ByteBuffer: 理解ByteBuffer

版本信息

北京中科创达软件股份有限公司
ThunderSoft Co., Ltd.

版本	0.1
制定部门	智能视觉北京研发中心
制定日期	2020-08-30
密 级	绝密
文档状态	初稿
修订人	岳宗哲
修订日期	2020-08-30

目录

参考文章

1. https://zhuanlan.zhihu.com/p/56876443
2. https://blog.csdn.net/kesalin/article/details/566354
3. https://blog.csdn.net/u012345283/article/details/38357851

前言

ByteBuffer前前后后看过好几次了，实际使用也用了一些，总觉得条理不够清晰 😳。

基础介绍

ByteBuffer类位于java.nio包下，所谓nio:代表new io, 另一种解释：N代表Non-blocking IO,非阻塞的IO。

Buffer是一个抽象的基类，派生类：

> IntBuffer (java.nio)
> CharBuffer (java.nio)
> FloatBuffer (java.nio)
> DoubleBuffer (java.nio)
> ShortBuffer (java.nio)
> LongBuffer (java.nio)
> ByteBuffer (java.nio)

缓冲区(Buffer)就是在内存中预留指定大小的存储空间用来对输入/输出(I/O)的数据作临时存储，这部分预留的内存空间就叫做缓冲区：
使用缓冲区有这么两个好处：
1、减少实际的物理读写次数;
2、缓冲区在创建时就被分配内存，这块内存区域一直被重用，可以减少动态分配和回收内存的次数;
在Java NIO中，缓冲区的作用也是用来临时存储数据，可以理解为是I/O操作中数据的**中转站**。
缓冲区直接为通道(`Channel`)服务，写入数据到通道或从通道读取数据，这样的操利用缓冲区数据来传递就可以达到对数据高效处理的目的。

详细分析

1.Fields

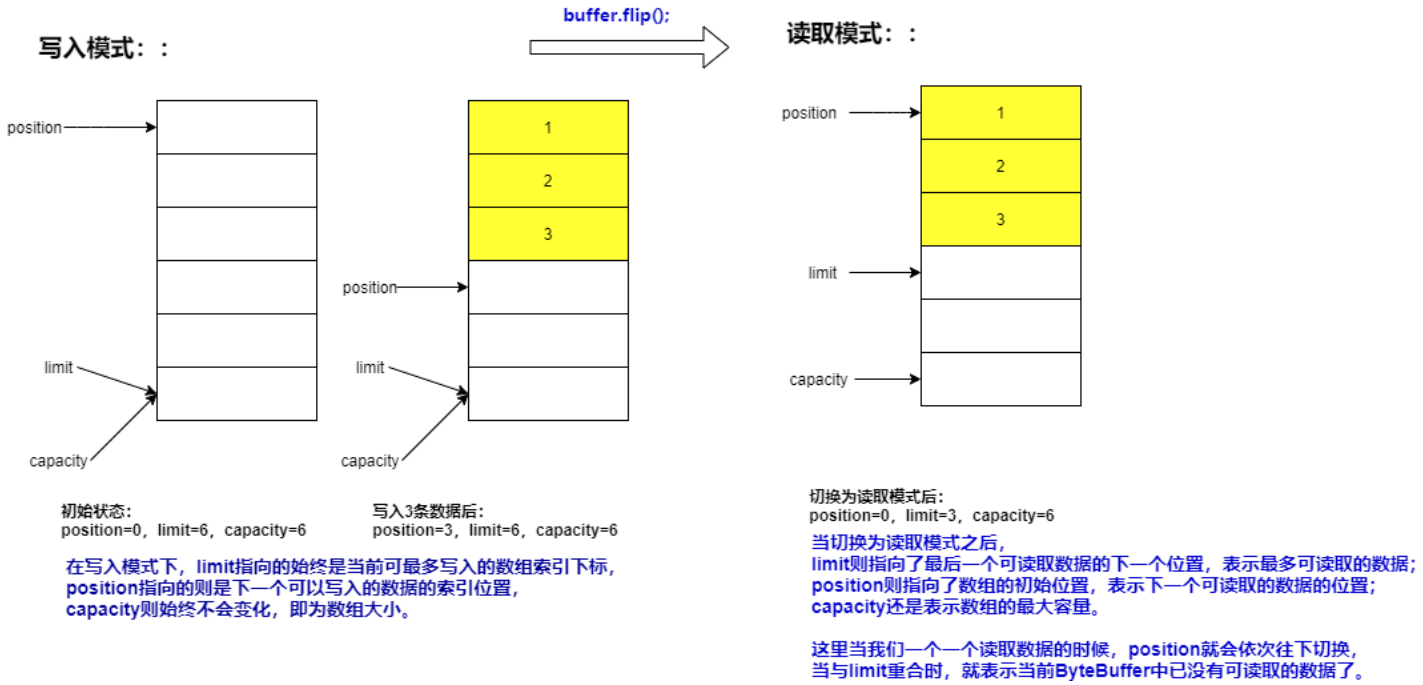
所有缓冲区都有4个属性：mark、position、limit、capacity，并遵循：mark <= position <= limit <= capacity，下表格是对着4个属性的解释：

```
// Invariants: mark <= position <= limit <= capacity
private int mark = -1;
private int position = 0;
private int limit;
private int capacity;
```

- mark：记录了当前所标记的索引下标；
- position：对于写入模式，表示当前可写入数据的下标，对于读取模式，表示接下来可以读取的数据的下标；
- limit：对于写入模式，表示当前可以写入的数组大小，默认为数组的最大长度，对于读取模式，表示当前最多可以读取的数据的位置下标；
- capacity：表示当前数组的容量大小；

最终的主要是position，limit和capacity三个属性，因为对于写入和读取模式，这三个属性的表示的含义大不一样。

1.1 写入模式 和 读取模式相应的图示



2.使用示例

2.1 写入模式 和 读取模式：

```
public class ByteBufferApp {
  @Test
  public void testBuffer() {
    // 初始化一个大小为6的ByteBuffer
    ByteBuffer buffer = ByteBuffer.allocate(6);
    print(buffer);  // 初始状态：position: 0, limit: 6, capacity: 6

    // 往buffer中写入3个字节的数据
    buffer.put((byte) 1);
    buffer.put((byte) 2);
    buffer.put((byte) 3);
    print(buffer);  // 写入之后的状态：position: 3, limit: 6, capacity: 6

    System.out.println("************** after flip **************");
    buffer.flip();
    print(buffer);  // 切换为读取模式之后的状态：position: 0, limit: 3, capacity: 6

    buffer.get();
    buffer.get();
    print(buffer);  // 读取两个数据之后的状态：position: 2, limit: 3, capacity: 6
  }

  private void print(ByteBuffer buffer) {
    System.out.printf("position: %d, limit: %d, capacity: %d\n",
      buffer.position(), buffer.limit(), buffer.capacity());
  }
}
```

2.2 mark()、reset()和flip()的使用:

mark属性，这个属性是一个标识的作用，即记录当前position的位置，在后续如果调用reset()或者flip()方法时，ByteBuffer的position就会被重置到mark所记录的位置。

```
public final Buffer mark() {
    mark = position;
    return this;
}

public final Buffer flip() {
    limit = position;
    position = 0;
    mark = -1;
    return this;
}

public final Buffer reset() {
    int m = mark;
    if (m < 0)
        throw new InvalidMarkException();
    position = m;
    return this;
}
```

> 对于写入模式，在mark()并reset()后，将会回到mark记录的可以写入数据的位置；
> 对于读取模式，在mark()并reset()后，将会回到mark记录的可以读取的数据的位置。

```
public class ByteBufferApp {
  @Test
  public void testMark() {
    ByteBuffer buffer = ByteBuffer.allocate(6);
    // position: 0, limit: 6, capacity: 6

    buffer.put((byte) 1);
    buffer.put((byte) 2);
    buffer.put((byte) 3);
    // position: 3, limit: 6, capacity: 6

    buffer.mark();  // 写入三个字节数据后进行标记
    // position: 3, limit: 6, capacity: 6

    buffer.put((byte) 4); // 再次写入一个字节数据
    // position: 4, limit: 6, capacity: 6

    buffer.reset(); // 对buffer进行重置，此时将恢复到Mark时的状态
    // position: 3, limit: 6, capacity: 6

    buffer.flip();  // 切换为读取模式，此时有三个数据可供读取
    // position: 0, limit: 3, capacity: 6

    buffer.get(); // 读取一个字节数据之后进行标记
    buffer.mark();
    // position: 1, limit: 3, capacity: 6

    buffer.get(); // 继续读取一个字节数据
    // position: 2, limit: 3, capacity: 6

    buffer.reset(); // 进行重置之后，将会恢复到mark的状态
    // position: 1, limit: 3, capacity: 6
  }
}
```

2.3 rewind()的使用：

rewind()和reset()方法都是进行重置的，但是reset()方法则是会优先重置到mark标记的位置。

> 对于写入模式，rewind()方法会重置为初始写入状态，
> 对于读取模式，rewind()则会重置为初始读取模式，其不会对limit属性有任何影响。

```
    public final Buffer rewind() {
        position = 0;
        mark = -1;
        return this;
    }

    public final Buffer reset() {
        int m = mark;
        if (m < 0)
            throw new InvalidMarkException();
        position = m;
        return this;
    }
```

```
public class ByteBufferApp {
  @Test
  public void testRewind() {
    ByteBuffer buffer = ByteBuffer.allocate(6);
    // position: 0, limit: 6, capacity: 6

    buffer.put((byte) 1);
    buffer.put((byte) 2);
    buffer.put((byte) 3);
    // position: 3, limit: 6, capacity: 6

    buffer.rewind();  // 调用rewind()方法之后，buffer状态将会重置
    // position: 0, limit: 6, capacity: 6
  }
}
```

2.4 compact()的使用：

对于compact()方法，其主要作用在于在读取模式下进行数据压缩，并且方便下一步继续写入数据。

```
public class ByteBufferApp {
  @Test
  public void testCompact() {
    ByteBuffer buffer = ByteBuffer.allocate(6);
    buffer.put((byte) 1);
    buffer.put((byte) 2);
    buffer.put((byte) 3);
    buffer.put((byte) 4);
    buffer.put((byte) 5);
    buffer.put((byte) 6); // 初始化一个写满的buffer

    buffer.flip();
    // position: 0, limit: 6, capacity: 6  -- 切换为读取模式

    buffer.get();
    buffer.get();
    // position: 2, limit: 6, capacity: 6  -- 读取两个字节后，还剩余四个字节

    buffer.compact();
    // position: 4, limit: 6, capacity: 6  -- 进行压缩之后将从第五个字节开始

    buffer.put((byte) 7);
    // position: 5, limit: 6, capacity: 6  -- 写入一个字节数据的状态
  }
}
```

compact()详细分析：：

比如在一个长度为6的ByteBuffer中写满了数据，然后在读取模式下读取了2个数据之后，我们想继续往buffer中写入数据，此时由于只有前2个字节是可用的，而后4个字节是有效的数据，此时如果写入的话是会把后面4个有效字节给覆盖掉的。因而需要将后面4个有效字节往前移动，以空出2个字节，并且将position指向下一个可供写入的位置，而不是迁移之后的索引0处。

HeapByteBuffer.java

```
public ByteBuffer compact() {
    System.arraycopy(hb, ix(position()), hb, ix(0), remaining());
    position(remaining());
    limit(capacity());
    discardMark();
    return this;
}

protected int ix(int i) {
    return i + offset;
}

public final int remaining() {
    return limit - position;
}
```

我们再来分析一下： **System.arraycopy**

```
public static native void arraycopy(Object src,  int  srcPos,
                                    Object dest, int destPos,
                                    int length);
```

src:源数组;　　srcPos:源数组要复制的起始位置;

dest:目的数组;　destPos:目的数组放置的起始位置;　length:复制的长度。

注意：src and dest都必须是同类型或者可以进行转换类型的数组.

有趣的是这个函数可以实现自己到自己复制，比如：

```
int[] fun ={0,1,2,3,4,5,6};

System.arraycopy(fun,0,fun,3,3);
```

则结果为：{0,1,2,0,1,2,6};

实现过程是这样的，先生成一个长度为length的临时数组,将fun数组中srcPos

到srcPos+length-1之间的数据拷贝到临时数组中，再执行System.arraycopy(临时数组,0,fun,3,3).

3.实例化

java.nio.Buffer类是一个抽象类，不能被实例化。Buffer类的直接子类，如ByteBuffer等也是抽象类，所以也不能被实例化。

但是ByteBuffer类提供了4个静态工厂方法来获得ByteBuffer的实例：

方法	描述
allocate(int capacity)	从堆空间中分配一个容量大小为capacity的byte数组作为缓冲区的byte数据存储器
allocateDirect(int capacity)	是不使用JVM堆栈而是通过操作系统来创建内存块用作缓冲区， 它与当前操作系统能够更好的耦合，因此能进一步提高I/O操作速度。 但是分配直接缓冲区的系统开销很大，因此只有在缓冲区较大并长期存在， 或者需要经常重用时，才使用这种缓冲区
wrap(byte[] array)	这个缓冲区的数据会存放在byte数组中， bytes数组或buff缓冲区任何一方中数据的改动都会影响另一方。 其实ByteBuffer底层本来就有一个bytes数组负责来保存buffer缓冲区中的数据， 通过allocate方法系统会帮你构造一个byte数组
wrap(byte[] array, int offset, int length)	在上一个方法的基础上可以指定偏移量和长度， 这个offset也就是包装后byteBuffer的position，而length呢就是limit-position的大小，从而我们可以得到limit的位置为length+position(offset)

详细方法的使用：：

```
        System.out.println("----------Test allocate--------");
        System.out.println("before alocate:" + Runtime.getRuntime().freeMemory());

        // 如果分配的内存过小，调用Runtime.getRuntime().freeMemory()大小不会变化?
        // 要超过多少内存大小JVM才能感觉到?
        ByteBuffer buffer = ByteBuffer.allocate(102400);
        System.out.println("buffer = " + buffer);

        System.out.println("after alocate:" + Runtime.getRuntime().freeMemory());

        // 这部分直接用的系统内存，所以对JVM的内存没有影响
        ByteBuffer directBuffer = ByteBuffer.allocateDirect(102400);
        System.out.println("directBuffer = " + directBuffer);
        System.out.println("after direct alocate:" + Runtime.getRuntime().freeMemory());

        System.out.println("----------Test wrap--------");
        byte[] bytes = new byte[32];
        buffer = ByteBuffer.wrap(bytes);
        System.out.println(buffer);

        buffer = ByteBuffer.wrap(bytes, 10, 10);
        System.out.println(buffer);
```

```
-----------------结果如下---------------------------------

before alocate:249989016
buffer = java.nio.HeapByteBuffer[pos=0 lim=102400 cap=102400]
after alocate:249989016
directBuffer = java.nio.DirectByteBuffer[pos=0 lim=102400 cap=102400]
after direct alocate:249989016
----------Test wrap--------
java.nio.HeapByteBuffer[pos=0 lim=32 cap=32]
java.nio.HeapByteBuffer[pos=10 lim=20 cap=32]
```

4.另外一些常用的方法

方法	描述
limit(), limit(10)	其中读取和设置这4个属性的方法的命名和jQuery中的val(),val(10)类似， 一个负责get，一个负责set
reset()	把position设置成mark的值，相当于之前做过一个标记， 现在要退回到之前标记的地方
clear()	position = 0;limit = capacity;mark = -1; 有点初始化的味道， 但是并不影响底层byte数组的内容

方法	描述
flip()	limit = position;position = 0;mark = -1; 翻转， 也就是让flip之后的position到limit这块区域变成之前的0到position这块， 翻转就是将一个处于存数据状态的缓冲区变为一个处于准备取数据的状态
rewind()	把position设为0，mark设为-1，不改变limit的值
remaining()	return limit - position; 返回limit和position之间相对位置差
hasRemaining()	return position < limit返回是否还有未读内容
compact()	把从position到limit中的内容移到0到limit-position的区域内， position和limit的取值也分别变成limit-position、capacity。 如果先将positon设置到limit，再compact，那么相当于clear()
get()	相对读，从position位置读取一个byte，并将position+1，为下次读写作准备
get(int index)	绝对读，读取byteBuffer底层的bytes中下标为index的byte，不改变position
get(byte[] dst, int offset, int length)	从position位置开始相对读，读length个byte， 并写入dst下标从offset到offset+length的区域
put(byte b)	相对写，向position的位置写入一个byte，并将postion+1，为下次读写作准备
put(int index, byte b)	绝对写，向byteBuffer底层的bytes中下标为index的位置插入byte b， 不改变position
put(ByteBuffer src)	用相对写，把src中可读的部分（也就是position到limit）写入此byteBuffer
put(byte[] src, int offset, int length)	从src数组中的offset到offset+length区域读取数据并使用相对写写入此byteBuffer

5.另外一些不常用的方法

```
public void testMethods() {
      ByteBuffer buffer = ByteBuffer.allocate(20);//分配20bytes大小的内存
      buffer.put((byte) 2);//1 byte
      buffer.get();
      buffer.putChar('a');//2 bytes
      buffer.getChar();
      buffer.putShort((short) 2);//2bytes
      buffer.getShort();
      buffer.putInt(123);//4bytes
      buffer.getInt();
      buffer.limit();
      //分为读写两种模式：当为写的模式时：返回值为缓存区的大小==buffer.capacity();
      //当为读的模式的时候，返回值为当前位置大小 == buffer.position()；以一个字节为计算单位。
      buffer.limit(0);//position=limit=0，写模式下重头覆盖缓冲区，与buffer.clear()效果相同。
      buffer.hasRemaining();//内存空间是否有剩余
      buffer.clear();//清除缓冲区
      buffer.flip().array();//将buffer中的内容以字节形式返回
  }
```