Windows 提供了以下三种机制来对内存进行操控  
虚拟内存：最适合用来管理大量对象数组或者大型数据结构  
内存映射文件：最适合用来管理大型数据流(通常是文件)，以及在同一机器上运行的多个进程之间的共享数据。  
堆：最适合用来管理大量的小型对象。

最后一种对内存进行操控的方法是堆。**堆适合分配大量小型数据，是管理链表和树最佳方式。优点是能让我们不必理会分配粒度和页面边界。缺点是分配和释放的内存块速度比其他方式慢，无法对物理存储器的调拨和撤销直接控制。**

从内部来讲，堆栈是保留的地址空间的一个区域。开始时，保留区域中的大多数页面没有被提交物理存储器。当从堆栈中进行越来越多的内存分配时，堆 栈管理器将把更多的物理存储器提交给堆栈。物理存储器总是从系统的页文 件中分配的，当释放堆栈中的内存块时，堆栈管理器将收回这些物理存储器。

**一、进程的默认堆**

进程初始化的时候,系统会在进程的地址空间中创建一个堆。这个堆被称为**进程的默认堆**（default heap）,**在默认的情况下,这个堆的地址空间区域的大小是1MB**,但是,系统可以增大进程的默认堆,使它大于1MB,我们也可以在创建应用程序的时候用/HEAP链接器开关来改变默认的区域大小。由于动态链接库(DLL)没有与之关联的堆,因此在创建DLL的时候不应该使用/HEAP开关。

**默认堆栈是在进程开始执行之前创建的，并且在进程终止运行时自动被撤消。**我们不能撤消进程的默认堆栈。每个堆栈均用它自己的堆栈句柄来标识，用于分配和释放堆栈中的内存块的所有堆栈函数都需要这个堆栈句柄作为其参数。 **可以通过调用 GetProcessHeap 函数获取你的进程默认堆栈的句柄**：

HANDLE GetProcessHeap();

**二、为什么要创建额外的堆**

除了进程的默认堆,我们可以在进程的地址空间中创建额外的堆。由于以下原因,我们可能希望在应用程序中创建额外的堆:

* 对组件进行保护（担心默认堆中的某一组件如链表存在缺陷，会导致覆盖其他的内容，所以要创建额外的堆）
* 更有效的内存管理（防止默认堆中数据碎片化）
* 局部访问（将需要同时访问的对象分配在相邻的内存地址，降低内存和磁盘交换的可能性）
* 避免线程同步的开销（默认情况下堆的访问是依次进行的，但系统有额外代码保证堆访问的线程安全性。如果创建额外的堆只有一个线程能访问，就不必执行额外的代码，但是要小心这样我们就承担起了堆的线程安全性的责任）
* 快速释放（把一些数据存入专门的堆中，如果要释放数据，直接释放整个堆而不必显示的释放堆中的每个数据的内存块）

**三、如何创建额外的堆**

此函数创建一个heap对象，该对象是调用进程的私有对象。**这将保留进程虚拟地址空间的一个连续块，并为该块的指定初始部分分配物理存储。**

HANDLE **HeapCreate**(  
DWORD flOptions,     // 对堆的操作该如何进行  
DWORD dwInitialSize,     //   
DWORD dwMaximumSize   
);

第1个参数dwOptions：表示对堆的操作该如何进行。可以指定0，HEAP\_NO\_SERIALIZE, HEAP\_GENERATE\_EXCEPTIONS, HEAP\_CREATE\_ENABL\_EXECUTE或这些标志的组合。

* HEAP\_NO\_SERIALIZE：不要贸然使用，没有使用的时候，只允许一个线程独占堆及其链表的访问，如果使用必须保证满足以下条件之一：  
      (1)进程中只有一个线程  
      (2)进程中只有一个线程会访问该堆  
      (3)进程中使用了线程同步的方式控制线程访问该堆。
* HEAP\_GENERATE\_EXCEPTIONS：标志告诉系统,每当在堆中分配或重新分配内存块失败的时候,抛出一个异常。
* HEAP\_CREATE\_ENABL\_EXECUTE：想在堆中存放可执行代码，如果不设置但执行堆中的代码时，会抛出异常 EXCEPTION\_ACCESS\_VIOLATION。

默认情况下,对堆的访问会依次进行,使多个线程可以从同一个堆中分配和释放内存块,同时也不会存在堆数据被破坏的危险。**当任何程序试图从堆中分配一块内存的时候,HeapAlloc 函数必须执行以下操作。**

(1)遍历已分配内存的链表和闲置内存的链表。  
(2)找到一块足够大的闲置内存块。  
(3)分配一块新的内存,也就是将刚找到的闲置内存块标记为已分配。  
(4)将新分配的内存块添加到已分配内存的链表中。

第2个参数dwInitialSize：表示一开始要调拨给堆的字节数。如果需要,HeapCreate会把这个值向上取整到CPU页面大小的整数倍。

第3个参数dwMaximumSize：表示堆所能增长到的最大大小(即系统为堆所预订的地址空间的最大大小)。如果dwMaximumSize大于0,那么创建的堆会有一个最大大小。这时,如果试图分配的内存块可能导致堆超过最大大小,分配操作会失败。如果dwMaximumSize为0,那么创建的堆将是可增长的,它没有一个指定的上限。从堆中分配内存会使堆不断增长,直到用尽所有的物理存储器为止。

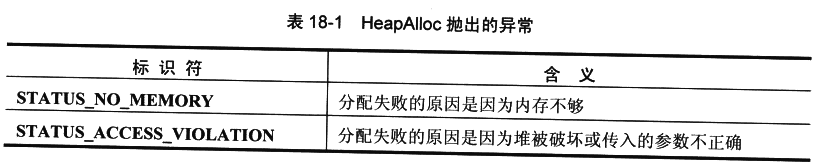
**四、从堆中分配内存块**

**这个函数从堆中分配一个内存块。分配的内存是不可移动的。**

LPVOID **HeapAlloc**(  
HANDLE hHeap,     // 堆的句柄  
DWORD dwFlags,       
DWORD dwBytes     // 指定要分配的字节数。  
);

第2参数dwFlags：HEAP\_GENERATE\_EXCEPTIONS、HEAP\_NO\_SERIALIZE、HEAP\_ZERO\_MEMORY

* HEAP\_ZERO\_MEMORY：让HeapAlloc在返回之前把内存块的内容都清零。
* HEAP\_GENERATE\_EXCEPTIONS：用来告诉系统,如果堆中没有足够的内存来满足内存分配请求, HeapAlloc就应该抛出异常。在调用HeapCreate来创建堆的时候,也可以使用HEAP\_GENERATE\_EXCEPTIONS标志,这告诉系统在无法分配内存块的时候抛出异常。如果在调用HeapCreate的时候指定了这个标志,在调用HeapAlloc的时候就不需要再指定了。

[](https://img2022.cnblogs.com/blog/2743864/202211/2743864-20221123112043863-1258096978.png)

   没有指定该标志的时候，HeapAlloc 无法分配内存会返回NULL。

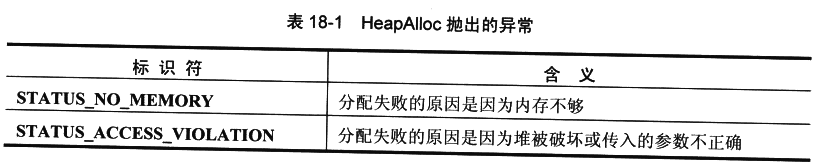
* HEAP\_NO\_SERIALIZE：用来强制系统不要把这次HeapAlloc调用与其他线程对同一个堆的访问依次排列起来。在使用这个标志的时候应该极其小心,因为如果其他线程也在同一时刻对堆进行操作,将破坏堆的完整性。在从进程的默认堆中分配内存的时候,绝对不要使用这个标志,否则可能会破坏数据,因为进程中的其他线程可能会在同时刻访问堆。

**五、调整堆中指定内存块大小**

HeapReAlloc函数从堆中重新分配一块内存。这个函数允许您调整内存块的大小和更改其他内存块属性。分配的内存是不可移动的。  
LPVOID HeapReAlloc(  
HANDLE hHeap,   // 堆的句柄  
DWORD dwFlags,  // heap reallocation options  
LPVOID lpMem,   // 想要调整大小的内存块的当前地址  
SIZE\_T dwBytes  // 内存块的新大小  
);

第2参数dwFlags：HEAP\_GENERATE\_EXCEPTIONS、HEAP\_NO\_SERIALIZE、HEAP\_REALLOC\_IN\_PLACE\_ONLY、HEAP\_ZERO\_MEMORY。

* HEAP\_GENERATE\_EXCEPTIONS：用来告诉系统,如果堆中没有足够的内存来满足内存分配请求, HeapAlloc就应该抛出异常。在调用HeapCreate来创建堆的时候,也可以使用HEAP\_GENERATE\_EXCEPTIONS标志,这告诉系统在无法分配内存块的时候抛出异常。如果在调用HeapCreate的时候指定了这个标志,在调用HeapAlloc的时候就不需要再指定了。

[](https://img2022.cnblogs.com/blog/2743864/202211/2743864-20221123112136776-434754925.png)

* HEAP\_NO\_SERIALIZE：用来强制系统不要把这次HeapAlloc调用与其他线程对同一个堆的访问依次排列起来。在使用这个标志的时候应该极其小心,因为如果其他线程也在同一时刻对堆进行操作,将破坏堆的完整性。在从进程的默认堆中分配内存的时候,绝对不要使用这个标志,否则可能会破坏数据,因为进程中的其他线程可能会在同时刻访问堆。
* HEAP\_ZERO\_MEMORY：只有增大内存块大小的时候才有用，则该值指定超出原始大小的额外内存区域将初始化为0。内存块的内容不受其原始大小的影响。减小内存块的大小的时候，这个标志不起任何作用。
* HEAP\_REALLOC\_IN\_PLACE\_ONLY：在增大内存块的时候, HeapReAlloc可能会在堆内部移动内存块,而HEAP\_REALLOC\_IN\_PLACE\_ONLY标志用来告诉HeapReAlloc不要移动内存块。如果HeapReAlloc能够在不移动内存块的前提下让它增大,它将返回内存块原来的地址。另一方面,如果HeapReAlloc必须移动内存块的内容,函数将返回一个新地址,指向一块更大的内存块。如果要把内存块减小,那么HeapReAlloc会返回内存块原来的地址。如果一个内存块是链表或树的一部分,则需要指定HEAP\_REALLOC\_IN\_PLACE\_ONLY。在这种情况下,链表或树中的其他节点可能包含指向当前节点的指针,把节点移动到堆中其他的地方会破坏链表或树的完整性。

**六、获得内存块大小**

HeapSize 函数检索由 HeapAlloc 或 HeapReAlloc 函数从堆中分配的内存块的实际大小。

DWORD HeapSize(  
HANDLE hHeap,  // handle to heap  
DWORD dwFlags, // heap size options  
LPCVOID lpMem  // pointer to memory  
);

参数hHeap用来标识堆,参数pvMem表示内存块的地址。参数fdwFlags可以是0或者HEAP\_NO\_SERIALIZE。

**七、释放内存块**

HeapFree函数释放由HeapAlloc或HeapReAlloc函数从堆中分配的内存块。  
BOOL HeapFree(  
  HANDLE hHeap,  // handle to heap  
  DWORD dwFlags, // heap free options  
  LPVOID lpMem   // pointer to memory  
  );

HeapFree会释放内存块,如果操作成功,它会返回TRUE,参数fdwFlags可以是0或者HEAP\_NO\_SERIALIZE,调用这个函数可能会使堆管理器撤销一些已经调拨的物理存储器,但这并不是一定的。参数pvMem表示内存块的地址。

**八、销毁堆**

如果应用程序不再需要自己创建的堆,则可以调用HeapDestroy来销毁它:

BOOL HeapDestroy (HANDLE hHeap);

调用HeapDestroy会释放堆中包含的所有内存块,同时系统会收回堆所占用的物理存储器和地址空间区域。如果函数调用成功, HeapDestroy将返回TRUE,如果我们不在进程终止之前显式地销毁堆,那么系统会替我们销毁。但只有在进程终止的时候,系统才会这样做。如果一个线程创建了堆,那么在这个线程终止的时候,堆是不会被销毁的。

在进程完全终止之前,系统不允许销毁进程的默认堆。如果把进程的默认堆的句柄传给HeapDestroy,系统将直接忽略该调用并返回FALSE。

**九、其他堆函数**

**由于在进程自己的地址空间中可以有多个堆，GetProcessHeaps 可以让我们都得到这些堆的句柄。**

DWORD **GetProcessHeaps**(DWORD dwNumHeaps,PHANDLE phHeaps);

HANDLE hHeaps[25];  
DWORD dwHeaps = GetProcessHeaps(25,hHeaps);  
if(dwHeaps >25)  
{  
  // 有更多的堆，需要分配更大的 HANDLE 数组。  
}  
else  
{  
  // 验证堆的完整性  
}

**验证堆的完整性需要用到函数 HeapValidate，会遍历堆的各个内存块，确保没有任何一块内存被破坏。**

// 验证堆未被破坏  
BOOL WINAPI **HeapValidate**(  
  \_In\_     HANDLE  hHeap,  
  // HEAP\_NO\_SERIALIZE 没有多线程互斥  
  \_In\_     DWORD   dwFlags,  
  // NULL验证整个堆  
  // 非NULL，验证堆内此块  
  \_In\_opt\_ LPCVOID lpMem  
);

**返回最大可用堆内块大小**  
SIZE\_T WINAPI HeapCompact(  
  \_In\_ HANDLE hHeap,  
  \_In\_ DWORD  dwFlags  
);

**多线程保护堆，在堆访问函数内部调用此两个函数**  
**锁定堆，堆被锁定，进程内其它线程不可用**  
BOOL WINAPI HeapLock(  
  \_In\_ HANDLE hHeap  
);

**解除锁定**  
BOOL WINAPI HeapUnlock(  
  \_In\_ HANDLE hHeap  
);

**枚举堆内的块**  
// 初始：lpEntry的lpData为NULL，每枚举成功一次，返回NULL。lpEntry被填充枚举的块信息。继续用此lpEntry执行调用，接着被填充下一块信息。枚举结束返回FALSE。  
// 枚举开始，一般用HeapLock锁定堆。  
// 枚举结束，一般用HeapUnlock解除锁定。  
BOOL WINAPI HeapWalk(  
  \_In\_    HANDLE               hHeap,  
  \_Inout\_ LPPROCESS\_HEAP\_ENTRY lpEntry  
);