链滴

Java NIO深入探究

作者：guobing

原文链接：https://ld246.com/article/1470842255537

来源网站：链滴

<pre class="vditor-yml-front-matter"><code class="language-yaml">### 1. socket / web socket区别
一直以来对两者的区别不是很清楚，今天尝试总结一下：
> * `socket是一般的app用的，客户端是任何的socket client`
> * `websocket 是web上用，客户端一般是浏览器上的js`
所以socket在web中是用不了的
> * `socket是应用层和传输层之间的一层抽象层。把复杂的tcp/ip操作封装成几个简单的接口。`
> * `websocket是应用层协议`

2. 流与块的比较
原来的 I/O 库(在 java.io.*中) 与 NIO 最重要的区别是数据打包和传输的方式。原来的 I/O 以流的方处理数据，而 NIO 以块的方式处理数据。
面向流 的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一字节的数据。

3. NIO概述
NIO的核心就是Channels、Buffers、Selectors这三部分。
Channel和Buffer有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：
`FileChannel`: 从文件中读写数据。
`DatagramChannel`: 通过UDP读写网络中的数据。
`SocketChannel`: 能通过TCP读写网络中的数据。
`ServerSocketChannel`: 可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都创建一个SocketChannel。
正如你所看到的，这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。
NIO中的buffer主要有：
```
ByteBuffer
CharBuffer
DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer
```
3.1 **Selector**
Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量很低，使用Selector就会很方便。例如，在一个聊天服务器中。
要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来数据接收等。

3.2 **buffer**
使用Buffer读写数据一般遵循以下四个步骤：

1. 写入数据到Buffer
2. 调用flip()方法
3. 从Buffer中读取数据
4. 调用clear()方法或者compact()方法

当向buffer写入数据时，buffer会记录下写了多少数据。一旦要读取数据，需要通过flip()方法将Buffe从写模式切换到读模式。在读模式下，可以读取之前写入到buffer的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用cl ar()或compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任

未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

下面是一个使用Buffer的例子：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();

//create buffer with capacity of 48 bytes
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //read into buffer.
while (bytesRead != -1) {

  buf.flip();  //make buffer ready for read

  while(buf.hasRemaining()){
      System.out.print((char) buf.get()); // read 1 byte at a time
  }

  buf.clear(); //make buffer ready for writing
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

3.3 **Buffer的capacity,position和limit**

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成NIO Buffer对，并提供了一组方法，用来方便的访问该块内存。

为了理解Buffer的工作原理，需要熟悉它的三个属性：

capacity
position
limit
position和limit的含义取决于Buffer处在读模式还是写模式。不管Buffer处在什么模式，capacity的义总是一样的。
capacity

作为一个内存块，Buffer有一个固定的大小值，也叫“capacity”.你只能往里写capacity个byte、lon，char等类型。一旦Buffer满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写据。

position

当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte、long等数写到Buffer后， position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1.

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0.从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

limit

在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于Buffer的capacity。

当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读模式时，limit被设置成写模式下的position值。换句话说，你能读到之前写入的所有数据（limit被设置成已写数据数量，这个值在写模式下就是position）

3.4. **Buffer的分配**
要想获得一个Buffer对象首先要进行分配。 每一个Buffer类都有一个allocate方法。下面是一个分配4字节capacity的ByteBuffer的例子。

`ByteBuffer buf = ByteBuffer.allocate(48);`
`flip()`方法
flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前postion的值。

换句话说，position现在用于标记读的位置，limit表示之前写进了多少个byte、char等 —— 现在能取多少个byte、char等。
`rewind()`方法
Buffer.rewind()将position设回0，所以你可以重读Buffer中的所有数据。limit保持不变，仍然表示从Buffer中读取多少个元素（byte、char等）。

`clear()与compact()方法`
一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或compact()方法来完。

如果调用的是clear()方法，position将被设回0，limit被设置成 capacity的值。换句话说，Buffer 被空了。Buffer中的数据并未清除，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。

如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告你哪些数据被读过，哪些还没有。

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用compct()方法。

compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未的数据。
mark()与reset()方法

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定position。之后可以通过调用Buffer.reset(方法恢复到这个position。例如：

```
buffer.mark();
//call buffer.get() a couple of times, e.g. during parsing.
buffer.reset();  //set position back to mark.
```

equals()与compareTo()方法

可以使用equals()和compareTo()方法两个Buffer。

equals()

当满足下列条件时，表示两个Buffer相等：

有相同的类型（byte、char、int等）。

Buffer中剩余的byte、char等的个数相等。
Buffer中所有剩余的byte、char等都相同。
如你所见，equals只是比较Buffer的一部分，不是每一个在它里面的元素都比较。实际上，它只比较Buffer中的剩余元素。

compareTo()方法

compareTo()方法比较两个Buffer的剩余元素(byte、char等)， 如果满足下列条件，则认为一个Buffer“小于”另一个Buffer：

第一个不相等的元素小于另一个Buffer中对应的元素 。
所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

4. 通道之间的数据传输
在Java NIO中，如果两个通道中有一个是FileChannel，那你可以直接将数据从一个channel传输到外一个channel。
`transferFrom()`
FileChannel的transferFrom()方法可以将数据从源通道传输到FileChannel中
```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count = fromChannel.size();

toChannel.transferFrom(position, count, fromChannel);
```
方法的输入参数position表示从position处开始向目标文件写入数据，count表示最多传输的字节数
如果源通道的剩余空间小于 count 个字节，则所传输的字节数要小于请求的字节数。
此外要注意，在SoketChannel的实现中，SocketChannel只会传输此刻准备好的数据（可能不足count字节）。因此，SocketChannel可能不会将请求的所有数据(count个字节)全部传输到FileChannel
。

`transferTo()`
transferTo()方法将数据从FileChannel传输到其他的channel中。
```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count = fromChannel.size();

fromChannel.transferTo(position, count, toChannel);
```
5. Java NIO之Selector
Selector（选择器）是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件好准备的组件。这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。

5.1. **为什么使用Selector?**
仅用单个线程来处理多个Channels的好处是，只需要更少的线程来处理通道。事实上，可以只用一个程处理所有的通道。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系的一些资源（如内存）。因此，使用的线程越少越好。
5.2. Selector的创建
通过调用Selector.open()方法创建一个Selector，如下：
`Selector selector = Selector.open();`
向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。通过SelectableChannel.register()方法来实现，如下：
```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector,Selectionkey.OP_READ);
```
与Selector一起使用时，Channel必须处于非阻塞模式下。这意味着不能将FileChannel与Selector一使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。
注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel对什么事件感兴趣。可以监听四种不同类型的事件：
```
Connect
Accept
Read
Write
```
通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。
这四种事件用SelectionKey的四个常量来表示：
```
SelectionKey.OP_CONNECT
SelectionKey.OP_ACCEPT
SelectionKey.OP_READ
SelectionKey.OP_WRITE
```
如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：
`int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;`
在下面还会继续提到interest集合。</code></pre>