第 1 章　图象文件的基本概念

要认识图象文件，必须从文件的编码原理、文件实例以及存取图象文件的程序三方面来着手。读者在研究数字图象文件时，可能会产生下列的疑问。

(1) 所谓数字图象到底是什么？

(2) 为什么图象文件需要有识别信息与压缩原理来进行编码？

(3) 什么样的文件实例最有代表性呢？

(4) 存取图象文件的程序要怎样编写？

这些都是在认识图象文件过程中，必须解决的基本问题。当这些问题取得圆满的答案时，才算是确实掌握了有关数字图象文件的知识。

1.1　何谓数字图象

图象数字化是电脑图象处理最基本的步骤，其意义就在于把真实的图象，转变成电脑所能接受的格式，也就是一连串特定的数字。一般在市面上常见的图象扫描仪所处理的就是这样的过程。通常这个数字化过程还可以再细分为采样与量化处理两个步骤。其中采样的结果就是通常所说的图象分辨率，而量化的结果则是图象所能容纳的颜色总数。

1.1.1　采样处理

采样的意义就是要使用多少点来表示一张图象。例如，一幅 640×480 的图象，就表示这幅图象是由 307200 个点所组成。当然，想要有更清楚的图象质量，就得使用更多的点来表示图象，也就是让这幅图象拥有较高的分辨率。但是，相对所付出的代价，就是需要更大的存储空间。如何在视觉效果可以接受的范围内，减少所需要的存储空间，就是采样时最基本的问题。

1.1.2　量化处理

量化的意义则是指要使用多大范围的数值，来表示图象采样之后的每一个点。这个数值范围包含了图象上所能使用的颜色总数。例如，以 4 个 Bits 存储一个点，就表示图象只能有 16 种颜色。数值范围越大，表示图象可以拥有更多的颜色，自然可以产生更为细致的图象效果。但是，也同样必须占用更大的存储空间。所以，量化时的基本问题与采样相同，都是视觉效果与存储空间的取舍。

当然，像这种取舍问题是不会有标准答案的，只能视实际应用的状况来决定。视觉效果与存储空间总是有一个要做为牺牲的代价。但是，不论最后的结果怎样，只有在采样与量化处理后，才能产生一张数字化的图象，再运用电脑上种种图象处理的技巧，或是修饰，或是转换，方能进一步达到用户所希望的图象效果。

那么，在图象数字化后，所转换出来的一连串数值，该如何存储呢？图象数据有两种存储方式：(1) 位映射(Bitmap)，(2)向量处理(Vector)。

位映射

可以将图象的每一点数值存放在以字节为单位的矩阵里。举例来说：当图象是单色时，一个字节可存放 8 点图象数据；16 色图象则是以一个字节存 2 点；256 色图象则是一个字节存储 1 点。如此就能表达各种不同颜色模式的图象画面。这种存储方式适用于内容复杂的图象。本书中所要介绍的四种图象文件格式，都是采用位映射方式存储图象数据。

向量处理

只记录图象内容的轮廓部分，而不存储图象数据的每一点。譬如，一个圆形图案只要存储圆心的坐标位置和半径长度，还有圆形边线及内部的颜色。矩形图案则是存储左上角和右下角两个点的坐标位置、矩形边线和内部的颜色。若是存储形状不规则的图案则比较费事，必须记录图案轮廓的每一点。

向量处理比较适合存储商用图表和工程设计图，这类图象内容多半是一些几何图形，如矩形、圆形、椭圆和多边形等等。以向量来记录几何图形，可为文件节省下许多数据量。但是，若采用向量记录一幅内容繁杂、图案形状多变的画面，反而会产生大量的向量数据，可能远超过该图象画面以位元映射存储的数据量。况且要计算出图象中每个图案的坐标位置，必须耗费很长的时间执行一些复杂的分析演算，才可能产生所有的向量数据而后存档。因此，一般图象文件在存储图象数据时，很少采用向量处理方式。

1.2 图象文件的编码原理与实例

各种图象文件的制作方式，其实有着共同的编码原理。每种图象文件内除了图象数据之外，都免不了要存储一些识别信息。试想一个图象文件若是只存储数字图象数据的话，程序必定难以解读出正确的图象数据。因此，在图象文件内部必须建立起一些识别信息，用以定义图象的各项参数，如图象的宽度和高度、颜色种类、调色板数据…等等，唯有如此才能避免程序读取数据时发生错误。

通常一个图象文件内只要有识别信息和图象数据，就已经是一个完整的图象文件，可以供程序任意存取。不过，图象内容经常是一批庞大的数据，若不经过压缩处理就直接存入文件，很容易耗尽磁盘的存储空间。所以，图象文件多半会运用某种压缩原理，减少存储图象所需的数据量，以达到节省存储空间的效果。

所以，在图象文件编码过程中，图象数据和识别信息是必不可少的两项基本单元，而压

缩原理则是经常被采用的要素。目前图象文件之所以会有种种不同类型的格式，主要在于在文件编码的过程中，定义了不同的识别信息和压缩方法。读者若能理解识别信息的用途和压缩原理的编码规则，就不难读写各类图象文件，甚至于自行设计出一种图象文件格式。

1.2.1 图象文件的识别信息

图象文件的识别信息除了有定义图象的各项参数之外，实际上，文件本身也需要有一些识别信息，才能供程序分辨出某个文件究竟属于何种图象文件格式。所以，图象文件的识别信息包括了文件识别与图象识别两类数据。这些识别信息通常都设计成固定的数据结构，并置于文件的最前端，以方便程序读取与识别。以下将分别说明文件识别信息与图象识别信息内容：

文件识别信息

文件识别信息通常包括图象文件的识别码与版本代号。识别码用以判断这个文件应为哪种图象文件格式，例如，PCX 图象文件的识别码为 10(0x0A)，GIF 图象文件的识别码则为"GIF"。版本代号则用来判断同类文件是属于哪一时期的版本。因为图象文件在不同时期所制定的版本，可能采用了不一样的编码方式。譬如，版本代号为 3 的 PCX 图象文件内没有调色板数据；而版本代号若为 2、4、5 的 PCX 图象文件内，则存放了调色板数据。通常文件识别信息都是放在图象文件的前端，称为图象文件的表头。

图象识别信息

图象识别信息方面的重要项目有：图象的宽度和高度、一个图象点所需的 Bits 数、数据压缩方式代码和调色板数据。这四项信息中的压缩方式代码和调色板数据，倒不是每个图象文件都有的数据。当文件所存的图象数据是不超过 256 色时，才会有调色板数据。图象文件若是允许采用多种不同的压缩方式，才需要一个代码做为压缩方式的识别码。除此之外，各类图象文件会基于个别的需要，定义一些其它的识别信息，如打印机的分辨率、图象每行的字节总数…等等。

虽然，在实际的图象文件格式中，识别信息的种类与排列位置，或多或少都会有一些各自不同的设计；但是，图象文件内总少不了上述几种重要的识别信息。图象文件必须提供这些识别信息，程序才能够正确无误地读取出图象数据。

1.2.2 图象文件常用的压缩原理

数字图象通常都会占用很大的存储空间，为减少这方面的需求，就要采用所谓的图象压缩原理。虽然，在理论上，可行的压缩方式是非常多样化，但是，在实际上，图象文件经常采用的却只有少数几种类型。所以，读者如能先弄清几种常用的压缩原理，自然可以掌握住图象

文件编码时的关键，并悠游自在畅行于图象文件世界里。

一般图象文件多半采用下列三种压缩原理：

RLE 压缩原理

这是 Run-Length Encoding 的缩写，可以翻译成行程编码。这类压缩原理是专门找寻连续重复的数值，再以两个字节值加以取代。当图象数据出现连续重复的数值时，就可以在这个数值的前面，加上一个长度值，表示这个数值重复的次数。然后就用这两个编码取代一串连续重复同一数值的数据。如此自然可以得到压缩的效果。

通常代表重复次数的编码，会将前端的一个或两个 Bits 定义为标志，提示程序这个编码的其余 Bits 值代表数据重复的次数，下一个编码即是重复的数据值。如此计算出来，这两个字节的编码最多一次可以取代 63 或 127 个字节长度的数据。或者是像 BMP 图象文件将第一个编码完全用来代表数据值重复的数量，可以用两个字节代表 255 个字节的数据，这就是 RLE 压缩原理编码所能达到的压缩上限。

不过，RLE 压缩原理只能压缩处理连续重复同一数值的数据串，若是遇到数值不同的连续数据，就只能够将数据原封不动地存入图象文件内。无法压缩处理不同值的数据串，是 RLE 压缩原理的一项严重缺陷。若是采用 RLE 压缩原理的图象文件，想要存储的图象数据内缺少同值的数据串，将无法得到良好的压缩效果，甚至于发生不减反增的后果。也就是说，图象数据经由 RLE 压缩原理处理过后，数据量没有减少，反而比未压缩前更多。

LZW 压缩原理

这是三个发明人名字的缩写(Lempel，Ziv，Welch)，这项压缩原理本来是专为文字数据所设计，但是有一些图象文件却利用 LZW 来压缩图象数据。这种压缩原理是将每一个字节的值都要与下一个字节的值配成一个字符对，并为每个字符对设定一个代码。当同样的一个字符对再度出现时，就用代号代替这一字符对，然后再以这个代号与下个字符配对。在配对过程中，必须建立三份表格，分别是：字首表、字符表和代号表。所有字符对和代号，便分别存入这三份表格内。

LZW 压缩原理规定代号的长度固定为 12 个 Bits，所以代号的最大值只有 4095，而三份表格只需要存放四千多个元素即可。不过，字符对可能不止 4095，所以代号很容易就会用尽，必须经常重新为字符配对和填表。另外，LZW 压缩原理规定 0 到 255 都是原始字符值，而 256 及 257 则是做为特殊码，不能当作代号使用，因此第一组字符对的代号是 258。

由于代号长度只有 12 个 Bits，比一对原始字符少了 4 个 Bits，所以用代号取代原始数据，能够得到压缩效果。除此之外，一个代号所代表的字符对，也有可能是另一个代号与原始字符的组合，因此这类所取代的字符数量就不止是两个原始字符了。

譬如，假设有一串数据一开始连续 9000 个字节的值都是 0x41，则在配对过程中，除了第一对字符是 0x41 和 0x41，其余字符对的字首都是代号，而字尾则固定是 0x41。因此，第二对字符代号 259 所代表的 258 和 0x41，实际上是 3 个原始字符 0x41。第三对字符代号 260

则代表 4 个原始字符 0x41,…最后一个代号 4095 所代表应该是 3839 个原始字符 0x41。

由此可知一个长度为 12Bits 的代号最多可代表 3839 个原始字符,这样的压缩效率即使折半计算也远超过 RLE 压缩原理。并且图象文件采用 LZW 压缩原理时,将固定长度为 12Bits 的代号,改为由 9 个 Bits 长度开始编码。当代码值递增到需要增加 Bit 数时,才将代码长度递增一个 Bit,这样的作法更是提高 LZW 压缩原理的效率。除了以上的优点,LZW 压缩原理还有一个极为重要的特性:代码不仅仅是能取代一串同值数据,也能够代替一串不同值的数据。

前面曾经提过,LZW 压缩原理专门处理文本数据,在文本数据中不可能出现一长串相同的字符。因此,LZW 压缩原理设计出字符配对方式,可以从文章中找出经常重复出现的字符串,然后再用代号取代这类字符串。例如,"information"这个英文单字若在一篇文章中多次重复出现,LZW 压缩原理即可找到一个代号来取代这个单字。同理,在图象数据中若有某些不同值的数据经常重复出现,也能找到一个代号取代这些数据串。而 RLE 压缩原理却无法压缩不同值的数据串,只能保持原状存档。在压缩处理不同值的数据串方面,LZW 压缩原理也是优于 RLE 压缩原理。

Huffman 压缩原理

Huffman 是创造这项压缩原理的发明人名字。这种压缩原理是采用不固定长度的编码,来取代原始数据。不过,Huffman 压缩原理不像前两种原理那样,以一个编码代表一串数据值,Huffman 压缩原理所产生的一个编码只代表一个 Byte 值。这些 Huffman 编码的长度最少可以只有一个 Bit,而 Huffman 压缩原理就是利用一些不足 8 个 Bits 的编码,来取代原始数据中经常出现的一些 Byte 的值,由此即可达到很不错的压缩效果。

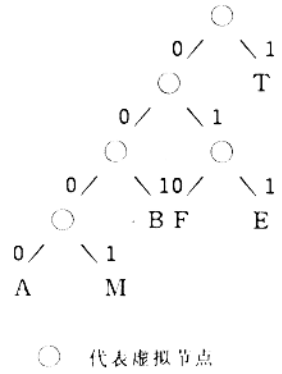
在压缩过程中必须读取所有原始数据两次。第一次是为了计算数据出现的频率,并以较短的码来代替较常出现的数据,以较长的码来代替较少出现的数据,而这些不同长度的编码,都存放在一个表格矩阵中。第二次读取原始数据时,再以表格内的编码取代原始数据存入图象文件内,这样一来大部分的原始数据就会被一些较短的编码所代替了。

当第一次读取原始数据,计算出不同 Byte 值的出现频率后,必须利用二元树结构来排列每个 Byte 值。首先将每个 Byte 值都视为最底层的节点,接着开始找寻出现频率最小的两个 Byte 值,然后将两个值的频率相加成为父节点的频率。凡是已经取得父节点的节点,就不再需要与最底层尚缺父节点的 Byte 值节点比较频率,而由父节点(为这个节点建立的上一层的父节点)加入比较频率的行列。

就是这样不断地找出频率最小的两个节点,然后建立起父节点。直到最后排列出一个根节点,代表整个二元树结构的最顶层。于是排在越上层的 Byte 值表示出现频率越高,位在越底层的 Byte 值表示出现频率越低。在这个二元树中会有许多个虚拟节点只代表着某个频率,并不代表 Byte 值。

以每个 Byte 值的频率顺序排出二元树结构之后,再建立一个表格存储 Byte 值及其编码。位于二元树结构内越高层的 Byte 值,将得到较短的编码。Huffman 压缩原理的编码方式是:左边节点为 0,右边节点为 1,位于下层的数据编码将继承其上层的节点编码。例如,在图

1.1 的二元树内，原始数据字符 E 的编码为 011，就是继承了上面第一层的节点编码 0，第二层的节点编码 1，以及第三层的节点编号 1，因此取得 3 个 Bits 长度的编码 011。



原始数据	编码
A	0000
B	001
E	011
F	010
M	0001
T	1

图 1.1

读者注意上表内的编码，将会发现较长编码的前端（由编码左边第一个 Bit 开始）绝对没有重复较短编码的值，这是 Huffman 压缩原理的一个重要编码原则。唯有如此，在程序解码时，才不会出现判断错误的情况。例如，字符 B 的节点位置若改变为字符 A 的父节点，则 B 的编码变成 00，而 A 的编码是 0000，前端重复了 B 的编码 00。解码程序读到压缩数据 00，就会与表格内编码做比较，而认为 00 是 B 的编码，将无法在压缩数据中找到字符 A。因此，在以二元树排列 Byte 值时，必须注意到：凡是存有 Byte 值的节点，其父节点和子节点都不能被其它 Byte 值所占用，而且每个 Byte 值的父节点都是一个虚拟节点。

使用 Huffman 压缩原理产生的压缩文件，必须在文件表头部分存放每个 Byte 值的出现频率。这个频率值是浮点数，所以一个频率值占用 4 个字节的数据长度，而 Byte 值为 0 到 255，一共 256 个元素。因为 $4\times256=1024$，所以表头内必须存储长度为 1024 个字节的数据出现频率。解码时，程序需要有这份频率数据，才能重新以二元树结构排列出不同频率的 Byte 值，进而建立起一份正确的编码表用来解读压缩数据。此外，表头内也得存放原始数据的总长度，解码程序才知道正确的原始数据共有多少个字节。

修改图象数据内容

在本书中要谈论的四种图象文件所运用的压缩法，大多数是出自于上述三种压缩原理。不过，有一些压缩方法在压缩数据之前，会先修改图象数据内容。将数据修改后再进行压缩编码，往往能够提高压缩效率。例如，TIFF 图象文件使用的 LZW 压缩法，可以在压缩数据前先将每个 Byte 值减去前一个 Byte 值，然后才开始压缩数据程序，这项修改数据的方法称为 Predictor。

最特别的是 TIFF 图象文件所采用的 JPEG 压缩法(TIFF 图象文件允许使用多种压缩方法),在数据压缩编码之前,图象数据得先经过三道处理程序:彩色模式变换及采样、DCT 变换和量化。彩色模式变换是将 RGB 全彩色值变换成 YCbCr 图象数据。采样则是只保留部份 Cb 和 Cr 数据。DCT 变换全名为离散余弦变换,是利用一个三角函数计算公式,将 YCbCr 图象数据变换成频率系数。这些频率系数都是浮点数,必须再以一次量化手续转换成整数,然后才开始进行压缩编码。

JPEG 是目前压缩效率最高的图象压缩方法,它主要是运用了修改数据内容和数据采样的方式,来提高数据压缩的效率。此外,这项压缩方式在压缩编码上,还结合了 RLE 与 Huffman 两种压缩原理。有关 JPEG 压缩方式的演算过程,在本书第五章中将会有更为详细的说明。

读者应当了解,通常具有良好压缩效率的方法,往往有着较为复杂的演算方法,需要花费比较多的时间来转换编码。至于像 RLE 这种压缩原理虽然压缩效果不佳,但是演算方法很简单,使得读写图象文件的速度相对提高了许多。就因为有着这种存储空间与读写时间快慢的冲突,才会造成不同压缩效率的方法得以并存于世。

1.2.3 具代表性的图象文件实例

虽然目前电脑界流行着多种图象文件,而这些图象文件格式可能截然不同,造成文件格式不同的主要差异在于识别信息的种类与数量,以及压缩原理的运用技巧。其实所有的图象文件都只是将编码原理各自变化应用。所以,只要挑出一些具代表性的文件实例,读者就可以完整掌握图象文件的编码原理,日后若是要再研究其它的图象文件,也能感受到驾轻就熟的快意。

作者认为能够称之为具有代表性的图象文件,必须符合下列两个条件。首先必须是相当常用,在电脑界广为流传的图象文件。其次则是足以示范上述所提的各种编码原理,让读者能够从其中了解到图象文件编码的要诀。下列四种图象文件,就是作者认为具备了这两项要素的代表性文件实例。

PCX

在 PC 世界里,历史最悠久,流行最广的图象文件格式。采取 RLE 压缩原理,适合做为进入图象文件领域的踏板。

BMP

是 Windows 3.0 所定义的基本图象文件。图象数据处理方式有不压缩与压缩两种,而压缩方式也是采用 RLE 压缩原理,可与 PCX 图象文件做一个对照。

GIF

是 BBS 上广为流传的文件格式。采用 LZW 压缩原理为基础来压缩图象数据,能够有效压缩文件容量,得以节省大量传输时间。

TIFF

是排版与图象扫描仪最常用的图象文件格式。其文件内部运用指针功能，建立了一个开放式的架构，可以包容多种不同的识别信息和压缩方式。譬如，TIFF 图象文件所能使用的多种压缩方式，便涵盖了前述三种压缩原理。这种开放多样的编码方式，足以提供读者一种新的图象文件设计观点。

作者相信通过对这四种文件格式的了解，将足以印证上述文件编码原理。并且当读者需要研究其它不同的图象文件格式时，以本书所得做为基础，也可以更容易入手。

1.3 本书程序结构介绍

本书所附程序内容，分别写于下面 10 个文件内。

IMAGE.H
IMAGEMEM.C
IMAGEERR.C
IMAGEPCX.C
IMAGEBMP.C
IMAGEGIF.C
IMAGETIF.C
IMAGEVGA.C
IMAGEDRV.ASM
IMAGERSW.C

1.3.1 程序功能说明

这 10 个文件是以 IMAGERSW.C 为主程序，若将它与其余八个程序及一个头文件编译、链接在一起，就成为一个小型的 IMAGERSW.EXE 执行文件。IMAGERSW.EXE 可算是一个简单的应用软件，具备以下四项功能：

(1) 能够读取 PCX、BMP、GIF 和 TIFF 等四种图象文件内容。

(2) 将图象数据显示于屏幕上。

(3) 图象若大于屏幕，可利用移位键移动画面。

(4) 在退出程序之前，可以按数字键<1>、<2>、<3>、<4>，选择将图象文件存入 PCX、BMP、GIF 和 TIFF 等四种图象文件格式；或者是按下<Esc>键，直接退出程序。

以下分别简介这十个文件的内容，并列举出文件内所定义的公用函数：

IMAGE.H

定义一个数据结构 Image、一些错误代码常数以及其它 9 个程序文件内定义的一些公用函数。其中数据结构 Image 的内容说明如下：

```
typedef struct{
        unsigned Type,
        unsigned Width,
        unsigned Height,
        unsigned PageSize;
        unsigned char * * Data,
        unsigned char * Color;
        } Image;
```

Type

表示图象类型，数值可能是 1 到 3，分别代表不同的图象模式：1 代表单色，2 代表 16 色，3 代表 256 色，这是本书程序所能处理的几种图象模式。以这三种图象模式为限，主要是符合标准 VGA 显示卡的功能。至于 24Bits 全彩色图象，因为标准 VGA 显示卡无法处理，所以本书所附程序不处理它们。

Width

代表图象水平方向长度，也称为图象宽度，是以点数为基本单位来计算的。

Heigth

代表图象垂直方向长度，也称为图象高度，是以点数为基本单位来计算的。

PageSize

指出图象分页的大小。由于单页数据有 64KB 的限制，所以必须将大于 64KB 限制的图象，以分页方式来存储。这个值代表每个分页存储几行。

Data

表示图象数据指针数组。由于图象数据可能需要存入数个分页内，所以利用指针数组，分别指向各个分页的起始地址。

Color

指示调色板数据的起始地址。如果值为 NULL，则表示图象文件内没有这个数据。

有了这个图象数据结构之后，不同的图象文件读写函数就有了图象存取的共同基础。既

可以借此存放取自图象文件的图象数据，又能够将图象数据从这里再写入另一种图象文件中，很自然地完成了转换文件格式的功能。

请读者特别注意：本书程序对 16 色图象数据的存储方式，是先将一行数据分成四个 Bit Planes，再存入存储单元内。不过，每一行的四个 Bit Planes 图象数据是连续存放在一起，而不是将四个 Bit Planes 图象数据分别存入不同页的存储单元中。所谓的连续存放如图 1.2 所示，在存储单元内 Line0 的 Bit Plane0 到 Bit Plane3 图象数据，是串联在一起存放，紧接在后面的则是 Line1 的四个 Bit Planes 图象数据。

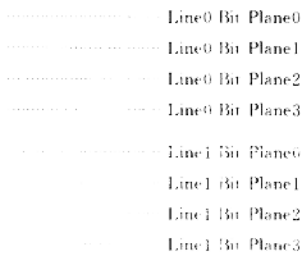


图 1.2

IMAGEMEM.C

申请与释放图象数据所需存储单元的公用函数：

ImageAlloc( )

ImageFree( )

转换 16 色图象数据的公用函数：

ImageByteToPlane( )

ImagePlaneToByte( )

IMAGEERR.C

设定与取得错误代码的公用函数：

ImageErrorSet ( )

ImageErrorGet ( )

IMAGEPCX.C

读取与写入 PCX 图象文件的公用函数：

ImagePcxFileRead ( )

ImagePcxFileWrite ( )

IMAGEBMP. C

读取与写入 BMP 图象文件的公用函数：
ImageBmpFileRead ( )
ImageBmpFileWrite ( )

IMAGEGIF. C

读取与写入 GIF 图象文件的公用函数：
ImageGifFileRead ( )
ImageGifFileWrite ( )

IMAGETIF. C

读取与写入 TIFF 图象文件的公用函数：
ImageTiffFileRead ( )
ImageTiffFileWrite ( )

IMAGEVGA. C

VGA 卡图象显示公用函数：
ImageVgaShow ( )

IMAGEDRV. ASM

负责屏幕显示与键盘控制的公用函数：
VgaCardReady ( )
VgaCardModeGet ( )
VgaCardModeSet ( )
VgaCardPalGet ( )
VgaCardPalSet ( )
KeyboardGet ( )

IMAGERSW. C

这是用来示范图象文件读写功能的主程序文件，运用其它程序文件所提供的公用函数，

来完成图象文件读取、写入和图象显示等功能。

上述 10 个文件中，前七个文件与图象文件读写功能息息相关。读者不妨将下列六个源程序编译后产生的 .OBJ 文件，合并成一个读写图象文件的程序库，方便日后程序设计上的需要。

IMAGEMEM.C
IMAGEERR.C
IMAGEPCX.C
IMAGEBMP.C
IMAGEGIF.C
IMAGETIF.C

至于最后的三个文件都是为了给读者提供一个实际应用的范例而设计的，与图象文件读写功能无关。IMAGERSW.C 主要是当作一个简单的主程序范例，提示读者如何使用本书所提供的公用函数，读取及写入图象文件数据，并且将图象画面显示于屏幕上。IMAGEVGA.C 和IMAGEDRV.ASM 则是执行屏幕显示与键盘控制的功能，颇具实用价值，也许可供读者参考使用。

1.3.2 程序清单配置说明

上述 10 个文件的完整内容，都分别在本书中不同的章节里列出。IMAGE.H、IMAGEMEM.C 和 IMAGEERR.C 就安排在本章的最后一节中。IMAGEPCX.C、IMAGEBMP.C、IMAGEGIF.C 和 IMAGETIF.C 分别安排在稍后四章的最后一节。至于 IMAGEVGA.C、IMAGEDRV.ASM 和 IMAGERSW.C 则放在附录 A 中。读者除了可以在上述章节中找到程序之外，本书所附磁盘也保存了这 10 个文件，以便于读者编译、链接所有的程序。

把 IMAGE.H、IMAGEMEM.C 和 IMAGEERR.C 安排在第一章，是因为必须有这三个文件内容做为图象文件读写函数的基础，才能够让图象文件读写函数有效地执行读写功能。IMAGE.H 声明图象数据结构及相关函数，专供其它程序文件引用，其重要性不必多说。另外两个文件内的函数，也提供了图象文件读写函数必须使用到的重要函数。所以，读者应当在先了解这三个文件的内容之后，才容易明白后面四章内图象文件读写函数的意义。

IMAGEPCX.C、IMAGEBMP.C、IMAGEGIF.C 和 IMAGETIF.C 分别放在与其相关的章节里，是希望读者能够先认识文件格式和压缩方法之后，再来阅读程序部分。按照这样循序渐进的阅读过程，在文件分析与程序演算的对照下，相信有助于读者理解程序文件的内容。

至于 IMAGEVGA.C 和 IMAGEDRV.ASM 这两个程序文件之所以列入附录 A 中，则是考虑到读者可能已经拥有一组专用的屏幕显示与键盘控制程序，不需要使用 IMAGEVGA.C 和 IMAGEDRV.ASM。况且屏幕显示与键盘控制并非本书谈论的主题，所以不把 IMAGEVGA.C 和 IMAGEDRV.ASM 列入正文之中。

另外，IMAGERSW.C 是一个主程序文件，可以和其它的程序文件链接成为一个可执

行文件。这个可执行文件能够读取本书所谈四种图象文件内容、显示图象画面，并可将图象数据写入图象文件内。作者的主要目的是想以 IMAGERSW.C 做为一个示范，让读者更容易明白本书所提供的函数应当如何使用。此外，IMAGERSW.C 也可以说是个具体的图象处理程序。若读者有心自行开发一套图象处理软件，也许 IMAGERSW.C 能够做为开发程序的参考。

1.4 程序清单——IMAGE.H IMAGEMEM.C IMAGEERR.C

1.4.1 IMAGE.H 头文件清单

```
/*********************************************************************
IMAGE.H
    声明图象数据结构与相关函数。
版本:1.00
日期:1995/03/01
作者:李振辉  李仁和
*********************************************************************/
/*C++使用的编译原则,保证C的函数可以在C++编译程序内正常使用。*/
#ifdef cplusplus
  extern "C"
    {

#endif

    /*定义数据类型Image ,其使用方式请参考本章说明。*/

    typedef struct{
                unsigned Type;
                unsigned Width;
                unsigned Height;
                unsigned PageSize;
                unsigned char **Data;
                unsigned char *Color;
                } Image;

    /*以下定义各项错误代码常数*/
```

```
#define SUCCESS      0        /*读文件或写文件成功*/
#define FILEERROR    1        /*找不到文件*/
#define MEMERROR     2        /*存储空间不足 */
#define LABELERROR   3        /*文件识别信息错误*/
#define COLORERROR   4        /*该颜色模式无法处理*/
#define READERROR    5        /*读取识别信息时发生错误*/
#define DECODEERROR  6        /*读取图象数据时发生错误*/
#define WRITEERROR   7        /*写入识别信息时发生错误*/
#define ENCODEERROR  8        /*写入图象数据时发生错误*/

/*以下定义各项可供用户调用的函数，其使用方法请参考各程序文件清单*/

/*IMAGEMEM.C源程序*/
Image *ImageAlloc(unsigned Type,
                  unsigned Width,
                  unsigned Height,
                  unsigned Color);
void   ImageFree(Image *Img);

void   ImageByteToPlane(unsigned char *ImgBuf,
                        unsigned char *LineBuf,
                        unsigned PLineBytes);
void   ImagePlaneToByte(unsigned char *LineBuf,
                        unsigned char *ImgBuf,
                        unsigned PLineBytes);

/*IMAGEERR.C源程序*/

void   ImageErrorSet(int);
int    ImageErrorGet();
void   ErrorMessage(int);

/*IMAGEPCX.C源程序*/

Image *ImagePcxFileRead(char *FileName);
int    ImagePcxFileWrite(Image *Img,
```

```
                        char *FileName,
                        unsigned Mode);

/*IMAGEBMP.C源程序*/

Image *ImageBmpFileRead(char *FileName);
int    ImageBmpFileWrite(Image *Img,
                        char *FileName,
                        unsigned Mode);
/*IMAGEGIF.C源程序*/

Image *ImageGifFileRead(char *FileName);
int    ImageGifFileWrite(Image *Img,
                        char *FileName,
                        unsigned Mode);

/*IMAGETIF.C源程序*/

Image *ImageTiffFileRead(char *FileName);
int    ImageTiffFileWrite(Image *Img,
                        char *FileName,
                        unsigned Mode);

/*IMAGEDRV.ASM源程序*/

int  VgaCardReady();
int  VgaCardModeGet();
int  VgaCardModeSet(int Mode);
void VgaCardPalGet(unsigned Type,
                   unsigned char *Palette);
void VgaCardPalSet(unsigned Type,
                   unsigned char *Palette);
void ImageVgaShow16(unsigned char *Image,
                    unsigned ImgW,
                    unsigned ImgX,
```

```
                        unsigned ImgY,
                        unsigned ScrW,
                        unsigned ScrH,
                        unsigned ScrX,
                        unsigned ScrY,
                        unsigned Mode);
    void ImageVgaShow256(unsigned char *Image,
                         unsigned ImgW,
                         unsigned ImgX,
                         unsigned ImgY,
                         unsigned ScrW,
                         unsigned ScrH,
                         unsigned ScrX,
                         unsigned ScrY);
    int  KeyboardGet();

    /*IMAGEVGA.C源程序*/

    void ImageVgaShow(Image *Img,
                      unsigned ImgX,
                      unsigned ImgY,
                      unsigned ScrW,
                      unsigned ScrH,
                      unsigned ScrX,
                      unsigned ScrY);

#ifdef __cplusplus

    }

#endif
```

1.4.2 IMAGEMEM.C 源程序清单

```
/*************************************************************************
IMAGEMEM.C
```

图象存储空间的申请与释放，16色图象数据转换。

函数：

ImageAlloc( )

ImageFree( )

ImageByteToPlane( )

ImagePlaneToByte( )

版本：1.00

日期：1995/03/01

作者：李振辉　李仁和

```
**********************************************************************/
#include "image.h"
#include "stdlib.h"
#include "limits.h"
/**********************************************************************
ImageAlloc( )
    依据所传入的各项参数，申请图象画面所需要的图象数据结构与存储空间，最后返回该图象数据结构
    的指针。
参数值
    Type    图象类型。1代表单色，2代表16色，3代表256色。
    Width   图象宽度。
    Height  图象高度。
    Color   有无图象调色板数据。
返回值
    成功时，返回所配置图象数据结构的指针。
    失败时，则返回NULL。
**********************************************************************/
Image *ImageAlloc(unsigned Type,
                  unsigned Width,
                  unsigned Height,
                  unsigned Color)
{
  Image *Img;

  /* 申请图象数据结构所需的存储单元 */

  if((Img = (Image*)malloc(sizeof(Image))) != NULL)
    {
```