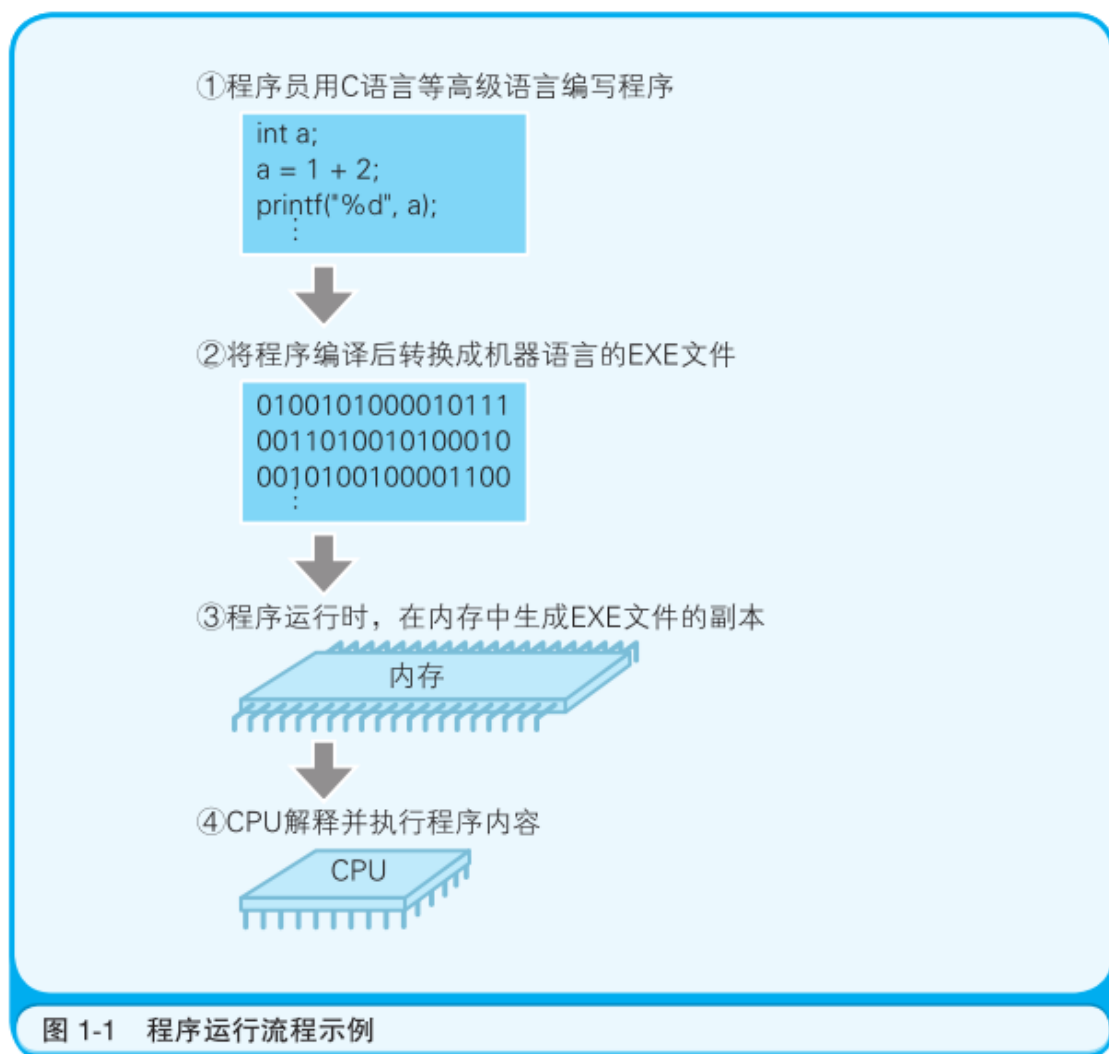


- [CPU](#)
- [数据](#)
  - [用二进制数表示计算机信息的原因](#)
  - [“补数”](#)
  - [逻辑右移和算术右移](#)
- [小数运算](#)
  - [如何避免计算机计算出错](#)
  - [十六进制](#)
- [内存](#)
- [内存和磁盘](#)
  - [虚拟内存把磁盘作为部分内存来使用](#)
  - [节约内存的编程方法](#)
    - [通过 DLL 文件实现函数共有](#)
    - [通过调用 stdcall 来减小程序文件的大小](#)
  - [磁盘的物理结构](#)
- [压缩数据](#)
  - [可逆压缩和非可逆压缩](#)
- [程序是在何种环境中运行的](#)
  - [运行环境 = 操作系统 + 硬件](#)
    - [提供相同运行环境的 Java 虚拟机](#)
  - [BIOS和引导](#)
- [从源文件到可执行文件](#)
  - [计算机只能运行本地代码](#)
  - [编译器负责转换源代码](#)
  - [启动及库文件](#)
  - [DLL文件及导入库](#)
  - [可执行文件运行时的必要条件](#)
  - [程序加载时会生成栈和堆](#)
- [操作系统和应用](#)

- 汇编
  - 汇编语言的语法是“操作码 + 操作数”
- 硬件控制
  - 外围设备的中断请求
  - DMA可以实现短时间内传送大量数据
  - 文字及图片的显示机制
- 思考
  - 程序生成随机数的方法

## CPU



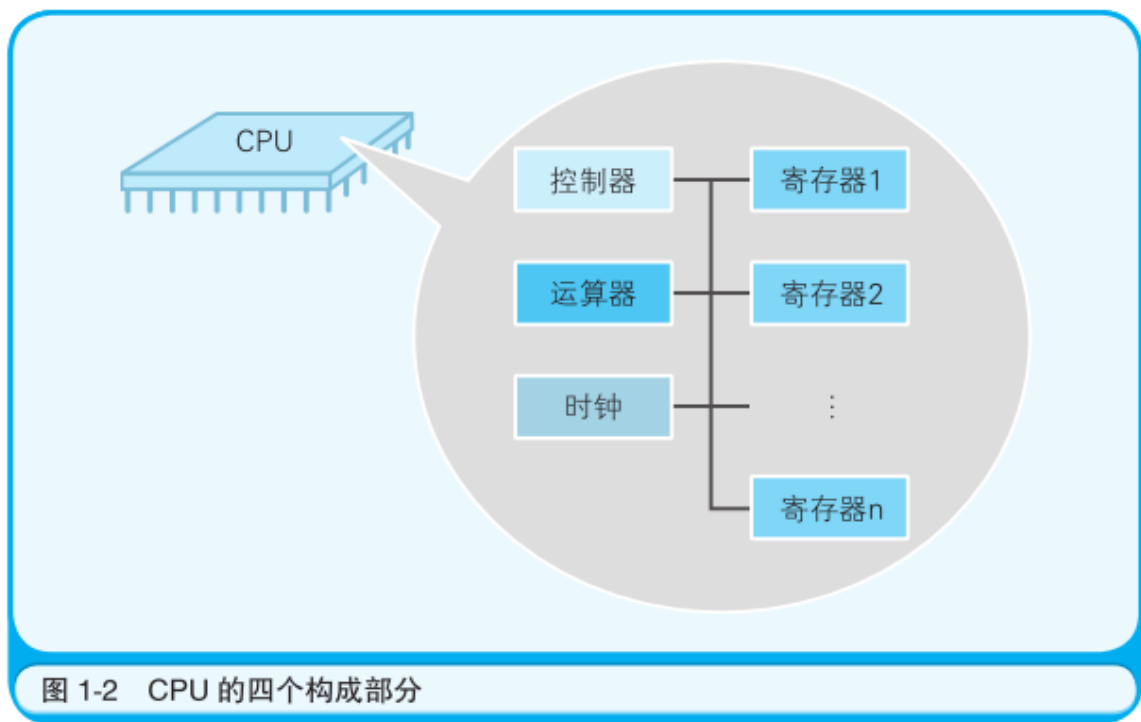


图 1-2 CPU 的四个构成部分

主存通常使用 DRAM（Dynamic Random Access Memory，动态随机存取存储器）芯片。DRAM 可以对任何地址进行数据的读写操作，但需要保持稳定的电源供给并时常刷新（确保是最新数据），关机后内容将自动清除。

程序启动后，根据时钟信号，控制器会从内存中读取指令和数据。通过对这些指令加以解释和运行，运算器就会对数据进行运算，控制器根据该运算结果来控制计算机。

其实所谓的控制就是指数据运算以外的处理（主要是数据输入输出的时机控制）。比如内存和磁盘等媒介的输入输出、键盘和鼠标的输入、显示器和打印机的输出等，这些都是控制的内容。

程序是把寄存器作为对象来描述的。

通常我们将汇编语言编写的程序转化成机器语言的过程称为汇编；反之，机器语言程序转化成汇编语言程序的过程则称为反汇编。

在程序员看来“CPU 是寄存器的集合体”。至于控制器、运算器和时钟，程序员只需要知道 CPU 中还有这几部分就足够了。

寄存器中存储的内容既可以是指令也可以是数据。其中，数据分为“用于运算的值”和“表示内存地址的数值”两种。

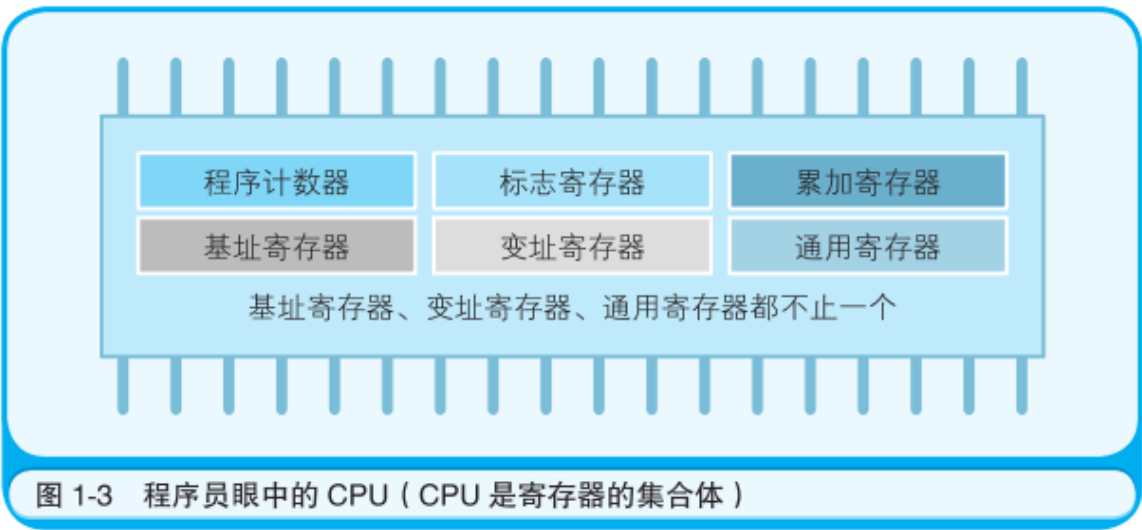
用于运算的数值放在累加寄存器中存储，表示内存地址的数值则放在基址寄存器和变址寄存器中 存储。

表 1-1 寄存器的主要种类和功能

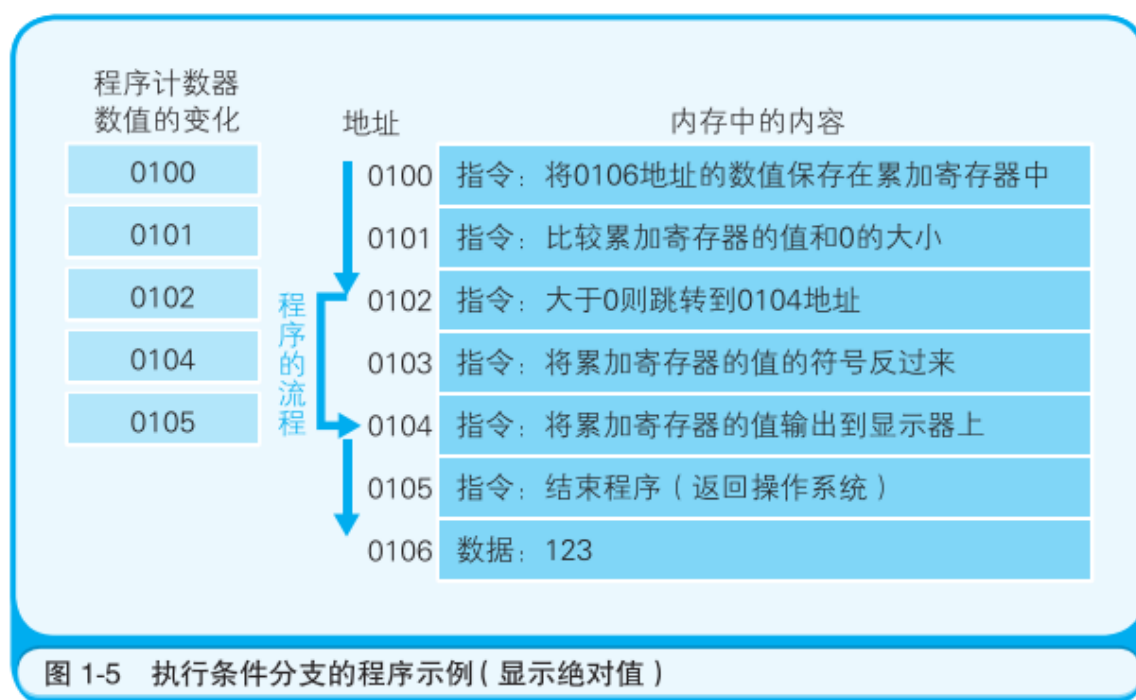
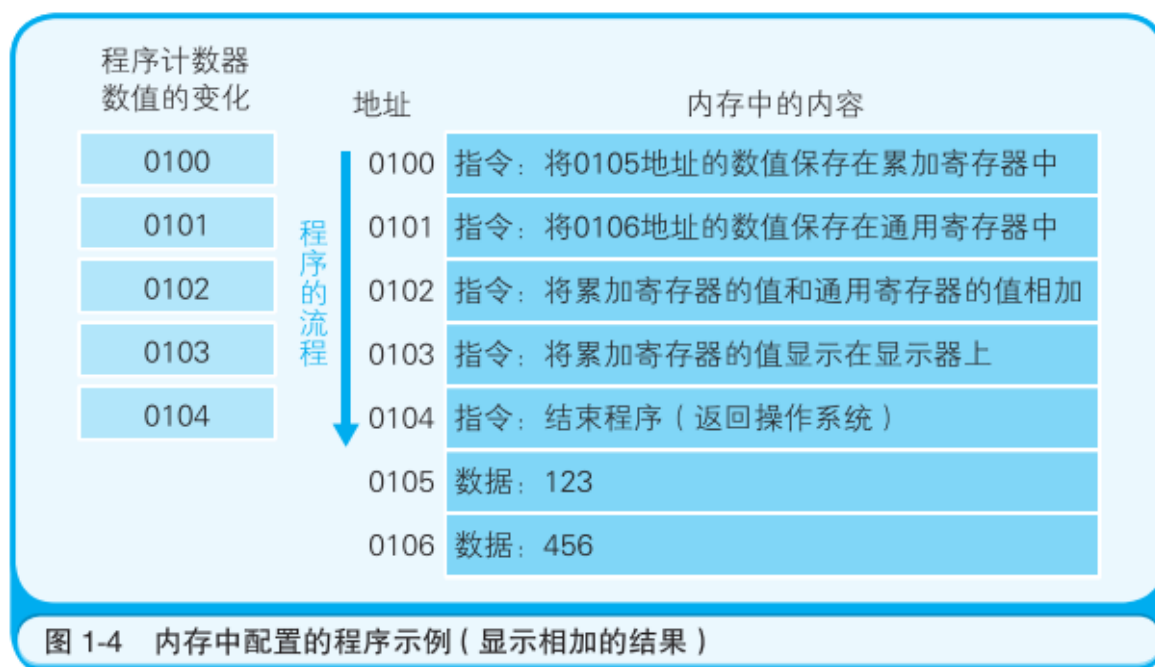
种 类	功 能
累加寄存器 ( accumulator register )	存储执行运算的数据和运算后的数据
标志寄存器 ( flag register )	存储运算处理后的 CPU 的状态
程序计数器 ( program counter )	存储下一条指令所在内存的地址
基址寄存器 ( base register )	存储数据内存的起始地址
变址寄存器 ( index register )	存储基址寄存器的相对地址
通用寄存器 ( general purpose register )	存储任意数据
指令寄存器 ( instruction register )	存储指令。CPU 内部使用，程序员无法通过程序对该寄存器进行读写操作
栈寄存器 ( stack register )	存储栈区域的起始地址

eax 和 ebp 分别是累加寄存器和基址寄存器。

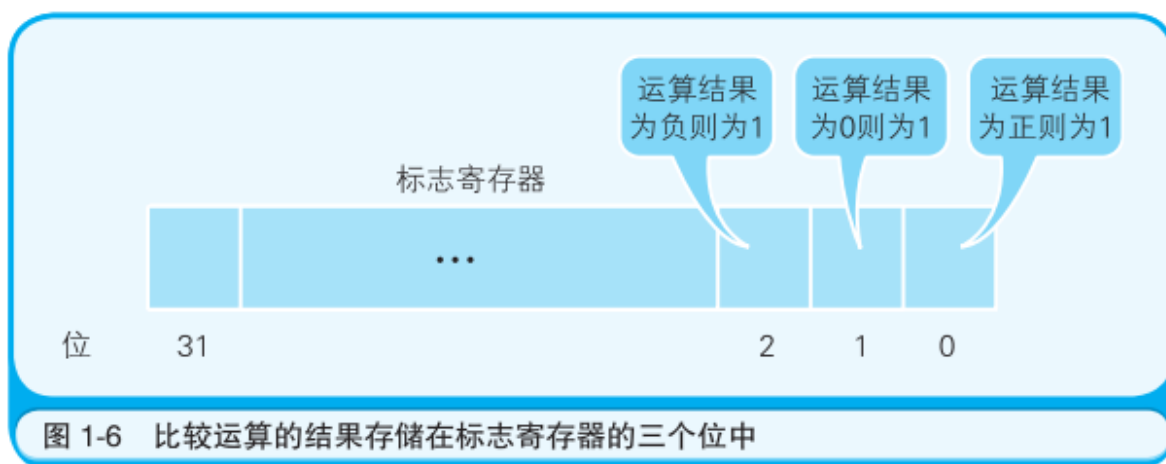
其中，程序计数器、累加寄存器、标志寄存器、指令寄存器和栈寄存器都只有一个，其他的寄存器一般有多个。



地址 0100 是程序运行的开始位置。Windows 等操作系统把程序从 硬盘复制到内存后，会将程序计数器（CPU 寄存器的一种）设定为0100，然后程序便开始运行。CPU 每执行一个指令，程序计数器的值 就会自动加 1。



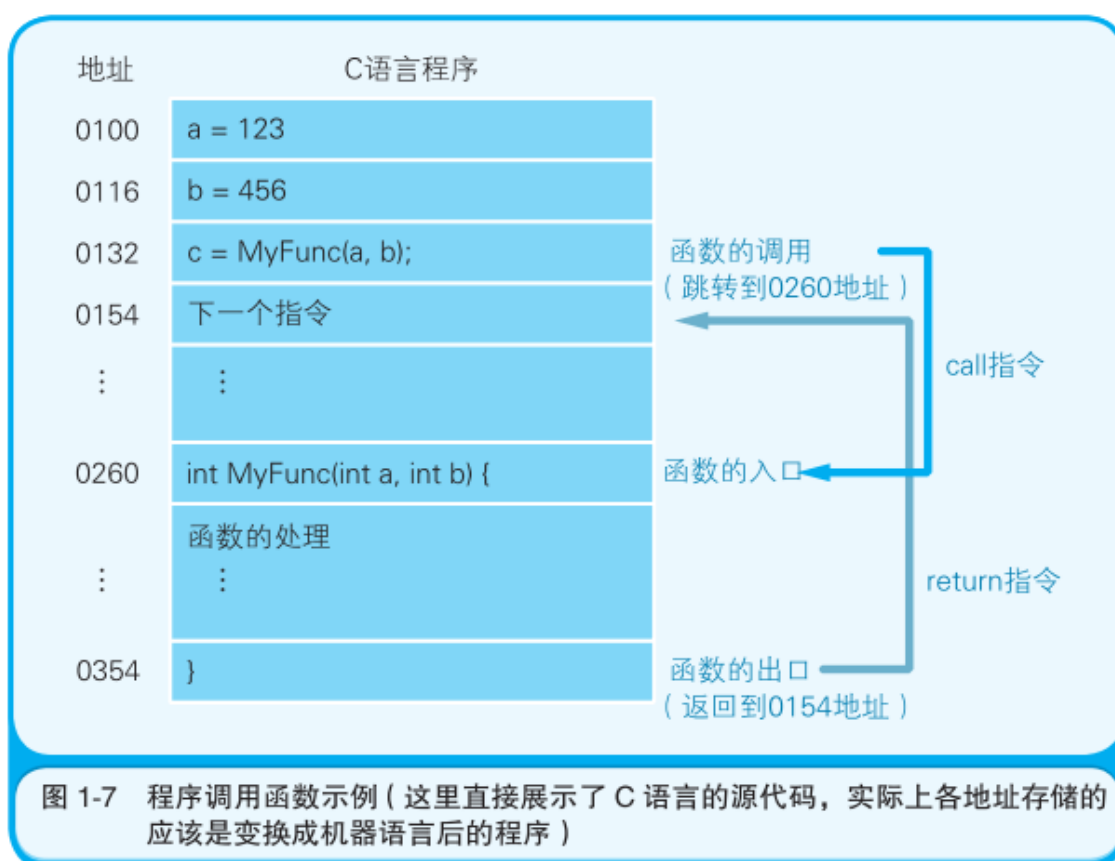
无论当前累加寄存器的运算结果是负数、零还是正数，标志寄存器都会将其保存（也负责存放溢出和奇偶校验的结果）。



程序中的比较指令，就是在 CPU 内部做减法运算。

假设要比较累加寄存器中存储的 XXX 值和通用寄存器中存储的 YYY 值，执行比较的指令后，CPU 的运算装置就会在内部（暗中）进行  $XXX - YYY$  的减法运算。而无论减法运算的结果是正数、零还是负数，都会保存到标志寄存器中。

函数调用使用的是 `call` 指令，而不是跳转指令。在将函数的入口地址设定到程序计数器之前，`call` 指令会把调用函数后要执行的指令地址存储在名为栈的主存内。函数处理完毕后，再通过函数的出口来执行 `return` 命令。`return` 命令的功能是把保存在栈中的地址设定到程序计数器中。



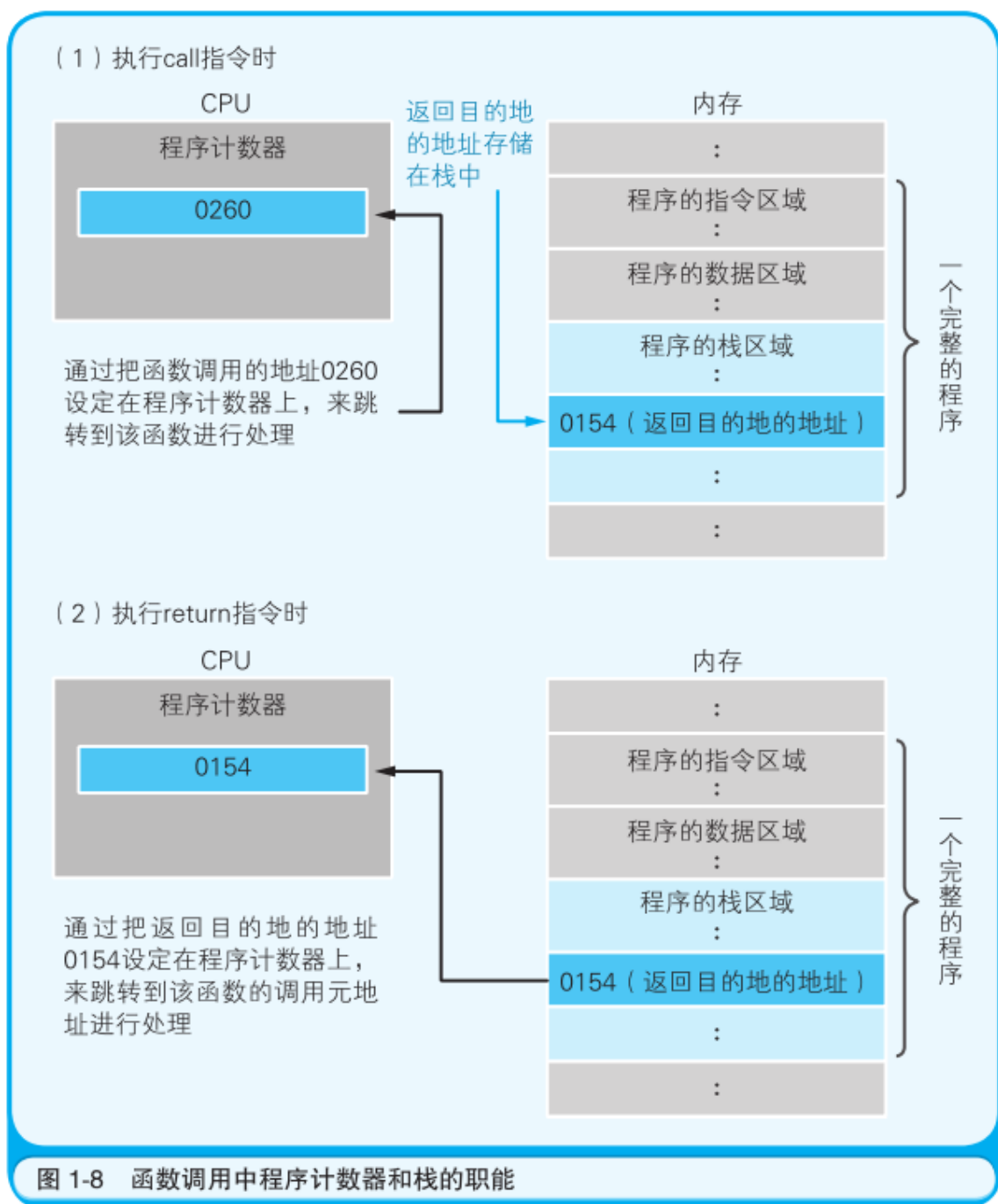


图 1-8 函数调用中程序计数器和栈的职能

CPU 会把基址寄存器+变址寄存器的值 解释为实际查看的内存地址。变址寄存器的值就相当于高级编程语言 程序中数组的索引功能。

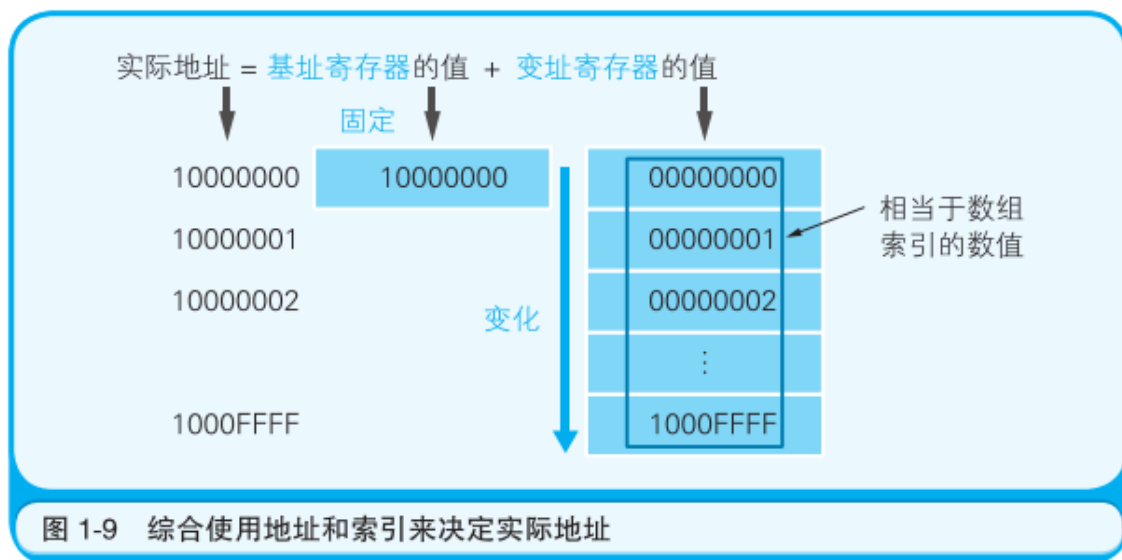


表 1-2 机器语言指令的主要类型和功能

类 型	功 能
数据转送指令	寄存器和内存、内存和内存、寄存器和外围设备 <sup>①</sup> 之间的数据读写操作
运算指令	用累加寄存器执行算术运算、逻辑运算、比较运算和移位运算
跳转指令	实现条件分支、循环、强制跳转等
call/return 指令	函数的调用 / 返回调用前的地址

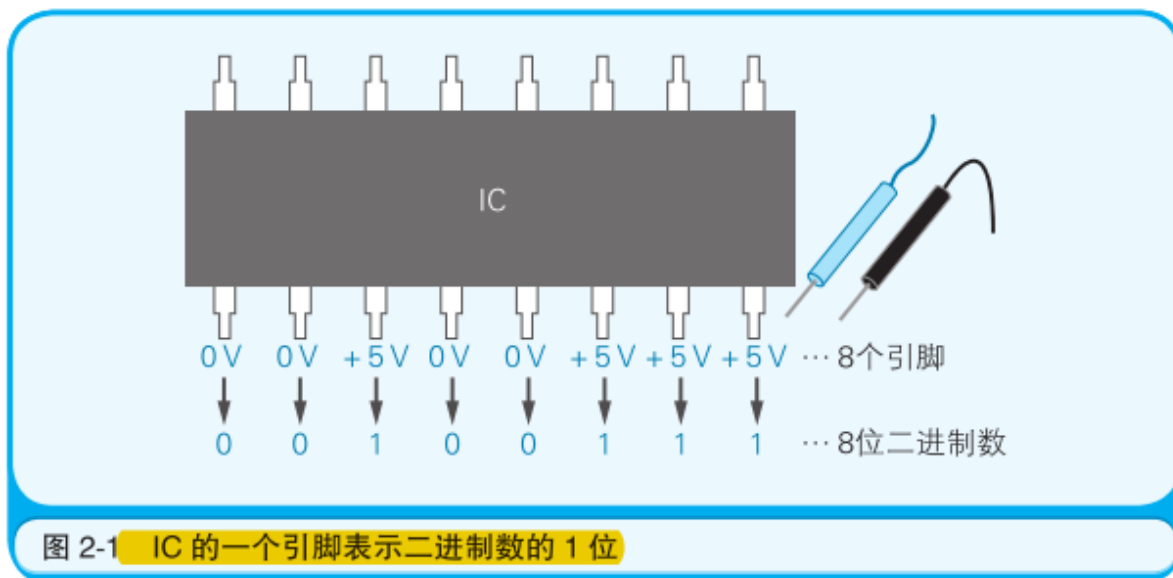
## 数据

### 用二进制数表示计算机信息的原因

IC的所有引脚，只有直流电压 0V或 5V 两个状态。也就是说，IC 的一个引脚，只能表示两个状态。

计算机处理信息的最小单位—— 位，就相当于二进制中的一位。





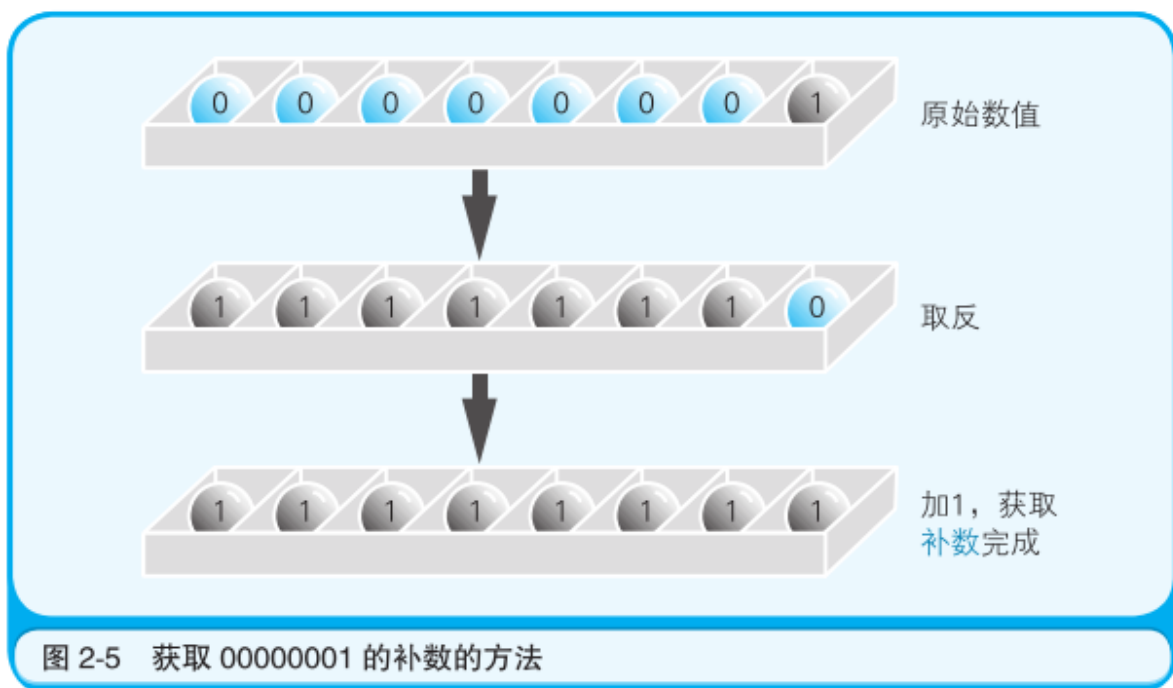
8 位二进制 数被称为一个 字节。

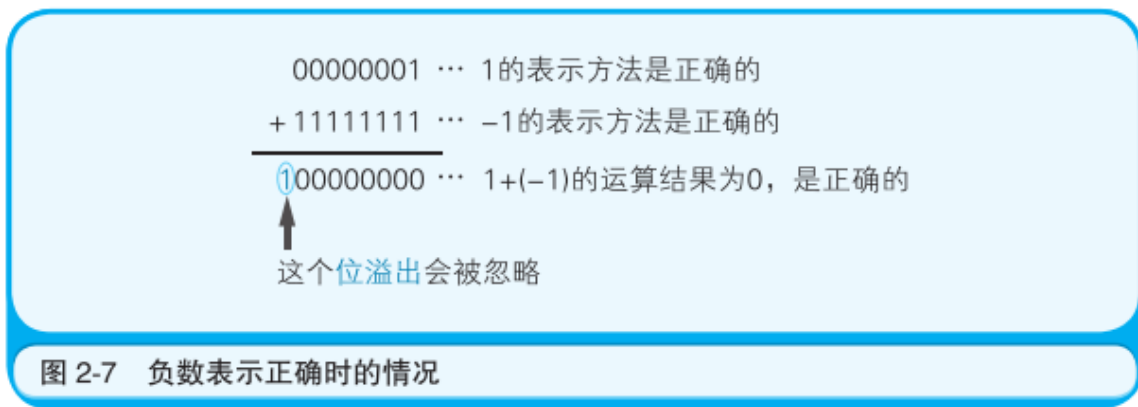
## “补数”

计算机在做减法运算时，实际上内部是在做加法运算。用加法运算来实现减法运算。

在表示负数时就需 要使用“二进制的补数”。补数就是用正数来表示负数。

具体来说，就是将各数位的 0 取反成 1，1 取反成 0，然后再将取反的结果加 1。



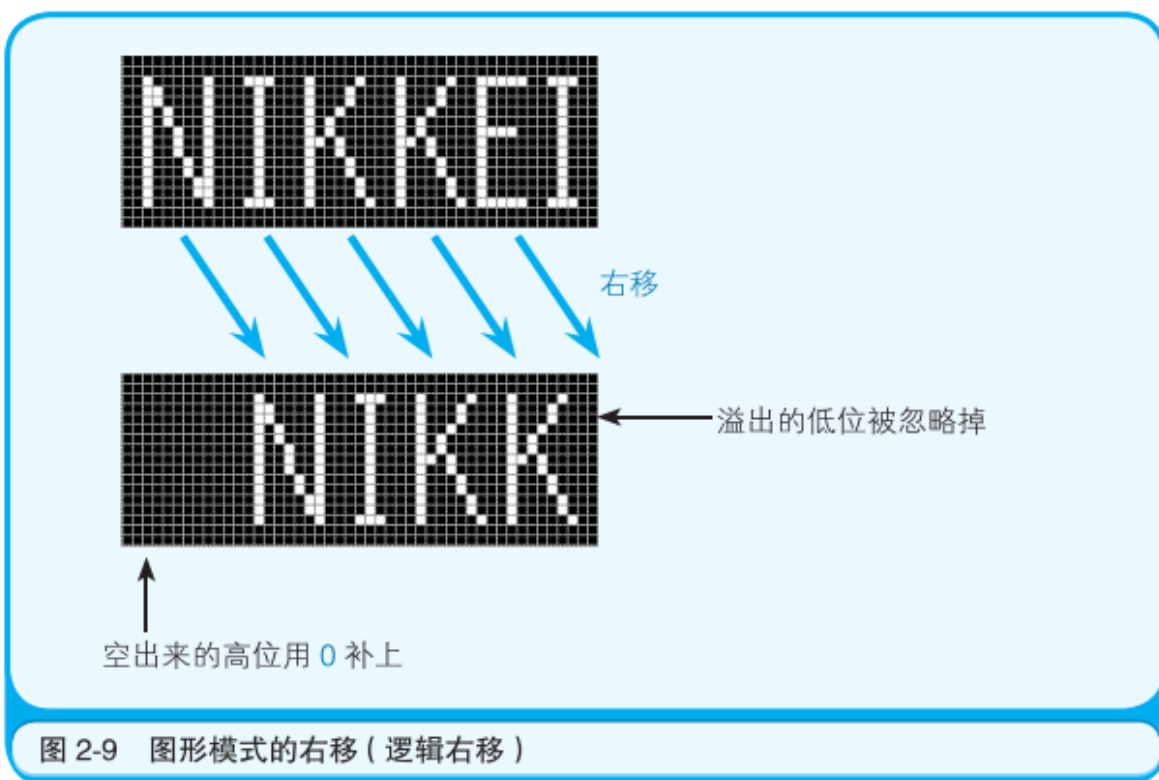


补数求解的变换方法就是“取反 + 1”

0 包含在正数范围内，所以负数就要比正数多 1 个。虽然 0 不是正数，但考虑到符号位，就将其划分到了正数中。

## 逻辑右移和算术右移

当二进制数的值表示图形模式而非数值时，移位后需要在最高位补 0。类似于霓虹灯往右滚动的效果。这就称为逻辑右移



将二进制数作为带符号的数值进行运算时，移位后要在最高位填充移位前符号位的值（0 或 1）。这就称为算术右移。

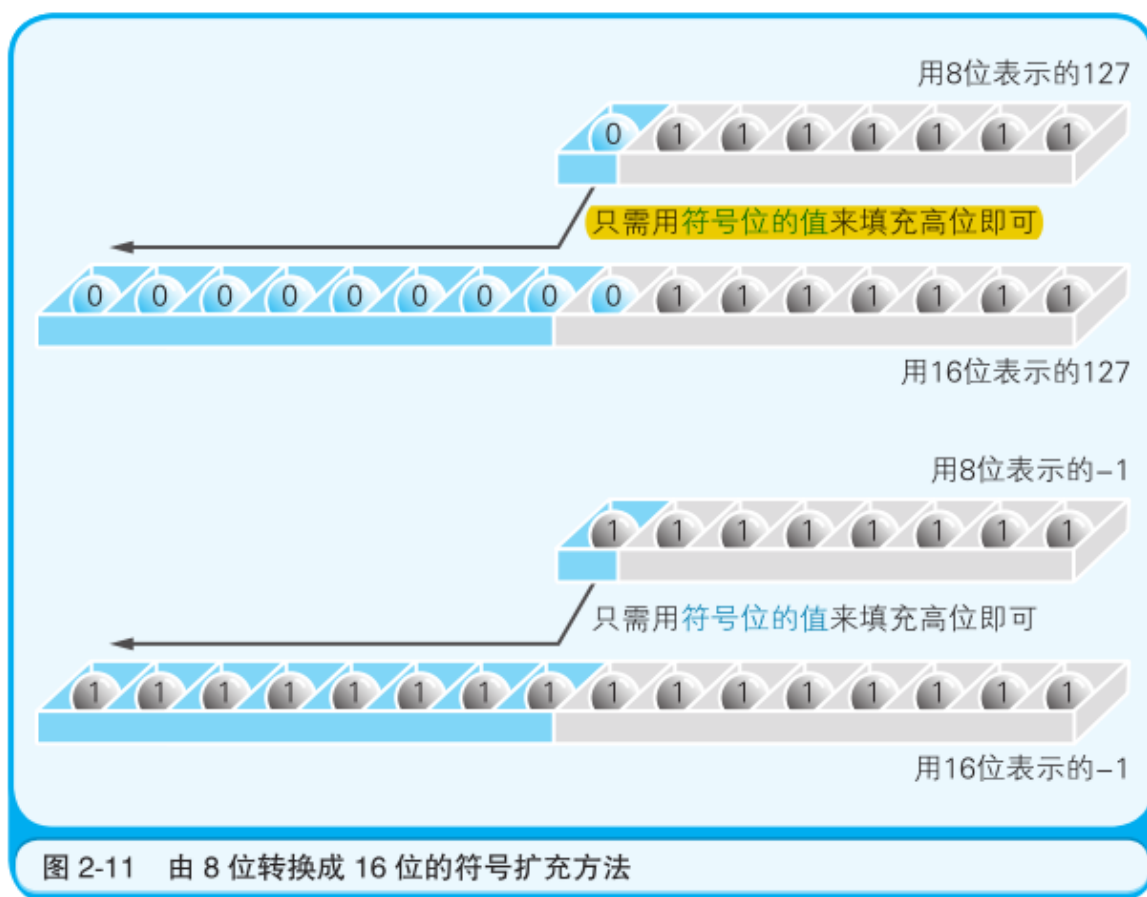
将  $-4$  ( $= 11111100$ ) 右移两位。这时，逻辑右移的情况下结果就会变成  $00111111$ ，也就是十进制数  $63$

而算术右移的情况下，结果就会变成  $11111111$ ，用补数表示就是  $-1$ ，即  $-4$  的  $1/4$

只有在右移时才必须区分逻辑位移和算术位移。左移时，无论是图形模式（逻辑左移）还是相乘运算（算术左移），都只需在空出来的低位补  $0$  即可

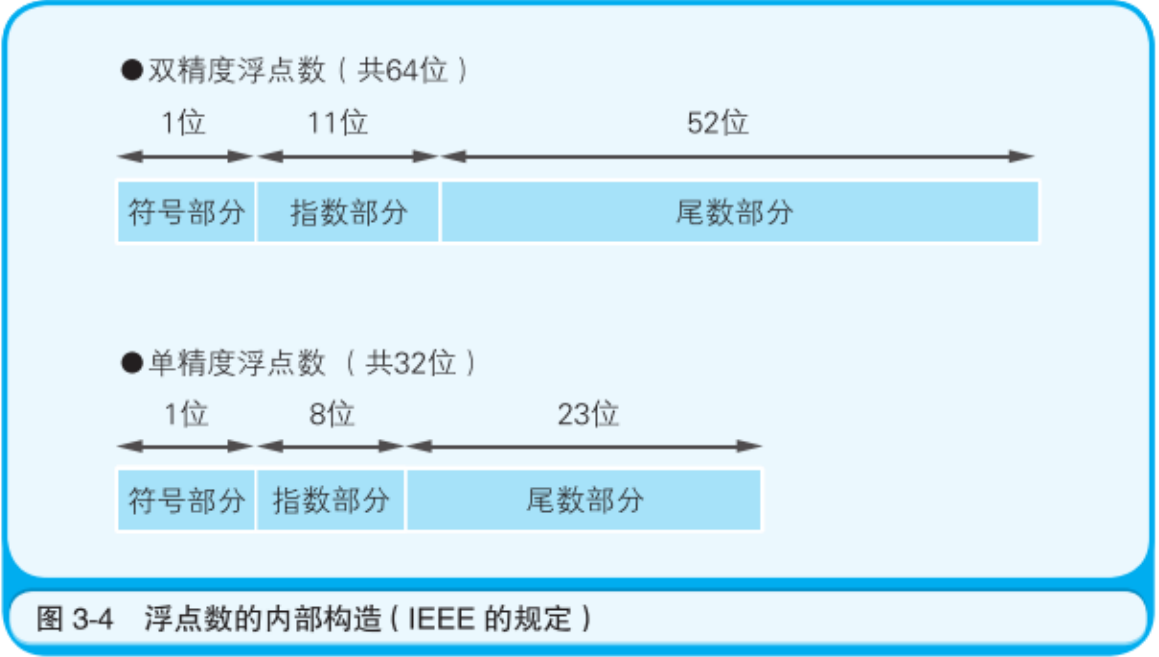
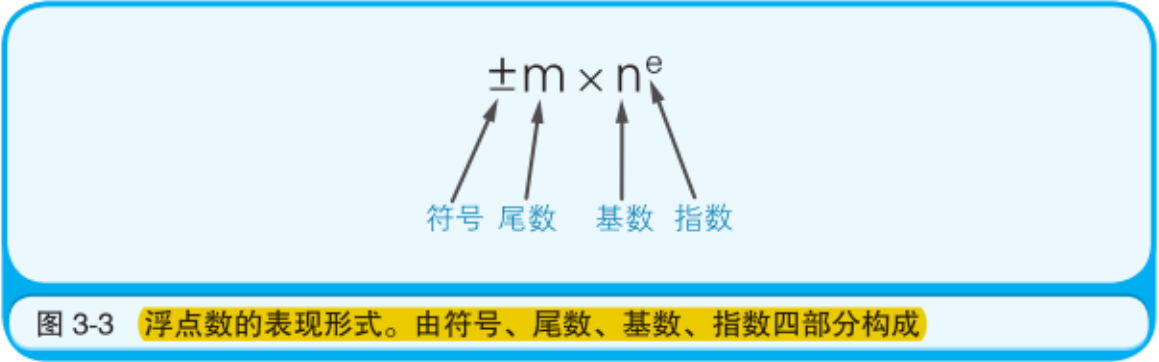
符号扩充就是指在保持值不变的前提下将其转换成  $16$  位和  $32$  位的二进制数。

不管是正数还是用补数表示的负数，都只需用符号位的值（ $0$  或者  $1$ ）填充高位即可

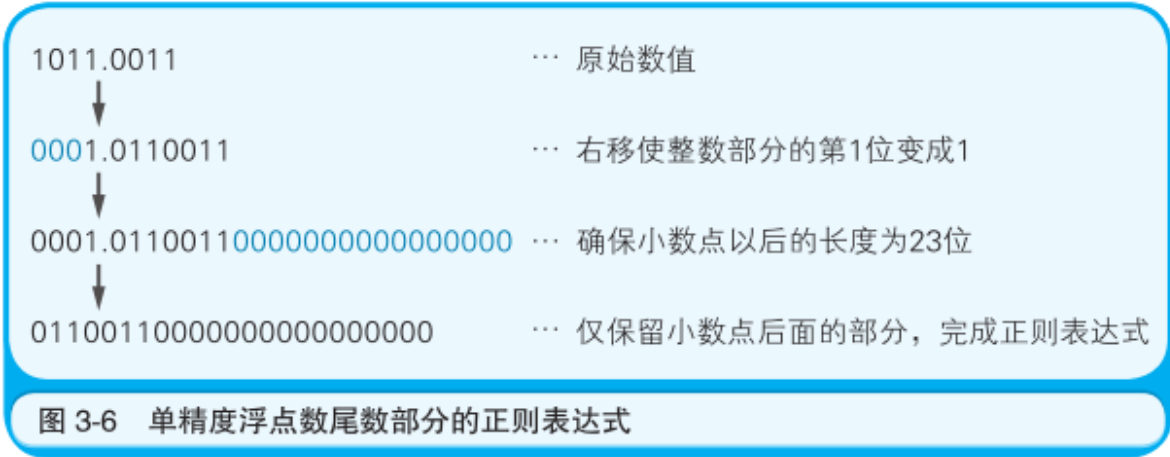


## 小数运算

浮点数是指用符号、尾数、基数和指数这四部分来表示的小数。



尾数部分使用 正则表达式：



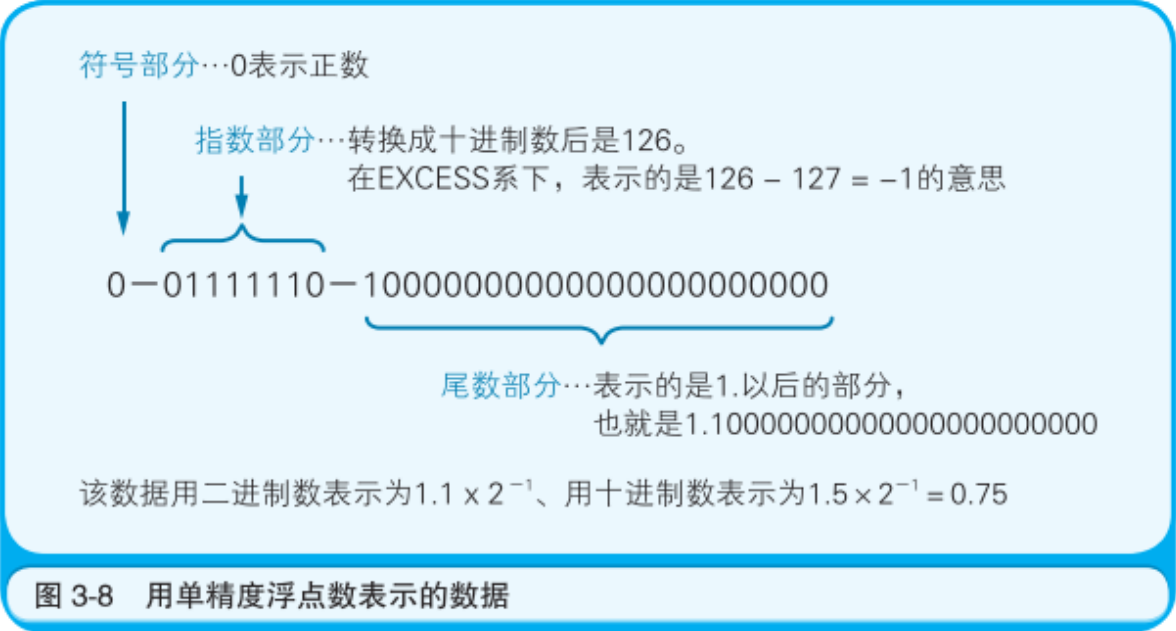
指数部分中使用的 EXCESS 系统：

用 1 ~ 13（A ~ K）的扑克牌来表示负数。这时，我们可以把中间的 7 这张牌当成 0。如果扑克牌 7 是 0，10 就表示 +3，3 就表示 -4。事实上，这个规则说的就是 EXCESS 系统。

指数部分为二进制数 11111111（十进制数 255），那么在 EXCESS 系统中则表示为 128 次幂。这是因为  $255 - 127 = 128$ 。因此，8 位的情况下，表示的范围就是 -127 次幂~128 次幂。

表 3-2 单精度浮点数指数部分的 EXCESS 系统表现

实际的值（二进制数）	实际的值（十进制数）	EXCESS 系统表现（十进制数）
11111111	255	128 ( = 255 - 127 )
11111110	254	127 ( = 254 - 127 )
⋮	⋮	⋮
01111111	127	0 ( = 127 - 127 )
01111110	126	- 1 ( = 126 - 127 )
⋮	⋮	⋮
00000001	1	- 126 ( = 1 - 127 )
00000000	0	- 127 ( = 0 - 127 )



## 如何避免计算机计算出错

首先是回避策略，即无视这些错误。

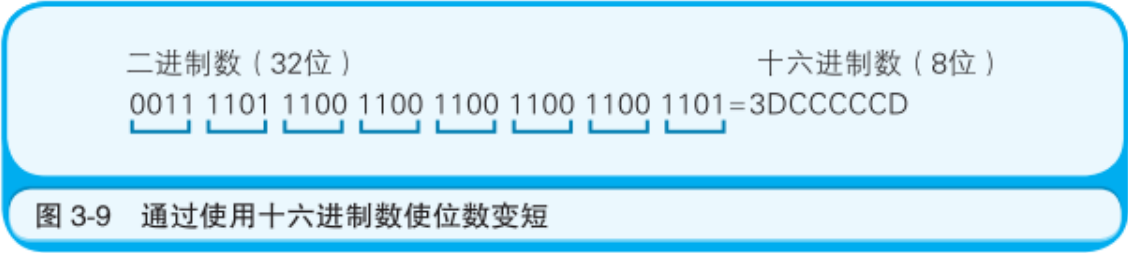
另一个策略是把小数转换成整数来计算。

进行小数的计算时可以暂时使用整数，然后再把计算结果用小数表示出来即可。

例如，将 0.1 相加 100 次这一计算，就可以转换为将 0.1 扩大 10 倍后再将 1 相加 100 次的计算，最后把结果除以 10 就可以了

## 十六进制

只需在数值的开头加上 0x（0 和 x）就可以表示 十六进制数。



二进制数（32位）	十六进制数（8位）
0011 1101 1100 1100 1100 1100 1100 1101	=3DCCCCD

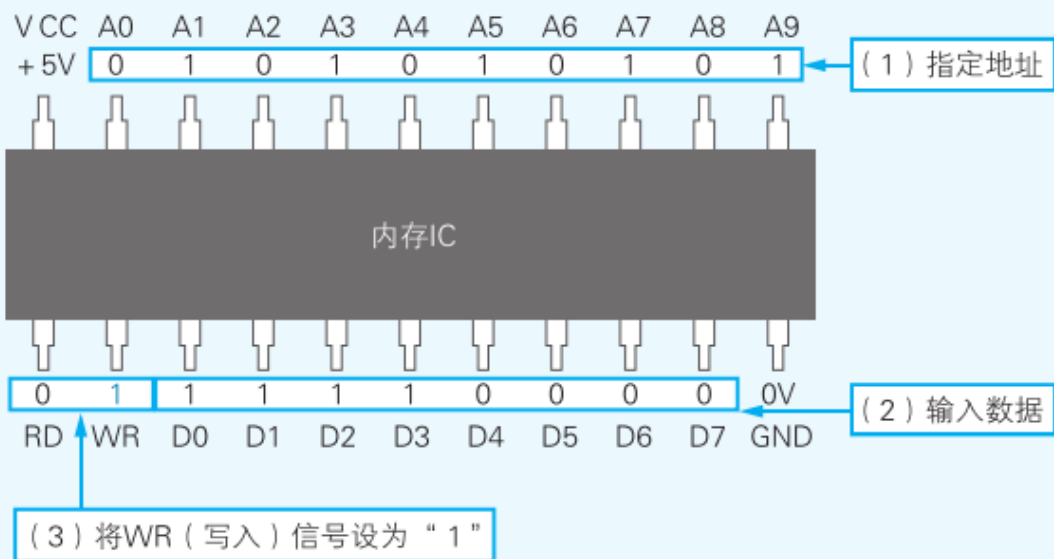
图 3-9 通过使用十六进制数使位数变短

## 内存

物理内存是以字节为单位进行数据存储的。

内存的物理机制

(a) 往0101010101地址写入11110000数据时



(b) 读出0101010101地址的数据时

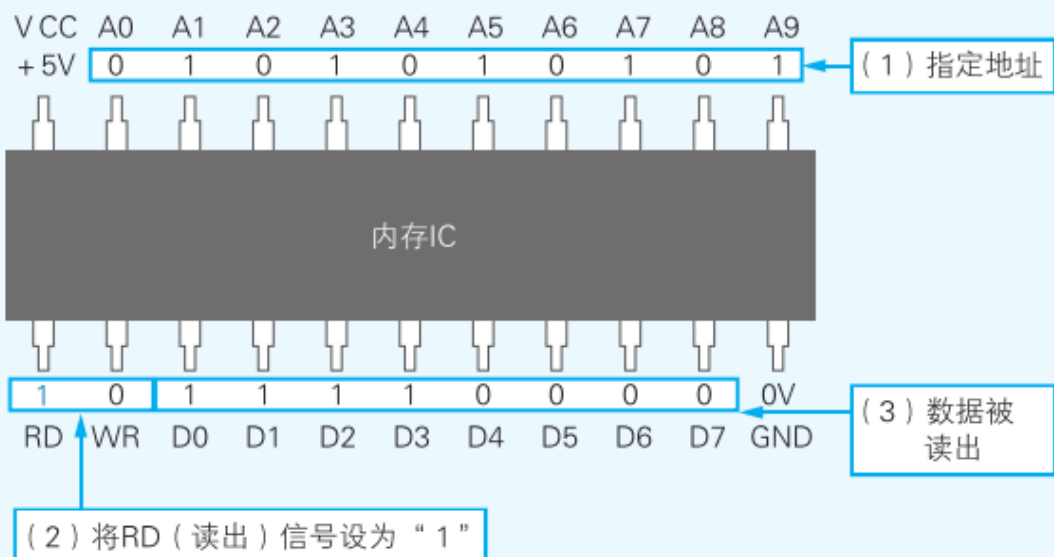
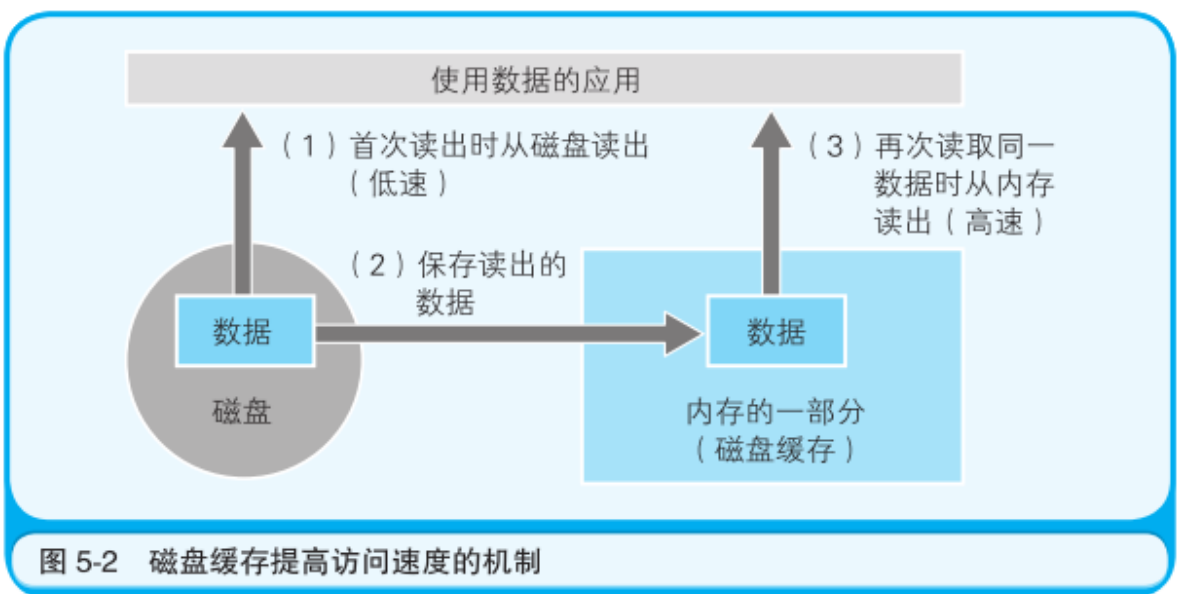
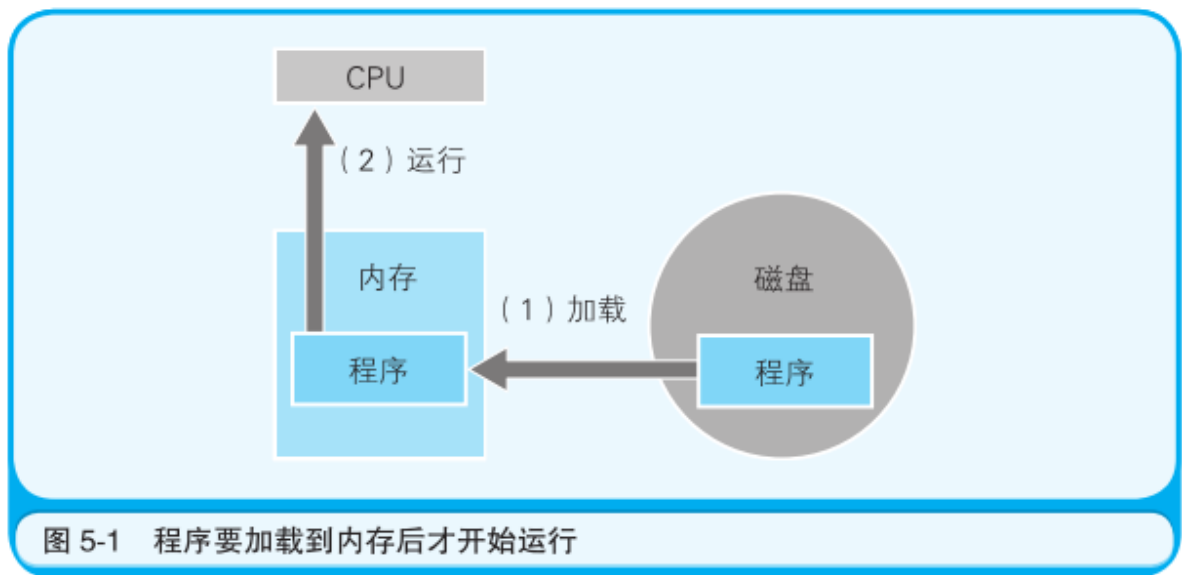


图 4-2 向内存 IC 中写入和读出数据的方法

## 内存和磁盘

在 Windows 计算机中，一般磁盘的 1 个扇区是512 字节。扇区是磁盘保存数据的物理单位。



## # 虚拟内存把磁盘作为部分内存来使用

虚拟内存是指把磁盘的一部分作为假想的内存来使用。这与磁盘缓存是假想的磁盘（实际上是内存）相对，虚拟内存是假想的内存（实际上是磁盘）。

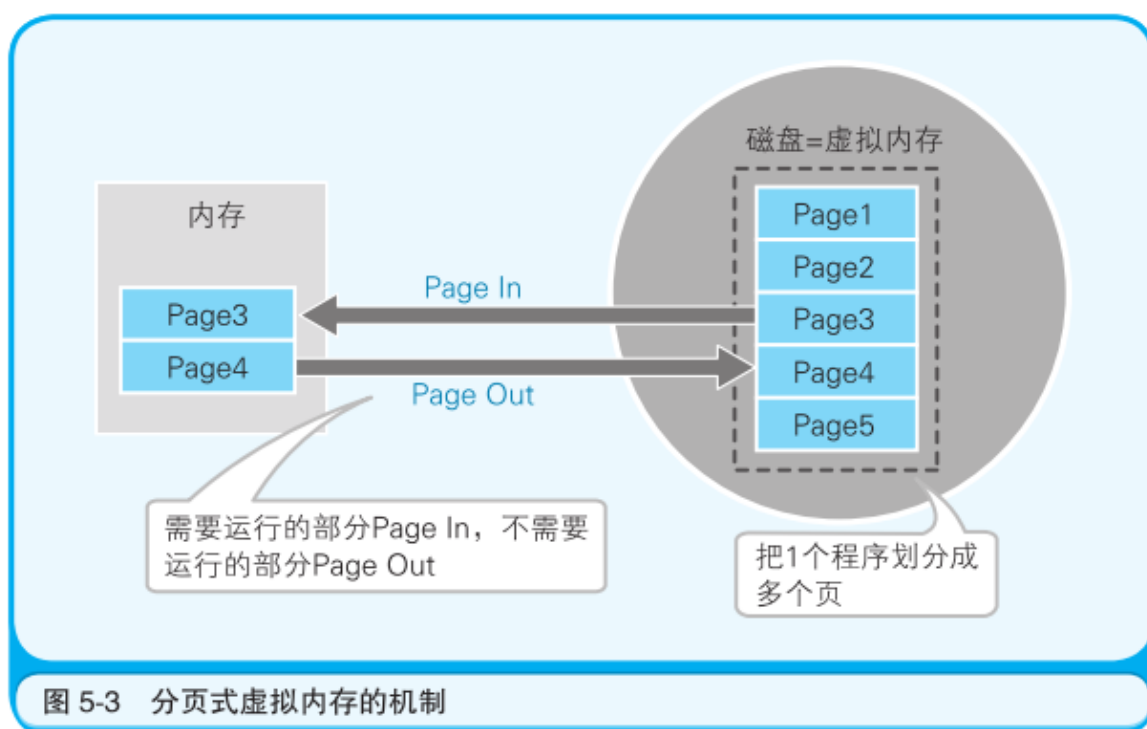
虚拟内存虽说是把磁盘作为内存的一部分来使用，但实际上正在运行的程序部分，在这个时间点上必须存在在内存中的。也就是说，为了实现虚拟内存，就必须把实际内存（也可称为物理内存）的内容，和磁盘上的虚拟内存的内容进行部分置换（swap），并同时运行程序。

虚拟内存的方法有分页式和分段式。



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。

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。

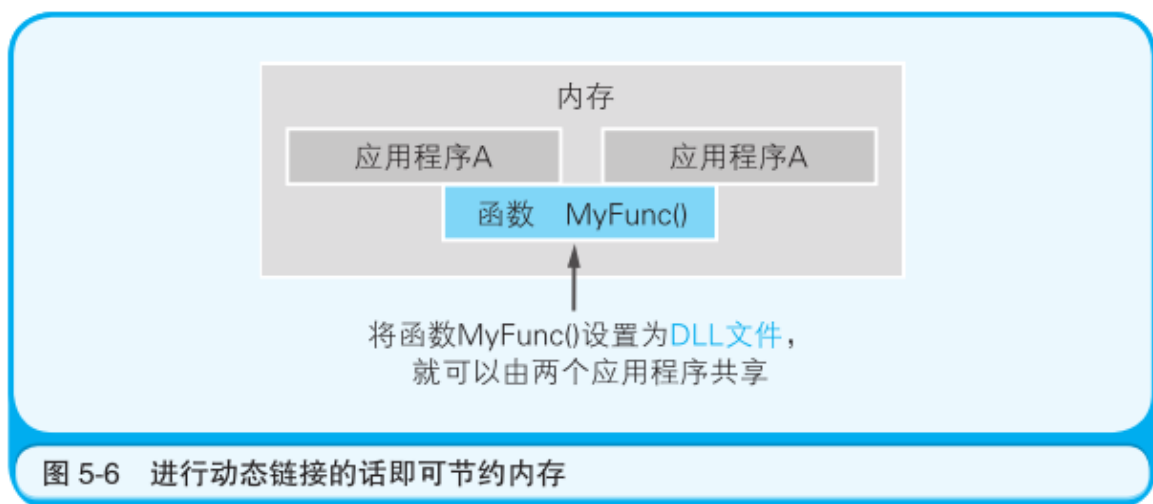
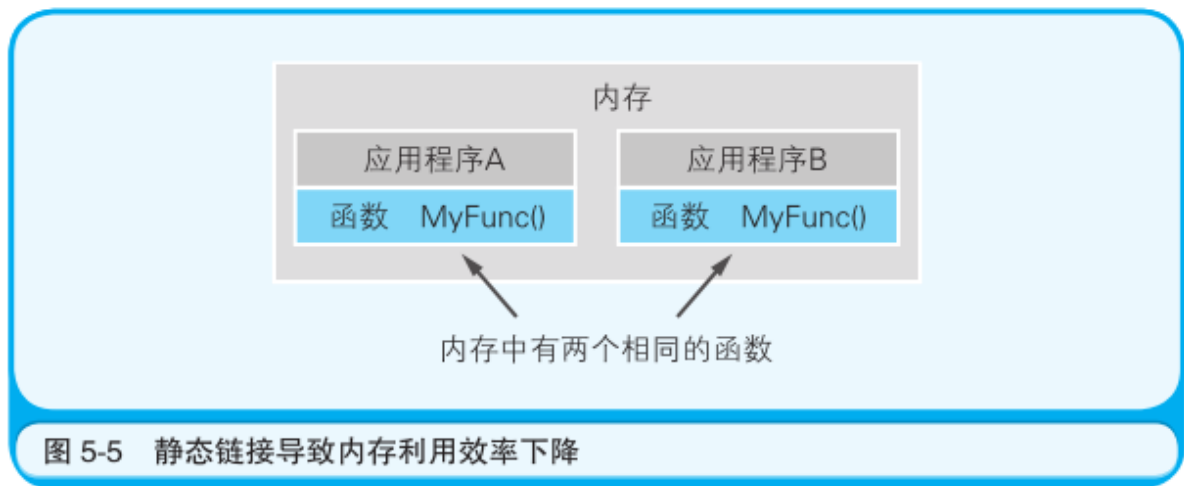


由于使用虚拟内存时发生的 Page In 和 Page Out 往往伴随着低速的磁盘访问，因此在这个过程中应用的运行会变得迟钝起来。虚拟内存无法彻底解决内存不足的问题。

## # 节约内存的编程方法

### 通过 DLL 文件实现函数共有

DLL（Dynamic Link Library）文件顾名思义，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。



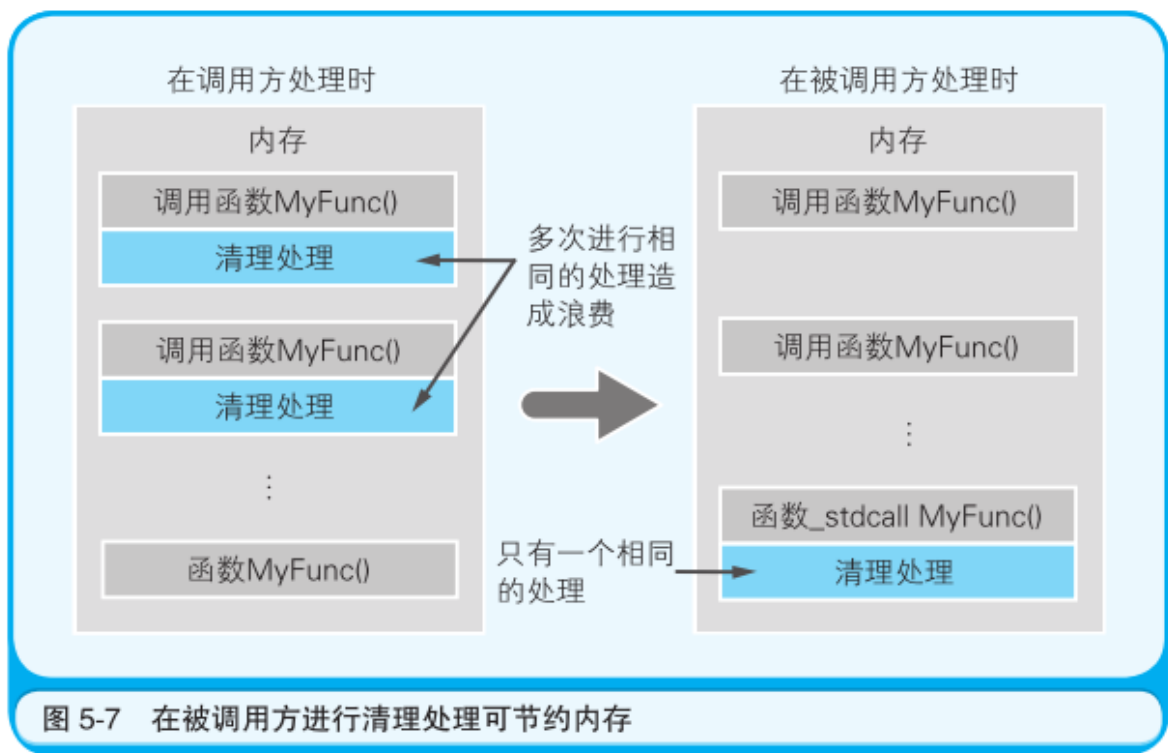
## 通过调用 `_stdcall` 来减小程序文件的大小

`_stdcall` 是 standard call（标准调用）的略称。Windows 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。

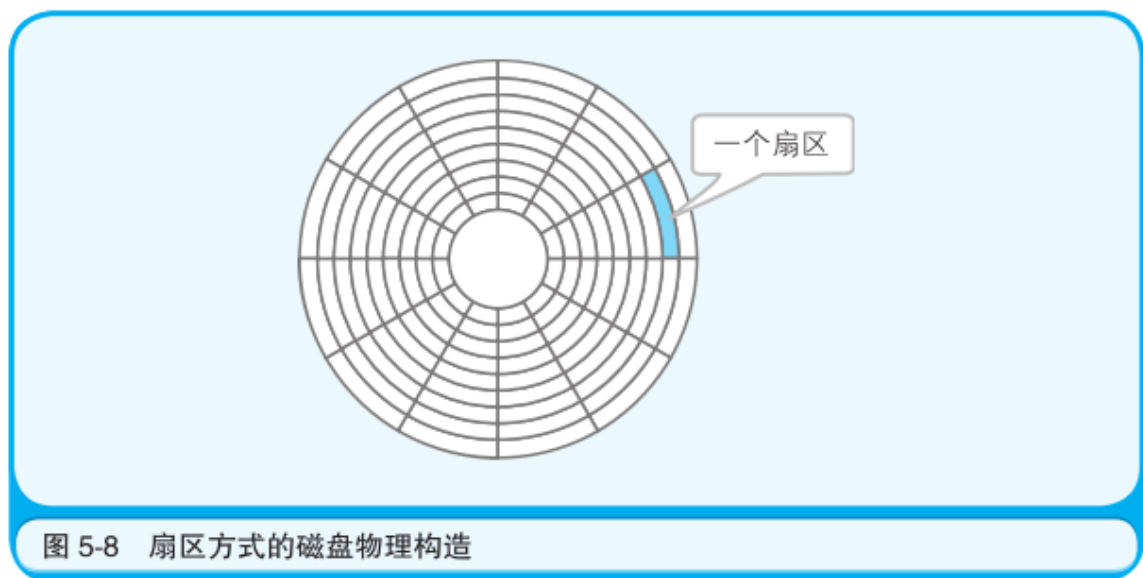
在 C 语言中，如果函数的参数数量固定的话，指定 `_stdcall` 是没有任何问题的。

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。



## # 磁盘的物理结构



## 压缩数据

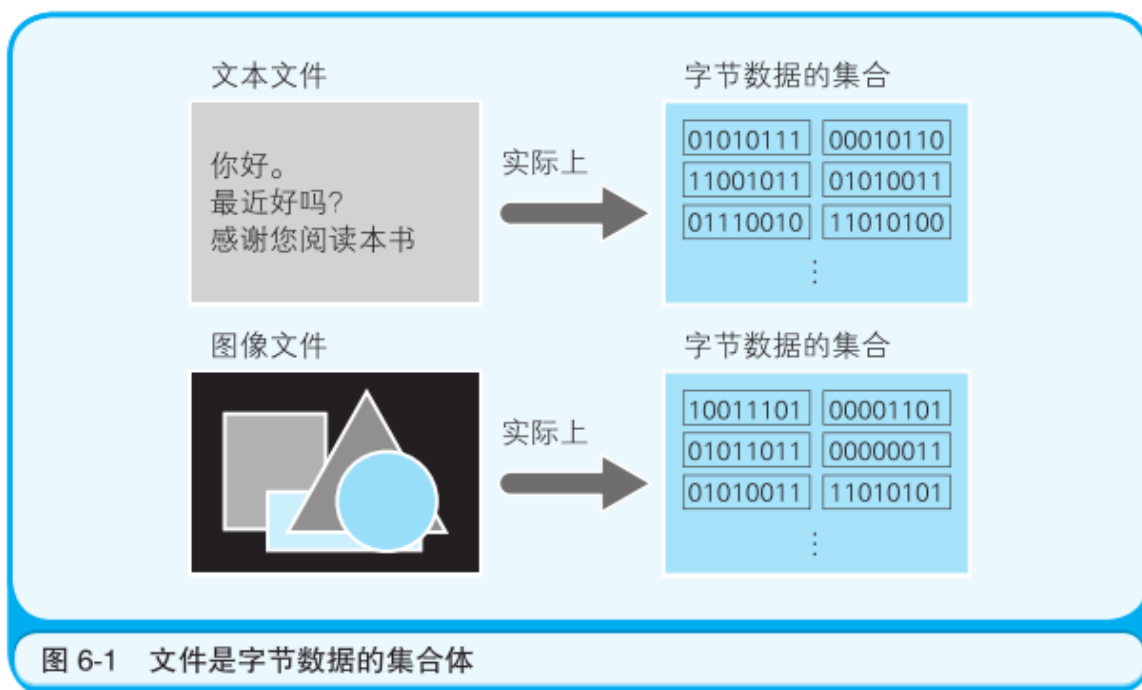
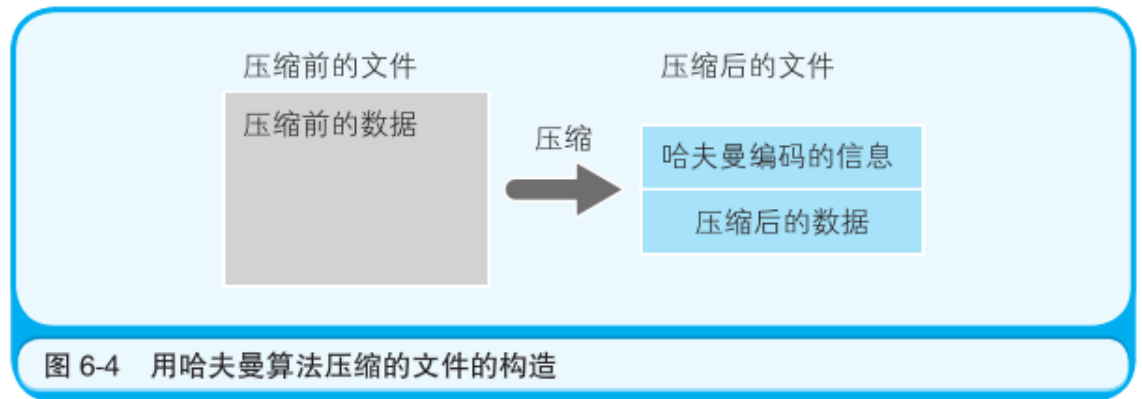


表 6-1 借助 RLE 算法对各文件进行压缩的结果

文件类型	压缩前文件大小	压缩后文件大小	压缩比率
文本文件	14862 字节	29506 字节	199%
图像文件	96062 字节	38328 字节	40%
EXE 文件	24576 字节	15198 字节	62%



哈夫曼算法能够大幅提升压缩比率

表 6-3 出现频率和编码 ( 方案 )

字符	出现频率	编码 ( 方案 )	位数
A	6	0	1
E	5	1	1
B	2	10	2
D	2	11	2
C	1	100	3
F	1	101	3

步骤1：列出数据及其出现频率，（）里面表示的是出现频率，这里按照降序排列

出现频率	(6)	(5)	(2)	(2)	(1)	(1)
数据	A	E	B	D	C	F

步骤2：选择两个出现频率最小的数字，拉出两条线，并在交叉地方写上这两位数字的和。当有多个选项时，任意选取即可

					(2)	
					└─┘	
出现频率	(6)	(5)	(2)	(2)	(1)	(1)
数据	A	E	B	D	C	F

步骤3：重复步骤2，可以连接任何位置的数值

				(4)		(2)	
				└─┘		└─┘	
出现频率	(6)	(5)	(2)	(2)	(1)	(1)	
数据	A	E	B	D	C	F	

步骤4：最后这些数字会被汇集到了1个点上，该点就是根，这样哈夫曼树也就完成了。按照从根部到底部的叶子这一顺序，在左边的树枝（线）处写上0，在右边的树枝（线）处写上1。然后从根部开始沿着树枝到达目标文字后，再按照顺序把通过的树枝上的0或者1写下来，就可以得到哈夫曼编码了

				(17)			
				└─┘			
				0		1	
					(6)		
					└─┘		
					0		1
						(4)	(2)
						└─┘	└─┘
						0	1
出现频率	(6)	(5)	(2)	(2)	(1)	(1)	
数据	A	E	B	D	C	F	
哈夫曼编码	00	01	100	101	110	111	

图 6-5 哈夫曼树的编码顺序

表 6-4 LHA 对各种文件的压缩结果

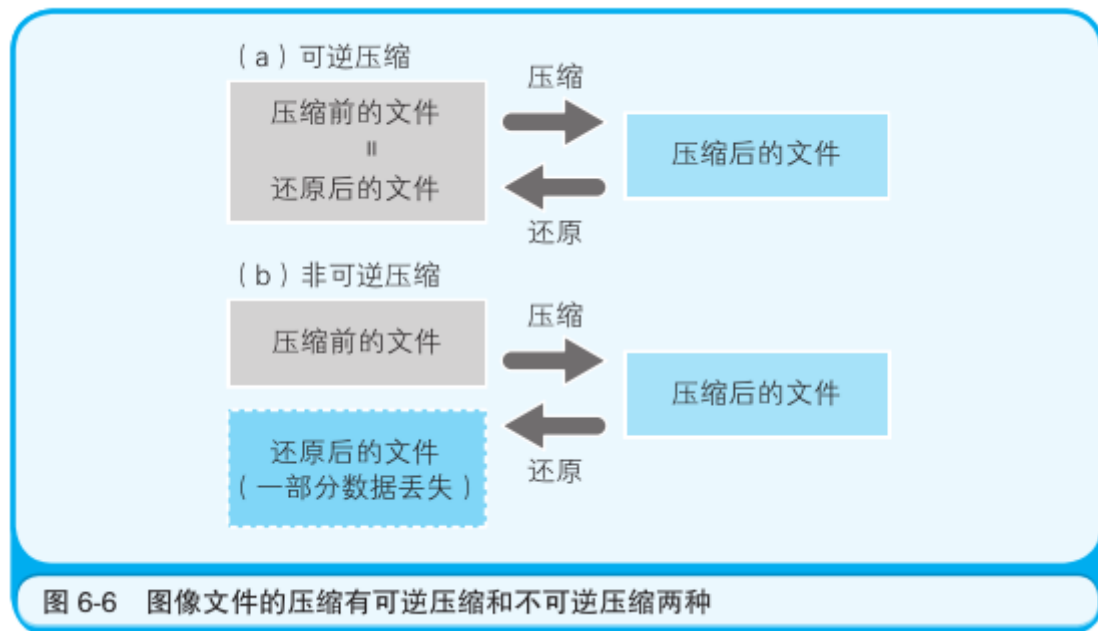
文件类型	压缩前	压缩后	压缩比率
文本文件	14 862 字节	4119 字节	28%
图像文件	96 062 字节	9456 字节	10%
EXE 文件	24 576 字节	4652 字节	19%

## # 可逆压缩和非可逆压缩

---

Windows 的标准图像数据形式为BMP，是完全未压缩的。

JPEG 格式、TIFF 格式、GIF 格式等。与BMP格式不同的是，这些图像数据都会用一些技法来对数据进行压缩。

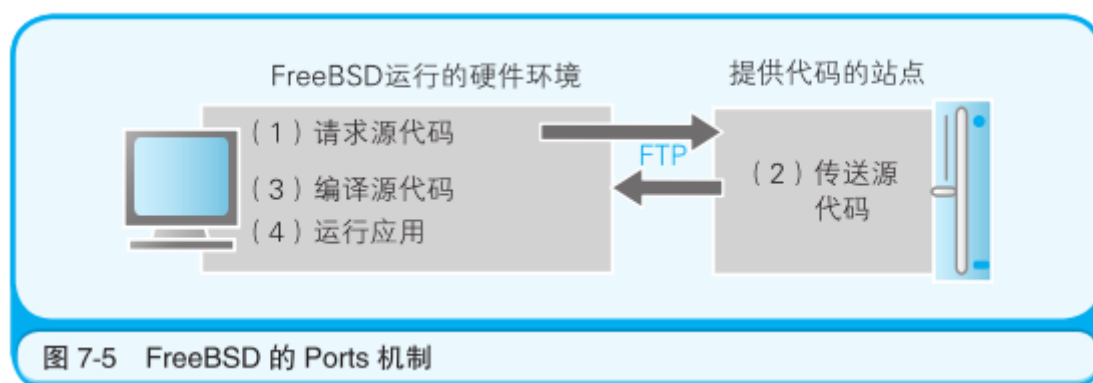
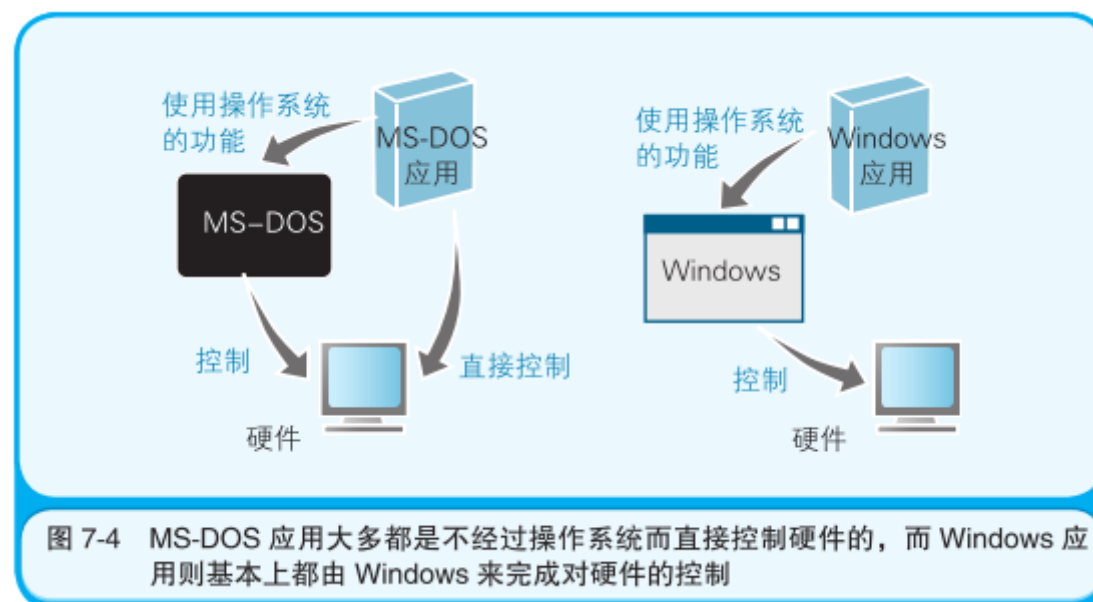
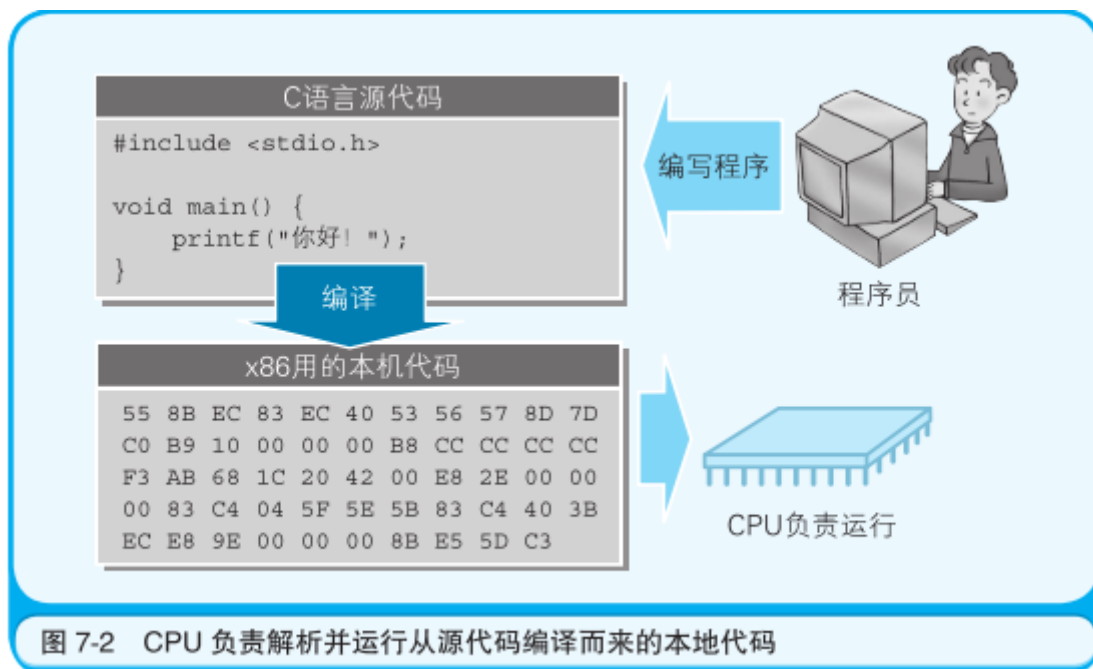


## 程序是在何种环境中运行的

---

# 运行环境 = 操作系统 + 硬件

---



提供相同运行环境的 **Java** 虚拟机

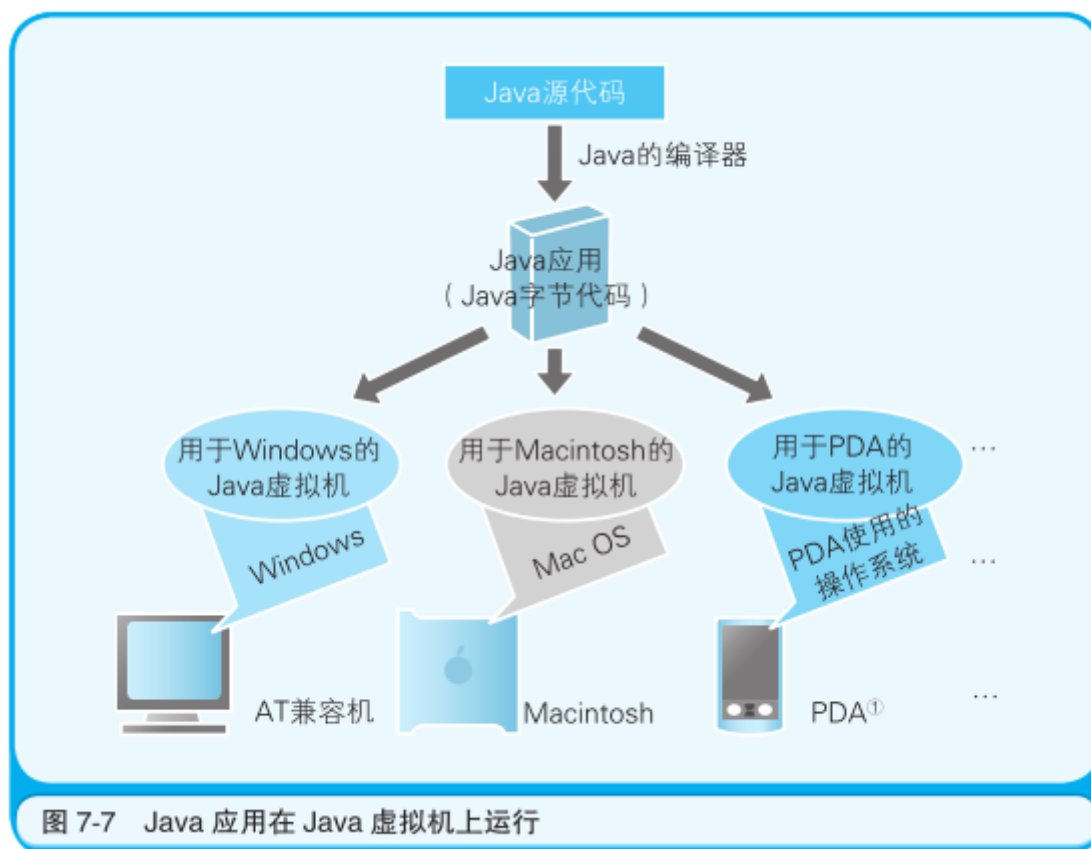


除虚拟机的方法之外，还有一种方法能够提供不依赖于特定硬件及操作系统的程序运行环境，那就是 Java。

大家说的 Java，有两个层面的意思。一个是作为编程语言的 Java，另一个是作为程序运行环境的 Java。

编译后生成的并不是特定 CPU 使用的本地代码，而是名为字节代码的程序。字节代码的运行环境就称为 Java 虚拟机（JavaVM，Java Virtual Machine）。

Java 虚拟机是一边把 Java 字节代码逐一转换成本地代码一边运行的。



## # BIOS和引导

程序的运行环境中，存在着名为BIOS（Basic Input/Output System）的系统。BIOS存储在ROM中，是预先内置在计算机主机内部的程序。

BIOS除了键盘、磁盘、显卡等基本控制程序外，还有启动“引导程序”的功能。引导程序是存储在启动驱动器起始区域的小程序。操作系统的启动驱动器一般是硬盘，不过有时也可以是CD-ROM或软盘。

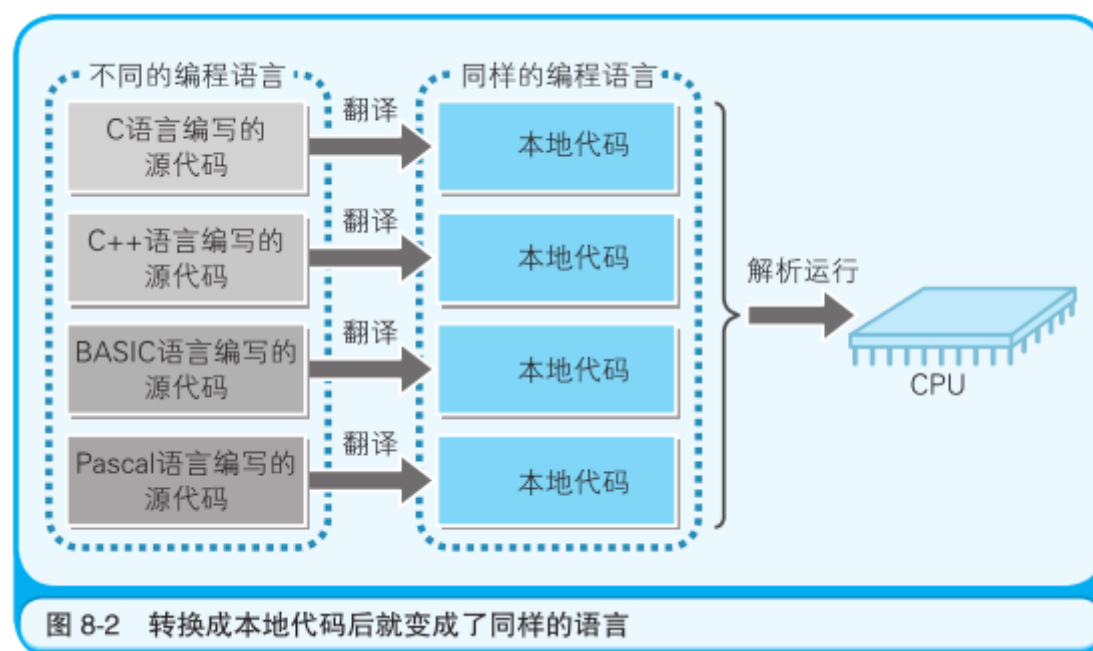
开机后，BIOS 会确认硬件是否正常运行，没有问题的话就会启动引导程序。引导程序的功能是把硬盘等记录的 OS 加载到内存中运行。虽然启动应用是 OS 的功能，但 OS 并不能自己启动自己，而是通过引导程序来启动。

Bootstrap 的原意是指靴子上部的“拔靴带”。

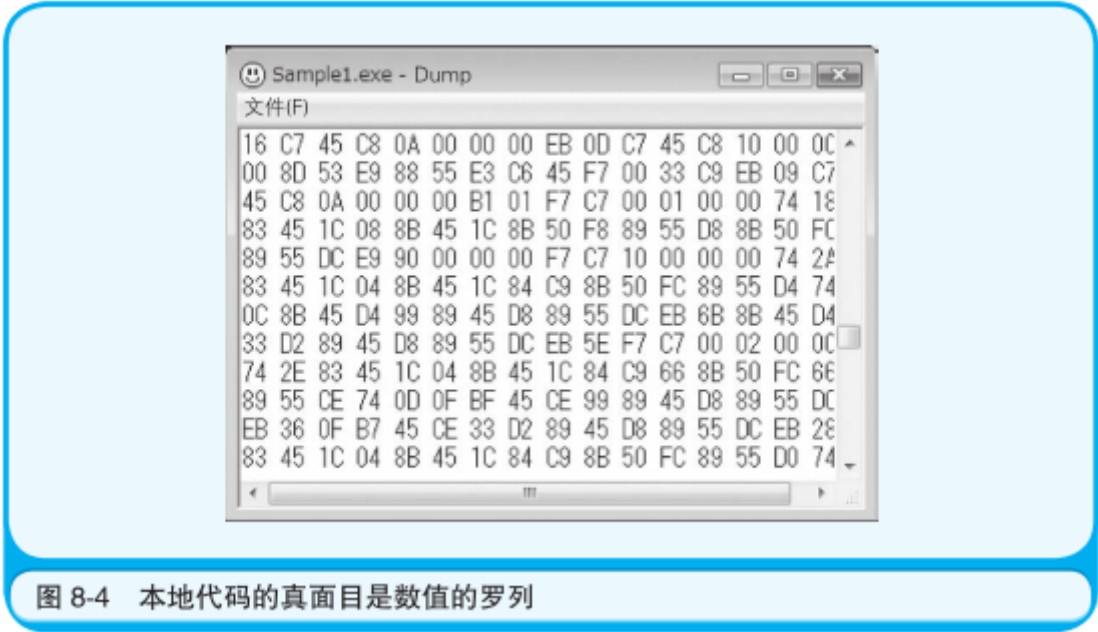


## 从源文件到可执行文件

计算机只能运行本地代码



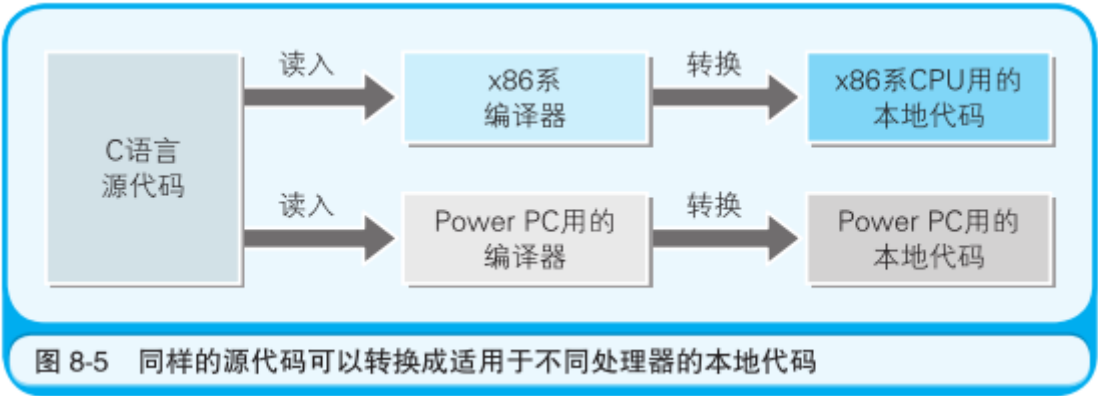
Windows 中 EXE 文件的程序内容，使用的就是本地代码。



## 编译器负责转换源代码

读入的源代码还要经过语法解析、句法解析、语义解析等，才能生成本地代码。

编译器不仅和编程语言的种类有关，和 CPU 的类型也是相关的。



把多个目标文件结合，生成 1 个 EXE 文件的处理就是 链接，运行链接的程序就称为 链接器（linkage editor 或 连结器）。

## 启动及库文件

链接选项“-Tpe-c-x-aa”是指定生成Windows 用的 EXE 文件的选项。

而该命令行中就指定了 c0w32.obj、Sample1.obj 这两个目标文件。

c0w32.obj 这个 目标文件记述的是同所有程序起始位置相结合的处理内容，称为程序的 启动。

像 import32.lib 及 cw32.lib 这样的文件称为库文件。

库文件指的是把多个目标文件集成保存到一个文件中的形式。

链接器指定库文件后， 就会从中把需要的目标文件抽取出来，并同其他目标文件结合生成 EXE 文件。

## # DLL文件及导入库

---

Windows 中，API 的目标文件，并不是存储在通常的库文件中，而是存储在名为 DLL（Dynamic Link Library）文件的特殊库文件中。

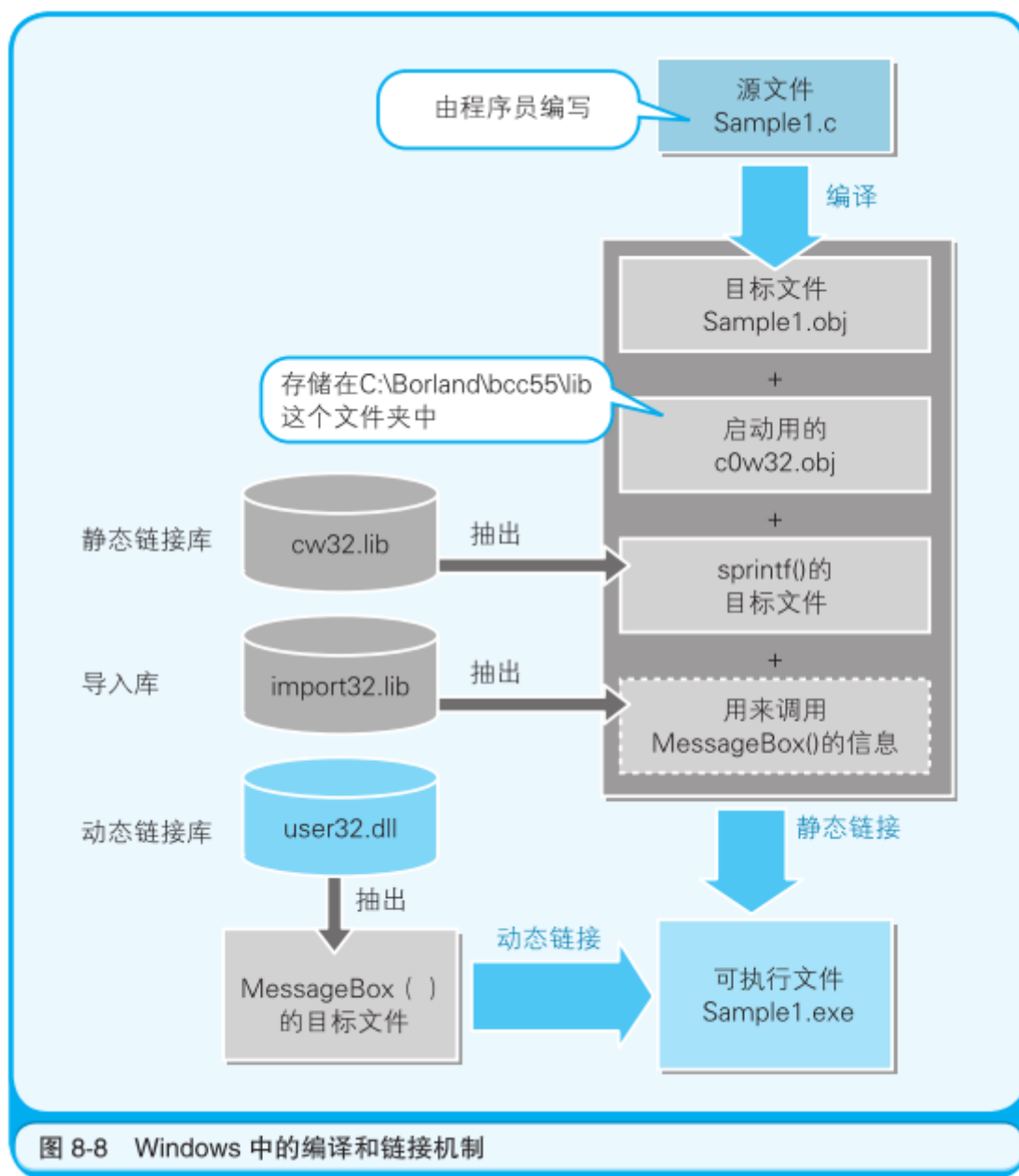


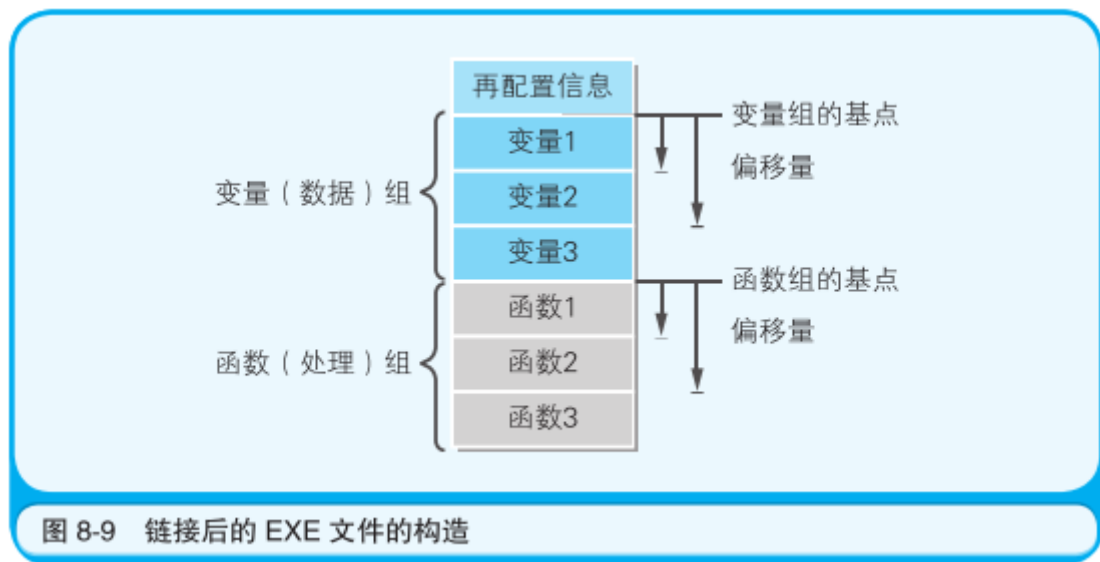
图 8-8 Windows 中的编译和链接机制

## # 可执行文件运行时的必要条件

每次运行时，程序内的变量及函数被分配到的内存地址都是不同的。

EXE 文件中给变量及函数分配了虚拟的内存地址。在程序运行时，虚拟的内存地址会转换成实际的内存地址。链接器会在 EXE 文件的开头，追加转换内存地址所需的必要信息。这个信息称为再配置信息。

EXE 文件的再配置信息，就成为了变量和函数的相对地址。相对地址表示的是相对于基点地址的偏移量，也就是相对距离。



## # 程序加载时会生成栈和堆

当程序加载到内存后，除此之外还会额外生成两个组，那就是栈和堆。栈是用来存储函数内部临时使用的变量（局部变量），以及函数调用时所用的参数的内存区域。堆是用来存储程序运行时的任意数据及对象的内存领域。

EXE 文件中并不存在栈及堆的组。栈和堆需要的内存空间是在 EXE 文件加载到内存后开始运行时得到分配的。因而，内存中的程序，就是由用于变量的内存空间、用于函数的内存空间、用于栈的内存空间、用于堆的内存空间这 4 部分构成的。

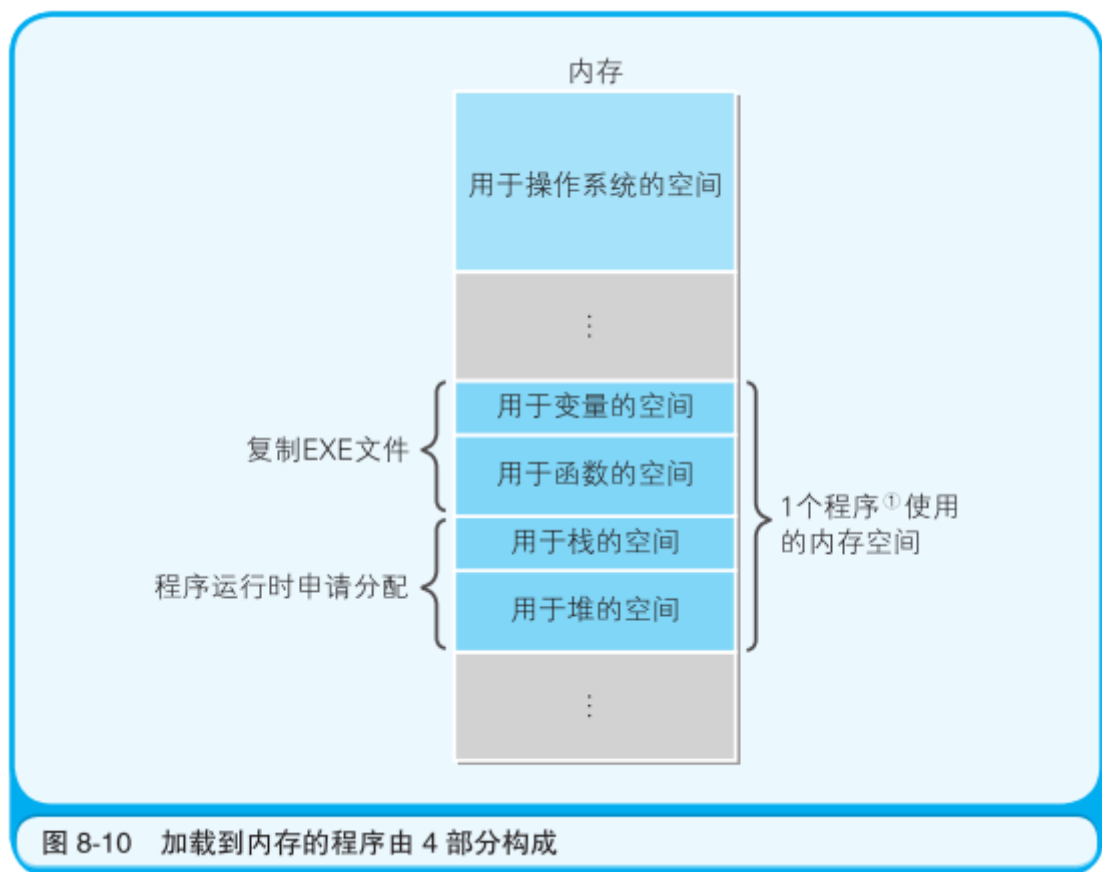


图 8-10 加载到内存的程序由 4 部分构成

编译器和解释器有什么不同？

编译器是在运行前对所有源代码进行解释处理的。而解释器则是在运行时对源代码的内容一行一行地进行解释处理的。

“分割编译”指的是什么？

将整个程序分为多个源代码来编写，然后分别进行编译，最后链接成一个 EXE 文件。这样每个源代码都相对变短，便于程序管理。

“Build”指的是什么？

根据开发工具种类的不同，有的编译器可以通过选择“Build”菜单来生成 EXE 文件。这种情况下，Build 指的是连续执行编译和链接。

使用 DLL 文件的好处是什么？

DLL 文件中的函数可以被多个程序共用。因此，借助该功能可以节约内存和磁盘。此外，在对函数的内容进行修正时，还不需要重新链接（静态链接）使用这个函数的程序

不链接导入库的话就无法调用 DLL 文件中的函数吗？

通过使用 LoadLibrary() 及 GetProcAddress() 这些 API，即使不链接导入库，也可以在程序运行时调用 DLL 文件中的函数。不过使用 导入库更简单一些。

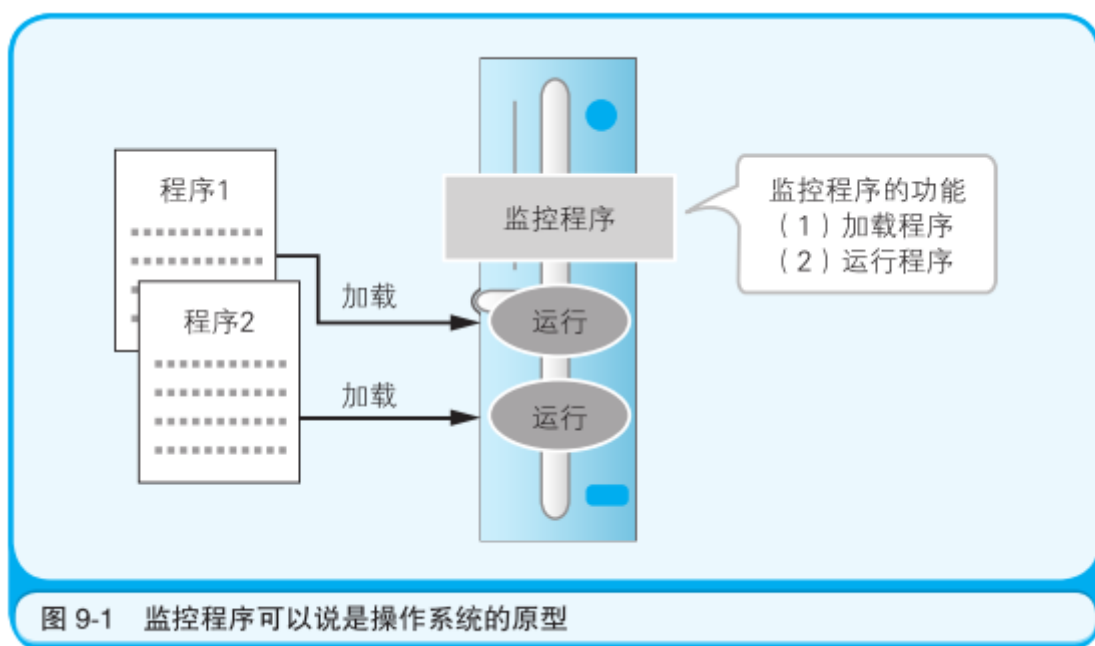
叠加链接”这个术语指的是什么？

将不会同时执行的函数，交替加载到同一个地址中运行。通过使用“叠加链接器”这一特殊的链接器即可实现。在计算机中配置的内存容量不多的 MS-DOS 时代，经常使用叠加链接。

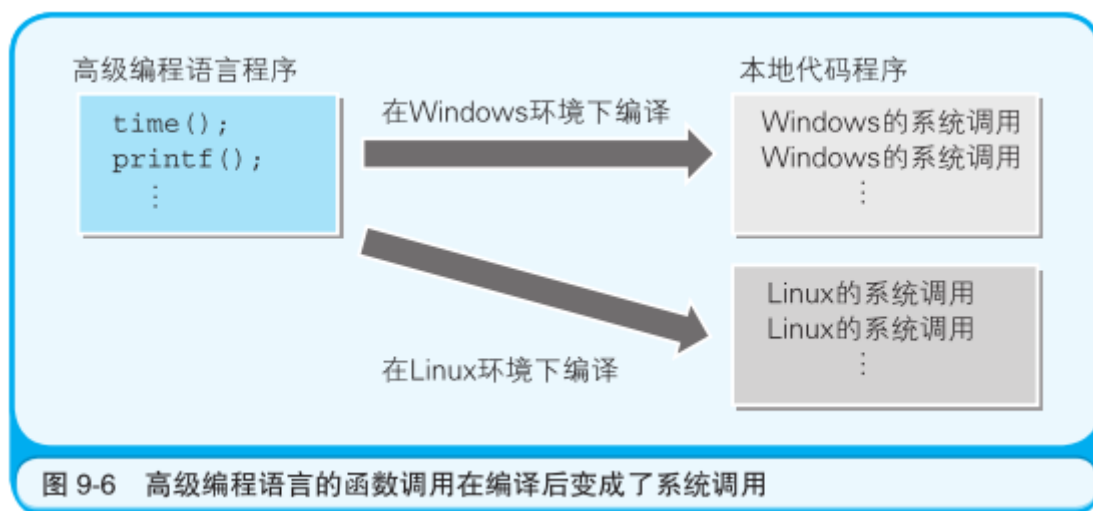
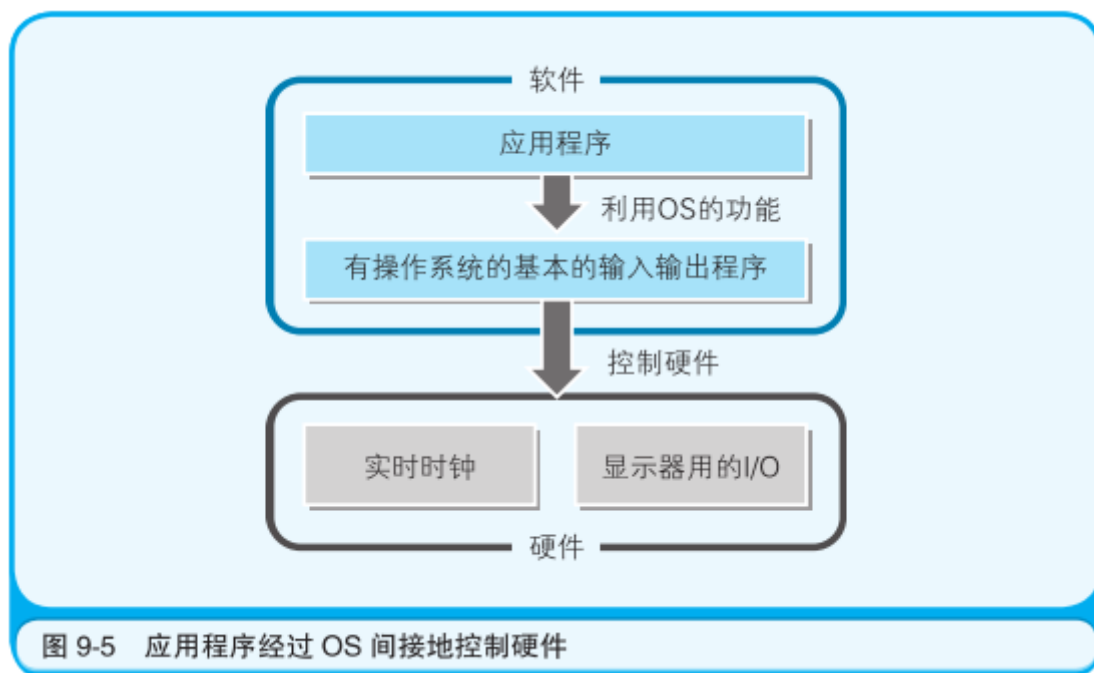
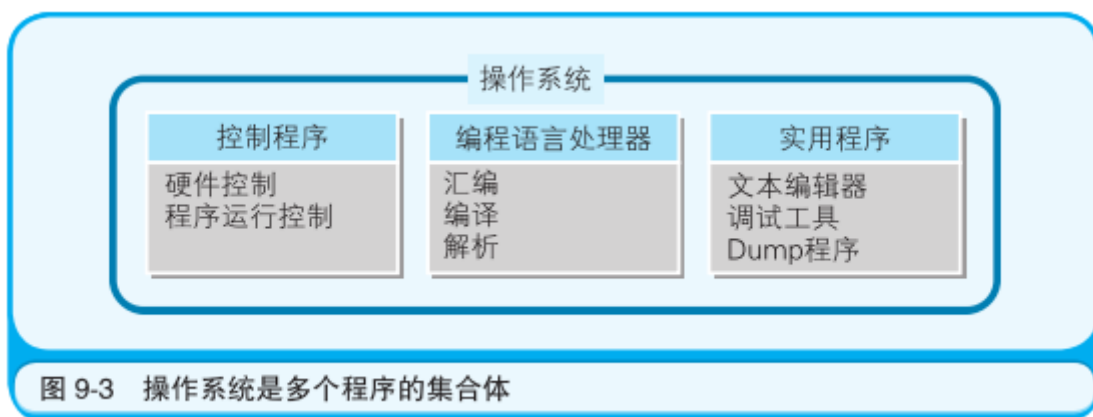
和内存管理相关的“垃圾回收机制”指的是什么呢？

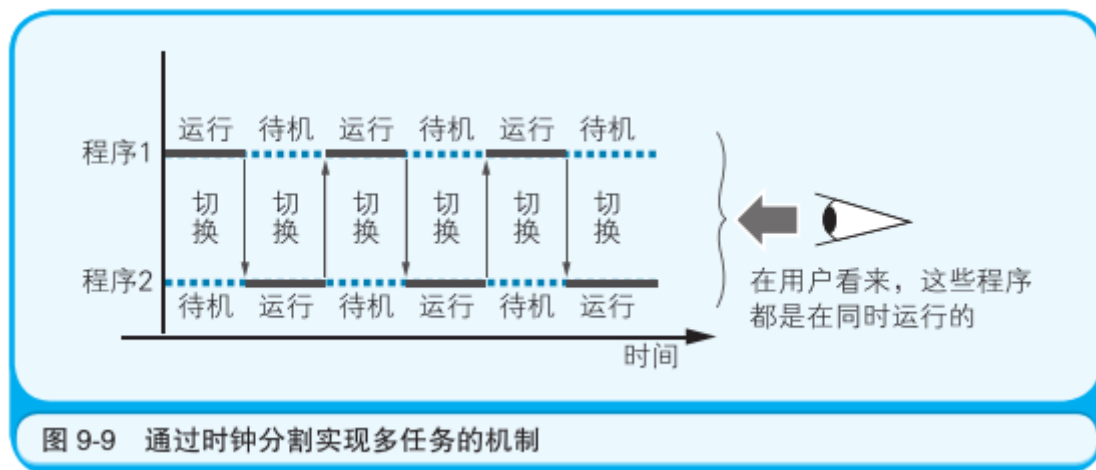
垃圾回收机制（garbage collection）指的是对处理完毕后不再需要的堆内存空间的数据和对象 进行清理，释放它们所使用的内存空间。在 C++ 的基础上开发出来的 Java 及 C# 这些编程语言中，程序运行环境会自动进行垃圾回收。

## 操作系统和应用

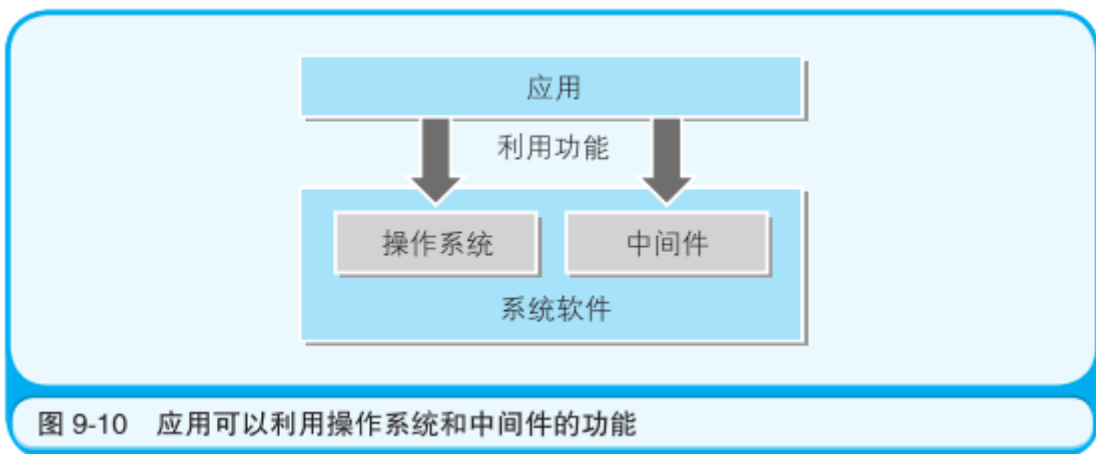








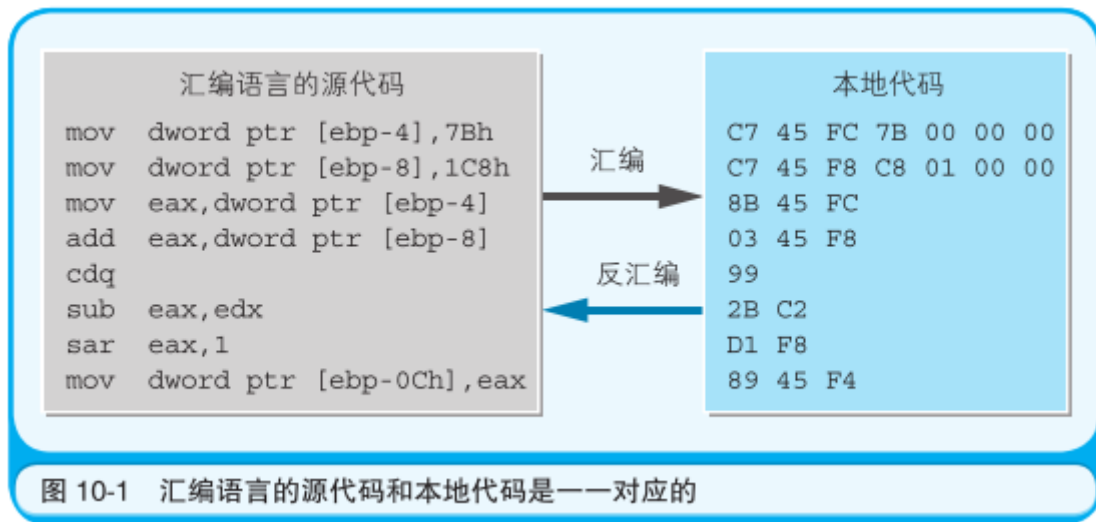
网络功能和数据库功能，虽并不是操作系统本身不可欠缺的功能，但因为它们和操作系统很接近，所以被统称为 中间件而不是应用。意思是处于操作系统和应用的中间（middle）。操作系统和中间件合在一起，也称为 系统软件。



## 汇编

汇编语言和本地代码是一一对应的。

在加法运算的本地代码中加上 `add`（`addition` 的缩写）、在比较运算的本地代码中加上 `cmp`（`compare` 的缩写）等。这些缩写称为 助记符，使用助记符的编程语言称为 汇编语言。



汇编语言的源代码，是由转换成本地代码的指令和针对汇编器的 伪指令构成的。伪指令负责把程序的构造及汇编的方法指示给汇编器（转换程序）。不过伪指令本身是无法汇编转换成本地代码的。

## 汇编语言的语法是“操作码 + 操作数”

32 位 x86 系列 CPU 用的操作码：

表 10-1 代码清单 10-2 中用到的操作码的功能

操作码	操作数	功 能
mov	A, B	把 B 的值赋给 A
and	A, B	把 A 同 B 的值相加，并将结果赋给 A
push	A	把 A 的值存储在栈中
pop	A	从栈中读取出值，并将其赋给 A
call	A	调用函数 A
ret	无	将处理返回到函数的调用源

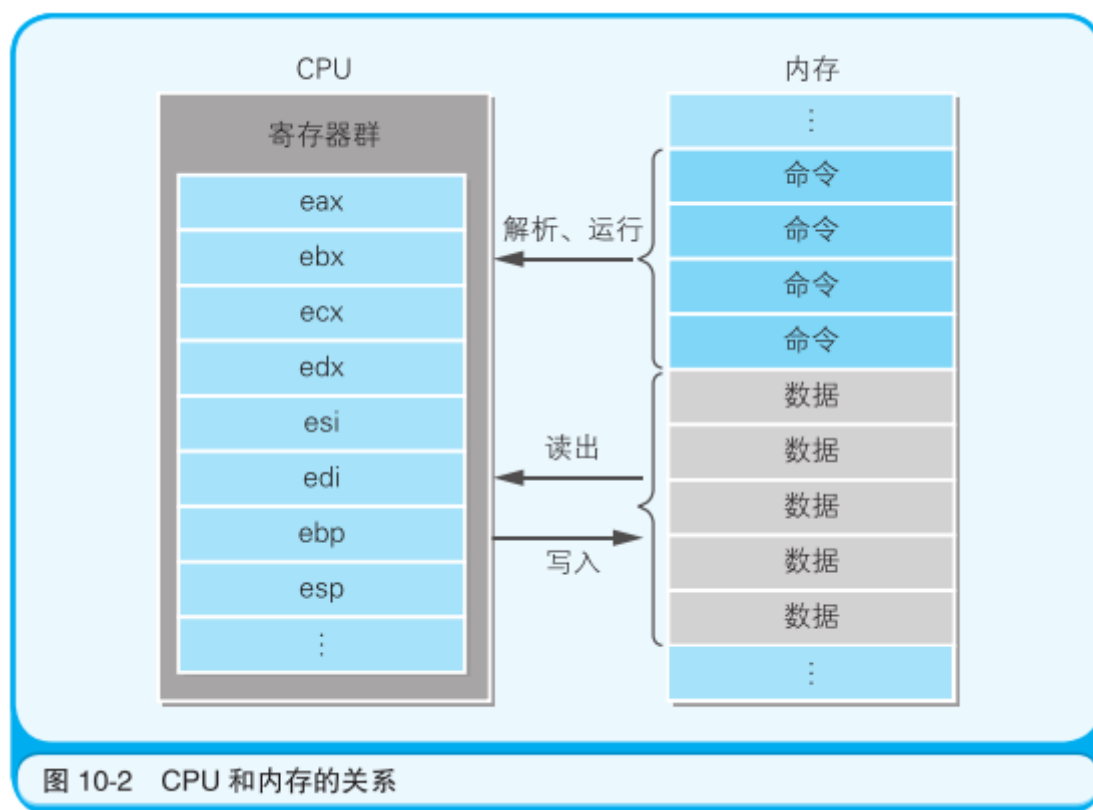
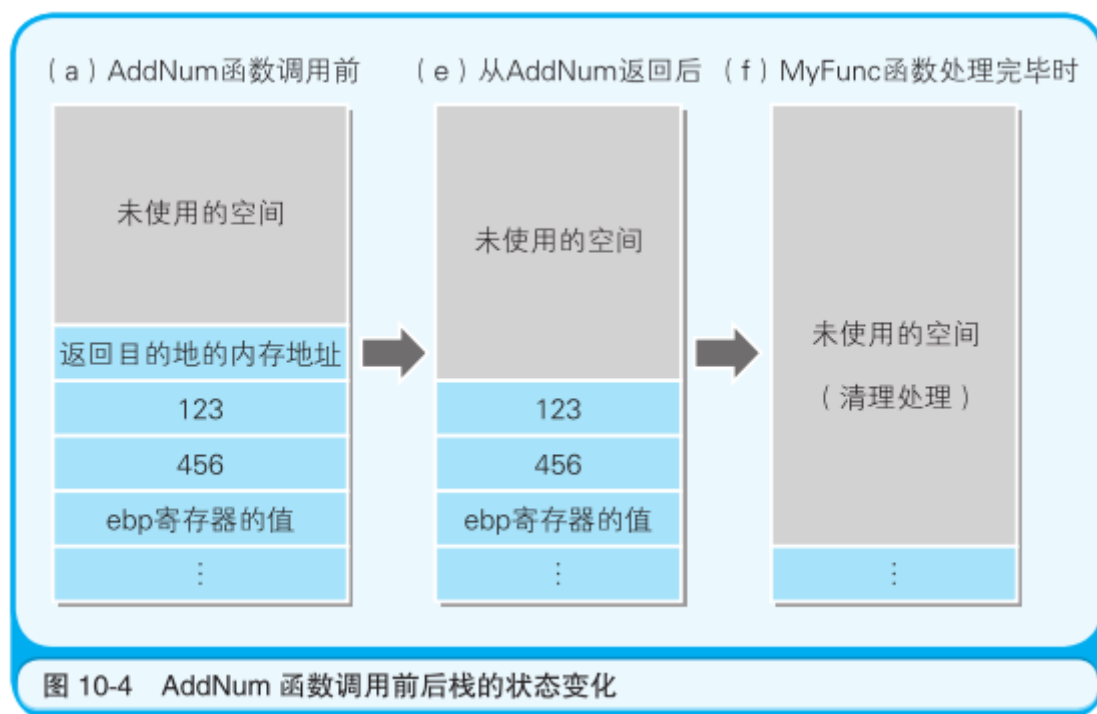
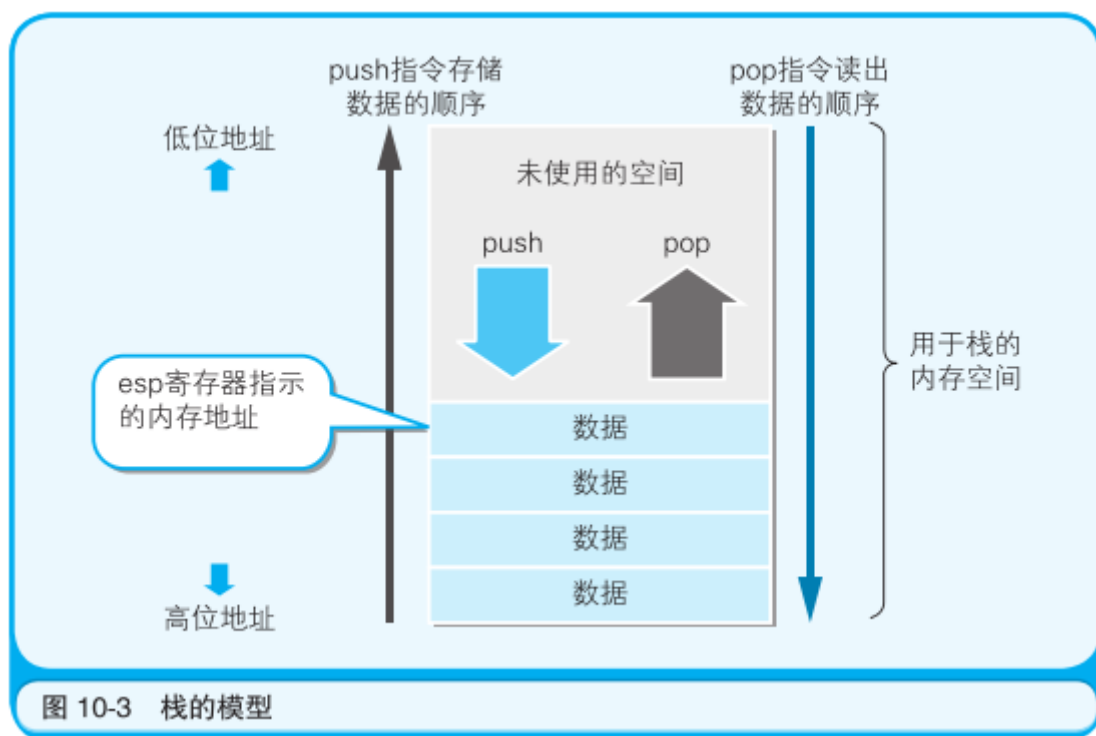


图 10-2 CPU 和内存的关系

寄存器并不仅仅具有存储指令和数据的功能，也有运算功能。

表 10-2 x86 系列 CPU 的主要寄存器<sup>①</sup>

寄存器名 <sup>②</sup>	名 称	主要功能
<code>eax</code>	累加寄存器	运算
<code>ebx</code>	基址寄存器	存储内存地址
<code>ecx</code>	计数寄存器	计算循环次数
<code>edx</code>	数据计数器	存储数据
<code>esi</code>	源基址寄存器	存储数据发送源的内存地址
<code>edi</code>	目标基址寄存器	存储数据发送目标的内存地址
<code>ebp</code>	扩展基址指针寄存器	存储数据存储领域基点的内存地址
<code>esp</code>	扩展栈指针寄存器	存储栈中最高位数据的内存地址



函数 的参数是通过栈来传递，返回值是通过寄存器来返回的。

## 硬件控制

IRQ 指的是用来执行硬件中断请求的编号。

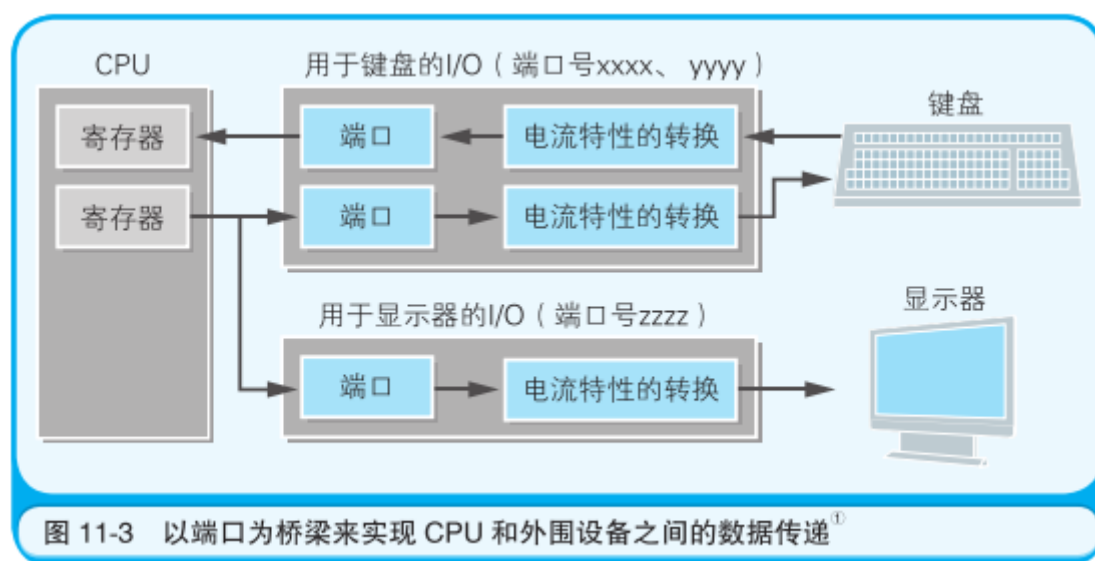
DMA指的是，不经过 CPU 中介处理，外围设备直接同计算机 的主内存进行数据传输。

像磁盘这样用来处理大量数据的外围设备都具有DMA功能。

计算机主机中，附带了用来连接显示器及键盘等外围设备的连接器。而各连接器的内部，都连接有用来交换计算机主机同外围设备之间电流特性的 IC。这些 IC，统称为 I/O 控制器。

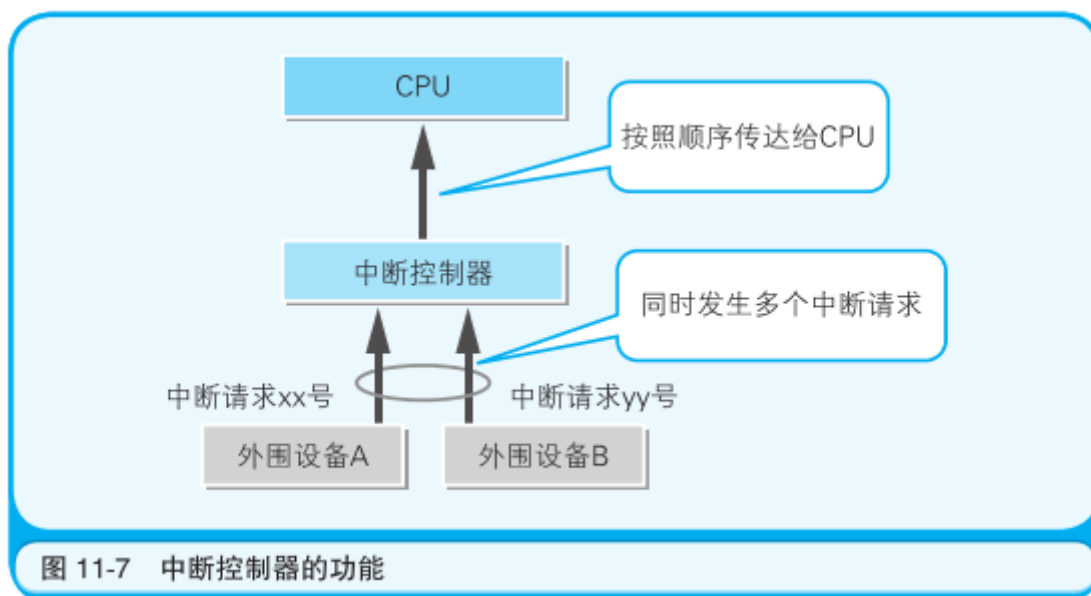
I/O 控制器中有用于临时保存输入输出数据的内存。这个内存就是 端口。端口（port）的字面意思是“港口”。由于端口就 像是在计算机主机和外围设备之间进行货物（数据）装卸的港口，所 以因此得名。

I/O 控制器内部的内存，也称为寄存器。虽然都是寄存 器，但它和 CPU 内部的寄存 器在功能上是不同的。CPU 内部的寄存 器是用来进行数据运算处理的，而 I/O 寄存 器则主要是用来临时存储 数据的。



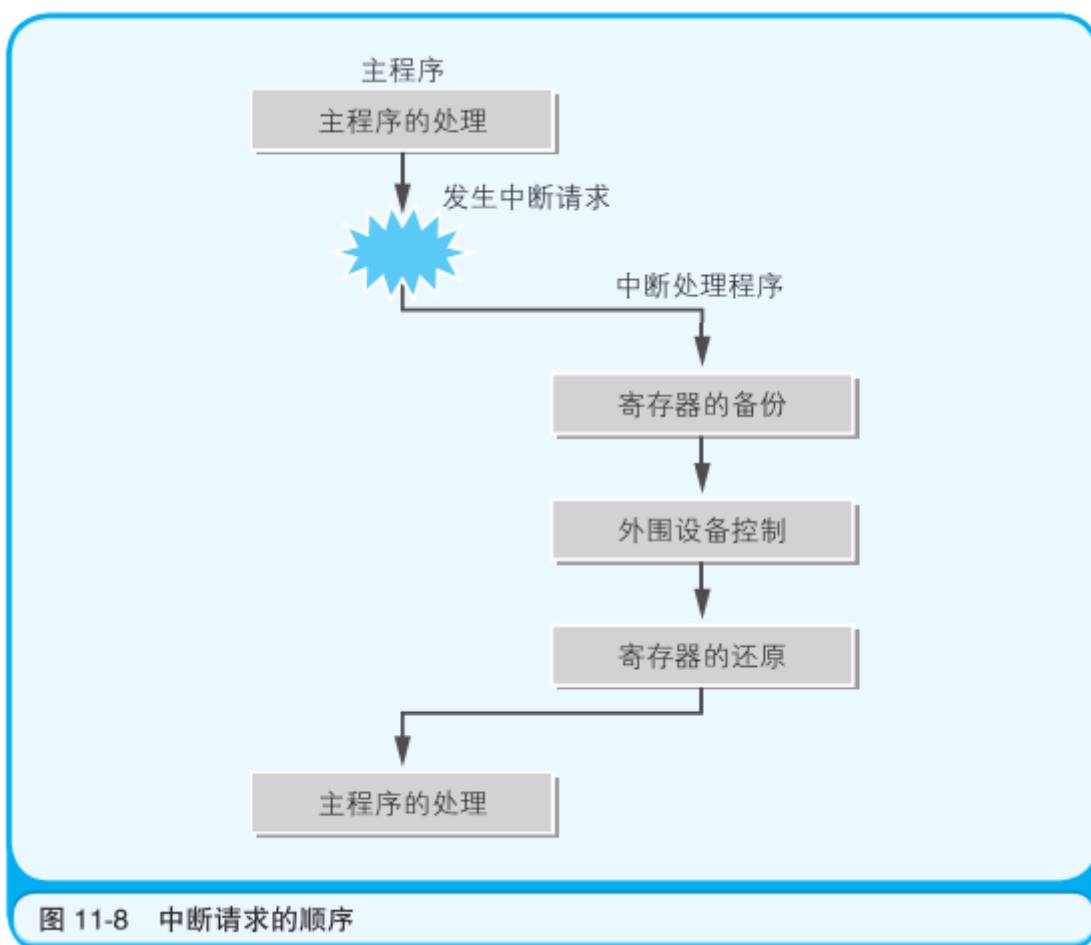
## # 外围设备的中断请求

IRQ 是用来暂停当前正在运行的程序，并跳转到其他程序运行的必要机制。该机制称为 中断处理。中断处理在硬件控制中担当着重要角 色。因为如果没有中断处理，就有可能出现处理无法顺畅进行的情况。



CPU 接收到来自中断控制器的中断请求后，会把当前正在运行的主程序中断，并切换到中断处理程序。中断处理程序的第一步处理，就是把 CPU 所有寄存器的数值保存到内存的栈中。在中断处理程序中完成外围设备的输入输出后，把栈中保存的数值还原到 CPU 寄存器中，然后再继续进行对主程序的处理。

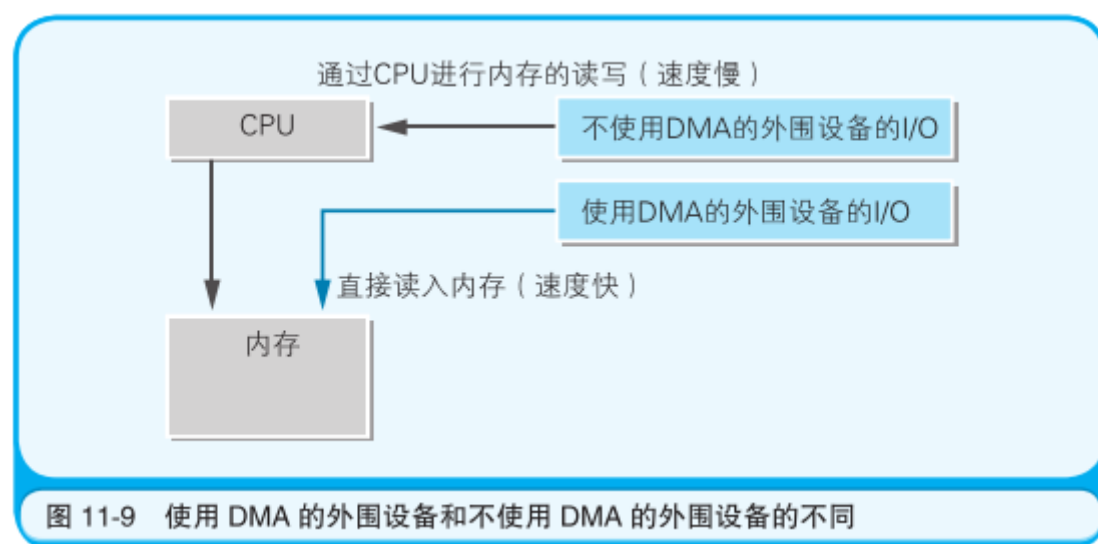
只要寄存器的值保持不变，主程序就可以像没有发生任何事情一样继续处理。



由于外围设备有很多个，因此就有必要按照顺序来调查。按照顺序调查多个外围设备的状态称为轮询□。对几乎不产生中断的系统来说，轮询是比较合适的处理。不过，对计算机来说就不适合了。举例来说，假如主程序正在调查是否有鼠标输入，这时如果发生了键盘输入的话，该如何处理呢？结果势必会导致键盘输入的文字无法实时地显示在显示器上。而通过使用中断，就可以实现实时显示了。

## # DMA可以实现短时间内传送大量数据

DMA是指在不通过CPU的情况下，外围设备直接和主内存进行数据传送。磁盘等都用了这个DMA机制。通过利用DMA，大量数据就可以在短时间内转送到主内存。之所以这么快，是因为CPU作为中介的时间被节省了。



I/O 端口号、IRQ、DMA通道可以说是识别外围设备的 3 点组合。不过，IRQ 和DMA通道并不是所有的外围设备都必须具备的。

## # 文字及图片的显示机制

那就是显示器中显示的信息一直存储在某内存中。该内存称为 VRAM（Video RAM）。在程序中，只要往VRAM中写入数据，该数据就会在显示器中显示出来。实现该功能的程序，是由操作系统或 BIOS 提供，并借助中断来进行处理的。



在现在的计算机中，显卡等专用硬件中一般都配置有与主内存相独立的 VRAM 和 GPU（Graphics Processing Unit，图形处理器，也称为图形芯片）。这是因为，对经常需要描绘图形的Windows来说，数百兆的VRAM是必需的。而为了提升图形的描绘速度，有时还需要专用的图形处理器。

但不管怎样，内存VRAM中存储的数据就是显示器上显示的信息，这一机制是不变的。

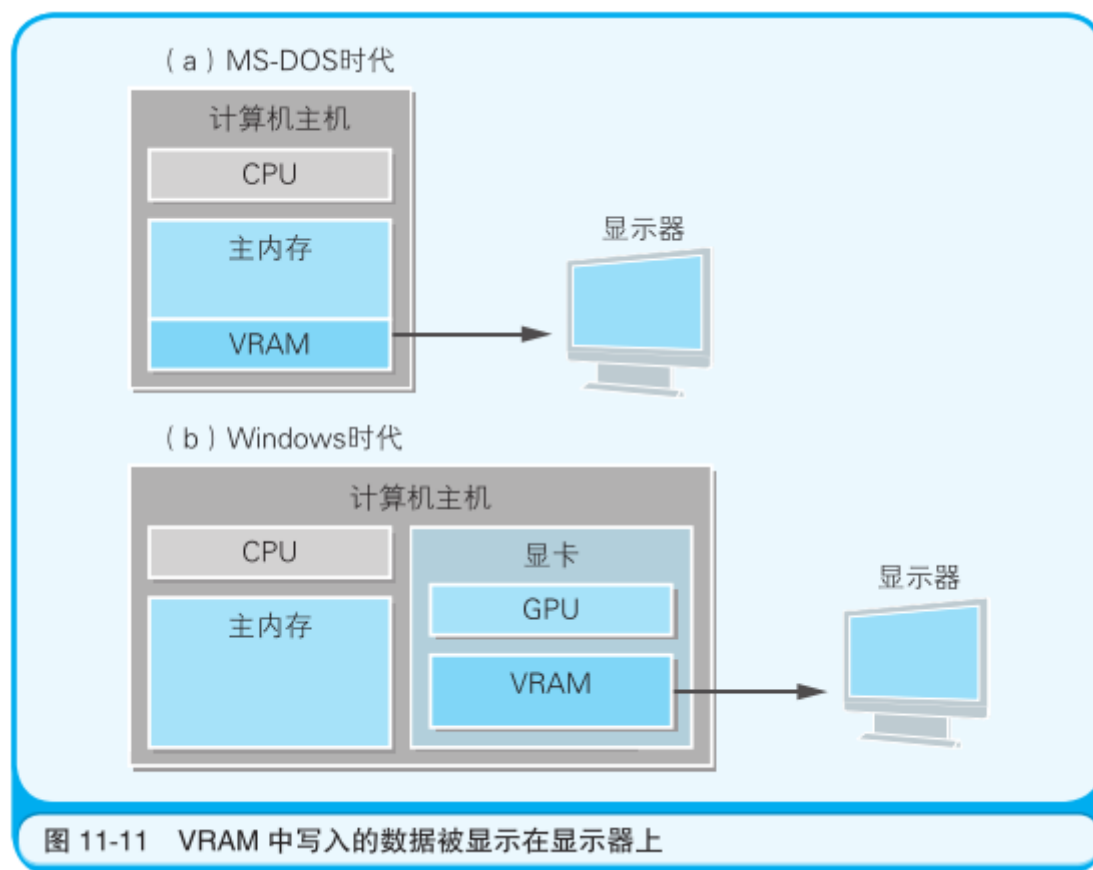


图 11-11 VRAM 中写入的数据被显示在显示器上

## 思考

### # 程序生成随机数的方法

线性同余法：

$$R_{i+1} = (a \times Ri + b) \bmod c$$

对  $a$ 、 $b$ 、 $c$  各参数设定合适的整数后，可以从该公 式获得的随机数的范围就是 0 到  $c$ 。

而假如在不运行 `srand(time(NULL));` 的情况下重复调用 `rand()` 函数 的话，会出现什么情况呢？因为  $Ri$ 、 $a$ 、 $b$ 、 $c$  的数值都有默认值，因此 每次都会生成以相同方式出现的随机数。这样一来，游戏以及计算机 模拟就都无法成立了。当然也就无法表示人类的思考了。