01讲二进制：不了解计算机的源头，你学什么编程

我们都知道，计算机的起源是数学中的二进制计数法。可以说，没有二进制，就没有如今的计算机系统。那什么是二进制呢？为什么计算机要使用二进制，而不是我们日常生活中的十进制呢？如何在代码中操作二进制呢？专栏开始，我们就从计算机认知的起源——二进制出发，讲讲它在计算机中的“玄机”。

什么是二进制计数法？

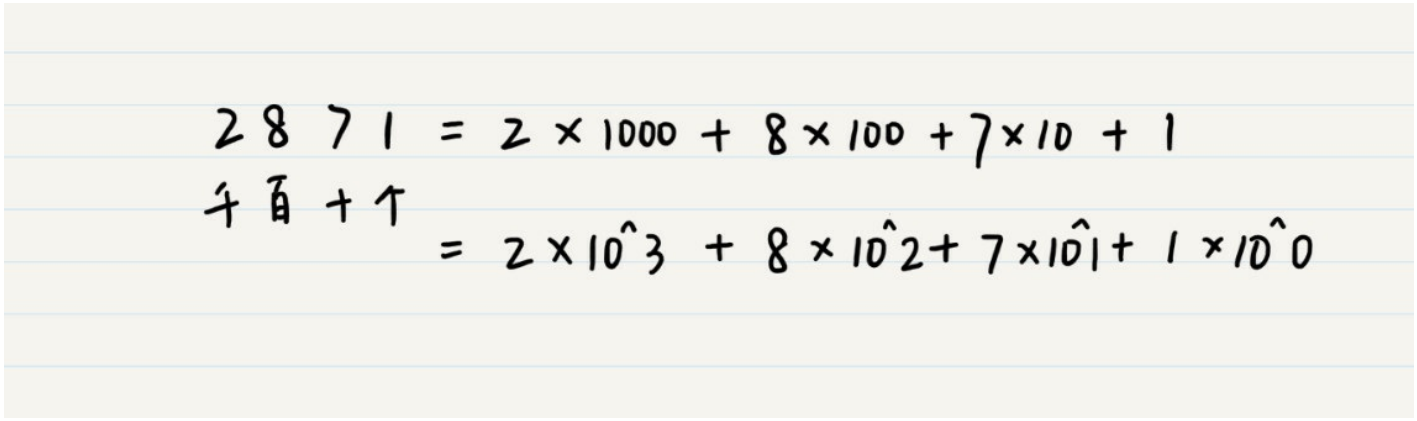
为了让你更好地理解二进制计数法，我们先来简单地回顾一下人类计数的发展史。

原始时代，人类用路边的小石子，来统计放牧归来的羊只数量，这表明我们很早就产生了计数的意识。后来，罗马人用手指作为计数的工具，并在羊皮上画出I、II、III来代替手指的数量。表示一只手时，就写成“V”形，表示两只手时，就画成“VV”形等等。

公元3世纪左右，印度数学家（也有说法是阿拉伯人）发明了阿拉伯数字。阿拉伯数字由从0到9这样10个计数符号组成，并采取**进位制法**，高位在左，低位在右，从左往右书写。由于阿拉伯数字本身笔画简单，演算便利，因此它们逐渐在各国流行起来，成为世界通用的数字。

日常生活中，我们广泛使用的十进制计数法，也是基于阿拉伯数字的。这也是十进制计数法的基础。因此，相对其他计数方法，十进制最容易被我们所理解。

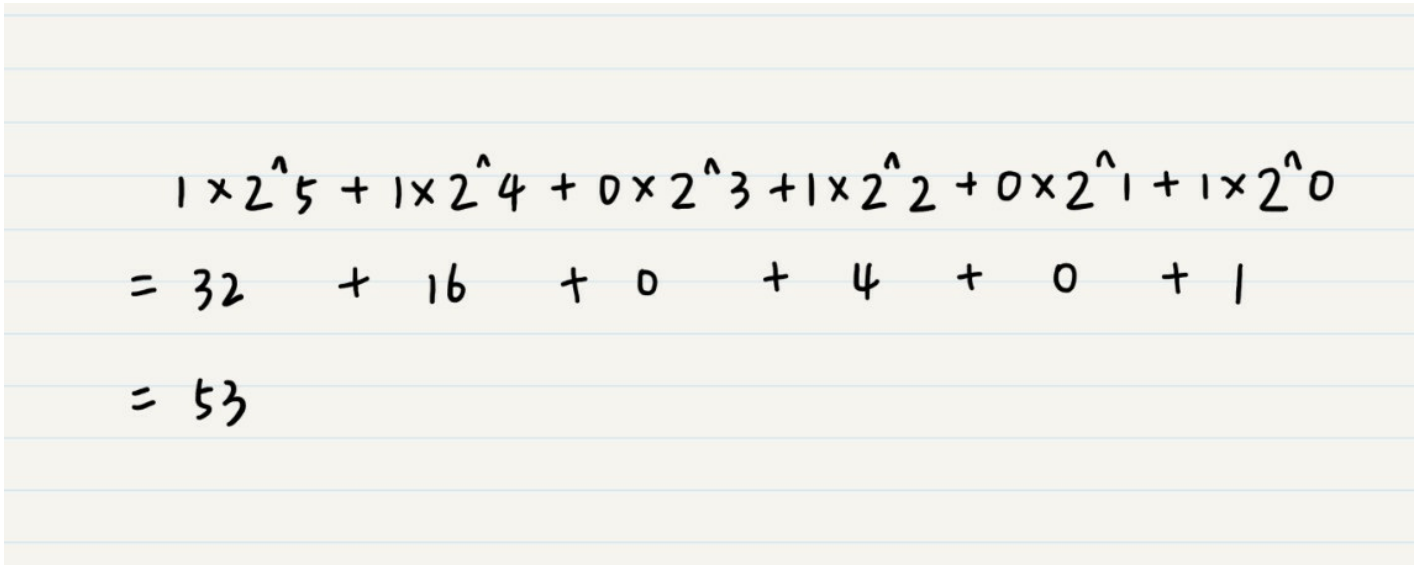
让我们来观察一个数字：2871。



其中^表示幂或次方运算。十进制的数位（千位、百位、十位等）全部都是10^n的形式。需要特别注意的是，任何非0数字的0次方均为1。在这个新的表示式里，10被称为十进制计数法的**基数**，也是十进制中“十”的由来。这个我想你应该好理解，因为这和我们日常生活的习惯是统一的。

明白了十进制，我们再试着用类似的思路来理解二进制的定义。我以二进制数字110101为例，解释给你听。我们先来看，这里110101究竟代表了十进制中的数字几呢？

刚才我们说了，十进制计数是使用10作为基数，那么二进制就是使用2作为基数，类比过来，**二进制的数位就是2^n的形式**。如果需要将这个数字转化为人们易于理解的十进制，我们就可以这样来计算：



按照这个思路，我们还可以推导出八进制（以8为基数）、十六进制（以16为基数）等等计数法，很简单，我在这里就不赘述了。

至此，你应该已经理解了什么是二进制。但是仅有数学的理论知识是不够的，结合相关的代码实践，相信你会有更深刻的印象。

基于此，我们来看看二进制和十进制数在Java语言中是如何互相转换的，并验证一下我们之前的推算。我这里使用的是Java语言来实现的，其他主流的编程语言实现方式都是类似的。

这段代码的实现采用了Java的BigInteger类及其API函数，我都加了代码注释，并且穿插一些解释，你应该可以看懂。

首先，我们引入BigInteger包，通过它和Integer类的API函数进行二进制和十进制的互相转换。

```
import java.math.BigInteger;

public class Lesson1_1 {

   /**

    * @Description: 十进制转换成二进制
    * @param decimalSource
    * @return String
    */
    public static String decimalToBinary(int decimalSource) {
       BigInteger bi = new BigInteger(String.valueOf(decimalSource)); //转换成BigInteger类型，默认是十进制
       return bi.toString(2); //参数2指定的是转化成二进制
    }

    /**
    * @Description: 二进制转换成十进制
    * @param binarySource
    * @return int
    */
    public static int binaryToDecimal(String binarySource) {
       BigInteger bi = new BigInteger(binarySource, 2);  //转换为BigInteger类型，参数2指定的是二进制
       return Integer.parseInt(bi.toString());     //默认转换成十进制
    }
}
```

然后，我们通过一个十进制数和一个二进制数，来验证一下上述代码的正确性。

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

      int a = 53;
      String b = "110101";
      System.out.println(String.format("数字%d的二进制是%s", a, Lesson1_1.decimalToBinary(a))); //获取十进制数53的二
      System.out.println(String.format("数字%s的十进制是%d", b, Lesson1_1.binaryToDecimal(b))); //获取二进制数11010

   }
```

这段代码运行的结果是：十进制数字53的二进制是110101，二进制数字110101的十进制是53。

好了，关于十进制和二进制的概念以及进制之间的相互转换，你应该都很清楚了。既然有十进制，又有二进制，你可能就要问了，为啥计算机使用的是二进制而不是十进制呢?

计算机为什么使用二进制?

我觉得，计算机使用二进制和现代计算机系统的硬件实现有关。组成计算机系统的逻辑电路通常只有两个状态，即开关的接通与断开。

断开的状态我们用“0”来表示，接通的状态用“1”来表示。由于每位数据只有断开与接通两种状态，所以即便系统受到一定程度的干扰时，它仍然能够可靠地分辨出数字是“0”还是“1”。因此，在具体的系统实现中，二进制的数据表达具有抗干扰能力强、可靠性高的优点。

相比之下，如果用十进制设计具有10种状态的电路，情况就会非常复杂，判断状态的时候出错的几率就会大大提高。

另外，二进制也非常适合逻辑运算。逻辑运算中的“真”和“假”，正好与二进制的“0”和“1”两个数字相对应。逻辑运算中的加法（“或”运算）、乘法（“与”运算）以及否定（“非”运算）都可以通过“0”和“1”的加法、乘法和减法来实现。

二进制的位操作

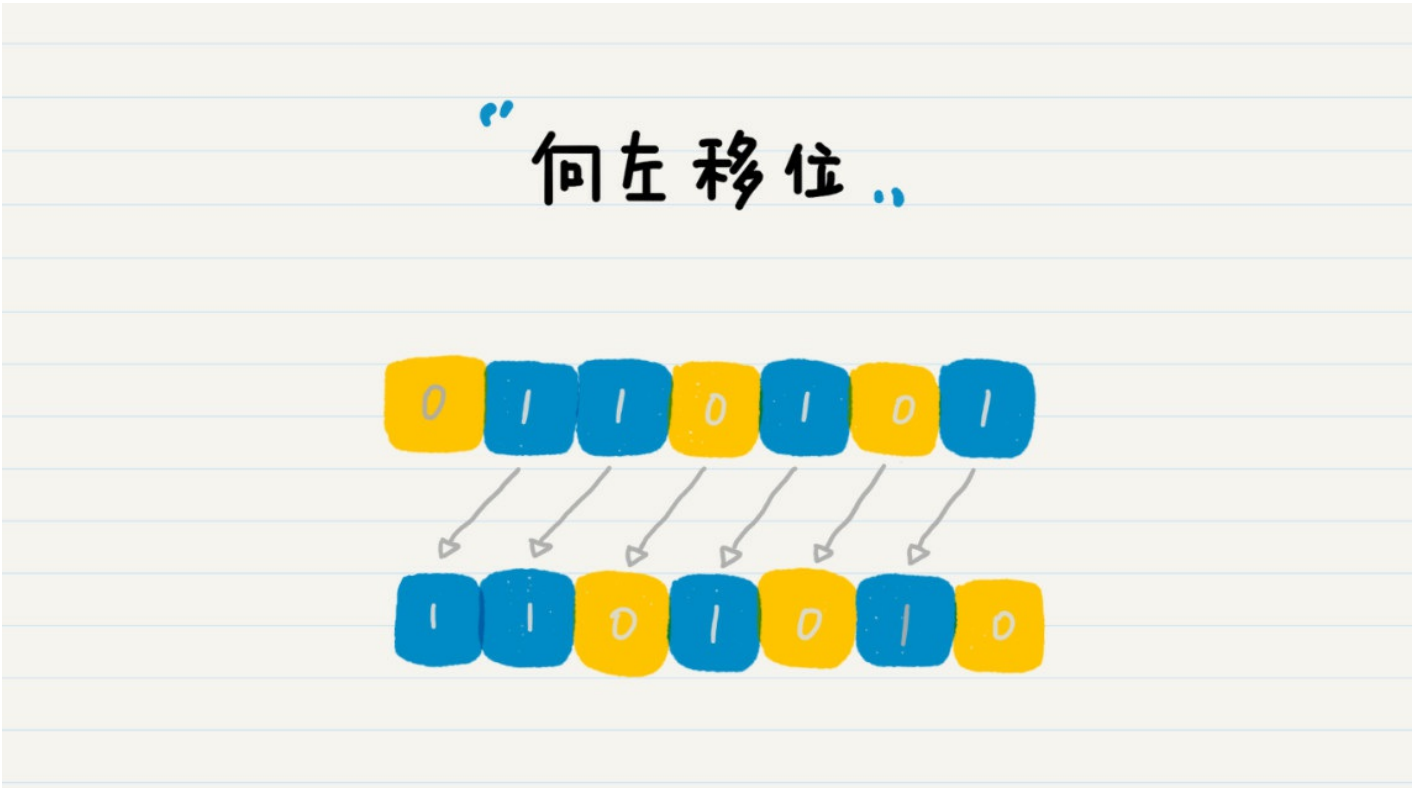
了解了现代计算机是基于二进制的，我们就来看看，计算机语言中针对二进制的位操作。这里的**位操作**，也叫作**位运算**，就是直接对内存中的二进制位进行操作。常见的二进制位操作包括向左移位和向右移位的移位操作，以及“或”“与”“异或”的逻辑操作。下面我们一一来看。

向左移位

我们先来看向左移位。

二进制110101向左移一位，就是在末尾添加一位0，因此110101就变成了1101010。请注意，这里讨论的是数字没有溢出的情况。

所谓**数字溢出**，就是二进制数的位数超过了系统所指定的位数。目前主流的系统都支持至少32位的整型数字，而1101010远未超过32位，所以不会溢出。如果进行左移操作的二进制已经超出了32位，左移后数字就会溢出，需要将溢出的位数去除。



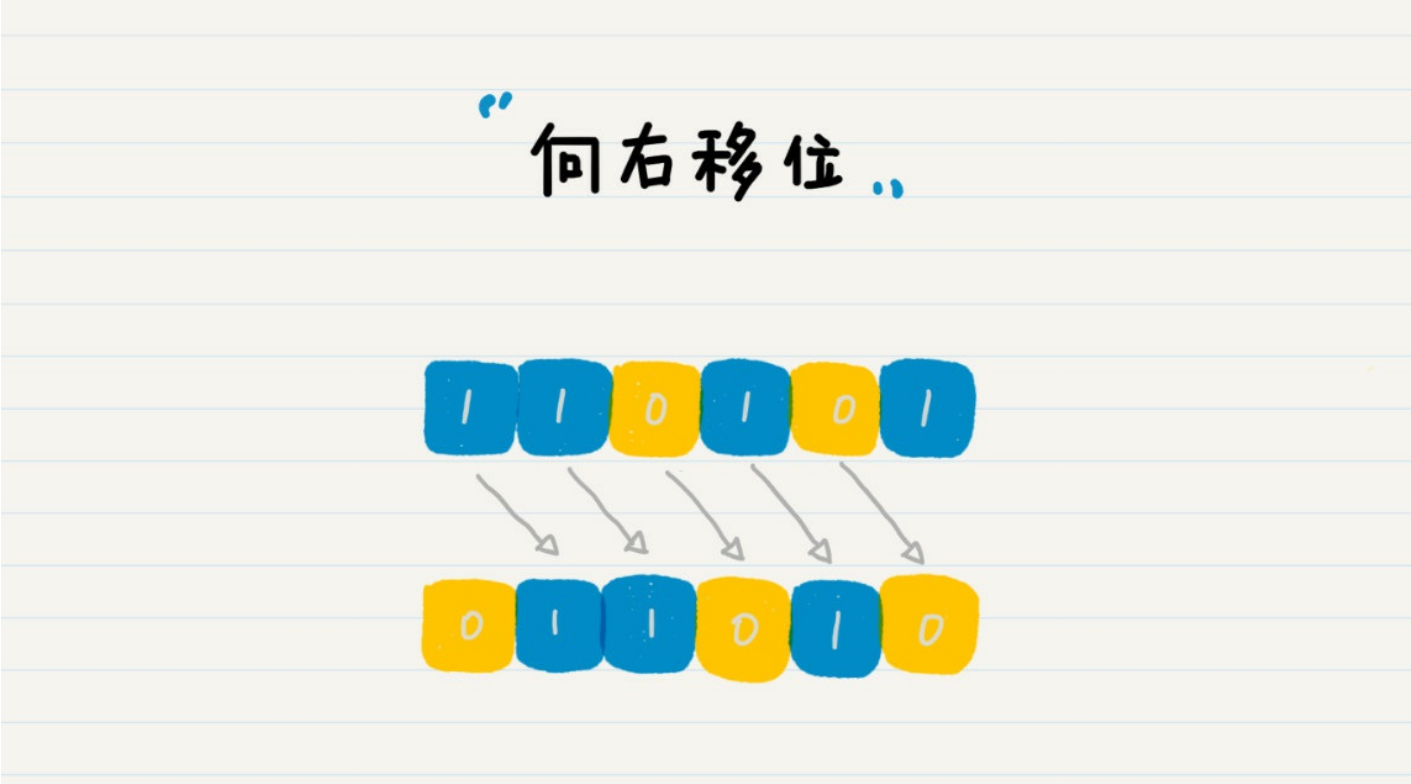
在这个例子中，如果将1101010换算为十进制，就是106，你有没有发现，106正好是53的2倍。所以，我们可以得出一个结

论：**二进制左移一位，其实就是将数字翻倍。**

向右移位

接下来我们来看向右移位。

二进制110101向右移一位，就是去除末尾的那一位，因此110101就变成了11010（最前面的0可以省略）。我们将11010换算为十进制，就是26，正好是53除以2的整数商。所以**二进制右移一位，就是将数字除以2并求整数商的操作**。



下面我们来看看，用代码如何进行移位操作。

```
import java.math.BigInteger;

public class Lesson1_2 {

   /**
    * @Description: 向左移位
    * @param num-等待移位的十进制数, m-向左移的位数
    * @return int-移位后的十进制数
    */
   public static int leftShift(int num, int m) {
      return num << m;
   }

   /**
    * @Description: 向右移位
    * @param num-等待移位的十进制数, m-向右移的位数
    * @return int-移位后的十进制数
    */
   public static int rightShift(int num, int m) {
      return num >>> m;
   }

}
```

然后，我们用一段测试代码验证下结果。

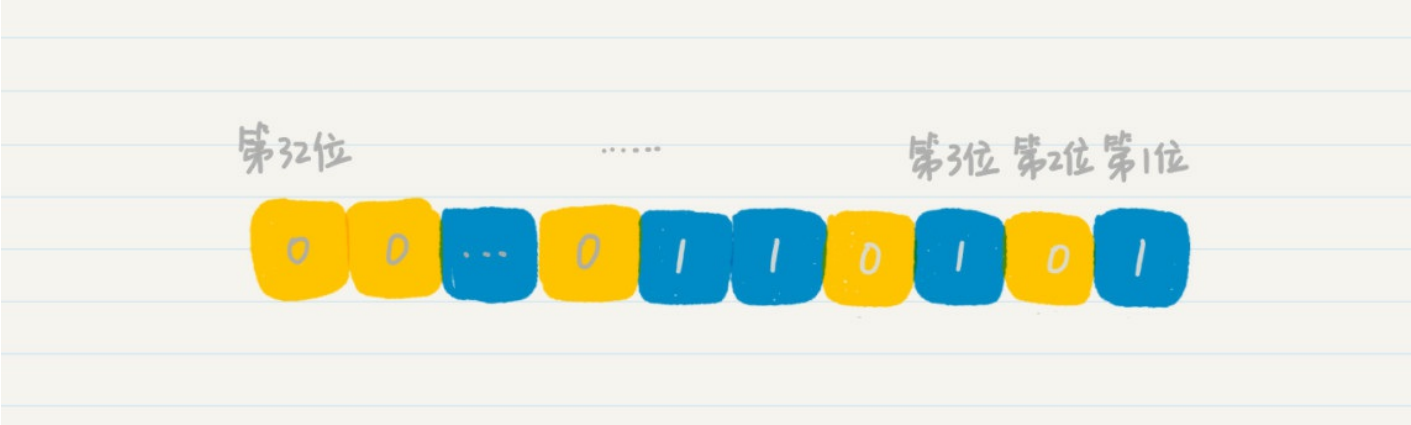
```
public static void main(String[] args) {

      int num = 53;
      int m = 1;
      System.out.println(String.format("数字%d的二进制向左移%d位是%d", num, m, Lesson1_2.leftShift(num, m)));   //
      System.out.println(String.format("数字%d的二进制向右移%d位是%d", num, m, Lesson1_2.rightShift(num, m)));   /

      System.out.println();

      m = 3;
      System.out.println(String.format("数字%d的二进制向左移%d位是%d", num, m, Lesson1_2.leftShift(num, m)));   //
      System.out.println(String.format("数字%d的二进制向右移%d位是%d", num, m, Lesson1_2.rightShift(num, m)));   /

   }
```

这段代码的运行结果是：数字53向左移1位是106；数字53向右移1位是26。数字53向左移3位是424，数字53向右移3位是6。

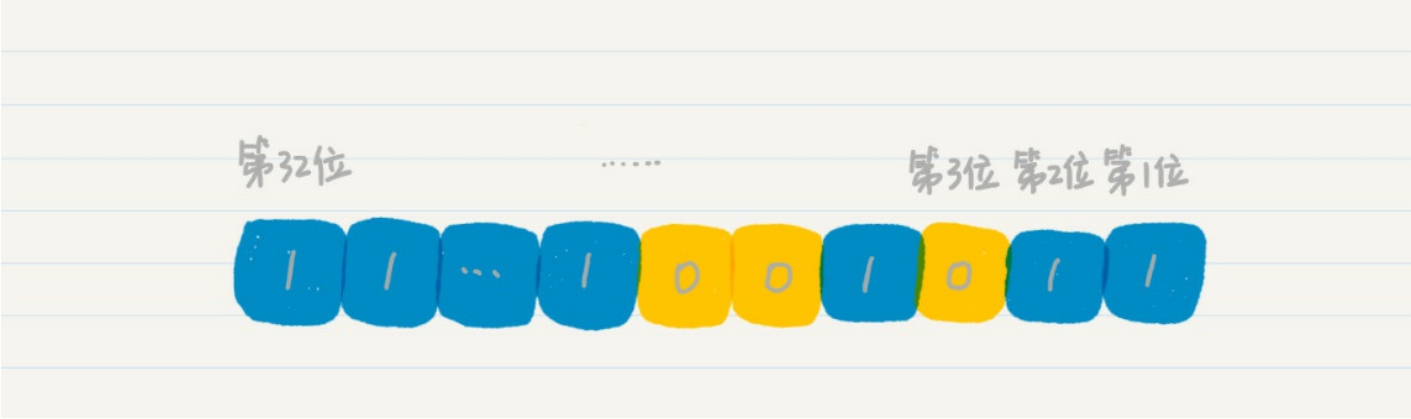
我来解释一下。其中，移位1次相当于乘以或除以2，而移位3次就相当于乘以或除以8（即2的3次方）。细心的话，你可能已经发现，Java中的左移位和右移位的表示是不太一样的。

左移位是<<，那右移位为什么是>>>而不是>>呢？实际上，>>也是右移操作。简单来说，之所以有这两种表达方式，根本原因是Java的二进制数值中最高一位是符号位。这里我给你详细解释一下。

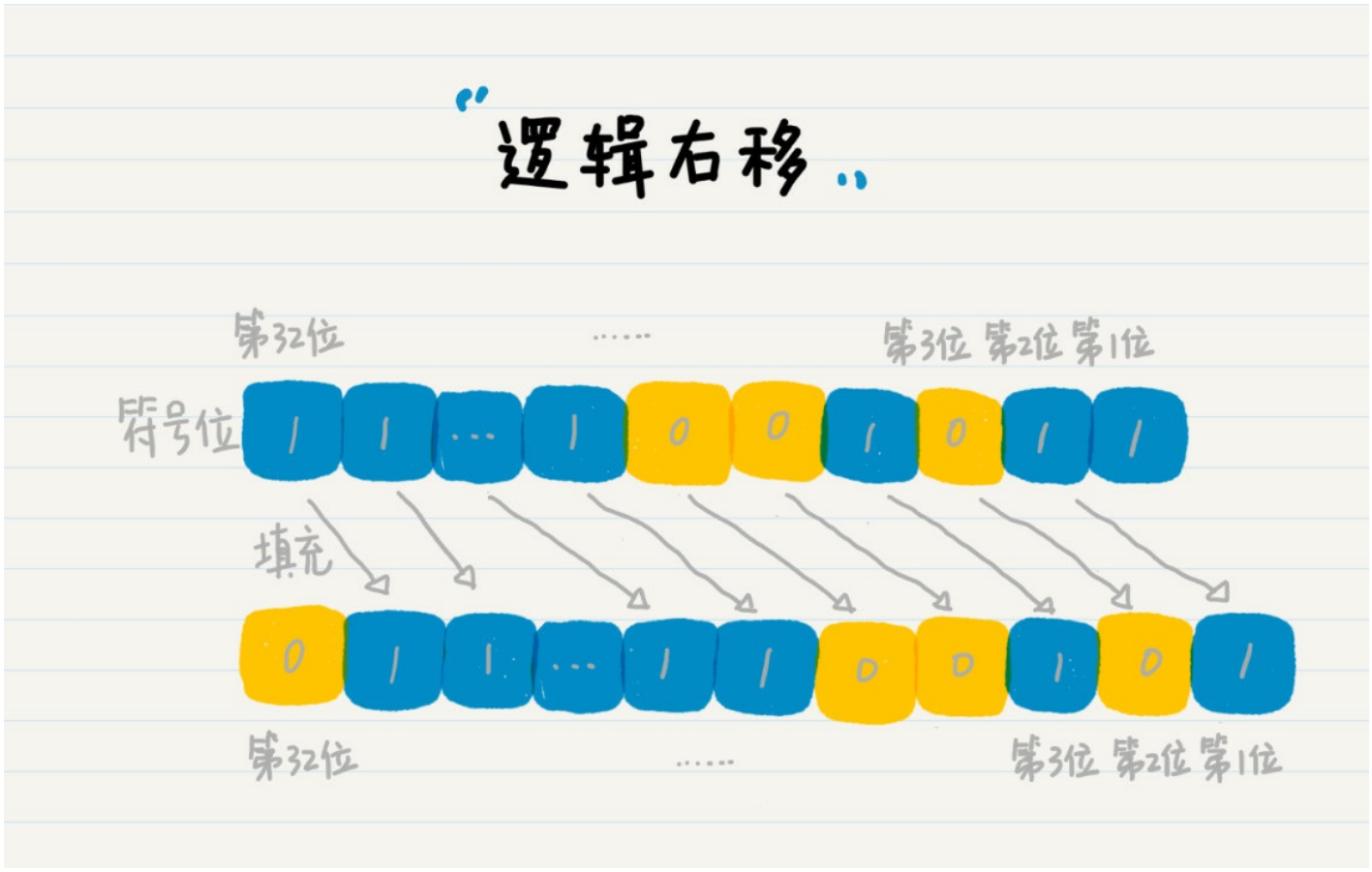
当符号位为0时，表示该数值为正数；当符号位为1时，表示该数值为负数。我们以32位Java为例，数字53的二进制为110101，从右往左数的第32位是0，表示该数是正数，只是通常我们都将其省略。



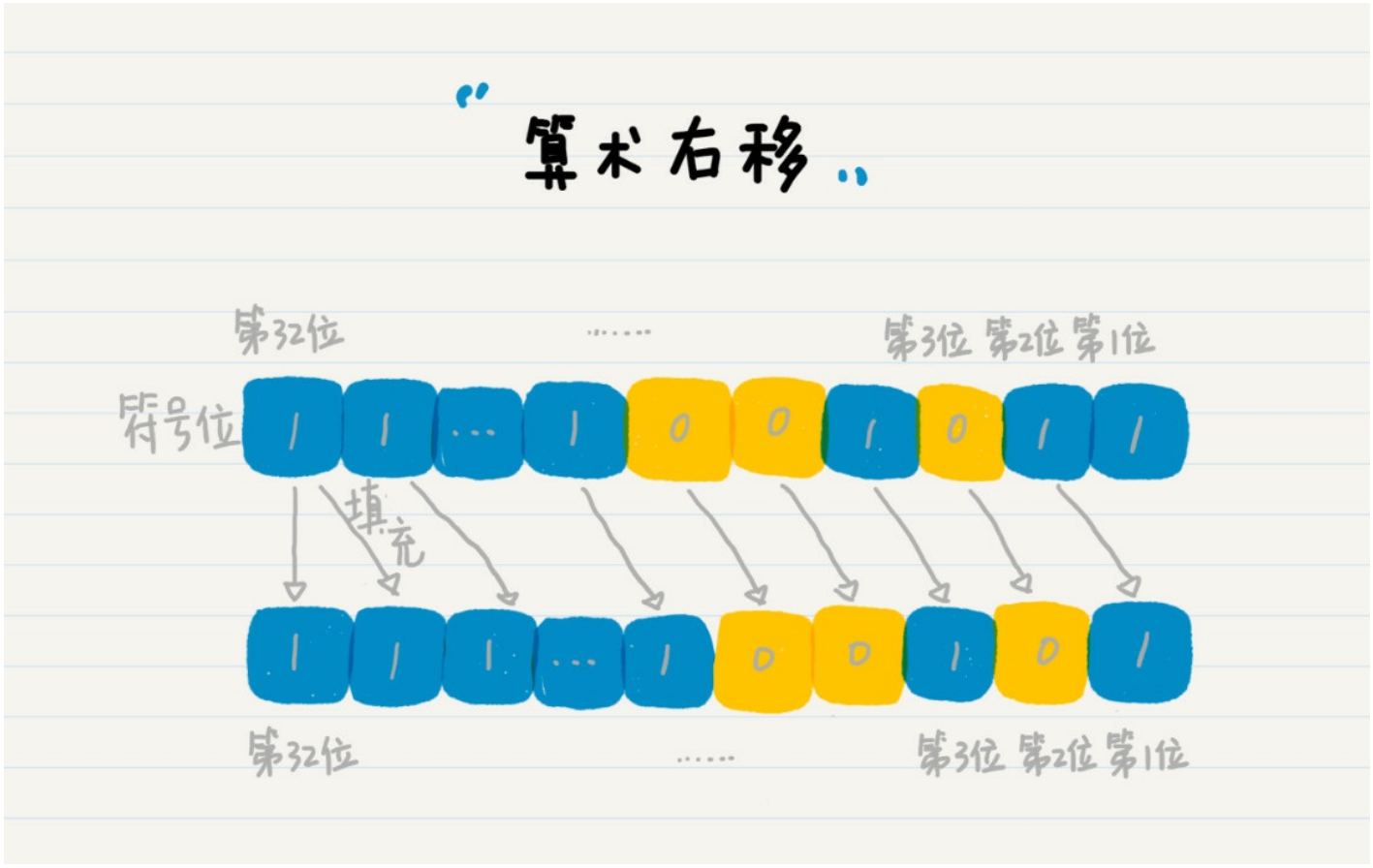
如果数字是-53呢？那么第32位就不是0，而是1。请注意我这里列出的是补码。



那么这个时候向右移位，就会产生一个问题：对于符号位（特别是符号位为1的时候），我们是否也需要将其右移呢？因此，Java里定义了两种右移，**逻辑右移**和**算术右移**。逻辑右移1位，左边补0即可。



算术右移时保持符号位不变，除符号位之外的右移一位并补符号位1。补的1仍然在符号位之后。



逻辑右移在Java和Python语言中使用>>>表示，而算术右移使用>>表示。如果你有兴趣，可以自己编码尝试一下，看看这两

种操作符输出的结果有何不同。

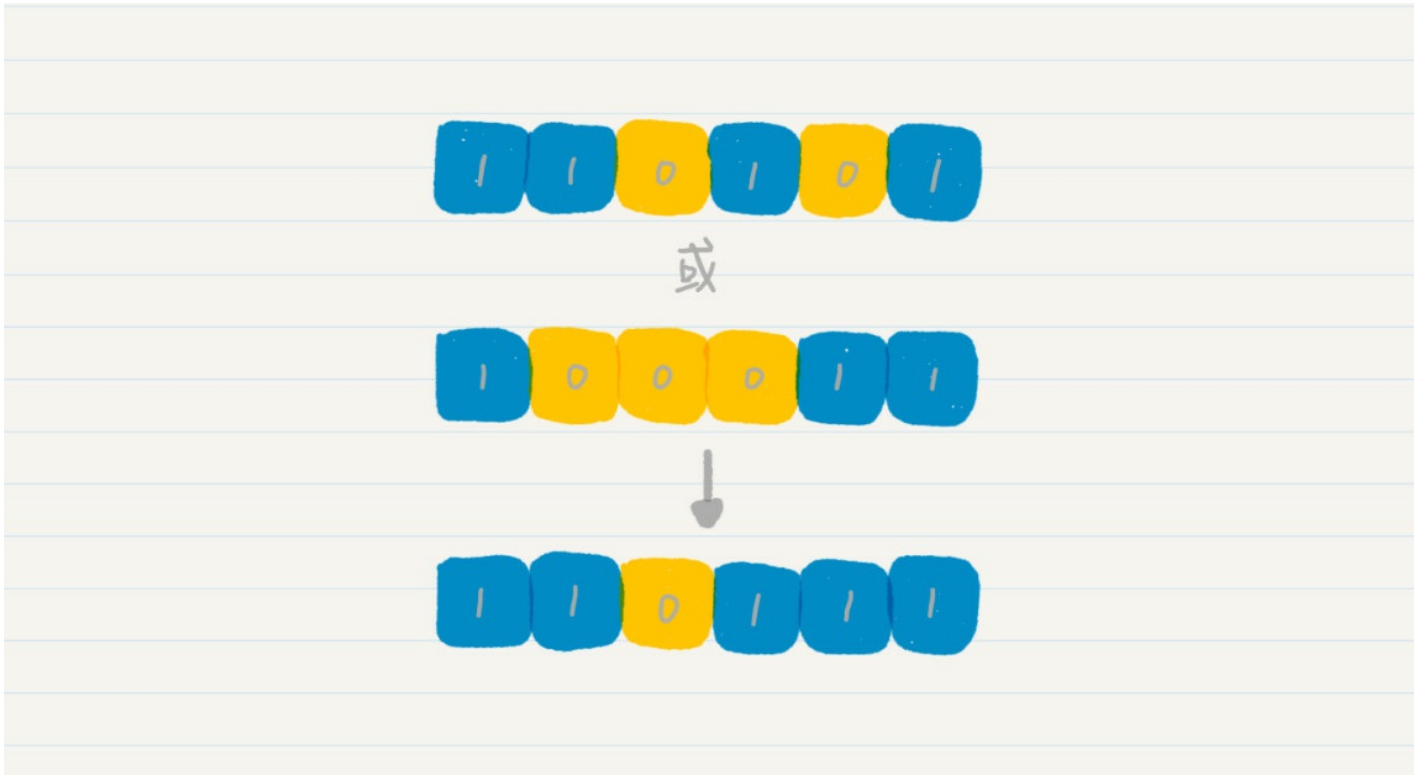
在C或C++语言中，逻辑右移和算数右移共享同一个运算符>>。那么，编译器是如何决定使用逻辑右移还是算数右移呢？答案是，取决于运算数的类型。如果运算数类型是unsigned，则采用逻辑右移；而是signed，则采用算数右移。如果你针对unsigned类型的数据使用算数右移，或者针对signed类型的数据使用逻辑右移，那么你首先需要进行类型的转换。

由于左移位无需考虑高位补1还是补0（符号位可能为1或0），所以不需要区分逻辑左移和算术左移。

位的“或”

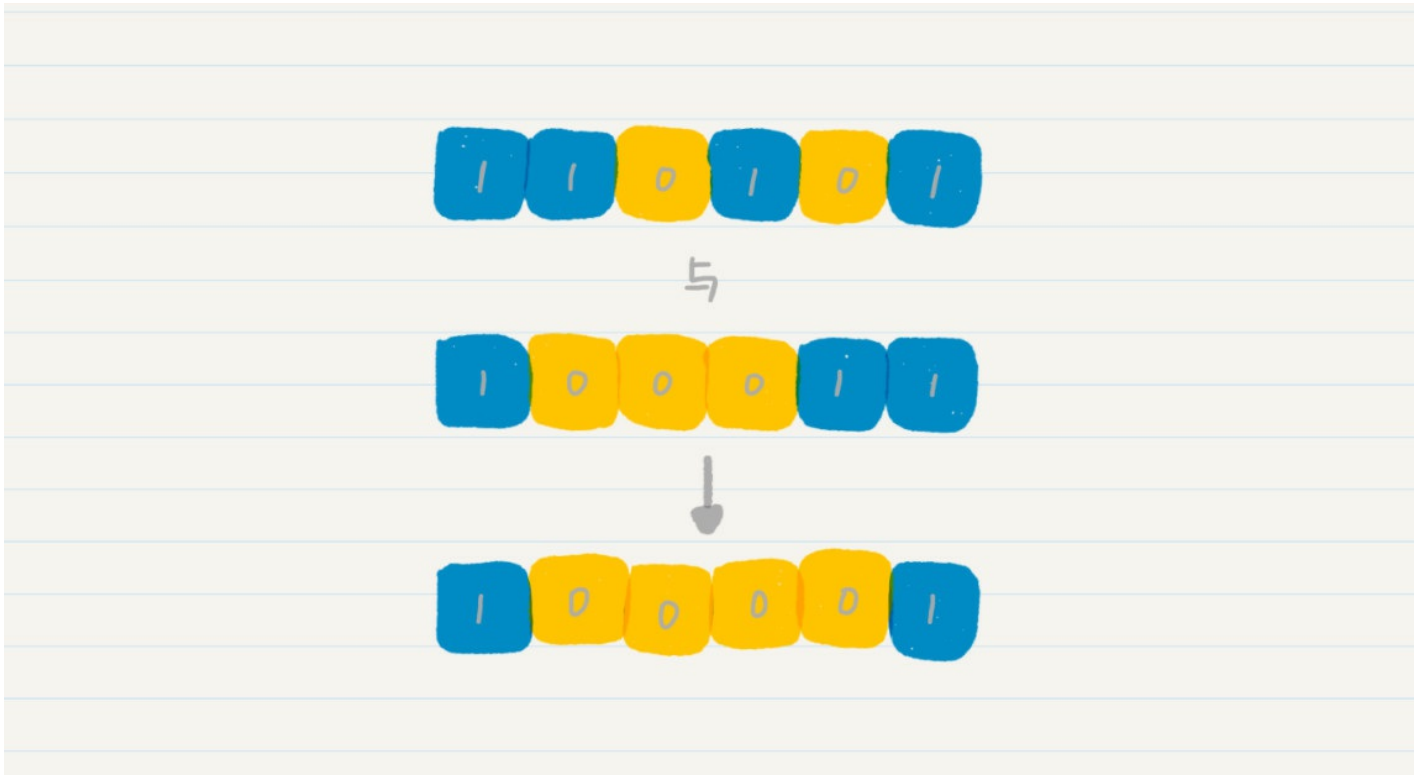
我们刚才说了，二进制的“1”和“0”分别对应逻辑中的“真”和“假”，因此可以针对位进行逻辑操作。

逻辑“或”的意思是，参与操作的位中只要有一个位是1，那么最终结果就是1，也就是“真”。如果我们将二进制110101和100011的每一位对齐，进行按位的“或”操作，就会得到110111。



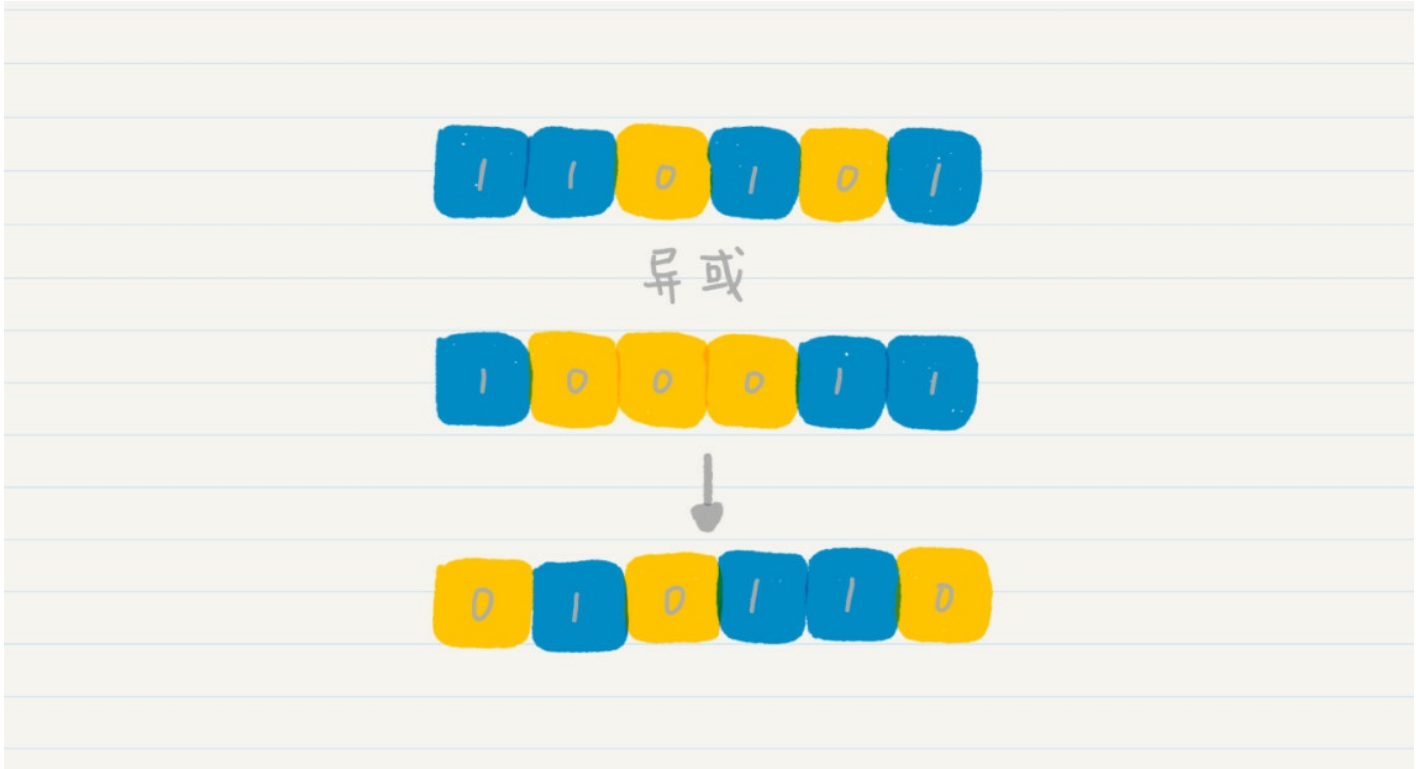
位的“与”

同理，我们也可以针对位进行逻辑“与”的操作。“与”的意思是，参与操作的位中必须全都是1，那么最终结果才是1（真），否则就为0（假）。如果我们将二进制110101和100011的每一位对齐，进行按位的“与”操作，就会得到100001。



位的“异或”

逻辑“异或”和“或”有所不同，它具有排异性，也就是说如果参与操作的位相同，那么最终结果就为0（假），否则为 1（真）。所以，如果要得到1，参与操作的两个位必须不同，这就是此处“异”的含义。我们将二进制110101和100011的每一位对齐，进行按位的“异或”操作，可以得到结果是10110。



我总结一下，“异或”操作的本质其实就是，所有数值和自身进行按位的“异或”操作之后都为0。而且要通过“异或”操作得到0，也必须通过两个相同的数值进行按位“异或”。这表明了两个数值按位“异或”结果为0，是这两个数值相等的必要充分条件，可以作为判断两个变量是否相等的条件。

接下来，我们来学习一下，在代码中如何实现二进制的逻辑操作。Java中使用|表示按位的“或”，&表示按位“与”，^表示按位“异或”。

```
import java.math.BigInteger;

public class Lesson1_3 {

   /**
    * @Description: 二进制按位“或”的操作
    * @param num1-第一个数字，num2-第二个数字
    * @return 二进制按位“或”的结果
    */
   public static int or(int num1, int num2) {

      return (num1 | num2);

   }

   /**
    * @Description: 二进制按位“与”的操作
    * @param num1-第一个数字，num2-第二个数字
    * @return 二进制按位“与”的结果
    */
   public static int and(int num1, int num2) {

      return (num1 & num2);

   }

   /**

    * @Description: 二进制按位“异或”的操作
    * @param num1-第一个数字，num2-第二个数字
    * @return 二进制按位“异或”的结果
    */

   public static int xor(int num1, int num2) {

      return (num1 ^ num2);

   }


}
```

同样，我们写一段测试代码，验证一下上面三个函数。

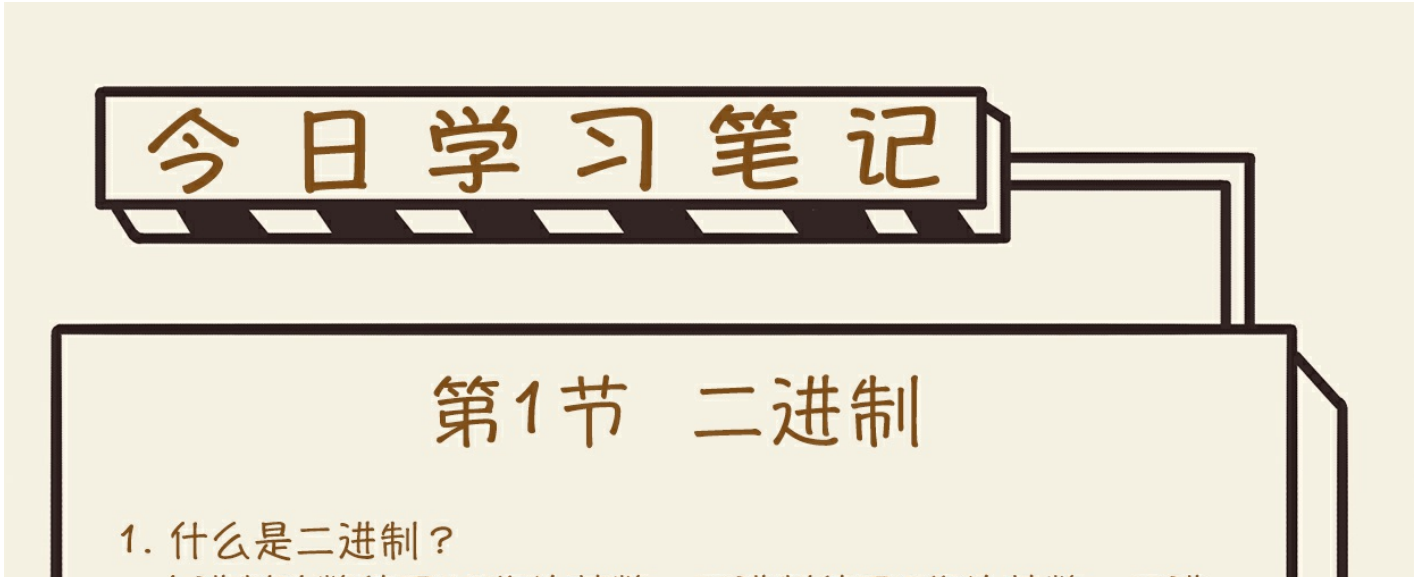
```
public static void main(String[] args) {

      int a = 53;
      int b = 35;

      System.out.println(String.format("数字%d(%s)和数字%d(%s)的按位‘或’结果是%d(%s)",
            a, decimalToBinary(a), b, decimalToBinary(b), Lesson2_3.or(a, b), decimalToBinary(Lesson1_3.or(a, b

      System.out.println(String.format("数字%d(%s)和数字%d(%s)的按位‘与’结果是%d(%s)",
            a, decimalToBinary(a), b, decimalToBinary(b), Lesson2_3.and(a, b), decimalToBinary(Lesson1_3.and(a,

      System.out.println(String.format("数字%d(%s)和数字%d(%s)的按位‘异或’结果是%d(%s)",
            a, decimalToBinary(a), a, decimalToBinary(a), Lesson2_3.xor(a, a), decimalToBinary(Lesson1_3.xor(a,

  }
```

这段代码的运行结果是：数字53(110101)和数字35(100011)的按位‘或’结果是55(110111)，数字53(110101)和数字35(100011)的按位‘与’结果是33(100001)，数字53(110101)和数字53(110101)的按位‘异或’结果是0(0)。

小结

今天我们聊了二进制，你可能会问：学习二进制究竟有什么用呢？平时的编程中，我们好像并没有使用相关的知识啊？确实，目前的高级语言可以帮助我们将人类的思维逻辑转换为使用0和1的机器语言，我们不用再为此操心了。但是，二进制作为现代计算机体系的基石，这些基础的概念和操作，你一定要非常了解。

二进制贯穿在很多常用的概念和思想中，例如逻辑判断、二分法、二叉树等等。逻辑判断中的真假值就是用二进制的1和0来表示的；二分法和二叉树都是把要处理的问题一分为二，正好也可以通过二进制的1和0来表示。因此，理解了二进制，你就能更加容易地理解很多计算机的数据结构和算法，也为我们后面的学习打下基础。



十进制计数使用10作为基数，二进制使用2作为基数，二进制的数位就是2^n的形式。

2. 计算机为什么使用二进制？

二进制的数据表达具有抗干扰能力强、可靠性高的优点；二进制非常适合逻辑运算。

3. 二进制的位操作

移位操作：

二进制左移一位，就是将数字翻倍。二进制右移一位，就是将数字除以2并求整数商。

逻辑操作：

“或”：参与操作的位中只要有一个是1，最终结果就是1。

“与”：参与操作的位中必须全都是1，最终结果才是1，否则就为0。

“异或”：参与操作的位相同，最终结果就为0，否则为 1。

黄申 · 程序员的数学基础课

思考题

如果不使用Java语言自带的BigInteger类，我们还有什么方法来实现十进制到二进制的转换呢？（提示：可以使用二进制的移位和按位逻辑操作来实现。）

欢迎在留言区交作业，并写下你今天的学习笔记。你可以点击“请朋友读”，把今天的内容分享给你的好友，和他一起精进。

精选留言

无双

请问有没有方法，快速实现进制转换，比如二进制、十进制、八进制、十六进制互相转化，我考试有要求要转，就是笔算，谢谢。

2018-12-10 09:26

作者回复

如果要快速在二进制、八进制和十六进制间转换，方法确实存在，可以上网查一些资料。如果很多人都感兴趣，我可以加入

2018-12-10 11:19

南山

逻辑或，与，异或一般有什么使用场景，平常写代码不怎么用

2018-12-10 13:13

作者回复

在elasticsearch的filter查询中，用到的bitset就是位运算，比查询倒排索引效率更高

2018-12-10 15:14

溯雪

老师，为什么不需要区分逻辑左移和算术左移呢?

比如十进制数-3，对应二进制1000...0011，那按照右移的思路，应该有两种移法，一种是符号位不动其它位置左移的1000...0110，一种是全部左移导致符号位被顶出去的0000....0110嘛

2018-12-10 22:17

作者回复

右移存在一个问题是，在高位补0还是1。但是左移只需要考虑后面补0就可以了

2018-12-10 23:17

Lugyedo

为什么不区分逻辑左移和算术左移

2018-12-10 09:51

清如許

#python实现

```
#encoding=utf-8
def int3binary(num):
result=[]
while num!=0:
result.append(num & 1)
num = num >> 1
result.reverse()
return result

print(*int2binary(10))

#输出 1010
```

2018-12-10 20:58

Libra

```
import java.util.Scanner;

public class Test1 {
public static void main(String[] args) {
Scanner scanner = new Scanner(System.in);

int value = scanner.nextInt();
boolean flag = false;

for (int i = 31; i >= 0; i--) {
int temp = value & (1 << i);
if (temp > 0){
flag = true;
}
if (flag){
if (temp > 0){
System.out.print(1);
}else {
System.out.print(0);
}
}
}
}
}
```

2018-12-10 10:42

作者回复

还可以考虑负数的情况

2018-12-20 08:32

Li Shunduo

请问文章里的图是用什么软件画的?

2018-12-10 16:49

桃园悠然在

被池大吓得赶紧买一个数学专栏，配合吴军老师的数学之美一起看。

2018-12-11 15:12

指间砂的宿命

数字与1做与操作，结果为1说明低位是1，否则为0，然后数字右移，重复以上操作，直到数字为0结束，倒序输出所有结果

2018-12-10 09:45

石佳佳_Gemtra

1.不考虑溢出的话，二进制数左移 n 位，即乘以 2^n；同理，右移 n 位，即除以 2^n，且向下取整数，因为移除的 n 位不全为 0 的话，除的结果就会包含小数。

2.不考虑溢出的话，有符号和无符号的左移运算结果相同，而右移的结果不同，所以会有逻辑右移和算术右移的区别。

3.两个数按位「异或」结果为 0，是这两个数值相等的必要充分条件。

4.思考题

Java 不太了解，根据提示，判断 n 位是否为 1，可与 1 左移n位后的数进行「与」运算，为真则为 1，反之为 0，循环即可。

2018-12-10 09:39

panda

异或 我想到一个算法题 判断很多数是不是有相等的

2018-12-11 22:12

作者回复

是的 很经典的一道面试题

2018-12-12 03:47

清如許

2^3<10<2^4

得到3,然后

10>>3=1

（10>>2）&1=0

（10>>1）&1=1

10&1=0

得到1010

2018-12-10 19:42

作者回复

再考虑一下负数的情况

2018-12-20 08:28

sloth-yp

最后的思考题， 是不是应该考虑负数，用补码表示?

```
public static String decimal2Binary(int decimal) {
// 负数的话,先换成正数然后取反再加1,再递归调用本函数
if (decimal < 0) {
int reverseNumber = ((decimal * -1) ^ Integer.MAX_VALUE) + 1;
return decimal2Binary (reverseNumber);
}
StringBuilder sb = new StringBuilder();
while (decimal > 0) {
// 跟0x0001 按位与,求得最低位的值
String lastNumber = String.valueOf(decimal & 1);
// 插入到字符串的最前面(这样才是原始的顺序)
sb.insert(0, lastNumber);
// 算术右移
decimal = decimal >> 1;
}
return sb.toString();
}
```

2018-12-18 16:52

作者回复

是的 需要考虑补码

2018-12-18 23:25

Transient

加密算法中也有许多用到二进制运算吧，而且二进制应该还有取反操作吧

2018-12-10 09:12

作者回复

是的 还有取反操作

2018-12-10 11:22

朱月俊

二进制左移1位相当于对应的十进制数字处以2取整数。因此，可以每次对十进制除以2并取整，将每次操作的余数按顺序从右到左排好序即可。

2018-12-10 00:53

Julian

01 | 二进制：不了解计算机的源头，你学什么编程的学习总结；

一：对于进制我现在的理解就是几进制就是以几为基数，然后按照左高右低规则进行基数幂运算然后在乘以数量然后在相加。例如：二进制110，首先基数是“2”；坐高右低原则就是“2 1 0”分别对应最左边的“1 1 0”；其实坐高右低就是从右边以0开始然后依次加一，这个是进行幂运算的多少次方的数字。所以这个二进制数转换成我们日常的十进制的计算规则就是：1*2^2 + 1*2^1 + 0*2^0;最终结果就是 4 + 2 + 1 = 7；

二：二进制的位操作；

1.二进制的移位操作；分为左移（<<）和右移；其中右移又分为算数右移(>>)和逻辑右移(>>>);

二进制左移<<：110 其实就是一次把数字往左移动一位，最右边补0所以最终就是1100；左移规律是数字的值翻倍；

二进制右移>>：同上就是把数字往右边移动一位，然后最左边数字看情况；如果是算术右移就要考虑正负数的问题；正负数在计算机里面java实现是看你操作系统位数，最后一位代表正负数的标识；0：代表正数；1：代表负数；所以算术右移我理解就是考虑计算，既然考虑计算也就是考虑正负数的问题；对于负数的算术右移左边是要上1的，对于正数的算数右移左边是上0的；但是对于逻辑右移，不管你是正数还是负数，左边都是上0的；

2.二进制的逻辑操作；

逻辑或 | ： 就是2个二进制数，从右到左，对相同位置的数字进行或运算；或运算就是 全是0才表示0，其余全是1；或顾名思义，只要有真就代表真；

逻辑与 & ： 就是2个二进制数，从右到左，对相同位置的数字进行与运算；与运算就是参与运算数字全1才是1，其余的都是0；与顾明思议，相同的而且都是真才为真；

逻辑异或 ^： 就是2个二进制数吗，从右到左，对相同位置的数字进行运算；异或运算就是参与运算的数字相同结果为0，其余全为1；异或顾明思议，不同的才为真；

在计算机世界，1代表真，0代表假；

2018-12-12 11:54

咕噜

打卡1/64，笔记已同步至博客。使用位操作实现进制转换还是第一次碰到，学习了！

2018-12-10 23:46

郑晨Cc

思考题

```
public class test {

public static void main(String[] args){
StringBuffer sb = null;
int a = 53;

String buffer = null;

while(a>1){
```

```
System.out.println("余数：" + a % 2);
String b = String.valueOf(a % 2);
if(null == sb){
sb = new StringBuffer(b);
}else{
sb.insert(0, b);
}
a = a >>>1;
System.out.println("商："+a);

}

sb.insert(0, a);
System.out.println(sb);
}

}
```

2018-12-10 15:24

作者回复

还需要考虑负数的情况

2018-12-20 08:31

maliming

```
/**
* @Title: decimalToBinary
* @Description: 十进制转二进制，方法1：余数短除法除以二
* @param decimalSource
* @return: String
*/
/*public static String decimalToBinary(int decimalSource) {
StringBuilder sb = new StringBuilder();
while (decimalSource != 0) {
sb.append(decimalSource % 2);
decimalSource = decimalSource >> 1;
}
return sb.reverse().toString();
}*/

/**
* @Title: decimalToBinary
* @Description: 十进制转二进制，方法2：降二次幂及减法混合运算
* @param decimalSource
* @return: String
*/
/*public static String decimalToBinary(int decimalSource) {
int length = (int) (Math.log(decimalSource) / Math.log(2));
StringBuffer sb = new StringBuffer();
do {
decimalSource = (int) (decimalSource - Math.pow(2, length));
int power = decimalSource <= 0 ? -1 : (int) (Math.log(decimalSource) / Math.log(2));
for (int i = length; i > power; i--) {
if (i == length) {
```

```
sb.append("1");
} else {
sb.append("0");
}
}
length = power;
} while (decimalSource > 0);
return sb.toString();
}*/
/**
*
* @Title: decimalToBinary
* @Description: 十进制转二进制，方法3：位运算法
* @param decimalSource
* @return
* @return: String
*/
public static String decimalToBinary(int decimalSource) {
StringBuffer sb = new StringBuffer();
while (decimalSource != 0) {
//此&运算，decimalSource & 1，目的是获取最低位的二进制数值
sb.append(decimalSource & 1);
//此>>运算，decimalSource >> 1，目的是将获取到的最低位二进制数值除去
decimalSource = decimalSource >> 1;
}
return sb.reverse().toString();
}
```

负整数转换为二进制 要点：
取反加一 解释：将该负整数对应的正整数先转换成二进制，然后对其“取补”，再对取补后的结果加1即可。
例如要把-52换算成二进制：
1.先取得52的二进制：00110100
2.对所得到的二进制数取反：11001011
3.将取反后的数值加一即可：11001100 即：(-52)10=(11001100)2

2019-01-12 14:22

海洋之心

```
public static String decimalToBinary(int decimalSource) {
if(decimalSource==-2147483648) {
return "10000000 00000000 00000000 00000000";
}
int[] bits = new int[32];
int i = 32;
StringBuffer sb = new StringBuffer();


int result = decimalSource;

if(decimalSource<0) {
result = -decimalSource;
}

while(result!=0) {
```

```
i--;
bits[i] = result%2;
result = result/2;
}

if(decimalSource<0) {
// 负数全部取反
for(int j = 0; j<bits.length; j++) {
bits[j] = bits[j]^1;
}
// 最高位置为1
bits[0] = 1;
// 补1
for(int j = 31; j>=0; j--) {
if(bits[j]==0) {
bits[j]=1;
break;
} else {
bits[j]=0;
continue;
}
}
}

for(int j = 0; j<bits.length; j++) {
if(j%8==0) {
sb.append(" ");
}
sb.append(bits[j]);
}



return sb.toString();
}

//感觉做的有些麻烦 还有什么简单的方法吗
```
2019-01-03 09:32