|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**Java**](http://lib.csdn.net/base/javase)的位运算（bitwise operators）直接对整数类型的位进行操作，这些整数类型包括long、int、short、char和 byte，位运算符具体如下表：   |  |  | | --- | --- | | 运算符 | 说明 | | << | 左移位，在低位处补0 | | >> | 右移位，若为正数则高位补0，若为负数则高位补1 | | >>> | 无符号右移位，无论正负都在高位补0 | | & | 与（AND），对两个整型操作数中对应位执行布尔代数，两个位都为1时输出1，否则0。 | | | | 或（OR），对两个整型操作数中对应位执行布尔代数，两个位都为0时输出0，否则1。 | | ~ | 非（NOT），一元运算符。 | | ^ | 异或（XOR），对两个整型操作数中对应位执行布尔代数，两个位相等0，不等1。 | | <<= | 左移位赋值。 | | >>= | 右移位赋值。 | | >>>= | 无符号右移位赋值。 | | &= | 按位与赋值。 | | |= | 按位或赋值。 | | ^= | 按位异或赋值。 |   **左移位（<<）**  程序：  public class LeftMoving{      public static void main(String[] args){             System.out.println("5<<3="+(5<<3));      }  }  输出结果：  5<<3=40  计算过程：  0000 0000 0000 0000 0000 0000 0000 0101         ? 5  0000 0000 0000 0000 0000 0000 0010 1000         ? 40  **右移位（>>）**  **正数**  程序：  public class PlusRightMoving{      public static void main(String[] args){             System.out.println("5>>1="+(5>>1));      }  }  输出结果：  5>>1=2  计算过程：  0000 0000 0000 0000 0000 0000 0000 0101         ? 5  0000 0000 0000 0000 0000 0000 0000 0010         ? 2  **负数**  程序：  public class NegativeRightMoving{      public static void main(String[] args){             System.out.println("-5>>1="+(-5>>1));      }  }  输出结果：  -5>>1=-3  计算过程：  1111 1111 1111 1111 1111 1111 1111 1011         ? -5  1111 1111 1111 1111 1111 1111 1111 1101         ? -3  **无符号右移位（>>>）**  程序：  public class UnsignedRightMoving{      public static void main(String[] args){  System.out.println("-5>>>1="+(-5>>>1));      }  }  输出结果：  -5>>>1=2147483645  计算过程：  1111 1111 1111 1111 1111 1111 1111 1011         ? -5         0111 1111 1111 1111 1111 1111 1111 1101          ? 2147483645 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 首先复习一下Java中的基本数据类型的相关知识。   |  |  |  |  | | --- | --- | --- | --- | | 数据类型 | 大小 | 最小值 | 最大值 | | boolean |  |  |  | | byte | 8-bit | -128 | +127 | | char | 16-bit | Unicode 0 | Unicode 216-1 | | short | 16-bit | -215 | +215-1 | | int | 32-bit | -231 | +231-1 | | float | 32-bit | IEEE754 | IEEE754 | | long | 64-bit | -263 | 263-1 | | double | 64-bit | IEEE754 | IEEE754 | | void |  |  |  |   这里包括了float和double两个浮点型，在本文中对其不予考虑，因为位运算是针对整型的。进行位操作时，除long型外，其他类型会自动转成int型，转换之后，可接受右操作数长度为32。进行位运算时，总是先将短整型和字节型值转换成整型值再进行移位操作的。  程序：  **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** ByteLeftMoving{ 3. **public** **static** **void** main(String[] args){ 5. **byte** b = 127; 7. System.out.println("b<<3="+(b<<3)); 9. System.out.println("(byte)(b<<3)="+(**byte**)(b<<3)); 11. } 13. }     输出结果：  b<<3=1016  (byte)(b<<3)=-8  程序：    **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** CharLeftMoving{ 3. **public** **static** **void** main(String[] args){ 5. **char** c = 'l'; 7. System.out.println("c<<3="+(c<<3)); 9. System.out.println("(char)(c<<3)="+(**char**)(c<<3)); 11. } 13. }   输出结果：  c<<3=864  (char)(c<<3)=?  以上两个例子全部编译通过，由此可以看出，当byte和char进行移位运算时不会发生错误，并且均按照整型进行计算，当计算结果超出byte或是char所能表示的范围时则进行相应的转换（分别输出了结果-8和?）。 |

|  |
| --- |
| **位运算中的操作数**  在进行移位运算时要注意整型和长整型在内存中的位数（整型是32位，长整型是64位），如果移位操作数超出了该位数则取模计算，例如：int型数据是32位的，如果左移35位是什么结果？  程序：  **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** LeftMoving{ 3. **public** **static** **void** main(String[] args){ 5. System.out.println("5<<35="+(5<<35)); 7. } 9. }     输出结果：  5<<35=40  该结果与5<<3完全相同。  无论正数、负数，它们的右移、左移、无符号右移 32位都是其本身，比如 -5<<32=-5、-5>>32=-5、-5>>>32=-5。  一个有趣的现象是，把 1 左移 31 位再右移 31位，其结果为 -1。  计算过程如下：  0000 0000 0000 0000 0000 0000 0000 0001  1000 0000 0000 0000 0000 0000 0000 0000  1111 1111 1111 1111 1111 1111 1111 1111  位运算要求操作数为整数，操作数不能是字符串也不能是小数。  如下列程序：  **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** BitMath{ 3. **public** **static** **void** main(String[] args){ 5. String s = "Hello"; 7. **long** l = 99; 9. **double** d = 1.11; 11. **int** i = 1; 13. **int** j = 0;  16. System.out.println("j<<s="+j<<s);    //编译错误语句 18. System.out.println("j<<d="+j<<d);    //编译错误语句 20. System.out.println("i<<j="+i<<j);    //编译可以通过 22. System.out.println("i<<l="+i<<l);    //编译可以通过 24. } 26. } |
| 由于位运算是二进制运算，不要与一些八进制数搞混，java中二进制数没有具体的表示方法。  public class BitMath{      public static void main(String[] args){          System.out.println("010|4="+(010|4));      }  }  输出结果：  010|4=12  计算过程：  0000 0000 0000 0000 0000 0000 0000 1000   ?8  0000 0000 0000 0000 0000 0000 0000 0100   ?4  进行“或”计算结果为：  0000 0000 0000 0000 0000 0000 0000 1100   ?12  当位运算中遇见负数，必须把它转成补码（不知道什么是补码的补习功课去）再进行计算，而不是使用原码。  程序：  public class BitMath{      public static void main(String[] args){          try {              int x = -7;              System.out.println("x>>1="+(x>>1));  } catch(Exception e) {              System.out.println("Exception");          }      }  }  输出结果：  x>>1=-4  计算过程：  1111 1111 1111 1111 1111 1111 1111 1001   ?-7  1111 1111 1111 1111 1111 1111 1111 1100   ?-4  **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** BitMath{ 3. **public** **static** **void** main(String[] args){ 5. **int** i = 1; 7. **int** j = -1; 9. System.out.println("1>>>31="+(i>>>31)); 11. System.out.println("-1>>31="+(j>>31)); 13. } 15. }     输出结果：  1>>>31=0  -1>>31=-1  程序：  **[java]** [view plain](http://blog.csdn.net/mw08091020/article/details/8010300) [copy](http://blog.csdn.net/mw08091020/article/details/8010300)   1. **public** **class** BitMath{ 3. **public** **static** **void** main(String[] args){ 5. **int** a = 1; 7. a <<= 31; 9. a >>= 31; 11. a >>= 1; 13. System.out.println("a="+a);  16. **int** b = 1; 18. b <<= 31; 20. b >>= 31; 22. System.out.println("b="+b);  25. **int** c = 1; 27. c >>= 31; 29. c <<= 31; 31. System.out.println("c="+c); 33. } 35. }   输出结果：  a=-1  b=-1  c=0  计算过程：  0000 0000 0000 0000 0000 0000 0000 0001   ?a=1  1000 0000 0000 0000 0000 0000 0000 0000   ?a=a<<31后，这里被当作是负数  1111 1111 1111 1111 1111 1111 1111 1111   ?a=a>>31后，结果为-1  1111 1111 1111 1111 1111 1111 1111 1111   ?a=a>>1后，结果仍为-1  0000 0000 0000 0000 0000 0000 0000 0001   ?c=1  0000 0000 0000 0000 0000 0000 0000 0000   ?c=c>>31后为0  0000 0000 0000 0000 0000 0000 0000 0000   ?0左移31位仍为0 |

**机器数**

一个数在计算机中的二进制表示形式，叫做这个数的机器数，机器数是带符号的，在计算机中用一个数的最高位存放符号，正数为0，负数为1，比如，十进制中的+3，假设计算机字长为8位，转换成二进制就是0000 0011，如果是-3，就是1000 0011.那么，这里0000 0011和1000 0011就是机器数，

**真值**

因为第一位为符号位，所以机器数的形式值就不等于真正的数值，例如上面的有符号数1000 0011，其最高位1代表负，其真正数值是-3,而不是形式值131（1000 0011转换成10进制等于131），所以为了区别起见，将带符号的机器数对应的真正数值称为机器数的真值。例如：0000 0001的真值=+000 0001=+1,1000 0001的真值=-000 0001=-1

**原码：**

原码就是符号位加上真值的绝对值，即用第一位表示符号，其余位表示值，比如如果是8位二进制，[+1]原=0000 0001.[-1]原=1000 0001.因为第一位是符号位，所以8位二进制的取值范围就是：[1111 1111,0111 1111]即[-127,127]，原码是人脑最容易理解和计算的表示方式。

**反码：**

反码的表示方法是:正数的反码是其本身，负数的反码是在其原码的基础上，符号位不变，其余各个位取反，[+1]=[0000 0001]原= [0000 0001]反，[-1]=[1000 0001]原=[1111 1110]反。可见如果一个反码表示的是负数，人脑无法直观的看出来它的数值，通常要将其转换成原码再计算。

**补码：**

补码的表示方法是：正整数的二进制补码与其二进制原码相同，负整数的二进制补码，先求与该负数相对应的正整数的二进制代码，然后所有位取反加1，不够位数时左边补1，例如，[+1]=[0000 0001]原=[0000 0001]反=[0000 0001]补，[-1]=[1000 0001]原=[1111 1110]反=[1111 1111]补，对于负数，补码表示方式也是人脑无法直观看出其数值的，通常也需要转换成原码再计算其数值。

**为什么要使用原码反码补码，**

现在我们知道了计算机可以用原码 反码 补码这三种编码方式表示一个数，对于正数因为三种编码方式都相同，没有什么好解释的，但是对于负数，负数的原码反码补码是完全不同的，既然原码才是被人脑直接识别并用于计算方式，那么为什么还要用反码和补码呢，首先，因为人脑可以知道原码的第一位是符号位，在计算的时候，我们会根据符号位，选择对真值区域的加减，但是对于计算机，加减乘除已经是最基础的运算，要设计的尽量简单，计算机辨别符号位显然会让计算机的基础电路设计变得十分复杂，于是人们想出了将符号位也参与运算的方法，我们知道，根据运算法则减去一个正数等于加上一个负数，即：1-1=1+（-1）=0；所以机器可以只有加法而没有减法，这样计算机运算的设计就更简单了，那么如果用原码计算，1-1=1+(-1)=[0000 0001]原+[1000 0001]原=[1000 0010]原=-2.如果用原码计算，让符号位也参与运算，显然对于减法来说，结果是不正确的，这也就是为什么计算机内部不用原码表示一个数，为了解决原码做减法的问题出现了反码，如果用反码计算减法，1-1=1+(-1)= [0000 0001]原+ [1000 0001]原=[0000 0001]反+[1111 1110反]=[1111 1111]反=[1000 0000]原=-0，发现用反码计算减法，结果的真值部分是正确的，而唯一的问题其实出现在0这个特殊的数值上，虽然人们理解上+0和-0是一样的，但是0带符号是没有任何意义的，而且会有[0000 0000]原和[1000 0000]原两个编码表示0；于是补码的出现，解决了0的符号以及两个编码的问题：1-1=1+(-1)=[0000 0001]原+[1000 0001]原=[0000 0001]补+[1111 1111]补=[0000 0000]补=[0000 0000]原，这样0用[0000 0000]表示，而以前出现的问题-0则不存在了，而且可以用[1000 0000]表示-128；(-1) + (-127) = [1000 0001]原 + [1111 1111]原 = [1111 1111]补 + [1000 0001]补 = [1000 0000]补。-1-127的结果应该是-128, 在用补码运算的结果中, [1000 0000]补 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示[1000 0000]补算出来的原码是[0000 0000]原, 这是不正确的)。使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].因为机器使用补码, 所以对于编程中常用到的32位int类型, 可以表示范围是: [-231, 231-1] 因为第一位表示的是符号位.而使用补码表示时又可以多保存一个最小值。

java中负数显示成补码

**package** com.example;  
  
*/\*\*  
 \*  
 \* <pre>  
 \* 基本数据类型转换(主要是byte和其它类型之间的互转).  
 \* </pre>  
 \*  
 \** ***@author*** *F.Fang  
 \** ***@version*** *$Id: ByteUtils.java, v 0.1 2014年11月9日 下午11:23:21 F.Fang Exp $  
 \*/***public class** ByteUtils {  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将4个byte数字组成的数组合并为一个float数.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static float** byte4ToFloat(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 4) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是4位!"**);  
 }  
 **int** i = *byte4ToInt*(arr);  
 **return** Float.*intBitsToFloat*(i);  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将一个float数字转换为4个byte数字组成的数组.  
 \* </pre>  
 \*  
 \** ***@param f*** *\** ***@return*** *\*/* **public static byte**[] floatToByte4(**float** f) {  
 **int** i = Float.*floatToIntBits*(f);  
 **return** *intToByte4*(i);  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将八个byte数字组成的数组转换为一个double数字.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static double** byte8ToDouble(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 8) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是8位!"**);  
 }  
 **long** l = *byte8ToLong*(arr);  
 **return** Double.*longBitsToDouble*(l);  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将一个double数字转换为8个byte数字组成的数组.  
 \* </pre>  
 \*  
 \** ***@param i*** *\** ***@return*** *\*/* **public static byte**[] doubleToByte8(**double** i) {  
 **long** j = Double.*doubleToLongBits*(i);  
 **return** *longToByte8*(j);  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将一个char字符转换为两个byte数字转换为的数组.  
 \* </pre>  
 \*  
 \** ***@param c*** *\** ***@return*** *\*/* **public static byte**[] charToByte2(**char** c) {  
 **byte**[] arr = **new byte**[2];  
 arr[0] = (**byte**) (c >> 8);  
 arr[1] = (**byte**) (c & 0xff);  
 **return** arr;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将2个byte数字组成的数组转换为一个char字符.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static char** byte2ToChar(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 2) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是2位!"**);  
 }  
 **return** (**char**) (((**char**) (arr[0] << 8)) | ((**char**) arr[1]));  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将一个16位的short转换为长度为2的8位byte数组.  
 \* </pre>  
 \*  
 \** ***@param s*** *\** ***@return*** *\*/* **public static byte**[] shortToByte2(Short s) {  
 **byte**[] arr = **new byte**[2];  
 arr[0] = (**byte**) (s >> 8);  
 arr[1] = (**byte**) (s & 0xff);  
 **return** arr;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 长度为2的8位byte数组转换为一个16位short数字.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static short** byte2ToShort(**byte**[] arr) {  
 **if** (arr != **null** && arr.**length** != 2) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是2位!"**);  
 }  
 **return** (**short**) (((**short**) arr[0] << 8) | ((**short**) arr[1] & 0xff));  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将short转换为长度为16的byte数组.  
 \* 实际上每个8位byte只存储了一个0或1的数字  
 \* 比较浪费.  
 \* </pre>  
 \*  
 \** ***@param s*** *\** ***@return*** *\*/* **public static byte**[] shortToByte16(**short** s) {  
 **byte**[] arr = **new byte**[16];  
 **for** (**int** i = 15; i >= 0; i--) {  
 arr[i] = (**byte**) (s & 1);  
 s >>= 1;  
 }  
 **return** arr;  
 }  
  
 **public static short** byte16ToShort(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 16) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且长度为16!"**);  
 }  
 **short** sum = 0;  
 **for** (**int** i = 0; i < 16; ++i) {  
 sum |= (arr[i] << (15 - i));  
 }  
 **return** sum;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将32位int转换为由四个8位byte数字.  
 \* </pre>  
 \*  
 \** ***@param sum*** *\** ***@return*** *\*/* **public static byte**[] intToByte4(**int** sum) {  
 **byte**[] arr = **new byte**[4];  
 arr[0] = (**byte**) (sum >> 24);  
 arr[1] = (**byte**) (sum >> 16);  
 arr[2] = (**byte**) (sum >> 8);  
 arr[3] = (**byte**) (sum & 0xff);  
 **return** arr;  
 }  
  
 */\*\*  
 \* <pre>  
 \* 将长度为4的8位byte数组转换为32位int.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static int** byte4ToInt(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 4) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是4位!"**);  
 }  
 **return** (**int**) (((arr[0] & 0xff) << 24) | ((arr[1] & 0xff) << 16) | ((arr[2] & 0xff) << 8) | ((arr[3] & 0xff)));  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将长度为8的8位byte数组转换为64位long.  
 \* </pre>  
 \*  
 \* 0xff对应16进制,f代表1111,0xff刚好是8位 byte[]  
 \* arr,byte[i]&0xff刚好满足一位byte计算,不会导致数据丢失. 如果是int计算. int[] arr,arr[i]&0xffff  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static long** byte8ToLong(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 8) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且是8位!"**);  
 }  
 **return** (**long**) (((**long**) (arr[0] & 0xff) << 56) | ((**long**) (arr[1] & 0xff) << 48) | ((**long**) (arr[2] & 0xff) << 40)  
 | ((**long**) (arr[3] & 0xff) << 32) | ((**long**) (arr[4] & 0xff) << 24)  
 | ((**long**) (arr[5] & 0xff) << 16) | ((**long**) (arr[6] & 0xff) << 8) | ((**long**) (arr[7] & 0xff)));  
 }  
  
 */\*\*  
 \* 将一个long数字转换为8个byte数组组成的数组.  
 \*/* **public static byte**[] longToByte8(**long** sum) {  
 **byte**[] arr = **new byte**[8];  
 arr[0] = (**byte**) (sum >> 56);  
 arr[1] = (**byte**) (sum >> 48);  
 arr[2] = (**byte**) (sum >> 40);  
 arr[3] = (**byte**) (sum >> 32);  
 arr[4] = (**byte**) (sum >> 24);  
 arr[5] = (**byte**) (sum >> 16);  
 arr[6] = (**byte**) (sum >> 8);  
 arr[7] = (**byte**) (sum & 0xff);  
 **return** arr;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将int转换为32位byte.  
 \* 实际上每个8位byte只存储了一个0或1的数字  
 \* 比较浪费.  
 \* </pre>  
 \*  
 \** ***@param num*** *\** ***@return*** *\*/* **public static byte**[] intToByte32(**int** num) {  
 **byte**[] arr = **new byte**[32];  
 **for** (**int** i = 31; i >= 0; i--) {  
 *// &1 也可以改为num&0x01,表示取最地位数字.* arr[i] = (**byte**) (num & 1);  
 *// 右移一位.* num >>= 1;  
 }  
 **return** arr;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将长度为32的byte数组转换为一个int类型值.  
 \* 每一个8位byte都只存储了0或1的数字.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static int** byte32ToInt(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 32) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且长度是32!"**);  
 }  
 **int** sum = 0;  
 **for** (**int** i = 0; i < 32; ++i) {  
 sum |= (arr[i] << (31 - i));  
 }  
 **return** sum;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将长度为64的byte数组转换为一个long类型值.  
 \* 每一个8位byte都只存储了0或1的数字.  
 \* </pre>  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static long** byte64ToLong(**byte**[] arr) {  
 **if** (arr == **null** || arr.**length** != 64) {  
 **throw new** IllegalArgumentException(**"byte数组必须不为空,并且长度是64!"**);  
 }  
 **long** sum = 0L;  
 **for** (**int** i = 0; i < 64; ++i) {  
 sum |= ((**long**) arr[i] << (63 - i));  
 }  
 **return** sum;  
 }  
  
 */\*\*  
 \*  
 \* <pre>  
 \* 将一个long值转换为长度为64的8位byte数组.  
 \* 每一个8位byte都只存储了0或1的数字.  
 \* </pre>  
 \*  
 \** ***@param sum*** *\** ***@return*** *\*/* **public static byte**[] longToByte64(**long** sum) {  
 **byte**[] arr = **new byte**[64];  
 **for** (**int** i = 63; i >= 0; i--) {  
 arr[i] = (**byte**) (sum & 1);  
 sum >>= 1;  
 }  
 **return** arr;  
 }  
  
 */\*\*  
 \* 将byte数组转string输出  
 \*  
 \** ***@param arr*** *\** ***@return*** *\*/* **public static** String byteToString(**byte**[] arr) {  
 StringBuffer buffer = **new** StringBuffer();  
 **for** (**byte** b : arr) {  
 buffer.append(b);  
 }  
 **return** buffer.toString();  
 }  
  
}