

## JDK10都发布了，nio你了解多少？

笔记本： 微信

创建时间： 2018/8/3 9:04

标签： 微信

URL: [http://mp.weixin.qq.com/s?\\_\\_biz=MzI4Njg5MDA5NA==&mid=2247484235&idx=1&sn=4c3b6d13335245d4de18...](http://mp.weixin.qq.com/s?__biz=MzI4Njg5MDA5NA==&mid=2247484235&idx=1&sn=4c3b6d13335245d4de18...)

## JDK10都发布了，nio你了解多少？

原创： [Java3y](#) [Java3y](#)

### 前言

只有光头才能变强

回顾前面：

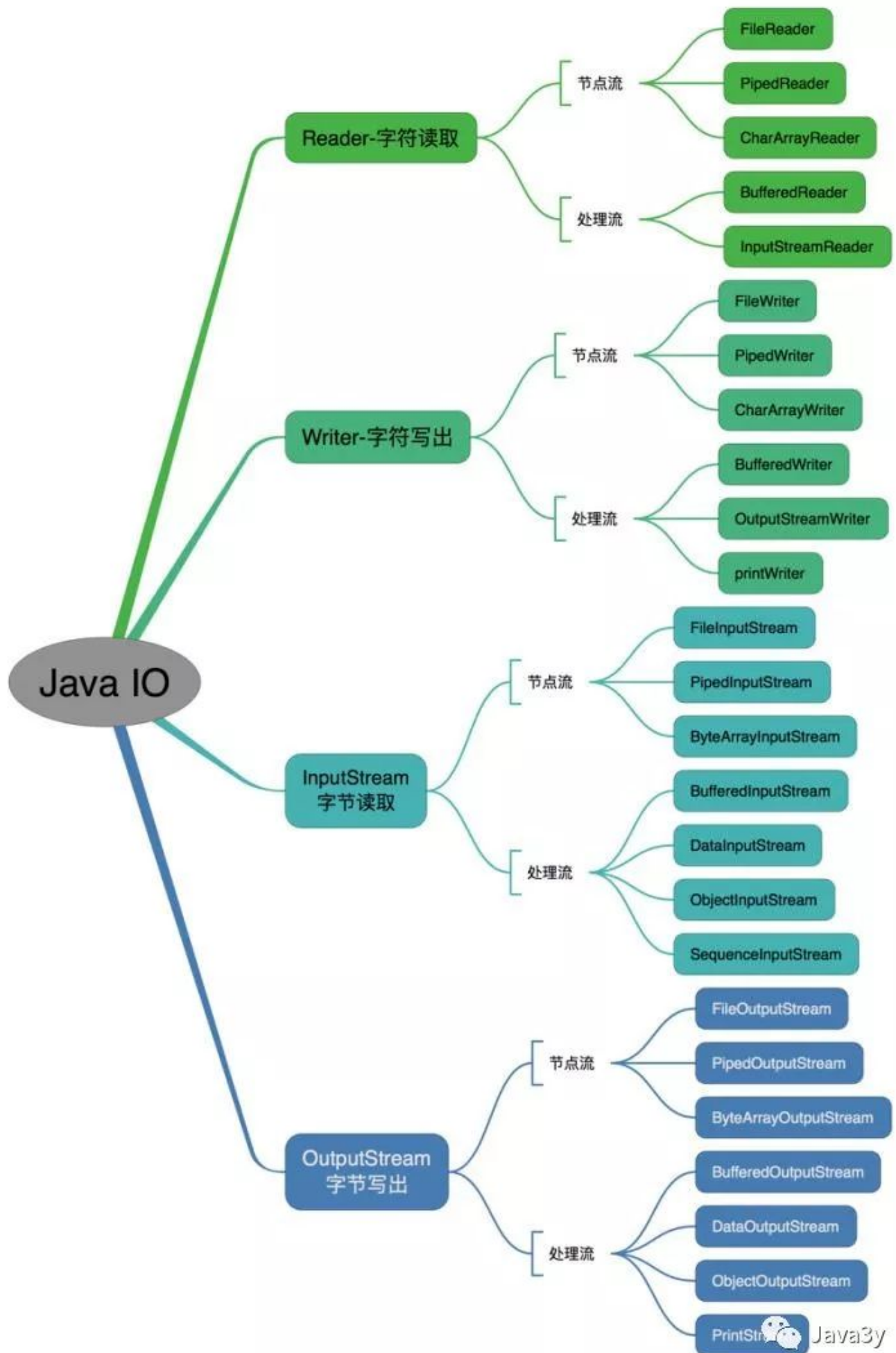
- [给女朋友讲解什么是代理模式](#)
- [包装模式就是这么简单啦](#)

本来我预想是先来回顾一下传统的IO模式的，将传统的IO模式的相关类理清楚(因为IO的类很多)。

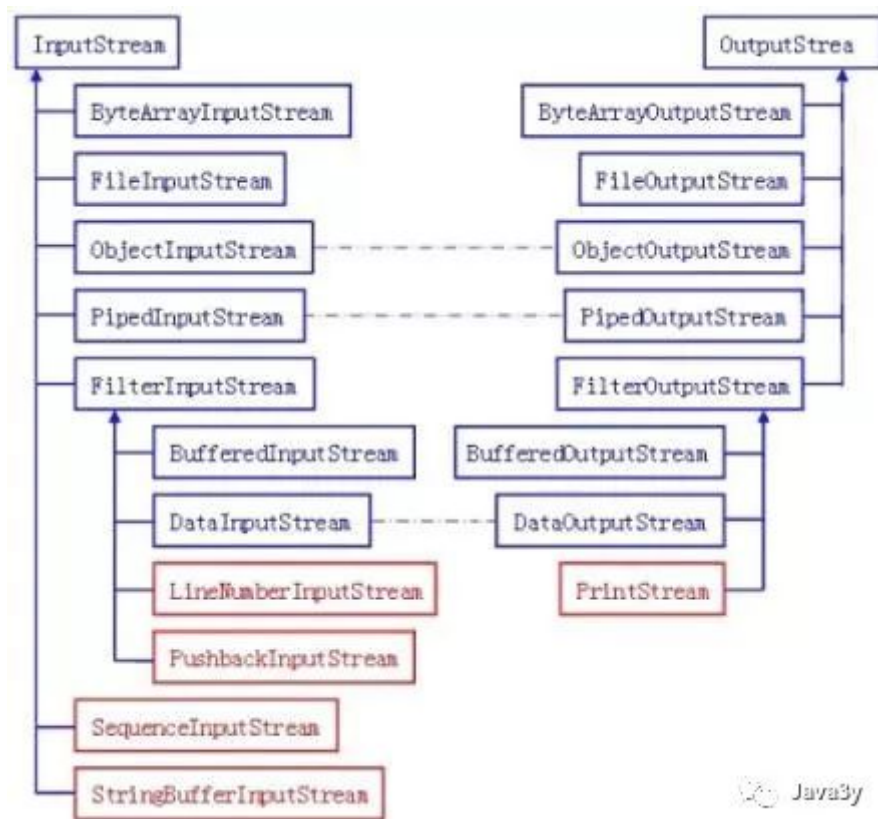
但是，发现在整理的过程已经有很多优秀的文章了，而我自己来整理的话可能达不到他们的水平。并且传统的IO估计大家都会用，而NIO就不一定了。

下面我就贴几张我认为整理比较优秀的思维导图(下面会给出图片来源地址，大家可前往阅读)：

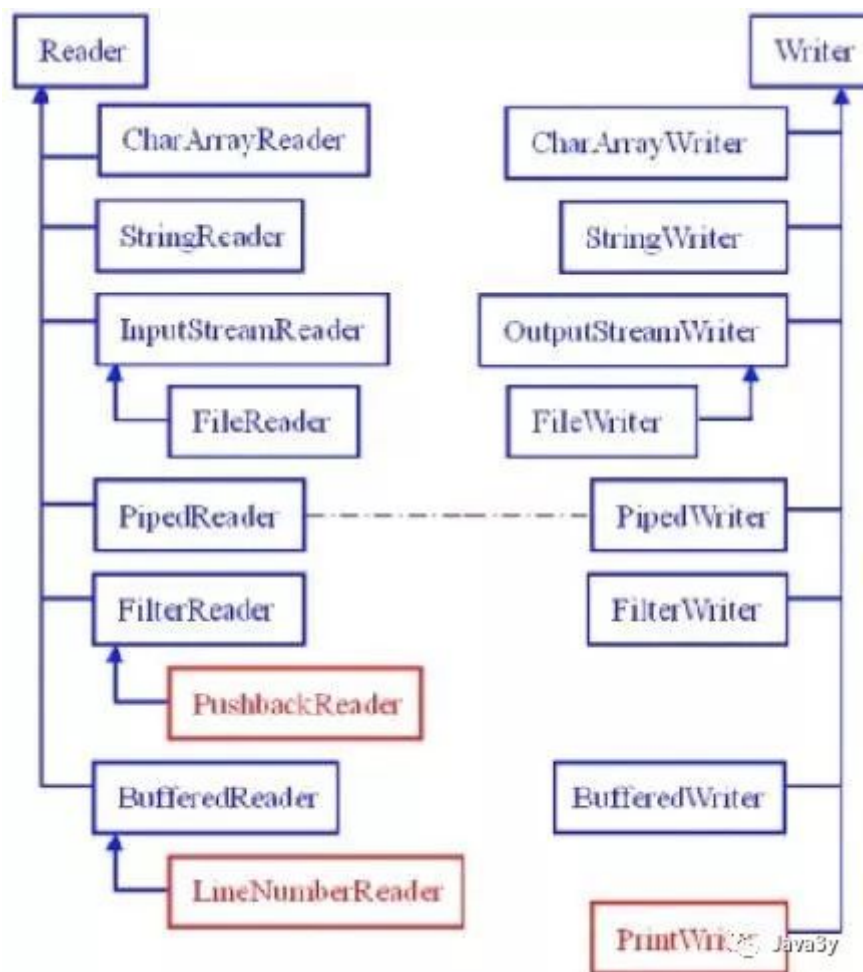
[按操作方式分类结构图：](#)



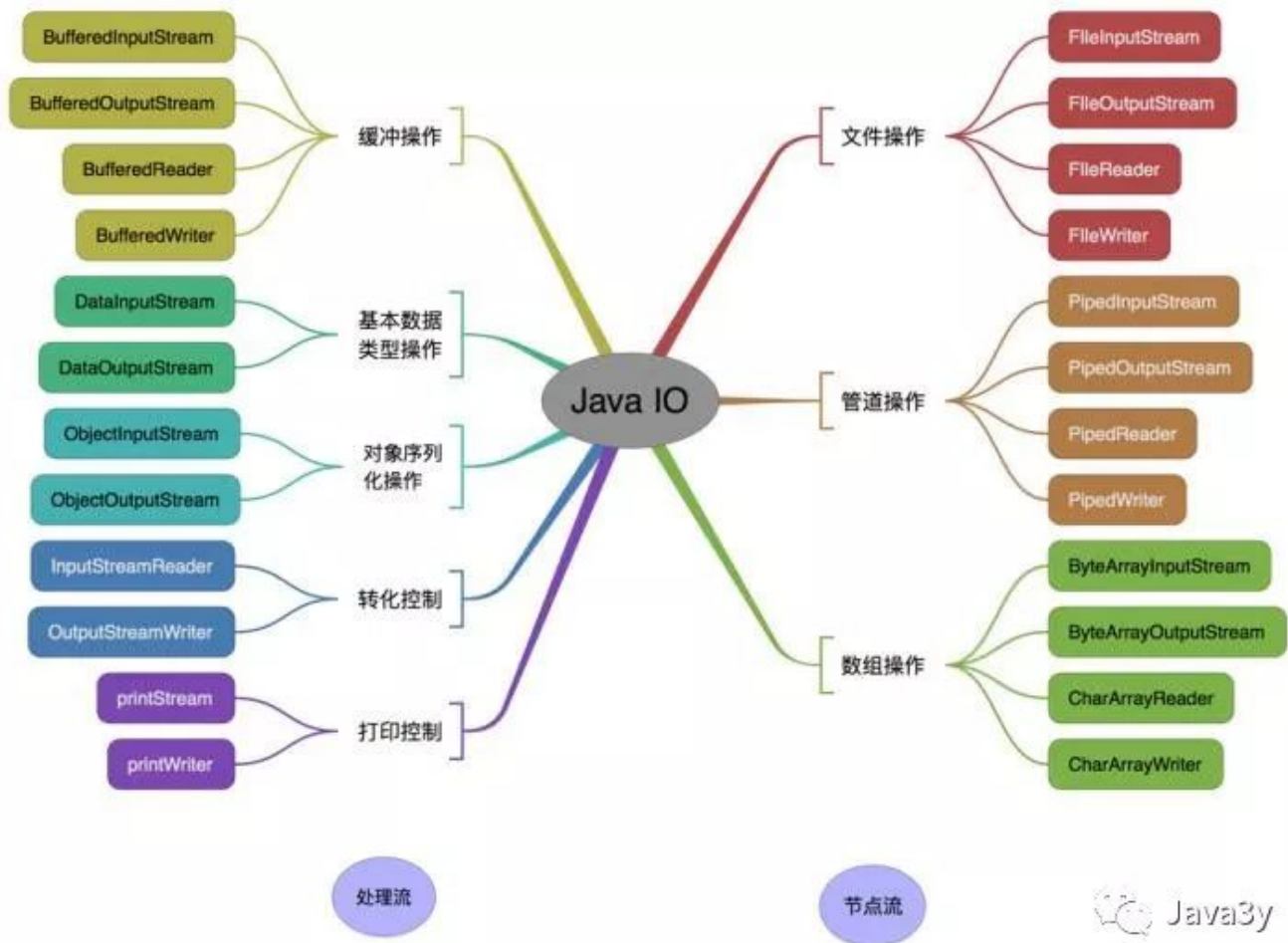
字节流的输入和输出对照图：



字符流的输入和输出对照图：



按操作对象分类结构图：



上述图片原文地址，知乎作者@小明：

- <https://zhuanlan.zhihu.com/p/28286559>

还有 [阅读传统IO源码](#) 的优秀文章：

- <https://blog.csdn.net/panweiwei1994/article/details/78046000>

相信大家看完上面两个给出的链接+理解了 [包装模式就是这么简单啦](#)，传统的IO应该就没什么事啦~~

而NIO对于我来说可以说是挺 [陌生](#) 的，在当初学的时候是接触过的。但是一直没有用它，所以停留认知：nio是jdk1.4开始有的，比传统IO高级。

相信很多初学者都跟我一样，对NIO是不太了解的。而我们现在jdk10都已经发布了，jdk1.4的nio都不知道，这有点说不过去了。

所以我花了几天去了解**NIO的核心知识点**，期间看了《Java 编程思想》和《疯狂Java讲义》的**nio**模块。但是，会发现看完了之后还是很**迷**，不知道**NIO**这是干嘛用的，而网上的资料与书上的知识点没有很好地对应。

- 网上的资料很多都以**IO**的五种模型为基础来讲解**NIO**，而**IO**这五种模型其中又涉及到了很多概念：**同步/异步/阻塞/非阻塞/多路复用**，而不同的人又有不同的理解方式。
- 还有涉及到了**unix**的 **select/epoll/poll/pselect**，**fd** 这些关键字，没有相关基础的人看起来简直是天书
- 这就导致了在初学时认为**nio**远不可及

我在找资料的过程中也收藏了好多讲解**NIO**的资料，这篇文章就是**以初学的角度来理解NIO**。也算是我这两天看**NIO**的一个总结吧。

- 希望大家可以看了之后知道什么是**NIO**，**NIO**的核心知识点是什么，会使用**NIO**~

那么接下来就开始吧，如果文章有错误的地方请大家多多包涵，不吝在评论区指正哦~

声明：本文使用JDK1.8

## 一、NIO的概述

JDK 1.4中的 **java.nio.\*包** 中引入新的Java I/O库，其目的是**提高速度**。实际上，“旧”的I/O包已经使用**NIO**重新实现过，即使我们不显式的使用**NIO**编程，也能从中**受益**。

- **nio**翻译成 **no-blocking io** 或者 **new io** 都无所谓啦，都说得通~

在《Java编程思想》读到“**即使我们不显式的使用NIO编程，也能从中受益**”的时候，我是挺在意的，所以：我们**测试**一下使用**NIO**复制文件和传统**IO**复制文件的性能：

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class SimpleFileTransferTest {
```

```

private long transferFile(File source, File des) throws IOException {
    long startTime = System.currentTimeMillis();

    if (!des.exists())
        des.createNewFile();

    BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source));
    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(des));

    //将数据源读到的内容写入目的地--使用数组
    byte[] bytes = new byte[1024 * 1024];
    int len;
    while ((len = bis.read(bytes)) != -1) {
        bos.write(bytes, 0, len);
    }

    long endTime = System.currentTimeMillis();
    return endTime - startTime;
}

private long transferFileWithNIO(File source, File des) throws IOException {
    long startTime = System.currentTimeMillis();

    if (!des.exists())
        des.createNewFile();

    RandomAccessFile read = new RandomAccessFile(source, "rw");
    RandomAccessFile write = new RandomAccessFile(des, "rw");

    FileChannel readChannel = read.getChannel();
    FileChannel writeChannel = write.getChannel();

    ByteBuffer byteBuffer = ByteBuffer.allocate(1024 * 1024); //1M缓冲区

    while (readChannel.read(byteBuffer) > 0) {
        byteBuffer.flip();
        writeChannel.write(byteBuffer);
        byteBuffer.clear();
    }

    writeChannel.close();
    readChannel.close();
    long endTime = System.currentTimeMillis();
    return endTime - startTime;
}

public static void main(String[] args) throws IOException {
    SimpleFileTransferTest simpleFileTransferTest = new SimpleFileTransferTest();
    File source = new File("F:\\电影\\[电影天堂www.dygod.cn]猜火车-cd1.rmvb");
    File des = new File("X:\\Users\\ozc\\Desktop\\io.avi");
    File nio = new File("X:\\Users\\ozc\\Desktop\\nio.avi");

    long time = simpleFileTransferTest.transferFile(source, des);
    System.out.println(time + ": 普通字节流时间");

    long timeNio = simpleFileTransferTest.transferFileWithNIO(source, nio);
    System.out.println(timeNio + ": NIO时间");

}
}

```



我分别测试了文件大小为13M，40M，200M的：

```
SimpleFileTransferTest
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
599: 普通字节流时间
910: NIO时间 | 文件大小200M

Process finished with exit code 0
Java3y

SimpleFileTransferTest > main()
SimpleFileTransferTest
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
98: 普通字节流时间
74: NIO时间 | 40M

Process finished with exit code 0
Java3y

SimpleFileTransferTest > main()
SimpleFileTransferTest
"X:\Program Files\Java\jdk1.8.0_91\bin\java" ...
33: 普通字节流时间
27: NIO时间 | 13M

Process finished with exit code 0
Java3y
```

## 1.1为什么要使用NIO

可以看到使用过NIO重新实现过的**传统IO根本不虚**，在大文件下效果还比NIO要好(当然了，个人几次的测试，或许不是很准)

- 而NIO要有一定的学习成本，也没有传统IO那么好理解。

那这意味着我们**可以不使用/学习NIO**了吗？

答案是**否定**的，IO操作往往在**两个场景**下会用到：

- 文件IO
- 网络IO



NIO的魅力：在网络中使用IO就可以体现出来了！

- 后面会说到网络中使用NIO，不急哈~

## 二、NIO快速入门

首先我们来看看IO和NIO的区别：

IO	NIO
面向流(Stream Oriented)	面向缓冲区(Buffer Oriented)
阻塞IO(Blocking IO)	非阻塞IO(Non Blocking IO)
(无)	选择器(Selectors)

仅限于  
网络IO  
Java3y

- 可简单认为：**IO**是面向流的处理，**NIO**是面向块(缓冲区)的处理
  - 面向流的I/O 系统一次一个字节地处理数据。
  - 一个面向块(缓冲区)的I/O系统以块的形式处理数据。

NIO主要有三个核心部分组成：

- **buffer**缓冲区
- **Channel**管道
- **Selector**选择器

### 2.1buffer缓冲区和Channel管道

在NIO中并不是以流的方式来处理数据的，而是以buffer缓冲区和Channel管道配合使用来处理数据。

简单理解一下：

- Channel管道比作成铁路，buffer缓冲区比作成火车(运载着货物)

而我们的NIO就是通过Channel管道运输着存储数据的Buffer缓冲区的来实现数据的处理！

- 要时刻记住：Channel不与数据打交道，它只负责运输数据。与数据打交道的是Buffer缓冲区
  - Channel-->运输
  - Buffer-->数据

相对于传统IO而言，流是单向的。对于NIO而言，有了Channel管道这个概念，我们的读写都是双向的(铁路上的火车能从广州去北京、自然就能从北京返还到广州)！

### 2.1.1buffer缓冲区核心要点

我们来看看Buffer缓冲区有什么值得我们注意的地方。

Buffer是缓冲区的抽象类：

```
public abstract class Buffer {  
    /**  
     * The characteristics of Spliterators that traverse and split elements  
     * maintained in Buffers.  
     */  
    static final int SPLITERATOR_CHARACTERISTICS =  
        Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.ORDERED;  
  
    // Invariants: mark <= position <= limit <= capacity  
    private int mark = -1;  
    private int position = 0;  
    private int limit;  
    private int capacity;  
  
    // Used only by direct buffers
```



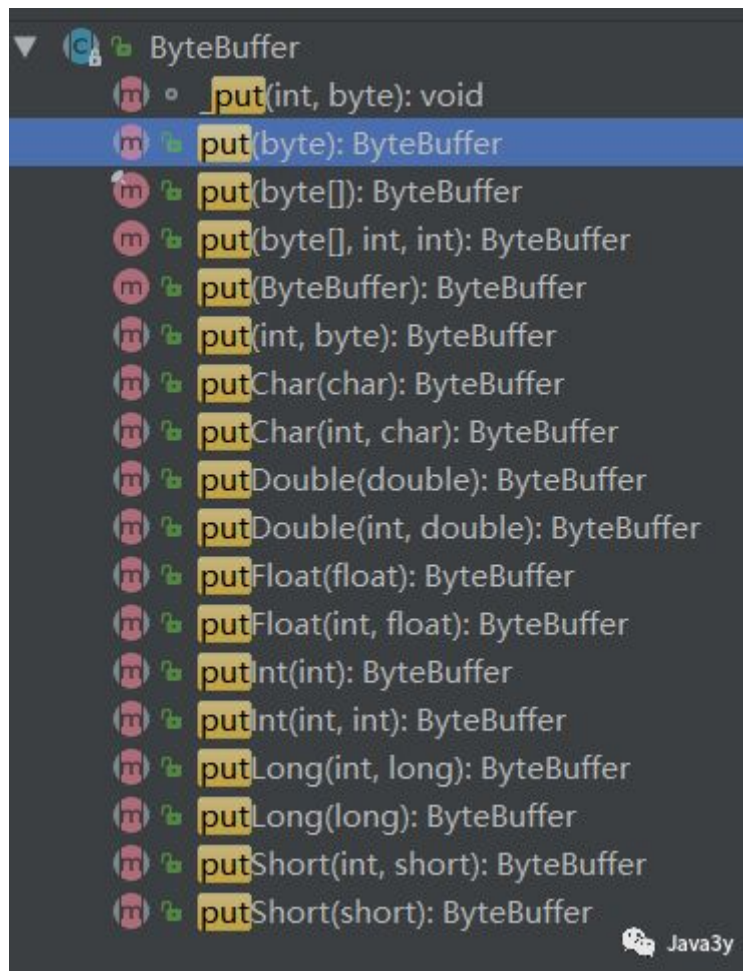
其中ByteBuffer是用得最多的实现类(在管道中读写字节数据)。

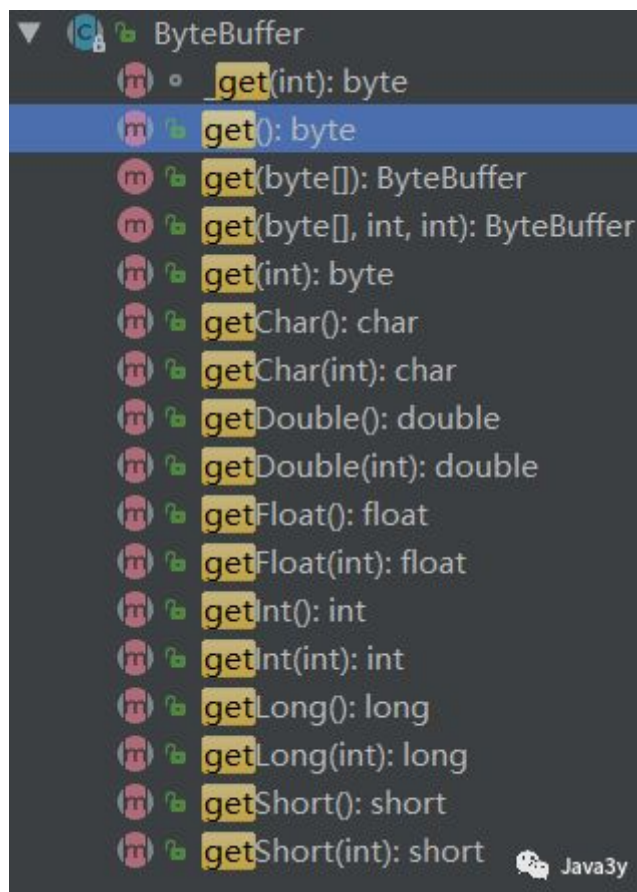
```
/*
 * 一、缓冲区 (Buffer)：在 Java NIO 中负责数据的存取。缓冲区就是数组。用于存储不同数据类型的数据
 *
 * 根据数据类型不同 (boolean 除外)，提供了相应类型的缓冲区：
 * ByteBuffer
 * CharBuffer
 * ShortBuffer
 * IntBuffer
 * LongBuffer
 * FloatBuffer
 * DoubleBuffer
 *
 * 上述缓冲区的管理方式几乎一致，通过 allocate() 获取缓冲区
 */
```



拿到一个缓冲区我们往往会做什么？很简单，就是读取缓冲区的数据 / 写数据到缓冲区中。所以，缓冲区的核心方法就是：

- put()
- get()





Buffer类维护了4个核心变量属性来提供关于其所包含的数组的信息。它们是：

- 容量Capacity

- 缓冲区能够容纳的数据元素的最大数量。容量在缓冲区创建时被设定，并且永远不能被改变。（不能被改变的原因也很简单，底层是数组嘛）

- 上界Limit

- 缓冲区里的数据的总数，代表了当前缓冲区中一共有多少数据。

- 位置Position

- 下一个要被读或写的元素的位置。Position会自动由相应的 `get( )` 和 `put( )` 函数更新。

- 标记Mark

- 一个备忘位置。用于记录上一次读写的位置。

二、缓冲区存取数据的两个核心方法：

**put()**：存入数据到缓冲区中

**get()**：获取缓冲区中的数据

四、缓冲区中的四个核心属性：

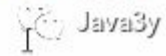
**capacity**：容量，表示缓冲区中最大存储数据的容量。一旦声明不能改变。

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

**position**：位置，表示缓冲区中正在操作数据的位置。

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

$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$



## 2.1.2 buffer 代码演示

首先展示一下是如何创建缓冲区的，核心变量的值是怎么变化的。

```
public static void main(String[] args) {  
    // 创建一个缓冲区  
    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  
  
    // 看一下初始时4个核心变量的值  
    System.out.println("初始时-->limit--->" + byteBuffer.limit());  
    System.out.println("初始时-->position--->" + byteBuffer.position());  
    System.out.println("初始时-->capacity--->" + byteBuffer.capacity());  
    System.out.println("初始时-->mark--->" + byteBuffer.mark());  
  
    System.out.println("-----");  
  
    // 添加一些数据到缓冲区中  
    String s = "Java3y";  
    byteBuffer.put(s.getBytes());  
  
    // 看一下初始时4个核心变量的值  
    System.out.println("put完之后-->limit--->" + byteBuffer.limit());  
    System.out.println("put完之后-->position--->" + byteBuffer.position());  
    System.out.println("put完之后-->capacity--->" + byteBuffer.capacity());  
    System.out.println("put完之后-->mark--->" + byteBuffer.mark());  
}
```

运行结果：

```
初始时-->limit--->1024  
初始时-->position--->0  
初始时-->capacity--->1024  
初始时-->mark--->java.nio.HeapByteBuffer[pos=0 lim=1024 cap=1024]  
-----  
put完之后-->limit--->1024  
put完之后-->position--->6  
put完之后-->capacity--->1024  
put完之后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=1024 cap=1024]
```



现在我想从缓存区拿数据，怎么拿呀？？NIO给了我们一个 `flip()` 方法。这个方法可以改动 `position` 和 `limit` 的位置！

还是上面的代码，我们 `flip()` 一下后，再看看4个核心属性的值会发生什么变化：

```
put完之后-->limit--->1024
put完之后-->position--->6
put完之后-->capacity--->1024
put完之后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=1024 cap=1024]
-----
flip完之后-->limit--->6
flip完之后-->position--->0
flip完之后-->capacity--->1024
flip完之后-->mark--->java.nio.HeapByteBuffer[pos=0 lim=6 cap=1024]
```

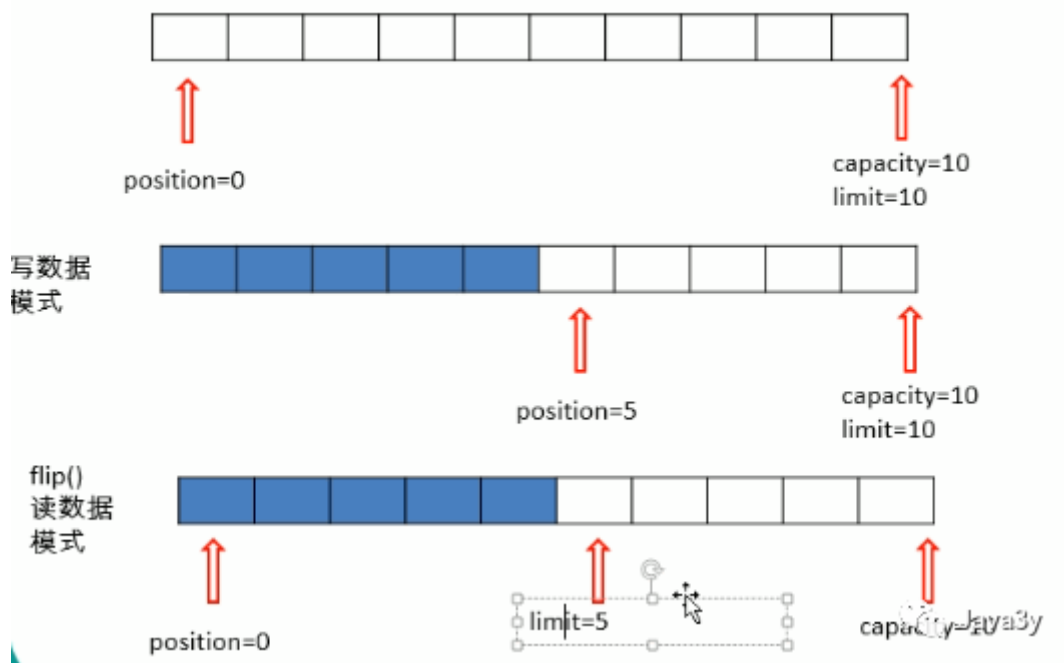
很明显的是：

- `limit`变成了`position`的位置了
- 而`position`变成了0

看到这里的同学可能就会想到了：当调用完 `flip()` 时：`limit`是限制读到哪里，而 `position`是从哪里读

一般我们称 `flip()` 为“切换成读模式”

- 每当要从缓存区的时候读取数据时，就调用 `flip()` “切换成读模式”。





切换到读模式之后，我们就可以读取缓冲区的数据了：

```
// 创建一个limit()大小的字节数组(因为就只有limit这么多个数据可读)
byte[] bytes = new byte[byteBuffer.limit()];

// 将读取的数据装进我们的字节数组中
byteBuffer.get(bytes);

// 输出数据
System.out.println(new String(bytes, 0, bytes.length));
```

Java3y

随后输出一下核心变量的值看看：

```
flip完之后-->limit--->6
flip完之后-->position--->0
flip完之后-->capacity--->1024
flip完之后-->mark--->java.nio.HeapByteBuffer[pos=0 lim=6 cap=1024]
-----
Java3y
-----
get完之后-->limit--->6
get完之后-->position--->6
get完之后-->capacity--->1024
get完之后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=6 cap=1024]
```

get同样会影响position的位置

读完我们还想写数据到缓冲区，那就使用 `clear()` 函数，这个函数会“清空”缓冲区：

- 数据没有真正被清空，只是被遗忘掉了

```
71     byteBuffer.clear();
72     System.out.println("clear完之后-->limit--->" + byteBuffer.limit());
73     System.out.println("clear完之后-->position--->" + byteBuffer.position());
74     System.out.println("clear完之后-->capacity--->" + byteBuffer.capacity());
75     System.out.println("clear完之后-->mark--->" + byteBuffer.mark());
76
```

```
-----
Java3y
-----
get完之后-->limit--->6
get完之后-->position--->6
get完之后-->capacity--->1024
get完之后-->mark--->java.nio.HeapByteBuffer[pos=6 lim=6 cap=1024]
-----
clear完之后-->limit--->1024
clear完之后-->position--->0
clear完之后-->capacity--->1024
clear完之后-->mark--->java.nio.HeapByteBuffer[pos=
```

“清空缓冲区”-->核心变量回归“写模式”，缓冲区数据是没有清空的，但被“遗忘了”，因为操作数据的核心变量都清空了

Java3y

### 2.1.3 FileChannel 通道核心要点



```

* 一、通道 (Channel)：用于源节点与目标节点的连接。在 Java NIO 中负责缓冲区中数据的传输。Channel 本身不存储数据，因此需要配合缓冲区进行传输。
*
* 二、通道的主要实现类
* java.nio.channels.Channel 接口：
*   |--FileChannel
*   |--SocketChannel
*   |--ServerSocketChannel
*   |--DatagramChannel
*
* 三、获取通道
* 1. Java 针对支持通道的类提供了 getChannel() 方法
*   本地 IO：
*   FileInputStream/FileOutputStream
*   RandomAccessFile
*
*   网络 IO：
*   Socket
*   ServerSocket
*   DatagramSocket
*
* 2. 在 JDK 1.7 中的 NIO.2 针对各个通道提供了静态方法 open()
* 2. 在 JDK 1.7 中的 NIO.2 的 Files 工具类的 newByteChannel()
*/
public class TestChannel {

```



Channel 通道只负责传输数据、不直接操作数据的。操作数据都是通过 Buffer 缓冲区来进行操作！

```

// 1. 通过本地IO的方式来获取通道
FileInputStream fileInputStream = new FileInputStream("F:\\3yBlog\\JavaEE常用框架\\Elasticsearch就

// 得到文件的输入通道
FileChannel inChannel = fileInputStream.getChannel();

// 2. jdk1.7后通过静态方法.open()获取通道
FileChannel.open(Paths.get("F:\\3yBlog\\JavaEE常用框架\\Elasticsearch就是这么简单2.md"), StandardO

```

使用 FileChannel 配合缓冲区实现文件复制的功能：

```

//③获取通道
FileChannel inChannel = null;
FileChannel outChannel = null;
try {
    fis = new FileInputStream("1.jpg");
    fos = new FileOutputStream("2.jpg");

    inChannel = fis.getChannel();
    outChannel = fos.getChannel();

    //③分配指定大小的缓冲区
    ByteBuffer buf = ByteBuffer.allocate(1024);

    //③将通道中的数据存入缓冲区中
    while(inChannel.read(buf) != -1){
        buf.flip(); //切换读取数据的模式
        //④将缓冲区中的数据写入通道中
        outChannel.write(buf);
        buf.clear(); //清空缓冲区
    }
} catch (IOException e) {

```



使用 **内存映射文件** 的方式实现 **文件复制** 的功能 (直接操作缓冲区):

```
//使用直接缓冲区完成文件的复制(内存映射文件)
@Test
public void test2() throws IOException{
    FileChannel inChannel = FileChannel.open(Paths.get("1.jpg"), StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get("2.jpg"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

    //内存映射文件
    MappedByteBuffer inMappedBuf = inChannel.map(MapMode.READ_ONLY, 0, inChannel.size());
    MappedByteBuffer outMappedBuf = outChannel.map(MapMode.READ_WRITE, 0, inChannel.size());

    //直接对缓冲区进行数据的读写操作
    byte[] dst = new byte[inMappedBuf.limit()];
    inMappedBuf.get(dst);
    outMappedBuf.put(dst);
}
```



这里还有个read模式



通道之间通过 `transfer()` 实现数据的传输 (直接操作缓冲区):

```
四、通过FileChannel实现文件复制
* transferFrom()
* transferTo()
*/
public class TestChannel {

    //通道之间的数据传输(直接缓冲区)
    @Test
    public void test3() throws IOException{
        FileChannel inChannel = FileChannel.open(Paths.get("d:/1.mkv"), StandardOpenOption.READ);
        FileChannel outChannel = FileChannel.open(Paths.get("d:/2.mkv"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

        inChannel.transferTo(0, inChannel.size(), outChannel);

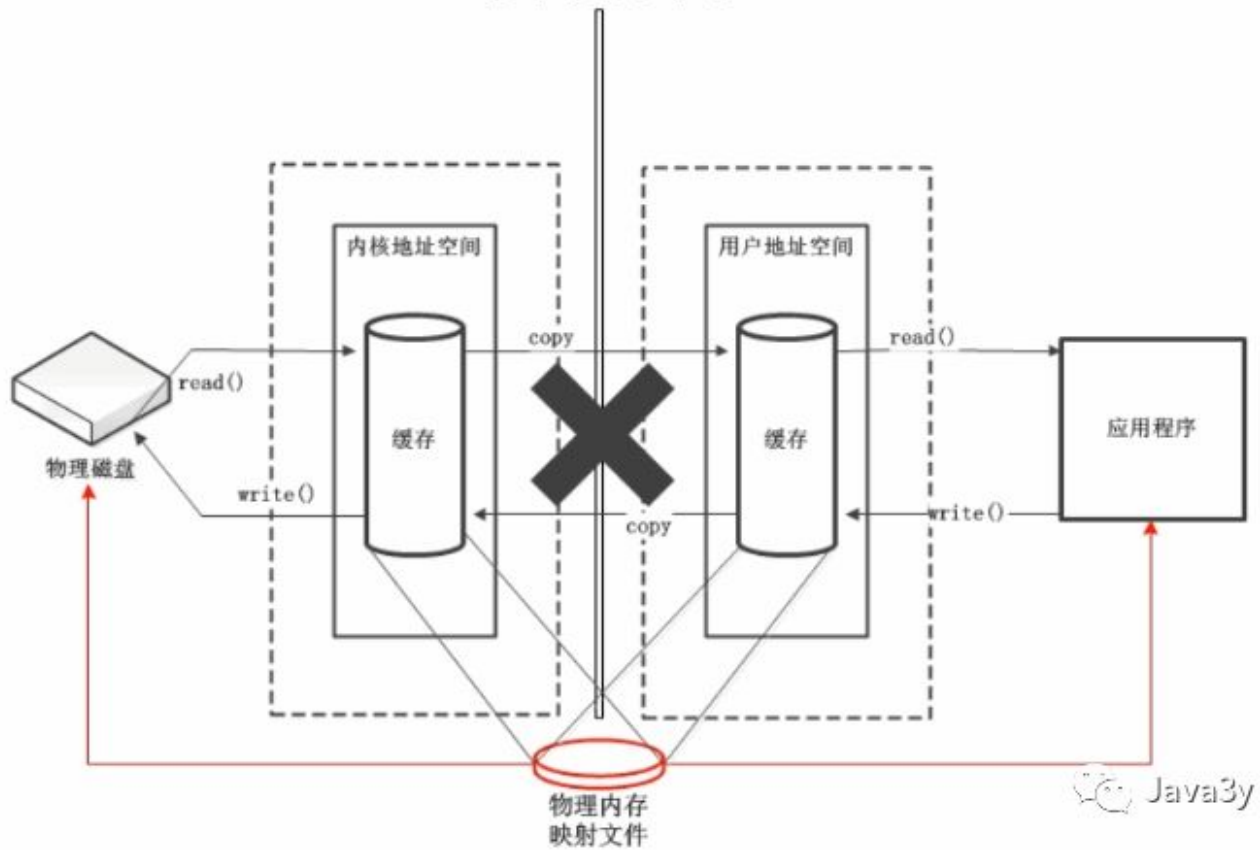
        inChannel.close();
        outChannel.close();
    }
}
```



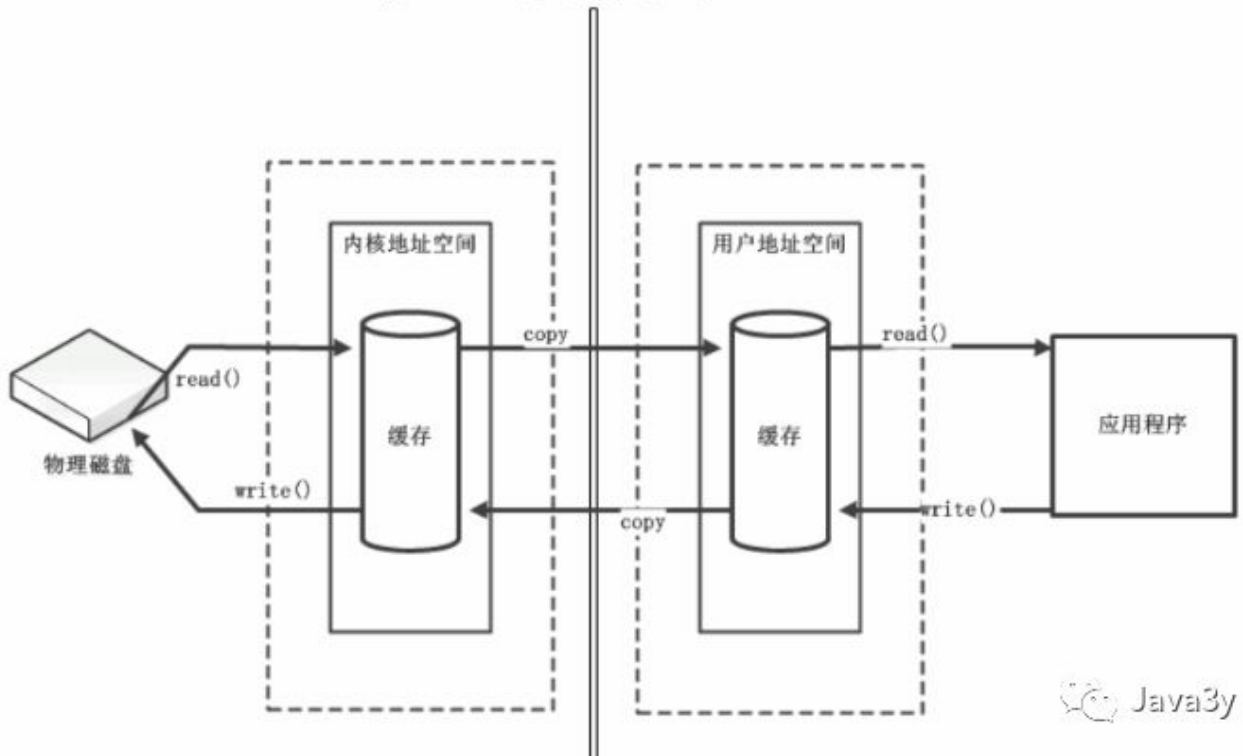
## 2.1.4 直接与非直接缓冲区

- 非直接缓冲区是 **需要** 经过一个: **copy** 的阶段 (从内核空间 **copy** 到用户空间)
- 直接缓冲区 **不需要** 经过 **copy** 阶段, 也可以理解成 ---> **内存映射文件**, (上面的图片也有过例子)。

# 直接缓冲区



# 非直接缓冲区



使用直接缓冲区有两种方式：

- 缓冲区创建的时候分配的是直接缓冲区
- 在FileChannel上调用 `map()` 方法，将文件直接映射到内存中创建

## 直接与非直接缓冲区

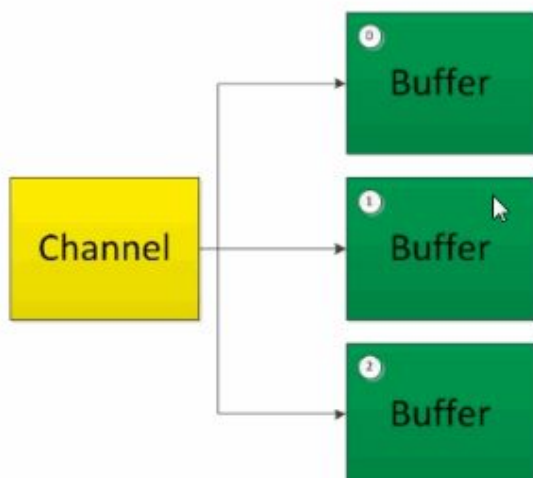
- 字节缓冲区要么是直接的，要么是非直接的。如果为直接字节缓冲区，则 Java 虚拟机会尽最大努力直接在此缓冲区上执行本机 I/O 操作。也就是说，在每次调用基础操作系统的一个本机 I/O 操作之前（或之后），虚拟机都会尽量避免将缓冲区的内容复制到中间缓冲区中（或从中间缓冲区中复制内容）。
- 直接字节缓冲区可以通过调用此类的 `allocateDirect()` 工厂方法来创建。此方法返回的缓冲区进行分配和取消分配所需成本通常高于非直接缓冲区。直接缓冲区的内容可以驻留在常规的垃圾回收堆之外，因此，它们对应用程序的内存需求量造成的影响可能并不明显。所以，建议将直接缓冲区主要分配给那些易受基础系统的本机 I/O 操作影响的大型、持久的缓冲区。一般情况下，最好仅在直接缓冲区能在程序性能方面带来明显好处时分配它们。
- 直接字节缓冲区还可以通过 FileChannel 的 `map()` 方法将文件区域直接映射到内存中来创建。该方法返回 `MappedByteBuffer`。Java 平台的实现有助于通过 JNI 从本机代码创建直接字节缓冲区。如果以上这些缓冲区中的某个缓冲区实例指的是不可访问的内存区域，则试图访问该区域不会更改该缓冲区的内容，并且将会在访问期间或稍后的某个时间导致抛出不确定的异常。
- 字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 `isDirect()` 方法来确定。提供此方法是为了能够在性能关键型代码中执行显式缓冲区管理。

### 2.1.5 scatter和gather、字符集

这个知识点我感觉用得挺少的，不过很多教程都有说这个知识点，我也拿过来说说吧：

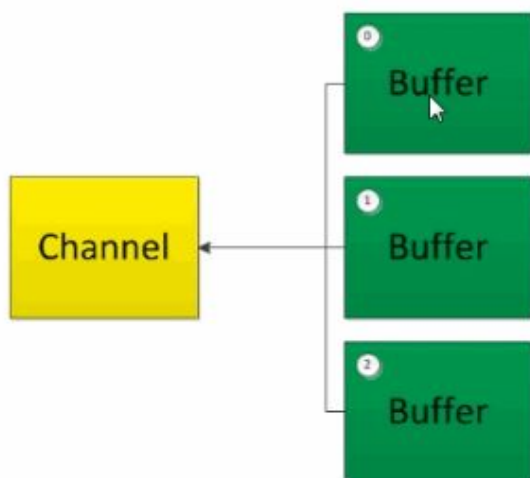
- 分散读取(`scatter`)：将一个通道中的数据分散读取到多个缓冲区中
- 聚集写入(`gather`)：将多个缓冲区中的数据集中写入到一个通道中

- 分散读取（Scattering Reads）是指从 Channel 中读取的数据“分散”到多个 Buffer 中。



注意：按照缓冲区的顺序，从 Channel 中读取的数据依次将 Buffer 填满。

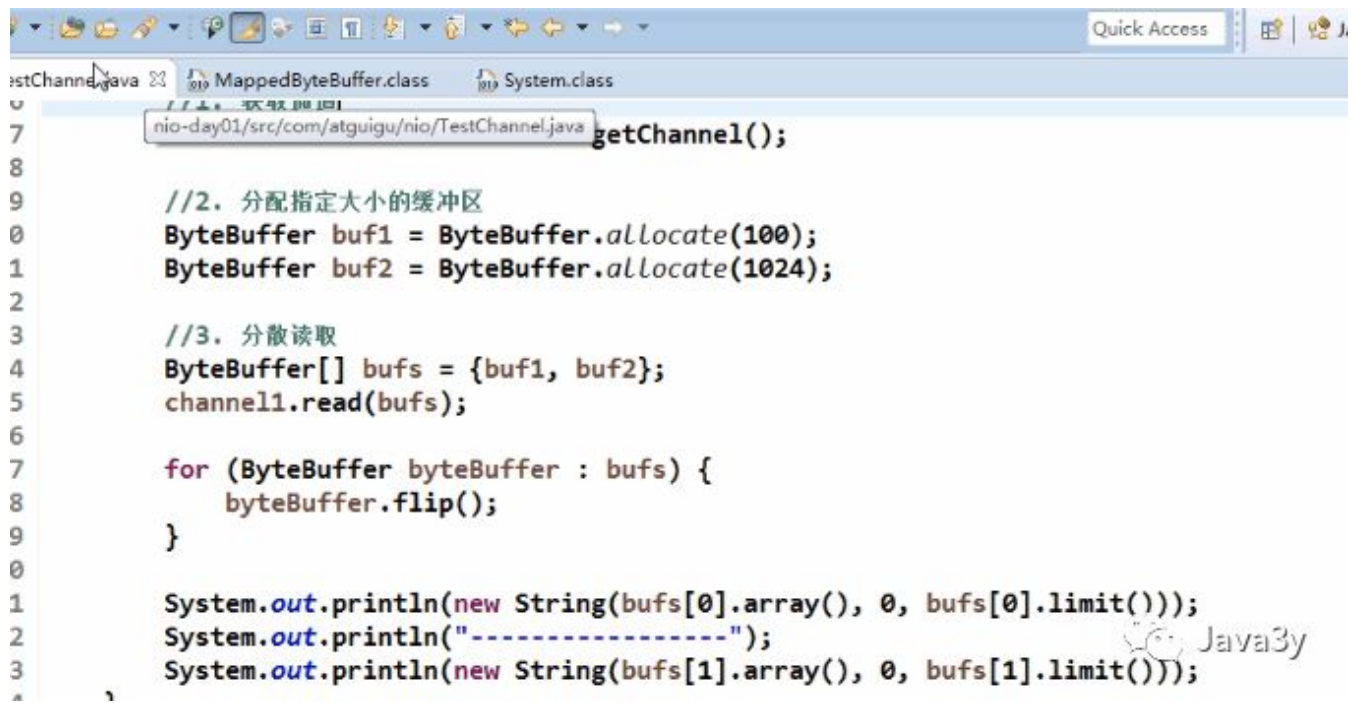
- 聚集写入（Gathering Writes）是指将多个 Buffer 中的数据“聚集”到 Channel。



注意：按照缓冲区的顺序，写入 position 和 limit 之间的数据到 Channel。

分散读取



A screenshot of an IDE window showing a Java file named TestChannel.java. The code is for a NIO channel test. It includes comments in Chinese and Java code for allocating buffers, reading from a channel, and printing the results. The code is as follows:

```
7 getChannel();
8
9 //2. 分配指定大小的缓冲区
10 ByteBuffer buf1 = ByteBuffer.allocate(100);
11 ByteBuffer buf2 = ByteBuffer.allocate(1024);
12
13 //3. 分散读取
14 ByteBuffer[] bufs = {buf1, buf2};
15 channel1.read(bufs);
16
17 for (ByteBuffer byteBuffer : bufs) {
18     byteBuffer.flip();
19 }
20
21 System.out.println(new String(bufs[0].array(), 0, bufs[0].limit()));
22 System.out.println("-----");
23 System.out.println(new String(bufs[1].array(), 0, bufs[1].limit()));
```

聚集写入

```
//4. 聚集写入
RandomAccessFile raf2 = new RandomAccessFile("2.txt", "rw");
FileChannel channel2 = raf2.getChannel();

channel2.write(bufs);
```

字符集(只要编码格式和解码格式一致, 就没问题了)

```

//字符集
@Test
public void test6() throws IOException{
    Charset cs1 = Charset.forName("GBK");

    //获取编码器
    CharsetEncoder ce = cs1.newEncoder();

    //获取解码器
    CharsetDecoder cd = cs1.newDecoder();

    CharBuffer cBuf = CharBuffer.allocate(1024);
    cBuf.put("尚硅谷威武!");
    cBuf.flip();

    //编码
    ByteBuffer bBuf = ce.encode(cBuf);

    for (int i = 0; i < 12; i++) {
        System.out.println(bBuf.get());
    }

    //解码
    bBuf.flip();
}

```

JavaBy

### 三、IO模型理解

文件的IO就告一段落了，我们来学习网络中的IO~~~为了更好地理解NIO，我们先来学习一下IO的模型~

根据UNIX网络编程对I/O模型的分类，在UNIX可以归纳成5种I/O模型：

- 阻塞I/O
- 非阻塞I/O
- I/O多路复用
- 信号驱动I/O
- 异步I/O

### 3.0学习I/O模型需要的基础

#### 3.0.1文件描述符



Linux 的内核将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令(api)，返回一个 file descriptor (fd, 文件描述符)。而对一个socket的读写也会有响应的描述符，称为 socket fd (socket文件描述符)，描述符就是一个数字，指向内核中的一个结构体(文件路径，数据区等一些属性)。

- 所以说：在Linux下对文件的操作是利用文件描述符(file descriptor)来实现的。

### 3.0.2 用户空间和内核空间

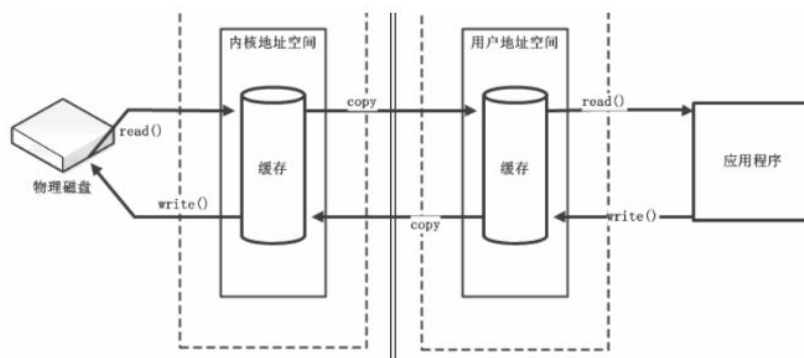
为了保证用户进程不能直接操作内核(kernel)，保证内核的安全，操心系统将虚拟空间划分为两部分

- 一部分为内核空间。
- 一部分为用户空间。

### 3.0.3 I/O运行过程

我们来看看IO在系统中的运行是怎么样的(我们以read为例)

- ① 应用程序调用系统提供的read接口API
- ② 此时系统会干两件事：
  1. 等待数据准备(查看内核空间有没有数据)
  2. 将数据从内核拷贝到进程中(将内核空间数据拷贝到用户空间)



- ③ 应用程序发现用户空间有数据了，读取

可以发现的是：当应用程序调用`read`方法时，是需要等待的--->从内核空间中找数据，再将内核空间的数据拷贝到用户空间的。

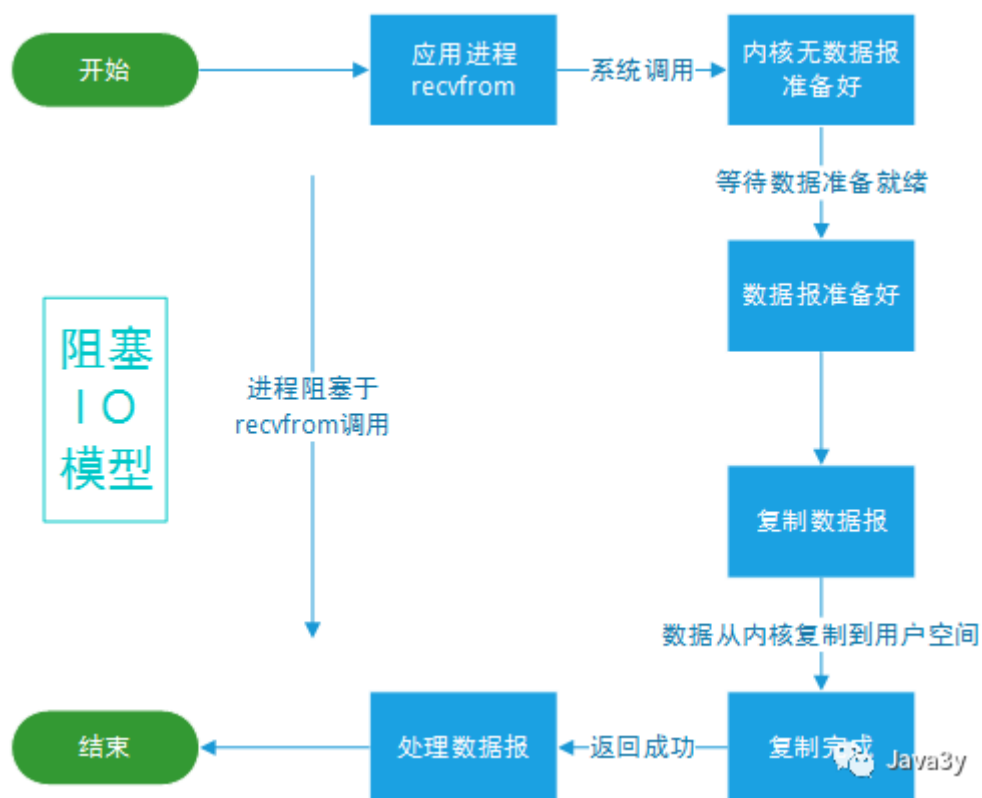
- 这个等待是必要的过程！

下面只讲解用得最多的3个I/O模型：

- 阻塞I/O
- 非阻塞I/O
- I/O多路复用

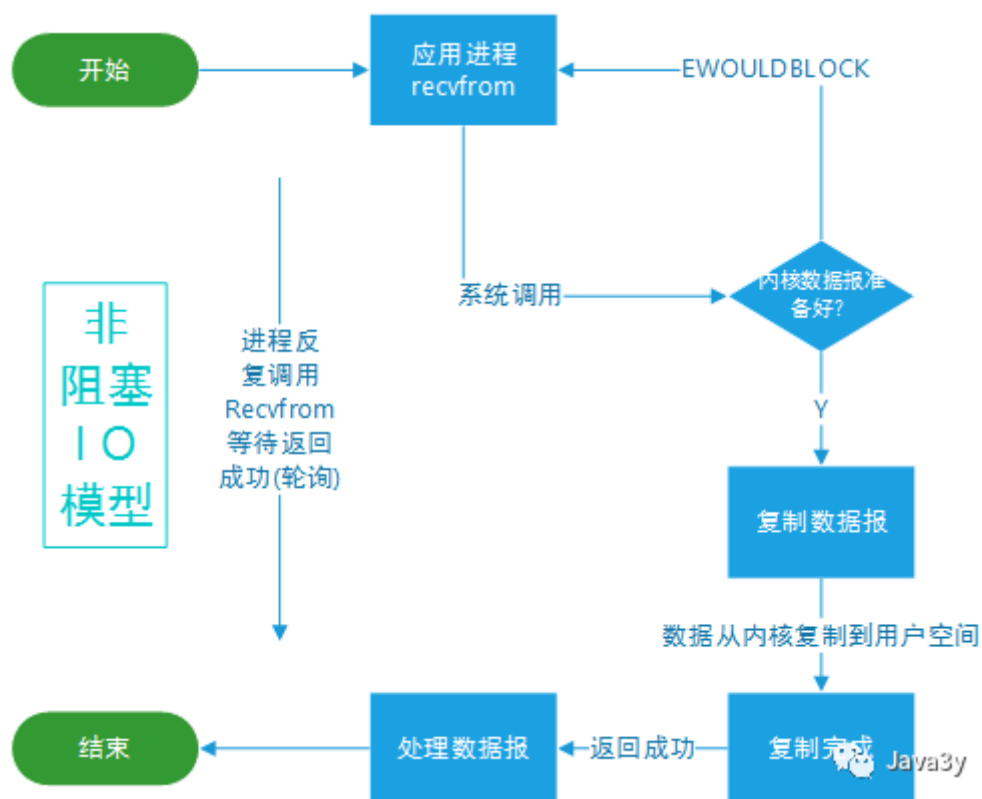
### 3.1 阻塞I/O模型

在进程(用户)空间中调用 `recvfrom`，其系统调用直到数据包到达且被复制到应用进程的缓冲区中或者发生错误时才返回，在此期间一直等待。



### 3.2 非阻塞I/O模型

`recvfrom` 从应用层到内核的时候，如果没有数据就**直接返回**一个 `EWOULDBLOCK` 错误，一般都**对非阻塞I/O模型进行轮询检查这个状态**，看内核是不是有数据到来。



### 3.3 I/O复用模型

前面也已经说了：在Linux下对文件的操作是**利用文件描述符(file descriptor)**来实现的。

在Linux下它是这样子实现I/O复用模型的：

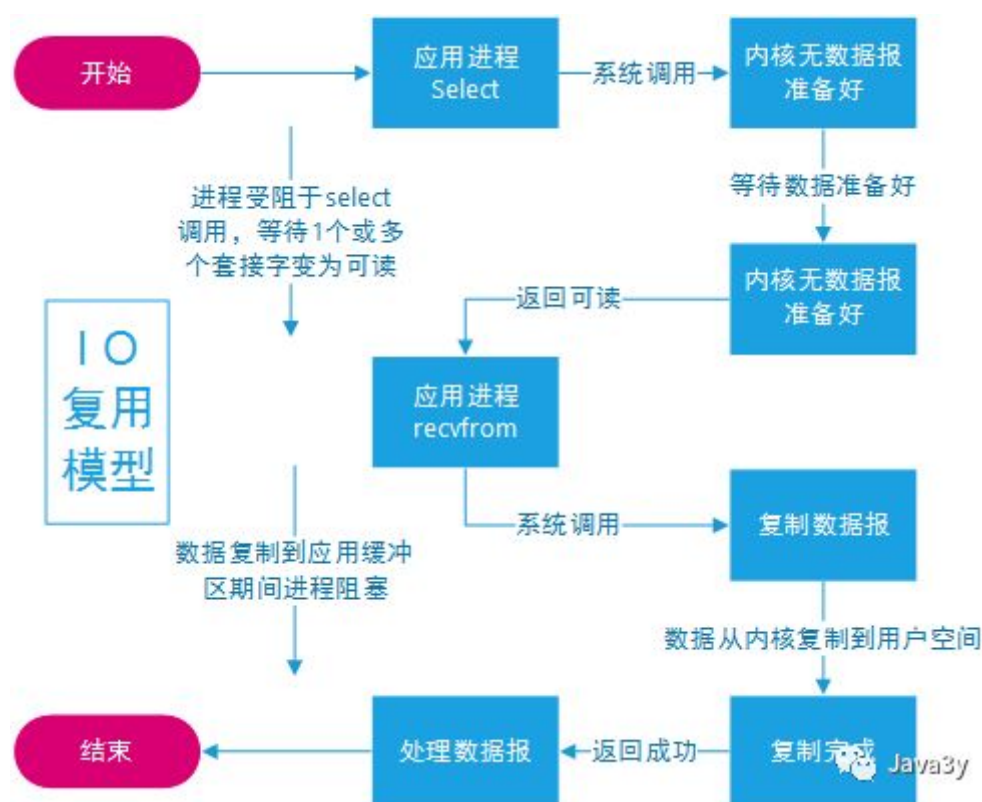
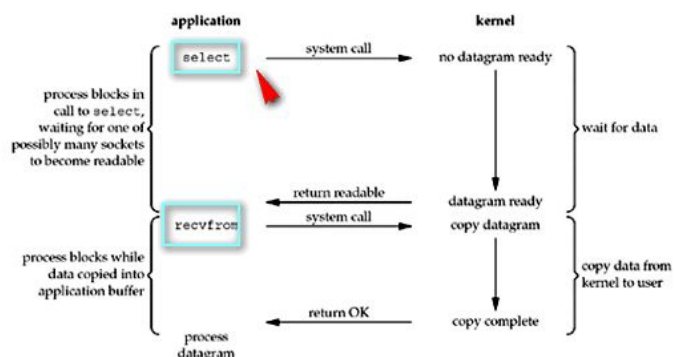
- 调用 `select/poll/epoll/pselect` 其中一个函数，**传入多个文件描述符**，如果有一个文件描述符**就绪，则返回**，否则阻塞直到超时。

比如 `poll()` 函数是这样子的：`int poll(struct pollfd *fds, nfd_t nfd, int timeout);`

其中 `pollfd` 结构定义如下：

```
struct pollfd {
    int fd; /* 文件描述符 */
    short events; /* 等待的事件 */
};
```

```
short revents;    /* 实际发生了的事件 */
};
```



- (1) 当用户进程调用了select，那么整个进程会被block；
- (2) 而同时，kernel会“监视”所有select负责的socket；
- (3) 当任何一个socket中的数据准备好了，select就会返回；
- (4) 这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程(空间)。
- 所以，I/O 多路复用的特点是**通过一种机制一个进程能同时等待多个文件描述符**，而这些文件描述符**其中的任意一个进入读就绪状态**，select()函数就可以返回。

select/epoll的优势并不是对于单个连接能处理得更快，而是**在于能处理更多的连接**。

## 3.4 I/O模型总结

正经的描述都在上面给出了，不知道大家理解了没有。下面我举几个例子总结一下这三种模型：

### 阻塞I/O：

- Java3y跟女朋友去买喜茶，排了很久的队终于可以点饮料了。我要绿研，谢谢。可是喜茶不是点了单就能立即拿，于是我**在喜茶门口等了一小时才拿到**绿研。
  - 在门口干等一小时

### 非阻塞I/O：

- Java3y跟女朋友去买一点点，排了很久的队终于可以点饮料了。我要波霸奶茶，谢谢。可是一点点不是点了单就能立即拿，**同时**服务员告诉我：你大概要等半小时哦。你们先去逛逛吧~于是Java3y跟女朋友去玩了几把斗地主，感觉时间差不多了。于是**又去一点点问**：请问到我了么？我的单号是xxx。服务员告诉Java3y：还没到呢，现在的单号是xxx，你还要等一会，可以去附近耍耍。问了好几次后，终于拿到我的波霸奶茶了。
  - 去逛了下街、斗了下地主，时不时问问到我了没有

### I/O复用模型：

- Java3y跟女朋友去麦当劳吃汉堡包，现在就厉害了可以使用微信小程序点餐了。于是跟女朋友找了个地方坐下就用小程序点餐了。点餐了之后玩玩斗地主、聊聊天什么的。**时不时听到广播在复述xxx请取餐**，反正我的单号还没到，就继续玩呗。~~**等听到广播的时候再取餐就是了**。时间过得挺快的，此时传来：Java3y请过来取餐。于是我就能拿到我的麦辣鸡翅汉堡了。
  - 听广播取餐，**广播不是为我一个人服务**。广播喊到我了，我过去取就Ok了。

## 四、使用NIO完成网络通信

### 4.1 NIO基础继续讲解

回到我们最开始的图：

IO	NIO
面向流 (Stream Oriented)	面向缓冲区 (Buffer Oriented)
阻塞IO (Blocking IO)	非阻塞IO (Non Blocking IO)
(无)	选择器 (Selectors)

仅限于网络IO  
Java3y

NIO被叫为 `no-blocking io`，其实是在网络这个层次中理解的，对于 `FileChannel` 来说一样是阻塞。

我们前面也仅仅讲解了 `FileChannel`，对于我们网络通信是还有几个 `Channel` 的~

```
3 /*
4  * 一、使用 NIO 完成网络通信的三个核心：
5  *
6  * 1. 通道 (Channel)：负责连接
7  *
8  *     java.nio.channels.Channel 接口：
9  *         |--SelectableChannel
10 *             |--SocketChannel
11 *             |--ServerSocketChannel
12 *             |--DatagramChannel
13 *
14 *             |--Pipe.SinkChannel
15 *             |--Pipe.SourceChannel
16 *
17 * 2. 缓冲区 (Buffer)：负责数据的存取
18 *
19 * 3. 选择器 (Selector)：是 SelectableChannel 的多路复用器。用于监控 SelectableChannel 的 IO
20 *
21 */
```

Java3y

所以说：我们通常使用NIO是在网络中使用的，网上大部分讨论NIO都是在网络通信的基础之上的！说NIO是非阻塞的NIO也是网络中体现的！

从上面的图我们可以发现还有一个 `Selector` 选择器这么一个东东。从一开始我们就说过了，nio的核心要素有：

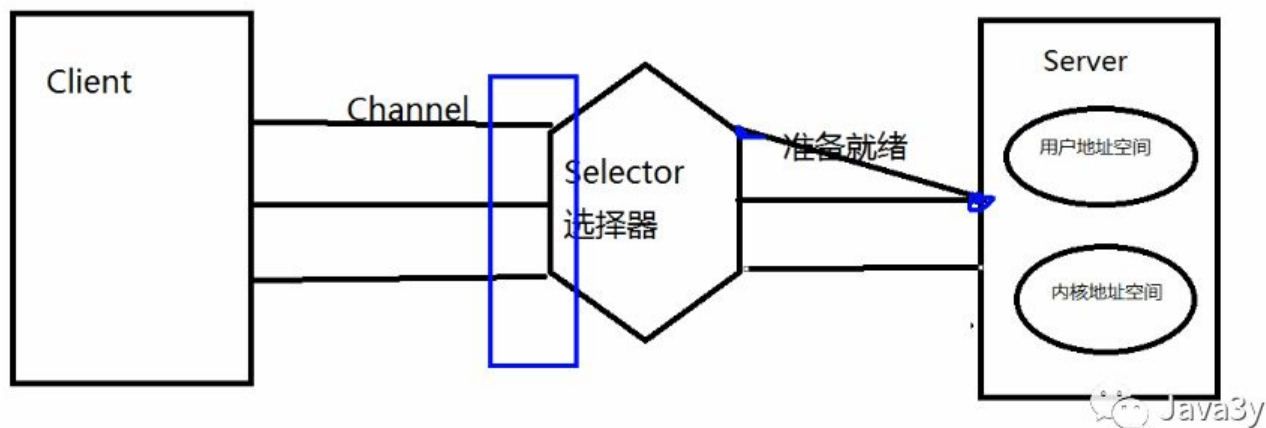
- Buffer缓冲区

- Channel 通道
- Selector 选择器

我们在网络中使用NIO往往是I/O模型的多路复用模型！

- Selector选择器就可以比喻成麦当劳的广播。
- 一个线程能够管理多个Channel的状态

NIO 的非阻塞模式



## 4.2 NIO 阻塞形态

为了更好地理解，我们先来写一下NIO在网络中是阻塞的状态代码，随后看看非阻塞是怎么写的就更容易理解了。

- 是阻塞的就没有Selector选择器了，就直接使用Channel和Buffer就完事了。

客户端：

```
public class BlockClient {
    public static void main(String[] args) throws IOException {
        // 1. 获取通道
        SocketChannel socketChannel = SocketChannel.open(new InetSocketAddress("127.0.0.1", 6666));

        // 2. 发送一张图片给服务端吧
        FileChannel fileChannel = FileChannel.open(Paths.get("X:\\Users\\ozc\\Desktop\\新建文件夹\\1.png"));

        // 3. 要使用NIO，有了Channel，就必然要有Buffer，Buffer是与数据打交道的呢
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 4. 读取本地文件(图片)，发送到服务器
        while (fileChannel.read(buffer) != -1) {
```



```

        // 在读之前都要切换成读模式
        buffer.flip();

        socketChannel.write(buffer);

        // 读完切换成写模式，能让管道继续读取文件的数据
        buffer.clear();
    }

    // 5. 关闭流
    fileChannel.close();
    socketChannel.close();
}
}

```

服务端：

```

public class BlockServer {

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

        // 1. 获取通道
        ServerSocketChannel server = ServerSocketChannel.open();

        // 2. 得到文件通道，将客户端传递过来的图片写到本地项目下(写模式、没有则创建)
        FileChannel outChannel = FileChannel.open(Paths.get("2.png"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

        // 3. 绑定链接
        server.bind(new InetSocketAddress(6666));

        // 4. 获取客户端的连接(阻塞的)
        SocketChannel client = server.accept();

        // 5. 要使用NIO，有了Channel，就必然要有Buffer，Buffer是与数据打交道的呢
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 6. 将客户端传递过来的图片保存在本地中
        while (client.read(buffer) != -1) {

            // 在读之前都要切换成读模式
            buffer.flip();

            outChannel.write(buffer);

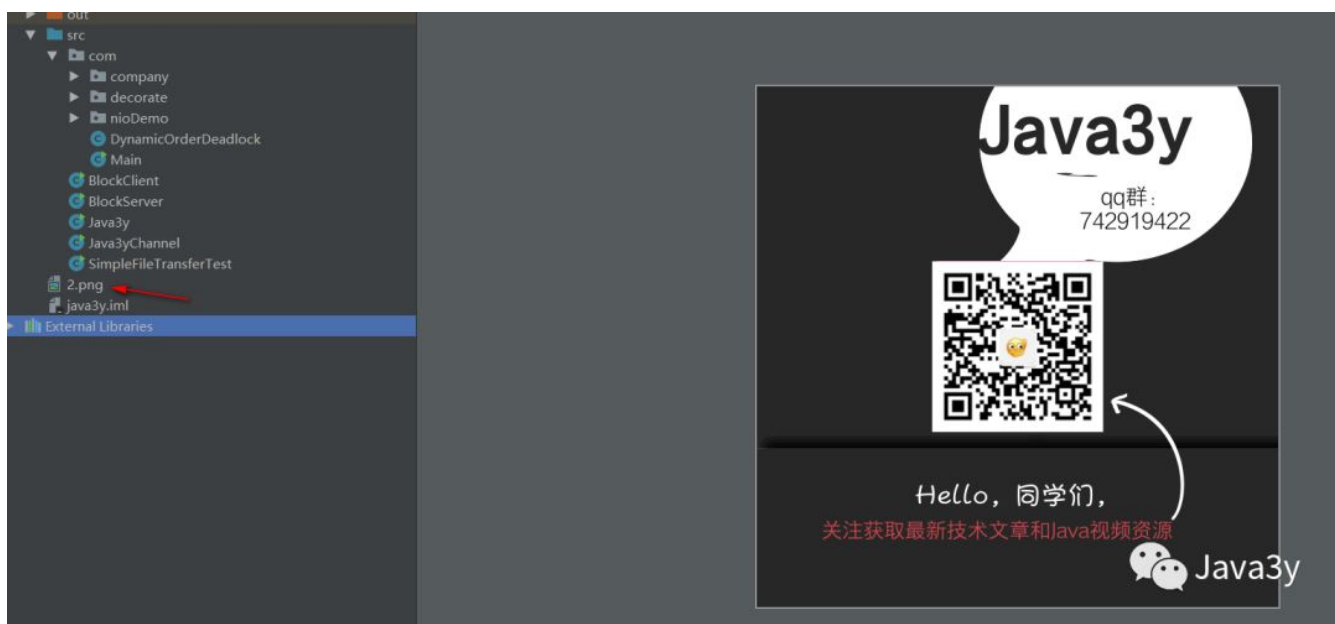
            // 读完切换成写模式，能让管道继续读取文件的数据
            buffer.clear();

        }

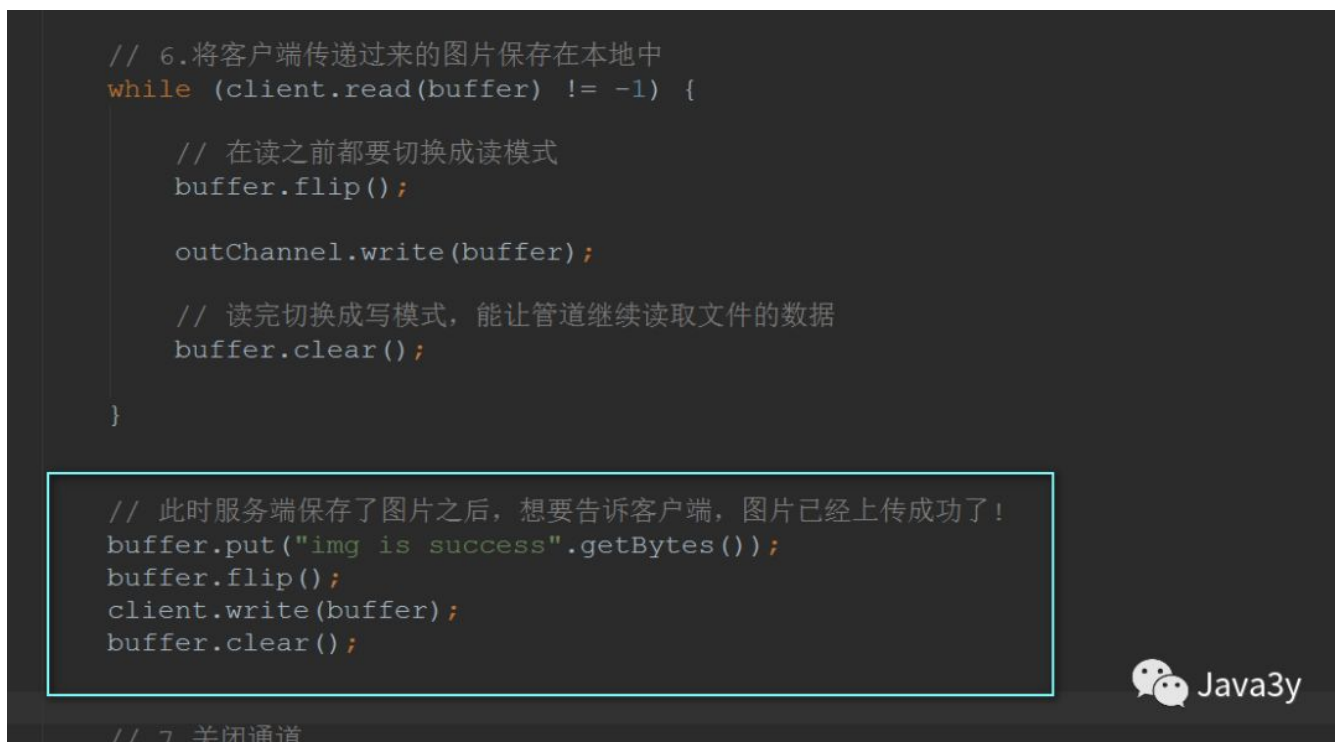
        // 7. 关闭通道
        outChannel.close();
        client.close();
        server.close();
    }
}

```

结果就可以将客户端传递过来的图片保存在本地了：



此时服务端保存完图片想要告诉客户端已经收到图片啦：



客户端接收服务端带过来的数据：

```
// 4.读取本地文件(图片)，发送到服务器
while (fileChannel.read(buffer) != -1) {

    // 在读之前都要切换成读模式
    buffer.flip();

    socketChannel.write(buffer);

    // 读完切换成写模式，能让管道继续读取文件的数据
    buffer.clear();
}

// 知道服务端要返回响应的数据给客户端，客户端在这里接收
int len = 0;
while ((len = socketChannel.read(buffer)) != -1) {

    // 切换读模式
    buffer.flip();

    System.out.println(new String(buffer.array(), offset: 0, len));

    // 切换写模式
    buffer.clear();
}
```



如果仅仅是上面的代码是不行的！这个程序会阻塞起来！

- 因为服务端不知道客户端还有没有数据要发过来(与刚开始不一样，客户端发完数据就将流关闭了，服务端可以知道客户端没数据发过来了)，导致服务端一直在读取客户端发过来的数据。
- 进而导致了阻塞！

于是客户端在写完数据给服务端时，显式告诉服务端已经发完数据了！

```
// 4.读取本地文件(图片), 发送到服务器
while (fileChannel.read(buffer) != -1) {

    // 在读之前都要切换成读模式
    buffer.flip();

    socketChannel.write(buffer);

    // 读完切换成写模式, 能让管道继续读取文件的数据
    buffer.clear();
}

// 告诉服务器已经写完了
socketChannel.shutdownOutput();

// 知道服务端要返回响应的数据给客户端, 客户端在这里接收
int len = 0;
while ((len = socketChannel.read(buffer)) != -1) {
```



## 4.3 NIO非阻塞形态

如果使用非阻塞模式的话, 那么我们就可以不显式告诉服务器已经发完数据了。我们下面来看看怎么写:

客户端:

```
public class NoBlockClient {

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

        // 1. 获取通道
        SocketChannel socketChannel = SocketChannel.open(new InetSocketAddress("127.0.0.1", 6666));

        // 1.1 切换成非阻塞模式
        socketChannel.configureBlocking(false);

        // 2. 发送一张图片给服务端吧
        FileChannel fileChannel = FileChannel.open(Paths.get("X:\\Users\\ozc\\Desktop\\新建文件夹\\1.png"));

        // 3. 要使用NIO, 有了Channel, 就必然要有Buffer, Buffer是与数据打交道的呢
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 4. 读取本地文件(图片), 发送到服务器
        while (fileChannel.read(buffer) != -1) {

            // 在读之前都要切换成读模式
            buffer.flip();

            socketChannel.write(buffer);
```

```

        // 读完切换成写模式，能让管道继续读取文件的数据
        buffer.clear();
    }

    // 5. 关闭流
    fileChannel.close();
    socketChannel.close();
}
}

```

服务端：

```

public class NoBlockServer {

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

        // 1. 获取通道
        ServerSocketChannel server = ServerSocketChannel.open();

        // 2. 切换成非阻塞模式
        server.configureBlocking(false);

        // 3. 绑定连接
        server.bind(new InetSocketAddress(6666));

        // 4. 获取选择器
        Selector selector = Selector.open();

        // 4.1 将通道注册到选择器上，指定接收“监听通道”事件
        server.register(selector, SelectionKey.OP_ACCEPT);

        // 5. 轮训地获取选择器上已“就绪”的事件--->只要select()>0，说明已就绪
        while (selector.select() > 0) {
            // 6. 获取当前选择器所有注册的“选择键”(已就绪的监听事件)
            Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

            // 7. 获取已“就绪”的事件，(不同的事件做不同的事)
            while (iterator.hasNext()) {

                SelectionKey selectionKey = iterator.next();

                // 接收事件就绪
                if (selectionKey.isAcceptable()) {

                    // 8. 获取客户端的连接
                    SocketChannel client = server.accept();

                    // 8.1 切换成非阻塞状态
                    client.configureBlocking(false);

                    // 8.2 注册到选择器上-->拿到客户端的连接为了读取通道的数据(监听读就绪事件)
                    client.register(selector, SelectionKey.OP_READ);

                } elseif (selectionKey.isReadable()) { // 读事件就绪

                    // 9. 获取当前选择器读就绪状态的通道
                    SocketChannel client = (SocketChannel) selectionKey.channel();

                    // 9.1 读取数据
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                }
            }
        }
    }
}

```

```

// 9.2得到文件通道，将客户端传递过来的图片写到本地项目下(写模式、没有则创建)
FileChannel outChannel = FileChannel.open(Paths.get("2.png"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

while (client.read(buffer) > 0) {
    // 在读之前都要切换成读模式
    buffer.flip();

    outChannel.write(buffer);

    // 读完切换成写模式，能让管道继续读取文件的数据
    buffer.clear();
}
}
// 10. 取消选择键(已经处理过的事件，就应该取消掉了)
iterator.remove();
}
}
}
}
}

```

还是刚才的需求：**服务端保存了图片以后，告诉客户端已经收到图片了。**

在服务端上只要在后面写些数据给客户端就好了：

```

outChannel.write(buffer);

// 读完切换成写模式，能让管道继续读取文件的数据
buffer.clear();
}

// 10. 此时服务端保存了图片之后，想要告诉客户端，图片已经上传成功了！
ByteBuffer writeBuffer = ByteBuffer.allocate(1024);
writeBuffer.put("the img is received ,thanks you client , I am Java3y".getBytes());
writeBuffer.flip();
client.write(writeBuffer);
}
}

```

在客户端上要想获取得到服务端的数据，也需要注册在**register**上(监听读事件)！

```

public class NoBlockClient2 {

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

        // 1. 获取通道
        SocketChannel socketChannel = SocketChannel.open(new InetSocketAddress("127.0.0.1", 6666));

        // 1.1 切换成非阻塞模式
        socketChannel.configureBlocking(false);

        // 1.2 获取选择器
        Selector selector = Selector.open();

        // 1.3 将通道注册到选择器中，获取服务端返回的数据
        socketChannel.register(selector, SelectionKey.OP_READ);
    }
}

```

```

// 2. 发送一张图片给服务端吧
FileChannel fileChannel = FileChannel.open(Paths.get("X:\\Users\\ozc\\Desktop\\新建文件夹\\1.png"));

// 3. 要使用NIO, 有了Channel, 就必然要有Buffer, Buffer是与数据打交道的呢
ByteBuffer buffer = ByteBuffer.allocate(1024);

// 4. 读取本地文件(图片), 发送到服务器
while (fileChannel.read(buffer) != -1) {

    // 在读之前都要切换成读模式
    buffer.flip();

    socketChannel.write(buffer);

    // 读完切换成写模式, 能让管道继续读取文件的数据
    buffer.clear();
}

// 5. 轮训地获取选择器上已“就绪”的事件--->只要select()>0, 说明已就绪
while (selector.select() > 0) {
    // 6. 获取当前选择器所有注册的“选择键”(已就绪的监听事件)
    Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

    // 7. 获取已“就绪”的事件, (不同的事件做不同的事)
    while (iterator.hasNext()) {

        SelectionKey selectionKey = iterator.next();

        // 8. 读事件就绪
        if (selectionKey.isReadable()) {

            // 8.1 得到对应的通道
            SocketChannel channel = (SocketChannel) selectionKey.channel();

            ByteBuffer responseBuffer = ByteBuffer.allocate(1024);

            // 9. 知道服务端要返回响应的数据给客户端, 客户端在这里接收
            int readBytes = channel.read(responseBuffer);

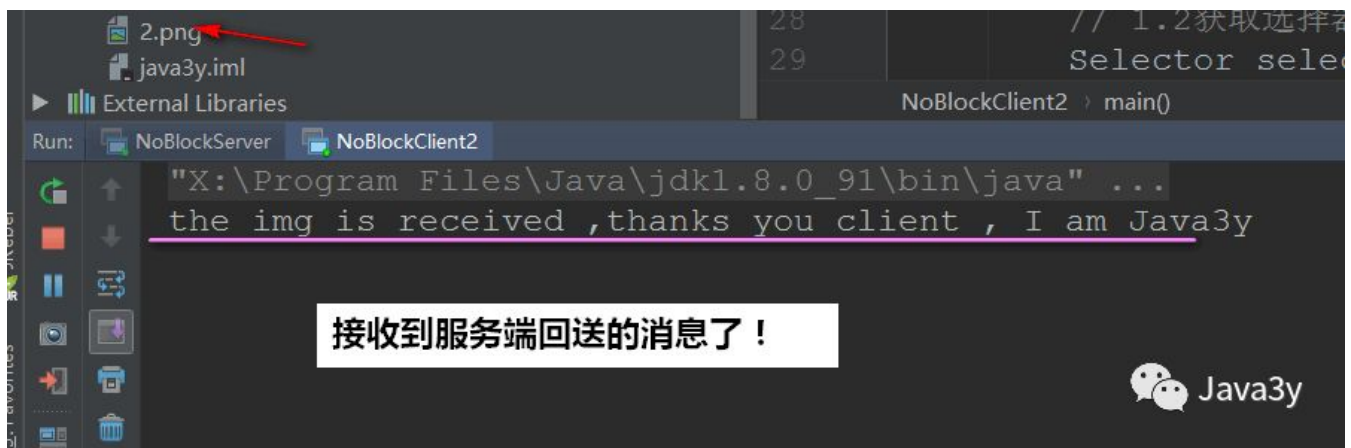
            if (readBytes > 0) {
                // 切换读模式
                responseBuffer.flip();
                System.out.println(new String(responseBuffer.array(), 0, readBytes));
            }
        }
    }

    // 10. 取消选择键(已经处理过的事件, 就应该取消掉了)
    iterator.remove();
}
}
}
}
}

```

测试结果：





下面就简单总结一下使用NIO时的要点：

- 将Socket通道注册到Selector中，监听感兴趣的事件
- 当感兴趣的时间就绪时，则会进去我们处理的方法进行处理
- 每处理完一次就绪事件，删除该选择键(因为我们已经处理完了)

## 4.4管道和DataGramChannel

这里我就不再讲述了，最难的TCP都讲了，UDP就很简单了。

UDP：

```
public void send() throws IOException {
    DatagramChannel dc = DatagramChannel.open();

    dc.configureBlocking(false);

    ByteBuffer buf = ByteBuffer.allocate(1024);

    Scanner scan = new Scanner(System.in);

    while(scan.hasNext()){
        String str = scan.next();
        buf.put((new Date().toString() + ":\n" + str).getBytes());
        buf.flip();
        dc.send(buf, new InetSocketAddress("127.0.0.1", 9898));
        buf.clear();
    }

    dc.close();
}
```

```

44 dc.bind(new InetSocketAddress(9898));
45
46 Selector selector = Selector.open();
47
48 dc.register(selector, SelectionKey.OP_READ);
49
50 while(selector.select() > 0){
51     Iterator<SelectionKey> it = selector.selectedKeys().iterator();
52
53     while(it.hasNext()){
54         SelectionKey sk = it.next();
55
56         if(sk.isReadable()){
57             ByteBuffer buf = ByteBuffer.allocate(1024);
58
59             dc.receive(buf);
60             buf.flip();
61             System.out.println(new String(buf.array(), 0, buf.limit()));
62             buf.clear();
63         }
64     }
65     it.remove();
66 }

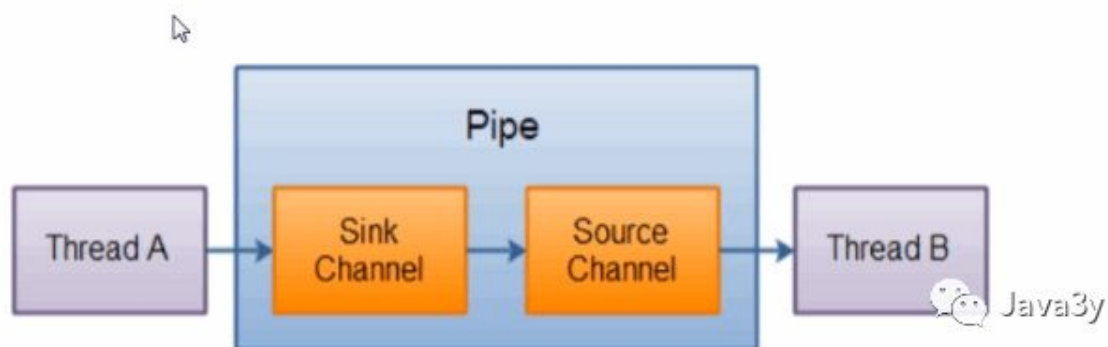
```

Java3y

管道：

## 管道 (Pipe)

- Java NIO 管道是2个线程之间的单向数据连接。Pipe有一个source通道和一个sink通道。数据会被写到sink通道，从source通道读取。



Java3y

```

@Test
public void test1() throws IOException{
    //1. 获取管道
    Pipe pipe = Pipe.open();

    //2. 将缓冲区中的数据写入管道
    ByteBuffer buf = ByteBuffer.allocate(1024);

    Pipe.SinkChannel sinkChannel = pipe.sink();
    buf.put("通过单向管道发送数据".getBytes());
    buf.flip();
    sinkChannel.write(buf);

    //3. 读取缓冲区中的数据
    Pipe.SourceChannel sourceChannel = pipe.source();
    buf.flip();
    int len = sourceChannel.read(buf);
    System.out.println(new String(buf.array(), 0, len));

    sourceChannel.close();
    sinkChannel.close();
}

```



## 五、总结

总的来说NIO也是一个比较重要的知识点，因为它是学习netty的基础~

想以一篇来完全讲解NIO显然是不可能的啦，想要更加深入了解NIO可以往下面的链接继续学习~

参考资料：

- <https://www.zhihu.com/question/29005375>---如何学习Java的NIO?
- <http://ifeve.com/java-nio-all/>---Java NIO 系列教程
- <https://www.ibm.com/developerworks/cn/education/java/j-nio/j-nio.html>-----NIO 入门
- <https://blog.csdn.net/anxpp/article/details/51503329>-----Linux 网络 I/O 模型简介（图文）
- <https://wangjingxin.top/2016/10/21/decoration/>-----谈谈 java 的 NIO和AIO
- [https://www.yiibai.com/java\\_nio/](https://www.yiibai.com/java_nio/)-----Java NIO教程

- <https://blog.csdn.net/cowthan/article/details/53563206>-----Java 8: Java 的新IO (nio)
- <https://blog.csdn.net/youyou1543724847/article/details/52748785>-----JAVA NIO(1.基本概念, 基本类)
- <https://www.cnblogs.com/zingp/p/6863170.html>-----IO模式和IO多路复用
- <https://www.cnblogs.com/Evsward/p/nio.html>----掌握NIO, 程序人生
- <https://blog.csdn.net/anxpp/article/details/51512200>----Java网络IO编程总结(BIO、NIO、AIO均含完整实例代码)
- <https://zhuanlan.zhihu.com/p/24393775?refer=hinus>---进击的Java新人
- 《Java编程思想》
- 《疯狂Java 讲义》

如果文章有错的地方欢迎指正, 大家互相交流。习惯在微信看技术文章, 想要获取更多的Java资源的同学, 可以[关注微信公众号:Java3y](#)。为了大家方便, 刚新建了一下[qq群:742919422](#), 大家也可以去交流交流。谢谢支持了! 希望能多介绍给其他有需要的朋友

文章的目录导航:

- <https://zhongfucheng.bitcron.com/post/shou-ji/wen-zhang-dao-hang>