浮点数在内存中是怎么存储的？

原创 展开

夜风~ 最后发布于2018-04-04 18:18:03 阅读数 8178 收藏

浮点数存储规则

根据国际标准IEEE（电气和电子工程协会）规定，任何一个浮点数NUM的二进制数可以写为：
NUM = (-1) ^ S * M * 2 ^ E;//(S表示符号，E表示阶乘，M表示有效数字)
①当S为0时，表示一个正数；当S为1时，表示一个负数
②M表示有效数字，1<= M <2
③2^E表示指数
比如十进制的3.0，二进制就是0011.0 就可以写成（-1）^ 0 * 1.1 * 2 ^ 1
在比如十进制的-3.0，二进制就是-0011.0 就可以写成（-1）^ 1 * 1.1 * 2 ^ 1
而规定float类型有一个符号位（S），有8个指数位（E），和23个有效数字位（M）
double类型有一个符号位（S），有11个指数位（E），和52个有效数字位（M）
以float类型为例：

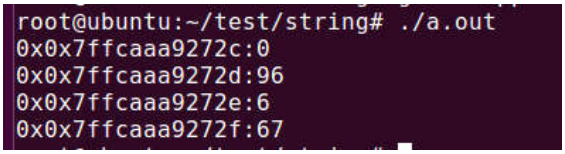
IEEE对于（有效数字）M和（指数）E有特殊的规定：（以float为例）
1.因为M的值一定是1<= M <2，所以它绝对可以写成1.xxxxxxx的形式，所以规定M在存储时舍去第一个1，只存储小数点之后的数字。这样做节省了空间，以float类型为例，就可以保存23位小数信息，加上舍去的1就可以用23位来表示24个有效的信息。
2.对于E（指数）E是一个无符号整数所以E的取值范围为（0~ 255），但是在计数中指数是可以为负的，所以规定在存入E时，在它原本的值上加上中间数（127），在使用时减去中间数（127），这样E的真正取值范围就成了（-127~128）。
对于E还分为三种情况：
①E不全为0，不全为1:
这时就用正常的计算规则，E的真实值就是E的字面值减去127（中间值），M的值要加上最前面的省去的1。
②E全为0
这时指数E等于1-127为真实值，M不在加上舍去的1，而是还原为0.xxxxxxxx小数。这样为了表示0，和一些很小的整数。
所以在进行浮点数与0的比较时，要注意。
③E全为1
当M全为0时，表示±无穷大（取决于符号位）；当M不全为1时，表示这数不是一个数（NaN）

测试

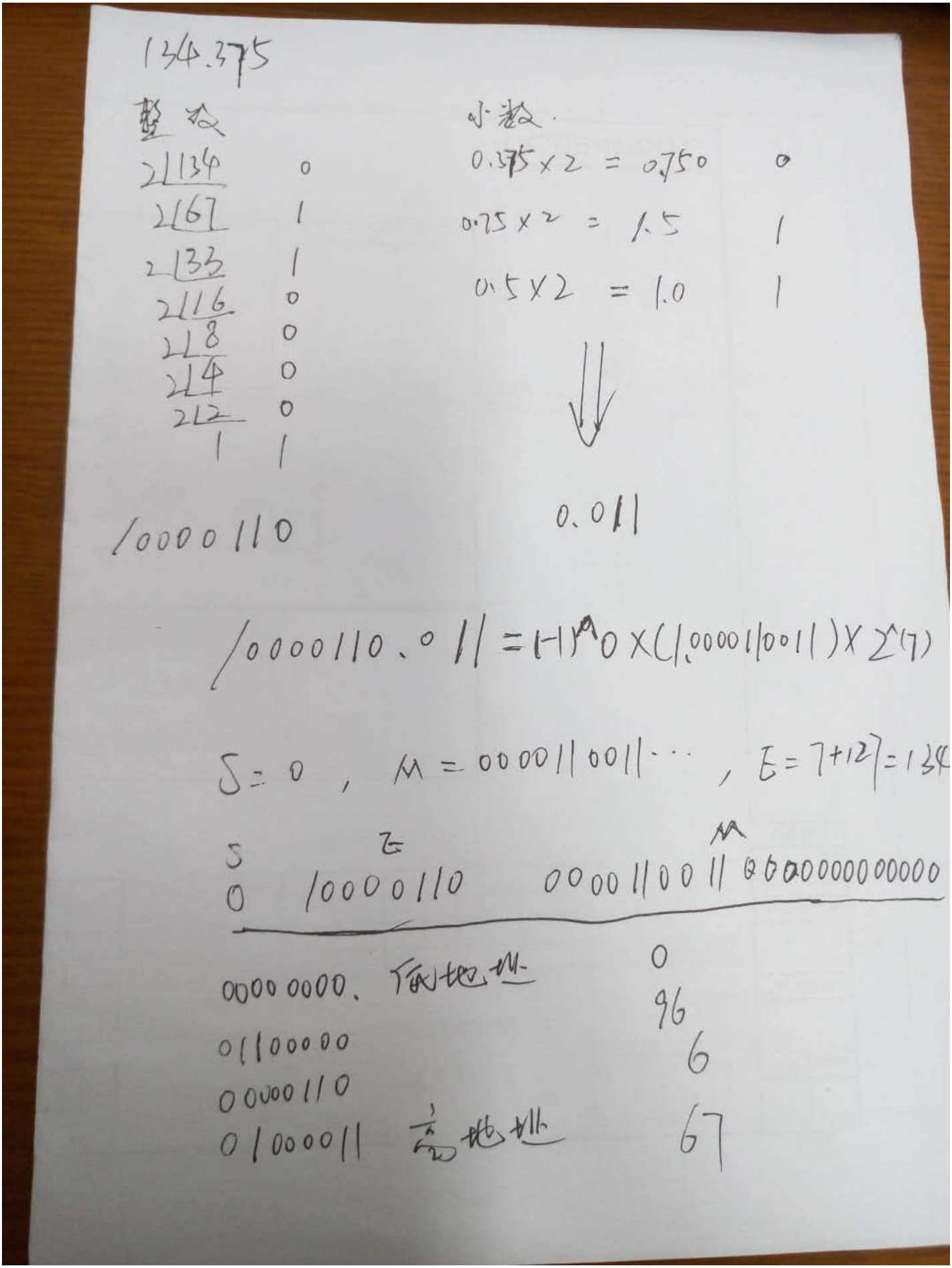
```
void test(void)
{
        float m=134.375;
        char *a=(char*)&m;

        printf("0x%p:%d\n",a,*a);
        printf("0x%p:%d\n",a+1,*(a+1) );
        printf("0x%p:%d\n",a+2,*(a+2) );
        printf("0x%p:%d\n",a+3,*(a+3) );
}
```

代码输出结果：


root@ubuntu:~/test/string# ./a.out
0x0x7ffcaaa9272c:0
0x0x7ffcaaa9272d:96
0x0x7ffcaaa9272e:6
0x0x7ffcaaa9272f:67

具体的计算过程如下：



精度损失

我们可以把十进制的小数部分乘以2，取整数部分作为二进制的一位，剩余小数继续乘以2，直至不存在剩余小数为止。
例如0.2可以转换为：

0.2 x 2 = 0.4 0

0.4 x 2 = 0.8 0

0.8 x 2 = 1.6 1

0.6 x 2 = 1.2 1

0.2 x 2 = 0.4 0

0.4 x 2 = 0.8 0

0.8 x 2 = 1.6 1

...

即：.0011001...
它是一个无限循环的二进制数，这就是为什么十进制小数转换成二进制小数的时候为什么会出现**精度损失**的情况了吗。

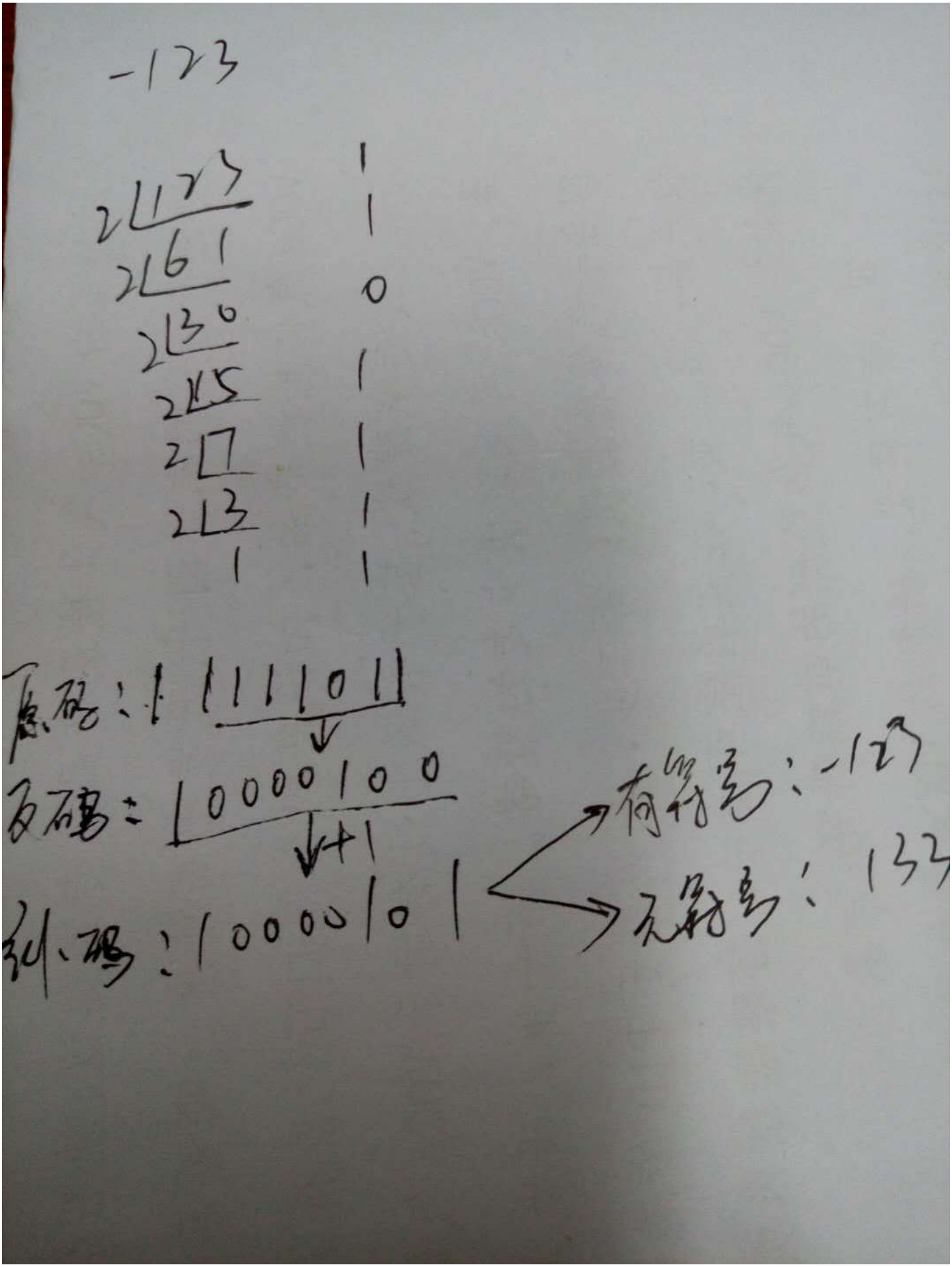
整数的存储规则

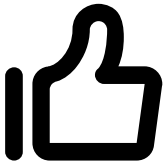
整数在内存中都是以补码的形式进行存储，整数有正负之分。当需存储有符号数时，用第一位来表示正（0）和负（1）。正数的反码和补码还是它本身，下面主要讨论下负数的反码和补码。反码是其原码除去最高符号位后其余位按位取反，补码是其反码在加上1 。
测试代码：

```
void test(void)
{
        int8_t n=-123;
        uint8_t *p=(uint8_t *)&n;

        printf("%d\n",n);
        printf("%d\n",*p);
}
```

输出结果：


-123
2|123 1
2|61 1
2|30 0
2|15 1
2|7 1
2|3 1
1 1
原码：1 1111011
反码：10000100
+1
补码：10000101
有符号：-123
无符号：133



点赞