

## 一、Java NIO 简介

- Java NIO（New IO）是从Java 1.4版本开始引入的一个新的IO API，可以替代标准的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支持面向缓冲区的、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。

## 二、Java NIO 与 IO 的主要区别

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)

## 三、缓冲区(Buffer)和通道(Channel)

- Java NIO系统的核心在于：通道(Channel)和缓冲区(Buffer)。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。

简而言之，**Channel** 负责传输，**Buffer** 负责存储

# 1、缓冲区 (Buffer)

---

## (1) 缓冲区概述及常用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。根据数据类型不同(boolean 除外)，有以下 **Buffer** 常用子类：
  - `ByteBuffer`
  - `CharBuffer`
  - `ShortBuffer`
  - `IntBuffer`
  - `LongBuffer`
  - `FloatBuffer`
  - `DoubleBuffer`

上述 **Buffer** 类 他们都采用相似的方法进行管理数据，只是各自管理的数据类型不同而已。都是通过如下方法获取一个 **Buffer** 对象：

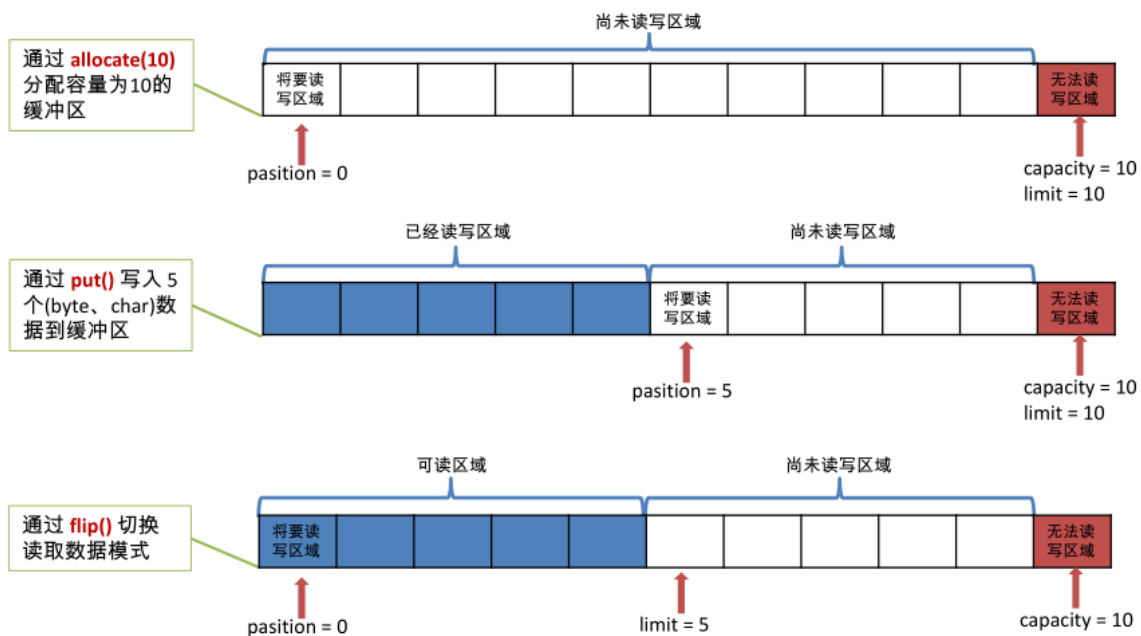
`static XxxBuffer allocate(int capacity)` : 创建一个容量为 `capacity` 的 `XxxBuffer` 对象

## (2) 缓冲区的基本属性

## ● Buffer 中的重要概念:

- **容量 (capacity):** 表示 Buffer 最大数据容量，缓冲区容量不能为负，并且创建后不能更改。
- **限制 (limit):** 第一个不应该读取或写入的数据的索引，即位于 limit 后的数据不可读写。缓冲区的限制不能为负，并且不能大于其容量。
- **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
- **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.

- 标记、位置、限制、容量遵守以下不变式:  $0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$



## (3) Buffer 的常用方法

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

## (4) 缓冲区的数据操作

### ● Buffer 所有子类提供了两个用于数据操作的方法：get() 与 put() 方法

#### ➤ 获取 Buffer 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)

#### ➤ 放入数据到 Buffer 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)

```

@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() 存入数据到缓冲区中
    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)); // abcde

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，但是处于“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()); // a

}

```

```

@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);
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
}
}

```

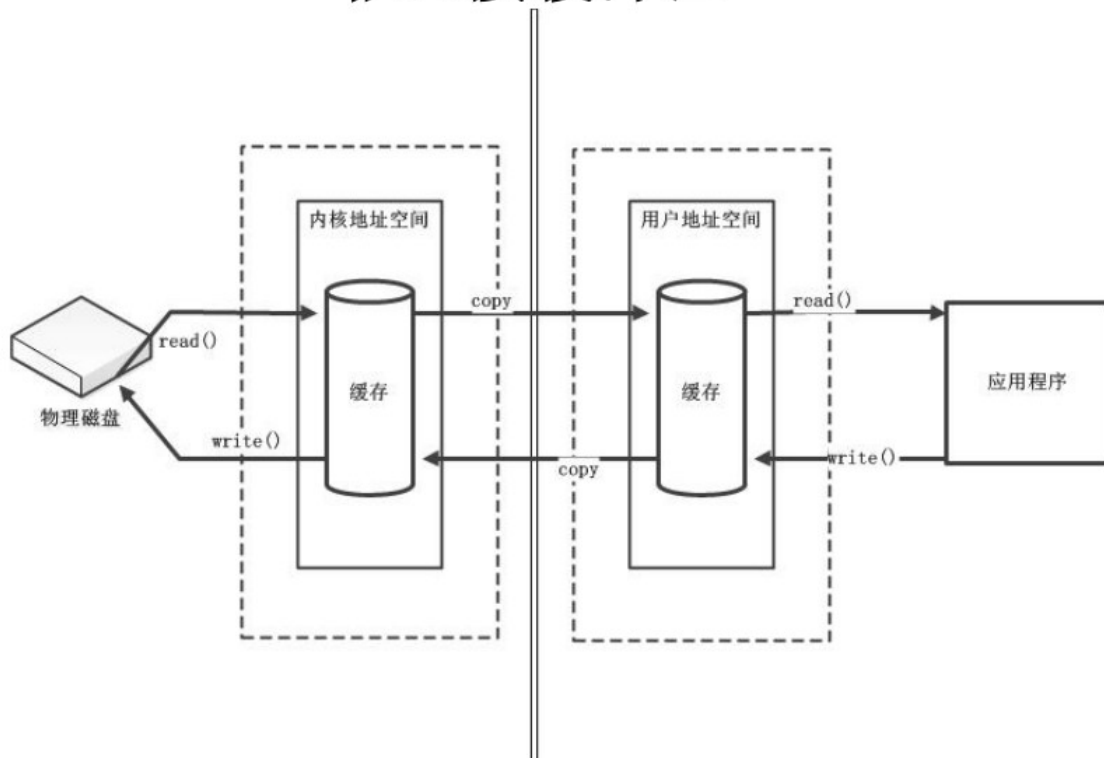
## (5) 直接与非直接缓冲区

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

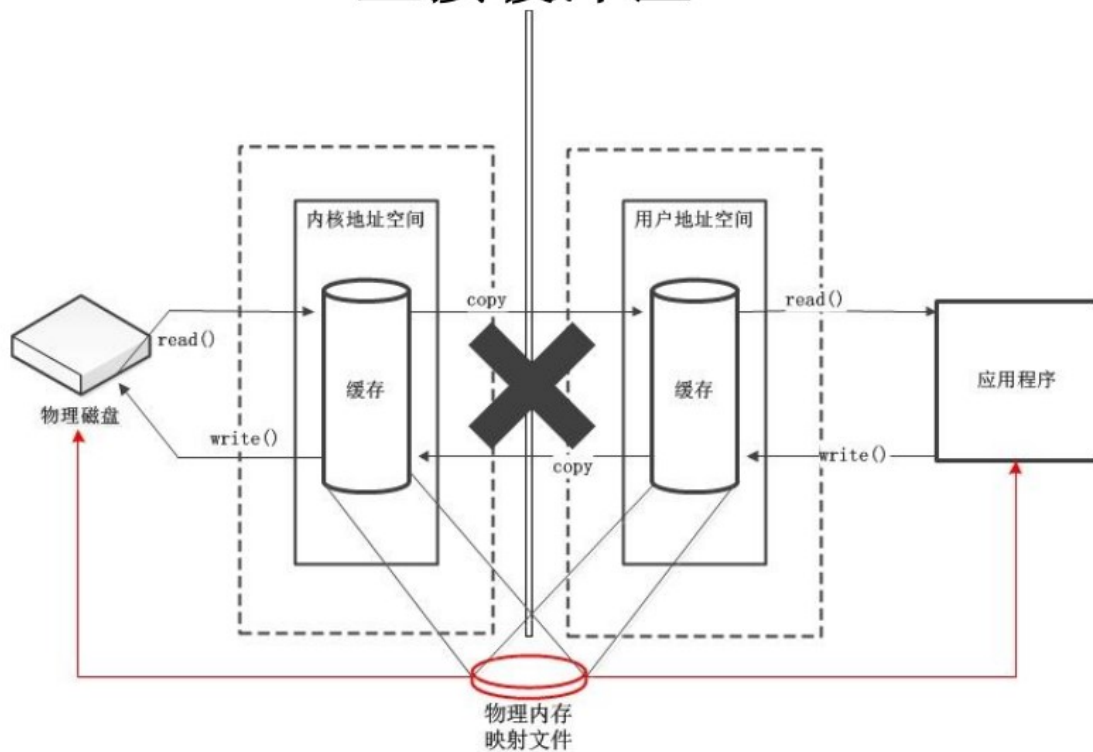
直接缓冲区：通过 `allocateDirect()` 方法分配直接缓冲区，将缓冲区建立在物理内存中。可以提高效率

- 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。

## 非直接缓冲区



## 直接缓冲区



```
@Test
public void test3(){
    //分配直接缓冲区
    ByteBuffer buf = ByteBuffer.allocateDirect(1024);

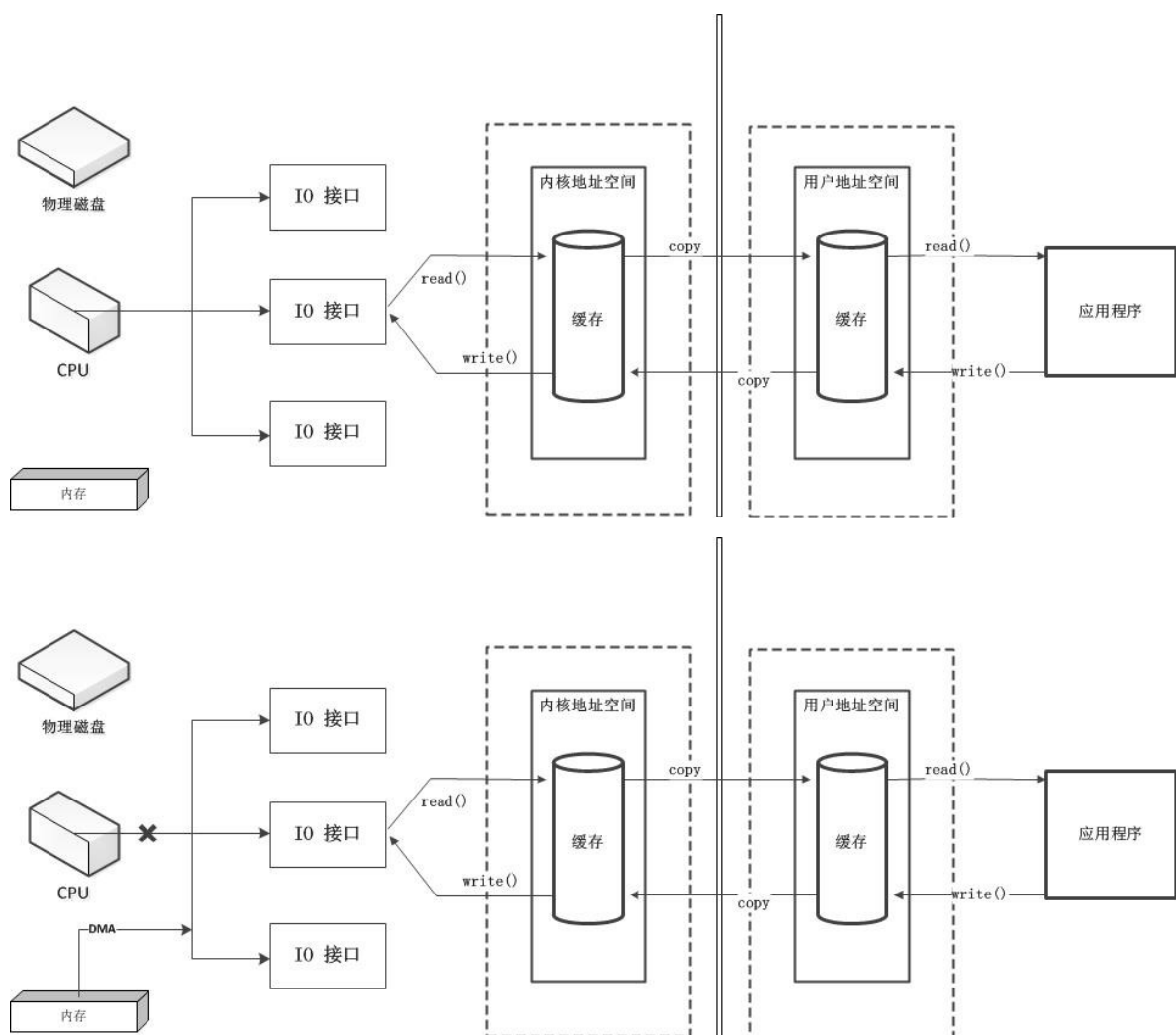
    System.out.println(buf.isDirect()); // true
}
```

## 2、通道(Channel)

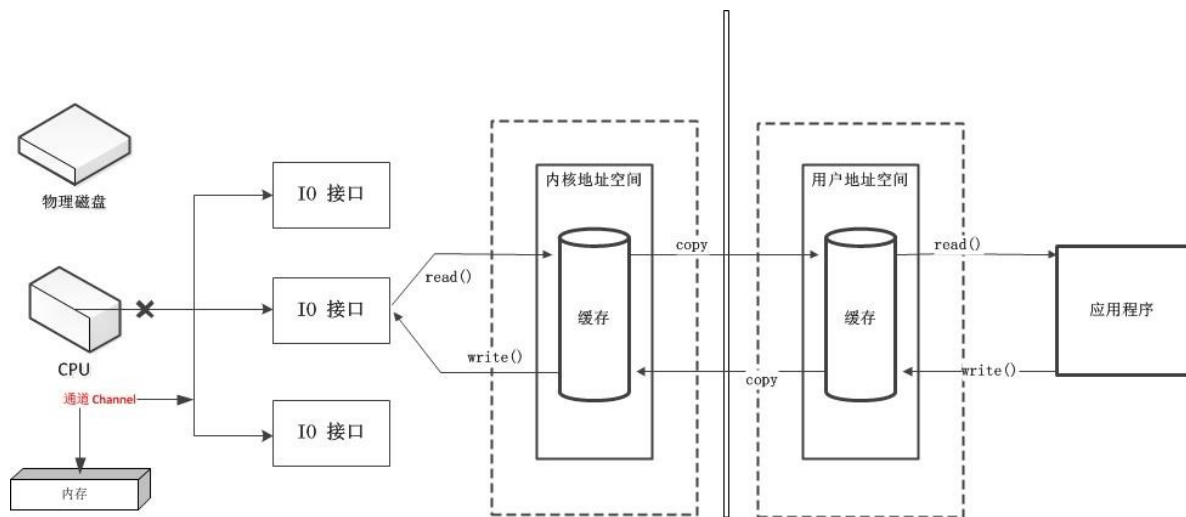
### (1) 通道概述

- 通道 (Channel)：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。







## (2) 通道的主要实现

### ● Java 为 Channel 接口提供的最主要实现类如下：

- **FileChannel**：用于读取、写入、映射和操作文件的通道。
- **DatagramChannel**：通过 UDP 读写网络中的数据通道。
- **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。

## (3) 获取通道

### ● 获取通道的一种方式是对支持通道的对象调用 `getChannel()` 方法。支持通道的类如下：

- `FileInputStream`
- `FileOutputStream`
- `RandomAccessFile`
- `DatagramSocket`
- `Socket`
- `ServerSocket`

获取通道的其他方式是使用 `Files` 类的静态方法 `newByteChannel()` 获取字节通道。或者通过通道的静态方法 `open()` 打开并返回指定通道。

## (4) 通道的数据传输

- 将 Buffer 中数据写入 Channel

例如：

```
//将 Buffer 中数据写入 Channel 中。  
int bytesWritten = inChannel.write(buf);
```

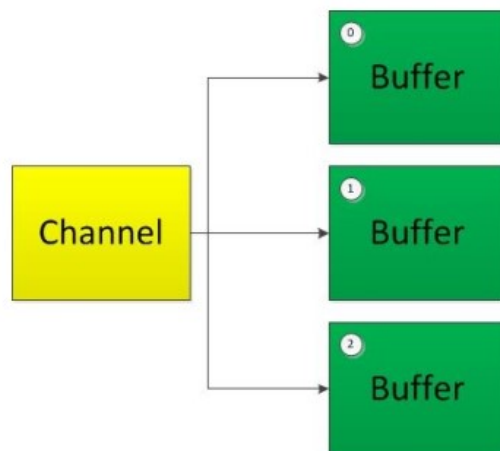
- 从 Channel 读取数据到 Buffer

例如：

```
//从 Channel 读取数据到 Buffer 中。  
int bytesRead = inChannel.read(buf);
```

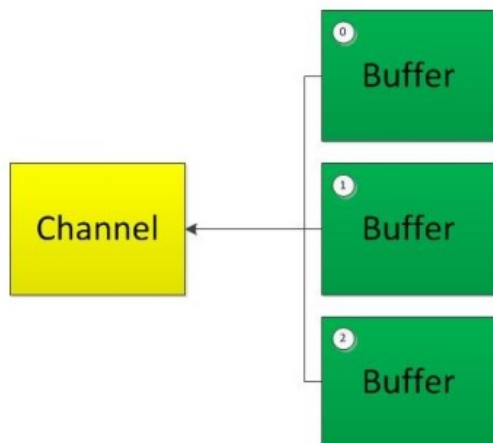
### (5) 分散 (Scatter) 和聚集 (Gather)

- 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。

```
//分散和聚集
@Test
public void test4() throws IOException{
    RandomAccessFile raf1 = new RandomAccessFile("1.txt", "rw");

    //1. 获取通道
    FileChannel 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
    RandomAccessFile raf2 = new RandomAccessFile("2.txt", "rw");
    FileChannel channel2 = raf2.getChannel();

    channel2.write(bufs);
}
```

## (6) transferFrom()

- 将数据从源通道传输到其他 Channel 中：

```
RandomAccessFile fromFile = new RandomAccessFile("data/fromFile.txt", "rw");  
//获取 FileChannel  
FileChannel fromChannel = fromFile.getChannel();  
  
RandomAccessFile toFile = new RandomAccessFile("data/toFile.txt", "rw");  
FileChannel toChannel = toFile.getChannel();  
  
//定义传输位置  
long position = 0L;  
  
//最多传输的字节数  
long count = fromChannel.size();  
  
//将数据从源通道传输到另一个通道  
toChannel.transferFrom(fromChannel, count, position);
```

```
//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.READ, StandardOpenOption.CREATE);  
  
    //    inChannel.transferTo(0, inChannel.size(), outChannel);  
    outChannel.transferFrom(inChannel, 0, inChannel.size());  
  
    inChannel.close();  
    outChannel.close();  
}
```

## (7) transferTo()

- 将数据从源通道传输到其他 Channel 中：

```
RandomAccessFile fromFile = new RandomAccessFile("data/fromFile.txt", "rw");  
//获取 FileChannel  
FileChannel fromChannel = fromFile.getChannel();  
  
RandomAccessFile toFile = new RandomAccessFile("data/toFile.txt", "rw");  
FileChannel toChannel = toFile.getChannel();  
  
//定义传输位置  
long position = 0L;  
  
//最多传输的字节数  
long count = fromChannel.size();  
  
//将数据从源通道传输到另一个通道  
fromChannel.transferTo(position, count, toChannel);
```

## (8) 字符集Charset

编码：字符串 -> 字节数组

解码：字节数组 -> 字符串

```
@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());
    }
}
```

```
Big5=Big5
Big5-HKSCS=Big5-HKSCS
CESU-8=CESU-8
EUC-JP=EUC-JP
EUC-KR=EUC-KR
GB18030=GB18030
GB2312=GB2312
GBK=GBK
IBM-Thai=IBM-Thai
IBM00858=IBM00858
IBM01140=IBM01140
IBM01141=IBM01141
IBM01142=IBM01142
IBM01143=IBM01143
IBM01144=IBM01144
IBM01145=IBM01145
IBM01146=IBM01146
IBM01147=IBM01147
IBM01148=IBM01148
IBM01149=IBM01149
IBM027=IBM027
Statistic
```

```
//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();
    CharBuffer cBuf2 = cd.decode(bBuf);
    System.out.println(cBuf2.toString());

    System.out.println("-----");

    Charset cs2 = Charset.forName("GBK");
    bBuf.flip();
    CharBuffer cBuf3 = cs2.decode(bBuf);
    System.out.println(cBuf3.toString());
}
```

-55

-48

-71

-24

-71

-56

-51

-2

-50

-28

-93

-95

尚硅谷威武!

-----

尚硅谷威武!

## 四、文件通道(FileChannel)

---

# FileChannel 的常用方法

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

```
//利用通道完成文件的复制（非直接缓冲区）
@Test
public void test1(){//10874-10953
    long start = System.currentTimeMillis();

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

        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("耗费时间为: " + (end - start));

}

```

```

//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{//2127-1902-1777
    long start = System.currentTimeMillis();

    FileChannel inChannel = FileChannel.open(Paths.get("d:/1.mkv"),
StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get("d:/2.mkv"),
StandardOpenOption.WRITE, StandardOpenOption.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,
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);

    inChannel.close();
}

```



```
outChannel.close();

long end = System.currentTimeMillis();
System.out.println("耗时时间为: " + (end - start));
}
```

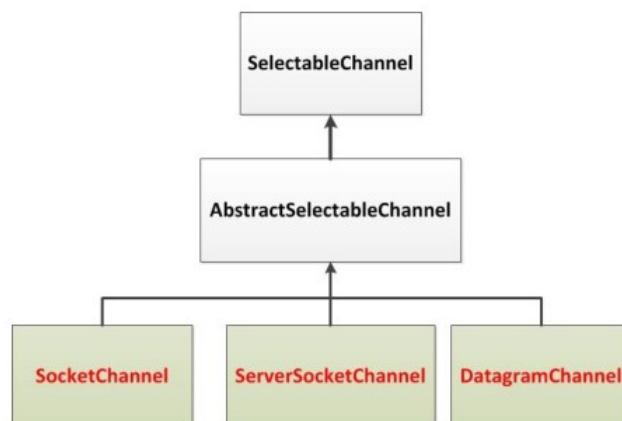
## 五、NIO 的非阻塞式网络通信

### 1、阻塞与非阻塞

- 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，直到有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，当服务器端需要处理大量客户端时，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，该线程可以进行其他任务。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理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端的所有客户端。

### 2、选择器 (Selector)

- 选择器 (Selector) 是 `SelectableChannel` 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 是非阻塞 IO 的核心。
- `SelectableChannel` 的结构如下图：



### 3、选择器 (Selector) 的应用

- 创建 Selector：通过调用 `Selector.open()` 方法创建一个 Selector。

```
//创建选择器。
Selector selector = Selector.open();
```

- 向选择器注册通道：`SelectableChannel.register(Selector sel, int ops)`

```
//创建一个 Socket 套接字。
Socket socket = new Socket(InetAddress.getByName("127.0.0.1"), 9898);

//获取 SocketChannel
SocketChannel channel = socket.getChannel();

//创建选择器。
Selector selector = Selector.open();

//将 SocketChannel 切换到非阻塞模式。
channel.configureBlocking(false);

//向 Selector 注册 Channel。
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

## 4、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，需要通过第二个参数 `ops` 指定。
- 可以监听的事件类型（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。

例：

```
//注册“监听事件”。
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

- **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会选择一个事件(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所支持的一类可选择操作。

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

## 5、Selector的常用方法

方 法	描 述
<code>Set&lt;SelectionKey&gt; keys()</code>	所有的 SelectionKey 集合。代表注册在该Selector上的Channel
<code>selectedKeys()</code>	被选择的 SelectionKey 集合。返回此Selector的已选择键集
<code>int select()</code>	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。
<code>int select(long timeout)</code>	可以设置超时时长的 select() 操作
<code>int selectNow()</code>	执行一个立即返回的 select() 操作，该方法不会阻塞线程
<code>Selector wakeup()</code>	使一个还未返回的 select() 方法立即返回
<code>void close()</code>	关闭该选择器

## 6、SocketChannel

- Java NIO中的SocketChannel是一个连接到TCP网络套接字的通道。

- 操作步骤:

- 打开 SocketChannel
- 读写数据
- 关闭 SocketChannel

## 7、ServerSocketChannel

---

- Java NIO中的 ServerSocketChannel 是一个可以监听新进来的TCP连接的通道，就像标准IO中的ServerSocket一样。

### 阻塞式:

```
package com.wl.java.nio2;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

import org.junit.Test;

/*
 * 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 {

    //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));

        FileChannel inChannel = FileChannel.open(Paths.get("1.jpg"),
        StandardOpenOption.READ);

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

        //3. 读取本地文件，并发送到服务端
        while(inChannel.read(buf) != -1){
            buf.flip();
            sChannel.write(buf);
            buf.clear();
        }

        //4. 关闭通道
        inChannel.close();
        sChannel.close();
    }

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

        FileChannel outChannel = FileChannel.open(Paths.get("2.jpg"),
        StandardOpenOption.WRITE, StandardOpenOption.CREATE);

        //2. 绑定连接
        ssChannel.bind(new InetSocketAddress(9898));

        //3. 获取客户端连接的通道
        SocketChannel sChannel = ssChannel.accept();

        //4. 分配指定大小的缓冲区
        ByteBuffer buf = ByteBuffer.allocate(1024);

        //5. 接收客户端的数据，并保存到本地
        while(sChannel.read(buf) != -1){
            buf.flip();
            outChannel.write(buf);
            buf.clear();
        }
    }
}

```

```

        //6. 关闭通道
        sChannel.close();
        outChannel.close();
        ssChannel.close();

    }

}

```

```

package com.wl.java.nio2;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

import org.junit.Test;

public class TestBlockingNIO2 {

    //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();

        //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.WRITE, 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();
    sChannel.write(buf);

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

```

## 非阻塞式:

```

package com.wl.java.nio2;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Date;
import java.util.Iterator;
import java.util.Scanner;

import org.junit.Test;

/*
 * 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 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”的是事件
        SelectionKey sk = it.next();

        //9. 判断具体是什么事件准备就绪
        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();
    }
}
}
}

```

## 8、DatagramChannel

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

- 操作步骤:

- 打开 DatagramChannel

- 接收/发送数据

**非阻塞式:**

```
package com.wl.java.nio2;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.util.Date;
import java.util.Iterator;
import java.util.Scanner;

import org.junit.Test;

public class TestNonBlockingNIO2 {

    @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();
        }
    }

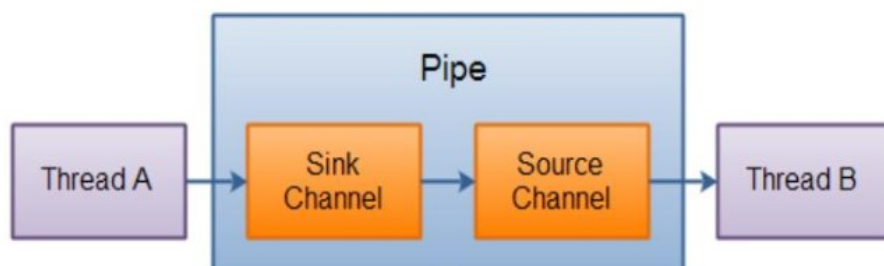
    it.remove();
}
}
}

```

## 六、管道(Pipe)

### 1、管道概述

- 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。



### 2、向管道写数据

```

@Test
public void test1() throws IOException{
    String str = "测试数据";

    //创建管道
    Pipe pipe = Pipe.open();

    //向管道写输入
    Pipe.SinkChannel sinkChannel = pipe.sink();

    //通过 SinkChannel 的 write() 方法写数据
    ByteBuffer buf = ByteBuffer.allocate(1024);
    buf.clear();
    buf.put(str.getBytes());
    buf.flip();

    while(buf.hasRemaining()){
        sinkChannel.write(buf);
    }
}

```

### 3、从管道读取数据

- 从读取管道的数据，需要访问source通道。

```

//从管道读取数据
Pipe.SourceChannel sourceChannel = pipe.source();

```

- 调用source通道的read()方法来读取数据

```

//调用 SourceChannel 的 read() 方法读取数据
ByteBuffer buf = ByteBuffer.allocate(1024);
sourceChannel.read(buf);

```

```

@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)); // 通过单向管道发送数据

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

## 七、Java NIO2 (Path、Paths 与 Files )

### 1、Java NIO概述

- Java NIO (New IO, Non-Blocking IO)是从Java 1.4版本开始引入的一套新的IO API，可以替代标准的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支持面向缓冲区的(IO是面向流的)、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。**
- Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

➤|----java.nio.channels.Channel

✓|----FileChannel:处理本地文件

✓|----SocketChannel: TCP网络编程的客户端的Channel

✓|----ServerSocketChannel:TCP网络编程的服务器端的Channel

✓|----DatagramChannel: UDP网络编程中发送端和接收端的Channel

### 2、NIO. 2

- 随着 JDK 7 的发布，Java对NIO进行了极大的扩展，增强了对文件处理和文件系统特性的支持，以至于我们称他们为 NIO.2。因为 NIO 提供的一些功能，NIO已经成为文件处理中越来越重要的部分。

### 3、Path 、 Paths 和Files 核心API

- 早期的Java只提供了一个File类来访问文件系统，但File类的功能比较有限，所提供的方法性能也不高。而且，大多数方法在出错时仅返回失败，并不会提供异常信息。
- NIO. 2为了弥补这种不足，引入了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。Path可以看成是File类的升级版本，实际引用的资源也可以不存在。
- 在以前IO操作都是这样写的：

```
import java.io.File;
File file = new File("index.html");
```
- 但在Java7 中，我们可以这样写：

```
import java.nio.file.Path;
import java.nio.file.Paths;
Path path = Paths.get("index.html");
```

让天下没有难懂的技术
- 同时，NIO.2在java.nio.file包下还提供了Files、Paths工具类，Files包含了大量静态的工具方法来操作文件；Paths则包含了两个返回Path的静态工厂方法。
- Paths 类提供的静态 get() 方法用来获取 Path 对象：
  - static Path get(String first, String ... more) : 用于将多个字符串串连成路径
  - static Path get(URI uri): 返回指定uri对应的Path路径

## 4、Path接口

### ● Path 常用方法:

- String toString() : 返回调用 Path 对象的字符串表示形式
- boolean startsWith(String path) : 判断是否以 path 路径开始
- boolean endsWith(String path) : 判断是否以 path 路径结束
- boolean isAbsolute() : 判断是否是绝对路径
- 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 getFileName() : 返回与调用 Path 对象关联的文件名
- int getNameCount() : 返回Path 根目录后面元素的数量
- Path getName(int idx) : 返回指定索引位置 idx 的路径名称
- Path toAbsolutePath() : 作为绝对路径返回调用 Path 对象
- Path resolve(Path p) : 合并两个路径，返回合并后的路径对应的Path对象
- File toFile() : 将Path转化为File类的对象

```
//如何使用Paths实例化Path
@Test
public void test1() {
    Path path1 = Paths.get("d:\\nio\\hello.txt");//new File(String filepath)

    Path path2 = Paths.get("d:\\", "nio\\hello.txt");//new File(String
parent,String filename);

    System.out.println(path1);// d:\nio\hello.txt
    System.out.println(path2);// d:\nio\hello.txt

    Path path3 = Paths.get("d:\\", "nio");
```

```

        System.out.println(path3); // d:\nio
    }

    //Path中的常用方法
    @Test
    public void test2() {
        Path path1 = Paths.get("d:\\", "nio\\nio1\\nio2\\hello.txt");
        Path path2 = Paths.get("hello.txt");

        // String toString() : 返回调用 Path 对象的字符串表示形式
        System.out.println(path1); // d:\nio\nio1\nio2\hello.txt

        // boolean startsWith(String path) : 判断是否以 path 路径开始
        System.out.println(path1.startsWith("d:\\nio")); // true
        // boolean endsWith(String path) : 判断是否以 path 路径结束
        System.out.println(path1.endsWith("hello.txt")); // true
        // boolean isAbsolute() : 判断是否是绝对路径
        System.out.println(path1.isAbsolute() + "~"); // true~
        System.out.println(path2.isAbsolute() + "~"); // false~
        // 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
        System.out.println(path1.getParent()); // d:\nio\nio1\nio2
        System.out.println(path2.getParent()); // null
        // Path getRoot() : 返回调用 Path 对象的根路径
        System.out.println(path1.getRoot()); // d:\
        System.out.println(path2.getRoot()); // null
        // Path getFileName() : 返回与调用 Path 对象关联的文件名
        System.out.println(path1.getFileName() + "~"); // hello.txt~
        System.out.println(path2.getFileName() + "~"); // hello.txt~
        // int getNameCount() : 返回Path 根目录后面元素的数量
        // Path getName(int idx) : 返回指定索引位置 idx 的路径名称
        for (int i = 0; i < path1.getNameCount(); i++) {
            System.out.println(path1.getName(i) + "*****");
            // nio*****
            // nio1*****
            // nio2*****
            // hello.txt*****
        }

        // Path toAbsolutePath() : 作为绝对路径返回调用 Path 对象
        System.out.println(path1.toAbsolutePath()); // d:\nio\nio1\nio2\hello.txt
        System.out.println(path2.toAbsolutePath()); // E:\学习资料\尚硅谷全套\尚硅谷
        Java学科全套教程\1. 尚硅谷全套JAVA教程--基础阶段\尚硅谷Java核心基础_2019年版-----重点
        \课件笔记源码资料\5_代码\第2部分: Java高级编程\JavaSenior\day10\hello.txt
        // Path resolve(Path p) : 合并两个路径, 返回合并后的路径对应的Path对象
        Path path3 = Paths.get("d:\\", "nio");
        Path path4 = Paths.get("nioo\\hi.txt");
        path3 = path3.resolve(path4);
        System.out.println(path3); // d:\nio\nioo\hi.txt

        // File toFile() : 将Path转化为File类的对象
        File file = path1.toFile(); // Path--->File的转换

        Path newPath = file.toPath(); // File--->Path的转换
    }

```



## 5、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<?> ... attr)` : 创建一个文件
  - `void delete(Path path)` : 删除一个文件/目录, 如果不存在, 执行报错
  - `void deleteIfExists(Path path)` : `Path`对应的文件/目录如果存在, 执行删除
  - `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 isRegularFile(Path path, LinkOption ... opts)` : 判断是否是文件
  - `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
- **Files常用方法: 用于操作内容**
  - `SeekableByteChannel newByteChannel(Path path, OpenOption...how)` : 获取与指定文件的连接, `how` 指定打开方式。
  - `DirectoryStream<Path> 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

```
/**
 * Files工具类的使用: 操作文件或目录的工具类
 * @author shkstart
 * @create 2019 下午 2:44
 */
public class FilesTest {

    @Test
    public void test1() throws IOException{
        Path path1 = Paths.get("d:\\", "hello.txt");
        Path path2 = Paths.get("atguigu.txt");
        Path path22 = Paths.get("d:\\home", "atguigu.txt");

        // Path copy(Path src, Path dest, CopyOption ... how) : 文件的复制
        //要想复制成功, 要求path1对应的物理上的文件存在。path1对应的文件没有要求。
        Files.copy(path1, path2, StandardCopyOption.REPLACE_EXISTING);

        // Path createDirectory(Path path, FileAttribute<?> ... attr) : 创建一个目录
        //要想执行成功, 要求path对应的物理上的文件目录不存在。一旦存在, 抛出异常。
        Path path3 = Paths.get("d:\\nio\\nio1");
        // Files.createDirectory(path3);// 只能一级一级创建, d:\\nio必须存在且nio1不存在才
        //不会抛异常
    }
}
```



```

//      Path createFile(Path path, FileAttribute<?> ... arr) : 创建一个文件
//      要想执行成功, 要求path对应的物理上的文件不存在。一旦存在, 抛出异常。
Path path4 = Paths.get("d:\\nio\\hi.txt");
//      Files.createFile(path4);

//      void delete(Path path) : 删除一个文件/目录, 如果不存在, 执行报错
//      Files.delete(path4);

//      void deleteIfExists(Path path) : Path对应的文件/目录如果存在, 执行删除. 如果不存在, 正常执行结束
Files.deleteIfExists(path3);

//      Path move(Path src, Path dest, CopyOption... how) : 将 src 移动到 dest 位置
//      要想执行成功, src对应的物理上的文件需要存在, dest对应的文件没有要求。
Files.move(path1, path22, StandardCopyOption.ATOMIC_MOVE);

//      long size(Path path) : 返回 path 指定文件的大小
long size = Files.size(path2);
System.out.println(size); // 53

}

@Test
public void test2() throws IOException{
    Path path1 = Paths.get("d:\\", "hello.txt");
    Path path2 = Paths.get("atguigu.txt");
//      boolean exists(Path path, LinkOption ... opts) : 判断文件是否存在
    System.out.println(Files.exists(path2, LinkOption.NOFOLLOW_LINKS)); //
true

//      boolean isDirectory(Path path, LinkOption ... opts) : 判断是否是目录
//      不要要求此path对应的物理文件存在。
    System.out.println(Files.isDirectory(path1, LinkOption.NOFOLLOW_LINKS));
// false

//      boolean isRegularFile(Path path, LinkOption ... opts) : 判断是否是文件

//      boolean isHidden(Path path) : 判断是否是隐藏文件
//      要求此path对应的物理上的文件需要存在。才可判断是否隐藏。否则, 抛异常。
    System.out.println(Files.isHidden(path1)); // false

//      boolean isReadable(Path path) : 判断文件是否可读
    System.out.println(Files.isReadable(path1)); // true
//      boolean isWritable(Path path) : 判断文件是否可写
    System.out.println(Files.isWritable(path1)); // true
//      boolean notExists(Path path, LinkOption ... opts) : 判断文件是否不存在
    System.out.println(Files.notExists(path1, LinkOption.NOFOLLOW_LINKS)); //
false
}

/**
 * StandardOpenOption.READ: 表示对应的Channel是可读的。
 * StandardOpenOption.WRITE: 表示对应的Channel是可写的。
 * StandardOpenOption.CREATE: 如果要写出的文件不存在, 则创建。如果存在, 忽略
 * StandardOpenOption.CREATE_NEW: 如果要写出的文件不存在, 则创建。如果存在, 抛异常
 *
 * @author shkstart 邮箱: shkstart@126.com
 * @throws IOException

```

```

    */
    @Test
    public void test3() throws IOException{
        Path path1 = Paths.get("d:\\", "hello.txt");

        //      InputStream newInputStream(Path path, OpenOption...how):获取 InputStream 对
        象
        InputStream inputStream = Files.newInputStream(path1,
        StandardOpenOption.READ);

        //      OutputStream newOutputStream(Path path, OpenOption...how) : 获取
        OutputStream 对象
        OutputStream outputStream = Files.newOutputStream(path1,
        StandardOpenOption.WRITE,StandardOpenOption.CREATE);

        //      SeekableByteChannel newByteChannel(Path path, OpenOption...how) : 获取与指定
        文件的连接, how 指定打开方式。
        SeekableByteChannel channel = Files.newByteChannel(path1,
        StandardOpenOption.READ,StandardOpenOption.WRITE,StandardOpenOption.CREATE);

        //      DirectoryStream<Path> newDirectoryStream(Path path) : 打开 path 指定的目
        录
        Path path2 = Paths.get("d:\\home\\upload");
        DirectoryStream<Path> directoryStream = Files.newDirectoryStream(path2);
        Iterator<Path> iterator = directoryStream.iterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }

    }
}

```

## 6、自动资源管理

- Java 7 增加了一个新特性，该特性提供了另外一种管理资源的方式，这种方式能自动关闭文件。这个特性有时被称为自动资源管理 (Automatic Resource Management, ARM)，该特性以 try 语句的扩展版为基础。自动资源管理主要用于，当不再需要文件（或其他资源）时，可以防止无意中忘记释放它们。

## ● 自动资源管理基于 try 语句的扩展形式：

```
try(需要关闭的资源声明){  
    //可能发生异常的语句  
}catch(异常类型 变量名){  
    //异常的处理语句  
}  
.....  
finally{  
    //一定执行的语句  
}
```

当 try 代码块结束时，自动释放资源。因此不需要显示的调用 close() 方法。该形式也称为“带资源的 try 语句”。

注意：

- ①try 语句中声明的资源被隐式声明为 final，资源的作用局限于带资源的 try 语句
- ②可以在一条 try 语句中管理多个资源，每个资源以“;” 隔开即可。
- ③需要关闭的资源，必须实现了 AutoCloseable 接口或其自接口 Closeable

```
//自动资源管理：自动关闭实现 AutoCloseable 接口的资源  
@Test  
public void test8(){  
    try(FileChannel inChannel = FileChannel.open(Paths.get("1.jpg"),  
StandardOpenOption.READ);  
        FileChannel outChannel = FileChannel.open(Paths.get("2.jpg"),  
StandardOpenOption.WRITE, StandardOpenOption.CREATE)){  
  
        ByteBuffer buf = ByteBuffer.allocate(1024);  
        inChannel.read(buf);  
  
    }catch(IOException e){  
  
    }  
}
```