**Разбираемся с java.nio.\***

Этот пакет был добавлен еще в Java 1.4, однако многие разработчики о нем либо не знают, либо не умеют пользоваться. В сети мало материалов на эту тему, особенно на русском.

**Введение**

Java New IO — «новая» реализация IO. Ее предназначение — решить проблемы производительности стандартного блокирующего IO. Почти все методы чтения-записи без блокировок, они читают или записывают **лишь уже доступную информацию**. Это позволяет в одном или нескольких потоках обрабатывать любое количество подключений.

Продолжение под хабракатом

**Buffers**

Более функциональная и удобная замена массивов. Используется для хранения считанной информации и в качестве источника для записи. Всего есть несколько типов буферов:

* **ByteBuffer** — хранит **байты**. Может быть представлен в виде других буферов.
* **ShortBuffer** — хранит **short**‘ы. Может быть представлен в виде **ByteBuffer**‘а
* **IntBuffer** — хранит **int**‘ы. Может быть представлен в виде **ByteBuffer**‘а
* **LongBuffer** — хранит **long**‘и. Может быть представлен в виде **ByteBuffer**‘а
* **FloatBuffer** — хранит **float**‘ы. Может быть представлен в виде **ByteBuffer**‘а
* **CharBuffer** — хранит **char**‘ы. Может быть представлен в виде **ByteBuffer**‘а

Помимо этого, можно создать **ReadOnlyBuffer** методом **asReadOnly()**; Каждый буфер имеет размер (**capacity**), лимит (**limit**), текущую позицию (**position**) и метку (**mark**):

* **размер** — сколько данных в себя физически может вместить буфер. Устанавливается при создании
* **лимит** — до какой позиции можно читать или записывать данные в буфер. Можно установить вручную
* **позиция** — сколько байт уже записано/прочитано. Можно установить вручную, по умолчанию равен нулю
* **метка** — сохраненная позиция, позволяет вернуться к нужному месту в буфере

Кроме этого, у каждого буфера есть пара методов, которые позволяют им управлять:

* **get(index)** — возвращает *элемент* на указанной позиции
* **put(index, type)** — устанавливает *элемент* на указанную позицию
* **get()** — возвращает *элемент* на текущей позиции, затем повышает позицию на 1
* **put(type)** — устанавливает *элемент type* на текущую позицию, затем повышает позицию на 1
* **clear()** — ставит позицию на 0, лимит на размер и удаляет метку. *Подготавливает буфер для записи*
* **flip()** — ставит лимит равным позиции, затем позицию на 0 и удаляет метку. *Подготавливает буфер для чтения*
* **rewind()** — ставит позицию на 0 и удаляет метку. Используется для того чтобы заново прочесть буфер
* **position(int), position()** — установка и получение позиции соответственно
* **limit(int), limit()** — установка и получение лимита соответственно
* **remaing()** — возвращает сколько еще элементов можно прочитать или записать
* **mark()** — устанавливает метку на текущую позицию
* **reset()** — возвращает позицию к метке

Создать буфер можно тоже разным способами:

* **(Type)Buffer.allocate(capacity)** — создает буфер в Heap. Можно преобразовать в массив с помощью метода **array()**
* **ByteBuffer.allocateDirect(capacity\*typesize).asType()** — создает буфер в системной памяти. Нельзя преобразовать в массив.

**Channels**

Заместо Stream’ов, в NIO используются каналы (**Channel**), которые могут объеденять функциональность InputStream и OutputStream.  
Сам по себе **Channel** имеет только методы **close()** и **isOpen()**. Остальные методы добавляются реализуемыми им интерфейсами:

* **ReadableChannel** — возможность чтения содержимого из канала в **ByteBuffer** (**channel.read(dst)**)
* **WriteableChannel** — возможность записи содержимого в канал из **ByteBuffer**(**channel.write(src)**)
* **SelectableChannel** — возможность использовать **Selector** и отключить блокировки (об этом ниже)
* **AsynchronousChannel** — возможность читать и записывать из нескольких потоков

Для удобного управления **SelectableChannel** есть специальный класс — **Selector**. Его можно использовать только после того, как Вы отключили блокировки (**channel.configureBlocking(false)**);

**Selectors и SelectionKeys**

**Selector** — своеобразный слушатель, который сообщает, когда с каналом можно совершить какое-то действие. Без него не получится сделать нормальное NIO приложение. Для начала его надо создать. **Selector** создается с статического метода **Selector.open()**. После создания селектора, необходимо его зарегестрировать на нужном канале. Это делается с помощью метода:

SelectionKey key = channel.register(selector, ops, [attach])

**Selection Op** определяет, какие события необходимо отслеживать:

* **SelectionKey.OP\_READ** — если в канале есть данные, доступные для чтения
* **SelectionKey.OP\_WRITE** — если канал доступен для записи.  
  **Внимание!** Ставьте этот op только если есть данные, доступные для записи.
* **SelectionKey.OP\_ACCEPT** — только для **ServerSocketChannel**. Если есть непринятые подключения
* **SelectionKey.OP\_CONNECT** — только для **\*SocketChannel**. Если подключение успешно закончилось

OP’ы можно объеденять с помощью **логического ИЛИ**:

int ops = SelectionKey.OP\_ACCEPT | SelectionKey.OP\_READ;

**SelectionKey** — объект, который провоцирует событие. Имеет несколько полезных методов:

* **attach(Object)** — добавляет «прикрепление» к ключу. Например, обработчик
* **attachment()** — возвращает ранее добавленное прикрепление
* **channel()** — возвращает канал, к которому прикреплен ключ
* **cancel()** — убирает ключ из селектора

Ниже приведен пример.

Чтобы обрабатывать каналы с **Selector**‘ом, необходимо сделать цикл, который работает до закрытия канала. Пример приведен ниже:

while(!serverKey.isCancelled())

{selector.select(); // Ждем до того, как появится хотя бы одно событие. Как появятся, выбираем ключи с этими событиями

Iterator<SelectionKey> iterator = selector.selectedKeys().iterator(); // Получаем итератор выбранных ключей

while(iterator.hasNext())

{

SelectionKey key = iterator.next();

SelectableChannel channel = key.channel();

if(key.isAcceptable())

{// принимаем подключение у сервера. Тут же его регистрируем в селекторе с OP\_READ.}

if(key.isReadable())

{// читаем данные, если длина -1, удаляем ключ с помощью key.cancel();}

if(key.isWriteable())

{// записываем данные}

iterator.remove(); // Удаляем ключ из выбранных, так как мы его обработали

}

}

selector.close();