1.NIO 简介

2020年8月22日 14:55

主要内容

1. Java NIO 简介
2. Java NIO 与 IO 的主要区别
3. 缓冲区（Buffer）和通道（Channel）
4. 文件通道（FileChannel）
5. NIO 的非阻塞式网络通信
   - 选择器（Selector）
   - SocketChannel、ServerSocketChannel、DatagramChannel
6. 管道（Pipe）
7. Java NIO2（Path、Paths 与Files）

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。

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)

要把目标数据读取到程序中来，必须建立一个用于传输数据的管道，我们把这个管道称为**输入流**；
相应的反过来，如果我们程序中有一定的数据想要写到目标地点去，我们也需要建立一个用来传输数据的管道，我们把这个管道成为**输出流**。
因为他们的传输都是单向的，所以说**传统的 IO 是面向流**的。
而 NIO 的话，也同样有用于传输数据的类似管道的东西，可是和传统的 IO 管道有一定不同：
此时用于传输数据的通道就不能理解为水流了，不过可以理解成铁路，铁路无法单独完成运输，必须依靠火车，
所以此处”铁路“仅用于连接，本身是没有任何数据的，要想传输数据必须依赖”火车“，即**缓冲区。**
NIO 中的缓冲区就类似与火车 —— 在程序处装载要传输的数据，然后发送到目标地点，进行“卸货”，并且如果有需要可以在目标地点再装载数据传输回程序中，
由此可以看出**缓冲区是双向的**，所以 **NIO 是面向缓冲区的**。

2.通道和缓冲区

2020年8月22日 15:38

通道和缓冲区

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（通道） 负责传输，Buffer（缓冲区） 负责存储

一、缓冲区（Buffer）：

在 JAVA NIO 中负责数据的存取，缓冲区的底层就是数组用于存储不同数据类型的数据

所以根据数据类型的不同，提供了对应类型的缓冲区（**Boolean 除外**）

如：**ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FlotBuffer、DoubleBuffer**

上述缓冲区的管理方式几乎一致，都是通过 allocate() 来获取一个缓冲区，其中最常用的是 ByteBuffer。

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

allocate 英 [ˈæləkeɪt] 美 [ˈæləkeɪt]

v. 拨…(给); 划…(归); 分配…(给);

二、缓冲区存取数据的两个核心方法：

put()：存入数据到缓冲区中；

get()：获取缓冲区中的数据；

要想对缓冲区执行操作，必须先了解 **Buffer** 中的几个参数：

```
public abstract class Buffer {

    /**
     * The characteristics of Spliterators that traverse and split elements
     * maintained in Buffers.
     */
    static final int SPLITERATOR_CHARACTERISTICS =
        Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.ORDERED;

    // Invariants: mark <= position <= limit <= capacity
    private int mark = -1;
    private int position = 0;
    private int limit;
    private int capacity;
```

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 <= mark <= position <= limit <= capacity**

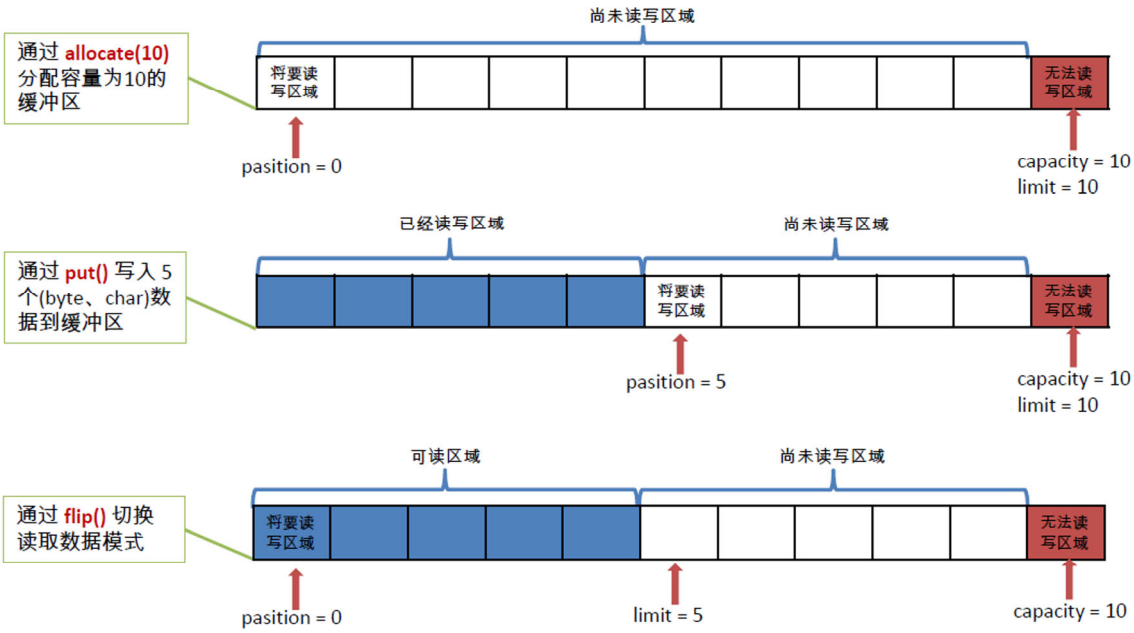
capacity（容量）：表示缓冲区中最大存储数据的容量，一旦声明无法改变【即数组初始化的时候的大小已经确定，无法扩容】；
limit（界限）：表示缓冲区中可以操作数据的的大小（limit 位置之后的数据是不能进行读写的，该数值不可以大于 capacity）；
position（位置）：表示缓冲区**当前可操作**数据的索引位置；**【position <= limit <= capacity】**
mark（标记）：用于记录当前 position 所处的位置，之后可以通过 reset() 恢复到之前 mark 记录的位置。

```java
@Test
public void test01() {
    //1.分配一个指定大小的缓冲区
    ByteBuffer buf = ByteBuffer.allocate(1024);

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

我们将对应的参数输出可以看到 capcity = 1024、limit = 1024、position = 0；

缓冲区的基本属性



我们对应图中的变化，写出代码测试 put() 方法

```
//2.利用 put() 存入数据到缓存区中
String str = "abcde";
buf.put(str.getBytes());
```

- 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)

此时 position 位置在 5，capacity 和 limit 的值不变

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

而通过 flip() 方法可则可以切换为读取数据的模式，则此时 buffer() 中已经有了五个数据，所以 limit() 的位置会移动直下标为 5 的地方：

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

相应的，通过 get() 方法可以对 buffer 内有效数据进行读取：

```
byte[] bytes = new byte[buf.limit()];
buf.get(bytes);
System.out.println("--------------------------------");
System.out.println(new String(bytes, offset: 0, bytes.length));
System.out.println(buf.position());      //5
System.out.println(buf.limit());         //5
System.out.println(buf.capacity());      //1024
```

➢ 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)

get() 中的参数即最终数据取出后的存放位置。

且使用 get() 之前必须先使用 flip() 切换到“读模式”

还有一个 rewind() 方法可以用来重复读取数据【即将 position 的位置初始化】；

clear() 可以用来清空缓冲区，不过调用该方法后缓冲区的数据依然会存在，不过是处于一种“被遗忘”状态；

hashRemaining() 可以用来判断缓冲区中是否还有可操作的数据

【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

3.直接与非直接缓冲区

直接与非直接缓冲区

- 字节缓冲区要么是直接的，要么是非直接的。如果为直接字节缓冲区，则 **Java** 虚拟机会尽最大努力直接在此缓冲区上执行本机 **I/O** 操作。也就是说，在每次调用基础操作系统的一个本机 **I/O** 操作之前（或之后），虚拟机都会尽量避免将缓冲区的内容复制到中间缓冲区中（或从中间缓冲区中复制内容）。

- 直接字节缓冲区可以通过调用此类的 **allocateDirect()** 工厂方法来创建。此方法返回的缓冲区进行分配和取消分配所需成本通常高于非直接缓冲区。直接缓冲区的内容可以驻留在常规的垃圾回收堆之外，因此，它们对应用程序的内存需求量造成的影响可能并不明显。所以，建议将直接缓冲区主要分配给那些易受基础系统的本机 **I/O** 操作影响的大型、持久的缓冲区。一般情况下，最好仅在直接缓冲区能在程序性能方面带来明显好处时分配它们。

- 直接字节缓冲区还可以通过 **FileChannel** 的 **map()** 方法 将文件区域直接映射到内存中来创建。该方法返回 **MappedByteBuffer** 。**Java** 平台的实现有助于通过 **JNI** 从本机代码创建直接字节缓冲区。如果以上这些缓冲区中的某个缓冲区实例指的是不可访问的内存区域，则试图访问该区域不会更改该缓冲区的内容，并且将会在访问期间或稍后的某个时间导致抛出不确定的异常。

- 字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 **isDirect()** 方法来确定。提供此方法是为了能够在性能关键型代码中执行显式缓冲区管理。

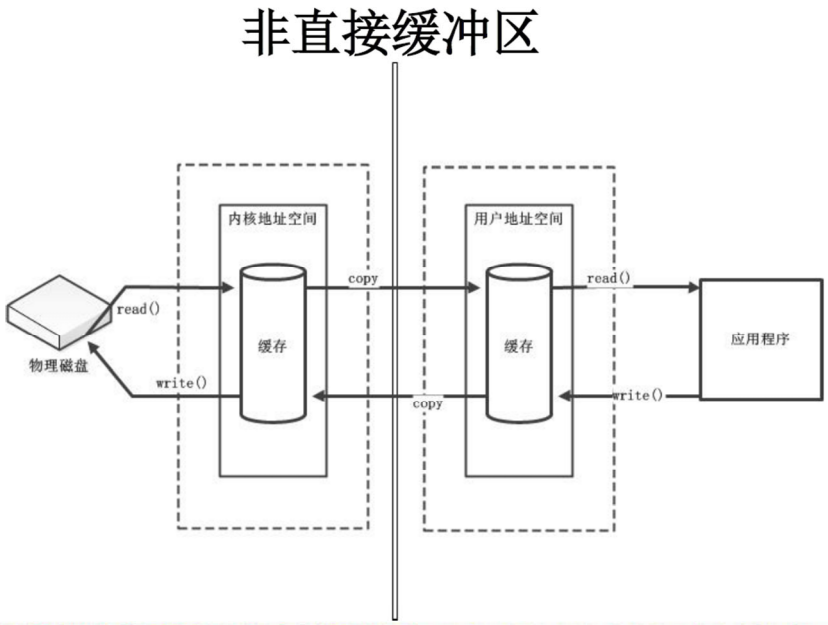
非直接缓冲区是通过 allocate() 方法分配缓冲区，将缓冲区建立在 **JVM 的内存中**：

直接缓冲区是通过 allocateDirect() 方法**【是个工厂方法】**分配直接缓冲区，将**缓冲区建立在操作系统的物理内存中**，在某种情况下可以提高效率：

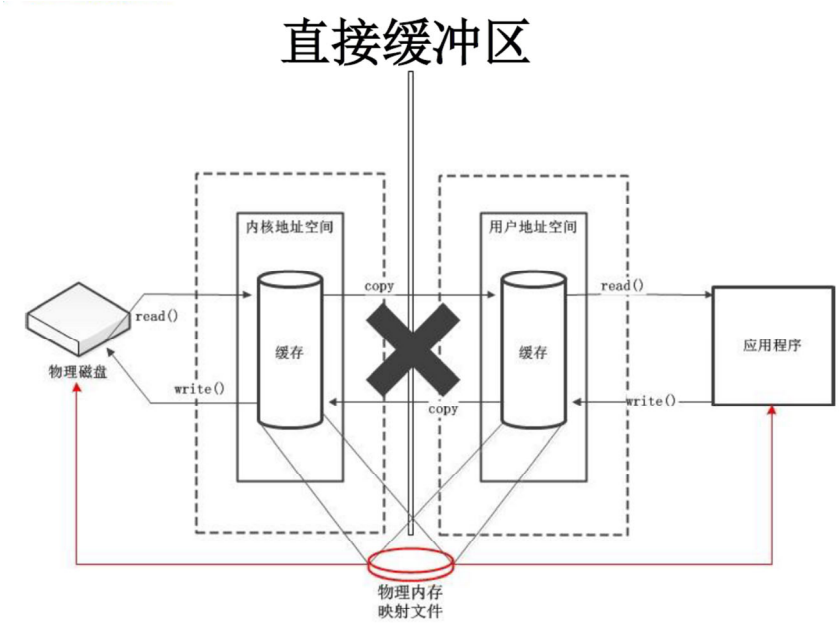
- 建议将直接缓冲区分配给那些**容易受到基础系统 IO 操作影响**的**大型、持久的缓冲区**，最好浸在直接缓冲区能在程序性能方面带来明显好处的时候分配他们；
- 直接字节缓冲区还可以通过 **FileChannel.map()** 方法将文件区域直接映射到内存中来创建，该方法返回一个 MappedByteBuffer；
- 字节缓冲区是直接缓冲区还是非直接缓冲区可以通过 **isDirect()** 来确定。

直接与非直接缓冲区

【左边是 OS，右边是 JVM】



我们要读取磁盘中数据的时候，首先会先向操作系统发起一个 read() 请求，而磁盘中的数据没法直接传输到我们的应用程序中来，
它要经历几个过程：首先会将数据读取到**内核地址空间**中，然后向**用户地址空间（即 JVM 内存）**中 copy。
写操作则同理，**传统的 IO 和非直接缓冲 allocate() 使用的都是这种方式。**



而在 JAVA NIO 中提出了**内存映射文件**，即直接缓冲区，该概念取消了内核地址空间和 JVM 内存之间的拷贝操作，
建立了一个物理内存映射文件，应用程序和物理磁盘进行数据交换不再需要经过繁琐的 copy 环节来实现，
直接**面向缓冲区操作即可**。该缓冲区能一定程度上提升效率，不过也有一定的缺点：如果在 JVM 中直接开辟一块缓冲区，
对于内存资源的分配和销毁耗费也是很大的，另外，在 JAVA 应用程序将数据写入物理内存映射文件以后，就脱离了 JVM 的管辖范围，
而物理内存映射文件中的数据什么时候写到物理磁盘中去，我们也无法进行操作及干涉，充满不确定性。

```
@Test
public void test02() {
    //分配直接缓冲区
    ByteBuffer byteBuffer = ByteBuffer.allocateDirect(1024);

}
```

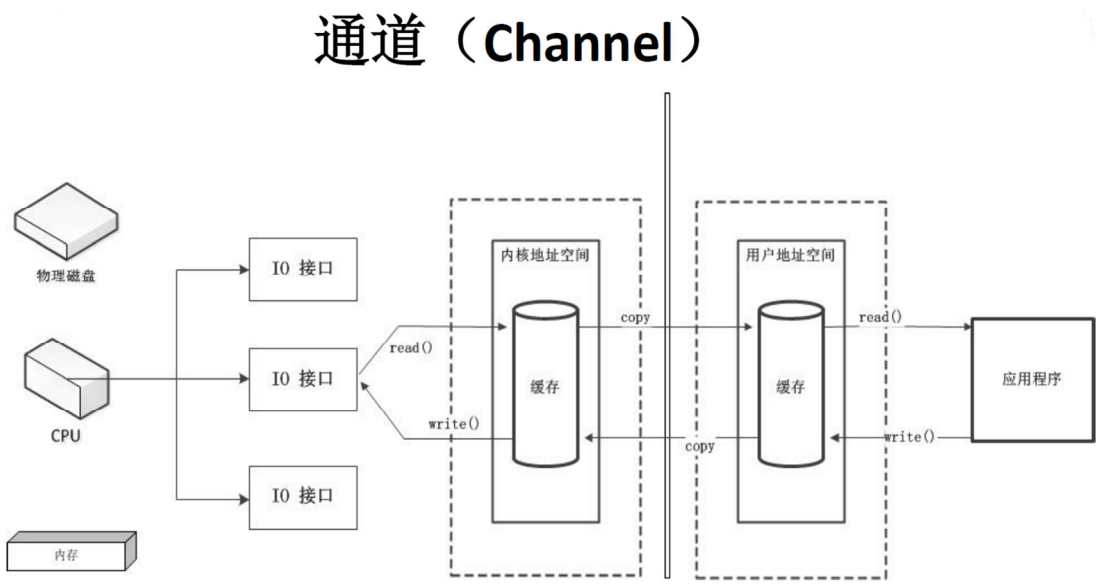
4.通道（Channel）的原理与获取

2020年8月25日 19:18

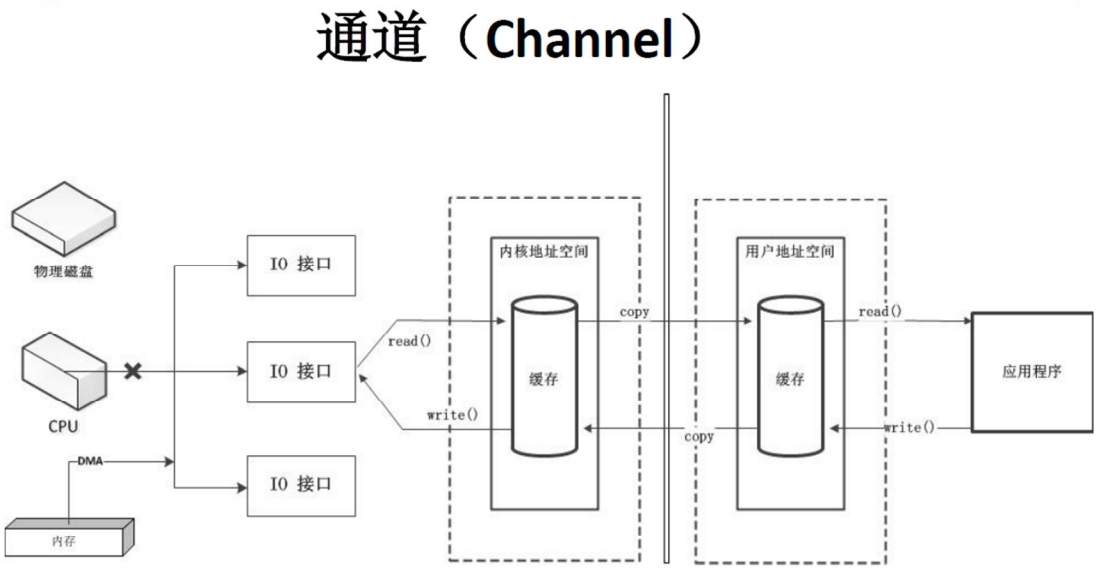
通道（Channel）

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。

【通道原理图】

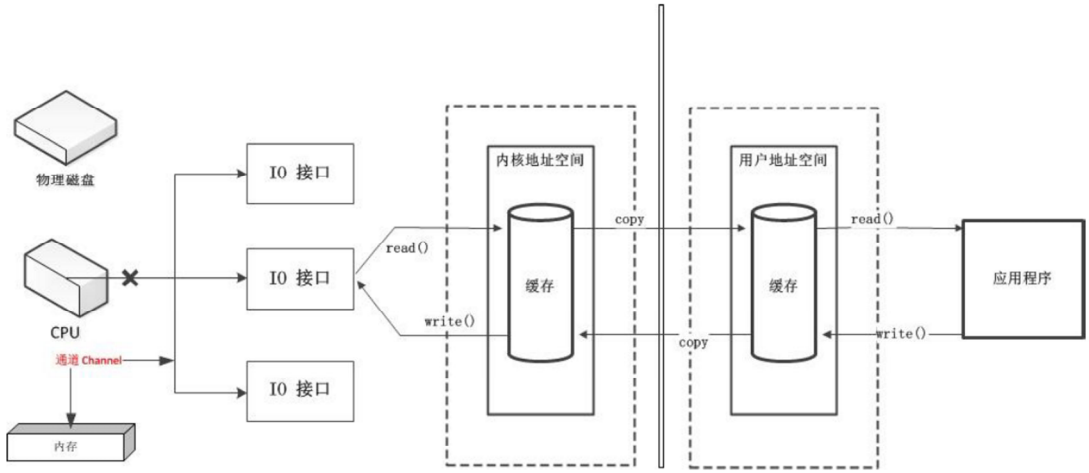


其实应用程序对于本地内存中数据读写实质上是通过 IO 接口来完成的，
由于 IO 接口都是交由 CPU 来管理，当多个 IO 接口一起工作的时候会造成 CPU 资源部分被浪费以及无法很好的利用，
所以后来改成如下图架构：



图中的 DMA 即直接存储器存取，这种架构的特点是：当有 IO 请求发到本地时，DMA 会向 CPU 申请权限，
让 CPU 将 IO 工作全权交给 DMA 来负责，这样 CPU 就不用干涉 IO 操作，可以腾出来去处理其他的操作，
传统的读写方式其实就是 DMA，也就是我们所说的 IO 流，这种 DMA 的架构也存在一些问题，
当大量的 IO 请求发送到本地时，DMA 也会向 CPU 申请资源，然后建立出更多的 DMA 总线，如果 DMA 总线过多，
就可能发生**总线冲突**的问题，最终也会影响性能，后来基于这个问题进行改进，就出现了**通道（Channel）**：

通道（Channel）



通道和 DMA 又有什么不同呢？通道是**完全独立的处理器**，专门用于 IO 操作（虽然它是完全独立的处理器，但仍附属于 CPU），并且拥有一套自己的命令。

Java 为 Channel 接口提供的主要实现类

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 本身不储存数据，因此需要配合缓冲区（Buffer）进行使用：

FileChannel：用于读取、写入、映射和操作本地文件的通道；
DatagramChannel：通过 UDP 读写网络中的数据通道**【用于网络 IO】**；
SocketChannel：通过 TCP 读写网络中的数据**【用于网络 IO】**；
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**【用于网络 IO】**。

如何获取通道？

在 JDK 1.7 以后支持三种方式获取通道：

（1）JAVA 针对支持通道的类提供了一个 **getChannel()** 方法；

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：

【本地 IO】FileInputStream、FileOutputStream、RandomAccessFile

【网络 IO】DatagramSocket、Socket、ServerSocket

如：FileInputStream.getChannel()

（2）JDK 1.7 中的 NIO 2 通过通道的静态方法 **open()** 打开并返回指定通道。

（3）JDK 1.7 中的 NIO 2 获取通道的方式是使用 Files（**是个工具类**） 类的静态方法 **newByteChannel()** 获取字节通道；

5.NIO 通道的数据传输与内存映射文件

2020年8月26日 23:56

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);
```

（1）利用通道完成文件的复制 **（非直接缓冲区）**

```java
public class TestChannel {
    //利用通道完成文件的复制
    @Test
    public void test01() throws IOException {
        FileInputStream fis = new FileInputStream( name: "1.jpg");
        FileOutputStream fos = new FileOutputStream( name: "2.jpg");

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

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

        //将通道中的数据存入缓冲区中
        while (inChannel.read(buf) != -1) {
            //要记得将缓冲区切换成读模式
            buf.flip();
            //将缓冲区中的数据写入通道中
            outChannel.read(buf);
            buf.clear();    //清空缓冲区
        }

        //操作完以后要记得关闭通道
        outChannel.close();
        inChannel.close();
    }
}
```

【其中注意两个点：读取的时候要记得将缓冲区切换成读模式；读取过后要清空缓冲区】

【代码中关闭步骤为简写，实际开发中一定要加 try catch 块，判断不为 null 以后进行关闭】

复制成功。

（2）利用通道完成文件的复制 **（直接缓冲区）（使用内存映射文件的方式）**

使用 FileChannel.open() 来对文件进行操作：

第一个参数 path，即文件的路径（文件路径可以使用逗号隔开，表示对路径进行拼接）；

第二个参数 options，即想要对文件进行的操作；

在 options 中可以使用 JDK 提供的枚举类 StandardOpenOption 来选择要对文件进行的操作：

```
valueOf(String name)          StandardOpenOption
values()                      StandardOpenOption[]
APPEND                        StandardOpenOption
CREATE                        StandardOpenOption
CREATE_NEW                    StandardOpenOption
class
DELETE_ON_CLOSE               StandardOpenOption
DSYNC                         StandardOpenOption
READ                          StandardOpenOption
SPARSE                        StandardOpenOption
SYNC                          StandardOpenOption
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards >>
StandardOpenOption.);
```

而 options 中也可以指定多个 StandardOpenOption 对数据进行操作，如我们在 WRITE 之后进行 CREATE 选项，
其中 CREATE 意为如果文件不存在就创建，文件存在则覆盖；而 CREATE_NEW 意为如果文件不存在就创建，文件存在则报错：

```
使用逗号隔开，表示对路径进行拼接）   CREATE                StandardOptio
                                    CREATE_NEW            StandardOpenOption
Paths.get( first: "1.jpg"), Standa  Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards >>
StandardOpenOption.WRITE,StandardOpenOption.Crea);
```

接下来我们基于内存映射文件的方法调用 map()，map 有 3 个参数分别是：
【该方法原理与 allocateDirect() 方法一样，只不过获取方式不一样而已】
【直接缓冲区的方式只支持 ByteBuffer，其他不支持】

mode：map 的模式，有 **PRIVATE、READ_ONLY【仅读取】、READ_WRITE【读写模式，可以理解为写】**；

```
blic void test02() throws IOException {
  //第一个参  MapMode (java.nio.channels.FileChannel)
  //第二个参  MapMode.PRIVATE (java.nio.channels.FileCha… MapMode
  FileChanne  MapMode.READ_ONLY (java.nio.channels.FileC… MapMode
  FileChanne  MapMode.READ_WRITE (java.nio.channels.File… MapMode
  Press Ctrl+空格 to see non-imported classes >>
  inChannel.map(MapMode);
```

position：与 Buffer 中的 position 属性同义；
size：要读的长度是多少。

这种方式和通道的方式有什么不同呢？
【效率会略高一些，因为它传输数据**不再需要通过通道完成**，**当前的缓存区直接存在于内存中，往缓冲区中存放数据就等同于直接放入物理内存中**】
【省去了操作通道的步骤，改为直接操作缓冲区】。

```
@Test
public void test02() throws IOException {
    //第一个参数 path，即文件的路径（文件路径可以使用逗号隔开，表示对路径进行拼接）
    //第二个参数 options，即想要对文件进行的操作
    FileChannel inChannel = FileChannel.open(Paths.get( first: "1.jpg"), StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get( first: "3.jpg"), StandardOpenOption.WRITE, StandardOpenOption.CREATE_NEW);

    //该方法原理与 allocateDirect() 方法一样，只不过获取方式不一样而已
    MappedByteBuffer inMapBuffer = inChannel.map(FileChannel.MapMode.READ_ONLY,  position: 0, inChannel.size());
    MappedByteBuffer outMapBuffer = outChannel.map(FileChannel.MapMode.READ_WRITE,  position: 0, outChannel.size());

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

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

```
java.nio.channels.NonReadableChannelException
    at sun.nio.ch.FileChannelImpl.map(FileChannelImpl.java:876)
    at TestChannel.test02(TestChannel.java:56) <25 internal calls>
```

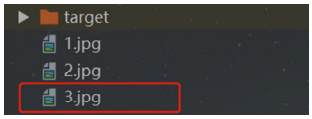
运行代码后发现报错，原因因为下面 outChannel.map 中定义的是 READ_WRITE 读写模式，
而上面 outChannel 定义中只定义了写，没有给他定义读模式，所以要加上才可以进行读写操作。

```java
@Test
public void test02() throws IOException {
    //第一个参数 path，即文件的路径（文件路径可以使用逗号隔开，表示对路径进行拼接）
    //第二个参数 options，即想要对文件进行的操作
    FileChannel inChannel = FileChannel.open(Paths.get( first: "1.jpg"), StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get( first: "3.jpg"), StandardOpenOption.WRITE,
                                              StandardOpenOption.READ, StandardOpenOption.CREATE_NEW);

    //该方法原理与 allocateDirect() 方法一样，只不过获取方式不一样而已
    MappedByteBuffer inMapBuffer = inChannel.map(FileChannel.MapMode.READ_ONLY,  position: 0, inChannel.size());
    MappedByteBuffer outMapBuffer = outChannel.map(FileChannel.MapMode.READ_WRITE,  position: 0, outChannel.size());

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

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



复制操作成功。

（3）通道之间的数据传输（直接缓冲区）

```java
@Test
public void test03() throws IOException {
    FileChannel inChannel = FileChannel.open(Paths.get( first: "1.jpg"), StandardOpenOption.READ);
    FileChannel outChannel = FileChannel.open(Paths.get( first: "4.jpg"), StandardOpenOption.WRITE,
            StandardOpenOption.READ, StandardOpenOption.CREATE_NEW);

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

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

利用 transferTo() 和 transferFrom() 来进行数据的传输，
其中三个参数为 position（同 buffer 中的 position）、count（传输的最大字节数，必须为非负数）、target（目标通道）。
该方式也是通过直接缓冲区进行数据的读写，相较于第二种方式代码行数更少，更简洁。

transferFrom()

- 将数据从源通道传输到其他 Channel 中：

```java
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);
```

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);
```

FileChannel 的常用方法

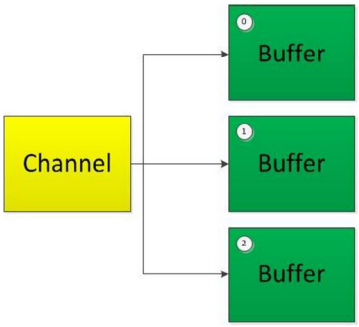
方　法	描　述
int read(ByteBuffer dst)	从 Channel 中读取数据到 ByteBuffer
long read(ByteBuffer[] dsts)	将 Channel 中的数据“分散”到 ByteBuffer[]
int write(ByteBuffer src)	将 ByteBuffer 中的数据写入到 Channel
long write(ByteBuffer[] srcs)	将 ByteBuffer[] 中的数据“聚集”到 Channel
long position()	返回此通道的文件位置
FileChannel position(long p)	设置此通道的文件位置
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

6.分散（Scatter）与聚集（Gather）

2020年8月27日 11:04

分散读取（Scattering Reads）

分散读取（Scattering Reads）是指从 Channel 中读取的数据“**分散**”到多个缓冲区（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。

```
//获取通道
File
    read(ByteBuffer dst)                                   int
    read(ByteBuffer[] dsts)                               long
//分 read(ByteBuffer dst, long position)                   int
Byte read(ByteBuffer[] dsts, int offset, int lengt…       long
Byte    reqnonnull              Objects.requireNonNull(expr)
        return                                   return expr
//分 Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards >>   π
channel.re
```

此处我们使用 RandomAcessFile**【本地 IO】** 来获取通道，在参数 mode 部分分有四种模式：

r	以只读的方式打开文本，也就意味着不能用write来操作文件
rw	读操作和写操作都是允许的
rws	每当进行写操作，同步的刷新到磁盘，刷新内容和元数据
rwd	每当进行写操作，同步的刷新到磁盘，刷新内容

建立好通道以后我们可以建立 2 个缓冲区进行数据的装填，再调用 read() 方法，其中第二个 read() 方法的参数为缓冲区数组对象，正是我们所需要的，所以我们将多个 Buffer 放入缓冲区数组中，再填入 read() 方法的参数里：

```
@Test
public void test04() throws IOException {

    RandomAccessFile raf1 = new RandomAccessFile( name: "1.txt", mode: "rw");

    //获取通道
    FileChannel channel = raf1.getChannel();

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

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

    for (ByteBuffer buf : bufs) {
        buf.flip();
    }

    System.out.println(new String(bufs[0].array(), offset: 0, bufs[0].limit()));
    System.out.println("----------------------------------------------------------");
    System.out.println(new String(bufs[1].array(), offset: 0, bufs[1].limit()));
}
```

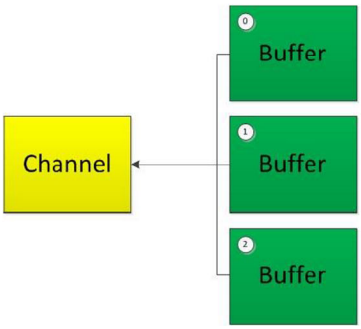
【然后我们对 buffer 内容进行读取并输出结果，第一个 buffer 装填了字节长度为 100 的内容，第二个 buffer 装填了字节长度为 1024 的内容】

```
反对外国侵略者的斗争
1836年，英国驻华商务义律扬言用武力打开国门
1839
----------
，林则徐 —《四洲志》 睁眼看世界第一人
魏源《海国图志》—“师夷长技以制夷”
19世纪初，英国向中国走私鸦片
1839年6月，林则徐虎门销烟
1840年，第一次鸦片战争：与英国签订《南京条约》（史上第一次不平等条约），割让香港岛；
与法国签订《黄埔条约》、与美国签订《望厦条约》
殉国将领：关天培、陈化成、海龄
人民群众第一次反侵略武装斗争：三元里人民的抗英斗争
1856年，英国利用“亚罗号事件”、法国利用“马神甫事件”，发起第二次鸦片战争
1858年，清政府与英、法、俄、美签订《天津条约》，
（1）增加牛庄、汉口、九江、南京等10处作为通商口岸；
（2）增加天津为商埠；
（3）割让九龙地方一区，即九龙半岛界限街以南地区，归入英属香港界内；
（4）赔偿英、法联军各增至白银800万两。
1860年，英法联军火烧圆明园、清政府和法
```

结果证明了该操作完成了**分散读取（Scattering Reads）**。

聚集写入（Gathering Writes）

聚集写入（Gathering Writes）是指将多个 Buffer 中的数据“**聚集**”到一个通道中（Channel）。



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

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

channel2.write(bufs);

channel.close();
channel2.close();
```

同样是进行通道的获取和创建，我们使用通道 2 进行数据的**写操作**，将 bufs 里面的内容写入到名为 "2.txt" 的目标文本中

【其中 write() 的参数放的是存放数据源的缓冲区数组**（源码中将该数组命名为 srcs，即"数据源"s，src 即 source，含义就是数据源文件）**】

```java
public final long write(ByteBuffer[] srcs) throws IOException {
    return write(srcs,  offset: 0, srcs.length);
}
```

执行代码过后：

2.txt

```
，林则徐 —《四洲志》 睁眼看世界第一人
魏源《海国图志》—“师夷长技以制夷”
19世纪初，英国向中国走私鸦片
1839年6月，林则徐虎门销烟
1840年，第一次鸦片战争：与英国签订《南京条约》（史上第一次不平等条约），割让香港岛；
与法国签订《黄埔条约》、与美国签订《望厦条约》
殉国将领：关天培、陈化成、海龄
人民群众第一次反侵略武装斗争：三元里人民的抗英斗争
1856年，英国利用“亚罗号事件”、法国利用“马神甫事件”，发起第二次鸦片战争
1858年，清政府与英、法、俄、美签订《天津条约》，
（1）增加牛庄、汉口、九江、南京等10处作为通商口岸；
（2）增加天津为商埠；
（3）割让九龙地方一区，即九龙半岛界限街以南地区，归入英属香港界内；
（4）赔偿英、法联军各增至白银800万两。
1860年，英法联军火烧圆明园、清政府和法
```

7.字符集（CharSet）

2020年8月27日 12:02

编码与解码

编码：字符串转换成字节数组的过程
解码：字节数组转换成字符串的过程

CharSet 支持多少种字符集呢

```
@Test
public void test05() {
    Map<String, Charset> map = Charset.availableCharsets();
    Set<Map.Entry<String, Charset>> entries = map.entrySet();

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

结果输出一百七十多个字符集

```
CESU-8=CESU-8
EUC-JP=EUC-JP
EUC-KR=EUC-KR
GB18030=GB18030
GB2312=GB2312
GBK=GBK
IBM-Thai=IBM-Thai
IBM00858=IBM00858
```

如何制定编码方式

我们可以通过 Charset.forName()，在其中以字符串的形式来指定编码格式：

```
Charset gbk = Charset.forName("GBK");
```

接着我们就可以通过这个返回的字符集来获取**编码器（EnCoder）**与**解码器（DeCoder）**：

```
//获取编码器与解码器
//编码器
CharsetEncoder charsetEncoder = gbk.newEncoder();

//解码器
CharsetDecoder charsetDecoder = gbk.newDecoder();
```

随后就可以对 Buffer 中的内容进行编码与解码

```java
@Test
public void test06() throws CharacterCodingException {

    Charset gbk = Charset.forName("GBK");

    //获取编码器与解码器
    //编码器
    CharsetEncoder charsetEncoder = gbk.newEncoder();

    //解码器
    CharsetDecoder charsetDecoder = gbk.newDecoder();

    CharBuffer cBuf = CharBuffer.allocate(1024);
    cBuf.put("EkoHwang");
    cBuf.flip();

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

    for (int i = 0; i < 8; i++) {
        System.out.println(bBuf.get());
    }

    //解码
    bBuf.flip();
    CharBuffer cBuff = charsetDecoder.decode(bBuf);
    System.out.println(cBuff.toString());
}
```

编码过程即在 CharBuffer 中存入数据“EkoHwang”，然后切换成读模式，调用编码器转换为 GBK 格式的内容，输入到 ByteBuffer 中；
解码过程则是调用解码器，将存在 ByteBuffer 中的 GBK 格式内容转换成未编码前的内容，然后进行输出。
而有时候输出的内容出现乱码，原因是因为编码集与解码集不是同一个字符集，
如使用 GBK 的编码器进行编码，却使用 UTF - 8 的解码器进行解码，则会出现乱码的现象。

```java
//使用 UTF-8 解 GBK 的编码集
Charset UTF8 = Charset.forName("UTF-8");
bBuf.flip();
CharBuffer decode = UTF8.decode(bBuf);
System.out.println(decode.toString());
```

```
你好世界
----------------------------------------
????????
```

8.NIO 的非阻塞式网络通信

2020年8月27日 15:36

阻塞式

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；

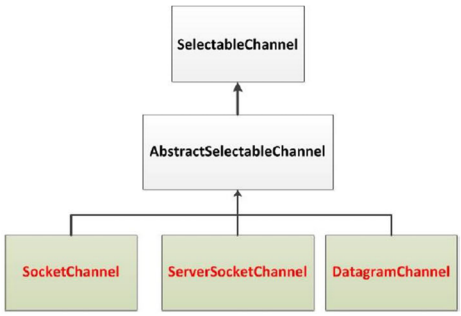
【个人理解】
通俗来讲就是当客户端向服务端发送一个请求以后，如果说**服务端无法确定客户端发送过来的读写请求数据有效**或者**无法判断里面是否存在数据时候**，
该**线程**就会一直处于**阻塞状态**，在阻塞期间，服务端的这个线程此时是不会去完成其他事情的，
这也就意味着，当客户端发送大量请求过来时，前面的如果完成不了，后面如果还有请求就得排队。
早期解决该问题的方法就是多线程，对于客户端发送过来的请求，我们给每一个请求都分配一个独立的线程让他们去完成任务，
多线程做法的好处就是充分利用了 CPU 资源，然而即便是用了多线程，应用服务器上的线程数量也是有限的，如果多数线程处于阻塞状态，仍然会造成整体的阻塞，
仍然是无法很好的利用 CPU 资源，只是**相对来说**提高了利用率而已。

非阻塞式

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**。

【个人理解】
NIO 相较于传统的 IO，提出了**选择器（Selector）**的概念，**选择器会将所有用于传输数据的通道都注册到选择器之中**，
选择器的功能就是用于**监控这些通道的 IO 状况**，即通道内的读、写、连接、接收数据等等情况...
当客户端发送过来的通道中的数据完全准备就绪的时候，选择器就会将这个任务分配给服务端的一个或多个线程上，再去运行
这就意味着如果客户端的这些请求尚未准备就绪，服务端是不会分配出线程来给这些未准备就绪的任务的。
这种非阻塞式的架构比起传统阻塞式更能发挥 CPU 的性能，**选择器则是非阻塞式 IO 的核心**。

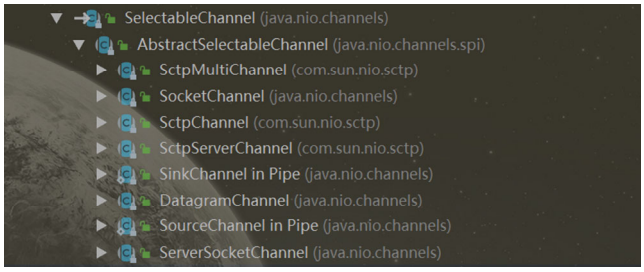
- 选择器（Selector）是 SelectableChannle 对象的多路复用器，Selector 可以同时监控多个 SelectableChannel 的 IO 状况，也就是说，利用 Selector 可使一个单独的线程管理多个 Channel。Selector 是非阻塞 IO 的核心。
- SelectableChannle 的结构如下图：



阻塞式和非阻塞式 IO 就像日常生活中取快递，如果你知道快递两点到，一点五十就跑去等快递，这时候你就做不了其他的事情，就是阻塞式的
而非阻塞式就像是快递已经到你楼下了，打电话叫你下楼去取件，这时候才下楼去取，即已经完全准备好了才去执行，这就是非阻塞式的。

使用代码来示范阻塞与非阻塞完成任务

若要使用**非阻塞式 NIO** 完成网络通信需要有三个核心：
（1）Channel：即通道，负责连接；
（2）Buffer：即缓冲区，负责数据的存取；
（3）Selector：即选择器，是 SelectableChannel 的多路复用器，用于监控 SelectableChannel 的 IO 状况。




```
public abstract class AbstractSelectableChannel
    extends SelectableChannel
{
```

【SelectableChannel 的包结构】：

间接继承了该接口的类有：**SocketChannel、ServerSocketChannel、DatagramChannel**；

以及：**Pipe.SinkChannle、Pipe.SourceChannel;**

可以观察到其中没有 FileChannel 的加入，由此可知，**FileChannel 不能切换成非阻塞模式**。

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);
```

【代码部分】

（1）模拟客户端与服务端**（阻塞式）**：

【客户端】：使用 SocketChannel 来传输数据

```
//客户端
@Test
public void client() throws IOException {
    //1.获取通道
    SocketChannel sChannel = SocketChannel.open(new InetSocketAddress( hostname: "127.0.0.1",  port: 9898));

    //用于读取本地文件的Channel
    FileChannel inChannel = FileChannel.open(Paths.get( first: "001.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，并发送数据到服务端
    while (inChannel.read(buf) != -1) {
        buf.flip();
        sChannel.write(buf);
        buf.clear();
    }

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

【服务端】：顾名思义使用 ServerSocketChannel 来获取数据：

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

    FileChannel outChannel = FileChannel.open(Paths.get( first: "002.jpg"), StandardOpenOption.WRITE, StandardOpenOption.CREATE);

    //2.绑定连接端口号
    ServerSocketChannel bind = ssChannel.bind(new InetSocketAddress( hostname: "127.0.0.1",  port: 9898));

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

    //分配一个指定大小的缓冲区来接收数据
    ByteBuffer buf = ByteBuffer.allocate(1024);

    //4.接收客户端数据并保存到本地（保存到本地也需要一个FileChannel）
    while (sChannel.read(buf) != -1) {
        buf.flip();
        outChannel.write(buf);
        buf.clear();
    }

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

先运行服务端再运行客户端，传输成功。
不过成功过后控制台没有输出任何信息，所以我们可以给他加上接受反馈的指令：

```java
//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(),  offset: 0, len));
    buf.clear();
}
```

```java
//发送反馈给客户端
buf.put("服务端接收客户端数据成功".getBytes());
buf.flip();
sChannel.write(buf);
```

运行后发现程序没有反应，呈阻塞状态，此时有两种方式解决，一种是让客户端通道 shutdown，另一种是将线程切换成非阻塞模式
此处先用 shutdown 来解除阻塞。

```java
sChannel.shutdownOutput();
```

```
"C:\Program Files\Java\jdk1.8.0_251\bin\java.exe" ...
服务端接收客户端数据成功
```

【代码部分】

（2）模拟客户端与服务端**（非阻塞式）**：
若要使用**非阻塞式 NIO** 完成网络通信需要有三个核心：

（1）Channel：即通道，负责连接；
（2）Buffer：即缓冲区，负责数据的存取；
（3）Selector：即选择器，是 SelectableChannel 的多路复用器，用于监控 SelectableChannel 的 IO 状况。

想要切换成非阻塞模式，必须调用方法 sChannel.configureBlocking（**false**）来完成**（客户端和服务端都要执行）**：

```java
//2.切换成非阻塞模式
sChannel.configureBlocking(false);
```

【客户端】

```java
//客户端
@Test
public void client() throws IOException {
    //1.获取通道
    SocketChannel sChannel = SocketChannel.open(new InetSocketAddress( hostname: "127.0.0.1",  port: 9898));

    //2.切换成非阻塞模式
    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 scanner = new Scanner(System.in);
    while (scanner.hasNext()) {
        String str = scanner.next();
        buf.put((LocalDateTime.now().toString() + "\n" + str).getBytes());
        buf.flip();
        sChannel.write(buf);
        buf.clear();
    }

    //5.关闭通道
    sChannel.close();
}
```

【服务端】
在服务端还需要开启选择器，使用 **Selector.open()** 来获取，
在获取到选择器以后，就需要**将通道注册到选择器上**，使用通道的 **register()** 方法，其中有两个参数，
第一个参数需要传入一个选择器，第二个参数则是**选择键**（**即需要选择器监控该通道的什么状态**）；

> **选择键（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。

选择键可以监听的事件有四种，可以用四个常量来表示：

读：SelectionKey.OP_READ （1）；
写：SelectionKey.OP_WRITE （4）；
连接：SelectionKey.OP_CONNECT（8）；
接收：SelectionKey.OP_ACCEPT （16）；

如果监听的时间不止一个，则可以通过**按位或（“|”）**来连接。

选择器（Selector）的应用

- 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：

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

而此时我们在选择器中暂定监听他的“接收状态”，和之前阻塞式的 **accept()** 不同的是，此处的 **OP_ACCPET 会等准备就绪之后再去获取连接**。

```java
//5.把通道注册到选择器上，并且指定监听"接收事件"
SelectionKey register = ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

【服务端】

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

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

    //3.绑定连接
    ssChannel.bind(new InetSocketAddress( hostname: "127.0.0.1",  port: 9898));

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

    //5.把通道注册到选择器上，并且指定监听"接收事件"
    SelectionKey register = ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

【1. 记得切换成非阻塞模式】；

【2. 客户端使用的是 SocketChannel，而服务端用的则是 ServerSocketChannel】；

【3. 通过通道可以将通道注册到选择器上，并指定需要选择器监控哪种行为】；

```java
//6.轮询式地获取选择器上已经准备就绪的事件
//当 selector.select() 的返回值大于 0 则代表当前已经至少有一个准备就绪的通道
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> iterator = selector.selectedKeys().iterator();

    //8.迭代获取
    while (iterator.hasNext()) {
        SelectionKey sKey = iterator.next();    //获取准备就绪的事件

        //9.判断具体是什么事件准备就绪了
        if (sKey.isAcceptable()) {
            //如果接收状态就绪，则获取客户端连接
            SocketChannel sChannel = ssChannel.accept();

            //10.记得要把客户端通道也切换成非阻塞
            sChannel.configureBlocking(false);

            //11.将该通道也注册到选择器上并监控他的"读数据状态"
            sChannel.register(selector, SelectionKey.OP_READ);

        } else if (sKey.isReadable()) {
            //12.如果是读就绪状态，则要获取当前选择器上"读就绪"的通道
            SocketChannel sChannel = (SocketChannel) sKey.channel();
```

【4. 要先获取选择器中所有的选择键，才能根据选择键的状态来判断具体是什么事件准备就绪了，需要执行什么操作】；

【5. 新声明的通道也要设置成非阻塞模式】；

```java
            //13.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(),  offset: 0, len));
                buf.clear();
            }
        }

        //14.SelectionKey使用完毕后要记得取消，不取消的话就会一直有效
        //取消则通过迭代器的 remove() 方法迭代删除即可
        iterator.remove();
    }
}
```

【6. 注意在使用完 SelectionKey 以后要记得取消，否则会一直有效下去，**使用迭代器的 remove()** 即可取消】；

客户端工作完之后，服务端仍未关闭，会持续进行轮询，客户端重复运行多次，就得出多个结果：

```
Run:  TestNonBlockingNIO.client   TestNonBlockingNIO.server
      Tests passed: 0 of 1 test
TestNonBlockingNIO
    server
Thu Aug 27 18:58:33 CST 2020
Thu Aug 27 18:58:55 CST 2020
```

9.DatagramChannel

2020年8月27日 20:45

DatagramChannel

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

datagramChannel 的用法也和其他几个 channel 大同小异，只是将其中 **read() 以及 write()** 换成了 **send() 和 receive()**

```java
@Test
public void send() throws IOException {
    DatagramChannel dc = DatagramChannel.open();
    dc.configureBlocking(false);
    ByteBuffer buf = ByteBuffer.allocate(1024);
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNext()) {
        String str = scanner.next();
        buf.put((LocalDateTime.now().toString() + "\n" + str).getBytes());
        buf.flip();
        dc.send(buf, new InetSocketAddress( hostname: "127.0.0.1",  port: 9898));
        buf.clear();
    }
    dc.close();
}
```

```java
@Test
public void receive() throws IOException {
    DatagramChannel dc = DatagramChannel.open();
    dc.configureBlocking(false);
    dc.bind(new InetSocketAddress( port: 9898));
    Selector selector = Selector.open();
    dc.register(selector, SelectionKey.OP_READ);
    while (selector.select() > 0) {
        Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
        while (iterator.hasNext()) {
            SelectionKey next = iterator.next();
            if (next.isReadable()) {
                ByteBuffer buf = ByteBuffer.allocate(1024);
                dc.receive(buf);
                buf.flip();
                System.out.println(new String(buf.array(),  offset: 0, buf.limit()));
                buf.clear();
            }
        }
    }
    selector.close();
    dc.close();
}
```

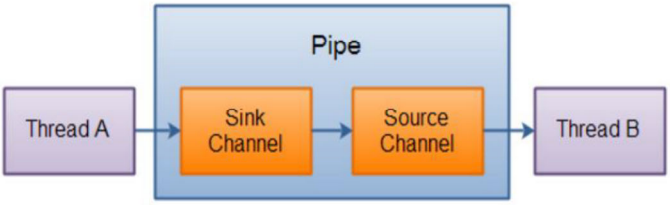
10.管道（Pipe）

2020年8月27日 20:48

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。

管道 (Pipe)

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



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);
    }
}
```

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);
```

代码实测

```java
@Test
public void test01() throws IOException {
    //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 sink = pipe.sink();
    buf.put("通过单向管道发送信息".getBytes());
    buf.flip();
    sink.write(buf);

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

    source.close();
    sink.close();
}
```

11.其他

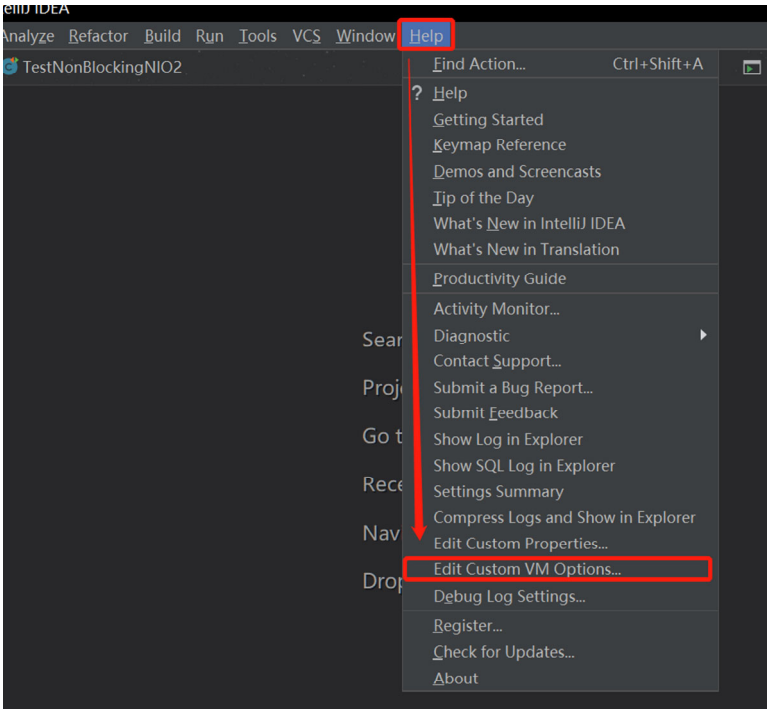
2020年8月27日 21:18

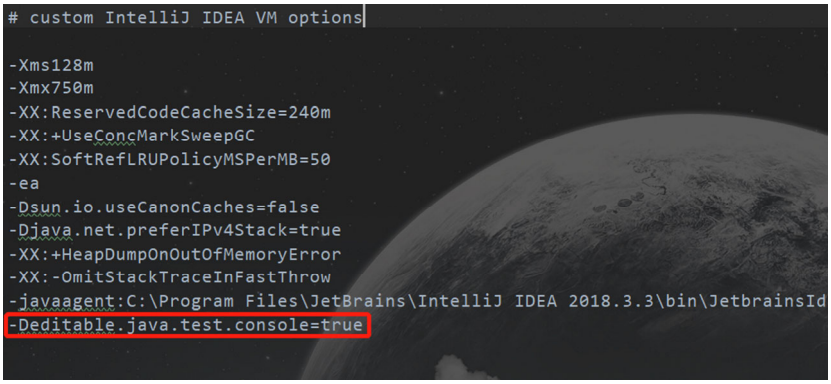
【可能遇见的问题】

解决 idea 单元测试不能使用 Scanner：

可能使用 @test 的话无法在控制台中输入信息，

此时需要添加一个 VM 参数，添加完毕后即可输入，操作如下：





加入这行在最底部即可：-Deditable.java.test.console=true

Files 类

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

Path 与 Paths

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) : 用于将多个字符串串连成路径。
- 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