVVAR\_NOT\_FOUND == dwErr)  {  printf("找不到环境变量\n");  }  else  {  printf("读取环境变量错误...\n");  }  HeapFree(GetProcessHeap(),0,szVal);//有错误就需要释放动态分配的内存  return FALSE;  }  else if(dwReturn > BUFSIZE)//如果动态分配的内存不够大  {  //需要重新根据我们需要的大小来重新分配内存  szVal = HeapReAlloc(GetProcessHeap(),0,szVal,dwReturn + dwNewValSize);  //重新分配内存后，需要再读取一次，因为上一次失败了嘛  dwReturn = GetEnvironmentVariable(szName,(LPTSTR)szVal,dwReturn);//注意此时的内存需要用大的那个而不是我们定义那个  //再检查一次  if(!dwReturn)  {  printf("读取环境变量错误...\n");  //读取失败也需要释放动态分配的内存  HeapFree(GetProcessHeap(),0,szVal);  return FALSE;  }  }  //读成功了，我们来添加新值  lstrcat((LPTSTR)szVal,TEXT(";"));//需要先加一个分号  lstrcat((LPTSTR)szVal,szNewValue);  //把新值重新设置回去，注意只在程序运行期间有效  if(!SetEnvironmentVariable(szName,(LPTSTR)szVal))  {  printf("设置环境变量错误...\n");  //出错了有需要释放动态分配的内存  HeapFree(GetProcessHeap(),0,szVal);  return FALSE;  }  //显示所以环境变量  //showAllEnvVariables();  HeapFree(GetProcessHeap(),0,szVal);//释放我们动态分配的内存  return TRUE;  } |