# Java NIO简介（已整理）

## 1.1 摘要

java NIO (New IO，或Non Blocking IO) 是从Java1.4 版本开始引入的一个新的IO API，可以代替标准的Java IO API。

NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同。

NIO支持面向缓冲区、基于通道的IO操作。NIO将以更加高效的方式进行文件的读写操作。

## 1.2 NIO与IO 的主要区别



面向流指的是输入输出是基于“流”的概念，而且，流是单向的，不管输入还是输出都必须使用对应的输入流和输出流。流的本身也是数据在传输当中的一种抽象形式。

在NIO 中有通道（channel）的概念，是输出端和输入端之间的桥梁。虽然流也有类似的含义，但是不同的是，通道仅仅作为连接，本身不包含任何数据。与之进行搭配的就是缓冲区（Buffer）。如果把通道理解为铁路，那么缓冲区就可以是火车。NIO中的“通道 + 缓冲区”的组合可以完成双向传输的目的。

阻塞与非阻塞、选择器都是针对网络通信而言的。后面在网络通讯的时候会详细讲解。

## 1.3 通道和缓冲区

Java NIO 系统的核心在于：通道和缓冲区。

通道表示打开到IO 设备（文件、套接字）的连接。若需要使用NIO 系统，需要就必须要获取连接 IO 设备的通道以及用于容纳数据的缓冲区，然后操作缓冲区，对数据进行操作。

简而言之，Channel 负责传输， Buffer 负责存储。

# 二、缓冲区（已整理）

一个用于特定基本数据类型的容器。由java.nio 包定义，所有缓冲区都是 Buffer 抽象类的子类。

Java NIO 中的Buffer 主要用于与NIO 通道进行交互。数据从通道存入缓冲区，从缓冲区取出到通道中。

## 2.1 创建缓冲区

缓冲区的本质是 **数组** ，用于存储不同类型的数据，根据数据类型（boolean 除外），提供了相应类型的缓冲区，如ByteBuffer、IntBuffer等。这些缓冲区的管理方式都是类似的，都是通过 **allocate()** 方法获创建缓冲区。



## 2.2 缓冲区的核心方法

### 2.2.1 存取数据

put() ：存入数据到缓冲区；

get() ：从缓冲区中取出数据。

### 2.2.2 flip()、rewind()、clear()

flip() : 翻转，将缓冲区进行读写切换。

rewind() : 倒带，可以将position 和 limit 回退到上一次操作前。

clear() ： 清空缓冲区，官方说明是“clears the buffer”，但详细解释是将 position 和 limit 恢复出厂设置，并丢弃 mark。注意，缓冲区中的数据并非清空，只是将两个指针重置，数据处在一种“被遗忘”状态，如果进行 get()操作依然可以取出。同时，clear 执行之后的缓冲区无法通过 rewind() 回退指针。

1. @Test
2. **public** **void** test1() {
3. // 分配 1 KB 大小的缓冲区
4. ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
5. System.out.println("=============allocate()===========");
6. System.out.println("position = " + byteBuffer.position());
7. System.out.println("limit = " + byteBuffer.limit());
8. System.out.println("capacity = " + byteBuffer.capacity());
10. // 写数据
11. System.out.println("=============put()===========");
12. String name = "Morty";
13. byteBuffer.put(name.getBytes());
14. System.out.println("position = " + byteBuffer.position());
15. System.out.println("limit = " + byteBuffer.limit());
16. System.out.println("capacity = " + byteBuffer.capacity());
17. // 翻转缓冲区
18. System.out.println("============flip()===========");
19. byteBuffer.flip();
20. System.out.println("position = " + byteBuffer.position());
21. System.out.println("limit = " + byteBuffer.limit());
22. System.out.println("capacity = " + byteBuffer.capacity());
23. // 读数据
24. System.out.println("============get()===========");
25. **byte**[] dst = **new** **byte**[byteBuffer.limit()];
26. byteBuffer.get(dst);
27. System.out.println("position = " + byteBuffer.position());
28. System.out.println("limit = " + byteBuffer.limit());
29. System.out.println("capacity = " + byteBuffer.capacity());
30. System.out.println(**new** String(dst));
32. // 倒带
33. System.out.println("============rewind()===========");
34. byteBuffer.rewind();
35. System.out.println("position = " + byteBuffer.position());
36. System.out.println("limit = " + byteBuffer.limit());
37. System.out.println("capacity = " + byteBuffer.capacity());
38. **byte**[] dst1 = **new** **byte**[byteBuffer.limit()];
39. byteBuffer.get(dst1);
40. System.out.println(**new** String(dst1));
42. // 清空，但缓冲区中的数据依然存在，只不过处于一种“被遗忘”状态
43. System.out.println("============clear()===========");
44. byteBuffer.clear();
45. System.out.println("position = " + byteBuffer.position());
46. System.out.println("limit = " + byteBuffer.limit());
47. System.out.println("capacity = " + byteBuffer.capacity());
48. System.out.println("--------");
49. byteBuffer.rewind();
50. System.out.println("position = " + byteBuffer.position());
51. System.out.println("limit = " + byteBuffer.limit());
52. System.out.println("capacity = " + byteBuffer.capacity());
53. **byte**[] dst2 = **new** **byte**[byteBuffer.limit()];
54. byteBuffer.get(dst2);
55. System.out.println(**new** String(dst2));
56. }

### 2.2.3 mark() 与 reset()

mark()方法可以记录当前 position 的位置，并可以通过 reset() 方法恢复到 mark() 的位置。

1. @Test
2. **public** **void** test2() {
3. String str = "abcde";
5. ByteBuffer buf = ByteBuffer.allocate(1024);
7. buf.put(str.getBytes());
9. buf.flip();
11. **byte**[] dst = **new** **byte**[buf.limit()];
12. buf.get(dst, 0, 2);
13. System.out.println(**new** String(dst));
15. System.out.println(buf.position());
17. // mark()标记
18. buf.mark();
20. buf.get(dst, 2, 2);
21. System.out.println(**new** String(dst));
22. System.out.println(buf.position());
23. // reset() 恢复到 mark 的位置
24. buf.reset();
25. System.out.println(buf.position());
27. }

### 2.2.4 其他方法

remaining()方法可以返回剩余可操作的元素个数。

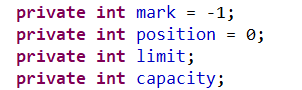
1. **if** (buf.hasRemaining()) {
2. System.out.println(buf.remaining());
3. System.out.println("limit - position = " + (buf.limit() - buf.position()));
4. }

输出：

1. 3
2. limit - position = 3

## 2.3 缓冲区中的 4 个核心属性

mark、position、limit、capacity



**capacity**：缓冲区容量，因为缓冲区本身是数组，因此，容量一旦声明无法改变。

**limit**：界限，表示缓冲区中可以操作数据的大小，limit 后的数据不能进行读写。

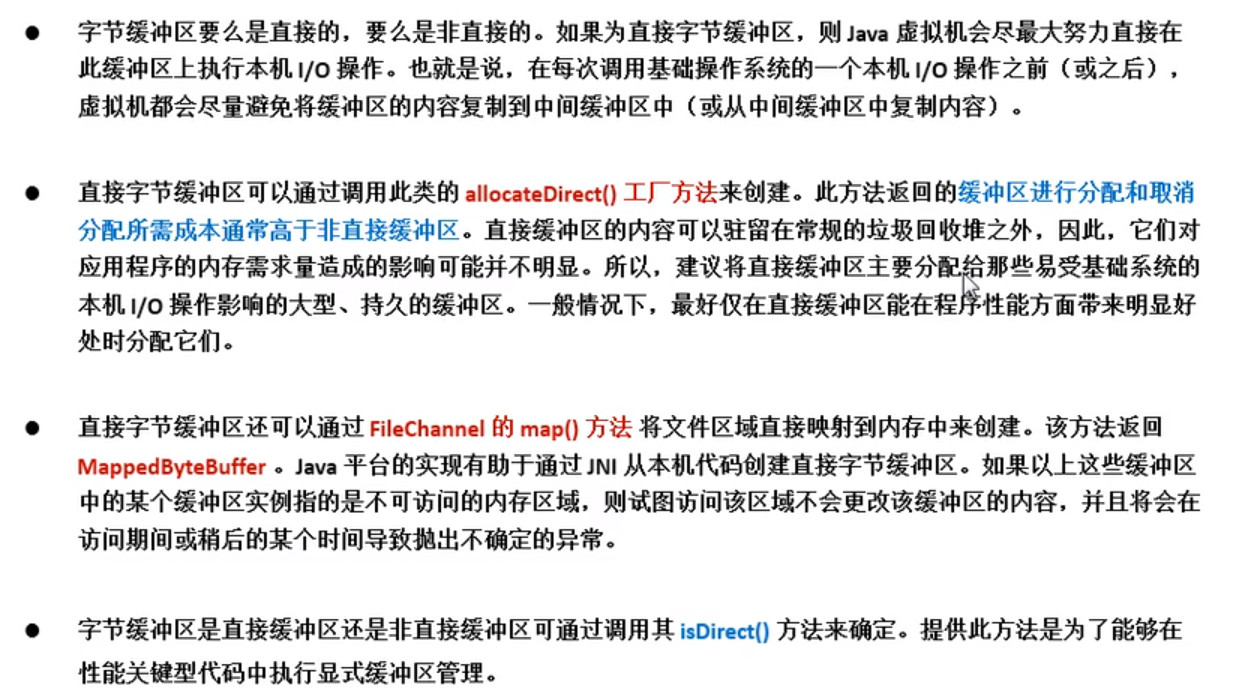
**position**：位置，正在操作的数据位置。

**position <= limit <= capacity**

**mark** ：标记，表示记录当前 position 的位置。可以通过 reset() 方法恢复到 mark 的位置。

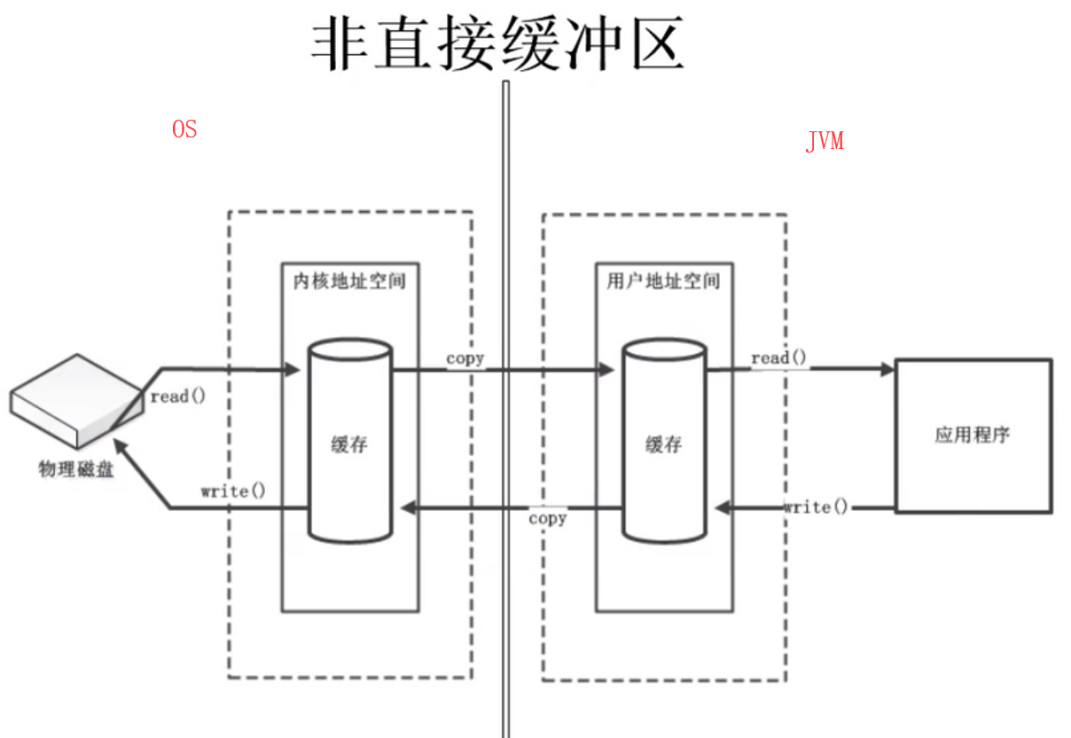
**0 <= mark <= position <= limit <= capacity**

## 2.4 直接缓冲区与非直接缓冲区



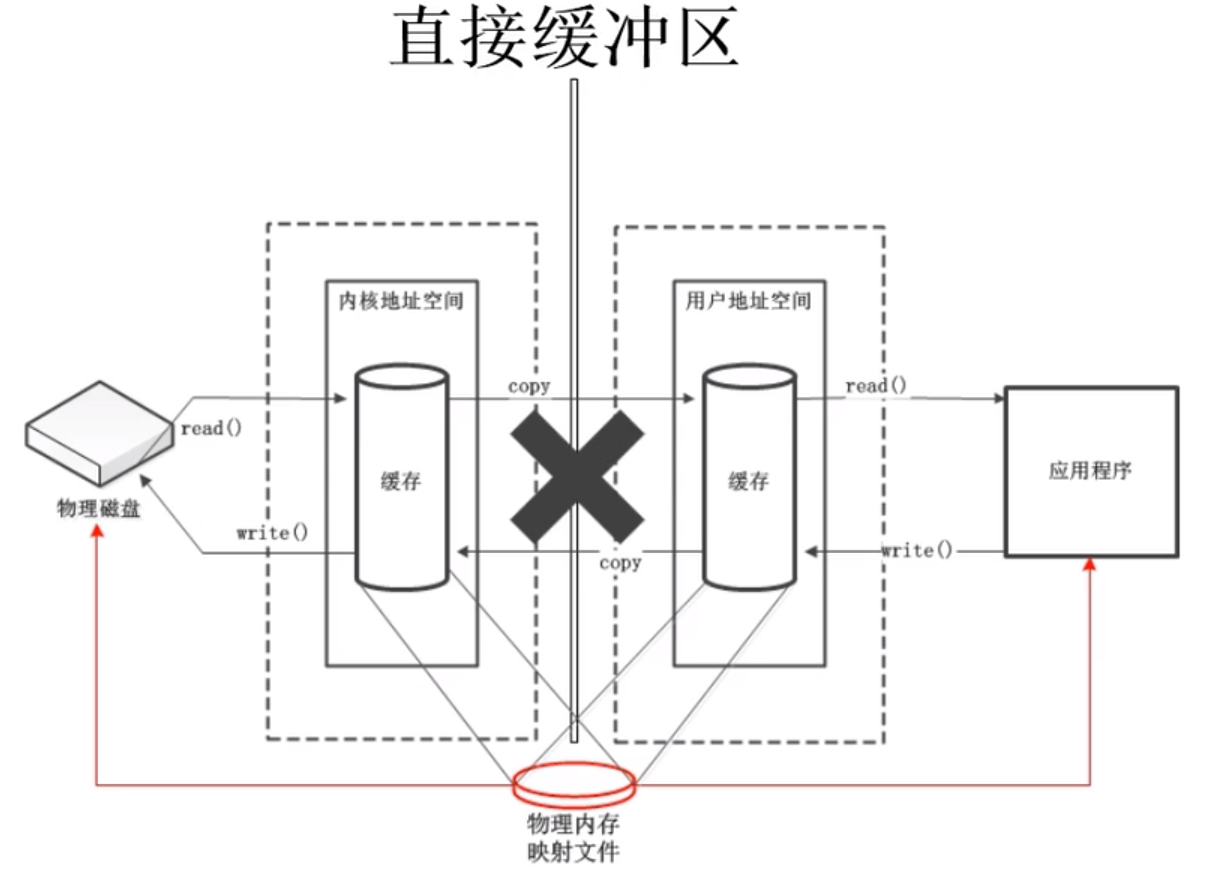
### 2.4.1 非直接缓冲区图解

传统的IO流和allocate()方法分配的缓冲区，都是下面这样的形式进行文件的传输。



### 2.4.2 直接缓冲区图解

直接缓冲区建立在物理内存中，省略了copy的过程，性能很好，但同时也存在一些致命的弊端：分配和销毁成本高、不易控制等。另外，直接缓冲区只有ByteBuffer支持。



### 2.4.3 创建两种缓冲区的方法

非直接缓冲区通过allocate()方法获得，缓冲区建立在 JVM 内存中。

直接缓冲区通过allocateDirect() 方法获得，缓冲区建立在物理内存中。提高效率。

判断一个缓冲区是否是为直接缓冲区，可以使用isDirect()方法，如果是true，就代表是直接缓冲区。

# 三、通道

通道：Channel，由java.nio.channels包定义，表示IO源与目标打开的连接。Channel类似于传统的“流”，只不过Channel本身不能直接访问数据，只能与Buffer 进行交互。

## 3.1 通道的主要实现类

java.nio.channels.Channel 接口：

FileChannel、SocketChannel、ServerSocketChannel、DatagramChannel等。

## 3.2 获取通道

在JDK1.7 后，通道的获取方式可以有三种。

1. getChannel()：Java针对支持通道的类提供了getChannel()方法，这些类包括：

本地IO：FileInputStream/FileOutputStream、RandomAccessFile；

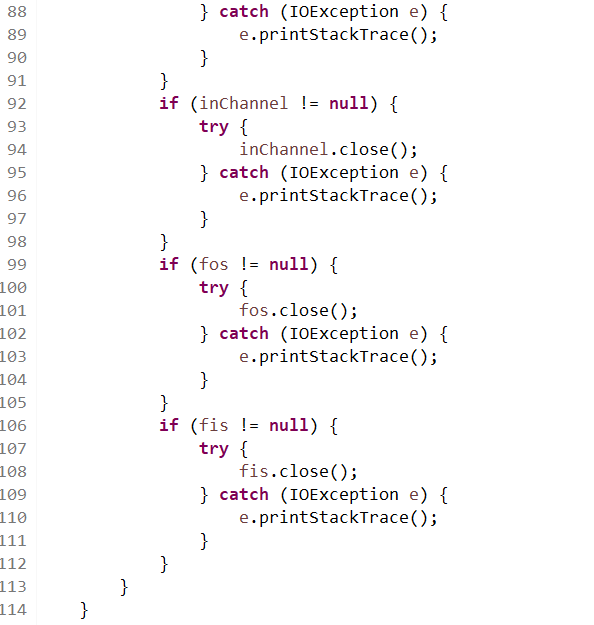
网络IO：Socket、ServerSocket、DatagramSocket

1. 在jdk1.7 中NIO.2，针对各个通道提供了静态方法open()。
2. 在jdk1.7 中NIO.2的Files工具类的 newByteChannel()。

## 3.3 案例分析-利用通道完成文件的复制

### 3.3.1 利用通道完成文件的复制（非直接缓冲区）



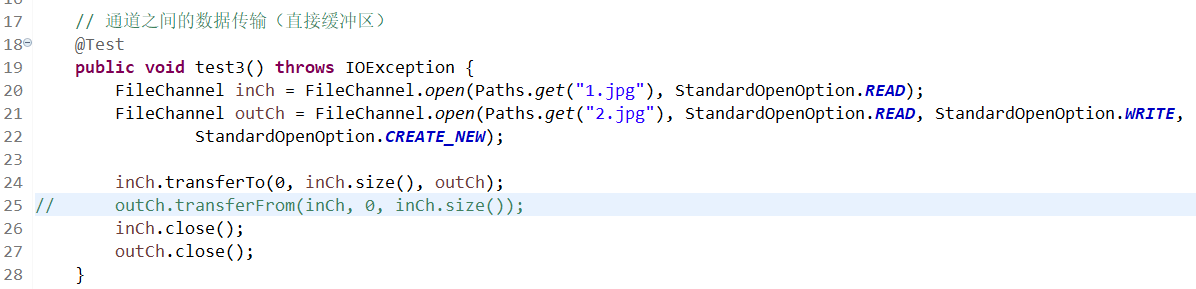


### 3.3.2 利用直接缓冲区完成文件的复制（内存映射文件）



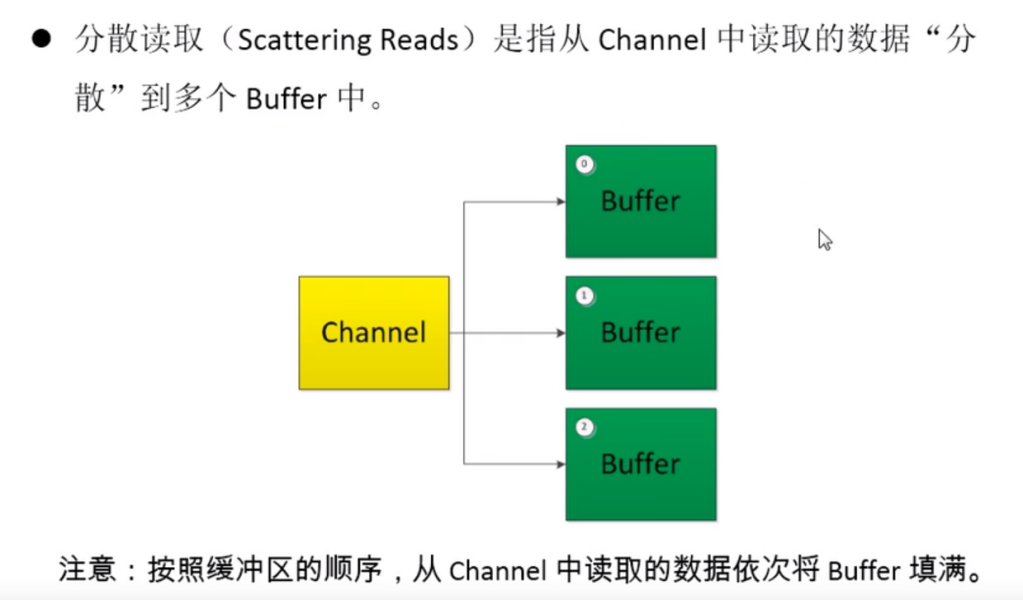
### 3.3.3 通道之间的数据传输

transferFrom()或transferTo()

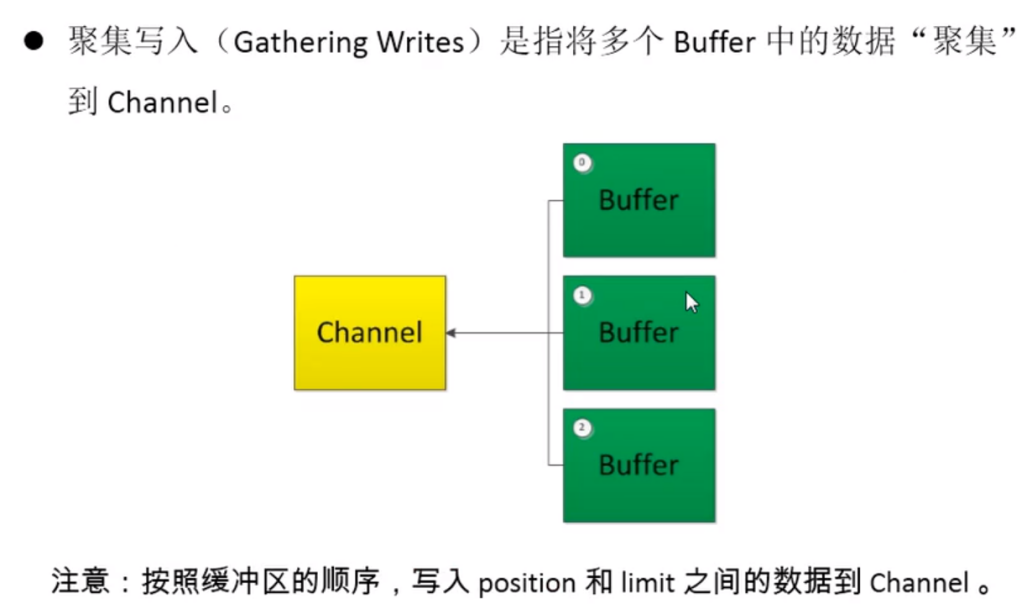


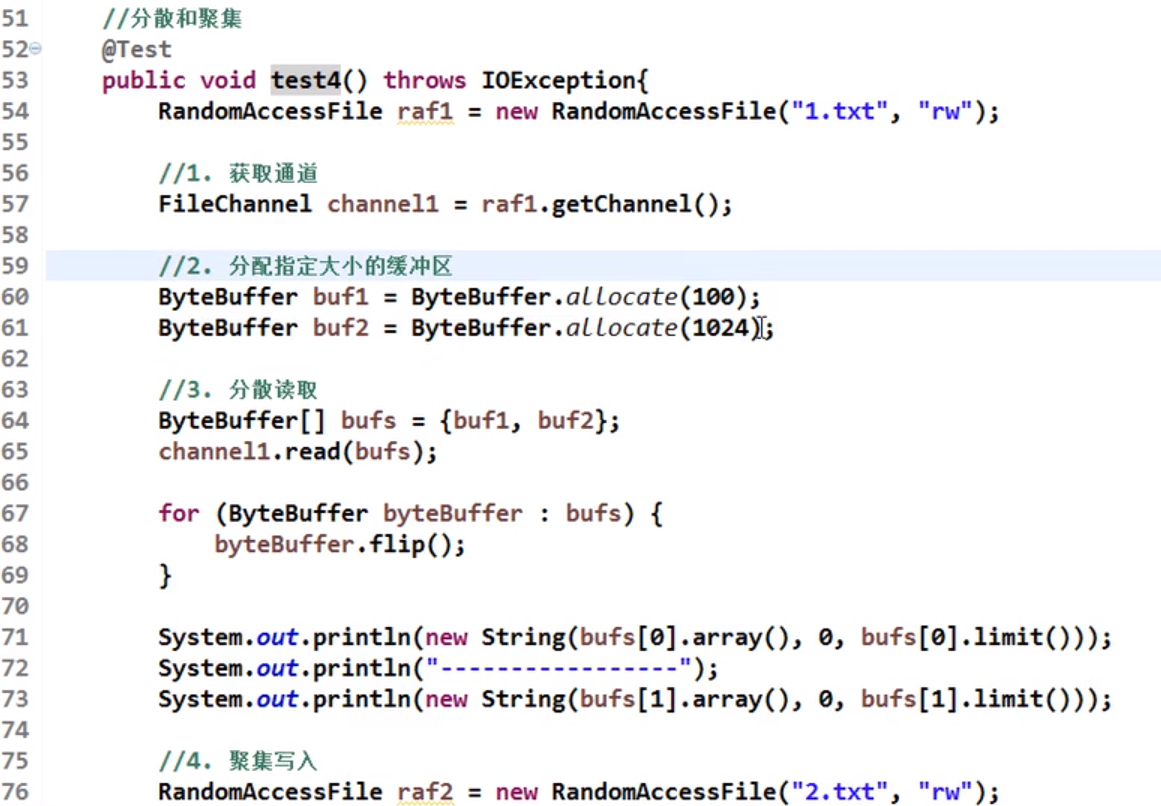
## 3.4 分散（Scatter）与聚集（Gather）

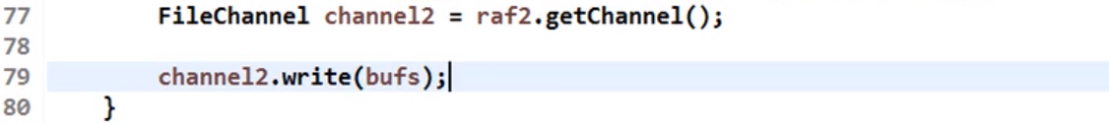
分散读取（Scattering Reads）：将通道中的数据分散到多个缓冲区中。



聚集写入（Gathering Writes）：将多个缓冲区中的数据聚集到通道中。





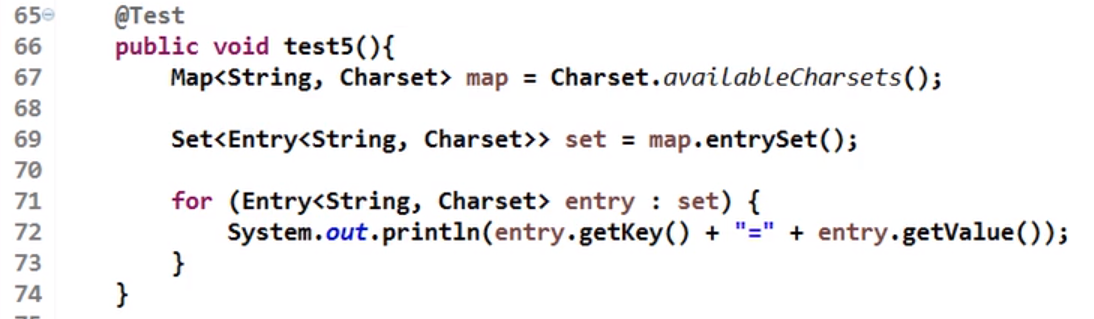


## 3.5 字符集：Charset

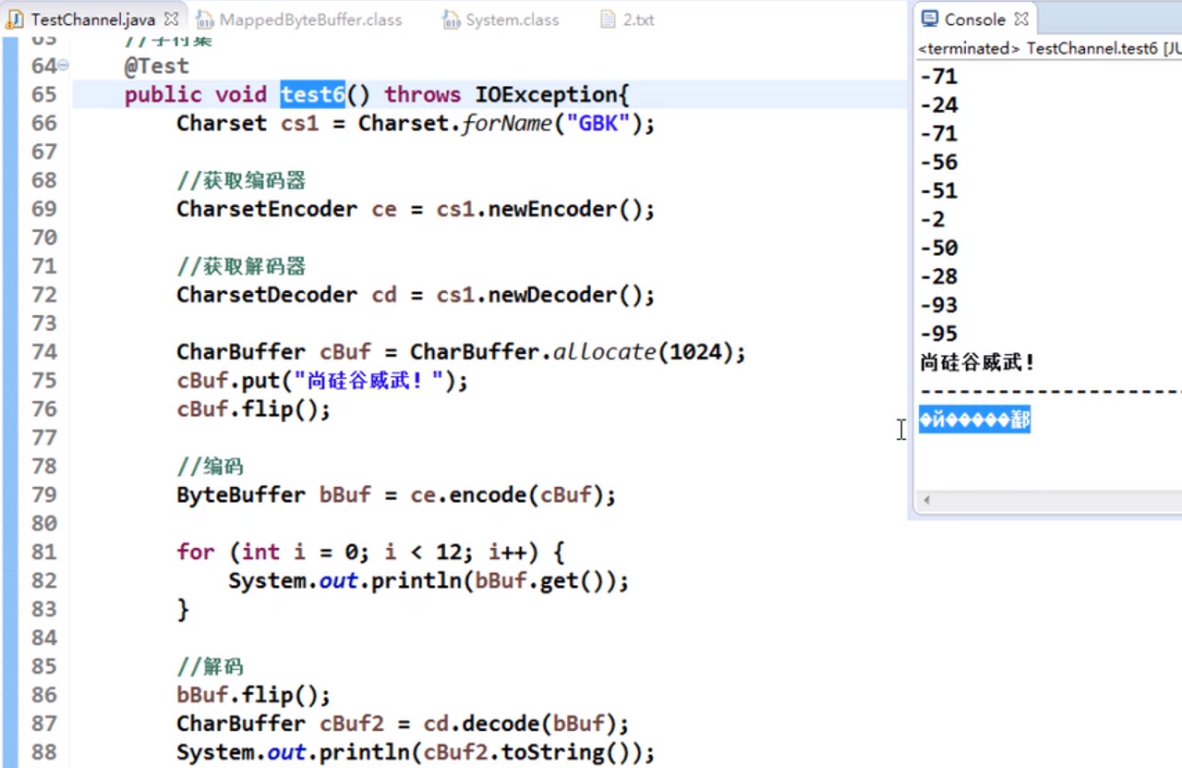
编码：字符串 -> 字节数组

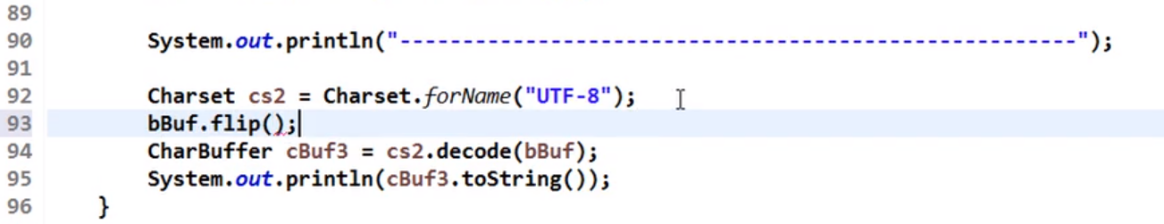
解码：字节数组 -> 字符串

**查看字符集种类**：



**编码与解码演示：**

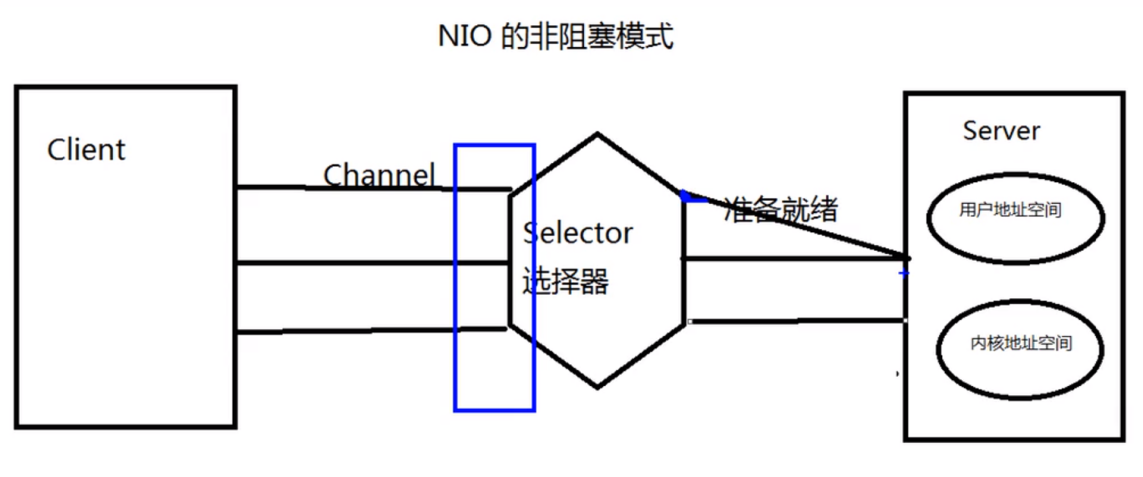




# 四、NIO的非阻塞式网络通信

<https://www.bilibili.com/video/av27062187/?p=8>

所谓的阻塞与非阻塞都是针对于网络通讯而言的。



## 4.1 利用NIO 完成一个网络通信的模型

使用 NIO 完成网络通信的三个核心组件：

1. 通道：负责连接

java.nio.channels.Channel 接口

|-- SelectableChannel

|-- SocketChannel

|-- ServerSocketChannel

|-- DatagramChannel

|-- Pipe.SinkChannel

|-- Pipe.SourceChannel

1. 缓冲区：负责数据的存取
2. 选择器（Selector）：是 SelectableChannel 的多路复用器。用于监控 SelectableChannel 的IO 状况。

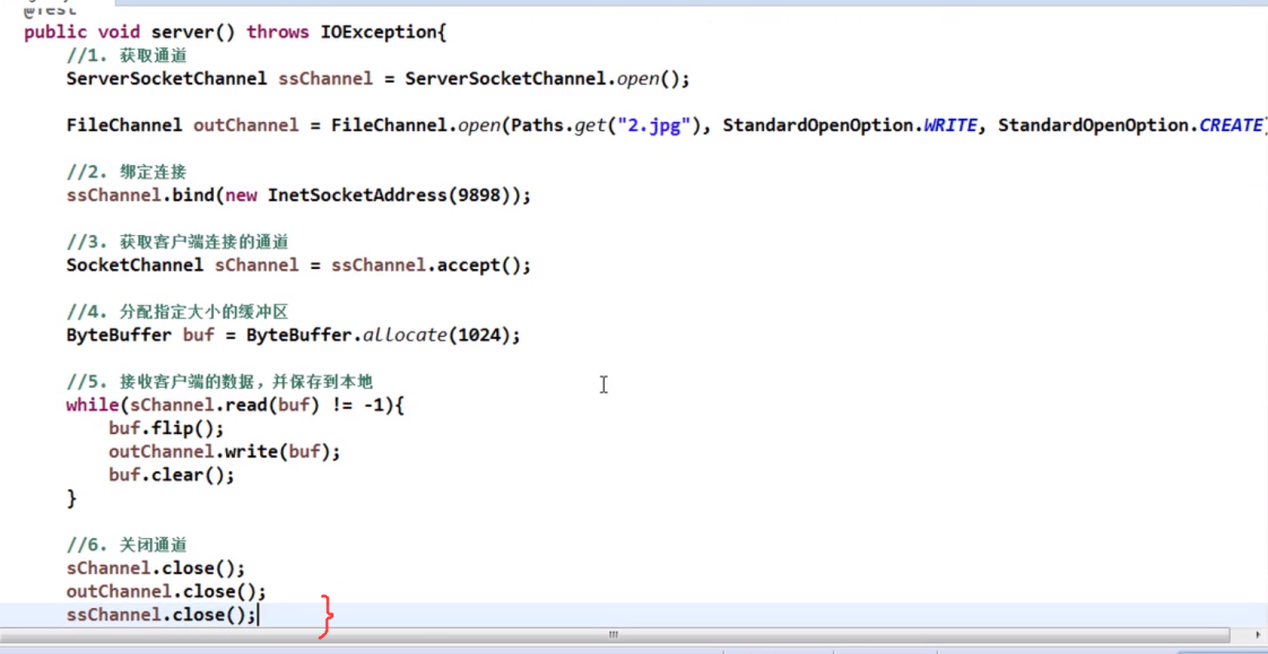
### 4.1.1 NIO的传统阻塞式网络通信

https://www.bilibili.com/video/av27062187/?p=9

客户端：



服务端：



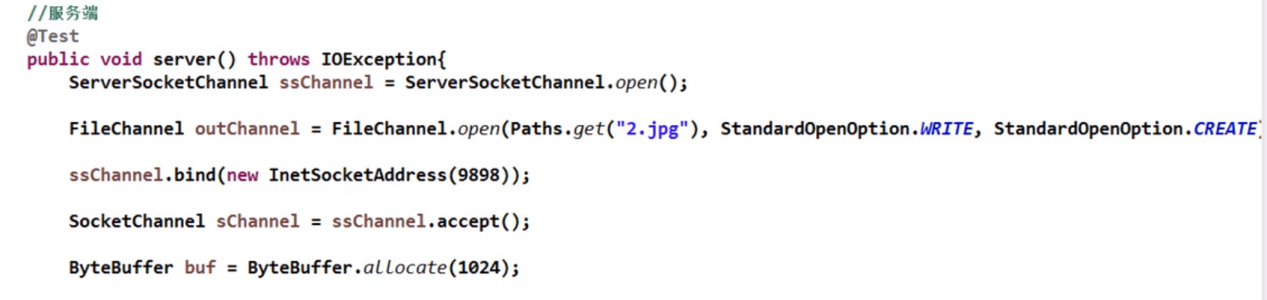
### 4.1.2 阻塞式通信2：成功消息反馈

https://www.bilibili.com/video/av27062187/?p=9

客户端：



服务端：



### 4.1.3 非阻塞式NIO

<https://www.bilibili.com/video/av27062187/?p=10>



