# 谈谈java中字节byte有负数的现象

在研究编码时，无意中发现java中输出编码后的字节数据的值有的是负值，比如utf-8编码后的字节数据，通过遍历，打印都是负值，java中字节byte有负数的现象让我产生了兴趣，在此探讨一下。

关于编码的字节有负数的现象，可以参考这篇博客：

<http://blog.csdn.net/csdn_ds/article/details/79077483>

下面我用java中的数据流去说说这个现象。

**实验一**

package com.anjz.test;

import java.io.ByteArrayInputStream;

import java.io.IOException;

public class ByteArrayTest {

public static void main(String[] args) throws IOException {

String str = "你好";

ByteArrayInputStream bis = new ByteArrayInputStream(str.getBytes("utf-8"));

byte[] bytes = new byte[str.getBytes("utf-8").length];

bis.read(bytes);

for(byte b :bytes){

System.out.print(b+",");

}

}

}

运行结果：

-28,-67,-96,-27,-91,-67,

**实验二**

package com.anjz.test;

import java.io.ByteArrayInputStream;

import java.io.IOException;

public class ByteArrayTest {

public static void main(String[] args) throws IOException {

String str = "你好";

ByteArrayInputStream bis = new ByteArrayInputStream(str.getBytes("utf-8"));

int temp = 0;

while((temp = bis.read())!=-1){

System.out.print(temp+",");

}

}

}

运行结果：

228,189,160,229,165,189,

实验一中直接输出的byte数据，实验二直接输出的是int数据，但两个数据是不一样的，我们把两个结果的数据放到一块。

-28,-67,-96,-27,-91,-67,

228,189,160,229,165,189,

发现一个规律：每列数据的绝对值加一起是个固定值256，这是一个巧合，还是一个规律？关于这个问题，首先我们看一下bis.read()的源码。

/\*\*

\* Reads the next byte of data from this input stream. The value

\* byte is returned as an <code>int</code> in the range

\* <code>0</code> to <code>255</code>. If no byte is available

\* because the end of the stream has been reached, the value

\* <code>-1</code> is returned.

\* <p>

\* This <code>read</code> method

\* cannot block.

\*

\* @return the next byte of data, or <code>-1</code> if the end of the

\* stream has been reached.

\*/

public synchronized int read() {

return (pos < count) ? (buf[pos++] & 0xff) : -1;

}

从上述代码的说明可以看出，此方法的返回值范围为[0,255],在方法体中，获取到字节后，进行了&0xff操作。

在此说明一个java中的几个规则：

1. **Java中byte的大小是8bits，int的大小是32bits，byte的范围是[-128,127],int的范围是[-231, 231-1]。**
2. **Java中数值的二进制是采用补码的形式表示的。**

其实从byte和int范围就可以看出，java中的二进制是采用补码表示的。关于原码、反码、补码的知识，可以参照这篇博客：

<http://blog.csdn.net/csdn_ds/article/details/79082640>

个人理解，**计算机不是所有的数据都是需要用补码表示的，补码的出现，主要是将计算机中的减法运算转化成加法运算，降低计算机底层的复杂性。Java中是数值类型的数据才使用补码表示，也就是数值类型在内存或磁盘中存储的都是补码，程序运行展示的数据是原码的十进制（或者说真值）。但对于字符来说，它是通过字符集（如UTF-8、GBK等）进行编码的，直接存储字节数即可。**

“你好”UTF-8编码对应的二进制：

11100100 10111101 10100000 11100101 10100101 10111101

查询UTF-8的编码可使用此地址：<http://www.mytju.com/classcode/tools/encode_utf8.asp>

转化成byte，当二进制以数值看待时，内存中的二进制要看成补码形式。

[11100100]补 = [10011100]原 = [-28]十进制(byte)

[10111101]补 = [11000011]原 = [-67]十进制(byte)

[10100000]补 = [11100000]原 = [-96]十进制(byte)

[11100101]补 = [10011011]原 = [-27]十进制(byte)

[10100101]补 = [11011011]原 = [-91]十进制(byte)

[10111101]补 = [11000011]原 = [-67]十进制(byte)

字节是计算机最小读取单位，如果最终转化成int类型，转化如下：

[-28]十进制（byte） = [10011100]原 = [11100100]补 ->转化成32位 [00000000 00000000 00000000 11100100]补 = [00000000 00000000 00000000 11100100]原 =[228]十进制(int)

[-67]十进制（byte） = [11000011]原 = [10111101]补 ->转化成32位 [00000000 00000000 00000000 10111101]补 = [00000000 00000000 00000000 10111101]原 =[189]十进制(int)

[-96]十进制（byte） = [11100000]原 = [10100000]补 ->转化成32位 [00000000 00000000 00000000 10100000]补 = [00000000 00000000 00000000 10100000]原 =[160]十进制(int)

[-27]十进制（byte） = [10011011]原 = [11100101]补 ->转化成32位 [00000000 00000000 00000000 11100101]补 = [00000000 00000000 00000000 11100101]原 =[229]十进制(int)

[-91]十进制（byte） = [11011011]原 = [10100101]补 ->转化成32位 [00000000 00000000 00000000 10100101]补 = [00000000 00000000 00000000 10100101]原 =[165]十进制(int)

[-67]十进制（byte） = [11000011]原 = [10111101]补 ->转化成32位 [00000000 00000000 00000000 10111101]补 = [00000000 00000000 00000000 10111101]原 =[189]十进制(int)

首先计算出内存中存储的补码二进制，再将值转化成32位的字节，高位无值的补0，在将得到的二进制转化成原码，再将原码转成十进制，就是int的数据了。

其实字节与0xff进行与运算，也可直接转化成int类型。

0xff我们可以理解它是int类型的，字节&0xff（补码和原码相同），就会强转成int类型。

字节-28对应的补码为11100100，与0xff进行与运算。

00000000 00000000 00000000 11100100

& 00000000 00000000 00000000 11111111

----------------------------------------------------------------

00000000 00000000 00000000 11100100

因高字节最高位为0，原码和补码相同，最终int类型

[00000000 00000000 00000000 11100100]补 = [00000000 00000000 00000000 11100100]原 = [128]十进制(int)

因为字节是8位，当转化为32位的int类型后，前三个字节都是0，只有后一个字节可以是非0的数，故转化后的int类型的范围为[0,255]。

按照上面的方式，**字节转int，直接就展示了内存中存储的补码对应于无符号的值**。一般可以进行其它操作，比如转化成十六进制，数据更具有可读性。

**实验三**

package com.anjz.test;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileStreamTest {

public static void main(String[] args) throws IOException {

fileStream(1);

}

public static void fileStream(int content) throws IOException{

File f = new File("C:\\Users\\Administrator\\Desktop\\a");

FileOutputStream fos = new FileOutputStream(f);

fos.write(content);

FileInputStream fis = new FileInputStream(f);

int a = fis.read();

System.out.println(a);

}

}

通过文件输出流，直接将int类型的数字写入文件中，再通过文件输入流，将内容读出来。

上述代码执行的结果为：1

将上述代码fileStream的参数修改为128，执行结果为：128

将上述代码fileStream的参数修改为256，执行结果为：0

将上述代码fileStream的参数修改为-1，执行结果为：255

上述实验的结果，有点让人摸不着头脑，通过输入流写入的数值，通过输出流读出的数值，有的是不一样的，这个是怎么回事呢？

首先我们看一下fos.write(content);的源码：

/\*\*

\* Writes the specified byte to this file output stream.

\*

\* @param b the byte to be written.

\* @param append {@code true} if the write operation first

\* advances the position to the end of file

\*/

private native void write(int b, boolean append) throws IOException;

/\*\*

\* Writes the specified byte to this file output stream. Implements

\* the <code>write</code> method of <code>OutputStream</code>.

\*

\* @param b the byte to be written.

\* @exception IOException if an I/O error occurs.

\*/

public void write(int b) throws IOException {

Object traceContext = IoTrace.fileWriteBegin(path);

int bytesWritten = 0;

try {

write(b, append);

bytesWritten = 1;

} finally {

IoTrace.fileWriteEnd(traceContext, bytesWritten);

}

}

通过查看源码，并没有看到什么特别之处，也没有看到特别需要注意的说明。但是通过实验发现一个规律：**当写入的值在[0,255]时，读出的值也是[0,255]，当值不在这个范围内，读出的值与写入的值是不样的**。通过观察，发现[0,255]是一个字节表示无符号数值的取值范围。虽然int类型用四个字节表示的，在这里，是不是进行了截断处理呢。按这个思路我们推测一下。

[1]十进制(int) = [00000000 00000000 00000000 00000001]原 = [00000000 00000000 00000000 00000001]补 ->截断取低8位 [00000001]补 （写入的值）->读取时，转化成32位[00000000 00000000 00000000 00000001]补 =[1]十进制(int)（读取的值）

[128]十进制(int) = [00000000 00000000 00000000 10000000]原 = [00000000 00000000 00000000 10000000]补 -> 截断取低8位 [10000000]补 （写入的值）->读取时，转化成32位[00000000 00000000 00000000 10000000]补 = [128]十进制(int)（读取的值）

[256]十进制(int) = [00000000 00000000 00000001 00000000]原 = [00000000 00000000 00000001 00000000]补 -> 截断取低8位 [00000000]补（写入的值）->读取时，转化成32位[00000000 00000000 00000000 00000000]补 = [0]十进制(int)（读取的值）

[-1]十进制(int) = [10000000 00000000 00000001 00000001]原 = [11111111 11111111 11111111 11111111]补 -> 截断取低8位 [11111111]补（写入的值）->读取时，转化成32位[00000000 00000000 00000000 11111111]补 = [255]十进制(int)（读取的值）

其实还有一种途径，去说明存入磁盘中的二进制是取低8位的补码，通过notepad++打开a文件。

文件流中输入：1 ，a文件展示的是：SOH，输出流中的值：1

文件流中输入：48 ，a文件中展示的是：0，输出流中的值：48

文件流中输入：65 ，a文件中展示的是：A，输出流中的值：65

文件流中输入：128 ，a文件展示的是：x80，输出流中的值：128

文件流中输入：258 ，a文件展示的是：STX，输出流中的值：2

通过查看ASCII表，可以发现文件中展示的都是ASCII码对应的字符，可以推测出，文件存入了一个8位的字节。

ASCII表查看地址：<http://tool.oschina.net/commons?type=4>

**实验四**

package com.anjz.test;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileStreamTest {

public static void main(String[] args) throws IOException {

fileStream(-1);

}

public static void fileStream(int content) throws IOException{

File f = new File("C:\\Users\\Administrator\\Desktop\\a");

FileOutputStream fos = new FileOutputStream(f);

fos.write(content);

FileInputStream fis = new FileInputStream(f);

byte[] b = new byte[1];

fis.read(b, 0, 1);

System.out.println(b[0]);

}

}

上述代码，输入为int类型，输出为byte类型，看看执行结果为：-1

将上述代码fileStream的参数修改为128，执行结果为：-128

将上述代码fileStream的参数修改为256，执行结果为：0

将上述代码fileStream的参数修改为1000，执行结果为：-24

我们通过实验三的理论推测一下：

[-1]十进制(int) = [10000000 00000000 00000001 00000001]原 = [11111111 11111111 11111111 11111111]补 -> 截断取低8位 [11111111]补 = [10000001]原 = [-1]十进制(byte)

[128]十进制(int) = [00000000 00000000 00000000 10000000]原 = [00000000 00000000 00000000 10000000]补 -> 截断取低8位 [10000000]补 =[-128]十进制(byte)（这个比较特殊10000000是没有原码和反码的，直接表示最小的数，主要还是因为不存在-0这么一说）

[256]十进制(int) = [00000000 00000000 00000001 00000000]原 =[00000000 00000000 00000001 00000000]补 -> 截断取低8位 [00000000]补 =[00000000]原 = [0]十进制(byte)

[1000]十进制(int) = [00000000 00000000 00000011 11101000]原 = [00000000 00000000 00000011 11101000]补 -> 截断取低8位 [11101000]补 =[10011000]原 =[-24]十进制(byte)

从上述实验得知，在流操作中，如果输入流写入的是int类型的值，一般写入低八位的数据，超出的部分都会被截断，为了防止写入的数据和读取的数据不一样，建议最好将int类型的范围控制在[0,127]上，这样读取的数据和写入的数据是一样的。如果写入的值不在[0,127]上，数据都是会发生变化的。最好在真实的项目中，直接用byte操作数据，就不会出现int类型转byte类型，截断的现象了。

总结

在分析这种问题时，总结了以下几条规则：

1. 字节是计算机读取的最小单位。
2. Java中数值是以补码的形式存在的，应用程序展示的十进制是补码对应真值。补码的存在主要为了简化计算机底层的运算，将减法运算直接当加法来做。
3. 字符串的编码是通过编码规范直接编码成二进制的，如果将编码后的二进制转化成字节数，就要将这些二进制当成补码来看，最终转化成数值。
4. Java中字节byte转化成整型int，可以理解成将有符号数转化成无符号数，通过扩展位数，来达到这种转化，也可以直接通过公式：字节数 & 0xff实现。

参考的文章：

<http://blog.csdn.net/xingtanzjr/article/details/50898122>

<http://blog.csdn.net/zdy10326621/article/details/50236529>