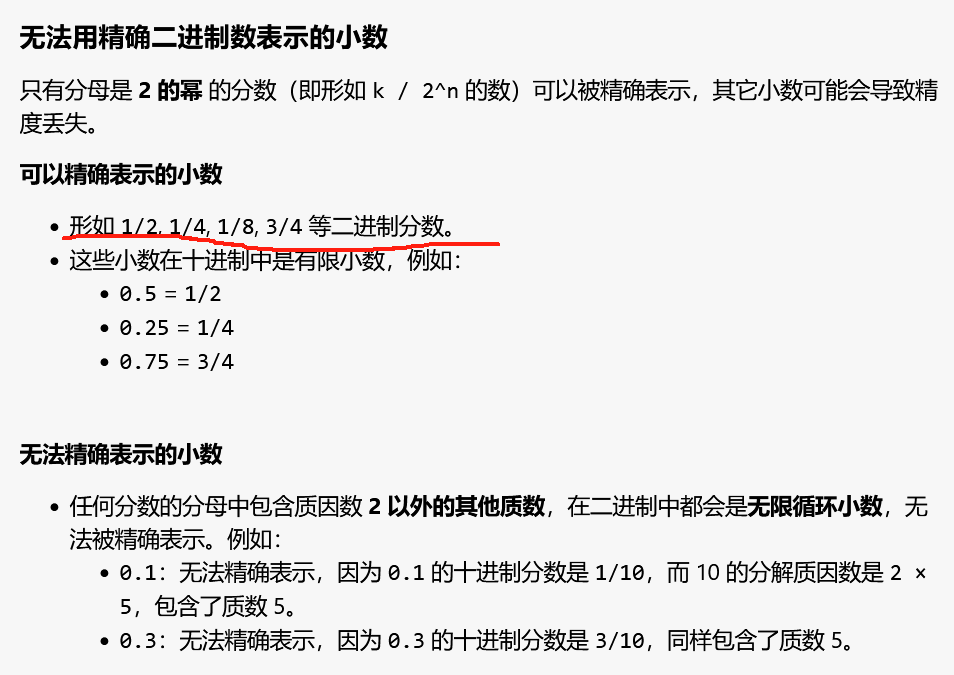
# 小数转二进制

<https://cloud.tencent.com/developer/article/2155741>

比如 十进制数字0.8125，我们发现 0.8125 \* 2 = 1.625，然后 1.625 \* 2 = 3.25，然后 3.25 \* 2 = 6.5，然后 6.5 \* 2 = 13，也就是 0.8125 \* 2 ^ 4 = 13，假设 0.8125 对应的二进制数字是 0.abcdef…，那么我们这个操作就是让这个数字左移4位，也就是 abcd.ef…. = 13，13的二进制我们知道是1101，并且13后面的小数都是0，因此abcd = 1101，所以0.8125 对应的二进制数字是0.1101。

上面的文章的算法，其实本质上就是我上面的这个过程，只不过每次把最高那一位给提取出来。可以看到，如果13这个数字后面还有小数，比如13.2，那么你还要重复\*2的过程，直到小数位归0，那么肯定有些数字是无法通过\*2的方式去消灭小数位的，比如下面说的这几个



因此，我们要直到，小数转二进制本身就存在一个误差，我们的计算机会通过某种方式去截断这个过程。

# 浮点数存储

<https://www.cnblogs.com/jillzhang/archive/2007/06/24/793901.html>

我们以一个数字为例子

21.235，先转成二进制小数，10101.0011110001 (二进制)，首先这里就已经有个转化误差了。

然后是，把所有数字都往右移，直到变成1.01010011110001，总共移动了4位，因此原来的式子表示为 1.01010011110001 \* 2^4，

A screenshot of a computer code

AI-generated content may be incorrect.

那么，符号位放第一个位，指数4放中间部分，最后的小数部分直接原封不动的放到后面，这就是表示办法，可以看到，随着你的数字越大，这个小数部分能表示的位数就会更少

A screenshot of a computer

AI-generated content may be incorrect.

这就是第二个误差，存储为浮点数的时候的误差。

# 不同硬件导致的误差

不同硬件可能对浮点数计算使用不同的临时精度。例如：

* 在一些 x86 架构的 CPU 上，浮点运算可能使用 80 位临时寄存器来计算中间结果。
* 在 ARM 架构（如移动设备）上，可能严格使用 64 位精度。

这种硬件差异会导致浮点数结果在客户端之间有所不同，从而破坏帧同步的核心假设。

在多线程或并行计算的环境中，线程的调度顺序可能不同，导致浮点数的计算顺序发生变化。例如：

* 多线程累加操作：不同线程可能以不同的顺序处理数据，最终结果可能略有差异。

**示例：**

假设有一组数 [1.1, 2.2, 3.3, 4.4]，需要在两个线程中并行累加：

* 客户端 A 的线程调度可能是 (1.1 + 2.2) 和 (3.3 + 4.4)，然后再计算总和。
* 客户端 B 的线程调度可能是 (1.1 + 3.3) 和 (2.2 + 4.4)，然后再计算总和。

由于浮点数的非交换性，结果可能不同。

## 为什么硬件实现不同？

硬件能够不经过软件直接计算浮点数，是通过专门的\*\*浮点运算单元（FPU, Floating Point Unit）\*\*来实现的。FPU 是现代处理器中负责执行浮点数运算的专用硬件模块。它能以硬件电路的形式直接处理浮点数的操作，包括加法、减法、乘法、除法、平方根等，而无需依赖软件的模拟运算

## 那么有办法解决这些误差么在帧同步的时候

1. 用软件库去模拟浮点数计算
2. 把浮点数计算提交给服务端来计算然后再下发
3. 严格控制浮点计算顺序，避免多线程导致的顺序问题