**[JS魔法堂：彻底理解0.1 + 0.2 === 0.30000000000000004的背后](https://www.cnblogs.com/fsjohnhuang/p/5115672.html)**

**Brief**

  一天有个朋友问我“JS中计算0.7 \* 180怎么会等于125.99999999998，坑也太多了吧！”那时我猜测是二进制表示数值时发生round-off error所导致，但并不清楚具体是如何导致，并且有什么方法去规避。于是用了3周时间静下心把这个问题搞懂，在学习的过程中还发现不仅0.7 \* 180==125.99999999998，还有以下的坑

  1. 著名的 0.1 + 0.2 === 0.30000000000000004

  2. 1000000000000000128 === 1000000000000000129

**IEEE 754 Floating-point**

  众所周知JS仅有Number这个数值类型，而Number采用的时IEEE 754 64位双精度浮点数编码。而浮点数表示方式具有以下特点：

  1. 浮点数可表示的值范围比同等位数的整数表示方式的值范围要大得多；

  2. 浮点数无法精确表示其值范围内的所有数值，而有符号和无符号整数则是精确表示其值范围内的每个数值；

  3. 浮点数只能精确表示m\*2e的数值；

  4. 当biased-exponent为2e-1-1时，浮点数能精确表示该范围内的各整数值；

  5. 当biased-exponent不为2e-1-1时，浮点数不能精确表示该范围内的各整数值。

  由于部分数值无法精确表示（存储），于是在运算统计后偏差会愈见明显。

  想了解更多浮点数的知识可参考以下文章：

[基础野：细说原码、反码和补码](http://www.cnblogs.com/fsjohnhuang/p/5060242.html)（http://www.cnblogs.com/fsjohnhuang/p/5060242.html）

[基础野：细说无符号整数](http://www.cnblogs.com/fsjohnhuang/p/5078290.html)（http://www.cnblogs.com/fsjohnhuang/p/5078290.html）

[基础野：细说有符号整数](http://www.cnblogs.com/fsjohnhuang/p/5082829.html)（http://www.cnblogs.com/fsjohnhuang/p/5082829.html）

[基础野：细说浮点数](http://www.cnblogs.com/fsjohnhuang/p/5109766.html)（http://www.cnblogs.com/fsjohnhuang/p/5109766.html）

**Why 0.1 + 0.2 === 0.30000000000000004 ?**

  在浮点数运算中产生误差值的示例中，最出名应该是0.1 + 0.2 === 0.30000000000000004了，到底有多有名？看看这个网站就知道了[http://0.30000000000000004.com/](http://0.30000000000000004.com/" \t "_blank)。也就是说不仅是JavaScript会产生这种问题，只要是采用IEEE 754 Floating-point的浮点数编码方式来表示浮点数时，则会产生这类问题。下面我们来分析整个运算过程。

  1. 0.1 的二进制表示为 1.1001100110011001100110011001100110011001100110011001 1(0011)+ \* 2^-4；

  2. 当64bit的存储空间无法存储完整的无限循环小数，而IEEE 754 Floating-point采用round to nearest, tie to even的舍入模式，因此0.1实际存储时的位模式是0-01111111011-1001100110011001100110011001100110011001100110011010；

  3. 0.2 的二进制表示为 1.1001100110011001100110011001100110011001100110011001 1(0011)+ \* 2^-3；

  4. 当64bit的存储空间无法存储完整的无限循环小数，而IEEE 754 Floating-point采用round to nearest, tie to even的舍入模式，因此0.2实际存储时的位模式是0-01111111100-1001100110011001100110011001100110011001100110011010；

  5. 实际存储的位模式作为操作数进行浮点数加法，得到 0-01111111101-0011001100110011001100110011001100110011001100110100。转换为十进制即为0.30000000000000004。

**Why 0.7 \* 180===125.99999999998 ?**

  1. 0.7实际存储时的位模式是0-01111111110-0110011001100110011001100110011001100110011001100110；

  2. 180实际存储时的位模式是0-10000000110-0110100000000000000000000000000000000000000000000000；

  3. 实际存储的位模式作为操作数进行浮点数乘法，得到0-10000000101-1111011111111111111111111111111111111111101010000001。转换为十进制即为125.99999999998。

**Why 1000000000000000128 === 1000000000000000129 ?**

  1. 1000000000000000128实际存储时的位模式是0-10000111010-1011110000010110110101100111010011101100100000000001；

  2. 1000000000000000129实际存储时的位模式是0-10000111010-1011110000010110110101100111010011101100100000000001；

  3. 因此1000000000000000128和1000000000000000129的实际存储的位模式是一样的。

**Solution**

  到这里我们都理解只要采取IEEE 754 FP的浮点数编码的语言均会出现上述问题，只是它们的标准类库已经为我们提供了解决方案而已。而JS呢？显然没有。坏处自然是掉坑了，而好处恰恰也是掉坑了:)

  针对不同的应用需求，我们有不同的实现方式。

***Solution 0x00 - Simple implementation***

    对于小数和小整数的简单运算可用如下方式

[复制代码](javascript:void(0);)

function numAdd(num1/\*:String\*/, num2/\*:String\*/) {

var baseNum, baseNum1, baseNum2;

try {

baseNum1 = num1.split(".")[1].length;

} catch (e) {

baseNum1 = 0;

}

try {

baseNum2 = num2.split(".")[1].length;

} catch (e) {

baseNum2 = 0;

}

baseNum = Math.pow(10, Math.max(baseNum1, baseNum2));

return (num1 \* baseNum + num2 \* baseNum) / baseNum;

};