519030910040 许灏瑄

在JavaScript乃至许多语言中,我们都会发现 $0.1 + 0.2 \neq 0.3$ 这个问题,接下来我将从小数的二进制表示及存储两个角度出发来尝试回答这个问题。

二进制小数表示

计算机是二进制的世界,一切数据、命令等均以二进制的形势存在,小数也不例外。但是小数的二进制表示法只能表示那些能够被写成 $x \times 2^y$ 的数,其他的值只能够被近似表示。以本问题中的0.1,0.2,0.3为例。它们在二进制下均无法用有限的位数表示,只能是通过1100的循环来不断逼近。(具体表示如下)

 $\begin{aligned} 0.1_{10} &= 0.000\dot{1}10\dot{0}_2\\ 0.2_{10} &= 0.00\dot{1}10\dot{0}_2\\ 0.3_{10} &= 0.0100\dot{1}10\dot{0}_2 \end{aligned}$

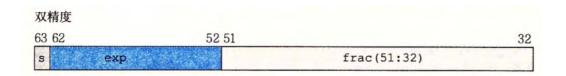
Javascript浮点数的存储

JavaScript的Number类型为双精度IEEE 754 64位浮点类型

--MDN

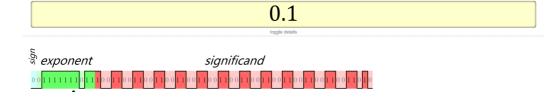
如前文所述,部分二进制小数无法用有限的位数表示,因此现实中IEEE就在上世纪80年代制定出一套浮点数的计算标准IEEE754,定义了浮点数的算术格式、交换格式、舍入规则、操作和异常处理,以期望通过更小和更快的系统表示范围更大和精度更高的实数。而在转换中,由于位数的有限,我们也毫无疑问会损失一些精度,这也就导致了我们最后的问题。

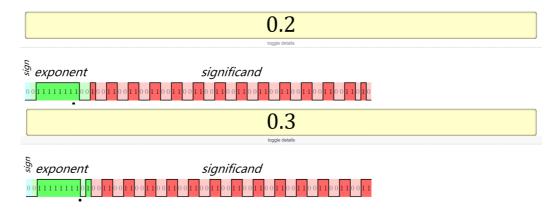
IEEE754中,64位浮点类型存储格式如下——由1位符号位,11位阶码字段,52位尾数字段组成。



根据阶码,被编码的值又可以分成三种情况——规格化、非规格化、特殊值。而0.1,0.2,0.3均是规格化的情况,因此我们今天只讨论规格化值的表示。规格化编码方式中,阶码值由其字段值减去1023的Bias得到,而尾数字段被解释位小数值,也即 $0.f_{51}f_{50}\dots f_0$,其尾数真实值还需加上1。

在了解了64位浮点类型的具体表示方法后,我们便可以直接写出0.1,0.2,0.3的64位表示。





之后再根据IEEE754规定的浮点数计算法进行求和便可知其和于0.3的二进制为表示并不相同,而运算法则较为复杂,因而不再展开。而不相等的直接原因便是在转换为二进制表示时由于尾数字段位数的有限会损失精度。

解决方案

如前文中所述,很多十进制有限小数转换到二进制后变为无限小数,这与有限的尾数阶 段尾数共同作用导致浮点数的不精确性。因此我们可以考虑构造一个专门的类来避免其不精 确性,如下。

1. 十进制小数Decimal类

大多数编程语言都原生支持或开源社区提供了十进制小数类。以Java为例,其 BigDecimal类如下。

```
public class BigDecimal extends Number implements
Comparable<BigDecimal> {
    private BigInteger intVal;
    private int scale;
    private int precision = 0;
    ...
}
```

以1234.56为例,我们来解释该类个属性的意义。

intval: 去除小数点后的所有数字 -- 123456
 scale: 小数的位数 -- 2
 precision: 全部的有效位数 -- 6

precision. Then, 11/4/EW

假如BigInteger用数组存储时,我们存储的精读将会变得近乎无限大。

2. 有理数类

存储的另一个思路便是用有理数来存储,分数可以精确地表示我们所用到的数。同样以 Java为例,其 Rational 类如下。

```
public class Rational implements Comparable<Rational> {
   private int num; // the numerator
   private int den; // the denominator
   public double toDouble() {
      return (double) num / den;
   }
   ...
}
```

其num, den分别为有理数的分子与分母,较为简单,也不再赘述。

以上两个类虽然可以很好的消除精度的问题,但带来了性能的问题,运算速率大大降低。因此,在实际应用中我们需要做好精度和性能两方面的trade-off来选择合理的实现方式。

References:

- [1] 为什么 0.1 + 0.2 = 0.300000004 面向信仰编程 (draveness.me)
- [2] IEEE-754 Floating Point representation explained (bartaz.github.io)