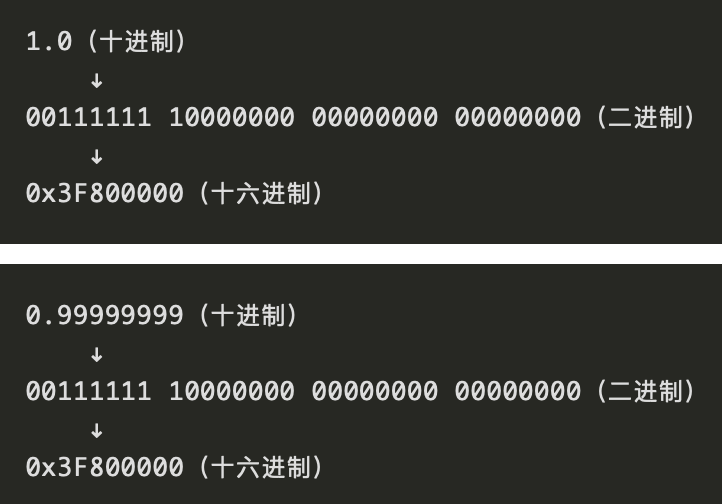
使用诸如float或者double这两种浮点型数据的处理时，典型现象。

1. 条件判断超预期
2. 数据转换超预期
3. 基本运算超预期
4. 数据自增超预期

System.*out*.println(1f == 0.99999999f);

打印出来true！！！

计算机底层在计算时可不是按照十进制来计算的，学过基本计组原理的都知道，计算机底层最终都是基于像010100100100110011011这种0、1二进制来完成的。把它转换到IEEE 754 Single precision 32-bit，也就float类型对应的精度。



很明显，它俩的二进制数字表示确实不一样，这是理所应当的结果。那么为什么0.99999999的底层二进制表示竟然是：00111111 10000000 00000000 00000000呢？这不明明是浮点数1.0的二进制表示吗？这就要谈一下浮点数的精度问题了。

### 浮点数的精度问题

浮点数在计算机中的存储方式遵循IEEE 754浮点数计数标准，可以用科学计数法表示为：

1.M...x2E

只要给出：符号（S）、阶码部分（E）、尾数部分（M） 这三个维度的信息，一个浮点数的表示就完全确定下来了，所以float和double这两种浮点数在内存中的存储结构如下所示：





1、符号部分（S）

0-正 1-负

2、阶码部分（E）（指数部分）：

对于float型浮点数，指数部分8位，考虑可正可负，因此可以表示的指数范围为-127 ~ 128

对于double型浮点数，指数部分11位，考虑可正可负，因此可以表示的指数范围为-1023 ~ 1024

3、尾数部分（M）：

浮点数的精度是由尾数的位数来决定的：

对于float型浮点数，尾数部分23位，换算成十进制就是 2^23=8388608，所以十进制精度只有6 ~ 7位；

对于double型浮点数，尾数部分52位，换算成十进制就是 2^52 = 4503599627370496，所以十进制精度只有15 ~ 16位

所以对于上面的数值0.99999999f，很明显已经超过了float型浮点数据的精度范围，出问题也是在所难免的。

### 精度问题如何解决

所以如果涉及商品金额、交易值、货币计算等这种对精度要求很高的场景该怎么办呢？

方法一：用字符串或者数组解决多位数问题

校招刷过算法题的小伙伴们应该都知道，用字符串或者数组表示大数是一个典型的解题思路。

比如经典面试题：编写两个任意位数大数的加法、减法、乘法等运算。

这时候我们我们可以用字符串或者数组来表示这种大数，然后按照四则运算的规则来手动模拟出具体计算过程，中间还需要考虑各种诸如：进位、借位、符号等等问题的处理，确实十分复杂，本文不做赘述。

方法二：Java的大数类是个好东西

JDK早已为我们考虑到了浮点数的计算精度问题，因此提供了专用于高精度数值计算的大数类来方便我们使用。

Java的大数类位于java.math包下：

常用的BigInteger 和 BigDecimal，像BigInteger 和 BigDecimal这种大数类的运算效率肯定是不如原生类型效率高，代价还是比较昂贵的，是否选用需要根据实际场景来评估。

### 1、BigDecimal初始化

对比了两种形式，第一种直接value写数字的值，第二种用string来表示：

BigDecimal num1 = new BigDecimal(0.005);

BigDecimal num2 = new BigDecimal(1000000);

//尽量用字符串的形式初始化  
BigDecimal num12 = new BigDecimal("0.005");  
BigDecimal num22 = new BigDecimal("1000000");

禁止使用构造方法BigDecimal(double)的方式把double值转化为BigDecimal对象。（说明：反编译出的字节码文件显示每次循环都会new出一个StringBuilder对象，然后进行append操作，最后通过toString方法返回String对象，造成内存资源浪费。）

### 2、加减乘除

// 1.加法

BigDecimal result1 = num1.add(num2);  
BigDecimal result12 = num12.add(num22);  
  
// 2.减法  
BigDecimal result2 = num1.subtract(num2);  
BigDecimal result22 = num12.subtract(num22);  
  
// 3.乘法  
BigDecimal result3 = num1.multiply(num2);  
BigDecimal result32 = num12.multiply(num22);  
  
// 4.绝对值  
BigDecimal result4 = num3.abs();  
BigDecimal result42 = num32.abs();  
  
// 5.除法  
BigDecimal result5 = num2.divide(num1,20,BigDecimal.*ROUND\_HALF\_UP*);  
BigDecimal result52 = num22.divide(num12,20,BigDecimal.*ROUND\_HALF\_UP*);

### 3、注意点

使用BigDecimal类构造方法传入double类型时，计算的结果也是不精确的！

因为不是所有的浮点数都能够被精确的表示成一个double 类型值，有些浮点数值不能够被精确的表示成 double 类型值，因此它会被表示成与它最接近的 double 类型的值。必须改用传入String的构造方法。

### 4、除法divide()参数使用

使用除法函数的时候要设置各种参数，要精确的小数位和摄入模式，不然会出现报错：

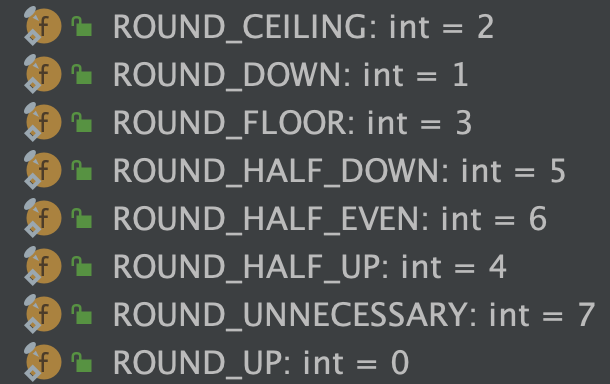
public [BigDecimal](file:///Users/chance/Library/Application%20Support/Dash/DocSets/Java_SE8/Java.docset/Contents/Resources/Documents/java/math/BigDecimal.html" \o "class in java.math) divide([BigDecimal](file:///Users/chance/Library/Application%20Support/Dash/DocSets/Java_SE8/Java.docset/Contents/Resources/Documents/java/math/BigDecimal.html" \o "class in java.math) divisor,

int scale,

[RoundingMode](file:///Users/chance/Library/Application%20Support/Dash/DocSets/Java_SE8/Java.docset/Contents/Resources/Documents/java/math/RoundingMode.html" \o "enum in java.math) roundingMode)

除数，精确小数位，舍入模式

### 5、8种舍入模式



#### 5.1 ROUND\_UP

舍入远离零的舍入模式。

在丢弃非零部分之前始终增加数字(始终对非零舍弃部分前面的数字加1)。

注意，此舍入模式始终不会减少计算值的大小。

#### 5.2 ROUND\_DOWN

接近零的舍入模式。

在丢弃某部分之前始终不增加数字(从不对舍弃部分前面的数字加1，即截短)。

注意，此舍入模式始终不会增加计算值的大小。

#### 5.3 ROUND\_CEILING

接近正无穷大的舍入模式。

如果 BigDecimal 为正，则舍入行为与 ROUND\_UP 相同;

如果为负，则舍入行为与 ROUND\_DOWN 相同。

注意，此舍入模式始终不会减少计算值。

#### 5.4 ROUND\_FLOOR

接近负无穷大的舍入模式。

如果 BigDecimal 为正，则舍入行为与 ROUND\_DOWN 相同;

如果为负，则舍入行为与 ROUND\_UP 相同。

注意，此舍入模式始终不会增加计算值。

#### 5.5 ROUND\_HALF\_UP

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为向上舍入的舍入模式。

如果舍弃部分 >= 0.5，则舍入行为与 ROUND\_UP 相同;否则舍入行为与 ROUND\_DOWN 相同。

注意，这是我们大多数人在小学时就学过的舍入模式(四舍五入)。

#### 5.6 ROUND\_HALF\_DOWN

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为上舍入的舍入模式。

如果舍弃部分 > 0.5，则舍入行为与 ROUND\_UP 相同;否则舍入行为与 ROUND\_DOWN 相同(五舍六入)。

#### 5.7 ROUND\_HALF\_EVEN

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。

如果舍弃部分左边的数字为奇数，则舍入行为与 ROUND\_HALF\_UP 相同;

如果为偶数，则舍入行为与 ROUND\_HALF\_DOWN 相同。

注意，在重复进行一系列计算时，此舍入模式可以将累加错误减到最小。

此舍入模式也称为“银行家舍入法”，主要在美国使用。四舍六入，五分两种情况。

如果前一位为奇数，则入位，否则舍去。

以下例子为保留小数点1位，那么这种舍入方式下的结果。

1.15>1.2 1.25>1.2

#### 5.8 ROUND\_UNNECESSARY

断言请求的操作具有精确的结果，因此不需要舍入。

如果对获得精确结果的操作指定此舍入模式，则抛出ArithmeticException。