HOPE

DOS
内存驻留技术

- PC中断系统
- DOS的弱任务性、实时性
- BIOS、DOS的重入分析
- 内存驻留的设计标准、安全要素、安装、检测与删除
- 热键、弹出式程序编程原理
- 驻留程序的磁盘I/O技术等

李振格
汪明里
编著

中国科学院希望高级电脑技术公司

IBM PC 程序设计高级专题

内存驻留技术

李振格　汪明坚　著

- PC 中断系统
- DOS 的弱多任务性、实时性
- BIOS、DOS 的重入分析
  - 系统内部堆栈
  - Control-Break、严重错误处理程序和 IN_DOS 标志
  - 键盘中断、磁盘中断、时钟中断与 BIOS、DOS 的关系
- 内存驻留程序(TSR)的设计标准
- 内存驻留程序的安全要素
- 内存驻留程序的安装、检测与删除
- 驻留程序的触发器
  - 键盘触发
  - 时钟触发
  - 其它硬件中断(如串口 RS-232)触发
- 热键、弹出式程序编程原理
- 驻留程序的显示技术
- 驻留程序的磁盘I/O技术

与操作系统并存的程序如 CCDOS SideKick SuperKey
可扩展 DOS 的功能、提高 DOS 的多任务、实时处理性能

中国科学院希望高级电脑公司

一九九一年元月

■ 北京市新闻出版局
准印证号：3192—90192
■ 订购单位：北京8721信箱资料部
■ 邮　　码：100080
■ 电　　话：2562329
■ 传　　真：01—2561057
■ 乘　　车：320、332、302路
车至海淀黄庄下车
■ 办公地点：希望公司大楼一楼
往里走101房间

目 录

第一章　揭开计算机的秘密

§1－0　前言

一本教认写程序的书，通常都希望能够立刻进入程序介绍的地方，而且希望读者能够马上从程序中吸收到重点。这种教法有时确实很有效，因为学习写程序最好的方法就是实际写程序。阅读别人写的程序很容易学习到新的技巧，但这必须有一个先决条件，那就是：对于问题必须深入地了解。我们必须知道：某些解决问题的方法有效，而其它的方法则未必有效。虽然我们未必需要知道所有的细节，但是至少要具备足够开始的条件。

有些人借助强记一些他们也不甚了解的字符来写程序。这种做法并不是不好；但是您必须确定，自己所需要的每一部分细节是否都能在书上找到，或是可以找到人问。而如果您希望从完全不了解进而变成了解时，记忆就没有太大的帮助了。

有些人学过修理摩托车，他们可能是把螺丝旋紧、换换管子，或是做一些自己也不知道的动作，就可以把车修好。但是如果要这些人使用别种汽缸设计出新的车子时，似乎就不大可能。因为设计车必须要了解汽缸的工作原理。

本书是针对希望了解 IBM PC 内部奥秘的读者而写的。您可以利用本书所介绍的一些程序，设法提高 PC 的功能，以便在某些特殊应用上帮助您。这不是一本基础的简介书籍，我们假设您对于汇编语言已经略有所知了。本书中有许多例子，您可以选择某些例子来编译，并且实际地执行；但是如果您希望从中获得更多的话，不妨把例题拆开来，然后略做修改，再把它们结合起来，看看结果如何。

本书的程序在编写时都遵循以下两项条件：清晰的设计思路和容易阅读，但是在速度上和大小上则未必是最佳。也许您在阅读过程序之后会说：“我可以让程序减化”。那么不妨自己动手试试。本书的程序并不保证，在任何情况下都可以正确地工作。您可以检查程序中的每一部分，然后问自己：“如果在这种情况时，会怎么样呢？”因此在读完本书之后，您若能够改写某些程序，使得程序执行得更快，程序更清晰，或是使程序的功能更强时，那么您就达到阅读本书的目的。

§1－1　系统程序和应用程序

介绍如何编写计算机程度的基础教科书中，大部分都会有一节专门说明“应用程序和系统程序的差别”。应用程序通常都是用来解决某一些特定的问题。譬如，您可能希望把一个文件中的名字根据字母顺序做排序，或是计算π值等。系统程序的本质则不相同。您的计算机上有一个核心程序永远都存在。这个核心程序提供了基本服务，以便让您执行自己所选择的应用程序；换句话说，系统程序是管理系统资源，而应用程序则是资源的使用者，系统程序长久地存在，应用程序则是暂时存在而已。

上面对于系统程序和应用程序的观点相当的普遍，而且多年来也一直成功地区分出两者的功能。但这只是一个设计者心中所想象的模式而已，并非铁的定律。通常，所有的

计算机都有系统， 而这些系统程序可以用来执行应用程序；但是在一台个人计算机上，为什么一定要把这两者划分的如此清楚，甚至造成不便呢？ 为什么不能让一个应用程序永远存在内存中，譬如：一个随时可以调出的工作计划？ 您也许希望某些应用程序永远可能随时昭唤。这种情况下，传统上对于应用程序和系统程序的区分模式就不适用了。

个人计算机和其它种类的计算机有一点主要差别，那就是：个人使用。它是您的计算机，您可以让它为您做任何事，买一台个人计算机就像是买一间房子。当您买回来时，它和别人的一模一样。但是它是您的，因此您可以随自己的意思装潢；最后就变成最适合自己需要的计算机。

§1－2 解决问题

如果您只是为了可以修改就动手去做的话，理由似乎不够充分。 当您在拆掉一片墙之前，一定会先仔细想清楚；同理，为您的计算机系统加入新的特性时， 也必须要很小心。在某一范围中加入的新功能也许不知不觉对别的方面产生不良的副作用。 这就好像是敲掉一片墙时，也许这片墙就是梁柱所在。

通常加入到操作系统的程序必须具备以下的特性：

- 无论您正在做什么，都可以取得使用。当您对系统做改变时，往往必须付出代价。每当您加入一项新的特性时，也许在执行效率、可用内存、或是磁盘空间上都会付出代价。新加入的特性必须在这些因素都列入考虑时，值得才可以。
- 降低您解决问题时必须花费的工夫。
  您绝对不可能加一项新功能，让您把原先可以容易解决的问题变成很难解决。在操作系统中加入一个程序帮助您计算π值也许很有意思，但是并不能帮助您写出更多的程序，您也不可能经常用到。只有您经常用到的东西写成系统程序才可能帮助您顺利开展工作。
- 帮助您执行原先很难做到的事情。
  通常您在应用程序中很难取得某些资料。如果您想知道自己使用的电传程序，如何设置串行口的状态，您就必须知道电传程序如何执行，这一点并不容易做到。而您若离开电传程序的话，原先所设定的状态很可能就改变了。此时您可能需要本书提供的内存驻程序来帮您忙。

§1－3 优良设计的特性

“品质”很难准确地定义。每一个人都想写出好的程序，但是什么样的程序才算是好的程序呢？每当您想出一项自认为是可能使程序“好”的标准时，可能就有人提出一个例子来反驳您的理论。但是，应该有一些建立优良程序的准则，而这些准则也应该不是很烦琐才对。

优良的程序必定能够做到它想做的事。不能正确执行的程序一定不是好的程序，无论程序写的多优美都是一样的。另外好的程序只做它该做的事，其它的事情绝对不予理会。譬如，如果一个程序必须列出目录的内容时，可是却在磁盘上写入一些垃圾数据，即使写入一点点，这个程序都不是好的程序，因为它具有破坏力。

为了确保程序能够产生最小的副作用，我们一定要把程序的目标定义清楚；让您的程序只作一件事，而且完美地达到目的。许多程序人员常常会在程序中多加入一点功能，或是处理略微不同的特殊情况，但这往往造成一些别的缺点。简单地说，使用钳子来转螺丝钉是可以用，但是远是不如使用螺丝起子来得好。

当您的操作系统中加入自己的程序时，就好像是在自己的工具箱中加入一项新的工具一样。如果您有了正确的工具时，您想解决的问题就变得比较容易解决。如果您所拥有的工具不够时，往往就必须采用比较不适当的方法解决问题。

工具太多时当然未必就一定好，因为杂而无章就令人无所适从。但是如果您能够把相类似的工具整理起来放在一起，形成一套特殊的工具箱，那么将来就可以很方便地使用。

§1－4　编写可以调试程序

应用程序在执行过程中都一直受到操作系统的保护。通常当应用程序错误时或是遇到突发状况时，系统程序就会接管，并且收拾残局。

系统程序的生命周期中就危险多了，如果稍有错误就会造成系统死机。除此之外，即使系统程序能够正确地执行时，程序人员也很难知道到底正在执行什么，因为系统程序的特性太复杂了。譬如，如果您改写 IBM PC 上从键盘读取字符的系统程序，但是在调试时死机了，这时候您就只好重新启动计算机，因此如何能知道发生错误的原因呢？双譬如您改写了执行磁盘输入输出的系统程序，而该程序调试时不但死机，而且破坏了 B 磁盘上的启动程序，那么该磁盘甚至都不能启动了。

为了要使系统程序尽可能容易调试，最好能遵循以下的规则：

- 不要自作聪明！一旦您想出解决问题的正确方法之后，就根据所想出来的方法，继续做下去。保持前后一致可以帮助您节省许多时间和金钱。虽然每一次都使用相同的方法解决同一问题，未必最好，但却能保证正常工作。
- 许多人一味地精减程序，或是使程序执行得更快，但却往往造成许多错误。如果某件事情只做一次时，其执行的速度并不是十分重要。如果您只针对 20 项数据做排序时，使用简单的排序方法可能和快速的排序方法费时相同；因此使用简单的排序方法可能更节省写程序的时间。
- 您也许现在能够了解某一段程序，但是两个月之后再回顾看时，是否必须做修改扩充时呢？因此如果您尽可能用最浅显的方式来写程序，将来您会发现对自己的帮助很大。
- 千万不要想一步登天！您必须一步一步来，才可能完成完整的程序。譬如，如果您希望写一个“常驻内存的文字编辑器”时，那就先写好一个一般的文字编辑器，然后再设法变成常驻程序。
- 按照模块的方式来编写程序。解决问题时最好把自己当成切割钻石的专家。如果您能够碰到问题的重点，就可能把问题分成清楚的模块；但是如果敲错地方，把问题分成细细碎碎的砂子时，就很难处理。
- 有时候您可能把问题简化，然后使用较简略的程序先写出粗略的应用程序。您可以模拟输入输出的程序，然后先使用固定的参数试试程序可否工作。最后再写

也可能让您设定不同参数的程序。您不需要一下子就写好程序。 把程序分成不同部分，然后让每一部分通过测试，可能很快就产生自己需要的程序。 这样子除了可能使用一些标准形态的程序以节省时间外， 而且您也可以很容易发现一些自己原先没想到的地方。

- 要使一个复杂的程序可以完全正确地工作前，需要经历两次设计。 当我们写出可以工作的程序稚型之后，最好能修改成结构比较好的最后版本， 这样做也可以帮助您进一步了解整个程序。第二次设计出来的程序会更清晰。 如果您未曾经历再一次设计，往往容易自陷于目光短浅的错误。
- 程序人员的必须具有以下的信念："开发出创新性的方法，节省更多时间"。这句话的意思是，程序人员必须能观察出重复性的工作， 并且设法使这些工作变成自动化。如果您可以把许多工作使用程序变成自动执行， 您就可以有更多的时间来创造。但是您也不能够花一百个小时把一件工作自动化， 可是那件工作原先只是每个月浪费您一个钟头而已。也许那件工作自动化很有趣， 但是却不值得去做。

每个人写程序时都会开发出一套自己解决问题的方法，这就好象写书法， 或是画图一样； 您也会渐渐产生一套自己的方法。许多资深的程序人员也许曾经有过以下的经验：他看到一段程序，觉得风格和自己很相似，但却没有认出，事实上就是自己写出来的。写程序时通常是受到当时的思想和心情所左右。

好的程序人员可以写出优美的程序， 而伟大的程度人员则可以快速地写出优美的程序。当然，两者所写出的程序都可以正确地执行。

第二章　基本原理

§2－0　前言

如果您从来没有在 IBM PC 上写过任何程序，而您也有自知之明，避免如此做的话，那么本书就不适合您阅读。

如果您曾经使用过 Pascal，C，或是 FORTRAN 写过一些简单的程序，但是却没有听过汇编语言(Assembly language)，只要您有兴趣的话，那么不妨尝试一下。当您遇到问题时，千万别丧气，请翻到附录 D，其中列有许多参考书籍，这些书籍对于编写汇编语言的程序帮助很大。

如果您曾经写过任何一种汇编语言的程序，即使只是打印出“Hello，world”，只要您能够投入并且从中得到乐趣的话，那么您就已经有了基础，而且足够阅读本书。本书并不是介绍如何编写 IBM PC 的汇编语言程序。如果您需要学习编写汇编语言的程序时，请参考附录 D，其中列出了其它作者所编写的这方面书籍。当您使用本书时，首先必须准备汇编器(assembler)和汇编语言参考手册(assembly language reference manual)，如此才可能尝试本书的程序。当然啦！您还需要一台 IBM PC，以及一些基本的外围设备。

本书有两种阅读方法。您可以坐下来，仔细地阅读完每一章，然后再输入程序玩玩。您也可以一边阅读，一边输入程序，尝试书中的结果是否正确。本书不仅要教您如何编写内存常驻程序；同时也希望让您学会设计良好的程序。

§2－1　基础

虽然本书并不是教您编写汇编语言的入门教科书，但是本章准备花些时间，介绍一些基本概念。任何计算机都有一组它自己才能够了解的基本指令。这些基本指令通常都很简单，譬如：“把 6 移到暂存器 AX 中”；或是“如果 ZF(Zero Flag)等于 1，就执行地址 1234 的指令”。

计算机并不了解以上的句子。事实上，上面的句子都是使用一组位(bits)来表示，计算机只了解所表示的二进制数字而已。计算机的线路设计成：可以不断地从内存读取二进制的数据，然后根据二进制数据的内容，执行一定的动作。计算机所执行的基本操作都很简单，但是如果组合起来时，就可能变成程序，完成许多事情。

如果您懂得计算机如何根据二进制数字操作的话，那么您就可以直接把二进制数据写到内存中，产生程序。这并不是不可能，因为过去曾经有数以万计的程序人员这样写程序。这和爬高山，或是了解量子力学来比较，并不算是很困难。介是这种做法需要一段长时间的学习，同时也需要一段长时间才能够写出程序。

很久以前，有一批工作辛苦的程序人员想到，要设计出一个程序，让不懂计算机奥秘的人也能写程序。他们发明了汇编器(assembler)，汇编器可能把像英文一样的指令（譬如，MOV AX ,6）转换成二进制的指令。早期的汇编器充满了许多错误，而且很难使用。但是即使如此，使用汇编器来写程序远比最古老的方法好太多了。而且有了汇编器之

后，就可以用来写出更好的汇编器，甚至写出高级语言的编译器。

为什么本书的程序都使用汇编语言呢？使用高级语言不是更好吗？ 这个问题没有正确的答案。对于许多种程序而言，使用汇编言语言实在是浪费时间。 但是对于有些程序而言，，却又非得使用汇编语言不可。为什么呢？答案是取舍的标准何在。 编写任何程序时，都有一些取舍存在。任何程序所呈现出来的结果， 绝非是设计时唯一的标准；事实上，也许是从数以千种决定，做取舍后的结果。

当一种程序语言被设计出来时，其本身的特性将影响到它的应用范围。譬如， 高级语言为了达到移植容易的特性，很可能就必须牺牲一些特性；譬如：直接存取内存、连结到中断处理程序等特性。程序设计人员必须判断什么比较重要， 是程序移植重要呢？还是完成其他的任务重要？这一点说明了， 为什么内存常驻程序需要用汇编语言编写。

使用汇编语言编写内存常驻程序可以让计算机发挥最大的效用。除此之外，使用汇编语言所编写的程序，可以尽量地使程序变小，这样子才不会占用太多有限的内存空间。

§2－2 8086/8088

IBM PC 中央处理单元(central processing unit)是微处理器 Intel 8088，8088 是 8086 最小的版本。 对于编写程序而言，两者几乎完全相同。两者之间的差别是在于：它们对外的沟通。8086 和外界沟通时是经由 16 位的输入输出通道，内存存取时也是每次以 16 位为单位。旧的 8 位微处理器无论在计算能力、或是输入输出通道都只是目前 16 位芯片的一半。IBM 选择了 INTEL 8088 做为中央处理单元，恰好是 8 位和 16 位之间的妥协。8088 的内部结构和 8086 极为相似，但是它和外界沟通时就必须经由 8 位的通道。这项妥协是以硬件价格来换取执行速度；以 8088 为 CPU 的计算系统可以使用较少的零件来组装。但是执行程序的时间就比较长。IBM 认为这是一项明智的选择，因此 PC 就诞生了。当硬件价格降低时，其它装配 8086 的 PC 兼容产品就出现在市面上。这些产品的执行速度增加了，但是原先的软件依然可以使用。

§2－2－1 寄存器

8086/8088 的结构简单。其中包含了一组一般用途的 16 位寄存器，分别是：AX，BX，CX，DX，BP，SI，DI。其中 AX，BX，CX 和 DX 还可以分成 8 位的寄存器，譬如：AX 就可以分成 AH 和 AL，BX 就可以分成 BH 和 BL，CX 就可以分成 CH 和 CL，DX 则可以 DH 和 AL。寄存器 BP，SI，DI 的用途也没有特别限制，但是却不能分成两个字节。另外寄存器 SP 主要是用来当做堆栈指针。除此之外，还有四个非常重要的段寄存器(segment register) ：CS，DS，SS 和 ES。指令指针(Instruction pointer)IP 是用来控制目前 CPU 执行到哪一个指令。

8086 设计时考虑到要和 8 位的 CPU 8080 兼容。8 位的计算机是使用两个字节（亦即 16 位） 来定址，因此其定址空间可以到达 64K 字节。16 位的 CPU 在地址设定上选择了完全不同的方法。CPU 以段(segment)为单位，每一段范围内包括 64K 字节，而内存中则可以包含许多段。因此，操作系统可以在一个段内执行。而使用者的程序则可以在另一个段内执行。在一个段内，程序可以把计算机视为只有 64K 字节内存空间。因此原先 8 位计算机上执行的程序就可以很容易地移植到 16 位计算机上。除此之外，内存段也可以

彼此重叠，因此两个不同的程序就可以共用某一块内存。Intel 选择“段落”(paragraph)做为段内的基本单位。如果您把段指针由 0700 往前移到 0701 时，就会把内存位置往前移动 16 个字节，也就是一个段落。事实上，段值是以段寄存器来设定的，而实际的地址值则是把段值（16 位）往左移 4 个位，然后再加上 16 位的位移(offest)，因此构成 20 位的地址值。因此 8086 可以直接做 20 位的地址，也就是可能存取到一兆字节的内存。在这 1M 字节的内存中，IBM PC 保留了最前面的 320K 字节给系统的 ROM BIOS 和显示内存等，因此使用者最多也中能使用 640K 字节。

编写程序的程序人员必须考虑到四个段。 第一个段是代码段（code segment 简称为 CS）它是放在寄存器 CS 之中，这个段是实际执行程序代码所放的地方。第二个段是数据段，它是放在寄存器 DS 之中。当一个程序要从内存读取数据，而且没有指示从哪一个段读取时，CPU 就当成是从数据段读取数据。第三个段是堆栈段，它是放在寄存器 SS 之中。 堆栈操作指令 PUSH 和 POP 都是针对这个段进行。最后一个是附加段，它是存放在寄存器 ES 之中。这个段寄存器当做一般用途使用，由程序人员决定该如何使用。

许多计算机在设计时都让所有的寄存器可以使用相同的方式读取和写入。计算机设计师把这种设计叫做垂直设计(orthogonal design)。但是 8086 不是使用这种方式设计。在 8086，对于某些寄存器做存取是无法操作的。譬如，程序人员可以把数字直接搬到一般用途的寄存器中，但是却不能搬到段寄存器之中。这表明，您只要使用一个单一指令就可以把 BX 设定成 6：

```
mov     Ax , 6
```

但是把 DS 设定成 6， 却需要两个指令，因为段寄存器的数值只能使用其它的一般寄存器来设定。

```
mov     bx , 6
mov     ds , bx
```

您在本书中会看到许多类似的例子。

§2－2－2 2寻址方式

寻址方式 (addressing mode)是一台计算机上许多复杂操作的重要关键所在。8086 提供了以下几种寻址方法：立即寻址，内存间接寻址，寄存器间接寻址等。这里只介绍几种常用的方法。

立即寻址　　这种方式是直接使用数字，譬如前面例子中的 6。

内存间接寻址　数值存放在数据段中的某个位置。譬如：

```
        mov     bx , foo
         .
         .
foo     dw      6
```

寄存器间接寻址，有两种寄存器可以使用在这种寻址方式下，它们分别是基址寄存器(Base Register)和索引寄存器(Index Register)。基址寄存器分别是 BX 和 BP，索引寄存器则是 SI 和 DI。在这种寻址方式下，寄存器存放了数据段中的地址值。譬如：

```
mov     ax , 0F000H
mov     es , ax
mov     si , oFFFEH
mov     dl , byte ptr es : [si]
```

上面的程序使用间接寻址方式，由寄存器 SI 读出位于 F000:FFFE 位置的数据。寄存器间接存取时，最多只能使用到一个基址寄存器和一个索引寄存器。

以上的寻址方式可以做不同的结合。因此组合后的结果很多。但是以上所介绍的寻址方式是本书使用最多的方法。请参考有关 IBM 汇编语言的书籍，其中会有更详细的介绍。

§2－2－3 标志

8086 有 9 个一位的标志(flag)，它们可以用指示 CPU 的各种状态。以下是这 9 个标志的简介：

CF (Carry flag) CF 为 1 时就表示算术运算的结果超出正确的长度。

PF (Parity flag) PF 为 1 就表示使用偶校验，PF 为 0 就表示使用奇校验。

AF (auxiliary carry flag) AF 和 CF 相同，只是它使用在低 4 位的结果。A F 通常都使用在 20 位的地址计算上。

ZF (Zero flag) ZF 为 1 就表示运算结果是 0，否则 ZF 就为 0。

SF (sign flag) SF 为 1 就表示运算结果的最高位是 1，否则 SF 就为 0。

TF (trap flag) TF 为 1 ，CPU 就单步地执行。在这种模式之下，每完成一个指令就发生一次特殊的中断。

IF (interrupt-enable flag) IF 为 1 时，允许 CPU 接收外界的中断，否则 IF 就为 0。

DF (direction flag) 这个标志使用在循环指令，譬如：MOVS ，MOVSB, MOVSW, CMPS, CMPSB 和 CMPSW。如果 DF 为 1，循环运行时就使地址值往前增加。如果 DF 为 0 时，循环运行时就使地址值往后减少。

OF (Overflow flag) OF 为 1 表示一个考虑正负号的运算超出正确的字节的长度。

本书大部分都只用到 ZF 和 CF 两个标志。请参考 IBM 的书籍，以了解其它标志的详细用法。

本书所使用到的条件转移(conditional branch)指令，都是以 ZF 和 CF 这两个标志为条件。以 ZF 为转移条件的指令是检查某一条件是否为 0。这是一个程序中，大部分逻辑操作的动作，譬如：检查是否相等，或是计数器是否为一定的数值等条件。8086 有几种转移指令，譬如：

```
cmp     ax , 6
jz      foo
```

就和

```
cmp     ax , 6
je      foo
```

表达了相同的意思。然而后者却比较比较能够表达程序的真正涵义。它可以让您知道，当寄存器 AX 等于 6 时，控制权就转移到 foo 的位置。

以下是另外一种常见到的习惯用法，但是这种用法却不是一定必要，譬如，以下的程序：

```
dec     ax
cmp     ax , 0
je      foo
```

就可以使用以下的程序取代

```
dec     ax
jz      foo
```

在某些情况下，拿来做比较的数字可能是任何数字，而不一定非是 0。上面的第一种情况中，我们是比较一个数字，而第二种情况则不是，以可读性的观点来看，第一个例子比较容易让人了解。第一个例子很明显是把一个计数器减 1，当计数器到达某一个数字之后，就跳到 foo 的地方执行。第二个例子的意思就比较含糊，因此本书的程序将避免使用这一类型的写法。如果您希望改写这样的程序，以节省内存，不妨自己动手试试。

另外一种转移指令是根据 CF 值为条件。本书中很少使用到这种程序。有些 DOS 的功能调用是以 CF 来表示执行成功或失败。如果要以 CF 来执行转移操作时，可以使用 JC (Jump if Carry Set)或 JNC (Jump if Carry Not Set)等指令。

§2－2－4　循环

计算机不会疲倦，这也是很好用的特性之一。我们可以让计算机反复地执行相同的工作。所有的程序语言都有循环(looping)这项基本流程控制的功能。如果一台计算机可以愈快地完成一个循环，其性能就可以认定为愈好。因为循环的用途很大，因此 8086 也有专门执行循环的指令。

所有的循环指令都是以 cx 做为计数器。一个循环会反复地执行，直到 cx 等于某一特定值为止。以下的程序是利用反复地相加，完成两个数的相乘：

```
mov     ax , 0
mov     cx , 4
```

```
next:
        add     ax , 6
        loop    next
done:
```

在上面的程序中，Loop 指令执行时会把 cx 减 1，并且检查 cx 的内容；如果 cx 等于 0，就转移到 done，否则就跳到 next 标示的地方执行。

您可以使用以下的程序，完成相同的工作。

```
        mov     ax , 0
        mov     cx , 4
next:
        add     ax , 6
        dec     cx
        cmp     cx , 0
        jne     next
done:
```

Loop 指令的种类很多，有些是计数器行于 0 时就停止执行循环，有些则刚好相反。转移指令限制了下一个执行指令的位置与转移指令的距离，不能超过-127 到+128 字节之内。这表明，您只可以往回跳 127 个字节，或是往前 128 个字节。

8086 有一个功能强大的循环指令，它可以让您反复地执行一个单一指令。最常见到的循环例子是：处理一连串的字节。譬如，如果您要把 60 个字节的阵列搬到另一个阵列中时，可以使用以下的 Loop 指令完成：

```
        mov     si , offset source
        mov     di , offset dest
        mov     cx , 60
mvloop:
        mov     ds:byte ptr [di] , ds:[si]
        inc     di
        inc     si
        loop    mvloop
```

上面的循环程序写得十分严密，因此应该是十分减化了；但是如果上面的程序很重要时，也许您会希望更进一步减化。8088 提供了一些字串移动的指令，可以完成以上的动作。譬如，您可以使用指令 MOVSB 完成以上的动作：

```
        mov     si , offset source
        mov     di , offset dest
        mov     cx , 60
mvloop:
```

```
movsb
loop    mvloop
```

字串指令执行时，SI 和 DI 的数值会随着 CX 的减少，而递增或递减。方向标志 DF 则决定了是要递增还是递减。如果您使用 CLD 指令把 DF 设置成 0，SI 和 DI 就会递增。如果您使用 STD 指令把 DF 设置成 1，SI 和 DI 就会递减。因此 CLD 指令就相当于“设置方向向前”；而 STD 指令则相当于“设置方向向后”。

在 8086/8088 上，上面的循环可以进一步减化。8086 提供了一个特殊的重复指令，可以进一步地减化以上的循环成为一个单一指令，以下是减化后的结果：

```
mov     si , offset source
mov     di , offset dest
mov     cx , 60
cld
rep     movsb
```

§ 2－2－5　内存内的数据结构

8088 是以字节为存取数据的基本单位。计算机的存贮结构是 8 位的字节，但是 CPU 本身处理数据则是以 16 位的字节为单位。从内存中找出正确的字节往往需要使用一些技巧。

任何计算机的内存都是由较小的存贮单位(bit)组成。在许多计算机上，基本的数据单位是字节(8 bit)，而在其它计算机上则可能是 16 位或 32 位。有些大型的计算机上，基本是数据单位也许不是 2 的几次方，譬如：一个字节是 36 位。但是即使说字节的长度有所不同，这些计算机依然可以分成一些基本种类。如果说您的计算机可以从内存装载一个字节，而不必装整个字之后，舍去不必要的部分，才能读出一个字节时，那么您的计算机就是：字节取址的结构。如果您必须取出一整个字，才有办法读出其中一个字节时，那么您的计算机就是字取址的结构。

8086/8088 是一种字节取址结构的 CPU。您可以直接地取出内存中的个别字节。譬如，我们可以根据以下的格式取出一个字节的内容：

```
mov     si , 1234H
mov     al , [si]
```

上面的指令曾以 SI 所存放的内容为地址(1234H)，把这个地址内的数据搬到寄存器 AX 中。

如果您要把哪个存贮单位的内容设置成 6 时，就可能使用以下的指令去执行：

```
mov     si , 1234H
mov     [si] , 6
```

但是上面的程序在汇编时就会产生错误，因为汇编器无法判断，您是要把 06H 放到