

# 进制

重庆市育才中学·信息学讲义·赵康·2020年01月

## 一、进制的定义

进制也就是**进位制**，是人们规定的某种进位方法。对于任何一种进制（记为  $P$  进制）， $dd$  表示某一位置上的数运算时逢  $P$  进 1。

十进制是逢十进一，十六进制是逢十六进一，二进制就是逢二进一，以此类推。

生活中常用的进制有十进制、六十进制、千进制、万进制、1024进制、二进制、八进制、十六进制、三十六进制等。

其中**十进制**为人类的**本位进制**，就像学习语言时我们以中文为本位。

只要有需要，可以用任何进制来表示数。

## 二、进制的标记：

如果在问题中，涉及多种进制表示的数（比如进制转换），需要做标记，否则会出现表示混乱。例如，

$7(8) = 7(10)$  --- 8 进制表示的 7 与 10 进制表示的 7 相等

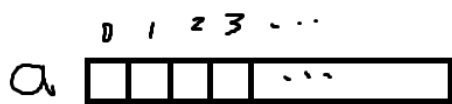
$83(10) = 123(8)$

括号中的数值表示进制的**基数**。

## 三、进制的转换：

我们来找规律：

○ -- 某个数  $X$   
 $\downarrow$   $P$ 进制



满 $P$ 进1, 位置 $3$ 上的数, 代表的值为其本身 $\times P^3$   
 (不足 $P$ 的留下)

满 $P$ 进1, 位置 $2$ 上的数, 代表的值为其本身 $\times P^2$   
 (不足 $P$ 的留下)

满 $P$ 进1, 位置 $1$ 上的数, 代表的值为其本身 $\times P$   
 (不足 $P$ 的留下)

按照以上数位分离方式, 任何数 $X$ 均可表示为:

$$X = a_0 \cdot P^0 + a_1 \cdot P^1 + a_2 \cdot P^2 + \dots + a_n \cdot P^n \quad (\text{直到数数值分配完毕})$$



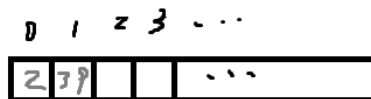
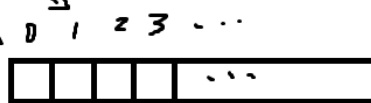
上式简记为:  $X = a_n a_{n-1} \dots a_0 (P)$



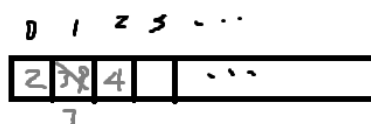
这不就是  $P$  进制的表示吗? 😊

举例:  $314(10)$  以  $8$  进制表示.

$$314 = \underbrace{314/8}_{\text{满8, 进位}} \times 8 + \underbrace{314 \% 8}_{\text{不满8, 留下}}$$



$\downarrow$  39 继续拆分



$$314(10) = 472(8)$$

总结规律：

1. 本位进制转  $P$  进制方法：不断对一个数  $X$  通过  $\%P$  留下不足  $P$  的数，并将  $X$  更新为剩下的数  $X/P$ ，直到  $X$  为 0（分完了）。（由于我们是由低位到高位进行分离，但人类的计数习惯是从高位到低位，所以最后对结果反序输出即可）
2.  $P$  进制转本位进制方法：（更加简单）将各数位代表的真实值累加即可。

随堂思考：

$P$  进制 转  $Q$  进制（非本位进制的两种进制互转）如何操作？

随堂练习：

1.  $123(10) = \_\_? \_\_(8)$
2.  $123(10) = \_\_? \_\_(2)$
3.  $666(3) = \_\_? \_\_(10)$
4.  $233(7) = \_\_? \_\_(5)$

## 四、一个漏洞

大家先做一个例题：

$$10000(10) = \_\_? \_\_(60)$$

你发新了什么问题？

你有什么办法解决这个问题？

（大家先思考，课上会讲解）

## 五、二进制

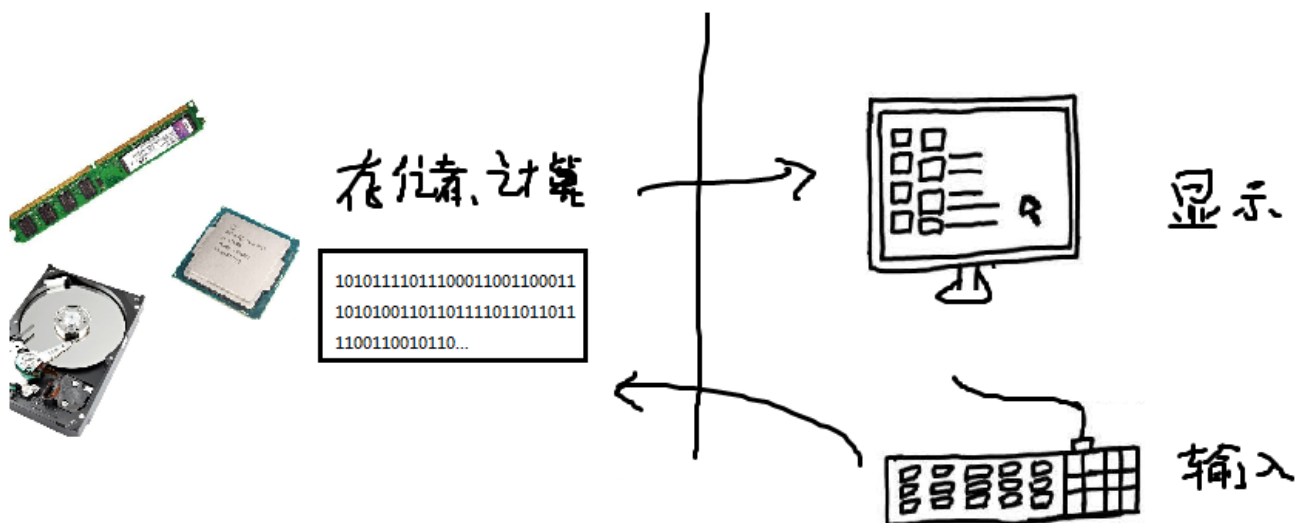
人类的本位进制是十进制，而计算机的本位进制是二进制

也许你会问：

我们使用的电脑的时候（比如编程时），不是直接用的十进制吗？

没错，但我们看到的数据是转换以后的结果。

也就是说计算机存储和计算的数据根本上是二进制数，显示时自动转为需要的进制表示。

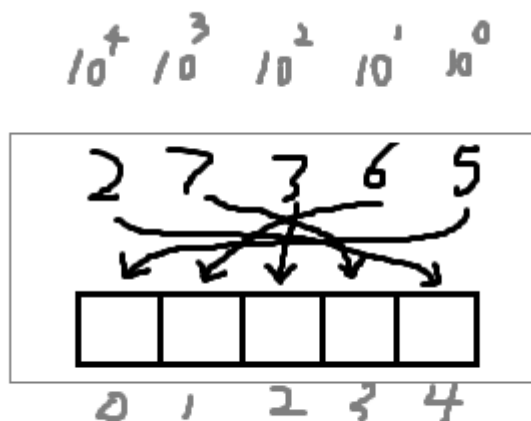


计算机采用二进制的主要原因如下：

1. 技术实现简单，逻辑电路的开关状态可以表示0或1；
2. 简化运算规则，适合逻辑真假运算且易于进制转换；
3. 抗干扰能力强，可靠性高，只有两个状态，受到干扰时容易分辨

## 知识延伸1. 高精度的万进制版本（不做要求）

之前学过高精度运算，第一步操作就是把数据按位拆分放到数组中



在计算过程中，数组每个位置不管累积了多大数，在处理进位之后，最终只留下一位数。通过这种方法，可以处理超大整数的基本运算；

我们知道 `int` 类型数组每个位置最大可存  $10^9$  级别的数；考虑做乘法时可能会有整数溢出，那么存  $10^4$  级别的数也是没有问题的。

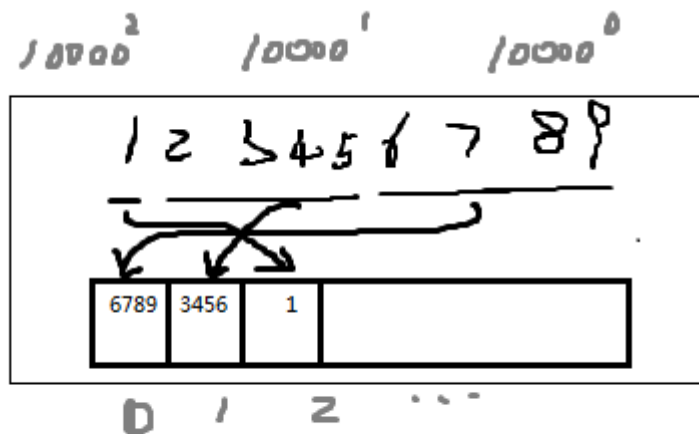
一个问题隐隐浮出：

数组每个位置只存1位数是不是有点浪费？

那我们让数组每个位置存4位数好不好？

好的！如下图

1 2 3 4 5 6 7 8 9  
亿 万  
读作：1 2 3 4 5 6 7 8 9



以上是以**万进制**方式做数据转换,优点是可以节省存储空间和运算时间，缺点是要考虑的细节较多。

下面是一个高精度加法的参考代码：

```
// 关键词：万进制，高精度加法
#include<bits/stdc++.h>
using namespace std;
int a[10010], b[10010];
int n1, n2;

void in1()
{
    string s1;
    cin >> s1;
    n1 = s1.size();
    for(int i = 0, j=n1-1; i < n1; i++,j--)
```

```

        a[j] = s1[i] - '0';
    for(int i = 0; i < n1; i+=4)
        a[i/4] = a[i] + a[i+1]*10 + a[i+2]*100 + a[i+3]*1000;
    for(int i = (n1-1)/4+1; i < n1; i++)
        a[i] = 0;
    n1 = ceil(n1/4.0); // ceil()函数对小数进行向上取整
}

void in2()
{
    string s2;
    cin >> s2;
    n2 = s2.size();
    for(int i = 0, j=n2-1; i < n2; i++,j--)
        b[j] = s2[i] - '0';
    for(int i = 0; i < n2; i+=4)
        b[i/4] = b[i] + b[i+1]*10 + b[i+2]*100 + b[i+3]*1000;
    for(int i = (n2-1)/4+1; i < n2; i++)
        b[i] = 0;
    n2 = ceil(n2/4.0);
}

void add()
{
    n1 = max(n1,n2);
    for(int i = 0; i < n1; i++)
    {
        a[i] += b[i];
        a[i+1] += a[i] / 10000;
        a[i] %= 10000;
    }
    if(a[n1]) n1++;
}

void out()
{
    for(int i = n1-1; i >= 0; i--)
    {
        if(i!=n1-1){
            if(a[i] < 1000) cout << 0;
            if(a[i] < 100) cout << 0;
            if(a[i] < 10) cout << 0;
        }
        cout << a[i];
    }
    cout << endl;
}

int main()
{
    in1();
    in2();
    add();
}

```

```
out();  
return 0;  
}
```

## 知识延伸2. 位运算

(下面给出一些常见操作, 课上会讲解, 但当前阶段不作要求)

1. 左移  $x \ll 1$  --- 相当于  $x * 2$
2. 右移  $x \gg 3$  --- 相当于  $x / 2 / 2 / 2$
3. 按位与  $3 \& 6$
4. 按位或  $3 | 6$
5. 按位异或  $6 \wedge 5$
6. 按位非  $\sim 6$

位运算有什么用呢?

1. 直接对二进制数操作, 比其他运算快;
2. 二进制数既可以直接加减, 又可以实现类似数组的**打标记**功能, 对于某些问题可以方便写出简洁高效的代码;
3. 二进制思想在很多高级算法和数据结构设计中有重要运用

我们来列举几个常见的需求:

检查第  $i$  位是否是 1

```
if(1 << (i-1) & x) ...
```

检查第  $i$  位是否是 0

```
if(1 << (i-1) & x == 0) ...
```

统计  $x$  中有多少个 1

```
while(x)  
{  
    cnt += x & 1;  
    x >>= 1;  
}
```

检查  $x$  中是否有相邻的 1

```
if(x & (x << 1)) ...
```

计算  $x$  最低位 1 代表的值

```
int lowbit(int x)
{
    return x & (-x); // 用到负数补码的性质，可以不深究
}
```

把第  $i$  位变成 1

```
x |= (1 << (i-1))
```

把第  $i$  位变成 0

```
x &= ~(1 << (i-1))
```

把第  $i$  位取反

```
x ^= (1 << (i-1))
```

$x$  包含  $y$

```
if(x & y == y) ...
    OR
if(x | y == x) ...
```

...