[进程空间](http://blog.csdn.net/yeming81/article/details/2046193)

**1.**      **进程地址空间**

**1.1地址空间**

        **32|64位的系统|CPU**

        操作系统运行在硬件CPU上，32位操作系统运行于32位CPU上，64位操作系统运行于64位CPU上；目前没有真正的64位CPU。

32位CPU一次只能操作32位二进制数；位数多CPU设计越复杂，软件设计越简单。

       软件的进程运行于32位系统上，其寻址位也是32位，能表示的空间是232=4G，范围从0x0000 0000~0xFFFF FFFF。

        **NULL指针分区**

范围：0x0000 0000~0x0000 FFFF

作用：保护内存非法访问

例子：分配内存时，如果由于某种原因分配不成功，则返回空指针0x0000 0000；当用户继续使用比如改写数据时，系统将因为发生访问违规而退出。

        那么，为什么需要那么大的区域呢，一个地址值不就行了吗？我在想，是不是因为不让8或16位的程序运行于32位的系统上呢？！因为NULL分区刚好范围是16的进程空间。

        **独享用户分区**

范围：0x0001 0000~0x7FFE FFFF

作用：进程只能读取或访问这个范围的虚拟地址；超越这个范围的行为都会产生违规退出。

例子：

        程序的二进制代码中所用的地址大部分将在这个范围，所有exe和dll文件都加载到这个。每个进程将近2G的空间是独享的。

注意：如果在boot.ini上设置了/3G，这个区域的范围从2G扩大为3G：0x0001 0000~0xBFFE FFFF。

        **共享内核分区**

范围：0x8000 0000~0xFFFF FFFF

作用：这个空间是供操作系统内核代码、设备驱动程序、设备I/O高速缓存、非页面内存池的分配、进程目表和页表等。

例子：

       这段地址各进程是可以共享的。

注意：如果在boot.ini上设置了/3G，这个区域的范围从2G缩小为1G：0xC000 0000~0xFFFF FFFF。

       通过以上分析，可以知道，如果系统有n个进程，它所需的虚拟空间是：2G\*n+2G (内核只需2G的共享空间)。

**1.2地址映射**

        **区域**

区域指的是上述地址空间中的一片连续地址。区域的大小必须是粒度(64k) 的整数倍，不是的话系统自动处理成整数倍。不同CPU粒度大小是不一样的，大部分都是64K。

区域的状态有：空闲、私有、映射、映像。

在你的应用程序中，申请空间的过程称作保留(预订)，可以用VirtualAlloc；删除空间的过程为释放，可以用VirtualFree。

        在程序里预订了地址空间以后，你还不可以存取数据，因为你还没有付钱，没有真实的RAM和它关联。

这时候的区域状态是私有；

默认情况下，区域状态是空闲；

当exe或DLL文件被映射进了进程空间后，区域状态变成映像；

当一般数据文件被映射进了进程空间后，区域状态变成映射。

        **物理存储器**

Windows各系列支持的内存上限是不一样的，从2G到64G不等。理论上32位CPU，硬件上只能支持4G内存的寻址；能支持超过4G的内存只能靠其他技术来弥补。顺便提一下，Windows个人版只能支持最大2G内存，Intel使用Address Windows Extension (AWE) 技术使得寻址范围为236=64G。当然，也得操作系统配合。

        内存分配的最小单位是4K或8K，一般来说，根据CPU不同而不同，后面你可以看到可以通过系统函数得到区域粒度和页面粒度。

        **页文件**

页文件是存在硬盘上的系统文件，它的大小可以在系统属性里面设置，它相当于物理内存，所以称为虚拟内存。事实上，它的大小是影响系统快慢的关键所在，如果物理内存不多的情况下。

       每页的大小和上述所说内存分配的最小单位是一样的，通常是4K或8K。

        **访问属性**

物理页面的访问属性指的是对页面进行的具体操作：可读、可写、可执行。CPU一般不支持可执行，它认为可读就是可执行。但是，操作系统提供这个可执行的权限。

PAGE\_NOACCESS

PAGE\_READONLY

PAGE\_READWRITE

PAGE\_EXECUTE

PAGE\_EXECUTE\_READ

PAGE\_EXECUTE\_READWRITE

这6个属性很好理解，第一个是拒绝所有操作，最后一个是接受收有操作；

PAGE\_WRITECOPY

PAGE\_EXECUTE\_WRITECOPY

这两个属性在运行同一个程序的多个实例时非常有用；它使得程序可以共享代码段和数据段。一般情况下，多个进程只读或执行页面，如果要写的话，将会Copy页面到新的页面。通过映射exe文件时设置这两个属性可以达到这个目的。

PAGE\_NOCACHE

PAGE\_WRITECOMBINE

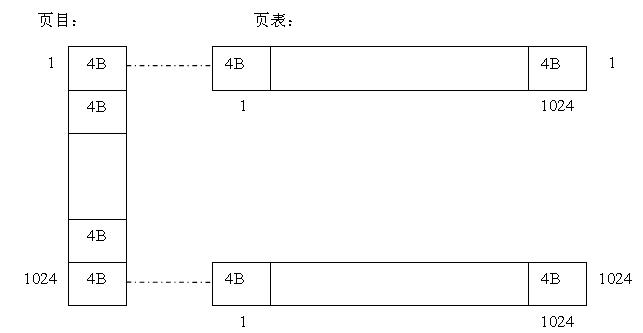
这两个是开发设备驱动的时候需要的。

PAGE\_GUARD

当往页面写入一个字节时，应用程序会收到堆栈溢出通知，在线程堆栈时有用。

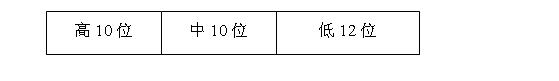
        **映射过程**

进程地址空间的地址是虚拟地址，也就是说，当取到指令时，需要把虚拟地址转化为物理地址才能够存取数据。这个工作通过页目和页表进行。



从图中可以看出，页目大小为4K，其中每一项(32位)保存一个页表的物理地址；每个页表大小为4K，其中每一项(32位)保存一个物理页的物理地址，一共有1024个页表。利用这4K+4K\*1K=4.4M的空间可以表示进程的1024\*1024\* (一页4K) =4G的地址空间。

进程空间中的32位地址如下：



高10位用来找到1024个页目项中的一项，取出页表的物理地址后，利用中10位来得到页表项的值，根据这个值得到物理页的地址，由于一页有4K大小，利用低12位得到单元地址，这样就可以访问这个内存单元了。

        每个进程都有自己的一个页目和页表，那么，刚开始进程是怎么找到页目所在的物理页呢？答案是CPU的CR3寄存器会保存当前进程的页目物理地址。

        当进程被创建时，同时需要创建页目和页表，一共需要4.4M。在进程的空间中，0xC030 0000~0xC030 0FFF是用来保存页目的4k空间。0xC000 0000~0xC03F FFFF是用来保存页表的4M空间。也就是说程序里面访问这些地址你是可以读取页目和页表的具体值的（要工作在内核方式下）。有一点我不明白的是，页表的空间包含了页目的空间！

        至于说，页目和页表是保存在物理内存还是页文件中，我觉得，页目比较常用，应该在物理内存的概率大点，页表需要时再从页文件导入物理内存中。

        页目项和页表项是一个32位的值，当页目项第0位为1时，表明页表已经在物理内存中；当页表项第0位为1时，表明访问的数据已经在内存中。还有很多数据是否已经被改变，是否可读写等标志。另外，当页目项第7位为1时，表明这是一个4M的页面，这值已经是物理页地址，用虚拟地址的低22位作为偏移量。还有很多：数据是否已经被改变、是否可读写等标志。

**1.3 一个例子**

        **编写生成软件程序exe**

软件描述如下：

Main ()

{

1：定义全局变量

2：处理函数逻辑（Load 所需DLL库，调用方法处理逻辑）

3：定义并实现各种方法（方法含有局部变量）

                       4：程序结束

}

将程序编译，生成exe文件，附带所需的DLL库。

        **exe文件格式**

exe文件有自己的格式，有若干节（section）：.text用来放二进制代码（exe或dll）；.data用来放各种全局数据。

.text

指令1：move a, b

指令2：add a, b

…

.data

数据1：a=2

数据2：b=1

…

这些地址都是虚拟地址，也就是进程的地址空间。

        **运行exe程序**

**建立进程：**运行这个exe程序时，系统会创建一个进程，建立进程控制块PCB，生成进程页目和页表，放到PCB中。

**数据对齐：**数据的内存地址除以数据的大小，余数为0时说明数据是对齐的。现在的编译器编译时就考虑数据对齐的问题，生成exe文件后，数据基本上是对齐的，CPU运行时，寄存器有标志标识CPU是否能够自动对齐数据，如果遇到不能对齐的情况，或者通过两次访问内存，或者通知操作系统处理。

要注意的是，如果数据没有对齐，CPU处理的效率是很低的。

**文件映射：**系统不会将整个exe文件和所有的DLL文件装载进物理内存中，同时它也不会装载进页面文件中。相反，它会建立文件映射，也就是利用exe本身当作页面文件。系统将部分二进制代码装载进内存，分配页面给它。

        假设分配了一个页面，物理地址为0x0232 FFF1。其中装载的一个指令虚拟地址为0x4000 1001=**0100 0000 00** 0000 0000 01 **0000 0000 0001**。一个页面有4K，系统会将指令保存在低12位0x0001的地址处。同时，系统根据高10位0x0100找到页目项，如果没有关联的页表，系统会生成一个页表，分配一个物理页；然后，根据中10位0x0001找到表项，将物理地址0x0232 FFF1存进去。

**执行过程：**

执行时，当系统拿到一个虚拟地址，就根据页目和页表找到数据的地址，根据页目上的值可以判断页表是在页文件中还是在内存中；

如果在页文件中，会将页面导入内存，更新页目项。读取页表项的值后，可以判断数据页文件中还是在物理内存中；如果在页文件中，会导入到内存中，更新页表项。最终，拿到了数据。

        在分配物理页的过程中，系统会根据内存分配的状况适当淘汰暂时不用的页面，如果页面内容改变了（通过页表项的标志位），保存到页文件中，系统会维护内存与页文件的对应关系。

由于将exe文件当作内存映射文件，当需要改变数据，如更改全局变量的值时，利用Copy-On-Write的机制，重新生成页文件，将结果保存在这个页文件中，原来的页文件还是需要被其他进程实例使用的。

        在清楚了指令和数据是如何导入内存，如何找到它们的情况下，剩下的就是CPU不断的取指令、运行、保存数据的过程了，当进程结束后，系统会清空之前的各种结构、释放相关的物理内存和删除页文件。

[内存状态查询](http://blog.csdn.net/yeming81/article/details/2046207)

**2.**      **内存状态查询函数**

**2.1系统信息**

Windows 提供API可以查询系统内存的一些属性，有时候我们需要获取一些页面大小、分配粒度等属性，在分配内存时用的上。

请看以下C++程序：

SYSTEM\_INFO sysInfo;

**GetSystemInfo**(&sysInfo);

            cout<<"机器属性："<<endl;

            cout<<"页大小="<<sysInfo.dwPageSize<<endl;

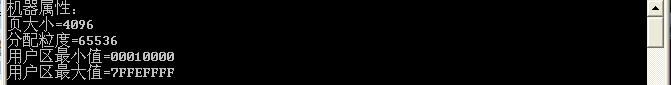
            cout<<"分配粒度="<<sysInfo.dwAllocationGranularity<<endl;

            cout<<"用户区最小值="<<sysInfo.lpMinimumApplicationAddress<<endl;

   cout<<"用户区最大值="

<<sysInfo.lpMaximumApplicationAddress<<endl<<endl;

**结果如下：**

****

可以看出，页面大小是4K，区域分配粒度是64K，进程用户区是0x0001 0000~0x7FFE FFFF。

**2.2内存状态**

        内存状态可以获取总内存和可用内存，包括页文件和物理内存。

请看以下C++程序：

MEMORYSTATUS memStatus;

**GlobalMemoryStatus**(&memStatus);

            cout<<"内存初始状态："<<endl;

            cout<<"内存繁忙程度="<<memStatus.dwMemoryLoad<<endl;

            cout<<"总物理内存="<<memStatus.dwTotalPhys<<endl;

            cout<<"可用物理内存="<<memStatus.dwAvailPhys<<endl;

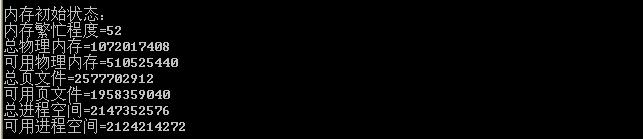
            cout<<"总页文件="<<memStatus.dwTotalPageFile<<endl;

            cout<<"可用页文件="<<memStatus.dwAvailPageFile<<endl;

            cout<<"总进程空间="<<memStatus.dwTotalVirtual<<endl;

   cout<<"可用进程空间="<<memStatus.dwAvailVirtual<<endl<<endl;

**结果如下：**

****

可以看出，总物理内存是1G，可用物理内存是510兆，总页文件是2.5G，这个是包含物理内存的页文件；可用页文件是1.9G。这里还标识了总进程空间,还有可用的进程空间，程序只用了22兆的内存空间。这里说的都是大约数。

内存繁忙程序是标识当前系统内存管理的繁忙程序，从0到100，其实用处不大。

        在函数里面**静态分配**一些内存后，看看究竟发生什么

char **stat**[65536];

            MEMORYSTATUS memStatus1;

**GlobalMemoryStatus**(&memStatus1);

            cout<<"静态分配空间："<<endl;

            printf("指针地址=%x/n",stat);

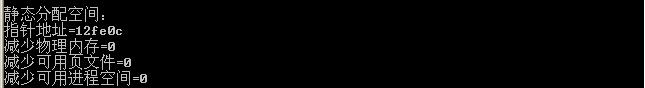
cout<<"减少物理内存="<<memStatus.dwAvailPhys-memStatus1.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus.dwAvailPageFile-memStatus1.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus.dwAvailVirtual-

memSta tus1.dwAvailVirtual<<endl<<endl;

**结果如下：**

****

可以看出，物理内存、可用页文件和进程空间都没有损耗。因为局部变量是分配在线程堆栈里面的，每个线程系统都会建立一个默认1M大小的堆栈给线程函数调用使用。如果分配超过1M，就会出现堆栈溢出。

        在函数里面**动态分配**300M的内存后，看看究竟发生什么

char \*dynamic=new char[300\*1024\*1024];

            MEMORYSTATUS memStatus2;

            GlobalMemoryStatus(&memStatus2);

            cout<<"动态分配空间："<<endl;

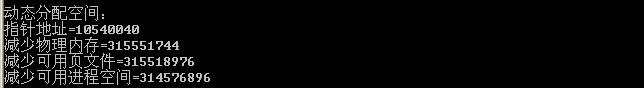
            printf("指针地址=%x/n",dynamic);

cout<<"减少物理内存="<<memStatus.dwAvailPhys-memStatus2.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus.dwAvailPageFile-memStatus2.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus.dwAvailVirtual-memStatus2.dwAvailVirtual<<endl<<endl;

**结果如下：**

****

动态分配情况下，系统分配直到内存页文件使用完为止，当然，系统要留一下系统使用的页面。

**2.3 进程区域地址查询**

在给定一个进程空间的地址后，可以查询它所在区域和相邻页面的状态，包括页面保护属性、存储器类型等。

        C++**静态分配**了两次内存，一次是4K大一点，一个是900K左右。

char arrayA[4097];

            char arrayB[900000];

第一次查询：

            long len=sizeof(MEMORY\_BASIC\_INFORMATION);

            MEMORY\_BASIC\_INFORMATION mbiA;

            VirtualQuery(arrayA,&mbiA,len);

            cout<<"静态内存地址属性："<<endl;

            cout<<"区域基地址="<<mbiA.AllocationBase<<endl;

            cout<<"区域邻近页面状态="<<mbiA.State<<endl;

            cout<<"区域保护属性="<<mbiA.AllocationProtect<<endl;

            cout<<"页面基地址="<<mbiA.BaseAddress<<endl;

            printf("arrayA指针地址=%x/n",arrayA);

            cout<<"从页面基地址开始的大小="<<mbiA.RegionSize<<endl;

            cout<<"邻近页面物理存储器类型="<<mbiA.Type<<endl;

            cout<<"页面保护属性="<<mbiA.Protect<<endl<<endl;

第二次查询：

            MEMORY\_BASIC\_INFORMATION mbiB;

            VirtualQuery(arrayB,&mbiB,len);

            cout<<"静态内存地址属性："<<endl;

            cout<<"区域基地址="<<mbiB.AllocationBase<<endl;

            cout<<"区域邻近页面状态="<<mbiB.State<<endl;

            cout<<"区域保护属性="<<mbiB.AllocationProtect<<endl;

            cout<<"页面基地址="<<mbiB.BaseAddress<<endl;

            printf("arrayB指针地址=%x/n",arrayB);

            cout<<"从页面基地址开始的大小="<<mbiB.RegionSize<<endl;

            cout<<"邻近页面物理存储器类型="<<mbiB.Type<<endl;

   cout<<"页面保护属性="<<mbiB.Protect<<endl<<endl;

说明：区域基地址指的是给定地址所在的进程空间区域；

邻近页面状态指的是与给定地址所在页面状态相同页面的属性：MEM\_FREE（空闲=65536）、MEM\_RESERVE（保留=8192）和MEM\_COMMIT（提交=4096）。

区域保护属性指的是区域初次被保留时被赋予的保护属性：PAGE\_READONLY（2）、PAGE\_READWRITE（4）、PAGE\_WRITECOPY（8）和PAGE\_EXECUTE\_WRITECOPY（128）等等。

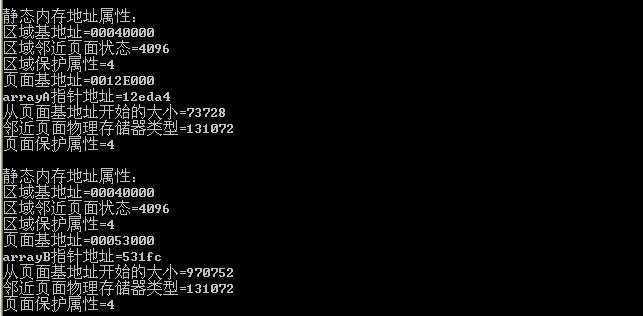
页面基地址指的是给定地址所在页面的基地址。

从页面基地址开始的区域页面的大小，指的是与给定地址所在页面状态、保护属性相同的页面。

邻近页面物理存储器类型指的是与给定地址所在页面相同的存储器类型，包括：MEM\_PRIVATE（页文件=131072）、MEM\_MAPPED（文件映射=262144）和MEM\_IMAGE（exe映像=16777216）。

页面保护属性指的是页面被指定的保护属性，在区域保护属性指定后更新。

**结果如下：**

****

如前所说，这是在堆栈区域0x0004 0000里分配的，后分配的地址arrayB反而更小，符合堆栈的特性。arrayA和arrayB它们处于不同的页面。页面都受页文件支持，并且区域都是提交的，是系统在线程创建时提交的。

        C++**动态分配**了两次内存，一次是1K大一点，一个是64K左右。所以应该不会在一个区域。

char \*dynamicA=new char[1024];

            char \*dynamicB=new char[65467];

            VirtualQuery(dynamicA,&mbiA,len);

            cout<<"动态内存地址属性："<<endl;

            cout<<"区域基地址="<<mbiA.AllocationBase<<endl;

            cout<<"区域邻近页面状态="<<mbiA.State<<endl;

            cout<<"区域保护属性="<<mbiA.AllocationProtect<<endl;

            cout<<"页面基地址="<<mbiA.BaseAddress<<endl;

            printf("dynamicA指针地址=%x/n",dynamicA);

            cout<<"从页面基地址开始的大小="<<mbiA.RegionSize<<endl;

            cout<<"邻近页面物理存储器类型="<<mbiA.Type<<endl;

            cout<<"页面保护属性="<<mbiA.Protect<<endl<<endl;

            VirtualQuery(dynamicB,&mbiB,len);

            cout<<"动态内存地址属性："<<endl;

            cout<<"区域基地址="<<mbiB.AllocationBase<<endl;

            cout<<"区域邻近页面状态="<<mbiB.State<<endl;

            cout<<"区域保护属性="<<mbiB.AllocationProtect<<endl;

            cout<<"页面基地址="<<mbiB.BaseAddress<<endl;

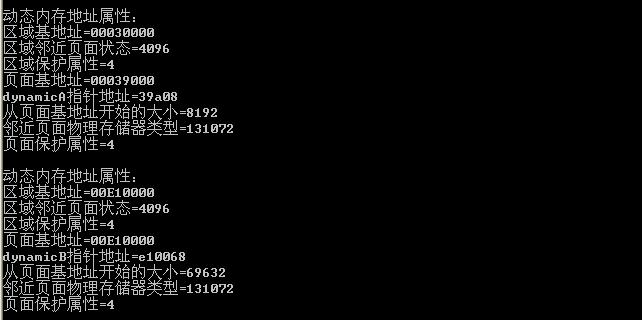
            printf("dynamicB指针地址=%x/n",dynamicB);

            cout<<"从页面基地址开始的大小="<<mbiB.RegionSize<<endl;

            cout<<"邻近页面物理存储器类型="<<mbiB.Type<<endl;

            cout<<"页面保护属性="<<mbiB.Protect<<endl;

**结果如下：**

****

这里是动态分配，dynamicA和dynamicB处于两个不同的区域；同样，页面都受页文件支持，并且区域都是提交的。

第二个区域是比64K大的，由分配粒度可知，区域至少是128K。那么，剩下的空间也是提交的吗，如果是的话那就太浪费了。看看就知道了：0x00E2 1000肯定在这个空间里，所以查询如下：

VirtualQuery((char\*)0xE23390,&mbiB,len);

            cout<<"动态内存地址属性："<<endl;

            cout<<"区域基地址="<<mbiB.AllocationBase<<endl;

            cout<<"区域邻近页面状态="<<mbiB.State<<endl;

            cout<<"区域保护属性="<<mbiB.AllocationProtect<<endl;

            cout<<"页面基地址="<<mbiB.BaseAddress<<endl;

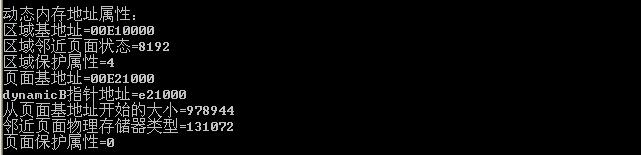
            printf("dynamicB指针地址=%x/n",0xE21000);

            cout<<"从页面基地址开始的大小="<<mbiB.RegionSize<<endl;

            cout<<"邻近页面物理存储器类型="<<mbiB.Type<<endl;

   cout<<"页面保护属性="<<mbiB.Protect<<endl;

**结果如下：**

****

可以看出，邻近页面状态为保留，还没提交，预料之中；**0x00E1 0000**这个区域的大小可以计算出来：**69632**+**978944**=**1024K**。系统动态分配了**1M**的空间，就为了**64K**左右大小的空间。可能是为了使得下次有要求分配时时不用再分配了。

[虚拟内存](http://blog.csdn.net/yeming81/article/details/2047879)

**3.**      **内存管理机制--虚拟内存 (VM)**

        **虚拟内存使用场合**

虚拟内存最适合用来管理大型对象或数据结构。比如说，电子表格程序，有很多单元格，但是也许大多数的单元格是没有数据的，用不着分配空间。也许，你会想到用动态链表，但是访问又没有数组快。定义二维数组，就会浪费很多空间。

它的优点是同时具有数组的快速和链表的小空间的优点。

        **分配虚拟内存**

如果你程序需要大块内存，你可以先保留内存，需要的时候再提交物理存储器。在需要的时候再提交才能有效的利用内存。一般来说，如果需要内存大于1M，用虚拟内存比较好。

        **保留**

用以下Windows 函数保留内存块

VirtualAlloc (PVOID 开始地址，SIZE\_T 大小，DWORD 类型，DWORD 保护属性)

一般情况下，你不需要指定“开始地址”，因为你不知道进程的那段空间是不是已经被占用了；所以你可以用NULL。“大小”是你需要的内存字节；“类型”有MEM\_RESERVE（保留）、MEM\_RELEASE（释放）和MEM\_COMMIT（提交）。“保护属性”在前面章节有详细介绍，只能用前六种属性。

如果你要保留的是**长久不会释放**的内存区，就保留在较高的空间区域，这样不会产生碎片。用这个类型标志可以达到：

MEM\_RESERVE|MEM\_TOP\_DOWN。

C++程序：保留1G的空间

LPVOID pV=VirtualAlloc(NULL,1000\*1024\*1024,MEM\_RESERVE|MEM\_TOP\_DOWN,PAGE\_READWRITE);

            if(pV==NULL)

            cout<<"没有那么多虚拟空间!"<<endl;

            MEMORYSTATUS memStatusVirtual1;

            GlobalMemoryStatus(&memStatusVirtual1);

            cout<<"虚拟内存分配："<<endl;

            printf("指针地址=%x/n",pV);

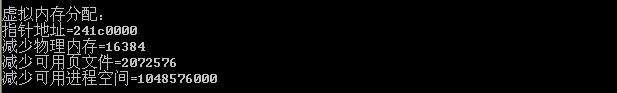
cout<<"减少物理内存="<<memStatusVirtual.dwAvailPhys-memStatusVirtual1.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatusVirtual.dwAvailPageFile-memStatusVirtual1.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatusVirtual.dwAvailVirtual-memStatusVirtual1.dwAvailVirtual<<endl<<endl;

结果如下：



可见，进程空间减少了1G；减少的物理内存和可用页文件用来管理页目和页表。但是，现在访问空间的话，会出错的：

int \* iV=(int\*)pV;

   //iV[0]=1;现在访问会出错，出现访问违规

        **提交**

你必须提供一个初始地址和提交的大小。提交的大小系统会变成页面的倍数，因为只能按页面提交。指定类型是MEM\_COMMIT。保护属性最好跟区域的保护属性一致，这样可以提高系统管理的效率。

C++程序：提交100M的空间

LPVOID pP=VirtualAlloc(pV,100\*1024\*1024,MEM\_COMMIT,PAGE\_READWRITE);

            if(pP==NULL)

            cout<<"没有那么多物理空间!"<<endl;

            int \* iP=(int\*)pP;

            iP[0]=3;

            iP[100/sizeof(int)\*1024\*1024-1]=5;//这是能访问的最后一个地址

            //iP[100/sizeof(int)\*1024\*1024]=5;访问出错

        **保留&提交**

你可以用类型MEM\_RESERVE|MEM\_COMMIT一次全部提交。但是这样的话，没有有效地利用内存，和使用一般的C++动态分配内存函数一样了。

        **更改保护属性**

更改已经提交的页面的保护属性，有时候会很有用处，假设你在访问数据后，不想别的函数再访问，或者出于防止指针乱指改变结构的目的，你可以更改数据所处的页面的属性，让别人无法访问。

VirtualProtect (PVOID 基地址，SIZE\_T 大小，DWORD 新属性，DWORD 旧属性)

“基地址”是你想改变的页面的地址，注意，不能跨区改变。

C++程序：更改一页的页面属性，改为只读，看看还能不能访问

DWORD protect;

            iP[0]=8;

            VirtualProtect(pV,4096,PAGE\_READONLY,&protect);

            int \* iP=(int\*)pV;

iP[1024]=9;//可以访问，因为在那一页之外

            //iP[0]=9;不可以访问，只读

            //还原保护属性

            VirtualProtect(pV,4096,PAGE\_READWRITE,&protect);

   cout<<"初始值="<<iP[0]<<endl;//可以访问

        **清除物理存储器内容**

清除页面指的是，将页面清零，也就是说当作页面没有改变。假设数据存在物理内存中，系统没有RAM页面后，会将这个页面暂时写进虚拟内存页文件中，这样来回的倒腾系统会很慢；如果那一页数据已经不需要的话，系统可以直接使用。当程序需要它那一页时，系统会分配另一页给它。

VirtualAlloc (PVOID 开始地址，SIZE\_T 大小，DWORD 类型，DWORD 保护属性)

“大小”如果小于一个页面的话，函数会执行失败，因为系统使用四舍五入的方法；“类型”是MEM\_RESET。

有人说，为什么需要清除呢，释放不就行了吗？你要知道，释放了后，程序就无法访问了。现在只是因为不需要结构的内容了，顺便提高一下系统的性能；之后程序仍然需要访问这个结构的。

C++程序：

清除1M的页面：

PVOID re=VirtualAlloc(pV,1024\*1024,MEM\_RESET,PAGE\_READWRITE);

            if(re==NULL)

   cout<<"清除失败!"<<endl;

这时候，页面可能还没有被清零，因为如果系统没有RAM请求的话，页面内存保存不变的，为了看看被清零的效果，程序人为的请求大量页面：

C++程序：

VirtualAlloc((char\*)pV+100\*1024\*1024+4096,memStatus.dwAvailPhys+10000000,MEM\_COMMIT,PAGE\_READWRITE);//没访问之前是不给物理内存的。

            char\* pp=(char\*)pV+100\*1024\*1024+4096;

            for(int i=0;i<memStatus.dwAvailPhys+10000000;i++)

            pp[i]='V';//逼他使用物理内存，而不使用页文件

            GlobalMemoryStatus(&memStatus);

            cout<<"内存初始状态："<<endl;

            cout<<"长度="<<memStatus.dwLength<<endl;

            cout<<"内存繁忙程度="<<memStatus.dwMemoryLoad<<endl;

            cout<<"总物理内存="<<memStatus.dwTotalPhys<<endl;

            cout<<"可用物理内存="<<memStatus.dwAvailPhys<<endl;

            cout<<"总页文件="<<memStatus.dwTotalPageFile<<endl;

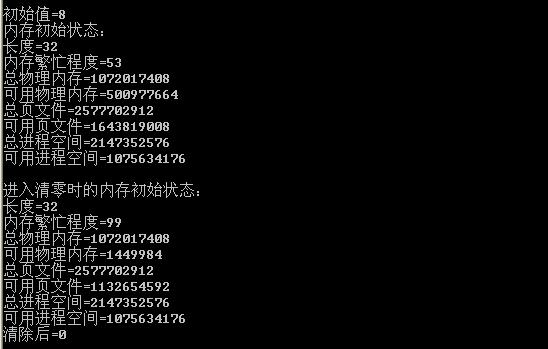
            cout<<"可用页文件="<<memStatus.dwAvailPageFile<<endl;

            cout<<"总进程空间="<<memStatus.dwTotalVirtual<<endl;

            cout<<"可用进程空间="<<memStatus.dwAvailVirtual<<end;

   cout<<"清除后="<<iP[0]<<endl;

结果如下:



当内存所剩无几时，系统将刚清除的内存页面分配出去，同时不会把页面的内存写到虚拟页面文件中。可以看见，原先是8的值现在是0了。

        **虚拟内存的关键之处**

虚拟内存存在的优点是，需要的时候才真正分配内存。那么程序必须决定何时才提交内存。

如果访问没有提交内存的数据结构，系统会产生访问违规的错误。提交的最好方法是，当你程序需要访问虚拟内存的数据结构时，假设它已经是分配内存的，然后异常处理可能出现的错误。对于访问违规的错误，就提交这个地址的内存。

        **释放**

可以释放整个保留的空间，或者只释放分配的一些物理内存。

释放特定分配的物理内存：

如果不想释放所有空间，可以只释放某些物理内存。

“开始地址”是页面的基地址，这个地址不一定是第一页的地址，一个窍门是提供一页中的某个地址就行了，因为系统会做页边界处理，取该页的首地址；“大小”是页面的要释放的字节数；“类型”是MEM\_DECOMMIT。

C++程序：

            //只释放物理内存

            VirtualFree((int\*)pV+2000,50\*1024\*1024,MEM\_DECOMMIT);

            int\* a=(int\*)pV;

            a[10]=2;//可以使用，没有释放这一页

            MEMORYSTATUS memStatusVirtual3;

            GlobalMemoryStatus(&memStatusVirtual3);

            cout<<"物理内存释放："<<endl;

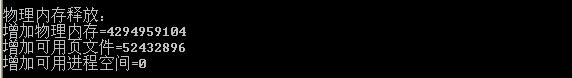
cout<<"增加物理内存="<<memStatusVirtual3.dwAvailPhys-memStatusVirtual2.dwAvailPhys<<endl;

cout<<"增加可用页文件="<<memStatusVirtual3.dwAvailPageFile-memStatusVirtual2.dwAvailPageFile<<endl;

   cout<<"增加可用进程空间="

<<memStatusVirtual3.dwAvailVirtual-memStatusVirtual2.dwAvailVirtual<<endl<<endl;

结果如下：



可以看见，只释放物理内存，没有释放进程的空间。

释放整个保留的空间：

VirtualFree (LPVOID 开始地址，SIZE\_T 大小，DWORD 类型)

“开始地址”一定是该区域的基地址；“大小”必须是0，因为只能释放整个保留的空间；“类型”是MEM\_RELEASE。

C++程序：

VirtualFree(pV,0,MEM\_RELEASE);

            //a[10]=2;不能使用了，进程空间也释放了

            MEMORYSTATUS memStatusVirtual4;

            GlobalMemoryStatus(&memStatusVirtual4);

            cout<<"虚拟内存释放："<<endl;

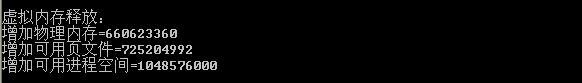
cout<<"增加物理内存="<<memStatusVirtual4.dwAvailPhys-memStatusVirtual3.dwAvailPhys <<endl;

cout<<"增加可用页文件="<<memStatusVirtual4.dwAvailPageFile-memStatusVirtual3.dwAvailPageFile<<endl;

cout<<"增加可用进程空间="

<<memStatusVirtual4.dwAvailVirtual-memStatusVirtual3.dwAvailVirtual<<endl<<endl;

结果如下：



整个分配的进程区域被释放了，包括所占的物理内存和页文件。

        **何时释放**

如果数组的元素大小是小于一个页面4K的话，你需要记录哪些空间不需要，哪些在一个页面上，可以用一个元素一个Bit来记录；另外，你可以创建一个线程定时检测无用单元。

        **扩展地址AWE**

AWE是内存管理器功能的一套应用程序编程接口 (API) ，它使程序能够将物理内存保留为非分页内存，然后将非分页内存部分动态映射到程序的内存工作集。此过程使内存密集型程序（如大型数据库系统）能够为数据保留大量的物理内存，而不必交换分页文件以供使用。相反，数据在工作集中进行交换，并且保留的内存超过 4 GB 范围。

对于物理内存小于2G进程空间时，它的作用是：不必要在物理内存和虚拟页文件中交换。

对于物理内存大于2G进程空间时，它的作用是：应用程序能够访问的物理内存大于2G，也就相当于进程空间超越了2G的范围；同时具有上述优点。

**3GB**

当在boot.ini 上加上 /3GB 选项时，应用程序的进程空间增加了1G，也就是说，你写程序时，可以分配的空间又增大了1G，而不管物理内存是多少，反正有虚拟内存的页文件，大不了慢点。

**PAE**

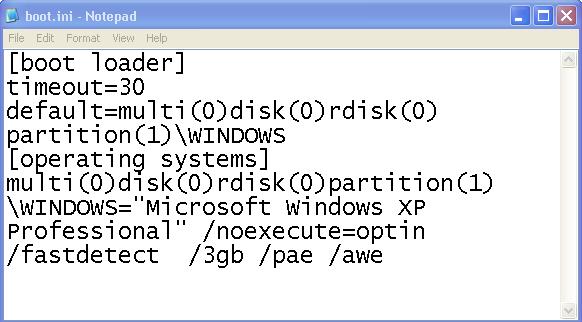
当在boot.ini上加上 /PAE 选项时，操作系统可以支持大于4G的物理内存，否则，你加再多内存操作系统也是不认的，因为管理这么大的内存需要特殊处理。所以，你内存小于4G是没有必要加这个选项的。注意，当要支持大于16G的物理内存时，不能使用/3G选项，因为，只有1G的系统空间是不能管理超过16G的内存的。

**AWE**

当在boot.ini上加上 /AWE选项时，应用程序可以为自己保留物理内存，直接的使用物理内存而不通过页文件，也不会被页文件交换出去。当内存大于3G时，就显得特别有用。因为可以充分利用物理内存。

当物理内存大于4G时，需要/PAE的支持。

以下是一个boot.ini的实例图，是我机器上的：



要使用AWE，需要用户具有Lock Pages in Memory权限，这个在控制面板中的本地计算机政策中设置。

**第一**，分配进程虚拟空间：

VirtualAlloc (PVOID 开始地址，SIZE\_T 大小，DWORD 类型，DWORD 保护属性)

“开始地址”可以是NULL，由系统分配进程空间；“类型”是MEM\_RESERVE|MEM\_PHYSICAL；“保护属性”只能是

PAGE\_READWRITE。

MEM\_PHYSICAL指的是区域将受物理存储器的支持。

**第二**，你要计算出分配的页面数目PageCount：

利用本文第二节的GetSystemInfo可以计算出来。

**第三**，分配物理内存页面：

AllocateUserPhysicalPages (HANDLE 进程句柄，SIZE\_T 页数，ULONG\_PTR 页面指针数组)

进程句柄可以用GetCurrentProcess()获得；页数是刚计算出来的页数PageCount；页面数组指针unsigned long\* Array[PageCount]。

系统会将分配结果存进这个数组。

**第四**，将物理内存与虚拟空间进行映射：

MapUserPhysicalPages (PVOID 开始地址，SIZE\_T 页数，ULONG\_PTR 页面指针数组)

“开始地址”是第一步分配的空间；

这样的话，虚拟地址就可以使用了。

如果“页面指针数组”是NULL，则取消映射。

**第五**，释放物理页面

FreeUserPhysicalPages (HANDLE 进程句柄，SIZE\_T 页数，ULONG\_PTR 页面指针数组)

这个除了释放物理页面外，还会取消物理页面的映射。

**第六**，释放进程空间

VirtualFree (PVOID 开始地址，0，MEM\_RELEASE)

C++程序：

首先，在登录用户有了Lock Pages in Memory权限以后，还需要调用Windows API激活这个权限。

BOOL VirtualMem::LoggedSetLockPagesPrivilege ( HANDLE hProcess,BOOL bEnable)

{

            struct {

                        DWORD Count;//数组的个数

                        LUID\_AND\_ATTRIBUTES Privilege [1];} Info;

            HANDLE Token;

            //打开本进程的权限句柄

            BOOL Result = OpenProcessToken ( hProcess,

                        TOKEN\_ADJUST\_PRIVILEGES,

                        & Token);

            If (Result!= TRUE )

            {

                        printf( "Cannot open process token./n" );

                        return FALSE;

            }

            //我们只改变一个属性

            Info.Count = 1;

            //准备激活

            if( bEnable )

                    Info.Privilege[0].Attributes = SE\_PRIVILEGE\_ENABLED;

            else

                        Info.Privilege[0].Attributes = 0;

            //根据权限名字找到LGUID

            Result = LookupPrivilegeValue ( NULL,

                        SE\_LOCK\_MEMORY\_NAME,

                        &(Info.Privilege[0].Luid));

            if( Result != TRUE )

            {

                        printf( "Cannot get privilege for %s./n", SE\_LOCK\_MEMORY\_NAME );

                        return FALSE;

            }

            // 激活Lock Pages in Memory权限

Result = AdjustTokenPrivileges ( Token, FALSE,(PTOKEN\_PRIVILEGES) &Info,0, NULL, NULL);

            if( Result != TRUE )

            {

                        printf ("Cannot adjust token privileges (%u)/n", GetLastError() );

                        return FALSE;

            }

            else

            {

                        if( GetLastError() != ERROR\_SUCCESS )

                        {

printf ("Cannot enable the SE\_LOCK\_MEMORY\_NAME privilege; ");

                                    printf ("please check the local policy./n");

                                    return FALSE;

                        }

            }

            CloseHandle( Token );

            return TRUE;

}

分配100M虚拟空间:

PVOID pVirtual=VirtualAlloc(NULL,100\*1024\*1024,MEM\_RESERVE|MEM\_PHYSICAL,PAGE\_READWRITE);

            if(pVirtual==NULL)

                        cout<<"没有那么大连续进程空间!"<<endl;

            MEMORYSTATUS memStatusVirtual5;

            GlobalMemoryStatus(&memStatusVirtual5);

            cout<<"虚拟内存分配："<<endl;

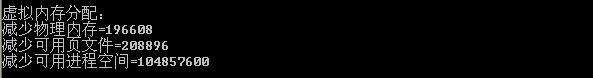
cout<<"减少物理内存="<<memStatusVirtual4.dwAvailPhys-memStatusVirtual5.dwAvailPhys<<endl

cout<<"减少可用页文件="<<memStatusVirtual4.dwAvailPageFile-memStatusVirtual5.dwAvailPageFile<<endl;

   cout<<"减少可用进程空间="

<<memStatusVirtual4.dwAvailVirtual-memStatusVirtual5.dwAvailVirtual<<endl<<endl;

结果如下：



可以看见，只分配了进程空间，没有分配物理内存。

分配物理内存：

ULONG\_PTR pages=(ULONG\_PTR)100\*1024\*1024/sysInfo.dwPageSize;

            ULONG\_PTR \*frameArray=new ULONG\_PTR[pages];

            //如果没激活权限，是不能调用这个方法的，可以调用，但是返回FALSE

BOOL flag=AllocateUserPhysicalPages(GetCurrentProcess(),

&pages,frameArray);

            if(flag==FALSE)

                        cout<<"分配物理内存失败!"<<endl;

            MEMORYSTATUS memStatusVirtual6;

            GlobalMemoryStatus(&memStatusVirtual6);

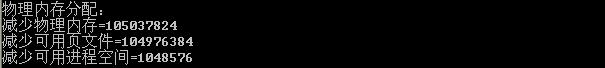
            cout<<"物理内存分配："<<endl;

cout<<"减少物理内存="<<memStatusVirtual5.dwAvailPhys-memStatusVirtual6.dwAvailPhys<<endl

cout<<"减少可用页文件="<<memStatusVirtual5.dwAvailPageFile-memStatusVirtual6.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatusVirtual5.dwAvailVirtual-memStatusVirtual6.dwAvailVirtual<<endl<<endl;

  结果如下：



分配了物理内存，可能分配时需要进程空间管理。

物理内存映射进程空间：

int\* pVInt=(int\*)pVirtual;

            //pVInt[0]=10;这时候访问会出错

            flag=MapUserPhysicalPages(pVirtual,1,frameArray);

            if(flag==FALSE)

                        cout<<"映射物理内存失败!"<<endl;

            MEMORYSTATUS memStatusVirtual7;

            GlobalMemoryStatus(&memStatusVirtual7);

            cout<<"物理内存分配："<<endl;

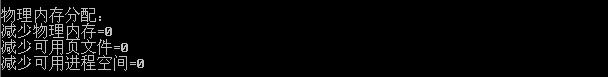
cout<<"减少物理内存="<<memStatusVirtual6.dwAvailPhys-memStatusVirtual7.dwAvailPhys<<endl

cout<<"减少可用页文件="<<memStatusVirtual6.dwAvailPageFile-memStatusVirtual7.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatusVirtual6.dwAvailVirtual-memStatusVirtual7.dwAvailVirtual<<endl<<endl;

结果如下：



这个过程没有损失任何东西。

看看第一次映射和第二次映射的值：

pVInt[0]=10;

            cout<<"第一次映射值="<<pVInt[0]<<endl;

                        flag=MapUserPhysicalPages(pVirtual,1,frameArray+1);

            if(flag==FALSE)

                        cout<<"映射物理内存失败!"<<endl;

            pVInt[0]=21;

            cout<<"第二次映射值="<<pVInt[0]<<endl;

            flag=MapUserPhysicalPages(pVirtual,1,frameArray);

            if(flag==FALSE)

                        cout<<"映射物理内存失败!"<<endl;

            cout<<"再现第一次映射值="<<pVInt[0]<<endl;

结果如下：

http://p.blog.csdn.net/images/p_blog_csdn_net/yeming81/awe4.JPG

可以看出，第二次映射的值没有覆盖第一次映射的值，也就是说，用同一个进程空间地址可以取出两份数据，这样的话，相当于进程的地址空间增大了。

[内存映射文件](http://blog.csdn.net/yeming81/article/details/2050521)

**4.**      **内存管理机制--内存映射文件 (Map)**

    和虚拟内存一样，内存映射文件可以用来保留一个进程地址区域；但是，与虚拟内存不同，它提交的不是物理内存或是虚拟页文件，而是硬盘上的文件。

        **使用场合**

它有三个主要用途：

**系统加载EXE和DLL文件**

操作系统就是用它来加载exe和dll文件建立进程，运行exe。这样可以节省页文件和启动时间。

**访问大数据文件**

如果文件太大，比如超过了进程用户区2G，用fopen是不能对文件进行操作的。这时，可用内存映射文件。对于大数据文件可以不必对文件执行I/O操作，不必对所有文件内容进行缓存。

**进程共享机制**

内存映射文件是多个进程共享数据的一种较高性能的有效方式，它也是操作系统进程通信机制的底层实现方法。RPC、COM、OLE、DDE、窗口消息、剪贴板、管道、Socket等都是使用内存映射文件实现的。

        **系统加载EXE和DLL文件**

      **EXE文件格式**

每个EXE和DLL文件由许多节(Section)组成，每个节都有保护属性：READ，WRITE，EXECUTE和SHARED（可以被多个进程共享，关闭页面的COPY-ON-WRITE属性）。

以下是常见的节和作用：

|  |  |
| --- | --- |
| 节名 | 作用 |
| .text | .exe和.dll文件的代码 |
| .data | 已经初始化的数据 |
| .bss | 未初始化的数据 |
| .reloc | 重定位表（装载进程的进程地址空间） |
| .rdata | 运行期只读数据 |
| .CRT | C运行期只读数据 |
| .debug | 调试信息 |
| .xdata | 异常处理表 |
| .tls | 线程的本地化存储 |
| .idata | 输入文件名表 |
| .edata | 输出文件名表 |
| .rsrc | 资源表 |
| .didata | 延迟输入文件名表 |

      **加载过程**

1.      系统根据exe文件名建立进程内核对象、页目和页表，也就是建立了进程的虚拟空间。

2.      读取exe文件的大小，在默认基地址0x0040 0000上保留适当大小的区域。可以在链接程序时用/BASE 选项更改基地址（在VC工程属性/链接器/高级上设置）。提交时，操作系统会管理页目和页表，将硬盘上的文件映射到进程空间中，页表中保存的地址是exe文件的页偏移。

3.      读取exe文件的.idata节，此节列出exe所用到的所有dll文件。然后和

exe文件一样，将dll文件映射到进程空间中。如果无法映射到基地址，系统会重新定位。

4.   映射成功后，系统会把第一页代码加载到内存，然后更新页目和页

表。将第一条指令的地址交给线程指令指针。当系统执行时，发现代码没有在内存中，会将exe文件中的代码加载到内存中。

      **第二次加载时（运行多个进程实例）**

1.      建立进程、映射进程空间都跟前面一样，只是当系统发现这个exe已

      经建立了内存映射文件对象时，它就直接映射到进程空间了；只是当

     系统分配物理页面时，根据节的保护属性赋予页面保护属性，对于代码

     节赋予READ属性，全局变量节赋予COPY-ON-WRITE属性。

2.      不同的实例共享代码节和其他的节，当实例需要改变页面内容时，会

      拷贝页面内容到新页面，更新页目和页表。

3.      对于不同进程实例需要共享的变量，exe文件有一

      个默认的节, 给这个节赋予SHARED属性。

4.      你也可以创建自己的SHARED节

#pragma data\_seg(“节名”)

Long instCount;

#pragma data\_seg()

然后，你需要在链接程序时告诉编译器节的默认属性。

/SECTION: 节名，RWS

或者，在程序里用以下表达式：

#pragma comment(linker,“/SECTION:节名，RWS”)

这样的话编译器会创建.drective节来保存上述命令，然后链接时会用它改变节属性。

注意，共享变量有可能有安全隐患，因为它可以读到其他进程的数据。

C++程序：多个进程共享变量举例

\*.cpp开始处：

#pragma data\_seg(".share")

long shareCount=0;

#pragma data\_seg()

#pragma comment(linker,"/SECTION:.share,RWS")

ShareCount++;

注意，同一个exe文件产生的进程会共享shareCount，必须是处于**同一个位置上**的exe。

        **访问大数据文件**

      **创建文件内核对象**

使用CreateFile（文件名，访问属性，共享模式，…） API可以创建。

其中，访问属性有：

0 不能读写 (用它可以访问文件属性)

GENERIC\_READ

GENERIC\_WRITE

GENERIC\_READ|GENERIC\_WRITE；

共享模式：

0 独享文件，其他应用程序无法打开

FILE\_SHARE\_WRITE

FILE\_SHARE\_READ|FILE\_SHARE\_WRITE

这个属性依赖于访问属性，必须和访问属性不冲突。

当创建失败时，返回INVALID\_HANDLE\_VALUE。

C++程序如下：

试图打开一个1G的文件：

MEMORYSTATUS memStatus;

GlobalMemoryStatus(&memStatus);

HANDLE hn=CreateFile(L"D://1G.rmvb",GENERIC\_READ|GENERIC\_WRITE,

FILE\_SHARE\_READ|FILE\_SHARE\_WRITE,NULL,OPEN\_EXISTING,FILE\_ATTRIBUTE\_NORMAL,NULL);

              if(hn==INVALID\_HANDLE\_VALUE)

                        cout<<"打开文件失败!"<<endl;

              FILE \*p=fopen("D://1G.rmvb","rb");

              if(p==NULL)

                        cout<<"用fopen不能打开大文件!"<<endl;

              MEMORYSTATUS memStatus2;

              GlobalMemoryStatus(&memStatus2);

              cout<<"打开文件后的空间："<<endl;

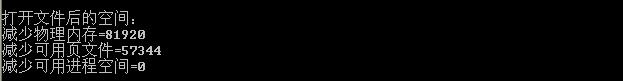
cout<<"减少物理内存="<<memStatus.dwAvailPhys-memStatus2.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus.dwAvailPageFile-memStatus2.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatus.dwAvailVirtual-memStatus2.dwAvailVirtual<<endl<<endl;

结果如下：



可见，系统需要一些内存来管理内核对象，每一次运行的结果都不一样，但差别不会太大。

用c语言的fopen不能打开这么大的文件。理论上，32位系统能支持232字节，但是，进程空间只有2G，它只能表示那么大的空间。

      **创建文件映射内核对象**

API如下：

HANDLE CreateFileMapping(Handle 文件，PSECURITY\_ATTRIBUTES 安全属性，DWORD保护属性，DWORD 文件大小高32位，DWORD 文件大小低32位，PCTSTR  映射名称)

“文件”是上面创建的句柄；

“安全属性”是内核对象需要的，NULL表示使用系统默认的安全属性；“保护属性”是当将存储器提交给进程空间时，需要的页面属性：PAGE\_READONLY, PAGE\_READWRITE和PAGE\_WRITECOPY。这个属性不能和文件对象的访问属性冲突。除了这三个外，还有两个属性可以和它们连接使用(|)。当更新文件内容时，不提供缓存，直接写入文件，可用SEC\_NOCACHE；当文件是可执行文件时，系统会根据节赋予不同的页面属性，可用SEC\_IMAGE。另外，SEC\_RESERVE和SEC\_COMMIT用于稀疏提交的文件映射，详细介绍请参考下文。

“文件大小高32位”和“文件大小低32位”联合起来告诉系统，这个映射所能支持的文件大小（操作系统支持264B文件大小）；当这个值大于实际的文件大小时，系统会扩大文件到这个值，因为系统需要保证进程空间能完全被映射。值为0默认为文件的大小，这时候如果文件大小为0，创建失败。

“映射名称”是给用户标识此内核对象，供各进程共享，如果为NULL，则不能共享。

对象创建失败时返回NULL。

创建成功后，系统仍未为文件保留进程空间。

C++程序：

                        MEMORYSTATUS memStatus2;

                        GlobalMemoryStatus(&memStatus2);

HANDLE hmap=CreateFileMapping(hn,NULL,PAGE\_READWRITE,0,0,L"Yeming-Map");

                        if(hmap==NULL)

                        cout<<"建立内存映射对象失败!"<<endl;

                        MEMORYSTATUS memStatus3;

                        GlobalMemoryStatus(&memStatus3);

                        cout<<"建立内存映射文件后的空间："<<endl;

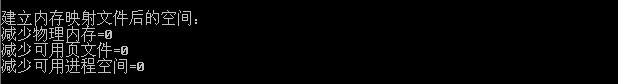
cout<<"减少物理内存="<<memStatus2.dwAvailPhys-memStatus3.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus2.dwAvailPageFile-memStatus3.dwAvailPageFile<<endl;

         cout<<"减少可用进程空间="

<<memStatus2.dwAvailVirtual-memStatus3.dwAvailVirtual<<endl<<endl;

            结果如下：



默认内存映射的大小是1G文件。没有损失内存和进程空间。它所做的是建立内核对象，收集一些属性。

      **文件映射内核对象映射到进程空间**

API如下：

PVOID MAPViewOfFile(HANDLE 映射对象，DWORD访问属性，DWORD 偏移量高32位，DWORD 偏移量低32位，SIZE\_T 字节数)

“映射对象”是前面建立的对象；

“访问属性”可以是下面的值：FILE\_MAP\_WRITE(读和写)、FILE\_MAP\_READ、FILE\_MAP\_ALL\_ACCESS(读和写)、FILE\_MAP\_COPY。当使用FILE\_MAP\_COPY时，系统分配虚拟页文件，当有写操作时，系统会拷贝数据到这些页面，并赋予PAGE\_READWRITE属性。

可以看到，每一步都需要设置这类属性，是为了可以多点控制，试想，如果在这一步想有多种不同的属性操作文件的不同部分，就比较有用。

“偏移高32位”和“偏移低32位”联合起来标识映射的开始字节（地址是分配粒度的倍数）；

“字节数”指映射的字节数，默认0为到文件尾。

当你需要指定映射到哪里时，你可以使用：

PVOID MAPViewOfFile(HANDLE 映射对象，DWORD访问属性，DWORD 偏移量高32位，DWORD 偏移量低32位，SIZE\_T 字节数，PVOID 基地址)

“基地址”是映射到进程空间的首地址，必须是分配粒度的倍数。

C++程序：

MEMORYSTATUS memStatus3;

            GlobalMemoryStatus(&memStatus3);

            LPVOID pMAP=MapViewOfFile(hmap,FILE\_MAP\_WRITE,0,0,0);

            cout<<"映射内存映射文件后的空间："<<endl;

if(pMAP==NULL)

               cout<<"映射进程空间失败!"<<endl;

            else

               printf("首地址=%x/n",pMAP);

            MEMORYSTATUS memStatus4;

            GlobalMemoryStatus(&memStatus4);

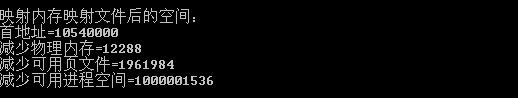
cout<<"减少物理内存="<<memStatus3.dwAvailPhys-memStatus4.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus3.dwAvailPageFile-memStatus4.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatus3.dwAvailVirtual-memStatus4.dwAvailVirtual<<endl<<endl;

结果如下：



进程空间减少了1G，系统同时会开辟一些内存来做文件缓存。

      **使用文件**

1.      对于大文件，可以用多次映射的方法达到访问的目的。有点像AWE技术。

2.      Windows只保证同一文件映射内核对象的多次映射的数据一致性，比如，当有两次映射同一对象到二个进程空间时，一个进程空间的数据改变后，另一个进程空间的数据也会跟着改变；不保证不同映射内核对象的多次映射的一致性。所以，使用文件映射时，最好在CreateFile时将共享模型设置为0独享，当然，对于只读文件没这个必要。

    C++程序：使用1G的文件

MEMORYSTATUS memStatus4;

                        GlobalMemoryStatus(&memStatus4);

                        cout<<"读取1G文件前："<<endl;

                        cout<<"可用物理内存="<<memStatus4.dwAvailPhys<<endl;

                        cout<<"可用页文件="<<memStatus4.dwAvailPageFile<<endl;

                        cout<<"可用进程空间="<<memStatus4.dwAvailVirtual<<endl<<endl;

                        int\* pInt=(int\*)pMAP;

                        cout<<"更改前="<<pInt[1000001536/4-1]<<endl;//文件的最后一个整数

                        for(int i=0;i<1000001536/4-1;i++)

                             pInt[i]++;

                        pInt[1000001536/4-1]=10;

                        pInt[100]=90;

                        pInt[101]=100;

                        cout<<"读取1G文件后："<<endl;

                        MEMORYSTATUS memStatus5;

                        GlobalMemoryStatus(&memStatus5);

                        cout<<"可用物理内存="<<memStatus5.dwAvailPhys<<endl;

                        cout<<"可用页文件="<<memStatus5.dwAvailPageFile<<endl;

                        cout<<"可用进程空间="<<memStatus5.dwAvailVirtual<<endl<<endl;

结果如下：



程序将1G文件的各个整型数据加1，从上图看出内存损失了600多兆，但有时候损失不过十几兆，可能跟系统当时的状态有关。

不管怎样，这样你完全看不到I/O操作，就像访问普通数据结构一样方便。

      **保存文件修改**

为了提高速度，更改文件时可能只更改到了系统缓存，这时，需要强制保存更改到硬盘，特别是撤销映射前。

BOOL FlushViewOfFile(PVOID 进程空间地址，SIZE\_T 字节数)

“进程空间地址”指的是需要更改的第一个字节地址，系统会变成页面的地址；

“字节数”，系统会变成页面大小的倍数。

写入磁盘后，函数返回，对于网络硬盘，如果希望写入网络硬盘后才返回的话，需要将FILE\_FLAG\_WRITE\_THROUGH参数传给CreateFile。

当使用FILE\_MAP\_COPY建立映射时，由于对数据的更改只是对虚拟页文件的修改而不是硬盘文件的修改，当撤销映射时，会丢失所做的修改。如果要保存，怎么办？

你可以用FILE\_MAP\_WRITE建立另外一个映射，它映射到进程的另外一段空间；扫描第一个映射的PAGE\_READWRITE页面(因为属性被更改)，如果页面改变，用MoveMemory或其他拷贝函数将页面内容拷贝到第二次映射的空间里，然后再调用FlushViewOfFile。当然，你要记录哪个页面被更改。

      **撤销映射**

用以下API可以撤销映射：

BOOL  UnmapViewOfFile(PVOID pvBaseAddress)

这个地址必须与MapViewOfFile返回值相同。

      **关闭内核对象**

在不需要内核对象时，尽早将其释放，防止内存泄露。由于它们是内核对象，调用CloseHandle(HANDLE)就可以了。

在CreateFileMapping后马上关闭文件句柄；

在MapViewOfFile后马上关闭内存映射句柄；

最后再撤销映射。

        **进程共享机制**

      基于硬盘文件的内存映射

如果进程需要共享文件，只要按照前面的方式建立内存映射对象，然后按照名字来共享，那么进程就可以映射这个对象到自己的进程空间中。

C++程序如下：

HANDLE mapYeming=OpenFileMapping(FILE\_MAP\_WRITE,true,L"Yeming-Map");

                        if(mapYeming==NULL)

                        cout<<"找不到内存映射对象:Yeming-Map!"<<endl;

                        MEMORYSTATUS memStatus3;

                        GlobalMemoryStatus(&memStatus3);

LPVOID pMAP=MapViewOfFile(mapYeming,FILE\_MAP\_WRITE,0,0,100000000);

                        cout<<"建立内存映射文件后的空间："<<endl;

                        if(pMAP==NULL)

                        cout<<"映射进程空间失败!"<<endl;

                        else

                        printf("首地址=%x/n",pMAP);

                        MEMORYSTATUS memStatus4;

                        GlobalMemoryStatus(&memStatus4);

cout<<"减少物理内存="<<memStatus3.dwAvailPhys-memStatus4.dwAvailPhys<<endl;

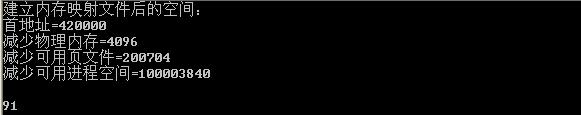
cout<<"减少可用页文件="<<memStatus3.dwAvailPageFile-memStatus4.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus3.dwAvailVirtual-memStatus4.dwAvailVirtual<<endl<<endl;

                        int\* pInt=(int\*)pMAP;

         cout<<pInt[100]<<endl;

         结果如下：



在2.exe中打开之前1.exe创建的内存映射对象(当然，1.exe得处于运行状态)，然后映射进自己的进程空间，当1.exe改变文件的值时，2.exe的文件对应值也跟着改变，Windows保证同一个内存映射对象映射出来的数据是一致的。可以看见，1.exe将值从90改为91，2.exe也跟着改变，因为它们有共同的缓冲页。

      基于页文件的内存映射

如果只想共享内存数据时，没有必要创建硬盘文件，再建立映射。可以直

接建立映射对象：

只要传给CreateFileMapping一个文件句柄INVALID\_HANDLE\_VALUE就行了。所以，CreateFile时，一定要检查返回值，否则会建立一个基于页文件的内存映射对象。接下来就是映射到进程空间了，这时，系统会分配页文件给它。

C++程序如下：

HANDLE hPageMap=CreateFileMapping(INVALID\_HANDLE\_VALUE,NULL,PAGE\_READWRITE,0,

100000000,L"Yeming-Map-Page");

            if(hPageMap==NULL)

                        cout<<"建立基于页文件的内存映射对象失败!"<<endl;

            MEMORYSTATUS memStatus6;

            GlobalMemoryStatus(&memStatus6);

            cout<<"建立基于页文件的内存映射文件后的空间："<<endl;

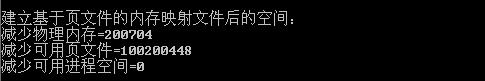
cout<<"减少物理内存="<<memStatus5.dwAvailPhys-memStatus6.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus5.dwAvailPageFile-memStatus6.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus5.dwAvailVirtual-memStatus6.dwAvailVirtual<<endl<<endl;

LPVOID pPageMAP=MapViewOfFile(hPageMap,FILE\_MAP\_WRITE,0,0,0);

            结果如下：



可见，和基于数据文件的内存映射不同，现在刚建立内核对象时就分配了所要的100M内存。好处是，别的进程可以通过这个内核对象共享这段内存，只要它也做了映射。

      稀疏内存映射文件

在虚拟内存一节中，提到了电子表格程序。虚拟内存解决了表示很少单元格有数据但必须分配所有内存的内存浪费问题；但是，如果想在多个进程之间共享这个电子表格结构呢？

如果用基于页文件的内存映射，需要先分配页文件，还是浪费了空间，没有了虚拟内存的优点。

Windows提供了稀疏提交的内存映射机制。

当使用CreateFileMapping时，保护属性用SEC\_RESERVE时，其不提交物理存储器，使用SEC\_COMMIT时，其马上提交物理存储器。注意，只有文件句柄为INVALID\_HANDLE\_VALUE时，才能使用这两个参数。

按照通常的方法映射时，系统只保留进程地址空间，不会提交物理存储器。

当需要提交物理内存时才提交，利用通常的VirtualAlloc函数就可以提交。

当释放内存时，不能调用VirtualFree函数，只能调用UnmapViewOfFile来撤销映射，从而释放内存。

C++程序如下：

HANDLE hVirtualMap=CreateFileMapping(INVALID\_HANDLE\_VALUE,NULL,PAGE\_READWRITE|SEC\_RESERVE,0,100000000,L"Yeming-Map-Virtual");

if(hPageMap==NULL)

                        cout<<"建立基于页文件的稀疏内存映射对象失败!"<<endl;

            MEMORYSTATUS memStatus8;

            GlobalMemoryStatus(&memStatus8);

            cout<<"建立基于页文件的稀疏内存映射文件后的空间："<<endl;

cout<<"减少物理内存="<<memStatus7.dwAvailPhys-memStatus8.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus7.dwAvailPageFile-memStatus8.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus7.dwAvailVirtual-memStatus8.dwAvailVirtual<<endl<<endl;

LPVOID pVirtualMAP=MapViewOfFile(hVirtualMap,FILE\_MAP\_WRITE,0,0,0);

            cout<<"内存映射进程后的空间："<<endl;

            if(pVirtualMAP==NULL)

                        cout<<"映射进程空间失败!"<<endl;

            else

                        printf("首地址=%x/n",pVirtualMAP);

            MEMORYSTATUS memStatus9;

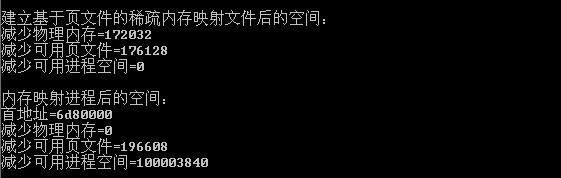
            GlobalMemoryStatus(&memStatus9);

cout<<"减少物理内存="<<memStatus8.dwAvailPhys-memStatus9.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus8.dwAvailPageFile-memStatus9.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus8.dwAvailVirtual-memStatus9.dwAvailVirtual<<endl<<endl;

结果如下：



用了SEC\_RESERVE后，只是建立了一个内存映射对象，和普通的一样；不同的是，它映射完后，得到了一个虚拟进程空间。现在，这个空间没有分配任何的物理存储器给它，你可以用VirtualAlloc 提交存储器给它，详细请参考上一篇<虚拟内存（VM）>。

注意，你不可以用VirtualFree来释放了,只能用UnmapViewOfFile来。

C++程序如下：

LPVOID pP=VirtualAlloc(pVirtualMAP,100\*1000\*1000,MEM\_COMMIT,PAGE\_READWRITE);

            MEMORYSTATUS memStatus10;

            GlobalMemoryStatus(&memStatus10);

cout<<"减少物理内存="<<memStatus9.dwAvailPhys-memStatus10.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus9.dwAvailPageFile-memStatus10.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus9.dwAvailVirtual-memStatus10.dwAvailVirtual<<endl<<endl;

            bool result=VirtualFree(pP,100000000,MEM\_DECOMMIT);

            if(!result)

                        cout<<"释放失败!"<<endl;

             result=VirtualFree(pP,100000000,MEM\_RELEASE);

            if(!result)

                        cout<<"释放失败!"<<endl;

            CloseHandle(hVirtualMap);

            MEMORYSTATUS memStatus11;

            GlobalMemoryStatus(&memStatus11);

cout<<"增加物理内存="<<memStatus11.dwAvailPhys-memStatus10.dwAvailPhys<<endl;

cout<<"增加可用页文件="<<memStatus11.dwAvailPageFile-memStatus10.dwAvailPageFile<<endl;

cout<<"增加可用进程空间="<<memStatus11.dwAvailVirtual-memStatus10.dwAvailVirtual<<endl<<endl;

            result=UnmapViewOfFile(pVirtualMAP);

            if(!result)

                        cout<<"撤销映射失败!"<<endl;

            MEMORYSTATUS memStatus12;

            GlobalMemoryStatus(&memStatus12);

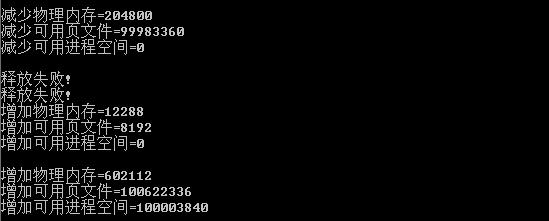
cout<<"增加物理内存="<<memStatus12.dwAvailPhys-memStatus11.dwAvailPhys<<endl;

cout<<"增加可用页文件="<<memStatus12.dwAvailPageFile-memStatus11.dwAvailPageFile<<endl;

cout<<"增加可用进程空间="

<<memStatus12.dwAvailVirtual-memStatus11.dwAvailVirtual<<endl<<endl;

结果如下：



可以看见，用VirtualFree是不能够释放这个稀疏映射的；最后用UnmapViewOfFile得以释放进程空间和物理内存。

[堆](http://blog.csdn.net/yeming81/article/details/2052311)

**5.**      **内存管理机制--堆 (Heap)**

        **使用场合**

堆是进程创建时在进程空间建立的区域，由堆管理器来管理。一个进程可以有很多个堆。进程有一个默认堆为1M，可以动态的扩大。

当程序需要管理很多小对象时，适合用堆；当需要的空间大于1M时，最好用虚拟内存来管理。

堆的优点是，有堆管理器来替它管理，不需管理具体的事情如页面边界

        和分配粒度等问题，你可以从调用函数看的出来，比VirtualAlloc的参数少了

        不少。

         堆的缺点是分配和释放的速度比前几种机制要慢，所以最好不要超过

         1M；不像虚拟内存那样随时提交和释放，因为它是由堆管理器决定的。如果

         用堆分配1G的空间，需要1分种，而用虚拟内存，则感觉不到任何延迟。

        **默认堆**

进程默认堆是供所有线程使用的，每当线程需要从堆中分配释放内存区时，系

 统会同步堆，所以访问速度较慢。

它的默认大小是1M，同样的，你可以通过以下链接命令改变其大小：

#pragma comment(linker,"/HEAP:102400000,1024000")

第一个值是堆的保留空间，第二个值是堆开始时提交的物理内存大小。本文将堆改变为100M。

**当你在程序中扩大了堆提交的物理内存时，进程运行时，物理内存将减少扩大的数量。但是，默认堆总是可以扩大的，不能限制它的最大值。**

**当你在程序中扩大了堆保留的空间时，进程运行时，可用进程空间将会减少扩大的数量。**

**每次你用New操作符分配内存时，进程空间会相应的减少，物理内存也会相应的减少。**

**一个重要的提示，本文经过测试，如果你需要的内存块大部分都超过512K，那么，建堆时给它的初始大小不应该很大，因为，如果你所需内存块大于512K的话，它不是从堆中分配的，也就是说不用堆中默认的空间，但其仍然属于堆管理。**

默认堆的一个用处是系统函数需要利用它运行。比如，Windows2000的字符集是UNICODE的，如果调用ANSI版本的函数，系统需要利用堆来从ANSI到UNICODE的转换，调用UNICODE版本的函数。

        **自建堆**

      **使用场合**

**保护数据结构：**

将不同的数据结构存在不同的堆中，可以防止不同的结构之间由于指针误操作而破坏了它们。

**消除内存碎片：**

将大小不同的结构保存在一个堆中，会导致碎片的产生，比如释放一个小结构时，大结构也不能利用它。

**独享堆的快速：**

如果用默认堆的话，线程之间是同步访问，速度慢；如果创建独享堆，则系统可以不需同步，比较快。

第二个快速体现在释放的快速，默认堆中，你只能释放某个内存块，而不能释放整个堆；而独享堆可以一次释放堆，也就是释放了所有的内存块。

      **开始使用**

**建立堆：**

使用以下API

            HANDLE HeapCreate(DWORD 选项，SIZE\_T 初始大小，SIZE\_T 最大值)

“选项” 取值为0 ，不是以下任意一个

HEAP\_NO\_SERIALIZE，系统无需同步堆

HEAP\_GENERATE\_EXCEPTIONS，当创建失败或分配失败时产生异常。

“初始大小”是堆的大小，系统会规整到页面的整数倍，如0~4096的任何数都为4096；但是，进程空间至少要64K。

“最大值”是堆允许的最大值；为0则无限。

使用HEAP\_NO\_SERIALIZE需确定只有单线程访问这个堆，否则有可能破坏堆；或程序有同步代码来同步堆。

C++程序如下：

pHeap=(char\*)GetProcessHeap();

printf("默认堆地址=%x/n",pHeap);

MEMORYSTATUS memStatus2;

            GlobalMemoryStatus(&memStatus2);

HANDLE hHeap=HeapCreate(HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS,1024\*1024\*50,0);

            char\* pHeap=(char\*)hHeap;

            printf("新建堆1地址=%x/n",pHeap);

            if(hHeap==NULL)

            {

                        cout<<"创建堆失败!"<<endl;

            }

            MEMORYSTATUS memStatus3;

            GlobalMemoryStatus(&memStatus3);

            cout<<"建立堆后："<<endl;

cout<<"减少物理内存="<<memStatus2.dwAvailPhys-memStatus3.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus2.dwAvailPageFile-memStatus3.dwAvailPageFile<<endl;

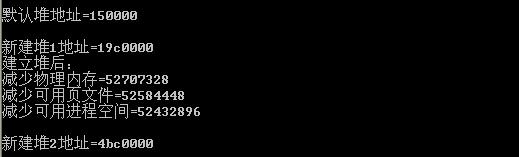
cout<<"减少可用进程空间="<<memStatus2.dwAvailVirtual-memStatus3.dwAvailVirtual<<endl<<endl;

HANDLE hHeap2=HeapCreate(HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS,1024\*1024\*10,0);

            char\* pHeap2=(char\*)hHeap2;

            printf("新建堆2地址=%x/n",pHeap2);

            结果如下：



当建立堆1时，它分配了50M的物理内存给堆使用；当建立堆2时，堆2的地址是0x04bc 0000=0x019c 0000+50\*1024\*1024.

**分配内存：**

使用以下API

            PVOID HeapAlloc(HANDLE 堆句柄，DWORD 选项，SIZE\_T 字节数)

            “选项”可以是，

            HEAP\_ZERO\_MEMORY，所有字节初始化为0

            HEAP\_NO\_SERIALIZE，堆这个内存区独享

HEAP\_GENERATE\_EXCEPTIONS，产生异常。如果创建堆有了它就不用再设了。异常可能为：STATUS\_NO\_MEMOR（无足够内存）和STATUS\_ACCESS\_VIOLATION（堆被破坏，分配失败）。

C++程序如下：

GlobalMemoryStatus(&memStatus3);

PVOID pV=HeapAlloc(hHeap,

HEAP\_ZERO\_MEMORY|HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS,1024\*507);

            if(pV==NULL)

            {

                        cout<<"分配堆内存失败!"<<endl;

            }

            char \* pC=(char\*)pV;

            printf("第一次分配地址=%x/n",pC);

            MEMORYSTATUS memStatus4;

            GlobalMemoryStatus(&memStatus4);

            cout<<"第一次堆分配后："<<endl;

cout<<"减少物理内存="<<memStatus3.dwAvailPhys-memStatus4.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus3.dwAvailPageFile-memStatus4.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus3.dwAvailVirtual-memStatus4.dwAvailVirtual<<endl<<endl;

PVOID pV2=HeapAlloc(hHeap,

HEAP\_ZERO\_MEMORY|HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS,1024\*508);

            if(pV2==NULL)

            {

                        cout<<"分配堆内存失败!"<<endl;

            }

            char \* pC2=(char\*)pV2;

            printf("第二次分配地址=%x/n",pC2);

            MEMORYSTATUS memStatus5;

            GlobalMemoryStatus(&memStatus5);

            cout<<"第二次堆分配后："<<endl;

cout<<"减少物理内存="<<memStatus4.dwAvailPhys-memStatus5.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus4.dwAvailPageFile-memStatus5.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus4.dwAvailVirtual-memStatus5.dwAvailVirtual<<endl<<endl;

            for(int i=0;i<200\*1024;i++)

                        pC2[i]=9;

            MEMORYSTATUS memStatus10;

            GlobalMemoryStatus(&memStatus10);

            cout<<"第二次堆使用一半后："<<endl;

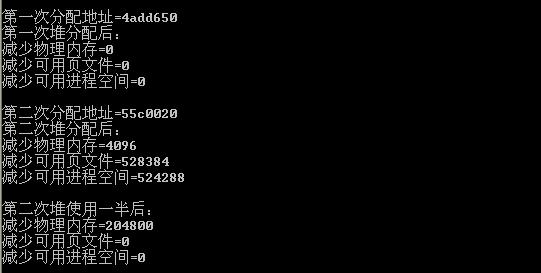
cout<<"减少物理内存="<<memStatus5.dwAvailPhys-memStatus10.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus5.dwAvailPageFile-memStatus10.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatus5.dwAvailVirtual-memStatus10.dwAvailVirtual<<endl<<endl;

结果如下：



可以看出，第一次分配507K的地址为0x04ad d650<0x04bc 0000，它是在堆中分配的；第二次分配508K的地址为0x055c 0020>0x04bc 0000，它是在堆外分配的；无论在多大的堆中，只要分配内存块大于507K时，都会在堆外分配，但是，它像在堆中一样，存在堆的链接表中，受堆管理。分配时，系统使用的是虚拟页文件；只有在真正使用时，才会分配物理内存。

至于为什么分配大于507K会在堆外分配而不直接使用堆中的内存，目前仍然不清楚。

**改变大小：**

PVOID HeapReAlloc(HANDLE 堆句柄，DWORD 选项，PVOID 旧内存块地址，SIZE\_T 新内存块大小)

“选项”除了以上三个外，还有HEAP\_REALLOC\_IN\_PLACE\_ONLY，指定不能移动原有内存块的地址。

C++程序如下：

GlobalMemoryStatus(&memStatus4);

            PVOID pV2New=HeapReAlloc(hHeap,0,pV2,1024\*1024\*2);

            if(pV2New!=NULL)

            {

            char \* pC2New=(char\*)pV2New;

            printf("改变分配地址=%x/n",pC2New);

            cout<<pC2New[0]<<endl;

            //cout<<pC2[0]<<endl;出现访问违规

            SIZE\_T lenNew=HeapSize(hHeap,0,pV2New);

            cout<<"改变后大小="<<lenNew<<endl;

            }

            GlobalMemoryStatus(&memStatus5);

            cout<<"改变分配后："<<endl;

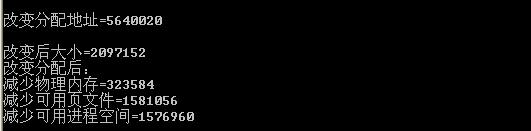
cout<<"减少物理内存="<<memStatus4.dwAvailPhys-memStatus5.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus4.dwAvailPageFile-memStatus5.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="

<<memStatus4.dwAvailVirtual-memStatus5.dwAvailVirtual<<endl<<endl;

结果如下：



可以看出，新内存块紧接着原来内存块结束的地方开始创建，大小为2M;原来的内存块的内容被销毁和释放，所以新内存块只减少了增加的内存量。一个缺点就是，新内存块居然不保留原来内存的内容！另外，如果采用HEAP\_REALLOC\_IN\_PLACE\_ONLY的话，出现Not Enough Quote异常。也就是说，当前内存的状况是，必须移动才可以扩大此内存块。

**查询内存：**

可以查询堆中一个内存块的大小。

SIZE\_T HeapSize(HANDLE 堆句柄，DWORD 选项，LPVOID 内存块地址)

            “选项”可为0或HEAP\_NO\_SERIALIZE。

            参考以上例子。

**释放内存块：**

            BOOL HeapFree(HANDLE 堆句柄，DWORD 选项，PVOID 内存块地址)

            “选项”可为0或HEAP\_NO\_SERIALIZE。

            C++程序如下：

            GlobalMemoryStatus(&memStatus5);

            HeapFree(hHeap,0,pV2New);

            MEMORYSTATUS memStatus6;

            GlobalMemoryStatus(&memStatus6);

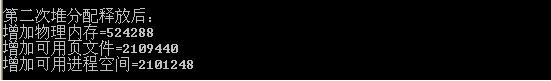
            cout<<"第二次堆分配释放后："<<endl;

cout<<"增加物理内存="<<memStatus6.dwAvailPhys-memStatus5.dwAvailPhys<<endl;

cout<<"增加可用页文件="<<memStatus6.dwAvailPageFile-memStatus5.dwAvailPageFile<<endl;

cout<<"增加可用进程空间="<<memStatus6.dwAvailVirtual-memStatus5.dwAvailVirtual<<endl<<endl;

结果如下：



            内存空间释放了原来的2M空间。

**释放堆：**

BOOL HeapDestroy(HANDLE 堆句柄)

            不能用它释放默认堆，系统忽略它的处理。

这一次，我们先在堆1中分配了70M的内存，由于它很大，所以，堆在堆外给它分配了内存，所以，堆1一共有50M+70M=120M。释放程序如下：

PVOID pV4=HeapAlloc(hHeap,HEAP\_ZERO\_MEMORY|HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS)

,1024\*1024\*70);

            if(pV4==NULL)

            {

                        cout<<"分配堆内存失败!"<<endl;

            }

            char \* pC4=(char\*)pV4;

            printf("第四次堆分配=%x/n",pC4);

            MEMORYSTATUS memStatus9;

            GlobalMemoryStatus(&memStatus9);

            cout<<"分配堆内存后："<<endl;

cout<<"减少物理内存="<<memStatus7.dwAvailPhys-memStatus9.dwAvailPhys<<endl;

cout<<"减少可用页文件="<<memStatus7.dwAvailPageFile-memStatus9.dwAvailPageFile<<endl;

cout<<"减少可用进程空间="<<memStatus7.dwAvailVirtual-memStatus9.dwAvailVirtual<<endl<<endl;

            SIZE\_T len=HeapSize(hHeap,0,pV4);

            cout<<"len="<<len<<endl;

            bool re=HeapDestroy(hHeap);

            if(re==false)

            {

                        cout<<"释放堆失败!"<<endl;

}

MEMORYSTATUS memStatus8;

            GlobalMemoryStatus(&memStatus8);

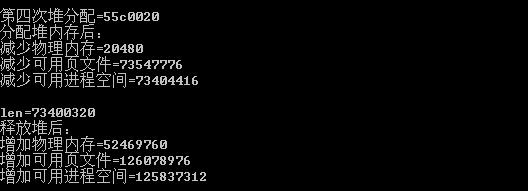
            cout<<"释放堆后："<<endl;

cout<<"增加物理内存="<<memStatus8.dwAvailPhys-memStatus9.dwAvailPhys<<endl;

cout<<"增加可用页文件="<<memStatus8.dwAvailPageFile-memStatus9.dwAvailPageFile<<endl;

cout<<"增加可用进程空间="<<memStatus8.dwAvailVirtual-memStatus9.dwAvailVirtual<<endl<<endl;

结果如下：



如所猜想一样，释放了120M内存。

**获取所有堆：**

DWORD GetProcessHeaps(DWORD 数量，PHANDLE 句柄数组)

            “数量”是你想获取的堆数目；

            “句柄数组”是获得的堆句柄。

            默认堆也可以获取。

            HANDLE        handles[10];

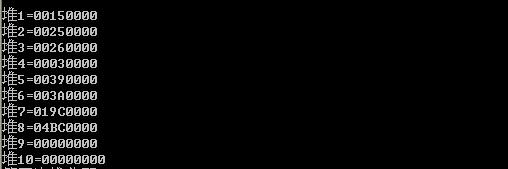
            memset(handles,0,sizeof(handles));

            GetProcessHeaps(10,handles);

            for(int i=0;i<10;i++)

                        cout<<"堆"<<i+1<<"="<<handles[i]<<endl;

            结果如下：



可以看见，一共有8个堆，堆1是默认堆，堆7和堆8是本文建立的堆。另外5个不知来源。

**验证堆：**

BOOL HeapValidate(HANDLE 堆句柄，DWORD 选项，LPVOID 内存块地址)

            “选项” 可为0或HEAP\_NO\_SERIALIZE；

            “内存块地址”为NULL时，验证所有内存块。

            C++程序如下：

HANDLE        handles[10];

            memset(handles,0,sizeof(handles));

            GetProcessHeaps(10,handles);

            for(int i=0;i<10;i++)

            {

                        cout<<"堆"<<i+1<<"="<<handles[i]<<"   ";

                        if(HeapValidate(handles[i],0,NULL))

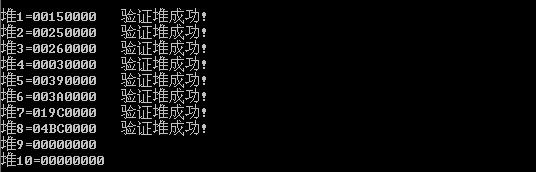
                                    cout<<"验证堆成功!"<<endl;

                        else

                                    cout<<endl;

            }

结果如下：



**合并内存块：**

UINT HeapCompact(HANDLE 堆句柄，DWORD 选项)

            “选项” 可为0或HEAP\_NO\_SERIALIZE；

            此函数可以合并空闲内存块。

**其他函数：**

HeapLock和HeapUnlock 通常是系统使用的；

            HeapWalk可以遍历堆内存，需要以上两个函数。

        **C++内存函数**

Malloc和Free

这是C语言使用的函数，只能从默认堆中分配内存，并且只是分配内存，不能调用构造函数，且只是按字节分配，不能按类型分配。

New 和Delete

这是C++语言使用的函数，默认情况下从默认堆中分配内存，但是也可以通过重载New函数，从自建堆中按类型分配；同时可以执行构造函数和析构函数。它底层是通过HeapAlloc和HeapFree实现的。 依赖于编译器的实现。

GlobalAlloc 和GlobalFree

这是比HeapAlloc和HeapFree更慢的函数，但是也没有比它们更好的优点，只能在默认堆中分配；16位操作系统下利用它们分配内存。

LocalAlloc和LocalFree

在WindowsNT 内核里，和GlobalAlloc、GlobalFree是一样的。

        **一个例子**

默认情况下，New关键字是利用HeapAlloc在默认堆上建立对象。本文重载了类的New方法，使得类在自己的堆中存放，这样可以与外面的对象隔离，以免重要的数据结构被意外破坏。由于类中的成员变量是在堆中存放，因此不局限于线程堆栈的1M空间。

C++程序如下：

class AllocateInOtherHeap

{

public:

            AllocateInOtherHeap(void);

            ~AllocateInOtherHeap(void);

            void\* operator new(size\_t size);

            static HANDLE heap;

         public:

            //类对象唯一所需的空间

            int iArray[1024\*1024\*10];

            AllocateInOtherHeap::AllocateInOtherHeap(void)

{

            cout<<"AllocateInOtherHeap()"<<endl;

            //如果New函数没有分配够空间，那么此处会出现访问违规

            memset(iArray,0,sizeof(AllocateInOtherHeap));

            iArray[1024]=8;

}

void\* AllocateInOtherHeap::operator new(size\_t size)

{

            if(heap==NULL)

heap=HeapCreate(HEAP\_NO\_SERIALIZE|HEAP\_GENERATE\_EXCEPTIONS,1024\*1024\*10,0);

            //分配足够这个类对象的空间

            void\* p=HeapAlloc(heap,0,sizeof(AllocateInOtherHeap));

            cout<<"堆的大小="<<HeapSize(heap,0,p)<<endl;

            printf("AllocateInOtherHeap堆地址=%x/n",heap);

            printf("AllocateInOtherHeap返回地址=%x/n",p);

            return p;

}

AllocateInOtherHeap::~AllocateInOtherHeap(void)

{

            cout<<"~AllocateInOtherHeap"<<endl;

}

void AllocateInOtherHeap::operator delete(void\* p)

{

            HeapFree(heap,0,p);

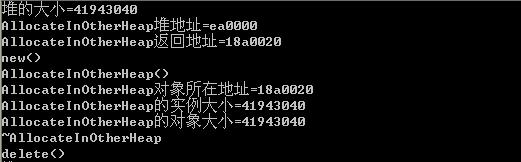
            HeapDestroy(heap);

            cout<<"delete()"<<endl;

}

};

结果如下：



可见，new函数先分配够空间，然后才能初始化对象变量；而delete函数得先做析构，才能释放空间。对象保存在堆外，因为大于512K；对象大小刚好是iArray变量的大小。

注意，如果没有分配足够的空间，虽然你可以得到对象指针，但是你访问数据时可能会出现访问违规，如果没出现，那更惨，意味着你读写了别人的数据。

[堆栈](http://blog.csdn.net/yeming81/article/details/2052312)

**6.**      **内存管理机制--堆栈 (Stack)**

        **使用场合**

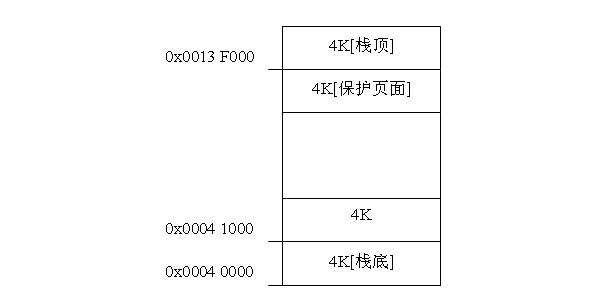
操作系统为每个线程都建立一个默认堆栈，大小为1M。这个堆栈是供函数调用时使用，线程内函数里的各种静态变量都是从这个默认堆栈里分配的。

        **堆栈结构**

默认1M的线程堆栈空间的结构举例如下，其中，基地址为0x0004 0000，刚开始时，CPU的堆栈指针寄存器保存的是栈顶的第一个页面地址0x0013 F000。第二页面为保护页面。这两页是已经分配物理存储器的可用页面。

随着函数的调用，系统将需要更多的页面，假设需要另外5页，则给这5页提交内存，删除原来页面的保护页面属性，最后一页赋予保护页面属性。

当分配倒数第二页0x0004 1000时，系统不再将保护属性赋予它，相反，它会产生堆栈溢出异常STATUS\_STACK\_OVERFLOW，如果程序没有处理它，则线程将退出。最后一页始终处于保留状态，也就是说可用堆栈数是没有1M的，之所以不用，是防止线程破坏栈底下面的内存（通过违规访问异常达到目的）。



当程序的函数里分配了临时变量时，编译器把堆栈指针递减相应的页数目，堆栈指针始终都是一个页面的整数倍。所以，当编译器发现堆栈指针位于保护页面之下时，会插入堆栈检查函数，改变堆栈指针及保护页面。这样，当程序运行时，就会分配物理内存，而不会出现访问违规。

        **使用例子**

**改变堆栈默认大小:**

有两个方法，一是在CreateThread()时传一个参数进去改变；

二是通过链接命令：

#pragma comment(linker,"/STACK:102400000,1024000")

第一个值是堆栈的保留空间，第二个值是堆栈开始时提交的物理内存大小。本文将堆栈改变为100M。

**堆栈溢出处理：**

        如果出现堆栈异常不处理，则导致线程终止；如果你只做了一般处理，内 存

        结构已经处于破坏状态，因为已经没有保护页面，系统没有办法再抛出堆栈溢

        出异常，这样的话，当再次出现溢出时，会出现访问违规操作

        STATUS\_ACCESS\_VIOLATION，这是线程将被系统终止。解决办法是，恢复

       堆栈的保护页面。请看以下例子：

       C++程序如下：

bool handle=true;

            static MEMORY\_BASIC\_INFORMATION mi;

            LPBYTE lpPage;

            //得到堆栈指针寄存器里的值

            \_asm mov lpPage, esp;

            // 得到当前堆栈的一些信息

            VirtualQuery(lpPage, &mi, sizeof(mi));

            //输出堆栈指针

            printf("堆栈指针=%x/n",lpPage);

            // 这里是堆栈的提交大小

            printf("已用堆栈大小=%d/n",mi.RegionSize);

            printf("堆栈基址=%x/n",mi.AllocationBase);

            for(int i=0;i<2;i++)

            {

                        \_\_try

                        {

                                    \_\_try

                                    {

                                                \_\_try

                                                {

                                                            cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

                        //如果是这样静态分配导致的堆栈异常，系统默认不抛出异常，捕获不到

                                                            //char a[1024\*1024];

                                                //动态分配栈空间，有系统调用Alloca实现，自动释放

                                                            Add(1000);

                                                            //系统可以捕获违规访问

                                                            int \* p=(int\*)0xC00000000;

                                                            \*p=3;

                                                            cout<<"执行结束"<<endl;

                                                }

                                                \_\_except(GetExceptionCode()==STATUS\_ACCESS\_VIOLATION ? EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

                                                {

                                                            cout<<"Excpetion 1"<<endl;

                                                }

                                    }

                                    \_\_except(GetExceptionCode()==STATUS\_STACK\_OVERFLOW ? EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

                                    {

                                                cout<<"Exception 2"<<endl;

                                                if(handle)

                                                {

                                                //做堆栈破坏状态恢复

                                                            LPBYTE lpPage;

                                                            static SYSTEM\_INFO si;

                                                            static MEMORY\_BASIC\_INFORMATION mi;

                                                            static DWORD dwOldProtect;

                                                            // 得到内存属性

                                                            GetSystemInfo(&si);

                                                            // 得到堆栈指针

                                                            \_asm mov lpPage, esp;

                                                            // 查询堆栈信息

                                                            VirtualQuery(lpPage, &mi, sizeof(mi));

                                                            printf("坏堆栈指针=%x/n",lpPage);

                                                            // 得到堆栈指针对应的下一页基址

lpPage = (LPBYTE)(mi.BaseAddress)-si.dwPageSize;

                                                            printf("已用堆栈大小=%d/n",mi.RegionSize);

                                                            printf("坏堆栈基址=%x/n",mi.AllocationBase);

                                                            //释放准保护页面的下面所有内存

                                                            if (!VirtualFree(mi.AllocationBase,

(LPBYTE)lpPage - (LPBYTE)mi.AllocationBase,

                                                                        MEM\_DECOMMIT))

                                                            {

                                                                        exit(1);

                                                            }

                                                            // 改页面为保护页面

                                                            if (!VirtualProtect(lpPage, si.dwPageSize,

                                                                        PAGE\_GUARD | PAGE\_READWRITE,

                                                                        &dwOldProtect))

                                                            {

                                                                        exit(1);

                                                            }

                                                }

                                                printf("Exception handler %lX/n", \_exception\_code());

                                    }

                        }

                        \_\_except(EXCEPTION\_EXECUTE\_HANDLER)

                        {

                                    cout<<"Default handler"<<endl;

                        }

            }

            cout<<"正常执行"<<endl;

            //分配空间，耗用堆栈

            char c[1024\*800];

            printf("c[0]=%x/n",c);

            printf("c[1024\*800]=%x/n",&c[1024\*800-1]);

}

void ThreadStack::Add(unsigned long a)

{

            //深递归，耗堆栈

            char b[1000];

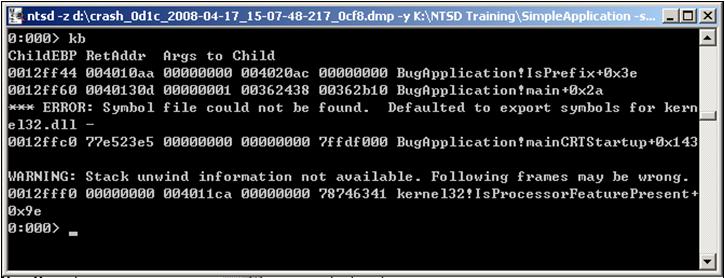
            if(a==0)

            return;

            Add(a-1);

}

程序运行结果如下：



可以看见，在执行递归前，堆栈已被用了800多K，这些是在编译时就静态决定了。它们不再占用进程空间，因为堆栈占用了默认的1M进程空间。分配是从栈顶到栈底的顺序。

当第一次递归调用后，系统捕获到了它的溢出异常，然后堆栈指针自动恢复到原来的指针值，并且在异常处理里，更改了保护页面，确保第二次递归调用时不会出现访问违规而退出线程，但是，它仍然会导致堆栈溢出，需要动态的增加堆栈大小，本文没有对这个进行研究，但是试图通过分配另外内存区，改变堆栈指针，但是没有奏效。

注意：在一个线程里，全局变量加上任何一个函数里的临时变量，如果超过堆栈大小，当调用这个函数时，都会出现堆栈溢出，这种溢出系统不会抛出堆栈溢出异常，而直接导致线程退出。

对于函数1调用函数2，而函数n-1又调用函数n的嵌套调用，每层调用不算临时变量将损失240字节，所以默认线程最多有1024\*(1024-2)/240=4360次调用。加上函数本身有变量，这个数目会大大减少。