【讲古堂】浮点数

（[dubenju@126.com](mailto:dubenju@126.com) 2015/12/19）

在处理小数时，会使用定点数，float，double，自己的大数处理比如java的BigDecimal。这里对它们进行简要的阐述。

众所周知，由于用高低电平的电路很容易实现二进制，所以在计算机中普遍采用二进制来存储数据。对应的二进制的位用Bit来表示。1字节=8Bits，如果考虑符号的话，那么一个字节能存储的数值范围是-128到127。只能是整数，不能是小数。那么小数怎么办呢？

小数的话，一定要有小数点的。如果把小数点固定在一个不变的位置的话，就成了定点数。比如Oracle数据库的NUMBER(4, 2)则能存储17.25或0.50这样的数。

定点数(Fixed Point Number)

所谓定点数，即约定数据的小数点位置是固定不变的。通常将定点数据表示成纯小数或纯整数。为了将数表示成纯小数，通常把小数点固定在数值部分的最高位之前；而为了把数表示成纯整数，则把小数点固定在数值部分的最后面。

对纯小数进行运算时，要用适当的比例因子进行折算，以免产生溢出，或过多损失精度。

假设用一个n位字来表示一个定点数x= x0 x1 x2 … xn-1，其中一位x0用来表示数的符号位，其余位数代表它的量值。为了对所有n位进行统一处理，符号位x0通常放在最左位置，并用数值0和1分别代表正号和负号。对于任意定点数x= x0 x1 x2 … xn-1，如果x表示的是纯小数，那么小数点位于x0和x1之间，数的表示范围为：0≤|ｘ|≤1-2-(n-1)；如果x 表示的是纯整数，则小数点位于最低位xn-1的右边，数的表示范围为：0≤|ｘ|≤2n-1-1。

在定点数表示中存在的一个问题是，难以表示数值很大的数据和数值很小的数据。例如，电子的质量（9×10^-28克）和太阳的质量（2×10^33克）相差甚远，在定点计算机中无法直接表示，因为小数点只能固定在某一个位置上，从而限制了数据的表示范围。一般来说，定点数可表示的数值的范围有限，但要求的处理硬件比较简单。比如用NUMBER(4, 2)来存储100.234是不行的。

使用定点数

例子1

FXPN0002.cob

000000 IDENTIFICATION DIVISION.

000000 PROGRAM-ID. FXPN0002.

000000 AUTHOR. dubenju@126.com.

000000 DATE-WRITTEN. 2015.12.18.

000000 DATE-COMPILED.

000000\*

000000 ENVIRONMENT DIVISION.

000000 CONFIGURATION SECTION.

000000 SOURCE-COMPUTER. HP.

000000 OBJECT-COMPUTER. HP.

000000\*

000000 INPUT-OUTPUT SECTION.

000000 FILE-CONTROL.

000000 SELECT A-FILE ASSIGN TO "FXPN0002a.txt".

000000 SELECT B-FILE ASSIGN TO "FXPN0002b.txt".

000000\*

000000 DATA DIVISION.

000000 FILE SECTION.

000000 FD A-FILE RECORDING MODE IS F.

000000 01 A-REC.

000000 05 ACC-NO PIC 9(3)V9(2).

000000 FD B-FILE RECORDING MODE IS F.

000000 01 B-REC.

000000 05 ACC-NO PIC 9(1)V9(4).

000000\*

000000\*

000000 WORKING-STORAGE SECTION.

000000\*

000000 01 A02 PIC 9(3)V9(2).

000000 01 A03 PIC 9(1)V9(4).

000000\*\*\*\* USER-WORK-AREA.

000000\*

000000 PROCEDURE DIVISION.

000000\*

000000 OPEN OUTPUT A-FILE

000000 OUTPUT B-FILE.

000000\*

000000 MOVE 123.45 TO A02.

000000 MOVE 1.2345 TO A03.

000000 DISPLAY '9(3)V9(2)=', A02.

000000 DISPLAY '9(1)V9(4)=', A03.

000000 MOVE A02 TO ACC-NO OF A-REC.

000000 MOVE A03 TO ACC-NO OF B-REC.

000000 WRITE A-REC.

000000 WRITE B-REC.

000000\*

000000 CLOSE A-FILE

000000 B-FILE.

000000\*

000000 STOP RUN.

000000\*

结果：

DBJ@DBJ-PC /prj/cobol/cob

$ ./FXPN0002.exe

9(3)V9(2)=123.45

9(1)V9(4)=1.2345

DBJ@DBJ-PC /prj/cobol/cob

$ cat FXPN0002a.txt

12345

DBJ@DBJ-PC /prj/cobol/cob

$ cat FXPN0002b.txt

12345

DBJ@DBJ-PC /prj/cobol/cob

$

123.45和1.2345写在文件中都是12345。即用五位数来存储12345，在3位整数2位小数定义时，就表示123.45，在1位整数4位小数定义时，就表示1.2345。

例子2

FXPN0001.cob

000000 IDENTIFICATION DIVISION.

000000 PROGRAM-ID. DaFXPN0001.

000000 AUTHOR. dubenju@126.com.

000000 DATE-WRITTEN. 2015.12.18.

000000 DATE-COMPILED.

000000\*

000000 ENVIRONMENT DIVISION.

000000 CONFIGURATION SECTION.

000000 SOURCE-COMPUTER. HP.

000000 OBJECT-COMPUTER. HP.

000000\*

000000 INPUT-OUTPUT SECTION.

000000\*

000000 DATA DIVISION.

000000 FILE SECTION.

000000\*

000000\*

000000 WORKING-STORAGE SECTION.

000000\*

000000 01 A02 PIC 9(3)V9(2).

000000 01 A03 PIC 9(1)V9(4).

000000\*\*\*\* USER-WORK-AREA.

000000\*

000000 PROCEDURE DIVISION.

000000\*

000000 MOVE 12345 TO A02.

000000 MOVE A02 TO A03.

000000 DISPLAY '9(3)V9(2)=', A02.

000000 DISPLAY '9(1)V9(4)=', A03.

000000\*

000000 STOP RUN.

000000\*

结果：

DBJ@DBJ-PC /prj/cobol/cob

$ ./FXPN0001.exe

9(3)V9(2)=345.00

9(1)V9(4)=5.0000

DBJ@DBJ-PC /prj/cobol/cob

在处理时是按照小数点对齐的。

浮点数(Floating Point Number)

为了表示更大范围的数据，数学上通常采用科学计数法，把数据表示成一个小数乘以一个以10为底的指数。

例如，在计算机中，电子的质量和太阳的质量可以分别取不同的比例因子，以使其数值部分的绝对值小于1，即：

9×10^-28 ＝ 0.9×10^-27

2×10^33 ＝ 0.2×10^34

这里的比例因子10^-27和10^34要分别存放在机器的某个单元中，以便以后对计算结果按此比例增大。显然，这要占用一定的存储空间和运算时间。

浮点表示法就是把一个数的有效数字和数的范围在计算机中分别予以表示。这种把数的范围和精度分别表示的方法，相当于数的小数点位置随比例因子的不同而在一定范围内自由浮动，改变指数部分的数值相当于改变小数点的位置。在这种表示法中，小数点的位置是可以浮动的，因此称为浮点表示法。

浮点数的一般表示形式为：

一个十进制数Ｎ可以写成：N = 10e×M

一个二进制数Ｎ可以写成：N = 2e×M

其中，M称为浮点数的尾数，是一个纯小数；e是比例因子的指数，称为浮点数的指数，是一个整数。在计算机中表示一个浮点数时，一是要给出尾数M，用小数形式表示；二是要给出指数e，用整数形式表示，常称为阶码。尾数部分给出有效数字的位数，因而决定了浮点数的表示精度；阶码部分指明了小数点在数据中的位置，因而决定了浮点数的表示范围。浮点数也是有符号数。

规格化浮点数

若不对浮点数的表示做出明确规定，同一个浮点数的表示就不是惟一的。例如：

(1.75)10 = (1.11)2

= 1.11×2^0

= 0.111×2^1

= 0.0111×2^2

= 0.00111×2^3

※、10进制小数二进制变换方法参照<http://my.oschina.net/dubenju/blog/535161>

为了提高数据的表示精度，需要充分利用尾数的有效位数。当尾数的值不为0时，尾数域的最高有效位应为1，否则就要用修改阶码同时左右移动小数点的办法，使其变成符合这一要求的表示形式，这称为浮点数的规格化。

IEEE-754标准浮点格式

在IEEE-754标准出现之前，业界并没有一个统一的浮点数标准，相反，很多计算机制造商都在设计自己的浮点数规则以及运算细节。为了便于软件的移植，浮点数的表示格式应该有一个统一的标准。

1985年，IEEE（Institute of Electrical and Electronics Engineers，美国电气和电子工程师协会）提出了IEEE-754标准，并以此作为浮点数表示格式的统一标准。目前，几乎所有的计算机都支持该标准，从而大大改善了科学应用程序的可移植性。

IEEE标准从逻辑上采用一个三元组{S, E, M}来表示一个数N，它规定基数为2，符号位S用0和1分别表示正和负，尾数M用原码表示，阶码E用移码表示。根据浮点数的规格化方法，尾数域的最高有效位总是1，由此，该标准约定这一位不予存储，而是认为隐藏在小数点的左边，因此，尾数域所表示的值是1.M（实际存储的是M），这样可使尾数的表示范围比实际存储多一位。为了表示指数的正负，阶码E通常采用移码方式来表示，将数据的指数e 加上一个固定的偏移量后作为该数的阶码，这样做既可避免出现正负指数，又可保持数据的原有大小顺序，便于进行比较操作。

目前，大多数高级语言都按照IEEE-754标准来规定浮点数的存储格式。IEEE-754标准规定了单精度浮点数(<http://my.oschina.net/dubenju/blog/425019>)和双精度浮点数(<http://my.oschina.net/dubenju/blog/543120>)。

单精度格式（32位）：符号位（S）1位；阶码（Ｅ）8位，阶码的偏移量为127（7FH）；尾数（M）23位，用小数表示，小数点放在尾数域的最前面；

双精度格式（64位）：符号位（S）1位；阶码（Ｅ）11位，阶码的偏移量为1023（3FFH）；尾数（M）52位，用小数表示，小数点放在尾数域的最前面。

在IEEE-754标准中，一个规格化的32位浮点数X的真值可表示为：

X = (-1)s×(1.M)×2^E-127 e = E-127 （式2-9）

在IEEE-754标准中，一个规格化的64位浮点数X的真值可表示为：

X = (-1)s×(1.M)×2^E-1023 e = E-1023 （式2-10）

使用单精度浮点数

flt.c

#include <stdio.h>

int main() {

float pi = 3.14159265;

float d = 30.0;

float a = d \* pi;

printf("%8.8f\n", a);

return 0;

}

C:\prj\c>flt.exe

94.24777985

C:\prj\c>

Flt.java

**package** javay.test.java;

**public** **class** Flt {

**public** **static** **void** main(String[] args) {

**float** **pi** = 3.14159265f;

**float** **d** = 30.0f;

**float** **a** = d \* pi;

System.***out***.println(a);

}

}

94.24778

3.14159265 \* 30.0=94.2477795

使用双精度浮点数

dbl.c

#include <stdio.h>

int main() {

double pi = 3.141592653589793;

double d = 30.0;

double a = d \* pi;

printf("%16.16f\n", a);

return 0;

}

C:\prj\c>dbl.exe

94.2477796076937860

C:\prj\c>

Dbl.java

**package** javay.test.java;

**public** **class** Dbl {

**public** **static** **void** main(String[] args) {

**double** **pi** = 3.141592653589793;

**double** **d** = 30.0;

**double** **a** = d \* pi;

System.***out***.println(a);

}

}

94.24777960769379

3.141592653589793×30.0=94.25777960769379

浮点数计算的精度误差问题

从上面的两个例子都可以看出，使用浮点数进行计算时，超过一定范围是出现误差。原因是超出精度范围，无法精确计算。

float和double的精度是由尾数的位数来决定的。浮点数在内存中是按科学计数法来存储的，其整数部分始终是一个隐含着的“1”，由于它是不变的，故不能对精度造成影响。

float：2^23 = 8388608，一共七位，这意味着最多能有7位有效数字，但绝对能保证的为6位，也即float的精度为6~7位有效数字；

double：2^52 = 4503599627370496，一共16位，同理，double的精度为15~16位。

四则运算误差

#include <stdio.h>

int main() {

printf("%15.15f\n", 0.05+0.01);

printf("%15.15f\n", 1.0-0.42);

printf("%15.15f\n", 4.015\*100);

printf("%15.15f\n", 123.3/100);

return 0;

}

结果：

C:\prj\c>dbl.exe

0.060000000000000

0.580000000000000

401.499999999999940

1.233000000000000

C:\prj\c>

**package** javay.test.java;

**public** **class** Testb {

**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(123.3/100);

}

}

0.060000000000000005

0.5800000000000001

401.49999999999994

1.2329999999999999

解决浮点数精确计算有误差的方法

由于浮点数的存储形式带来的误差是无法避免和解决的。在Java中通常使用BigDecimal来解决精确的计算的。

BigDecimal(double val)

BigDecimal(String val)

BigDecimal用哪个构造函数呢？

很简单，double的存储就不精确的，所以尽量使用String的那个。

BigDecimal的比较

**package** javay.test.java;

**import** java.math.BigDecimal;

**public** **class** TestBigDecimal {

**public** **static** **void** main(String[] args) {

BigDecimal **a** = **new** BigDecimal("15.0");

BigDecimal **b** = **new** BigDecimal("15.00");

BigDecimal **c** = **new** BigDecimal(15);

BigDecimal **d** = **new** BigDecimal("15.0");

System.***out***.println("(a == b)=" + (a == b));

System.***out***.println("a.equals(b)=" + a.equals(b));

System.***out***.println("a.compareTo(b)=" + a.compareTo(b));

System.***out***.println("(a == c)=" + (a == c));

System.***out***.println("a.equals(c)=" + a.equals(c));

System.***out***.println("a.compareTo(c)=" + a.compareTo(c));

System.***out***.println("(a == d)=" + (a == d));

System.***out***.println("a.equals(d)=" + a.equals(d));

System.***out***.println("a.compareTo(d)=" + a.compareTo(d));

}

}

(a == b)=false

a.equals(b)=false

a.compareTo(b)=0

(a == c)=false

a.equals(c)=false

a.compareTo(c)=0

(a == d)=false

a.equals(d)=true

a.compareTo(d)=0

看！你看到了什么？在某些时候compareTo是优秀的。

（完）