ByteBuffer类是在Java NIO中常常使用的一个缓冲区类，使用它可以进行高效的IO操作，但是，如果对常用方法的理解有错误，那么就会出现意想不到的bug。

ByteBuffer类的常用方法

先来看看一个基本的程序

public void test() throws IOException

{

ByteBuffer buff = ByteBuffer.allocate(128);

FileChannel fin = null;

FileChannel fout = null;

try

{

fin = new FileInputStream("filein").getChannel();

fout = new FileOutputStream("fileout").getChannel();

while(fin.read(buff) != -1) {

buff.flip();

fout.write(buff);

buff.clear();

}

}

catch (FileNotFoundException e)

{

} finally {

try {

if(fin != null) {

fin.close();

}

if(fout != null) {

fout.close();

}

} catch(IOException e) {

throw e;

}

}

}

在test方法中，首先通过ByteBuffer.allocate()方法分配了一段内存空间，作为缓存，allocate方法对缓存自动清零，然后打开一个输入文件管道fin和一个输出文件管道fout，在循环中先从fin读出数据存放到buff缓冲区中，再将buff缓冲中的内容写入fout。当然这对于先从文件中读，然后再写这样的场景，这不是高效的做法。   
可以看到先从fin中读出数据后，首先要调用ByteBuffer.flip()方法，若将数据写入输出文件后，还要调用ByteBuffer.clear()方法，为什么要这样做呢？

ByteBuffer可以作为一个缓冲区，是因为它是内存中的一段连续的空间，在ByteBuffer对象内部定义了四个索引，分别是mark，position，limit，capacity，其中

mark用于对当前position的标记

position表示当前可读写的指针，如果是向ByteBuffer对象中写入一个字节，那么就会向position所指向的地址写入这个字节，如果是从ByteBuffer读出一个字节，那么就会读出position所指向的地址读出这个字节，读写完成后，position加1

limit是可以读写的边界，当position到达limit时，就表示将ByteBuffer中的内容读完，或者将ByteBuffer写满了。

capacity是这个ByteBuffer的容量，上面的程序中调用ByteBuffer.allocate(128)就表示创建了一个容量为capacity字节的ByteBuffer对象。

了解了这四个变量之后，再来看看前面的程序。之所以调用ByteBuffer.flip()方法是因为在向ByteBuffer写入数据后，position为缓冲区中刚刚读入的数据的最后一个字节的位置，flip方法将limit值置为position值，position置0，这样在调用get\*()方法从ByteBuffer中取数据时就可以取到ByteBuffer中的有效数据，JDK中flip方法的代码如下:

public final Buffer flip() {

limit = position;

position = 0;

mark = -1;

return this;}

在调用four.write(buff)时，就将buff缓冲区中的数据写入到输出管道，此时调用ByteBuffer.clear()方法为下次从管道中读取数据做准备，但是调用clear方法并不将缓冲区的数据清空，而是设置position，mark，limit这三个变量的值，JDK中clear方法的代码如下：

public final Buffer clear() {

position = 0;

limit = capacity;

mark = -1;

return this;}

这个方法命名给人的感觉就是将数据清空了，但是实际上却不是的，它并没有清空缓冲区中的数据，而至重置了对象中的三个索引值，如果不清空的话，假设此次该ByteBuffer中的数据是满的，下次读取的数据不足以填满缓冲区，那么就会存在上一次已经处理的的数据，所以在判断缓冲区中是否还有可用数据时，使用ByteBuffer.hasRemaining()方法，在JDK中，这个方法的代码如下：

public final boolean hasRemaining() {

return position < limit;}

在该方法中，比较了position和limit的值，用以判断是否还有可用数据。

在ByteBuffer类中，还有个方法是compact，对于ByteBuffer，其子类HeapByteBuffer的compact方法实现是这样的：

public ByteBuffer compact() {

System.arraycopy(hb, ix(position()), hb, ix(0), remaining());

position(remaining());

limit(capacity());

return this;}

如果position()方法返回当前缓冲区中的position值，remaining()方法返回limit与position这段区间的长度，JDK中的remaining()方法代码如下

public final int remaining() {

return limit - position;}

所以compact()方法中第一条语句作用是将数组hb当前position所指向的位置开始复制长度为limit-position的数据到hb数组的开始出，其中使用到了ix()函数，这个函数是将参数值加上一个offset值，offset即一个偏移值，在这样的比如一个这样的场景对于一个很大的缓冲区，将其分成两段，第一段的起始位置是p1，长度是q1,第二段起始位置是p2，长度是q2，那么可以分别将这两段包装成一个HeapByteBuffer对象，然后这两个HeapByteBuffer对象(ByteBuffer的子类，默认实现)的offset属性分别设置为p1和p2，这样就可以通过在内部使用ix()函数来简化ByteBuffer对外提供的接口，在使用者看来，与默认的ByteBuffer并没有区别。

在compact函数中，接着将当前的缓冲区的position索引置为limit-position，limit索引置为缓冲区的容量，这样调用compact方法中就可以将缓冲区的有效数据全部移到缓冲区的首部，而position指向下一个可写位置。

比如刚刚创建一个ByteBuffer对象buff时，position=0，limit=capacity，那么此时调用buff.hasRemaining()则会返回true，这样来判断缓冲区中是否有数据是不行的，因为此时缓冲区中的存储的全部是0，但是调用一次compact()方法就可以将position置为limit值，这样再通过buff.hasRemaining()就会返回false，可以与后面的逻辑一起处理了。

ByteBuffer还有一个名为mark的方法，该方法设置mark索引为position的值，JDK中的代码如下：

public final Buffer mark() {

mark = position;

return this;}

与其功能相反的方法为reset方法，即将position的值设置为mark，JDK中的代码如下：

public final Buffer reset() {

int m = mark;

if (m < 0)

throw new InvalidMarkException();

position = m;

return this;}

此外还有一个名为rewind的方法，这个方法将position索引置为0，mark索引置为-1，JDK中的代码如下：

public final Buffer rewind() {

position = 0;

mark = -1;

return this;}

通过这些方法，就可以很方便的操作一个缓冲区，关键是要理解这些方法具体的作用，以及对三个索引值的影响(capacity是不变的)。

ByteBuffer继承自Buffer类，上面的方法四个索引值都定义在Buffer类中，操作索引值的方法也都定义在Buffer类中。

总结

通过对ByteBuffer中的四个索引值操作方法的分析，加深了对ByteBuffer的理解。理解ByteBuffer和其他几种Buffer的关键是要理解在使用中各个方法是如何操作索引值的，特别要注意的是clear方法并没有清除缓冲区的内容。