**数据运算**

数据上的运算可以分为三类：逻辑运算、移位运算和算术运算。

逻辑运算

逻辑运算就是所谓的与AND或OR非NOT异或NOR, 当然我们不排除还有其他可能的运算形式要求更多的操作数个数。

如果（单个）操作数只是一个二进制位（一般是从模式中独立提取出来的），则这种逻辑运算就是**位层次上**的。我们很容易定义位层次上的逻辑运算规则，甚至画出它们的**真值表**[[1]](#footnote-1)（没有给出NOT的真值表）：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | y | x and y |  | x | y | x or y |  | x | y | x and y |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 |  | 0 | 1 | 1 |  | 0 | 1 | 1 |
| 1 | 0 | 1 |  | 1 | 0 | 1 |  | 1 | 0 | 1 |
| 1 | 1 | 1 |  | 1 | 1 | 1 |  | 1 | 1 | 0 |

AND, OR, NOR都有一些奇妙的特性，这些特性在模式层次上有着重要应用：

对于AND而言，只要其中一个操作数是0, 则结果也必然是0（而不必考虑另外一个数究竟是什么）；对于OR而言，只要其中一个操作数是1, 则结果也必然是1; 对于XOR要稍微复杂一点：如果其中一个操作数是1, 则结果与另一个操作数的值相反。

基于为模式的逻辑运算，我们可以进一步规定模式层次上的逻辑运算，很简单，就是将每个位（单操作数）或对应位（多操作数）进行位层次运算即可。

利用上面提到的运算特性，模式层次上的逻辑运算有许多重要应用：

1. 对于AND, 0使得对应位被置0（称为**复位**）, 1使得对应位保持原来的数不变，因此与运算可以使指定的位复位。举个例子，如果要复位一个八位二进制数x的前五位，后三位保持不变，则我们可以利用x AND 00000111运算完成这一点，这时我们构造出的后操作数被称为**掩码**。

2. 对于OR, 0使得对应位保持不变, 1使得对应位变成1（称为**置位**）, 因此或运算可以使指定的位置位。举个例子，如果要置位一个八位二进制数x的前五位，后三位保持不变，则我们可以用x AND 11111000完成这一点，这个二进制数仍然被称为掩码。

3. 最后，对于XOR, 0使得对应位保持不变, 1使得对应位反转，因此异或运算可以反转指定的位，不再举例，第二个操作数仍然称为掩码。因此从掩码这个名称我们是无法判定究竟进行的是复位操作、置位操作还是反转操作的。

移位运算

移位运算就是以某种方式移动模式中的位。根据移位运算的需求可以分为逻辑移位运算和算术移位运算两类，逻辑移位运算仅仅只是描述了一系列的操作，而算术移位运算则基于具体的计算需求来考虑移位方法。事实上可以认为算术移位建立在逻辑移位的基础上。

逻辑移位运算被用于处理**不带符号位的整数**。

（普通）逻辑移位运算：逻辑右移运算将模式中的每个位向右移动一个位置，最右位被丢弃，最左位填0; 逻辑左移运算算法类似。

循环逻辑移位运算（或称为旋转运算）则将最右位（或最左位）填补到新数的最左位（最右位），使得移动形成一个回环。

算术移位运算被用于处理以**二进制补码格式**表示的带符号整数。算术右移运算的操作规定如下：保留符号位（即最左位），同时将其复制放入右边相邻的位中，因此符号被保留下来。算术右移运算被用于将整数除以2（并取整）。算术左移运算则丢弃符号位并在最右补0（实际上和普通逻辑左移运算完全等效），用于将整数乘2.

算术运算

由于复杂度的原因，这里只讨论整数的加法和减法。

1. 二进制补码整数的加减法

加法操作非常简单：将两个数对应的二进制位相加，服从一般的进位原理，如果最高位出现进位则丢弃它（请注意当两个数都是正数时这一举动会导致**上溢**）。

由于二进制补码整数的相反数即其补码，因此我们有

式中为的反码，即的补码。由上式可知减法可以很容易地转化为加法。

2. 符号加绝对值整数的加减法

**数据储存**

**数据类型**

**储存整数**

1. 概述

整数储存主要有**无符号表示法**（储存无符号整数）、**符号加绝对值表示法**和**二进制补码表示法**（储存带符号整数）三种类型。以下分别叙述三种方法储存的格式。

2. 无符号表示法

将十进制数转为二进制，位数不够的在前面填0. 用n位可储存的数值范围是

3. 符号加绝对值表示法

与无符号表示法类似，但最高位被用于表征数的符号，正数为0负数为1，注意到该表示法中存在+0和-0的区分。用n位可储存的数值范围是

4. 二进制补码表示法

正数与符号加绝对值表示法类似。对于某一数的相反数，取其补码表示，以产生相应的负数。并规定(除0外)补码与自身相同的1000....0(n-1个0)用于表示(而不是,这是为了统一正负数的判定法则，见下面第三段)

补码运算的规则说明如下：从二进制数的最右边开始复制，直到第一个1出现，以后的各数（不包括这个1）取其反码。例如00110100的补码为11001100.

根据以上规则我们知道对于二进制补码表示法仍然可以通过最高位来判定数的正负：0为正数1为负数。

该表示法中不存在+0和-0的区别，这是由于对0的二进制编码补码与自身相同。用n位可储存的数值范围是, 注意到相对于符号加绝对值表示法能够多表示一个负数，这是节约下来的-0所贡献的一位.

这样规定整数编码能够使得运算变得简单，因此在实践中被广泛采用。

5. 溢出

**储存实数：浮点法**

1. 为什么要用浮点法储存实数

考虑储存实数的基本思路就是将其转化为整数，一个自然的想法是将数据从小数点截断，分别将整数部分和小数部分作为两个整数来储存，称为**定点法**（小数点的位置固定不动）。这样做虽然可行，但却减少了能够表示的数的范围。假设规定小数点左右各8个数码，考虑这样两种情景：（以下用十进制举例，对二进制也是一样的道理）

112233445566.77由于左边位数不足，会被截成33445566.77

11.223344556677由于右边位数不足，会被截成11.22334455(或11.22334456)

对于实数采用**浮点表示法**（实质是**科学记数法**），能够用更小的空间来表示整数部分很短而小数部分很长，或整数部分很长而小数部分很短的这样两类数。（否则我们就不得不为整数和小数部分分别规定更大的长度）下一节所述的规则阐明了浮点表示法是如何起到作用的。

2. 浮点表示法的一般规则

首先仍然需要将十进制数转化为二进制，然后采用科学技术法来表示这个数（称为**规范化**）：

其中小数点左边显然只能是1, 那么实际上1和小数点都可以不保存，需要保存的只有**符号**、小数点后的数xxxxx（称为**定点数**）和指数n（称为**位移量**），因此在浮点表示法中实数（按在内存中储存的顺序）由符号、位移数、定点数三个部分来构成。一般而言符号占1位(0正1负)，位移数位数<定点数位数。

其中定点数可以通过无符号表示法储存，位移量由于带有符号，**理论上**可以通过二进制补码表示法储存，但实践中采用的是**余码表示法**。该表示法解决符号问题的方案是，在位移量上加一个正数（称为**偏移量bias**），从而将位移量的范围向右平移，使得所有位移量（在表示上）**都是非负的**。偏移量为m的系统常常又可以称为**余m码系统**。在位移量占n位的标准中，最终位移量的取值范围是（正如无符号表示法），偏移量m规定为，则原始位移量的取值范围是. [[2]](#footnote-2)

3. IEEE 754: **单精度Single**(32bit,4Byte)和**双精度Double**(64bit,8Byte)

在单精度中符号占1位，位移量占8位，定点数占23位。则可知偏移量 因此单精度又称为余127码系统。

在双精度中符号占1位，位移量占11位，定点数占52位。则可知偏移量 因此单精度又称为余1023码系统。

=====================================Example===================================

例如将5.75转换为单精度数，先换成二进制数101.11，规范化为

符号+, 表示为0

位移量2, 加上偏移量127后为129, 转化为二进制为10000001, 满8位不用补0

定点数0111, 在后面补0至23位为01110000000000000000000

因此单精度的5.75储存为01000000101110000000000000000000

=====================================Example===================================

【一些特殊规定】

（以下均以单精度举例，双精度是类似的）

注意到由于二进制科学记数法中小数点和小数点左边的1是默认的，因此在不加特殊规定的情况下是无法表示0的，为了避免这一点，IEEE将**偏移后的指数为0**（所对应的原始指数[[3]](#footnote-3)为**-127**）且**定点数部分为0**的数规定为表示0. 由于符号位可以是0或1，因此浮点数表示法中**存在+0和-0的区分**.

相应的，IEEE标准将偏移后的指数为255（所对应的原始指数为128）且定点数部分为0的数规定为表示∞. 依据符号位的不同则可区分为+∞和-∞.

由于-127和128都已经被用掉，因此在IEEE标准中原始指数的范围被缩小为

【单精度上下限推算】

由此经过简单推算可知，单精度型所能够表示的最大正数为

解释一下：上式中的是为了凑出定点数部分所能表示的最大数值1.1111…1(小数点后23个1)

所能够表示的最小正数为

负数部分同理。

【双精度上下限推算】

双精度型所能够表示的最大正数为

所能够表示的最小正数为

4. 溢出

当给浮点数赋值的绝对值超过最大值时，就会发生**上溢出**；绝对值小于最小值（不是0）时，就会发生**下溢出**。

**储存文本**

1. 代码

计算机通过**位模式**来储存文本。位的不同组合（当然实际上可以理解成不同的二进制数）表示不同的字符。位模式的集合称为**代码**。

容易证明，n个位可以表示个字符，表示m个字符所需要的位数n可以由公式

计算得到。例如要表示英文26个字母，理论上需要个位来编码所有不同的字符。

2. ASCII(已经被Unicode兼容为**基本Latin**)

美国信息标准交换码(ASCII)用7位表示字符，共定义了128个字符，其大致编码如下：（以十六进制叙述）

00~1F: 控制字符，大部分用在过时的通信协议中

20: Space

21~2F:符号

30~39:0~9 这说明一个数字字符对应的ASCII值是它本身加上

3A~40:符号

41~5A:大写英文字母

5B~60:符号

61~7A:小写英文字母

7B~7E:符号

7F: DEL

对照大小写字母的编码可知，将一个大写字母转化成小写只需加

3. Unicode

统一字符编码(Unicode)起初用2字节(16位)表示字符，在Unicode第5版以后则在前面又添加了2个字节，用4字节(32位)表示字符，所能定义的字符个数为

Unicode花费前2字节将编码空间划分为**平面**(**plane**)，目前的编码实际上只用到了其中极少的平面，大多数平面是闲置的。以下叙述已经定义的平面：

0000**基本多语言平面BMP**：用于与之前的16位版本兼容，编码在显示时通常省略掉高位的四个零，只写出后四位。该平面用于定义不同语言中的字符集。

BMP中的前128个字符(编码为0000 0000~0000 007F)用于兼容ASCII(基本Latin)

0001**辅助多语言平面SMP**：为没有被包含在BMP中的多语言字符提供编码。

0002辅助象形文字平面SIP：为象形文字提供编码。

000E辅助特殊平面SSP：表示不在基本Latin和Latin-1中的特殊字符。

000F和001O私有用户平面PUP：为私有用户预留。

因此0003~000D和0010~FFFF平面都是被保留的，足见其剩余空间之多。

**储存音频和图像**

与数字、文本的离散性不同，音频和图像本质上是机械振动模式（即响度和频率）关于时间的连续函数，以及电磁振动模式（即亮度和频率，或者说是颜色）关于空间的连续函数，由于它们是连续的，因此不可能把稠密的实数轴上的每一个点的具体值都记录下来，那将要求无限的内存空间，为了应对这一点，我们实际上有两种可用策略：一是**以离散模拟连续**，只要样本取的足够细就可以近似反应原来的连续情况；二是强行求出函数的方程并通过记录方程的形式来记录原始信号。对于图像，我们同时采用了两种方法，采用前者的图像储存技术称为**位图**（光栅图；栅格图）技术，而采用后者的图像储存技术称为**矢量图**技术；但对于音频，目前主流的做法只是前者，即取样法。

1. 取样法

先讨论取样法。取样法对于原始信号的逼近程度取决于两方面：单位时间(1s)或单位空间(1英寸)内的样本数、以及储存每个样本所用的空间，由于人的感官的感受精度有限，这二者并不需要不加限制地无限提高，到了后期由于感官的限制再提高精度将变得没有意义。

对于音频，每个样本所分配的位数被称为**每样本位**（**位深度**），目前常见的位深度为16,24,32.则每秒钟的音频需要的位数（定义为**位率**）

目前的音频编码主流标准是**MP3标准**, 采用每秒44100个样本、每样本位16, 并且对人耳无法分辨的信息进行**有损压缩**。

对于位图有也一些特殊称呼（虽然并没有什么卵用但是蛮认识一下）：我们把图像的采样称为**扫描**，每一个样本称为**像素**，单位空间内的样本数称为**解析度**，每个像素所分配的位数称为**色彩深度**。

目前对于色彩深度有两种编码方式，一种是真彩色模式(RGB)，采用24位进行编码，每8位代表R,G,B中的一种成分的浓度，真彩色模式可以表达种颜色；另一种是索引色（调色板色）模式，采用8位进行编码，因此只能够表达种颜色。

目前有许多种图像编码标准，JPEG（联合图像专家组）使用真彩色模式，但对图像进行有损压缩以减少位数，GIF（图形交换格式）采用索引色模式。

2. 模拟法（方程法）

上面已经介绍了模拟法的思想，现在直接解释其相对于取样法的优劣。对于位图而言，放大图像意味着扩大像素，那将使图像变得更加粗糙，而矢量图则保存图像的几何本质，当我们缩放矢量图时，图像中的点将被重新计算，因此其图像质量永远不会减损；但问题在于日常图像难以一律被方程所模拟，因此事实上矢量图一般只用在计算机制图中（例如CAD工程绘图）。

**储存视频**

视频就是一系列的图像随时间变化，因此储存图像的技术即可储存视频。不过通常视频一般都针对其特点进行了一些压缩，而不再是简单的一个接一个的图像数据。

**数据结构**

 数据结构是相互之间存在**一种**或**多种**特定关系的数据元素的**集合**，这些元素称为**数据**，而元素之间的关系则称为**结构**。这个定义表明，数据本身是没有所谓结构的，是由于它们之间存在着某种（或多种）的关系才有了结构。

通常可以将数据结构分为集合、线性结构、树形结构和图四种类型。

概括地说，集合中的元素除了“同属于一个集合”以外就没有任何关系[[4]](#footnote-4)；线性结构是一种“一对一”的关系，树形结构是一种“一对多”的关系，图则是一种“多对多”的关系。关于这些概念的精确定义在以后还会再提到，这里只需要形象地理解一下即可。

我们定义线性结构是这样的关系集合：

i. 存在唯一一个被称为“第一个”的元素；

ii. 存在唯一一个被称为“最后一个”的元素；

iii. 除了第一个元素外，每一个元素均有且只有一个前驱；

iv. 除了最后一个元素外，每一个元素均有且只有一个后继.

**数据压缩**

**数据压缩方法**

**无损压缩方法**

1. 游程长度编码

该算法的核心思想是将数据中连续出现的符号用一个字符和这个字符重复的次数来代替，重复次数的编码一般需要统一位数。如下例：



游程长度编码在只由两种符号表示（通常是0和1），并且一种符号比另一种符号使用更为频繁时十分有效。如当1很少0很多时，我们只标记两个1之间0的个数，如下例：



这里需要注意的是当0的个数超出的单个压缩单位时（这里是4位）就需要用8位来编码，例如25个0编码为1111 1010，为了消歧义（即把这段编码理解成是15个0+1+10个0），规定读取到1111则默认后面仍是连续的0，作出这一妥协以后，对于15个连续的0就被迫用1111 0000来编码）

2. 霍夫曼编码

注意：霍夫曼编码的操作是为了给出现频率高的字符较短的编码，频率低的字符较长的编码，以尽量地减少压缩后的文件长度。

 ①首先统计每个字符的使用频率。

②然后我们为它构造一个二叉树，自下而上，所结合的两个节点的权值之和必须是最小的：



③构造好了二叉树以后就可以对字符分配编码（注意到尽管我们最后的目的是给叶编码，但操作的对象是分支）：从根结点开始，左分支分配0，右分支分配1[[5]](#footnote-5)，循环这一过程迭代到叶。



注意到这种编码方式可以使得任何编码都不会成为另外一个编码的前缀[[6]](#footnote-6)，因此无论在编码和译码（压缩和解压）过程中不会产生二义性，这样在译码的过程中，任何字符在最小的位数下即可被解压出来，因此霍夫曼编码是一种即时码。

3. Lempel Ziv编码

**有损压缩方法**

1. 联合图像专家组JPEG（以灰度图像为例）

JPEG的步骤：分块→离散余弦变换DCT→量化→无损压缩

①将图像分为许多8×8的像素块，以减少计算量。

②对每个8×8像素块进行DCT，

2. 运动图像专家组MPEG（以MPEG-1为例）

视频的本质是帧的序列，对视频进行压缩包括空间压缩和时间压缩两个方面。空间压缩是对每一帧的独立压缩（使用压缩静态图像的JPEG方法），而时间压缩基于这样一个事实：许多连续的帧是几乎一样的，因此我们可以根据前后帧与本帧的差别来获得本帧的数据。

MPEG方法将帧分为三类：I-帧、P-帧、B-帧。

I-帧（内部编码帧）是独立的帧，不依赖于其他任何帧的数据，周期性出现。

P-帧（预帧）依赖于前面的I-帧或P-帧。

B-帧（双向帧）依赖于前后的I-帧或P-帧，注意B-帧不与B-帧发生联系。

1. 所谓的真值表是一种遍历运算的所有输入及其输出所作出的表格，它反映了这一运算的全貌。当两种运算的真值表完全相同时，我们就可以知道这两种运算等效，这是真值表的一个重要应用之一（如果它还有其他应用的话）。 [↑](#footnote-ref-1)
2. 注意到规定的后果就是浮点表示法中的原始指数取值范围和二进制补码表示法的范围在正负数上下限的处理上不统一（而是完全相反了）. 这一规定的理由不得而知. [↑](#footnote-ref-2)
3. 但是实际上这样的原始指数是被排除在外的。 [↑](#footnote-ref-3)
4. 这和数学中的集合定义是一致的。 [↑](#footnote-ref-4)
5. 这里左1右0也并无不妥。 [↑](#footnote-ref-5)
6. 这很好理解：毕竟它们存在于不同的分支上，前缀必然是不同的。 [↑](#footnote-ref-6)