Модуль 1. Основы программирования

Тема* 1.5. Представление отрицательных и вещественных чисел. Поразрядные операции

Оглавление

1.5. Представление отрицательных и вещественных чисел. Поразрядные операции

1.5.1. Представление отрицательных целых чисел

Проблема хранения отрицательных чисел в компьютере в том, что нельзя использовать знак минус. Отрицательные числа должны, также как и положительные состоять **только из нулей и единиц**. При этом надо понимать, что данное представление возможно, если число разрядов на представление числа фиксировано.

Естественной идеей является отведение какого-нибудь разряда, (например, старшего, самого левого под знак). Число -4 (-0100) можно записать как 1100, а положительное число 4 будет храниться как 0100 (здесь и далее мы для упрощения примеров мы будем рассматривать все на 4 разрядах (говорят на четырехбитной разрядной сетке).

Рассмотренный способ называется **прямой код**. Он интуитивно-понятен, но вычисления производить в нем сложно. Недостаточно их складывать “как обычно”. Для положительных все работает хорошо:

```
   0011
+  0010
   ----
   0101
```

(3 плюс 2 получается 5)

Но аналогичные действия для чисел с разными знаками или двумя отрицательными дают неверный результат:

```
   0011
+  1010
   ----
   1101
```

(3 плюс -1 получается -5)

Проблему можно решить, выполняя для чисел с разными знаками другие действия: в этом случае можно определять большое по модулю число, производить вычитание чисел и присваивать разности знак большего по модулю числа.

Но есть более простой универсальный способ – представлять числа в так называемом **дополнительном коде**. Причем положительные числа в дополнительном коде представляются так же как и в прямом, но для отрицательных чисел представление меняется.

Для получения дополнительного кода отрицательного числа сначала его представляют в **обратном коде** (еще его называют дополнением до единицы - one’s complement) - это инвертирование прямого кода (поэтому его еще называют инверсный код). Чтобы получить обратный код из прямого во всех разрядах кроме знакового все нули заменяются на единицы, а единицы на нули.

Дополнительный код числа, или дополнение до двойки (two’s complement) - это обратный код, к которому прибавлена единица.

При этом помним, что **положительные числа в прямом, обратном и дополнительном кодах выглядят одинаково**!

Пример 1

Представим -2 : 1010 (-2 в прямом коде)

Инвертируем: 1101 (-2 в обратном коде)

Прибавляем 1: 1110 (-2 в дополнительном коде)

При этом свойство прямого кода сохраняется – единица в старшем разряде свидетельствует об отрицательности числа.

Удивительно, что обратное преобразование можно произвести теми же самыми действиями, не вычитать единицу, а потом инвертировать, а прибавить единицу и инвертировать:

1110 (-2 в дополнительном коде)

Инвертируем: 1001

Прибавляем 1: 1010 (-2 в прямом коде)

Это становится ясно, если понять что дополнительный код – именно дополняет число до степени двойки. Т.е. это величина, полученная вычитанием числа из ближайшей наибольшей степени двух (из 2^N для N-битного дополнения до 2).

Действительно если число в обратном коде сложить с исходным по модулю, то получится число состоящее из всех единиц. Если добавить к нему 1 – получим следующую степень двойки.

```
  1101
+ 0010
  ----
  1111
+ 0001
  ----
  0000
```

(в разрядной сетке останутся все нули, левая единица «уйдет»)

Все целочисленные типы Java хранят числа в дополнительном коде.

Чтобы легче понять идею дополнения числа до степени двойки рассмотрим пример представления в дополнительном коде чисел в десятичной системе:

10-чная система счисления	*10-чная система счисления, дополнительный код*
10	*0010*
...	*...*
2	*0002*
1	*0001*
0	*0000*
-1	*9999*
-2	*9998*
...	*...*
-10	*9990*

Обратите внимание, что для нуля в этом коде только одно представление: 0000. Число 1000 отрицательное при преобразовании в дополнительный код оно переходит в себя – это -8. Теперь понятно, почему во всех диапазонах в таблице отрицательных чисел на одно больше.

Естественно, при фиксированном количестве разрядов возникает переполнение. Например: 7 + 7

```
  0111
+ 0111
  1110
```

В старшем разряде единица – число отрицательное. Найдем его модуль: 0001 + 1 = 0010 – два.

То есть при сложении двух семерок на четырехразрядной двоичной сетке получается минус 2. (Действительно, 2 – это дополнение к 14: 16 -2 = 14)

Пожалуй, самое главное, что нужно учитывать при работе с действительными числами, это погрешность, которая неизбежно возникает. Даже конечные десятичные дроби представляются в двоичной системе периодическими дробями и при представлении происходит отсечение.

Проверьте, что выведет команда:

```
out.println (3.0 - 2.1);
```

Если работать с денежными суммами при помощи вещественных чисел, представляя копейки дробной частью, может возникать расхождение в копейку. Чтобы округлить полученную в вычислениях сумму до копеек, можно умножить число на 100, добавить к нему 0.5, отсечь целую часть и поделить на 100. Если исходное число было с недостатком, скажем, 3.46499999.... вместо 3.465, то в результате этих действий мы получим 3.46, хотя должны получить 3.47. Потом, например, при умножении на большое число это может превратиться в серьезную ошибку.

Роберт Мартин в своей книге "Чистый код" пишет: «Использовать числа с плавающей точкой для представления денежных сумм — почти преступление».

Для работы с денежными суммами в Java можно пользоваться целыми числами, производя вычисления с копейками отдельно или использовать объекты класса BigDecimal

На Хабре http://habrahabr.ru/post/124233/ есть перевод интерсной статьи “Мои любимые ошибки в программировании”, в которой обсуждается и эта проблема на примере любопытной реальной истории (раздел “Ошибки одного пенса”)

1.5.2. Представление вещественных чисел

Для представления вещественных чисел принят способ представления **с плавающей точкой**.

Этот способ представления опирается на нормализованную (экспоненциальную) запись действительных чисел.

Нормализованная двоичная запись отличного от нуля действительного числа - это запись вида

a= ± m·2q,

где q - целое число (положительное, отрицательное или ноль), m – правильная двоичная дробь, у которой первая цифра после запятой не равна нулю, то есть $1/p \le m < 1$.

При этом m называется **мантиссой** числа, q - **порядком** числа.

Пример 2

$3{,}1415926 = 0{,}31415926 \cdot 10^1$;

$1000=0{,}1 \cdot 10^4$;

$0{,}123456789 = 0{,}123456789 \cdot 10^0$;

$0{,}0000107_8 = 0{,}1078 \cdot 8^{-4}$; (порядок записан в 10-й системе)

$1000{,}0001_2 = 0{,}100000012 \cdot 2^4$.

Так как число ноль не может быть записано в нормализованной форме в том виде, в каком она была определена, то считаем, что нормализованная запись нуля в десятичной системе будет такой: $0 = 0{,}0 \cdot 10^0$.

При представлении чисел с плавающей запятой часть разрядов ячейки отводится для записи порядка числа, остальные разряды - для записи мантиссы. По одному разряду в каждой группе отводится для изображения знака порядка и знака мантиссы. Для того, чтобы не хранить знак порядка, был придуман так называемый смещённый порядок, который рассчитывается по формуле

2^{a-1}+ истинный порядок, где a - количество разрядов, отводимых под порядок.

Пример 3

Если истинный порядок равен -5, и на хранение порядка отводится 1 байт, тогда смещённый порядок будет равен 127-5=122.

Таким образом представление числа можно сделать так:

- перевести число из p-ичной системы счисления в двоичную;
- представить двоичное число в нормализованной экспоненциальной форме;
- рассчитать смещённый порядок числа;
- разместить знак, порядок и мантиссу в соответствующие разряды сетки.

Пример 4

Представим число -25,625 в машинном виде с использованием 4 байтового представления (где 1 бит отводится под знак числа, 8 бит - под смещённый порядок, остальные биты - под мантиссу).

1. $25_{10}=100011_2$
   a. $0{,}625_{10}=0{,}101_2$
   b. $-25{,}625_{10}= -100011{,}101_2$
2. $-100011{,}101_2= -1{,}00011101_2 * 2^4$
3. СП = 127 + 4 = 131 = 1000011
4. Результат:

Знак числа

1	1	0	0	0	0	0	1	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Смещенный порядок **Мантисса**

1.5.3. Поразрядные операции

Нам уже известно, что вся информация представляется в компьютере двоичном виде нулями и единицами, битами. Но не всем известно, что доступ к конкретным нулям и единицам невозможен. Те, кто считает, что байт это восемь бит, правы лишь отчасти. На самом деле байт – это минимальная единица адресации компьютерной памяти. Для большинства современных компьютеров это действительно восемь бит. То есть прочитать или записать можно только сразу по восемь бит, а команд для чтения и записи бита не существует.

Но есть команды, которые работают **с каждым битом** числа. Например, мы можем заменить все биты в числе на противоположные. Если у нас есть два байта, то мы можем сделать логическую операцию И, ИЛИ или ИСКЛЮЧАЮЩЕЕ ИЛИ с каждой парой битов этих двух байтов. Это так называемые поразрядные операции. (Их не надо путать с логическими && и ||).

Рассмотрим их подробно и сразу поговорим о применениях.

Побитовое отрицание(NOT)

Обозначается символом ~

Это унарная операция, т.е. применяется к одному числу (последовательности битов), которая меняет каждый бит на противоположный.

```
~ 00000000 00000000 00000000 01111011 (123)
=
11111111 11111111 11111111 10000100 (-124)
```

Операция инвертирования применяется в компьютере очень часто. Попробуйте написать программу вычитающую одно число из другого без операции вычитания. В компьютере это выполняется именно так: при помощи побитового отрицания и сложения.

Побитовое И (AND)

Обозначается символом &

Эта операция выставляет значение бита в 1, только в том случае, если соответствующие биты в первом и втором числе равны 1 одновременно

```
00000000 00000000 00000000 01111011 (123)
&
00000000 00000000 00000001 11001000 (456)
=
00000000 00000000 00000000 01001000 (72)
```

Побитовое И зачастую применяют для того, чтобы узнать какое значение 0 или 1 стоит в определенном бите числа:

- Тот бит, значение которого необходимо узнать устанавливается в 1, остальные в 0. Полученное таким образом число назвают **маской**.

- Выполняют операцию побитового И между числом и маской
- Результат даст информацию о конкретном бите числа.

Пример 5

Пусть нужно узнать ноль или единица содержится во втором бите числа X, тогда маска будет иметь вид: 00000100 (число 4).

```
xxxxx?xx
& (поразрядное И)
00000100
--------
00000?00
```

Если искомый бит (?) нулевой, то в байте-результате будут все нули, то есть в результате будет ноль, если нет – то результат будет 4, то есть не ноль.

Побитовое ИЛИ (OR)

Операция обозначается символом |

Выставляет значение 1, если хотя бы один соответствующий бит в первом или во втором числе установлен в 1.

```
00000000 00000000 00000000 01111011 (123)
|
00000000 00000000 00000001 11001000 (456)
=
00000000 00000000 00000001 11111011 (507)
```

Пример 6

При помощи масок можно не только узнать, чему равно значение конкретного бита числа (см. Пример 1), но и установить его значение:

```
xxxxx?xx
| (поразрядное ИЛИ)
00000100
--------
xxxxx1xx
```

Все биты результата будут такими же, что и в исходномчисле X, кроме второго. Второй бит гарантированно станет единицей.

Два последних примера на Java записываются так:

```
X & 4
X | 4
```

Благодаря маскам можно в определенных заранее битах одной переменной сохранять несколько двоичных значений (например, истина/ложь) и считывать их.

Исключающее ИЛИ(XOR)

Обозначается символом **^**

Эта операция выставляет значение в 1, если соответствующий бит равен 1 только в одном из операндов, но не одновременно.

Пример 7

```
00000000 00000000 00000000 01111011 (123)
^
00000000 00000000 00000001 11001000 (456)
=
00000000 00000000 00000001 10110011 (435)
```

Операция XOR имеет много практических применений. Самое известное и часто применяемое из которых – шифрование. Если примерить эту операцию к значению с каким-то ключом (другим значением) первое изменится, но при повторном применении восстанавливается.
Например:
5 ^ 7 → 2 (значение изменилось - защифровалось)
2 ^ 7 → 5 (используя ключ можно расшифровать)
Понятно, что символы текста можно рассматривать как их числовые коды.
Идея шифрования при помощи XOR проста: выполним операцию XOR каждого символа с ключом. Текст станет нечитаемым - зашифруется. Получатель, знающий ключ, сможет легко восстановить текст, опять выполняя операцию XOR с каждым символом **с тем же ключом**.
Конечно, ключ можно подобрать. Но систему можно сильно улучшить, если выбирать для каждого шифруемого символа новый ключ из известной передающему и принимающему последовательности. Например, можно шифровать при помощи текста этого пособия. В этом случае по сути пособие выступает в роли некого сложного ключа. Расшифровка будет возможна, только если злоумышленник будет знать, как зашифровано сообщение и иметь это пособие. Такой способ кодирования называется «одноразовый блокнот».

Интересный пример практического применения операции исключающего ИЛИ:
Поменять местами значения двух переменных без использования третьей:
a=a^b;
b=a^b;
a=a^b;

Знаковый оператор сдвига влево <<

Операция, в результате которой все биты первого операнда смещаются влево и число справа дополняется нулем. Это операция повторяется столько раз, сколько задано вторым операндом. Один сдвиг влево соответствует умножению на 2.

Таким образом, запись 5 << 3 соответствует 5*8 (2 встепени3). Результатом будет 40.

Если оператор применяется к числу, умножение на 2 которого будет больше максимального значения int (2147483647), то в результате будет отрицательное число. Это происходит потому, что крайний левый бит, который отвечает за знак числа, выставляется в единицу, что соответствует отрицательным числам.

Пример 8

```
01111111 11111111 11111111 11111111 (2147483647)
<<
11111111 11111111 11111111 11111110 (-2 в дополнительном коде)
```

Знаковый оператор сдвига вправо >>

Операция, в результате которой все биты первого операндасмещаются вправо и число слева дополняется нулем, если число положительное и единицей, если отрицательное. Это повторяется столько раз, сколько задано вторым операндом. Один сдвиг вправо соответствует делению на 2. Если делится нечетное число, то остаток отбрасывается для положительных чисел и сохраняется для отрицательных.

Таким образом, 42 >> 3 = 5. Число 42 было поделено нацело на 8 (2 в степени 3).

Операции сдвига выполняются гораздо быстрее операций умножения и деления и по возможности следует применять именно их.

Пример 9

Фрагмент программы на Java с примерами быстрого умножения и деления на 2 с помощью операций поразрядного сдвига :

```
int i = 192;  //  00000000 00000000 00000000 11000000 (192)
i <<= 1;      //  00000000 00000000 00000001 10000000 (384)
i >>= 2;      //  00000000 00000000 00000000 01100000 (96)
int j =- 192; //  11111111 11111111 11111111 01000000 (-192)
j <<= 1;      //  11111111 11111111 11111110 10000000 (-384)
j >>= 2;      //  11111111 11111111 11111111 10100000 (-96)
```

Как и для других операций для поразрядных существуют операции с присваиванием.

Беззнаковый оператор сдвига >>>

Операция, в результате которой все биты первого операнда смещаются вправо и число слева дополняется нулем, даже если операция выполняется с отрицательными числами. Операция повторяется столько раз, сколько задано вторым операндом.

Отсюда и название оператора — беззнаковый. В результате применения оператора всегда получается положительное число, т.к. в Java левый бит отвечает за знак числа. Операция так же, как и знаковый оператор сдвига вправо, соответствует делению числа на два за исключением первого сдвига в отрицительном числе.

11111111 11111111 11111111 10000101 (-123)
>>>
01111111 11111111 11111111 11000010 (2147483586)

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Ильину Владимиру Владимировичу.