NIO 编程

NIO 概述

什么是 NIO?

Java NIO(New IO)是一个可以替代标准 Java IO API 的 IO API（从 Java 1.4 开始），Java NIO 提供了与标准 IO 不同的 IO 工作方式。

Java NIO: Channels and Buffers（通道和缓冲区）

标准的 IO 基于字节流和字符流进行操作的，而 NIO 是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

Java NIO: Non-blocking IO（非阻塞 IO）

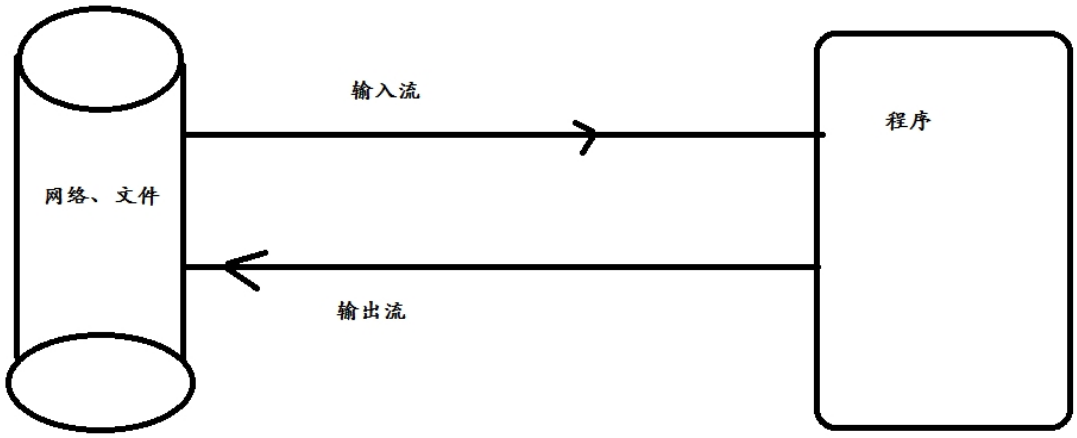
Java NIO 可以让你非阻塞的使用 IO，例如：当线程从通道读取数据到缓冲区时，线程还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

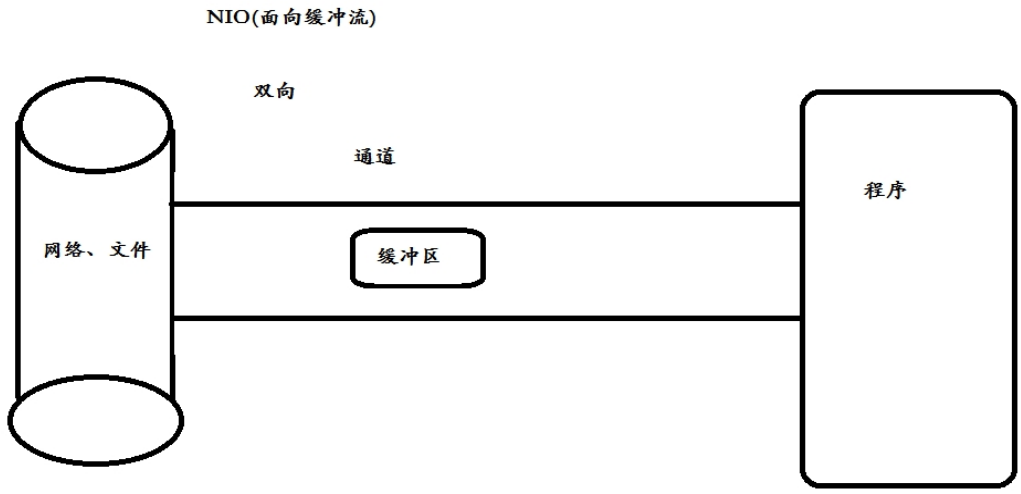
Java NIO: Selectors（选择器）

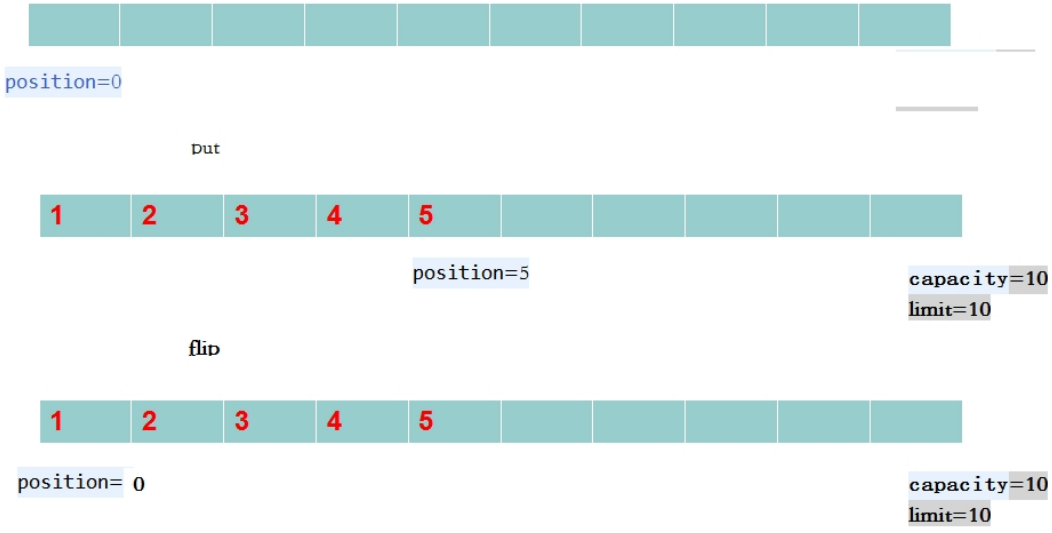
Java NIO 引入了选择器的概念，选择器用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

注意:传统 IT 是单向。 NIO 类似

IO流







区别

IO	NIO
面向流	面向缓冲区
阻塞 IO	非阻塞 IO
无	选择器

Buffer 的数据存取

一个用于特定基本数据类行的容器。有 java.nio 包定义的，所有缓冲区都是抽象类 Buffer 的子类。

Java NIO 中的 Buffer 主要用于与 NIO 通道进行交互，数据是从通道读入到缓冲区，从缓冲区写入通道中的。

Buffer 就像一个数组，可以保存多个相同类型的数据。根据类型不同（boolean 除外），有以下 Buffer 常用子类：

ByteBuffer

CharBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

DoubleBuffer

Buffer 的概述

1）容量（capacity）：表示 Buffer 最大数据容量，缓冲区容量不能为负，并且建立后不能修改。

2）限制（limit）：第一个不应该读取或者写入的数据的索引，即位于 limit 后的数据不可以读写。缓冲区的限制不能为负，并且不能大于其容量（capacity）。

3）位置（position）：下一个要读取或写入的数据的索引。缓冲区的位置不能为负，并且不能大于其限制（limit）。

4）标记（mark）与重置（reset）：标记是一个索引，通过 Buffer 中的 mark() 方法指定 Buffer 中一个特定的 position，之后可以通过调用 reset() 方法恢复到这个 position。

```
/**
 * (缓冲区)buffer 用于NIO存储数据 支持多种不同的数据类型 <br>
 * 1.byteBuffer <br>
 * 2.charBuffer <br>
 * 3.shortBuffer<br>
 * 4.IntBuffer<br>
 * 5.LongBuffer<br>
 * 6.FloatBuffer <br>
 * 7.DubooBuffer <br>
 * 上述缓冲区管理的方式 几乎<br>
 * 通过allocate () 获取缓冲区 <br>
 * 二、缓冲区核心的方法 put 存入数据到缓冲区 get <br> 获取缓冲区数据 flip 开启读模式
 * 三、缓冲区四个核心属性<br>
 * capacity:缓冲区最大容量，一旦声明不能改变。 limit:界面(缓冲区可以操作的数据大小) limit后面的数据不能读写。
 * position:缓冲区正在操作的位置
 */
public class Test004 {


	public static void main(String[] args) {
		// 1.指定缓冲区大小1024
		ByteBuffer buf = ByteBuffer.allocate(1024);
		System.out.println("--------------------");
		System.out.println(buf.position());
```

```
            System.out.println(buf.limit());
            System.out.println(buf.capacity());
            // 2.向缓冲区存放5个数据
            buf.put("abcd1".getBytes());
            System.out.println("--------------------");
            System.out.println(buf.position());
            System.out.println(buf.limit());
            System.out.println(buf.capacity());
            // 3.开启读模式
            buf.flip();
            System.out.println("----------开启读模式...----------");
            System.out.println(buf.position());
            System.out.println(buf.limit());
            System.out.println(buf.capacity());
            byte[] bytes = new byte[buf.limit()];
            buf.get(bytes);
            System.out.println(new String(bytes, 0, bytes.length));
            System.out.println("----------重复读模式...----------");
            // 4.开启重复读模式
            buf.rewind();
            System.out.println(buf.position());
            System.out.println(buf.limit());
            System.out.println(buf.capacity());
            byte[] bytes2 = new byte[buf.limit()];
            buf.get(bytes2);
            System.out.println(new String(bytes2, 0, bytes2.length));
            // 5.clean 清空缓冲区  数据依然存在,只不过数据被遗忘
            System.out.println("----------清空缓冲区...----------");
            buf.clear();
            System.out.println(buf.position());
            System.out.println(buf.limit());
            System.out.println(buf.capacity());
            System.out.println((char)buf.get());
        }

}
```

make 与 rest 用法

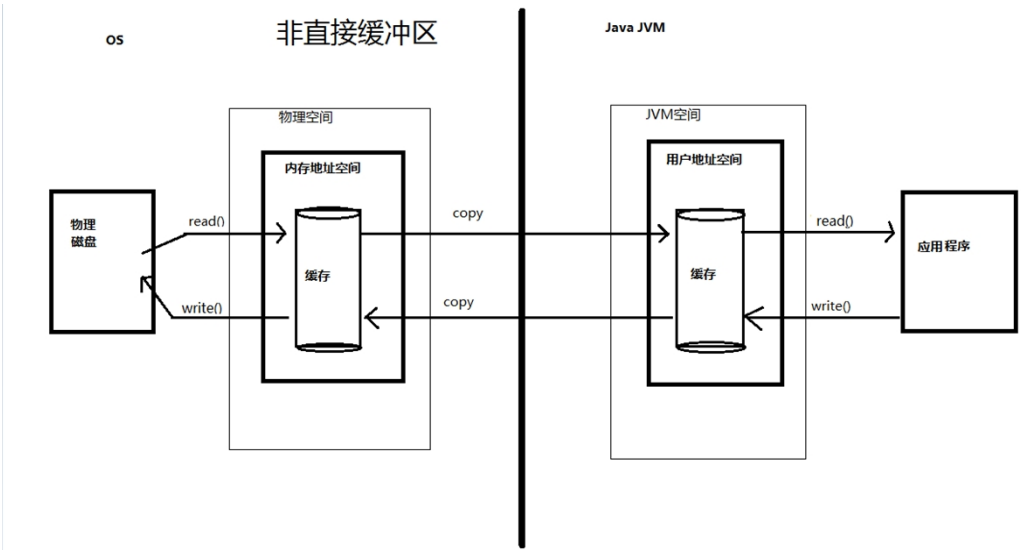
标记（mark）与重置（reset）：标记是一个索引，通过 Buffer 中的 mark() 方法指定 Buffer 中一个特定的 position，之后可以通过调用 reset() 方法恢复到这个 position。

```
public class Test002 {
```

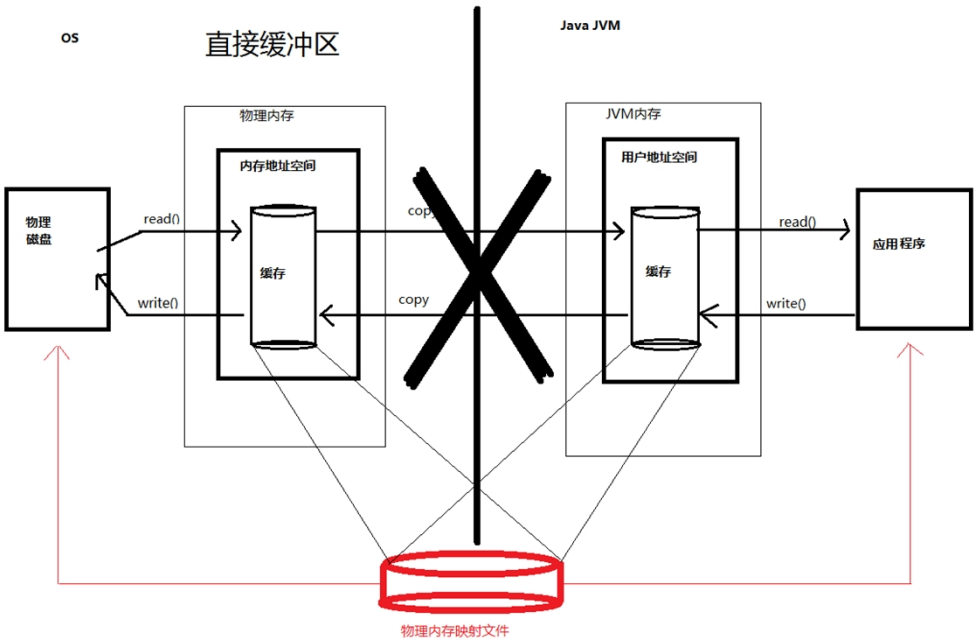
```
    public static void main(String[] args) {
        ByteBuffer buf = ByteBuffer.allocate(1024);
        String str = "abcd1";
        buf.put(str.getBytes());
        // 开启读取模式
        buf.flip();
        byte[] dst = new byte[buf.limit()];
        buf.get(dst, 0, 2);
        buf.mark();
        System.out.println(new String(dst, 0, 2));
        System.out.println(buf.position());
        buf.get(dst, 2, 2);
        System.out.println(new String(dst, 2, 2));
        System.out.println(buf.position());
        buf.reset();
        System.out.println("重置恢复到mark位置..");
        System.out.println(buf.position());
    }

}
```

直接缓冲区与非直接缓冲区别

非直接缓冲区：通过 allocate() 方法分配缓冲区，将缓冲区建立在 JVM 的内存中



直接缓冲区：通过 allocateDirect() 方法分配直接缓冲区，将缓冲区建立在物理内存中。可以提高效率



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。

```
// 使用直接缓冲区完成文件的复制(内存映射文件)
    static public void test2() throws IOException {
        long start = System.currentTimeMillis();
        FileChannel inChannel = FileChannel.open(Paths.get("f://1.mp4"), StandardOpenOption.READ);
        FileChannel outChannel = FileChannel.open(Paths.get("f://2.mp4"), StandardOpenOption.WRITE,
                StandardOpenOption.READ, StandardOpenOption.CREATE);
        // 内存映射文件
        MappedByteBuffer inMappedByteBuf = inChannel.map(MapMode.READ_ONLY, 0, inChannel.size());
        MappedByteBuffer outMappedByteBuffer = outChannel.map(MapMode.READ_WRITE, 0, inChannel.size());
        // 直接对缓冲区进行数据的读写操作
        byte[] dsf = new byte[inMappedByteBuf.limit()];
        inMappedByteBuf.get(dsf);
```

```
        outMappedByteBuffer.put(dsf);
        inChannel.close();
        outChannel.close();
        long end = System.currentTimeMillis();
        System.out.println(end - start);
    }

    // 1.利用通道完成文件的复制(非直接缓冲区)
    static public void test1() throws IOException { // 4400
        long start = System.currentTimeMillis();
        FileInputStream fis = new FileInputStream("f://1.mp4");
        FileOutputStream fos = new FileOutputStream("f://2.mp4");
        // ①获取通道
        FileChannel inChannel = fis.getChannel();
        FileChannel outChannel = fos.getChannel();
        // ②分配指定大小的缓冲区
        ByteBuffer buf = ByteBuffer.allocate(1024);
        while (inChannel.read(buf) != -1) {
            buf.flip();// 切换为读取数据
            // ③将缓冲区中的数据写入通道中
            outChannel.write(buf);
            buf.clear();
        }
        outChannel.close();
        inChannel.close();
        fos.close();
        fis.close();
        long end = System.currentTimeMillis();
        System.out.println(end - start);
    }
```

通道(Channel)的原理获取

通道表示打开到 IO 设备(例如：文件、套接字)的连接。若需要使用 NIO 系统，需要获取用于连接 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，对数据进行处理。Channel 负责传输， Buffer 负责存储。通道是由 java.nio.channels 包定义的。 Channel 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，Channel 只能与 Buffer 进行交互。

```
java.nio.channels.Channel 接口：
        |--FileChannel
        |--SocketChannel
        |--ServerSocketChannel
        |--DatagramChannel
```

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()

3. 在 JDK 1.7 中的 NIO.2 的 Files 工具类的 newByteChannel()

```
@Test
    // 使用直接缓冲区完成文件的复制(内存映射文件)
    public void test2() throws IOException {
        FileChannel inChannel = FileChannel.open(Paths.get("1.png"), StandardOpenOption.READ);
        FileChannel outChannel = FileChannel.open(Paths.get("2.png"), StandardOpenOption.READ,
StandardOpenOption.WRITE,
                StandardOpenOption.CREATE);
        // 映射文件
        MappedByteBuffer inMapperBuff = inChannel.map(MapMode.READ_ONLY, 0, inChannel.size());
        MappedByteBuffer outMapperBuff = outChannel.map(MapMode.READ_WRITE, 0, inChannel.size());
        // 直接对缓冲区进行数据读写操作
        byte[] dst = new byte[inMapperBuff.limit()];
        inMapperBuff.get(dst);
        outMapperBuff.put(dst);
        outChannel.close();
        inChannel.close();
    }

    @Test
    // 1.利用通道完成文件复制(非直接缓冲区)
    public void test1() throws IOException {
        FileInputStream fis = new FileInputStream("1.png");
        FileOutputStream fos = new FileOutputStream("2.png");
        // ①获取到通道
        FileChannel inChannel = fis.getChannel();
        FileChannel outChannel = fos.getChannel();

        // ②分配指定大小的缓冲区
        ByteBuffer buf = ByteBuffer.allocate(1024);
        while (inChannel.read(buf) != -1) {
```

```
                buf.flip();// 切换到读取模式
                outChannel.write(buf);
                buf.clear();// 清空缓冲区
            }
            // 关闭连接
            outChannel.close();
            inChannel.close();
            fos.close();
            fis.close();
        }
```

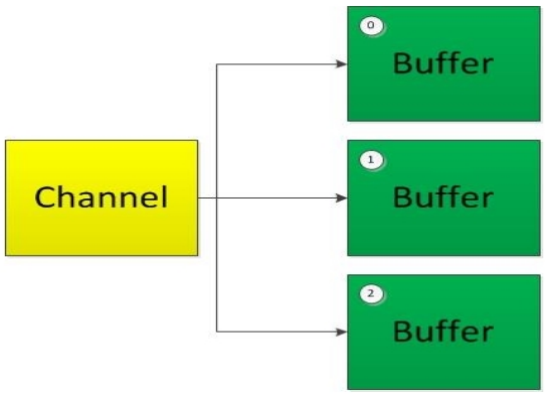
直接缓冲区与非直接缓冲耗时计算

```
    @Test
    // 使用直接缓冲区完成文件的复制(内存映射文件) //428、357
    public void test2() throws IOException {
        long startTime = System.currentTimeMillis();
        FileChannel inChannel = FileChannel.open(Paths.get("f://1.mp4"), StandardOpenOption.READ);
        FileChannel outChannel = FileChannel.open(Paths.get("f://2.mp4"), StandardOpenOption.READ,
StandardOpenOption.WRITE,
                StandardOpenOption.CREATE);
        // 映射文件
        MappedByteBuffer inMapperBuff = inChannel.map(MapMode.READ_ONLY, 0, inChannel.size());
        MappedByteBuffer outMapperBuff = outChannel.map(MapMode.READ_WRITE, 0, inChannel.size());
        // 直接对缓冲区进行数据读写操作
        byte[] dst = new byte[inMapperBuff.limit()];
        inMapperBuff.get(dst);
        outMapperBuff.put(dst);
        outChannel.close();
        inChannel.close();
        long endTime = System.currentTimeMillis();
        System.out.println("内存映射文件耗时:"+(endTime-startTime));
    }

    @Test
    // 1.利用通道完成文件复制(非直接缓冲区)
    public void test1() throws IOException {  //11953 、3207、3337
        long startTime = System.currentTimeMillis();
        FileInputStream fis = new FileInputStream("f://1.mp4");
        FileOutputStream fos = new FileOutputStream("f://2.mp4");
        // ①获取到通道
        FileChannel inChannel = fis.getChannel();
        FileChannel outChannel = fos.getChannel();
```

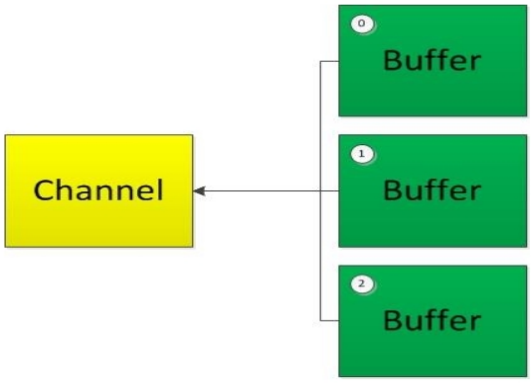
```
        // ②分配指定大小的缓冲区
        ByteBuffer buf = ByteBuffer.allocate(1024);
        while (inChannel.read(buf) != -1) {
              buf.flip();// 切换到读取模式
              outChannel.write(buf);
              buf.clear();// 清空缓冲区
        }
        // 关闭连接
        outChannel.close();
        inChannel.close();
        fos.close();
        fis.close();
        long endTime = System.currentTimeMillis();
        System.out.println("非缓冲区:"+(endTime-startTime));
  }
```

分散读取与聚集写入

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)：将多个缓冲区的数据聚集到通道中



```
RandomAccessFile raf1 = new RandomAccessFile("test.txt", "rw");

            // 1.获取通道

            FileChannel channel = raf1.getChannel();

            // 2.分配指定大小的指定缓冲区

            ByteBuffer buf1 = ByteBuffer.allocate(100);

            ByteBuffer buf2 = ByteBuffer.allocate(1024);

            // 3.分散读取

            ByteBuffer[] bufs = { buf1, buf2 };

            channel.read(bufs);

            for (ByteBuffer byteBuffer : bufs) {

                  // 切换为读取模式

                  byteBuffer.flip();

            }

            System.out.println(new String(bufs[0].array(), 0, bufs[0].limit()));

            System.out.println("------------------分算读取线分割--------------------");

            System.out.println(new String(bufs[1].array(), 0, bufs[1].limit()));

            // 聚集写入

            RandomAccessFile raf2 = new RandomAccessFile("2.txt", "rw");

            FileChannel channel2 = raf2.getChannel();

            channel2.write(bufs);
```

字符集 Charset

编码：字符串->字节数组

解码：字节数组 -> 字符串

```
public class Test005 {


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

            // 获取编码器

            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("蚂蚁课堂牛逼!");

            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();
            CharBuffer cBuf2 = cd.decode(bBuf);
            System.out.println(cBuf2.toString());
            System.out.println("------------------------------------");
            Charset cs2 = Charset.forName("GBK");
            bBuf.flip();
            CharBuffer cbeef = cs2.decode(bBuf);
            System.out.println(cbeef.toString());
        }

}
```