JavaNIO指南

极客学院出版

前言

Java NIO(New IO)是一个可以替代标准 Java IO API 的 IO API（从 Java 1.4 开始），Java NIO 提供了与标准 IO 不同的 IO 工作方式。本教程讲解了 Java NIO 的三个核心组件，并介绍了跟核心组件相关的内容，旨在帮助读者 NIO 的原理。

适用人群

本教程是 Java 中高级教程，帮助那些想要学习 Java 并发的开发者。

学习前提

学习本教程前，你需要了解 Java 这门开发语言。

鸣谢：http://ifeve.com/java-nio-all/

目录

前言 1

第 1 章　概述 3

第 2 章　通道 6

第 3 章　缓冲区 9

第 4 章　Scatter/Gather 17

第 5 章　通道之间的数据传输 20

第 6 章　选择器 22

第 7 章　FileChannel 30

第 8 章　SocketChannel 34

第 9 章　ServerSocketChannel 38

第 10 章　DatagramChannel 41

第 11 章　Pipe 44

第 12 章　Java NIO 与 IO 47

. 49

. 49

. 49

. 49

. 49

. 49

. 49

1

概述

Java NIO 由以下几个核心部分组成：

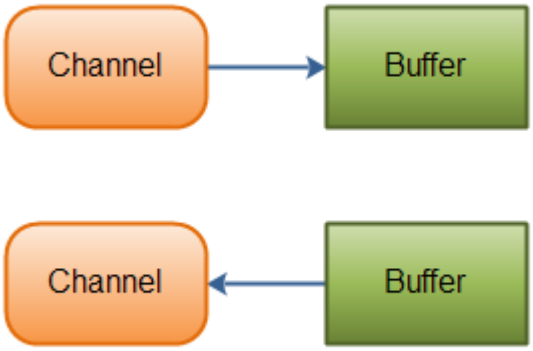
- Channels
- Buffers
- Selectors

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 Selector 构成了核心的 API。其它组件，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，在概述中我将集中在这三个组件上。其它组件会在单独的章节中讲到。

#

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。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。本教程其它章节与它们相关的地方我会进行解释。

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，double 和 char。

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

#

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。

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。

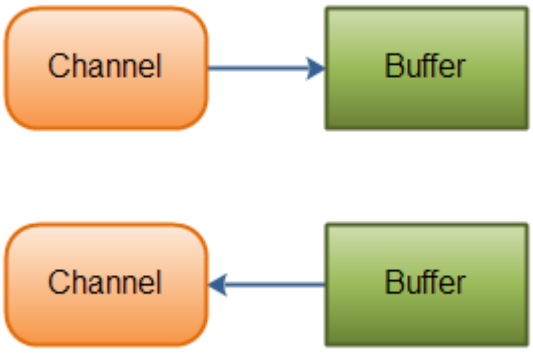
2

通道

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：



#

Channel 的实现

这些是 Java NIO 中最重要的通道的实现：

- FileChannel
- DatagramChannel
- SocketChannel
- ServerSocketChannel

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。对每一个新进来的连接都会创建一个 SocketChannel。

#

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(&amp;quot;Read &amp;quot; + 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 中读取数据。下一节会深入讲解 Buffer 的更多细节。

3

缓冲区

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，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。

下面是 NIO Buffer 相关的话题列表：

1. Buffer 的基本用法
2. Buffer 的 capacity,position 和 limit
3. Buffer 的类型
4. Buffer 的分配
5. 向 Buffer 中写数据
6. flip() 方法
7. 从 Buffer 中读取数据
8. clear() 与 compact() 方法
9. mark() 与 reset() 方法
10. equals() 与 compareTo() 方法

#

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
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。

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.
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();
```

#

Buffer 的 capacity,position 和 limit

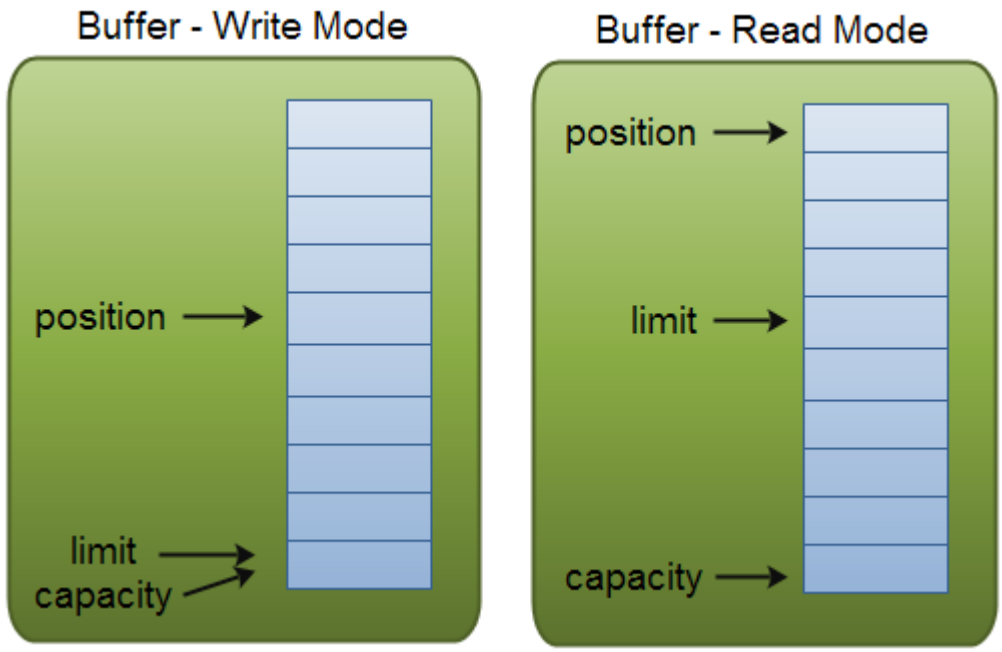
缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。

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，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，详细的解释在插图后面。



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。

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、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。

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 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

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 Buffer 的 capacity。

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）

#

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，short，int，long，float 或 double 类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。

#

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);
```

#

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

```
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。 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。

#

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：

```
//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。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，仍然表示能从 Buffer 中读取多少个元素（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 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。

#

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.
```

#

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：

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() 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)。

（译注：剩余元素是从 position 到 limit 之间的元素）

4

Scatter/Gather

Java NIO 开始支持 scatter/gather，scatter/gather 用于描述从 Channel（译者注：Channel 在中文经常翻译为通道）中读取或者写入到 Channel 的操作。

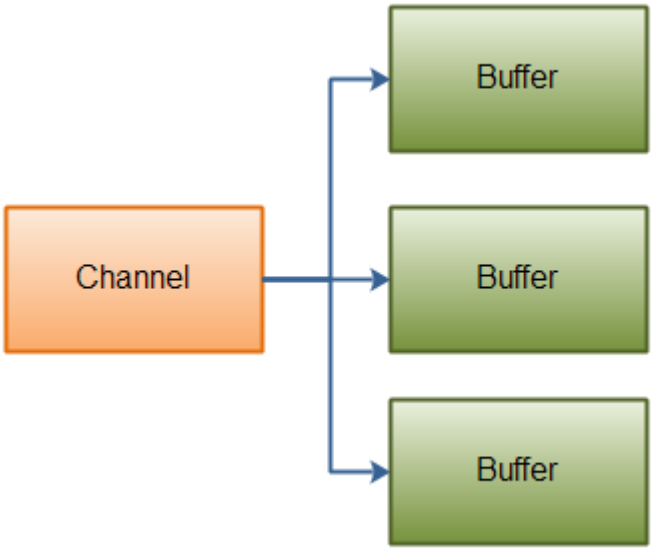
分散（scatter）从 Channel 中读取是指在读操作时将读取的数据写入多个 buffer 中。因此，Channel 将从 Channel 中读取的数据 “分散（scatter）” 到多个 Buffer 中。

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。

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：

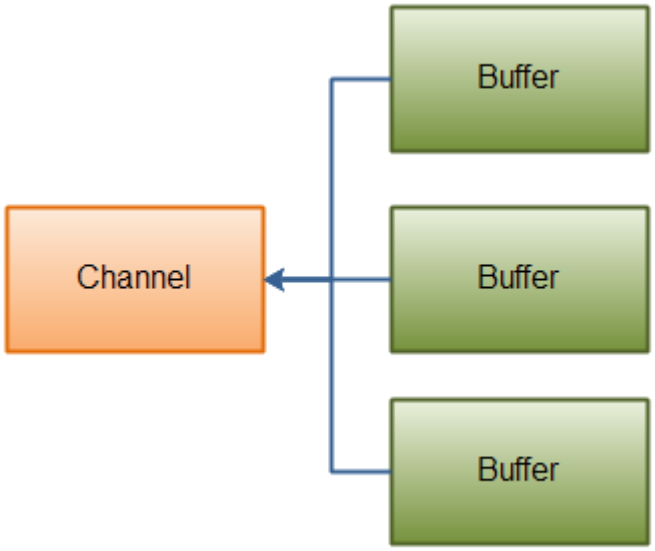
```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { header, body };
channel.read(bufferArray);
```

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() 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。

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。

5

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（译者注：channel 中文常译作通道）传输到另外一个 channel。

transferFrom()

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 SoketChannel 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。

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();
long position = 0;
long count = fromChannel.size();
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 对象不一样外，其他的都一样。

上面所说的关于 SocketChannel 的问题在 transferTo() 方法中同样存在。SocketChannel 会一直传输数据直到目标 buffer 被填满。

6

选择器

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。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。

下面是本文所涉及到的主题列表：

1. 为什么使用 Selector?
2. Selector 的创建
3. 向 Selector 注册通道
4. SelectionKey
5. 通过 Selector 选择通道
6. wakeUp()
7. close()
8. 完整的示例

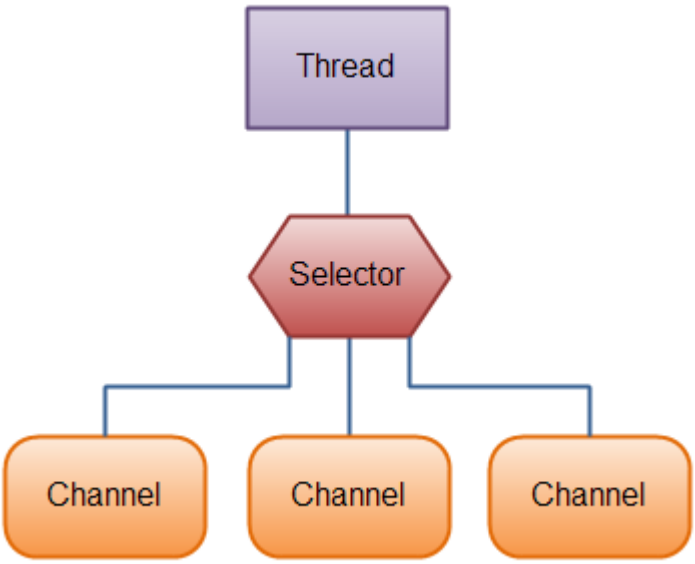
#

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，可以只用一个线程处理所有的通道。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统的一些资源（如内存）。因此，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，所以多线程的开销随着时间的推移，变得越来越小了。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。不管怎么说，关于那种设计的讨论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够了。

下面是单线程使用一个 Selector 处理 3 个 channel 的示例图：



#

Selector 的创建

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

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

#

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 SelectableChannel.register() 方法来实现，如下：

```
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。

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 Channel 时对什么事件感兴趣。可以监听四种不同类型的事件：

1. Connect
2. Accept
3. Read
4. 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务器称为 “连接就绪”。一个 server 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说是 “读就绪”。等待写数据的通道可以说是 “写就绪”。

这四种事件用 SelectionKey 的四个常量来表示：

1. SelectionKey.OP_CONNECT
2. SelectionKey.OP_ACCEPT
3. SelectionKey.OP_READ
4. 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

#

SelectionKey

在上一小节中，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）

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。可以通过 SelectionKey 读写 interest 集合，像这样：

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept  = (interestSet & SelectionKey.OP_ACCEPT) == 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事件是否在 interest 集合中。

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，你会首先访问这个 ready set。Selection 将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

从 SelectionKey 访问 Channel 和 Selector 很简单。如下：

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

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, theObject);
```

#

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。换句话说，如果你对 “读就绪” 的通道感兴趣，select() 方法会返回读事件已经就绪的那些通道。

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选择操作。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。）。

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() 方法后有多少通道变成就绪状态。如果调用 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，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。

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();
```

当像 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();
}
```

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 SelectionKey 实例。必须在处理完通道时自己移除。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 ServerSocketChannel 或 SocketChannel 等。

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() 方法返回。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 Selector.wakeup() 方法即可。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）”。

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。

#

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 (通道的初始化过程略去), 然后持续监控这个 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();
  }
}
```

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。

#

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

#

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。

#

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.currentTimeMillis();
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 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。

#

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 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。

#

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();
```

#

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。

#

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。

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);
```

8

SocketChannel

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。

#

打开 SocketChannel

下面是 SocketChannel 的打开方式：

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

#

关闭 SocketChannel

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

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

#

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：

```
ByteBuffer buf = ByteBuffer.allocate(48);
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。read() 方法返回的 int 值表示读了多少字节进 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）。

#

写入 SocketChannel

写数据到 SocketChannel 用的是 SocketChannel.write() 方法，该方法以一个 Buffer 作为参数。示例如下：

```
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);
}
```

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

#

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。

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前就返回了。为了确定连接是否建立，可以调用 finishConnect() 的方法。像这样：

```
socketChannel.configureBlocking(false);
socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));
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用 write()。前面已经有例子了，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 int 返回值，它会告诉你读取了多少字节。

#

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 Selector，可以询问选择器哪个通道已经准备好了读取，写入等。Selector 与 SocketChannel 的搭配使用会在后面详讲。

9

ServerSocketChannel

Java NIO 中的 ServerSocketChannel 是一个可以监听新进来的 TCP 连接的通道, 就像标准 IO 中的 ServerSocket 一样。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...
}
```

#

打开 ServerSocketChannel

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

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

#

关闭 ServerSocketChannel

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

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

#

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。

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。

#

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 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...
    }
}
```

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。

#

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。

#

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 容不下收到的数据，多出的数据将被丢弃。

#

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.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com", 80));
```

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这个端口，所以什么也不会发生。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。

#

连接到特定的地址

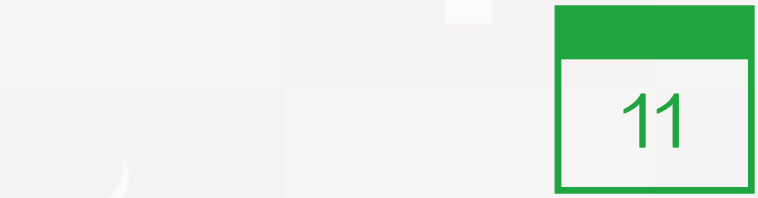
可以将 DatagramChannel“连接” 到网络中的特定地址的。由于 UDP 是无连接的，连接到特定地址并不会像 TCP 通道那样创建一个真正的连接。而是锁住 DatagramChannel ，让其只能从特定地址收发数据。

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。只是在数据传送方面没有任何保证。这里有几个例子:

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

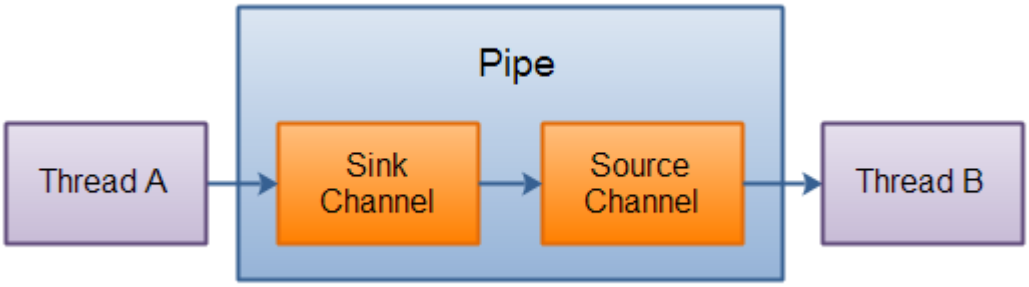


11

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。

这里是 Pipe 原理的图示：



#

创建管道

通过Pipe.open()方法打开管道。例如：

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

#

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.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining()) {
    sinkChannel.write(buf);
}
```

#

从管道读取数据

从读取管道的数据，需要访问 source 通道，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 = 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。

12

Java NIO 与 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 IO 的差异、它们的使用场景，以及它们如何影响您的代码设计。

#

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
面向流	面向缓冲
阻塞IO	非阻塞IO
无	选择器

#

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 IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要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不同。数据读取到一个它稍后处理的缓冲区，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，还需要检查是否该缓冲区中包含所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未处理的数据。

# #

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，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，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。而不是保持线程阻塞，所以直至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。一个线程请求写入一些数据到某通道，但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 的空闲时间用于在其它通道上执行 IO 操作，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 ）。

#

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。

# #

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：

1. 对 NIO 或 IO 类的 API 调用。
2. 数据处理。
3. 用来处理数据的线程数。

#

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，因为并不是仅从一个 InputStream 逐字节读取，而是数据必须先读入缓冲区再处理。

#

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的文本数据流，例如：

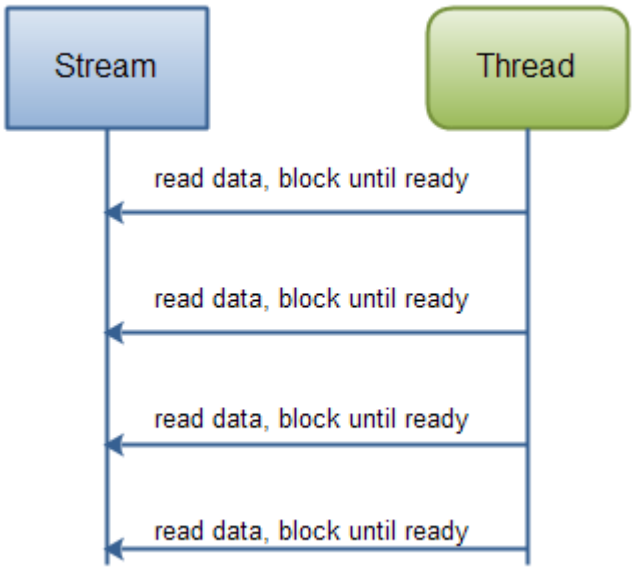
```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

该文本行的流可以这样处理：

```
InputStream input = … ; // get the InputStream from the client socket
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine   = reader.readLine();
String ageLine    = reader.readLine();
String emailLine  = reader.readLine();
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，你就知道肯定文本行就已读完， 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 readline() 调用返回的时候，你知道这行包含年龄等。 正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线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）。下图也说明了这条原则：



（Java IO: 从一个阻塞的流中读数据）而一个 NIO 的实现会有所不同，下面是一个简单的例子：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
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，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，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”，你能处理数据吗？显然不能，需要等待，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。

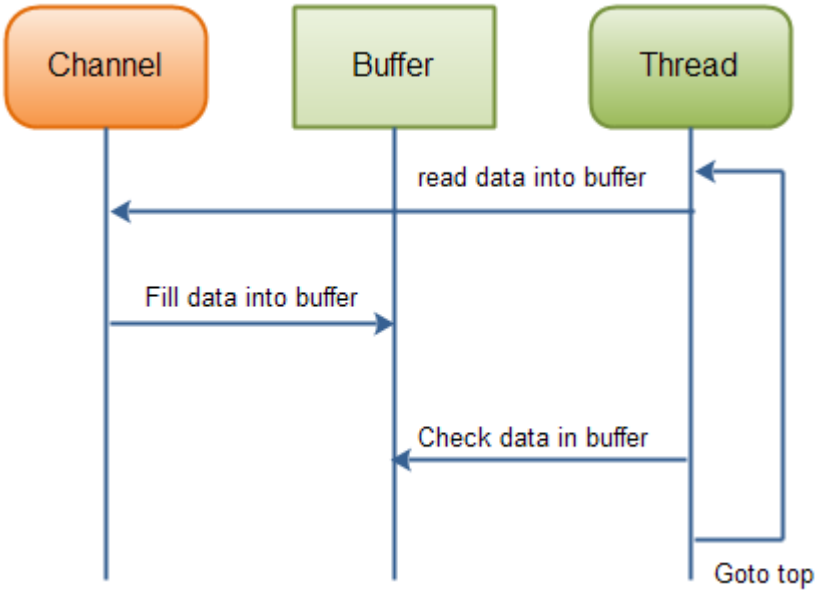
所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
bytesRead = inChannel.read(buffer);
}
```

bufferFull() 方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，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。如果没有，下一个读入缓冲区的数据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，你或许能处理其中的部分数据。但是许多情况下并非如此。下图展示了 “缓冲区数据循环就绪”：



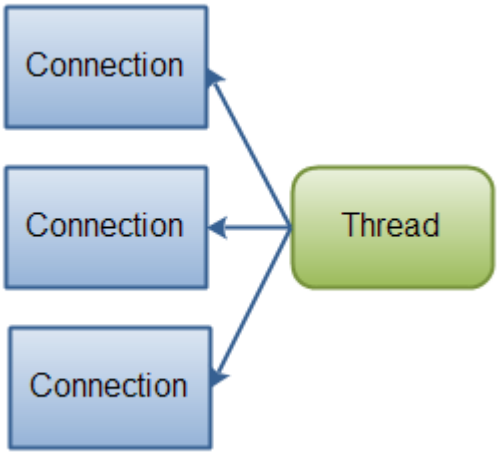
Java NIO：从一个通道里读数据，直到所有的数据都读到缓冲区里。

#

用来处理数据的线程数

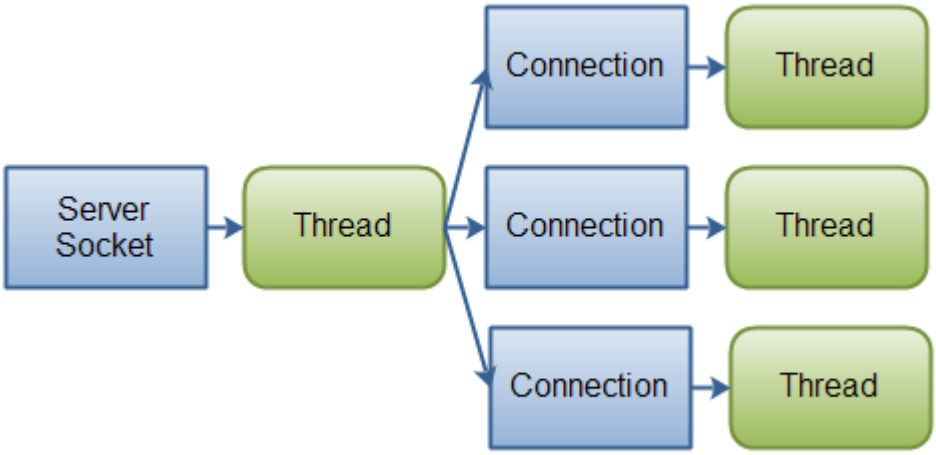
NIO 可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

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，例如聊天服务器，实现 NIO 的服务器可能是一个优势。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，使用一个单独的线程来管理你所有出站连接，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：



Java NIO: 单线程管理多个连接。

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 服务器实现可能非常契合。下图说明了一个典型的 IO 服务器设计：



Java IO：一个典型的 IO 服务器设计 – 一个连接通过一个线程处理。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/java-nio/