通过借助虚拟内存，在内存不足时也可以运行程序。例如，在只剩下 5MB 内存空间的情况下也能运行 10MB 大小的程序。不过，就如本章开头所讲述的那样，CPU 只能执行加载到内存中的程序。虚拟内存虽说是把磁盘作为内存的一部分来使用，但实际上正在运行的程序部分，在这个时间点上是必须存在在内存中的。也就是说，为了实现虚拟内存，就必须把**实际内存**（也可称为**物理内存**）的内容，和磁盘上的虚拟内存的内容进行部分置换（swap），并同时运行程序。

刚才已经说过，Windows 提供了虚拟内存机制作为操作系统。在当前的 Windows 中，虚拟内存依然发挥着很大的作用。虚拟内存的方法有**分页式**和**分段式**[①]两种。Windows 采用的是分页式。该方式是指，在不考虑程序构造的情况下，把运行的程序按照一定大小的页（page）进行分割，并以页为单位在内存和磁盘间进行置换。在分页式中，我们把磁盘的内容读出到内存称为 Page In，把内存的内容写入磁盘称为 Page Out。一般情况下，Windows 计算机的页的大小是 4KB。也就是说，把大程序用 4KB 的页来进行切分，并以页为单位放入磁盘（虚拟内存）或内存中（图 5-3）。

为了实现虚拟内存功能，Windows 在磁盘上提供了虚拟内存用的文件（page file，**页文件**）。该文件由 Windows 自动做成和管理。文件的大小也就是虚拟内存的大小，通常是实际内存的相同程度至两倍程度。通过 Windows 的控制面板，可以查看或变更当前虚拟内存的设定。

下面就让我们来看一下虚拟内存的设定。作者自己的计算机是 2GB 的内存。当前的页文件的大小是 2345MB ≒ 2GB（图 5-4）。

① 分段式虚拟内存是指，把要运行的程序分割成以处理集合及数据集合等为单位的段落，然后再以分割后的段落为单位在内存和磁盘之间进行数据置换。

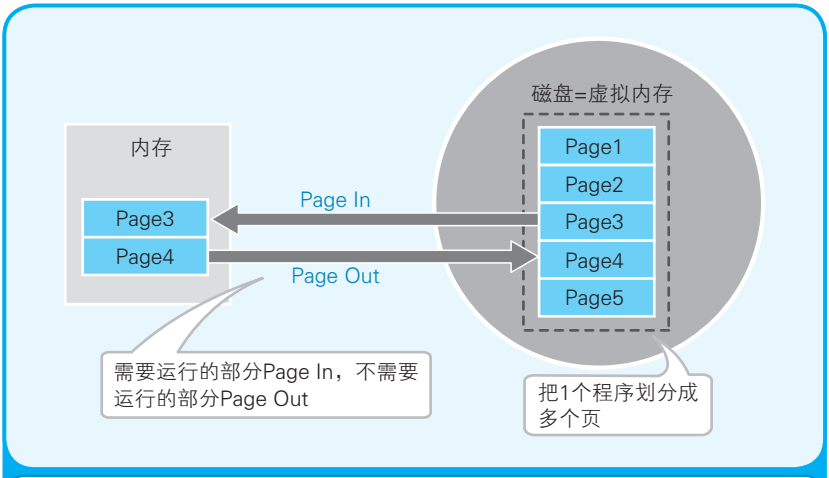

图 5-3　分页式虚拟内存的机制

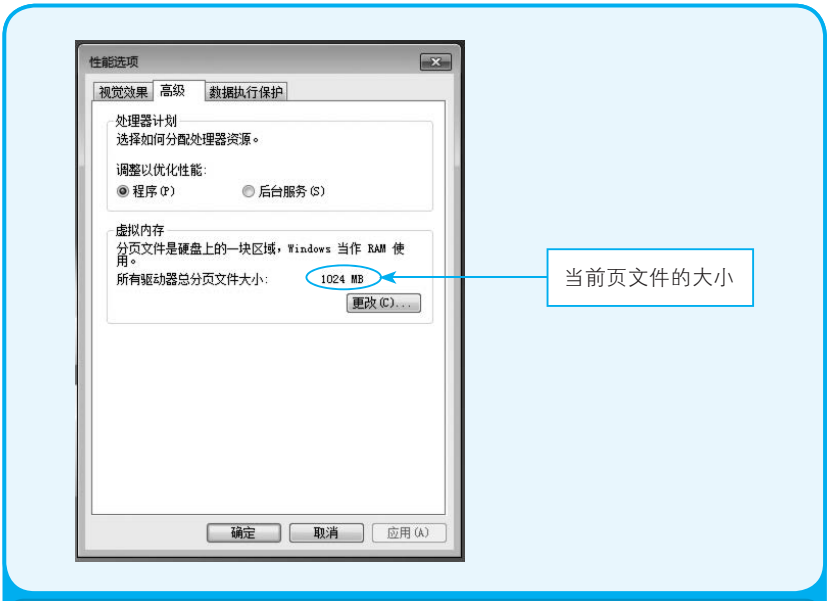

图 5-4　查看虚拟内存的设定

5.4　节约内存的编程方法

以图形用户界面（GUI，Graphical User Interface）[①]为基础的Windows，可以说是一个巨大的操作系统。Windows的前身是MS-DOS操作系统，最初版本可以在128KB左右的内存上运行，而想要Windows流畅运行的话，至少需要512MB的内存。而且，由于Windows具有多任务功能，在巨大的Windows操作系统中可以同时运行多个应用，因此，即使是512MB的内存，有时也无法保证流畅运行。Windows操作系统经常为内存不足所困。

许多人可能会认为，通过借助磁盘虚拟内存就可以解决内存不足的问题。而虚拟内存也确实能避免因内存不足导致的应用无法启动。不过，由于使用虚拟内存时发生的Page In和Page Out往往伴随着低速的磁盘访问，因此在这个过程中应用的运行会变得迟钝起来。想必大家也都有过在操作应用的过程中硬盘访问灯一直亮着（这时正在进行Page In和Page Out），导致应用一时无法操作的不愉快经历吧。也就是说，虚拟内存无法彻底解决内存不足的问题。

为了从根本上解决内存不足的问题，需要增加内存的容量，或者尽量把运行的应用文件变小。接下来会向大家介绍两个把应用文件变小的编程方法。虽然增加内存容量更为便捷，但是花费也高，所以大家还是需要先看一下口袋里面的银子再来做决定。

① 像Windows这样，窗口的菜单及图表等都可以进行可视化操作的方式称为图形用户界面。Windows的前身MS-DOS操作系统，是由键盘输入命令来进行操作的CLI（命令行界面）。

(1) 通过 DLL 文件实现函数共有

DLL(Dynamic Link Library)**文件**[①]，顾名思义，是在程序运行时可以动态加载 Library(函数和数据的集合)的文件。此外，还有一个需要大家注意的地方，那就是多个应用可以共有同一个 DLL 文件。而通过共有同一个 DLL 文件则可以达到节约内存的效果。

例如，假设我们编写了一个具有某些处理功能的函数 MyFunc()。应用 A 和应用 B 都会使用这个函数。在各个应用的运行文件中内置函数 MyFunc()(这个称为 Static Link，静态链接)后同时运行这两个应用，内存中就存在了具有同一函数的两个程序。但这会导致内存的利用效率降低。所以，有两个同样的函数，还是有点浪费(图 5-5)。

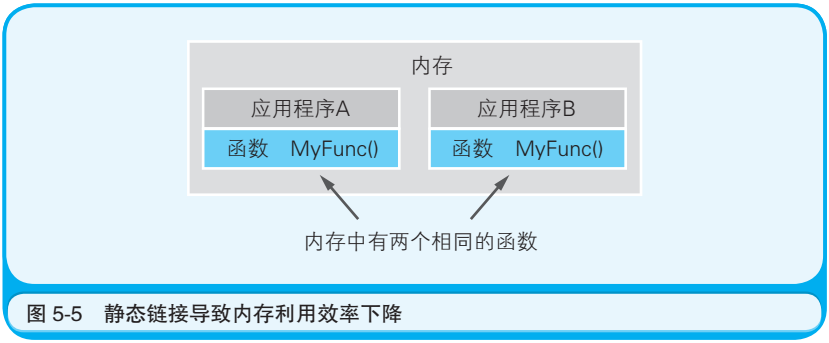

图 5-5 静态链接导致内存利用效率下降

那么，如果函数 MyFunc() 是独立的 DLL 文件而不是应用的执行文件(**EXE 文件**[②])，那结果会怎样呢？由于同一个 DLL 文件的内容在运行时可以被多个应用共有，因此内存中存在的函数 MyFunc() 的程序

① 关于 DLL 文件，会在第 8 章进行详细说明。

② Windows 中，可以执行的应用文件的扩展名是 .exe，这样的文件就称为 EXE 文件。exe 是 executable(可执行)的略写。另一方面，DLL 文件的扩展名为 .dll。

就只有 1 个。这样一来，内存的利用效率也就提高了。

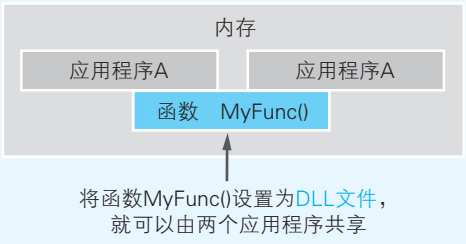

图 5-6　进行动态链接的话即可节约内存

Windows 的操作系统本身也是多个 DLL 文件的集合体。有时在安装新应用时，DLL 文件也会被追加。应用则会通过利用这些 DLL 文件的功能来运行。像这样，之所以要利用多个 DLL 文件，其中一个原因就是可以节约内存。而且 DLL 文件还有一个优点就是，在不变更 EXE 文件的情况下，只通过升级 DLL 文件就可以更新。

(2) 通过调用 _stdcall 来减小程序文件的大小

通过调用 _stdcall[①] 来减小程序文件的方法，是用 C 语言编写应用时可以利用的高级技巧。不过，这一思路应该也可以应用在其他编程语言中，因此大家一定要记住。

① _stdcall 是 standard call（标准调用）的略称。Windwos 提供的 DLL 文件内的函数，基本上都是 _stdcall 调用方式。这主要是为了节约内存。另一方面，用 C 语言编写的程序内的函数，默认设置都不是 _stdcall。C 语言特有的调用方式称为 C 调用。C 语言之所以默认不使用 _stdcall，是因为 C 语言所对应的函数的传入参数是可变的（可以设定任意参数），只有函数调用方才能知道到底有多少个参数，而这种情况下，栈的清理作业便无法进行。不过，在 C 语言中，如果函数的参数数量固定的话，指定 _stdcall 是没有任何问题的。

C 语言中，在调用函数后，需要执行栈清理处理指令。**栈清理处理**是指，把不需要的数据从接收和传递函数的参数时使用的内存上的栈区域中清理出去。该命令不是程序记述的，而是在程序编译时由编译器自动附加到程序中的。编译器默认将该处理附加在函数调用方。

例如，在代码清单 5-1 中，从函数 main() 中调用了函数 MyFunc()。按照默认设定，栈的清理处理会附加在函数 main() 这一方。在同一个程序中，同样的函数可能会被多次反复调用。而如果是同样的函数，栈清理处理的内容也是一样的。由于该处理是在调用函数一方，因此就会导致同一处理被反复进行。这就造成了内存的浪费。

代码清单 5-1　C 语言的函数调用程序示例

```
// 函数调用方
void main()
{
    int a;
    a = MyFunc(123, 456);
}

// 被调用方
int MyFunc(int a, int b)
{
    ...
}
```

虽然通过调查编译器生成的机器语言执行文件就可以得知栈清理的处理内容，不过鉴于原始的机器语言不太容易理解，所以这里我们用汇编语言的代码清单将其显示了出来。将代码清单 5-1 中调用函数 MyFunc() 的部分用汇编语言来表示，就如代码清单 5-2 所示。最后 1 行的处理就是清理处理。

代码清单 5-2　调用 MyFunc() 的部分程序（汇编语言）

```
push 1C8h      ←将参数 456 (= 1c8h) 存入栈中
push 7Bh       ← 将参数 123 (= 7Bh) 存入栈中
call @LTD+15 (MyFunc)(00401014) ←调用 MyFunc() 函数
add  esp, 8    ←运行栈清理
```

C 语言通过栈来传递函数的参数。push[①] 是往栈中存入数据的指令。32 位 CPU 中，1 次 push 指令可以存储 4 个字节的数据。代码清单 5-2 中，由于使用了两次 push 指令把两个参数（456 和 123）存入到了栈中，因此总的来说就是存储了 8 字节的数据。通过 call 指令调用函数 MyFunc() 后，栈中存储的数据就不再需要了。于是这时就通过 add esp, 8 这个指令，使存储着栈数据的 esp 寄存器[②] 前进 8 位（设定为指向高 8 位字节地址），来进行数据清理。由于栈是在各种情况下都可以再利用的内存领域，因此使用完毕后有必要将其恢复到原状态。上述这些操作就是栈的清理处理。另外，在 C 语言中，函数的返回值，是通过寄存器而非栈来返回的。

栈清理处理，比起在函数调用方进行，在反复被调用的函数一方进行时，程序整体要小一些。这时所使用的就是 _stdcall。在函数前加上 _stdcall，就可以把栈清理处理变为在被调用函数一方进行。把代码清单 5-1 中的 int MyFunc(int a, int b) 部分转成 int _stdcall MyFunc(int a, int b) 进行再编译后，和代码清单 5-2 中 add esp, 8 同样的处理就会在函数 MyFunc() 一方执行。虽然该处理只能节约 3 个字节（add esp, 8 是机器语

① CPU 会提前准备好栈机制。往栈中存储数据的汇编语言指令是 push。从栈中取出数据的汇编语言指令是 pop。栈一般是用来实现函数调用机制的。如果想任意利用栈，程序员就需要自己用程序来实现所需要的栈机制。

② CPU 中，栈中堆积的最高位的数据地址是保存在 esp（esp 是 Pentium 系列 CPU 的栈指针名）中的。连续运行两次 pop 指令，可以消除两个存储在栈中的 4 字节数据，而同样的功能也可以通过把 esp 的数值加 8 来实现。

的 3 个字节）的程序大小，不过在整个程序中还是有效果的（图 5-7）。

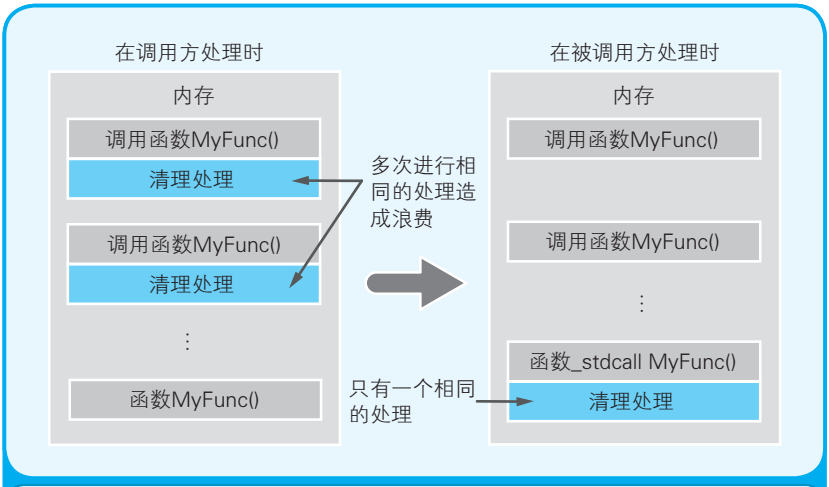

图 5-7　在被调用方进行清理处理可节约内存

5.5　磁盘的物理结构

第 4 章中我们介绍了内存的物理结构，本章就让我们来看一下磁盘的物理结构。磁盘的物理结构是指磁盘存储数据的形式。

磁盘是通过把其物理表面划分成多个空间来使用的。划分的方式有**扇区方式**和**可变长方式**两种，前者是指将磁盘划分为固定长度的空间，后者则是指把磁盘划分为长度可变的空间。一般的 Windows 计算机所使用的硬盘和软盘，采用的都是扇区方式。扇区方式中，把磁盘表面分成若干个同心圆的空间就是**磁道**，把磁道按照固定大小（能存储的数据长度相同）划分而成的空间就是**扇区**（图 5-8）。

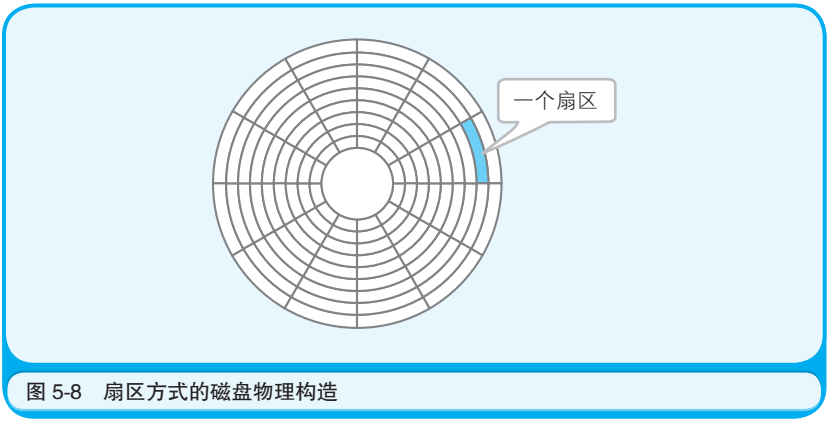

图 5-8　扇区方式的磁盘物理构造

扇区是对磁盘进行物理读写的最小单位。Windows 中使用的磁盘，一般 1 个扇区是 512 字节。不过，Windows 在逻辑方面（软件方面）对磁盘进行读写的单位是扇区整数倍**簇**。根据磁盘容量的不同，1 簇可以是 512 字节（1 簇 = 1 扇区）、1KB（1 簇 = 2 扇区）、2KB、4KB、8KB、16KB、32KB（1 簇 = 64 扇区）。磁盘的容量越大，簇的容量也越大。不过，在软盘中，1 簇 = 512 字节 = 1 扇区，簇和扇区的大小是相等的。

不管是硬盘还是软盘，不同的文件是不能存储在同一个簇中的，否则就会导致只有一方的文件不能被删除。因此，不管是多么小的文件，都会占用 1 簇的空间。这样一来，所有的文件都会占用 1 簇的整数倍的磁盘空间。我们可以通过试验来确认这一点。

由于在硬盘上做试验比较麻烦，所以我们选择在软盘上进行。首先，把软盘按照“1.44MB，512 字节 / 扇区”进行格式化。软盘中，1 扇区 = 1 簇。格式化完成后，我们可以看一下磁盘的属性，这时的已用空间应该是 0 字节，因为没有存储任何文件（图 5-9）。

图 5-9 格式化后磁盘的已用空间是 0 字节

接下来，让我们用记事本等文本编辑工具[①]做成一个只有 1 个半角文字的文件，并将其保存到软盘中，然后再来看一下磁盘的属性。这时我们就会发现，虽然文件的大小只有 1 字节，但使用空间却变成了 512 字节。

再次打开上述文件，并增加一些文字，然后覆盖保存。这时再查看一下磁盘的属性就会发现，当文件大小未达到 512 个半角文字（=512 字节）时，已用空间一直是 512 字节。一旦达到 513 个文字，已用空间就会一下子变成 1024 字节（=2 簇）。通过这个实验，想必大家都应该明白磁盘的数据保存是以簇为单位来进行了吧（图 5-10）。

① 文本编辑工具指的是像简易的文字处理机那样可以输入文字的应用。标准的 Windows 中都带有记事本（notepad.exe）这一文字编辑工具。