**17.6 Java NIO API**

Что вы узнаете

В этом материале вы узнаете, что такое Java NIO и чем он отличается от Java IO. Разберёте, какие инструменты для работы с потоком ввода-вывода содержатся в нём и как их можно использовать.

Что такое Java NIO

**Java NIO**— это API (Application Programming Interface) ввода-вывода, включённый в Java 1.4 и предоставляющий более гибкие и производительные средства для обработки данных, чем API Java IO.

Java NIO предлагает асинхронный ввод-вывод, в котором поток не блокируется, ожидая завершения операции. Вместо этого поток может продолжать работу и выполнять другие задачи. Это особенно полезно в приложениях, обрабатывающих множество соединений одновременно.

Основные компоненты Java NIO:

* **Buffer** (буфер). Используется для хранения данных и обмена ими между буфером и каналом.
* **Charset** (чарсет). Используется для отображения необработанных байтов в читаемые символы и наоборот.
* **Channel** (канал). Представляет собой открытое соединение между источником данных и целью, с которыми выполняются операции ввода-вывода.
* **Selector** (селектор). Используется для мониторинга каналов и определения, готовы ли они к чтению или записи.

Сравнение Java IO и NIO

Java IO и NIO — это два разных подхода к работе с вводом-выводом данных.

|  |  |
| --- | --- |
| **Java IO** | **Java NIO** |
| Блокирующая модель ввода-вывода: поток ожидает завершения операции, прежде чем продолжить выполнение кода | Неблокирующая модель ввода-вывода: поток может продолжать работу, не дожидаясь завершения операции |
| Потоко-ориентированный: данные считываются и записываются в поток байтов | Буферо-ориентированный: данные считываются и записываются в буфер |
| Имеет только одно соединение на один поток | Может обрабатывать несколько соединений на поток |
| Базируется на потоке: передаёт данные в одну сторону (one-directional) | Базируется на канале:  передаёт данные в обе стороны (bi-directional) |
| Использует Heap — основной вариант работы с памятью в JVM | Использует ресурсы вне JVM, что позволяет снизить накладные расходы памяти |
| Подходит для простой синхронной обработки данных, например для парсинга XML-документов небольшого объёма в одном потоке | Подходит для обработки больших объёмов данных с использованием множества сетевых соединений одновременно, например для многопоточной асинхронной обработки файлов |
| **Итог:**  + простота использования, + надёжность, - производительность. | **Итог:**  + производительность, + гибкость, - сложности в настройке и оптимизации. |

У каждого API свои преимущества и недостатки. Важно иметь о них представление, чтобы понимать, какой API в каких ситуациях использовать.

Буферы

Рассмотрим базовый класс Buffer. Он содержит индексы, отвечающие за запись состояния хранящихся в нём данных:

* **capacity**— максимальное количество элементов, содержащихся в буфере (неизменяемое значение, задаётся при инициализации);
* **limit**— индекс, до которого производится чтение или запись данных (по умолчанию равен capacity);
* **position**— индекс, с которого начинается чтение или запись данных (по умолчанию равен 0);
* **mark**— отмеченный индекс (по умолчанию равен −1).

Взаимоотношения индексов можно выразить через следующую зависимость:

**0 ≤ mark ≤ position ≤ limit ≤ capacity**.

Изображение выглядит как снимок экрана, Прямоугольник, дизайн

Автоматически созданное описание

Здесь и далее — изображения Skillbox

Работа с индексами

Метод mark() позволяет запоминать текущее положение индекса position и возвращать его в положение индекса mark.

Метод reset() позволяет возвращать индекс position на место индекса mark. Вызов этого метода не меняет и не сбрасывает значение mark.

**Важно:** reset() выбросит исключение, если mark не определён (−1).

ByteBuffer buff = ByteBuffer.allocate(32); //выделяем буфер размером 32 байта, mark = -1, position = 0  
buff.position(10); // mark = -1, position = 10  
buff.mark(); // mark = 10, position = 10  
buff.position(20); // mark = 10, position = 20  
buff.reset(); // mark = 10, position = 10

Метод clear() меняет индексы limit = capacity, position = 0, mark = −1. Применяется для повторного использования буфера, когда нужно сбросить индексы на начальные позиции.

buff.limit(25); // mark = 10, position = 10, limit = 25  
buff.clear(); // mark = -1, position = 0, limit = 32

Метод flip() меняет индексы limit = position, position = 0, mark = −1. Применяется для переключения из режима записи в режим чтения.

**Важно:** если вызвать flip() два раза подряд, установится limit = 0 и данные прочитать не получится.

buff.limit(25); // mark = 10, position = 10, limit = 25  
buff.flip(); // mark = -1, position = 0, limit = 10

Метод rewind() меняет индексы position = 0, mark = −1. Применяется для повторного чтения данных.

buff.limit(25); // mark = 10, position = 10, limit = 25  
buff.rewind(); // mark = -1, position = 0, limit = 25

Метод compact() принадлежит классу ByteBuffer и меняет индексы position = limit − position, limit = capacity, mark = −1. Применяется для частичной перезаписи буфера. Например, после частичного чтения данных нужно записать в буфер данные, compact() скопирует непрочитанные данные в начало буфера и подготовит его для записи.

buff.limit(25); // mark = 10, position = 10, limit = 25  
buff.compact(); // mark = -1, position = 15, limit = 32

Методы hasRemaining() и remaining() определяют, осталось ли место между индексами limit и position.

ByteBuffer buff = ByteBuffer.allocate(32); // mark = -1, position = 0  
buff.position(10); // mark = -1, position = 10  
buff.limit(15); // mark = -1, position = 10, limit = 15  
boolean isRemaining = buff.hasRemaining(); // true  
int remainingSize = buff.remaining(); // 5

Передача данных

Рассмотрим способы передачи на примере того же универсального подкласса ByteBuffer, который может передавать данные разных типов (числа и символы).

Данные для записи и чтения могут передаваться как по одному байту, так и в виде массива.

ByteBuffer buff = ByteBuffer.allocate(16);  
  
//Запись в буфер  
buff.put((byte) 'H'); // position = 1, одиночная запись  
// буфер содержит [H]  
buff.put("ello world".getBytes()); // position = 11, массив байт  
// буфер содержит [Hello world]  
buff.put(6, "Skillbox".getBytes()); // position = 11, массив байт по конкретному индексу, позиция не меняется!  
// буфер содержит [Hello Skillbox]  
  
// Чтение из буфера  
buff.flip(); // position = 0, limit = 11  
char h = (char) buff.get(); // position = 1, char = 'H'  
char e = (char) buff.get(); // position = 2, char = 'e'  
char s = (char) buff.get(6); // position = 2, char = 'S', чтение по конкретному индексу, позиция не меняется!  
char k = (char) buff.get(7); // position = 2, char = 'k', чтение по конкретному индексу, позиция не меняется!  
  
byte[] hello = new byte[5];  
buff.get(0 , hello); // чтение с индекса 0 массива байт и запись в массив hello, заполнится массив hello данными с 0 индекса по 4 включительно  
String result = new String(hello, StandardCharsets.UTF\_8);  
System.out.println(result); // Hello  
  
char x = (char) buff.get(12); // IndexOutOfBoundsException, обращаемся к существующему символу 'o', но за пределами limit, равного 11

**Важно:** чтение и запись с индекса put(int index, byte b), get(int index) не изменяет position. Минусы этого — можно обратиться за пределы limit и сложно отследить position. Плюсы — возможно гибкое управление буфером.

Помимо чтения и записи байтовых данных, ByteBuffer поддерживает обработку примитивных типов: byte, char, short, int, long, float, double.

ByteBuffer view

ByteBuffer view — способ создания представления (view) данных в ByteBuffer в другой форме (тип данных) или с другой интерпретацией (только чтение, копия, остаток).

Это позволяет использовать **одни и те же данные в разных контекстах**, не создавая новые объекты, и копировать данные между существующими объектами, что помогает ускорить работу и снизить использование памяти.

Как и в предыдущих примерах, создадим буфер.

ByteBuffer buff = ByteBuffer.allocate(16); // mark = -1, position = 0, limit = 16, capacity = 16  
buff.put("He".getBytes()); // mark = -1, position = 2, limit = 16, capacity = 16  
buff.mark(); // mark = 2, position = 2, limit = 16, capacity = 16  
buff.put("llo".getBytes()); // mark = 2, position = 5, limit = 16, capacity = 16  
buff.limit(8); // mark = 2, position = 5, limit = 8, capacity = 16

Метод duplicate() создаст новый экземпляр ByteBuffer, который будет ссылаться на ту же область памяти, но иметь свои индексы, скопированные с родительского экземпляра. Изменения данных будут отражаться сразу в двух экземплярах.

ByteBuffer duplicateView = buff.duplicate();// mark = 2, position = 5, limit = 8, capacity = 16  
duplicateView.position(0);// mark = 2, position = 0, limit = 8, capacity = 16  
duplicateView.put("Good day".getBytes());// mark = 2, position = 8, limit = 8, capacity = 16  
System.out.println(new String(buff.array(), StandardCharsets.UTF\_8));// Good day

Метод slice() создаст новый экземпляр ByteBuffer начиная с позиции, указанной в родительском экземпляре, сбросит индексы mark и position, а также установит limit и capacity равными размеру нового буфера.

duplicateView.limit(16).put(" fellow".getBytes());// mark = 2, position = 15, limit = 16, capacity = 16  
duplicateView.position(9).limit(15);// mark = 2, position = 9, limit = 15, capacity = 16  
ByteBuffer slicedView = duplicateView.slice();// mark = -1, position = 0, limit = 7, capacity = 7  
byte [] slicedByteArr = new byte[slicedView.capacity()];// создаём массив байт размером с ёмкостью slicedView  
slicedView.get(slicedByteArr);// передаём данные начиная с позиции 0  
System.out.println(new String(slicedByteArr, StandardCharsets.UTF\_8));// fellow

**Обратите внимание**: записывать методы работы с буфером можно и в строку.

Существует также метод asReadOnlyBuffer(). Он отработает аналогично методу duplicate(), с той лишь разницей, что при попытке изменить дочерний буфер получается исключение ReadOnlyBufferException. То есть asReadOnlyBuffer() создаёт буфер только для чтения.

Методы преобразования данных работают аналогично slice(), однако количество оставшихся (remaining) элементов будет рассчитываться по-разному в зависимости от байтового размера типа данных. Например, при переводе в тип long количество оставшихся элементов будет делиться на 8, потому что long занимает 8 байт.

Сравнение буферов

Мы рассмотрели большинство вспомогательных методов. Поговорим о способах сравнения буферов compareTo() и equals().

Эти методы производят сравнение, ограничиваясь диапазоном между индексами position и limit.

byte[] bytes1 = "Skillbox".getBytes();  
byte[] bytes2 = "HelloSkillbox".getBytes();  
  
ByteBuffer buffer1 = ByteBuffer.wrap(bytes1);  
ByteBuffer buffer2 = ByteBuffer.wrap(bytes2);  
buffer2.position(5);  
boolean equal = buffer1.equals(buffer2); *// true*  
*int result = buffer1.compareTo(buffer2); // 0*

Буферы можно представить таким образом:

buffer1          buffer2

 S ← position     H

k                          e

i                           l

l                           l

l                           o

b                          S ← position

o                          k

x ← limit             i

                            l

                            l

                            b

                            o

                            x ← limit

Сравниваться будут байты от S до x в каждом буфере, если position или limit изменится в одном из буферов, equals вернёт false и compateTo не будет равен нулю.

Самостоятельная работа

Прежде чем продолжить изучение Java NIO, стоит самостоятельно попрактиковаться работать с индексами и ByteBuffer.

Запишите, какие значения будут принимать индексы в процессе исполнения кода и какая фраза выводится в консоль.

ByteBuffer buff = ByteBuffer.allocate(32); // mark = \*, position = \*, limit = \*, capacity = \*  
buff.mark(); // mark = \*, position = \*, limit = \*, capacity = \*  
buff.put("Byte".getBytes());// mark = \*, position = \*, limit = \*, capacity = \*  
buff.reset(); // mark = \*, position = \*, limit = \*, capacity = \*  
buff.put("Buffer are part of NIO".getBytes()); // mark = \*, position = \*, limit = \*, capacity = \*  
buff.limit(30); // mark = \*, position = \*, limit = \*, capacity = \*  
buff.flip(); // mark = \*, position = \*, limit = \*, capacity = \*  
  
byte[] result = new byte[buff.limit()];  
buff.get(result); // mark = \*, position = \*, limit = \*, capacity = \*  
System.out.println(new String(result, StandardCharsets.UTF\_8)); // \*\*\*

Проверьте себя

Надеемся, что у вас всё получилось. Если что-то не вышло, ничего страшного: сравните свой результат с решением и проанализируйте его.

ByteBuffer buff = ByteBuffer.allocate(32); // mark = -1, position = 0, limit = 32, capacity = 32  
buff.mark(); // mark = 0, position = 0, limit = 32, capacity = 32  
buff.put("Byte".getBytes());// mark = 0, position = 4, limit = 32, capacity = 32  
buff.reset(); // mark = 0, position = 0, limit = 32, capacity = 32  
buff.put("Buffer are part of NIO".getBytes()); // mark = 0, position = 22, limit = 32, capacity = 32  
buff.limit(30); // mark = 0, position = 22, limit = 30, capacity = 32  
buff.flip();// mark = -1, position = 0, limit = 22, capacity = 32  
  
byte[] result = new byte[buff.limit()];  
buff.get(result);// mark = -1, position = 22, limit = 22, capacity = 32  
System.out.println(new String(result, StandardCharsets.UTF\_8));// Buffer are part of NIO

Charset

Класс **Charset** — это инструмент сопоставления последовательности байтов с конкретной кодировкой текста, определённой в стандартах [IANA](https://www.ietf.org/rfc/rfc2278.txt) (Internet Assigned Numbers Authority). Его можно использовать для кодирования текста в байтовые последовательности и декодирования. Некоторые кодировки, которые можно использовать с помощью класса Charset, включают UTF-8, ISO-8859-1 и Windows-1251.

Например, чтобы закодировать строку в UTF-8 при помощи класса Charset, можно использовать следующий код:

String text = "Hello Skillbox";  
Charset winCharset = Charset.forName("windows-1251");  
ByteBuffer byteBuffer = utf8Charset.encode(text);

А затем успешно декодировать его из байтовой последовательности в строку:

CharBuffer charBuffer = winCharset.decode(byteBuffer);  
String textFromChar = charBuffer.toString();  
System.out.println(textFromChar);*// Hello Skillbox*

Классы CharsetEncoder и CharsetDecoder используют и для преобразования текста между кодировками. Также в них есть методы для проверки возможности декодирования, обработки ошибок и проверки результата преобразования символов в байты и наоборот.

Применение классов из пакета java.nio.charset может пригодиться, если нужно оперативно перевести из одной кодировки в другую большой объём текста, например, полученного по сети.

Каналы и селекторы

**Channel** — это базовый интерфейс, который предоставляет API для чтения данных из разных источников и записи в них. Например, так можно работать с файлами, сокетами, буферами и прочим. В Java Channel используется неблокирующий ввод-вывод, который позволяет приложению продолжать работу, пока операция ввода-вывода не завершена.

Channel можно использовать для решения задач, связанных с вводом-выводом данных, например:

* чтение и запись файлов: Channel позволяет работать с файлами эффективнее традиционных потоков ввода-вывода;
* работа с сокетами: Channel можно использовать для чтения данных из сокетов и записи в них, что помогает реализовать сетевые приложения;
* работа с буфером: Channel позволяет работать с буферами данных, что повышает эффективность операций ввода/вывода.

При работе с интерфейсами и классами, имплементирующими или наследующими интерфейс AutoСlosable, можно использовать конструкцию try-with-resources. Она предоставляет удобный способ автоматического закрытия ресурсов, открытых в блоке try, после завершения работы с ними или при возникновении исключения в блоке try. При необходимости вызова нескольких ресурсов их можно перечислять, разделяя точкой с запятой.

Чтобы воспроизвести работу примеров на своём ПК, воспользуйтесь [этим репозиторием](https://github.com/skillbox-java/nio_example).

Реализуем класс FileCopySimpleExample — простой пример функции, копирующей объёмный файл из входящего канала в исходящий при помощи ByteBuffer.

Класс FileCopySimpleExample:

* определяет путь до файлов;
* открывает каналы для чтения и записи;
* создаёт буфер объёмом в 1 024 байта;
* записывает данные в целевой FileChannel.

import java.io.IOException;  
import java.nio.ByteBuffer;  
import java.nio.channels.FileChannel;  
import java.nio.file.Path;  
import java.nio.file.StandardOpenOption;  
  
public class FileCopySimpleExample {  
   public static void main(String[] args) {  
       Path inputPath = Path.of("src/file\_copy\_example  
/input.txt");  
       Path outputPath = Path.of("src/file\_copy\_example  
/output.txt");  
       try (FileChannel inputChannel = FileChannel.open(inputPath, StandardOpenOption.READ);  
            FileChannel outputChannel = FileChannel.open(outputPath, StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {  
           ByteBuffer buffer = ByteBuffer.allocate(1024);  
           while (inputChannel.read(buffer) != -1) {  
               buffer.flip();  
               outputChannel.write(buffer);  
               buffer.clear();  
           }  
       } catch (IOException e) {  
           e.printStackTrace();  
       }  
   }  
}

Так мы быстро сделали копию объёмного файла, используя исключительно инструменты из пакета java.nio.

Передача данных не ограничивается локальной системой, так как NIO API позволяет передавать информацию через сетевое подключение. В качестве примера реализуем клиент-серверное соединение, используя имеющиеся базовые инструменты.

Проверьте, свободен ли порт 8088 на вашем ПК. Для этого запустите следующую команду:

* в терминале Windows 10 — **netstat -ano | findstr :8088**;
* в терминале Mac — **lsof -i :8088**;
* в терминале Linux — **netstat -tln | grep :8088**.

Если результат запроса пустой, значит, порт не занят. Если порт занят, найдите другой, добавляя единицу, пока не увидите свободный. Например, можно попробовать 8089, 8090 и так далее.

Создадим два класса.

Серверный (NetworkServerExample) для приёма сообщения:

* создаёт серверный сокет-канал и привязывает его к порту 8088;
* ожидает подключение клиента;
* принимает сообщение от клиента;
* закрывает сокет-каналы.

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.ServerSocketChannel;  
import java.nio.channels.SocketChannel;  
  
public class NetworkServerExample {  
   *// если ваш порт занят, укажите свободный порт*  
   private static final int PORT = 8088;  
   public static void main(String[] args) throws IOException {  
       ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
      serverSocketChannel.socket().bind(new InetSocketAddress(PORT));  
      System.out.println("Server started on port " + PORT);  
  
       SocketChannel socketChannel = serverSocketChannel.accept();  
       System.out.println("Client connected: " + socketChannel.getRemoteAddress());  
  
       ByteBuffer buffer = ByteBuffer.allocate(1024);  
       int bytesRead = socketChannel.read(buffer);  
       String message = new String(buffer.array(), 0, bytesRead);  
       System.out.println("Received message from client: " + message);  
  
       socketChannel.close();  
       serverSocketChannel.close();  
   }  
}

Клиентский (NetworkClientExample) для отправки сообщения:

* создаёт клиентский сокет-канал и подключается к серверу по адресу и порту;
* отправляет сообщение на сервер;
* закрывает сокет-канал.

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
  
public class NetworkClientExample {  
   private static final String HOSTNAME = "localhost";  
   private static final int PORT = 8088;  
   public static void main(String[] args) throws IOException {  
       SocketChannel socketChannel = SocketChannel.open();  
       socketChannel.connect(new InetSocketAddress(HOSTNAME, PORT));  
       System.out.println("Connected to server");  
  
       String message = "Hello, server!";  
       ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());  
       socketChannel.write(buffer);  
       System.out.println("Sent message to server: " + message);  
       socketChannel.close();  
   }  
}

Первым запустите серверный класс и дождитесь сообщения: Server started on port 8088. Затем запустите клиентский.

Вывод программы будет следующим:

* NetworkClientExample   
  Connected to server  
  Sent message to server: Hello, server!
* NetworkServerExample  
  Server started on port 8088  
  Client connected: /127.0.0.1:54763  
  Received message from client: Hello, server!

Так мы можем передавать данные по сети, используя только имеющийся функционал Java NIO.

**В завершение изучения данного блока затронем тему селекторов.**

**Селекторы** используются для одновременного обслуживания нескольких каналов ввода-вывода в одном потоке. Это позволяет обрабатывать несколько соединений в одном потоке и может быть полезно для обработки большого количества соединений или повышения производительности ввода-вывода.

Рассмотрим пример с сервером и тремя клиентами.

Создадим класс, имитирующий работу сервера SelectorServerExample:

* создаёт селектор;
* создаёт два серверных канала и регистрирует их в селекторе;
* блокирует работу до поднятия каналов;
* получает ключи для каналов;
* проверяет подключение от клиентов;
* считывает данные из канала в буфер и выводит их в консоль.

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SelectionKey;  
import java.nio.channels.Selector;  
import java.nio.channels.ServerSocketChannel;  
import java.nio.channels.SocketChannel;  
import java.util.Iterator;  
import java.util.Set;  
  
public class SelectorServerExample {  
  
   private static final int PORT1 = 8088;  
   private static final int PORT2 = 8089;  
  
   public static void main(String[] args) throws IOException {  
  
       Selector selector = Selector.open();  
  
       ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
       serverSocketChannel.socket().bind(new InetSocketAddress("localhost", PORT1));  
       serverSocketChannel.configureBlocking(false);  
       serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  
  
       ServerSocketChannel serverSocketChannel2 = ServerSocketChannel.open();  
       serverSocketChannel2.socket().bind(new InetSocketAddress("localhost", PORT2));  
       serverSocketChannel2.configureBlocking(false);  
       serverSocketChannel2.register(selector, SelectionKey.OP\_ACCEPT);  
  
       while (true) {  
           selector.select();  
  
           Set<SelectionKey> selectedKeys = selector.selectedKeys();  
           Iterator<SelectionKey> keyIterator = selectedKeys.iterator();  
  
           while (keyIterator.hasNext()) {  
  
               SelectionKey key = keyIterator.next();  
  
               if (key.isAcceptable()) {  
                   ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();  
                   SocketChannel clientChannel = serverChannel.accept();  
                   clientChannel.configureBlocking(false);  
                   clientChannel.register(selector, SelectionKey.OP\_READ);  
                   System.out.println("Новое подключение: " + clientChannel);  
               } else if (key.isReadable()) {  
                   SocketChannel channel = (SocketChannel) key.channel();  
                   ByteBuffer buffer = ByteBuffer.allocate(1024);  
                   channel.read(buffer);  
                   String message = new String(buffer.array()).trim();  
                   System.out.println("Получено сообщение от " + channel + ": " + message);  
               }  
               keyIterator.remove();  
           }  
       }  
   }  
}

Реализуем и запустим после запуска сервера три клиентских класса SelectorClientExample:

* создают канал соединения с сервером;
* считывают пользовательский ввод;
* закрывают канал;
* два клиента обращаются по одному порту.

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
import java.util.Scanner;  
public class SelectorClient1Example {  
   public static void main(String[] args) throws IOException {  
       SocketChannel clientChannel = SocketChannel.open(new InetSocketAddress("localhost", 8088));  
       Scanner scanner = new Scanner(System.in);  
       while (true) {  
           System.out.print("Введите сообщение для отправки на сервер: ");  
           String message = scanner.nextLine();  
           if ("exit".equals(message)) {  
               break;  
           }  
           ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());  
           clientChannel.write(buffer);  
       }  
       clientChannel.close();  
   }  
}

Второй клиентский класс:

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
import java.util.Scanner;  
public class SelectorClient2Example {  
   public static void main(String[] args) throws IOException {  
       SocketChannel clientChannel = SocketChannel.open(new InetSocketAddress("localhost", 8088));  
       Scanner scanner = new Scanner(System.in);  
       while (true) {  
           System.out.print("Введите сообщение для отправки на сервер: ");  
           String message = scanner.nextLine();  
           if ("exit".equals(message)) {  
               break;  
           }  
           ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());  
           clientChannel.write(buffer);  
       }  
       clientChannel.close();  
   }  
}

Третий клиентский класс:

import java.io.IOException;  
import java.net.InetSocketAddress;  
import java.nio.ByteBuffer;  
import java.nio.channels.SocketChannel;  
import java.util.Scanner;  
public class SelectorClient3Example {  
   public static void main(String[] args) throws IOException {  
       SocketChannel clientChannel = SocketChannel.open(new InetSocketAddress("localhost", 8089));  
       Scanner scanner = new Scanner(System.in);  
       while (true) {  
           System.out.print("Введите сообщение для отправки на сервер: ");  
           String message = scanner.nextLine();  
           if ("exit".equals(message)) {  
               break;  
           }  
           ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());  
           clientChannel.write(buffer);  
       }  
       clientChannel.close();  
   }  
}

Вывод серверного класса выглядит так:

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание

Посмотрим на схему работы программы:

Изображение выглядит как текст, диаграмма, снимок экрана, План

Автоматически созданное описание

С помощью селектора можно регистрировать каналы, которые могут быть готовы к операциям (OP) чтения (OP\_READ), записи (OP\_WRITE), подключения (OP\_CONNECT) или принятия нового соединения (OP\_ACCEPT). Селектор также позволяет определять, какие каналы готовы к обработке ввода-вывода, и в соответствии с этим обрабатывать их. Это сильно упрощает код для работы с несколькими каналами ввода-вывода и устраняет необходимость использования нескольких потоков для обработки каждого соединения.

Селекторы также могут использоваться для реализации асинхронного ввода-вывода, когда приложение может продолжать работу, в то время как данные считываются или записываются в фоновом режиме. Это может улучшить производительность и отзывчивость приложения.

Выводы

В этом материале вы познакомились с Java NIO API, изучили его основные компоненты и рассмотрели примеры их использования. Знание механизмов работы с потоками ввода-вывода помогут вам создавать гибкие и высокопроизводительные приложения.

Пример кода из статьи вы найдёте на [GitHub](https://github.com/skillbox-java/nio_example" \t "_blank).