Java NIO

Presenter: Eric (Haotao) Lai
Contact: haotao.lai@gmail.com

What is I/O operation

Usually, I/O operation contains the read/write operation of the hard disk, socket or other peripherals.

It is consist of two parts:

1. check the data is ready or not;
2. move the data from one place to another;

Blocking I/O vs. Non-blocking I/O

They are different in the first part, check the data is ready or not, if the data is NOT ready to be moved from one place to another:

- blocking I/O: do nothing but wait until the data get ready;
- non-blocking I/O: go to do another job until the signal told the data is ready;

Synchronous I/O vs. Asynchronous I/O

[ei’siŋkrənəs]

- A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;
- An asynchronous I/O operation does not cause the requesting process to be blocked;

The key to identify those two concepts is to see who is responsible for moving the data, synchronous I/O need the user process to check the data status and for asynchronous I/O all jobs are done by the OS.

Note: please don’t confused with the BIO and NIO concepts, they are talking about different aspects.

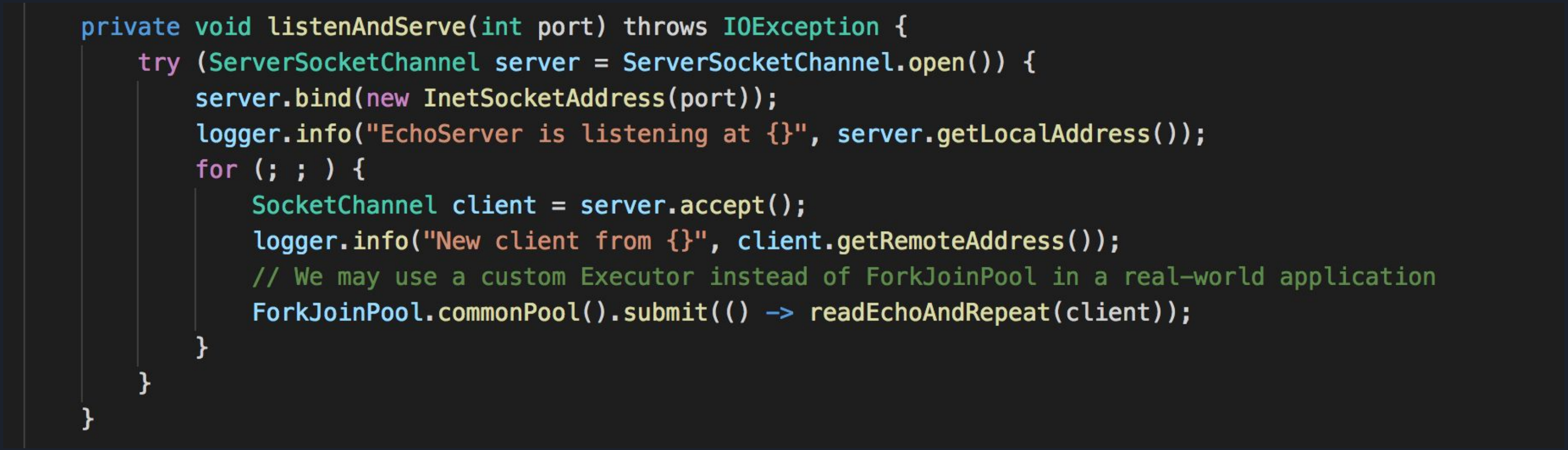
Java NIO Programming

Channel: similar to InputStream/OutputStream in traditional I/O programming, you can read OR write data through the channel.

Buffer: a place to hold the data.

Selector: loop to check all register channel, once found a register event occurred, catch it and handle it.

Channel



```java
private void listenAndServe(int port) throws IOException {
    try (ServerSocketChannel server = ServerSocketChannel.open()) {
        server.bind(new InetSocketAddress(port));
        logger.info("EchoServer is listening at {}", server.getLocalAddress());
        for (; ; ) {
            SocketChannel client = server.accept();
            logger.info("New client from {}", client.getRemoteAddress());
            // We may use a custom Executor instead of ForkJoinPool in a real-world application
            ForkJoinPool.commonPool().submit(() -> readEchoAndRepeat(client));
        }
    }
}
```

Buffer

Buffer actually is an array with some build-in operation which can help you to easily play with that array. We can use three properties to describe the buffer:

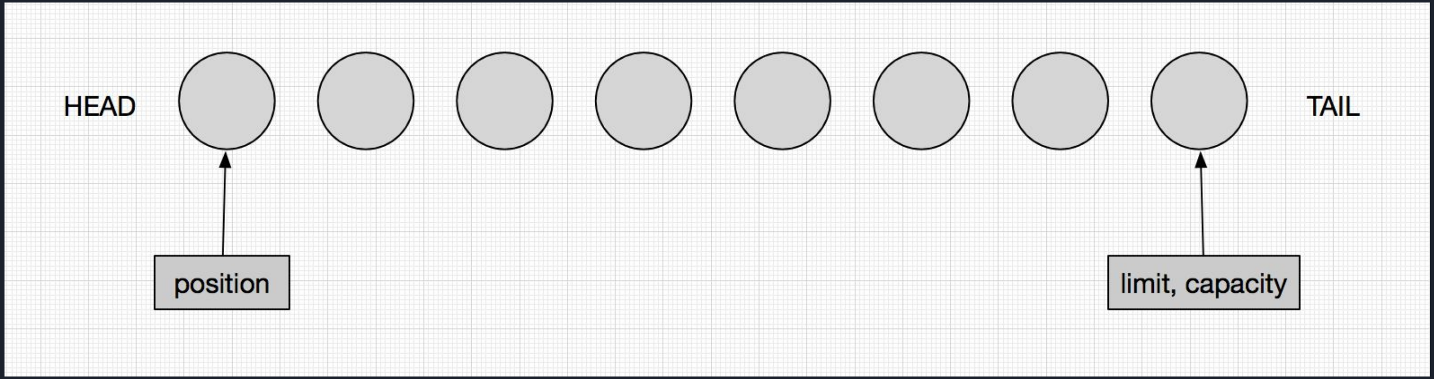
- position, it points out where the next data can be placed (write) / picked (read);
- limit, it tells us how many data need to be handled;
- capacity, it shown the length of the array;

Let’s go to an example.

Buffer Example (1)

Assume we have a ByteBuffer with capacity 8 bytes.

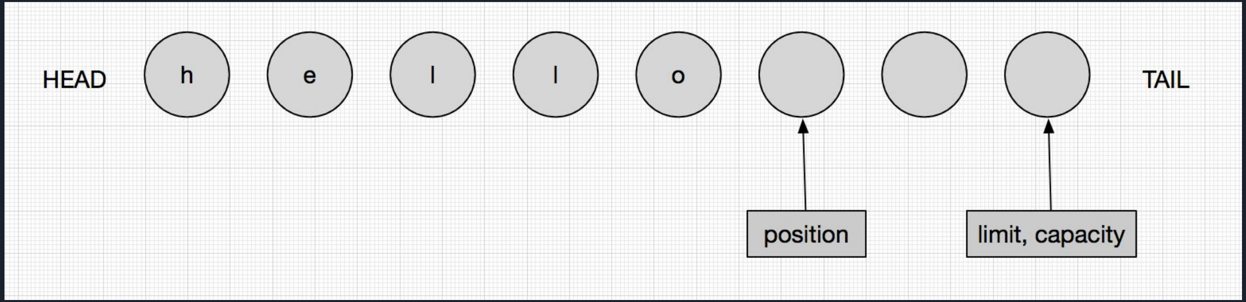
At the very beginning when I create the buffer, it should look like:



Buffer Example (2)

Assume now we read some data into the buffer, since the【position】points to the first slot, it indicate that the coming data should be put into that place.

When we finish reading “hello” (in reality world the data in each slot should NOT be the letter shows below, here just for make you understanding the idea), the buffer looks like:



Buffer Example (3)

Assume now, we want to write the data in the buffer out to some other place. But please keep in mind that

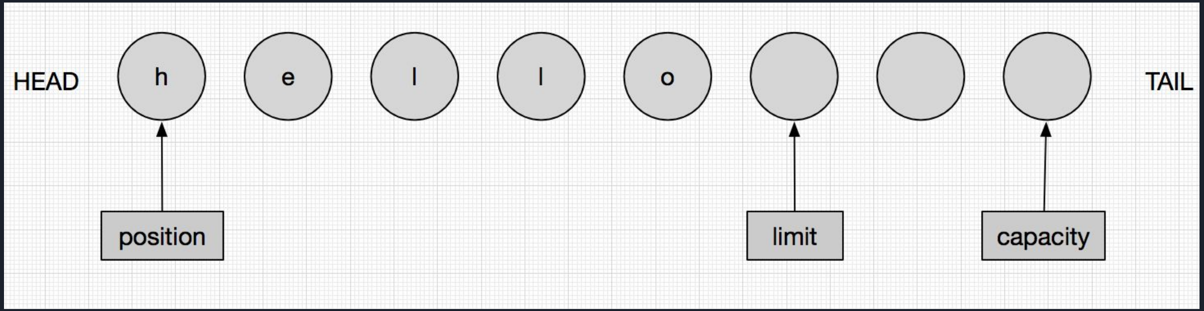
1. 【position】always point to the next operate element
2. 【limit】tells us how many data we need to deal with;

If we write the data directly, we will have a problem (please see the previous slide, you can write nothing out)!

Buffer Example (4)

The solution is to make the【position】back to the beginning of the array. So if we say【flip】then, the result is shown below.

Also please know that【position】pointer can <u>NEVER</u> go behind【limit】pointer.



Buffer Example (5)

After a read and write, if you want to make the buffer clean and repeat the operation, you can just simply say【clean】which will reset all three pointer to their original place.

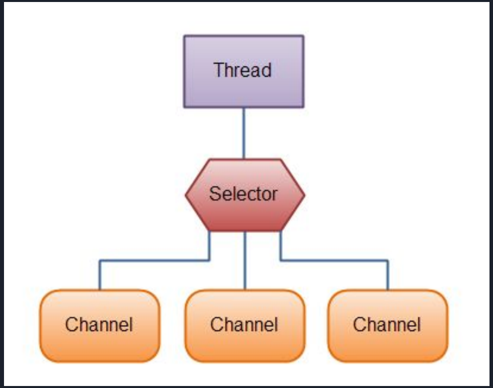
There is also a【reset】operation, it is not the same as【clean】.

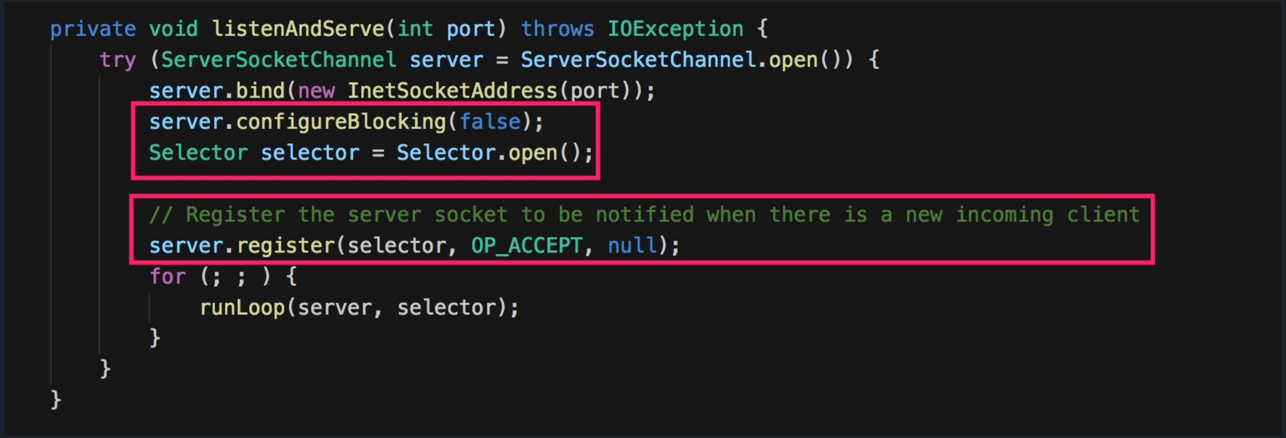
Selector

The core feature inside asynchronous in Java NIO.

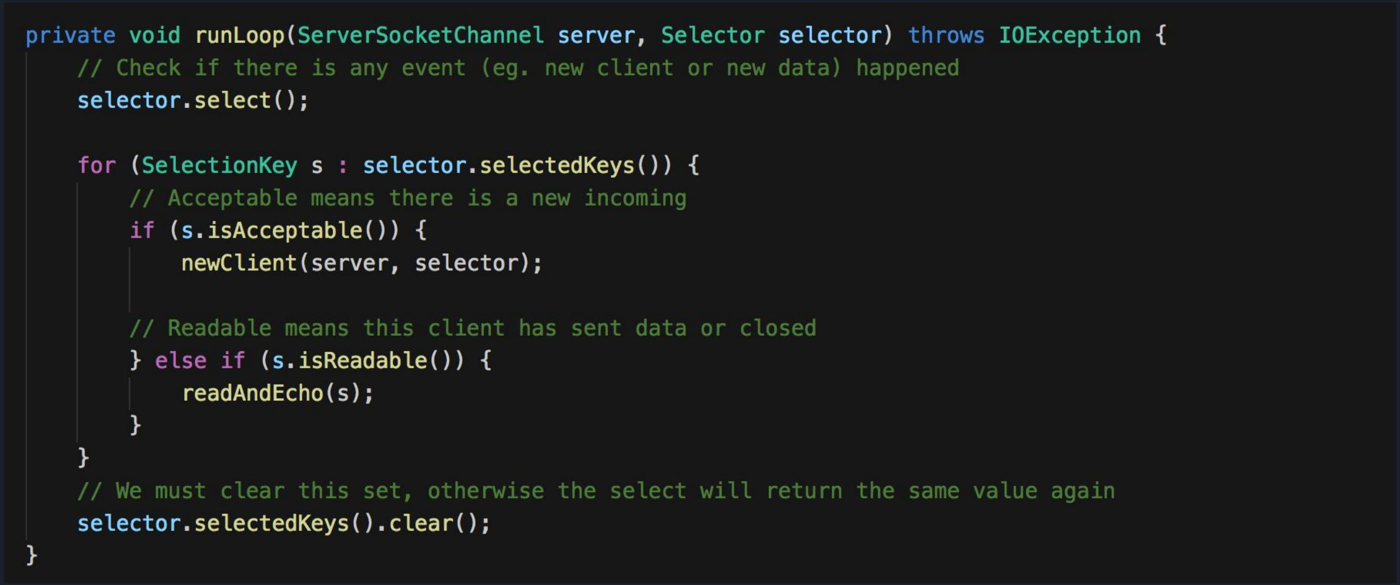
We use Selector to register the event which we are interested in, when the registered event happen it will tell us.

With Selector, we can use only one thread to manage multiple channels.

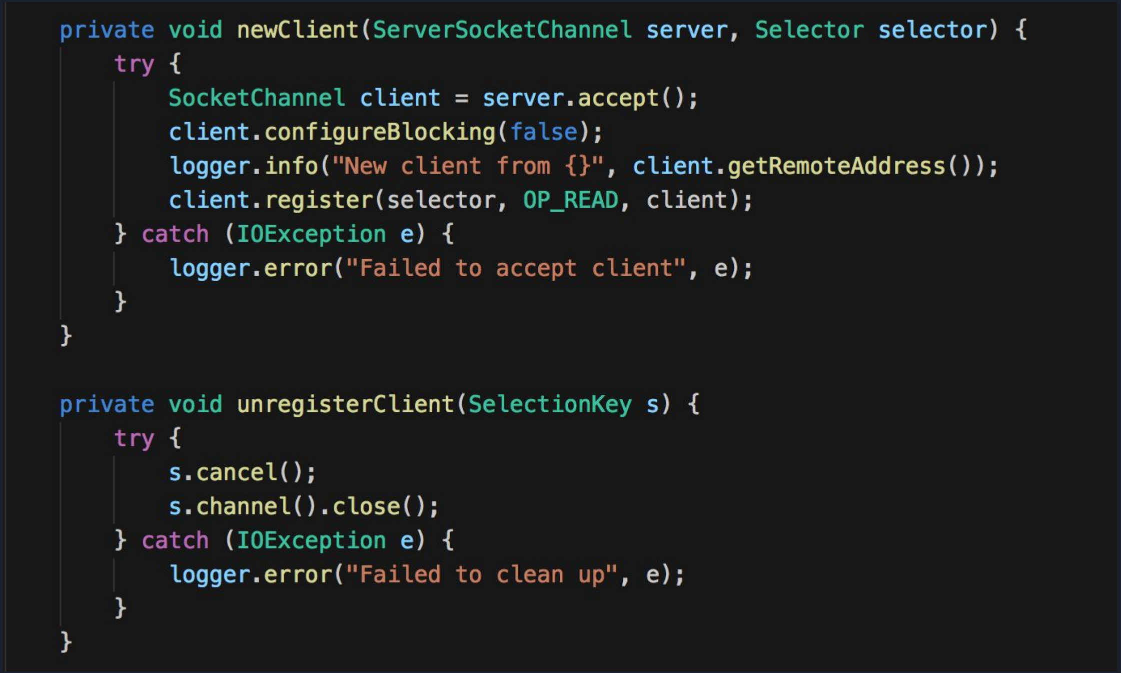




```java
private void listenAndServe(int port) throws IOException {
    try (ServerSocketChannel server = ServerSocketChannel.open()) {
        server.bind(new InetSocketAddress(port));
        server.configureBlocking(false);
        Selector selector = Selector.open();

        // Register the server socket to be notified when there is a new incoming client
        server.register(selector, OP_ACCEPT, null);
        for (; ; ) {
            runLoop(server, selector);
        }
    }
}
```



```java
private void runLoop(ServerSocketChannel server, Selector selector) throws IOException {
    // Check if there is any event (eg. new client or new data) happened
    selector.select();

    for (SelectionKey s : selector.selectedKeys()) {
        // Acceptable means there is a new incoming
        if (s.isAcceptable()) {
            newClient(server, selector);

        // Readable means this client has sent data or closed
        } else if (s.isReadable()) {
            readAndEcho(s);
        }
    }
    // We must clear this set, otherwise the select will return the same value again
    selector.selectedKeys().clear();
}
```



```java
private void newClient(ServerSocketChannel server, Selector selector) {
    try {
        SocketChannel client = server.accept();
        client.configureBlocking(false);
        logger.info("New client from {}", client.getRemoteAddress());
        client.register(selector, OP_READ, client);
    } catch (IOException e) {
        logger.error("Failed to accept client", e);
    }
}

private void unregisterClient(SelectionKey s) {
    try {
        s.cancel();
        s.channel().close();
    } catch (IOException e) {
        logger.error("Failed to clean up", e);
    }
}
```