

昵称: 沧海一滴
园龄: 6年1个月
粉丝: 106
关注: 308
[+加关注](#)

< 2017年5月 >						
日	一	二	三	四	五	六
30	<u>1</u>	<u>2</u>	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

我的标签

- [lua\(14\)](#)
- [git\(12\)](#)
- [Mysql\(11\)](#)
- [maven\(11\)](#)
- [Mybatis\(9\)](#)
- [groovy\(7\)](#)
- [spring\(7\)](#)
- [Spring Boot\(6\)](#)
- [jvm\(6\)](#)
- [Oracle\(6\)](#)
- [更多](#)

阅读排行榜

- 1. 四种常见的 POST 提交数据方式--good (26612)
- 2. Spring Boot 属性配置和使用 (转) (8233)
- 3. Spring Boot 配置优先级顺序(8192)
- 4. SpringBoot优化内嵌的Tomcat(7579)
- 5. Spring Boot使用redis做数据缓存(6533)

评论排行榜


- 1. webapi文档描述-swagger(10)
- 2. restful风格, restcontroller与controller(5)
- 3. Iterator、Iterable接口的使用及详解(4)
- 4. 验证码识别的一些总结及相关代码(4)
- 5. java extends与implements在使用时的一个差异(3)

推荐排行榜


- 1. 四种常见的 POST 提交数据方式--good (3)
- 2. webapi文档描述-swagger(3)
- 3. 如果看了这篇文章你还不理解傅里叶变换，那就过来掐死我吧(转)(2)
- 4. lua.c:80:31: fatal error: readline/readline.h: No such file or directory(2)
- 5. Https 客户端与服务器交互过程梳理 (转) (2)

java中浮点数的比较 (double, float)(转)

问题的提出：
如果我们编译运行下面这个程序会看到什么？



```
public static void main(String args[]){
    System.out.println(0.05+0.01);
    System.out.println(1.0-0.42);
    System.out.println(4.015*100);
    System.out.println("BigDecimal:"+new
BigDecimal(Double.toString(4.015)).multiply(new BigDecimal(Double.toString(100))));
    System.out.println(123.3/100);
}
```



你没有看错！结果确实是
0.060000000000000005
0.5800000000000001
401.49999999999994
BigDecimal:401.5000
1.2329999999999999
Java中的简单浮点数类型float和double不能够进行运算。不光是Java，在其它很多编程语言中也有这样的问题。在大多数情况下，计算的结果是准确的，但是多试几次（可以做一个循环）就可以试出类似上面的错误。现在终于理解为什么要有BCD码了。
这个问题相当严重，如果你有9.999999999999元，你的计算机是不会认为你可以购买10元的商品的。
在有的编程语言中提供了专门的货币类型来处理这种情况，但是Java没有。现在让我们看看如何解决这个问题。

四舍五入
我们的第一个反应是做四舍五入。Math类中的round方法不能设置保留几位小数，我们只能象这样（保留两位）：
public double round(double value){
 return Math.round(value*100)/100.0;
}
非常不幸，上面的代码并不能正常工作，给这个方法传入4.015它将返回4.01而不是4.02，如我们在上面看到的
4.015*100=401.49999999999994
因此如果我们要做到精确的四舍五入，不能利用简单类型做任何运算
java.text.DecimalFormat也不能解决这个问题：
System.out.println(new java.text.DecimalFormat("0.00").format(4.025));
输出是4.02

BigDecimal
在《Effective Java》这本书中也提到这个原则，float和double只能用来做**科学计算**或者是**工程计算**，在商业计算中我们要用 java.math.BigDecimal。BigDecimal一共有4个够造方法，我们不关心用BigInteger来够造的那两个，那么还有两个，它们是：
BigDecimal(double val)
 Translates a double into a BigDecimal.
BigDecimal(String val)
 Translates the String repre sentation of a BigDecimal into a BigDecimal.
上面的API简要描述相当的明确，而且通常情况下，上面的那一个使用起来要方便一些。我们可能想都不想就用上了，会有什么问题呢？等到出了问题的时候，才发现上面哪个够造方法的详细说明中有这么一段：
Note: the results of this constructor can be somewhat unpredictable. One might assume that new BigDecimal(.1) i s exactly equal to .1, but it is actually equal to .1000000000000000555111512312578270211815834045410156 25. This is so because .1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the long value that is being passed in to the constructor is not exactly equal to .1, appearance

s notwithstanding.

The (String) constructor, on the other hand, is perfectly predictable: new BigDecimal(".1") is exactly equal to .1, as one would expect. Therefore, it is generally recommended that the (String) constructor be used in preference to this one.

原来我们**如果需要精确计算，非要用String来构造BigDecimal不可！**在《Effective Java》一书中的例子是用String来构造BigDecimal的，但是书上却没有强调这一点，这也许是一个小小的失误吧。

解决方案

现在我们已经可以解决这个问题了，原则是使用BigDecimal并且一定要用String来构造。

但是想像一下吧，如果我们要做一个加法运算，需要先将两个浮点数转为String，然后够造成BigDecimal，在其中一个上调用add方法，传入另一个作为参数，然后把运算的结果（BigDecimal）再转换为浮点数。你能够忍受这么烦琐的过程吗？下面我们提供一个工具类Arith来简化操作。它提供以下静态方法，包括加减乘除和四舍五入：

```
public static double add(double v1,double v2)
public static double sub(double v1,double v2)
public static double mul(double v1,double v2)
public static double div(double v1,double v2)
public static double div(double v1,double v2,int scale)
public static double round(double v,int scale)
```

附录

源文件Arith.java：

```
import java.math.BigDecimal;

/**
 * 由于Java的简单类型不能够精确的对浮点数进行运算，这个工具类提供精
 * 确的浮点数运算，包括加减乘除和四舍五入。
 */
public class Arith{
    //默认除法运算精度
    private static final int DEF_DIV_SCALE = 10;
    //这个类不能实例化
    private Arith(){
    }

    /**
     * 提供精确的加法运算。
     * @param v1 被加数
     * @param v2 加数
     * @return 两个参数的和
     */
    public static double add(double v1,double v2){
        BigDecimal b1 = new BigDecimal(Double.toString(v1));
        BigDecimal b2 = new BigDecimal(Double.toString(v2));
        return b1.add(b2).doubleValue();
    }

    /**
     * 提供精确的减法运算。
     * @param v1 被减数
     * @param v2 减数
     * @return 两个参数的差
     */
    public static double sub(double v1,double v2){
        BigDecimal b1 = new BigDecimal(Double.toString(v1));
        BigDecimal b2 = new BigDecimal(Double.toString(v2));
        return b1.subtract(b2).doubleValue();
    }

    /**
     * 提供精确的乘法运算。
     * @param v1 被乘数
     * @param v2 乘数
     * @return 两个参数的积
     */
    public static double mul(double v1,double v2){
        BigDecimal b1 = new BigDecimal(Double.toString(v1));
        BigDecimal b2 = new BigDecimal(Double.toString(v2));
        return b1.multiply(b2).doubleValue();
    }

    /**
     * 提供（相对）精确的除法运算，当发生除不尽的情况时，精确到
     * 小数点以后10位，以后的数字四舍五入。
     * @param v1 被除数
     * @param v2 除数
     * @return 两个参数的商
     */
}
```

```

public static double div(double v1,double v2){
    return div(v1,v2,DEF_DIV_SCALE);
}

/**
 * 提供（相对）精确的除法运算。当发生除不尽的情况时，由scale参数指
 * 定精度，以后的数字四舍五入。
 * @param v1 被除数
 * @param v2 除数
 * @param scale 表示表示需要精确到小数点以后几位。
 * @return 两个参数的商
 */
public static double div(double v1,double v2,int scale){
    if(scale<0){
        throw new IllegalArgumentException(
            "The scale must be a positive integer or zero");
    }
    BigDecimal b1 = new BigDecimal(Double.toString(v1));
    BigDecimal b2 = new BigDecimal(Double.toString(v2));
    return b1.divide(b2,scale,BigDecimal.ROUND_HALF_UP).doubleValue();
}

/**
 * 提供精确的小数位四舍五入处理。
 * @param v 需要四舍五入的数字
 * @param scale 小数点后保留几位
 * @return 四舍五入后的结果
 */
public static double round(double v,int scale){
    if(scale<0){
        throw new IllegalArgumentException(
            "The scale must be a positive integer or zero");
    }
    BigDecimal b = new BigDecimal(Double.toString(v));
    BigDecimal one = new BigDecimal("1");
    return b.divide(one,scale,BigDecimal.ROUND_HALF_UP).doubleValue();
}
}

```

<http://blog.csdn.net/pttaag/article/details/5912171>

最近在项目中碰到了有一个业务逻辑计算,代码如下(示例代码)

```

double val1 = ...;

double val2 = ...,

double dif = ...,

if (Math.abs(val1 - val2-dif) == 0){

    //do things

}

```

结果发现有一组数据:61.5,60.4,1.1无法达到正确的结果.有经验的开发人员一眼就可以发现问题所在,也知道应该采用如下的方式修改代码(产品模式下要进行代码的抽取和封装):

```

double exp = 10E-10;

if (Math.abs(val1 - val2-dif)>-1*exp && Math.abs(val1 - val2-dif)<exp){

    //do things

}

```

那为什么上面代码中"Math.abs(val1 - val2-dif) == 0"的值为什么会是false呢?这就引申到java的一个基础问题,即java中浮点数的存储机制.

Java 中的浮点数分为单精度和双精度数,也就是float和double.

float在内存中跟int一样,占4个字节,32 bit.

第1个bit表示符号,0表示正数,1表示负数,这个很好理解,不用多管.

第2-9个bit表示指数,一共8位(可以表示0-255),这里的底数是2,为了同时表示正数和负数,这里要减去127的偏移量.这样的话范围就是(-127到128),

另外全0和全1作为特殊处理,所以直接表示-126到127.

剩下的23位表示小数部分,这里23位表示了24位的数字,因为有一个默认的前导1(只有二进制才有这个特性).

最后结果是: $(-1)^{\text{sign}} * 1.f * 2^{\text{exponent}}$

这里:sign是符号位,f是23bit的小数部分,exponent是指数部分,最后表示范围是(因为正负数是对称的,这里只关心正数)

$2^{-(126)} \sim 2(1-2^{-(24)}) * 2^{127}$

这个还不是float的取值范围,因为标准中还规定了非规格化表示法,另外还有一些特殊规定.

非规格化表示:

当指数部分全0而且小数部分不全0时表示的是非规格化的浮点数,因为这里默认没有前导1,而是0.

取值位 $0.f * 2^{-(126)}$,表示范围位 $2^{-(149)} \sim (1-2^{-(23)}) * 2^{-(126)}$ 这里没有考虑符号.这里为什么是-126而不是-127? 如果是-127的话,那么最大表示为

$2^{-(127)}-2^{-(149)}$,很显然 $2^{-(127)} \sim 2^{-(126)}$ 就没法表示了.

其他特殊表示

- 1.当指数部分和小数部分全为0时,表示0值,有+0和-0之分(符号位决定),0x00000000表示正0,0x80000000表示负0.
- 2.指数部分全1,小数部分全0时,表示无穷大,有正无穷和负无穷,0x7f800000表示正无穷,0xff800000表示负无穷.
- 3.指数部分全1,小数部分不全0时,表示NaN,分为QNaN和SNaN,Java中都是NaN.

结论:

可以看出浮点数的取值范围是: $2^{-(149)} \sim (2-2^{-(23)}) * 2^{127}$,也就是Float.MIN_VALUE和Float.MAX_VALUE.

References:

<http://blog.csdn.net/treeroot/archive/2004/09/05/95071.aspx>

<http://hi.baidu.com/520miner/blog/item/698266ed9ee000d7b31cb1aa.html>

<http://blog.csdn.net/running8063/article/details/4093261>

Java中关于 BigDecimal 的一个导致double精度损失的"bug"

背景

在博客 [恶心的0.5四舍五入问题](#) 一文中看到一个关于 0.5 不能正确的四舍五入的问题。主要说的是 double 转换到 BigDecimal 后, 进行四舍五入得不到正确的结果:

```
public class BigDecimalTest {
    public static void main(String[] args){
        double d = 301353.05;
        BigDecimal decimal = new BigDecimal(d);
        System.out.println(decimal);//301353.04999999999883584678173065185546875
        System.out.println(decimal.setScale(1, RoundingMode.HALF_UP));//301353.0
    }
}
```

输出的结果为:

```
301353.0499999999883584678173065185546875
301353.0
```

这个结果显然不是我们所期望的，我们希望的是得到 301353.1。

原因

允许明眼人一眼就看出另外问题所在——BigDecimal的构造函数 `public BigDecimal(double val)` 损失了 `double` 参数的精度，最后才导致了错误的结果。所以问题的关键是：BigDecimal的构造函数 `public BigDecimal(double val)` 损失了 `double` 参数的精度。

解决之道

因为上面找到了原因，所以也就很好解决了。只要防止了 `double` 到 `BigDecimal` 的精度损失，也就不会出现问

1) 很容易想到第一个解决办法：使用BigDecimal的以String为参数的构造函数：`public BigDecimal(String val)` 来替代。

```
public class BigDecimalTest {
    public static void main(String[] args){
        double d = 301353.05;
        System.out.println(new BigDecimal(new Double(d).toString()));
        System.out.println(new BigDecimal("301353.05"));
        System.out.println(new BigDecimal("301353.895898895455898954895989"));
    }
}
```

输出结果：

```
301353.05
301353.05
301353.895898895455898954895989
```

我们看到了没有任何的精度损失，四舍五入也就肯定不会出错了。

2) BigDecimal的构造函数 `public BigDecimal(double val)` 会损失了 `double` 参数的精度，这个也许应该可以算是 JDK 中的一个 bug 了。既然存在bug，那么我们就应该解决它。上面的办法是绕过了它。现在我们实现自己的 `double` 到 `BigDecimal` 的转换，并且保证在某些情况下可以完全不损失 `double` 的精度。

```
import java.math.BigDecimal;

public class BigDecimalUtil {

    public static BigDecimal doubleToBigDecimal(double d){
        String doubleStr = String.valueOf(d);
        if(doubleStr.indexOf(".") != -1){
            int pointLen = doubleStr.replaceAll("\\d+\\.","").length();    // 取得小数点后的数字的位数
            pointLen = pointLen > 16 ? 16 : pointLen;    // double最大有效小数点后的位数为16
            double pow = Math.pow(10, pointLen);

            long tmp = (long)(d * pow);
            return new BigDecimal(tmp).divide(new BigDecimal(pow));
        }
        return new BigDecimal(d);
    }

    public static void main(String[] args){
        // System.out.println(doubleToBigDecimal(301353.05));
        // System.out.println(doubleToBigDecimal(-301353.05));
        // System.out.println(doubleToBigDecimal(new Double(-301353.05)));
        // System.out.println(doubleToBigDecimal(301353));
        // System.out.println(doubleToBigDecimal(new Double(-301353)));

        double d = 301353.05;//5898895455898954895989;
        System.out.println(doubleToBigDecimal(d));
    }
}
```

```

        System.out.println(d);
        System.out.println(new Double(d).toString());
        System.out.println(new BigDecimal(new Double(d).toString()));
        System.out.println(new BigDecimal(d));
    }
}

```

输出结果:

```

301353.05
301353.05
301353.05
301353.05
301353.04999999999883584678173065185546875

```

上面我们自己写了一个工具类，实现了 double 到 BigDecimal 的“无损失”double精度的转换。方法是将小数点后有效数字的double先转换到小数点后没有有效数字的double，然后在转换到 BigDecimal，之后使用 BigDecimal的 divide 返回之前的大小。

上面的结果看起来好像十分的完美，但是其实是存在问题的。上面我们也说到了“某些情况下可以完全不损失double的精度”，我们先看一个例子：

```

public static void main(String[] args){
    double d = 301353.05;
    System.out.println(doubleToBigDecimal(d));
    System.out.println(d);
    System.out.println(new Double(d).toString());
    System.out.println(new BigDecimal(new Double(d).toString()));
    System.out.println(new BigDecimal(d));

    System.out.println("=====");
    d = 301353.895898895455898954895989;
    System.out.println(doubleToBigDecimal(d));
    System.out.println(d);
    System.out.println(new Double(d).toString());
    System.out.println(new BigDecimal(new Double(d).toString()));
    System.out.println(new BigDecimal(d));
    System.out.println(new BigDecimal("301353.895898895455898954895989"));

    System.out.println("=====");
    d = 301353.46899434;
    System.out.println(doubleToBigDecimal(d));
    System.out.println(d);
    System.out.println(new Double(d).toString());
    System.out.println(new BigDecimal(new Double(d).toString()));
    System.out.println(new BigDecimal(d));

    System.out.println("=====");
    d = 301353.45789666;
    System.out.println(doubleToBigDecimal(d));
    System.out.println(d);
    System.out.println(new Double(d).toString());
    System.out.println(new BigDecimal(new Double(d).toString()));
    System.out.println(new BigDecimal(d));
}

```

输出结果:

```

301353.05
301353.05
301353.05
301353.05
301353.04999999999883584678173065185546875
=====
301353.89589889544
301353.89589889545
301353.89589889545

```

```

301353.89589889545
301353.895898895454593002796173095703125
301353.895898895455898954895989
=====
301353.46899434
301353.46899434
301353.46899434
301353.46899434
301353.4689943399862386286258697509765625
=====
301353.45789666
301353.45789666
301353.45789666
301353.45789666
301353.4578966600238345563411712646484375

```

我们可以看到：我们自己实现的 `doubleToBigDecimal` 方法只有在 `double` 的小数点后的数字位数比较少时(比如只有5,6位)，才能保证完全的不损失精度。

在 `double` 的小数点后的数字位数比较多时，`d * pow` 会存在精度损失，所以最终的结果也会存在精度损失。所以如果小数点后的位数比较多时，还是使用 `BigDecimal` 的 `String` 参数的构造函数为好，只有在小数点后的位数比较少时，才可以采用自己实现的 `doubleToBigDecimal` 方法。

因为我们看到原始的`double`的转换之后的`BigDecimal`的数字的最后一位一个时5，一个是4，原因是在上面的转换方法中：

```
long tmp = (long)(d * pow);
```

这一步可能存在很小的精度损失，因为 `d` 是一个 `double`，`d * pow` 之后还是一个 `double`(但是小数点之后都是0了，所以到`long`的转换没有精度损失)，所以会存在很小的精度损失(`double`的计算总是有可能存在精度损失的)。但是这个精度损失和 `BigDecimal` 的构造函数 `public BigDecimal(double val)` 的精度损失相比而言，不会显得那么的突兀(也许我们自己写的`doubleToBigDecimal`也是存在问题的，欢迎指点)。

总结：

如果需要保证精度，最好是不要使用`BigDecimal`的`double`参数的构造函数，因为存在损失`double`参数精度的可能，最好是使用`BigDecimal`的`String`参数的构造函数。最好是杜绝使用`BigDecimal`的`double`参数的构造函数。

后记：

其实说这是`BigDecimal`的一个bug，有标题党的嫌疑，最多可以算是`BigDecimal`的一个“坑”。

<http://www.cnblogs.com/digdeep/p/4459781.html>

恶心的0.5四舍五入问题

四舍五入是财务类应用中常见的需求，按中国人的财务习惯，遇到0.5统一向上进位，但是`c#`与`java`中默认的不是这样。

见`c#`代码：

```

1      static void Main(string[] args)
2      {
3          Decimal d = 301353.05M;
4          Console.WriteLine(d);//301353.05
5          Console.WriteLine(Math.Round(d, 1));//301353.0
6          Console.WriteLine(Math.Round(d, 1,
MidpointRounding.AwayFromZero));//301353.1
7
8          Console.ReadKey();
9      }

```

默认情况下，如果要舍弃的位置上，正好值是5，系统会看前一位是奇数还是偶数，如果是偶数，则丢弃最后1位，即上面代码行5，输出的结果为 301353.0，这不符合国人的习惯，所以要人为指定第3个参数“`MidpointRounding.AwayFromZero`”

java中也提出了类似的做法，但是有“缺陷”

```
1  @Test
2  public void testScale(){
3      double d = 301353.05;
4      BigDecimal decimal = new BigDecimal(d);
5      System.out.println(decimal);//301353.04999999999883584678173065185546875
6      System.out.println(decimal.setScale(1, RoundingMode.HALF_UP));//301353.0
7  }
```

类似的，在设置精度时，可以指定一个额外的参数RoundingMode.HALF_UP，表示如果要舍弃的这一位正好是5，则向上进位，代码看似没有问题，但是输出值却是301353.0

原因在于BigDecimal在计算机内部的存储值为"301353.04999999999883584678173065185546875"，即小数点第2位是4，上面的代码要求精度到1位，所以代码执行时，只看第2个小数位，其值为4，没有到HALF的标准，因此直接扔掉

改进方法：

```
1  @Test
2  public void testScale(){
3      double d = 301353.05 + 0.0000000001;
4      BigDecimal decimal = new BigDecimal(d);
5      System.out.println(decimal);//301353.05000000001047737896442413330078125
6      System.out.println(decimal.setScale(1, RoundingMode.HALF_UP));//301353.1
7  }
```

在满足财务精度的前提下，将要处理的数字加1个微小的偏移量，这样计算机内部存储时，值变成301353.0500000001047737896442413330078125，这样小数位第2位变成了5，满足了HALF_UP的条件。

<http://www.cnblogs.com/yjmyzz/p/4427669.html>

最根本的原因是使用了错误的构造函数：BigDecimal decimal = new BigDecimal(d);导致了精度损失。使用下面的构造函数就不会导致精度损失了：

BigDecimal decimal = new BigDecimal(String.valueOf(d));

或者

BigDecimal decimal = new BigDecimal(new Double(d).toString());

这是BigDecimal的一个坑。

标签: [java](#) [double](#) [float](#) [BigDecimal](#)

好文要顶

关注我

收藏该文





沧海一滴

关注 - 308

粉丝 - 106

+加关注

0

0

« 上一篇: [SVN与TortoiseSVN实战: 补丁详解\(转\)](#)

» 下一篇: [Groovy常见语法汇总](#)

posted @ 2015-03-24 23:01 沧海一滴 阅读(4554) 评论(0) 编辑 收藏

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [访问网站首页](#)。

- 【推荐】50万行VC++源码: 大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】中铁、中石油等大型企业的复杂报表解决方案
- 【福利】阿里云免费套餐升级, 更多产品, 更久时长



最新IT新闻:

- 当今世界最有价值的资源是什么? 不是石油, 而是数据
- 智能家居时代的第一场谋杀案故事
- 苹果实习生日记: 每周都和副总裁吃饭
- 传摩托罗拉正在研发首款Android平板电脑
- 周亚辉回应空空狐事件: 断章取义, 拼凑故事, 自我炒作
- » 更多新闻...



最新知识库文章:

- 唱吧DevOps的落地, 微服务CI/CD的范本技术解读
- 程序员, 如何从平庸走向理想?
- 我为什么鼓励工程师写blog
- 怎么轻松学习JavaScript
- 如何打好前端游击战
- » 更多知识库文章...