链滴

Java NIO（New IO）

作者：wlgzs-sjl
原文链接：https://ld246.com/article/1604150261787
来源网站：链滴

Java NIO（New IO或 Non Blocking IO）是从Java 1.4版本开始引入的一个新的IO API，可以替代准的Java IO API。NIO支持面向缓冲区的、基于通道的IO操作。NIO将以更加高效的方式进行文件的写操作。

java IO 与 java 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)

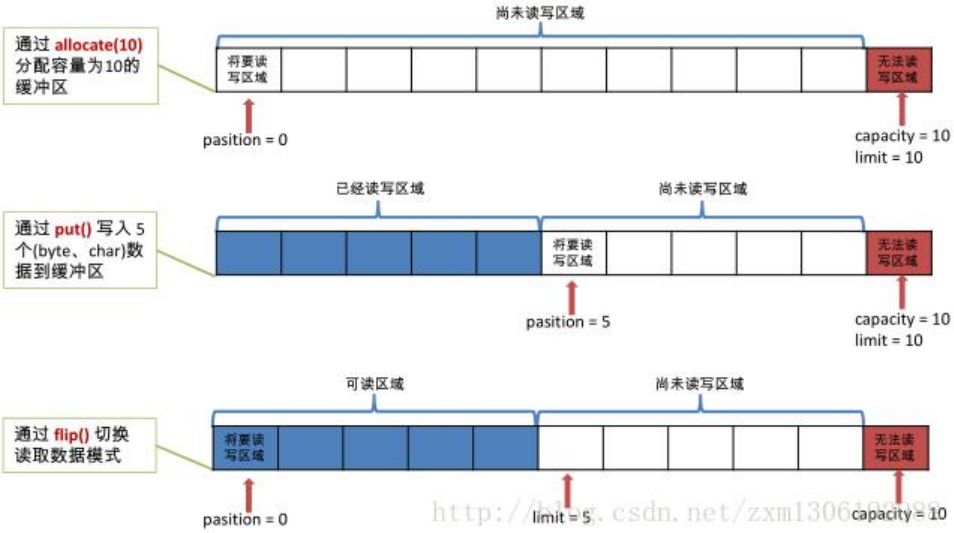
一、通道（Channel）与缓冲区（Buffer)

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，对数据进行处理。简而言之，Channel 负责传输， Buffer 负责存储。

1、缓冲区（Buffer)

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

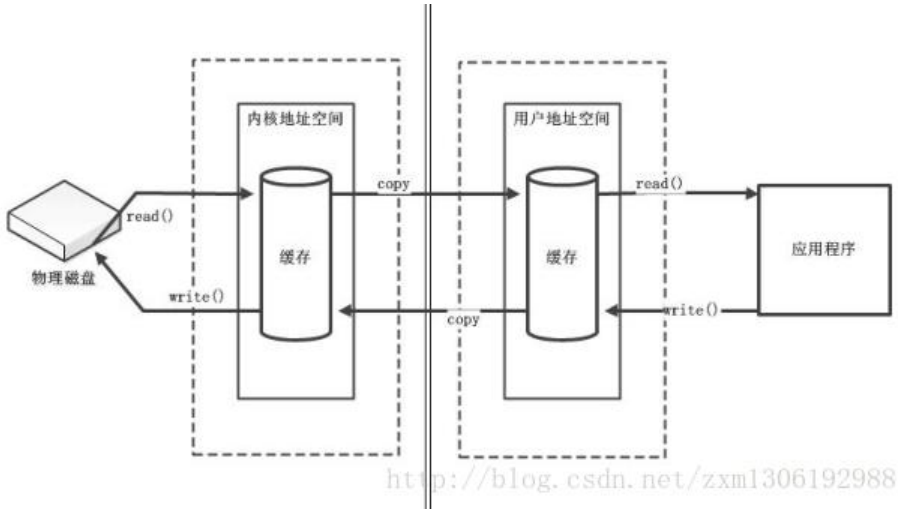
Java NIO 中的 Buffer 主要用于与 NIO 通道进行交互，数据是从通道读入缓冲区，从缓冲区写入通中的。



Buffer 的常用方法

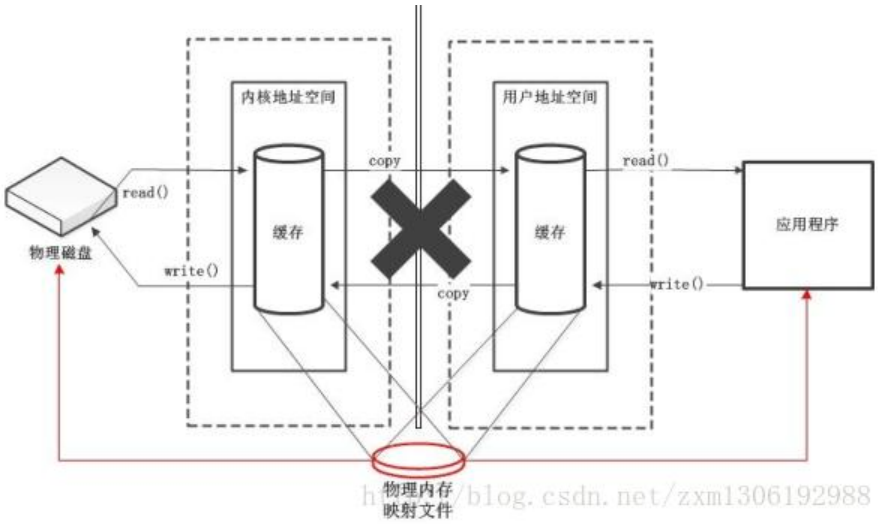
方 法	描 述
Buffer clear()	清空缓冲区并返回对缓冲区的引用
Buffer flip()	将缓冲区的界限设置为当前位置，并将当前位置充值为 0
int capacity()	返回 Buffer 的 capacity 大小
boolean hasRemaining()	判断缓冲区中是否还有元素
int limit()	返回 Buffer 的界限(limit) 的位置
Buffer limit(int n)	将设置缓冲区界限为 n，并返回一个具有新 limit 的缓冲区对象
Buffer mark()	对缓冲区设置标记
int position()	返回缓冲区的当前位置 position
Buffer position(int n)	将设置缓冲区的当前位置为 n ，并返回修改后的 Buffer 对象
int remaining()	返回 position 和 limit 之间的元素个数
Buffer reset()	将位置 position 转到以前设置的 mark 所在的位置
Buffer rewind()	将位置设为为 0， 取消设置的 mark

非直接缓冲区



直接缓冲区

原文链接：Java NIO（New IO）




```
/*
 * 一、缓冲区（Buffer）：在java NIO 中负者数据的存储。缓冲区就是数组。用于存储不同类型的数
。
 *
 * 根据数据类型的不同(boolean 除外)，有以下 Buffer 常用子类：
 * 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
 *
 * 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
 *      put(byte b)：将给定单个字节写入缓冲区的当前位置
 *      put(byte[] src)：将 src 中的字节写入缓冲区的当前位置
 *      put(int index, byte b)：将指定字节写入缓冲区的索引位置(不会移动 position)
 * get():获取缓存区中的数据
 *      get() ：读取单个字节
 *      get(byte[] dst)：批量读取多个字节到 dst 中
 *      get(int index)：读取指定索引位置的字节(不会移动 position)
 *
 * 三、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位置。可以通过reset()恢复到mark的位置。
 *
 * 0<=mark<=position<=limit<=capacity
 *
 * 四、直接缓冲区与非直接缓冲区：
 * 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

 *        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.
 *        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。
 *        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过 通过FileChannel 的 map() 方法 将文件区域直接映射到内存中来
建 。该方法返回MappedByteBuffe
 */
public class TestBuffer {
    @Test
    public void test1(){
        String str="abcde";

        //1.分配一个指定大小的缓冲区
        ByteBuffer buf=ByteBuffer.allocate(1024);

        System.out.println("--------------allocate()----------------");
        System.out.println(buf.position());//0
        System.out.println(buf.limit());//1024
        System.out.println(buf.capacity());//1024

        //2.利用put()存放数据到缓冲区中
        buf.put(str.getBytes());

        System.out.println("-------------put()-------------");
        System.out.println(buf.position());//5
        System.out.println(buf.limit());//1024
        System.out.println(buf.capacity());//1024

        //3.切换读取数据模式
        buf.flip();
        System.out.println("--------------flip()------------");
        System.out.println(buf.position());//0
        System.out.println(buf.limit());//5
        System.out.println(buf.capacity());//1024

        //4.利用get()读取缓冲区中的数据
        byte[] dst=new byte[buf.limit()];
        buf.get(dst);
        System.out.println(new String(dst,0,dst.length));//abcd

        System.out.println("--------------get()------------");
        System.out.println(buf.position());//5
        System.out.println(buf.limit());//5
        System.out.println(buf.capacity());//1024

        //5.rewind():可重复读
        buf.rewind();

        System.out.println("--------------rewind()------------");
        System.out.println(buf.position());//0
        System.out.println(buf.limit());//5
        System.out.println(buf.capacity());//1024

        //6.clear():清空缓冲区。但是缓冲区中的数据依然存在，但是处在“被遗忘”状态
        buf.clear();
```

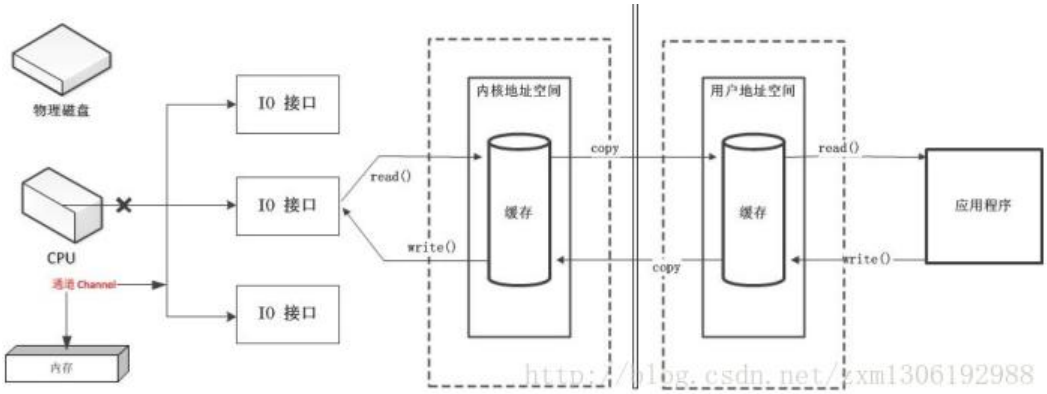
```
        System.out.println("--------------clear()------------");
        System.out.println(buf.position());//0
        System.out.println(buf.limit());//1024
        System.out.println(buf.capacity());//1024

        System.out.println((char)buf.get());
    }

    @Test
    public void test2(){
        String str="abcde";

        ByteBuffer buf=ByteBuffer.allocate(1024);

        buf.put(str.getBytes());

        buf.flip();

        byte[] dst=new byte[buf.limit()];
        buf.get(dst,0,2);
        System.out.println(new String(dst,0,2));//ab
        System.out.println(buf.position());//2

        //mark():标记
        buf.mark();

        buf.get(dst,2,2);//再读两个位置
        System.out.println(new String(dst, 2, 2));//cd
        System.out.println(buf.position());//4

        //reset():恢复到mark的位置
        buf.reset();
        System.out.println(buf.position());//2

        //判断缓冲区中是否还有剩余数据
        if(buf.hasRemaining()){
            //获取缓冲区中可以操作的数量
            System.out.println(buf.remaining());//3
        }
    }

    @Test
    public void test3(){
        //分配直接缓冲区
        ByteBuffer buf=ByteBuffer.allocate(1024);
        System.out.println(buf.isDirect());//false
    }
}
```

2、通道（Channel）

通道：由java.nio.channels包定义。

Channel表示IO源与目标打开的连接。

Channel类似于传统的“流”。但其自身不能直接访问数据，Channel只能与Buffer进行交互。



操作系统中：通道是一种通过执行通道程序管理I/O操作的控制器，它使主机（CPU和内存）与I/O操之间达到更高的并行程度。需要进行I/O操作时，CPU只需启动通道，然后可以继续执行自身程序，道则执行通道程序，管理与实现I/O操作。

FileChannel 的常用方法

方　法	描　述
int read(ByteBuffer dst)	从 Channel 中读取数据到 ByteBuffer
long read(ByteBuffer[] dsts)	将 Channel 中的数据“分散”到 ByteBuffer[]
int write(ByteBuffer src)	将 ByteBuffer 中的数据写入到 Channel
long write(ByteBuffer[] srcs)	将 ByteBuffer[] 中的数据“聚集”到 Channel
long position()	返回此通道的文件位置
FileChannel position(long p)	设置此通道的文件位置
long size()	返回此通道的文件的当前大小
FileChannel truncate(long s)	将此通道的文件截取为给定大小
void force(boolean metaData)	强制将所有对此通道的文件更新写入到存储设备中

```
/*
 * 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，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：用于读取、写入、映射和操作文件的通道。
 *        |--SocketChannel：通过 TCP 读写网络中的数据。
 *        |--ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个SocketChannel。
 *        |--DatagramChannel：通过 UDP 读写网络中的数据通道。
 *
 * 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()
 * 3.在JDK 1.7 中的NIO.2 的Files工具类的newByteChannel()
 *
 * 四、通道之间的数据传输
 * transferFrom()
 * transferTo()
 *
 * 五、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
 *
 * 六、字符集：Charset
 * 编码：字符串-》字符数组
 * 解码：字符数组-》字符串
 */
public class TestChannel {

    //利用通道完成文件的复制(非直接缓冲区)
    @Test
    public void test1(){
        long start=System.currentTimeMillis();

        FileInputStream fis=null;
        FileOutputStream fos=null;

        FileChannel inChannel=null;
        FileChannel outChannel=null;
        try{
            fis=new FileInputStream("d:/1.avi");
            fos=new FileOutputStream("d:/2.avi");

            //1.获取通道
            inChannel=fis.getChannel();
            outChannel=fos.getChannel();

            //2.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){
            e.printStackTrace();
        }finally{
            if(outChannel!=null){
                try {
```

```
                outChannel.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(inChannel!=null){
            try {
                inChannel.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(fos!=null){
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(fis!=null){
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    long end=System.currentTimeMillis();
    System.out.println("耗费时间："+(end-start));//耗费时间：1094
}

//使用直接缓冲区完成文件的复制(内存映射文件)
@Test
public void test2() {
    long start=System.currentTimeMillis();

    FileChannel inChannel=null;
    FileChannel outChannel=null;
    try {
        inChannel = FileChannel.open(Paths.get("d:/1.avi"), StandardOpenOption.READ);
        outChannel=FileChannel.open(Paths.get("d:/2.avi"), StandardOpenOption.WRITE,Stan
ardOpenOption.READ,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, inCh
nnel.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);
    } catch (IOException e) {
        e.printStackTrace();
```

```
        }finally{
            if(outChannel!=null){
                try {
                    outChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(inChannel!=null){
                try {
                    inChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        long end=System.currentTimeMillis();
        System.out.println("耗费的时间为："+(end-start));//耗费的时间为：200
    }

    //通道之间的数据传输(直接缓冲区)
    @Test
    public void test3(){
        long start=System.currentTimeMillis();

        FileChannel inChannel=null;
        FileChannel outChannel=null;
        try {
            inChannel = FileChannel.open(Paths.get("d:/1.avi"), StandardOpenOption.READ);
            outChannel=FileChannel.open(Paths.get("d:/2.avi"), StandardOpenOption.WRITE,Stan
ardOpenOption.READ,StandardOpenOption.CREATE);

            inChannel.transferTo(0, inChannel.size(), outChannel);
            outChannel.transferFrom(inChannel, 0, inChannel.size());
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            if(outChannel!=null){
                try {
                    outChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(inChannel!=null){
                try {
                    inChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        long end=System.currentTimeMillis();
```

```
    System.out.println("耗费的时间为："+(end-start));//耗费的时间为：147
}

//分散和聚集
@Test
public void test4(){
    RandomAccessFile raf1=null;
    FileChannel channel1=null;
    RandomAccessFile raf2=null;
    FileChannel channel2=null;
    try {
        raf1=new RandomAccessFile("1.txt","rw");

        //1.获取通道
        channel1=raf1.getChannel();

        //2.分配指定大小的缓冲区
        ByteBuffer buf1=ByteBuffer.allocate(100);
        ByteBuffer buf2=ByteBuffer.allocate(1024);

        //3.分散读取
        ByteBuffer[] bufs={buf1,buf2};
        channel1.read(bufs);

        for(ByteBuffer byteBuffer : bufs){
            byteBuffer.flip();
        }
        System.out.println(new String(bufs[0].array(),0,bufs[0].limit()));
        System.out.println("--------------------");
        System.out.println(new String(bufs[1].array(),0,bufs[1].limit()));

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

        channel2.write(bufs);

    }catch (IOException e) {
        e.printStackTrace();
    }finally{
        if(channel2!=null){
            try {
                channel2.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(channel1!=null){
            try {
                channel1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

```
        if(raf2!=null){
            try {
                raf2.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(raf1!=null){
            try {
                raf1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

//输出支持的字符集
@Test
public void test5(){
    Map<String,Charset> map=Charset.availableCharsets();
    Set<Entry<String,Charset>> set=map.entrySet();

    for(Entry<String,Charset> entry:set){
        System.out.println(entry.getKey()+"="+entry.getValue());
    }
}

//字符集
@Test
public void test6(){
    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("啦啦哈哈吧吧");
    cBuf.flip();

    //编码
    ByteBuffer bBuf=null;
    try {
        bBuf = ce.encode(cBuf);
    } catch (CharacterCodingException e) {
        e.printStackTrace();
    }

    for(int i=0;i<12;i++){
        System.out.println(bBuf.get());//-64-78-64-78-71-2-7-2-80-55-80-55
    }
```

```
        //解码
        bBuf.flip();
        CharBuffer cBuf2=null;
        try {
            cBuf2 = cd.decode(bBuf);
        } catch (CharacterCodingException e) {
            e.printStackTrace();
        }
        System.out.println(cBuf2.toString());//啦啦哈哈吧吧
    }
}
```

二、NIO 的非阻塞式网络通信

传统的 IO 流都是阻塞式的。也就是说，当一个线程调用 read() 或 write()时，该线程被阻塞，直到一些数据被读取或写入，该线程在此期间不能执行其他任务。因此，在完成网络通信进行 IO 操作时由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要理大量客户端时，性能急剧下降。

Java NIO 是非阻塞模式的。当线程从某通道进行读写数据时，若没有数据可用时，该线程可以进行他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管多个输入和输出通道。因此，NIO 可以让服务器端使用一个或有限几个线程来同时处理连接到服务器的所有客户端。

选择器（Selector）

选择器（Selector） 是 SelectableChannle 对象的多路复用器，Selector 可以同时监控多个SelectableChannel 的 IO 状况，也就是说，利用 Selector可使一个单独的线程管理多个 Channel。Selector 非阻塞 IO 的核心。

```
/*
 * 一、使用NIO 完成网络通信的三个核心:
 *
 * 1、通道(Channel):负责连接
 *     java.nio.channels.Channel 接口:
 *         |--SelectableChannel
 *             |--SocketChannel
 *             |--ServerSocketChannel
 *             |--DatagramChannel
 *
 *             |--Pipe.SinkChannel
 *             |--Pipe.SourceChannel
 *
 * 2.缓冲区(Buffer):负责数据的存取
 *
 * 3.选择器(Selector):是 SelectableChannel 的多路复用器。用于监控SelectableChannel的IO状况
 */
public class TestBlockingNIO {//没用Selector，阻塞型的

    //客户端
    @Test
    public void client() throws IOException{
        SocketChannel sChannel=SocketChannel.open(new InetSocketAddress("127.0.0.1",9898))
```

```
        FileChannel inChannel=FileChannel.open(Paths.get("1.jpg"), StandardOpenOption.READ);
        ByteBuffer buf=ByteBuffer.allocate(1024);
        while(inChannel.read(buf)!=-1){
            buf.flip();
            sChannel.write(buf);
            buf.clear();
        }
        sChannel.shutdownOutput();//关闭发送通道，表明发送完毕

        //接收服务端的反馈
        int len=0;
        while((len=sChannel.read(buf))!=-1){
            buf.flip();
            System.out.println(new String(buf.array(),0,len));
            buf.clear();
        }
        inChannel.close();
        sChannel.close();
    }

    //服务端
    @Test
    public void server() throws IOException{
        ServerSocketChannel ssChannel=ServerSocketChannel.open();
        FileChannel outChannel=FileChannel.open(Paths.get("2.jpg"), StandardOpenOption.WRI
E,StandardOpenOption.CREATE);
        ssChannel.bind(new InetSocketAddress(9898));
        SocketChannel sChannel=ssChannel.accept();
        ByteBuffer buf=ByteBuffer.allocate(1024);
        while(sChannel.read(buf)!=-1){
            buf.flip();
            outChannel.write(buf);
            buf.clear();
        }

        //发送反馈给客户端
        buf.put("服务端接收数据成功".getBytes());
        buf.flip();//给为读模式
        sChannel.write(buf);

        sChannel.close();
        outChannel.close();
        ssChannel.close();
    }
}
```

SelectionKey

当调用 register(Selector sel, int ops) 将通道注册选择器时，选择器对通道的监听事件，需要通过第个参数 ops 指定。

可以监听的事件类型（用 可使用 SelectionKey 的四个常量 表示）：

- 读 : SelectionKey.OP_READ （1）

- 写 : SelectionKey.OP_WRITE （4）
- 连接 : SelectionKey.OP_CONNECT （8）
- 接收 : SelectionKey.OP_ACCEPT （16）

若注册时不止监听一个事件，则可以使用“位或”操作符连接。

SelectionKey：表示 SelectableChannel 和 Selector 之间的注册关系。每次向选择器注册通道时就选择一个事件(选择键)。选择键包含两个表示为整数值的操作集。操作集的每一位都表示该键的通道支持的一类可选择操作。

方　法	描　述
int　interestOps()	获取感兴趣事件集合
int　readyOps()	获取通道已经准备就绪的操作的集合
SelectableChannel　channel()	获取注册通道
Selector　selector()	返回选择器
boolean isReadable()	检测 Channal 中读事件是否就绪
boolean isWritable()	检测 Channal 中写事件是否就绪
boolean isConnectable()	检测 Channel 中连接是否就绪
boolean isAcceptable()	检测 Channel 中接收是否就绪

Selector 的常用方法

方　法	描　述
Set<SelectionKey> keys()	所有的 SelectionKey 集合。代表注册在该Selector上的Channel
selectedKeys()	被选择的 SelectionKey 集合。返回此Selector的已选择键集
int select()	监控所有注册的Channel，当它们中间有需要处理的 IO 操作时，该方法返回，并将对应得的 SelectionKey 加入被选择的 SelectionKey 集合中，该方法返回这些 Channel 的数量。
int select(long timeout)	可以设置超时时长的 select() 操作
int selectNow()	执行一个立即返回的 select() 操作，该方法不会阻塞线程
Selector wakeup()	使一个还未返回的 select() 方法立即返回
void close()	关闭该选择器

```
public class TestNonBlockingNIO {
    //客户端
    @Test
    public void client()throws IOException{
        //1.获取通道
        SocketChannel sChannel=SocketChannel.open(new InetSocketAddress("127.0.0.1", 9898)
;
        //2.切换非阻塞模式
        sChannel.configureBlocking(false);
        //3.分配指定大小的缓冲区
        ByteBuffer buf=ByteBuffer.allocate(1024);
        //4.发送数据给服务端
        Scanner scan=new Scanner(System.in);
        while(scan.hasNext()){
            String str=scan.next();
            buf.put((new Date().toString()+"\n"+str).getBytes());
```

```
        buf.flip();
        sChannel.write(buf);
        buf.clear();
    }
    //5.关闭通道
    sChannel.close();
}

//服务端
@Test
public void server() throws IOException{
    //1.获取通道
    ServerSocketChannel ssChannel=ServerSocketChannel.open();

    //2.切换非阻塞式模式
    ssChannel.configureBlocking(false);

    //3.绑定连接
    ssChannel.bind(new InetSocketAddress(9898));

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

    //5.将通道注册到选择器上，并且指定“监听接收事件”
    ssChannel.register(selector,SelectionKey.OP_ACCEPT);

    //6.轮询式的获取选择器上已经“准备就绪”的事件
    while(selector.select()>0){

        //7.获取当前选择器中所有注册的“选择键（已就绪的监听事件）”
        Iterator<SelectionKey> it=selector.selectedKeys().iterator();

        while(it.hasNext()){
            //8.获取准备“就绪”的事件
            SelectionKey sk=it.next();

            //9.判断具体是什么时间准备就绪
            if(sk.isAcceptable()){
                //10.若“接收就绪”，获取客户端连接
                SocketChannel sChannel=ssChannel.accept();

                //11.切换非阻塞模式
                sChannel.configureBlocking(false);

                //12.将该通道注册到选择器上
                sChannel.register(selector, SelectionKey.OP_READ);
            }else if(sk.isReadable()){
                //13.获取当前选择器上“读就绪”状态的通道
                SocketChannel sChannel=(SocketChannel)sk.channel();
                //14.读取数据
                ByteBuffer buf=ByteBuffer.allocate(1024);
                int len=0;
                while((len=sChannel.read(buf))>0){
                    buf.flip();
```

```
                System.out.println(new String(buf.array(),0,len));
                buf.clear();
            }
        }
        //15.取消选择键SelectionKey
        it.remove();
    }
  }
 }
}
```

DatagramChannel

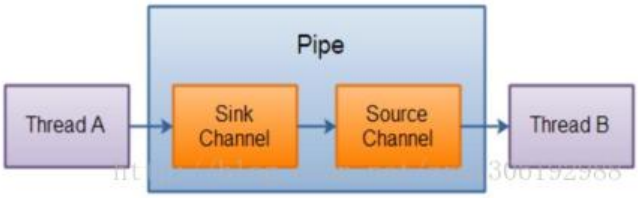
Java NIO中的DatagramChannel是一个能收发UDP包的通道。

```
public class TestNonBlockNIO2 {
    @Test
    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();
    }

    @Test
    public void receive() throws IOException{
        DatagramChannel dc=DatagramChannel.open();
        dc.configureBlocking(false);
        dc.bind(new InetSocketAddress(9898));
        Selector selector=Selector.open();
        dc.register(selector, SelectionKey.OP_READ);
        while(selector.select()>0){
            Iterator<SelectionKey> it=selector.selectedKeys().iterator();
            while(it.hasNext()){
                SelectionKey sk=it.next();

                if(sk.isReadable()){
                    ByteBuffer buf=ByteBuffer.allocate(1024);
                    dc.receive(buf)
;
                    buf.flip();
                    System.out.println(new String(buf.array(),0,buf.limit()));
                    buf.clear();
                }
            }
            it.remove();
        }
```

```
    }
}
```

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sink通道，从source通道读取。




```
public class TestPipe {
    @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();
    }
}
```

三、NIO.2 – Path 、Paths 、Files

Path 与 Paths

- java.nio.file.Path 接口代表一个平台无关的平台路径，描述了目录结构中文件的位置。
- Paths 提供的 get() 方法用来获取 Path 对象：Path get(String first, String ... more) : 用于将多个符串串连成路径。
- Path 常用方法：
    - boolean endsWith(String path) : 判断是否以 path 路径结束
    - boolean startsWith(String path) : 判断是否以 path 路径开始
    - boolean isAbsolute() : 判断是否是绝对路径
    - Path getFileName() : 返回与调用 Path 对象关联的文件名
    - Path getName(int idx) : 返回的指定索引位置 idx 的路径名称

- int getNameCount() : 返回Path 根目录后面元素的数量
  - Path getParent() ： 返回Path对象包含整个路径，不包含Path 对象指定的文件路径
  - Path getRoot() ： 返回调用 Path 对象的根路径
  - Path resolve(Path p) :将相对路径解析为绝对路径
  - Path toAbsolutePath() : 作为绝对路径返回调用 Path 对象
  - String toString() ： 返回调用 Path 对象的字符串表示形式

Files 类

java.nio.file.Files 用于操作文件或目录的工具类。

- Files常用方法:
  - Path copy(Path src, Path dest, CopyOption ... how) : 文件的复制
  - Path createDirectory(Path path, FileAttribute< ? > ... attr) : 创建一个目录
  - Path createFile(Path path, FileAttribute< ? > ... arr) : 创建一个文件
  - void delete(Path path) : 删除一个文件
  - Path move(Path src, Path dest, CopyOption...how) : 将 src 移动到 dest 位置
  - long size(Path path) : 返回 path 指定文件的大小
- Files常用方法：用于判断
  - boolean exists(Path path, LinkOption ... opts) : 判断文件是否存在
  - boolean isDirectory(Path path, LinkOption ... opts) : 判断是否是目录
  - boolean isExecutable(Path path) : 判断是否是可执行文件
  - boolean isHidden(Path path) : 判断是否是隐藏文件
  - boolean isReadable(Path path) : 判断文件是否可读
  - boolean isWritable(Path path) : 判断文件是否可写
  - boolean notExists(Path path, LinkOption ... opts) : 判断文件是否不存在
  - public static < A extends BasicFileAttributes> A readAttributes(Path path,Class< A > type LinkOption...options) : 获取与 path 指定的文件相关联的属性。
- Files常用方法：用于操作内容
  - SeekableByteChannel newByteChannel(Path path, OpenOption...how) : 获取与指定文件连接，how 指定打开方式。
  - DirectoryStream newDirectoryStream(Path path) : 打开 path 指定的目录
  - InputStream newInputStream(Path path, OpenOption...how):获取 InputStream 对象
  - OutputStream newOutputStream(Path path, OpenOption...how) : 获取 OutputStream 对象