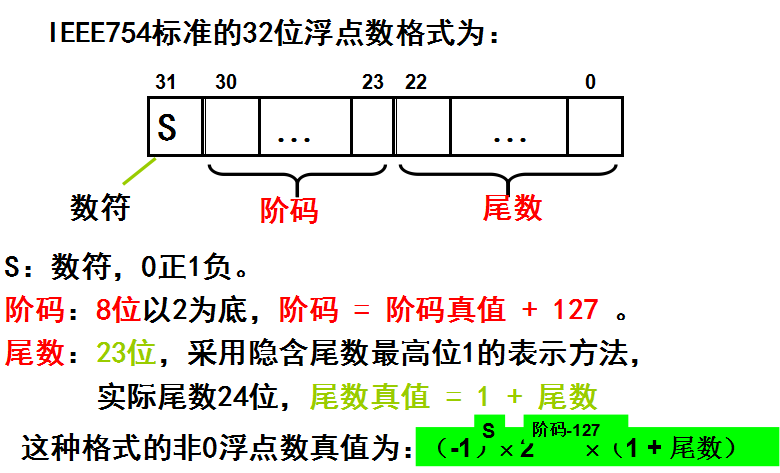
# 原因

在JavaScript中，数字是以IEEE 754 双精度64位浮点数来存储的。标准如下：（以32位为例）



一个数由**符号位s+阶码e+尾数m**来表：**(s) \* (m) \* (2 ^ e)**。这实际上是一种科学计数法。

我们都知道，由于浮点数位数有限，因此在表示精度较高的数时会发生截断误差。但是由于采用了IEEE754标准表示浮点数，即使在浮点数小数位数很低时，也会发生算术运算导致的误差。这就是为什么在js中0.1+0.2!=0.3.（这个问题也出现在除了js以外的很多其他语言中，如python等）。

这个问题的本质实际上就是：利用**二进制（2的幂之和）**无法准确地表示**十进制（10的幂之和）**浮点数。

按IEEE754标准，0.1表示为：

e = -4;

m = 1.1001100110011001100110011001100110011001100110011010 (52位)

0.2表示为

e = -3;

m = 1.1001100110011001100110011001100110011001100110011010 (52位)

二者相加

e = -4; m = 1.1001100110011001100110011001100110011001100110011010 (52位)

+

e = -3; m = 1.1001100110011001100110011001100110011001100110011010 (52位)

由于阶码（e）不同，统一转化为-3。

e = -3; m = 0.1100110011001100110011001100110011001100110011001101 (52位)

+

e = -3; m = 1.1001100110011001100110011001100110011001100110011010 (52位)

得到

e = -3; m = 10.0110011001100110011001100110011001100110011001100111 (52位)

保留一位整数

e = -2; m = 1.00110011001100110011001100110011001100110011001100111 (53位)

这个数的m部分超过了52位，要进行截断

取m=1.0011001100110011001100110011001100110011001100110100 （52位）

最终得到1.0011001100110011001100110011001100110011001100110100 \* 2 ^ -2

这个数的十进制为：

0.30000000000000004

这就是为什么js中0.1+0.2!=0.3的原因。

# 解决方案

可以通过指定精度的方式避免误差。但是这样不够优雅。

parseFloat((0.1+0.2).toFixed(1)) ===0.3

可以通过转为正数计算的方式避免误差。但是这样在小数位数不确定或者较大时失效。

0.1+0.2 => (0.1\*10+0.2\*10)/10

使用第三方专门库。

Math.js//这个是专门用来解决JS 计算精度问题的类库