第一章　引言

近年来，随着计算机技术的发展，数据压缩技术的研究受到人们越来越多的关注。尤其是多媒体技术的兴起，更加促进了有关理论和技术的研究与开发，并使之应用于工程实践。我们知道，多媒体计算技术的关键问题之一是实时地处理语音、图象和文本数据。除了硬件设备和系统软件的支持外，需要解决的一个重要的问题是大量数据的存储、处理和传输。数据压缩技术就能够比较有效的解决这个问题。

在多年研究和探讨的基础上，已经产生了许多能实现各种不同数据压缩技术的硬件和软件产品。在这本书中，我们不可能、也不准备全面介绍每一种技术和方法，仅从应用的角度介绍一些常用的数据压缩技术、算法及其软件实现。

1.1　什么是数据压缩

数据压缩最初是做为信息论研究中的一个重要课题，在信息论中被称为信源编码。但近年来，数据压缩已不仅限于编码方法的研究与探讨，已逐步形成较为独立的体系。它主要研究数据的表示、传输和转换方法，目的是减少数据所占据的存储空间和传输时所需用的时间。例如，在通讯工程中为了能在存储设备容量、信道带宽、或通讯链路容量等工作环境有限的情况下，通过采用相应的编码技术，可以大大减少数据所占的存储空间，从而达到提高工作效率，或降低系统的工作成本的目的。

我们知道，大型应用系统中的数据库以及数字化图象和语音信号的数据量是非常大的。若不进行压缩处理，如此大量的数据很难在计算机及其网络上存储、处理和传输。可以这样说，没有数据压缩技术的进步，大数据量的存储和传输很难实现，多媒体计算技术也难以得到实际的应用。

数据压缩技术研究和处理的对象可以是数据的物理容积，如所占的空间；也可以是时间间隔，例如，传输某个指定数据集合所需的时间；还可以是传输指定数据集合所需的频带宽度。数据压缩的这三个对象是相互关联的：

$$\text{物理容积} = f(\text{时间} * \text{带宽})$$

在某一特定的应用环境中，一般说来不太可能同时对全部三个参数进行压缩，通常只有一项做为压缩的关键。不过，在这些年的研究中，人们对这几个参数的关注程度发生了一些变化。最初，大家对在通讯系统中减少传输模拟信号所需的带宽有极大的兴趣（现在该问题仍是人们关注的重点问题之一）。随后，在传真类系统中，提高传输速度也变得重要起来。近年来，在许多系统，特别是计算机与数字通讯系统中，数据所占存储空间的大小成为压缩的关键参数。这本书中，我们所要讨论的就是这种减少数据所占存储空间的压缩技术。

数据压缩技术的一般的处理框图如图 1-1。其中的原始数据（又称源数据）经过压缩处理，得到的输出即是被压缩的数据。当这些数据所占的存储空间和传输中所用时间的开销小于原始数据时，即实现了数据压缩。在需要使用这些数据时，只要经过还原（或称释放）处理

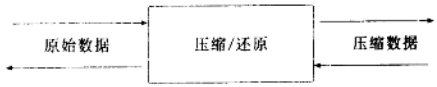

图1-1 压缩处理示意图

即可。这种数据压缩/还原模块可以在用户现有的硬件(如个人计算机、智能终端或其他的设备)上通过软件方式来实现,也可以使用综合了一种或多种压缩技术的专用硬件设备来实现。

数据压缩技术的应用很广泛,如文本压缩、图象压缩、数据库压缩、电视信号的压缩以及传输信号的压缩等。

文本压缩可以说是最基本的一种文件压缩,其目的是要减少文件所占的存储空间。这种压缩要求在压缩过程中不丢失信息,即要求压缩了的文件释放以后,应与压缩之前完全相同。关于文本压缩有许多简单且容易实现的算法,如空白压缩、位映射等。这些算法均是压缩掉数据流中的冗余信息,不会造成信息的损失。

用计算机处理语音数据时,首先要将连续的语音信号转化为数字量,然后才能进行相应的处理、传输、存储,或根据需要再转换为能被人收听到的模拟量重放。在把模拟量转换为数字量时,首先要进行采样,即每隔一固定的时间间隔测量输入信号的值,然后对这个数据按某种精度进行量化。在量化的过程中,量化值与实际值间的误差将会影响到把数字语音信号再重放为语音输出时的语音质量。另一个影响重放语音质量的因素是对连续信号进行采样时的采样速率。采样间隔越大,损失的信息越多,压缩比(衡量压缩效果的物理量)越大,但同时语音信号的失真也越大。在采样和量化过程中实际已对语音信号进行了压缩。即使这样,采样和量化后的数字语音数据仍占用相当大的存储空间。若不使用相应的方法进行数据压缩,用计算机处理语音的技术将很难得到推广和使用。可以利用去掉重复代码的方法对语音数据文件进行压缩,也可以根据语音数据的特点进行压缩。例如,语音信号中常常出现一些无声信号序列。对这种无声序列进行压缩,可以去除语音数据文件中的冗余,达到压缩的目的。不过,这种方法并非总是有效,它与所处理的语音文件的内容密切相关。显然,在无声序列很短的文件中,压缩比一定很小。在实际应用中,为达到提高系统压缩比的目的,经常把几种压缩方法综合使用。

在计算机中,数字图象信息是按象素来描述的。因此,在存储图象信息时,要占用大量的存储空间。在数据传输时,要占据很大的信道容量。例如,一幅 640 * 480 中分辨率的彩色图象(24bit/象素)的数据量约为 7.37Mbit/帧。对于运动图象,若以每秒 30 或 25 帧的速度播出时,要求视频信号传输速率约为 220Mbit/s。如果将这些信息存放于 600MB 的光盘中,只能播放 20 秒钟左右。可以这样说,大量数据存储的问题已成为计算机图象处理的瓶颈。数据压缩技术对解决这个问题起着非常重要的作用。对图象进行压缩编码的方法主要是利用图象固有的统计特性、视觉的心理学特性和显示设备的特性,从原始的数据中提取有效的信息,去除冗余信息,实现压缩编码。

近年来计算机越来越多的应用于非数值计算领域,如信息管理系统等。这些系统中数据库的规模也越来越大。采用数据压缩方法,可以降低存储的开销。另外在网络环境下,通过通讯线路进行数据库数据的存取所花费的时间可能会长得令人难以容忍。采用压缩技术可以减少数据的传输量,从而缩短在原通讯线路传输所用的时间,降低了系统的开销。

数据库压缩包括逻辑压缩和物理压缩。逻辑压缩是在设计数据库的结构时，通过对系统需求、所使用的数据及数据间的关系的分析，合理地确定数据类型和长度；或是通过代码的设计和使用，尽可能地减少数据的存储空间。物理压缩则是对具体的数据库文件进行压缩处理。它又包括对数据库结构的处理和压缩数据库中的实际数据等方面的工作。对数据库物理结构的压缩需针对不同数据库管理系统进行。对数据库中数据的压缩又包括字符数据与非字符数据的压缩。对前者，可以使用文本数据压缩的方法进行压缩，而后者可能是语音，图象，图形等不同类型的数据。对这些数据的压缩则要根据各自的特点采用相应的方法进行压缩。

1.2 数据压缩的分类

数据压缩技术有多种不同的分类方法。一种是按压缩技术所使用的方法进行分类，可分为预测编码(Predictive Coding)、变换编码(Transform Coding)和统计编码(Statistical Coding)三大类。这种分类方法是以不同的数学理论和方法为准则设计编码模型，进行压缩编码。

另一种是按压缩过程的可逆性进行分类。通常分为熵压缩(Entropy Compression)和冗余度压缩(Redundancy Reduction)两种。熵压缩是不可逆压缩。在熵压缩的过程中，会损失掉一部分信息(即损失了信息熵)。这样，在还原压缩文件时就无法做到无失真地再现被压缩的数据。因此这种压缩又称为有损压缩。它是以丢失部分信息为代价而获得较好的压缩效果。当然，为确保还原后的数据能基本保持原数据的特征，这种失真应被限制在某个规定的范围内。熵压缩主要用于图象和语音数据的压缩。冗余度压缩又称可逆压缩。冗余度压缩的工作机理是除去或尽量除去数据中重复和冗余的部分，而不丢失其中的任何信息，从而确保被压缩了的数据还原后与压缩前完全一致。因此可逆压缩又称为无失真编码(Noiseless Coding)。这种压缩方法主要用于文本、程序文件等不允许出现任何数据失真的场合。本书以数据压缩的应用为主要目的，因此采用后一种分类方法。

1.3 数据压缩术语简介

1.3.1 信息、熵和冗余度

1. 信息(Information)

我们经常听到、用到的"信息"这个词按日常理解的含义类似于"消息"。其实，消息与信息是完全不同的概念。消息是由符号、文字、数字或语音组成的表达一定含义的一个序列。一份电报，一句话，一段文字或报纸上刊登的新闻都是消息。在消息中，有一些内容是我们事先所不确知的。这些不确定的内容称为信息。对同一条消息，不同的人会有不同的感觉和反应，因为不同的人从这条消息中获得不同的信息。例如，将"太阳系有九大行星"这句话告诉一个成年人，他不会从中获得任何信息。因为这个内容是他确切知道的，不存在任何二意性。而对于不具备这种知识的儿童来说，他则会从中获得较多的信息，知道了一些他们原来所不知道的内容。消息是信息的载体，是表达信息的工具。信息是消息的内含，是消息中的不确定

性内容。如果这种不确定性被消除，那么其中的信息也就不存在了。很显然，昨天的新闻在今天就不再具有什么信息量了。

2. 熵 (Entropy)

既然信息是对不确定性的描述，那么就存在一个定量测量信息的问题。显然，消息中所含信息量的多少与不确定性的程度有关。在数学上，不确定性就是随机性，因此人们用概率论的方法来测度不确定性的大小。

一般说来，某件事出现的不确定性与其发生的概率有关。例如，“星期日我们可能去长城”。那么，星期日去长城的活动有两种可能，“去”或“不去”。二者出现的机会均等，即这两件事发生的概率各为 1/2。在这种情况下，不确定性不太大，只不过是“二者中择其一”。由此可以看到，消息中的不确定性的大小与可能发生的消息的数目以及各消息发生的概率有关。

我们将以上消息改为“如果星期日天气好，我们去长城”。此时，“去”与“不去”两件事发生的可能性不再是简单的各 1/2 的概率，要取决于天气的情况。假设是十月份的北京，天气以“晴”或“晴转多云”为主，其他则有“多云”，“多云转阴”，“阴”，“阴有小雪”等。一般，各种天气发生的概率是不相等的，“下小雪”的概率是极小的，“下大雪”的可能性更小。在听气象预报前，我们大概能猜出天气的情况，并可基本上确定有很大的可能性去长城。若预报为“晴”或“多云”，一般不会感觉意外，因与猜测基本一致，由此获得的信息也相应较少。如果预报有小雪，则会感到气候反常，从而获得较多的信息量。如果听到“阴有大雪”的预报，就会大吃一惊，因为完全出乎我们的意料，这时将会获得相当大的信息量。由此可知，某一事物状态出现的概率越小，其不确定性越大，一旦这种状态出现，将会获得很大的信息量。

在信息论中用“熵”来测量信息量的大小。对于单个事件(如一个字符)来说，其熵定义为：

$$H(i) = -\log2(Pi) \quad (bit)$$

该式表示发生的概率为 Pi 的事件(字符)i 所具有的信息量。度量信息量大小的单位是“位”(bit)。其物理涵义为表示该事件(字符)所需要的最少位数。

对于一个消息队列 X(X=x1,x2,...,xn)的平均信息熵定义为：

$$H(X) = -\sum_{i=1}^{n} P(ai) * \log(P(ai))$$

式中的 P(ai)表示某一事件 ai 发生的概率。由于这个表达式在形式上与物理学中热熵的表达式相似，因而借用“熵”这个词表示对信息量测度。有时为了区别于热力学熵，又称其为信息熵。一个事件发生的概率越小，其信息熵越高，所含的信息量越大。

3. 冗余度 (Redundancy)

早期的数据压缩之所以成为信息论的一部分，是因为它涉及冗余度的问题。消息中这些冗余信息将会产生额外的编码。如果去掉这些冗余信息，就会减少消息所占的空间。例如，“中华人民共和国”可以压缩成“中国”。这样，能大大节省存储和传输的开销。

当然，冗余信息在某些情况下是非常有用的。具有一定冗余度的消息能有较强的抗干扰能力。当消息在传输中受到干扰而出现错误时，这些冗余信息可以帮助我们根据上下文纠正错误。如果我们接收到“中华人 X 共 X 国”(其中的“X”表示由于干扰或其他原因造成丢失或误传的字符)，由上下文可找出丢掉的两个字。但若收到的是“X 国”，就很难判断丢掉的是什么字，不知到底是“中国”、“法国”还是“美国”等。

1.3.2 压缩、还原、压缩比

不论是文本文件、图形文件、数据文件还是程序文件，其中都有大量的重复部分。压缩(Compression)就是设法去掉部分或全部冗余，从而减少数据或文件所占的存储空间。还原(或称释放)则是利用相反的算法使数据或文件恢复原状。

衡量压缩程度的指标之一是压缩比。到目前为止，关于压缩比的定义尚无统一的方法。在信息论中，压缩比定义为压缩前后数据的熵之比。这种定义方法基于对要压缩数据的统计分析结果而提出。而现在所使用的许多压缩技术并不依赖于数据的统计结果。因此采用这种定义方法有相当的局限。为与实际的概念相一致，我们拟采用如下形式定义压缩比：

$$\text{压缩比} = \frac{\text{源代码长度} - \text{压缩后代码长度}}{\text{源代码长度}} \times 100\%$$

其含义是被压缩掉的数据占源数据的比例。例如，若压缩后的代码长度与源数据相同，则压缩比为0；若压缩后的代码长度为0，则达到100%的压缩比；若压缩后的代码长度是源数据的30%，则表示被压缩了70%，即其压缩比是70%。

数据压缩技术所探讨和研究的各种压缩方法，都是为了提高压缩比。不同的文件，不同的压缩方法，不同的图象分辨率都将影响到压缩比的大小。从数据压缩的目的来说，我们希望压缩比越大越好。但实际上，压缩比是有上限的。对基于统计的编码方法而言，这个上限与信息的熵有着密切的关系。如果压缩比超过了这个上限，还原时将无法恢复原状，出现失真。

1.3.3 编码模型

一般说来，数据压缩的工作是将符号流或数据流转换成相应的代码。显然，只有输出的代码流小于源符号流时，所做的数据压缩才是有效的。数据压缩的工作是在模型的基础上进行的。这里提到的模型是数据和规则的集合。规则用来处理输入符号，使之转换成相应的输出代码。构造数据压缩的模型和编码是两件不同的事情。但许多人常常把编码看作是数据压缩的全部内容。实际上，编码仅仅是数据压缩的一部分。例如，HUFFMAN码是一种常用的压缩代码。图1-2表示了用HUFFMAN编码法进行数据压缩的过程。在此过程中，模型的构造和编码是两个分离的部分。如果我们用汽车做比喻，那么，编码可以被看做是车轮，模型的构造则被看做是发动机。模型若不能提供合适的规则和数据，编码器的编码效果再好也无法完成预定的压缩工作。

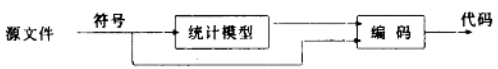

图1-2 Huffman编码过程

无损数据压缩通常使用两种不同类型的模型：统计模型和基于字典的模型。最简单的统计模型是读入字符，同时根据这个字符出现的概率进行编码。基于字典的压缩方法则采用完全不同的概念。首先，要为源文件建立数据字典。字典中列出了较长的字符串及对应于该串的代码。当然，代码一定短于字符串，否则，就不成其为数据压缩了。进行压缩处理时，由源文件读入数据，并在字典中寻找与之匹配的字符串。若能找到，则输出字符串所对应的代码

流。这部分输出就是被压缩了的数据或文件。源文件与字典相匹配的字符串越多，压缩的效果越好。

有损数据压缩较之无损数据压缩的一个基本不同在于它允许所处理的数据出现有限的数据损失。一般被用在存储数字化了的模拟数据，其主要应用于图形、图象和声音文件的处理。

有损数据压缩通常要通过两次处理：用信号处理的方法对其进行量化，这时将会发生数据的损失；然后用有关的压缩算法再进行压缩处理。

1.4 数据压缩技术的发展

关于数据压缩的理论研究或实践，有些学者认为始于十八世末 W. F. Sheppards 所做的"实数舍入为固定十进制数"(The rounding off real numbers to a fixed number of decimal places)的研究以及他的论文<关于按等距比例因子除法排列数据的频率常数最大概率值的计算>("ON the Calculation of the Most Probable Value of Frequency Constants for Data Arranged According to Equidistant Divisions of a Scale")；也有人认为十九世纪研制的莫尔斯代码是数据压缩的第一次尝试。1939 年，达德利(Dudley)发明了声码器(声音编码器)。它把声音的频谱能量划分为有限数目的频带，并且在每个频带内传输相应的能级，因此能提供较高的压缩比。不过，比较系统的研究始于四十年代初形成的信息论。早期信息论研究的内容之一是在已知消息中各符号出现的概率时，设法构造一种编码方法使消息所占空间尽可能少。尽管当时数字式计算机尚未出现，但所进行的研究与当今计算机中所使用的数据压缩技术有着密切的联系，许多算法在现在仍有很大的使用价值，如 HUFFMAN 编码、变换编码、预测编码等压缩技术和方法。到六十年代，已有许多压缩技术被应用于实际系统。近二十年，由于模式识别、图象处理、计算机通讯以及近年来倍受关注的多媒体等新技术的应用，促进了数据压缩在理论研究和实际应用方面的更大进展。

数据压缩的研究过程中一直有两个主要的方向。一个是许多数学家所进行的建立信源和数据压缩的数学模型，并从中找出其衡量数据压缩质量的技术指标及最优压缩性能指标；另一个则是更多的工程技术人员所进行的工作，他们的重点在于如何建立一个能实现数据压缩功能的系统，以服务于工程应用，或者对这些数据压缩系统进行分析或模拟，以确定它们的性能指标。但不论是理论研究还是工程实践，总的说来，1977 年以前，数据压缩主要是做为信息论研究中的一项内容，主要是有关信息熵、数据压缩比和各种编码方法的研究，即按某种算法对源数据流进行编码，使得经过编码的数据流较源数据流，占用少得多的空间。其中，基于符号频率统计的 HUFFMAN 编码方法具有良好的压缩效果，一直占有较为重要的位置。不断有以基本 HUFFMAN 编码方法为基础而派生出的 HUFFMAN 改进算法推出。

随着计算机技术的飞速发展，数据压缩做为解决海量信息存储和传输的支撑技术受到人们的极大重视，对数据压缩算法的研究也不仅局限于信息论中有关信源编码的范畴，数字图象信号、语音信号的分析、处理技术被大量引入到有关的研究领域。1977 年，两位以色列科学家 Jacob Ziv 和 Abraham Lempel 发表了论文"顺序数据压缩的通用算法"(A Universal Algorithm for Sequential Data Compression)，提出了一种不同于以往的基于字典的压缩算法

LZ77。1978年又推出了改进算法LZ78。他们的工作把数据压缩的研究推向一个全新的阶段。1984年，Terry Welch发表的论文"高性能数据压缩技术"(A Technique for High Performance Data Compression)描述了对LZ78算法的改进和具体实现技术，称为LZW算法。至此，压缩技术进入了实用化的阶段。目前，无损压缩领域中最为流行的压缩方法多是这种基于字典的压缩技术。

正是由于数据压缩技术的重大进展，近年计算机在语音处理、数字图象处理、数据存储以及数字通讯等方面的应用也取得令人瞩目的成就。

在数据压缩的研究中，语音压缩也是较早取得进展并应用于工程实践的领域。这项技术最初是为了解决通讯带宽等环境限制而提出的。六十年代初，AT&T公司开始研究用于电话通讯的数字音频数据编码和压缩问题，以解决传输质量和开销的问题。1962年，该公司建立了第一条商用数字电话线路。有关的研究对现在的计算机语音处理有很大的益处。但当时的研究仅限于一些特殊用途的设备，真正使用计算机处理语音是在近些年硬件设备开销大幅度降低的情况下才开始的。

在五十和六十年代，对图象压缩的研究限于静止图象的编码。主要方法有：预测编码法等。六十年代末陆续提出了若干变换编码方法。进入八十年代后，研究重点又由静止图象的编码转向活动图象的帧间编码，有关专家对此进行了大量工作，并取得了进展。国际标准化组织(ISO—International Standards Organization)和国际电报电话协商委员会(CCITT—Consultive Committee for International Telegraph and Telephone)共同组织了两个委员会：联合影象专家小组(JPEG—Joint Photographic Experts Group)和运动图象专家组(MPGE—Moving Picture Experts Group)，并推出了有关的压缩标准。JPEG标准是面向连续色调静止图象的压缩标准，其中定义了两种基本算法，一种是基于离散余弦变换(DCT—Discrete Cosine Transform)有失真的压缩算法，另一种是基于空间线性预测技术差分脉冲码调制(DPCM—Differential Pulse Code Modulation))的无失真压缩算法。MPEG标准是应用于多媒体环境的压缩标准，包括三个部分：MPEG视频、MPEG音频和MPEG系统。它所面向的问题是视频压缩、音频压缩及多种压缩数据流的复合问题。近几年，专家学者就小波变换(Wavelet Transformation)和分数维(fractal Coding)技术做了大量的工作，对图象数据处理和压缩的研究给予很大的推动。

适用于多种文件、多种环境的通用数据压缩软件是近十几年才开始得到广泛使用的。在Unix环境下，较早得到使用的多用途程序是COMPACT。COMPACT使用自适应HUFFMAN编码算法，能够产生比较好的压缩效果。但由于COMPACT的运行速度比较慢，因此它的使用受到很大的限制。在LZW算法提出不久后推出的基于字典的COMPRESS压缩软件不仅运行速度比COMPACT快得多，而且没有版权的限制，所有的Unix用户都可以很方便的使用和移植COMPRESS软件，因此受到了用户的欢迎。

八十年代初，CP/M和MS-DOS环境的用户广泛使用的是可与COMPACT相比拟的SQ软件。它采用静态HUFFMAN编码方法进行编码。1985年，以LZW算法为基础的ARC通用压缩程序出台，并很快替代了SQ。ARC具有文件压缩和归档两种功能。在此之前，用户必须先运行一个归档软件对文件进行归档，然后再进行压缩处理。ARC可一次运行完成这两项工作。现在，ARC已做了多次改版，仍是一个非常流行的商用软件。这几年，又相继出现了PKZIP、ARJ、LHarc等软件，它们均是基于字典的压缩方法。进入九十年代，个人计算机

上压缩软件的配备和使用更加广泛。1991 年上市的 DRDOS6.0 配备了美国 Addstor 公司的工具。1993 年 Microsoft 公司发表的 MS-DOS6.0 中，配备了“Double Space”数据压缩功能。Windows 的使用，使得对硬盘容量的需求和存取速度的要求大大增加，这从另一方面促进了数据压缩技术的推广。

尽管用软件方法可以较好地实现数据压缩的目的，但由于压缩算法的运算量较大，需要很高的运算速度和存储空间，这对现有系统来说是很大的负担。为了解决这个问题，人们在继续探索数据压缩技术的同时，着手研制生产高性能的芯片和系统。一般在对时间要求不高的应用场合，可以采用软件压缩，而在对运行速度有特殊要求的情况下，可使用硬件压缩。不过，目前硬件压缩的开销远大于软件压缩的开销。

在硬件设备中，最早是七十年代及八十年代初出现的数据集中器和统计多路转换器。由于在数据传输中，为节省通讯带宽而采用压缩数据的方法时，需要确保发送和接收端的压缩软件兼容。为解决这个问题，八十年代中期，制造商直接把数据压缩产品建立到网关的调制解调器上。此后，许多的调制解调器制造厂把各种压缩算法做到他们的产品中。

总之，数据压缩技术在计算机技术飞速发展的今天，有着很重要的作用。数据压缩技术除了可以减少系统开销外，还有另外的好处。当数据压缩被用于减少存储空间时，可以减少程序的总体执行时间。这是因为存储量的减少将导致磁盘存取次数的减少。虽然数据的压缩/还原会增加额外的程序指令，但由于程序的执行时间通常少于数据存取和转换的时间，故其总的执行时间将减少。当数据压缩被用于数据传输时，由于传输的字符较少，可以减少传输数据发生错误的概率；由于提高了效率，可减少工作量；由于传输的是压缩了的数据，而不是 ASCII 码，增加了系统的安全性。

在目前，数据压缩技术的发展还存在着一些需要注意的问题，其中主要的有：第一，数据压缩的标准化。现在用不同的压缩软件处理的文件具有不同的格式，因此不能用其他的软件进行释放。第二，压缩软件的专利权问题。现在，许多压缩软件申请了专利，有关公司对此的控制也很严格。对有关的问题应给予重视。

第二章　数据压缩技术基础

2.1　编码技术概述

编码实质上是对要处理的源数据按一定的规则进行变换。这里提到的变换规则就是从源数据到代码的处理模型，它可以是针对源数据符号的性质而选用的数学方法，也可以是针对某一种具体的数据格式而采用的变换策略。这些数学方法和变换策略的目的都是力图用尽可能少的符号或位来表示较多、较长的符号及信息。因此，根据这些方法和策略对源数据的编码处理，就是数据压缩的过程。鉴于此，在本书中我们将视编码与压缩为同一概念。

我们知道，数据压缩是对源数据集 D 进行编码，产生一个比 D 小的数据集 D′的过程，并且应能由 D′恢复为 D(在有损编码情况下，D′不能完全恢复为 D，但应尽可能使其接近 D)。在计算机中，这种编码就是用 0 和 1 构成代码序列表示源数据集的每个符号。为了提高编码和传输的效率，人们研究、设计并使用了许多种针对不同对象的编码方法，由此产生不同的代码。这些代码一般可分为固定长度码和可变长度码。固定长度码又称等长码，这种代码中所有的码符号个数都相同(如表 2-1 中的代码 1 及代码 3)，否则就是变长码或不等长码(如表 2-1 中的代码 2)。

表 2-1

源数据符号	代码 1	代码 2	代码 3
S1	00	0	001
S2	01	01	010
S3	10	001	011
S4	11	111	100
S5	00	101	101

不论何种类型的代码，若要达到无失真的编码和译码还原，必须保证源数据符号 Si 与码符号序列中的代码一一对应，并且码符号序列的反变换也是唯一的，即一种代码的任意一串有限长的码符号序列只能被唯一的译成所对应的源数据符号。显然，表 2-1 中的代码 1 不满足唯一可译的条件，因为 S1 与 S5 两个符号对应于相同的码符号 00。直观地看，代码 2 满足唯一可译的条件。但若接收到一个码序列 001111，则很难确定这串代码所对应的源数据符号是 S1、S2、S4 三个符号，还是 S3、S4 两个符号。因此，尽管代码 2 与源数据集中的字符满足一一对应的条件，但它的有限长序列不能满足唯一可译的条件，代码 2 仍然不是唯一可译码。

为了使代码唯一可译，源数据符号集中的符号种类越多，表示这些符号所需的码符号长度就越长。例如，在表 2-1 中可以看到，用长度为 2 的代码表示有五种符号的源数据时，无法使代码做到唯一可译。当然采用代码 3 可以得到唯一可译码，但代码的长度却大大增加了。过长的代码将会耗费大量的系统资源，编码效率较低。因此，从某种意义上讲，数据压缩是在

保证代码唯一可译性的前提下，用尽可能短的代码表示源数据序列。

2.2 HUFFMAN 编码

2.2.1 HUFFMAN 方法

HUFFMAN 编码是 1952 年由 HUFFMAN 提出的一种编码方法。这种编码方法的主导思想是根据源数据符号发生的概率进行编码。在源数据中出现概率越高的符号，相应的码长越短；出现概率越小的符号，其码长越长，从而达到用尽可能少的码符号表示源数据。理论研究表明，HUFFMAN 编码方法是接近压缩比上限的一种较好的编码方法。

我们通过一个实例说明 HUFFMAN 编码的步骤。

设有一源数据序列，包括 A、B、C、D、E 五个符号，它们出现的概率分别是 0.40、0.18、0.15、0.15 和 0.12。HUFFMAN 编码的过程可用编码树来表示。HUFFMAN 树的构造过程如下。

1)将源数据符号按概率递减的顺序排列；

2)取两个最小概率所对应的符号为叶结点，为这两个结点构造一个双亲结点，其概率为两个叶结点的概率之和，如图 2-1；

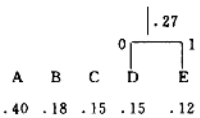

图2-1

3)把 D、E 结点的双亲结点按其概率的大小插入到源数据符号列表中。如图 2-2；

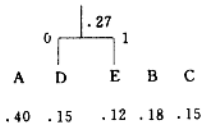

图2-2

4)重复步骤 2)、3)，直到全部结点被构造到 HUFFMAN 树中，即到达根结点为止；

5)设所有结点的左后代为 0，右后代为 1。从根开始经各中间结点到叶结点的路径代码即是该叶结点的 HUFFMAN 码。

按上述步骤构造的树为 HUFFMAN 树。整理后得到图 2-3。

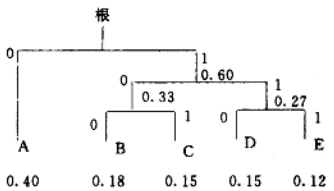

图2-3 HUFFMAN树

相应的源数据符号集的 HUFFMAN 代码表为：

A：0

B：100

C：101

D：110

E：111

符号 A 在源数据中出现的概率最大，其代码最短；其它出现概率较小的符号的代码相对较长。这样，将会使代码的平均码长尽可能短。

用 HUFFMAN 方法对数据进行编码的模型如第一章中的图 1-2。由这个模型产生的代码，可以根据它的 HUFFMAN 树，比较容易的完成译码——代码还原的操作。

2.2.2 编码算法流程

尽管 HUFFMAN 编码的效率很高，但在实际应用中，由于很难事先得知源数据中各符号发生的概率，因此无法保证按各符号发生的概率进行编码。人们最初用 HUFFMAN 方法进行编码时，往往根据一般文本中字符出现的统计规律得出的字符集概率进行编码。但在实际系统中，由于不同文件中字符出现的概率有较大的差异，编码效果不很理想。对此，最简单的方法是先扫描要编码的文件，计算出各字符出现的概率，然后再进行编码。我们在这一节中介绍的就是这种基本的 HUFFMAN 编码方法。

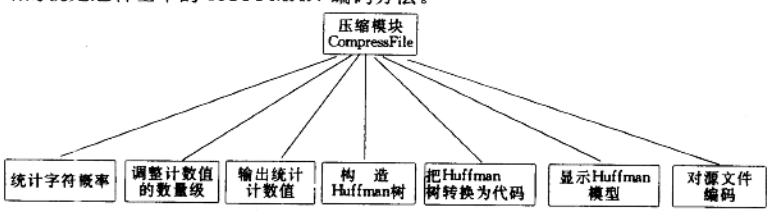

图2-4 基本Huffman压缩模块图

我们用程序模块图表示基本 HUFFMAN 编码方法如图 2-4。程序模块图是软件工程中用来描述程序功能模块划分和调用的工具。它并不指明各模块具体的输入、输出和执行方式。图中的矩形框表示的是功能模块，连线连接了相互间有关系的模块，相互关系不同，连接方式也不同。图 2-4 所示的模块图表示主模块 CompressFile 依次调用下层各模块。

在算法模块图中，首先统计源数据中各符号发生的概率，然后根据统计结果进行 HUFFMAN 编码。这样，可以有效的解决构造源数据概率模型的问题。但这种方法需要对源数据进行两次处理——统计计数和数据编码。一般情况下，这样做仅仅影响到压缩的速度。但在一些实际应用系统中，这种情况有时是不能容许的。例如，在数据实时传输的应用中，无法先对源数据进行统计，再进行编码压缩。若要解决这个问题，我们必须进一步修改算法模型，即采用自适应 HUFFMAN 编码模型。具体算法，将在第三章中讨论。这里介绍的是基本 HUFFMAN 编码方法。

2.2.3 编码算法实现中的一些问题

1. 统计计数

统计字符出现的频度是 HUFFMAN 编码的第一步，即需要统计出在源文件中各字符出现的次数。为保存统计的结果，我们建立一个长度为 256 个元素的长整型数组 counts，每个数组元素表示一个 ASCII 码的统计计数值。在程序中，这项工作由 void count_bytes(input,counts)函数完成。统计的结果是构造 HUFFMAN 树的依据。

建立长整型的数组是为了适应大文件或文件中出现高频度字符的情况。这势必会增加存储 HUFFMAN 树所必需的开销。若要减少这部分开销，就需要把 32 位的统计计数值变换为能用 unsigned char 类型表达和存储的数据，即进行数据的数量级调整。函数 void scale_counts(counts,nodes)先找出所有符号的统计值中的最大数，再用由此得到的缩放因子调整所有的统计计数值，使得都能存放在八个字节里。运算结果做为初始频度统计值存入结构数组 NODE 中。这样做有利于提高程序的运行速度，减少存储 NODE 数组所占的空间。同时也可以限制最终生成的 HUFFMAN 码的长度。在这项工作中，需要特别注意的是所有非零的统计计数值不能被缩放因子处理为 0 。因为凡是发生频度为 0 的字符均不参与 HUFFMAN 编码。若把一些字符的频度调整为 0 后，就会影响编码的正确性。

2. HUFFMAN 树的构造和存储问题

不论是压缩还是释放操作，HUFFMAN 树的构造都是必不可少的。程序中，HUFFMAN 树的构造是由函数 build_tree() 完成的。程序中定义的全程结构变量如下所示：

```
typedef struct tree_node{
    unsigned int count ;
    unsigned int saved_count ;
    int child_0;
    int child_1;
}NODE;
```

该结构包括四个成员变量：count 表示该结点的权重；child_0、child_1 是该结点的两个后代；saved_count 则是为了便于程序的调试而设的一个变量，它用来保存每个符号的初始计数值，因为，成员变量 count 中所保存的权重在 HUFFMAN 树的构造过程中会发生变化。

HUFFMAN 树的构造是一个循环的过程。当结点的计数为非零值时，表明该结点为活动结点。在所有的活动结点中找出两个最小权重的结点。将这两个结点的权重求和，形成一个新的内部结点，并用变量 next_free 指出。它的两个后代则指向前面找到的两个最小结点。这时，再把这两个后代结点的权重设为零，表明这两个结点为非活动结点，不再参加后续的构造 HUFFMAN 树的操作。如果只有一个活动结点，则该结点为根结点，表明树的构造过程结束。

函数中所设的第 513 号结点用做标志结点。对 ASCII 字符集中 256 个字符构造的 HUFFMAN 树而言，任何一个有效活动结点的序号都不会超过这个值。这是由二叉树的性质和特点所决定的。

HUFFMAN 树在 HUFFMAN 编码中起着非常重要的作用。若要保证准确无误地释放被压缩的文件，就必须保证译码时所使用的 HUFFMAN 树与编码时建立的树完全相同。这棵 HUFFMAN 树不仅是编码的重要依据，也是正确译码的基础。因此，有关这棵树的信息要作为编码后产生的压缩文件的附加信息同时保存。有两种方法可以实现对有关信息的保存。一种方法是用结构数组保存 HUFFMAN 树。一般说来，HUFFMAN 树的每个结点都

应包括该结点的字符、该字符在源数据流中出现的频度(即权重)、双亲结点、左后代分支和右后代分支等五个数据项。考虑到译码的需要，若译码是从根结点开始，则在存储这棵树时，只需保存左分支、右分支和数据项三个部分。由二叉树的定义可知，树的结点总数是叶结点数的二倍。那么，有 256 叶结点(字符)的最大 HUFFMAN 树的全部结点数是 256 * 2=512。按我们对这个数据结构的定义，即使只存储左分支、右分支和数据项，每个结点也要占 5 个字节。这样，为节省空间进行编码而产生的文件却要附带一个 512 * 5=2304 字节的 HUFFMAN 树。在源文件不太大的情况下，这棵相对庞大的 HUFFMAN 树将会抵消用 HUFFMAN 方法进行数据压缩的效果。因此，必须尽量减少 HUFFMAN 树的存储开销。如果考虑到在 HUFFMAN 树中，叶结点的左分支和右分支的两个数据项永远是 0，因此叶结点的数据项不必存储。再又虑及到所有的字符都在叶结点上，所以全部的非叶结点的数据项在译码中是不起作用的。那么，对于最大的 HUFFMAN 树必须要保存的内容所占空间为 1280 字节。释放压缩文件时，先读出 HUFFMAN 树，然后再按有关内容对压缩文件进行译码。

另一种方法是不直接保存编码用的 HUFFMAN 树，而只保存对字符进行统计计数的结果。由于编码时所用的 HUFFMAN 树是根据这些统计结果得到的，因此只要有了这个字符统计计数结果就可以构造出与编码时所使用的模型完全相同的 HUFFMAN 树。根据这种思路，在压缩文件中，只需保存各字符的统计计数结果。当对压缩文件进行释放操作时，先读出字符的统计结果，再据此构造出 HUFFMAN 树，最后完成其他的译码操作。由于已对统计计数的结果进行了数量级调整，因此，计数数组将只占 256 个字节。这比保存完整的 HUFFMAN 树的存储开销要小得多。我们选择这后一种方法来存储用于构造 HUFFMAN 树的信息。

尽管附加信息已被减少到 256 字节，但我们经过分析会发现还可再节省一些存储开销。对于许多 ASCII 的文本文件来说，一般最多用到 128 个可显示字符。实际上，甚至常常使用其中的不到 100 个字符。文件没有出现过的字符，完全没有必要存储相应的计数值 0。因为，它即不参加编码，更不参加译码。这样，只需存储出现在文件中的字符的计数值即可。尽管采用这种方法进行 HUFFMAN 编码，不论是压缩还是释放都需要构造 HUFFMAN 树。但所换取的是使为构造 HUFFMAN 树而附加的信息占据尽可能少的存储空间。对于数据压缩来说，这是非常重要的。程序中，这项工作由函数 void output_counts(output,nodes)完成。

译码时，用函数 void input_courts(input,nodes) 从压缩的文件中读取字符的统计计数，提供构造 HUFFMAN 树之用。

用 HUFFMAN 方法进行数据压缩时，不论是编码压缩还是译码释放都要构造 HUFFMAN 树。在程序中，这部分工作是通过对 int build_tree(nodes) 函数的调用来完成的。

3. HUFFMAN 树的使用

在 HUFFMAN 编码算法中，不论是压缩还是释放操作，都要使用 HUFFMAN 树。

使用 HUFFMAN 树进行压缩时，需要先构造出每个源符号对应的代码。这项工作由函数 convert_tree_to_code() 完成。针对我们所使用树的数据结构 NODES，采用递归算法能较方便的得到所需的代码表。设树根处为 0，由此开始遍历整个 HUFFMAN 树。每遍历一个树枝，在代码中相应的增加一个 1 或 0。到达叶结点后，将该叶结点的代码值存入代码数组，同时返回到上一个结点，继续遍历 HUFFMAN 树，直到完成全部代码表的构造。有了这张

代码表，就可以很方便的把源文件转换为用 HUFFMAN 编码表示的压缩文件。把用 HUFFMAN 码表示的文件还原为 ASCII 码的文件的过程是译码或释放的过程。其主要工作是先从压缩的代码文件中读出各字符的统计计数，再构造出 HUFFMAN 树，得到 HUFFMAN 码，最后完成 HUFFMAN 码到 ASCII 码的转换。其程序模块图如图 2-5。

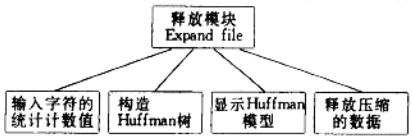

图2-5 释放算法模块图

释放压缩数据的操作由 expend_node() 函数实现。函数的流程比较简单。具体过程是：从根结点开始译码过程。每次从压缩文件中读出一"位"。若该位的值是"0"，则下一个结点由结构中的成员变量 child_0 指明。若为"1"，则下一个结点由 child_1 指明。如果结点号小于或等于 256，为叶结点，即是压缩代码所对应的源字符。

4. 位的读写操作

我们知道，计算机中的数据是按字节存放的，而 HUFFMAN 方法却是用位(而不是字节)表示字符。我们所熟悉的 C 语言的标准 I/O 函数库 stdio. h 只提供了字节或字节块的操作。这就给 HUFFMAN 方法的实现带来了一些困难。为了解决这个问题，我们建立一个程序 bitio. c。bitio. c 中定义了若干个按位进行读写操作的函数例程，包括对文件的打开、关闭、按位读入和输出、多位的读入和输出等，并将其放到 bitio. h 头文件中。在使用时，只需把 bitio. h 包括到程序中即可。bitio. h 清单如下所示。

```
/* * * * * * * * * * * * * * * * * Start of BITIO. H * * * * * * * * * * * * * * * * */
#ifndef BITIO_H
#define BITIO_H

#include <stdio. h>

typedef struct bit_file {
    FILE * file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
    }BIT_FILE;

BIT_FILE * OpenInputBitFile(char * name);
BIT_FILE * OpenOutputBitFile(char * name);
void OutputBit(BIT_FILE * bit_file,int bit);
void OutputBits(BIT_FILE * bit_file,unsigned long code, int count);
int InputBit(BIT_FILE * bit_file);
unsigned long InputBits(BIT_FILE * bit,int bit_count);
```

```
void CloseInputBitFile(BIT_FILE * bit_file);
void CloseOutputBitFile(BIT_FILE * bit_file);
void FilePrintBinary(FILE * file, unsigned int code,int bits);

#endif /* _BITIO_H */
/* * * * * * * * * * * * * * * * End of BITIO.H * * * * * * * * * * * * * * * */
```

为完成各项按位操作，在 bitio.c 中定义了一个结构：

```
typedef struct bit_file {
    FILE            * file;
    unsigned char     mask;
    int               rack;
    int               pacifier_counter;
} BIT_FILE;
```

结构 bit_file 中第一个成员变量是文件指针“* file”，用以保存对函数 OpenInputBitFile() 或 OpenOutputBitFile() 调用后返回的 FILE 结构指针。结构中的 mask" 和 “rack”是为 BIT_FILE 的面向位的处理而设立的两个变量。rack 包含了从文件读入或要写到文件中的数据的当前字节。该字节中各位的读入是按由高位到低位的顺序进行的。“mask”则是用于设置/清除当前的输出位，或屏蔽当前的某些输入位。当第一次打开 BIT_FILE 时，mask 元素的初值是 0x80。输入数据时，从文件中读入一个单字节放到 rack 中，并和屏蔽字 mask 进行与运算，即可得到输入位的信息。然后，mask 右移一位，继续重复以上操作，直到 rack 中所有的位都被读入，完成了对一个字节的按位输入。重新设置 mask 的值后，再读入下一字节，直到文件结束。bitio.c 中提供的一次读入多个位的函数例程也按类似的方法进行处理。BIT_FILE 结构中定义的 pocifier_counter 用于检查压缩编码工作进行的情况。它的计数值随着读入一个新字节或把一个字节写到文件中而增加。每当达到 2048 字节时，向标准输出设备输出一个字符，如本程序中用的是“.”。这样，可以给用户以确切地提示，表明编码工作正在进行。bitio.c 程序清单如下。

```
/* * * * * * * * * * * * * * * Start of BITIO.C * * * * * * * * * * * * * * * * * */
/* 本程序包括的位运算例程应连接到有关程序中。                                    */

#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"
#include "errhand.h"

#define PACIFIER_COUNT 2047
```

```
BIT_FILE * OpenOutputBitFile( name )
char * name;
{
    BIT_FILE * bit_file;

    bit_file = (BIT_FILE * ) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE * OpenInputBitFile( name )
char * name;
{
    BIT_FILE * bit_file;

    bit_file = (BIT_FILE * ) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
      return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE * bit_file;
{
    if ( bit_file->mask ! = 0x80 )
      if ( putc( bit_file->rack, bit_file->file ) ! = bit_file->rack )
        fatal_error( "Fatal error in CloseBitFile! \n" );
    fclose( bit_file->file );
    free( (char * ) bit_file );
```

```
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
{
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void OutputBit( bit_file, bit )
BIT_FILE *bit_file;
int bit;
{
    if ( bit )
      bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
      if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
        fatal_error( "Fatal error in OutputBit! \n" );
    else
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
          putc( '.', stdout );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
   }
}

void OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;

    mask = 1L << ( count - 1 );
    while ( mask != 0) {
      if ( mask & code )
        bit_file->rack |= bit_file->mask;
    bit_file ->mask >>= 1;
```