链滴

Java 移位运算符介绍

作者：teneous

原文链接：https://ld246.com/article/1557365230125

来源网站：链滴

<h2 id="位算符">位算符</h2>
@(syoka)
Preface:我们知道在计算机中,数据都是以 2 进制存储的，且越贴近机器语，运行效率就越高，但是同时会变得晦涩难懂。 但是如果在程序中能够使用位运算还是尽量使用位运。
今天主要是针对 Java 中的移位运算符进行简单介绍，我将以举例的形式来帮助大家更加理解，希望完对大家有所帮助。 如果不想看过程可以直接调至文章末尾结论

<hr>
<h3 id="简单说明">简单说明</h3>

int 作为 Java 基本类型中的一个，将有幸成为今天用例的主角
int 不管在 32X 还是 64X 机器中都占位 4 个字节也就是 32 位（注:1 个字节 = 8 位）
左边 8 位是高位，右边 8 位属于低位
例：十进制 10 对应的二进制数就是 <code>00000000 00000000 00000000 00001010 = 2^3 + ^1</code>

<h3 id="区别">区别</h3>

有符号偏移和无符号偏移 的区别在于偏移时符号位否也跟着偏移。（2 进制最高位为符号位 1 为正数，0 为负数）
有符号右偏移高位补符号位，有符号左偏移,低位补 0
无符号右偏移高位补 0

<hr>
<h3 id="用例介绍--我们以正负10来介绍">用例介绍: 我们以正负 10 来介绍</h3>
<h4 id="正数10情况">正数 10 情况</h4>
（2 进制: [0]0000000 00000000 00000000 00001010 ）
<h3 id="有符号偏移">有符号偏移</h3>
<h5 id="有符号右移----10--2-">有符号右移 >> (10>>2)</h5>
<blockquote>
按照规则符号位不动，高位补符号位(0),低位超出位数舍弃。
[0]0000000 00000000 00000000 00000010
结果等于 10 / 2^2 = 2 因此随着有符号正数的右偏移，数值会越来越小
</blockquote>
<h5 id="有符号左移---10--2-">有符号左移 <<(10<<2)</h5>
<blockquote>
按照规则符号位不动，低位补(0),高位超出位数舍弃
[0]0000000 00000000 00000000 00101000
结果等于 10 * 2^2 = 40 因此随着有符号正数的左偏移，数值会越来越大
</blockquote>
<h4 id="负数10情况">负数 10 情况</h4>
（2 进制: [1]0000000 00000000 00000000 00001010 ）
注:负数偏移时需要原码 --> 反码 --> 补码 --> 偏移 --> 反码 --> 补码 --> 才等于偏移后的原码 #
<h4 id="有符号右移-----10--2-">有符号右移 >> (-10>>2)</h4>
<blockquote>
负数移位时需要先取得其补码
[1]1111111 11111111 11111111 11110101 获取反码（按位取反）
[1]1111111 11111111 11111111 11110110 获取补码（反码基础上 +1）

在补码的基础上，按照规则符号位不动，高位补符号位(1),低位超出位数舍弃<b
>
[1]1111111 11111111 11111111 11111101 获取补码偏移后的值
但这不是我们想要的最终结果，我们还需要在偏移后的补码基础上再进行反码补码从而获取原码
[1]0000000 00000000 00000000 00000010 再取反码（按位取反）
[1]0000000 00000000 00000000 00000011 再取补码（反码基础上 +1）<br

结果等于-3 因此随着有符号负数的右偏移，数值会越来越大
</blockquote>
<h5 id="有符号左移-----10--2-">有符号左移 << (-10<<2)</h5>
<blockquote>
负数移位时需要先取得其补码
[1]1111111 11111111 11111111 11110101 获取反码（按位取反）
[1]1111111 11111111 11111111 11110110 获取补码（反码基础上 +1）<br

在补码的基础上，按照规则符号位不动，低位补(0),高位超出位数舍弃
[1]1111111 11111111 11111111 11011000 获取补码偏移后的值
但这不是我们想要的最终结果，我们还需要在偏移后的补码基础上再进行反码补码从而获取原码
[1]0000000 00000000 00000000 00100111 再取反码（按位取反）
[1]0000000 00000000 00000000 00101000 再取补码（反码基础上 +1）<br

结果等于-40 因此随着有符号负数的左偏移，数值会越来越小
</blockquote>
<hr>
<h3 id="无符号偏移">无符号偏移</h3>
<h4 id="负数10情况-">负数 10 情况</h4>
（2 进制: [1]0000000 00000000 00000000 00001010</stron
> ）
正数情况下的无符号偏移和有符号偏移的情况是一样的，因此我们只需看看负数情况即可。
<h5 id="无符号右移------10---2-">无符号右移 >>> (-10>>>2)</h5>
<blockquote>
负数移位时需要先取得其补码
[1]1111111 11111111 11111111 11110101 获取反码（按位取反）
[1]1111111 11111111 11111111 11110110 获取补码（反码基础上 +1）<br

【符号位也要偏移】，高位补符号位(0),低位超出位数舍弃
[0]011111 11111111 11111111 11111101 获取补码偏移后的值
我们发现数据变正数，正数的补码反码都是它自己
假设现在数是 01000000 00000000 00000000 00000000 = 2^30
它与原来的数 00111111 11111111 11111111 11111101 刚好差 3
结果 1073741821 是一个很大的数
</blockquote>
<h5 id="不存在无符号左移------10---2-">不存在无符号左移 <<< (-10<<<2)</h5>
因为无符号左移和有符号左移一样都是低位补 0，结果来说一样的，没有必要
结论

负数偏移是通过补码进行偏移的(如果高位是 1，则在偏移完后在进行一次反码补码)
有符号偏移：左偏移是增大，右偏移是变小（负数则相反）
有符号偏移：左偏移低位补 0，右偏移高位补符号位
JAVA 不支持无符号类型，但是支持无符号偏移，且只能右移
无符号右移比较特殊，高位是补 0，在偏移完后成为正数，因此会变为一个很大的数
int -1 的二进制是 16 个 1 [11111111 11111111 11111111 11111111]常参与偏移，与或，异
运算 例：snowflake 算法中求每个生成部分的最大值

例：snowflake 算法中求每个生成部分的最大值
<pre><code class="highlight-chroma">/**
* 每一部分占用的数
*/
private final static long SEQUENCE_BIT = 12; //序列号占用的位数
private final static long MACHINE_BIT = 5; //机器标识占用的位数
private final static long DATACENTER_BIT = 5;//数据中心占用的位数

/**
* 每一部分的最大值

*/
private final static long MAX_DATACENTER_NUM = ~(-1L << DATACENTER_BIT);
private final static long MAX_MACHINE_NUM = ~(-1L << MACHINE_BIT);
private final static long MAX_SEQUENCE = ~(-1L << SEQUENCE_BIT);

</code></pre>
我们解析下：
第一步：
假设这里只给出低 8 位 -1 左偏 5 位则低位是[1]110 0000，再进行反码（[1]001 1111），再进行码（[1]010 0000）此时的值位-2^5
第二步：
再取其取非运算（[1]010 0000） = 再反码 ([1]101 1111)，再进行补码（[1]110 0000）在取反（001 1111） = 31
这里我当时想了很久 ~-32 进行运算的时候要把-32 再重新算补码再取反可以。
有错误或是不足之处请多多指正。