# 1. NIO简介

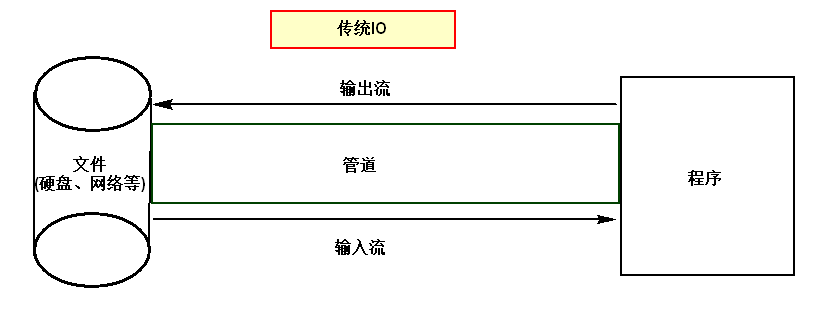
Java NIO（New IO，新IO；或者也可以理解为Non-blocking IO，非阻塞式IO） 是JDK1.4就引入的一系列新的输入输出API，可以替代标准的输入输出API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同。NIO支持面向缓冲区的、基于通道的IO操作。 NIO将以更加高效的方式进行文件的读写操作。

随着高并发和大数据量处理的需求，NIO使用的场景越来越多，因此Java人员需要掌握NIO。在JDK1.7时，又对NIO进行了改动，我们把改动的又称之为NIO2。

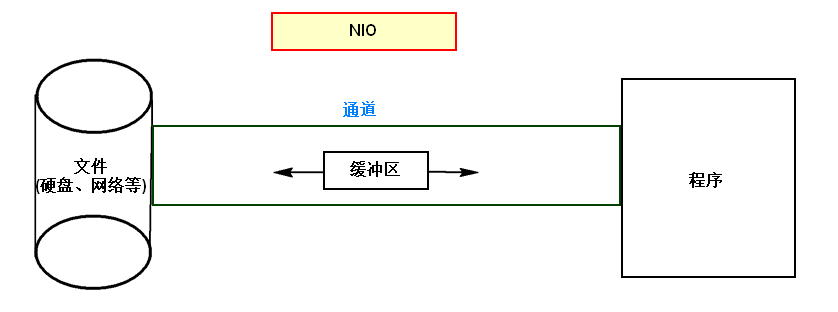
# 2. NIO与IO的主要区别

（1）IO是面向流的（Stream Oriented），NIO是面向缓冲区的（Buffer Oriented）。

什么是面向流呢？在传统IO中，首先需要建立用于数据传输的管道，这时管道面对的直接就是要传输的数据流。传统的IO是单向的，即数据的流动是有方向的：如果想把文件A数据送入程序，则建立的输入流管道只能从A流向程序，如果想把程序的数据输出到文件A中，还要新建一个新的输出流管道来输出数据。如图：



那什么是面向缓冲区的呢？同样，使用NIO传输数据，还是要建立传输的通道，注意将之称为“通道”。但这里的通道和传统IO中的流的管道是有不同之处的。现在不可以把该通道理解为“流”的管道，我们可以把通道理解为“铁路”，铁路本身不完成运输，就用于A地与B地的连接通道。要想运输，必须有火车承载乘客或货物进行运输。类似的，要想在通道上传送数据，需要使用“缓冲区”。缓冲区就用来承载数据，然后在通过通道运输。要输出时，可在程序中“装载”缓冲区，缓冲区运输的数据到达目的文件后，就可“卸载”拿到数据。当然，若需要再把数据输入到程序，还可利用该缓冲区装载文件数据，再运送到程序。相同“地点”之前的传输无需新开通道，因为缓冲区是双向的。显然，数据的传输时面向缓冲区的。如下图所示：



（2）传统IO是阻塞IO，NIO是非阻塞IO。

（3）NIO有选择器Selectors。

上述的（2）（3）是针对网络编程而言的。以后再讲。

# 3. 通道（Channel）和缓冲区（Buffer）

## 3.1 简要

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 负责存储数据。

## 3.2 缓冲区

在NIO中，缓冲区（Buffer）负责数据的存储，即缓冲区是一个存储特定数据类型的容器。缓冲区中实际就是维护了“数组”，根据数据类型的不同，JDK提供了如下几种类型的缓冲区：

ByteBuffer：字节缓冲区；

CharBuffer：字符缓冲区；

ShortBuffer、IntBuffer、LongBuffer、FloatBuffer和DoubleBuffer：整型和浮点数缓冲区。

所有的缓冲区都是Buffer抽象类的子类，因此他们的使用和管理方式类似。比如都是通过allocate()静态方法获取缓冲区。我们常使用的缓冲区就是字节缓冲区，因为文件、网络中传送的都是字节数据。

下面我们了解一下缓冲区的细节，但是我们最终会使用Buffer和NIO的通道进行交互，以达到传输数据的目的。

### 3.2.1 缓冲区的基本属性

使用allocate()静态方法创建好缓冲区后，我们使用以下两个核心方法存取缓冲区的数据：

（1）put()：向缓冲区中存入数据：

* 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)

（2）get()：向缓冲区中获取数据：

* 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)

如果想使用上述方法正确地存取缓冲区数据，还必须先了解缓冲区的四个核心属性。这四个属性均定义在Buffer抽象类中，解释如下：

（1）capacity：容量，表示缓冲区中最大存储数据的容量。一旦声明就不能改变（即通过allocate()声明的）。

（2）limit：界限，表示第一个不允许读写的数据的索引，即位于 limit 后的数据不可读写。

（3）position：位置，表示下一个要读写的数据的索引（即将要操作的位置）。

（4）mark：稍后讲。

分别用capacity()、limit()和position()方法获得当前缓冲区中这些属性的值。

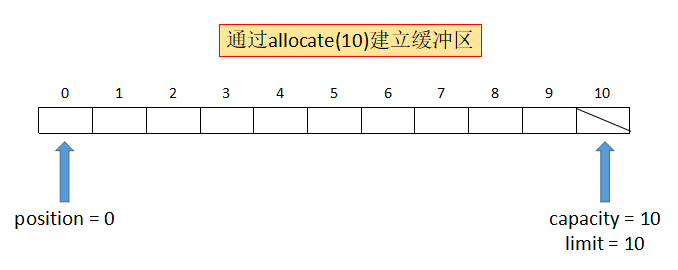
现在，我们查看在调用allocate()后，这些属性是什么值：

|  |
| --- |
| **package** juc.demo;  **import** java.nio.ByteBuffer;  **public class** Demo {  **public static void** main(String[] args) {  *// 分配一个指定大小的缓冲区。容量是10* ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  System.***out***.println(byteBuffer.capacity()); *// 容量* System.***out***.println(byteBuffer.limit()); *// 界限* System.***out***.println(byteBuffer.position()); *// 位置* } } |

输出的结果是：

|  |
| --- |
| 10  10  0 |

示意图：

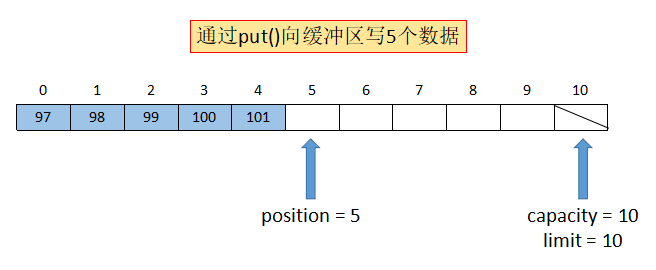


也就是说，分配完缓冲区后，limit和capacity相等，即允许对整个缓冲区进行操作。position位置是0，意味着允许从第一个位置开始向缓冲区写入数据。

现在我们就使用put()方法在缓冲区中写数据：

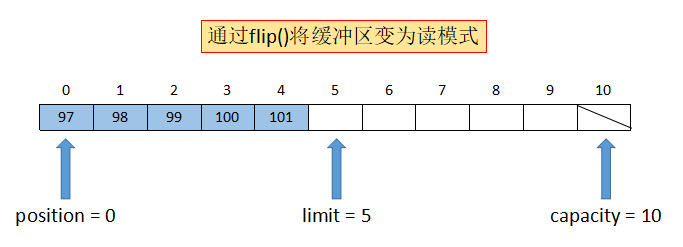
|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  *// 将“ABCDE”一个字符串写入缓冲区* byteBuffer.put(**"ABCDE"**.getBytes());  *// 输出属性* System.***out***.println(byteBuffer.capacity());  System.***out***.println(byteBuffer.limit());  System.***out***.println(byteBuffer.position());  } } |

这时程序输出的结果是“10 10 5”，说明limit和capacity没有变化，position变为5，说明现在写了5个数据，因为“ABCDE”分别占据一个字节，对应的数值为“97 98 99 100 101”。示意图如下：



我们可以把上述情况看成是写数据的模式。

现在缓冲区中已经有数据了，那么如何读取缓冲区的数据呢？首先，需要用flip()方法将缓冲区切换到“读数据”模式。当调用完flip()方法后，属性变化如下：position变为0，limit变为5，capacity还是不变的。示意图如下：



这就意味着我们可以从头开始读取已存入的5个数据，limit之后的空间是不允许操作的。这就是“读模式”。

接下来我们就能用get()方法读取数据。我们把数据读取到字节数组中：

|  |
| --- |
| **package** juc.demo;  **import** java.nio.ByteBuffer;  **public class** Demo {  **public static void** main(String[] args) {  ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  *// 将“ABCDE”一个字符串写入缓冲区* byteBuffer.put(**"ABCDE"**.getBytes());  *// 转换模式* byteBuffer.flip();  *// 读取数据* **byte**[] data = **new byte**[byteBuffer.limit()]; // 该数组用于存储数据  byteBuffer.get(data);  System.***out***.println(**new** String(data));  *// 输出属性* System.***out***.println(byteBuffer.capacity());  System.***out***.println(byteBuffer.limit());  System.***out***.println(byteBuffer.position());  } } |

程序输出的就是“ABCDE”。当然，再输出的各个属性位置就是position=5，limit不变还是5，capacity不变还是10。这里就不画示意图了。

那么如果想重读数据怎么办呢？就需要调用rewind()方法，该方法可将position置为0，这样就能重读数据了。

另外，还有clear()方法，该方法是清空缓冲区，会使各个属性回到最初状态，即position=0，limit=10，capacity=10。但是缓冲区的数据依然存在，只是出于“被遗忘”状态，因为limit被重置，因此不知道之前缓冲区有多少有效数据，就不能正确地读取了。

最后，还有一个mark属性。mark，标记，可以调用mark()方法记录当前position的位置，然后，通过reset()方法可以将position恢复到mark之前标记的位置。例如现在想要重复读取“CDE”，那么可以在读完“AB”之后做个标记，然后再重新读：

|  |
| --- |
| **package** juc.demo;  **import** java.nio.ByteBuffer;  **public class** Demo {  **public static void** main(String[] args) {  ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  *// 将“ABCDE”一个字符串写入缓冲区* byteBuffer.put(**"ABCDE"**.getBytes());  *// 转换模式* byteBuffer.flip();  *// 先读取AB* **byte**[] abByte = **new byte**[2];  byteBuffer.get(abByte);  *// 标记* byteBuffer.mark();  System.***out***.println(**new** String(abByte));  *// 再读取CDE* **byte**[] cdeByte = **new byte**[3];  byteBuffer.get(cdeByte);  System.***out***.println(**new** String(cdeByte));  *// 重置mark* byteBuffer.reset();  byteBuffer.get(cdeByte);  System.***out***.println(**new** String(cdeByte));  } } |

可以看出：0 <= mark <= position <= limit <= capacity。

还可以通过hasRemaining()判断是否还有剩余数据，remaining()返回剩余的个数：

|  |
| --- |
| **package** juc.demo;  **import** java.nio.ByteBuffer;  **public class** Demo {  **public static void** main(String[] args) {  ByteBuffer byteBuffer = ByteBuffer.*allocate*(10);  *// 将“ABCDE”一个字符串写入缓冲区* byteBuffer.put(**"ABCDE"**.getBytes());  *// 转换模式* byteBuffer.flip();  *// 先读取AB* **byte**[] abByte = **new byte**[2];  byteBuffer.get(abByte);  *// 是否有剩余* **if** (byteBuffer.hasRemaining()) {  System.***out***.println(byteBuffer.remaining());  }  } } |

会输出3，即3个未读取。

利用上述讲的东西，操作缓冲区数据基本OK了。

### 3.2.2 直接缓冲区与非直接缓冲区

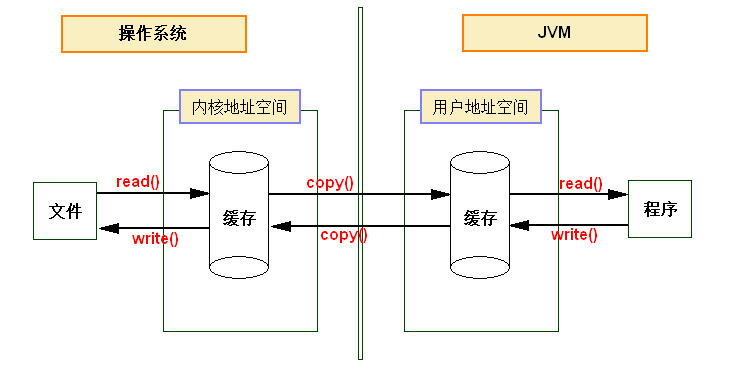
字节缓冲区要么是直接的，要么是非直接的。

（1）非直接缓冲区：通过ByteBuffer.allocate()方法分配非直接缓冲区，是将缓冲区建立在JVM的内存中。

（2）直接缓冲区：通过ByteBuffer.allocateDirect()方法分配直接缓冲区，将缓冲区建立在计算机的物理内存中，可以提高效率。

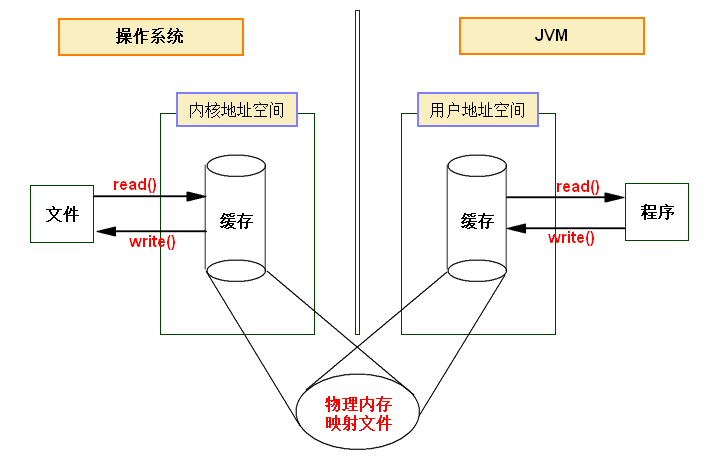
为什么直接缓冲区能提高效率呢？

我们要知道，要操作文件，JVM始终需要调用操作系统的I/O操作，那么操作系统内存中有一份数据的缓存，JVM进行I/O时，JVM也有自己的缓存，那么在读写时，JVM会使用复制的方法，保持操作系统中缓存和JVM中缓存的数据交换，进而进行真正的读写操作。如下图所示：



显然，此种方式产生了“冗余”的内存空间（重复的中间缓冲区），且效率不高。这种方式就是非直接缓冲区的IO方式。

那么如果为直接字节缓冲区，JVM会尽可能地直接在此缓冲区上执行本机 I/O 操作。也就是说，在每次调用操作系统的IO操作时，JVM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）。直接缓冲区可以采用“物理内存映射文件”来操作数据，避免了复制的开销：



直接字节缓冲区可以通过调用ByteBuffer类的allocateDirect()工厂方法来创建。直接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，但由图可看出，使用直接缓冲区能够提高内存和读写的效率（因为通过映射文件，操作系统和JVM相当于共用缓存数据，无需相互复制数据），并且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，当操作的数据使用时间长、数据量大时，就使用直接缓冲区。

可以使用ByteBuffer对象的isDirect()方法查看该缓冲区对象是否是直接缓冲区。例如：

|  |
| --- |
| **package** juc.demo;  **import** java.nio.ByteBuffer;  **public class** Demo {  **public static void** main(String[] args) {  ByteBuffer notDirectBuffer = ByteBuffer.*allocate*(10);  ByteBuffer directBuffer = ByteBuffer.*allocateDirect*(10);  System.***out***.println(notDirectBuffer.isDirect()); *// false* System.***out***.println(directBuffer.isDirect()); *// true* } } |

在下面通道的讲解中，还会用到其他方法来使用直接缓冲区。

## 3.3 通道Channel

### 3.3. 1 简述

通道（Channel）相关类定义在java.nio.channels包中，通道表示IO源与目标打开的连接。Channel本身不能直接访问数据，Channel需要与缓冲区Buffer进行交互。

我们主要使用的通道类有：

（1）FileChannel：操作文件的通道类；

（2）SocketChannel：读写TCP数据的通道类；

（3）ServerSocketChannel：监听TCP连接的通道类，对每一个新进来的连接都会创建一个 SocketChannel；

（4）DatagramChannel：读写UDP数据的通道类。

上述这些通道类都实现java.nio.channels.Channel接口。

### 3.3.2 获取Channel的方式

现在，一共有三种方式获取Channel通道对象。

（1）Java给以下的支持通道的类提供了getChannel()方法，可以此获得Channel对象：

本地IO类：

FileInputStream、FileOutputStream和RandomAccessFile类；

网络IO类：

Socket、ServerSocket和DatagramSocket。

（2）JDK 1.7中，为各个Channel实现类提供了open()静态方法用于获得Channel对象。例如：FileChannel.open()。

（3）JDK 1.7中，Files工具类提供了newByteChannel()方法，可以此获取Channel对象。

例1：利用FileInputStream和FileOutputStream获取通道来复制文件。

|  |
| --- |
| **package** juc.demo;  **import** java.io.FileInputStream; **import** java.io.FileOutputStream; **import** java.io.IOException; **import** java.nio.ByteBuffer; **import** java.nio.channels.FileChannel;  **public class** Demo {  **public static void** main(String[] args) {  FileInputStream fis = **null**;  FileOutputStream fos = **null**;  FileChannel inChannel = **null**;  FileChannel outChannel = **null**;  **try** {  fis = **new** FileInputStream(**"d:/1.txt"**);  fos = **new** FileOutputStream(**"d:/2.txt"**);  *// 1. 分别获得输入和输出的channel* inChannel = fis.getChannel();  outChannel = fos.getChannel();  *// 2. 创建指定大小的缓冲区以便传送数据* ByteBuffer buf = ByteBuffer.*allocate*(1024);  *// 3. 读取数据。我们可以利用inChannel.read(buf)方法读数据  // 读到的数据写入buf缓冲区中。也是需要循环读取，如果读不到了，就返回-1* **while** (inChannel.read(buf) != -1) {  *// 现在需要将buf中的数据再写入到outChannel中，调用outChannel.write()方法即可。  // 需要注意的是：  // 1. 需要先调用buf.flip()方法将buf转换为读模式，因为先要读到缓冲区数据，再写入outChannel。  // 2. 无需像之前传统IO读取那样要记录本次读到的真实数据的长度，因为在读模式中，buf中的limit属性就表示了数据的有效长度，所以在write中传一个buf就足够了。* buf.flip(); *// 切换到读模式* outChannel.write(buf);  *// 3. 最后，一定要将buf清空，以便可以循环写入buf、再读取、再清空...* buf.clear(); *// 清空缓冲区* }  } **catch** (IOException e) {   } **finally** {  *// 关闭资源* **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();  }  }  }  } } |

上述代码使用的非直接缓冲区。现在我们使用直接缓冲区完成文件的复制：

例2：使用直接缓冲区完成文件的复制（内存映射文件的方式）。

|  |
| --- |
| **package** juc.demo;  **import** java.io.IOException; **import** java.nio.MappedByteBuffer; **import** java.nio.channels.FileChannel; **import** java.nio.channels.FileChannel.MapMode; **import** java.nio.file.Paths; **import** java.nio.file.StandardOpenOption;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 1. 现在用open()方式打开通道。  // open(Path path, OpenOption... options)方法解释：  // 参数1，Path，即路径，可用Paths.get(String first, String... more)获取。可通过拼接多个文件路径得到Path对象。比如Paths.get("d:/", "study/", "1.txt")。  // 参数2，表示文件通道的打开方式。也是可变参数，表示可有多种方式，比如读和写。* FileChannel inChannel = FileChannel.*open*(Paths.*get*(**"d:/1.txt"**), StandardOpenOption.***READ***);  FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"d:/2.txt"**), StandardOpenOption.***WRITE***, StandardOpenOption.CREATE); *// CREATE表示每次都会重新创建文件。如果使用CREATE\_NEW，那么文件不存在则创建，若存在则报错。  // 2. 写到这里，我们获得了inChannel和outChannel，下面的代码就和例1中相同了。  // 但是我们这里要求是使用直接缓冲区。  // 当然可以使用ByteBuffer.allocateDirect()方式创建直接缓冲区，这个和例1也是大同小异，我们换个方法，使用inChannel.map()方法返回一个MappedByteBuffer，这个也是直接缓冲区。* MappedByteBuffer inMappedBuffer = inChannel.map(MapMode.***READ\_ONLY***, 0, inChannel.size());  *// 同样，再用outChannel.map()方法也得到直接缓冲区。只是现在的模式是写。* MappedByteBuffer outMappedBuffer = outChannel.map(MapMode.***READ\_WRITE***, 0, inChannel.size()); *// 这里的模式只有“READ\_WRITE”，即读写，没有只写的模式。  //3. 直接对缓冲区数据进行读写操作* **byte**[] buf = **new byte**[inMappedBuffer.limit()];  inMappedBuffer.get(buf); *// 读取数据* outMappedBuffer.put(buf); *// 写入数据  // 关闭连接。这样就完成了复制文件* outChannel.close();  inChannel.close();  } } |

但上述程序运行出现异常“NonReadableChannelException”，为什么会出现“不可读通道异常”呢？原来我们给outChannel分配的权限是写和创建，但是outChannel.map()写入数据时，模式却是“READ\_WRITE”，即读写，但是outChannel并没有读权限，因此出错。解决办法：给outChannel加上读权限：

|  |
| --- |
| FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"d:/2.txt"**), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***, StandardOpenOption.***READ***); |

实际上文件的复制无需这么麻烦。可以使用通道之前的传输，使用的方法：

（1）transferFrom()

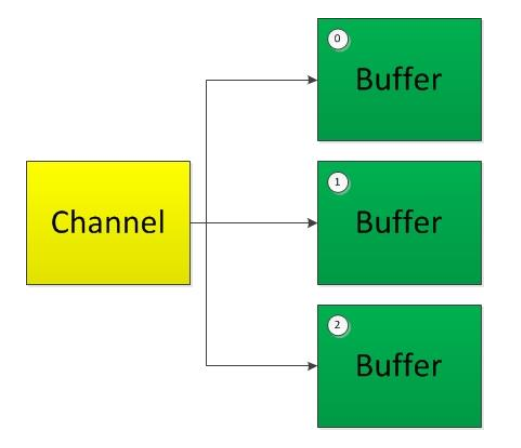
（2）transferTo()

上述方法用的也是直接缓冲区的方式。

|  |
| --- |
| **package** juc.demo;  **import** java.io.IOException; **import** java.nio.MappedByteBuffer; **import** java.nio.channels.FileChannel; **import** java.nio.channels.FileChannel.MapMode; **import** java.nio.file.Paths; **import** java.nio.file.StandardOpenOption;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 获得读写通道* FileChannel inChannel = FileChannel.*open*(Paths.*get*(**"d:/1.txt"**), StandardOpenOption.***READ***);  FileChannel outChannel = FileChannel.*open*(Paths.*get*(**"d:/2.txt"**), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);  *// 复制。调用inChannel的transferTo方法将数据传输到outChannel中* inChannel.transferTo(0, inChannel.size(), outChannel);  *// 同理，也可以用outChannel的transferFrom方法。如下：  // outChannel.transferFrom(inChannel, 0, inChannel.size());  // 最后关闭通道* outChannel.close();  inChannel.close();  } } |

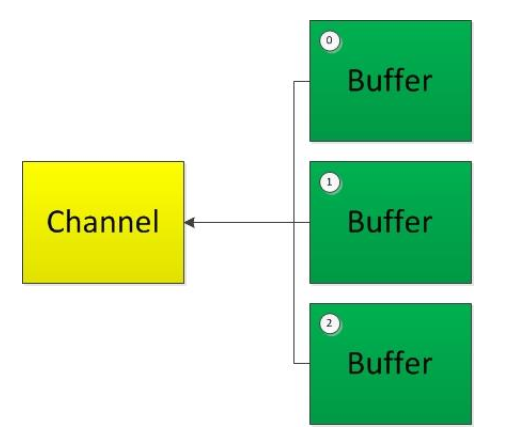
### 3.3.3 分散Scatter与聚集Gather

分散读取（Scattering Reads）是指将从Channel中读取的数据“分散” 到多个 Buffer 中。如图：



分散读取时，会按照缓冲区的顺序，将从Channel中读取的数据依次填满Buffer。

相反，聚集写入（Gathering Writes）是指将多个 Buffer 中的数据“聚集”到 Channel中：



聚集写入时，会按照缓冲区的顺序，将Buffer中position和limit之间的数据写到 Channel中。

代码示例：

|  |
| --- |
| **package** juc.demo;  **import** java.io.IOException; **import** java.io.RandomAccessFile; **import** java.nio.ByteBuffer; **import** java.nio.channels.FileChannel;  **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 这里使用RandomAccessFile演示* RandomAccessFile raf1 = **new** RandomAccessFile(**"d:/1.txt"**, **"rw"**);  FileChannel channel = raf1.getChannel();  *// 新建多个缓冲区* ByteBuffer buf1 = ByteBuffer.*allocate*(1024 \* 3);  ByteBuffer buf2 = ByteBuffer.*allocate*(1024 \* 4);  *// 将channel中数据分散读取到Buffer中。* ByteBuffer[] byteBuffers = {buf1, buf2}; *// 先包装成数组* channel.read(byteBuffers); *// read可接收ByteBuffer数组，因此将byteBuffers传入  // 现在buf1和buf2中顺序存储了1.txt文件了，我们可输出查看* **for**(**int** i = 0; i < byteBuffers.**length**; i++) {  byteBuffers[i].flip(); *// 首先要将缓冲区切换为读模式* **byte**[] b = **new byte**[byteBuffers[i].limit()];  byteBuffers[i].get(b, 0, byteBuffers[i].limit());  System.***out***.println(**new** String(b));  }   *// 正好再在这里演示聚集的代码* RandomAccessFile raf2 = **new** RandomAccessFile(**"d:/2.txt"**, **"rw"**);  FileChannel outChannel = raf2.getChannel();  **for**(**int** i = 0; i < byteBuffers.**length**; i++) {  byteBuffers[i].flip(); *// 首先要将缓冲区切换为读模式。否则下面的outChannel.write不能写。* }  **long** len = outChannel.write(byteBuffers); *// 利用上面的byteBuffers，将多个buffer聚集写入到channel。这里完成了文件的复制* System.***out***.println(len);  *// 关闭资源* outChannel.close();  channel.close();  raf2.close();  raf1.close();  } } |

即注意，只要将缓冲区数据写入到channel，那么在此之前一定要flip()转换一下！！！。

### 3.3.4 字符集

（1）查看JDK支持的字符集：

|  |
| --- |
| @Test **public void** test() {  SortedMap<String, Charset> map = Charset.*availableCharsets*(); *// 支持的字符集  // 遍历* **for** (Map.Entry<String, Charset> entry : map.entrySet()) {  System.***out***.println(entry.getValue());  } } |

输出（因为比较多，下面省略了大部分输出）：

|  |
| --- |
| Big5  Big5-HKSCS  CESU-8  EUC-JP  EUC-KR  GB18030  GB2312  GBK  ISO-2022-CN  ISO-2022-JP  ISO-2022-JP-2  ISO-2022-KR  ISO-8859-1  US-ASCII  UTF-16  UTF-16BE  UTF-16LE  UTF-32  UTF-32BE  UTF-32LE  UTF-8  windows-1250  x-windows-iso2022jp |

我们可以使用Charset charset = Charset.forName(name)方法得到某编码字符集对象。然后使用charset对象的encode()和decode()方法进行编码和解码。编码返回的结果是ByteBuffer，解码返回的结果是CharBuffer。

现在用UTF-8编码写内容到1.txt文件中：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 1. 将要写入的内容放入CharBuffer中* CharBuffer charBuffer = CharBuffer.*allocate*(1024);  charBuffer.put(**"这是要写入的内容 UTF-8编码"**);  *// 2. 获得UTF-8编码字符集，并将charBuffer转成编码后的字符数组* Charset charset = Charset.*forName*(**"UTF-8"**);  charBuffer.flip(); *// 注意首先要flip()* ByteBuffer byteBuffer = charset.encode(charBuffer); *// 编码  // 3. 打开Channel并写入* FileChannel channel = FileChannel.*open*(Paths.*get*(**"d:/1.txt"**), StandardOpenOption.***WRITE***, StandardOpenOption.***CREATE***);  channel.write(byteBuffer); *// 这里write之前就不需要再flip()了。因为encode()返回的byteBuffer就是读模式，如果再flip()，将导致写入数据失败（虽然程序不报错）。  // 4. 关闭channel* channel.close();  } } |

说明：上述代码中，用charset.encode(str)也可直接将字符串编码成ByteBuffer，可按需使用。

而且，由于操作了多个Buffer，有时候我们并不确定是不是在操作每个Buffer之前都要flip()一下，我现在的笨方法就是先执行程序再说，如果文件中没有数据（或者程序中没读到数据），就断点调试看看每个buffer的数据内容和write()返回值（表示数据长度）是否是0，如果某个buffer的数据内容都是“0”或者write()返回值是0，说明有问题，很可能需要flip()，但是如果是flip()之后的数据内容是“0”或者write()返回值是0，那么就说明前面不应该flip()。

那么如果要读取1.txt内容，则这样：

|  |
| --- |
| **package** juc.demo; **public class** Demo {  **public static void** main(String[] args) **throws** IOException {  *// 创建charset* Charset charset = Charset.*forName*(**"UTF-8"**);  *// 打开channel* FileChannel fileChannel = FileChannel.*open*(Paths.*get*(**"d:/1.txt"**), StandardOpenOption.***READ***);  *// 分配缓冲区，以便下面循环操作* ByteBuffer byteBuffer = ByteBuffer.*allocate*(1024);  *// 循环读取和操作* **while** (fileChannel.read(byteBuffer) != -1) {  *// 先flip()* byteBuffer.flip();  CharBuffer charBuffer = charset.decode(byteBuffer); *// 解码为charBuffer* **char**[] chs = **new char**[charBuffer.limit()]; *// char数组用于存储从缓冲区中得到的数据* charBuffer.get(chs); *// 从缓冲区读数据到chs中* System.***out***.println(chs); *// 输出* byteBuffer.clear(); *// 最后清空缓冲区以便循环* }  *// 关闭channel* fileChannel.close();  } } |

上述就是利用charset对象进行编码和解码。另外，charset对象提供了newEncoder()和newDecoder()方法，用于获取编码器和解码器，编码器和解码器中也分别提供了encode()和decode()方法，和直接使用charset.encode()和charset.decode()是一样的，随便你使用。

# 4. NIO的非阻塞式网络通信

## 4.1 阻塞与非阻塞

阻塞与非阻塞是对于网络通信而言的。

传统的网络IO是阻塞式的。例如，我们之前写的第一个TCP服务端代码，当调用IO读写操作（例如抽象的read()和write()）时，程序将会阻塞，直到IO中有可以读写的数据时，程序才会继续执行。在此期间，如果有其他客户端Socket请求连接，那么程序是不能及时响应的。由于线程会阻塞（包括accept()也会阻塞），因此一般会将服务端设计为多线程模式，但是当要处理的客户端Socket连接非常多时，服务端性能将急剧下降。

Java NIO是非阻塞模式的。NIO提出了一个叫“选择器（Selector）”的东西。同样，使用NIO进行网络通信时，还需用到缓冲区和通道。选择器就具有如下作用：首先每个用于网络连接的通道将注册到选择器上，选择器监测这些通道的IO状况。当某个通道完全准备好传输数据时，选择器才会将这个任务分配到服务端的一个或多个线程上运行。这样的优点是线程无需阻塞等待。

## 4.2 NIO网络通信

使用NIO进行网络通信的核心：

（1）通道：负责连接；（2）缓冲区，负责存放数据；（3）选择器，是SelectableChannel的多路复用器，用于监控SelectableChannel接口实现类的IO状况。

哪些类实现了SelectableChannel接口呢？

前面提到的SocketChannel、SocketServerChannel和DatagramChannel都实现了SelectableChannel接口，还有以后用到的Pipe.SinkChannel和Pipe.SourceChannel也实现了此接口。

而对于FileChannel，它是不能切换成非阻塞模式的。因为非阻塞模式是相较于网络IO而言的。

### 4.2.1 阻塞式NIO通信

我们先用NIO来完成一个阻塞式的网络通信：

（1）客户端

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.InetSocketAddress; **import** java.nio.ByteBuffer; **import** java.nio.channels.FileChannel; **import** java.nio.channels.SocketChannel; **import** java.nio.file.Paths; **import** java.nio.file.StandardOpenOption;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  *// 1. 获取通道* SocketChannel socketChannel = SocketChannel.*open*(**new** InetSocketAddress(**"127.0.0.1"**, 10000));  *// 我们打开本地文件的通道。下面将文件数据发送过去* FileChannel fileChannel = FileChannel.*open*(Paths.*get*(**"d:/1.txt"**), 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，并发送到服务器  // 循环读取，并通过socketChannel发送* **while** (fileChannel.read(buf) != -1) {  buf.flip();  socketChannel.write(buf);  buf.clear();  }  *// 4. 关闭通道* fileChannel.close();  socketChannel.close();  } } |

（2）服务端

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **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;  **public class** Server {  **public static void** main(String[] args) **throws** IOException {  *// 1. 获取通道* ServerSocketChannel ssChannel = ServerSocketChannel.*open*();  *// 2. 绑定端口* ssChannel.bind(**new** InetSocketAddress(10000));  *// 服务端要保存的文件为2.txt* FileChannel fileChannel = FileChannel.*open*(Paths.*get*(**"d:/2.txt"**), StandardOpenOption.***CREATE***, StandardOpenOption.***WRITE***);  *// 3. 获取客户端连接的通道* SocketChannel socketChannel = ssChannel.accept();  *// 4. 分配指定大小的缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);  **while** (socketChannel.read(buf) != -1) {  buf.flip();  fileChannel.write(buf);  buf.clear();  }  *// 6. 关闭* fileChannel.close();  socketChannel.close();  ssChannel.close();  } } |

上述的代码还是阻塞式的，如果想要客户端和服务端相互收发数据，也像之前一样，发送（或接收）完数据后再接收（或发送）数据，并且发送或接收完后要调用shutdownInput()或shutdownOutput()方法，只是现在的shutdownXxx()方法是SocketServerChannel和ServerSocketChannel提供的。并且，最好将服务端改为多线程模式。这些大家自行去做。

### 4.2.2 非阻塞式网络通信

非阻塞式的TCP客户端比较简单，只需要设置为非阻塞的即可。例如下述代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.InetSocketAddress; **import** java.nio.ByteBuffer; **import** java.nio.channels.SocketChannel; **import** java.util.Date;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  *// 1. 获取通道* SocketChannel socketChannel = SocketChannel.*open*(**new** InetSocketAddress(**"127.0.0.1"**, 10000));  *// 2. 切换为非阻塞模式* socketChannel.configureBlocking(**false**);  *// 3. 分配缓冲区* ByteBuffer buf = ByteBuffer.*allocate*(1024);  *// 4. 发送数据* buf.put(**new** Date().toString().getBytes()); *// 发送当前时间* buf.flip();  socketChannel.write(buf);  buf.clear();  *// 5. 关闭通道* socketChannel.close();  } } |

非阻塞式的服务器端较复杂。服务器端需要使用选择器。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。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。

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

（2）向选择器中注册通道：调用待注册通道的register(Selector sel, int ops)方法完成注册。参数1是选择器，参数2是指定该选择器监听该通道的哪些类型事件。参数2有下面几种选项（即可以监听的事件类型）：

读事件：SelectionKey.OP\_READ（1）

写事件：SelectionKey.OP\_WRITE（4）

连接事件：SelectionKey.OP\_CONNECT（8）

接收事件：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。例如：

|  |
| --- |
| ssChannel.register(selector, SelectionKey.***OP\_ACCEPT*** | SelectionKey.***OP\_CONNECT***); |

SelectionKey也叫做选择键。

下面是服务器端的完整代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.InetSocketAddress; **import** java.nio.ByteBuffer; **import** java.nio.channels.\*; **import** java.util.Iterator;  **public class** Server {  **public static void** main(String[] args) **throws** IOException {  *// 1. 获取通道* ServerSocketChannel ssChannel = ServerSocketChannel.*open*();  *// 2. 切换为非阻塞模式* ssChannel.configureBlocking(**false**);  *// 3. 绑定连接* ssChannel.bind(**new** InetSocketAddress(10000));  *// 4. 获取选择器* Selector selector = Selector.*open*();  *// 5. 将上面的服务器端通道注册到选择器上。  // 并且指定监听“接收”事件，即需要监听客户端过来的连接  // 注册该事件后，当有该事件到达时，selector.select()会返回，否则下面的select()会一直阻塞。* ssChannel.register(selector, SelectionKey.***OP\_ACCEPT***);  *// 6. 轮询式地获取选择器上已经“准备就绪”的通道。根据上面，select()在没有接收事件达到时，会一直阻塞* **while** (selector.select() > 0) {  *// 7. 获取当前选择器中所有已经就绪的“选择键”。  // 并且遍历这些选择键事件对象，查看他们的状态。根据不同的状态进行不同的操作* **for** (Iterator<SelectionKey> it = selector.selectedKeys().iterator(); it.hasNext(); ) {  *// 获得其中的准备就绪的选择键* SelectionKey sk = it.next();  *// 8. 判断就绪的状态是哪一种（即什么事件准备就绪）* **if** (sk.isAcceptable()) {  *// 若接收事件就绪，则接收客户端的套接字* SocketChannel socketChannel = ssChannel.accept();  *// 还要把客户端的这个socket通道切换成非阻塞模式* socketChannel.configureBlocking(**false**);  *// 再将该通道注册到选择器上，并监听读数据事件。  // 因为获得客户端套接字的首要目的就是读数据* socketChannel.register(selector, SelectionKey.***OP\_READ***);  } **else if** (sk.isReadable()) {  *// 若读就绪，就开始读数据  // 先获取当前选择器上“读数据”状态的通道* SocketChannel socketChannel = (SocketChannel) sk.channel();  *// 下面就可读取数据了* ByteBuffer buf = ByteBuffer.*allocate*(1024);  **while** (socketChannel.read(buf) != -1) {  buf.flip();  System.***out***.println(**new** String(buf.array(), 0, buf.limit()));  buf.clear();  }  }  *// 以上就监控了两种状态，别的暂时不用，以后可根据需要使用。  // 9. 最后要注意，用完当前的选择键后，要在集合中把它取消掉，否则该选择键的状态是不会改变的  // 因此用完就清除。我们调用迭代器的remove()方法，该方法会移除当前迭代的元素。这是在遍历时移除元素的方法。* it.remove();  }  }  } } |

看完整个代码，有事件处理的感觉，根据事件走。上述服务器能够一直运行，等待响应请求。

下面再编写非阻塞的UDP通信。两者大同小异：

（1）客户端：

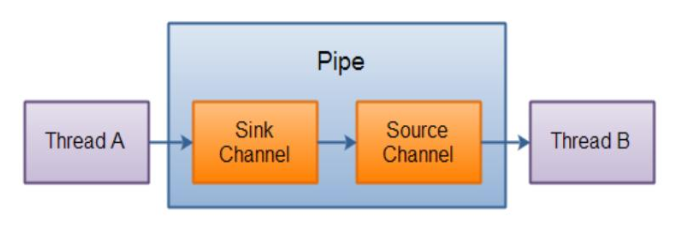
|  |
| --- |
| **package** com.demo.udp;  **import** java.io.\*; **import** java.net.InetSocketAddress; **import** java.nio.ByteBuffer; **import** java.nio.channels.DatagramChannel;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  DatagramChannel datagramChannel = DatagramChannel.*open*();  datagramChannel.configureBlocking(**false**);  ByteBuffer buffer = ByteBuffer.*allocate*(1024);  buffer.put(**"UDP数据"**.getBytes());  buffer.flip();  datagramChannel.send(buffer, **new** InetSocketAddress(**"127.0.0.1"**, 10000));  buffer.clear();  datagramChannel.close();  } } |

（2）服务端：

|  |
| --- |
| **package** com.demo.udp;  **import** java.io.\*; **import** java.net.InetSocketAddress; **import** java.nio.ByteBuffer; **import** java.nio.channels.\*; **import** java.util.Iterator;  **public class** Server {  **public static void** main(String[] args) **throws** IOException {  DatagramChannel datagramChannel = DatagramChannel.*open*();  datagramChannel.configureBlocking(**false**);  datagramChannel.bind(**new** InetSocketAddress(10000));  Selector selector = Selector.*open*();  datagramChannel.register(selector, SelectionKey.***OP\_READ***); *// UDP直接监听读事件* **while** (selector.select() > 0) {  **for** (Iterator<SelectionKey> it = selector.selectedKeys().iterator(); it.hasNext(); ) {  SelectionKey sk = it.next();  *// 判断状态* **if** (sk.isReadable()) {  *// 可读状态，开始读* ByteBuffer buf = ByteBuffer.*allocate*(1024);  datagramChannel.receive(buf);  buf.flip();  System.***out***.println(**new** String(buf.array(), 0, buf.limit()));  buf.clear();  } **else if** (sk.isWritable()) {  *// ...这些操作自行处理* }  it.remove();  }  }  } } |

### 4.2.3 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sink通道和一个source通道。数据会被写到sink通道，从source通道读取。



示例代码：

|  |
| --- |
| **package** com.demo.pipe;  **import** java.io.IOException; **import** java.nio.ByteBuffer; **import** java.nio.channels.Pipe;  **public class** PipeDemo {  **public static void** main(String[] args) **throws** IOException {  *// 1. 获取管道* Pipe pipe = Pipe.*open*();  *// 2. 创建缓冲区以便读写数据* ByteBuffer buf = ByteBuffer.*allocate*(1024);  buf.put(**"数据"**.getBytes()); *// 存入数据  // 3. 获得SinkChannel，将缓冲区中的数据写入管道* Pipe.SinkChannel sinkChannel = pipe.sink();  sinkChannel.write(buf);  *// 4. 获得SourceChannel，读取缓冲区的数据* Pipe.SourceChannel sourceChannel = pipe.source();  sourceChannel.read(buf); *// 使用同一个缓冲区* System.***out***.println(**new** String(buf.array(), 0, buf.limit()));  *// 5. 关闭资源* sourceChannel.close();  sinkChannel.close();  } } |

当然，可将上述代码的第三步和第四步分别写到不同的线程中，以实现单向通信。

# 5. NIO2

Java 7对NIO进行了扩展，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NIO2。

## 5.1 Path和Paths

java.nio.file.Path接口代表一个与平台无关的平台路径，用于描述目录结构中文件的位置。而Paths类提供了get()静态方法用于获取Path对象，即（两个重载）：

|  |
| --- |
| **public static** Path get(URI uri); // 通过uri得到Path对象  **public static** Path get(String first, String... more); // 通过多个字符串路径进行拼接得到Path对象 |

Path接口中提供的常用方法：

|  |
| --- |
| 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 |

## 5.2 Files类

java.nio.file.Files类是操作文件或目录的工具类。

（1）常用方法：

|  |
| --- |
| Path copy(Path src, Path dest, CopyOption … how)：复制文件  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 |

（2）常用判断方法：

|  |
| --- |
| 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。 |

（3）常用操作方法：

|  |
| --- |
| SeekableByteChannel newByteChannel(Path path, OpenOption... how)：获取与指定文件的连接，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 |