

## 第18章 堆 栈

对内存进行操作的第三个机制是使用堆栈。堆栈可以用来分配许多较小的数据块。例如，若要对链接表和链接树进行管理，最好的方法是使用堆栈，而不是第15章介绍的虚拟内存操作方法或第17章介绍的内存映射文件操作方法。堆栈的优点是，可以不考虑分配粒度和页面边界之类的问题，集中精力处理手头的任务。堆栈的缺点是，分配和释放内存块的速度比其他机制要慢，并且无法直接控制物理存储器的提交和回收。

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。

Microsoft并没有以文档的形式来规定堆栈释放和收回存储器时应该遵循的具体规则，Windows 98与Windows 2000的规则是不同的。可以这样说，Windows 98更加注重内存的使用，因此只要可能，它就收回堆栈。Windows 2000更加注重速度，因此它往往较长时间占用物理存储器，只有在一段时间后页面不再使用时，才将它返回给页文件。Microsoft常常进行适应性测试并运行各种不同的条件，以确定在大部分时间内最适合的规则。随着使用这些规则的应用程序和硬件的变更，这些规则也会有所变化。如果了解这些规则对你的应用程序非常关键，那么请不要使用堆栈。相反，可以使用虚拟内存函数（即VirtualAlloc和VirtualFree），这样，就能够控制这些规则。

### 18.1 进程的默认堆栈

当进程初始化时，系统在进程的地址空间中创建一个堆栈。该堆栈称为进程的默认堆栈。按照默认设置，该堆栈的地址空间区域的大小是1 MB。但是，系统可以扩大进程的默认堆栈，使它大于其默认值。当创建应用程序时，可以使用/HEAP链接开关，改变堆栈的1MB默认区域大小。由于DLL没有与其相关的堆栈，所以当链接DLL时，不应该使用/HEAP链接开关。/HEAP链接开关的句法如下：

```
/HEAP:reserve[,commit]
```

许多Windows函数要求进程使用其默认堆栈。例如，Windows 2000的核心函数均使用Unicode字符和字符串执行它们的全部操作。如果调用Windows函数的ANSI版本，那么该ANSI版本必须将ANSI字符串转换成Unicode字符串，然后调用同一个函数的Unicode版本。为了进行字符串的转换，ANSI函数必须分配一个内存块，以便放置Unicode版本的字符串。该内存块是从你的进程的默认堆栈中分配的。Windows的其他许多函数需要使用一些临时内存块，这些内存块是从进程的默认堆栈中分配的。另外，老的16位Windows函数LocalAlloc和GlobalAlloc也是从进程的默认堆栈中进行它们的内存分配的。

由于进程的默认堆栈可供许多Windows函数使用，你的应用程序有许多线程同时调用各种Windows函数，因此对默认堆栈的访问是顺序进行的。换句话说，系统必须保证在规定的时间内，每次只有一个线程能够分配和释放默认堆栈中的内存块。如果两个线程试图同时分配默认堆栈中的内存块，那么只有一个线程能够分配内存块，另一个线程必须等待第一个线程的内存

块分配之后，才能分配它的内存块。一旦第一个线程的内存块分配完，堆栈函数将允许第二个线程分配内存块。这种顺序访问方法对速度有一定的影响。如果你的应用程序只有一个线程，并且你想要以最快的速度访问堆栈，那么应该创建你自己的独立的堆栈，不要使用进程的默认堆栈。不幸的是，你无法告诉 Windows 函数不要使用默认堆栈，因此，它们对堆栈的访问总是顺序进行的。

单个进程可以同时拥有若干个堆栈。这些堆栈可以在进程的寿命期中创建和撤消。但是，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。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();
```

## 18.2 为什么要创建辅助堆栈

除了进程的默认堆栈外，可以在进程的地址空间中创建一些辅助堆栈。由于下列原因，你可能想要在自己的应用程序中创建一些辅助堆栈：

- 保护组件。
- 更加有效地进行内存管理。
- 进行本地访问。
- 减少线程同步的开销。
- 迅速释放。

下面让我们来详细说明每个原因。

### 18.2.1 保护组件

假如你的应用程序需要保护两个组件，一个是节点结构的链接表，一个是 `BRANCH` 结构的二进制树。你有两个源代码文件，一个是 `LnkLst.cpp`，它包含负责处理 `NODE` 链接表的各个函数，另一个文件是 `BinTree.cpp`，它包含负责处理分支的二进制树的各个函数。

如果节点和分支一道存储在单个堆栈中，那么这个组合堆栈将类似图 18-1 所示的样子。

现在假设链接表代码中有一个错误，它使节点 1 后面的 8 个字节不小心被改写了，从而导致分支 3 中的数据被破坏。当 `BinTree.cpp` 文件中的代码后来试图遍历二进制树时，它将无法进行这项操作，因为它的内存已经被破坏。当然，这使你认为二进制树代码中存在一个错误，而实际上错误是在链接表代码中。由于不同类型的对象混合放在单个堆栈中，因此跟踪和确定错误将变得非常困难。

通过创建两个独立的堆栈，一个堆栈用于存放节点，另一个堆栈用于存放分支，就能够确定你的问题。你的链接表代码中的一个小错误不会破坏你的二进制树的完整性。反过来，二进制树中的小错误也不会影响链接表代码中的数据完整性。但是，你的代码中的错误仍然可能导致对堆栈进行杂乱的内存写操作，不过出现这种情况的可能性很小。

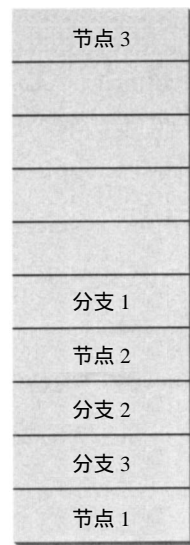


图 18-1 将节点和分支存放在一起的单个堆栈

### 18.2.2 更有效的内存管理

通过在堆栈中分配同样大小的对象，就可以更加有效地管理堆栈。例如，假设每个节点结构需要24字节，每个分支结构需要32字节。所有这些对象均从单个堆栈中分配。图18-2显示了单个堆栈中已经分配的若干个节点和分支对象占满了这个堆栈。如果节点2和节点4被释放，堆栈中的内存将变成许多碎片。这时，如果试图分配分支结构，那么尽管分支只需要32个字节，而实际上可以使用的有48个字节，但是分配仍将失败。

如果每个堆栈只包含大小相同的对象，那么释放一个对象后，另一个对象就可以恰好放入被释放的对象空间中。

### 18.2.3 进行本地访问

每当系统必须在RAM与系统的页文件之间进行RAM页面的交换时，系统的运行性能就会受到很大的影响。如果经常访问局限于一个小范围地址的内存，那么系统就不太可能需要在RAM与磁盘之间进行页面的交换。

所以，在设计应用程序的时候，如果有些数据将被同时访问，那么最好把它们分配在互相靠近的位置上。让我们回到链接表和二进制树的例子上来，遍历链接表与遍历二进制树之间并无什么关系。如果将所有的节点放在一起（放在一个堆栈中），就可以使这些节点位于相邻的页面上。实际上，若干个节点很可能恰好放入单个物理内存页面上。遍历链接表将不需要CPU为了访问每个节点而引用若干不同的内存页面。

如果将节点和分支分配在单个页面上，那么节点就不一定会互相靠在一起。在最坏的情况下，每个内存页面上可能只有一个节点，而其余的每个页面则由分支占用。在这种情况下，遍历链接表将可能导致每个节点的页面出错，从而使进程运行得极慢。

### 18.2.4 减少线程同步的开销

正如下面就要介绍的那样，按照默认设置，堆栈是顺序运行的，这样，如果多个线程试图同时访问堆栈，就不会使数据受到破坏。但是，堆栈函数必须执行额外的代码，以保证堆栈对线程的安全性。如果要进行大量的堆栈分配操作，那么执行这些额外的代码会增加很大的负担，从而降低你的应用程序的运行性能。当你创建一个新堆栈时，可以告诉系统，只有一个线程将访问该堆栈，因此额外的代码将不执行。但是要注意，现在你要负责保证堆栈对线程的安全性。系统将不对此负责。

### 18.2.5 迅速释放堆栈

最后要说明的是，将专用堆栈用于某些数据结构后，就可以释放整个堆栈，而不必显式释放堆栈中的每个内存块。例如，当Windows Explorer遍历硬盘驱动器的目录层次结构时，它必须在内存中建立一个树状结构。如果你告诉Windows Explorer刷新它的显示器，它只需要撤消包含这个树状结构的堆栈并且重新运行即可（当然，假定它将专用堆栈用于存放目录树信息）。对于许多应用程序来说，这是非常方便的，并且它们也能更快地运行。



图18-2 变成碎片的单个堆栈包含若干个节点和分支对象

### 18.3 如何创建辅助堆栈

你可以在进程中创建辅助堆栈，方法是让线程调用 HeapCreate 函数：

```
HANDLE HeapCreate(  
    DWORD fdwOptions,  
    SIZE_T dwInitialSize,  
    SIZE_T dwMaximumSize);
```

第一个参数 fdwOptions 用于修改如何在堆栈上执行各种操作。你可以设定 0、HEAP\_NO\_SERIALIZE、HEAP\_GENERATE\_EXCEPTIONS 或者是这两个标志的组合。

按照默认设置，堆栈将顺序访问它自己，这样，多个线程就能够分配和释放堆栈中的内存块而不至于破坏堆栈。当试图从堆栈分配一个内存块时，HeapAlloc 函数（下面将要介绍）必须执行下列操作：

- 1) 遍历分配的和释放的内存块的链接表。
- 2) 寻找一个空闲内存块的地址。
- 3) 通过将空闲内存块标记为“已分配”分配新内存块。
- 4) 将新内存块添加给内存块链接表。

下面这个例子说明为什么应该避免使用 HEAP\_NO\_SERIALIZE 标志。假定有两个线程试图同时从同一个堆栈中分配内存块。线程 1 执行上面的第一步和第二步，获得了空闲内存块的地址。但是，在该线程可以执行第三步之前，它的运行被线程 2 抢占，线程 2 得到一个机会来执行上面的第一步和第二步。由于线程 1 尚未执行第三步，因此线程 2 发现了同一个空闲内存块的地址。

由于这两个线程都发现了堆栈中它们认为是空闲的内存块，因此线程 1 更新了链接表，给新内存块做上了“已分配”的标记。然后线程 2 也更新了链接表，给同一个内存块做上了“已分配”标记。到现在为止，两个线程都没有发现问题，但是两个线程得到的是完全相同的内存块的地址。

这种类型的错误是很难跟踪的，因为它不会立即表现出来。相反，这个错误会在后台等待着，直到很不适合的时候才显示出来。可能出现的问题是：

- 内存块的链接表已经被破坏。在试图分配或释放内存块之前，这个问题不会被发现。
- 两个线程共享同一个内存块。线程 1 和线程 2 会将信息写入同一个内存块。当线程 1 查看该内存块的内容时，它将无法识别线程 2 提供的的数据。
- 一个线程可能继续使用该内存块并且将它释放，导致另一个线程改写未分配的内存。这将破坏该堆栈。

解决这个问题的办法是让单个线程独占对堆栈和它的链接表的访问权，直到该线程执行了对堆栈的全部必要的操作。如果不使用 HEAP\_NO\_SERIALIZE 标志，就能够达到这个目的。只有当你的进程具备下面的一个或多个条件时，才能安全地使用 HEAP\_NO\_SERIALIZE 标志：

- 你的进程只使用一个线程。
- 你的进程使用多个线程，但是只有单个线程访问该堆栈。
- 你的进程使用多个线程，但是它设法使用其他形式的互斥机制，如关键代码段、互斥对象和信标（第 8、9 章中介绍），以便设法自己访问堆栈。

如果对是否可以使用 HEAP\_NO\_SERIALIZE 标志没有把握，那么请不要使用它。如果不使用该标志，每当调用堆栈函数时，线程的运行速度会受到一定的影响，但是不会破坏你的堆栈及其数据。



另一个标志HEAP\_GENERATE\_EXCEPTIONS，会在分配或重新分配堆栈中的内存块的尝试失败时，导致系统引发一个异常条件。所谓异常条件，只不过是系统使用的另一种方法，以便将已经出现错误的情况通知你的应用程序。有时在设计应用程序时让它查看异常条件比查看返回值要更加容易些。异常条件将在第23、24和25章中介绍。

HeapCreate的第二个参数dwInitialSize用于指明最初提交给堆栈的字节数。如果必要的话，HeapCreate函数会将这个值圆整为CPU页面大小的倍数。最后一个参数dwMaximumSize用于指明堆栈能够扩展到的最大值（即系统能够为堆栈保留的地址空间的最大数量）。如果dwMaximumSize大于0，那么你创建的堆栈将具有最大值。如果尝试分配的内存块会导致堆栈超过其最大值，那么这种尝试就会失败。

如果dwMaximumSize的值是0，那么可以创建一个能够扩展的堆栈，它没有内在的限制。从堆栈中分配内存块只需要使堆栈不断扩展，直到物理存储器用完为止。如果堆栈创建成功，HeapCreate函数返回一个句柄以标识新堆栈。该句柄可以被其他堆栈函数使用。

### 18.3.1 从堆栈中分配内存块

若要从堆栈中分配内存块，只需要调用HeapAlloc函数：

```
PVOID HeapAlloc(  
    HANDLE hHeap,  
    DWORD fdwFlags,  
    SIZE_T dwBytes);
```

第一个参数hHeap用于标识分配的内存块来自的堆栈的句柄。dwBytes参数用于设定从堆栈中分配的内存块的字节数。参数fdwFlags用于设定影响分配的各个标志。目前支持的标志只有3个，即HEAP\_ZERO\_MEMORY、HEAP\_GENERATE\_EXCEPTIONS和HEAP\_NO\_SERIALIZE。

HEAP\_ZERO\_MEMORY标志的作用应该是非常清楚的。该标志使得HeapAlloc在返回前用0来填写内存块的内容。第二个标志HEAP\_GENERATE\_EXCEPTIONS用于在堆栈中没有足够的内存来满足需求时使HeapAlloc函数引发一个软件异常条件。当用HeapCreate函数创建堆栈时，可以设定HEAP\_GENERATE\_EXCEPTIONS标志，它告诉堆栈，当不能分配内存块时，就应该引发一个异常条件。如果在调用HeapCreate函数时设定了这个标志，那么当调用HeapAlloc函数时，就不需要设定该标志。另外，你可能想要不使用该标志来创建堆栈。在这种情况下，为HeapAlloc函数设定该标志只会影响对HeapAlloc函数的一次调用，并不是每次调用都会受到影响。

如果HeapAlloc运行失败，引发一个异常条件，那么这个异常条件将是表18-1中的两个异常条件之一。

表18-1 异常条件

标 志	含 义
STATUS_NO_MEMORY	由于内存不够，分配内存块的尝试失败
STATUS_ACCESS_VIOLATION	由于堆栈被破坏，或者函数的参数不正确，分配内存块的尝试失败

如果内存块已经成功地分配，HeapAlloc返回内存块的地址。如果内存不能分配并且没有设定HEAP\_GENERATE\_EXCEPTIONS标志，那么HeapAlloc函数返回NULL。

最后一个标志HEAP\_NO\_SERIALIZE可以用来强制对HeapAlloc函数的调用与访问同一个堆栈的其他线程不按照顺序进行。在使用这个标志时应该格外小心，因为如果其他线程在同一

时间使用该堆栈，那么堆栈就会被破坏。当从你的进程的默认堆栈中分配内存块时，决不要使用这个标志，因为数据可能被破坏，你的进程中的其他线程可能在同一时间访问默认堆栈。

Windows 98 如果调用HeapAlloc函数并且要求分配大于256 MB的内存块，Windows 98 就将它看成是一个错误，函数的调用将失败。注意，在这种情况下，该函数总是返回NULL，并且不会引发异常条件，即使你在创建堆栈或者试图分配内存块时使用HEAP\_GENERATE\_EXCEPTIONS标志，也不会引发异常条件。

注意 当你分配较大的内存块（大约1 MB或者更大）时，最好使用VirtualAlloc函数，应该避免使用堆栈函数。

### 18.3.2 改变内存块的大小

常常需要改变内存块的大小。有些应用程序开始时分配的内存块比较大，然后，当所有数据放入内存块后，再缩小内存块的大小。有些应用程序开始时分配的内存块比较小，后来需要将更多的数据拷贝到内存块中去时，再设法扩大它的大小。如果要改变内存块的大小，可以调用HeapReAlloc函数：

```
PVOID HeapReAlloc(  
    HANDLE hHeap,  
    DWORD fdwFlags,  
    PVOID pvMem,  
    SIZE_T dwBytes);
```

与其他情况一样，hHeap参数用于指明包含你要改变其大小的内存块的堆栈。fdwFlags参数用于设定改变内存块大小时HeapReAlloc函数应该使用的标志。可以使用的标志只有下面4个，即HEAP\_GENERATE\_EXCEPTIONS、HEAP\_NO\_SERIALIZE、HEAP\_ZERO\_MEMORY和HEAP\_REALLOC\_IN\_PLACE\_ONLY。

前面两个标志在用于HeapAlloc时，其作用相同。HEAP\_ZERO\_MEMORY标志只有在你扩大内存块时才使用。在这种情况下，内存块中增加的字节将被置0。如果内存块已经被缩小，那么该标志不起作用。

HEAP\_REALLOC\_IN\_PLACE\_ONLY标志告诉HeapReAlloc函数，它不能移动堆栈中的内存块。如果内存块在增大，HeapReAlloc函数可能试图移动内存块。如果HeapReAlloc能够扩大内存块而不移动它，那么它将会这样做并且返回内存块的原始地址。另外，如果HeapReAlloc必须移动内存块的内容，则返回新的较大内存块的地址。如果内存块被缩小，HeapReAlloc将返回内存块的原始地址。如果内存块是链接表或二进制树的组成部分，那么可以设定HEAP\_REALLOC\_IN\_PLACE\_ONLY标志。在这种情况下，链接表或二进制树中的其他节点可能拥有该节点的指针，改变堆栈中的节点位置会破坏链接表的完整性。

其余的两个参数pvMem和dwBytes用于设定你要改变其大小的内存块的地址和内存块的新的大小（以字节为计量单位）。HeapReAlloc既可以返回新的改变了大小的内存块的地址，也可以在内存块不能改变大小时返回NULL。

### 18.3.3 了解内存块的大小

当内存块分配后，可以调用HeapSize函数来检索内存块的实际大小：

```
SIZE_T HeapSize(  
    HANDLE hHeap,
```

```
DWORD fdwFlags,  
LPCVOID pvMem);
```

参数hHeap用于标识堆栈，参数pvMem用于指明内存块的地址。参数fdwFlags既可以是0，也可以是HEAP\_NO\_SERIALIZE。

#### 18.3.4 释放内存块

当不再需要内存块时，可以调用HeapFree函数将它释放：

```
BOOL HeapFree(  
HANDLE hHeap,  
DWORD fdwFlags,  
PVOID pvMem);
```

HeapFree函数用于释放内存块，如果它运行成功，便返回TRUE。参数fdwFlags既可以是0，也可以是HEAP\_NO\_SERIALIZE。调用这个函数可使堆栈管理器收回某些物理存储器，但是这没有保证。

#### 18.3.5 撤消堆栈

如果应用程序不再需要它创建的堆栈，可以通过调用HeapDestroy函数将它撤消：

```
BOOL HeapDestroy(HANDLE hHeap);
```

调用HeapDestroy函数可以释放堆栈中包含的所有内存块，也可以将堆栈占用的物理存储器和保留的地址空间区域重新返回给系统。如果该函数运行成功，HeapDestroy返回TRUE。如果在进程终止运行之前没有显式撤消堆栈，那么系统将为你将它撤消。但是，只有当进程终止运行时，堆栈才能被撤消。如果线程创建了一个堆栈，当线程终止运行时，该堆栈将不会被撤消。

在进程完全终止运行之前，系统不允许进程的默认堆栈被撤消。如果将进程的默认堆栈的句柄传递给HeapDestroy函数，系统将忽略对该函数的调用。

#### 18.3.6 用C++程序来使用堆栈

使用堆栈的最好方法之一是将堆栈纳入现有的C++程序。在C++中，调用new操作符，而不是调用通常的C运行期例程malloc，就可以执行类对象的分配操作。然后，当我们不再需要这个类对象时，调用delete操作符，而不是调用通常的C运行期例程free将它释放。例如，我们有一个称为CSomeClass的类，我们想要分配这个类的一个实例，那么可以使用类似下面的句法：

```
CSomeClass* pSomeClass = new CSomeClass;
```

当C++编译器查看这一行代码时，它首先查看CSomeClass类是否包含new操作符的成员函数。如果包含，那么编译器就生成调用该函数的代码。如果编译器没有找到重载new操作符的函数，那么编译器将生成调用标准C++的new操作符函数的代码。

当完成对已分配对象的使用后，可以通过调用delete操作符将它撤消：

```
delete pSomeClass;
```

通过为我们的C++类重载new和delete操作符，就能够很容易地利用堆栈函数。为此，让我们在头文件中将我们的CSomeClass类定义为如下的形式：

```
class CSomeClass {  
private:
```

```

static HANDLE s_hHeap;
static UINT s_uNumAllocsInHeap;

// Other private data and member functions
:
public:
    void* operator new (size_t size);
    void operator delete (void* p);
    // Other public data and member functions
    :
};

```

在这个代码段中，我声明了两个成员变量，即 `s_hHeap` 和 `s_uNumAllocsInHeap` 作为静态变量。由于它们是静态变量，因此 C++ 将使 `CSomeClass` 的所有实例共享相同的变量，也就是说，C++ 将不为已经创建的该类的每个实例分配独立的 `s_hHeap` 和 `s_uNumAllocsInHeap` 变量。这个情况对我们来说是非常重要的，因为我们的 `CSomeClass` 类的所有实例都在相同的堆栈中分配。

变量 `s_hHeap` 将包含分配 `CSomeClass` 对象时所在堆栈的句柄。`s_uNumAllocsInHeap` 变量只是一个计数器，用于计算堆栈中已经分配了多少个 `CSomeClass` 对象。每次在堆栈中分配一个新的 `CSomeClass` 对象时，`s_uNumAllocsInHeap` 的数字就递减。当 `s_uNumAllocsInHeap` 的数字到达 0 时，堆栈就不再需要并被释放。用于对堆栈进行操作的代码应该包括在类似下面的 .cpp 文件中：

```

HANDLE CSomeClass::s_hHeap = NULL;
UINT CSomeClass::s_uNumAllocsInHeap = 0;

void* CSomeClass::operator new (size_t size) {
    if (s_hHeap == NULL) {
        // Heap does not exist; create it.
        s_hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);

        if (s_hHeap == NULL)
            return(NULL);
    }
    // The heap exists for CSomeClass objects.
    void* p = HeapAlloc(s_hHeap, 0, size);

    if (p != NULL) {
        // Memory was allocated successfully; increment
        // the count of CSomeClass objects in the heap.
        s_uNumAllocsInHeap++;
    }

    // Return the address of the allocated CSomeClass object.
    return(p);
}

```

注意，我首先在代码的开始处定义了两个静态数字变量，即 `s_hHeap` 和 `s_uNumAllocsInHeap`，并且分别将它们初始化为 `NULL` 和 0。

C++ 的 `new` 操作符接受一个参数，即 `Size`。该参数用于指明存放 `CSomeClass` 对象所需要的字节数。`new` 操作符函数的第一个任务是创建一个堆栈，如果这样的堆栈尚未创建的话。这只



需要查看 `s_hHeap` 变量来了解它的值是否是 `NULL`。如果是 `NULL`，那么就调用 `HeapCreate` 函数，创建一个新堆栈，同时将 `HeapCreate` 返回的句柄保存在 `s_hHeap` 中，这样，下次调用 `new` 操作符时，就不会创建另一个堆栈，而是使用我们刚刚创建的堆栈。

当调用上面的 `HeapCreate` 函数时，我使用了 `HEAP_NO_SERIALIZE` 标志，因为示例代码的剩余部分不具备对多线程安全的特性。调用 `HeapCreate` 函数时使用的另外两个参数分别用于指明堆栈的初始大小和它的最大值。这里我为这两个值都选择了 0。第一个 0 表示该堆栈没有初始大小的值。第二个 0 表示该堆栈可以根据需要进行扩展。根据你的需要，可以改变这两个值中的任何一个，也可以同时改变两个值。

你可能认为将 `new` 操作符函数的 `size` 参数作为第二个参数传递给 `HeapCreate` 函数是值得的。如果这样的话，你可以对堆栈进行初始化，使它大得足以包含该类的一个实例。然后，当 `HeapAlloc` 第一次被调用时，它将以更快的速度运行，因为堆栈不必改变它的大小以便存放类的实例。但是，事情并不总是按照你的想像来进行的。由于堆栈中的每个已分配内存块都需要与之相关的开销，因此调用 `HeapAlloc` 时仍然必须改变堆栈的大小，这样它才能变得足够大，以便放置一个类的实例和它相关的开销。

一旦堆栈创建完成，就可以使用 `HeapAlloc` 函数从该堆栈中分配新的 `CSomeClass` 对象。第一个参数是堆栈的句柄，第二个参数是 `CSomeClass` 对象的大小。`HeapAlloc` 返回分配的内存块的地址。

当这个分配操作成功地执行时，我对 `s_uNumAllocsInHeap` 变量进行了递增，这样就可以知道堆栈中已分配了一个内存块。`new` 操作符函数做的最后一项工作是返回新分配的 `CSomeClass` 对象的地址。

这就是创建新 `CSomeClass` 对象的整个过程。下面要介绍的是，当应用程序不再需要 `CSomeClass` 时，如何将它撤消。这是 `delete` 操作符函数的责任，它的代码如下：

```
void CSomeClass::operator delete (void* p) {
    if (HeapFree(s_hHeap, 0, p)) {
        // Object was deleted successfully.
        s_uNumAllocsInHeap--;
    }

    if (s_uNumAllocsInHeap == 0) {
        // If there are no more objects in the heap,
        // destroy the heap.
        if (HeapDestroy(s_hHeap)) {
            // Set the heap handle to NULL so that the new operator
            // will know to create a new heap if a new CSomeClass
            // object is created.
            s_hHeap = NULL;
        }
    }
}
```

`delete` 操作符函数只接受一个参数，即被删除的对象的地址。该函数进行的第一项操作是调用 `HeapFree`，将堆栈的句柄和被释放的对象的地址传递给它。如果该对象被成功地释放了，`s_uNumAllocsInHeap` 的值就被递减 1，表示堆栈中又少了一个 `CSomeClass` 对象。接着该函数要检查 `s_uNumAllocsInHeap` 的值是否是 0。如果是 0，那么该函数就调用 `HeapDestroy`，将堆栈的句柄传递给它。如果堆栈被成功地撤消了，`s_hHeap` 将被设置为 `NULL`。这一点非常重要，因为我们的程序可能在将来的某个时候分配另一个 `CSomeClass` 对象。当它进行这项操作时，`new`

操作符将被调用，同时该操作符将查看 `s_hHeap` 变量，以确定它是应该使用现有的堆栈还是创建一个新堆栈。

这个例子显示了一种使用多个堆栈的简便方法。这个例子很容易建立，并且可以纳入若干个类中。不过应该对继承性问题有所考虑。如果你用 `CsomeClass` 类作为一个基类，派生一个新类，那么这个新类就可以继承 `CsomeClass` 的 `new` 和 `delete` 操作符。这个新类也可以继承 `CsomeClass` 的堆栈，这意味着当 `new` 操作符用于派生类时，该派生类对象的内存将从 `CsomeClass` 使用的同一个堆栈中分配。根据具体情况，你也许希望这样，也许不希望这样。如果对象的大小差别很大，建立的堆栈环境可能使你的堆栈变得支离破碎。正如本章前面部分中的“组件保护”和“更加有效地进行内存管理”两节所说的那样，你可能更加难以跟踪代码中的错误。

如果想将一个独立的堆栈用于各个派生类，只需要重复我在 `CsomeClass` 类中所进行的操作。也就是说，加上另一组 `s_hHeap` 和 `s_uNumAllocsInHeap` 变量，为 `new` 和 `delete` 操作符拷贝该代码。当进行编译时，编译器将发现你已经为该派生类重载了 `new` 和 `delete` 操作符，并将调用这些函数，而不是调用基类中的那些函数。

不为每个类创建新堆栈的唯一优点是，不必为每个堆栈分配开销和内存。但是，与这些堆栈相关的开销和内存并不很大，并且与带来的好处相比，这样做也许是值得的。我们采取的折中方案是，当你的应用程序已经经过很好的测试并且将要推向市场时，让每个类使用它自己的堆栈，让派生类共享基类的堆栈。不过堆栈碎片问题仍然可能存在。

## 18.4 其他堆栈函数

除了上面介绍的堆栈函数外，Windows 还提供了若干个别的函数。下面对它们作一个简单的介绍。

`ToolHelp` 的各个函数（第4章后面部分讲过）可以用来枚举进程的各个堆栈和这些堆栈中分配的内存块。关于这些函数的详细说明，请参见 Platform SDK 文档中的下列函数：`Heap32First`、`Heap32Next`、`Heap32ListFirst` 和 `Heap32ListNext`。`ToolHelp` 函数的优点在于，在 Windows 98 和 Windows 2000 中都能够使用它们。

本节中介绍的其他堆栈函数只存在于 Windows 2000 中。

由于进程的地址空间中可以存在多个堆栈，因此可以使用 `GetProcessHeaps` 函数来获取现有堆栈的句柄：

```
DWORD GetProcessHeaps(  
    DWORD dwNumHeaps,  
    PHANDLE pHeaps);
```

若要调用 `GetProcessHeaps` 函数，必须首先分配一个 `HANDLE` 数组，然后调用下面的函数：

```
HANDLE hHeaps[25];  
DWORD dwHeaps = GetProcessHeaps(25, hHeaps);  
if (dwHeaps > 25) {  
    // More heaps are in this process than we expected.  
} else {  
    // hHeaps[0] through hHeap[dwHeaps - 1]  
    // identify the existing heaps.  
}
```

注意，当该函数返回时，你的进程的默认堆栈的句柄也包含在堆栈句柄的数组中。

HeapValidate函数用于验证堆栈的完整性：

```
BOOL HeapValidate(  
    HANDLE hHeap,  
    DWORD fdwFlags,  
    LPCVOID pvMem);
```

调用该函数时，通常要传递一个堆栈句柄，一个值为 0 的标志（唯一的另一个合法标志是 HEAP\_NO\_SERIALIZE），并且为 pvMem 传递 NULL。然后，该函数将遍历堆栈中的内存块以确保所有内存块都完好无损。为了使该函数运行得更快，可以为参数 pvMem 传递一个特定的内存块的地址。这样做可使该函数只检查单个内存块的有效性。

若要合并地址中的空闲内存块并收回不包含已经分配的地址内存块的存储器页面，可以调用下面的函数：

```
UINT HeapCompact(  
    HANDLE hHeap,  
    DWORD fdwFlags);
```

通常情况下，可以为参数 fdwFlags 传递 0，但是也可以传递 HEAP\_NO\_SERIALIZE。

下面两个函数 HeapLock 和 HeapUnlock 是结合在一起使用的：

```
BOOL HeapLock(HANDLE hHeap);  
BOOL HeapUnlock(HANDLE hHeap);
```

这些函数是用于线程同步的。当调用 HeapLock 函数时，调用线程将成为特定堆栈的所有者。如果其他任何线程调用堆栈函数（设定相同的堆栈句柄），系统将暂停调用线程的运行，并且在堆栈被 HeapUnlock 函数解锁之前不允许它醒来。

HeapAlloc、HeapSize 和 HeapFree 等函数在内部调用 HeapLock 和 HeapUnlock 函数来确保对堆栈的访问能够顺序进行。自己调用 HeapLock 或 HeapUnlock 这种情况是不常见的。

最后一个堆栈函数是 HeapWalk：

```
BOOL HeapWalk(  
    HANDLE hHeap,  
    PPROCESS_HEAP_ENTRY pHeapEntry);
```

该函数只用于调试目的。它使你能够遍历堆栈的内容。可以多次调用该函数。每次调用该函数时，将传递必须分配和初始化的 PROCESS\_HEAP\_ENTRY 结构的地址：

```
typedef struct _PROCESS_HEAP_ENTRY {  
    PVOID lpData;  
    DWORD cbData;  
    BYTE cbOverhead;  
    BYTE iRegionIndex;  
    WORD wFlags;  
    union {  
        struct {  
            HANDLE hMem;  
            DWORD dwReserved[ 3 ];  
        } Block;  
        struct {  
            DWORD dwCommittedSize;  
            DWORD dwUnCommittedSize;  
            LPVOID lpFirstBlock;  
            LPVOID lpLastBlock;
```

```
        } Region;
    };
} PROCESS_HEAP_ENTRY, *LPPROCESS_HEAP_ENTRY, *PPROCESS_HEAP_ENTRY;
```

当开始枚举堆栈中的内存块时，必须将成员 `lpData` 设置为 `NULL`。这将告诉 `HeapWalk` 对该结构中的成员进行初始化。当成功地调用 `HeapWalk` 后，可以查看该结构的成员。若要进入堆栈的下一个内存块，只需要再次调用 `HeapWalk`，传递相同的堆栈句柄和在上次调用该函数时传递的 `PROCESS_HEAP_ENTRY` 结构的地址。当 `HeapWalk` 返回 `FALSE` 时，堆栈中就没有更多的内存块了。关于该结构中的成员的说明，请参见 Platform SDK 文档。

在循环调用 `HeapWalk` 的时候，必须使用 `HeapLock` 和 `HeapUnlock` 函数，这样，当遍历堆栈时，其他线程将无法分配和释放堆栈中的内存块。