Java NIO 概述

# 一、Java NIO概述

NIO（Non-blocking I/O， Java中也称为New I/O），是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

**Channel和Buffer**

在标准 IO API 中，使用字节流和字符流。 在 NIO 中使用通道和缓冲区。 数据总是从通道读入缓冲区，或从缓冲区写入通道。

**非阻塞IO**

NIO 可以执行非阻塞 IO 。 例如，当通道将数据读入缓冲区时，线程可以执行其他操作。 并且一旦数据被读入缓冲区，线程就可以继续处理它。 将数据写入通道也是如此。

**Selector**

NIO选择器是一个可以监视多个事件通道的对象（例如：连接打开，数据到达等）。 因此，单个线程可以监视多个通道的数据。

Channel，Buffer 和 Selector 构成了NIO API 的核心。 其余的组件，如 Pipe 和 FileLock ，只是与三个核心组件一起使用的实用程序类。

## **1.1 NIO与传统BIO区别**

|  |  |
| --- | --- |
| BIO | NIO |
| Stream oriented | Buffer oriented |
| Blocking IO | Non-blocking IO |
|  | Selectors |

## **1.2 面向流与面向缓冲**

Java BIO面向流的; 意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据，如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。

Java NIO是面向缓冲区的; NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是需要检查该缓冲区中是否包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。 

## **1.3 阻塞与非阻塞**

Java BIO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。

Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。

非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

NIO通讯是将整个任务切换成许多小任务，由一个线程负责处理所有IO事件，并负责分发。它是利用**事件驱动机制**，而不是监听机制，事件到的时候再触发。NIO线程之间通过wait，notify等方式通讯。保证了每次上下文切换都有意义，减少无谓的进程切换。

## **1.4 选择器Selectors**

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

# 二、Channel

## **2.1 通道简介**

|  |  |
| --- | --- |
| FileChannel | 读写文件中的数据  (无法设置为非阻塞模式，他总是运行在阻塞模式下) |
| SocketChannel | 通过TCP读写网络中的数据 |
| ServerSocketChannel | 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会创建一个SocketChannel  (read，write 前必须建立连接，线程安全，任意时刻只能有一个线程进行读取和写入) |
| DatagramChannel | 通过UDP读写网络中的数据  (send，receive 前无需建立连接，read，write前必须建立连接，线程安全，任意时刻只能有一个线程进行读取和写入) |
| AsynchronousFileChannel | 异步读取数据并将数据写入文件 |

Java NIO的通道类似流，但又有些不同：

* 既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。
* 通道可以异步地读写。
* 通道中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。

从通道读取数据到缓冲区，从缓冲区写入数据到通道。如下图所示：



## **2.2 通道常用方法**

| **方 法** | **描 述** |
| --- | --- |
| int read(ByteBuffer dst) | 从Channel到中读取数据到ByteBuffer |
| long read(ByteBuffer[] dsts) | 将Channel到中的数据“分散”到ByteBuffer[] |
| int write(ByteBuffer src) | 将ByteBuffer到中的数据写入到Channel |
| long write(ByteBuffer[] srcs) | 将ByteBuffer[]到中的数据“聚集”到Channel |
| long position() | 返回此通道的文件位置 |
| FileChannel position(long p) | 设置此通道的文件位置 |
| long size() | 返回此通道的文件的当前大小 |
| FileChannel truncate(long s) | 将此通道的文件截取为给定大小 |
| void force(boolean metaData) | 强制将所有对此通道的文件更新写入到存储设备中 |

## **2.3 获取通道--getChannel方法**

java针对支持通道的类提供了FileChannel getChannel()方法。

|  |  |
| --- | --- |
| 本地IO | 网络IO |
| FileInputStream | Socket |
| FileOutputStream | Socket |
| RandomAccessFile | ServerSocket |

## **2.4 通道间数据传输--read & write**

outChannel.write(buff); //将 Buffer 中数据写入 Channel

inChannel.read(buff) ; //从 Channel 读取数据到 Buffer

## **2.5 通道间数据传输--transferFrom**

从源信道读取字节到这个通道的文件中。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。

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）

* src : 源通道
* position : 调动开始的文件内的位置，必须是非负的
* count : 要传输的最大字节数，必须是非负

public abstract long transferFrom(ReadableByteChannel src, long position, long count) throws IOException;

## **2.6 通道间数据传输--transferTo**

将字节从此通道的文件传输到给定的可写入字节通道。

* position : 调动开始的文件内的位置，必须是非负的
* count : 要传输的最大字节数，必须是非负
* target : 目标通道

public abstract long transferTo(long position, long count, WritableByteChannel target) throws IOException;

# 三、Buffer

* ByteBuffer
* MappedByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

NIO中的Buffer用于和通道进行交互。数据是从通道读入缓冲区，从缓冲区写入到通道中的。



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对象，并提供了一组方法，用来方便的访问该块内存。

## **3.1 ByteBuffer方法**

| **方法** | **描述** |
| --- | --- |
| byte[] array() | 返回实现此缓冲区的 byte 数组，此缓冲区的内容修改将导致返回的数组内容修改，反之亦然。 |
| CharBuffer asCharBuffer() | 创建此字节缓冲区作为新的独立的char 缓冲区。新缓冲区的内容将从此缓冲区的当前位置开始 |
| XxxBuffer asXxxBuffer() | 同上，创建对应的 Xxx 缓冲区，Xxx 可为 Short/Int/Long/Float/Double |
| byte get() | 相对 get 方法。读取此缓冲区当前位置的字节，然后该 position 递增。 |
| ByteBuffer get(byte[] dst, int offset, int length) | 相对批量 get 方法，后2个参数可省略 |
| byte get(int index) | 绝对 get 方法。读取指定索引处的字节。 |
| char getChar() | 用于读取 char 值的相对 get 方法。 |
| char getChar(int index) | 用于读取 char 值的绝对 get 方法。 |
| xxx getXxx(int index) | 用于读取 xxx 值的绝对 get 方法。index 可以选，指定位置。 |
| 众多 put() 方法 | 参考以上 get() 方法 |
| static ByteBuffer wrap(byte[] array) | 将 byte 数组包装到缓冲区中。 |

## **3.2 capacity,position,limit**



### **1. 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、long，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数据。

### **2. 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等数据写到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. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

### **3. 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会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被设置成已写数据的数量，这个值在写模式下就是position）

### **3. mark**

位置标记，用于记录某一次的读写位置，可以通过 reset 重新回到这个位置;

标记、位置、限制和容量值遵守以下不变式：  
0 <= mark<= position <= limit<= capacity

## **3.3 allocate(int capacity)**

分配容量為capacity的缓冲区。

**源码：**

public static ByteBuffer allocate(int capacity) {

if (capacity < 0)

throw new IllegalArgumentException();

return new HeapByteBuffer(capacity, capacity);

}

## **3.4 flip()--切换读写模式**

flip() 方法将 Buffer 从写入模式切换到读取模式;

会将 position 设置回 0，并将 limit 的值设置为切换之前的 position 值。

**源码：**

public final Buffer flip() {

limit = position;

position = 0;

mark = -1;

return this;

}

## **3.5 rewind()--倒带**

Rewinds this buffer. The position is set to zero and the mark is discarded。

**源码：**

public final Buffer rewind() {

position = 0;

mark = -1;

return this;

}

## **3.6 clear() 和 compact()**

如果调用 clear() ,则将 position 设置回 0 ，并将 limit 被设置成 capacity 的值。换句话说，Buffer 被清空了。 但是 Buffer 中的实际存放的数据并未清除。

如果在调用 clear() 时缓冲区中有任何未读数据，数据将被“遗忘”，这意味着不再有任何标记告诉读取了哪些数据，**还没有**读取哪些数据。

**源码：**

public final Buffer clear() {

position = 0;

limit = capacity;

mark = -1;

return this;

}

如果缓冲区中仍有未读数据，并且想稍后读取它，但需要先写入一些数据，这时候应该调用 compact() ，它会将所有未读数据复制到 Buffer 的开头，然后它将 position 设置在最后一个未读元素之后。 limit 属性仍设置为 capacity ，就像 clear() 一样。 现在缓冲区已准备好写入，并且不会覆盖未读数据。

## **3.7 mark() 和 reset()**

以通过调用 Buffer 对象的 mark() 方法在 Buffer 中标记给定位置。 然后，可以通过调用 Buffer.reset() 方法将位置重置回标记位置，就像在标准 IO 中一样。

**源码：**

public final Buffer mark**() {**

mark **=** position;

return this;

}

public final Buffer reset() {

int m = mark;

if (m < 0)

throw new InvalidMarkException();

position = m;

return this;

}

## **3.8 equals() 和 compareTo()**

### **1, equals()**

当满足下列条件时，表示两个Buffer相等：

a) 有相同的类型（byte、char、int等）。

b) Buffer中剩余的byte、char等的个数相等。

c) 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。

### **2, compareTo()**

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：

a) 第一个不相等的元素小于另一个Buffer中对应的元素 。

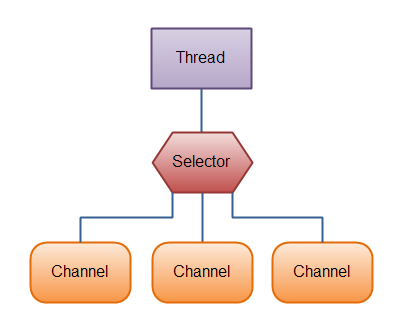
b) 所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

**（译注：剩余元素是从 position到limit之间的元素）**

# 四、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做好准备的组件。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。

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，可以只用一个线程处理所有的通道。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统的一些资源（如内存）。因此，使用的线程越少越好。



## **4.1 SelectionKey-- interest 集合**

Channel并非一定要支持所有的四种操作。比如服务器通道ServerSocketChannel支持Accept接受操作，而SocketChannel客户端通道则不支持。

可以通过通道上的validOps()方法，来获取特定通道下所有支持的操作集合

* SelectionKey.OP\_CONNECT 连接
* SelectionKey.OP\_ACCEPT 接收
* SelectionKey.OP\_READ 读
* SelectionKey.OP\_WRITE 写

## **4.2 SelectionKey-- register(Selector sel, int ops)**

**SelectionKey selectionKey = channel.register(selector, interestSet)**

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一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

register() 方法返回一个 SelectionKey 对象。 这个 SelectionKey 对象包含的属性：

* interest 集合
* ready 集合
* 对应 Channel
* 对应 Selector
* 附加对象（可选）

## **4.3 select(long timeout)**

**select()** 阻塞到至少有一个通道在你注册的事件上就绪了,方法返回的int值表示有多少通道已经就绪

**select(timeout)** 和select()一样，除了最长会阻塞timeout毫秒(参数)

**(** 注：**自前一次select方法到当前select方法之间的时间段上，有多少通道变成就绪状**)

**selectNow()** 不会阻塞，不管什么通道就绪都立刻返回（注：此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零）

## **4.4 selectedKeys()**

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。

SelectionKey.isAcceptable()

SelectionKey.isConnectable()

SelectionKey.isReadable()

SelectionKey.isWritable()

( 注：**每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除**)

## **4.5 wakeUp()和close()**

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，如果让其从select()方法返回，只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在select()方法上的线程会立马返回。

如果有其它线程调用了wakeUp()方法，但当前没有线程阻塞在select()方法上，下个调用select()方法的线程会立即“醒来（wake up）” 。

**close()** 方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭。

# 六、同步/异步、阻塞/非阻塞关系

同步/异步和阻塞/非阻塞没有关系(一般异步都是和非阻塞组合使用)：

## **.1 同步/异步**

**同步**是同时只能有一个线程处理某个对象或者操作，例如一个线程占用了一个对象，其他线程如果要访问此对象，则需要等之前得线程操作完成返回；相关概念有同步方法、同步代码块、对象锁。

**异步**是线程处理某个耗时操作（或者对象已经被别的线程占用了），不等待而是去做其他事情，也不主动查看是否完成，而是等耗时操作完成，发通知再叫线程回来处理结果；常见的例子就是 Ajax ，相关概念有回调函数等。

## **.2 阻塞/非阻塞**

**阻塞**是线程的一个状态，线程发起任务请求然后一直等，直到到任务完成再把结果返回，如果任务未完成当前线程会被挂起。

**非阻塞**是发起任务请求之后先马上返回去做别的事，然后再时不时主动查看任务请求是否被完成（轮询）。