Integer.highestOneBit(int i)方法的作用与底层实现

在Integer类中有这么一个方法，你可以给它传入一个数字，它将返回最大的小于等于这个数字的一个2的幂次方数。这个方法就是highestOneBit(int i)。

比如下面的Demo，注意方法的输入与返回值：

```
System.out.println(Integer.highestOneBit(15));  // 输出8
System.out.println(Integer.highestOneBit(16));  // 输出16
System.out.println(Integer.highestOneBit(17));  // 输出16
```

这个方法的实现代码量也是非常少的：

```
public static int highestOneBit(int i) {
    // HD, Figure 3-1
    i |= (i >>  1);
    i |= (i >>  2);
    i |= (i >>  4);
    i |= (i >>  8);
    i |= (i >> 16);
    return i - (i >>> 1);
}
```

接下来，我们就来详细分析一下这块代码的逻辑。

首先，对于这个方法的功能：**给定一个数字，找到小于或等于这个数字的一个2的幂次方数。**

如果我们要自己来实现的话，我们需要知道：**怎么判断一个数字是2的幂次方数。**

说真的，我一下想不到什么好方法来判断，唯一能想到的就是一个数字如果把它转换成二进制表示的话，它会有一个规律：**如果一个数字是2的幂次方数，那么它对应的二进制表示仅有一个bit位上是1，其他bit位全为0。**

比如：
十进制6，二进制表示为：0000 0110
十进制8，二进制表示为：0000 1000
十进制9，二进制表示为：0000 1001
所以，我们可以利用一个数字的二进制表示来判断这个数字是不是2的幂次方数。关键代码怎么实现呢？去遍历每个bit位？可以，但是不好。那怎么办？我们还是回头仔细看看Integer是如何实现的吧？

```
public static int highestOneBit(int i) {
    // HD, Figure 3-1
    i |= (i >>  1);
    i |= (i >>  2);
    i |= (i >>  4);
    i |= (i >>  8);
    i |= (i >> 16);
    return i - (i >>> 1);
}
```

我们发现这段代码中没有任何的遍历，只有位运算与一个减法，也就是说它的实现思路和我们自己的实现思路完全不一样，它的思路就是：**给定一个数字，通过一系列的运算，得到一个小于或等于该数字的一个2的幂次方数。**

也就是：如果给定一个数字18，通过运算后，要得到16。
18用二进制表示为：　0001 0010
想要得到的结果(16)是：0001 0000

那么这个运算的过程无非就是将18对应的二进制数中除最高位的1之外的其他bit位都清零，则拿到了我们想要的结果。

那怎么通过位运算来实现这个过程呢？

我们拿18对应的二进制数`0001 0010`来举个例子就行了：
先将`0001 0010`右移1位，
得到`0000 1001`，再与自身进行或运算：
得到`0001 1011`。

再将`0001 1011`右移2位，
得到`0000 0110`，再与自身进行或运算：
得到`0001 1111`。

再将`0001 1111`右移4位，
得到`0000 0001`，再与自身进行或运算：
得到`0001 1111`。

再将`0001 1111`右移8位，
得到`0000 0000`，再与自身进行或运算：
得到`0001 1111`。

再将`0001 1111`右移16位，
得到`0000 0000`，再与自身进行或运算：
得到`0001 1111`。

再将`0001 1111`无符号右移1位，
得到`0000 1111`。

> 关于无符号右移，可以看我之前写的文章。

最后用`0001 1111 - 0000 1111 = 0001 0000`
震惊！得到了我们想要的结果。

其实这个过程可以抽象成这样：
现在有一个二进制数据，`0001****`，我们不关心低位的取值情况，我们对其进行右移并且进行或运算。

先将`0001****`右移1位，
得到`00001***`，再与自身进行或运算：
得到`00011***`。

再将`00011***`右移2位，
得到`0000011*`，再与自身进行或运算：
得到`0001111*`。

再将`0001111*`右移4位，
得到`00000001`，再与自身进行或运算：
得到`00011111`。

后面不用再推算了，到这里我们其实可以发现一个规律：
右移与或运算的目的就是想让某个数字的低位都变为1，再用该结果 减去 该结果右移一位后的结果，则相当于清零了原数字的低位，即得到了我们想要的结果。

到此，只能感叹JDK作者对于位运算的使用已经达到了出神入化的境界了。

如果想学习更多的精彩的Java、分布式、微服务等方面的知识，**请关注微信公众号：1点25**

1点25

微信扫描二维码，关注我的公众号

周瑜　2019-12-01 17:24　1053　2　投诉

分享到：

本文来自
1点25
高并发、分布式、微服务、性能调优
关注

上一篇
一文搞明白位运算、补码、反码、原码

下一篇
String.intern()使用总结

2 条回复

小明快跑　2019-12-19 17:59
方法的描述说成下面的描述比较好把
`Integer类中有这么一个方法，你可以给它传入一个非负整数，*它将返回最大的小于等于这个数字的一个2的幂次方数。`

周瑜　2019-12-01 17:24
回复 @小明快跑
是的，我改一下