一、Java NIO概述

Java NIO 由以下几个核心部分组成：

- Channels
- Buffers
- Selectors

1. Channel和Buffer

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中 。

2. Selector

Selector允许单线程处理多个 Channel。

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，事件的例子有如新连接进来，数据接收等。

二、Channel

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。

1. 基本的Channel示例

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();     //打开一个通道，类似打开一个流
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);        //从通道写入缓冲区
while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip();     //反转Buffer，详见下一节
    while(buf.hasRemaining()){
        System.out.print((char) buf.get());     //从Buffer读取数据1 byte数据
    }
    buf.clear();
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

三、Buffer

Java NIO中的Buffer用于和NIO通道进行交互。

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。

1. Buffer的基本用法

使用Buffer读写数据一般遵循以下四个步骤：

1. 写入数据到Buffer

   Buffer会记录下写了多少数据

2. 调用`flip()`方法

   把Buffer从写模式切换到读模式

3. 从Buffer中读取数据

4. 调用`clear()`方法或者`compact()`方法

   读完所有数据化，需要清空缓冲区，让它可以再次被写入。

   - `clear()`：清空整个缓冲区
   - `compact()`：清除已经读过的数据，未读数据都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. Buffer的capacity,position和limit

- **capacity**

  只能往Buffer中写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。

- **position**

  - *写*：position表示当前的位置。初始的position值为0.当一个byte、long等数据写到Buffer后， position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1。
  - *读*：当将Buffer从写模式切换到读模式，position会被重置为0. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

- **limit**

  - 写：limit等于Buffer的capacity。
  - 读：limit表示你最多能读到多少数据 。因此，当切换Buffer到读模式时，limit会被设置成写模式下的position值。换句话说，你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）

3. Buffer的类型

Java NIO 有以下Buffer类型

- ByteBuffer
- MappedByteBuffer
- CharBuffer

- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

可以通过char，short，int，long，float 或 double类型来操作缓冲区中的字节。

4. Buffer的分配

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。

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

5. 向Buffer中写数据

写数据到Buffer有两种方式：

- 从Channel写到Buffer。

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

- 通过Buffer的put()方法写到Buffer里。

  ```
  buf.put(127);       //put方法有很多重载版本,详情参考JavaDoc。
  ```

6. Buffer反转

`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。

7. 从Buffer中读取数据

从Buffer中读取数据有两种方式：

- 从Buffer读取数据到Channel。

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

- 使用get()方法从Buffer中读取数据。

  ```
  byte aByte = buf.get(); //get方法有很多重载版本
  ```

8. 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）。

9. clear()与compact()方法

- `clear()`：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。

- `compact()`：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。

10. 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。

```
buffer.mark();

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

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

11. 比较

可以使用`equals()`和`compareTo()`方法两个Buffer。

`equals()`

当满足下列条件时，表示两个Buffer相等：

1. 有相同的类型（byte、char、int等）。
2. Buffer中剩余的byte、char等的个数相等。
3. Buffer中所有剩余的byte、char等都相同。

如你所见，equals只是比较Buffer的一部分，不是每一个在它里面的元素都比较。实际上，它***只比较Buffer中的剩余元素***。

`compareTo()`

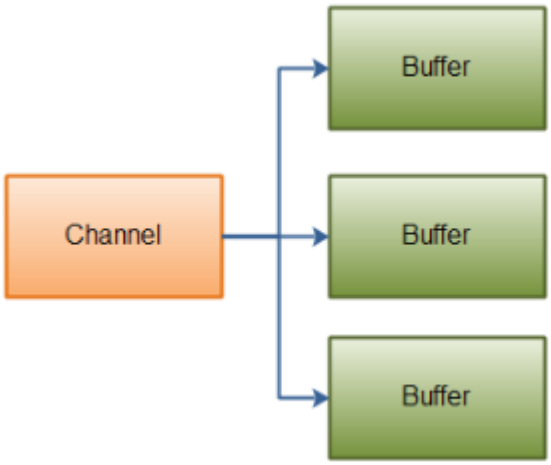
compareTo()方法比较两个Buffer的剩余元素(byte、char等)， 如果满足下列条件，则认为一个Buffer“小于”另一个Buffer：

1. 第一个不相等的元素小于另一个Buffer中对应的元素 。
2. 所有元素都相等，但第一个Buffer比另一个先耗尽(第一个Buffer的元素个数比另一个少)。

四、Scatter/Gather

- scatter（分散）：Channel将从Channel中读取的数据“分散（scatter）”到多个Buffer中。
- gather（聚集）：Channel 将多个Buffer中的数据“聚集（gather）”后发送到Channel。

1. Scattering Reads

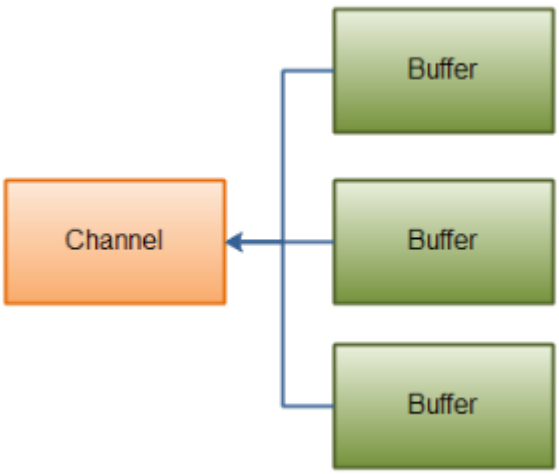



```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { header, body };
channel.read(bufferArray);
```

`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，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(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. Gathering Writes



```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);
//write data into buffers
ByteBuffer[] bufferArray = { header, body };
channel.write(bufferArray);
```

`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***。

五、通道之间的数据传输

在Java NIO中，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()

FileChannel的`transferFrom()`方法可以将数据从源通道传输到FileChannel中

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count = fromChannel.size();

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

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。

2. transferTo()

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

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel      fromChannel = fromFile.getChannel();

RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel      toChannel = toFile.getChannel();

long position = 0;
long count = fromChannel.size();

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

六、Selector

Selector（选择器）是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件做好准备的组件。

1. 为什么使用Selector?

仅用单个线程来处理多个Channels的好处是，只需要更少的线程来处理通道。

2. Selector的创建

```
Selector selector = Selector.open();
```

3.向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, Selectionkey.OP_READ);
```

与Selector一起使用时，**Channel必须处于非阻塞模式下** 。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。而套接字通道都可以。

`register()`的第二个参数表示在通过Selector监听Channel时对什么事件感兴趣。可以监听四种事件：

- Connect

  某个channel成功连接到另一个服务器称为“连接就绪”。

- Accept

  一个server socket channel准备好接收新进入的连接称为“接收就绪” 。

- Read

  一个有数据可读的通道可以说是“读就绪”。

- Write

  等待写数据的通道可以说是“写就绪”。

如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

在下面还会继续提到interest集合。

4. SelectionKey

当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些你感兴趣的属性：

- interest集合
- ready集合
- Channel
- Selector
- 附加的对象（可选）

interest集合

interest集合是你所选择的感兴趣的事件集合。可以通过SelectionKey读写interest集合。

```
int interestSet = selectionKey.interestOps();
```

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

Channel + Selector

```
Channel  channel  = selectionKey.channel();
Selector selector = selectionKey.selector();
```

附加的对象

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加 与通道一起使用的Buffer，或是包含聚集数据的某个对象。

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
key.attach(new Object());

//也可以在用register()方法向Selector注册Channel的时候附加对象
key = channel.register(selector, SelectionKey.OP_READ, new Object());
```

5. 通过Selector选择通道

一旦Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道 。

下面是select()方法：

- `int select()`

  阻塞到至少有一个通道在你注册的事件上就绪了。
- `int select(long timeout)`

和`select()`一样，除了最长会阻塞timeout毫秒(参数)。

- `int selectNow()`

  不会阻塞，不管什么通道就绪都立刻返回，如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。

返回的int值表示多少通道已经就绪。

6. selectedKeys()

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的`selectedKeys()`方法，访问“已选择键集（selected key set）”中的就绪通道。如下所示：

```
Set selectedKeys = selector.selectedKeys();
```

当向Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。可以通过SelectionKey的selectedKeySet()方法访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。如下：

```
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();  //Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。否则下次该通道变成就绪时，Selector会再次将其放入已选择键集中。
}
```

`key.channel()`方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

7. wakeUp()

某个线程调用`select()`方法后阻塞了，只要让其它线程在第一个线程调用select()方法的那个对象上调用`wakeup()`方法，阻塞在`select()`方法上的线程会立马返回。

如果有其它线程调用了`wakeup()`方法，但当前没有线程阻塞在`select()`方法上，下个调用select()方法的线程会立即“醒来（wake up）”。

8. close()

用完selector后调用其`close()`方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭。

9. 完成示例

```
Selector selector = Selector.open();
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_READ);
while(true) {
    int readyChannels = selector.select();
    if(readyChannels == 0) continue;
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> 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();
    }
}
```

七、FileChannel

Java NIO中的FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。

FileChannel无法设置为非阻塞模式，它总是运行在阻塞模式下。

1. 打开FileChannel

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。

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

2. 从FileChannel读取数据

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

3. 向FileChannel写数据

```
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()) {
    channel.write(buf);
}
```

4. 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。

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

八、SocketChannel

可以通过以下2种方式创建SocketChannel：

1. 打开一个SocketChannel并连接到互联网上的某台服务器。
2. 一个新连接到达ServerSocketChannel时，会创建一个SocketChannel。

1. 打开 SocketChannel

```
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));
```

2. 非阻塞模式

可以设置 SocketChannel 为非阻塞模式（non-blocking mode）.设置之后，就可以在异步模式下调用connect(), read() 和write()了。

3. connect()

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));

while(! socketChannel.finishConnect() ){
    //wait, or do something else...
}
```

九、ServerSocketChannel

1. 打开 ServerSocketChannel

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

2. 监听新进来的连接

```
while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();
    //do something with socketChannel...
}
```

当 accept()方法返回的时候,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。

3. 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。

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

serverSocketChannel.socket().bind(new InetSocketAddress(9999));
serverSocketChannel.configureBlocking(false);

while(true){
    SocketChannel socketChannel = serverSocketChannel.accept();

    if(socketChannel != null){
        //do something with socketChannel...
    }
}
```

十、Java NIO DatagramChannel

Java NIO中的DatagramChannel是一个能收发UDP包的通道。因为UDP是无连接的网络协议，所以不能像其它通道那样读取和写入。它发送和接收的是数据包。

1. 打开 DatagramChannel

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
```

2. 接收数据

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
channel.receive(buf);
```

receive()方法会将接收到的数据包内容复制到指定的Buffer。 如果Buffer容不下收到的数据，多出的数据将被丢弃。

3. 发送数据

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

ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();

int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));
```

十一、Java NIO与IO

1. Java NIO和IO的主要区别

面向流与面向缓冲

- IO：面向流
- NIO：面向缓冲区

阻塞与非阻塞IO

- IO：阻塞
- NIO：非阻塞

选择器

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。