Java NIO

Java NIO (new IO) — расширение системы ввода/вывода Java, добавляющий множество функций и утилит, как например, буферы, отображенные в памяти файлы, и многое другое. Но главное, NIO меняет подход к работе с вводом/выводом, к примеру, возможно работать в неблокирующем режиме.

NIO вводит дополнительные понятия:

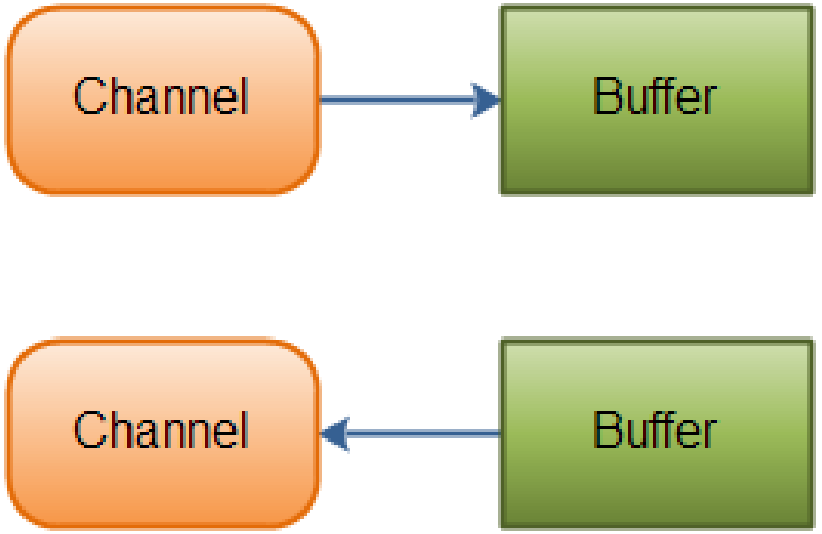
- Channel — канал ввода/вывода;
- Buffer — буфер, при помощи которого происходит чтение и запись из/в канал;
- Selector — позволяет однопоточно переключатся между каналами по событиям.

Канал (Channel) имеет следующие различия с потоками ввода\вывода:

- Канал можно использовать как для записи, так и для чтения, в то время как потоки ввода/вывода однонаправленные;
- Писать и читать из/в канала можно в неблокирующем режиме;
- Работа с каналами происходит при помощи буфера.

Наиболее часто используемые реализации Channel:

- FileChannel — работа с файлами;
- DatagramChannel — передача по UDP протоколу;
- SocketChannel — TCP клиент;
- ServerSocketChannel — TCP сервер.



Пример записи в файл:

```
private static void write(File file) throws IOException {
        try (RandomAccessFile raf = new RandomAccessFile(file, "rw")) {
            // Канал можно получить из RandomAccessFile или
            // FileInputStream/FileOutputStream
            FileChannel ch = raf.getChannel();

            String text = "Hello!\n" +
                    "Hello!\n" +
                    "Hello!!!\n";

            byte[] bytes = text.getBytes(Charset.forName("utf-8"));

            // Создаем буфер для записи данных.
            ByteBuffer buf = ByteBuffer.allocate(512);
//            ByteBuffer buf = ByteBuffer.wrap(bytes);

            // Заполняем буфер данными.
            buf.put(bytes);

            // Ставим лимит на место курсора, а курсор перемещаем в начало.
            buf.flip();

            // Записываем данные из буфера в канал (курсор будет перемещен
            // на позицию равную количеству записанных байт)
            int written = ch.write(buf);

            System.out.println(">> Written " + written + " bytes to file " +
                                                        file.getAbsolutePath());
        }
    }
```

Пример чтения из файла:

```
private static void read(File file) throws IOException {
    try (RandomAccessFile raf = new RandomAccessFile(file, "rw")) {
        // Канал можно получить из RandomAccessFile или
        // FileInputStream/FileOutputStream
        FileChannel ch = raf.getChannel();

        // Создаем буфер для чтениия данных.
        ByteBuffer buf = ByteBuffer.allocate(512);

        StringBuilder sb = new StringBuilder();

        // Читаем данные в буфер.
        while (ch.read(buf) != -1) {
            sb.append(new String(buf.array(), 0, buf.position()));

            // Очищаем буфер для следующего чтения.
            buf.clear();
        }

        System.out.println(">> Read from file " + file + ":");
        System.out.println(sb);
    }
}
```

FileChannel имеет очень удобные методы transferTo() и transferFrom(), позволяющие копировать содержимое другого канала.

Например, так выглядит копирование файла:

```
private static void copy_1(File src, File dst) throws IOException {
        try (RandomAccessFile rafSrc = new RandomAccessFile(src, "r");
             RandomAccessFile rafDst = new RandomAccessFile(dst, "rw")) {
             FileChannel srcCh = rafSrc.getChannel();
             FileChannel dstCh = rafDst.getChannel();

             srcCh.transferTo(0, rafSrc.length(), dstCh);
//             dstCh.transferFrom(srcCh, 0, rafSrc.length());
        }
    }
```

FileChannel имеет очень удобные методы transferTo() и transferFrom(), позволяющие копировать содержимое другого канала.

Например, так выглядит копирование файла:

```
private static void copy_1(File src, File dst) throws IOException {
        try (RandomAccessFile rafSrc = new RandomAccessFile(src, "r");
             RandomAccessFile rafDst = new RandomAccessFile(dst, "rw")) {
             FileChannel srcCh = rafSrc.getChannel();
             FileChannel dstCh = rafDst.getChannel();

             srcCh.transferTo(0, rafSrc.length(), dstCh);
//             dstCh.transferFrom(srcCh, 0, rafSrc.length());
        }
    }
```

По правде говоря, можно сделать проще:

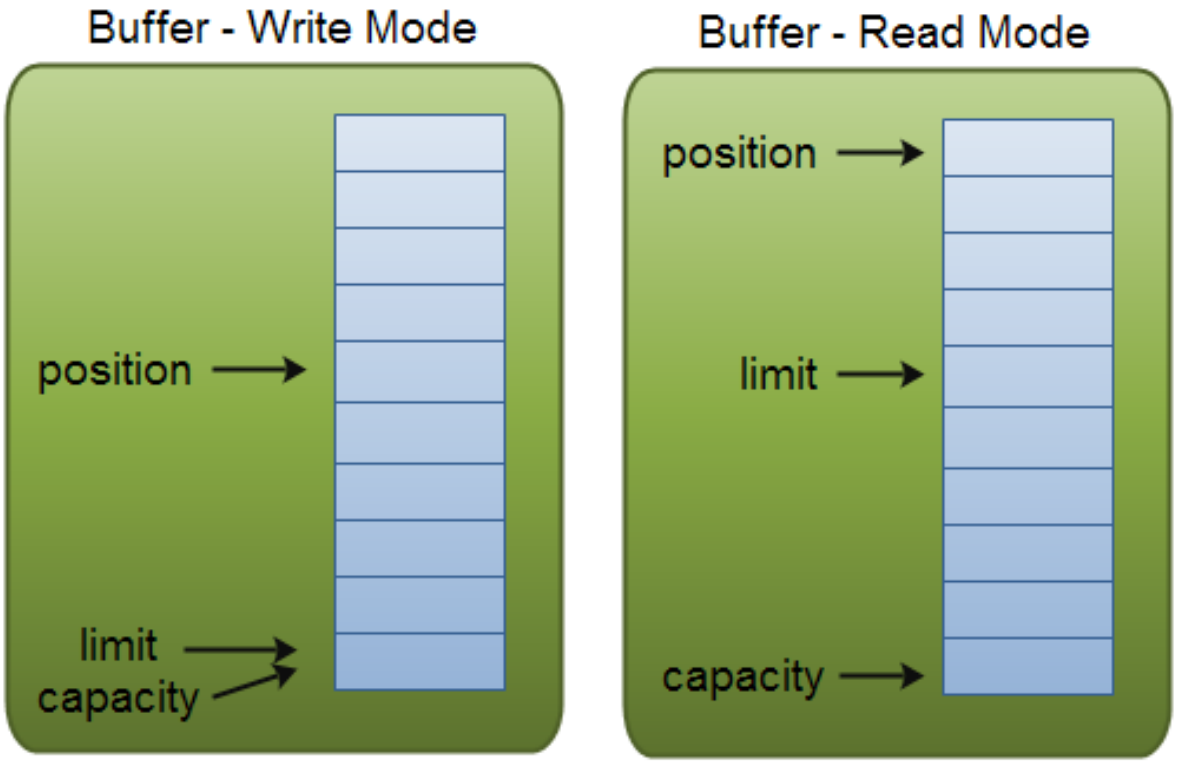
```
private static void copy_2(Path src, Path dst) throws IOException {
    Files.copy(src, dst);
}
```

NIO буфер значительно упрощает работу с данными в памяти, т. к. он автоматически следит за прочитанными и записанными данными. Для этого он имеет следующие параметры:

- capacity — емкость буфера, не изменяется;
- position — текущее положение курсора;
- limit — отметка указывающая предел для позиции.

Позиция изменяется равно как для записи, так и для чтения. Например, если было прочитано 2 байта, то позиция увеличится на 2, если записано, то так же увеличена на 2.

Фактически, позиция и лимит определяют тот участок памяти в буфере, который можно прочесть, либо, наоборот, заполнить чем-то.


Buffer - Write Mode
position
limit
capacity
Buffer - Read Mode
position
limit
capacity

Буфер имеет следующие методы:

- putxxx()/getxxx() - записывает/читает переданные данные и увеличивает позицию в зависимости от объема данных;
- flip() - подготавливает буфер для следующей записи или чтения, а именно: устанавливает лимит в текущую позицию, а позицию сбрасывает в 0;
- rewind() - перематывает буфер для повторного чтения - сбрасывает позицию в 0;
- clear() - ставит позицию в 0, а лимит равным емкости буфера;
- remaining() - показывает сколько осталось данных для чтения (или места для записи), а именно разницу между лимитом и позицией.

```
ByteBuffer buffer = ByteBuffer.allocate(16);

assert buffer.position() == 0;
assert buffer.capacity() == 16;
assert buffer.limit() == 16;
assert buffer.remaining() == 16;

// Увеличивает позицию на 4.
buffer.putInt(100);

assert buffer.position() == 4;
assert buffer.remaining() == 12;

// Увеличивает позицию на 8.
buffer.putDouble(100.25);

assert buffer.position() == 12;
assert buffer.remaining() == 4;

// Устанавливает лимит на место позиции,
// сбрасывает позицию в 0.
buffer.flip();

assert buffer.position() == 0;
assert buffer.limit() == 12;
assert buffer.remaining() == 12;
```

```
// Увеличивает позицию на 4.
int anInt = buffer.getInt();

assert buffer.position() == 4;
assert buffer.remaining() == 8;

// Увеличивает позицию на 8.
double aDouble = buffer.getDouble();

assert buffer.position() == 12;
assert buffer.remaining() == 0;

// Сбрасывает позицию в 0.
buffer.rewind();

assert buffer.position() == 0;
assert buffer.limit() == 12;
assert buffer.remaining() == 12;

// Увеличивает позицию на 4.
assert anInt == buffer.getInt();
// Увеличивает позицию на 8.
assert aDouble == buffer.getDouble();

// Сбрасывает позицию в 0, ставит лимит равный емкости буфера
buffer.clear();

assert buffer.position() == 0;
assert buffer.capacity() == 16;
assert buffer.limit() == 16;
assert buffer.remaining() == 16;
```

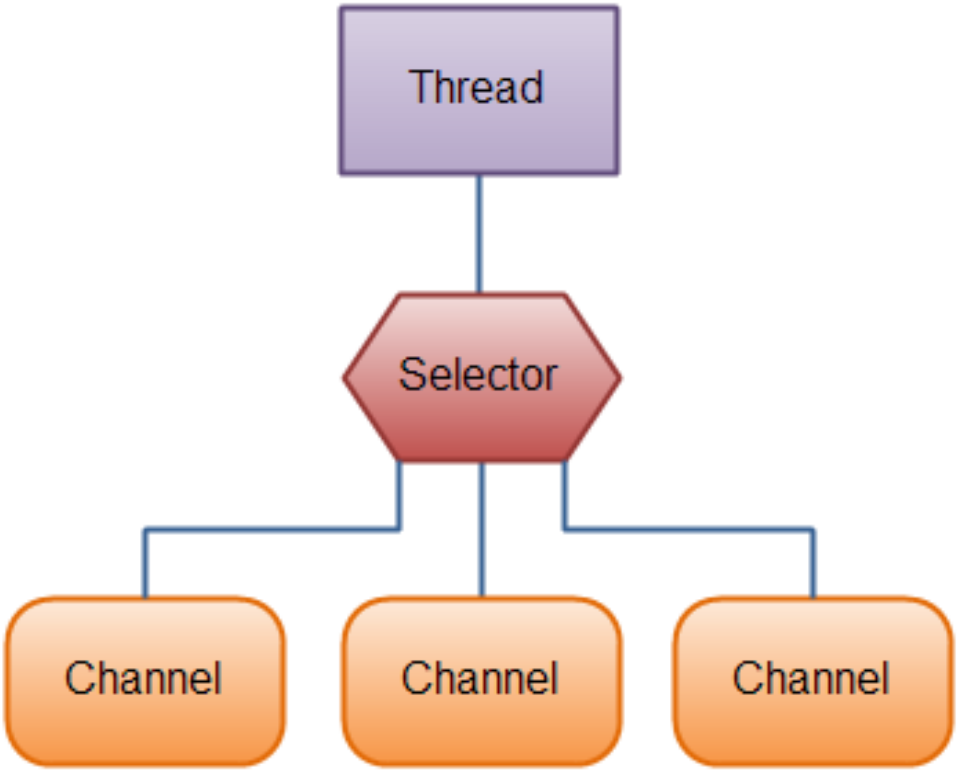
Типичная работа с буфером и каналами состоит из следующих шагов:
1) Запись данных в буфер.
2) flip().
3) Чтение данных из буфера.
4) clear() или compact().

Java NIO предоставляет следующие реализации буфера:
- ByteBuffer;
- MappedByteBuffer;
- CharBuffer;
- DoubleBuffer;
- FloatBuffer;
- IntBuffer;
- LongBuffer;
- ShortBuffer.

Наиболее востребованной и важной функцией NIO является возможность работать в неблокирующем режиме, а именно, обслуживать множество подключений из одного потока исполнения.

Такой подход можно называть событийным, т. к. поток реагирует на некоторые события ввода/вывода и выполняет некоторую работу, а в остальное время просто спит.

Для утилизации такой возможности используется селектор (Selector).



Работа селектора заключается в том, что он по некоторому событию (например, входящее соединение или полученные данные) будит обслуживающий поток, который в свою очередь выполняет нужную операцию.

Создается селектор просто:

```
Selector selector = Selector.open();
```

После чего необходимо зарегистрировать в нем канал и указать на какое событие его выбирать:

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

Канал должен быть переведен в неблокирующий режим. В данном случае оформлена подписка на опреацию чтения. Что означает, когда в системном сетевом буфере будут доступны данные, обслуживающий поток будет оповещен.

SelectionKey представляет канал в селекторе.

Селектор поддерживает следующие события:

- Connect — установлено исходящее подключение;
- Accept — установлено входящее подключение;
- Read — канал имеет данные для чтения;
- Write — канал готов к отправке данных.

Этим событиям соответствуют константы:

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

```
Selector selector = Selector.open();

channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);

while(true) {
    int readyChannels = selector.select();

    if(readyChannels == 0) continue;

    Set<SelectionKey> selectedKeys = selector.selectedKeys();

    Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

    while(keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();

        if(key.isAcceptable()) {
            // Было принято входящее подключение в ServerSocketChannel.

        } else if (key.isConnectable()) {
            // Установлено исходящее подключение.

        } else if (key.isReadable()) {
            // Канал готов к чтению.

        } else if (key.isWritable()) {
            // канал готов к записи
        }

        keyIterator.remove();
    }
}
```