第3章　信息表示

从前面的介绍中可知：计算机元器件的基本物理特性使得它们容易产生两种明显不同的状态(电路的开与关、磁极的南与北、材料表面的凸与凹等)，通常用0和1抽象地表示它们。那么，世界上纷繁多样的信息是怎样在计算机中用0和1表示出来的呢？本章就介绍与此有关的知识。

计算机中只能存储0和1，如果要用它们表示数值，则需要采用二进制。当然，计算机中表示的信息不一定都是数值。在表示非数值信息时，所谓“进制”的概念是没有意义的，但人们习惯上还是笼统地说“计算机是用二进制表示信息的”。那么，到底什么是二进制呢？它和常用的十进制之间有什么关系呢？针对这些问题，本章将首先介绍进位制的有关内容。在弄清楚进位制的问题之后，再说明计算机中是怎样用0和1表示信息的。本章主要介绍数字和各种字符的表示，其他诸如多媒体信息的表示比较复杂，本章不做介绍。本章最后介绍指令和程序的表示。

3.1　进位制及其转换

采用进位制表示数是人类的一个重要的发明。我们在学算术时都知道一个道理：“满十进一”，即数数的时候数到10就进一位。这种记数方法称为十进制。我们把记数进位的规则称为进位制。习惯上用进位制规定的进位的值命名记数方法。例如，“满二进一”的记数方法就称为二进制，“满八进一”的记数方法就称为八进制。

进位制就好像一种打包方法。假设有一大堆苹果，现在要把它们装到箱子里。首先把苹果分成10个一组，装在一个个小箱子里，把它们称为1号箱。如果最后一组不到10个，就把它们放在外边。然后，再把1号箱分成10个一组，每组再装到大一点儿的2号箱里。同样，如果最后一组不够10个，就把它们放在外边，不再装入2号箱。这样继续下去，一直到不存在10个一样的箱子。最后清点每种箱子以及没有装箱的苹果的个数，并把这些数字按照箱子从大到小的顺序写在一起。例如，最大的4号箱有6个，没装进4号箱的3号箱有7个，没装进3号箱的2号箱有0个，没装进2号箱的1号箱有9个，外面还剩3个苹果。这样就得出一个数字：67093，这个数字正好是这些苹果的个数，准确地说是苹果个数的十进制表示。因为只要有10个一样的箱子，就把它装到大一号的箱子中，这就是“满十进一”。

如果你不是按照10个一组来装箱，而是按照2个一组来装箱，即只要有两个一样的箱子，就把它装到大一号的箱子中。这样最后从大到小清点箱子形成的数字就是苹果个数的二进制表示。以此类推，以8为单位装箱就可以得到这个数字的八进制表示。

日常生活中用得较多的记数方法是十进制。有时也会用到其他进制的记数方法。例如，在计时上，采用的就是六十进位制，60秒为一分钟，60分钟为一小时。

下面介绍与计算机有关的几种进位制。

3.1.1　进位制

1. 十进制

十进制是我们最熟悉的进位制。下面通过十进制了解与进位制有关的属性，仍然使用苹果装箱的例子。使用十进制，也就是说把苹果按照 10 个一组进行装箱。如果最后一组苹果不够装满一箱，那这几个苹果就不装箱。那么剩下的苹果数可能是 1～9 的任何一种情况或者一个不剩(为 0)。为了表示没有装到 1 号箱中的苹果个数，一共需要 10 个符号区分从“没有”到“九”这 10 种不同的情况。用 0～9 这 10 个阿拉伯数字完成这个任务，这 10 个阿拉伯数字称为十进制的数码。每个进位制的数码的个数称为该进位制的基数。例如，十进制有 0～9 一共 10 个数码，那么十进制的基数就是 10。十进制用英文字母 D 表示，它是英文单词 decimal 的首字母。

重　点　提　示

记数进位的规则称为进位制。
习惯上用进位制规定的进位的值命名记数方法。
每个进位制使用的数码个数称为该进位制的基数。

假定按照 10 个一组为单位进行苹果装箱，最后清点得出的数字是 6789。也就是说最后我们看到 6 个 3 号箱、7 个 2 号箱、8 个 1 号箱以及 9 个苹果。这也就意味着苹果的个数是 6789。为什么呢？因为每个 1 号箱中可以装 10 个苹果；每个 2 号箱中有 10 个 1 号箱，所以每个 2 号箱中有 10×10 个苹果；每个 3 号箱中有 10×10×10 个苹果。所以一共有

$$6\times10^3+7\times10^2+8\times10^1+9=6789$$

个苹果。把 6789 这个数右起第一位记作位置 1，第二位记作位置 2……第 n 位记作位置 n。可以看出，6789 这个数的每个位置和一个特殊的常数有关。例如，位置 4 对应 10^3，位置 2 对应 10^1……位置 n 对应 10^{n-1}。这个常数称为该位置的位权。例如，4 号位置的位权是 10^3。按照上面的记法，不同箱子的个数在数字中的位置是固定的。例如，3 号箱的个数出现在位置 4。其实每位上的位权就是对应箱子中苹果的个数。例如，4 号位置的位权就是 3 号箱中的苹果个数 10^3。通常位权用十进制数表示。位权是和数字的进位制有关系的。现在 4 号位置的位权为 10^3 是因为采用的是十进制。也就是说，苹果是按照 10 个一组装箱的，所以 3 号箱中的苹果数是 10^3。如果苹果是按照 2 个一组装箱的，那么 3 号箱中苹果的个数应该是 2^3，此时 4 号位的位权应该是 2^3，这是因为采用的是二进制记数法。

概括一下，用 q 进制表示的数，第 n 位(最右边一位为第 1 位)上的位权是 q^{n-1}。例如，二进制数的第 3 位上的位权为 2^2，十六进制数的第 2 位上的位权为 16。

接下来讨论小数的问题。看下面的例子：

$$0.1234=1\times10^{-1}+2\times10^{-2}+3\times10^{-3}+4\times10^{-4}$$

因此，规定小数点右边第一位的位权为 10^{-1}，第二位为 10^{-2}，以此类推。

现在把整数和小数的情况归纳在一起：

对于用 q 进制表示的数，其数码的顺序定义如下：小数点左边第一位定义为第 1 位，向左第二位定义为第 2 位，以此类推；小数点右边第一位定义为第 0 位，右边第二位定义为第 −1 位，以此类推，那么，第 n 位上的位权就是 q^{n-1}。

例如，二进制数小数点后第 3 位上的位权为 2^{-3}，十六进制数小数点后第 2 位上的位权为 16^{-2}。

从上面的分析可以看出，每个数字表示的实际数值（用十进制表示）应该是经过下列运算得出的值：把每个位置上的数码乘以该位置上的位权，然后再把这些乘积累加。例如，

$$6789=6\times10^3+7\times10^2+8\times10^1+9=6789$$

因为现在用到的数字 6789 和它所表示的数值 6789 都是十进制表示，所以当然一样了。如果不是十进制，这个数看起来就会不一样了。

重点提示

对于用 q 进制表示的数，其数码的顺序定义如下：小数点左边第一位定义为第 1 位，向左第二位定义为第 2 位，以此类推；小数点右边第一位定义为第 0 位，右边第二位定义为第 -1 位，以此类推，那么，第 n 位上的位权是 q^{n-1}。

2. 二进制

二进制就是“满二进一”。用苹果装箱的例子说，就是按照 2 个一组进行装箱。装到最后，未装箱的苹果和每种箱子最多只有一个，只需要 0 和 1 两个数码就够了，因此二进制的基数为 2。

现在假定只有 2 个苹果，那么我们可以把它们装入一个 1 号箱中。清点箱子和未装箱的苹果的个数，并按照箱子大小顺序写出来就是 10。最后一位应该是未装箱的苹果，如果没剩，应该用 0 表示。所以，十进制的数 2_{10} 用二进制表示就是 10_2（数字的后面用一个十进制数作为下标，指出这个数的进位制）。也可以用符号 B(Binary)表示二进制数，如 10B。

那么 3_{10} 用二进制怎么表示呢？现在重新清点箱子和苹果的个数：1 个 1 号箱和 1 个没装箱的苹果，得出数字 11。也就是说 3_{10} 的二进制表示是 11_2。

现在假定按照 2 个一组把苹果装箱，最后清点得出的数字是 1011_2。那么，一共有多少个苹果呢（用十进制表示）？

按照装箱和清点规则，1011_2 这个数字表示最后看到 1 个 3 号箱、0 个 2 号箱、1 个 1 号箱和 1 个苹果。按照 2 个一组装箱，现在每个 1 号箱中应该有 2 个苹果；每个 2 号箱中有 2 个 1 号箱，有 2×2 个苹果；每个 3 号箱中有 2 个 2 号箱，有 $2\times2\times2$ 个苹果。所以苹果的个数为

$$1\times2^3+0\times2^2+1\times2^1+1\times2^0=11$$

其实可以换一种说法，前面已经得出这个结论：每个数字表示的数值（用十进制表示）应该是每个位置上的数码乘以该位置上的位权再求和得出的值。

因为 1011_2 是二进制数，即按照 2 个一组进行苹果装箱，则二进制数第 4 位上的位权就应该是 3 号箱子中的苹果数 2^3，第 n 位上的位权应该是 $n-1$ 号箱子中的苹果数 2^{n-1}。那么二进制数 1011_2 表示的数值（用十进制表示）应该是

$$1011_2=1\times2^3+0\times2^2+1\times2^1+1\times2^0=11_{10}$$

可以得到同样的结果。

3. 八进制

八进制，顾名思义，就是“满八进一”。为什么要讲八进制呢？因为八进制和后面马上要说到的十六进制与二进制有很方便的转换关系，所以在计算机领域这两种进位制的使用非

常广泛。

计算机里面存储的是二进制数，但是因为二进制数每位只能表示两个数值，所以要表示一个较大的数就会用很多位。这在计算机中问题不大，但是书写起来就很不方便。例如，6789_{10} 写成二进制数就是 1101010000101_2。所以，尽管计算机实际上用的是二进制，但人们书写时通常将它们“浓缩”成八进制或十六进制。

八进制通常用字母 O(Octal)表示。八进制需要 8 个数码符号，它们就是 0～7。八进制数中肯定不会出现数码 8。6789_{10} 写成八进制数应该是 15205_8。因为八进制也就是按照 8 个一组进行苹果装箱，所以八进制数中位置 n 的位权为 8^{n-1}。八进制数 15205_8 表示的十进制数值可以用下面这个式子计算：

$$15205_8 = 1\times8^4 + 5\times8^3 + 2\times8^2 + 0\times8^1 + 5\times8^0$$

结果是 6789_{10}。

4. 十六进制

十六进制也就是“逢十六进一”，那么就需要 16 个不同的符号。人们用 0～9 这 10 个阿拉伯数字再加上字母 A～F 表示 0～15 这 16 个数值。6789_{10} 写成十六进制数就是 $1A85_{16}$。十六进制也就是按照 16 个一组进行苹果装箱，所以十六进制数中位置 n 的位权为 16^{n-1}。$1A85_{16}$ 表示的十进制数值就是 $1\times16^3 + 10\times16^2 + 8\times16^1 + 5\times16^0$。通常用 H(Hexadecimal)表示十六进制。

3.1.2　数的进位制转换

有时候需要查看计算机存储器的内容，尤其是寄存器的内容。例如，程序出错时，有些调试软件会显示相关寄存器的内容。不过这些内容不是以十进制的形式表示的，而是二进制。说得更精确一些，多数软件会以十六进制的形式(二进制的紧凑方式)显示寄存器的内容。如果想了解寄存器中的数值到底是多少，就要掌握把这些数转换成十进制表示的方法。有时候用户还要知道一个数在计算机中应该以什么样的一个 0/1 串表示，那么就需要把它转换成二进制表示。所以，要掌握数的不同进制表示之间转换的方法。需要注意的是，这里讨论的只是同一个数的不同表示方法，所谓转换，并不是将一个数变成了另一个数。

1. 转换成十进制表示

把二进制、八进制和十六进制数转换成十进制表示非常简单，只要把数的每位上的数码乘以该位上的位权，然后把这些乘积加在一起即可。下面通过实例分别看看这几种进制表示的转换。

1) 二进制数转换成十进制表示

例如，将二进制数 $1111\ 1111_2$ 和 0.1101_2 转换成十进制表示。因为二进制数第 n 位上的位权为 2^{n-1}，所以，

$$1111\ 1111_2 = 1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 255_{10}$$

$$0.1101_2 = 1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4} = 0.8125_{10}$$

2) 八进制数转换成十进制表示

例如，将八进制数 377_8 和 0.377_8 转换成十进制表示。因为八进制数第 n 位上的位权为 8^{n-1}，所以，

$$377_8 = 3\times8^2 + 7\times8^1 + 7\times8^0 = 255_{10}$$

$$0.377_8 = 3 \times 8^{-1} + 7 \times 8^{-2} + 7 \times 8^{-3} = 0.498\ 046\ 875_{10}$$

3）十六进制数转换成十进制表示

例如，将十六进制数 FF_{16} 和 $F.F_{16}$ 转换成十进制表示。因为十六进制数第 n 位上的位权为 16^{n-1}，所以，

$$FF_{16} = 15 \times 16 + 15 \times 16^0 = 255_{10}$$

$$F.F_{16} = 15 \times 16^0 + 15 \times 16^{-1} = 15.9375_{10}$$

清楚了这几种进位制的数到十进制数的转换，接下来看看这些不同进位制的数转换成二进制表示应该怎样做。

重点提示

把二进制、八进制和十六进制的数转换成十进制表示，只需把数的每位上的数码乘以该位上的位权，然后把这些乘积相加即可。

2. 转换成二进制表示

1）十进制数转换成二进制表示

还是用苹果装箱的例子说明。假设有 89 个苹果，现在要把这些苹果装箱。要得出二进制表示，所以装箱单位是 2，也就是“满二进一”。这样，最后各种箱子和未装箱的苹果的个数按顺序放在一起形成的数就是 89 对应的二进制表示。

现在转换问题就变成求各种箱子个数的问题。可以有两种算法：一种是从计算最大箱子的个数开始，然后再依次算小一号的箱子的个数；另一种则反过来，先从计算最小箱子的个数开始，一直算到最大号的箱子。两种算法得出的结果应该一样。

先说第一种算法。

这种算法首先要判断最大的箱子是几号。我们已经知道，在以 2 为单位装箱的情况下，n 号箱中苹果的个数是 2^n。要判断出使用的最大号箱子，就是要看看刚好比 89 小一点儿的 2 的幂是多少。因为 $2^6 < 89_{10} < 2^7$，所以最大的箱子应该是 6 号箱。装满一个 6 号箱之后，还剩 $89 - 2^6 = 25$ 个苹果。接下来要看看剩下的这些苹果能装满的最大箱是几号。因为 $2^4 < 25 < 2^5$，所以接下来用到的应该是 4 号箱。现在还剩 $25 - 2^4 = 9$ 个苹果。继续为这 9 个苹果找最大的箱子，以此类推。最后得到的结果是：1 个 6 号箱，1 个 4 号箱，1 个 3 号箱，还有一个苹果未装箱。按顺序把这些数放在一起，中间没有用到的箱子用 0 表示，最后得到的数字串是 1011001。这个数字串就是 89 的二进制表示。

这种算法被称为减权定位法。其过程是：将要转换的十进制数和相近的权值比较，从中减去比该数小的最大权值，并确定与该权值对应的数位上的数码为 1。以此类推，直到余数是 0 为止，未被记 1 的数位均记为 0，即可得到相应的二进制数。

使用减权定位法需要记住每位上的权值。相较于此，第二种算法更容易一些。这种算法就是从小号箱的个数算起。首先要算这些苹果能够装满多少个 1 号箱，剩下多少苹果。剩下的苹果数就是二进制数的最后一位。因为每个 1 号箱中能装两个苹果，89 除以 2 等于 44 余 1，所以要用到 44 个 1 号箱，剩下 1 个苹果。因为 44/2=22，所以这些 1 号箱正好装入 22 个 2 号箱中，一个没剩，因此最后能看到的 1 号箱的个数是 0。继续将 2 号箱装入 3 号箱，因为 22/2=11，所以这些 2 号箱正好装入 11 个 3 号箱中，一个没剩，因此最后能看到的

2 号箱的个数是 0。以此类推，直到剩余的箱子不够再装入大一号的箱子。最后清点箱子和未装箱的苹果的个数，把这些数按顺序形成一个数字串，这个数字串就是 89 的二进制表示。当然，结果肯定与第一种算法相同。

这种算法其实就是反复地把十进制数除以 2，直到商为 0，最后形成的二进制数就是从最后一次除法到第一次除法的余数排列起来。这种算法被称为除基取余法（“基”就是进位制的基数，二进制的基就是 2），先得到的余数为低位，后得到的余数为高位。

一般采用下面的竖式表示上面的过程。

例如，求 89 的二进制表示：

$$
\begin{array}{r|r|ll}
2 & 89 & 1 & p_1 \\
2 & 44 & 0 & p_2 \\
2 & 22 & 0 & p_3 \\
2 & 11 & 1 & p_4 \\
2 & 5 & 1 & p_5 \\
2 & 2 & 0 & p_6 \\
2 & 1 & 1 & p_7 \\
 & 0 & &
\end{array}
$$

则　　$89_{10}=p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1=1011001_2$

小数的转换也有两种方法。一种是减权定位法，与前面整数的情况一样。另一种方法略有不同，小数的转换不再采用除法，而是采用乘法，称为乘基取整法。具体的做法是：乘以基数并取整，先得到的整数为高位，后得到的整数为低位。也就是小数部分不断地乘以 2，直到乘积的小数部分为 0，然后把每次乘积的整数部分按照先后顺序排列在一起。

例如，求 0.8125 的二进制表示：

$$
\begin{array}{rll}
 & 0.8125 & \\
\times & 2 & \\
\hline
 & 1.6250 & \text{整数 } 1 \quad p_1 \\
 & 0.6250 & \\
\times & 2 & \\
\hline
 & 1.2500 & \text{整数 } 1 \quad p_2 \\
 & 0.2500 & \\
\times & 2 & \\
\hline
 & 0.5000 & \text{整数 } 0 \quad p_3 \\
 & 0.5000 & \\
\times & 2 & \\
\hline
 & 1.0000 & \text{整数 } 1 \quad p_4
\end{array}
$$

则　　$0.8125_{10}=p_1\ p_2\ p_3\ p_4=0.1101_2$

如果要转换的十进制数既有整数部分又有小数部分，那么要将这两部分分开，分别用除基取余法和乘基取整法求得转换后的整数部分和小数部分，最后再把它们合在一起。

2）八进制数转换成二进制表示

八进制数和十六进制数转换成二进制数则比较简单。因为 $2^3=8$，所以 3 位二进制数正好等于一位八进制数。以下是八进制的 8 个数码对应的二进制数码：

0	1	2	3	4	5	6	7
000	001	010	111	100	101	110	111

把八进制数转换成二进制数时，直接把这些数码对换成相应的三位二进制数码就可以了。例如：

$$765_8 = 111\ 110\ 101_2$$

$$0.765_8 = 0.111\ 110\ 101_2$$

3）十六进制数转换成二进制表示

因为 $2^4=16$，所以一位十六进制数正好等于 4 位二进制数，它们的关系也是一一对应的。16 个十六进制数码对应的二进制数码如下：

0	1	2	3	4	5	6	7
0000	0001	0010	0011	0100	0101	0110	0111
8	9	A	B	C	D	E	F
1000	1001	1010	1011	1100	1101	1110	1111

把十六进制数转换成二进制数非常简单，只要把每位上的十六进制数码替换成 4 位二进制数码即可。例如：

$$A13F_{16} = 1010\ 0001\ 0011\ 1111_2$$

$$0.A13F_{16} = 0.1010\ 0001\ 0011\ 1111_2$$

当然和八进制数转换成二进制数一样，这需要熟记这些数码的对应关系。因为十六进制数和二进制数有这样简单的对应关系，而且十六进制数比较简短，比二进制数更容易读写，所以平时都用十六进制数描述计算机中的二进制数。例如，需要显示计算机中寄存器的内容时，通常都以十六进制表示。

重点提示

十进制数转换成二进制表示有减权定位法和除基取余法。
八进制数转换成二进制表示，只要把每位上的八进制数码替换成相应的 3 位二进制数码即可。
十六进制数转换成二进制表示，只要把每位上的十六进制数码替换成相应的 4 位二进制数码即可。

3. 转换成八进制表示

有了前面的铺垫，现在你对到八进制数的转换已经能够举一反三了吧？

1）十进制数转换成八进制表示

十进制数转换成八进制表示用除基取余法，只不过现在的基数是 8。

例如，求 89 的八进制表示：

$$\begin{array}{r|l|ll} 8 & 89 & 1 & p_1 \\ 8 & 11 & 3 & p_2 \\ 8 & 1 & 1 & p_3 \\ & 0 & & \end{array}$$

所以 $$89_{10} = p_3\ p_2\ p_1 = 131_8$$

请你把这个八进制数转换成二进制表示，看看与前面直接从 89 转换成的二进制数是否相同。

2）二进制数转换成八进制表示

把二进制数从最后一位开始分成 3 位一组，然后替换成相应的八进制数码就可以了。

例如，求 1010000100111111_2 的八进制表示，转换如下：

$$1010000100111111_2 = 1\ 010\ 000\ 100\ 111\ 111_2 = 120477_8$$

对于小数，要从小数点后开始分成 3 位一组，最后的一组如果不够 3 位，则要补 0。例如，求 0.1010000100111111_2 的八进制表示，转换如下：

$$0.1010000100111111_2 = 0.101\ 000\ 010\ 011\ 111\ 100_2 = 0.502374_8$$

重　点　提　示

二进制数转换成八进制表示，把二进制整数从最后一位开始向前分成 3 位一组，然后把每组替换成相应的八进制数码即可。

二进制小数则从小数点后第一位开始向后分成 3 位一组，如果最后一组不足 3 位，则用 0 补齐，然后把每组替换成相应的八进制数码即可。

3）十六进制数转换成八进制表示

十六进制数转换成八进制表示，最简单的办法是把它先转换成二进制表示，然后再转换成八进制表示。例如，求 $A13F_{16}$ 的八进制表示：

$$A13F_{16} = 1010\ 0001\ 0011\ 1111_2 = 1\ 010\ 000\ 100\ 111\ 111_2 = 120477_8$$

求 $0.A13F_{16}$ 的八进制表示：

$$0.A13F_{16} = 0.1010\ 0001\ 0011\ 1111_2 = 0.101\ 000\ 010\ 011\ 111\ 100_2 = 0.502374_8$$

4. 转换成十六进制表示

1）十进制数转换成十六进制表示

十进制数转换成十六进制表示采用除基取余法，基数为 16。例如，求 89_{10} 的十六进制表示：

16	89	9	p_1
16	5	5	p_2
	0		

所以　　$89_{10} = p_2\ p_1 = 59_{16}$

重　点　提　示

把十进制数转换成二进制、八进制和十六进制表示，可以采用减权定位法或者除基取余法（小数部分采用乘基取整法）。

2）二进制数转换成十六进制表示

将二进制数转换成十六进制表示时，把二进制整数从最后一位开始分成 4 位一组，然后把每组替换成相应的十六进制数码就可以了。例如，求 1010000100111111_2 的十六进制表示，转换如下：

$$1010000100111111_2 = 1010\ 0001\ 0011\ 1111_2 = A13F_{16}$$

二进制小数则从小数点后第一位开始分成 4 位一组，如果最后一组不足 4 位，则用 0 补齐，然后把每组替换成相应的十六进制数码。例如，求 0.1010000100111111_2 的十六进制表示，转换如下：

$$0.1010000100111111_2 = 0.1010\ 0001\ 0011\ 1111_2 = 0.A13F_{16}$$

重点提示

二进制数转换成十六进制表示，把二进制整数从最后一位开始向前分成 4 位一组，然后把每组替换成相应的十六进制数码即可。

二进制小数转换成十六进制表示，则从小数点后第一位开始向后分成 4 位一组，如果最后一组不足 4 位，则用 0 补齐，然后把每组替换成相应的十六进制数码即可。

3）八进制数转换成十六进制表示

将八进制数转换成十六进制表示时，把它先转换成二进制表示，然后再转换成十六进制表示。例如，求 120477_8 的十六进制表示：

$$120477_8 = 1\ 010\ 000\ 100\ 111\ 111_2 = 1010\ 0001\ 0011\ 1111_2 = A13F_{16}$$

重点提示

十六进制数转换成八进制表示，把它先转换成二进制表示，然后再转换成八进制表示。

八进制数转换成十六进制表示，把它先转换成二进制表示，然后再转换成十六进制表示。

3.1.3 二进制数的运算

进入计算机的世界后，我们将频繁地接触二进制数。下面简要介绍二进制数的运算规则。

二进制数主要涉及的运算有两类：算术运算和逻辑运算。算术运算主要包括加、减、乘、除，而逻辑运算主要包括逻辑与、逻辑或、逻辑非以及逻辑异或。

1. 算术运算

1）加法运算

二进制数的加法运算和十进制数的加法运算类似，只是进位时要记住“满二进一”。例如，计算 $1101_2 + 1011_2$：

$$\begin{array}{r} 1101 \\ +\ 1011 \\ \hline 11000 \end{array}$$

因此 $$0101_2 + 1011_2 = 11000_2$$

2）减法运算

二进制数的减法运算也同样与十进制数的减法运算类似，只是从上一位借位时“借一当二”。例如，计算 $1101_2 - 1011_2$

$$\begin{array}{r} 1101 \\ -\ 1011 \\ \hline 10 \end{array}$$

因此 $$1101_2 + 1011_2 = 10_2$$

3）乘法运算

二进制数的乘法运算的法则也和十进制数相同，即分别用乘数的每一位去乘被乘数，然后把这些结果累加。例如，计算 $1101_2 \times 1011_2$：

$$\begin{array}{r} 1101 \\ \times\ 1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ +\ 1101 \\ \hline 10001111 \end{array}$$

所以　　$1101_2 \times 1011_2 = 10001111_2$

4）除法运算

二进制数的除法运算法则也和十进制数相同，由减法、逐位上商等操作分步完成。例如，计算 $10001111_2/1101_2$：

$$\begin{array}{r} \quad\ 1011 \\ 1101\overline{)\,10001111} \\ \quad\ 1101 \\ \hline \quad\ 10011 \\ \quad\ 1101 \\ \hline \quad\ 1101 \\ \quad\ 1101 \\ \hline \quad\ 0 \end{array}$$

所以　　$10001111_2/1101_2 = 1011_2$

重点提示

二进制数主要的算术运算有加、减、乘、除。

二进制数算术运算的法则和十进制数相同，只是“满二进一”和“借一当二”。

2. 逻辑运算

逻辑运算是十进制数中没有的。常见的逻辑运算有逻辑与、逻辑或、逻辑非以及逻辑异或。其实逻辑运算中二进制数的 0 和 1 不再是数值的含义，而是分别代表真和假两个值，1 代表真，而 0 代表假。所以实际上逻辑运算是在真值和假值上的运算。在信息技术应用数学中还会涉及这些逻辑运算。对于两位以上的二进制数来说，逻辑运算是在每个操作数对应的位之间进行的，所以也称为按位与、按位或等。下面通过具体的示例介绍逻辑运算规则。

1）逻辑与

逻辑与的运算符号为 Λ 或者 &。它的运算规则如下：

$$1\&1=1,1\&0=0,0\&1=0,0\&0=0$$

即，只有两个参与逻辑与运算的数都是 1 时运算结果是 1，其余情况结果都为 0。其实逻辑与有“和”“并且”的意思。如果 1 代表真，0 代表假，那么对于事情 A、B，要“A 并且 B”为真时，显然必须 A 是真的并且 B 也是真的才行。

例如，求 1101 & 1011：

$$\begin{array}{r} 1101 \\ \&\ 1011 \\ \hline 1001 \end{array}$$

因此　　1101 & 1011＝1001

2）逻辑或

逻辑或的运算符号为∨或者|。它的运算规则如下：

1|1＝1,1|0＝1,0|1＝1,0|0＝0

即1和任何数进行逻辑或运算结果都为1,其余情况结果为0。逻辑或有“或者”的意思。对于事情A，B,如果要“A或者B”为真,那么只要A和B中至少有一个是真的就行了。

例如,求1101 | 1011：

$$\begin{array}{r} 1101 \\ |\ \ 1011 \\ \hline 1111 \end{array}$$

因此　　1101 | 1011＝1111

3）逻辑非

逻辑非是单值运算,也就是只对一个数进行操作。它的运算符号是～或者在数字的上面画一条横线。它的运算规则如下：

～1＝0，　～0＝1

也就是逐位取反。例如：

～1101＝0010

4）逻辑异或

逻辑异或的运算符号是⊕。它的运算规则如下：

1⊕1＝0,1⊕0＝1,0⊕1＝1,0⊕0＝0

可以看出,逻辑异或的意思就是看看两个操作数是否不一样。如果不一样,结果为1;否则,结果为0。

例如,计算1101⊕1011：

$$\begin{array}{r} 1101 \\ \oplus\ \ 1011 \\ \hline 0110 \end{array}$$

因此　　1101⊕1011＝0110

重点提示

二进制数主要的逻辑运算有逻辑与、逻辑或、逻辑非以及逻辑异或。

3.2　数字的编码

我们现在已经知道两个事实：一是计算机中存储的是0/1串;二是我们向计算机中存储的是程序和数据。程序是用某种计算机程序设计语言编写的,具体体现形式就是一个字符串,包括数字、字母、汉字、运算符以及一些特别规定的符号等。而数据则包括用来运算的数以及用户的一些文档,如最近新写的一首小诗。那么,这些日常使用的数字、字母等与0/1串有什么关系呢？它们是怎样互相转换的呢？这就是编码问题。

下面就来说说数字、字母以及汉字在计算机中是怎样表示的。先从数字说起。

3.2.1　原码

从 3.1 节的介绍可知，数的表示比较简单。你现在一定在想：把各种数字转换成二进制表示就行了。你说的没错。不过还有一个小问题。我们平时用的数是分正负的。如果是负数，前面的负号怎么表示呢？你肯定会说，在转换后的二进制数前单独加一位表示符号就行了。如果是负数，符号位就为 1；如果是正数，符号位就为 0。这确实是一种办法。这种数据表示法被称为原码。

一般情况下，计算机中的数都是要参与运算的。最常见的运算就是加、减。现在看一看用原码表示的数在计算机中进行加法运算时的步骤。假设要计算 57 和－68 的和。

先把它们转换成二进制表示：

$$57_{10}=111001_2$$

$$68_{10}=1000100_2$$

接下来要做的事情就是加上符号位。现在出现了一个小麻烦。如果直接在 1000100_2 前面加上一个 1 表示负号，变成 11000100，那么怎样才能区分它表示负的 1000100_2 还是正的 11000100_2 呢？为此，需要把符号位的位置固定下来，例如，在从最右边开始向左数的第 8 位（从 1 开始计数）作为符号位，则

$$57_{10}=111001_2,[57_{10}]_{原}=0\ 011\ 1001$$

$$68_{10}=1000100_2,[-68_{10}]_{原}=1\ 100\ 0100$$

这意味着后面有 7 位可以用来表示数值。前面已经讲过，地址总线的宽度决定最大能访问的地址。同样道理，如果已经决定用 8 位表示数，并且 1 位作为符号位，余下 7 位表示数的绝对值，那么可以表示的数的范围就固定了。就目前这种假设情况而言，能表示的数的范围是什么？

具体地说，如果把原码的长度确定为 n 位，则数 x 的原码定义如下：

$$[x]_{原}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ 2^{n-1}-x, & -2^{n-1}<x\leqslant 0\end{cases}$$

注意，0 的原码有两个。

现在要通过 57 和－68 这两个数的原码 0 011 1001 和 1 100 0100 计算它们的和，并且把和保存在计算机中，也就是说结果也要转换成原码的形式。两个带符号数的加法步骤如下：

```
首先判断两个加数的符号位；
如果符号位相同
    则将两个数相加，和的符号位与加数相同；
否则（如果符号位不同）
    比较两个数绝对值的大小；
    用绝对值大的数减去绝对值小的数，得出结果的绝对值；
    结果的符号位与绝对值大的加数相同。
```

本例求和步骤如下：

（1）通过 57 和－68 的原码判断它们的符号位是否相同。可以看到 57 和－68 的原码符号位分别为 0 和 1，不相同。

(2) 比较这两个数的绝对值的大小，也就是原码的数值部分。可以发现 100 0100 大于 011 1001，所以用 100 0100 减去 011 1001，结果为 000 1011。

(3) 确定结果的符号位，它应该与绝对值大的数(也就是 100 0100)的符号位相同，即为 1。所以

$$[-68_{10}]_{原}+[57_{10}]_{原}=1\ 000\ 1011=[-11_{10}]_{原}$$

可以发现，对于这个求和过程，最后运算过程是加法还是减法取决于具体加数的情况。所以，为了实现原码表示数的加法，计算机中需要有能够判断符号位的硬件逻辑，还需要有能计算加法的逻辑电路，以及能够计算减法的逻辑电路。这样的硬件设计有点儿复杂。如果不用考虑符号位，并且也不用判断两个数的大小，只要把计算机中两个数的编码直接相加，就能得出正确结果，那该多好！这个目标是通过补码实现的。

最后还要说一点，原码除了与数的真值有简单的对应关系，比较容易相互转换之外，还有一个优点，就是乘除法的运算规则简单。等学习完后面几种数字的编码，读者可以自己想想怎样实现这些编码的乘除法，然后看看是不是这样。

重点提示

原码表示法直接用数字绝对值对应的二进制数表示，并在前面加一位符号位，1 表示负号，0 表示正号。

数的原码同其真值有简单的对应关系，比较容易相互转换。

原码实现加减法很不方便，但实现乘除法的运算规则简单。

3.2.2 补码

前面讨论原码时，假设计算机用 8 位表示一个数。这种情况下，可以出现的不同 0/1 串范围为 0000 0000～1111 1111，一共有 256 种。对于原码表示法，第一位是符号位，因此可以表示的正数为 0000 0000～0111 1111，换算成十进制整数就是 0～127；而负数为 1000 0000～1111 1111，即 0～－127。所以 8 位原码可以表示的整数范围为－127～127，一共 255 个整数(数字 0 占用了两个码：1000 0000 和 0000 0000)。

现在讨论 3.2.1 节的目标，找到一个数字编码形式，它在进行加法运算时不用考虑符号位，也不用判断两个数的大小，只要把计算机中两个数的编码直接相加，就能得出和的相应编码结果。

现在设计这种编码。还是假定用 8 位表示整数。我们已经知道，8 位一共可以表示 256 个不同数。如果全用来表示正数，则可以表示 0～255。现在希望用它表示有符号数，并且 0 只占用一个 0/1 串，由此确定新的编码所表示的数的范围为－128～127。仍然用 0000 0000～0111 1111 这 128 个 0/1 串表示和它们的二进制数值相同的数 0～127；而对应 128～255 这 128 个正数的 0/1 串，即 1000 0000～1111 1111，则用来表示－128～－1 这 128 个负数。用 128 对应的二进制串 1000 0000 表示－128，用 129 对应的二进制串 1000 0001 表示－127，以此类推，用 255 对应的二进制串 1000 0010 表示－1。从中可以发现一个规律：128＝－128＋256，129＝－127＋256，…，255＝－1＋256。我们把这两个数的关系称为模 256 的补数。现在假设用这种编码，暂时把这种编码称为简单码，因为它的确很简单。现在计算 57 和－68 在这种编码规则下对应的数据表示。

因为 $57_{10}=111001_2$，所以$[57_{10}]_{简}=0011\ 1001$。

因为 $256-68=188$，$188_{10}=1011\ 1100_2$，所以$[-68_{10}]_{简}=1011\ 1100$。

现在把$[57_{10}]_{简}$和$[-68_{10}]_{简}$加在一起：

$$\begin{array}{rr} & 0011\ 1001 \\ + & 1011\ 1100 \\ \hline & 1111\ 0101 \end{array}$$

$$1111\ 0101_2=245_{10}=256-11=[-11_{10}]_{简}$$

也就是说$[-68_{10}]_{简}+[57_{10}]_{简}=[-11_{10}]_{简}$。

这是巧合吗？再找另外两个数试验一下，计算－68 和－57 的和。

因为　　$256-57=199$，$199_{10}=1100\ 0111_2$

所以　　$[-57_{10}]_{简}=1100\ 0111$

而　　$[-68_{10}]_{简}=1011\ 1100$

那么　　$[-68_{10}]_{简}+[-57_{10}]_{简}=1011\ 1100+1100\ 0111=1\ 1000\ 0011$

因为现在的数据只能有 8 位，也就是说存储器中只为每个数留了 8 位的空间，所以，最前面的进位 1 没地方放，被丢弃了。结果就是 1000 0011。

$$1000\ 0011_2=131_{10}=256-125=[-125_{10}]_{简}$$

而－68 和－57 的和正好是－125。看来不是巧合。那么我们就找到了希望的编码：计算两个有符号数的加法时，只要直接把它们的简单码相加，就可以得出和，而不用考虑加数的符号。

其实，这样的结果完全是由于前面发现的那个规律，$129=-127+256$，即$[-127]_{简}=129=256+(-127)$。这里面蕴含着两个概念，就是“模”与“同余”。

模是指一个计量器的容量。例如，日常使用的钟表上面用 12 个格表示整点，所以钟表的模就是 12。可以用钟表表示 19 点，可是表盘上的指针指向 7 点，所以说 19 和 7 是模 12 同余的，因为 19 和 7 除以 12 的余数都是 7，也就是说，在 12 个格的表盘上，7 点和 19 点的表示是相同的，都是 7。

现在再来看同余。假设现在是 7 点，如果需要把时间调整到 3 点，有两种办法：可以把表针向回拨 4 个格，也可以把表针向前拨 8 个格。结论是，在只有 12 个格的表盘上，$7-4=7+8$。现在可以发现－4 和 8 是有一定关系的。仔细再观察一下可以发现，在表盘上表示 8 和－4 的其实是同一个格，都是 8 点，只不过看你从哪个方向数数。从 0 点开始顺时针数，那个格就表示 8；如果从 0 开始逆时针数，那个格就表示－4。实际上，－4 和 8 也是模 12 同余的，称－4 和 8 互为补数。这里有两个规律。首先，将负数加上模就可以得出它的正补数，例如，$-4+12=8$。其次，与一个负数的加法可以转换成与这个负数补数的加法，例如，$7+(-4)=7+8(\mathrm{mod}\ 12)$。为了避免混淆，在算式中加上(mod 12)表示以 12 为模。

现在就清楚为什么前面的编码能把同负数的加法直接转换成两个数的加法，而不用考虑符号了。在这种情况下，因为只有 8 位的存储空间存放整数，所以只能表示 256 个数，这样存放整数的容器的模就是 256。为了把减法转换成加法，要把负数用它的正补数表示，即－127 用 129(＝256＋(－127))表示。然后，当要计算与某个负数的加法，例如 25 加上－127 时，则直接把它们的编码表示相加，也就是 25＋129＝154。因为表示数的范围是－128～127，所以会把 154 解释成它的负补数，就是－102(＝154－256)。实际上是用补数表示每个

数(包括正数,因为正数的补数就是它自己),所以这种编码的名称不是简单码,而是补码。

正数的补码就是它本身。负数的补码等于模加上这个负数,实际上就是模减去负数的绝对值。模是多少呢?对于 8 位二进制数是 256($=2^8$),对于 n 位二进制数就是 2^n。

总结一下,如果补码的长度为 n 位,那么数 x 的补码定义为

$$[x]_{补}=\begin{cases}x, & 0\leqslant x<2^{n-1}\\ 2^n+x, & -2^{n-1}\leqslant x<0\end{cases}$$

从定义可以看出,正数的补码比较好求,而负数的补码要做一个减法运算。

例如,求 -68 的补码:

$$68_{10}=1000100_2$$

$$2^8_{10}=1\ 0000\ 0000_2$$

所以 $$[-68_{10}]_{补}=100000000-0100\ 0100=1011\ 1100$$

实际上,通常不用计算这个二进制减法,而是用另一种简单的算法,即把 68 对应的二进制数 0100 0100(一定要记住在最前面加 0,补全 8 位)按位取反(即变成 1011 1011),然后在末位加 1(1011 1011 + 1=1011 1100),这样就把减法换算成取反加一这样简单的运算。这种算法用硬件实现非常简单。

再介绍一种求负数补码的简单办法。首先,把负数的绝对值对应的二进制数求出来,并在前面补上必要的 0,补全位数,例如 8 位;然后,从后面开始找到第一个 1,把它记下来,并把其后的 0 也都写下来;最后,把这个 1 前面的各位取反,即 1 变 0、0 变 1。以 -68 为例,假设这次用 16 位编码。

第一步,求出 68 的二进制表示,并补全位数(千万别忘了):

$$68_{10}=0000\ 0000\ 0100\ 0100_2$$

第二步,找到第一个 1,并将其以及后面的 0 照抄下来。第三位上的 1 是从右边数第一个遇到的 1,所以将右边的 3 位(100)保留下来。

第三步,将第三位上的 1 前面的各位都取反,就得到了补码:

$$[-68]_{补}=1111\ 1111\ 1011\ 1100$$

用补码的定义验证一下,看看结果对不对。

补码这种把减法转换成加法的特性可以大大简化计算机硬件的设计。想象一下,如果用原码,那么设计加法运算的硬件时必须有两套电路,一套算加法,一套算减法,而且在运算之前还要有一套电路对加数的符号进行比较,然后决定用加法电路还是减法电路,运算的结果还要根据前面比较的情况确定符号,非常复杂。而使用补码就很简单,只要一套加法电路就行了,根本不用考虑符号,并且减法都可以转换成加法实现。负数的补码也很好求,取反加 1,在电路上实现也非常简单。所以,补码是目前计算机中使用很广的一种数字表示方式。

重点提示

正数的补码就是它的二进制表示。

负数的补码可以通过将其二进制表示按位取反、末位加 1 的方法获得。

补码实现加减法的规则比较简单。

3.2.3　反码

数的第三种表示法称为反码。它的定义很简单：正数的反码等于正数本身，负数的反码等于把负数绝对值的二进制表示每位取反。其实这就是求补码的中间步骤，即按位取反得出的结果。只不过补码多了个加 1 的步骤。

计算机中很少用反码直接进行运算。

3.2.4　移码

除了上述几种数字编码外，有时也会用到一种称为移码的表示法。如果移码的长度为 n 位，那么数 x 的移码定义为

$$[x]_{移} = 2^{n-1} + x, \quad -2^{n-1} \leqslant x < 2^{n-1}$$

为什么叫移码呢？因为移码是原来的数加上 2^{n-1}，在数轴上也就是将数的位置向数轴的正方向移动 2^{n-1}，因此得名。移码和补码有一个很有用的关系：一个数的补码和移码的符号位正好相反，而其他位则完全相同。例如，采用 8 位编码，十进制数 7 的移码是 1000 0111，补码是 0000 0111；而 −7 的移码是 0111 1001，补码是 1111 1001。

3.2.5　小数的表示

前面都是在讲整数怎样表示，关键在于解决符号的表示问题。不过，除了整数，很多地方还要用到小数。小数比整数多了一个小数点，它把数值分成整数部分和小数部分。所以，要表示小数就要解决小数点的表示问题。

通常有两种办法可以解决这个问题。第一种办法是把小数点固定在某个默认位置，小数点并不在数的表示中出现。这种方法表示的数被称为定点数。通常小数点的固定位置有两种情况。一是默认固定在数的最右边。也就是说，这种数只有整数部分，实际上也就是整数，因此也把这种数称为定点纯整数。二是把小数点固定在符号位之后。那么，这种数也就只有小数部分，因此也被称为定点纯小数。

定点表示法可以表示数的范围很有限，要么是整数，要么是小数。可是，通常使用的数这两部分都是有的。要表示这样的数，就应该允许小数点位置变化，也就需要在表示中给出小数点位置。这就是第二种小数的表示法，称为浮点表示法，意思是小数点可以浮动位置。这种表示法表示的数称为浮点数。

那么，怎样表示小数点的位置呢？计算机采纳了科学记数法的方式。小数 0.000001234 用科学记数法表示就是 1.234×10^{-6}。

在科学记数法中，乘号前面的小数部分，如 1.234，给出了有效数字；而后面的幂，如 10^{-6}，给出了小数点的位置。计算机中的浮点数也采用这种方式。因此，一个浮点数分为两部分：一部分称为尾数，就是科学记数法中的有效数字部分，如 1.234；另一部分称为阶码，记录的是后面的幂次，如前面的 −6。幂 10^{-6} 中的 10 给出的是这个数的基数，即这是一个十进制数。但是计算机系统并没有记录基数。通常在一个系统中会使用默认的基数，如 2 或者 8，并不是 10。

浮点数的尾数一般用补码或原码表示，阶码则用补码、原码或移码表示。与科学记数法不同的是，浮点数的尾数只记录小数部分，省略了科学记数法中的 1 位整数部分（因为对于

二进制数来说，这 1 位整数部分的值只能是 1，所以就可以省略了）。另外，阶码只记录了幂的次数。计算机中使用的基数有 2、8、16 等，通常默认的基数是 2，而不是科学记数法通常采用的 10。

前面介绍了这么多种浮点表示可以采用的编码，如原码、补码，数字的各种编码都可以使用。当然，为了便于软件的移植，最好都用相同的编码。为此，IEEE 在 1985 年给出了 IEEE 754 标准。这个标准规定，浮点数的基数为 2，阶码用移码表示，尾数用原码表示。它规定了单精度和双精度两个基本格式。单精度格式浮点数为 32 位，分别是 1 位符号位、8 位指数位（阶码）、23 位有效位（尾数）。

双精度是什么就不介绍了。如果对具体的定义感兴趣，请查阅其他计算机原理方面的书，你会发现在数的表示上还有好多要仔细考究的地方。

3.3 字符编码

前面介绍了数字在计算机中是怎样表示的。但是程序在计算机中是怎样存储的呢？程序一般是由英文字母加上一些符号构成的。因为计算机中只能存储 0 和 1，所以必须为每个字母和符号规定一个 0/1 串表示，称为字符的编码。当然每个人都可能会给出一种编码。例如，小明希望用 0111 0000 表示小写字母 a，而用 1111 0000 表示大写字母 F；可是 Tom 喜欢用 0111 0000 表示字母 T，而用 1111 0000 表示字母 o，用 0000 0100 表示字母 m。显然，要想使不同的计算机能够互相识别各自表示的字符，就必须使用相同的编码，这就是编码标准。不过，标准也有很多。各个国家和很多公司都制定了标准。本节介绍 3 个有代表性的编码标准，它们分别是：使用最多的字符编码——ASCII 码，汉字编码标准之一——GB 2312，以及力争统一全世界字符的统一码——Unicode。

3.3.1 简单字符的编码：ASCII 码

使用最广泛的编码标准是美国信息交换标准代码（American Standard Code for Information Interchange），其英文首字母缩写是 ASCII，所以这种编码被称为 ASCII 码。这种编码用 8 位表示 128 个字符。这些字符包括 10 个阿拉伯数字（0～9）、52 个英文字母（a～z 和A～Z）、34 个专用符号和 32 个控制符号。表示 128 个不同的符号只要 7 位二进制数就够了，而 8 位的 0/1 串可以有 256 个不同序列。实际上 ASCII 码的最高位都为 0，也就是说它只用了后面的 7 位，所以一共只表示了 128 个不同的字符。具体的编码规则如表 3-1 所示。有些公司会对最高位进行定义，这样就可以再多表示 128 个字符，这种编码被称为扩展的 ASCII 码。

在表 3-1 中可以看到，ASCII 码用十进制数 97 对应的二进制 0/1 串，也就是 0110 0001，表示小写字母 a。根据这种编码规则，如果想在计算机中存储小写字母 a，计算机就在存储器中存储 0110 0001。现在问题出现了。如果存储器某个单元的内容是 0110 0001，它表示的是字母 a 还是数字 97 呢？

在实际的计算机系统中，如果计算机处于指令周期的取指令阶段，CPU 从存储器取出的内容会被当作机器指令来解释。机器指令也是一种 0/1 串，将在 3.4 节说明。在执行指

表 3-1　ASCII 码表

十进制	十六进制	八进制	字符	十进制	十六进制	八进制	字符	十进制	十六进制	八进制	字符	十进制	十六进制	八进制	字符
0	0	000	NUL	32	20	040	SPACE	64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

令阶段，CPU 需要从存储器取操作数进行运算，这时从存储器取出的内容就会被解释成数据。但是，这并不能回答前面提出的问题。如果取数据时取出 0110 0001，那么可以解释成 97，也可以解释成字母 a。这个问题的解决留给了程序设计。当用某种程序设计语言写程序时，如果需要在存储器中存储数据，必须首先申请存储单元，并且要指明这个存储单元准备存储什么类型的数据。用程序设计语言的说法就是必须首先定义一个变量并且声明变量的类型。变量其实就是给存储空间起的别名，就好像把中央大街 115 号起名为“空间站”。这样，只要说“空间站”，指的就是中央大街 115 号。定义变量就是申请存储空间，当程序中定义了一个变量时，计算机会分配一个存储单元，并且用程序中给出的变量名标识这个存储单元。例如，程序中定义了一个名叫 age 的变量。定义变量除了要起名外，还要声明变量的类型，也就是说明该存储单元要用来存储什么类型的数据。例如，int age 表示变量 age 标识的存储单元要存储整数。这样，当计算机从 age 这个存储单元取出 0110 0001 时，就会把这个 0/1 串解释成整数 97，而不是字母 a。与之相对的是，如果定义 age 时写的是 char age，表示变量 age 所对应的存储单元要存储字符。那么，当计算机从 age 中取出 0110 0001 时，就会把它解释成字母 a，而不是整数 97。

重点提示

为字符规定一个对应的 0/1 序列，称为字符编码。
现在存在很多个字符编码标准。
计算机中的 0/1 串可以被解释成字符、数字、指令等，计算机系统根据具体情况进行相应的解释。
程序中的变量必须声明类型，计算机才能正确解释它的内容。

3.3.2 汉字字符的编码：GB 2312

ASCII 码解决了简单字母和符号在计算机中的表示问题。可是，要在计算机中存储的内容还有很多。例如，你想把最近写的一首小诗存在计算机中，然后把它放到网上，让大家欣赏。当然，你的诗是用汉字写的。再如，要把原版的《一千零一夜》做成电子版。它是用阿拉伯文写的，同英文以及汉字完全不同。其实，仍然只要为每个需要表示的字符定义一个 0/1 串就行了。每一种定义就形成一种编码。当然，在解释这些 0/1 串时，要知道用的是哪种编码方案，才能还原出最初的内容。上网时用户经常会遇到一种情况：网页上显示的都是一些看不懂的符号，人们将其称为乱码。这时需要在浏览器的菜单栏中找到“查看”→“编码”选项(见图 3-1)，然后就会看到很多名称，例如阿拉伯文、简体中文、繁体中文、希伯来文等。需要在这些名字中间试验选择，直到网页上的内容看起来像是可以理解的文字为止。从这个菜单上列出的选项，可以对各种文字的编码有初步的认识：编码方案真是够多的！每种文字至少有一种编码。

这里不能对所有编码都进行介绍，只对汉字编码方案进行简要介绍。GB 2312 是我国在 1980 年颁布的第一个汉字编码国家标准。它的全称是《信息交换用汉字编码字符集 基本集》，它的标准号是 GB 2312—1980。在这个标准中，汉字以及一些序号、字母等被称为图形字符。GB 2312 一共对 7445 个图形字符进行了编码，其中包括一般符号、序号、数字、拉丁字母、日文假名、希腊字母、俄文字母、汉语拼音符号、汉语注音字母、汉字等。GB 2312 能够表示的汉字数量为 6763 个。每一个图形字符都用两字节表示。

GB 2312 定义了一个代码表，对其支持的 7445 个图形字符进行了编号。代码表分为

图 3-1　浏览器的“编码”选项

94 个区，每个区中有 94 个图形字符的位置，GB 2312 把这些位置称为位。所以，一个区中就有 94 个位。区的编号(区号)是 1～94，位的编号(位号)也是 1～94，所以一个图形字符的编码就可以用它的区号和位号表示了。区号加上位号也被称为区位码。例如，汉字“啊”在 16 区的第一位，所以它的区位码是 16-01，写的时候区号和位号用连字符相连。如果计算机上有区位码输入法，那么用的就是这个编码方案。

但是，GB 2312 并没有用区位码作为每个图形字符的最终编码。GB 2312 最终定义并使用的编码被称为国标码。国标码实际上是在区位码的基础上加上十六进制数 2020H。那么，最后计算机中存储汉字“啊”用的是哪个 0/1 串？是 16-01 换算成十六进制数加上 2020H 的和的二进制表示吗？非也！因为计算机系统中可能同时支持 ASCII 码和国标码。这时候就会带来一些麻烦。例如，两个字节单元的内容是 3021H 时，如果解释成国标码，那么表示的就是汉字“啊”(用区位码 16-01 计算一下看看对不对)；如果解释成 ASCII 码，那么就是字符'0'和'!'(查表 3-1 核实一下)。所以，为了解决这个问题，计算机中并不存储汉字的国标码，而是再把它加上十六进制数 8080H，这样就保证两个字节单元的最高位都是 1，也就能与 ASCII 码区分开了。最后存储在计算机中的这个编码称为机内码。

图 3-1 中的“简体中文(GB 18030)”是我国于 2000 年发布的国家标准，全称为《信息技术 中文编码字符集》，完全兼容 GB 2312。

重点提示

ASCII 码是使用最广泛的编码标准，它使用 7 位编码表示 128 个字符。
GB 2312 是我国为了能在计算机内存储和表示汉字，在 1980 年颁布的国家标准。