信息学奥赛笔记17-18

进制转换 计算机的码 位运算

进制转换

进制

古老的人类就已经开始有了数字的概念，最初，他们选择在绳子上打结来统计数量的多少，但是当数量累计到一定程度时，普通的绳结计数已经无法满足人类的需求。那如果能把多个结打包看成是1个结呢?比如：我们把12个鸡蛋放在一个盒子里可以称之为`一打`，那从数学的角度来看，原本计量为12个的东西因为换了计量名词，数量也随之发生改变，这只是称呼的不同，鸡蛋仍然是12个。

刚刚这种计数行为也可以称之为逢`12`进`1`。

生活中还有哪些这样的例子呢?

在我们钟表当中，秒针每动`60`次，就是`1`分钟，那么这个`1`分钟和`60`秒是等量的，我们也可以认为在秒和分的计量上是逢`60`进`1`

再具体的，在数学上，我们把逢几进一的这种计数行为称之为——**进制**

逢`12`进`1`则是一种`12`进制计量，逢`60`进`1`是一种`60`进制计量

在人类生活的普遍认知当中，我们日常生活使用的计量均为逢`10`进`1`，也就是`10`进制。

既然我们刚刚也说了，只是计量的名称变化了，实际上所表示的量是相同的，那么也就意味着，进制与进制之间可以互相转换，接下来我们就来说进制转换的方法。

10进制转K进制(整数部分)

把10进制转成k进制需要使用：列**竖式除法，倒序求余**

例如，我们需要将`10`进制的`123`，记为$123_{(10)}$转成`2`进制的数，我们需要列出竖式除法

$$
\begin{array}{llr}
2 & |\underline{123\quad} & 1 \\
2 & |\underline{\ 61\quad} & 1 \\
2 & |\underline{\ 30\quad\ } & 0 \\
2 & |\underline{\ 15\quad} & 1 \\
2 & |\underline{\quad 7\quad} & 1 \\
2 & |\underline{\quad 3\quad} & 1 \\
2 & |\underline{\quad 1\quad} & 1
\end{array}
$$

我们将获得的余数写在每次除法的右边，然后从下往上！！从下往上！！！从下往上！！！！！看余数。

所以结果是$1111011_{(2)}$

我们想转成`K`进制，则每次除`K`即可。

K进制转10进制(整数部分)

给你一个二进制的数$10101_{(2)}$，你也得会把它转成`10`进制。

我们采用的方法是，**逐位求幂法**(我是这么称呼的)。

第一步，我们列个表

数位	4	3	2	1	0	数位
二进制的值	1	0	1	0	1	具体的值
幂次系数积	1×2^4	0×2^3	1×2^2	0×2^1	1×2^0	$值\times K进制^{数位}$
具体的值	16	0	4	0	1	

这个数就是$16+4+1=21_{(10)}$。

那么，我们在算幂次系数积的时候，乘的数是该K进制的数位上的值，另外一个幂次的底数是K，具体几次方是当前的数位值。

再举一个例子

我们想把八进制的数$567_{(8)}$转成10进制数

数位	2	1	0
进制的值	5	6	7
幂次系数积	5×8^2	6×8^1	7×8^0
具体的值	320	48	7

所以这个数是$320+48+7=375_{(10)}$。

10进制转K进制(小数部分)

和整数部分的**竖式相除，倒序取余**相反，小数部分的做法是**竖式相乘，正序取整**

$0.65625_{(10)}$ = ($)_{(2)}$

$$
\begin{array}{r l}
0.65625 & \\
\times \quad\quad 2 & \\
\hline
1.3125 & 1 \\
\times \quad\quad 2 & \\
\hline
0.625 & 0 \\
\times \quad 2 & \\
\hline
1.25 & 1 \\
\times \quad 2 & \\
\hline
0.5 & 0 \\
\times \ 2 & \\
\hline
1 & 1
\end{array}
$$

每次乘上的数都是进制，并且，我们取出整数部分作为答案，整数将不再参与进下次乘法，比如第一步中的`1.3125`下次参与运算的只有`0.3125 × 2`。

从上往下取整数结果，结果为$(0.10101)_{(2)}$

K进制转10进制(小数部分)

该问题的做法和整数部分几乎没有区别，只不过我们需要从小数点断开继续画表求解。

$(11.10101)_{(2)} = (\quad)_{(10)}$

数位	1	0	小数点	-1	-2	-3	-4	-5
进制的值	1	1	小数点	1	0	1	0	1
幂次系数积	$1{\times}2^1$	$1{\times}2^0$	小数点	$1{\times}2^{-1}$	$0{\times}2^{-2}$	$1{\times}2^{-3}$	$0{\times}2^{-4}$	$1{\times}2^{-5}$
具体的值	2	1	小数点	0.5	0	0.125	0	0.03125

结果为$2+1+0.5+0.125+0.03125$ =$3.65625_{(10)}$

特别的，对于

$$a^{-b} = \frac{1}{a^b} = 1{\div}a^b$$

大于10进制

对于任何一个K进制来说，组成这些数的基本单元是`1` 到 `K - 1`这些数，那么比如说，`16`进制，我们需要`1`到`15`这些单独的数，但是`15`这个数本身它就是`10`进制体系下的一个进了位的数，不符合要求，所以我们用字母来代替，对于`16`进制当中由`A`替代`10`， `B`替代`11`...... `F`替代`15`。所以理论上来说，用26个字母加0-9这些数字能表述的最高为`36`进制，逢`Z`进`1`。

计算机的码

我们要搞明白一个问题，计算机是如何储存我们人类熟知的数值的，讲白了它只是一个由各种电路信号芯片组成的单元，是怎么样清楚现实世界人类的行为活动的，其实最根本来说都是由电信号组成的，而电路分为`闭合(1)`和`断开(0)`。所以冯·诺依曼提出，使用二进制来存储值。

字节与位

各位同学应该都听过字节(Byte)，一个字节能存储8个二进制位，我们称每个二进制位叫做**位**

1字节(B) = 8位

对于一个10进制数来说，我们需要把他转换成计算机能读懂的数值，也就是转换成二进制，但是各位同学可以发现，二进制是没办法表示负数的

为了解决这个问题，我们把一个字节的8个位进行了一个划分

`0000 0000`

其中最高位表示符号位，他代表了这个二进制数是正数还是负数，如果是0则是正数，如果是1则是负数。

比如说

`0000 0001`是十进制的1

`1000 0001`是十进制的-1

那么大家可以思考一个问题

`0000 0000` 和 `1000 0000`表示的数到底是不是相同的?

那么此时翻译过来则是一个正`0`，一个负`0`，这样是不利于我们计算的，因为同一个数有两个不同的定义。

为了统一规范，我们把`1000 0000` 定义为了最小的那个数——$(-128)_{(10)}$。

所以一个字节，8位能够表示的最大的数就是`0111 1111` ， 也就是$(127)_{(10)}$

能够表示的最小的数是`1000 0000`，也就是$(-128)_{(10)}$。

原码

原码(True Form)是一种计算机中对数字的二进制定点表示方法。

原码非常简单，我们直接把一个10进制的-128 ~ 127的数直接转成8位二进制的数，这就是原码

$(100)_{(10)}$的原码是`0110 0100`

$(-110)_{(10)}$的原码是`1110 1110`

反码

反码(Inverse Code)是数值存储的一种，多应用于系统环境设置，如linux平台的目录和文件的默认权限的设置umask，就是使用反码原理。

正数的反码就是原码

对于一个负数来说，它的反码就是原码除了符号位全部取反`1`变`0`， `0`变`1`。

$(-110)_{(10)}$的反码是`1001 0001`

补码

补码(Two's Complement)是计算机中算是最为重要的码了，因为补码主要是用于计算的，由于CPU内只有加法器，所以需要用加法来模拟出其他所有的减法乘法除法，而减法的过程我们看成是加上了一个负数，万物皆可加法。

正数的补码还是原码

负数的补码 = 负数的反码 + 1

$(-110)_{(10)}$的补码为`1001 0010`

我们现在尝试用二进制的100 + (-110) = -10

$$
\begin{array}{r}
0110\ 0100 \\
+\quad 1001\ 0010 \\
\hline
1111\ 0110
\end{array}
$$

这是结果的补码，现在我们要把这个数转换成它的原码，首先应该先求这个数的反码再转换成原码。

`1111 0110(补)` = `1111 0101(反) = 1000 1010(原)`

口算得出结果是-10，这就是$100-110$的答案。

如果不用补码能完成这一要求吗？

接下来我们使用原码计算：

$$
\begin{array}{r}
0110\ 0100 \\
+\quad 1110\ 1110 \\
\hline
11101\ 0010
\end{array}
$$

不仅最终得到的结果会溢出不谈，而且可能会导致8位二进制并不够存储，而且答案还是错的，这就体现出了补码存在的意义。

如果要计算$100+(-110)$。 那么根据原码来计算的话，必然会计算出$100+110$的结果，已经大于`8`位二进制能表示的`127`。

补码更相当于是我们把这个二进制反过来转，正数从`0000 0000`到`0000 0001`往正拨， 负数则是从`0000 0000`到`1000 0000往负拨`，然后正负相抵，这就是补码最初创建的缘由，根据这个理论，我们继续拓展

数据类型与码的关系

interger整型

我们在很早的时候就已经知道了C++的数据类型int。各位同学有没有想过为什么int的数据范围是`-2147483648`到`2147483647`，这样的一个数，而不是正好的一个整百整千的数字。这是因为一个int数，占用了4个字节。

大家想想，1B = 8位，它的弊端是什么？是不是没有办法存储大数，那如果我们的数字变大呢？我们可以用多个字节来存储一个数，4字节也就是32位来存储。

所以int的结构是

`0000 0000 0000 0000 0000 0000 0000 0000`这是int的0。因为int包含负数，所以我们定第一个数位是符号位。所以能表示数值的只有31个值。最大的值是

`0111 1111 1111 1111 1111 1111 1111 1111` 也就是 `2147483647`，$2^{31}-1$。

最小的值是`1000 0000 0000 0000 0000 0000 0000 0000`，也就是`-2147483648`， -2^{31}

long long int 长整型

$longlong$类型的变量所占用的字节是`8`个字节，所以一共有`64`个二进制位，最大能表示的数是$2^{63}-1$，`9223372036854775807`。

最小的数是-2^{63}，`-9223372036854775808`。

__int128型

计算机还有个数据类型是`__int128`，我们用的比较少，但在这老师还是提一下，这个数据类型的变量一个占用`16`字节

所以它一共有$16*8=128$个二进制位，它可以表示的最大数为$2^{127}-1$。

这种数据类型的变量不可以用于`cin` `cout`输入输出，只可以用于四则运算。

位运算

在先前学习到的选择结构中，我们已经了解到了逻辑运算符$\&\&$(逻辑与) $||$(逻辑或) !(非)。

他们可以用来判断条件的成立与否以及多条件判断，今天我们要来说几个运算符，既然是运算符，他们可以像加减乘除那样去直接对数做运算。

与普通的运算符不同的是，他们并不是在10进制上对数值进行运算的，而是转换成二进制后再进行**逐位**运算

& 与

两个数的&，就是先把两个数都转换成二进制，并且逐位的进行&运算，每次&有四种情况

$0\&0=0$

$0\&1=0$

$1\&0=0$

$1\&1=1$

总结就是，同为真时则为真，这与我们的逻辑表达式&&有异曲同工之妙，只不过一个比较的是逻辑式，一个比较的是具体的二进制数值

计算：4 & 7

也就是计算`100` & `111` 结果为100 也就是4，通常情况下，与运算大多情况下会抹去原有的`1`，所以会导致结果变小或不变。

| 或

$0|0=0$

$0|1=1$

$1|0=1$

$1|1=1$

总结：只要有一个为真则为真，这和||也是基本上一样的，也是换成了二进制比较

计算：5 | 2

也就是计算`101` | `010`,结果为`111`也就是`7`，因为或运算会带来新的二进制`1`，所以会导致结果变大或不变。

^ 异或

计算机中有一个非常特殊的运算符——异或。

它的定义规范是：相同为`0`，不同为`1`。

0 ^ 0 = 0

0 ^ 1 = 1

1 ^ 0 = 1

1 ^ 1 = 0

计算6 ^ 3

也就是计算`110` ^ `011` 结果为`101` 为`5`异或具有隐藏性，我们可以用刚刚的结果`5`再去异或`3`看看结果

答案是`6`也就是一开始的结果。

那么我们可以把`3`看作是密钥，`6`是原文，`5`是密文，我们可以通过用`6`^`3`将原文加密，再通过对密钥异或运算的方式来进行解密。

所以异或符号在计算机密码学中应用广泛，比较知名的有：RSA加密和MD5加密，都使用了异或来对原文进行掩盖。