Nio缓冲区分析二

本文主要讲解缓冲区的分配、缓冲区分片和只读缓冲区。由于ByteBuffer是经常使用的缓冲区，所有使用ByteBuffer为例。

1. **缓冲区的分配（也是创建buffer的方法）**

1）调用ByteBuffer的静态方法allocate(capacity)方法，去分配一个指定容量的ByteBuffer缓冲区，该缓冲区的元素的值都为0。实质是调用ByteBuffer的构造函数来分配的。

/\*\*

\* 缓冲区的分配

\* **@author** 21463

\*

\*/

**public** **class** BufferAllocate {

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

//1.分配一个容量为10 的缓冲区，实质指定了数组的大小为10

ByteBuffer buffer = ByteBuffer.*allocate*(10);

}

}

2）调用ByteBuffer的静态方法wrap(array)方法，通过一个byte数组来初始化ByteBuffer缓冲区。

/\*\*

\* 缓冲区的分配

\* **@author** 21463

\*

\*/

**public** **class** BufferAllocate {

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

//1.分配一个容量为10 的缓冲区，实质指定了数组的大小为10

ByteBuffer buffer = ByteBuffer.*allocate*(10);

//2.通过调用wrap方法来，使用一个byte数组来分配一个ByteBuffer缓冲区

//并使用数组来初始化缓冲区。

**byte**[] array = **new** **byte**[10];

**for**(**byte** i = 0;i < array.length ; i++){

array[i] = i;

}

ByteBuffer buffer2 = ByteBuffer.*wrap*(array);

ByteBuffer buffer3 = ByteBuffer.*wrap*(array, 1, 5);

}

}

1. **缓冲区的分片（实质就是获取buffer的子缓冲区，子缓冲区和父缓冲区共享数据）**

缓冲区的分片是调用ByteBuffer的slice()方法来实现的，调用该方法会返回buffer的一

个子buffer，并且改子缓冲区中数据的修改会体现到buffer中，说明slice()方法创建的子buffer和buffer是共享数据的。

slice()方法创建子buffer，实质是使用buffer中的hb即缓冲数据所有的数组的引用去创建的子buffer，因此buffer和其子buffer是共享数据的，更改子buffer中的数据可以体现到buffer中。ByteBuffer的slice()方法创建子ByteBuffer的代码如下：

**public** ByteBuffer slice() {

**return** **new** HeapByteBuffer(hb,

-1,

0,

**this**.remaining(),

**this**.remaining(),

**this**.position() + offset);

}

其中hb就是ByteBuffer实例中缓冲数据所用的数组，remaining()方法返回的是position和limit之间的元素个数，在这里就是子buffer的容量和limit的值。并且通过buffer的slice()方法返回的是子buffer是buffer当前的position和limit之间的元素。因此，可以通过设置buffer的position和limit位置，来获取buffer中指定的区域的数据。

缓冲区简单分片例子如下：

/\*\*

\* 缓冲区的分片

\* **@author** 21463

\*

\*/

**public** **class** BufferSlience {

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

//分配缓冲区

ByteBuffer buffer = ByteBuffer.*allocate*(10);

//向缓冲区中写入1-10

buffer.clear();//重置缓冲区，一般在写入数据前调用该方法

**for**(**byte** i = 1;i <= 10 ;i++){

buffer.put(i);

}

buffer.flip();//固定缓冲数据区域

//position()是返回当前缓冲区的position值

//将position的值定位到newPosition的位置

buffer.position(3);

//设置limit的值定位到newLimit的位置。

buffer.limit(7);

//获取子缓冲区

ByteBuffer slice = buffer.slice();

**for**(**byte** i = 0;i < slice.capacity();i ++){

**byte** b = slice.get(i);//通过索引获取buffer中的元素值

b \*= 10;

//将数据写入到指定的位置

slice.put(i, b);//通过元素索引，设置元素的值

}

buffer.clear();//重置将position 设置为0，limit设置为capacity

**while**(buffer.hasRemaining()){

System.***out***.println(buffer.get());

}

}

}

上述代码运行的效果如下：

1 2 3 40 50 60 70 8 9 10

**3.只读缓冲区**

只读缓冲区，只能读取缓冲区的数据，不能向中写入数据。只读缓冲区可以通过调用adReadOnlyBuffer()方法来创建，创建的只读缓冲区和源缓冲区是共享数据缓冲的数组。因此，原缓冲区中的数据发送改变，那么对应的只读缓冲区也相应发生变化。通过asReadOnlyBuffer()创建的只读缓冲区的代码如下：

**public** ByteBuffer asReadOnlyBuffer() {

**return** **new** HeapByteBufferR(hb,

**this**.markValue(),

**this**.position(),

**this**.limit(),

**this**.capacity(),

offset);

}

从代码中可以看出，创建的只读缓冲区是和buffer共享数据的新的缓冲区，相当于是当前buffer部分区域数据只读的快照，和buffer共享数据。

示例代码如下：

/\*\*

\* 只读缓冲区

\* **@author** 21463

\*

\*/

**public** **class** BufferRead {

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

ByteBuffer buffer = ByteBuffer.*allocate*(10);

**for**(**byte** i = 1 ;i <= buffer.capacity();i++){

buffer.put(i);

}

buffer.flip();

//实质是HeapByteBufferR缓冲对象，该缓冲对象的修改操作是直接报错的。

//只能读取缓冲区中的对象，不能修改。并且和buffer共享hb数组，即缓冲区

ByteBuffer readOnlyBuffer = buffer.asReadOnlyBuffer();

//readOnlyBuffer.put((byte)12);//直接报java.nio.ReadOnlyBufferException异常

readOnlyBuffer.clear();

**while**(readOnlyBuffer.hasRemaining()){

System.***out***.print(readOnlyBuffer.get() + " ");

}//1 2 3 4 5 6 7 8 9 10

System.***out***.println();

buffer.clear();//初始化buffer

buffer.position(3);

buffer.limit(7);

//对buffer的修改

**for**(**byte** i = (**byte**) buffer.position();i < buffer.limit();i ++ ){

**byte** b = buffer.get(i);

b \*= 10;

buffer.put(i, b);

}

//在读取时，需要对确定读取数据的范围。

buffer.flip();

readOnlyBuffer.flip();

**while**(readOnlyBuffer.hasRemaining()){

System.***out***.print(readOnlyBuffer.get() + " ");

}//1 2 3 40 50 60 70 8 9 10

}

}

4.直接缓冲区（）

直接缓冲区是为加快I/O速度，使用一种特殊方式为其分配内存的缓冲区，JDK文档中的描述为：**给定一个直接字节缓冲区，Java虚拟机将尽最大努 力直接对它执行本机I/O操作。也就是说，它会在每一次调用底层操作系统的本机I/O操作之前(或之后)，尝试避免将缓冲区的内容拷贝到一个中间缓冲区中 或者从一个中间缓冲区中拷贝数据。**要分配直接缓冲区，需要调用allocateDirect()方法，而不是allocate()方法，使用方式与普通缓冲区并无区别，如下面的拷贝文件示例：

/\*\*

\* 直接缓冲区

\* **@author** 21463

\*

\*/

**public** **class** BufferDirect {

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

FileInputStream fis = **new** FileInputStream("hello.txt");

FileChannel fisChannel = fis.getChannel();

FileOutputStream fos = **new** FileOutputStream("hello\_copy.txt");

FileChannel fosChannel = fos.getChannel();

//开辟一个直接缓冲区

ByteBuffer directBuffer = ByteBuffer.*allocateDirect*(1024);

**while**(fisChannel.read(directBuffer) != -1){

directBuffer.flip();

fosChannel.write(directBuffer);

directBuffer.clear();

}

fis.close();

fisChannel.close();

fos.close();

fosChannel.close();

}

}

从直接缓冲区创建的过程来看，调用allocateDirect()方法实质是创建DirectByteBuffer对象，而DirectByteBuffer是继承于MappedByteBuffer，MappedByteBuffer是文件的映射区域的直接缓冲区。

5. **内存映射文件I/O**

内存映射文件I/O是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的I/O快的多。内存映射文件I/O是通过使文件中的数据出现为 内存数组的内容来完成的，这其初听起来似乎不过就是将整个文件读到内存中，但是事实上并不是这样。一般来说，**只有文件中实际读取或者写入的部分才会映射到内存中**。如下面的示例代码：

**public** **class** BufferMapped {

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

RandomAccessFile raf = **new** RandomAccessFile("hello.txt", "rw");

FileChannel channel = raf.getChannel();

/\*\*

\* MappedByteByteBuffer是通过流对象的getChannel()方法去获取

\* FileChannel对象，然后通过该channel对象的map方法来创建MappedByteBuffer

\* 对象，map方法有三个参数：

\* mode：是MappedByteBuffer的模式，由FileChannel.MapMode指定：READ\_ONLY、READ\_WRITE、PRIVATE

\* position:指定从文件的字节数组中的哪个位置开始创建MappedByteBuffer缓冲区。

\* size：指定获取的容器。

\*/

MappedByteBuffer map = channel.map(FileChannel.MapMode.***READ\_WRITE***, 0, 1024);

System.***out***.println("文件中的内容的字符的ASCII码为：");

**while**(map.hasRemaining()){

System.***out***.print(map.get() + " ");

}

System.***out***.println();

map.flip();

channel.write(map);

raf.close(); }

}

从代码中可以看出，MappedByteBuffer是文件的内存映射的直接缓冲区。MappedByteBuffer中的数据就是从文件中读取的。MappedByteBuffer对象是通过FileChannel对象调用其map(mode,position,size)方法来创建的，由此可以看出，MappedByteBuffer是用于操作文件时的首选缓冲区，处理速度比传统IO和通道IO快很多。