Введение в Java NIO Selector

**Главное помнить, что на сервере уровень операций класса ServerSocket/ServerSocketChannel заканчивается тогда когда сервер приняв соединение от клиента и получив при этом объект Socket/SocketChannel создаст на сервере такого-же клиента (*как-бы объединяя два компьютера в один, просто “типо” с двумя дисками C и D*). И вся дальнейшая работа по чтению или записи информации с сервера и на сервер идет в режиме такого-же клиента Socket/SocketChannel.**

**1. Обзор**

В этой статье мы рассмотрим вводные части компонента Java NIO *Selector*.

Селектор предоставляет механизм для мониторинга одного или нескольких каналов NIO и определения, когда один или несколько каналов становятся доступными для передачи данных.

Таким образом, **один поток может использоваться для управления несколькими каналами**и, следовательно, несколькими сетевыми подключениями.

**2. Зачем использовать селектор?**

С помощью селектора мы можем использовать один поток вместо нескольких для управления несколькими каналами. **Переключение контекста между потоками является дорогостоящим для операционной системы**, и, кроме того, **каждый поток занимает память.**

Поэтому чем меньше ниток мы используем, тем лучше. Однако важно помнить, что **современные операционные системы и процессоры становятся лучше в многозадачном режиме**, поэтому издержки многопоточности со временем уменьшаются.

Здесь мы будем иметь дело с тем, как мы можем обрабатывать несколько каналов в одном потоке, используя селектор.

Обратите внимание, что селекторы не просто помогают вам читать данные; они также могут прослушивать входящие сетевые соединения и записывать данные по медленным каналам.

**3. Настроить**

Чтобы использовать селектор, нам не нужно никаких специальных настроек. Все классы, которые нам нужны, являются базовым пакетом *java.nio*, и мы просто должны импортировать то, что нам нужно.

После этого мы можем зарегистрировать несколько каналов с помощью селектора объекта.

Когда активность ввода-вывода происходит на каком-либо из каналов, селектор уведомляет нас. Вот как мы можем читать из большого количества источников данных из одного потока.

Любой канал, который мы регистрируем с помощью селектора, должен быть подклассом*SelectableChannel*. Это специальный тип каналов, которые можно перевести в неблокирующий режим.

**4. Создание селектора**

Селектор может быть создан путем вызова статического метода *open*класса *Selector*, который будет использовать системный поставщик по умолчанию для создания нового селектора:

Selector selector = Selector.open();

**5. Регистрация выбираемых каналов**

Чтобы селектор мог отслеживать любые каналы, мы должны зарегистрировать эти каналы в селекторе. Мы делаем это, вызывая метод *register*для выбранного канала.

Но прежде чем канал будет зарегистрирован с помощью селектора, он должен быть в неблокирующем режиме:

channel.configureBlocking(false);

SelectionKey key = channel.register(selector, SelectionKey.OP\_\_READ);

Это означает, что мы не можем использовать *\_\_FileChannel*\_\_s с селектором, поскольку они не могут быть переключены в неблокирующий режим, как мы делаем с сокетными каналами.

Первый параметр - это объект *Selector*, который мы создали ранее, второй параметр определяет набор интересов , означающий, какие события мы хотим прослушивать в отслеживаемом канале с помощью селектора.

Мы можем прослушать четыре различных события, каждое из которых представлено константой в классе *SelectionKey*:

* *Connect***-**когда клиент пытается подключиться к серверу.

Представлено *SelectionKey.OP*CONNECT Accept\_\_ - \*\* когда сервер принимает соединение от клиента.

Представлено *SelectionKey.OP*ACCEPT Read\_\_ - \*\* когда сервер готов к чтению с канала.

Представлено *SelectionKey.OP*READ Write\_\_ - \*\* когда сервер готов к записи в канал.

Представлено *SelectionKey.OP*WRITE\_\_

Возвращенный объект *SelectionKey*представляет регистрацию выбранного канала с помощью селектора. Мы рассмотрим это далее в следующем разделе.

**6. Объект *SelectionKey***

Как мы видели в предыдущем разделе, когда мы регистрируем канал с помощью селектора, мы получаем объект *SelectionKey*. Этот объект содержит данные, представляющие регистрацию канала.

Он содержит некоторые важные свойства, которые мы должны хорошо понимать, чтобы иметь возможность использовать селектор на канале. Мы рассмотрим эти свойства в следующих подразделах.

**6.1. Набор интересов**

Набор интересов определяет набор событий, которые мы хотим, чтобы селектор наблюдал на этом канале. Это целочисленное значение; мы можем получить эту информацию следующим образом.

Во-первых, у нас есть набор интересов, возвращаемый методом *SelectionKey*‘s *interestOps*. Тогда у нас есть константа события в *SelectionKey*, на которую мы смотрели ранее.

Когда мы И эти два значения, мы получаем логическое значение, которое сообщает нам, отслеживается ли событие или нет:

int interestSet = selectionKey.interestOps();

boolean isInterestedInAccept = interestSet & SelectionKey.OP\_\_ACCEPT;

boolean isInterestedInConnect = interestSet & SelectionKey.OP\_\_CONNECT;

boolean isInterestedInRead = interestSet & SelectionKey.OP\_\_READ;

boolean isInterestedInWrite = interestSet & SelectionKey.OP\_\_WRITE;

**6.2. Готовый комплект**

Готовый набор определяет набор событий, к которым канал готов.

Это также целочисленное значение; мы можем получить эту информацию следующим образом.

Мы получили готовый набор, возвращенный методом *SelectionKey*‘s *readyOps*.

Когда мы И это значение с константами событий, как мы это сделали в случае набора интереса, мы получаем логическое значение, представляющее, готов ли канал для определенного значения или нет.

Другой альтернативный и более короткий способ сделать это - использовать удобные методы \_\_\_\_SelectionKey для этой же цели:

selectionKey.isAcceptable();

selectionKey.isConnectable();

selectionKey.isReadable();

selectionKey.isWriteable();

**6.3. Канал**

Доступ к каналу, который просматривается из объекта *SelectionKey*, очень прост. Мы просто вызываем метод *channel*:

Channel channel = key.channel();

**6.4. Селектор**

Так же, как получение канала, очень просто получить объект *Selector*из объекта*SelectionKey*:

Selector selector = key.selector();

**6.5. Прикрепление объектов**

Мы можем прикрепить объект к *SelectionKey.*Иногда нам может потребоваться присвоить каналу собственный идентификатор или присоединить любой тип Java-объекта, который мы хотим отслеживать.

Присоединение объектов - удобный способ сделать это. Вот как вы прикрепляете и получаете объекты из *SelectionKey*:

key.attach(Object);

Object object = key.attachment();

Кроме того, мы можем выбрать прикрепить объект во время регистрации канала. Мы добавляем его в качестве третьего параметра в метод *register*канала, например так:

SelectionKey key = channel.register(

selector, SelectionKey.OP\_\_ACCEPT, object);

**7. Выбор ключа канала**

До сих пор мы рассмотрели, как создать селектор, зарегистрировать в нем каналы и проверить свойства объекта *SelectionKey*, который представляет регистрацию канала для селектора.

Это только половина процесса, теперь мы должны выполнить непрерывный процесс выбора готового набора, который мы рассматривали ранее. Мы делаем выбор, используя метод селектора *select*, например:

int channels = selector.select();

Этот метод блокируется, пока хотя бы один канал не будет готов к работе.

Возвращаемое целое число представляет количество ключей, каналы которых готовы к операции.

Далее мы обычно получаем набор выбранных ключей для обработки:

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

Мы получили набор объектов *SelectionKey*, каждый ключ представляет зарегистрированный канал, который готов к операции.

После этого мы обычно перебираем этот набор и для каждого ключа мы получаем канал и выполняем любые операции, которые появляются в нашем наборе интересов на нем.

За время существования канала его можно выбрать несколько раз, так как его ключ появляется в готовом наборе для разных событий. Вот почему у нас должен быть непрерывный цикл для захвата и обработки событий канала, как и когда они происходят.

**8. Полный пример**

Чтобы закрепить знания, которые мы получили в предыдущих разделах, мы собираемся создать полный пример клиент-сервер.

Для удобства тестирования нашего кода мы создадим эхо-сервер и эхо-клиент. При такой настройке клиент подключается к серверу и начинает отправлять на него сообщения. Сервер возвращает сообщения, отправленные каждым клиентом.

Когда сервер встречает определенное сообщение, такое как *end*, он интерпретирует его как конец связи и закрывает соединение с клиентом

**8.1. Сервер**

Вот наш код для *EchoServer.java*:

public class EchoServer {

private static final String POISON\_\_PILL = "POISON\_\_PILL";

public static void main(String[]args) throws IOException {

// создаем объект Selector

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

// создаем канал для потоко-ориентированного сокета прослушивания

ServerSocket**Channel** serverSocket = ServerSocketChannel.open();

// связываем канал с портом

serverSocket.bind(new InetSocketAddress("localhost", 5454));

// установливаем канал в неблокирующий режим

serverSocket.configureBlocking(false);

// регистрируем канал для селектора, с настройкой “ сервер принимает соединение от клиента”

serverSocket.register(selector, SelectionKey.OP\_\_ACCEPT);

// создаем новый ByteBuffer , с которого сервер будет писать и читать (т.к. NIO это буферно-ориентированную модель)

ByteBuffer buffer = ByteBuffer.allocate(256);

//создаем бесконечный цикл, поскольку серверы обычно должны продолжать работать независимо от того, есть активность или нет

while (true) {

// выбираем готовые каналы

selector.select();

//извлекаем их ключи выбора

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

//перебираем ключи

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

//выполняем операции, к которым готов каждый канал

while (iter.hasNext()) {

SelectionKey key = iter.next();

if (key.isAcceptable()) {

register(selector, serverSocket);

}

if (key.isReadable()) {

answerWithEcho(buffer, key);

}

iter.remove();

}

}

}

private static void **answerWithEcho**(ByteBuffer buffer, SelectionKey key)

throws IOException {

SocketChannel client = (SocketChannel) key.channel();

client.read(buffer);

Дальше идет блок, проверяющий полноту заполнения буфера. Чтобы начать работать только при получении всей информации от клиента, а не с ее частью. Например, “NAME An”. Для этого размер создаваемого буфера должен быть равен размеру получаемой от клиента информации.

/\* while(!bufferFull(client.read(buffer)))

{client.read(buffer);} \*/

if (new String(buffer.array()).trim().equals(POISON\_\_PILL)) {

client.close();

System.out.println("Not accepting client messages anymore");

}

buffer.flip();

client.write(buffer);

buffer.clear();

}

private static void **register**(Selector selector, ServerSocketChannel serverSocket)

throws IOException {

SocketChannel client = serverSocket.accept();

client.configureBlocking(false);

client.register(selector, SelectionKey.OP\_\_READ);

}

// Метод для запуска эхо-сервера как отдельного процесса во время модульного тестирования

public static Process **start**() throws IOException, InterruptedException {

String javaHome = System.getProperty("java.home");

String javaBin = javaHome + File.separator + "bin" + File.separator + "java";

String classpath = System.getProperty("java.class.path");

String className = EchoServer.class.getCanonicalName();

ProcessBuilder builder = new ProcessBuilder(javaBin, "-cp", classpath, className);

return builder.start();

}

}

Это то, что происходит; мы создаем объект *Selector*, вызывая статический метод *open*. Затем мы создаем канал, также вызывая его статический метод *open*, в частности экземпляр *ServerSocketChannel*

Это потому, что ***ServerSocketChannel*выбирается и подходит для потоково-ориентированного сокета прослушивания**.

Затем мы связываем его с портом по нашему выбору.

Помните, мы говорили ранее, что прежде чем регистрировать выбираемый канал для селектора, мы должны сначала установить его в неблокирующий режим. Итак, затем мы делаем это, а затем регистрируем канал для селектора.

На данном этапе нам не нужен экземпляр *SelectionKey*этого канала, поэтому мы его не запомним.

Java NIO использует буферно-ориентированную модель, отличную от потоково-ориентированной модели. Поэтому общение через сокеты обычно происходит путем записи в буфер и чтения из него.

Поэтому мы создаем новый *ByteBuffer*, с которого сервер будет писать и читать. Мы инициализируем его до 256 байт, это просто произвольное значение, в зависимости от того, сколько данных мы планируем передавать туда и обратно.

Наконец, мы выполняем процесс отбора. Мы выбираем готовые каналы, извлекаем их ключи выбора, перебираем ключи и выполняем операции, к которым готов каждый канал.

Мы делаем это в бесконечном цикле, поскольку серверы обычно должны продолжать работать независимо от того, есть активность или нет.

Единственная операция, которую может выполнить *ServerSocketChannel*, - это операция*ACCEPT*. Когда мы принимаем соединение от клиента, мы получаем объект *SocketChannel*, для которого мы можем выполнять чтение и запись. Мы устанавливаем его в неблокирующий режим и регистрируем его для операции чтения для селектора.

Во время одного из следующих выборов этот новый канал станет готовым для чтения. Мы извлекаем его и читаем его содержимое в буфер.

Так как наш сервер это эхо-сервер, мы должны записать этот контент обратно клиенту:

а. метод flip() переводит позицию в буфере на 0, чтобы мы могли считать с буфера информацию (*для отправки клиенту*) с самого ее начала;

б. считываем информацию с буфера и отправляем клиенту;

в. Метод clear() обнуляет информацию в буфере (*переводит позицию на 0, сбрасывает все метки, границу делает равной вместимости*).

Метод *start ()*определен так, что эхо-сервер может быть запущен как отдельный процесс во время модульного тестирования.

**8.2. Клиент**

Вот наш код для *EchoClient.java*:

public class EchoClient {

private static SocketChannel client;

private static ByteBuffer buffer;

private static EchoClient instance;

public static EchoClient start() {

if (instance == null)

instance = new EchoClient();

return instance;

}

public static void stop() throws IOException {

client.close();

buffer = null;

}

private EchoClient() {

try {

client = SocketChannel.open(new InetSocketAddress("localhost", 5454));

buffer = ByteBuffer.allocate(256);

} catch (IOException e) {

e.printStackTrace();

}

}

public String sendMessage(String msg) {

buffer = ByteBuffer.wrap(msg.getBytes());

String response = null;

try {

client.write(buffer);

buffer.clear();

client.read(buffer);

response = new String(buffer.array()).trim();

System.out.println("response=" + response);

buffer.clear();

} catch (IOException e) {

e.printStackTrace();

}

return response;

}

}

Клиент проще сервера.

Мы используем одноэлементный шаблон, чтобы создать его экземпляр внутри статического метода *start*. Мы вызываем приватный конструктор из этого метода.

В приватном конструкторе мы открываем соединение на том же порту, к которому был привязан канал сервера, и на том же хосте.

Затем мы создаем буфер, в который мы можем писать и из которого мы можем читать.

Наконец, у нас есть метод *sendMessage*, который читает обертки любой передаваемой ей строки в байтовый буфер, который передается по каналу на сервер.

Затем мы читаем из клиентского канала, чтобы получить сообщение, отправленное сервером. Мы возвращаем это как эхо нашего сообщения.

**8.3. Тестирование**

Внутри класса *EchoTest.java*мы собираемся создать тестовый пример, который запускает сервер, отправляет сообщения на сервер и проходит только тогда, когда те же сообщения возвращаются с сервера. На последнем этапе тестовый пример останавливает сервер до его завершения.

Теперь мы можем запустить тест:

public class EchoTest {

Process server;

EchoClient client;

@Before

public void setup() throws IOException, InterruptedException {

server = EchoServer.start();

client = EchoClient.start();

}

@Test

public void givenServerClient\_\_whenServerEchosMessage\_\_thenCorrect() {

String resp1 = client.sendMessage("hello");

String resp2 = client.sendMessage("world");

assertEquals("hello", resp1);

assertEquals("world", resp2);

}

@After

public void teardown() throws IOException {

server.destroy();

EchoClient.stop();

}

}

**9. Заключение**

В этой статье мы рассмотрели базовое использование компонента Java NIO Selector.

Полный исходный код и все фрагменты кода для этой статьи доступны в моем [проект GitHub](https://github.com/eugenp/tutorials/tree/master/core-java-io/src/main/java/com/baeldung/java/nio/%D1%81%D0%B5%D0%BB%D0%B5%D0%BA%D1%82%D0%BE%D1%80).