Java NIO 教程

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。

Java NIO: 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的，而 NIO 是基于通道（Channel）和缓冲区（Buffer）进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。

Java NIO: Non-blocking IO（非阻塞 IO）

Java NIO 可以让你非阻塞的使用 IO，例如：当线程从通道读取数据到缓冲区时，线程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。从缓冲区写入通道也类似。

Java NIO: 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的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道。

Java NIO 教程(一) 概述

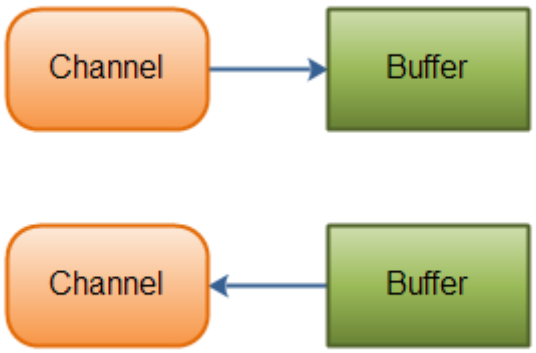
Java NIO 由以下几个核心部分组成：

- 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 有点像流。 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：



Java NIO 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`

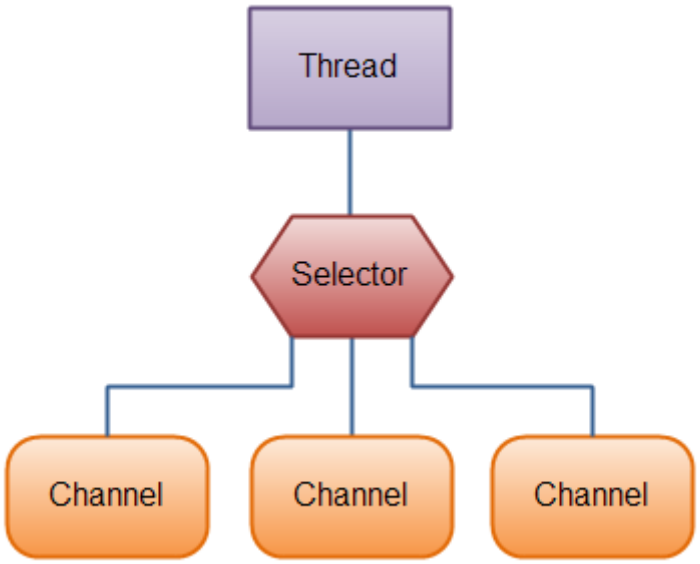
这些 Buffer 覆盖了你能通过 IO 发送的基本数据类型：`byte`, `short`, `int`, `long`, `float`, `double` 和 `char`。

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：



单线程使用 Selector 处理 3 个 Channel

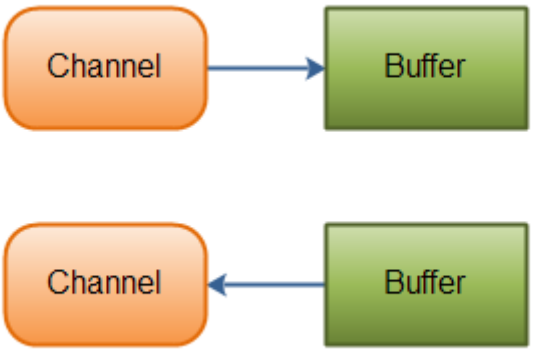
要使用 Selector，得向 Selector 注册 Channel，然后调用它的 `select()`方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

Java NIO 教程(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。

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：



NIO 数据读写流程

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：

```
    private static void useNio(){
```

```
        RandomAccessFile aFile = null;
        try {
            aFile = new
RandomAccessFile("/Users/sschen/Documents/SerialVersion.txt",
"rw");
            FileChannel inChannel = aFile.getChannel();

            ByteBuffer byteBuffer = ByteBuffer.allocate(48);
            int byteReader = inChannel.read(byteBuffer);

            while (byteReader != -1) {
                System.out.println("Read:" + byteReader);
                byteBuffer.flip();

                while (byteBuffer.hasRemaining()) {
                    System.out.println((char)byteBuffer.get());
                }
```

```
            byteBuffer.clear();


            byteReader = inChannel.read(byteBuffer);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {
            aFile.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        }

    }
```

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。

Java NIO 教程(三) 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，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。

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();



//创建容量为 48byte 的 buffer

ByteBuffer buf = ByteBuffer.allocate(48);



int bytesRead = inChannel.read(buf); //读取数据，放入 buffer

while (bytesRead != -1) {



  buf.flip();  //设置 buffer 切换模式为读模式



  while(buf.hasRemaining()){
```

```
        System.out.print((char) buf.get()); // 每次读取 1byte，也就是一个字节
    }


    buf.clear(); //清空 buffer，准备再次写入
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

Buffer 的 capacity,position 和 limit

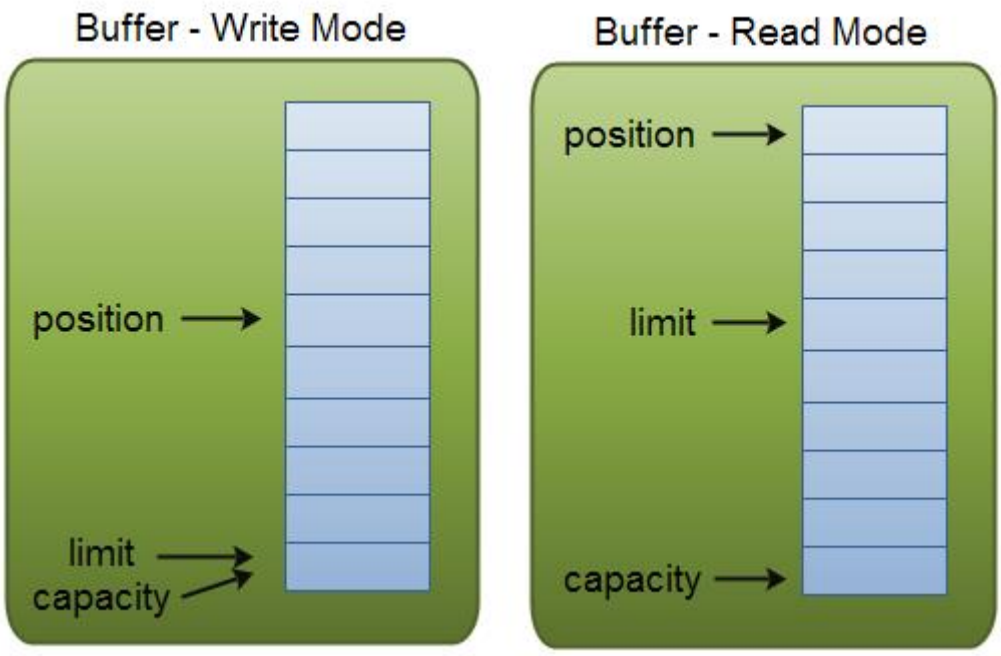
缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。

为了理解 Buffer 的工作原理，需要熟悉它的三个属性：

- capacity 容量
- position 位置
- limit 限制

`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` 等类型数据。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：

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。 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：

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：

```
//从 buffer 中读取数据到 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();



//调用几次 buffer.get()方法。例如在解析过程中



buffer.reset();  //设置 position 恢复到标记的位置.
```

equals()与 compareTo()方法

可以使用 `equals()`和 `compareTo()`方法比较两个 `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）

Java NIO 教程(四) 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。

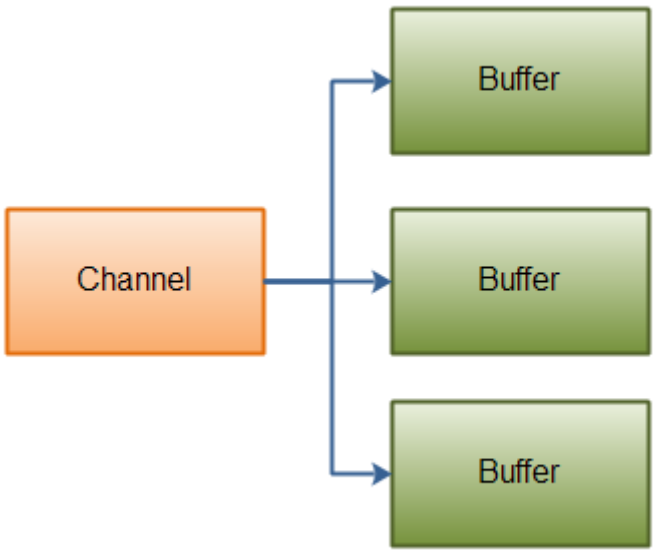
从 Channel 中分散（scatter）读取，是指在读操作时将读取的数据写入多个`buffer`中。因此，从`Channel`中读取的数据将“分散（`scatter`）”到多个`Buffer`中。

聚集（gather）写入一个 Channel，是指在写操作时将多个`buffer`的数据写入同一个`Channel`，因此，多个`Buffer`中的数据将“聚集（`gather`）”后写入到一个`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：



Java NIO: Scattering Read

代码示例如下：

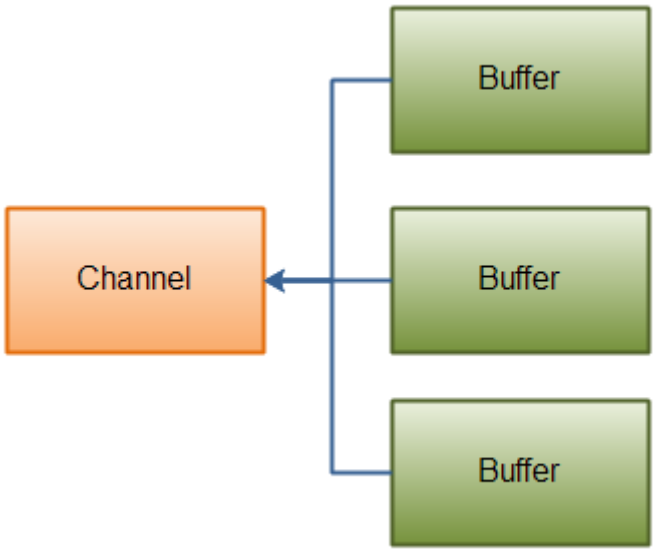
```
ByteBuffer header = ByteBuffer.allocate(128);

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：



Java NIO: Gathering Write

代码示例如下：

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

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

//此处写数据到 buffer 中

ByteBuffer[] bufferArray = { header, body };

channel.write(bufferArray);
```

`buffers` 数组是 `write()`方法的输入参数，`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。

Java NIO 教程(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`（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。

Java NIO 教程(六) Selector

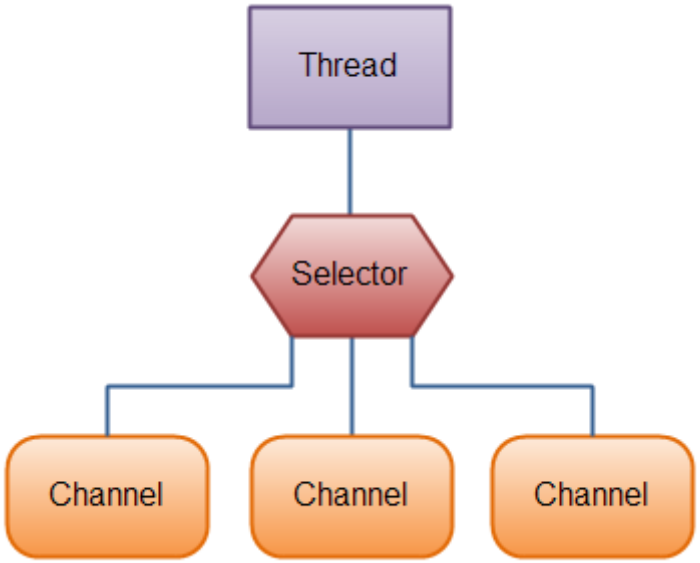
`Selector`（选择器）是 Java NIO 中能够检测一到多个 NIO 通道，并能够知晓通道是否为诸如读写事件做好准备的组件。这样，一个单独的线程可以管理多个 `channel`，从而管理多个网络连接。

为什么使用 Selector?

仅用单个线程来处理多个 `Channel` 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：



Java NIO: 一个线程使用一个 Selector 处理 3 个 Channel

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“*兴趣(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

`Channel` 能够触发了一个事件，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 说明

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“和”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();
```

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。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, theObject);
```

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<SelectionKey> selectedKeys = selector.selectedKeys();



Iterator<SelectionKey> keyIterator = selectedKeys.iterator();



while(keyIterator.hasNext()) {

```

```
SelectionKey key = keyIterator.next();


if(key.isAcceptable()) {
    // 一个连接被 ServerSocketChannel 接受

} else if (key.isConnectable()) {
    // 与远程服务器建立了连接

} else if (key.isReadable()) {
    // 一个 channel 做好了读准备

} else if (key.isWritable()) {
    // 一个 channel 做好了写准备
}

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<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

  SelectionKey key = keyIterator.next();

  if(key.isAcceptable()) {
      // 一个连接被 ServerSocketChannel 接受

  } else if (key.isConnectable()) {
      // 与远程服务器建立了连接
```

```
    } else if (key.isReadable()) {

        // 一个 channel 做好了读准备



    } else if (key.isWritable()) {

        // 一个 channel 做好了写准备

    }



    keyIterator.remove();

  }

}
```

Java NIO 教程(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。

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);
```

Java NIO 教程(八) 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();



//生成 Buffer，并向 Buffer 中写数据

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

buf.put(newData.getBytes());



//切换 buffer 为读模式

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。

Java NIO 教程(九) 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();

    //使用 socketChannel 做一些工作...
}
```

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();



    //使用 socketChannel 做一些工作...

}
```

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){

        //使用 socketChannel 做一些工作...

        }
```

```
}
```

Java NIO 教程(十) 非阻塞式服务器

即使你知道 Java NIO 非阻塞的工作特性（如 `Selector`,`Channel`,`Buffer` 等组件），但是想要设计一个非阻塞的服务器仍然是一件很困难的事。非阻塞式服务器相较于阻塞式来说要多上许多挑战。本文将会讨论非阻塞式服务器的主要几个难题，并针对这些难题给出一些可能的解决方案。

查找关于非阻塞式服务器设计方面的资料实在不太容易，所以本文提供的解决方案都是基于本人工作和想法上的。如果各位有其他的替代方案或者更好的想法，我会很乐意听取这些方案和想法！你可以在文章下方留下你的评论，或者发邮件给我（邮箱为：info@jenkov.com ）。

本文的设计思路想法都是基于 Java NIO 的。但是我相信如果某些语言中也有像 `Selector` 之类的组件的话，文中的想法也能用于该语言。据我所知，类似的组件底层操作系统会提供，所以对你来说也可以根据其中的思想运用在其他语言上。

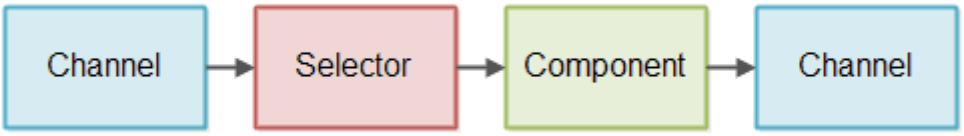
非阻塞式服务器– GitHub 仓库

我已经创建了一些简单的这些思想的概念验证呈现在这篇教程中，并且为了让你可以看到，我把源码放到了 github 资源库上了。这里是 GitHub 资源库地址:

https://github.com/jjenkov/java-nio-server

非阻塞式 IO 管道(Pipelines)

一个非阻塞式 IO 管道是由各个处理非阻塞式 IO 组件组成的链。其中包括读/写 IO。下图就是一个简单的非阻塞式 IO 管道组成：



非阻塞式 IO 管道组成

一个组件使用 `Selector` 监控 `Channel` 什么时候有可读数据。然后这个组件读取输入并且根据输入生成相应的输出。最后输出将会再次写入到一个 `Channel` 中。

一个非阻塞式 IO 管道不需要将读数据和写数据都包含，有一些管道可能只会读数据，另一些可能只会写数据。

上图仅显示了一个单一的组件。一个非阻塞式 IO 管道可能拥有超过一个以上的组件去处理输入数据。一个非阻塞式管道的长度是由他的所要完成的任务决定。

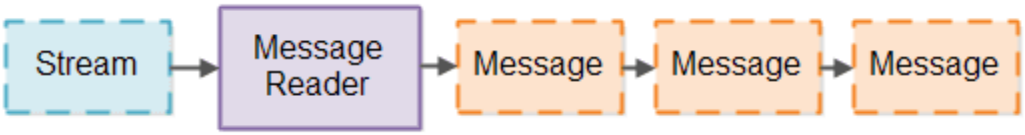
一个非阻塞 IO 管道可能同时读取多个 `Channel` 里的数据。举个例子：从多个 `SocketChannel` 管道读取数据。

其实上图的控制流程还是太简单了。这里是组件从 `Selector` 开始从 `Channel` 中读取数据，而不是 `Channel` 将数据推送给 `Selector` 进入组件中，即便上图画的就是这样。

非阻塞式 vs 阻塞式管道

非阻塞和阻塞 IO 管道两者之间最大的区别在于他们如何从底层 `Channel`(`Socket` 或者 `file`)读取数据。

IO 管道通常从流中读取数据（来自 `socket` 或者 `file`）并且将这些数据拆分为一系列连贯的消息。这和使用 `tokenizer`（这里估计是解析器之类的意思）将数据流解析为 `token`（这里应该是数据包的意思）类似。相反你只是将数据流分解为更大的消息体。我将拆分数据流成消息这一组件称为“消息读取器”（`Message Reader`）下面是 `Message Reader` 拆分流为消息的示意图：



Message Reader 拆分流为消息

一个阻塞 IO 管道可以使用类似 `InputStream` 的接口每次一个字节地从底层 `Channel` 读取数据，并且这个接口阻塞直到有数据可以读取。这就是阻塞式 `Message Reader` 的实现过程。

使用阻塞式 IO 接口简化了 `Message Reader` 的实现。阻塞式 `Message Reader` 从不用处理在流没有数据可读的情况，或者它只读取流中的部分数据并且对于消息的恢复也要延迟处理的情况。

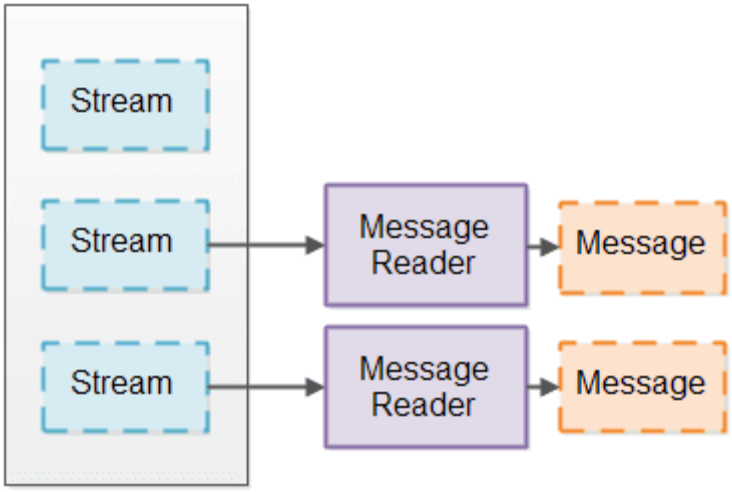
同样，阻塞式 `Message Writer`(一个将数据写入流中组件)也从不用处理只有部分数据被写入和写入消息要延迟恢复的情况。

阻塞式 IO 管道的缺陷

虽然阻塞式 `Message Reader` 容易实现，但是也有一个不幸的缺点：每一个要分解成消息的流都需要一个独立的线程。必须要这样做的理由是每一个流的 IO 接口会阻塞，直到它有数据读取。这就意味着一个单独的线程是无法尝试从一个没有数据的流中读取数据转去读另一个流。一旦一个线程尝试从一个流中读取数据，那么这个线程将会阻塞直到有数据可以读取。

如果 IO 管道是必须要处理大量并发链接服务器的一部分的话，那么服务器就需要为每一个链接维护一个线程。对于任何时间都只有几百条并发链接的服务器这确实不是什么问题。但是如果服务器拥有百万级别的并发链接量，这种设计方式就没有良好收放。每个线程都会占用栈 32bit-64bit 的内存。所以一百万个线程占用的内存将会达到 1TB！不过在此之前服务器将会把所有的内存用以处理传过来的消息（例如：分配给消息处理期间使用对象的内存）

为了将线程数量降下来，许多服务器使用了服务器维持线程池（例如：常用线程为 100）的设计，从而一次一个地从入站链接（`inbound connections`）地读取。入站链接保存在一个队列中，线程按照进入队列的顺序处理入站链接。这一设计如下图所示：(译者注：Tomcat 就是这样的)



然而，这一设计需要入站链接合理地发送数据。如果入站链接长时间不活跃，那么大量的不活跃链接实际上就造成了线程池中所有线程阻塞。这意味着服务器响应变慢甚至是没有反应。

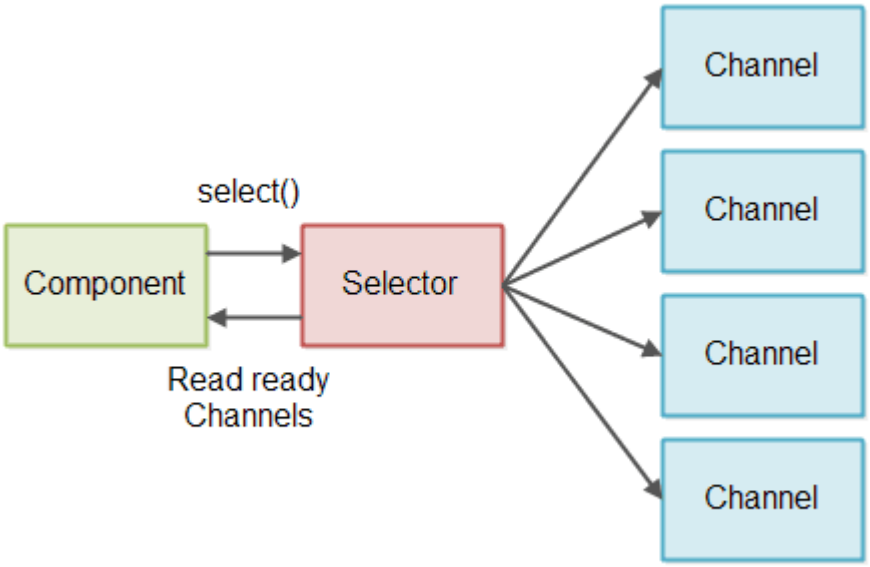
一些服务器尝试通过弹性控制线程池的核心线程数量这一设计减轻这一问题。例如，如果线程池线程不足时，线程池可能开启更多的线程处理请求。这一方案意味着需要大量的长时链接才能使服务器不响应。但是记住，对于并发线程数任然是有一个上限的。因此，这一方案仍然无法很好地解决一百万个长时链接。

基础非阻塞式 IO 管道设计

一个非阻塞式 IO 管道可以使用一个单独的线程向多个流读取数据。这需要流可以被切换到非阻塞模式。在非阻塞模式下，当你读取流信息时可能会返回 0 个字节或更多字节的信息。如果流中没有数据可读就返回 0 字节，如果流中有数据可读就返回 1+字节。

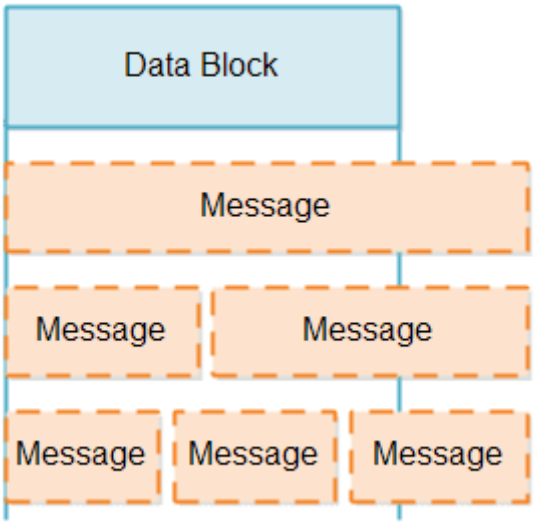
为了避免检查没有可读数据的流我们可以使用 Java NIO Selector. 一个或多个 `SelectableChannel` 实例可以同时被一个 `Selector` 注册..当你调用 `Selector`

的 `select()`或者 `selectNow()`方法它只会返回有数据读取的 `SelectableChannel` 的实例. 下图是该设计的示意图：



读取部分消息

当我们从一个 `SelectableChannel` 读取一个数据包时，我们不知道这个数据包相比于源文件是否有丢失或者重复数据（原文是：When we read a block of data from a SelectableChannel we do not know if that data block contains less or more than a message）。一个数据包可能的情况有：缺失数据（比原有消息的数据少）、与原有一致、比原来的消息的数据更多（例如：是原来的 1.5 或者 2.5 倍）。数据包可能出现的情况如下图所示：

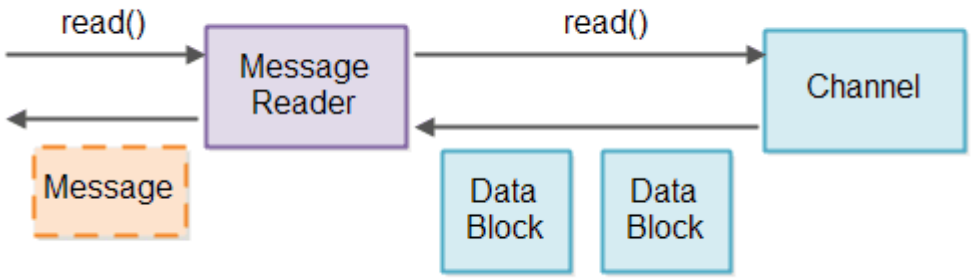


在处理类似上面这样部分信息时，有两个问题:

1. 判断你是否能在数据包中获取完整的消息。
2. 在其余消息到达之前如何处理已到达的部分消息。

判断消息的完整性需要消息读取器（`Message Reader`）在数据包中寻找是否存在至少一个完整消息体的数据。如果一个数据包包含一个或多个完整消息体，这些消息就能够被发送到管道进行处理。寻找完整消息体这一处理可能会重复多次，因此这一操作应该尽可能的快。

判断消息完整性和存储部分消息都是消息读取器(`Message Reader`)的责任。为了避免混合来自不同`Channel`的消息，我们将对每一个`Channel`使用一个`Message Reader`。设计如下图所示:



在从 `Selector` 得到可从中读取数据的 `Channel` 实例之后,与该 `Channel` 相关联的 `Message Reader` 读取数据并尝试将他们分解为消息。这样读出的任何完整消息可以被传到读取通道(`read pipeline`)任何需要处理这些消息的组件中。

一个 `Message Reader` 一定满足特定的协议。`Message Reader` 需要知道它尝试读取的消息的消息格式。如果我们的服务器可以通过协议来复用，那它需要有能够插入 `Message Reader` 实现的功能 – 可能通过接收一个 `Message Reader` 工厂作为配置参数。

存储部分消息

现在我们已经确定 `Message Reader` 有责任存储部分消息，直到收到完整的消息，我们需要弄清楚这些部分消息的存储应该如何实现。

有两个设计因素我们要考虑：

1. 我们想尽可能少地复制消息数据。复制越多，性能越低。
2. 我们希望将完整的消息存储在连续的字节序列中，使解析消息更容易。

每个 Message Reader 的缓冲区

很显然部分消息需要存储某些缓冲区中。简单的实现方式可以是每一个 `Message Reader` 内部简单地有一个缓冲区。但是这个缓冲区应该多大？它要大到足够储存最大允许储存消息。因此，如果最大允许储存消息是 1MB，那么 `Message Reader` 内部缓冲区将至少需要 1MB。

当我们的链接达到百万数量级，每个链接都使用 1MB 并没有什么作用。1,000,000 * 1MB 仍然是 1TB 的内存！那如果最大的消息是 16MB 甚至是 128MB 呢？

大小可调的缓冲区

另一个选择是在 `Message Reader` 内部实现一个大小可调的缓冲区。大小可调的缓冲区开始的时候很小，如果它获取的消息过大，那缓冲区会扩大。这样每一条链接就不一定需要如 1MB 的缓冲区。每条链接的缓冲区只要需要足够储存下一条消息的内存就行了。

有几个可实现可调大小缓冲区的方法。它们都各自有自己的优缺点，所以接下来的部分我将逐个讨论。

通过复制调整大小

实现可调大小缓冲区的第一种方式是从一个大小(例如:4KB)的缓冲区开始。如果 4KB 的缓冲区装不下一个消息，则会分配一个更大的缓冲区(如:8KB),并将大小为 4KB 的缓冲区数据复制到这个更大的缓冲区中去。

通过复制实现大小可调缓冲区的优点在于消息的所有数据被保存在一个连续的字节数组中，这就使得消息的解析更加容易。它的缺点就是在复制更大消息的时候会导致大量的数据。

为了减少消息的复制，你可以分析流进你系统的消息的大小，并找出尽量减少复制量的缓冲区的大小。例如，你可能看到大多数消息都小于 4KB，这是因为它们都仅包含很小的 `request/responses`。这意味着缓冲区的初始值应该设为 4KB。

然后你可能有一个消息大于 4KB，这通常是因为它里面包含一个文件。你可能注意到大多数流进系统的文件都是小于 128KB 的。这样第二个缓冲区的大小设置为 128KB 就较为合理。

最后你可能会发现一旦消息超过 128KB 之后，消息的大小就没有什么固定的模式，因此缓冲区最终的大小可能就是最大消息的大小。

根据流经系统的消息大小，上面三种缓冲区大小可以减少数据的复制。小于 4KB 的消息将不会复制。对于一百万个并发链接其结果是：1,000,000 * 4KB = 4GB，对于目前大多数服务器还是有可能的。介于 4KB – 128KB 的消息将只会复制一次，并且只有 4KB 的数据复制进 128KB 的缓冲区中。介于 128KB 至最大消息大小的消息将会复制两次。第一次复制 4KB，第二次复制 128KB，所以最大的消息总共复制了 132KB。假设没有那么多超过 128KB 大小的消息那还是可以接受的。

一旦消息处理完毕，那么分配的内存将会被清空。这样在同一链接接收到的下一条消息将会再次从最小缓冲区大小开始算。这样做的必要性是确保了不同连接间内存的有效共享。所有的连接很有可能在同一时间并不需要打的缓冲区。

我有一篇介绍如何实现这样支持可调整大小的数组的内存缓冲区的完整文章:

Resizable Arrays

文章包含一个 GitHub 仓库连接，其中的代码演示了是如何实现的。

通过追加调整大小

调整缓冲区大小的另一种方法是使缓冲区由多个数组组成。当你需要调整缓冲区大小时，你只需要另一个字节数组并将数据写进去就行了。

这里有两种方法扩张一个缓冲区。一个方法是分配单独的字节数组，并将这些数组保存在一个列表中。另一个方法是分配较大的共享字节数组的片段，然后保留

分配给缓冲区的片段的列表。就个人而言，我觉得片段的方式会好些，但是差别不大。

通过追加单独的数组或片段来扩展缓冲区的优点在于写入过程中不需要复制数据。所有的数据可以直接从 `socket` (`Channel`)复制到一个数组或片段中。

以这种方式扩展缓冲区的缺点是在于数据不是存储在单独且连续的数组中。这将使得消息的解析更困难，因为解析器需要同时查找每个单独数组的结尾处和所有数组的结尾处。由于你需要在写入的数据中查找消息的结尾，所以该模型并不容易使用。

TLV 编码消息

一些协议消息格式是使用 TLV 格式（类型(`Type`)、长度(`Length`)、值(`Value`)）编码。这意味着当消息到达时，消息的总长度被存储在消息的开头。这一方式你可以立即知道应该对整个消息分配多大的内存。

TLV 编码使得内存管理变得更加容易。你可以立即知道要分配多大的内存给这个消息。只有部分在结束时使用的缓冲区才会使得内存浪费。

TLV 编码的一个缺点是你要在消息的所有数据到达之前就分配好这个消息需要的所有内存。一些慢连接可能因此分配完你所有可用内存，从而使得你的服务器无法响应。

此问题的解决方法是使用包含多个 TLV 字段的消息格式。因此，服务器是为每个字段分配内存而不是为整个消息分配内存，并且是字段到达之后再分配内存。然而，一个大消息中的一个大字段在你的内存管理有同样的影响。

另外一个方案就是对于还未到达的信息设置超时时间，例如 10-15 秒。当恰好有许多大消息到达服务器时，这个方案能够使得你的服务器可以恢复，但是仍然会造成服务器一段时间无法响应。另外，恶意的 DoS（Denial of Service 拒绝服务）攻击仍然可以分配完你服务器的所有内存。

TLV 编码存在许多不同的形式。实际使用的字节数、自定字段的类型和长度都依赖于每一个 TLV 编码。TLV 编码首先放置字段的长度、然后是类型、然后是值（一个 LTV 编码）。 虽然字段的顺序不同，但它仍然是 TLV 的一种。

TLV 编码使内存管理更容易这一事实，其实是 HTTP 1.1 是如此可怕的协议的原因之一。 这是他们试图在 HTTP 2.0 中修复数据的问题之一，数据在 LTV 编码帧中传输。这也是为什么我们使用 TLV 编码的 VStack.co project 设计了我们自己的网络协议。

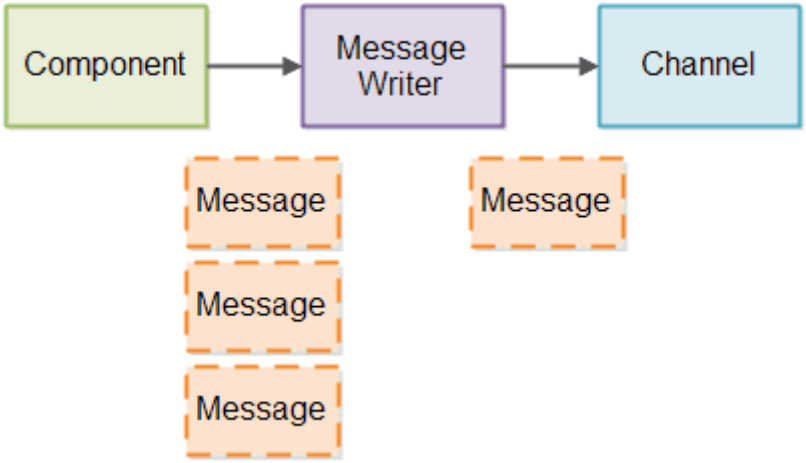
写部分数据

在非阻塞 IO 管道中写数据仍然是一个挑战。当你调用一个处于非阻塞式 Channel 对象的 `write(ByteBuffer)`方法时，`ByteBuffer` 写入多少数据是无法保证的。`write(ByteBuffer)`方法会返回写入的字节数，因此可以跟踪写入的字节数。这就是挑战：跟踪部分写入的消息，以便最终可以发送一条消息的所有字节。

为了管理部分消息写入`Channel`，我们将创建一个消息写入器（`Message Writer`）。就像 `Message Reader` 一样，每一个要写入消息的 `Channel` 我们都需要一个 `Message Writer`。在每个 `Message Writer` 中，我们跟踪正在写入的消息的字节数。

如果达到的消息量超过 `Message Writer` 可直接写入 `Channel` 的消息量，消息就需要在 `Message Writer` 排队。然后 `Message Writer` 尽快地将消息写入到 `Channel` 中。

下图是部分消息如何写入的设计图：



为了使 `Message Writer` 能够尽快发送数据，`Message Writer` 需要能够不时被调用，这样就能发送更多的消息。

如果你有大量的连接那你将需要大量的 `Message Writer` 实例。检查 `Message Writer` 实例(如:一百万个)看写任何数据时是否缓慢。 首先，许多 `Message Writer` 实例都没有任何消息要发送，我们并不想检查那些 `Message Writer` 实例。其次，并不是所有的 `Channel` 实例都可以准备好写入数据。 我们不想浪费时间尝试将数据写入无法接受任何数据的 `Channel`。

为了检查 `Channel` 是否准备好进行写入，您可以使用 `Selector` 注册 `Channel`。然而我们并不想将所有的 `Channel` 实例注册到 `Selector` 中去。想象一下，如果你有 1,000,000 个连接且其中大多是空闲的，并且所有的连接已经与 `Selector` 注册。然后当你调用 `select()`时，这些 `Channel` 实例的大部分将被写入就绪（它

们大都是空闲的，记得吗？）然后你必须检查所有这些连接的`Message Writer`，以查看他们是否有任何数据要写入。

为了避免检查所有消息的 Message Writer 实例和所有不可能被写入任何信息的 Channel 实例，我们使用这两步的方法：

1. 当一个消息被写入 Message Writer，Message Writer 向 Selector 注册其相关 Channel（如果尚未注册）。
2. 当你的服务器有时间时，它检查 Selector 以查看哪些注册的 Channel 实例已准备好进行写入。 对于每个写就绪 Channel，请求其关联的 Message Writer 将数据写入 Channel。 如果 Message Writer 将其所有消息写入其 Channel，则 Channel 将再次从 Selector 注册。

这两个小步骤确保了有消息写入的 Channel 实际上已经被 Selector 注册了。

汇总

正如你所见，一个非阻塞式服务器需要时不时检查输入的消息来判断是否有任何的新的完整的消息发送过来。服务器可能会在一个或多个完整消息发来之前就检查了多次。检查一次是不够的。

同样，一个非阻塞式服务器需要时不时检查是否有任何数据需要写入。如果有，服务器需要检查是否有任何相应的连接准备好将该数据写入它们。只有在第一次排队消息时才检查是不够的，因为消息可能被部分写入。

所有这些非阻塞服务器最终都需要定期执行的三个“管道”（`pipelines`）：：

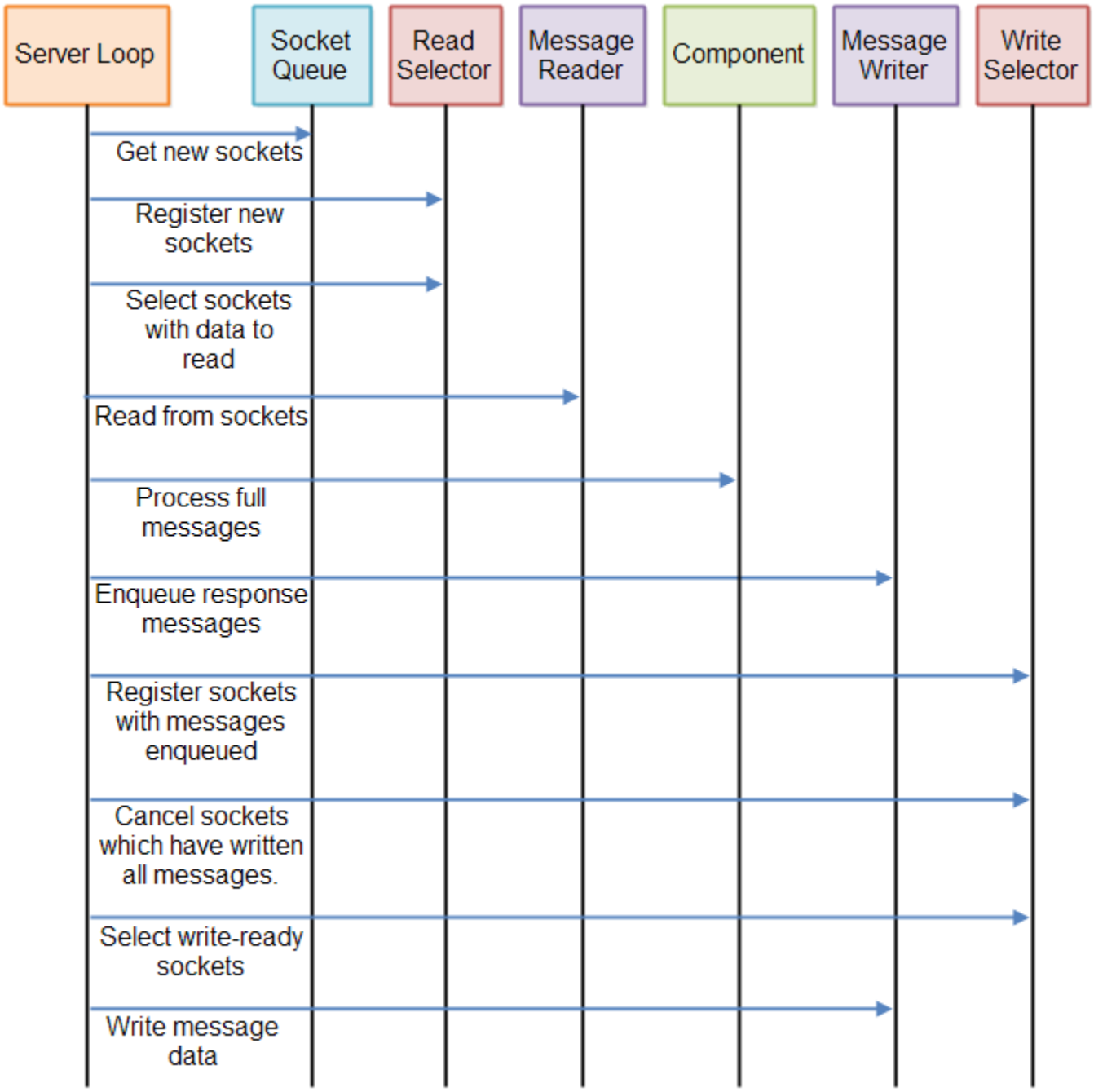
1. **读取管道（The read pipeline）**，用于检查是否有新数据从开放连接进来的。

2. **处理管道(The process pipeline)**，用于所有任何完整消息。

3. **写入管道（The write pipeline**），用于检查是否可以将任何传出的消息写入任何打开的连接。

这三条管道在循环中重复执行。你可能可以稍微优化执行。例如，如果没有排队的消息可以跳过写入管道。 或者，如果我们没有收到新的，完整的消息，也许您可以跳过处理管道。

以下是说明完整服务器循环的图：



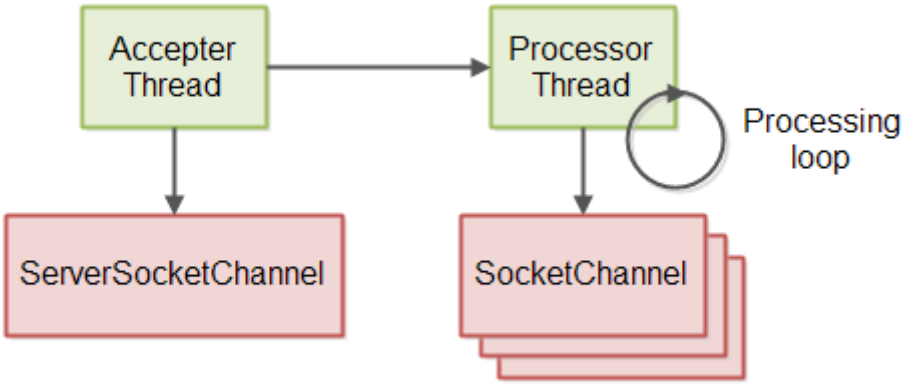
如果仍然发现这有点复杂，请记住查看 GitHub 资料库：

https://github.com/jjenkov/java-nio-server

也许看到正在执行的代码可能会帮助你了解如何实现这一点。

服务器线程模型

GitHub 资源库里面的非阻塞式服务器实现使用了两个线程的线程模式。第一个线程用来接收来自 `ServerSocketChannel` 的传入连接。第二个线程处理接受的连接，意思是读取消息，处理消息并将响应写回连接。这两个线程模型的图解如下：



上一节中说到的服务器循环处理是由处理线程（`Processor Thread`）执行。

Java NIO 教程(十一) 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。

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));
```

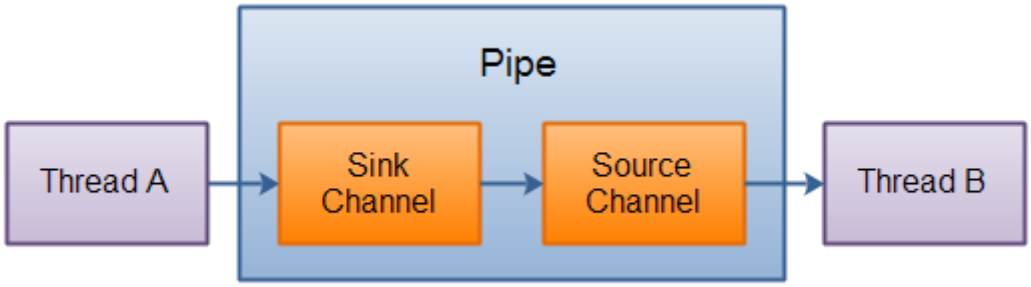
当连接后，也可以使用 `read()`和 `write()`方法，就像在用传统的通道一样。只是在数据传送方面没有任何保证。这里有几个例子：

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

int bytesWritten = channel.write(buf);
```

Java NIO 教程(十二) Pipe

Java NIO `Pipe` 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：



Pipe 原理

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 = 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。

Java NIO 教程(十三) 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 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：

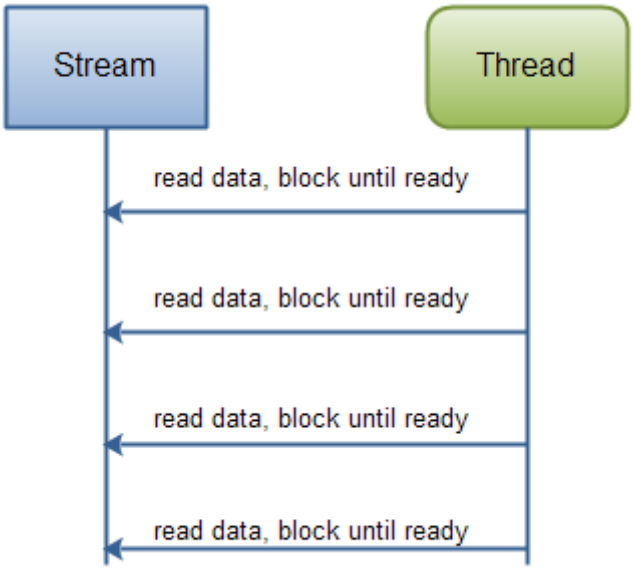
```
Name: Anna

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

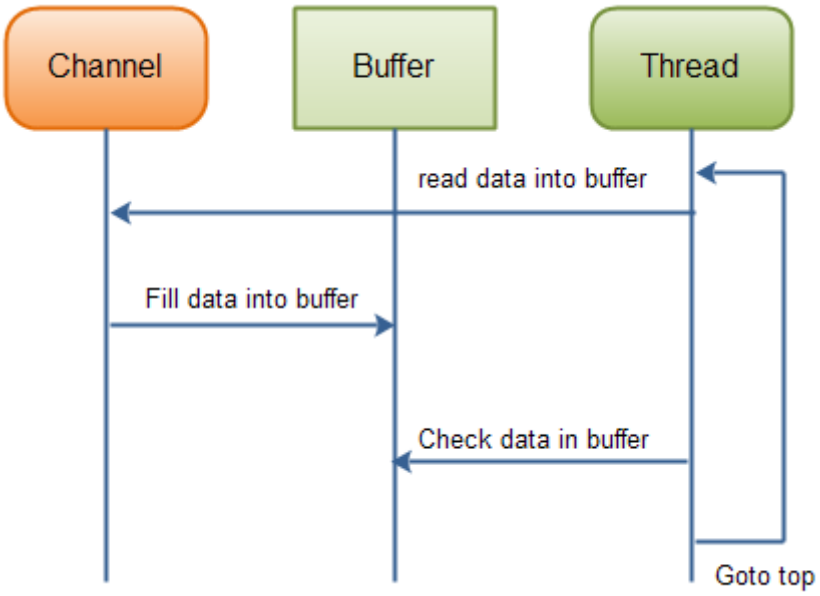
里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

```
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”：

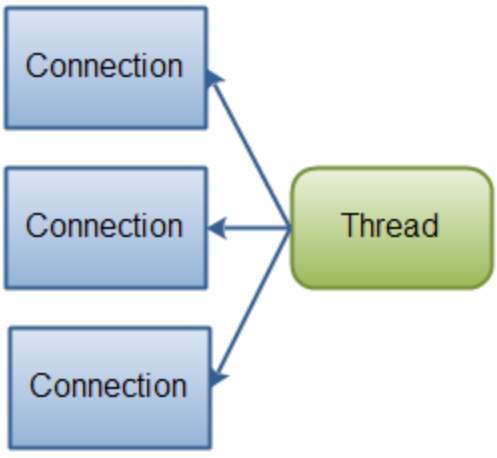


Java NIO:从一个通道里读数据，直到所有的数据都读到缓冲区里.

概要

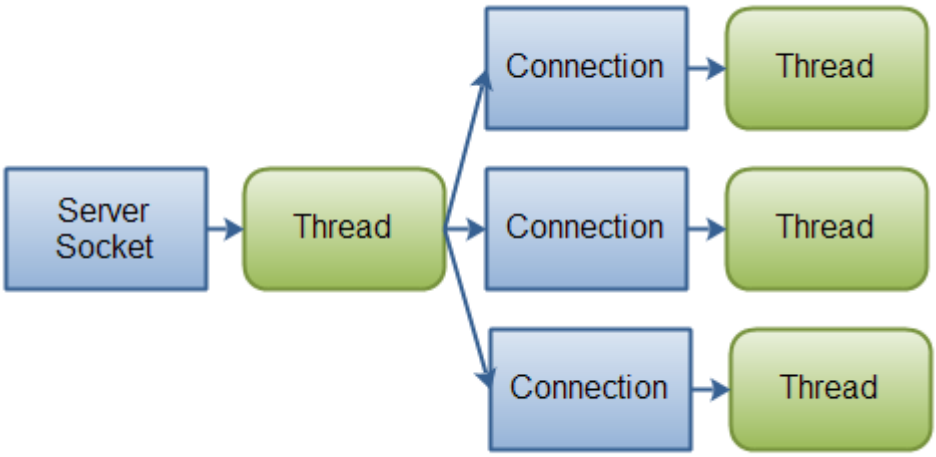
NIO 可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

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

Java NIO 教程(十四) Java NIO Path

Java Path 接口是 Java NIO 2 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。相对路径可能听起来有点混乱。别担心。我将在稍后的 Java NIO 路径教程中详细解释相关路径。

不要将文件系统路径与某些操作系统中的 path 环境变量混淆。`java.nio.file.Path` 接口与 path 环境变量没有任何关系。

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

创建一个 Path 实例

为了使用 `java.nio.file.Path` 实例必须创建一个 `Path` 实例。您可以使用 `Paths` 类(`java.nio.file.Paths`)中的静态方法来创建路径实例，名为 `Paths.get()`。下面是一个 Java `Paths.get()`示例:

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

import java.nio.file.Paths;



public class PathExample {



    public static void main(String[] args) {



        Path path = Paths.get("c:\\data\\myfile.txt");



    }
```

```
}
```

请注意示例顶部的两个导入语句。要使用 `Path` 接口和 `Paths` 类，我们必须首先导入它们。

其次，注意 `Paths.get("c:\data\myfile.txt")`方法调用。它是调用 `Path` 实例的 `Paths.get()`方法。换句话说，`Paths.get()`方法是 `Path` 实例的工厂方法。

创建一个绝对路径

创建绝对路径是通过调用 `Paths.get()`工厂方法，给定绝对路径文件作为参数来完成的。下面是创建一个表示绝对路径的路径实例的例子:

```
Path path = Paths.get("c:\\data\\myfile.txt");
```

绝对路径是 `c:\data\myfile.txt`。在 Java 字符串中，重复`` ` ``字符是必需的，因为`` ` ``是一个转义字符，这意味着下面的字符告诉我们在字符串中的这个位置要定位什么字符。通过编写`` \` ``，您可以告诉 Java 编译器在字符串中写入一个`` ` ``字符。

上面的路径是一个 Windows 文件系统路径。在 Unix 系统(Linux、MacOS、FreeBSD 等)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`.

如果您在 Windows 机器上使用了这种路径(从`/`开始的路径)，那么路径将被解释为相对于当前驱动器。例如,路径

```
/home/jakobjenkov/myfile.txt
```

可以将其解释为位于 C 盘驱动器上。那么这条路径就会对应这条完整的路径:

```
C:/home/jakobjenkov/myfile.txt
```

创建一个相对路径

相对路径是指从一条路径(基本路径)指向一个目录或文件的路径。一条相对路径的完整路径(绝对路径)是通过将基本路径与相对路径相结合而得到的。

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。下面是 Java 中的两个相对路径示例:

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



Path file     = Paths.get("d:\\data",
"projects\\a-project\\myfile.txt");
```

第一个例子创建了一个 Java `Path`的实例，指向路径(目录):`d:\data\projects`，第二个例子创建了一个`Path`的实例，指向路径(文件):`d:\data\projects\a-project\myfile.txt`

当在工作中使用相对路径时，你可以在你的路径字符串中使用两个特殊代码，它们是：

- `.`
- `..`

代码`.`表示“当前目录”，例如，如果你创建了这样一个相对路径：

```
Path currentDir = Paths.get(".");

System.out.println(currentDir.toAbsolutePath());
```

然后，Java `Path`实例对应的绝对路径将是执行上述代码的应用程序的目录。

如果。在路径字符串的中间使用`.`，表示同样的目录作为路径指向那个点。这里有一个例子说明了这一点:

```
Path currentDir = Paths.get("d:\\data\\projects\.\a-project");
```

这条路径将对应于路径:

```
d:\data\projects\a-project
```

`..`表示“父目录”或者“上一级目录”，这里有一个 Path 的 Java 例子表明这一点：

```
Path parentDir = Paths.get("..");
```

这个例子创建的 `Path` 实例对应于运行该代码的应用程序的父目录。

如果你在路径字符串代码中间使用`..`，它将对应于在路径字符串的那个点上改变一个目录。例如:

```
String path =
"d:\\data\\projects\\a-project\\..\\another-project";

Path parentDir2 = Paths.get(path);
```

这个示例创建的 Java `Path` 实例将对应于这个绝对路径:

```
d:\data\projects\another-project
```

`a-project` 目录之后的`..`代码，会修改目录到项目的父目录中，所以这个目录是指向 `another-project` 目录。

`.`和`..`代码也可以用于两个子富川的合并方法`Paths.get()`中。下面是两个简单演示 Java `Paths.get()`的例子：

```
Path path1 = Paths.get("d:\\data\\projects", ".\\a-project");



Path path2 = Paths.get("d:\\data\\projects\\a-project",

                       "..\\another-project");
```

有更多的方法可以使用 Java NIO `Path` 类来处理相对路径。在本教程中，您将了解到更多相关知识。

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。下面是一个 Java `Path.normalize()` 示例:

```
String originalPath =

        "d:\\data\\projects\\a-project\\..\\another-project";



Path path1 = Paths.get(originalPath);

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



Path path2 = path1.normalize();

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

这个 `Path` 示例首先创建一个中间带有 `..` 代码的路径字符串。然后，这个示例从这个路径字符串创建一个 `Path` 实例，并将该 `Path` 实例打印出来(实际上它会打印 `Path.toString()`)。

然后，该示例在创建的 `Path` 实例上调用 `normalize()`方法，该实例返回一个新的 `Path` 实例。这个新的、标准化的路径实例也会被打印出来。

下面是上述示例的输出:

```
path1 = d:\data\projects\a-project\..\another-project

path2 = d:\data\projects\another-project
```

正如您所看到的，标准化的路径不包含 `a-project\..`部分，因为这是多余的。移除的部分不会增加最终的绝对路径。

Java NIO 教程(十五) Java NIO 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。这个 Java NIO `Files` 教程将介绍最常用的这些方法。`Files` 类包含许多方法，所以如果您需要一个在这里没有描述的方法，那么请检查 JavaDoc。`Files` 类可能还会有一个方法来实现它。

`java.nio.file.Files` 类与 `java.nio.file.Path` 实例一起工作，因此在处理 `Files` 类之前，您需要了解 `Path` 类。

Files.exists()

`Files.exists()`方法检查给定的 `Path` 在文件系统中是否存在。

可以创建在文件系统中不存在的 `Path` 实例。例如，如果您计划创建一个新目录，您首先要创建相应的 `Path` 实例，然后创建目录。

由于 `Path` 实例可能指向，也可能没有指向文件系统中存在的路径，你可以使用 `Files.exists()`方法来确定它们是否存在(如果需要检查的话)。

这里是一个 Java `Files.exists()`的例子：

```
Path path = Paths.get("data/logging.properties");



boolean pathExists =

       Files.exists(path,

           new LinkOption[]{ LinkOption.NOFOLLOW_LINKS});
```

这个例子首先创建一个 `Path` 实例指向一个路径，我们想要检查这个路径是否存在。然后，这个例子调用 `Files.exists()`方法，然后将 `Path` 实例作为第一个参数。

注意 `Files.exists()`方法的第二个参数。这个参数是一个选项数组，它影响 `Files.exists()`如何确定路径是否存在。在上面的例子中的数组包含 `LinkOption.NOFOLLOW_LINKS`，这意味着 `Files.exists()`方法不应该在文件系统中跟踪符号链接，以确定文件是否存在。

Files.createDirectory()

`Files.createDirectory()`方法，用于根据 `Path` 实例创建一个新目录，下面是一个 `Files.createDirectory()`例子：

```
Path path = Paths.get("data/subdir");



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`。父目录是您想要创建新目录的目录。因此，它表示新目录的父目录。

Files.copy()

`Files.copy()`方法从一个路径拷贝一个文件到另外一个目录，这里是一个 Java `Files.copy()`例子：

```
Path sourcePath      = Paths.get("data/logging.properties");

Path destinationPath = Paths.get("data/logging-copy.properties");



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，该示例创建一个源和目标`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`。

重写已存在的文件

可以强制`Files.copy()`覆盖现有的文件。这里有一个示例，演示如何使用`Files.copy()`覆盖现有文件。

```
Path sourcePath      = Paths.get("data/logging.properties");
Path destinationPath = Paths.get("data/logging-copy.properties");

try {
    Files.copy(sourcePath, destinationPath,
            StandardCopyOption.REPLACE_EXISTING);
} catch(FileAlreadyExistsException e) {
    // 目标文件已存在
} catch (IOException e) {
    // 其他发生的异常
    e.printStackTrace();
```

```
}
```

请注意 `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.move()

Java NIO `Files` 还包含一个函数，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。是的,`java.io.File` 类也可以使用它的 `renameTo()`方法来完成这个操作，但是现在已经在 `java.nio.file.Files` 中有了文件移动功能。

这里有一个 Java `Files.move()`例子：

```
Path sourcePath      = Paths.get("data/logging-copy.properties");

Path destinationPath =
Paths.get("data/subdir/logging-moved.properties");



try {

    Files.move(sourcePath, destinationPath,

            StandardCopyOption.REPLACE_EXISTING);

} catch (IOException e) {
```

```
    //移动文件失败

    e.printStackTrace();

}
```

首先创建源路径和目标路径。源路径指向要移动的文件，而目标路径指向文件应该移动到的位置。然后调用`Files.move()`方法。这会导致文件被移动。

请注意传递给`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。这个参数实际上是可选的。

如果移动文件失败，`Files.move()`方法可能抛出一个`IOException`。例如，如果一个文件已经存在于目标路径中，并且您已经排除了`StandardCopyOption.REPLACE_EXISTING`选项，或者被移动的文件不存在等等。

Files.delete()

`Files.delete()`方法可以删除一个文件或者目录。下面是一个Java `Files.delete()`例子：

```
Path path = Paths.get("data/subdir/logging-moved.properties");



try {

    Files.delete(path);
```

```
} catch (IOException e) {

    // 删除文件失败

    e.printStackTrace();

}
```

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

Files.walkFileTree()

`Files.walkFileTree()`方法包含递归遍历目录树的功能。`walkFileTree()`方法将 `Path` 实例和 `FileVisitor` 作为参数。`Path` 实例指向您想要遍历的目录。`FileVisitor` 在遍历期间被调用。

在我解释遍历是如何工作之前，这里我们先了解 `FileVisitor` 接口:

```
public interface FileVisitor {

    public FileVisitResult preVisitDirectory(

        Path dir, BasicFileAttributes attrs) throws IOException;



    public FileVisitResult visitFile(
```

```
        Path file, BasicFileAttributes attrs) throws IOException;


    public FileVisitResult visitFileFailed(
        Path file, IOException exc) throws IOException;


    public FileVisitResult postVisitDirectory(
        Path dir, IOException exc) throws IOException {
}
```

您必须自己实现 `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。

这里是一个 `walkFileTree()`的例子：

```
Files.walkFileTree(path, new FileVisitor<Path>() {

  @Override

  public FileVisitResult preVisitDirectory(Path dir,
BasicFileAttributes attrs) throws IOException {
```

```
    System.out.println("pre visit dir:" + dir);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFile(Path file, BasicFileAttributes
attrs) throws IOException {
    System.out.println("visit file: " + file);
    return FileVisitResult.CONTINUE;
  }

  @Override
  public FileVisitResult visitFileFailed(Path file, IOException
exc) throws IOException {
    System.out.println("visit file failed: " + file);
    return FileVisitResult.CONTINUE;
  }
```

```
  @Override
  public FileVisitResult postVisitDirectory(Path dir, IOException
exc) throws IOException {
    System.out.println("post visit directory: " + dir);
    return FileVisitResult.CONTINUE;
  }
});
```

`FileVisitor` 实现中的每个方法在遍历过程中的不同时间都被调用:

在访问任何目录之前调用 `preVisitDirectory()`方法。在访问一个目录之后调用 `postVisitDirectory()`方法。

调用 `visitFile()`在文件遍历过程中访问的每一个文件。它不会访问目录-只会访问文件。在访问文件失败时调用 `visitFileFailed()`方法。例如，如果您没有正确的权限，或者其他什么地方出错了。

这四个方法中的每个都返回一个 `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

通过返回其中一个值，调用方法可以决定如何继续执行文件。

`CONTINUE` **继续**意味着文件的执行应该像正常一样继续。

`TERMINATE` **终止**意味着文件遍历现在应该终止。

`SKIP_SIBLINGS` **跳过同级**意味着文件遍历应该继续，但不需要访问该文件或目录的任何同级。

`SKIP_SUBTREE` **跳过子级**意味着文件遍历应该继续，但是不需要访问这个目录中的子目录。这个值只有从 `preVisitDirectory()`返回时才是一个函数。如果从任何其他方法返回，它将被解释为一个 `CONTINUE` 继续。

文件搜索

这里是一个用于扩展 `SimpleFileVisitor` 的 `walkFileTree()`，以查找一个名为 `README.txt` 的文件:

```
Path rootPath = Paths.get("data");

String fileToFind = File.separator + "README.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();
```

```
}
```

递归删除目录

`Files.walkFileTree()`也可以用来删除包含所有文件和子目录的目录。`Files.delete()`方法只会删除一个目录，如果它是空的。通过遍历所有目录并删除每个目录中的所有文件(在`visitFile()`)中，然后删除目录本身(在`postVisitDirectory()`中)，您可以删除包含所有子目录和文件的目录。下面是一个递归目录删除示例:

```
Path rootPath = Paths.get("data/to-delete");



try {

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

    @Override

    public FileVisitResult visitFile(Path file,
BasicFileAttributes attrs) throws IOException {

      System.out.println("delete file: " + file.toString());

      Files.delete(file);

      return FileVisitResult.CONTINUE;
```

```
    }


    @Override
    public FileVisitResult postVisitDirectory(Path dir,
IOException exc) throws IOException {
      Files.delete(dir);
      System.out.println("delete dir: " + dir.toString());
      return FileVisitResult.CONTINUE;
    }
  });
} catch(IOException e){
  e.printStackTrace();
}
```

文件类中的其他方法

`java.nio.file.Files` 类包含许多其他有用的函数，比如用于创建符号链接的函数、确定文件大小、设置文件权限等等。有关这些方法的更多信息，请查看 `java.nio.file.Files` 类的 JavaDoc。

Java NIO 教程(十六) Java NIO AsynchronousFileChannel

在 Java 7 中，`AsynchronousFileChannel` 被添加到 Java NIO。`AsynchronousFileChannel` 使读取数据，并异步地将数据写入文件成为可能。本教程将解释如何使用 `AsynchronousFileChannel`。

创建一个 AsynchronousFileChannel

您可以通过它的静态方法 `open()`创建一个 `AsynchronousFileChannel`。下面是创建 `AsynchronousFileChannel` 的示例:

```
Path path = Paths.get("data/test.xml");



AsynchronousFileChannel fileChannel =

    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
```

`open()`方法的第一个参数是指向与 `AsynchronousFileChannel` 相关联的文件的 `Path` 实例。

第二个参数是一个或多个打开选项，它告诉 `AsynchronousFileChannel` 在底层文件上执行哪些操作。在本例中，我们使用了 `StandardOpenOption.READ` 选项。阅读意味着该文件将被打开以供阅读。

读取数据

您可以通过两种方式从 `AsynchronousFileChannel` 读取数据。读取数据的每一种方法都调用 `AsynchronousFileChannel` 的 `read()`方法之一。这两种读取数据的方法都将在下面的部分中介绍。

通过 Future 阅读数据

从 `AsynchronousFileChannel` 读取数据的第一种方法是调用返回 `Future` 的 read()方法。下面是如何调用这个 read()方法的示例:

```
Future<Integer> operation = fileChannel.read(buffer, 0);
```

`read()`方法的这个版本将 `ByteBuffer` 作为第一个参数。从 `AsynchronousFileChannel` 读取的数据被读入这个 `ByteBuffer`。第二个参数是文件中的字节位置，以便开始读取。

`read()`方法会立即返回，即使读操作还没有完成。通过调用 `read()`方法返回的 `Future` 实例的 `isDone()`方法，您可以检查读取操作是否完成。

下面是一个更长的示例，展示如何使用 `read()`方法的这个版本:

```
AsynchronousFileChannel fileChannel =

    AsynchronousFileChannel.open(path, StandardOpenOption.READ);



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();
```

这个例子创建了一个 `AsynchronousFileChannel`，然后创建一个 `ByteBuffer`，它被传递给 `read()`方法作为参数，以及一个 `0` 的位置。在调用 `read()`之后，这个示例循环，直到返回的 `isDone()`方法返回 true。当然，这不是非常有效地使用 CPU，但是您需要等到读取操作完成之后才会执行。

读取操作完成后，数据读取到 `ByteBuffer` 中，然后进入一个字符串并打印到 `System.out` 中。

通过一个 CompletionHandler 读取数据

从`AsynchronousFileChannel`读取数据的第二种方法是调用`read()`方法版本，该方法将一个`CompletionHandler`作为参数。下面是如何调用`read()`方法:

```
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) {


    }
});
```

一旦读取操作完成，将调用 `CompletionHandler` 的 `completed()`方法。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。在本例中，它是 `ByteBuffer`，数据也被读取。您可以自由选择要附加的对象。

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

写数据

就像阅读一样，您可以通过两种方式将数据写入一个 `AsynchronousFileChannel`。写入数据的每一种方法都调用异步文件通道的 `write()`方法之一。这两种方法都将在下面的部分中介绍。

通过 `Future` 写数据

`AsynchronousFileChannel` 还允许您异步地写数据。下面是一个完整的 Java `AsynchronousFileChannel` 示例:

```
Path path = Paths.get("data/test-write.txt");
```

```
AsynchronousFileChannel fileChannel =

    AsynchronousFileChannel.open(path,
StandardOpenOption.WRITE);



ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;



buffer.put("test data".getBytes());

buffer.flip();



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

buffer.clear();



while(!operation.isDone());



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

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，在此代码生效之前，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`。

您可以确保该 `Path` 指向的文件具有以下代码:

```
if(!Files.exists(path)){

    Files.createFile(path);

}
```

通过一个 CompletionHandler 写入数据

您还可以使用一个 `CompletionHandler` 将数据写入到 `AsynchronousFileChannel` 中，以告诉您何时完成写入，而不是 `Future`。下面是一个将数据写入到 `AsynchronousFileChannel` 的示例，该通道有一个 `CompletionHandler`:

```
Path path = Paths.get("data/test-write.txt");

if(!Files.exists(path)){

    Files.createFile(path);

}
```

```
AsynchronousFileChannel fileChannel =

    AsynchronousFileChannel.open(path,
StandardOpenOption.WRITE);



ByteBuffer buffer = ByteBuffer.allocate(1024);

long position = 0;



buffer.put("test 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。如果由于某种原因而写失败，则会调用 `failed()`方法。

注意如何将 `ByteBuffer` 用作附件——该对象被传递给 `CompletionHandler` 的方法。