Java NIO core knowledge summary

Guide Java Java IO, Java Basics About 3899 words About 13 minutes

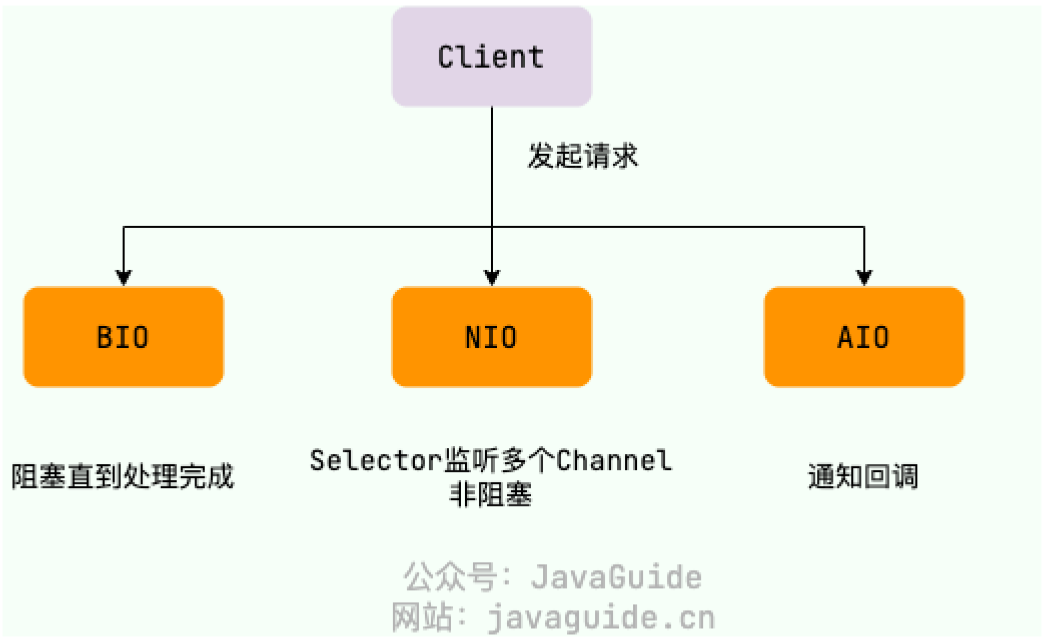
Before learning NIO, you need to understand the basic theoretical knowledge of the computer I/O model. If you don't understand it yet, you can refer to this article I wrote: Detailed Explanation of the Java IO Model .

About NIO

In the traditional Java I/O model (BIO), I/O operations are performed in a blocking manner. That is, when a thread performs an I/O operation, it is blocked until the operation completes. This blocking model can lead to performance bottlenecks when handling multiple concurrent connections because a thread must be created for each connection, and thread creation and switching are both expensive.

To address this issue, Java 1.4 introduced a new I/O model— **NIO** (New IO, also known as Non-blocking IO). NIO addresses the shortcomings of synchronous blocking I/O by providing non-blocking, buffered, channel-based I/O within standard Java code. It can handle multiple connections using a small number of threads, significantly improving I/O efficiency and concurrency.

The figure below is a simple comparison of BIO, NIO and AIO processing client requests (for an introduction to AIO, you can read this article I wrote: Detailed Explanation of Java IO Model , which is not the focus, just understand it).



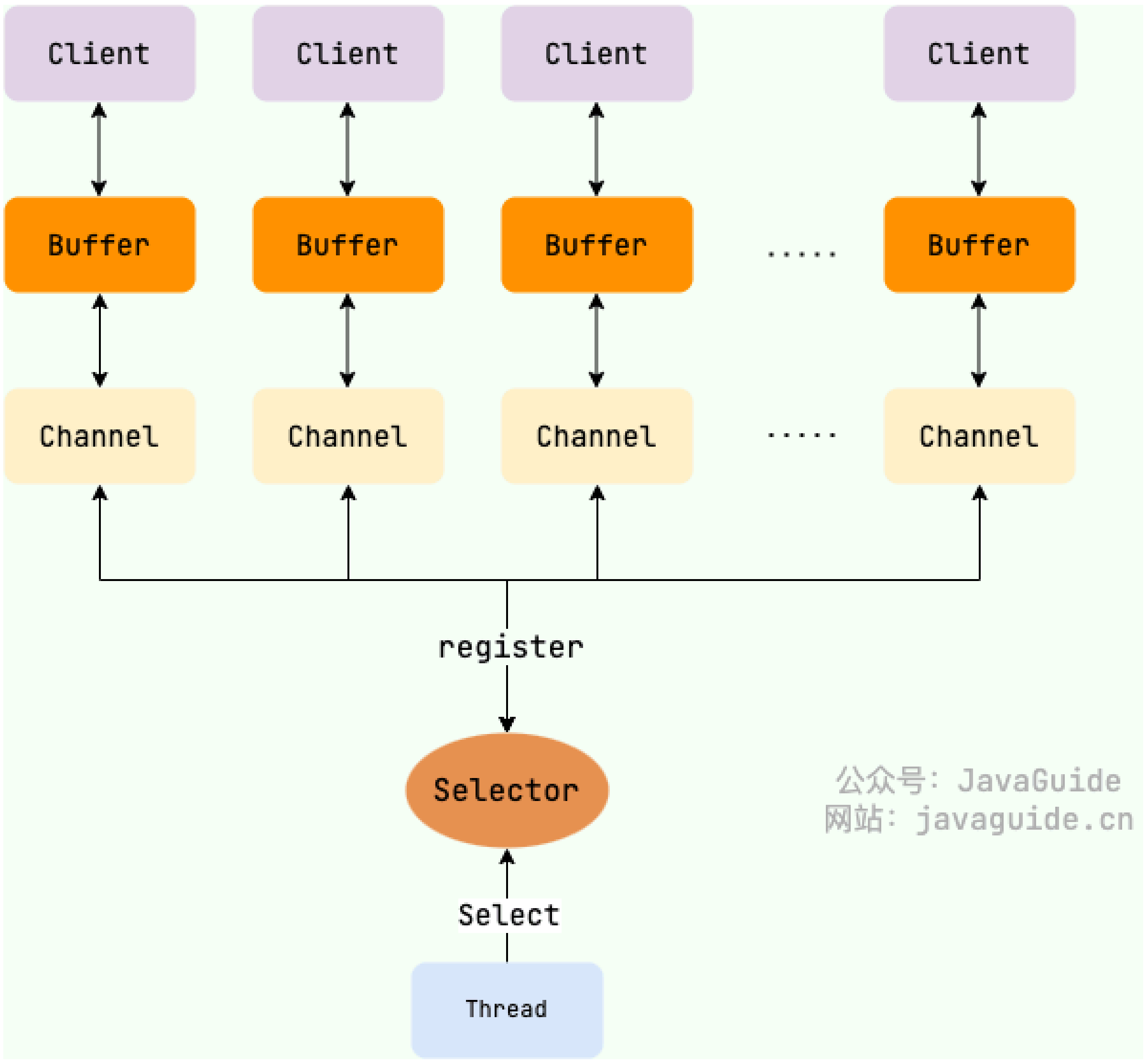
⚠️ Note: Using NIO doesn't necessarily guarantee high performance. Its performance advantage is primarily in high-concurrency and high-latency network environments. When there are fewer connections, lower concurrency, or faster network transmission speeds, NIO's performance may not necessarily outperform traditional BIO.

NIO core components

NIO mainly includes the following three core components:

- **Buffer** : NIO reads and writes data through the buffer. During a read operation, the data in the channel is filled into the buffer, and during a write operation, the data in the buffer is written into the channel.
- **Channel** : A channel is a bidirectional, readable and writable data transmission channel. NIO uses channels to implement data input and output. A channel is an abstract concept that can represent a connection between files, sockets, or other data sources.
- **Selector** : Allows a single thread to handle multiple channels, based on an event-driven I/O multiplexing model. All channels can be registered with a Selector, which then assigns threads to handle events.

The relationship between the three is shown in the figure below (it doesn’t matter if you don’t understand it for now, it will be explained in detail later):



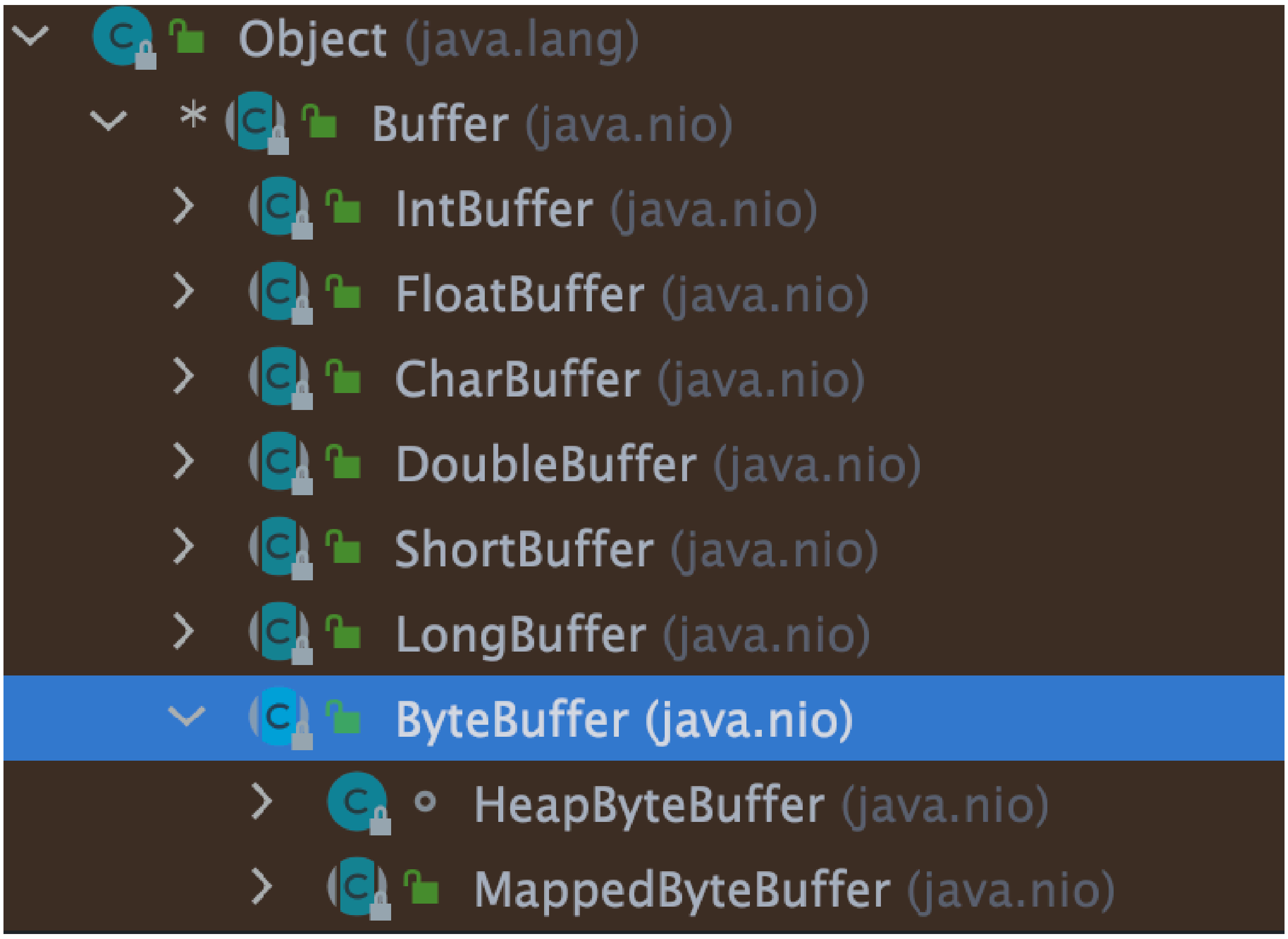
The following is a detailed introduction to these three components.

Buffer

In traditional BIO, data reading and writing are stream-oriented, divided into byte stream and character stream.

In Java 1.4's NIO library, all data is handled using buffers. This is a key difference between the new library and the previous BIO library, somewhat similar to the buffered streams in BIO. When reading data, NIO reads it directly into the buffer. When writing data, it writes it directly into the buffer. When using NIO, both reading and writing data are performed through the buffer.

`Buffer` The subclasses of are shown in the figure below. Among them, the most commonly used is `ByteBuffer` , it can be used to store and operate byte data.



```
Object (java.lang)
    * Buffer (java.nio)
        IntBuffer (java.nio)
        FloatBuffer (java.nio)
        CharBuffer (java.nio)
        DoubleBuffer (java.nio)
        ShortBuffer (java.nio)
        LongBuffer (java.nio)
        ByteBuffer (java.nio)
            HeapByteBuffer (java.nio)
            MappedByteBuffer (java.nio)
```

You can think of Buffer as an array, `IntBuffer` where , `FloatBuffer` , etc. correspond to , , etc. `CharBuffer` respectively . `int[]` `float[]` `char[]`

To understand the buffer more clearly, let's take a quick look at `Buffer` the four member variables defined in the class:

```java
public abstract class Buffer {
    // Invariants: mark <= position <= limit <= capacity
    private int mark = -1;
    private int position = 0;
    private int limit;
    private int capacity;
}
```

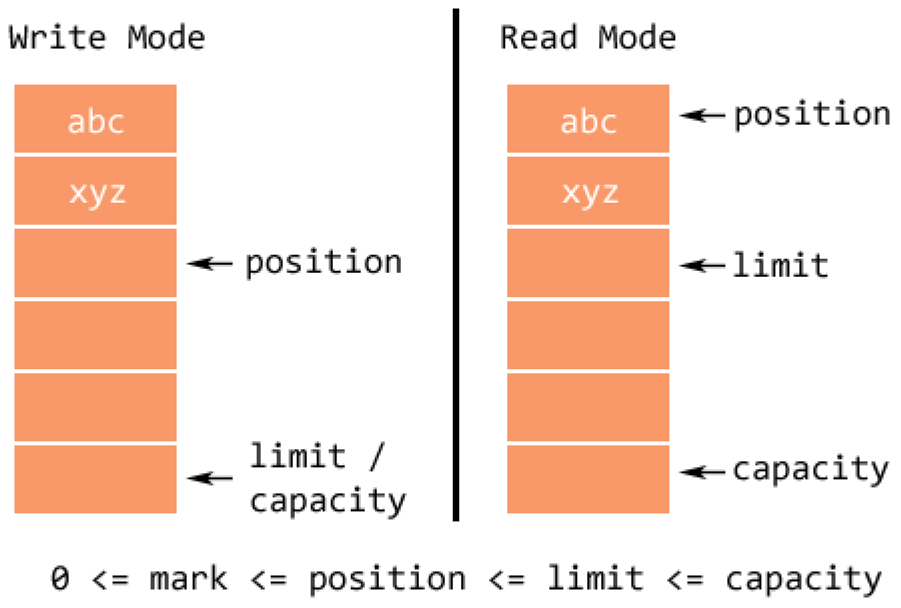
The specific meanings of these four member variables are as follows:

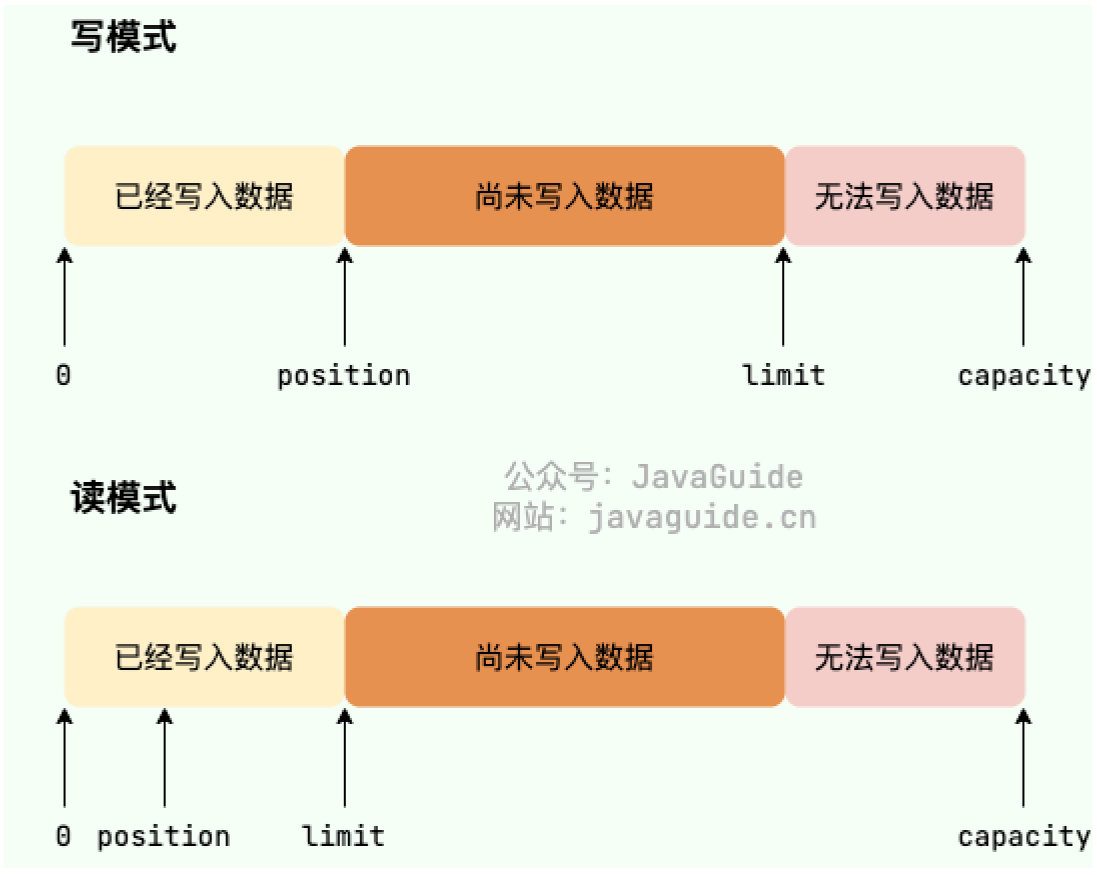
1. Capacity (`capacity`): `Buffer` The maximum amount of data that can be stored, `Buffer` set at creation time and cannot be changed;
2. Limit (`limit`): `Buffer` The boundary of the data that can be read/written. In wr mode, `limit` it represents the maximum amount of data that can be written, generally

equal to `capacity` (can `limit(int newLimit)` be set through methods); in read mode, `limit` it is equal to the actual size of the data written to the buffer.
3. Position (`position`): The position (index) of the next data that can be read or written. When switching from write mode to read mode (flip), `position` it will be reset to zero so that reading and writing can start from the beginning.
4. Marker (`mark`): `Buffer` allows the position to be directly positioned at the marker, which is an optional attribute;

Moreover, the above variables satisfy the following relationship: **0 <= mark <= position <= limit <= capacity** .

Buffers have two modes: read mode and write mode, used to read and write data to the buffer, respectively. When a buffer is created, it is in write mode by default `flip()` . You can switch to read mode by calling Buffer() . To switch back to write mode, you can call the Buffer() or Buffer `clear()` () `compact()` methods.





`Buffer` Objects cannot `new` be created by calling constructors and can only be instantiated through static methods `Buffer` .

Here `ByteBuffer` we take as an example:

```java
//
public static ByteBuffer allocate(int capacity);
//
public static ByteBuffer allocateDirect(int capacity);
```

The two core methods of Buffer are:

1. `get` : Read data from the buffer
2. `put` : Write data to the buffer

In addition to the above two methods, other important methods are:

- `flip` : Switches the buffer from write mode to read mode, which `limit` sets the value of to the current `position` value of and `position` sets the value of to 0.
- `clear` : Clear the buffer, switch the buffer from read mode to write mode, and `position` set the value of to 0 and `limit` the value of to `capacity` the value of .
- ...

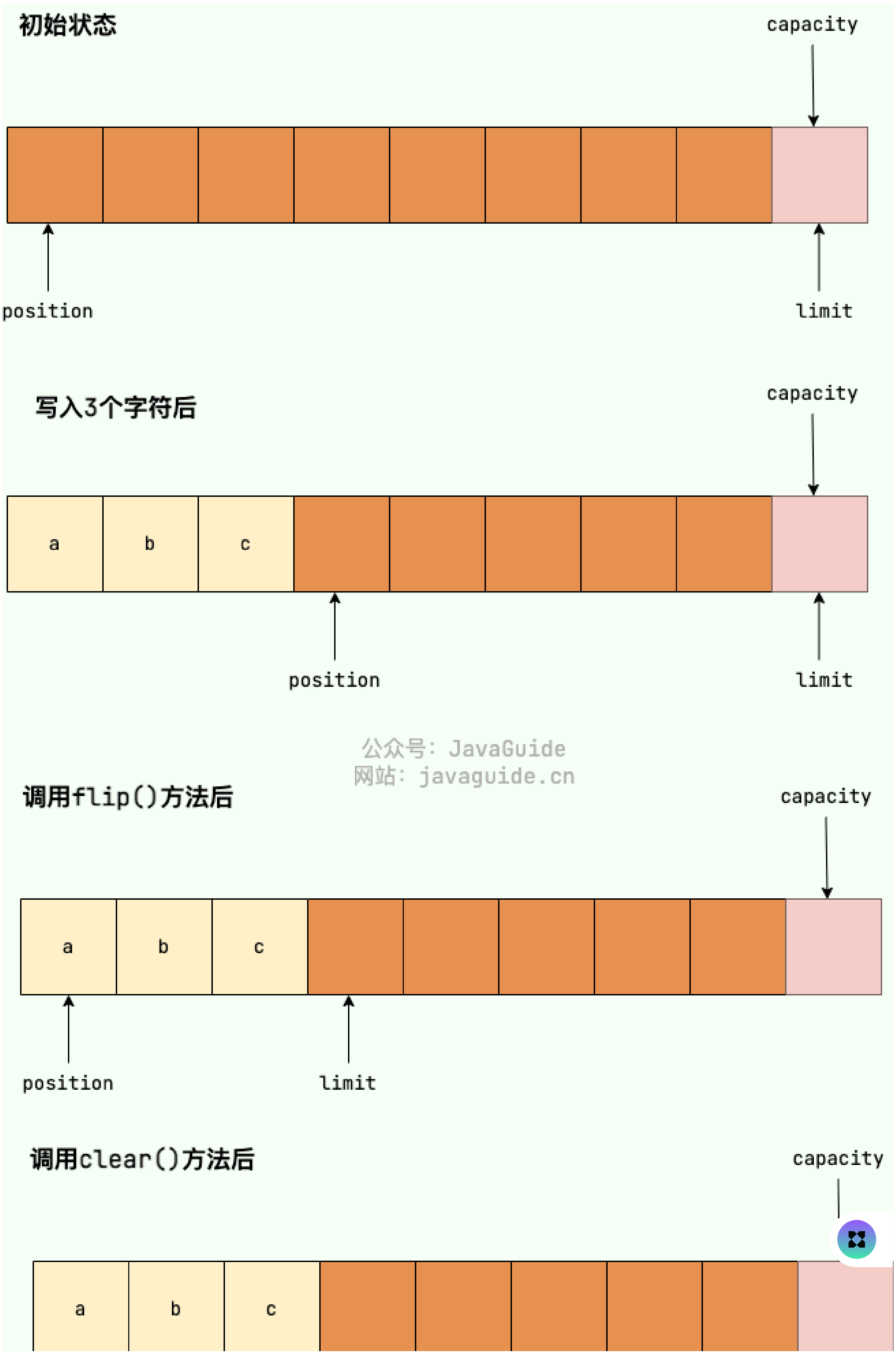
The process of data change in Buffer:

```java
import java.nio.*;

public class CharBufferDemo {
    public static void main(String[] args) {
        //             8  CharBuffer
        CharBuffer buffer = CharBuffer.allocate(8);
        System.out.println("        ");
        printState(buffer);

        //   buffer   3
        buffer.put('a').put('b').put('c');
        System.out.println("   3             ");
        printState(buffer);

        //    flip()            buffer           position
0,limit     3
        buffer.flip();
        System.out.println("   flip()           ");
        printState(buffer);

        //
        while (buffer.hasRemaining()) {
            System.out.print(buffer.get());
        }

        //    clear()                 position        0     limit
        capacity
        buffer.clear();
        System.out.println("   clear()           ");
        printState(buffer);

    }

    //    buffer  capacity  limit  position  mark
    private static void printState(CharBuffer buffer) {
        System.out.print("capacity: " + buffer.capacity());
        System.out.print(", limit: " + buffer.limit());
        System.out.print(", position: " + buffer.position());
        System.out.print(", mark             : " + buffer.mark());
        System.out.println("\n");
    }
}
```

Output:

```bash

capacity: 8, limit: 8, position: 0

   3
capacity: 8, limit: 8, position: 3

      buffer

   flip()
capacity: 8, limit: 3, position: 0

           abc

   clear()
capacity: 8, limit: 8, position: 0
```

To help you understand, I drew a picture showing the process `capacity` and changes `limit` at `position` each stage.

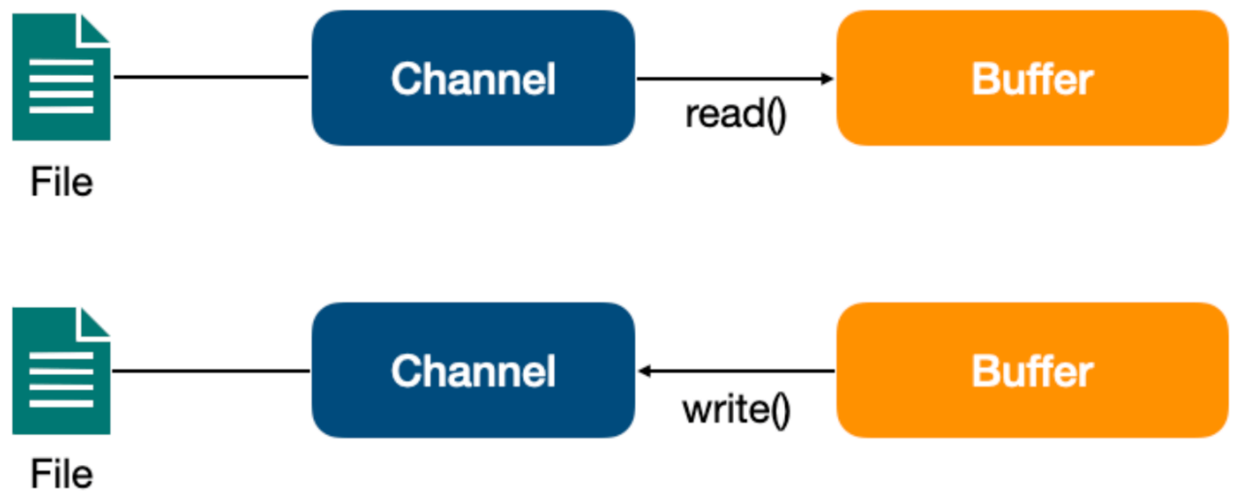

初始状态
capacity
position
limit
写入3个字符后
capacity
a
b
c
position
limit
公众号：JavaGuide
网站：javaguide.cn
调用flip()方法后
capacity
a
b
c
position
limit
调用clear()方法后
capacity
a
b
c

Channel

A Channel is a channel that establishes a connection with a data source (such as a file, network socket, etc.). We can use it to read and write data, just like opening a water pipe and letting data flow freely in the Channel.

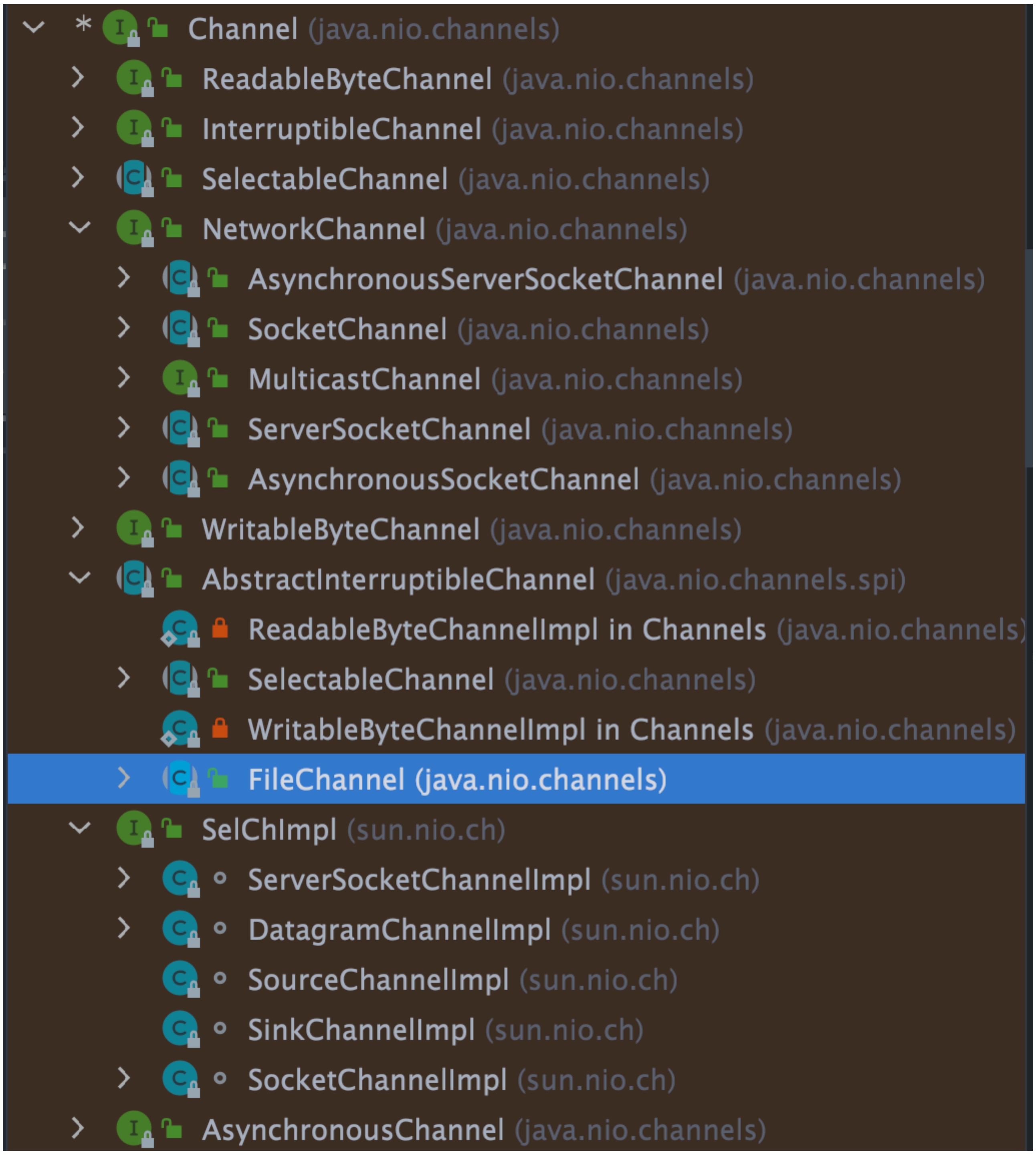
Streams in BIO are unidirectional and are classified as `InputStream` input streams and `OutputStream` output streams. Data is transmitted in one direction only. Channels differ from streams in that they are bidirectional and can be used for reading, writing, or both.

The Channel interacts with the Buffer introduced earlier. During a read operation, the data in the Channel is filled into the Buffer, while during a write operation, the data in the Buffer is written into the Channel.



In addition, because Channel is full-duplex, it can better map the API of the underlying operating system than streams. In particular, in the UNIX network programming model, the channels of the underlying operating system are full-duplex and support both read and write operations.

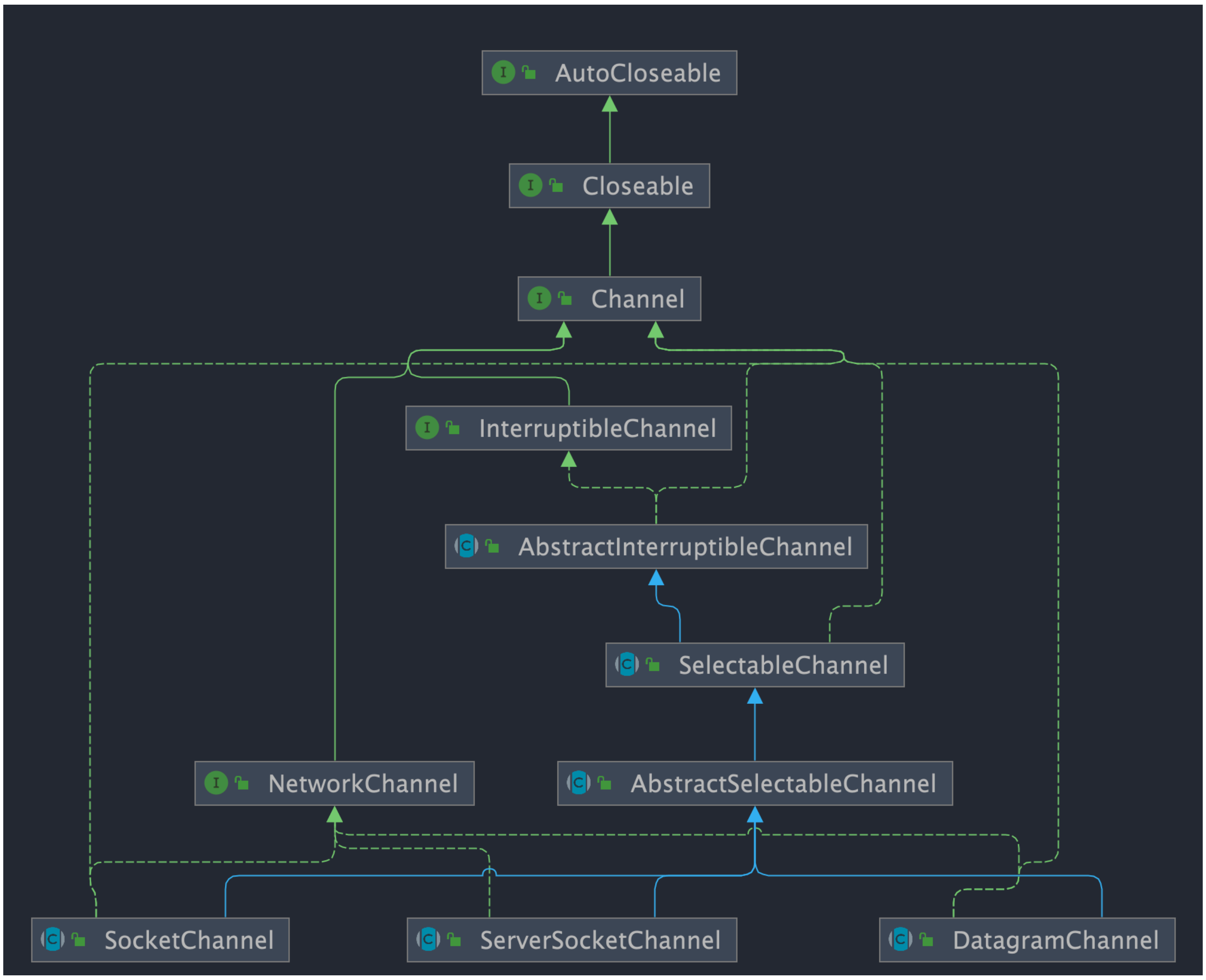
`Channel` The subclasses are shown below.



```
Channel (java.nio.channels)
    ReadableByteChannel (java.nio.channels)
    InterruptibleChannel (java.nio.channels)
    SelectableChannel (java.nio.channels)
    NetworkChannel (java.nio.channels)
        AsynchronousServerSocketChannel (java.nio.channels)
        SocketChannel (java.nio.channels)
        MulticastChannel (java.nio.channels)
        ServerSocketChannel (java.nio.channels)
        AsynchronousSocketChannel (java.nio.channels)
    WritableByteChannel (java.nio.channels)
    AbstractInterruptibleChannel (java.nio.channels.spi)
        ReadableByteChannelImpl in Channels (java.nio.channels)
        SelectableChannel (java.nio.channels)
        WritableByteChannelImpl in Channels (java.nio.channels)
        FileChannel (java.nio.channels)
    SelChImpl (sun.nio.ch)
        ServerSocketChannelImpl (sun.nio.ch)
        DatagramChannelImpl (sun.nio.ch)
        SourceChannelImpl (sun.nio.ch)
        SinkChannelImpl (sun.nio.ch)
        SocketChannelImpl (sun.nio.ch)
    AsynchronousChannel (java.nio.channels)
```

Among them, the most commonly used are the following types of channels:

- `FileChannel` : File access channel;
- `SocketChannel` , `ServerSocketChannel` : TCP communication channel;
- `DatagramChannel` : UDP communication channel;



The two core methods of Channel are:

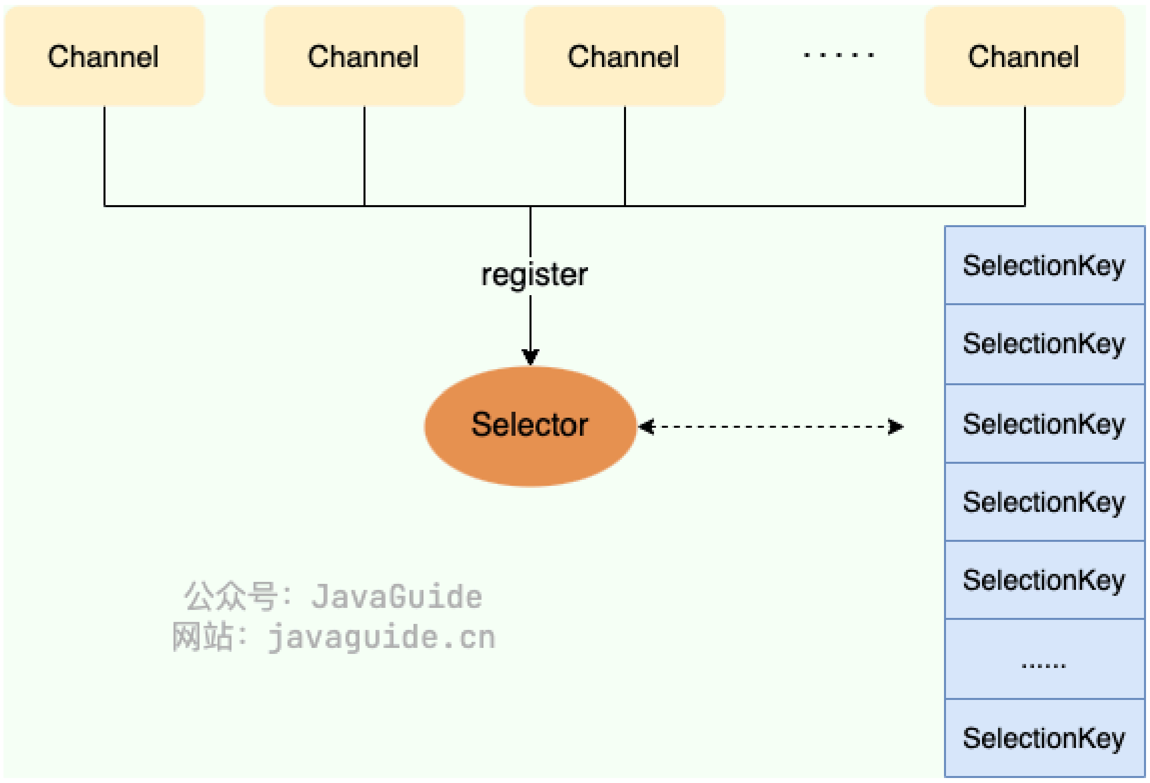
1. `read` : Read data and write it into Buffer.
2. `write` : Write the data in the Buffer to the Channel.

Here we take as `FileChannel` an example to demonstrate how to read file data.

```java
RandomAccessFile reader = new
RandomAccessFile("/Users/guide/Documents/test_read.in", "r"))
FileChannel channel = reader.getChannel();
ByteBuffer buffer = ByteBuffer.allocate(1024);
channel.read(buffer);
```

Selector

The Selector is a key component in NIO, allowing a single thread to handle multiple Channels. The Selector is based on an event-driven I/O multiplexing model. Its main operating principle is: Channel events are registered through the Selector, and the Selector continuously polls the registered Channels. When an event occurs, such as a new TCP connection, read, or write event on a Channel, the Channel enters the Ready state and is polled by the Selector. The Selector adds the relevant Channels to the Ready Set. The SelectionKey allows you to obtain a collection of Ready Channels, and then perform corresponding I/O operations on these Ready Channels.



A multiplexer Selector can poll multiple Channels simultaneously. Since the JDK uses `epoll()` instead of the traditional `select` implementation, it does not have a maximum connection handle `1024/2048` limit. This means that only one thread is needed to poll the Selector and connect to thousands of clients.

Selector can listen for the following four event types:

1. `SelectionKey.OP_ACCEPT` : Indicates that the channel accepts the connection event, which is usually used `ServerSocketChannel` .
2. `SelectionKey.OP_CONNECT` : Indicates that the channel has completed the connection event, which is usually used `SocketChannel` .
3. `SelectionKey.OP_READ` : Indicates that the channel is ready for reading, that is, there is data to be read.
4. `SelectionKey.OP_WRITE` : Indicates that the channel is ready for writing, that is, data can be written.

`Selector` It is an abstract class. You can create a Selector instance by calling its `open()` static methods. Selector can monitor the status `SelectableChannel` of multiple at the same time `IO` and is `IO` the core of non-blocking .

A Selector instance has three `SelectionKey` collections:

1. All `SelectionKey` collections: represents the ones registered on this Selector `Channel` . This collection can be `keys()` returned by the method.
2. The selected `SelectionKey` collection: represents all Channels that can `select()` be obtained through the method and need to be processed . This collection can be returned through . `IO` `selectedKeys()`
3. Cancelled `SelectionKey` collection: represents all the canceled registration relationships `Channel` . When the method is executed next time `select()` , these `Channel` corresponding `SelectionKey` will be completely deleted. The program usually does not need to directly access this collection, and there is no exposed access method.

Here is a brief demonstration of how to traverse the selected `SelectionKey` collection and process it:

```java
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while (keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if (key != null) {
        if (key.isAcceptable()) {
            // ServerSocketChannel
        } else if (key.isConnectable()) {
            //
        } else if (key.isReadable()) {
            // Channel
        } else if (key.isWritable()) {
            // Channel         Buffer
        }
    }
    keyIterator.remove();
}
```

Selector also provides a series of `select()` related methods:

- `int select()` : Monitors all registered ones `Channel` . When there is an operation that needs to be processed among them `IO` , this method returns and

`SelectionKey` adds the corresponding to the selected `SelectionKey` set. This method returns `Channel` the number of these.

- `int select(long timeout)` : You can set the timeout period for `select()` the operation.
- `int selectNow()` : Executes an operation that returns immediately `select()` . Compared with the method without parameters `select()` , this method will not block the thread.
- `Selector wakeup()` : Make a method that has not yet returned `select()` return immediately.
- ...

A simple example of using Selector to implement network reading and writing:

```java
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 NioSelectorExample {

  public static void main(String[] args) {
    try {
      ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
      serverSocketChannel.configureBlocking(false);
      serverSocketChannel.socket().bind(new
InetSocketAddress(8080));

      Selector selector = Selector.open();
      //    ServerSocketChannel        Selector        OP_ACCEPT
      serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);

      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()) {
            //
            ServerSocketChannel server = (ServerSocketChannel)
key.channel();
            SocketChannel client = server.accept();
            client.configureBlocking(false);

            //                  Selector       OP_READ
            client.register(selector, SelectionKey.OP_READ);
          } else if (key.isReadable()) {
            //
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            int bytesRead = client.read(buffer);

            if (bytesRead > 0) {
              buffer.flip();
              System.out.println("        " +new
String(buffer.array(), 0, bytesRead));
              //                  Selector       OP_WRITE
              client.register(selector, SelectionKey.OP_WRITE);
            } else if (bytesRead < 0) {
              //
              client.close();
            }
          } else if (key.isWritable()) {
            //
            SocketChannel client = (SocketChannel) key.channel();
            ByteBuffer buffer = ByteBuffer.wrap("Hello,
Client!".getBytes());
            client.write(buffer);

            //                  Selector       OP_READ
```

```
                client.register(selector, SelectionKey.OP_READ);
            }

            keyIterator.remove();
          }
        }
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

In this example, we create a simple server that listens on port 8080 and uses Selector to handle connection, read, and write events. When receiving data from the client, the server reads the data and prints it to the console, then replies "Hello, Client!" to the client.

NIO Zero Copy

Zero copy is a common method to improve IO operation performance. Top open source projects such as ActiveMQ, Kafka, RocketMQ, QMQ, Netty, etc. all use zero copy.

Zero copy means that when a computer performs I/O operations, the CPU does not need to copy data from one storage area to another, thereby reducing context switches and CPU copy time. In other words, zero copy primarily addresses the problem of the operating system frequently copying data when processing I/O operations. Common zero copy implementation technologies include: `mmap+write` ,, `sendfile` and `sendfile + DMA gather copy` .

The following figure shows a comparison of various zero-copy technologies:

	CPU copy	DMA copy	System calls	Context Switching
Traditional methods	2	2	read+write	4
mmap+write	1	2	mmap+write	4
sendfile	1	2	sendfile	2

	CPU copy	DMA copy	System calls	Context Switching
sendfile + DMA gather copy	0	2	sendfile	2

As can be seen, both traditional I/O methods and those using zero-copy require two DMA (Direct Memory Access) copies. This is because both DMA operations rely on hardware. Zero-copy primarily reduces CPU copies and context switches.

Java supports zero copy:

- `MappedByteBuffer` NIO `mmap` provides a zero-copy implementation based on memory mapping ( ). Under the hood, it actually invokes the Linux kernel's `mmap` system call. It can map a file or part of a file into memory, forming a virtual memory file. This allows direct access to data in memory without requiring system calls to read and write files.
- `FileChannel` This `transferTo()/transferFrom()` is an NIO `sendfile` implementation based on the zero-copy send file method (sendFile()). It actually uses a Linux kernel `sendfile` system call. It can send file data directly from disk to the network without going through a user-space buffer. For more information on `FileChannel` its usage, please refer to the article: Java NIO FileChannel Usage .

Code example:

```java
private void loadFileIntoMemory(File xmlFile) throws IOException {
  FileInputStream fis = new FileInputStream(xmlFile);
  //      FileChannel
  FileChannel fc = fis.getChannel();
  // FileChannel.map()                         MappedByteBuffer
  MappedByteBuffer mmb = fc.map(FileChannel.MapMode.READ_ONLY, 0,
fc.size());
  xmlFileBuffer = new byte[(int)fc.size()];
  mmb.get(xmlFileBuffer);
  fis.close();
}
```

Summarize

In this article, we mainly introduced the core knowledge points of NIO, including NIO's core components and zero copy.

If you need to use NIO to build a network application, we recommend against using native NIO directly, as programming is complex and functionality is limited. Instead, we recommend using a mature NIO-based network programming framework, such as Netty. Netty offers several optimizations and extensions based on NIO, including support for multiple protocols and SSL/TLS.

refer to

- A brief analysis of Java NIO: https://tech.meituan.com/2016/11/04/nio.html
- Interviewer: Do you know Java NIO? https://mp.weixin.qq.com/s/mZobf-U8OSYQfHfYBEB6KA
- Java NIO: Buffer, Channel and Selector: https://www.javadoop.com/post/java-nio

Recently Updated2024/7/17 16:02
Contributors: dandelionstar , Guide , Mr.Hope , aucub , FengWei2000 , Guevara