

动手动脑课堂练习

一、原码、反码和补码

(出处: <http://www.cnblogs.com/zhangzhiqiu/>)

1、机器数和真值

在学习原码, 反码和补码之前, 需要先了解机器数和真值的概念.

(1)、机器数

一个数在计算机中的二进制表示形式, 叫做这个数的机器数。机器数是带符号的, 在计算机用一个数的最高位存放符号, 正数为0, 负数为1. 比如, 十进制中的数 +3, 计算机字长为8位, 转换成二进制就是00000011。如果是 -3, 就是 10000011。那么, 这里的 00000011 和 10000011 就是机器数。

(2)、真值

因为第一位是符号位, 所以机器数的形式值就不等于真正的数值。例如上面的有符号数 10000011, 其最高位1代表负, 其真正数值是 -3 而不是形式值131 (10000011转换成十进制等于131)。所以, 为区别起见, 将带符号位的机器数对应的真正数值称为机器数的真值。例: 0000 0001的真值 = +000 0001 = +1, 1000 0001的真值 = - 000 0001 = - 1

2、原码

原码就是符号位加上真值的绝对值, 即用第一位表示符号, 其余位表示值. 比如如果是8位二进制:

[+1]原 = 0000 0001

[-1]原 = 1000 0001

第一位是符号位. 因为第一位是符号位, 所以8位二进制数的取值范围就是:

[1111 1111, 0111 1111]

即[-127 , 127]

原码是人脑最容易理解和计算的表示方式.

3、反码

反码的表示方法是:

正数的反码是其本身

负数的反码是在其原码的基础上, 符号位不变, 其余各个位取反.

[+1] = [00000001]原 = [00000001]反

[-1] = [10000001]原 = [11111110]反

可见如果一个反码表示的是负数, 人脑无法直观的看出来它的数值. 通常要将其转换成原码再计算.

4、补码

补码的表示方法是:

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1. (即在反码的基础上+1)

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

对于负数, 补码表示方式也是人脑无法直观看出其数值的. 通常也需要转换成原码在计算其数值.

5、Java对正负数进行各种位操作使用的是补码

现在我们知道了计算机可以有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

[+1] = [00000001]原 = [00000001]反 = [00000001]补

所以不需要过多解释. 但是对于负数:

$$[-1] = [10000001]_{\text{原}} = [11111110]_{\text{反}} = [11111111]_{\text{补}}$$

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. (真值的概念在本文最开头). 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: $1-1 = 1 + (-1) = 0$, 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了. 于是人们开始探索 将符号位参与运算, 并且只保留加法的方法. 首先来看原码:

计算十进制的表达式: $1-1=0$

$$1 - 1 = 1 + (-1) = [00000001]_{\text{原}} + [10000001]_{\text{原}} = [10000010]_{\text{原}} = -2$$

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的. 这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:

计算十进制的表达式: $1-1=0$

$$1 - 1 = 1 + (-1) = [0000\ 0001]_{\text{原}} + [1000\ 0001]_{\text{原}} = [0000\ 0001]_{\text{反}} + [1111\ 1110]_{\text{反}} = [1111\ 1111]_{\text{反}} = [1000\ 0000]_{\text{原}} = -0$$

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有[0000 0000]原和[1000 0000]原两个编码表示0.

于是补码的出现, 解决了0的符号以及两个编码的问题:

$$1-1 = 1 + (-1) = [0000\ 0001]_{\text{原}} + [1000\ 0001]_{\text{原}} = [0000\ 0001]_{\text{补}} + [1111\ 1111]_{\text{补}} = [0000\ 0000]_{\text{补}} = [0000\ 0000]_{\text{原}}$$

这样0用[0000 0000]表示, 而以前出现问题的-0则不存在了. 而且可以用[1000 0000]表示-128:

$$(-1) + (-127) = [1000\ 0001]_{\text{原}} + [1111\ 1111]_{\text{原}} = [1111\ 1111]_{\text{补}} + [1000\ 0001]_{\text{补}} = [1000\ 0000]_{\text{补}}$$

-1-127的结果应该是-128, 在用补码运算的结果中, [1000 0000]补 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示[1000 0000]补算出来的原码是[0000 0000]原, 这是不正确的)

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值.

二、仔细阅读示例: **EnumTest.java**, 运行它, 分析运行结果? **你能得到什么结论?**

1、源程序

```
package demo;

public class EnumTest {

    public static void main(String[] args) {

        Size s=Size.SMALL;

        Size t=Size.LARGE;

        //s和t引用同一个对象?

        System.out.println(s==t);

        //是原始数据类型吗?

        System.out.println(s.getClass().isPrimitive());

        //从字符串中转换

        Size u=Size.valueOf("SMALL");

        System.out.println(s==u); // //列出它的所有值

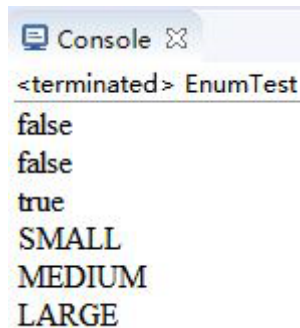
        for(Size value:Size.values()){


            System.out.println(value);
```

```
}  
  
}  
  
}
```

```
enum Size{SMALL,MEDIUM,LARGE;
```

2、结果



```
Console   
<terminated> EnumTest  
false  
false  
true  
SMALL  
MEDIUM  
LARGE
```

3、结果分析

枚举类型的表示方法：enum Size{ SMALL , MEDIUM , LARGE }

使用方法：Size s=Size.SMALL;

//从字符串转换为枚举

Size t=Size.valueOf("SMALL");

枚举类型是引用类型，枚举不属于原始数据类型，它的每个具体值都引用一个特定的对象。相同的值则引用同一个对象。可以使用“==”和equals()方法直接比对枚举变量的值，换句话说，对于枚举类型的变量，“==”和equals()方法执行的结果是等价的。当判断两个赋值是否相同时，输出结果只能是ture或false。当输出value时，需要Size value:Size.values()循环类型的数量的次数。enum Size{SMALL,MEDIUM,LARGE};语句可以写在程序的最后，也可以写在前面，但必须自己引用。枚举的赋值可以有两种表示方法：1.Size s=Size.SMALL; 2.Size t=Size.valueOf(“SMALL”)。

三、Java变量遵循“同名变量的屏蔽原则”，请课后阅读相关资料弄清楚相关知识，然后自己编写一些测试代码，就象本示例一样，有意识地在不同地方定义一些同名变量，看看输出的到底是哪个值。

1、源程序

```
package demo;

public class My1 {

    private static int value=1;

    public static void main(String[] args){

        int value=2;

        System.out.println(value);

    }

}
```

结果：2

2、源程序

```
package demo;

public class My1 {

    private static int value=10;

    public static void main(String[] args){

        int value=6;
```

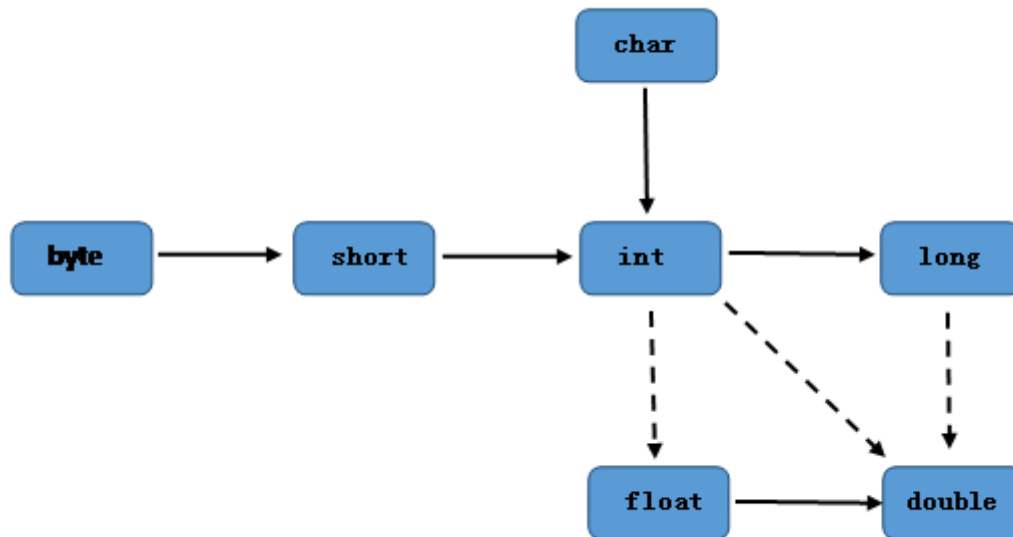
```
System.out.println(value);  
}  
}
```

结果：6

3、结论

在函数里面的赋值优先于在函数外赋值，属于局部变量。函数外的赋值可以赋给类中的多个函数，属于全局变量。如果函数里面没有重复的赋值，那么函数的值为函数外的。

四、看着这个图，再查查**Java中每个数据类型所占的位数，和表示数值的范围，你能得出什么结论？**




答：自动类型转换是安全的，强制类型转换时，可能会引起信息的损失。实线代表无精度损失，虚线代表有精度损失，一般来说在实线两端都是由低精度指向高精度的类型，所占的位数从低到高，范围从小到大，所以可得出，低精度向高精度转化不丢失精度，反之，从高精度传向低精度则会损失。

五、请运行以下代码（**TestDouble.java**），你看到了什么样的输出，意外吗？

```
public class TestDouble {  
    public static void main(String args[]) {  
        System.out.println("0.05 + 0.01 = " + (0.05 + 0.01));  
        System.out.println("1.0 - 0.42 = " + (1.0 - 0.42));  
        System.out.println("4.015 * 100 = " + (4.015 * 100));  
        System.out.println("123.3 / 100 = " + (123.3 / 100));  
    }  
}
```

答：结果为

```
Console   
<terminated> TestDouble [Java Application] |  
0.05 + 0.01 = 0.060000000000000005  
1.0 - 0.42 = 0.58000000000000001  
4.015 * 100 = 401.49999999999994  
123.3 / 100 = 1.2329999999999999
```

感到意外，使用double类型的数值进行计算，其结果是不精确的。计算机只能识别二进制，一切的数据最后都要转换为二进制。例如源程序中的0.05是十进制的，要转换为二进制，但0.05的二进制不是精确的0.05，只是接近0.05，实际为0.04999 999 999 999 999，浮点数由两部分组成：指数和尾数，再进行浮点数的二进制与十进制的转换时，浮点数参与了计算，那么转换过程就变的不可预测，并且变得不可逆。

六、在构建**BigDecimal**对象时应使用字符串而不是double数值，否则，仍有可能引发计算精度问题。（为什么会这样呢？）

```
package demo;
```



```
import java.math.BigDecimal;

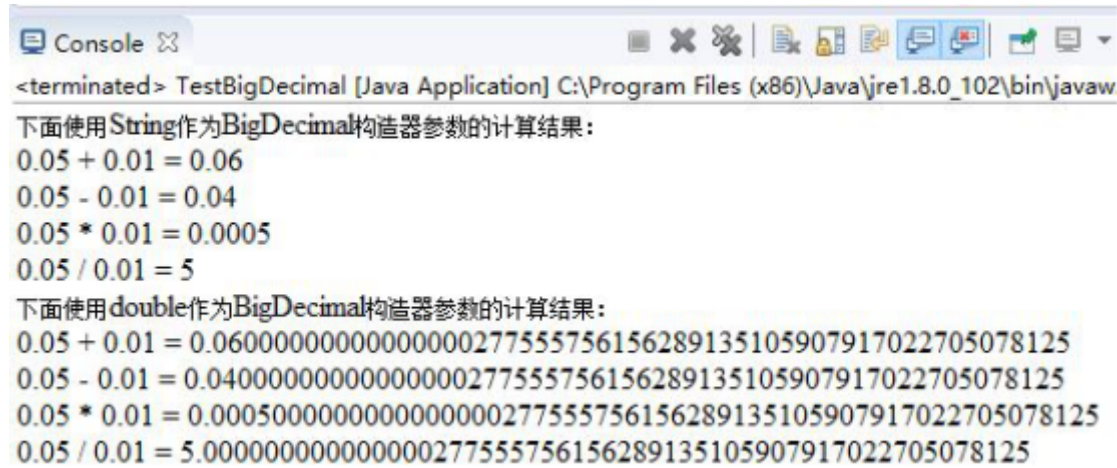
public class TestBigDecimal
{
    public static void main(String[] args)
    {
        BigDecimal f1 = new BigDecimal("0.05");
        BigDecimal f2 = BigDecimal.valueOf(0.01);
        BigDecimal f3 = new BigDecimal(0.05);

        System.out.println("下面使用String作为BigDecimal构造器参数的计算结果: ");
        System.out.println("0.05 + 0.01 = " + f1.add(f2));
        System.out.println("0.05 - 0.01 = " + f1.subtract(f2));
        System.out.println("0.05 * 0.01 = " + f1.multiply(f2));
        System.out.println("0.05 / 0.01 = " + f1.divide(f2));

        System.out.println("下面使用double作为BigDecimal构造器参数的计算结果: ");
        System.out.println("0.05 + 0.01 = " + f3.add(f2));
        System.out.println("0.05 - 0.01 = " + f3.subtract(f2));
        System.out.println("0.05 * 0.01 = " + f3.multiply(f2));
        System.out.println("0.05 / 0.01 = " + f3.divide(f2));
    }
}
```

```
}
```

答：结果为



The screenshot shows a Java IDE console window titled "Console" with the following output:

```
<terminated> TestBigDecimal [Java Application] C:\Program Files (x86)\Java\jre1.8.0_102\bin\javaw  
下面使用String作为BigDecimal构造器参数的计算结果：  
0.05 + 0.01 = 0.06  
0.05 - 0.01 = 0.04  
0.05 * 0.01 = 0.0005  
0.05 / 0.01 = 5  
下面使用double作为BigDecimal构造器参数的计算结果：  
0.05 + 0.01 = 0.06000000000000000277555756156289135105907917022705078125  
0.05 - 0.01 = 0.04000000000000000277555756156289135105907917022705078125  
0.05 * 0.01 = 0.0005000000000000000277555756156289135105907917022705078125  
0.05 / 0.01 = 5.000000000000000277555756156289135105907917022705078125
```

由此可见，构建BigDecimal对象可以解决不精确的问题。但是使用时应使用字符串而不是double数值，否则，仍有可能引发计算精度问题。

七、以下代码的输出结果是什么？

```
int X=100;
```

```
int Y=200;
```

```
System.out.println("X+Y="+X+Y);
```

```
System.out.println(X+Y+"=X+Y");
```

为什么会有这样的输出结果？

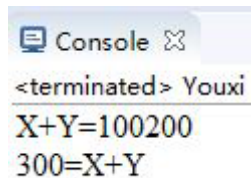
答：源程序：

```
package demo;
```

```
public class Youxi {
```

```
public static void main(String[] args){  
  
    int X=100;  
  
    int Y=200;  
  
    System.out.println("X+Y="+X+Y);  
  
    System.out.println(X+Y+"=X+Y");  
  
    }  
}
```

结果:



The screenshot shows a console window titled "Console" with a close button. It displays the output of the Java program: "<terminated> Youxi", "X+Y=100200", and "300=X+Y".

在System.out.println()中，如果在string字符串后面是+和变量，会把变量转换成string类型，加号起连接作用，然后把两个字符串连接成一个新的字符串输出；如果先有变量的加减运算再有字符串，那么会从左到右先计算变量的加减，然后再与后面的string结合成一个新的字符串。也就是说加号只有在两个string类型或者其中一个是string类型的时候才起到连接作用，否则仍然是运算符。

八、Java中出现0的情况

答：0.0/0.0 得到的结果是NaN(not an number的简称，即"不是数字")。通过Double.isNaN(double x)来判断。

正数/0.0 得到的结果是正无穷大，即Infenity

负数/0.0 得到的结果是负无穷大，即Infenity。通过Double.isInfinite(double x)来判断。

源程序：

```
package demo;
```

```
public class My3{  
    public static void main(String[] args) {  
        double a=0,  
            b=50,  
            c=-50;  
        System.out.println("0/50= \n" + (a/b));  
        System.out.println("0/(-50)= \n" + (a/c));  
        System.out.println("-50/0= \n" + (c/a));  
        System.out.println("0/0= \n" + (a/a));  
        System.out.println("50/0= \n" + (b/a));  
    }  
}
```

结果：

Console

<terminated> My3

0/50=

0.0

0/(-50)=

-0.0

-50/0=

-Infinity

0/0=

NaN

50/0=

Infinity