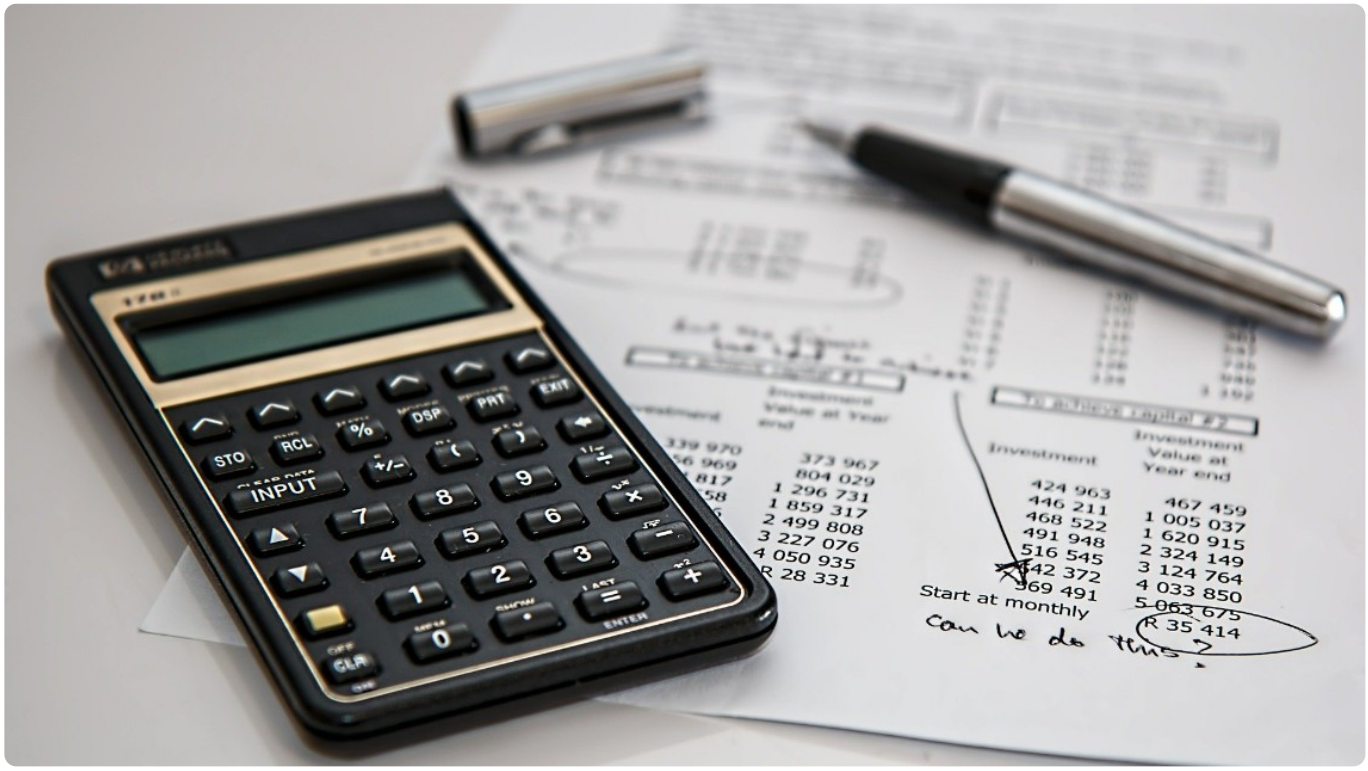
09 | 数值计算：注意精度、舍入和溢出问题

2020-03-28 朱晔



讲述：王少泽

时长 14:42　大小 13.48M

你好，我是朱晔。今天，我要和你说说数值计算的精度、舍入和溢出问题。

之所以要单独分享数值计算，是因为很多时候我们习惯的或者说认为理所当然的计算，在计算器或计算机看来并不是那么回事儿。就比如前段时间爆出的一条新闻，说是手机计算器把 10%+10% 算成了 0.11 而不是 0.2。

出现这种问题的原因在于，国外的计算程序使用的是单步计算法。在单步计算法中，a+b% 代表的是 a*(1+b%)。所以，手机计算器计算 10%+10% 时，其实计算的是 10%*（1+10%），所以得到的是 0.11 而不是 0.2。

在我看来，计算器或计算机会得到反直觉的计算结果的原因，可以归结为：

在人看来，浮点数只是具有小数点的数字，0.1 和 1 都是一样精确的数字。但，计算机其实无法精确保存浮点数，因此浮点数的计算结果也不可能精确。

在人看来，一个超大的数字只是位数多一点而已，多写几个 1 并不会让大脑死机。但，计算机是把数值保存在了变量中，不同类型的数值变量能保存的数值范围不同，当数值超过类型能表达的数值上限则会发生溢出问题。

接下来，我们就具体看看这些问题吧。

“危险”的 Double

我们先从简单的反直觉的四则运算看起。对几个简单的浮点数进行加减乘除运算：

复制代码

```
System.out.println(0.1+0.2);
System.out.println(1.0-0.8);
System.out.println(4.015*100);
System.out.println(123.3/100);

double amount1 = 2.15;
double amount2 = 1.10;
if (amount1 - amount2 == 1.05)
    System.out.println("OK");
```

输出结果如下：

复制代码

```
0.30000000000000004
0.19999999999999996
401.49999999999994
1.2329999999999999
```

可以看到，输出结果和我们预期的很不一样。比如，0.1+0.2 输出的不是 0.3 而是 0.30000000000000004；再比如，对 2.15-1.10 和 1.05 判等，结果判等不成立。

出现这种问题的主要原因是，计算机是以二进制存储数值的，浮点数也不例外。Java 采用了IEEE 754 标准实现浮点数的表达和运算，你可以通过这里查看数值转化为二进制的结果。

比如，0.1 的二进制表示为 0.0 0011 0011 0011…（0011 无限循环），再转换为十进制就是 0.1000000000000000055511151231257827021181583404541015625。**对于计算机而言，0.1 无法精确表达，这是浮点数计算造成精度损失的根源。**

你可能会说，以 0.1 为例，其十进制和二进制间转换后相差非常小，不会对计算产生什么影响。但，所谓积土成山，如果大量使用 double 来作大量的金钱计算，最终损失的精度就是大量的资金出入。比如，每天有一百万次交易，每次交易都差一分钱，一个月下来就差 30 万。这就不是小事儿了。那，如何解决这个问题呢?

我们大都听说过 BigDecimal 类型，浮点数精确表达和运算的场景，一定要使用这个类型。不过，在使用 BigDecimal 时有几个坑需要避开。我们用 BigDecimal 把之前的四则运算改一下：

复制代码

```
System.out.println(new BigDecimal(0.1).add(new BigDecimal(0.2)));
System.out.println(new BigDecimal(1.0).subtract(new BigDecimal(0.8)));
System.out.println(new BigDecimal(4.015).multiply(new BigDecimal(100)));
System.out.println(new BigDecimal(123.3).divide(new BigDecimal(100)));
```

输出如下：

复制代码

```
0.3000000000000000016653345369377348106354475021362304687 5
0.1999999999999999955591079014993738383054733276367187 5
401.4999999999999680255768907954916357994079589843750 0
1.2329999999999999715782905695959925655155029296875
```

可以看到，运算结果还是不精确，只不过是精度高了而已。这里给出浮点数运算避坑第一原则：**使用 BigDecimal 表示和计算浮点数，且务必使用字符串的构造方法来初始化 BigDecimal**：

复制代码

```
System.out.println(new BigDecimal("0.1").add(new BigDecimal("0.2")));
System.out.println(new BigDecimal("1.0").subtract(new BigDecimal("0.8")));
System.out.println(new BigDecimal("4.015").multiply(new BigDecimal("100")));
System.out.println(new BigDecimal("123.3").divide(new BigDecimal("100")));
```

改进后，就能得到我们想要的输出了：

```
0.3
0.2
401.500
1.233
```

到这里，你可能会继续问，不能调用 BigDecimal 传入 Double 的构造方法，但手头只有一个 Double，如何转换为精确表达的 BigDecimal 呢?

我们试试用 Double.toString 把 double 转换为字符串，看看行不行?

```
System.out.println(new BigDecimal("4.015").multiply(new BigDecimal(Double.toSt
```

输出为 401.5000。与上面字符串初始化 100 和 4.015 相乘得到的结果 401.500 相比，这里为什么多了 1 个 0 呢？原因就是，BigDecimal 有 scale 和 precision 的概念，scale 表示小数点右边的位数，而 precision 表示精度，也就是有效数字的长度。

调试一下可以发现，new BigDecimal(Double.toString(100)) 得到的 BigDecimal 的 scale=1、precision=4；而 new BigDecimal(“100”) 得到的 BigDecimal 的 scale=0、precision=3。对于 BigDecimal 乘法操作，返回值的 scale 是两个数的 scale 相加。所以，初始化 100 的两种不同方式，导致最后结果的 scale 分别是 4 和 3。

如果一定要用 Double 来初始化 BigDecimal 的话，可以使用 BigDecimal.valueOf 方法，以确保其表现和字符串形式的构造方法一致，这也是官方文档更推荐的方式：

```
System.out.println(new BigDecimal("4.015").multiply(BigDecimal.valueOf(100)));
```

BigDecimal 的 toString 方法得到的字符串和 scale 相关，又会引出了另一个问题：对于浮点数的字符串形式输出和格式化，我们应该考虑显式进行，通过格式化表达式或格式化工具来明确小数位数和舍入方式。接下来，我们就聊聊浮点数舍入和格式化。

考虑浮点数舍入和格式化的方式

除了使用 Double 保存浮点数可能带来精度问题外，更匪夷所思的是这种精度问题，加上 String.format 的格式化舍入方式，可能得到让人摸不着头脑的结果。

我们看一个例子吧。首先用 double 和 float 初始化两个 3.35 的浮点数，然后通过 String.format 使用 %.1f 来格式化这 2 个数字：

复制代码

```
1 double num1 = 3.35;
2 float num2 = 3.35f;
3 System.out.println(String.format("%.1f", num1));//四舍五入
4 System.out.println(String.format("%.1f", num2));
```

得到的结果居然是 3.4 和 3.3。

这就是由精度问题和舍入方式共同导致的，double 和 float 的 3.35 其实相当于 3.350xxx 和 3.349xxx：

复制代码

```
1 3.350000000000000088817841970012523233890533447265625
2 3.349999904632568359375
```

String.format 采用四舍五入的方式进行舍入，取 1 位小数，double 的 3.350 四舍五入为 3.4，而 float 的 3.349 四舍五入为 3.3。

我们看一下 Formatter 类的相关源码，可以发现使用的舍入模式是 HALF_UP（代码第 11 行）：

复制代码

```
1 else if (c == Conversion.DECIMAL_FLOAT) {
2     // Create a new BigDecimal with the desired precision.
3     int prec = (precision == -1 ? 6 : precision);
4     int scale = value.scale();
5
6     if (scale > prec) {
7         // more "scale" digits than the requested "precision"
8         int compPrec = value.precision();
```

```
        if (compPrec <= scale) {
            // case of 0.xxxxxx
            value = value.setScale(prec, RoundingMode.HALF_UP);
        } else {
            compPrec -= (scale - prec);
            value = new BigDecimal(value.unscaledValue(),
                                   scale,
                                   new MathContext(compPrec));
        }
    }
```

如果我们希望使用其他舍入方式来格式化字符串的话，可以设置 DecimalFormat，如下代码所示：

复制代码

```
double num1 = 3.35;
float num2 = 3.35f;
DecimalFormat format = new DecimalFormat("#.##");
format.setRoundingMode(RoundingMode.DOWN);
System.out.println(format.format(num1));
format.setRoundingMode(RoundingMode.DOWN);
System.out.println(format.format(num2));
```

当我们把这 2 个浮点数向下舍入取 2 位小数时，输出分别是 3.35 和 3.34，还是我们之前说的浮点数无法精确存储的问题。

因此，即使通过 DecimalFormat 来精确控制舍入方式，double 和 float 的问题也可能产生意想不到的结果，所以浮点数避坑第二原则：**浮点数的字符串格式化也要通过 BigDecimal 进行。**

比如下面这段代码，使用 BigDecimal 来格式化数字 3.35，分别使用向下舍入和四舍五入方式取 1 位小数进行格式化：

复制代码

```
BigDecimal num1 = new BigDecimal("3.35");
BigDecimal num2 = num1.setScale(1, BigDecimal.ROUND_DOWN);
System.out.println(num2);
BigDecimal num3 = num1.setScale(1, BigDecimal.ROUND_HALF_UP);
System.out.println(num3);
```

这次得到的结果是 3.3 和 3.4，符合预期。

用 equals 做判等，就一定是对的吗?

现在我们知道了，应该使用 BigDecimal 来进行浮点数的表示、计算、格式化。在上一讲介绍判等问题时，我提到一个原则：包装类的比较要通过 equals 进行，而不能使用 ==。那么，使用 equals 方法对两个 BigDecimal 判等，一定能得到我们想要的结果吗?

我们来看下面的例子。使用 equals 方法比较 1.0 和 1 这两个 BigDecimal：

复制代码

```
System.out.println(new BigDecimal("1.0").equals(new BigDecimal("1")))
```

你可能已经猜到我要说什么了，结果当然是 false。BigDecimal 的 equals 方法的注释中说明了原因，equals 比较的是 BigDecimal 的 value 和 scale，1.0 的 scale 是 1，1 的 scale 是 0，所以结果一定是 false：

复制代码

```
/**
 * Compares this {@code BigDecimal} with the specified
 * {@code Object} for equality.  Unlike {@link
 * #compareTo(BigDecimal) compareTo}, this method considers two
 * {@code BigDecimal} objects equal only if they are equal in
 * value and scale (thus 2.0 is not equal to 2.00 when compared by
 * this method).
 *
 * @param  x {@code Object} to which this {@code BigDecimal} is
 *         to be compared.
 * @return {@code true} if and only if the specified {@code Object} is a
 *         {@code BigDecimal} whose value and scale are equal to this
 *         {@code BigDecimal}'s.
 * @see    #compareTo(java.math.BigDecimal)
 * @see    #hashCode
 */
@Override
public boolean equals(Object x)
```

如果我们希望只比较 BigDecimal 的 value，可以使用 compareTo 方法，修改后代码如下：

```
System.out.println(new BigDecimal("1.0").compareTo(new BigDecimal("1"))==0);
```

学过上一讲，你可能会意识到 BigDecimal 的 equals 和 hashCode 方法会同时考虑 value 和 scale，如果结合 HashSet 或 HashMap 使用的话就可能会出现麻烦。比如，我们把值为 1.0 的 BigDecimal 加入 HashSet，然后判断其是否存在值为 1 的 BigDecimal，得到的结果是 false：

```
Set<BigDecimal> hashSet1 = new HashSet<>();
hashSet1.add(new BigDecimal("1.0"));
System.out.println(hashSet1.contains(new BigDecimal("1")));//返回false
```

解决这个问题的办法有两个：

第一个方法是，使用 TreeSet 替换 HashSet。TreeSet 不使用 hashCode 方法，也不使用 equals 比较元素，而是使用 compareTo 方法，所以不会有问题。

```
Set<BigDecimal> treeSet = new TreeSet<>();
treeSet.add(new BigDecimal("1.0"));
System.out.println(treeSet.contains(new BigDecimal("1")));//返回true
```

第二个方法是，把 BigDecimal 存入 HashSet 或 HashMap 前，先使用 stripTrailingZeros 方法去掉尾部的零，比较的时候也去掉尾部的 0，确保 value 相同的 BigDecimal，scale 也是一致的：

```
Set<BigDecimal> hashSet2 = new HashSet<>();
hashSet2.add(new BigDecimal("1.0").stripTrailingZeros());
System.out.println(hashSet2.contains(new BigDecimal("1.000").stripTrailingZero
```

小心数值溢出问题

数值计算还有一个要小心的点是溢出，不管是 int 还是 long，所有的基本数值类型都有超出表达范围的可能性。

比如，对 Long 的最大值进行 +1 操作：

复制代码

```
long l = Long.MAX_VALUE;
System.out.println(l + 1);
System.out.println(l + 1 == Long.MIN_VALUE);
```

输出结果是一个负数，因为 Long 的最大值 +1 变为了 Long 的最小值：

复制代码

```
-9223372036854775808
true
```

显然这是发生了溢出，而且是默默的溢出，并没有任何异常。这类问题非常容易被忽略，改进方式有下面 2 种。

方法一是，考虑使用 Math 类的 addExact、subtractExact 等 xxExact 方法进行数值运算，这些方法可以在数值溢出时主动抛出异常。我们来测试一下，使用 Math.addExact 对 Long 最大值做 +1 操作：

复制代码

```
try {
    long l = Long.MAX_VALUE;
    System.out.println(Math.addExact(l, 1));
} catch (Exception ex) {
    ex.printStackTrace();
}
```

执行后，可以得到 ArithmeticException，这是一个 RuntimeException：

复制代码

```
java.lang.ArithmeticException: long overflow
  at java.lang.Math.addExact(Math.java:809)
```

```
3    at org.geekbang.time.commonmistakes.numeralcalculations.demo3.CommonMistakes
4    at org.geekbang.time.commonmistakes.numeralcalculations.demo3.CommonMistakes
```

方法二是，使用大数类 BigInteger。BigDecimal 是处理浮点数的专家，而 BigInteger 则是对大数进行科学计算的专家。

如下代码，使用 BigInteger 对 Long 最大值进行 +1 操作；如果希望把计算结果转换一个 Long 变量的话，可以使用 BigInteger 的 longValueExact 方法，在转换出现溢出时，同样会抛出 ArithmeticException：

复制代码

```
BigInteger i = new BigInteger(String.valueOf(Long.MAX_VALUE));
System.out.println(i.add(BigInteger.ONE).toString());

try {
    long l = i.add(BigInteger.ONE).longValueExact();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

输出结果如下：

复制代码

```
9223372036854775808
java.lang.ArithmeticException: BigInteger out of long range
  at java.math.BigInteger.longValueExact(BigInteger.java:4632)
  at org.geekbang.time.commonmistakes.numeralcalculations.demo3.CommonMistakes
  at org.geekbang.time.commonmistakes.numeralcalculations.demo3.CommonMistakes
```

可以看到，通过 BigInteger 对 Long 的最大值加 1 一点问题都没有，当尝试把结果转换为 Long 类型时，则会提示 BigInteger out of long range。

重点回顾

今天，我与你分享了浮点数的表示、计算、舍入和格式化、溢出等涉及的一些坑。

第一，切记，要精确表示浮点数应该使用 BigDecimal。并且，使用 BigDecimal 的 Double 入参的构造方法同样存在精度丢失问题，应该使用 String 入参的构造方法或者 BigDecimal.valueOf 方法来初始化。

第二，对浮点数做精确计算，参与计算的各种数值应该始终使用 BigDecimal，所有的计算都要通过 BigDecimal 的方法进行，切勿只是让 BigDecimal 来走过场。任何一个环节出现精度损失，最后的计算结果可能都会出现误差。

第三，对于浮点数的格式化，如果使用 String.format 的话，需要认识到它使用的是四舍五入，可以考虑使用 DecimalFormat 来明确指定舍入方式。但考虑到精度问题，我更建议使用 BigDecimal 来表示浮点数，并使用其 setScale 方法指定舍入的位数和方式。

第四，进行数值运算时要小心溢出问题，虽然溢出后不会出现异常，但得到的计算结果是完全错误的。我们考虑使用 Math.xxxExact 方法来进行运算，在溢出时能抛出异常，更建议对于可能会出现溢出的大数运算使用 BigInteger 类。

总之，对于金融、科学计算等场景，请尽可能使用 BigDecimal 和 BigInteger，避免由精度和溢出问题引发难以发现，但影响重大的 Bug。

今天用到的代码，我都放在了 GitHub 上，你可以点击这个链接查看。

思考与讨论

1. BigDecimal提供了 8 种舍入模式，你能通过一些例子说说它们的区别吗？
2. 数据库（比如 MySQL）中的浮点数和整型数字，你知道应该怎样定义吗？又如何实现浮点数的准确计算呢？

针对数值运算，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

上一篇　08 | 判等问题：程序里如何确定你就是你?

下一篇　10 | 集合类：坑满地的List列表操作

精选留言 (13)

写留言

Darren
2020-03-28

精度问题遇到的比较少，可能与从事非金融行业有关系，试着回答下问题
第一种问题
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）。 ...
展开

作者回复: 👍

2　9

2020-03-28

double\float精度问题，会导致一些结果不是我们想要的。比如3.35 其实如果用double表示，则是3.34900000，如果用float表示，则是3.500000所以控制精度不能用他们。
浮点数的字符串格式化也要通过 BigDecimal 进行。

BigDecimal num1 = new BigDecimal("3.35");...

展开

2

2020-03-28

想请教一下。关于金额。
还存在 使用Long类型的分存储，以及封装的money对象存储的方式。这两种方式适合解决金额类的精度丢失问题嘛?

作者回复: 用分存储是可以（解决精度问题），但是容易出错，万一读的时候忘记/100或者是存的时候忘记*100，可能会引起重大问题，还是使用DECIMAL(13, 2) /DECIMAL(13, 4) 存比较好。

2

Geek_3b1096
2020-03-29

用equals对两BigDecimal判等...之前就被坑了

展开

1

hellojd
2020-03-28

还有 bigdecimal的值对比，也是问题

展开

1

hellojd
2020-03-28

对账时，涉及double求和 ,遇到了

展开

1

pedro
2020-03-28

第一个问题，BigDecimal 的 8 中 Round模式，分别是
1.ROUND_UP：向上取整，如 5.1 被格式化后为 6，如果是负数则与直观上不一致，如 -1.1 会变成 -2。2.ROUND_DOWN：向下取整，与 ROUND_UP 相反。
3.ROUND_CEILING：正负数分开版的取整，如果是正数，则与 ROUND_UP 一样，如果是负数则与 ROUND_DOWN 一样。...
展开

作者回复: 👍

1

梦倚栏杆
2020-03-28

0.45和0.55也有这个问题。double a=0.45,可以输出0.45,double b =1-0.55=0.444444449

1

伍嘉儒
2020-04-01

感谢老师，看完这篇文章，改了BigDecimal工具类，避免了一个事故。

作者回复: 赞

Monday
2020-03-29

手机计算器把 10%+10% 算成了 0.11 而不是 0.2。
读到这里，吓得我赶快掏出安卓机算了下

吴国帅
2020-03-29

真棒 get到知识了！
展开

作者回复: 觉得好可以多转发分享

许童童

2020-03-28

浮点数我印象很深刻的一个问题就是大数吃小数的问题

展开

每天晒白牙

2020-03-28

我们现在对金额的计算都是用分做单位处理

展开