

Java 网络编程系列之 NIO

尚硅谷 JavaEE 教研组

版本：V2021

内容概览

- 1、Java NIO 概述
- 2、Java NIO (Channel)
 - (1) FileChannel
 - (2) Scatter/Gather
- 3、Java NIO (SocketChannel)
- 4、Java NIO (Buffer)
- 5、Java NIO (Selector)
- 6、Java NIO (Pipe 和 FileLock)
- 7、Java NIO (其他)
 - (1) Path 和 Files
 - (2) AsynchronousFileChannel
- 8、Java NIO 综合案例

第 1 章 Java NIO 概述

1.1 IO 概述

IO 的操作方式通常分为几种：同步阻塞 BIO、同步非阻塞 NIO、异步非阻塞 AIO。

(1) 在 JDK1.4 之前，我们建立网络连接的时候采用的是 BIO 模式。

(2) Java NIO (New IO 或 Non Blocking IO) 是从 Java 1.4 版本开始引入的一个新的 IO API，可以替代标准的 Java IO API。NIO 支持面向缓冲区的、基于通道的 IO 操作。NIO 将以更加高效的方式进行文件的读写操作。BIO 与 NIO 一个比较重要的不同是，我们使用 BIO 的时候往往会引入多线程，每个连接对应一个单独的线程；而 NIO 则是使用单线程或者只使用少量的多线程，让连接共用一个线程。

(3) AIO 也就是 NIO 2，在 Java 7 中引入了 NIO 的改进版 NIO 2，它是异步非阻塞的 IO 模型。

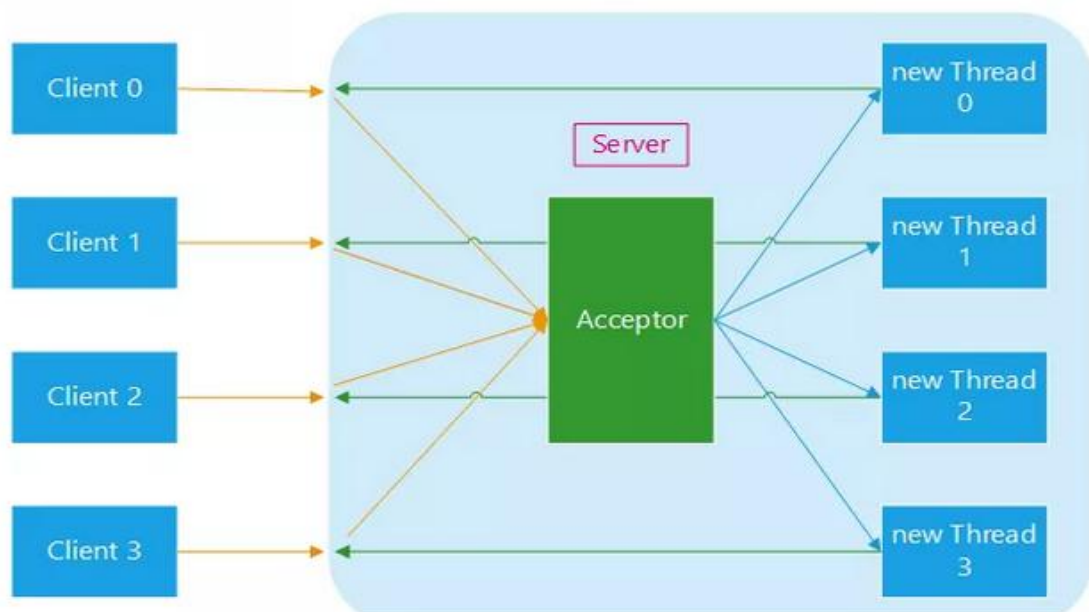
下面我们来详细介绍这几种 IO 方式

1.2 阻塞 IO (BIO)

阻塞 IO (BIO) 是最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象，直至有可供读取的数据或者数据能够写入。

(1) 在 BIO 模式中，服务器会为每个客户端请求建立一个线程，由该线程单独负责处理一个客户请求，这种模式虽然简单方便，但由于服务器为每个客户端的连接都采用一个线程去处理，使得资源占用非常大。因此，当连接数量达到上限时，如果再有用户请求连接，直接会导致资源瓶颈，严重的可能会直接导致服务器崩溃。

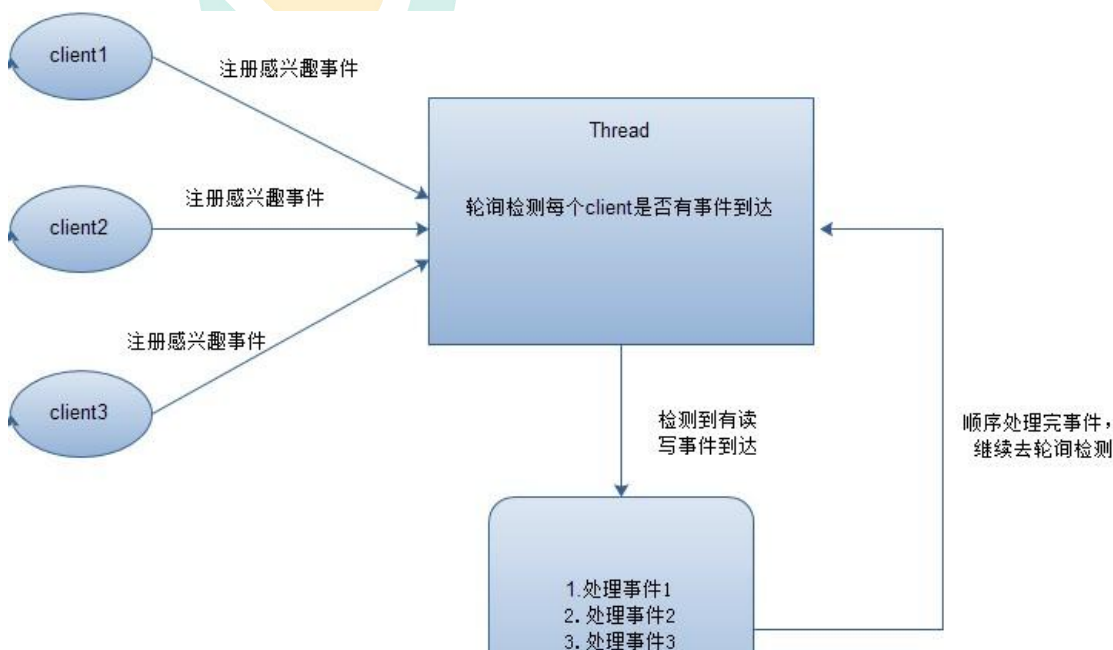
(2) 大多数情况下为了避免上述问题，都采用了线程池模型。也就是创建一个固定大小的线程池，如果有客户端请求，就从线程池中取一个空闲线程来处理，当客户端处理完操作之后，就会释放对线程的占用。因此这样就避免为每一个客户端都要创建线程带来的资源浪费，使得线程可以重用。但线程池也有它的弊端，如果连接大多是长连接，可能会导致在一段时间内，线程池中的线程都被占用，那么当再有客户端请求连接时，由于没有空闲线程来处理，就会导致客户端连接失败。传统的 BIO 模式如下图所示：



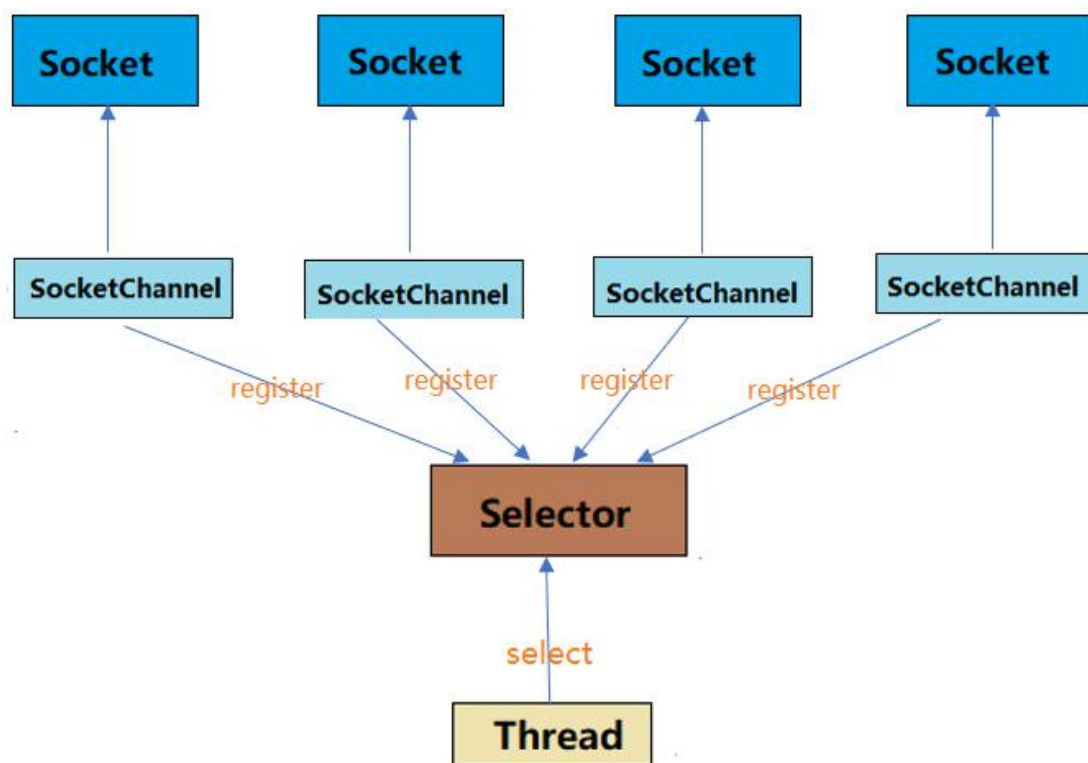
1.3 非阻塞 IO(NIO)

基于 BIO 的各种弊端，在 JDK1.4 开始出现了高性能 IO 设计模式非阻塞 IO（NIO）。

(1) NIO 采用非阻塞模式，基于 Reactor 模式的工作方式，I/O 调用不会被阻塞，它的实现过程是：会先对每个客户端注册感兴趣的事件，然后有一个线程专门去轮询每个客户端是否有事件发生，当有事件发生时，便顺序处理每个事件，当所有事件处理完之后，便再转去继续轮询。如下图所示：



(2) NIO 中实现非阻塞 I/O 的核心对象就是 Selector，Selector 就是注册各种 I/O 事件地方，而且当我们感兴趣的事件发生时，就是这个对象告诉我们所发生的事件，如下图所示：



(3) NIO 的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，一个选择器线程可以同时处理成千上万个连接，系统不必创建大量的线程，也不必维护这些线程，从而大大减小了系统的开销。

I/O	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)

1.4 异步非阻塞 IO(AIO)

(1) AIO 也就是 NIO 2，在 Java 7 中引入了 NIO 的改进版 NIO 2，它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是说 AIO 模式不需要

selector 操作，而是是事件驱动形式，也就是当客户端发送数据之后，会主动通知服务器，接着服务器再进行读写操作。

(2) Java 的 AIO API 其实就是 Proactor 模式的应用，和 Reactor 模式类似。Reactor 和 Proactor 模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor 中需要应用程序自己读取或者写入数据，而 Proactor 模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的 IO 设备。

1.5 NIO 概述

Java NIO 由以下几个核心部分组成：

- Channels
- Buffers
- Selectors

虽然 Java NIO 中除此之外还有很多类和组件，但 Channel，Buffer 和 Selector 构成了核心的 API。其它组件，如 Pipe 和 FileLock，只不过是三个核心组件共同使用的工具类。

1.5.1 Channel

首先说一下 Channel，可以翻译成“通道”。Channel 和 IO 中的 Stream(流)是差不多一个等级的。只不过 Stream 是单向的，譬如：InputStream, OutputStream.而 Channel 是双向的，既可以用来进行读操作，又可以用来进行写操作。

NIO 中的 Channel 的主要实现有：FileChannel、DatagramChannel、SocketChannel 和 ServerSocketChannel，这里看名字就可以猜出个所以然来：分别可以对应文件 IO、UDP 和 TCP (Server 和 Client) 。

1.5.2 Buffer

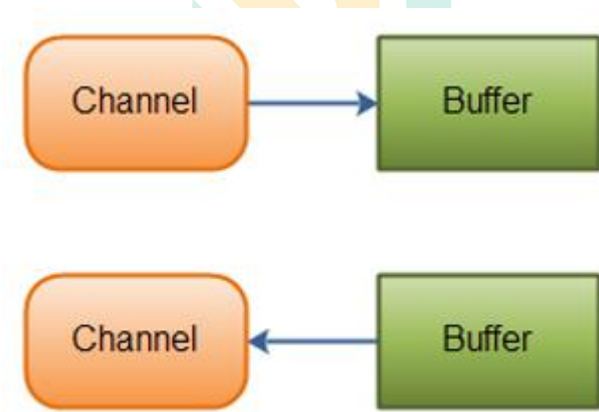
NIO 中的关键 Buffer 实现有：ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer, 分别对应基本数据类型: byte, char, double, float, int, long, short。

1.5.3 Selector

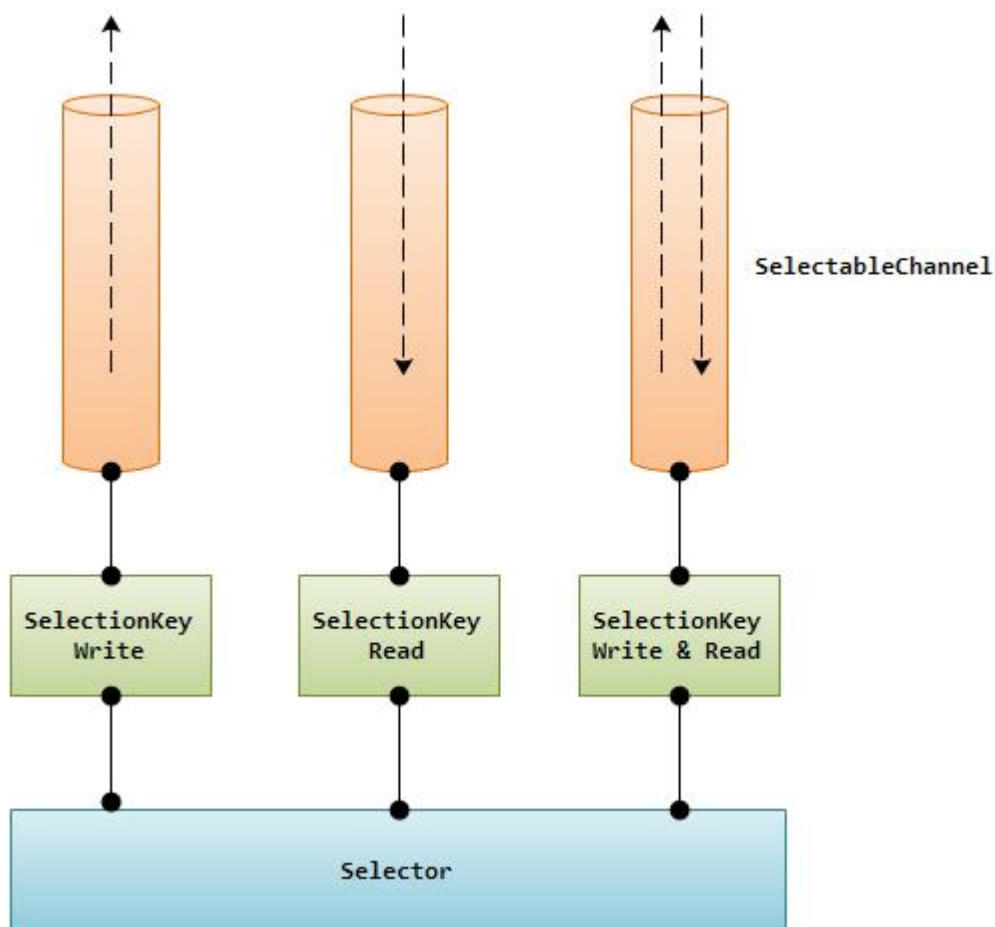
Selector 运行单线程处理多个 Channel, 如果你的应用打开了多个通道, 但每个连接的流量都很低, 使用 Selector 就会很方便。例如在一个聊天服务器中。要使用 Selector, 得向 Selector 注册 Channel, 然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回, 线程就可以处理这些事件, 事件的例子有如新的连接进来、数据接收等。

1.5.4 Channel Buffer Selector 三者关系

(1) 一个 Channel 就像一个流, 只是 Channel 是双向的, Channel 读数据到 Buffer, Buffer 写数据到 Channel。



(2) 一个 selector 允许一个线程处理多个 channel。



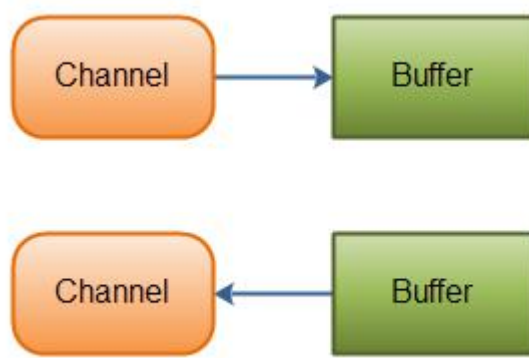
第 2 章 Java NIO (Channel)

2.1 Channel 概述

Java NIO 的通道类似流，但又有些不同：

- 既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。
- 通道可以异步地读写。
- 通道中的数据总是要先读到一个 Buffer，或者总是要从一个 Buffer 中写入。

正如上面所说，从通道读取数据到缓冲区，从缓冲区写入数据到通道。如下图所示：



2.2 Channel 实现

下面是 Java NIO 中最重要的 Channel 的实现：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

(1) FileChannel 从文件中读写数据。

(2) DatagramChannel 能通过 UDP 读写网络中的数据。

(3) SocketChannel 能通过 TCP 读写网络中的数据。

(4) ServerSocketChannel 可以监听新进来的 TCP 连接，像 Web 服务器那样。对每一个新进来的连接都会创建一个 SocketChannel。

正如你所看到的，这些通道涵盖了 UDP 和 TCP 网络 IO，以及文件 IO

2.3 FileChannel 介绍和示例

FileChannel 类可以实现常用的 read，write 以及 scatter/gather 操作，同时它也提供了很多专用于文件的新方法。这些方法中的许多都是我们所熟悉的文件操作。

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

下面是一个使用 FileChannel 读取数据到 Buffer 中的示例：

```
public class FileChannelDemo {

    public static void main(String[] args) throws IOException {
        RandomAccessFile aFile = new
RandomAccessFile("d:\\atguigu\\01.txt", "rw");
        FileChannel inChannel = aFile.getChannel();
        ByteBuffer buf = ByteBuffer.allocate(48);
        int bytesRead = inChannel.read(buf);
        while (bytesRead != -1) {
            System.out.println("读取: " + bytesRead);
            buf.flip();
            while (buf.hasRemaining()) {
                System.out.print((char) buf.get());
            }
            buf.clear();
            bytesRead = inChannel.read(buf);
        }
        aFile.close();
        System.out.println("操作结束");
    }
}
```

Buffer 通常的操作

将数据写入缓冲区

调用 `buffer.flip()` 反转读写模式

从缓冲区读取数据

调用 `buffer.clear()` 或 `buffer.compact()` 清除缓冲区内容

2.4 FileChannel 操作详解

2.4.1 打开 FileChannel

在使用 `FileChannel` 之前，必须先打开它。但是，我们无法直接打开一个 `FileChannel`，需要通过使用一个 `InputStream`、`OutputStream` 或 `RandomAccessFile` 来获取一个 `FileChannel` 实例。下面是通过 `RandomAccessFile` 打开 `FileChannel` 的示例：

```
RandomAccessFile aFile = new RandomAccessFile("d:\\atguigu\\01.txt",  
"rw");  
FileChannel inChannel = aFile.getChannel();
```

2.4.2 从 FileChannel 读取数据

调用多个 `read()` 方法之一从 `FileChannel` 中读取数据。如：

```
ByteBuffer buf = ByteBuffer.allocate(48);  
int bytesRead = inChannel.read(buf);
```

首先，分配一个 `Buffer`。从 `FileChannel` 中读取的数据将被读到 `Buffer` 中。然后，调用 `FileChannel.read()` 方法。该方法将数据从 `FileChannel` 读取到 `Buffer` 中。`read()` 方法返回的 `int` 值表示了有多少字节被读到了 `Buffer` 中。如果返回 -1，表示到了文件末尾。

2.4.3 向 FileChannel 写数据

使用 `FileChannel.write()` 方法向 `FileChannel` 写数据，该方法的参数是一个 `Buffer`。
如：

```
public class FileChannelDemo {  
  
    public static void main(String[] args) throws IOException {  
        RandomAccessFile aFile = new  
RandomAccessFile("d:\\atguigu\\01.txt", "rw");  
        FileChannel inChannel = aFile.getChannel();  
  
        String newData = "New String to write to file..." +  
System.currentTimeMillis();  
  
        ByteBuffer buf1 = ByteBuffer.allocate(48);  
        buf1.clear();  
        buf1.put(newData.getBytes());  
  
        buf1.flip();  
        while(buf1.hasRemaining()) {  
            inChannel.write(buf1);  
        }  
        inChannel.close();  
    }  
}
```

注意 `FileChannel.write()` 是在 `while` 循环中调用的。因为无法保证 `write()` 方法一次能向 `FileChannel` 写入多少字节，因此需要重复调用 `write()` 方法，直到 `Buffer` 中已经没有尚未写入通道的字节。

2.4.4 关闭 FileChannel

用完 `FileChannel` 后必须将其关闭。如：

```
inChannel.close();
```

2.4.5 FileChannel 的 position 方法

有时可能需要在 FileChannel 的某个特定位置进行数据的读/写操作。可以通过调用 position()方法获取 FileChannel 的当前位置。也可以通过调用 position(long pos)方法设置 FileChannel 的当前位置。

这里有两个例子:

```
long pos = channel.position();
```

```
channel.position(pos + 123);
```

如果将位置设置在文件结束符之后，然后试图从文件通道中读取数据，读方法将返回-1（文件结束标志）。

如果将位置设置在文件结束符之后，然后向通道中写数据，文件将撑大到当前位置并写入数据。这可能导致“文件空洞”，磁盘上物理文件中写入的数据间有空隙。

2.4.6 FileChannel 的 size 方法

FileChannel 实例的 size()方法将返回该实例所关联文件的大小。如:

```
long fileSize = channel.size();
```

2.4.7 FileChannel 的 truncate 方法

可以使用 FileChannel.truncate()方法截取一个文件。截取文件时，文件将中指定长度后面的部分将被删除。如:

```
channel.truncate(1024);
```

这个例子截取文件的前 1024 个字节。

2.4.8 FileChannel 的 force 方法

FileChannel.force()方法将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，操作系统会将数据缓存在内存中，所以无法保证写入到 FileChannel 里的数据一定会即时写到磁盘上。要保证这一点，需要调用 force()方法。

force()方法有一个 boolean 类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

2.4.9 FileChannel 的 transferTo 和 transferFrom 方法

通道之间的数据传输：

如果两个通道中有一个是 FileChannel，那你可以直接将数据从一个 channel 传输到另外一个 channel。

(1) transferFrom()方法

FileChannel 的 transferFrom()方法可以将数据从源通道传输到 FileChannel 中（译者注：这个方法在 JDK 文档中的解释为将字节从给定的可读取字节通道传输到此通道的文件中）。下面是一个 FileChannel 完成文件间的复制的例子：

```
public class FileChannelWrite {  
  
    public static void main(String args[]) throws Exception {  
        RandomAccessFile aFile = new  
RandomAccessFile("d:\\atguigu\\01.txt", "rw");  
        FileChannel fromChannel = aFile.getChannel();  
  
        RandomAccessFile bFile = new  
RandomAccessFile("d:\\atguigu\\02.txt", "rw");  
        FileChannel toChannel = bFile.getChannel();  
  
        long position = 0;  
        long count = fromChannel.size();
```

```
toChannel.transferFrom(fromChannel, position, count);

aFile.close();
bFile.close();
System.out.println("over!");
}
}
```

方法的输入参数 position 表示从 position 处开始向目标文件写入数据，count 表示最多传输的字节数。如果源通道的剩余空间小于 count 个字节，则所传输的字节数要小于请求的字节数。此外要注意，在 SocketChannel 的实现中，SocketChannel 只会传输此刻准备好的数据（可能不足 count 字节）。因此，SocketChannel 可能不会将请求的所有数据(count 个字节)全部传输到 FileChannel 中。

(2) transferTo()方法

transferTo()方法将数据从 FileChannel 传输到其他的 channel 中。

下面是一个 transferTo()方法的例子：

```
public class FileChannelDemo {

    public static void main(String args[]) throws Exception {
        RandomAccessFile aFile = new
RandomAccessFile("d:\\atguigu\\02.txt", "rw");
        FileChannel fromChannel = aFile.getChannel();

        RandomAccessFile bFile = new
RandomAccessFile("d:\\atguigu\\03.txt", "rw");
        FileChannel toChannel = bFile.getChannel();

        long position = 0;
        long count = fromChannel.size();
        fromChannel.transferTo(position, count, toChannel);

        aFile.close();
        bFile.close();
        System.out.println("over!");
    }
}
```

```
}
```

2.5 Scatter/Gather

Java NIO 开始支持 scatter/gather, scatter/gather 用于描述从 Channel 中读取或者写入到 Channel 的操作。

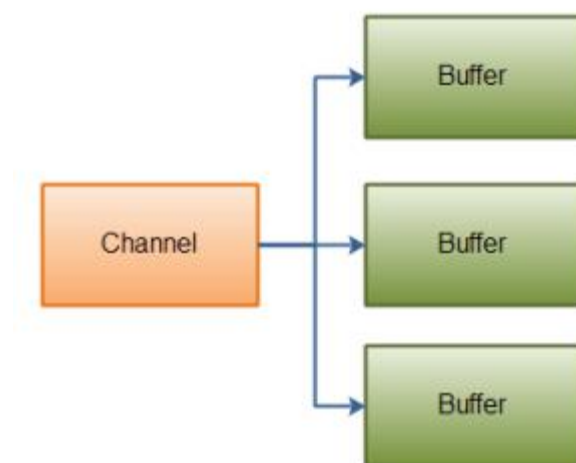
分散 (scatter) 从 Channel 中读取是指在读操作时将读取的数据写入多个 buffer 中。因此, Channel 将从 Channel 中读取的数据 “分散 (scatter)” 到多个 Buffer 中。

聚集 (gather) 写入 Channel 是指在写操作时将多个 buffer 的数据写入同一个 Channel, 因此, Channel 将多个 Buffer 中的数据 “聚集 (gather)” 后发送到 Channel。

scatter / gather 经常用于需要将传输的数据分开处理的场合, 例如传输一个由消息头和消息体组成的消息, 你可能会将消息体和消息头分散到不同的 buffer 中, 这样你可以方便的处理消息头和消息体。

2.5.1 Scattering Reads

Scattering Reads 是指数据从一个 channel 读取到多个 buffer 中。如下图描述:



```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);
```



```
ByteBuffer[] bufferArray = { header, body };
```

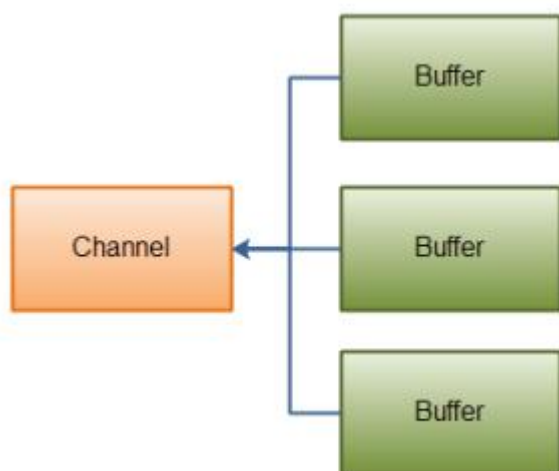
```
channel.read(bufferArray);
```

注意 buffer 首先被插入到数组，然后再将数组作为 channel.read() 的输入参数。read()方法按照 buffer 在数组中的顺序将从 channel 中读取的数据写入到 buffer，当一个 buffer 被写满后，channel 紧接着向另一个 buffer 中写。

Scattering Reads 在移动下一个 buffer 前，必须填满当前的 buffer，这也意味着它不适用于动态消息(译者注：消息大小不固定)。换句话说，如果存在消息头和消息体，消息头必须完成填充（例如 128byte），Scattering Reads 才能正常工作。

2.5.2 Gathering Writes

Gathering Writes 是指数据从多个 buffer 写入到同一个 channel。如下图描述：



```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body = ByteBuffer.allocate(1024);
```

```
//write data into buffers
```

```
ByteBuffer[] bufferArray = { header, body };
```

```
channel.write(bufferArray);
```

buffers 数组是 write()方法的入参，write()方法会按照 buffer 在数组中的顺序，将数据写入到 channel，注意只有 position 和 limit 之间的数据才会被写入。因此，如果一个 buffer 的容量为 128byte，但是仅仅包含 58byte 的数据，那么这 58byte 的数据将被写入到 channel 中。因此与 Scattering Reads 相反，Gathering Writes 能较好的处理动态消息。

第 3 章 Java NIO（SocketChannel）

(1) SocketChannel 就是 NIO 对于非阻塞 socket 操作的支持的组件，其在 socket 上封装了一层，主要是支持了非阻塞的读写。同时改进了传统的单向流 API，Channel 同时支持读写。

(2) socket 通道类主要分为 DatagramChannel、SocketChannel 和 ServerSocketChannel，它们在被实例化时都会创建一个对等 socket 对象。要把一个 socket 通道置于非阻塞模式，我们要依靠所有 socket 通道类的公有超级类：SelectableChannel。就绪选择（readiness selection）是一种可以用来查询通道的机制，该查询可以判断通道是否准备好执行一个目标操作，如读或写。非阻塞 I/O 和可选择性是紧密相连的，那也正是管理阻塞模式的 API 代码要在 SelectableChannel 超级类中定义的原因。

(3) 设置或重新设置一个通道的阻塞模式是很简单的，只要调用 configureBlocking()方法即可，传递参数值为 true 则设为阻塞模式，参数值为 false 值设为非阻塞模式。可以通过调用 isBlocking()方法来判断某个 socket 通道当前处于哪种模式。

AbstractSelectableChannel.java 中实现的 configureBlocking()方法如下：

```
public final SelectableChannel configureBlocking(boolean block)
    throws IOException
{
    synchronized (regLock) {
        if (!isOpen())
            throw new ClosedChannelException();
        if (blocking == block)
            return this;
        if (block && haveValidKeys())
            throw new IllegalBlockingModeException();
        implConfigureBlocking(block);
        blocking = block;
    }
    return this;
}
```

下面分别介绍这 3 个通道

3.1 ServerSocketChannel

ServerSocketChannel 是一个基于通道的 socket 监听器。它同我们所熟悉的 java.net.ServerSocket 执行相同的任务，不过它增加了通道语义，因此能够在非阻塞模式下运行。

由于 ServerSocketChannel 没有 bind() 方法，因此有必要取出对等的 socket 并使用它来绑定到一个端口以开始监听连接。我们也是使用对等 ServerSocket 的 API 来根据需要设置其他的 socket 选项。

同 java.net.ServerSocket 一样，ServerSocketChannel 也有 accept() 方法。ServerSocketChannel 的 accept() 方法会返回 SocketChannel 类型对象，SocketChannel 可以在非阻塞模式下运行。

以下代码演示了如何使用一个非阻塞的 accept() 方法：

```
public class FileChannelAccept {

    public static final String GREETING = "Hello java.nio.\r\n";
```

```
public static void main(String[] argv) throws Exception {
    int port = 1234; // default
    if (argv.length > 0) {
        port = Integer.parseInt(argv[0]);
    }
    ByteBuffer buffer = ByteBuffer.wrap(GREETING.getBytes());
    ServerSocketChannel ssc = ServerSocketChannel.open();
    ssc.socket().bind(new InetSocketAddress(port));
    ssc.configureBlocking(false);
    while (true) {
        System.out.println("Waiting for connections");
        SocketChannel sc = ssc.accept();
        if (sc == null) {
            System.out.println("null");
            Thread.sleep(2000);
        } else {
            System.out.println("Incoming connection from: " +
sc.socket().getRemoteSocketAddress());
            buffer.rewind();
            sc.write(buffer);
            sc.close();
        }
    }
}
```

```
FileChannelAccept x
null
Waiting for connections
null
Waiting for connections
Incoming connection from: /127.0.0.1:64664
Waiting for connections
Incoming connection from: /127.0.0.1:64666
Waiting for connections
Incoming connection from: /127.0.0.1:64667
Waiting for connections
```

(1) 打开 ServerSocketChannel

通过调用 `ServerSocketChannel.open()` 方法来打开 `ServerSocketChannel`。

```
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
```

(2) 关闭 ServerSocketChannel

通过调用 `ServerSocketChannel.close()` 方法来关闭 `ServerSocketChannel`。

```
serverSocketChannel.close();
```

(3) 监听新的连接

通过 `ServerSocketChannel.accept()` 方法监听新进的连接。当 `accept()` 方法返回时, 它返回一个包含新进来的连接的 `SocketChannel`。因此, `accept()` 方法会一直阻塞到有新连接到达。

通常不会仅仅只监听一个连接, 在 `while` 循环中调用 `accept()` 方法。如下面的例子:

```
while (true) {
    System.out.println("Waiting for connections");
    SocketChannel sc = ssc.accept();
```

(4) 阻塞模式

```
FileChannelAccept x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Waiting for connections
```

会在 `SocketChannel sc = ssc.accept();` 这里阻塞住进程。

(5) 非阻塞模式

`ServerSocketChannel` 可以设置成非阻塞模式。在非阻塞模式下, `accept()` 方法会立刻返回, 如果还没有新进来的连接, 返回的将是 `null`。因此, 需要检查返回的 `SocketChannel` 是否是 `null`。如:

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.socket().bind(new InetSocketAddress(port));
ssc.configureBlocking(false);
while (true) {
    System.out.println("Waiting for connections");
    SocketChannel sc = ssc.accept();
    if (sc == null) {
        System.out.println("null");
    }
}
```

3.2 SocketChannel

3.2.1、SocketChannel 介绍

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

A selectable channel for stream-oriented connecting sockets.

以上是 Java docs 中对于 `SocketChannel` 的描述: `SocketChannel` 是一种面向流连接 sockets 套接字的可选择通道。从这里可以看出:

- `SocketChannel` 是用来连接 Socket 套接字
- `SocketChannel` 主要用途用来处理网络 I/O 的通道
- `SocketChannel` 是基于 TCP 连接传输

- SocketChannel 实现了可选择通道，可以被多路复用的

3.2.2、SocketChannel 特征：

- (1) 对于已经存在的 socket 不能创建 SocketChannel
- (2) SocketChannel 中提供的 open 接口创建的 Channel 并没有进行网络级联，需要使用 connect 接口连接到指定地址
- (3) 未进行连接的 SocketChannel 执行 I/O 操作时，会抛出 NotYetConnectedException
- (4) SocketChannel 支持两种 I/O 模式：阻塞式和非阻塞式
- (5) SocketChannel 支持异步关闭。如果 SocketChannel 在一个线程上 read 阻塞，另一个线程对该 SocketChannel 调用 shutdownInput，则读阻塞的线程将返回 -1 表示没有读取任何数据；如果 SocketChannel 在一个线程上 write 阻塞，另一个线程对该 SocketChannel 调用 shutdownWrite，则写阻塞的线程将抛出

AsynchronousCloseException

- (6) SocketChannel 支持设定参数
 - SO_SNDBUF 套接字发送缓冲区大小
 - SO_RCVBUF 套接字接收缓冲区大小
 - SO_KEEPALIVE 保活连接
 - O_REUSEADDR 复用地址
 - SO_LINGER 有数据传输时延缓关闭 Channel (只有在非阻塞模式下有用)
 - TCP_NODELAY 禁用 Nagle 算法

3.2.3、SocketChannel 的使用

(1) 创建 SocketChannel

方式一：

```
SocketChannel socketChannel = SocketChannel.open(new  
InetSocketAddress("www.baidu.com", 80));
```

方式二：

```
SocketChannel socketChanne2 = SocketChannel.open();  
socketChanne2.connect(new InetSocketAddress("www.baidu.com", 80));
```


直接使用有参 open api 或者使用无参 open api，但是在无参 open 只是创建了一个 SocketChannel 对象，并没有进行实质的 tcp 连接。

(2) 连接校验

```
socketChannel.isOpen();    // 测试 SocketChannel 是否为 open 状态
socketChannel.isConnected(); // 测试 SocketChannel 是否已经被连接
socketChannel.isConnectionPending(); // 测试 SocketChannel 是否正在进行连接
socketChannel.finishConnect(); // 校验正在进行套接字连接的 SocketChannel 是否已经完成连接
```

(3) 读写模式

前面提到 SocketChannel 支持阻塞和非阻塞两种模式：

```
socketChannel.configureBlocking(false);
```

通过以上方法设置 SocketChannel 的读写模式。false 表示非阻塞，true 表示阻塞。

(4) 读写

```
SocketChannel socketChannel = SocketChannel.open(
    new InetSocketAddress("www.baidu.com", 80));
ByteBuffer byteBuffer = ByteBuffer.allocate(16);
socketChannel.read(byteBuffer);
socketChannel.close();
System.out.println("read over");
```

以上为阻塞式读，当执行到 read 出，线程将阻塞，控制台将无法打印 read over

```
SocketChannel socketChannel = SocketChannel.open(
    new InetSocketAddress("www.baidu.com", 80));
socketChannel.configureBlocking(false);
ByteBuffer byteBuffer = ByteBuffer.allocate(16);
socketChannel.read(byteBuffer);
socketChannel.close();
```

```
System.out.println("read over");
```

以上为非阻塞读，控制台将打印 read over

读写都是面向缓冲区，这个读写方式与前文中的 FileChannel 相同。

(5) 设置和获取参数

```
socketChannel.setOption(StandardSocketOptions.SO_KEEPALIVE,  
Boolean.TRUE)  
    .setOption(StandardSocketOptions.TCP_NODELAY, Boolean.TRUE);
```

通过 setOptions 方法可以设置 socket 套接字的相关参数

```
socketChannel.getOption(StandardSocketOptions.SO_KEEPALIVE);  
socketChannel.getOption(StandardSocketOptions.SO_RCVBUF);
```

可以通过 getOption 获取相关参数的值。如默认接收缓冲区大小是 8192byte。

SocketChannel 还支持多路复用，但是多路复用在后续内容中会介绍到。

3.3 DatagramChannel

正如 SocketChannel 对应 Socket，ServerSocketChannel 对应 ServerSocket，每一个 DatagramChannel 对象也有一个关联的 DatagramSocket 对象。正如 SocketChannel 模拟连接导向的流协议（如 TCP/IP），DatagramChannel 则模拟包导向的无连接协议（如 UDP/IP）。DatagramChannel 是无连接的，每个数据报（datagram）都是一个自包含的实体，拥有它自己的目的地址及不依赖其他数据报的数据负载。与面向流的 socket 不同，DatagramChannel 可以发送单独的数据报给不同的目的地址。同样，DatagramChannel 对象也可以接收来自任意地址的数据包。每个到达的数据报都含有关于它来自何处的信息（源地址）

1、打开 DatagramChannel

```
DatagramChannel server = DatagramChannel.open();  
server.socket().bind(new InetSocketAddress(10086));
```

此例子是打开 10086 端口接收 UDP 数据包

2、接收数据

通过 `receive()`接收 UDP 包

```
ByteBuffer receiveBuffer = ByteBuffer.allocate(64);
receiveBuffer.clear();
SocketAddress receiveAddr = server.receive(receiveBuffer);
```

`SocketAddress` 可以获得发包的 ip、端口等信息，用 `toString` 查看，格式如下
/127.0.0.1:57126

3、发送数据

通过 `send()`发送 UDP 包

```
DatagramChannel server = DatagramChannel.open();
ByteBuffer sendBuffer = ByteBuffer.wrap("client send".getBytes());
server.send(sendBuffer, new InetSocketAddress("127.0.0.1",10086));
```

4、连接

UDP 不存在真正意义上的连接，这里的连接是向特定服务地址用 `read` 和 `write` 接收发送数据包。

```
client.connect(new InetSocketAddress("127.0.0.1",10086));
int readSize= client.read(sendBuffer);
server.write(sendBuffer);
```

`read()`和 `write()`只有在 `connect()`后才能使用，不然会抛 `NotYetConnectedException` 异常。用 `read()`接收时，如果没有接收到包，会抛 `PortUnreachableException` 异常。

5、DatagramChannel 示例

客户端发送，服务端接收的例子

```
/**
 * 发包的 datagram
 *
 * @throws IOException
 * @throws InterruptedException
 */
@Test
public void sendDatagram() throws IOException, InterruptedException {
    DatagramChannel sendChannel= DatagramChannel.open();
    InetSocketAddress sendAddress= new InetSocketAddress("127.0.0.1",
9999);
    while (true) {
        sendChannel.send(ByteBuffer.wrap("发包".getBytes("UTF-8")),
sendAddress);
        System.out.println("发包端发包");
        Thread.sleep(1000);
    }
}

/**
 * 收包端
 *
 * @throws IOException
 */
@Test
public void receive() throws IOException {
    DatagramChannel receiveChannel= DatagramChannel.open();
    InetSocketAddress receiveAddress= new InetSocketAddress(9999);
    receiveChannel.bind(receiveAddress);
    ByteBuffer receiveBuffer= ByteBuffer.allocate(512);
    while (true) {
        receiveBuffer.clear();
```

```
        SocketAddress sendAddress= receiveChannel.receive(receiveBuffer);
        receiveBuffer.flip();
        System.out.print(sendAddress.toString() + " ");
        System.out.println(Charset.forName("UTF-8").decode(receiveBuffer));
    }
}

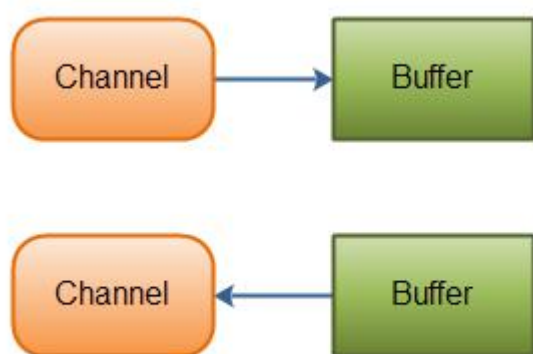
/**
 * 只接收和发送 9999 的数据包
 *
 * @throws IOException
 */
@Test
public void testConect1() throws IOException {
    DatagramChannel connChannel= DatagramChannel.open();
    connChannel.bind(new InetSocketAddress(9998));
    connChannel.connect(new InetSocketAddress("127.0.0.1",9999));
    connChannel.write(ByteBuffer.wrap("发包".getBytes("UTF-8")));
    ByteBuffer readBuffer= ByteBuffer.allocate(512);
    while (true) {
        try {
            readBuffer.clear();
            connChannel.read(readBuffer);
            readBuffer.flip();
            System.out.println(Charset.forName("UTF-8").decode(readBuffer));
        } catch (Exception e) {

        }
    }
}
```

第 4 章 Java NIO (Buffer)

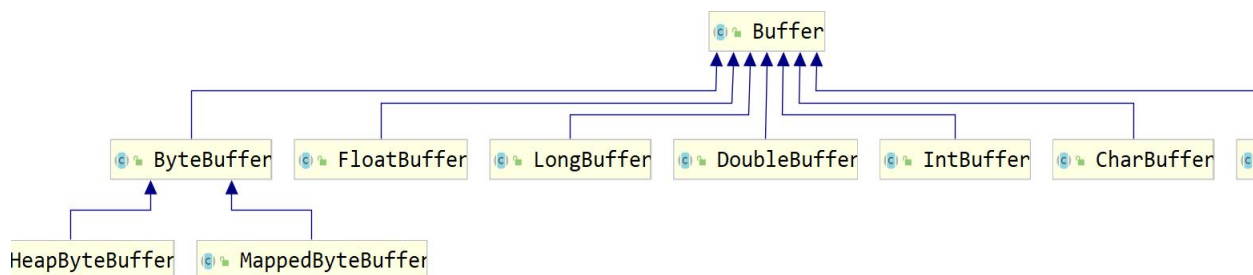
4.1 Buffer 简介

Java NIO 中的 Buffer 用于和 NIO 通道进行交互。数据是从通道读入缓冲区，从缓冲区写入到通道中的。



缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。缓冲区实际上是一个容器对象，更直接的说，其实就是一个数组，在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的；在写入数据时，它也是写入到缓冲区中的；任何时候访问 NIO 中的数据，都是将它放到缓冲区中。而在面向流 I/O 系统中，所有数据都是直接写入或者直接将数据读取到 Stream 对象中。

在 NIO 中，所有的缓冲区类型都继承于抽象类 Buffer，最常用的就是 ByteBuffer，对于 Java 中的基本类型，基本都有一个具体 Buffer 类型与之相对应，它们之间的继承关系如下图所示：



4.2 Buffer 的基本用法

1、使用 Buffer 读写数据，一般遵循以下四个步骤：

- (1) 写入数据到 Buffer
- (2) 调用 flip()方法
- (3) 从 Buffer 中读取数据
- (4) 调用 clear()方法或者 compact()方法

当向 buffer 写入数据时，buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip()方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 的所有数据。一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear()或 compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

2、使用 Buffer 的例子

```
@Test
public void testConect2() throws IOException {
    RandomAccessFile aFile = new RandomAccessFile("d:\\atguigu/01.txt",
"rw");
    FileChannel inChannel = aFile.getChannel();

    //create buffer with capacity of 48 bytes
    ByteBuffer buf = ByteBuffer.allocate(48);

    int bytesRead = inChannel.read(buf); //read into buffer.
    while (bytesRead != -1) {

        buf.flip(); //make buffer ready for read

        while(buf.hasRemaining()){
            System.out.print((char) buf.get()); // read 1 byte at a time
        }
    }
}
```



```
    buf.clear(); //make buffer ready for writing
    bytesRead = inChannel.read(buf);
}
aFile.close();
}
```

3、使用 IntBuffer 的例子

```
@Test
public void testConect3() throws IOException {
    // 分配新的 int 缓冲区, 参数为缓冲区容量
    // 新缓冲区的当前位置将为零, 其界限(限制位置)将为其容量。
    // 它将具有一个底层实现数组, 其数组偏移量将为零。
    IntBuffer buffer = IntBuffer.allocate(8);

    for (int i = 0; i < buffer.capacity(); ++i) {
        int j = 2 * (i + 1);
        // 将给定整数写入此缓冲区的当前位置, 当前位置递增
        buffer.put(j);
    }

    // 重设此缓冲区, 将限制设置为当前位置, 然后将当前位置设置为 0
    buffer.flip();

    // 查看在当前位置和限制位置之间是否有元素
    while (buffer.hasRemaining()) {
        // 读取此缓冲区当前位置的整数, 然后当前位置递增
        int j = buffer.get();
        System.out.print(j + " ");
    }
}
```

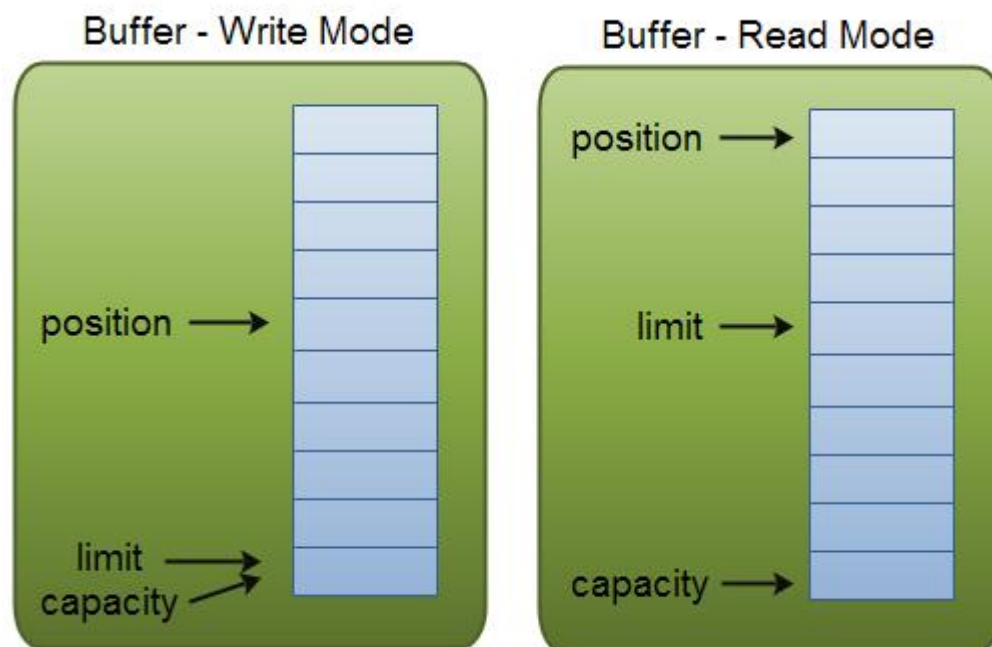
4.3 Buffer 的 capacity、position 和 limit

为了理解 Buffer 的工作原理，需要熟悉它的三个属性：

- Capacity
- Position
- limit

position 和 limit 的含义取决于 Buffer 处在读模式还是写模式。不管 Buffer 处在什么模式，capacity 的含义总是一样的。

这里有一个关于 capacity，position 和 limit 在读写模式中的说明



(1) capacity

作为一个内存块，Buffer 有一个固定的大小值，也叫 “capacity”。你只能往里写 capacity 个 byte、long，char 等类型。一旦 Buffer 满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

(2) position

1) 写数据到 Buffer 中时，position 表示写入数据的当前位置，position 的初始值为 0。当一个 byte、long 等数据写到 Buffer 后，position 会向下移动到下一个可插入数据的 Buffer 单元。position 最大可为 capacity - 1（因为 position 的初始值为 0）。

2) **读数据到 Buffer 中时**，position 表示读入数据的当前位置，如 position=2 时表示已开始读入了 3 个 byte，或从第 3 个 byte 开始读取。通过 ByteBuffer.flip() 切换到读模式时 position 会被重置为 0，当 Buffer 从 position 读入数据后，position 会下移到下一个可读入的数据 Buffer 单元。

(3) limit

1) **写数据时**，limit 表示可对 Buffer 最多写入多少个数据。写模式下，limit 等于 Buffer 的 capacity。

2) **读数据时**，limit 表示 Buffer 里有多少可读数据（not null 的数据），因此能读到之前写入的所有数据（limit 被设置成已写数据的数量，这个值在写模式下就是 position）。

4.4 Buffer 的类型

Java NIO 有以下 Buffer 类型

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

这些 Buffer 类型代表了不同的数据类型。换句话说，就是可以通过 char, short, int, long, float 或 double 类型来操作缓冲区中的字节。

4.5 Buffer 分配和写数据

1、Buffer 分配

要想获得一个 Buffer 对象首先要进行分配。每一个 Buffer 类都有一个 allocate 方法。

下面是一个分配 48 字节 capacity 的 ByteBuffer 的例子。

```
ByteBuffer buf = ByteBuffer.allocate(48);
```

这是分配一个可存储 1024 个字符的 CharBuffer:

```
CharBuffer buf = CharBuffer.allocate(1024);
```

2、向 Buffer 中写数据

写数据到 Buffer 有两种方式:

- (1) 从 Channel 写到 Buffer。
- (2) 通过 Buffer 的 put()方法写到 Buffer 里。

从 Channel 写到 Buffer 的例子

```
int bytesRead = inChannel.read(buf); //read into buffer.
```

通过 put 方法写 Buffer 的例子:

```
buf.put(127);
```

put 方法有很多版本, 允许你以不同的方式把数据写入到 Buffer 中。例如, 写到一个指定的位置, 或者把一个字节数组写入到 Buffer

3、flip()方法

flip 方法将 Buffer 从写模式切换到读模式。调用 flip()方法会将 position 设回 0，并将 limit 设置成之前 position 的值。换句话说，position 现在用于标记读的位置，limit 表示之前写进了多少个 byte、char 等（现在能读取多少个 byte、char 等）。

4.6 从 Buffer 中读取数据

从 Buffer 中读取数据有两种方式：

- (1) 从 Buffer 读取数据到 Channel。
- (2) 使用 get()方法从 Buffer 中读取数据。

从 Buffer 读取数据到 Channel 的例子：

```
//read from buffer into channel.  
int bytesWritten = inChannel.write(buf);
```

使用 get()方法从 Buffer 中读取数据的例子

```
byte aByte = buf.get();
```

get 方法有很多版本，允许你以不同的方式从 Buffer 中读取数据。例如，从指定 position 读取，或者从 Buffer 中读取数据到字节数组。

4.7 Buffer 几个方法

1、rewind()方法

Buffer.rewind()将 position 设回 0，所以你可以重读 Buffer 中的所有数据。limit 保持不变，仍然表示能从 Buffer 中读取多少个元素（byte、char 等）。

2、clear()与 compact()方法

一旦读完 Buffer 中的数据，需要让 Buffer 准备好再次被写入。可以通过 clear()或 compact()方法来完成。

如果调用的是 clear()方法，position 将被设回 0，limit 被设置成 capacity 的值。换句话说，Buffer 被清空了。Buffer 中的数据并未清除，只是这些标记告诉我们可以从哪里开始往 Buffer 里写数据。

如果 Buffer 中有一些未读的数据，调用 clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果 Buffer 中仍有未读的数据，且后续还需要这些数据，但是此时想要先写些数据，那么使用 compact()方法。

compact()方法将所有未读的数据拷贝到 Buffer 起始处。然后将 position 设到最后一个未读元素正后面。limit 属性依然像 clear()方法一样，设置成 capacity。现在 Buffer 准备好写数据了，但是不会覆盖未读的数据。

3、mark()与 reset()方法

通过调用 Buffer.mark()方法，可以标记 Buffer 中的一个特定 position。之后可以通过调用 Buffer.reset()方法恢复到这个 position。例如：

```
buffer.mark();
```

```
//call buffer.get() a couple of times, e.g. during parsing.
```

```
buffer.reset(); //set position back to mark.
```

4.8 缓冲区操作

1、缓冲区分片

在 NIO 中，除了可以分配或者包装一个缓冲区对象外，还可以根据现有的缓冲区对象来创建一个子缓冲区，即在现有缓冲区上切出一片来作为一个新的缓冲区，但现有的缓冲区与创建的子缓冲区在底层数组层面上是数据共享的，也就是说，子缓冲区相当于是现有缓冲区的一个视图窗口。调用 `slice()` 方法可以创建一个子缓冲区。

@Test

```
public void testConect3() throws IOException {
```

```
    ByteBuffer buffer = ByteBuffer.allocate(10);
```

```
    // 缓冲区中的数据 0-9
```

```
    for (int i = 0; i < buffer.capacity(); ++i) {
```

```
        buffer.put((byte) i);
```

```
    }
```

```
    // 创建子缓冲区
```

```
        buffer.position(3);
```

```
    buffer.limit(7);
```

```
    ByteBuffer slice = buffer.slice();
```

```
    // 改变子缓冲区的内容
```

```
    for (int i = 0; i < slice.capacity(); ++i) {
```

```
        byte b = slice.get(i);
```

```
        b *= 10;
```

```
        slice.put(i, b);
```



```
}

buffer.position(0);
buffer.limit(buffer.capacity());

while (buffer.remaining() > 0) {
    System.out.println(buffer.get());
}
}
```

2、只读缓冲区

只读缓冲区非常简单，可以读取它们，但是不能向它们写入数据。可以通过调用缓冲区的 `asReadOnlyBuffer()` 方法，将任何常规缓冲区转换为只读缓冲区，这个方法返回一个与原缓冲区完全相同的缓冲区，并与原缓冲区共享数据，只不过它是只读的。如果原缓冲区的内容发生了变化，只读缓冲区的内容也随之发生变化：

```
@Test
public void testConect4() throws IOException {
    ByteBuffer buffer = ByteBuffer.allocate(10);

    // 缓冲区中的数据 0-9
    for (int i = 0; i < buffer.capacity(); ++i) {
        buffer.put((byte) i);
    }

    // 创建只读缓冲区
    ByteBuffer readonly = buffer.asReadOnlyBuffer();

    // 改变原缓冲区的内容
    for (int i = 0; i < buffer.capacity(); ++i) {
        byte b = buffer.get(i);
        b *= 10;
        buffer.put(i, b);
    }
}
```

```
readonly.position(0);
readonly.limit(buffer.capacity());

// 只读缓冲区的内容也随之改变
while (readonly.remaining() > 0) {
    System.out.println(readonly.get());
}
}
```

如果尝试修改只读缓冲区的内容，则会报 `ReadOnlyBufferException` 异常。只读缓冲区对于保护数据很有用。在将缓冲区传递给某个对象的方法时，无法知道这个方法是否会修改缓冲区中的数据。创建一个只读的缓冲区可以保证该缓冲区不会被修改。只可以把常规缓冲区转换为只读缓冲区，而不能将只读的缓冲区转换为可写的缓冲区。

3、直接缓冲区

直接缓冲区是为加快 I/O 速度，使用一种特殊方式为其分配内存的缓冲区，JDK 文档中的描述为：给定一个直接字节缓冲区，Java 虚拟机将尽最大努力直接对它执行本机 I/O 操作。也就是说，它会在每一次调用底层操作系统的本机 I/O 操作之前(或之后)，尝试避免将缓冲区的内容拷贝到一个中间缓冲区中 或者从一个中间缓冲区中拷贝数据。要分配直接缓冲区，需要调用 `allocateDirect()` 方法，而不是 `allocate()` 方法，使用方式与普通缓冲区并无区别。

拷贝文件示例：

```
@Test
public void testConect5() throws IOException {
    String infile = "d:\\atguigu\\01.txt";
    FileInputStream fin = new FileInputStream(infile);
    FileChannel fcin = fin.getChannel();

    String outfile = String.format("d:\\atguigu\\02.txt");
    FileOutputStream fout = new FileOutputStream(outfile);
    FileChannel fcout = fout.getChannel();
}
```

```
// 使用 allocateDirect, 而不是 allocate
ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

while (true) {
    buffer.clear();
    int r = fcin.read(buffer);
    if (r == -1) {
        break;
    }
    buffer.flip();
    fcout.write(buffer);
}
}
```

4、内存映射文件 I/O

内存映射文件 I/O 是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的 I/O 快的多。内存映射文件 I/O 是通过使文件中的数据出现为 内存数组的内容来完成的，这其初听起来似乎不过就是将整个文件读到内存中，但是事实上并不是这样。一般来说，只有文件中实际读取或者写入的部分才会映射到内存中。

示例代码：

```
static private final int start = 0;
static private final int size = 1024;

static public void main(String args[]) throws Exception {
    RandomAccessFile raf = new RandomAccessFile("d:\\atguigu\\01.txt",
"rw");
    FileChannel fc = raf.getChannel();
    MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE,
start, size);

    mbb.put(0, (byte) 97);
    mbb.put(1023, (byte) 122);
}
```

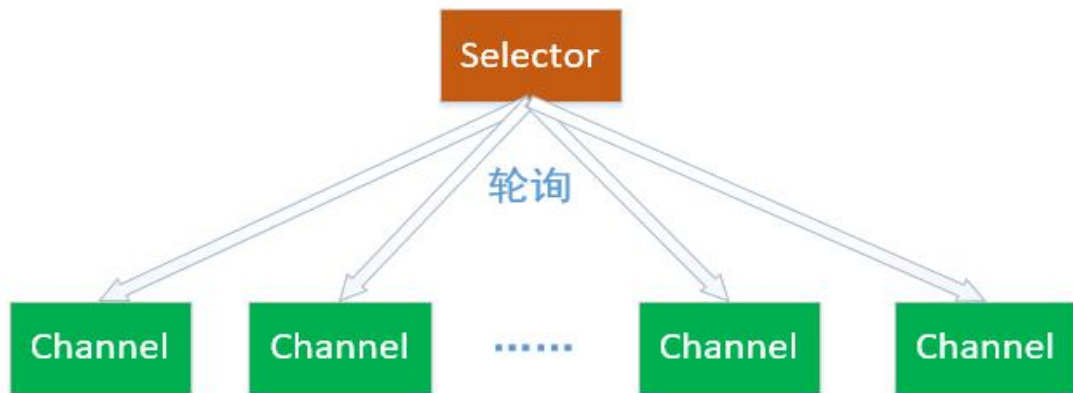
```
raf.close();
}
```

第 5 章 Java NIO (Selector)

5.1 Selector 简介

1、Selector 和 Channel 关系

Selector 一般称为选择器，也可以翻译为多路复用器。它是 Java NIO 核心组件中的一个，用于检查一个或多个 NIO Channel（通道）的状态是否处于可读、可写。如此可以实现单线程管理多个 channels,也就是可以管理多个网络链接。



使用 Selector 的好处在于：使用更少的线程来就可以来处理通道了，相比使用多个线程，避免了线程上下文切换带来的开销。

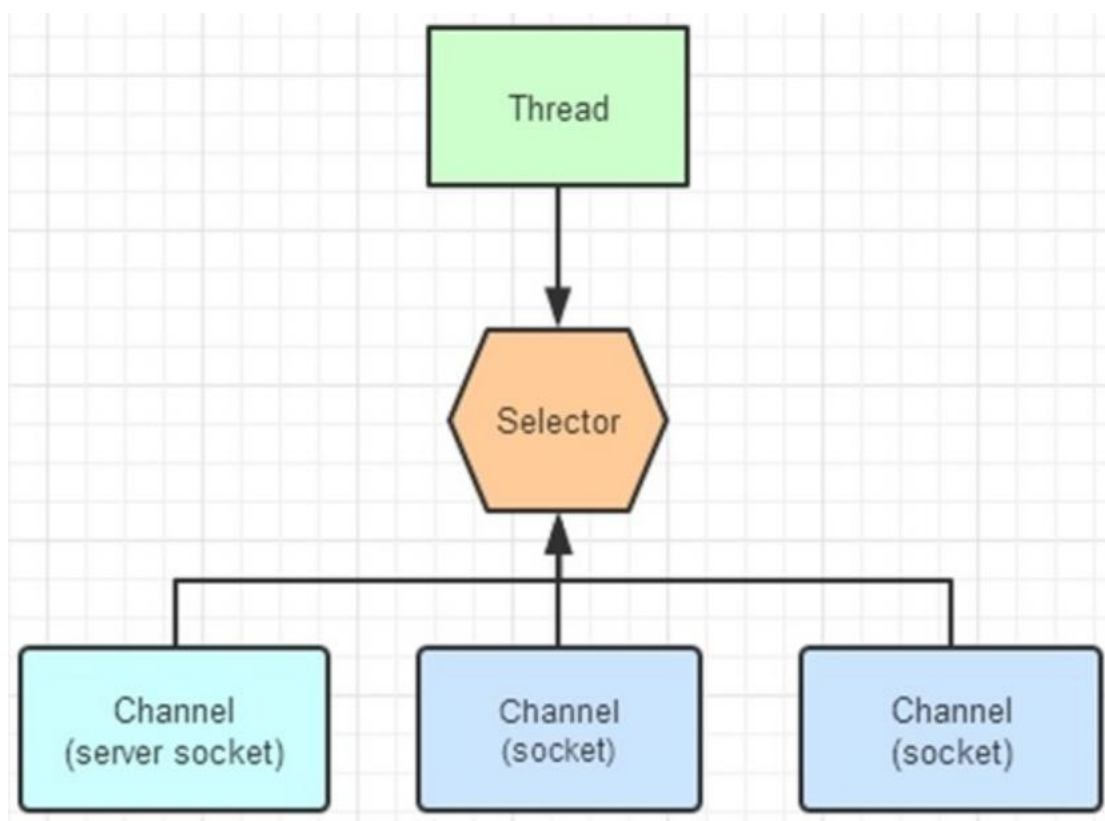
2、可选择通道(SelectableChannel)

(1) 不是所有的 Channel 都可以被 Selector 复用的。比方说，FileChannel 就不能被选择器复用。判断一个 Channel 能被 Selector 复用，有一个前提：判断他是否继

承了一个抽象类 `SelectableChannel`。如果继承了 `SelectableChannel`，则可以被复用，否则不能。

(2) `SelectableChannel` 类提供了实现通道的可选择性所需要的公共方法。它是所有支持就绪检查的通道类的父类。所有 `socket` 通道，都继承了 `SelectableChannel` 类都是可选择的，包括从管道(Pipe)对象的中获得的通道。而 `FileChannel` 类，没有继承 `SelectableChannel`，因此是不是可选通道。

(3) 一个通道可以被注册到多个选择器上，但对每个选择器而言只能被注册一次。通道和选择器之间的关系，使用注册的方式完成。`SelectableChannel` 可以被注册到 `Selector` 对象上，在注册的时候，需要指定通道的哪些操作，是 `Selector` 感兴趣的。



3、Channel 注册到 Selector

(1) 使用 `Channel.register (Selector sel, int ops)` 方法，将一个通道注册到一个选择器时。第一个参数，指定通道要注册的选择器。第二个参数指定选择器需要查询的通道操作。

(2) 可以供选择器查询的通道操作，从类型来分，包括以下四种：

- 可读 : `SelectionKey.OP_READ`
- 可写 : `SelectionKey.OP_WRITE`
- 连接 : `SelectionKey.OP_CONNECT`
- 接收 : `SelectionKey.OP_ACCEPT`

如果 Selector 对通道的多操作类型感兴趣, 可以用 “位或” 操作符来实现:

比如: `int key = SelectionKey.OP_READ | SelectionKey.OP_WRITE ;`

(3) 选择器查询的不是通道的操作, 而是通道的某个操作的一种就绪状态。什么是操作的就绪状态? 一旦通道具备完成某个操作的条件, 表示该通道的某个操作已经就绪, 就可以被 Selector 查询到, 程序可以对通道进行对应的操作。比方说, 某个 `SocketChannel` 通道可以连接到一个服务器, 则处于 “连接就绪” (`OP_CONNECT`)。再比方说, 一个 `ServerSocketChannel` 服务器通道准备好接收新进入的连接, 则处于 “接收就绪” (`OP_ACCEPT`) 状态。还比方说, 一个有数据可读的通道, 可以说是 “读就绪” (`OP_READ`)。一个等待写数据的通道可以说是 “写就绪” (`OP_WRITE`)。

4、选择键(SelectionKey)

(1) Channel 注册到后, 并且一旦通道处于某种就绪的状态, 就可以被选择器查询到。这个工作, 使用选择器 Selector 的 `select ()` 方法完成。select 方法的作用, 对感兴趣的通道操作, 进行就绪状态的查询。

(2) Selector 可以不断的查询 Channel 中发生的操作的就绪状态。并且挑选感兴趣的操作就绪状态。一旦通道有操作的就绪状态达成, 并且是 Selector 感兴趣的操作, 就会被 Selector 选中, 放入选择键集合中。

(3) 一个选择键, 首先是包含了注册在 Selector 的通道操作的类型, 比方说 `SelectionKey.OP_READ`。也包含了特定的通道与特定的选择器之间的注册关系。

开发应用程序是, 选择键是编程的关键。NIO 的编程, 就是根据对应的选择键, 进行不同的业务逻辑处理。

(4) 选择键的概念, 和事件的概念比较相似。一个选择键类似监听器模式里边的一个事件。由于 Selector 不是事件触发的模式, 而是主动去查询的模式, 所以不叫事件 Event, 而是叫 SelectionKey 选择键。

5.2 Selector 的使用方法

1. Selector 的创建

通过调用 `Selector.open()` 方法创建一个 `Selector` 对象，如下：

```
// 1、获取 Selector 选择器
Selector selector = Selector.open();
```

2. 注册 Channel 到 Selector

要实现 `Selector` 管理 `Channel`，需要将 `channel` 注册到相应的 `Selector` 上

```
// 1、获取 Selector 选择器
Selector selector = Selector.open();

// 2、获取通道
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

// 3. 设置为非阻塞
serverSocketChannel.configureBlocking(false);

// 4、绑定连接
serverSocketChannel.bind(new InetSocketAddress(9999));

// 5、将通道注册到选择器上, 并制定监听事件为: "接收" 事件
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
```

上面通过调用通道的 `register()` 方法会将它注册到一个选择器上。

首先需要注意的是：

(1) 与 Selector 一起使用时, **Channel 必须处于非阻塞模式下**, 否则将抛出异常 `IllegalBlockingModeException`。这意味着, `FileChannel` 不能与 Selector 一起使用, 因为 `FileChannel` 不能切换到非阻塞模式, 而套接字相关的所有的通道都可以。

(2) 一个通道, 并没有一定要支持所有的四种操作。比如服务器通道 `ServerSocketChannel` 支持 `Accept` 接受操作, 而 `SocketChannel` 客户端通道则不支持。可以通过通道上的 `validOps()` 方法, 来获取特定通道下所有支持的操作集合。

3. 轮询查询就绪操作

(1) 通过 Selector 的 `select ()` 方法, 可以查询出已经就绪的通道操作, 这些就绪的状态集合, 包存在一个元素是 `SelectionKey` 对象的 `Set` 集合中。

(2) 下面是 Selector 几个重载的查询 `select()` 方法:

- `select()`: 阻塞到至少有一个通道在你注册的事件上就绪了。
- `select(long timeout)`: 和 `select()` 一样, 但最长阻塞事件为 `timeout` 毫秒。
- `selectNow()`: 非阻塞, 只要有通道就绪就立刻返回。

`select()` 方法返回的 `int` 值, 表示有多少通道已经就绪, 更准确的说, 是自前一次 `select` 方法以来到这一次 `select` 方法之间的时间段上, 有多少通道变成就绪状态。

例如: 首次调用 `select()` 方法, 如果有一个通道变成就绪状态, 返回了 1, 若再次调用 `select()` 方法, 如果另一个通道就绪了, 它会再次返回 1。如果对第一个就绪的 `channel` 没有做任何操作, 现在就有两个就绪的通道, 但在每次 `select()` 方法调用之间, 只有一个通道就绪了。

一旦调用 `select()` 方法, 并且返回值不为 0 时, 在 Selector 中有一个 `selectedKeys()` 方法, 用来访问已选择键集合, 迭代集合的每一个选择键元素, 根据就绪操作的类型, 完成对应的操作:

```
Set selectedKeys = selector.selectedKeys();  
  
Iterator keyIterator = selectedKeys.iterator();
```



```
while(keyIterator.hasNext()) {  
  
    SelectionKey key = keyIterator.next();  
  
    if(key.isAcceptable()) {  
  
        // a connection was accepted by a ServerSocketChannel.  
  
    } else if (key.isConnectable()) {  
  
        // a connection was established with a remote server.  
  
    } else if (key.isReadable()) {  
  
        // a channel is ready for reading  
  
    } else if (key.isWritable()) {  
  
        // a channel is ready for writing  
  
    }  
  
    keyIterator.remove();  
  
}
```

5. 停止选择的方法

选择器执行选择的过程，系统底层会依次询问每个通道是否已经就绪，这个过程可能会造成调用线程进入阻塞状态,那么我们有以下三种方式可以唤醒在 `select ()` 方法中阻塞的线程。

`wakeup()`方法：通过调用 `Selector` 对象的 `wakeup ()` 方法让处在阻塞状态的 `select()`方法立刻返回

该方法使得选择器上的第一个还没有返回的选择操作立即返回。如果当前没有进行中的选择操作，那么下一次对 `select()` 方法的一次调用将立即返回。

`close()` 方法：通过 `close ()` 方法关闭 Selector，

该方法使得任何一个在选择操作中阻塞的线程都被唤醒（类似 `wakeup ()`），同时使得注册到该 Selector 的所有 Channel 被注销，所有的键将被取消，但是 Channel 本身并不会关闭。

5.3 NIO 编程步骤

第一步：创建 Selector 选择器

第二步：创建 `ServerSocketChannel` 通道，并绑定监听端口

第三步：设置 Channel 通道是非阻塞模式

第四步：把 Channel 注册到 Selector 选择器上，监听连接事件

第五步：调用 Selector 的 `select` 方法（循环调用），监测通道的就绪状况

第六步：调用 `selectKeys` 方法获取就绪 channel 集合

第七步：遍历就绪 channel 集合，判断就绪事件类型，实现具体的业务操作

第八步：根据业务，决定是否需要再次注册监听事件，重复执行第三步操作

5.4 示例代码

1、服务端代码

```
@Test
public void ServerDemo() {
    try {
        ServerSocketChannel ssc = ServerSocketChannel.open();
```

```
ssc.socket().bind(new InetSocketAddress("127.0.0.1", 8000));
ssc.configureBlocking(false);

Selector selector = Selector.open();
// 注册 channel, 并且指定感兴趣的事件是 Accept
ssc.register(selector, SelectionKey.OP_ACCEPT);

ByteBuffer readBuff = ByteBuffer.allocate(1024);
ByteBuffer writeBuff = ByteBuffer.allocate(128);
writeBuff.put("received".getBytes());
writeBuff.flip();

while (true) {
    int nReady = selector.select();
    Set<SelectionKey> keys = selector.selectedKeys();
    Iterator<SelectionKey> it = keys.iterator();

    while (it.hasNext()) {
        SelectionKey key = it.next();
        it.remove();

        if (key.isAcceptable()) {
            // 创建新的连接, 并且把连接注册到 selector 上, 而且,
            // 声明这个 channel 只对读操作感兴趣。
            SocketChannel socketChannel = ssc.accept();
            socketChannel.configureBlocking(false);
            socketChannel.register(selector, SelectionKey.OP_READ);
        }
        else if (key.isReadable()) {
            SocketChannel socketChannel = (SocketChannel) key.channel();
            readBuff.clear();
            socketChannel.read(readBuff);

            readBuff.flip();
            System.out.println("received : " + new String(readBuff.array()));
            key.interestOps(SelectionKey.OP_WRITE);
        }
    }
}
```

```
    }  
    else if (key.isWritable()) {  
        writeBuff.rewind();  
        SocketChannel socketChannel = (SocketChannel) key.channel();  
        socketChannel.write(writeBuff);  
        key.interestOps(SelectionKey.OP_READ);  
    }  
}  
}  
}  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}
```

2、客户端代码

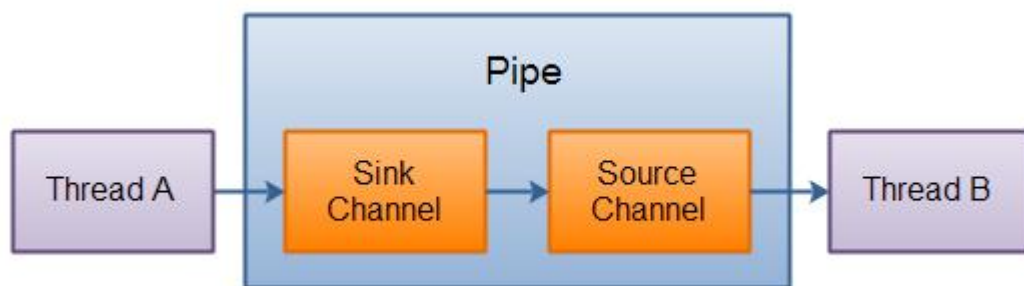
```
@Test  
public void ClientDemo() {  
    try {  
        SocketChannel socketChannel = SocketChannel.open();  
        socketChannel.connect(new InetSocketAddress("127.0.0.1", 8000));  
  
        ByteBuffer writeBuffer = ByteBuffer.allocate(32);  
        ByteBuffer readBuffer = ByteBuffer.allocate(32);  
  
        writeBuffer.put("hello".getBytes());  
        writeBuffer.flip();  
  
        while (true) {  
            writeBuffer.rewind();  
            socketChannel.write(writeBuffer);  
            readBuffer.clear();  
            socketChannel.read(readBuffer);  
        }  
    }  
}
```

```
} catch (IOException e) {  
}  
}
```

第 6 章 Java NIO (Pipe 和 FileLock)

6.1 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。



1、创建管道

通过 `Pipe.open()` 方法打开管道。

```
Pipe pipe = Pipe.open();
```

2、写入管道

要向管道写数据，需要访问 sink 通道。：

```
Pipe.SinkChannel sinkChannel = pipe.sink();
```

通过调用 SinkChannel 的 write()方法，将数据写入 SinkChannel：

```
String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

buf.put(newData.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，像这样：

```
Pipe.SourceChannel sourceChannel = pipe.source();
```

调用 source 通道的 read()方法来读取数据：

```
ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = sourceChannel.read(buf);
```

read()方法返回的 int 值会告诉我们多少字节被读进了缓冲区。

4、示例

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

```
// 1、获取通道
Pipe pipe = Pipe.open();
// 2、获取 sink 管道，用来传送数据
Pipe.SinkChannel sinkChannel = pipe.sink();
// 3、申请一定大小的缓冲区
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
byteBuffer.put("atguigu".getBytes());
byteBuffer.flip();
// 4、sink 发送数据
sinkChannel.write(byteBuffer);
// 5、创建接收 pipe 数据的 source 管道
Pipe.SourceChannel sourceChannel = pipe.source();
// 6、接收数据，并保存到缓冲区中
ByteBuffer byteBuffer2 = ByteBuffer.allocate(1024);
int length = sourceChannel.read(byteBuffer2);
System.out.println(new String(byteBuffer2.array(), 0, length));

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

6.2 FileLock

1、FileLock 简介

文件锁在 OS 中很常见，如果多个程序同时访问、修改同一个文件，很容易因为文件数据不同步而出现问题。给文件加一个锁，同一时间，只能有一个程序修改此文件，或者程序都只能读此文件，这就解决了同步问题。

文件锁是进程级别的，不是线程级别的。文件锁可以解决多个进程并发访问、修改同一个文件的问题，但不能解决多线程并发访问、修改同一文件的问题。使用文件锁时，同一进程内的多个线程，可以同时访问、修改此文件。

文件锁是当前程序所属的 JVM 实例持有的，一旦获取到文件锁（对文件加锁），要调用 `release()`，或者关闭对应的 `FileChannel` 对象，或者当前 JVM 退出，才会释放这个锁。

一旦某个进程（比如说 JVM 实例）对某个文件加锁，则在释放这个锁之前，此进程不能再对此文件加锁，就是说 JVM 实例在同一文件上的文件锁是不重叠的（进程级别不能重复在同一文件上获取锁）。

2、文件锁分类：

排它锁：又叫独占锁。对文件加排它锁后，该进程可以对此文件进行读写，该进程独占此文件，其他进程不能读写此文件，直到该进程释放文件锁。

共享锁：某个进程对文件加共享锁，其他进程也可以访问此文件，但这些进程都只能读此文件，不能写。线程是安全的。只要还有一个进程持有共享锁，此文件就只能读，不能写。

3、使用示例：

```
//创建 FileChannel 对象，文件锁只能通过 FileChannel 对象来使用
FileChannel fileChannel=new FileOutputStream("./1.txt").getChannel();

//对文件加锁
FileLock lock=fileChannel.lock();

//对此文件进行一些读写操作。
//.....

//释放锁
lock.release();
```

文件锁要通过 `FileChannel` 对象使用。

4、获取文件锁方法

有 4 种获取文件锁的方法：

`lock()` //对整个文件加锁，默认为排它锁。

`lock(long position, long size, boolean shared)` //自定义加锁方式。前 2 个参数指定要加锁的部分（可以只对此文件的部分内容加锁），第三个参数值指定是否是共享锁。

`tryLock()` //对整个文件加锁，默认为排它锁。

`tryLock(long position, long size, boolean shared)` //自定义加锁方式。

如果指定为共享锁，则其它进程可读此文件，所有进程均不能写此文件，如果某进程试图对此文件进行写操作，会抛出异常。

5、lock 与 tryLock 的区别：

`lock` 是阻塞式的，如果未获取到文件锁，会一直阻塞当前线程，直到获取文件锁

`tryLock` 和 `lock` 的作用相同，只不过 `tryLock` 是非阻塞式的，`tryLock` 是尝试获取文件锁，获取成功就返回锁对象，否则返回 `null`，不会阻塞当前线程。

6、FileLock 两个方法：

`boolean isShared()` //此文件锁是否是共享锁

`boolean isValid()` //此文件锁是否还有效

在某些 OS 上，对某个文件加锁后，不能对此文件使用通道映射。

7、完整例子

```
public class Demo1 {

    public static void main(String[] args) throws IOException {
        String input = "atguigu";
        System.out.println("输入:" + input);

        ByteBuffer buf = ByteBuffer.wrap(input.getBytes());
        String fp = "D:\\atguigu\\01.txt";
        Path pt = Paths.get(fp);
        FileChannel channel = FileChannel.open(pt,
        StandardOpenOption.WRITE, StandardOpenOption.APPEND);
        channel.position(channel.size() - 1); // position of a cursor at the end of file

        // 获得锁方法一: lock(), 阻塞方法, 当文件锁不可用时, 当前进程会被挂起
        // lock = channel.lock(); // 无参 lock() 为独占锁
        // lock = channel.lock(0L, Long.MAX_VALUE, true); // 有参 lock() 为共享锁, 有写操作会报异常
        // 获得锁方法二: trylock(), 非阻塞的方法, 当文件锁不可用时, tryLock() 会得到 null 值
        FileLock lock = channel.tryLock(0, Long.MAX_VALUE, false);
        System.out.println("共享锁 shared: " + lock.isShared());

        channel.write(buf);
        channel.close(); // Releases the Lock
        System.out.println("写操作完成.");
        // 读取数据
        readPrint(fp);
    }

    public static void readPrint(String path) throws IOException {
        FileReader filereader = new FileReader(path);
        BufferedReader bufferedreader = new BufferedReader(filereader);
        String tr = bufferedreader.readLine();
        System.out.println("读取内容: ");
    }
}
```

```
while (tr != null) {  
    System.out.println(" " + tr);  
    tr = bufferedreader.readLine();  
}  
filereader.close();  
bufferedReader.close();  
}  
}
```

第 7 章 Java NIO (其他)

7.1 Path

1、Path 简介

Java Path 接口是 Java NIO 更新的一部分，同 Java NIO 一起已经包括在 Java6 和 Java7 中。Java Path 接口是在 Java7 中添加到 Java NIO 的。Path 接口位于 `java.nio.file` 包中，所以 Path 接口的完全限定名称为 `java.nio.file.Path`。

Java Path 实例表示文件系统中的路径。一个路径可以指向一个文件或一个目录。路径可以是绝对路径，也可以是相对路径。绝对路径包含从文件系统的根目录到它指向的文件或目录的完整路径。相对路径包含相对于其他路径的文件或目录的路径。

在许多方面，`java.nio.file.Path` 接口类似于 `java.io.File` 类，但是有一些差别。不过，在许多情况下，可以使用 Path 接口来替换 File 类的使用。

2、创建 Path 实例

使用 `java.nio.file.Path` 实例必须创建一个 Path 实例。可以使用 `Paths` 类 (`java.nio.file.Paths`) 中的静态方法 `Paths.get()` 来创建路径实例。

示例代码:

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class PathDemo {

    public static void main(String[] args) {

        Path path = Paths.get("d:\\atguigu\\001.txt");

    }
}
```

上述代码，可以理解为，Paths.get()方法是 Path 实例的工厂方法。

3、创建绝对路径

(1) 创建绝对路径，通过调用 Paths.get()方法，给定绝对路径文件作为参数来完成。

示例代码:

```
Path path = Paths.get("d:\\atguigu\\001.txt");
```

上述代码中，绝对路径是 d:\atguigu\001.txt。在 Java 字符串中，\是一个转义字符，需要编写\\，告诉 Java 编译器在字符串中写入一个\字符。

(2) 如果在 Linux、MacOS 等操作字体上，上面的绝对路径可能如下:

```
Path path = Paths.get("/home/jakobjenkov/myfile.txt");
```

绝对路径现在为/home/jakobjenkov/myfile.txt.

(3) 如果在 Windows 机器上使用了从/开始的路径，那么路径将被解释为相对于当前驱动器。

4、创建相对路径

Java NIO Path 类也可以用于处理相对路径。您可以使用 Paths.get(basePath, relativePath)方法创建一个相对路径。

示例代码:

//代码 1

```
Path projects = Paths.get("d:\\atguigu", "projects");
```

//代码 2

```
Path file = Paths.get("d:\\atguigu", "projects\\002.txt");
```

代码 1 创建了一个 Java Path 的实例, 指向路径(目录):d:\atguigu\projects

代码 2 创建了一个 Path 的实例, 指向路径(文件):d:\atguigu\projects\002.txt

5、Path.normalize()

Path 接口的 normalize()方法可以使路径标准化。标准化意味着它将移除所有在路径字符串的中间的.和..代码, 并解析路径字符串所引用的路径。

Path.normalize()示例:

```
String originalPath =
```

```
    "d:\\atguigu\\projects\\..\\yygh-project";
```

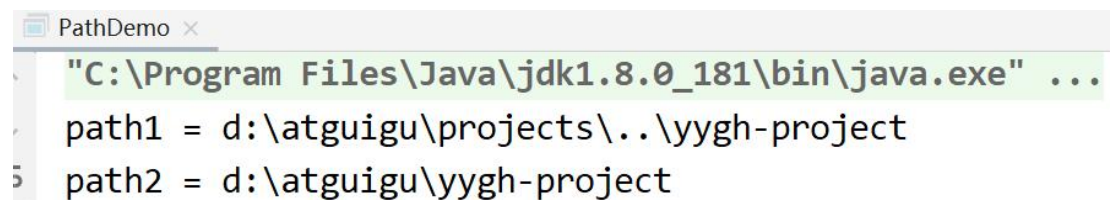
```
Path path1 = Paths.get(originalPath);
```

```
System.out.println("path1 = " + path1);
```

```
Path path2 = path1.normalize();
```

```
System.out.println("path2 = " + path2);
```

输出结果: 标准化的路径不包含 projects\\..部分



```
PathDemo x
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
path1 = d:\atguigu\projects\..\yygh-project
path2 = d:\atguigu\yygh-project
```

7.2 Files

Java NIO Files 类(`java.nio.file.Files`)提供了几种操作文件系统中的文件的方法。以下内容介绍 Java NIO Files 最常用的一些方法。`java.nio.file.Files` 类与 `java.nio.file.Path` 实例一起工作, 因此在学习 Files 类之前, 需要先了解 Path 类。

1、Files.createDirectory()

`Files.createDirectory()`方法, 用于根据 Path 实例创建一个新目录

示例:

```
Path path = Paths.get("d:\\sgg");

try {
    Path newDir = Files.createDirectory(path);
} catch (FileAlreadyExistsException e){
    // 目录已经存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
}
```

第一行创建表示要创建的目录的 Path 实例。在 try-catch 块中, 用路径作为参数调用 `Files.createDirectory()`方法。如果创建目录成功, 将返回一个 Path 实例, 该实例指向新创建的路径。

如果该目录已经存在, 则是抛出一个 `java.nio.file.FileAlreadyExistsException`。如果出现其他错误, 可能会抛出 `IOException`。例如, 如果想要的新目录的父目录不存在, 则可能会抛出 `IOException`。

2、Files.copy()

(1) `Files.copy()`方法从一个路径拷贝一个文件到另外一个目录

示例：

```
Path sourcePath = Paths.get("d:\\atguigu\\01.txt");
Path destinationPath = Paths.get("d:\\atguigu\\002.txt");

try {
    Files.copy(sourcePath, destinationPath);
} catch (FileAlreadyExistsException e) {
    // 目录已经存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
}
```

首先，该示例创建两个 Path 实例。然后，这个例子调用 Files.copy()，将两个 Path 实例作为参数传递。这可以让源路径引用的文件被复制到目标路径引用的文件中。

如果目标文件已经存在，则抛出一个 java.nio.file.FileAlreadyExistsException 异常。如果有其他错误，则会抛出一个 IOException。例如，如果将该文件复制到不存在的目录，则会抛出 IOException。

(2) 覆盖已存在的文件

Files.copy()方法的第三个参数。如果目标文件已经存在，这个参数指示 copy()方法覆盖现有的文件。

```
Files.copy(sourcePath, destinationPath,
    StandardCopyOption.REPLACE_EXISTING);
```

3、Files.move()

Files.move()用于将文件从一个路径移动到另一个路径。移动文件与重命名相同，但是移动文件既可以移动到不同的目录，也可以在相同的操作中更改它的名称。

示例：

```
Path sourcePath = Paths.get("d:\\atguigu\\01.txt");
Path destinationPath = Paths.get("d:\\atguigu\\001.txt");
```



```
try {  
    Files.move(sourcePath, destinationPath,  
        StandardCopyOption.REPLACE_EXISTING);  
} catch (IOException e) {  
    //移动文件失败  
    e.printStackTrace();  
}
```

Files.move()的第三个参数。这个参数告诉 Files.move()方法来覆盖目标路径上的任何现有文件。

4、Files.delete()

Files.delete()方法可以删除一个文件或者目录。

示例：

```
Path path = Paths.get("d:\\atguigu\\001.txt");  
  
try {  
    Files.delete(path);  
} catch (IOException e) {  
    // 删除文件失败  
    e.printStackTrace();  
}
```

创建指向要删除的文件的 Path。然后调用 Files.delete()方法。如果 Files.delete()不能删除文件(例如，文件或目录不存在)，会抛出一个 IOException。

5、Files.walkFileTree()

(1) Files.walkFileTree()方法包含递归遍历目录树功能，将 Path 实例和 FileVisitor 作为参数。Path 实例指向要遍历的目录，FileVisitor 在遍历期间被调用。

(2) FileVisitor 是一个接口，必须自己实现 FileVisitor 接口，并将实现的实例传递给 walkFileTree() 方法。在目录遍历过程中，您的 FileVisitor 实现的每个方法都将被调用。如果不需要实现所有这些方法，那么可以扩展 SimpleFileVisitor 类，它包含 FileVisitor 接口中所有方法的默认实现。

(3) FileVisitor 接口的方法中，每个都返回一个 FileVisitResult 枚举实例。FileVisitResult 枚举包含以下四个选项：

CONTINUE 继续

TERMINATE 终止

SKIP_SIBLING 跳过同级

SKIP_SUBTREE 跳过子级

(4) 查找一个名为 001.txt 的文件示例：

```
Path rootPath = Paths.get("d:\\atguigu");
String fileToFind = File.separator + "001.txt";

try {
    Files.walkFileTree(rootPath, new SimpleFileVisitor<Path>() {

        @Override
        public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws
IOException {
            String fileString = file.toAbsolutePath().toString();
            //System.out.println("pathString = " + fileString);

            if(fileString.endsWith(fileToFind)){
                System.out.println("file found at path: " + file.toAbsolutePath());
                return FileVisitResult.TERMINATE;
            }
            return FileVisitResult.CONTINUE;
        }
    });
} catch (IOException e){
    e.printStackTrace();
}
```

(5) `java.nio.file.Files` 类包含许多其他的函数，有关这些方法的更多信息，请查看 `java.nio.file.Files` 类的 JavaDoc。

7.3 AsynchronousFileChannel

在 Java 7 中，Java NIO 中添加了 `AsynchronousFileChannel`，也就是异步地将数据写入文件。

1、创建 `AsynchronousFileChannel`

通过静态方法 `open()` 创建

示例：

```
Path path = Paths.get("d:\\atguigu\\01.txt");

try {
    AsynchronousFileChannel fileChannel =
        AsynchronousFileChannel.open(path, StandardOpenOption.READ);
} catch (IOException e) {
    e.printStackTrace();
}
```

`open()` 方法的第一个参数指向与 `AsynchronousFileChannel` 相关联文件的 `Path` 实例。

第二个参数是一个或多个打开选项，它告诉 `AsynchronousFileChannel` 在文件上执行什么操作。在本例中，我们使用了 `StandardOpenOption.READ` 选项，表示该文件将被打开阅读。

2、通过 Future 读取数据

可以通过两种方式从 `AsynchronousFileChannel` 读取数据。第一种方式是调用返回 `Future` 的 `read()` 方法

示例：

```
Path path = Paths.get("d:\\atguigu\\001.txt");
AsynchronousFileChannel fileChannel = null;
try {
    fileChannel = AsynchronousFileChannel.open(path,
        StandardOpenOption.READ);
} catch (IOException e) {
    e.printStackTrace();
}

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

Future<Integer> operation = fileChannel.read(buffer, position);

while(!operation.isDone());

buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

上述代码：

- (1) 创建了一个 `AsynchronousFileChannel`,
- (2) 创建一个 `ByteBuffer`, 它被传递给 `read()` 方法作为参数, 以及一个 0 的位置。
- (3) 在调用 `read()` 之后, 循环, 直到返回的 `isDone()` 方法返回 `true`。
- (4) 读取操作完成后, 数据读取到 `ByteBuffer` 中, 然后打印到 `System.out` 中。

3、通过 CompletionHandler 读取数据

第二种方法是调用 `read()` 方法，该方法将一个 `CompletionHandler` 作为参数

示例：

```
Path path = Paths.get("d:\\atguigu\\001.txt");
AsynchronousFileChannel fileChannel = null;
try {
    fileChannel = AsynchronousFileChannel.open(path,
        StandardOpenOption.READ);
} catch (IOException e) {
    e.printStackTrace();
}

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

fileChannel.read(buffer, position, buffer, new CompletionHandler<Integer,
ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("result = " + result);

        attachment.flip();
        byte[] data = new byte[attachment.limit()];
        attachment.get(data);
        System.out.println(new String(data));
        attachment.clear();
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
    }
}
```

```
});
```

(1) 读取操作完成，将调用 CompletionHandler 的 completed()方法。

(2) 对于 completed()方法的参数传递一个整数，它告诉我们读取了多少字节，以及传递给 read()方法的“附件”。 “附件” 是 read()方法的第三个参数。在本代码中，它是 ByteBuffer，数据也被读取。

(3) 如果读取操作失败，则将调用 CompletionHandler 的 failed()方法。

4、通过 Future 写数据

和读取一样，可以通过两种方式将数据写入一个 AsynchronousFileChannel

示例：

```
Path path = Paths.get("d:\\atguigu\\001.txt");
AsynchronousFileChannel fileChannel = null;
try {
    fileChannel = AsynchronousFileChannel.open(path,
        StandardOpenOption.WRITE);
} catch (IOException e) {
    e.printStackTrace();
}

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("atguigu data".getBytes());
buffer.flip();

Future<Integer> operation = fileChannel.write(buffer, position);
buffer.clear();

while(!operation.isDone());

System.out.println("Write over");
```

首先, `AsynchronousFileChannel` 以写模式打开。然后创建一个 `ByteBuffer`, 并将一些数据写入其中。然后, `ByteBuffer` 中的数据被写入到文件中。最后, 示例检查返回的 `Future`, 以查看写操作完成时的情况。

注意, 文件必须已经存在。如果该文件不存在, 那么 `write()` 方法将抛出一个 `java.nio.file.NoSuchFileException`。

5、通过 `CompletionHandler` 写数据

示例:

```
Path path = Paths.get("d:\\atguigu\\001.txt");
if(!Files.exists(path)){
    try {
        Files.createFile(path);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
AsynchronousFileChannel fileChannel = null;
try {
    fileChannel = AsynchronousFileChannel.open(path,
        StandardOpenOption.WRITE);
} catch (IOException e) {
    e.printStackTrace();
}

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("atguigu data".getBytes());
buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer,
    ByteBuffer>() {
```



```
@Override
public void completed(Integer result, ByteBuffer attachment) {
    System.out.println("bytes written: " + result);
}

@Override
public void failed(Throwable exc, ByteBuffer attachment) {
    System.out.println("Write failed");
    exc.printStackTrace();
}
});
```

当写操作完成时，将会调用 CompletionHandler 的 completed()方法。如果写失败，则会调用 failed()方法。

7.4 字符集（Charset）

java 中使用 Charset 来表示字符集编码对象

Charset 常用静态方法

```
public static Charset forName(String charsetName)//通过编码类型获得 Charset 对象

public static SortedMap<String,Charset> availableCharsets()//获得系统支持的所有编码方式

public static Charset defaultCharset()//获得虚拟机默认的编码方式

public static boolean isSupported(String charsetName)//判断是否支持该编码类型
```

Charset 常用普通方法

```
public final String name()//获得 Charset 对象的编码类型(String)

public abstract CharsetEncoder newEncoder()//获得编码器对象
```

```
public abstract CharsetDecoder newDecoder();//获得解码器对象
```

代码示例：

```
@Test
public void charSetEncoderAndDecoder() throws
CharacterCodingException {
    Charset charset=Charset.forName("UTF-8");
    //1.获取编码器
    CharsetEncoder charsetEncoder=charset.newEncoder();
    //2.获取解码器
    CharsetDecoder charsetDecoder=charset.newDecoder();

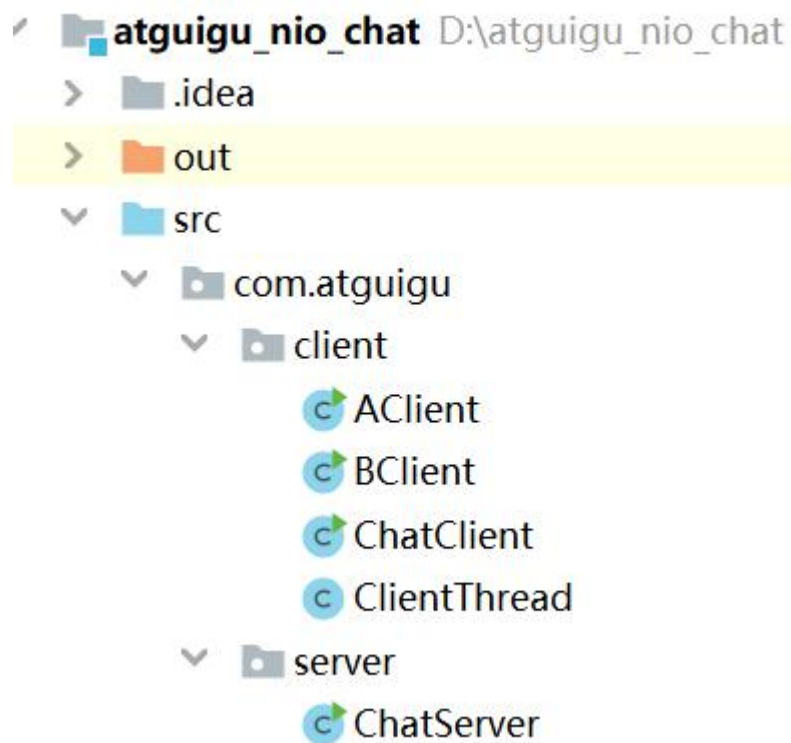
    //3.获取需要解码编码的数据
    CharBuffer charBuffer=CharBuffer.allocate(1024);
    charBuffer.put("字符集编码解码");
    charBuffer.flip();

    //4.编码
    ByteBuffer byteBuffer=charsetEncoder.encode(charBuffer);
    System.out.println("编码后: ");
    for (int i=0;i<byteBuffer.limit();i++) {
        System.out.println(byteBuffer.get());
    }
    //5.解码
    byteBuffer.flip();
    CharBuffer charBuffer1=charsetDecoder.decode(byteBuffer);
    System.out.println("解码后: ");
    System.out.println(charBuffer1.toString());
    System.out.println("指定其他格式解码:");
    Charset charset1=Charset.forName("GBK");
    byteBuffer.flip();
    CharBuffer charBuffer2 =charset1.decode(byteBuffer);
    System.out.println(charBuffer2.toString());
    //6.获取 Charset 所支持的字符编码
    Map<String ,Charset> map= Charset.availableCharsets();
```

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

第 8 章 Java NIO 综合案例

使用 Java NIO 完成一个多人聊天室功能



8.1 服务端代码

```
//服务端  
public class ChatServer {  
  
    //服务端启动的方法  
    public void startServer() throws IOException {
```

```
//1 创建 Selector 选择器
Selector selector = Selector.open();

//2 创建 ServerSocketChannel 通道
ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();

//3 为 channel 通道绑定监听端口
serverSocketChannel.bind(new InetSocketAddress(8000));
//设置非阻塞模式
serverSocketChannel.configureBlocking(false);

//4 把 channel 通道注册到 selector 选择器上
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
System.out.println("服务器已经启动成功了");

//5 循环, 等待有新链接接入
//while(true)
for(;;) {
    //获取 channel 数量
    int readChannels = selector.select();

    if(readChannels == 0) {
        continue;
    }

    //获取可用的 channel
    Set<SelectionKey> selectionKeys = selector.selectedKeys();
    //遍历集合
    Iterator<SelectionKey> iterator = selectionKeys.iterator();
    while (iterator.hasNext()) {
        SelectionKey selectionKey = iterator.next();

        //移除 set 集合当前 selectionKey
        iterator.remove();
    }
}
```

```
//6 根据就绪状态, 调用对应方法实现具体业务操作
//6.1 如果 accept 状态
if(selectionKey.isAcceptable()) {
    acceptOperator(serverSocketChannel,selector);
}
//6.2 如果可读状态
if(selectionKey.isReadable()) {
    readOperator(selector,selectionKey);
}
}
}

//处理可读状态操作
private void readOperator(Selector selector, SelectionKey selectionKey)
throws IOException {
    //1 从 SelectionKey 获取到已经就绪的通道
    SocketChannel socketChannel =
(SocketChannel)selectionKey.channel();

    //2 创建 buffer
    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);

    //3 循环读取客户端消息
    int readLength = socketChannel.read(byteBuffer);
    String message = "";
    if(readLength > 0) {
        //切换读模式
        byteBuffer.flip();

        //读取内容
        message += Charset.forName("UTF-8").decode(byteBuffer);
    }

    //4 将 channel 再次注册到选择器上, 监听可读状态
    socketChannel.register(selector,SelectionKey.OP_READ);
}
```

```
//5 把客户端发送消息, 广播到其他客户端
if(message.length()>0) {
    //广播给其他客户端
    System.out.println(message);
    castOtherClient(message,selector,socketChannel);
}
}

//广播到其他客户端
private void castOtherClient(String message, Selector selector,
SocketChannel socketChannel) throws IOException {
    //1 获取所有已经接入 channel
    Set<SelectionKey> selectionKeySet = selector.keys();

    //2 循环想所有 channel 广播消息
    for(SelectionKey selectionKey : selectionKeySet) {
        //获取每个 channel
        Channel tarChannel = selectionKey.channel();
        //不需要给自己发送
        if(tarChannel instanceof SocketChannel && tarChannel !=
socketChannel) {
            ((SocketChannel)tarChannel).write(Charset.forName("UTF-
8").encode(message));
        }
    }
}

//处理接入状态操作
private void acceptOperator(ServerSocketChannel serverSocketChannel,
Selector selector) throws IOException {
    //1 接入状态, 创建 socketChannel
    SocketChannel socketChannel = serverSocketChannel.accept();

    //2 把 socketChannel 设置非阻塞模式
    socketChannel.configureBlocking(false);
```

```
//3 把 channel 注册到 selector 选择器上, 监听可读状态
socketChannel.register(selector, SelectionKey.OP_READ);

//4 客户端回复信息
socketChannel.write(Charset.forName("UTF-8")
    .encode("欢迎进入聊天室, 请注意隐私安全"));
}

//启动主方法
public static void main(String[] args) {
    try {
        new ChatServer().startServer();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

8.2 客户端代码

ChatClient 类

```
//客户端
public class ChatClient {

    //启动客户端方法
    public void startClient(String name) throws IOException {
        //连接服务端
        SocketChannel socketChannel =
            SocketChannel.open(new InetSocketAddress("127.0.0.1", 8000));

        //接收服务端响应数据
        Selector selector = Selector.open();
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_READ);
    }
}
```

```
//创建线程
new Thread(new ClientThread(selector)).start();

//向服务器端发送消息
Scanner scanner = new Scanner(System.in);
while(scanner.hasNextLine()) {
    String msg = scanner.nextLine();
    if(msg.length() > 0) {
        socketChannel.write(Charset.forName("UTF-8").encode(name + " : " + msg));
    }
}
}
```

ClientThread 类

```
public class ClientThread implements Runnable {

    private Selector selector;
    public ClientThread(Selector selector) {
        this.selector = selector;
    }

    @Override
    public void run() {
        try {
            for(;;) {
                //获取 channel 数量
                int readChannels = selector.select();
                if(readChannels == 0) {
                    continue;
                }
                //获取可用的 channel
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                //遍历集合
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



```
        Iterator<SelectionKey> iterator = selectionKeys.iterator();
        while (iterator.hasNext()) {
            SelectionKey selectionKey = iterator.next();
            //移除 set 集合当前 selectionKey
            iterator.remove();
            //如果可读状态
            if(selectionKey.isReadable()) {
                readOperator(selector,selectionKey);
            }
        }
    }
} catch (Exception e) {
}
}

//处理可读状态操作
private void readOperator(Selector selector, SelectionKey selectionKey)
throws IOException {
    //1 从 SelectionKey 获取到已经就绪的通道
    SocketChannel socketChannel =
(SocketChannel)selectionKey.channel();
    //2 创建 buffer
    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
    //3 循环读取客户端消息
    int readLength = socketChannel.read(byteBuffer);
    String message = "";
    if(readLength > 0) {
        //切换读模式
        byteBuffer.flip();
        //读取内容
        message += Charset.forName("UTF-8").decode(byteBuffer);
    }
    //4 将 channel 再次注册到选择器上, 监听可读状态
    socketChannel.register(selector,SelectionKey.OP_READ);
    //5 把客户端发送消息, 广播到其他客户端
    if(message.length()>0) {
```

```
//广播给其他客户端  
System.out.println(message);  
}  
}  
}
```

