Java NIO 系列教程

转载 Jack-Chan 2017-03-12 13:35:56 377 收藏

分类专栏： Java基础 Java基础入门 文章标签： java nio io 输入输出流

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，你学习和理解Java NIO。

Java NIO提供了与标准IO不同的IO工作方式：

- Channels and Buffers（通道和缓冲区）：标准的IO基于字节流和字符流进行操作（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或
- Asynchronous IO（异步IO）：Java NIO可以让你异步的使用IO，例如：当线程从还是可以进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓
- Selectors（选择器）：Java NIO引入了选择器的概念，选择器用于监听多个通道达）。因此，单个的线程可以监听多个数据通道。

下面就来详细介绍Java NIO的相关知识。

1. Java NIO 概述

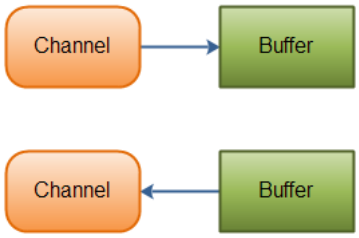
Java NIO 由以下几个核心部分组成：

- Channels
- Buffers
- Selectors

虽然Java NIO 中除此之外还有很多类和组件，但在我看来，Channel，Buffer 和 Sele件，如Pipe和FileLock，只不过是与三个核心组件共同使用的工具类。因此，在概述中组件会在单独的章节中讲到。

1.1 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Buffer 写到Channel中。这里有个图示：



Channel和Buffer有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

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。

与这些类一起的有一些有趣的接口，但为简单起见，我尽量在概述中不提到它们。本教会进行解释。

以下是Java NIO里关键的Buffer实现：

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

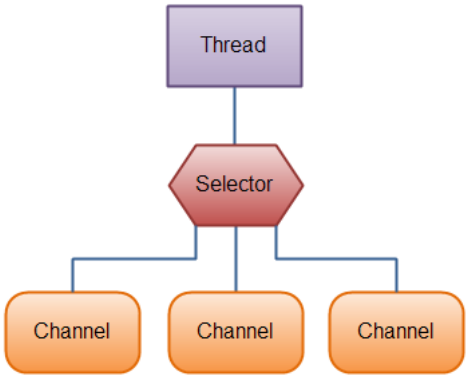
这些Buffer覆盖了你能通过IO发送的基本数据类型：byte，short，int，long，float，d

Java NIO 还有个 Mappedyteuffer，用于表示内存映射文件， 我也不打算在概述中说明

1.2 Selector

Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每
Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：



要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会
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

2. Java NIO vs IO

当学习了Java NIO和IO的API后，一个问题马上涌入脑海：

> 我应该何时使用IO，何时使用NIO呢？在本文中，我会尽量清晰地解析Java NIO和
> 以及它们如何影响您的代码设计。

2.1 Java NIO和IO的主要区别

下表总结了Java NIO和IO之间的主要差别，我会更详细地描述表中每部分的差异。

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

2.2 面向流与面向缓冲

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java
个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移
动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有

的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，这
有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚

2.3 阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被
数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一
据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。
数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线
不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空
作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

2.4 选择器（Selectors）

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
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
的线程很容易来管理多个通道。

2.5 NIO和IO如何影响应用程序的设计

无论您选择IO或NIO工具箱，可能会影响您应用程序设计的以下几个方面：

- 对NIO或IO类的API调用
- 数据处理
- 用来处理数据的线程数

2.5.1 API调用

当然，使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是
取，而是数据必须先读入缓冲区再处理。

2.5.2 数据处理

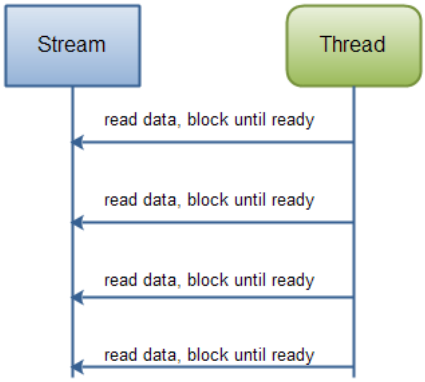
使用纯粹的NIO设计相较IO设计，数据处理也受到影响。

在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行

该文本行的流可以这样处理：

```
1  InputStream input = … ; // get the InputStream from the client socket
2  BufferedReader reader = new BufferedReader(new InputStreamReader(input)
3
4  String nameLine   = reader.readLine();
5  String ageLine    = reader.readLine();
6  String emailLine  = reader.readLine();
7  String phoneLine  = reader.readLine();
```

请注意处理状态由程序执行多久决定。换句话说，一旦reader.readLine()方法返回，你
readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个rea
行包含年龄等。 正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步
程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。下图也说明了这条



而一个NIO的实现会有所不同，下面是一个简单的例子：

```
1  ByteBuffer buffer = ByteBuffer.allocate(48);
2
3  int bytesRead = inChannel.read(buffer);
```

注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需
所知道的是，该缓冲区包含一些字节，这使得处理有点困难。

假设第一次 read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你
待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

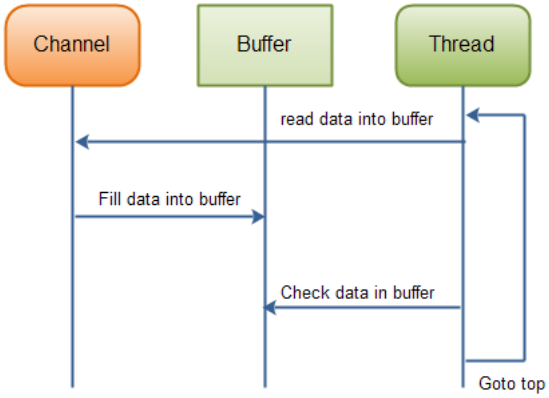
所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现
其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这
计方案杂乱不堪。例如：

```
1  ByteBuffer buffer = ByteBuffer.allocate(48);
2  int bytesRead = inChannel.read(buffer);
3  while(! bufferFull(bytesRead) ) {
4  bytesRead = inChannel.read(buffer);
5  }
```

bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是
备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull()方法被调用之前状态相同。如
可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

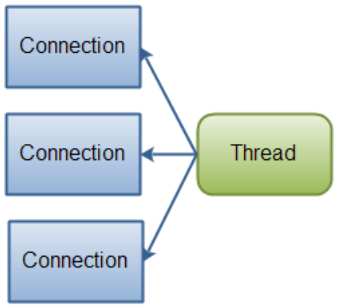
如果缓冲区已满，它可以被处理。如果它不满，并且在你的实际案例中有意义，你或
多情况下并非如此。下图展示了“缓冲区数据循环就绪”：



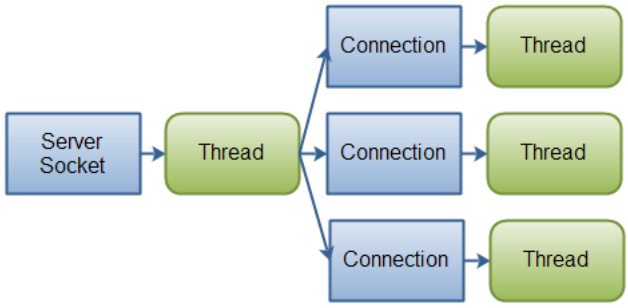
2.6 总结

NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的
阻塞流中读取数据更复杂。

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊
能是一个优势。同样，如果你需要维持许多打开的连接到其他计算机上，如P2P网络中
所有出站连接，可能是一个优势。一个线程多个连接的设计方案如下图所示：



如果你有少量的连接使用非常高的带宽，一次发送大量的数据，也许典型的IO服务器实
个典型的IO服务器设计：

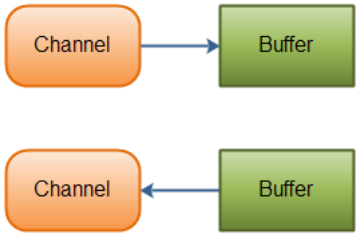


3. 通道（Channel）

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。

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：



3.1 Channel的实现

这些是Java NIO中最重要的通道的实现：

- FileChannel：从文件中读写数据
- DatagramChannel：能通过UDP读写网络中的数据
- SocketChannel：能通过TCP读写网络中的数据
- ServerSocketChannel：可以监听新进来的TCP连接，像Web服务器那样。对每一SocketChannel

3.2 基本的 Channel 示例

下面是一个使用FileChannel读取数据到Buffer中的示例：

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

ByteBuffer buf = ByteBuffer.allocate(48);

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

System.out.println("Read " + bytesRead);
buf.flip();

while(buf.hasRemaining()){
System.out.print((char) buf.get());
}

buf.clear();
bytesRead = inChannel.read(buf);
}
aFile.close();
```

注意 buf.flip() 的调用，首先读取数据到Buffer，然后反转Buffer,接着再从Buffer中读取更多细节。

4. 缓冲区（Buffer）

Java NIO中的Buffer用于和NIO通道进行交互。如你所知，数据是从通道读入缓冲区，

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。这块内存被包装成方法，用来方便的访问该块内存。

4.1 Buffer的基本用法

使用Buffer读写数据一般遵循以下四个步骤：

- 写入数据到Buffer
- 调用flip()方法
- 从Buffer中读取数据
- 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。任何处，新写入的数据将放到缓冲区未读数据的后面。

下面是一个使用Buffer的例子：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.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.
```

```
8  while (bytesRead != -1) {
9
10   buf.flip();  //make buffer ready for read
11
12   while(buf.hasRemaining()){
13       System.out.print((char) buf.get()); // read 1 byte at a time
14   }
15
16   buf.clear(); //make buffer ready for writing
17   bytesRead = inChannel.read(buf);
18 }
19 aFile.close();
```

4.2 Buffer的capacity、position、limit

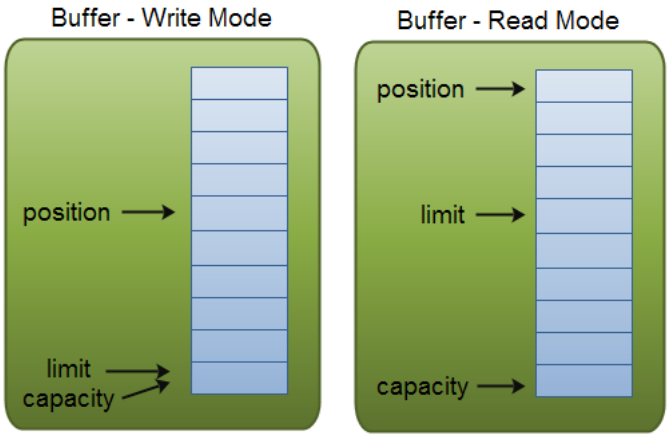
缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成
方法，用来方便的访问该块内存。

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，

这里有一个关于capacity，position和limit在读写模式中的说明，详细的解释在插图后面



4.2.1 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
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。

4.2.2 position

当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte
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。

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position
position处读取数据时，position向前移动到下一个可读的位置。

4.2.3 limit

在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据。 写模式下，limit等于B

当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读
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
position）

4.3 Buffer的类型

Java NIO 有以下Buffer类型：

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

如你所见，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，sh
型来操作缓冲区中的字节。

MappedByteBuffer 有些特别，在涉及它的专门章节中再讲。

4.4 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。下面
ByteBuffer的例子。

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

这是分配一个可存储1024个字符的CharBuffer：

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

4.5 向Buffer中写数据

写数据到Buffer有两种方式：

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

从Channel写到Buffer的例子

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

通过put方法写Buffer的例子：

```
1  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
入到Buffer。 更多Buffer实现的细节参考JavaDoc。

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

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
等。

4.5 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。
- 使用get()方法从Buffer中读取数据。

从Buffer读取数据到Channel的例子：

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

使用get()方法从Buffer中读取数据的例子

```
1  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
字节数组。更多Buffer实现的细节参考JavaDoc。

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，
个元素（byte、char等）。

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compac

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
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
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，但是此时想要先先写些数据，

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
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

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
position。例如：

```
1  buffer.mark();
2
3  //call buffer.get() a couple of times, e.g. during parsing.
4
5  buffer.reset();  //set position back to mark.
```

equals()与compareTo()方法

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：

- 有相同的类型（byte、char、int等）。
- Buffer中剩余的byte、char等的个数相等。
- 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
素。

compareTo()方法

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
Buffer：

- 第一个不相等的元素小于另一个Buffer中对应的元素。
- 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

（译注：剩余元素是从 position到limit之间的元素）

5. 分散Scatter和聚集Gather

Java NIO开始支持scatter/gather，scatter/gather用于描述从Channel（译者注：Char
取或者写入到Channel的操作。

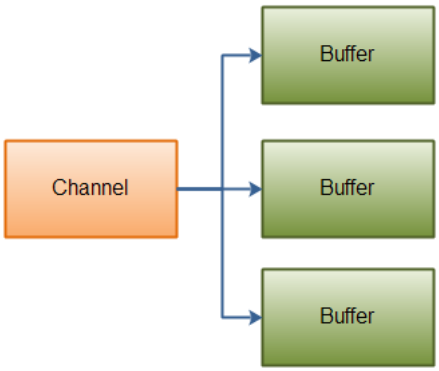
分散（scatter）从Channel中读取是指在读操作时将读取的数据写入多个buffer中。因
的数据“分散（scatter）”到多个Buffer中。

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，
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。

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：



代码示例如下：

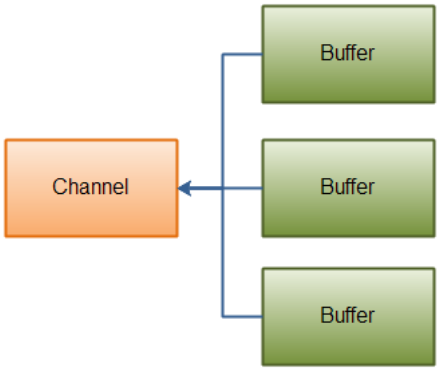
```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

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()
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

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
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）
作。

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：



代码示例如下：

```
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
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5
据将被写入到channel中。因此与Scattering Reads相反，Gathering Writes能较好的处

6. 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ch
作通道）传输到另外一个channel。

6.1 transferFrom()

FileChannel的transferFrom()方法可以将数据从源通道传输到FileChannel中（译者注:
将字节从给定的可读取字节通道传输到此通道的文件中）。下面是一个简单的例子：

```
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);
```

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
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SoketChannel的实现中，SocketChannel只会传输此刻准备好的数据
SocketChannel可能不会将请求的所有数据(count个字节)全部传输到FileChannel中。

6.2 transferTo()

transferTo()方法将数据从FileChannel传输到其他的channel中。下面是一个简单的例子

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

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

```
6
7  long position = 0;
8  long count = fromChannel.size();
9
10 fromChannel.transferTo(position, count, toChannel);
```

是不是发现这个例子和前面那个例子特别相似？除了调用方法的FileChannel对象不一

上面所说的关于SocketChannel的问题在transferTo()方法中同样存在。SocketChanne
填满。

7. 选择器（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
这样，一个单独的线程可以管理多个channel，从而管理多个网络连接。

7.1 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。事实上
道。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系
使用的线程越少越好。

但是，需要记住，现代的操作系统和CPU在多任务方面表现的越来越好，所以多线程
越小了。实际上，如果一个CPU有多个内核，不使用多任务可能是在浪费CPU能力。
应该放在另一篇不同的文章中。在这里，只要知道使用Selector能够处理多个通道就足

下面是单线程使用一个Selector处理3个channel的示例图：

7.2 Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

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

7.3 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。通过Selectab
如下：

```
1  channel.configureBlocking(false);
2  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FileChanne
FileChannel不能切换到非阻塞模式。而套接字通道都可以。

注意register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听
以监听四种不同类型的事件：

- Connect
- Accept
- Read
- Write

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服
socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以
可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

- SelectionKey.OP_CONNECT
- SelectionKey.OP_ACCEPT
- SelectionKey.OP_READ
- SelectionKey.OP_WRITE

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。

7.4 SelectionKey

在上一小节中，当向Selector注册Channel时，register()方法会返回一个SelectionKey对
趣的属性：

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

下面我会描述这些属性。

interest集合

就像向Selector注册通道一节中所描述的，interest集合是你所选择的感兴趣的事件集合
interest集合，像这样：

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

boolean isInterestedInAccept  = (interestSet & SelectionKey.OP_ACCEPT)
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE;
```

可以看到，用“位与”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的

ready集合

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先
在下一小节进行解释。可以这样访问ready集合：

```
int readySet = selectionKey.readyOps();
```

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
它们都会返回一个布尔类型：

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

Channel + Selector

从SelectionKey访问Channel和Selector很简单。如下：

```
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
起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

```
selectionKey.attach(theObject);
Object attachedObj = selectionKey.attachment();
```

还可以在用register()方法向Selector注册Channel的时候附加对象。如：

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, the
```

7.5 通过Selector选择通道

一旦向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
接受、读或写）已经准备就绪的那些通道。换句话说，如果你对“读就绪”的通道感兴趣
就绪的那些通道。

下面是select()方法：

```
int select()
int select(long timeout)
int selectNow()
```

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，除了最长会阻塞timeout毫秒(参数)。

selectNow()不会阻塞，不管什么通道就绪都立刻返回（译者注：此方法执行非阻塞的
作后，没有通道变成可选择的，则此方法直接返回零。）。

select()方法返回的int值表示有多少通道已经就绪。亦即，自上次调用select()方法后有
用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调用select()方法，
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
一个通道就绪了。

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
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();
```

当像Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。
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();
}
```

这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除
通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocket

7.6 wakeUp()

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select(
一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在selec

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
（wake up）”。

7.7 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S
并不会关闭。

7.7 完整的示例

这里有一个完整的示例，打开一个Selector，注册一个通道注册到这个Selector上(通道
这个Selector的四种事件（接受，连接，读，写）是否就绪。

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

8. 文件通道

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。

8.1 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，
OutputStream或RandomAccessFile来获取一个FileChannel实例。下面是通过Randor
例：

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

8.2 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。rea
字节被读到了Buffer中。如果返回-1，表示到了文件末尾。

8.3 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：

```
String newData = "New String to write to file..." + System.currentTimeM

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

buf.flip();

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

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Fi
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。

8.4 关闭FileChannel

用完FileChannel后必须将其关闭。如：

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

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posi
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-

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
洞”，磁盘上物理文件中写入的数据间有空隙。

8.5 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();
```

8.6 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

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

这个例子截取文件的前1024个字节。

8.7 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
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

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

下面的例子同时将文件数据和元数据强制写到磁盘上：

```
channel.force(true);
```

9. Socket 通道

Java NIO中的SocketChannel是一个连接到TCP网络套接字的通道。可以通过以下2种

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

9.1 打开 SocketChannel

下面是SocketChannel的打开方式：

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

9.2 关闭 SocketChannel

当用完SocketChannel之后调用SocketChannel.close()关闭SocketChannel：

```
1  socketChannel.close();
```

9.3 从 SocketChannel 读取数据

要从SocketChannel中读取数据，调用一个read()的方法之一。以下是例子：

```
1  ByteBuffer buf = ByteBuffer.allocate(48);
2  int bytesRead = socketChannel.read(buf);
```

首先，分配一个Buffer。从SocketChannel读取到的数据将会放到这个Buffer中。

然后，调用SocketChannel.read()。该方法将数据从SocketChannel 读到Buffer中。re
字节进Buffer里。如果返回的是-1，表示已经读到了流的末尾（连接关闭了）。

9.4 写入 SocketChannel

写数据到SocketChannel用的是SocketChannel.write()方法，该方法以一个Buffer作为

```
1  String newData = "New String to write to file..." + System.currentTimeM
2
3  ByteBuffer buf = ByteBuffer.allocate(48);
4  buf.clear();
5  buf.put(newData.getBytes());
6
7  buf.flip();
8
9  while(buf.hasRemaining()) {
10     channel.write(buf);
11 }
```

注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证
所以，我们重复调用write()直到Buffer没有要写的字节为止。

9.5 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
和write()了。

connect()

如果SocketChannel在非阻塞模式下，此时调用connect()，该方法可能在连接建立之
立，可以调用finishConnect()的方法。像这样：

```
1  socketChannel.configureBlocking(false);
2  socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));
3
4  while(! socketChannel.finishConnect() ){
```

```
    //wait, or do something else...
}
```

write()

非阻塞模式下，write()方法在尚未写出任何内容时可能就返回了。所以需要在循环中调
这里就不赘述了。

read()

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的i
字节。

9.6 非阻塞模式与选择器

非阻塞模式与选择器搭配会工作的更好，通过将一或多个SocketChannel注册到Selec
准备好了读取，写入等。Selector与SocketChannel的搭配使用会在后面详讲。

10. ServerSocket 通道

Java NIO中的 ServerSocketChannel 是一个可以监听新进来的TCP连接的通道，就像
ServerSocketChannel类在 java.nio.channels包中。

这里有个例子：

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

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

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

    //do something with socketChannel...
}
```

10.1 打开 ServerSocketChannel

通过调用 ServerSocketChannel.open() 方法来打开ServerSocketChannel.如：

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

10.2 关闭 ServerSocketChannel

通过调用ServerSocketChannel.close() 方法来关闭ServerSocketChannel. 如：

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

10.3 监听新进来的连接

通过 ServerSocketChannel.accept() 方法监听新进来的连接。当 accept()方法返回的
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){
    SocketChannel socketChannel =
            serverSocketChannel.accept();

    //do something with socketChannel...
}
```

当然，也可以在while循环中使用除了true以外的其它退出准则。

10.4 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
接，返回的将是null。 因此，需要检查返回的SocketChannel是否是null。如：

```
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...
    }
}
```

11. Datagram 通道

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
读取和写入。它发送和接收的是数据包。

11.1 打开 DatagramChannel

下面是 DatagramChannel 的打开方式：

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

这个例子打开的 DatagramChannel可以在UDP端口9999上接收数据包。

11.2 接收数据

通过receive()方法从DatagramChannel接收数据，如：

```
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

11.3 发送数据

通过send()方法从DatagramChannel发送数据，如:

```
String newData = "New String to write to file..." + System.currentTimeM

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

int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 8
```

这个例子发送一串字符到"jenkov.com"服务器的UDP端口80。 因为服务端并没有监控
也不会通知你发出的数据包是否已收到，因为UDP在数据传送方面没有任何保证。

11.4 连接到特定的地址

可以将DatagramChannel"连接"到网络中的特定地址的。由于UDP是无连接的，连接到
创建一个真正的连接。而是锁住DatagramChannel ，让其只能从特定地址收发数据。

这里有个例子:

```
channel.connect(new InetSocketAddress("jenkov.com", 80));
```

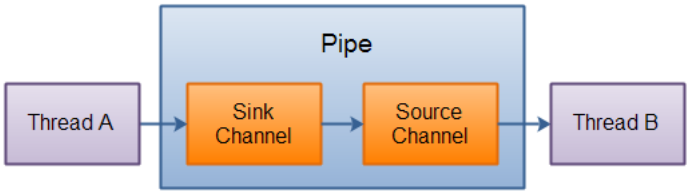
当连接后，也可以使用read()和write()方法，就像在用传统的通道一样。只是在数据传
例子：

```
int bytesRead = channel.read(buf);
int bytesWritten = channel.write(but);
```

12. 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
source通道读取。

这里是Pipe原理的图示：



12.1 创建管道

通过Pipe.open()方法打开管道。例如：

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

12.2 向管道写数据

要向管道写数据，需要访问sink通道。像这样：

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

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

```
String newData = "New String to write to file..." + System.currentTimeM
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());

buf.flip();

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

12.3 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，像这样：

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

int bytesRead = inChannel.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

13. 总结

本文中说了最重要的3个概念

- Channel 通道
- Buffer 缓冲区
- Selector 选择器

其中Channel对应以前的流，Buffer不是什么新东西，Selector是因为nio可以使用异步

以前的流总是堵塞的，一个线程只要对它进行操作，其它操作就会被堵塞，也就相当于
候，不管水到了没有，你就都只能耗在接水（流）上。

nio的Channel的加入，相当于增加了水龙头（有阀门），虽然一个时刻也只能接一个水
量不大的时候，各个水管里流出来的水，都可以得到妥善接纳，这个关键之处就是增加
Selector，他负责协调，也就是看哪根水管有水了的话，在当前水管的水接到一定程度
当前水龙头，试着打开另一个水龙头（看看有没有水）。

当其他人需要用水的时候，不是直接去接水，而是事前提了一个水桶给接水工，这个水
虽然也可能要等，但不会在现场等，而是回家等，可以做其它事去，水接满了，接水工

这其实也是非常接近当前社会分工细化的现实，也是统分利用现有资源达到并发效果的
动就来个并行处理，虽然那样是最简单的，但也是最浪费资源的方式。

原文链接：http://www.iteye.com/magazines/132-Java-NIO

相关推荐