05 Java NIO的核心组件有哪些

更新时间：2020-07-14 15:23:07

> 只有在那崎岖的小路上不畏艰险奋勇攀登的人,才有希望达到光辉的顶点。——马克思

前言

你好，我是彤哥。

上一节我们一起学习了在 Java 中如何编写 BIO/NIO/AIO 的程序，并给出了最小化的示例代码，通过上一节的学习，相信你一定可以写出很优秀的网络应用程序了。并且，在最后我们用 NIO 写了一个简单的群聊系统，领悟了 NIO 编程的魅力。

不过，对于 Java NIO，你会发现，它不仅仅可以用在网络编程中，还能用在文件读写等其它场景，因此，有必要从根本上来了解 Java NIO。学习它的核心组件是一种最佳途径，能够使我们对 Java NIO 有一个完整的认识，而不仅仅局限于网络编程场景中。

所以，本节，我们将对 Java NIO 的核心组件做一个完整的剖析，带你从根上理解 Java NIO。

好了，让我们一起进入今天的学习吧。

Channel

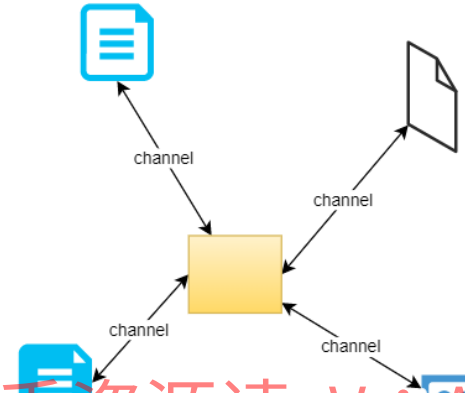
什么是 Channel 呢？ 让我们来看看 JDK 怎么说：

```
// java.nio.channels.Channel
```

> A nexus for I/O operations.
>
> A channel represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, for example reading or writing.

Channel 是一种 IO 操作的连接（nexus，连接的意思），它代表的是到实体的开放连接，这个实体可以是硬件设备、文件、网络套接字或者可执行 IO 操作（比如读、写）的程序组件。

在 linux 系统中，一切皆可看作是文件，所以，简单点讲，Channel 就是到文件的连接，并可以通过 IO 操作这些文件。



因此，针对不同的文件类型又衍生出了**不同类型的 Channel**：

- FileChannel：操作普通文件
- DatagramChannel：用于 UDP 协议
- SocketChannel：用于 TCP 协议，客户端与服务端之间的 Channel
- ServerSocketChannel：用于 TCP 协议，仅用于服务端的 Channel

> ServerSocketChannel 和 SocketChannel 是专门用于 TCP 协议中的。
> ServerSocketChannel 是一种服务端的 Channel，只能用在服务端，可以看作是到网卡的一种 Channel，它监听着网卡的某个端口。
> SocketChannel 是一种客户端与服务端之间的 Channel，客户端连接到服务器的网卡之后，被服务端的 Channel 监听到，然后与客户端之间建立一个 Channel，这个 Channel 就是 SocketChannel。

那么，这些 **Channel 又该如何使用呢？** 我们以 FileChannel 为代表来写一个简单的示例。

```
public class FileChannelTest {
    public static void main(String[] args) throws IOException {
        // 从文件获取一个FileChannel
        FileChannel fileChannel = new RandomAccessFile("D:\\object.txt", "rw").getChannel();
        // 声明一个Byte类型的Buffer
        ByteBuffer buffer = ByteBuffer.allocate(10);
        // 将FileChannel中的数据读出到buffer中，-1表示读取完毕
        // buffer默认为写模式
        // read()方法是相对channel而言的，相对buffer就是写
        while ((fileChannel.read(buffer)) != -1) {
            // buffer切换为读模式
            buffer.flip();
            // buffer中是否有未读数据
            while (buffer.hasRemaining()) {
                // 未读数据的长度
                int remain = buffer.remaining();
                // 声明一个字节数组
                byte[] bytes = new byte[remain];
                // 将buffer中数据读出到字节数组中
                buffer.get(bytes);
                // 打印出来
                System.out.println(new String(bytes, StandardCharsets.UTF_8));
            }
            // 清空buffer，为下一次写入数据做准备
            // clear()会将buffer再次切换为写模式
            buffer.clear();
        }
    }
}
```

通过上面的示例，我们可以发现，Channel 是和 Buffer 一起使用的，那么，什么是 Buffer 呢？为什么要和 Buffer 一起使用呢？必须吗？

Buffer

什么是 **Buffer** 呢？ 让我们再来看看 JDK 怎么说：

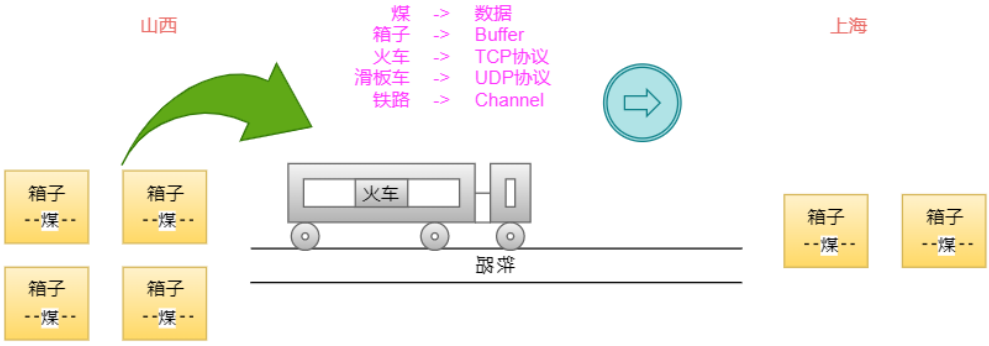
> // java.nio.Buffer
>
> A container for data of a specific primitive type.
>
> A buffer is a linear, finite sequence of elements of a specific primitive type. Aside from its content, the essential properties of a buffer are its capacity, limit, and position.

Buffer 是一个容器，什么样的容器呢？存放数据的容器。存放什么样的数据呢？特定基本类型的数据。这个容器有哪些特点呢？它是线性的，有限的序列，元素是某种基本类型的数据。它又有哪些属性呢？主要有三个属性：capacity、limit、position。

那么，**Buffer** 为什么要和 **Channel** 一起使用呢？必须一起使用吗？

打个比方，我们知道，山西盛产煤这种资源，一粒一粒的煤我们可以看作是数据。煤要往外运，那就需要修铁路，比如从山西运到上海，那就要修一条从山西到上海的铁路，这条铁路就相当于是连接山西和上海的通道（Channel）。数据和通道都有了，煤要放在哪里运过去呢？那就需要一种容器，有人可能会想到火车，其实火车可以看作是运输的一种方式或者叫协议，不使用火车，使用滑板车可不可以呢？其实也可以，只是运输的风险比较大而已，所以，火车可以看作是 TCP 协议，而滑板车是 UDP 协议。真正的容器应该是装煤的箱子，也就是 Buffer。

它们之间的关系如下图所示：



所以，Channel 和 Buffer 能不能单独使用呢？其实也可以，只是意义不大，比如，声明了一个 Channel 啥也不干，声明了一个 Buffer 里面存放一些数据啥也不干，意义不大，数据有交互才有意义，所以我们一般把 Channel 和 Buffer 一起使用，从 Channel 读取数据到 Buffer 中，或者从 Buffer 写入数据到 Channel 中。

细心的同学可能会发现，**NIO 的传输方式和传统的基于 BIO 的传输方式**基本是类似的，那么，它们**有什么区别呢？**

首先，BIO 是面向流的，而 NIO 是面向 Channel 或者面向缓冲区的，它的效率更高。

其次，流是单向的，所以又分成 InputStream 和 OutputStream，而 Channel 是双向的，既可读也可写。

然后，流只支持同步读写，而 Channel 是可以支持异步读写的。

最后，流一般与字节数组或者字符数组配合使用，而 Channel 一般与 Buffer 配合使用。

好了，通过前面的学习，我们知道了 Buffer 是一个容器，它里面存储的是特定的基本类型，那么，**有哪些类型的 Buffer 呢？**

我们知道基本类型有：byte、char、short、int、long、float、double、boolean，那么是不是每一种基本类型对应一种 Buffer 呢？嗯，基本上是这样，除了 boolean 没有对应的 Buffer 以外，其它的类型都有对应的 Buffer，因为 boolean 本质上就是 0 和 1 两种情况，Java 字节码层面也是用 0 和 1 来表示 boolean 类型的 false 和 true 的。

所以，Buffer 的类型有：ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。

上面是按照基本类型的角度来划分的，其实针对每一种类型还有不同的内存实现，分为**堆内存实现和直接内存实现**，比如，ByteBuffer 又分为 HeapByteBuffer 和 DirectByteBuffer 两种不同的内存实现，关于堆内存和直接内存的话题，我们后面再细说。

OK，Buffer 有这么多种不同的实现，那么，**它们该如何使用呢？**

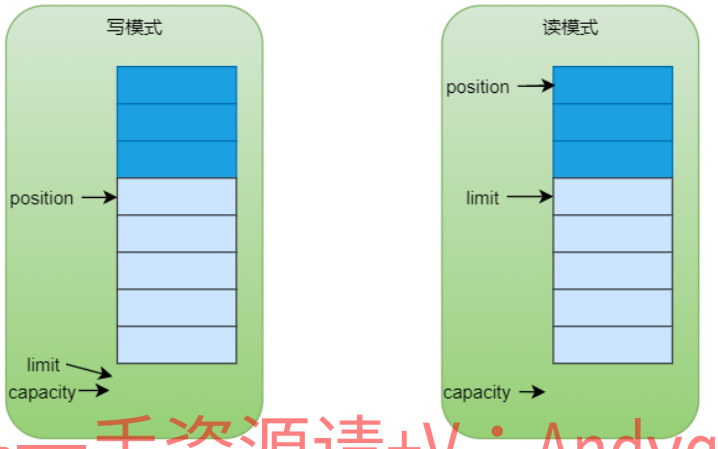
从前面 Buffer 的定义，我们知道 Buffer 有三个非常重要的属性，分别为：capacity、limit、position。

capacity，比较好理解，Buffer 的容量，即能够容纳多少数据。

limit，这个稍微费脑一些，表示的是最大可写或者最大可读的数据。

position，这个就更难理解一些，表示下一次可使用的位置，针对读模式表示下一个可读的位置，针对写模式表示下一个可写的位置。

上面的描述可能比较抽象，让我们上一张图来细细品味一下：



上图中，深蓝色表示已经写入的空间，这部分有数据可读；淡蓝色表示未被写入的空间，这部分没有数据可读。

写模式下，position 指向下一个可写的位置，limit 表示最大可写的数据，capacity 表示容量。比如，Buffer 的大小为 8，已经写了三个单位的数据，则 capacity=8，limit=8，position=3。

读模式下，position 指向下一个可读的位置，limit 表示最大可读的数据，capacity 表示容量。比如，Buffer 的大小为 8，已经写了三个单位的数据，此时切换为读模式，则 capacity=8，limit=3，position=0。

> 注意，position 表示的是位置，类似于数组的下标，是从 0 开始的。而 limit 和 capacity 表示的是大小，类似于数组的长度，是从 1 开始的。当 Buffer 从写模式切换为读模式时，limit 变为原 position 的值，position 变为 0。

好了，Buffer 的结构我们了解了，那么，要如何使用 Buffer 呢？

Buffer 提供了一系列的方法，供我们简单快捷地使用 Buffer，我们从使用的流程上来说的话，大概有下面这么**几个重要的方法：**

- 分配一个 Buffer：allocate ()
- 写入数据：buf.put () 或者 channel.read (buf)，read 为 read to 的意思，从 channel 读出并写入 buffer
- 切换为读模式：buf.flip ()
- 读取数据：buf.read () 或者 channel.write (buf)，write 为 write from 的意思，从 buffer 读出并写入 channel

- 重新读取或重新写入：rewind ()，重置 position 为 0，limit 和 capacity 保持不变，可以重新读取或重新写入数据
- 清空数据：buf.clear ()，清空所有数据
- 压缩数据：buf.compact ()，清除已读取的数据，并将未读取的数据往前移

用生活中的案例来解释的话，就如同饭店门口的排队策略。今天我们来到一家叫“椰子鸡”的餐厅，人爆满。服务员在门口依次摆了 10 张椅子（分配 Buffer），过来一对情侣，他们分别坐在 0 号和 1 号椅子上（position 从 0 开始，写入数据），过了一会儿，又过来一对情侣（别扎心老铁），他们紧跟着坐在了 2 号和 3 号椅子上（position 从 2 开始，写入数据），又过了一会儿，有一桌吃完了，且收拾完毕，0 号和 1 号椅子上的情侣可以出列了（切换为读模式，并读取数据），后面排队的人往前移（压缩数据 compact ()，清除已读取的数据，并将未读取的数据前移），我和女朋友一直排队到晚上 10 点，服务员过来说，我们要打烊了，明天再来吧，我的内心有一万只草尼玛跑过，无奈地离开了座位（清空数据 clear ()），带着女朋友去吃了肯德基。

好了，扯淡结束，让我们对照着代码来看看如何使用 Buffer。

```
public class FileChannelTest {
    public static void main(String[] args) throws IOException {
        // 从文件获取一个FileChannel
        FileChannel fileChannel = new RandomAccessFile("D:\\object.txt", "rw").getChannel();
        // 分配一个Byte类型的Buffer
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        // 将FileChannel中的数据读出到buffer中，-1表示读取完毕
        // buffer默认为写模式
        // read()方法是相对channel而言的，相对buffer就是写
        while ((fileChannel.read(buffer)) != -1) {
            // buffer切换为读模式
            buffer.flip();
            // buffer中是否有未读数据
            while (buffer.hasRemaining()) {
                // 读取数据
                System.out.print((char)buffer.get());
            }
            // 清空buffer，为下一次写入数据做准备
            // clear()会将buffer再次切换为写模式
            buffer.clear();
        }
    }
}
```


细心的同学会发现，跟上面 Channel 的例子基本上是一样的，是的了，因为 Channel 和 Buffer 要配合着一起使用嘛 ^^。

好了，介绍完了 Channel 和 Buffer，我们再来看看 NIO 的另一个核心组件 ——Selector，可以毫不夸张地说，没有 Selector 就无法使用 NIO 来进行网络编程，那么，**Selector 有哪些过人之处呢？**

Selecor

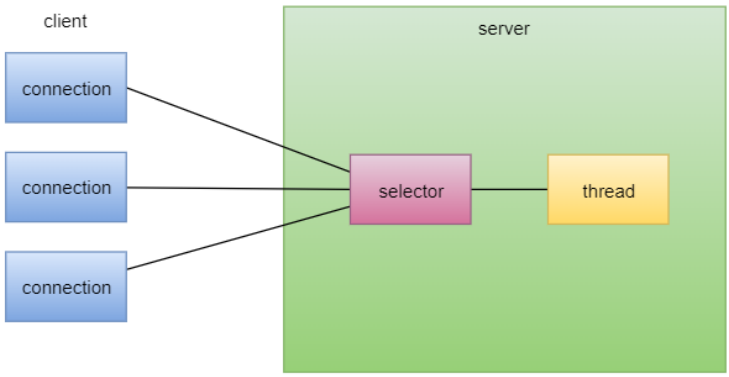
什么是 **Selector**？ 让我们再来看看 JDK 怎么说：

> // java.nio.channels.Selector
>
> A multiplexor of {@link SelectableChannel} objects.

首先，Selector 是一个多路复用器。什么的多路复用器？SelectableChannel 对象的多路复用器，注意，这里的对象是复数，说明一个 Selector 可以关联到多个 SelectableChannel。另外，它是 **SelectableChannel** 的多路复用器，可以跟 FileChannel 配合使用吗？不可以，因为 FileChannel 不是 SelectableChannel。那么 SelectableChannel 有哪些呢？跟网络编程相关的那些 Channel 基本上都是 SelectableChannel，比如 SocketChannel、ServerSocketChannel、DatagramChannel 等。

那么，**Selector 跟 Channel 究竟是怎样的关系呢？**

从上面的描述中，我们也能够大胆猜测，Selector 和 Channel 是一对多的关系，一个 Selector 可以为多个 Channel 服务，监听它们准备好的事件。Selector 就像饭店中的服务员一样，一个服务员是可以服务于多位顾客的，时刻监听着顾客的吩咐。



OK，那么，**Selector** 又该怎么使用呢？我们还是以饭店来举例。

首先，饭店需要先聘请一个服务员，然后这个服务员来上班。同样地，Selector 也需要先创建出来，创建的方式有两种，一种是调用 `Selector.open()` 方法，一种是调用 `SelectorProvider.openSelector()` 方法，其中 SelectorProvider 是自定义的。鉴于第二种方式不太常用，所以我们只讲第一种方式。

```
// 创建一个Selector
Selector selector = Selector.open();
```

接着，有顾客上门了，服务员去接待，“吃啥啊兄弟？”，“小炒黄牛肉。”，顾客把点的菜告诉服务员。同样地，你需要 Selector 干什么，也需要告诉他，在 Java 里面，我们叫作注册事件到 Selector 上，当然了，我们是非阻塞式的，所以，注册之前还要先设置为非阻塞式。

```
// 注册事件到Selector上
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

Channel 注册到 Selector 上之后，返回了一个叫作 SelectionKey 的对象，**SelectionKey** 又是什么呢？

一般地，我们在饭店点完菜之后，都会给一个牌子到你手上，服务员通过这个牌子可以找到你，你通过这个牌子可以去拿饭。SelectionKey 就相当于是这个牌子，它将 Channel 和 Selector 的牢牢地绑定在一起，并保存着你感兴趣的事件。

事件又是什么东西？

事件是 Channel 感兴趣的事情，比如读事件、写事件等等，在 Java 中，定义了四种事件，位于 SelectionKey 这个类中：

- 读事件：SelectionKey.OP_READ = 1 << 0 = 0000 0001
- 写事件：SelectionKey.OP_WRITE = 1 << 2 = 0000 0100
- 连接事件：SelectionKey.OP_CONNECT = 1 << 3 = 0000 1000
- 接受连接事件：SelectionKey.OP_ACCEPT = 1 << 4 = 0001 0000

细心的同学会发现，四种事件的位正好是错开的，所以，我们可以使用“位或”操作监听多种感兴趣的事件：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

然后，服务员拿到了好几个顾客的菜单后，不断地去后厨询问，看看有没有做好的，有做好的就通知到这些顾客。在 Java 中，这叫作轮询，“轮”是一次又一次的意思，包含循环的含义。

```
// select()只有询问的意思，加上循环才是轮询的意思
while(true) {
    selector.select(); // 一直阻塞直到有感兴趣的事件
    // selector.selectNow(); // 立即返回，不阻塞
    // selector.select(timeout); // 阻塞一段时间返回
    // ...
}
```

与服务员不断地询问后厨不同的是，服务员没问到结果，可能就去忙其它的事了，比如玩手机，而 select () 没询问到结果会一直阻塞着，直到有感兴趣的事件来了为止。当然，你也可以使用的它的兄弟方法 selectNow() 不阻塞立即返回，或者 select(timeout) 阻塞一段时间后返回。

最后，服务员拿到这些做好的菜单，通知顾客自己过来聚餐（这个服务员比较懒）。在 Java 中，通过 selector.selectedKeys() 返回就绪的事件，然后遍历这些事件就可以拿到想要的结果。

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // 接受连接事件已就绪
    } else if (key.isConnectable()) {
        // 连接事件已就绪
    } else if (key.isReadable()) {
        // 读事件已就绪
    } else if (key.isWritable()) {
        // 写事件已就绪
    }
    keyIterator.remove();
}
```

剩下的就是从 Channel 中取数据了，也就是 Channel 和 Buffer 的交互了，在上面已经介绍过了，这里我们就不重复介绍了。

Channel、Buffer、Selector 三者如何联合使用

Channel、Buffer、Selector 这三个 NIO 的核心组件我们都剖析完了，那么，它们该如何联合起来使用呢？让我们看一个完整的案例：

```
public class NIOEchoServer {
    public static void main(String[] args) throws IOException {
        // 创建一个Selector
        Selector selector = Selector.open();
        // 创建ServerSocketChannel
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        // 绑定8080端口
        serverSocketChannel.bind(new InetSocketAddress(8002));
        // 设置为非阻塞模式
        serverSocketChannel.configureBlocking(false);
        // 将Channel注册到selector上，并注册Accept事件
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        System.out.println("server start");

        while (true) {
            // 阻塞在select上（第一阶段阻塞）
            selector.select();

            // 如果使用的是select(timeout)或selectNow()需要判断返回值是否大于0

            // 有就绪的Channel
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            // 遍历selectKeys
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                // 如果是accept事件
                if (selectionKey.isAcceptable()) {
                    // 强制转换为ServerSocketChannel
                    ServerSocketChannel ssc = (ServerSocketChannel) selectionKey.channel();
                    SocketChannel socketChannel = ssc.accept();
                    System.out.println("accept new conn: " + socketChannel.getRemoteAddress());
                    socketChannel.configureBlocking(false);
                    // 将SocketChannel注册到Selector上，并注册读事件
                    socketChannel.register(selector, SelectionKey.OP_READ);
                } else if (selectionKey.isReadable()) {
                    // 如果是读取事件
                    // 强制转换为SocketChannel
                    SocketChannel socketChannel = (SocketChannel) selectionKey.channel();
                    // 创建Buffer用于读取数据
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    // 将数据读入到buffer中（第二阶段阻塞）
                    int length = socketChannel.read(buffer);
                    if (length > 0) {
                        buffer.flip();
                        byte[] bytes = new byte[buffer.remaining()];
                        // 将数据读入到byte数组中
                        buffer.get(bytes);

                        // 换行符会跟着消息一起传过来
                        String content = new String(bytes, "UTF-8").replace("\r\n", "");
                        System.out.println("receive msg: " + content);
                    }
                }
                iterator.remove();
            }
        }
    }
}
```

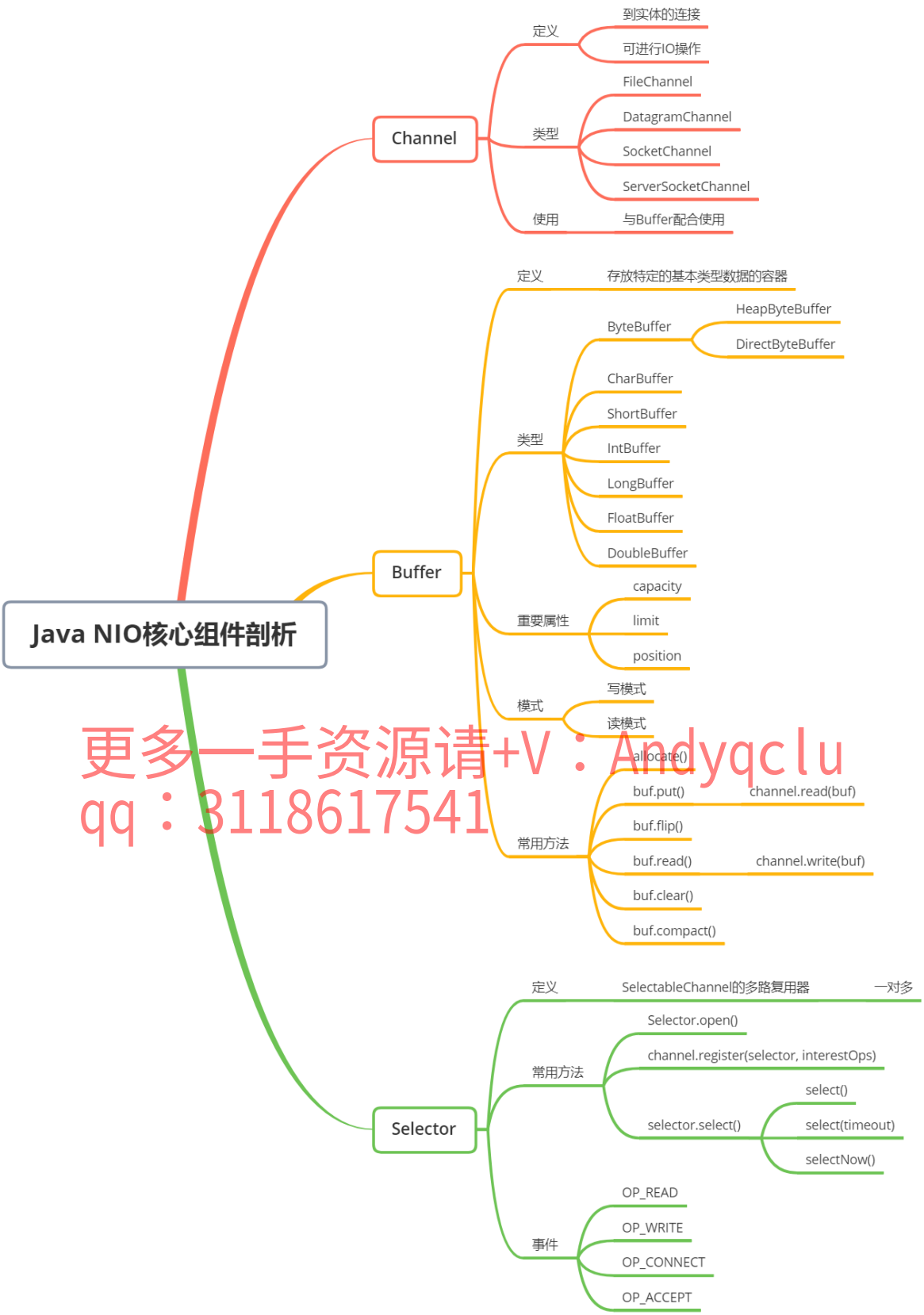

是不是很简单？三者一起组成了 Java NIO 网络编程的利器。

后记

本节，我们一起学习了 Java NIO 的三大核心组件 ——Channel、Buffer、Selector。NIO 不仅仅指代网络编程，还可以进行文件操作等，而 Channel 和 Buffer 一般是联合起来使用，共同进行 NIO 操作。再加上 Selector，就形成了 NIO 网络编程的利器。

通过这几节的学习，不知道你有没有发现，其实 NIO 编程也是蛮简单的嘛，使用 Java NIO 似乎也能写出高性能的网络应用，那么，我们为什么还要学习 Netty 呢？它又有什么过人之处呢？从下一节开始我们将全面进入 Netty 的学习进程，等你来找答案哦。

思维导图



}