

Java NIO 框架 Netty 教程 (一) – Hello Netty

先啰嗦两句，如果你还不知道 Netty 是做什么的能做什么。那可以先简单的搜索了解一下。我只能说 Netty 是一个 NIO 的框架，可以用于开发分布式的 Java 程序。具体能做什么，各位可以尽量发挥想象。技术，是服务于人而不是局限住人的。

Netty 的简介和下载可参考：《[开源 Java 高性能 NIO 框架推荐 – Netty](#)》。注意，此时的最新版已经为 3.5.2.Final。

如果你已经万事具备，那么我们先从一段代码开始。程序员们习惯的上手第一步，自然是"Hello world"，不过 Netty 官网的例子却偏偏抛弃了"Hello world"。那我们就自己写一个最简单的"Hello world"的例子，作为上手。

?

HelloServer.java

```
01/**  
02 * Netty 服务端代码  
03 *  
04 * @author lihz  
05 * @alia OneCoder  
06 * @blog http://www.coderli.com  
07 */  
08public class HelloServer {  
09  
10    public static void main(String args[]) {  
11        // Server 服务启动器  
12        ServerBootstrap bootstrap = new ServerBootstrap(  
13            new NioServerSocketChannelFactory(  
14                Executors.newCachedThreadPool(),  
15                Executors.newCachedThreadPool()));  
16        // 设置一个处理客户端消息和各种消息事件的类(Handler)  
17        bootstrap  
18            .setPipelineFactory(new ChannelPipelineFactory() {  
19                @Override  
20                public ChannelPipeline getPipeline()  
21                    throws Exception {  
22                    return Channels  
23                        .pipeline(new  
24HelloServerHandler());  
25                }  
26            });  
27        // 开放 8000 端口供客户端访问。  
28        bootstrap.bind(new InetSocketAddress(8000));  
29    }  
30  
31    private static class HelloServerHandler extends  
32        SimpleChannelHandler {  
33  
34        /**
```

```
35     * 当有客户端绑定到服务端的时候触发，打印"Hello world, I'm server."
36     *
37     * @author OneCoder
38     * @author lihz
39     */
40     @Override
41     public void channelConnected(
42             ChannelHandlerContext ctx,
43             ChannelStateEvent e) {
44         System.out.println("Hello world, I'm server.");
45     }
46 }
?
```

HelloClient.java

```
01 /**
02  * Netty 客户端代码
03  *
04  * @author lihz
05  * @author OneCoder
06  * @blog http://www.coderli.com
07  */
08 public class HelloClient {
09
10     public static void main(String args[]) {
11         // Client 服务启动器
12         ClientBootstrap bootstrap = new ClientBootstrap(
13             new NioClientSocketChannelFactory(
14                 Executors.newCachedThreadPool(),
15                 Executors.newCachedThreadPool()));
16         // 设置一个处理服务端消息和各种消息事件的类(Handler)
17         bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
18             @Override
19             public ChannelPipeline getPipeline() throws Exception {
20                 return Channels.pipeline(new HelloClientHandler());
21             }
22         });
23         // 连接到本地的 8000 端口的服务端
24         bootstrap.connect(new InetSocketAddress(
25             "127.0.0.1", 8000));
26     }
27
28     private static class HelloClientHandler extends SimpleChannelHandler {
```

29

```
30
31     /**
32      * 当绑定到服务端的时候触发，打印"Hello world, I'm client."
33      *
34      * @alia OneCoder
35      * @author lihz
36      */
37
38     @Override
39     public void channelConnected(ChannelHandlerContext ctx,
40                               ChannelStateEvent e) {
41         System.out.println("Hello world, I'm client.");
42     }
43}
```

既然是分布式的，自然要分多个服务。Netty 中，需要区分 Server 和 Client 服务。所有的 Client 都是绑定在 Server 上的，他们之间是不能通过 Netty 直接通信的。（自己采用的其他手段，不包括在内。）。白话一下这个通信过程，Server 端开放端口，供 Client 连接，Client 发起请求，连接到 Server 指定的端口，完成绑定。随后便可自由通信。其实就是普通 Socket 连接通信的过程。

Netty 框架是基于事件机制的，简单说，就是发生什么事，就找相关处理方法。就跟着火了找 119，抢劫了找 110 一个道理。所以，这里，我们处理的是当客户端和服务端完成连接以后的这个事件。什么时候完成的连接，Netty 知道，他告诉我了，我就负责处理。这就是框架的作用。Netty，提供的事件还有很多，以后会慢慢的接触和介绍。

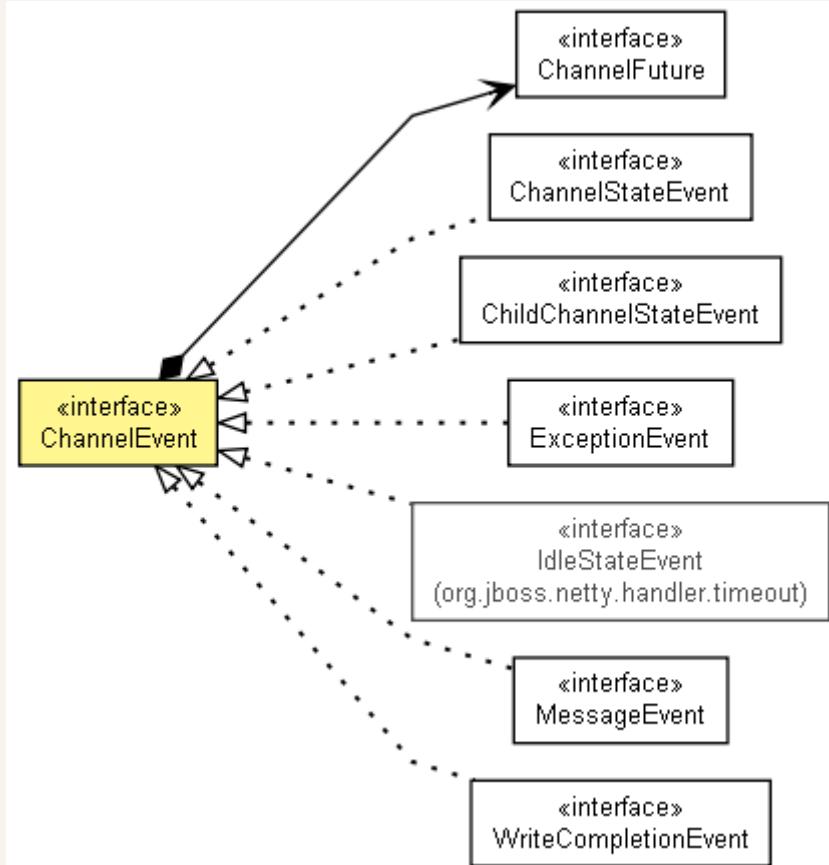
你应该已经可以上手了：）

Java NIO 框架 Netty 教程（二） – 白话概念

["Hello World"](#)的代码固然简单，不过其中的几个重要概念（类）和 Netty 的工作原理还是需要简单明确一下，至少知道其是负责什。方便自己以后更灵活的使用和扩展。

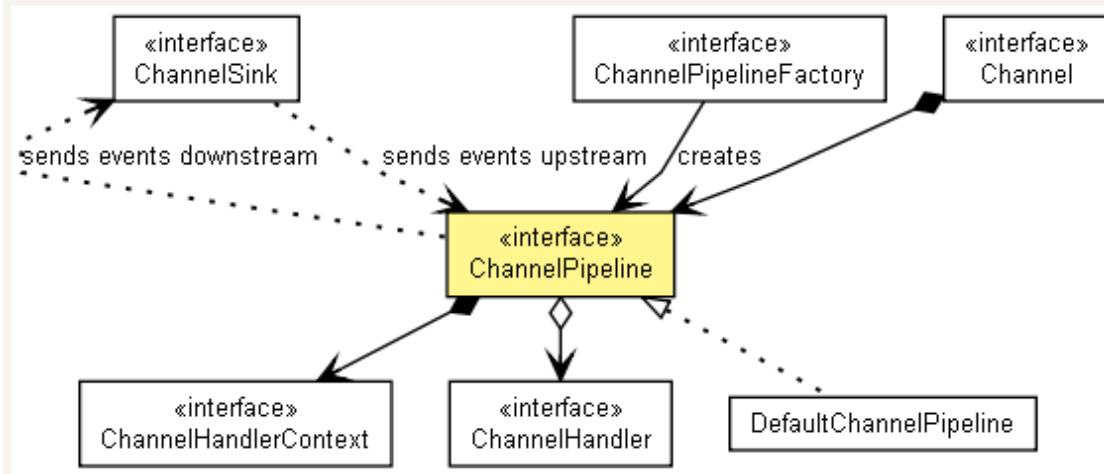
声明，笔者一介码农，不会那么多专业的词汇和缩写，只能以最简单苍白的话来形容个人的感受和体会。如果您觉得这太不专业，笔者首先只能抱歉。然后，笔者曾转过《[Netty 代码分析](#)》，您可参考。

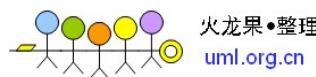
• ChannelEvent



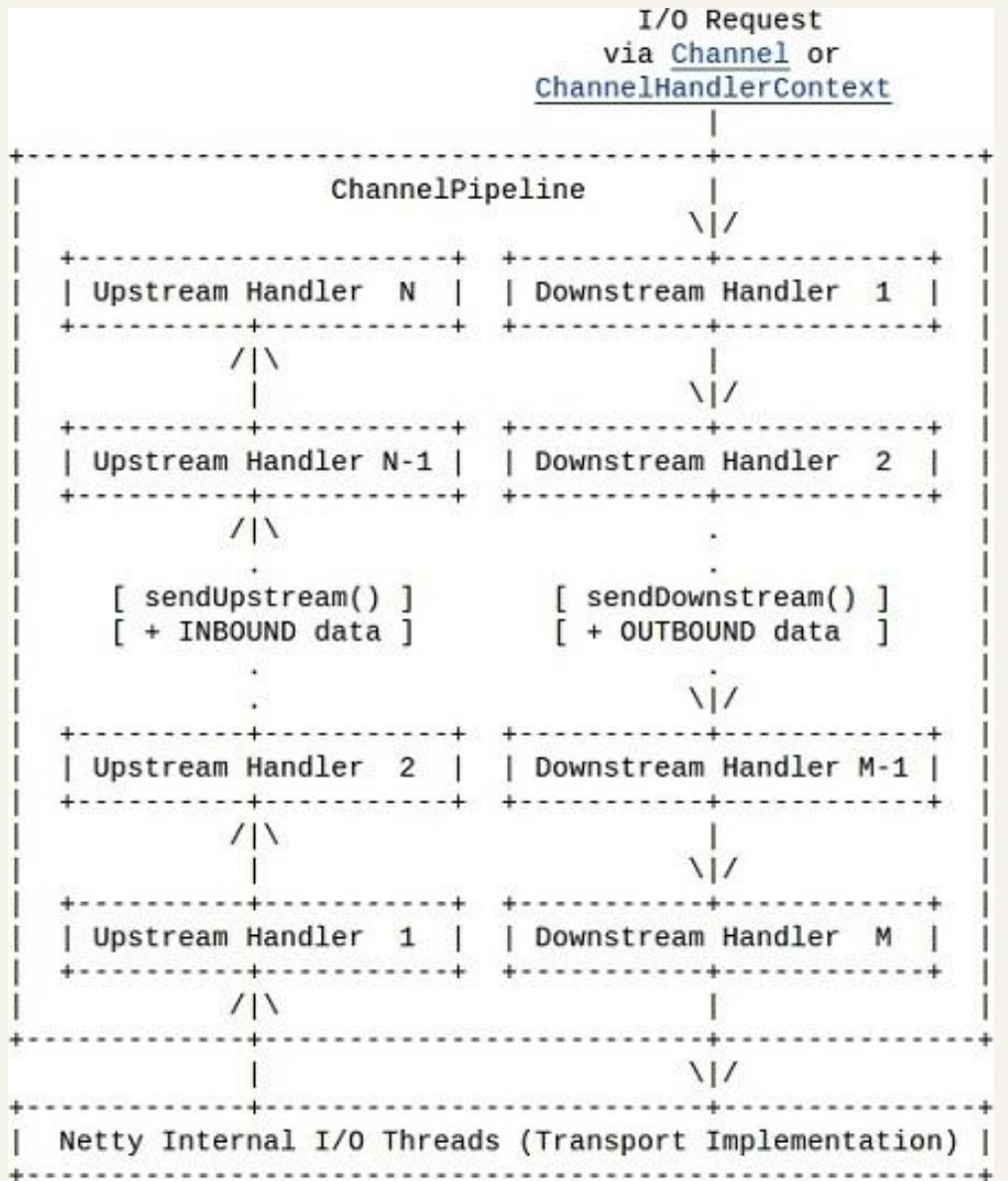
先说这个 ChannelEvent，因为 Netty 是基于事件驱动的，就是我们[上文](#)提到的，发生什么事，就通知“有关部门”。所以，不难理解，我们自己的业务代码中，一定有跟这些事件相关的处理。在[样例代码](#)，我们处理的事件，就是 channelConnected。以后，我们还会处理更多的事件。

• ChannelPipeline





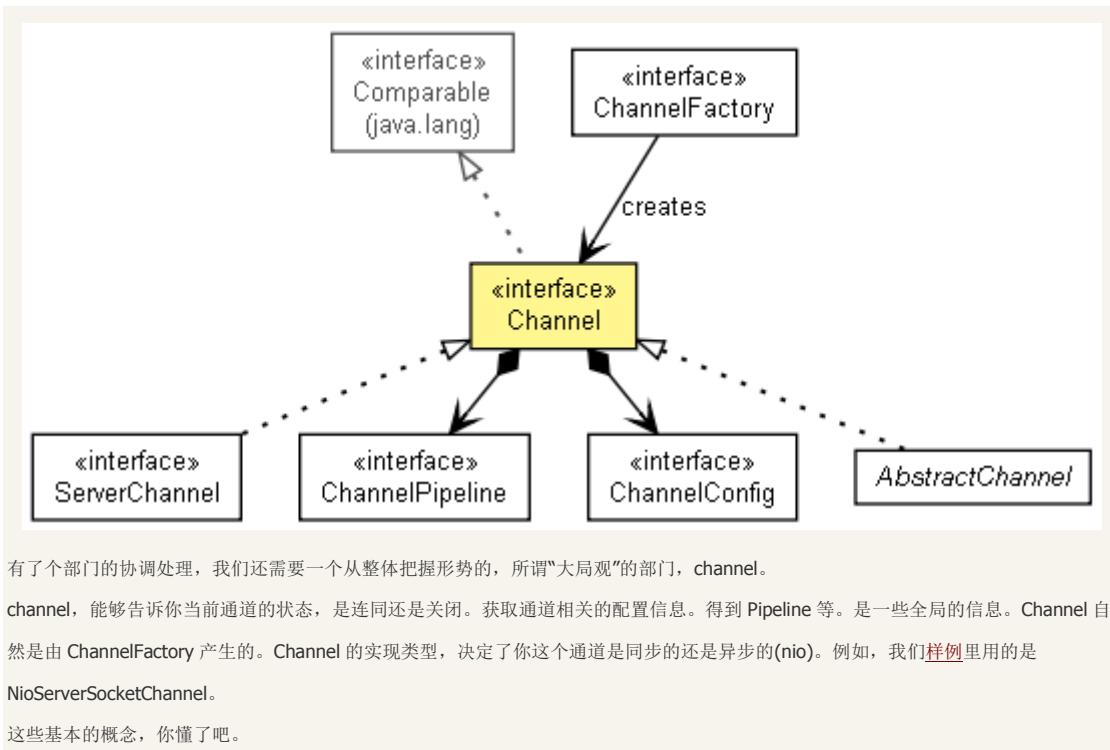
Pipeline, 翻译成中文的意思是：管道，传输途径。也就是说，在这里他是控制 ChannelEvent 事件分发和传递的。事件在管道中流转，第一站到哪，第二站到哪，到哪是终点，就是用这个 ChannelPipeline 处理的。比如：开发事件。先给 A 设计，然后给 B 开发。一个流转图，希望能给你更直观的感觉。



ChannelHandler

刚说 Pipeline 负责把事件分发到相应的站点，那个这个站点在 Netty 里，就是指 ChannelHandler。事件到了 ChannelHandler 这里，就要被具体的进行处理了，我们的[样例代码](#)里，实现的就是这样一个处理事件的“站点”，也就是说，你自己的业务逻辑一般都是从这里开始的。

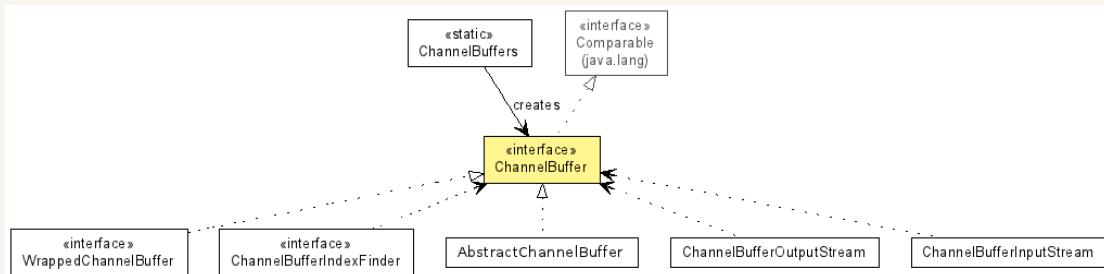
- Channel



Java NIO 框架 Netty 教程（三）- 字符串消息收发

了解了 Netty 的[基本概念](#)，开发起来应该会顺手很多。在["Hello World"](#)代码中，我们只是在完成绑定的时候，在各自的本地打印了简单的信息，并没有客户端和服务端的消息传递。这个肯定是最基本的功能。在上代码之前，先补充一个 Netty 中重要的概念，ChannelBuffer。

• ChannelBuffer



Netty 中的消息传递，都必须以字节的形式，以 ChannelBuffer 为载体传递。简单的说，就是你想直接写个字符串过去，对不起，抛异常。

虽然，Netty 定义的 writer 的接口参数是 Object 的，这可能也是会给新上手的朋友容易造成误会的地方。Netty 源码中，是这样判断的：

?

```

01     SendBuffer acquire(Object message) {
02         if (message instanceof ChannelBuffer) {
03             return acquire((ChannelBuffer) message);
04         } else if (message instanceof FileRegion) {
05             return acquire((FileRegion) message);
06         }
07
08         throw new IllegalArgumentException(
09                 "unsupported message type: " +
10             message.getClass());
11     }
  
```

所以，我们要想传递字符串，那么就必须转换成 ChannelBuffer。明确了这一点，接下来我们上代码：

?

```

01 /**
02  * @author lihz
03  * @alia OneCoder
04  * @blog http://www.coderli.com
05 */
06 public class MessageServer {
07
08     public static void main(String args[]) {
09         // Server 服务启动器
10         ServerBootstrap bootstrap = new ServerBootstrap(
11             new NioServerSocketChannelFactory(
12                 Executors.newCachedThreadPool(),
13                 Executors.newCachedThreadPool()));
14         // 设置一个处理客户端消息和各种消息事件的类(Handler)
15         bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
16             @Override
17             public ChannelPipeline getPipeline() throws Exception {
  
```

```
18                     return Channels.pipeline(new MessageServerHandler());
19                 }
20             });
21             // 开放 8000 端口供客户端访问。
22             bootstrap.bind(new InetSocketAddress(8000));
23         }
24
25     private static class MessageServerHandler extends SimpleChannelHandler {
26
27         /**
28          * 用户接受客户端发来的消息，在有客户端消息到达时触发
29          *
30          * @author lihz
31          * @alia OneCoder
32          */
33         @Override
34         public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
35             ChannelBuffer buffer = (ChannelBuffer) e.getMessage();
36             System.out.println(buffer.toString(Charset.defaultCharset()));
37         }
38
39     }
40}
```

?

```
01/**
02 * @author lihz
03 * @alia OneCoder
04 * @blog http://www.coderli.com
05 */
06public class MessageClient {
```

```
07
08     public static void main(String args[]) {
09         // Client 服务启动器
10         ClientBootstrap bootstrap = new ClientBootstrap(
11             new NioClientSocketChannelFactory(
12                 Executors.newCachedThreadPool(),
13                 Executors.newCachedThreadPool()));
14         // 设置一个处理服务端消息和各种消息事件的类(Handler)
15         bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
16             @Override
17             public ChannelPipeline getPipeline() throws Exception {
18                 return Channels.pipeline(new
19 MessageClientHandler());
20             }
21 }
```

```
21         });
22         // 连接到本地的 8000 端口的服务端
23         bootstrap.connect(new InetSocketAddress("127.0.0.1", 8000));
24     }
25
26     private static class MessageClientHandler extends SimpleChannelHandler {
27
28         /**
29          * 当绑定到服务端的时候触发，给服务端发消息。
30          *
31          * @author lihz
32          */
33
34         @Override
35         public void channelConnected(ChannelHandlerContext ctx,
36             ChannelStateEvent e) {
37             // 将字符串，构造成 ChannelBuffer，传递给服务端
38             String msg = "Hello, I'm client.";
39             ChannelBuffer buffer = ChannelBuffers.buffer(msg.length());
40             buffer.writeBytes(msg.getBytes());
41             e.getChannel().write(buffer);
42         }
43     }
44 }
```

与 "[Hello World](#)" 样例代码不同的是，客户端在 `channel` 连通后，不是在本地打印，而是将消息转换成 `ChannelBuffer` 传递给服务端，服务端接受到 `ChannelBuffer` 后，解码成字符串打印出来。

同时，通过对比可以发现，变动的只是 `Handler` 里的代码，启动服务和绑定服务的代码没有变化，也就是我们在[概念介绍](#)里提到了，关注 `Handler`，在 `Handler` 里处理我们自己的业务。所以，以后我们会只给出业务中关键代码，不会在上重复的代码：）

由于在 `Netty` 中消息的收发全依赖于 `ChannelBuffer`，所以，下一章我们将会详细的介绍 `ChannelBuffer` 的使用。我们一起学习。

Java NIO 框架 Netty 教程（四）- ChannelBuffer

在学[字符串消息收发](#)的时候，已经提到过。ChannelBuffer 是 Netty 中非常重要的概念。所有消息的收发都依赖于这个 Buffer。我们通过 Netty 的官方的文档来了解一下，基于流的消息传递机制。

In a stream-based transport such as TCP/IP, received data is stored into a socket receive buffer.

Unfortunately, the buffer of a stream-based transport is not a queue of packets but a queue of bytes. It means, even if you sent two messages as two independent packets, an operating system will not treat them as two messages but as just a bunch of bytes. Therefore, there is no guarantee that what you read is exactly what your remote peer wrote. For example, let us assume that the TCP/IP stack of an operating system has received three packets:

```
+---+---+---+
| ABC | DEF | GHI |
+---+---+---+
```

Because of this general property of a stream-based protocol, there's high chance of reading them in the following fragmented form in your application:

```
+---+---+---+
```

```
| AB | CDEFG | H | I |
```

```
+---+---+---+
```

Therefore, a receiving part, regardless it is server-side or client-side, should defrag the received data into one or more meaningful frames that could be easily understood by the application logic. In case of the example above, the received data should be framed like the following:

```
+---+---+---+
```

```
| ABC | DEF | GHI |
```

```
+---+---+---+
```

不知道您理解了没，简单翻译一下就是说。在 TCP/IP 这种基于流传递的协议中。他识别的不是你每一次发送来的消息，不是分包的。而是，只认识一个整体的流，即使分三次分别发送三段话：ABC、DEF、GHI。在传递的过程中，他就是一个具有整体长度的流。在读流的过程中，如果我一次读取的长度选择的不是三个，我可以收到类似 AB、CDEFG、H、I 这样的信息。这显然是我们不想看到的。所以说，在你写的消息收发的系统里，需要预先定义好这种解析机制，规定每帧(次)读取的长度。通过代码来理解一下：

?

```
01 /**
02 * @author lihz
03 * @alia OneCoder
04 * @blog http://www.coderli.com
05 */
06public class ServerBufferHandler extends SimpleChannelHandler {
07
08    /**
09     * 用户接受客户端发来的消息，在有客户端消息到达时触发
10     *
11     * @author lihz
12     * @alia OneCoder
13     */
14
15    @Override
16    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
17        ChannelBuffer buffer = (ChannelBuffer) e.getMessage();
18        // 五位读取
19        while (buffer.readableBytes() >= 5) {
20            ChannelBuffer tempBuffer = buffer.readBytes(5);
21            System.out.println(tempBuffer.toString(Charset.defaultCharset()));
22        }
23        // 读取剩下的信息
24        System.out.println(buffer.toString(Charset.defaultCharset()));
25    }
26}
```

?

```
01 /**
02 * @author lihz
03 * @alia OneCoder
04 * @blog http://www.coderli.com
```

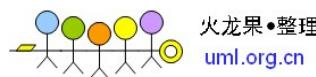
```
05     */
06 public class ClientBufferHandler extends SimpleChannelHandler {
07
08     /**
09      * 当绑定到服务端的时候触发，给服务端发消息。
10      *
11      * @alisa OneCoder
12      * @author lihz
13      */
14
15     @Override
16     public void channelConnected(ChannelHandlerContext ctx,
17         ChannelStateEvent e) {
18
19         /**
20          * 分段发送信息
21          * sendMessageByFrame(e);
22
23         /**
24          * 将<b>"Hello, I'm client."</b>分成三份发送。<br>
25          * Hello, <br>
26          * I'm<br>
27          * client.<br>
28          *
29          * @param e
30          * Netty 事件
31          * @author lihz
32          */
33
34     private void sendMessageByFrame(ChannelStateEvent e) {
35
36         String msgOne = "Hello, ";
37         String msgTwo = "I'm ";
38         String msgThree = "client.";
39
40         /**
41          * 将字符串转换成 {@link ChannelBuffer}，私有方法不进行字符串的非空判
42          * 断。
43          *
44          * @param str
45          * 待转换字符串，要求非 null
46          * @return 转换后的 ChannelBuffer
47          * @author lihz
48          */
```

```
49     private ChannelBuffer tranStr2Buffer(String str) {  
50         ChannelBuffer buffer = ChannelBuffers.buffer(str.length());  
51         buffer.writeBytes(str.getBytes());  
52         return buffer;  
53     }  
}
```

服务端输出结果:

```
?  
1 Hello  
2 , I'm  
3 clie  
4 nt.
```

这里其实，服务端是否分段发送并不会影响输出结果，也就是说，你一次性的把"Hi, I'm client."这段信息发送过来，输出的结果也是一样的。这就是开头说的，传输的是流，不分包。而只在于你如何分段读写。



Java NIO 框架 Netty 教程（五）- 消息收发次数不匹配的问题

本来周末是最好的学习时间，不过这周末收房子，可想而知事情自然也不会少。这段时间的周末，可能很少有时间学习了。见缝插针吧。不说废话了，好好学习。[上回](#)通过代码理解了 Netty 底层信息的流的传递机制，不过只是一个感性上的认识。教会你应该如何使用和使用的时候应该注意的方面。但是有一些细节的问题，并没有提及。比如在上回（[《Java NIO 框架 Netty 教程（四）- ChannelBuffer》](#)）的代码里，我们通过：

?

```
1private void sendMessageByFrame(ChannelStateEvent e) {  
2            String msgOne = "Hello, ";  
3            String msgTwo = "I'm ";  
4            String msgThree = "client. ";  
5            e.getChannel().write(tranStr2Buffer(msgOne));  
6            e.getChannel().write(tranStr2Buffer(msgTwo));  
7            e.getChannel().write(tranStr2Buffer(msgThree));  
8        }  
}
```

这样的方式，连续返送三次消息。但是如果你在服务端进行接收计数的话，你会发现，大部分时候都是接收到两次的事件请求。不过消息都是完整的。网上也有人提到过，进行 10000 次的连续放松，往往接受到的消息个数是 999X 的，总是就是消息数目上不匹配，这又是为何呢？笔者也只能通过阅读 Netty 的源码来找原因，我们一起来慢慢分析吧。

起点自然是选择在 `e.getChannel().writer()` 方法上。一路跟踪首先来到了：`AbstractNioWorker.java` 类

?

```
001protected void write0(AbstractNioChannel<?> channel) {  
002        boolean open = true;  
003        boolean addOpWrite = false;  
004        boolean removeOpWrite = false;  
005        boolean iothread = isIoThread(channel);  
006  
007        long writtenBytes = 0;  
008  
009        final SocketSendBufferPool sendBufferPool = this.sendBufferPool;  
010        final WritableByteChannel ch = channel.channel;  
011        final Queue<MessageEvent> writeBuffer = channel.writeBufferQueue;  
012        final int writeSpinCount = channel.getConfig().getWriteSpinCount();  
013        synchronized (channel.writeLock) {  
014            channel.inWriteNowLoop = true;  
015            for (;;) {  
016                MessageEvent evt = channel.currentWriteEvent;  
017                SendBuffer buf;  
018                if (evt == null) {  
019                    if ((channel.currentWriteEvent = evt =  
020writeBuffer.poll()) == null) {  
021                        removeOpWrite = true;  
022                        channel.writeSuspended = false;  
023                        break;  
024                    }  
025                }  
026            }  
027        }  
028    }  
029}
```

```
025
026                     channel.currentWriteBuffer = buf =
027sendBufferPool.acquire(evt.getMessage());
028                 } else {
029                     buf = channel.currentWriteBuffer;
030                 }
031
032             ChannelFuture future = evt.getFuture();
033             try {
034                 long localWrittenBytes = 0;
035                 for (int i = writeSpinCount; i > 0; i --) {
036                     localWrittenBytes = buf.transferTo(ch);
037                     if (localWrittenBytes != 0) {
038                         writtenBytes += localWrittenBytes;
039                         break;
040                     }
041                     if (buf.finished()) {
042                         break;
043                     }
044                 }
045
046                 if (buf.finished()) {
047                     // Successful write - proceed to the next
048message.
049                     buf.release();
050                     channel.currentWriteEvent = null;
051                     channel.currentWriteBuffer = null;
052                     evt = null;
053                     buf = null;
054                     future.setSuccess();
055                 } else {
056                     // Not written fully - perhaps the kernel
057buffer is full.
058                     addOpWrite = true;
059                     channel.writeSuspended = true;
060
061                     if (localWrittenBytes > 0) {
062                         // Notify progress listeners if
063necessary.
064                     future.setProgress(
065                             localWrittenBytes,
066                             buf.writtenBytes(),
067buf.totalBytes());
068                 }
```

```
069                     break;
070                 }
071             } catch (AsynchronousCloseException e) {
072                 // Doesn't need a user attention - ignore.
073             } catch (Throwable t) {
074                 if (buf != null) {
075                     buf.release();
076                 }
077                 channel.currentWriteEvent = null;
078                 channel.currentWriteBuffer = null;
079                 buf = null;
080                 evt = null;
081                 future.setFailure(t);
082                 if (iothread) {
083                     fireExceptionCaught(channel, t);
084                 } else {
085                     fireExceptionCaughtLater(channel, t);
086                 }
087                 if (t instanceof IOException) {
088                     open = false;
089                     close(channel, succeededFuture(channel));
090                 }
091             }
092         }
093         channel.inWriteNowLoop = false;
094
095         // Initially, the following block was executed after releasing
096         // the writeLock, but there was a race condition, and it has to be
097         // executed before releasing the writeLock:
098         //
099         //          https://issues.jboss.org/browse/NETTY-410
100         //
101         if (open) {
102             if (addOpWrite) {
103                 setOpWrite(channel);
104             } else if (removeOpWrite) {
105                 clearOpWrite(channel);
106             }
107         }
108     }
109     if (iothread) {
110         fireWriteComplete(channel, writtenBytes);
111     } else {
112         fireWriteCompleteLater(channel, writtenBytes);
113     }
114 }
```

```
}
```

```
}
```

这里，`buf.transferTo(ch);`的就是调用底层 `WritableByteChannel` 的 `write` 方法，把 `buffer` 写到管道里，传递过去。通过 `Debug` 可以看到，没调用一次这个方法，服务端的 `messageReceived` 方法就会进入断点一次。当然这个也只是表相，或者说也是在预料之内的。因为笔者从一开始就怀疑是连续写入过快导致的问题，所以测试过每次 `write` 后停顿 1 秒。再 `write` 下一次。结果一切正常。

那么我们跟到这里的意义何在呢？笔者的思路是先证明不是在 `write` 端出现的写覆盖的问题，这样就可以从 `read` 端寻找问题。这里笔者也在这里加入了一个计数，测试究竟 `transferTo` 了几次。结果确实是 3 次。

?

```
1   for (int i = writeSpinCount; i > 0; i --) {
2
3           localWrittenBytes = buf.transferTo(ch);
4
5           System.out.println(++count);
```

接下来就从接收端找找原因，在 `NioWorker` 的 `read` 方法，实现如下：

?

```
01      @Override
02      protected boolean read(SelectionKey k) {
03          final SocketChannel ch = (SocketChannel) k.channel();
04          final NioSocketChannel channel = (NioSocketChannel) k.attachment();
05
06          final ReceiveBufferSizePredictor predictor =
07              channel.getConfig().getReceiveBufferSizePredictor();
08          final int predictedRecvBufSize = predictor.nextReceiveBufferSize();
09
10          int ret = 0;
11          int readBytes = 0;
12          boolean failure = true;
13
14          ByteBuffer bb = recvBufferPool.acquire(predictedRecvBufSize);
15          try {
16              while ((ret = ch.read(bb)) > 0) {
17                  readBytes += ret;
18                  if (!bb.hasRemaining()) {
19                      break;
20                  }
21              }
22              failure = false;
23          } catch (ClosedChannelException e) {
24              // Can happen, and does not need a user attention.
25          } catch (Throwable t) {
26              fireExceptionCaught(channel, t);
27          }
28
29          if (readBytes > 0) {
30              bb.flip();
```

```
31
32         final ChannelBufferFactory bufferFactory =
33             channel.getConfig().getBufferFactory();
34         final ChannelBuffer buffer =
35     bufferFactory.getBuffer(readBytes);
36             buffer.setBytes(0, bb);
37             buffer.writerIndex(readBytes);
38
39             recvBufferPool.release(bb);
40
41             // Update the predictor.
42             predictor.previousReceiveBufferSize(readBytes);
43
44             // Fire the event.
45             fireMessageReceived(channel, buffer);
46         } else {
47             recvBufferPool.release(bb);
48         }
49
50         if (ret < 0 || failure) {
51             k.cancel(); // Some JDK implementations run into an infinite
52             loop without this.
53             close(channel, succeededFuture(channel));
54             return false;
55         }
56
57         return true;
58     }
```

在这个方法的外层是一个循环，不停的遍历，如果有 SelectionKey k 存在，则进入此方法读取 buffer 中的数据。这个 SelectionKey 区分

只是一种类型，这个设计到 Java NIO 中的 Selector 机制，这个笔者准备下讲穿插一下。属于 Netty 底层的一个重要的机制。

messageReceived 事件的触发，是在读取完当前缓冲池中所有的信息之后在触发的。这倒是可以解释，为什么即使我们收到事件的次数少，但是消息是完整的。

从目前来看，Netty 通过 Java 的 NIO 机制传递数据，数据读写跟事件没有严格的绑定机制。数据是以流的形式独立存在，读写都有一个缓冲池。

不过，这些还远未解决笔者的疑惑。笔者决定先了解一下 Selector 机制，再回头来探索这个问题。

待解决.....，如果您知道，热切期待您的指导。

Java NIO 框架 Netty 教程(六)-Java NIO Selector 模式

看到标题，您可能觉得，这跟 Netty 有什么关系呢？确实，如果你完全是使用 Netty 的，那么可能你可以完全不需要了解 Selector。但是，不得不提的是，Netty 底层关于 NIO 的实现也是基于 Java 的 Selector 的，是对 Selector 的封装。所以，我个人认为理解好 Selector 对于使用和理解 Netty 都是很多有帮助的。当然，如果您确实不关心这些，只想会用 Netty 就可以了。那么下文，您可以略过：）

笔者对于 Selector 也是新上手学习的。之前很多新人跟我交流，都会提到一个新框架或者一个新开源工具的使用和上手的问题。他们会觉得上手困难，耗费事件。不过笔者，从来没有此种感觉。这里正好，借用 Selector 的学习过程，跟大家交流一下，我上手的过程。

想要使用一个工具，自然是先了解其定位，解决问题的原理或者工作流程。所以，笔者先从网上了解了一下 Selector 大概的工作流程。

NIO 有一个主要的类 Selector,这个类似一个观察者，只要我们把需要探知的 socketchannel 告诉 Selector,我们接着做别的事情，当有事件发生时，他会通知我们，传回一组 SelectionKey,我们读取这些 Key,就会获得我们刚刚注册过的 socketchannel,然后，我们从这个 Channel 中读取数据，放心，包准能够读到，接着我们可以处理这些数据。

这是笔者摘录的一小段总结，就这一小段基本已经可以说明问题了。接下来，我们要考虑的就是，要实现这个过程，我们需要做什么？顺着描述，我们可以想象，需要选择器，需要消息传送的通道，需要注册一个事件，用于识别。通道自然需要绑定到一个地址。有了这样大概的想法，我们就可以去 API 里找相关的接口。

Selector 服务端样例代码：

```
?  
01  /**
02   * Java NIO Select 模式服务端样例代码
03   *
04   * @author lihz
05   * @alisa OneCoder
06   * @Blog http://www.coderli.com
07   * @date 2012-7-16 下午 9:22:53
08   */
09  public class NioSelectorServer {
10
11      /**
12       * @author lihz
13       * @throws IOException
14       * @alisa OneCoder
15       * @date 2012-7-16 下午 9:22:53
16       */
17      public static void main(String[] args) throws IOException {
18          // 创建一个 selector 选择器
19          Selector selector = Selector.open();
```

```

20         // 打开一个通道
21         ServerSocketChannel socketChannel = ServerSocketChannel.open();
22         // 绑定到 9000 端口
23         socketChannel.socket().bind(new InetSocketAddress(8000));
24         // 使设定 non-blocking 的方式。
25         socketChannel.configureBlocking(false);
26         // 向 Selector 注册 Channel 及我们有兴趣的事件
27         socketChannel.register(selector, SelectionKey.OP_ACCEPT);
28         for (;;) {
29             // 选择事件
30             selector.select();
31             // 当有客户端准备连接到服务端时，便会出发请求
32             for (Iterator<SelectionKey> keyIter =
33                 selector.selectedKeys()
34                         .iterator(); keyIter.hasNext();) {
35                 SelectionKey key = keyIter.next();
36                 keyIter.remove();
37                 System.out.println(key.readyOps());
38                 if (key.isAcceptable()) {
39                     System.out.println("Accept");
40                     // 接受连接到此 Channel 的连接
41                     socketChannel.accept();
42                 }
43             }
44         }
45     }
}

```

Selector 客户端样例代码:

[?](#)

```

01 /**
02  * Java NIO Selector 模式，客户端代码
03 *
04  * @author lihz
05  * @alia OneCoder
06  * @blog http://www.coderli.com
07 */
08 public class NioSelectorClient {
09
10     /**
11      * @author lihz
12      * @throws IOException
13      * @alia OneCoder
14      */
15     public static void main(String[] args) throws IOException {

```

```
16     SocketChannel channel = SocketChannel.open();
17     channel.configureBlocking(false);
18     channel.connect(new InetSocketAddress("127.0.0.1", 8000));
19 }
20 }
```

代码很简单，服务端接受到客户端的连接请求后，会打印出"Accept"信息。

简单概括就是，整一个通道，通道加个选择过滤器，看来的事件是不是我想要的，不想要的干脆不管，想要的，我就存起来，留着慢慢处理。

现在感觉是不是 Netty 确实跟这个机制比较想，如果让你去实现 Netty 先有的功能，也有思路可想了吧。

Java NIO 框架 Netty 教程(七)-再谈收发信息次数问题

在[《Java NIO 框架 Netty 教程（五） - 消息收发次数不匹配的问题》](#)里我们试图分析一个消息收发次数不匹配的问题。当时笔者还是心存疑惑的。所以决定先学习一下 Java NIO 的 [Selector](#) 机制。

经过简单的了解，笔者大胆的猜测和“武断”一下该问题的原因。

首先，[Selector](#) 机制让我们注册一个感兴趣的时间，然后只要有该时间发生，就会传递给接收端。我们写了三次，接收端一定会出发三次的。

然后，Netty 实现机制里，有个 [Buffer](#) 缓冲池，把收到的信息都缓存在里面，通过一个线程统一处理。也就是我们看到的那个 [buffer](#) 的处理过程。

Netty 的设置中，有一个一次性最多读取字节大小的设定。并且，事件的触发是在处理过缓冲池中的消息之后。我们再来看看 Netty 中读取信息的那段代码：

?

```
01 ByteBuffer bb = recvBufferPool.acquire(predictedRecvBufSize);
02     try {
03         while ((ret = ch.read(bb)) > 0) {
04             readBytes += ret;
05             if (!bb.hasRemaining()) {
06                 break;
07             }
08         }
09         failure = false;
10     } catch (ClosedChannelException e) {
11         // Can happen, and does not need a user attention.
12     } catch (Throwable t) {
13         fireExceptionCaught(channel, t);
14     }
15
16     if (readBytes > 0) {
17         bb.flip();
18
19         final ChannelBufferFactory bufferFactory =
20             channel.getConfig().getBufferFactory();
```

```

21         final ChannelBuffer buffer = bufferFactory.getBuffer(readBytes);
22         buffer.setBytes(0, bb);
23         buffer.writerIndex(readBytes);
24
25         recvBufferPool.release(bb);
26
27         // Update the predictor.
28         predictor.previousReceiveBufferSize(readBytes);
29
30         // Fire the event.
31         fireMessageReceived(channel, buffer);
32     } else {
33         recvBufferPool.release(bb);
34     }

```

可以看到，如果没有读取到字节是不会触发事件的，所以我们可能会收到 2 次或者 3 次信息。（如果发的快，解析的慢，后两次信息，一次性读取了，就 2 次，如果发送间隔长，分次解析，就收到 3 次。）原因应该就是如此。跟我们开始猜的差不多，只是不敢确认。

Java NIO 框架 Netty 教程(八)-Object 对象传递

说了这么多废话，才提到对象的传输，不知道您是不是已经不耐烦了。一个系统内部的消息传递，没有对象传递是不太现实的。下面就来说说，怎么传递对象。

如果，您看过前面的介绍，如果您善于专注本质，勤于思考。您应该也会想到，我们说过，Netty 的消息传递都是基于流，通过 ChannelBuffer 传递的，那么自然，Object 也需要转换成 ChannelBuffer 来传递。好在 Netty 本身已经给我们写好了这样的转换工具。ObjectEncoder 和 ObjectDecoder。

工具怎么用？再一次说说所谓的本质，我们之前也说过，Netty 给我们处理自己业务的空间是在灵活的可子定义的 Handler 上的，也就是说，如果我们自己去做这个转换工作，那么也应该在 Handler 里去做。而 Netty，提供给我们的 ObjectEncoder 和 Decoder 也恰恰是一组 Handler。于是，修改 Server 和 Client 的启动代码：

[?](#)

服务端

```

01 // 设置一个处理客户端消息和各种消息事件的类(Handler)
02 bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
03     @Override
04     public ChannelPipeline getPipeline() throws Exception {
05         return Channels.pipeline(
06             new ObjectDecoder(ClassResolvers.cacheDisabled(this
07                 .getClass().getClassLoader())),
08             new ObjectServerHandler());
09     }
10 });

```

[?](#)

客户端

```

1 // 设置一个处理服务端消息和各种消息事件的类(Handler)
2 bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
3     @Override

```

```
4     public ChannelPipeline getPipeline() throws Exception {
5         return Channels.pipeline(new ObjectEncoder(),
6             new ObjectClientHandler());
7     }
8 };
```

要传递对象，自然要有一个被传递模型，一个简单的 **Pojo**，当然，实现序列化接口是必须的。

?

```
1  /**
2  * @author lihz
3  * @alia OneCoder
4  * @blog http://www.coderli.com
5  */
6  public class Command implements Serializable {
7
8      private static final long serialVersionUID = 7590999461767050471L;
9
10     private String actionPerformed;
11
12     public String actionPerformed() {
13
14         return actionPerformed;
15     }
16
17     public void actionPerformed(String actionPerformed) {
18
19         this.actionPerformed = actionPerformed;
20     }
21 }
```

服务端和客户端里，我们自定义的 **Handler** 实现如下：

?

ObjectServerHandler . java

```
1  /**
2  * 对象传递服务端代码
3  *
4  * @author lihz
5  * @alia OneCoder
6  * @blog http://www.coderli.com
7  */
8  public class ObjectServerHandler extends SimpleChannelHandler {
9
10     /**
11      * 当接收到消息的时候触发
12      */
13     @Override
14     public void messageReceived(ChannelHandlerContext ctx, MessageEvent e)
15         throws Exception {
```

```
16     Command command = (Command) e.getMessage();  
17     // 打印看看是不是我们刚才传过来的那个  
18     System.out.println(command.getActionName());  
19 }  
20 }
```

?

ObjectClientHandler.java

```
1  /**  
2  * 对象传递，客户端代码  
3  *  
4  * @author lihz  
5  * @alia OneCoder  
6  * @blog http://www.coderli.com  
7  */  
8 public class ObjectClientHandler extends SimpleChannelHandler {  
9  
10    /**  
11     * 当绑定到服务端的时候触发，给服务端发消息。  
12     *  
13     * @author lihz  
14     * @alia OneCoder  
15     */  
16    @Override  
17    public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {  
18        // 向服务端发送 Object 信息  
19        sendObject(e.getChannel());  
20    }  
21  
22    /**  
23     * 发送 Object  
24     *  
25     * @param channel  
26     * @author lihz  
27     * @alia OneCoder  
28     */  
29    private void sendObject(Channel channel) {  
30        Command command = new Command();  
31        command.setActionName("Hello action.");  
32        channel.write(command);  
33    }  
34 }  
35 }
```

启动后，服务端正常打印结果：Hello action.

简单梳理一下思路：

1. 通过 Netty 传递，都需要基于流，以 ChannelBuffer 的形式传递。所以，Object -> ChannelBuffer.
2. Netty 提供了转换工具，需要我们配置到 Handler.
3. 样例从客户端 -> 服务端，单向发消息，所以在客户端配置了编码，服务端解码。如果双向收发，则需要全部配置 Encoder 和 Decoder。

这里需要注意，注册到 Server 的 Handler 是有顺序的，如果你颠倒一下注册顺序：

?

```

1   bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
2       @Override
3       public ChannelPipeline getPipeline() throws Exception {
4           return Channels.pipeline(new ObjectServerHandler(),
5               new ObjectDecoder(ClassResolvers.cacheDisabled(this
6                   .getClass().getClassLoader())));
7       }
8   });
9 });

```

结果就是，会先进入我们自己的业务，再进行解码。这自然是不行的，会强转失败。至此，你应该会用 Netty 传递对象了吧。

Java NIO 框架 Netty 教程(九)-Object 对象编/解码

看到题目，有的同学可能会想，上回不是说过对象传递了吗？是的，只是在[《Java NIO 框架 Netty 教程\(八\)-Object 对象传递》](#)中，我们只是介绍如何使用 Netty 提供的编/解码工具，完成对象的序列化。这节是想告诉你 Netty 具体是怎么做的，也许有的同学想自己完成序列化呢？况且，对象的序列化，随处可用：）

先看怎么编码。

?

ObjectEncoder.java

```

01  @Override
02      protected Object encode(ChannelHandlerContext ctx, Channel channel, Object msg) throws Exception {
03          ChannelBufferOutputStream bout =
04              new ChannelBufferOutputStream(dynamicBuffer(
05                  estimatedLength, ctx.getChannel().getConfig().getBufferFactory()));
06          bout.write(LENGTH_PLACEHOLDER);
07          ObjectOutputStream oout = new CompactObjectOutputStream(bout);
08          oout.writeObject(msg);
09          oout.flush();
10          oout.close();
11
12          ChannelBuffer encoded = bout.buffer();
13          encoded.setInt(0, encoded.writerIndex() - 4);

```

```

14         return encoded;
15     }

```

其实你早已经应该想到了，在 Java 中对对象的序列化自然跑不出 `ObjectOutputStream` 了。Netty 这里只是又做了一层包装，在流的开头增加了一个 4 字节的标志位。所以，Netty 声明，该编码和解码的类必须配套使用，与单纯的 `ObjectInputStream` 不兼容。

- * An encoder which serializes a Java object into a {@link ChannelBuffer}.
- * <p>
- * Please note that the serialized form this encoder produces is not
- * compatible with the standard {@link ObjectInputStream}. Please use
- * {@link ObjectDecoder} or {@link ObjectDecoderInputStream} to ensure the
- * interoperability with this encoder.

解码自然是先解析出多余的 4 位，然后再通过 `ObjectInputStream` 解析。

关于 Java 对象序列化的细节问题，不在文本讨论的范围内，不过不知您是否感兴趣试试自己写一个呢？所谓，多动手嘛。

[?](#)

```

01  /**
02   * Object 编码类
03   *
04   * @author lihz
05   * @alisa OneCoder
06   * @blog http://www.coderli.com
07   */
08  public class MyObjEncoder implements ChannelDownstreamHandler {
09
10     @Override
11     public void handleDownstream(ChannelHandlerContext ctx, ChannelEvent e)
12             throws Exception {
13         // 处理收发信息的情形
14         if (e instanceof MessageEvent) {
15             MessageEvent mEvent = (MessageEvent) e;
16             Object obj = mEvent.getMessage();
17             if (!(obj instanceof Command)) {
18                 ctx.sendDownstream(e);
19                 return;
20             }
21             ByteArrayOutputStream out = new ByteArrayOutputStream();
22             ObjectOutputStream oos = new ObjectOutputStream(out);
23             oos.writeObject(obj);
24             oos.flush();
25             oos.close();
26             ChannelBuffer buffer = ChannelBuffers.dynamicBuffer();
27             buffer.writeBytes(out.toByteArray());
28             e.getChannel().write(buffer);
29         } else {
30             // 其他事件，自动流转。比如，bind，connected
31             ctx.sendDownstream(e);

```

```

32         }
33     }
34 }

?

01 /**
02 * Object 解码类
03 *
04 * @author lihz
05 * @alisa OneCoder
06 * @blog http://www.coderli.com
07 */
08
09 public class MyObjDecoder implements ChannelUpstreamHandler {
10
11     @Override
12     public void handleUpstream(ChannelHandlerContext ctx, ChannelEvent e)
13         throws Exception {
14         if (e instanceof MessageEvent) {
15             MessageEvent mEvent = (MessageEvent) e;
16             if (!(mEvent.getMessage() instanceof ChannelBuffer)) {
17                 ctx.sendUpstream(mEvent);
18                 return;
19             }
20             ChannelBuffer buffer = (ChannelBuffer) mEvent.getMessage();
21             ByteArrayInputStream input = new
22             ByteArrayInputStream(buffer.array());
23             ObjectInputStream ois = new ObjectInputStream(input);
24             Object obj = ois.readObject();
25             Channels.fireMessageReceived(e.getChannel(), obj);
26         }
27     }
28 }

```

怎么样，是不是也好用？所谓，模仿，学以致用。

不过，提醒一下大家，这个实现里有很多硬编码的东西，切勿模仿，只是为了展示 Object 编解码的处理方式和在 Netty 中的应用而已。

Java NIO 框架 Netty 教程(十)-Object 对象的连续收发解析分析

如果您一直关注 OneCoder，我们之前有两篇文章介绍关于 Netty 消息连续收发的问题。（[《Java NIO 框架 Netty 教程（五）- 消息收发次数不匹配的问题》](#)、[《Java NIO 框架 Netty 教程（七）- 再谈收发信息次数问题》](#)）。如果您经常的“怀疑”和思考，我们刚介绍了 Object 的传递，您是否好奇，在 Object 传递中是否会有这样的问题？如果 Object 流的字节截断错乱，那肯定是会出错的。Netty 一定不会这么傻的，那么 Netty 是怎么做的呢？

我们先通过代码验证一下是否有这样的问题。（有问题的可能性几乎没有。）

```

?

01 /**
02 * 当绑定到服务端的时候触发，给服务端发消息。

```

```

03      *
04      * @author lihz
05      * @alia OneCoder
06      */
07      @Override
08      public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) {
09          // 向服务端发送 Object 信息
10          sendObject(e.getChannel());
11      }
12
13      /**
14      * 发送 Object
15      *
16      * @param channel
17      * @author lihz
18      * @alia OneCoder
19      */
20      private void sendObject(Channel channel) {
21          Command command = new Command();
22          command.setActionName("Hello action.");
23          Command commandOne = new Command();
24          commandOne.setActionName("Hello action. One");
25          Command commandTwo = new Command();
26          commandTwo.setActionName("Hello action. Two");
27          channel.write(commandTwo);
28          channel.write(command);
29          channel.write(commandOne);
30      }

```

打印结果：

Hello action. Two

Hello action.

Hello action. One

一切正常。那么 Netty 是怎么分割对象流的呢？看看 ObjectDecoder 怎么做的。

在 ObjectDecoder 的基类 LengthFieldBasedFrameDecoder 中注释中有详细的说明。我们这里主要介绍一下关键的代码逻辑：

[?](#)

```

01      @Override
02      protected Object decode(
03          ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) throws Exception {
04
05          if (discardingTooLongFrame) {
06              long bytesToDiscard = this.bytesToDiscard;
07              int localBytesToDiscard = (int) Math.min(bytesToDiscard, buffer.readableBytes());
08              buffer.skipBytes(localBytesToDiscard);
09              bytesToDiscard -= localBytesToDiscard;

```

```

10         this.bytesToDiscard = bytesToDiscard;
11         failIfNecessary(ctx, false);
12         return null;
13     }
14
15     if (buffer.readableBytes() < lengthFieldEndOffset) {
16         return null;
17     }
18
19     int actualLengthFieldOffset = buffer.readerIndex() + lengthFieldOffset;
20     long frameLength;
21     switch (lengthFieldLength) {
22     case 1:
23         frameLength = buffer.getUnsignedByte(actualLengthFieldOffset);
24         break;
25     case 2:
26         frameLength = buffer.getUnsignedShort(actualLengthFieldOffset);
27         break;
28     case 3:
29         frameLength = buffer.getUnsignedMedium(actualLengthFieldOffset);
30         break;
31     case 4:
32         frameLength = buffer.getUnsignedInt(actualLengthFieldOffset);
33         break;
34     .....

```

我们这里进入的是 4，还记得在编码时候的开头的 4 位占位字节吗？跟踪进去发现。

?

```

1  public int getInt(int index) {
2      return (array[index]          & 0xff) << 24 |
3             (array[index + 1] & 0xff) << 16 |
4             (array[index + 2] & 0xff) <<   8 |
5             (array[index + 3] & 0xff) <<   0;
6  }

```

原来，当初在编码时，在流开头增加的 4 字节的字符是做这个的。他记录了当前了这个对象流的长度，便于在解码时候准确的计算出该对象流的长度，正确解码。看来，我们如果自己写的对象编码解码的工具，要考虑的还有很多啊。

附：LengthFieldBasedFrameDecoder 的 JavaDoc

```

/**
 * A decoder that splits the received {@link ChannelBuffer}s dynamically by the
 * value of the length field in the message. It is particularly useful when you
 * decode a binary message which has an integer header field that represents the
 * length of the message body or the whole message.
 *
 * <p>
 * {@link LengthFieldBasedFrameDecoder} has many configuration parameters so
 * that it can decode any message with a length field, which is often seen in

```

```

* proprietary client-server protocols. Here are some example that will give
* you the basic idea on which option does what.
*
* <h3>2 bytes length field at offset 0, do not strip header</h3>
*
* The value of the length field in this example is <tt>12 (0x0C)</tt> which
* represents the length of "HELLO, WORLD". By default, the decoder assumes
* that the length field represents the number of the bytes that follows the
* length field. Therefore, it can be decoded with the simplistic parameter
* combination.
*
* <pre>
* <b>lengthFieldOffset</b> = <b>0</b>
* <b>lengthFieldLength</b> = <b>2</b>
* lengthAdjustment = 0
* initialBytesToStrip = 0 (= do not strip header)
*
* BEFORE DECODE (14 bytes)      AFTER DECODE (14 bytes)
* +---+-----+     +---+-----+
* | Length | Actual Content |-->| Length | Actual Content |
* | 0x000C | "HELLO, WORLD" |     | 0x000C | "HELLO, WORLD" |
* +---+-----+     +---+-----+
* </pre>
*
* <h3>2 bytes length field at offset 0, strip header</h3>
*
* Because we can get the length of the content by calling
* {@link ChannelBuffer#readableBytes()}, you might want to strip the length
* field by specifying <tt>initialBytesToStrip</tt>. In this example, we
* specified <tt>2</tt>, that is same with the length of the length field, to
* strip the first two bytes.
*
* <pre>
* lengthFieldOffset = 0
* lengthFieldLength = 2
* lengthAdjustment = 0
* <b>initialBytesToStrip</b> = <b>2</b> (= the length of the Length field)
*
* BEFORE DECODE (14 bytes)      AFTER DECODE (12 bytes)
* +---+-----+     +-----+
* | Length | Actual Content |-->| Actual Content |
* | 0x000C | "HELLO, WORLD" |     | "HELLO, WORLD" |
* +---+-----+     +-----+
* </pre>
*
* <h3>2 bytes length field at offset 0, do not strip header, the length field

```

```

*   represents the length of the whole message</h3>
*
* In most cases, the length field represents the length of the message body
* only, as shown in the previous examples. However, in some protocols, the
* length field represents the length of the whole message, including the
* message header. In such a case, we specify a non-zero
* <tt>lengthAdjustment</tt>. Because the length value in this example message
* is always greater than the body length by <tt>2</tt>, we specify <tt>-2</tt>
* as <tt>lengthAdjustment</tt> for compensation.

* <pre>
* lengthFieldOffset  = 0
* lengthFieldLength = 2
* <b>lengthAdjustment</b> = <b>-2</b> (= the length of the Length field)
* initialBytesToStrip = 0
*
* BEFORE DECODE (14 bytes)      AFTER DECODE (14 bytes)
* +---+-----+     +---+-----+
* | Length | Actual Content |-->| Length | Actual Content |
* | 0x000E | "HELLO, WORLD" |     | 0x000E | "HELLO, WORLD" |
* +---+-----+     +---+-----+
* </pre>
*
* <h3>3 bytes length field at the end of 5 bytes header, do not strip header</h3>
*
* The following message is a simple variation of the first example. An extra
* header value is prepended to the message. <tt>lengthAdjustment</tt> is zero
* again because the decoder always takes the length of the prepended data into
* account during frame length calculation.

* <pre>
* <b>lengthFieldOffset</b> = <b>2</b> (= the length of Header 1)
* <b>lengthFieldLength</b> = <b>3</b>
* lengthAdjustment = 0
* initialBytesToStrip = 0
*
* BEFORE DECODE (17 bytes)      AFTER DECODE (17 bytes)
* +---+-----+-----+     +---+-----+-----+
* | Header 1 | Length | Actual Content |-->| Header 1 | Length | Actual Content |
* | 0xCAFE | 0x00000C | "HELLO, WORLD" |     | 0xCAFE | 0x00000C | "HELLO, WORLD" |
* +---+-----+-----+     +---+-----+-----+
* </pre>
*
* <h3>3 bytes length field at the beginning of 5 bytes header, do not strip header</h3>
*
* This is an advanced example that shows the case where there is an extra

```

```

* header between the length field and the message body. You have to specify a
* positive <tt>lengthAdjustment</tt> so that the decoder counts the extra
* header into the frame length calculation.

* <pre>

* lengthFieldOffset  = 0
* lengthFieldLength  = 3
* <b>lengthAdjustment</b>  = <b>2</b> (= the length of Header 1)
* initialBytesToStrip = 0
*
* BEFORE DECODE (17 bytes)           AFTER DECODE (17 bytes)
* +-----+-----+-----+ +-----+-----+-----+
* | Length | Header 1 | Actual Content |-->| Length | Header 1 | Actual Content |
* | 0x00000C | 0xCAFE | "HELLO, WORLD" |     | 0x00000C | 0xCAFE | "HELLO, WORLD" |
* +-----+-----+-----+ +-----+-----+-----+
* </pre>
*
* <h3>2 bytes length field at offset 1 in the middle of 4 bytes header,
* strip the first header field and the length field</h3>
*
* This is a combination of all the examples above. There are the prepended
* header before the length field and the extra header after the length field.
* The prepended header affects the <tt>lengthFieldOffset</tt> and the extra
* header affects the <tt>lengthAdjustment</tt>. We also specified a non-zero
* <tt>initialBytesToStrip</tt> to strip the length field and the prepended
* header from the frame. If you don't want to strip the prepended header, you
* could specify <tt>0</tt> for <tt>initialBytesToSkip</tt>.

* <pre>

* lengthFieldOffset  = 1 (= the length of HDR1)
* lengthFieldLength  = 2
* <b>lengthAdjustment</b>  = <b>1</b> (= the length of HDR2)
* <b>initialBytesToStrip</b> = <b>3</b> (= the length of HDR1 + LEN)
*
* BEFORE DECODE (16 bytes)           AFTER DECODE (13 bytes)
* +---+---+---+-----+ +---+-----+
* | HDR1 | Length | HDR2 | Actual Content |-->| HDR2 | Actual Content |
* | 0xCA | 0x000C | 0xFE | "HELLO, WORLD" |     | 0xFE | "HELLO, WORLD" |
* +---+---+---+-----+ +---+-----+
* </pre>
*
* <h3>2 bytes length field at offset 1 in the middle of 4 bytes header,
* strip the first header field and the length field, the length field
* represents the length of the whole message</h3>
*
* Let's give another twist to the previous example. The only difference from

```

```

* the previous example is that the length field represents the length of the
* whole message instead of the message body, just like the third example.
* We have to count the length of HDR1 and Length into <tt>lengthAdjustment</tt>.
* Please note that we don't need to take the length of HDR2 into account
* because the length field already includes the whole header length.
* <pre>
* lengthFieldOffset  =  1
* lengthFieldLength  =  2
* <b>lengthAdjustment</b>  = <b>-3</b> (= the length of HDR1 + LEN, negative)
* <b>initialBytesToStrip</b> = <b> 3</b>
*
* BEFORE DECODE (16 bytes)           AFTER DECODE (13 bytes)
* +---+---+---+-----+    +---+-----+
* | HDR1 | Length | HDR2 | Actual Content |-->| HDR2 | Actual Content |
* | 0xCA | 0x0010 | 0xFE | "HELLO, WORLD" |    | 0xFE | "HELLO, WORLD" |
* +---+---+---+-----+    +---+-----+
* </pre>
*
* @see LengthFieldPrepender
*/

```

Java NIO 框架 Netty 教程(十一)-并发访问测试(上)

之前更新了几篇关于 JVM 研究的文章，其实也是在做本篇文章验证的时候，跑的有点远，呵呵。回归 Netty 教程，这次要讲的其实是针对一个问题的研究和验证结论。另外，最近工作比较忙，所以可能会分文章更新一些阶段性的成果，而不是全部汇总更新，以免间隔过久。

起因是一个朋友，通过微博([OneCoder 腾讯微博](#)、[OneCoder 新浪微博](#)、[OneCoder 网易微博](#)、[OneCoder 搜狐微博](#))私信给我一个问题，大意是说他在用 Netty 做并发测试的时候，会出现大量的 connection refuse 信息，问我如何解决。

没动手就没有发言权，所以 [OneCoder](#) 决定测试一下：

```

?
01  /**
02   * @author lihz
03   * @alia OneCoder
04   * @blog http://www.coderli.com
05   */
06  public class ConcurrencyNettyTestHandler extends SimpleChannelHandler {
07
08      private static int count = 0;
09
10     /**
11      * 当接受到消息的时候触发
12     */

```

```
13     @Override
14     public void channelConnected(ChannelHandlerContext ctx,
15         final ChannelStateEvent e) throws Exception {
16         for (int i = 0; i < 100000; i++) {
17             Thread t = new Thread(new Runnable() {
18                 @Override
19                 public void run() {
20                     sendObject(e.getChannel());
21                 }
22             });
23             System.out.println("Thread count: " + i);
24             t.start();
25         }
26     }
27 }
28
29 /**
30 * 发送 Object
31 *
32 * @author lihz
33 * @alia OneCoder
34 */
35 private void sendObject(Channel channel) {
36     count++;
37     Command command = new Command();
38     command.setActionName("Hello action.");
39     System.out.println("Write: " + count);
40     channel.write(command);
41 }
42 }
```

运行结果:

```
Hello action.: 99996
Hello action.: 99997
Hello action.: 99998
Hello action.: 99999
Hello action.: 100000
```

你可能会惊讶，10w 个请求都能通过？呵呵，细心的同学，可能会发现，这其实并不是并发，而只是所谓 10w 个线程的，单 channel 的伪并发，或者说是一种持续的连续访问。并且，如果你跑一下测试用例，会发现，Server 端开始接受处理消息，是在 Client 端 10w 个线程请求都结束之后再开始的。这是为什么？

其实，如果您看过 [OneCoder](#) 的《Java NIO 框架 Netty 教程(七)-再谈收发信息次数问题》，应该会有所联想。不过坦白的说，[OneCoder](#) 也是在经过一番周折，一番 Debug 以后，才发现了这个问题。当 [OneCoder](#) 在线程内断点以后，放过一个线程，接收端就会有一条信息出现，这其实是和之前文章里介绍的场景是一样的。所以，呵呵，可能对您来说，看了这篇文章，并没有更多的收获，但是对 [OneCoder](#) 来说，确实是经历了不小的周折，绕了挺大的弯子，也算是对代码的再熟悉过程吧。

下篇我们会面对真正并发的问题：）

Java NIO 框架 Netty 教程(十二)-并发访问测试(中)

写在前面：对 Netty 并发问题的测试和解决完全超出了我的预期，想说的东西越来越多。所以才出现这个中篇，也就是说，一定会有下篇。至于问题点的发现，[OneCoder](#) 也在努力验证中。

继续并发的问题。在[《Java NIO 框架 Netty 教程\(十一\)-并发访问测试\(上\)》](#)中，我们测试的其实是一种伪并发的情景。底层实际只有一个 Channel 在运作，不会出现什么无响应的问题。而实际的并发不是这个意思的，独立的客户端，自然就会有独立的 channel。也就是一个服务端很可能和很多的客户端建立关系，就有很多的 Channel，进行了多次的绑定。下面我们来模拟一下这种场景。

[?](#)

```

1  /**
2  * Netty 并发，多 connection，socket 并发测试
3  *
4  * @author lihz (OneCoder)
5  * @alisa OneCoder
6  * @Blog http://www.coderli.com
7  * @date 2012-8-13 下午 10:47:28
8  */
9
10 public class ConcurrencyNettyMultiConnection {
11
12     public static void main(String args[]) {
13         final ClientBootstrap bootstrap = new ClientBootstrap(
14             new NioClientSocketChannelFactory(
15                 Executors.newCachedThreadPool(),
16                 Executors.newCachedThreadPool()));
17
18         // 设置一个处理服务端消息和各种消息事件的类(Handler)
19         bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
20
21             @Override
22             public ChannelPipeline getPipeline() throws Exception {
23                 return Channels.pipeline(new ObjectEncoder(),
24                     new ObjectClientHandler());
25             }
26         });
27
28         for (int i = 0; i < 3; i++) {
29             bootstrap.connect(new InetSocketAddress("127.0.0.1",
30
31                 8000));
32         }
33     }
34 }
```

看到这段代码，你可能会疑问，这是在做什么。这个验证是基于酷壳的文章[《Java NIO 类库 Selector 机制解析（上）》](#)。他是基于 Java Selector 直接进行的验证，Netty 是在此之上封装了一层。其实[OneCoder](#) 也做了 Selector 层的验证。

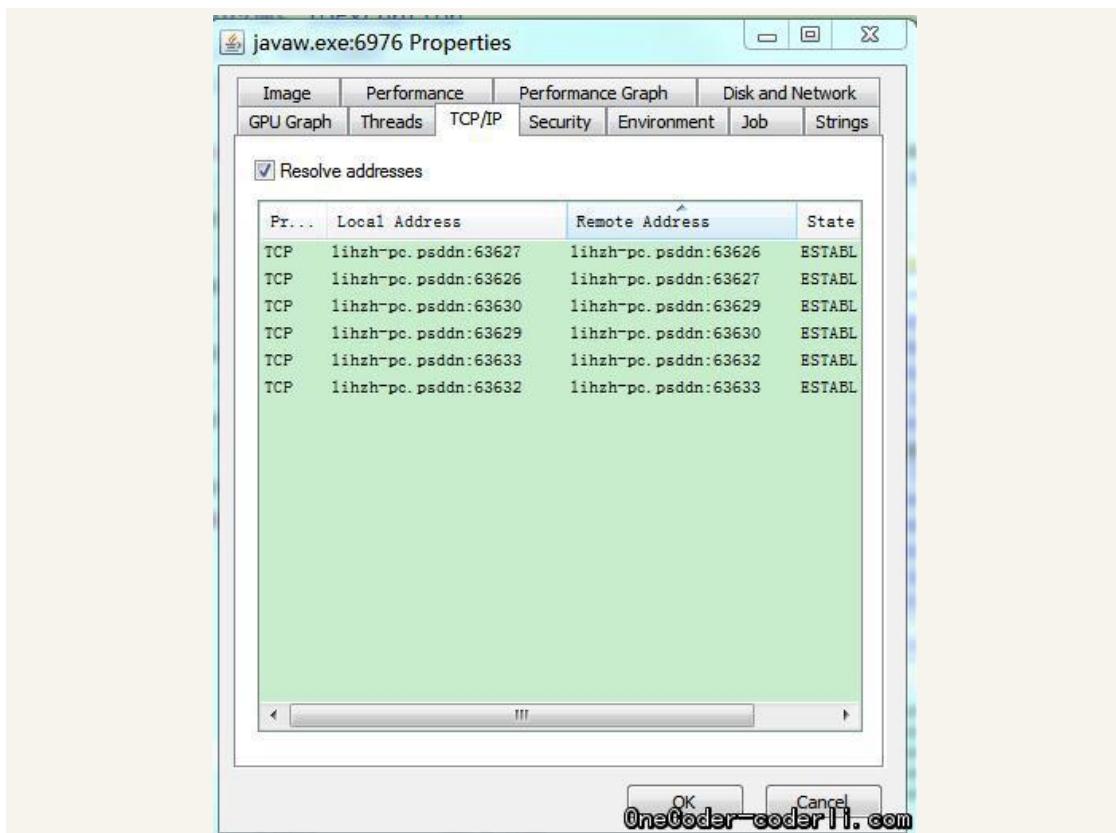
[?](#)

```

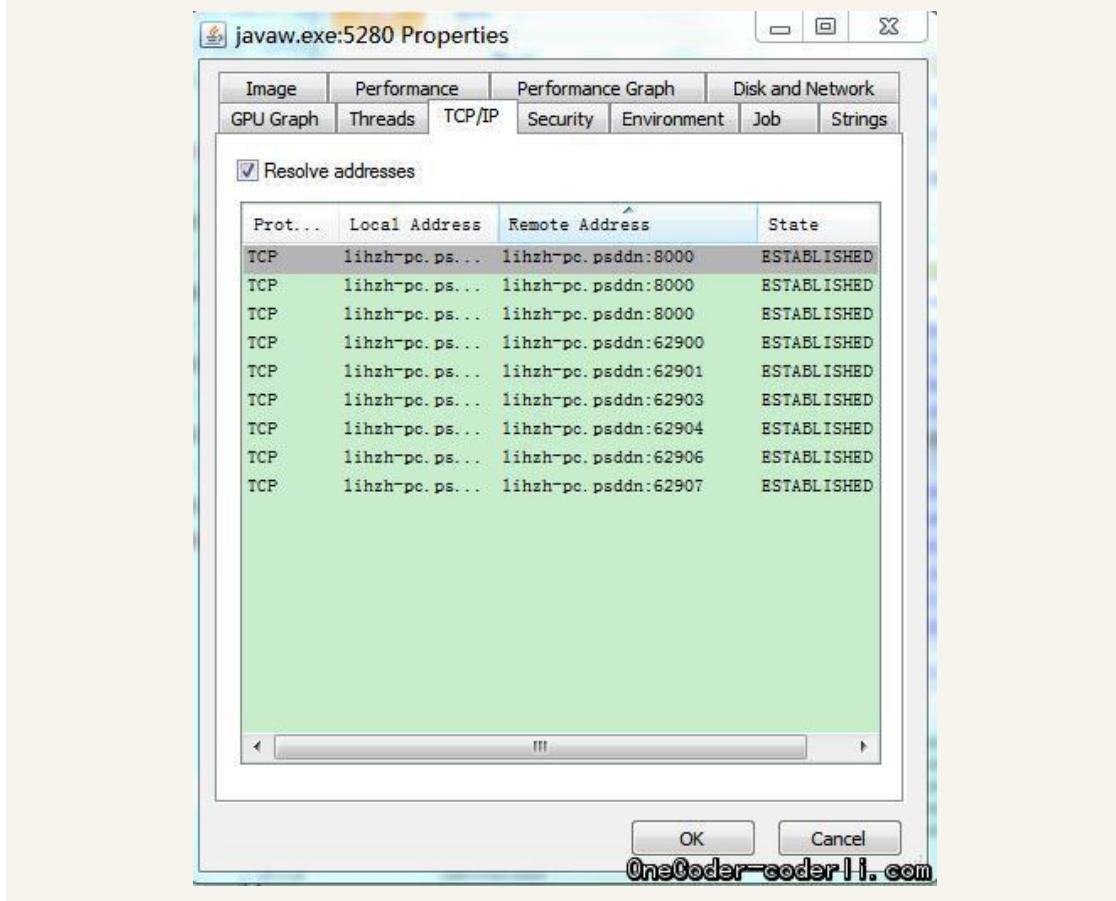
01 /**
02  * Selector 打开端口数量测试
```

```
03  *
04  * @author lihz
05  * @alia OneCoder
06  * @blog http://www.coderli.com
07  */
08 public class ConcurrencySelectorOpen {
09
10     /**
11      * @author lihz
12      * @alia OneCoder
13      * @throws IOException
14      * @throws InterruptedException
15      * @date 2012-8-15 下午 1:57:56
16      */
17     public static void main(String[] args) throws IOException, InterruptedException {
18         for (int i = 0; i < 3; i++) {
19             Selector selector = Selector.open();
20             SocketChannel channel = SocketChannel.open();
21             channel.configureBlocking(false);
22             channel.connect(new InetSocketAddress("127.0.0.1", 8000));
23             channel.register(selector, SelectionKey.OP_READ);
24         }
25         Thread.sleep(300000);
26     }
27 }
```

同样，通过 Process Explorer 去观察端口占用情况，开始跟酷壳大哥的观察到的效果一样。当不启动 Server 端，也就是不实际跟 Server 端建立链接的时候，3 次 selector open，占用了 6 个端口。

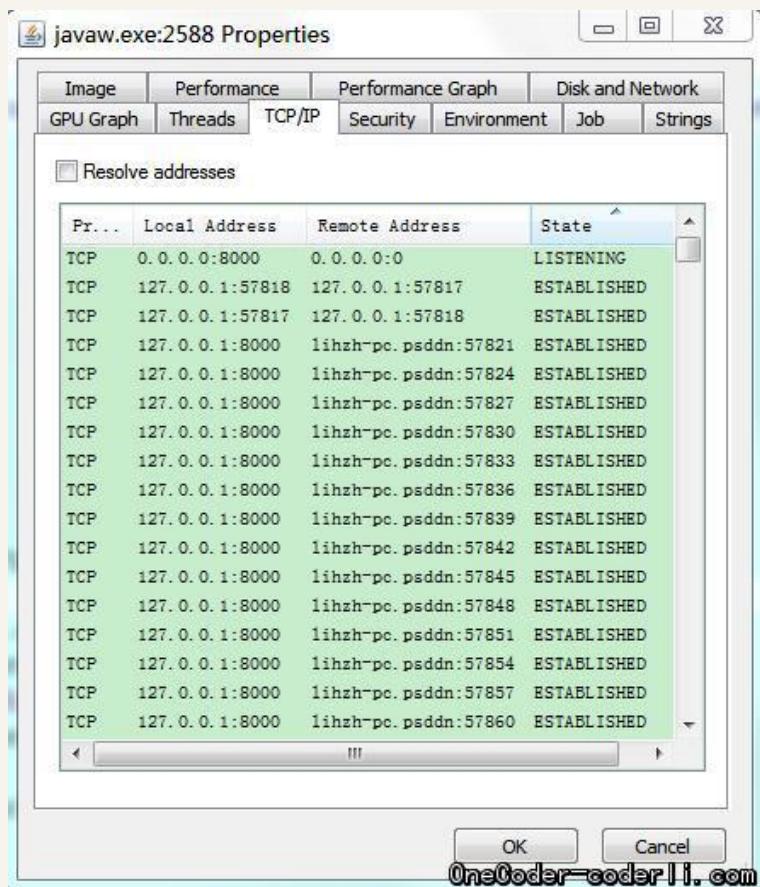


当启动 Server 端，进行绑定的时候，占用了 9 个端口



其实，多的三个，就是实际绑定的 8000 端口。其余就是酷壳大哥说的内部 Loop 链接。也就是说，对于我们实际场景，一次链接需要的端口数是 3 个。操作系统的端口数和资源当然有限的，也就是说当我们增大这个链接的时候，错误必然会发生。OneCoder 尝试一下 3000 次的场景，并没有出现错误，不过这么下去出错肯定是必然的。

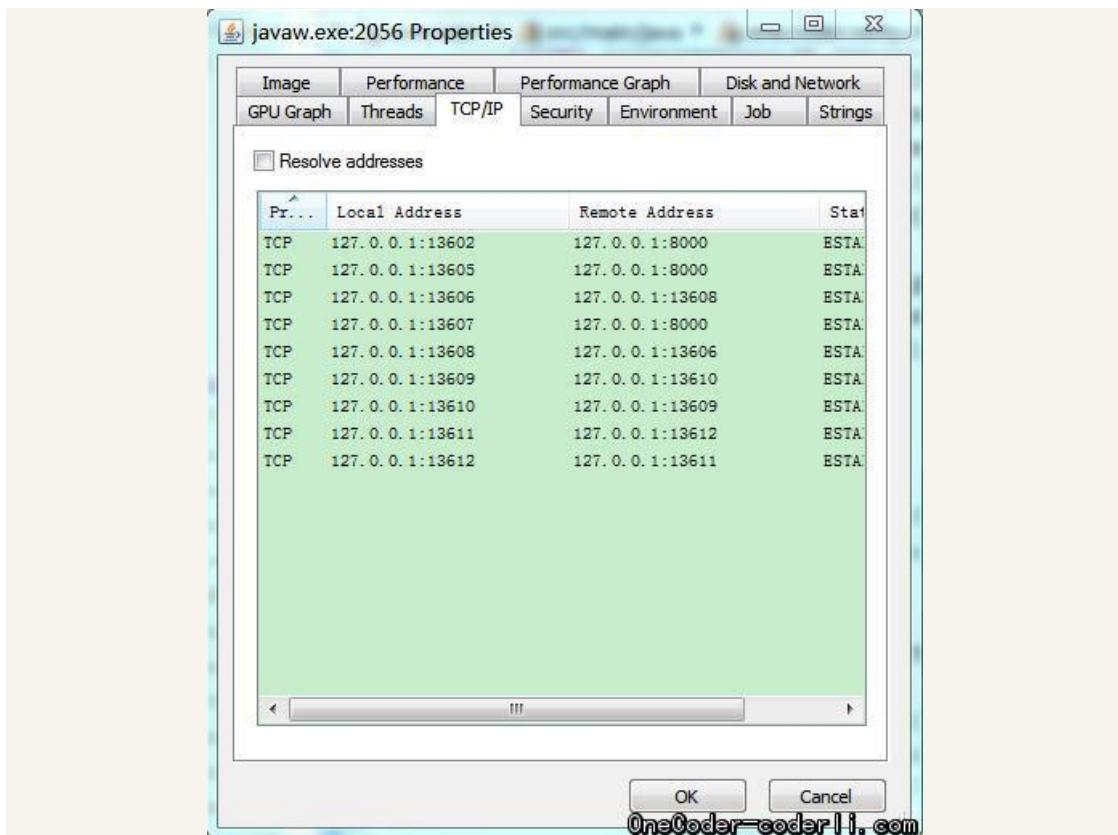
再看看服务端的情况，



这个还是直接通过 Selector 连接的时候的服务端的情况，除了两个内部回环端口以外，都是通过监听的 8000 的端口与外界通信，所以，服务端不会有端口限制的问题。不过，也可以看到，如果对链接不控制，服务端也维持大量的连接耗费系统资源！所以，良好的编码是多么的重要。

回到我们的主题，Netty 的场景。先使用跟 Selector 测试一样的场景，单线程，一个 bootstrap，连续多次 connect 看看错误和端口占用的情况。代码也就是开头提供的那段代码。

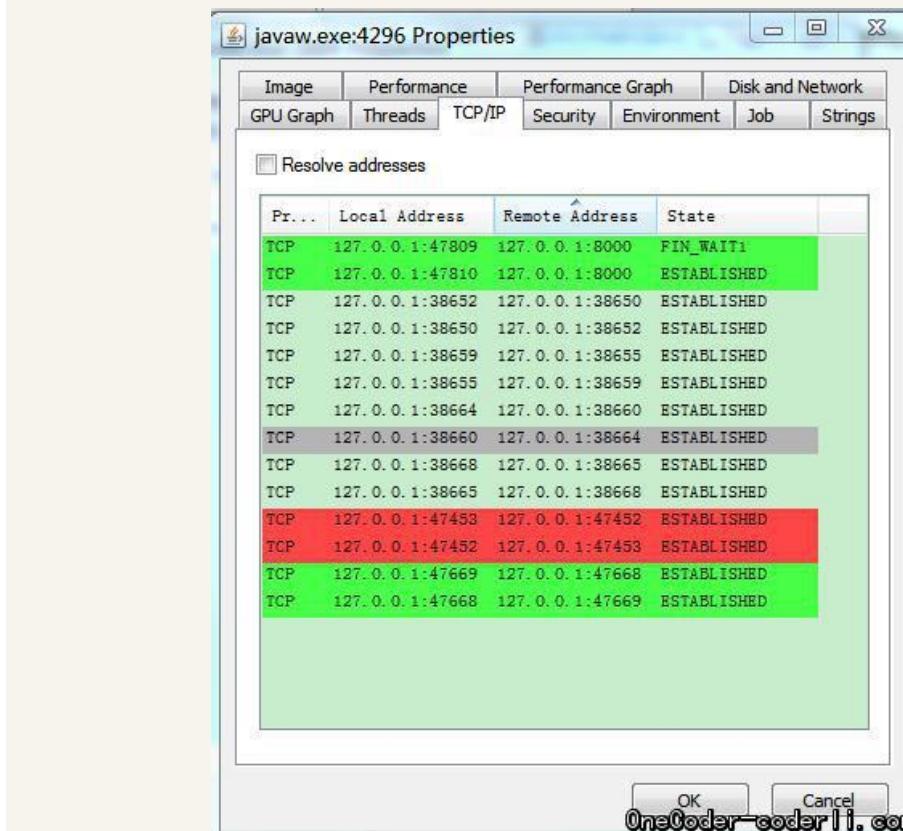
看看测试结果，



同样连接后还是占用 9 个端口。如果手动调用了 channel.close()方法，则会释放与 8000 链接的端口，也就是变成 6 个端口占用。

增大连续连接数到 10000。

首先没有报错，在每次 close channel 情况下，客户端端口占用情况如图（服务端情况类似）。



可见，并没有像 selector 那样无限的增加下去。这正是 Netty 中 worker 的巨大作用，帮我们管理这些琐碎的链接。具体分析，我们留待下章，您也可以自己研究一下。（[OneCoder](#)注：如果没关闭 channel，会发现对 8000 端口的占用，会迅速增加。系统会很卡。）调整测试代码，用一个线程来模拟一个客户端的请求。

[?](#)

```

01  /**
02   * Netty 并发，多 connection，socket 并发测试
03   *
04   * @author lihz (OneCoder)
05   * @alia OneCoder
06   * @Blog http://www.coderli.com
07   * @date 2012-8-13 下午 10:47:28
08   */
09  public class ConcurrencyNettyMultiConnection {
10
11      public static void main(String args[]) {
12          final ClientBootstrap bootstrap = new ClientBootstrap(
13              new NioClientSocketChannelFactory(
14                  Executors.newCachedThreadPool(),
15                  Executors.newCachedThreadPool()));
16          // 设置一个处理服务端消息和各种消息事件的类(Handler)
17          bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
18              @Override
19              public ChannelPipeline getPipeline() throws Exception {
20                  return Channels.pipeline(new ObjectEncoder(),
21                      new ObjectClientHandler()
22                  );
23              }
24          });
25          for (int i = 0; i < 1000; i++) {
26              // 连接到本地的 8000 端口的服务端
27              Thread t = new Thread(new Runnable() {
28                  @Override
29                  public void run() {
30                      bootstrap.connect(new
31                          InetSocketAddress("127.0.0.1", 8000));
32                      try {
33                          Thread.sleep(300000);
34                      } catch (InterruptedException e) {
35                          e.printStackTrace();
36                      }
37                  }
38              });
39              t.start();
40          }

```

```
41      }
}
```

不出所料，跟微博上问我问题的朋友出现的问题应该是一样的：

Write: 973

Write: 974

八月 17, 2012 9:57:28 下午 org.jboss.netty.channel.SimpleChannelHandler

警告: EXCEPTION, please implement one.coder.netty.chapter.eight.ObjectClientHandler.exceptionCaught() for proper handling.

java.net.ConnectException: Connection refused: no further information

这个问题，笔者确实尚未分析出原因，仅从端口占用情况来看，跟前面的测试用例跑出的效果基本类似。应该说明不是端口不足导致的，不过 [OneCoder](#) 尚不敢妄下结论。待研究后，我们下回分解吧：）您有任何想法，也可以提供给我，我来进行验证。

Java NIO 框架 Netty 教程(十三)-并发访问测试(下)

在上节([《Java NIO 框架 Netty 教程\(十二\)-并发访问测试\(中\)》](#))，我们从各个角度对 Netty 并发的场景进行了测试。这节，我们将重点关注上节最后提到的问题。在多线程并发访问的情况下，会出现

警告: EXCEPTION, please implement one.coder.netty.chapter.eight.ObjectClientHandler.exceptionCaught() for proper handling.

java.net.ConnectException: Connection refused: no further information

的错误警告。

之前 [OneCoder](#) 层怀疑过是端口数不够的问题，所以还准备了一套修改操作系统端口数限制的配置。

a) windows –

- 1、打开注册表: regedit
- 2、HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ Services\TCP/IP\Parameters
- 3、新建 DWORD 值, name: TcpTimedWaitDelay, value: 0 (十进制) -> 设置为 0
- 4、新建 DWORD 值, name: MaxUserPort, value: 65534 (十进制) -> 设置最大连接数 65534
- 5、重启系统

b) linux –

- 1、查看有关的选项 /sbin/sysctl -a|grep net.ipv4.tcp_tw_reuse

```
net.ipv4.tcp_tw_reuse = 0
```

#表示开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接，默认为 0，表示关闭;
- 2、修改 vi /etc/sysctl.conf

```
net.ipv4.tcp_tw_reuse = 1
```

```
net.ipv4.tcp_tw_recycle = 1
```

#表示开启 TCP 连接中 TIME-WAIT sockets 的快速回收，默认为 0，表示关闭

2、修改 vi /etc/sysctl.conf

```
net.ipv4.tcp_tw_reuse = 1
```

```
net.ipv4.tcp_tw_recycle = 1
```

3、使内核参数生效 sysctl -p

这套配置在测试 Restlet 框架并发的时候，起到了明显的效果。

然后，这次即使 [OneCoder](#) 修改配置，并发连接也没有明显的上升。[OneCoder](#) 决定换个思路，启动多个进程对同一个服务进行持续访问，以证明之前的连接拒绝就是因为客户端多线程并发自身的问题(其实 [OneCoder](#) 一直非常怀疑是这个问题)还是服务端连接处理的问题。

修改了一下客户端发动消息的代码，使其在其线程内部，不停的给服务端发送信息。

```
?  
01  /**  
02   * 发送 Object  
03   *  
04   * @param channel  
05   * @author lihz  
06   * @alisa OneCoder  
07   * @blog http://www.coderli.com  
08   */  
09  private void sendObject(final Channel channel) {  
10      new Thread(new Runnable() {  
11          @Override  
12          public void run() {  
13              // TODO Auto-generated method stub  
14              for (;;) {  
15                  Command command = new Command();  
16                  command.setActionName("Hello action.");  
17                  channel.write(command);  
18                  try {  
19                      Thread.sleep(1000);  
20                  } catch (InterruptedException e) {  
21                      e.printStackTrace();  
22                  }  
23              }  
24          }  
25      }).start();  
26  }  
27 }
```

启动多个客户端，效果如图：

	1 Server [Java Application] C:\Program Files\Java\jdk1.7.0_05\bin\javaw
	2 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	3 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	4 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	5 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	6 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	7 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	8 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	9 ConcurrencyNettyMultiConnection [Java Application] C:\Program File
	ConcurrencyNettyMultiConnection [Java Application] C:\Program Files\
	ConcurrencyNettyMultiConnection [Java Application] C:\Program Files\
	ConcurrencyNettyMultiConnection [Java Application] C:\Program Files\OneCoder-coderli.com

果然，在单个进程数量控制合理的情况下，服务端可以处理所有请求，不会出现链接拒绝的情况。总连接数轻松达到 4,5k。([OneCoder](#) 注：以前超过 1000 都容易出错。这里只是测试到以前完全没有办法支持的情形，并没有测试最大压力值。)

对于测试 Netty 服务端压力来说，这样的测试， [OneCoder](#) 认为完全可以起到效果，有参考价值。因为即使单进程网络连接方面无上限，单进程能启动的线程数也是有限制的，效率也一定会受到影响。所以，对于并发测试来说， [OneCoder](#) 认为可以采用上面的方式。

对于，单进程多线程的时，拒绝连接的问题。是在 sun.nio.ch.SocketChannelImpl 中的 native 方法 checkConnect 中抛出的。这应该是跟操作系统密切相关的。[OneCoder](#) 没有在 linux 下进行测试，但是猜测在 linux 下，上面的设置参数是可以起到作用的，也就是会比 windows 下可以开启的并发线程数多。当然这只是猜测。

对于 windows 来说，揪净能开启多个线程的并发，这个数据在 [OneCoder](#) 的环境下也是非常不稳定的。最开始的时候是，起 1000 个线程，成功的 350 个线程左右。后来， [OneCoder](#) 怀疑启动的程序过多，尤其是跟网络相关的程序会影响测试结果，关闭了很多程序。结果，多次 1000 线程都成功连接。

[OneCoder](#) 仔细排查了一下，猛然发现 [OneCoder](#) 使用了 Proxifier 这个代理。在代理打开的情况下，一般只能跑到 300 左右。关闭有 1000 个线程基本稳定通过。最多可以跑到 1500 左右。目前被列为最大“嫌疑人”

以 [OneCoder](#) 目前的知识构来说，Netty 并发的测试基本可以告一段落了。再简单的总结唠到几句：

1. 如果需要测试并发，可以考虑多进程，进程内多线程的方式测试服务端压力。
2. [OneCoder](#) 没有测试 Netty 最大可以支持多少并发，因为从目前测试的效果来看。（5 个进程，每个进程 1000 线程，持续访问同一个服务），已经完全可以满足 [OneCoder](#) 的要求了。您也可以继续测试下去。
3. [OneCoder](#) 使用的是 windows7 32 位操作系统，在测试过程其实也修改了注册表中的若干参数，包括上面提到的两个。不知道是否起到了一定的作用，也就是是否使单进程可以支持的多线程数增加，或者服务端可以支持的连接数增加，您在测试的过程中，可以配合考虑这些参数。
4. 对于，connection refuse 的具体原因，[OneCoder](#) 希望能随着自己知识的慢慢积累，找到其真正的答案。

Java NIO 框架 Netty 教程(十四)-Netty 中 OIO 模型(对比 NIO)

[OneCoder](#) 这个周末搬家，并且新家目前还没有网络，本周的翻译的任务尚未完成，下周一起补上，先上一篇 OIO 和 NIO 对比的小研究。

Netty 中不光支持了 Java 中 NIO 模型，同时也提供了对 OIO 模型的支持。（New IO vs Old IO）。

首先，在 Netty 中，切换 OIO 和 NIO 两种模式是非常方便的，只需要初始化不同的 Channel 工程即可。

?

```
1ServerBootstrap bootstrap = new ServerBootstrap()  
2new OioServerSocketChannelFactory(  
3Executors.newCachedThreadPool(),  
4Executors.newFixedThreadPool(4));
```

?

```

1ServerBootstrap bootstrap = new ServerBootstrap(
2new NioServerSocketChannelFactory(
3Executors.newCachedThreadPool(),
4Executors.newFixedThreadPool(4)));

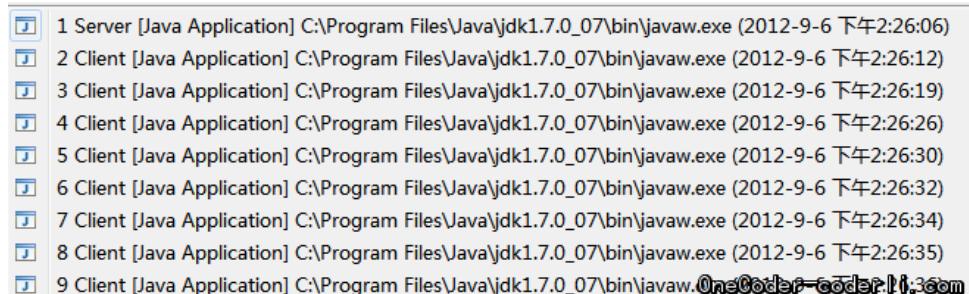
```

这就是 Netty 框架为我们做的贡献。

再说说，这两种情况的区别。OneCoder 根据网上的资料和自己的理解，总结了一下：在 Netty 中，是通过 worker 执行任务的。也就是我们在构造 bootstrap 时传入的 worker 线程池。对于传统 OI0 来说，一个 worker 对应的 channel，从读到操作再到回写，只要是这个 channel 的操作都通过这个 worker 来完成，对于 NIO 来说，到 MessageReceived 之后，该 worker 的任务就完成了。所以，从这个角度来说，非常建议你在 Recieve 之后，立即启动线程去执行耗时逻辑，以释放 worker。

基于这个分析，你可能也发现了，上面的代码中我们用的是 FixedThreadPool。固定大小为 4，从理论上来说，OI0 支持的 client 数应该是 4。而 NIO 应该不受此影响。测试效果如下图：

8 个 Client 连接：



```

1 Server [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:06)
2 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:12)
3 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:19)
4 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:26)
5 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:30)
6 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:32)
7 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:34)
8 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:35)
9 Client [Java Application] C:\Program Files\Java\jdk1.7.0_07\bin\javaw.exe (2012-9-6 下午2:26:35)

```

NIOServer 的线程情况：

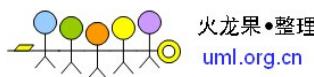


Server [Java Application]

- one.coder.netty.chapter.common.Server at localhost:53639
 - Thread [New I/O server boss #1 ([id: 0x016b4be5, /0.0.0.0:8000])] (Running)
 - Thread [DestroyJavaVM] (Running)
 - Thread [New I/O worker #1] (Running)
 - Thread [New I/O worker #2] (Running)
 - Thread [New I/O worker #3] (Running)
 - Thread [New I/O worker #4] (Running)

并且 8 个 Client 的请求都正常处理了。

对于 OI0 来说，如果你对 worker 池没有控制，那么支持 8 个 client 需要 8 个 worker，8 个线程，这也就是传统 OI0 并发数受限的原因，如图：



当 OIO 使用 FixedThreadPool 的时候：

- one.coder.netty.chapter.fourteen.OioServer at localhost:53968
 - Thread [DestroyJavaVM] (Running)
 - Thread [Old I/O server boss ([id: 0x01f84d01, 0.0.0/0.0.0:8000])] (Running)
 - Thread [Old I/O server worker (parentId: 33049857, [id: 0x01f84d01, 0.0.0/0.0.0:8000])] (Running)
 - Thread [Old I/O server worker (parentId: 33049857, [id: 0x01f84d01, 0.0.0/0.0.0:8000])] (Running)
 - Thread [Old I/O server worker (parentId: 33049857, [id: 0x01f84d01, 0.0.0/0.0.0:8000])] (Running)
 - Thread [Old I/O server worker (parentId: 33049857, [id: 0x01f84d01, 0.0.0/0.0.0:8000])] (Running)

只能处理头四个 client 的请求，其他的被堵塞了。

Hello action. : 32

Hello action. : 33

Hello action. : 34

Hello action. : 35

Hello action. : 36

Hello action. : 37

Hello action. : 38

Hello action. : 39

Hello action. : 40

但是，在 Netty 框架

Java NIO 框架 Netty 教程(十五)-利用 Netty 进行文件传输

如果您持续关注 [OneCoder](#), 您可能会问, 在[《Java NIO 框架 Netty 教程\(十四\)- Netty 中 OIO 模型\(对比 NIO\)》](#)中不是说下节介绍的是, NIO 和 OIO 中的 worker 处理方式吗。这个一定会有的, 只是在研究的过程中, [OneCoder](#)发现了之前遗留的文件传输的代码, 所以决定先完成它。

其实, Netty 的样例代码中也提供了文件上传下载的代码样例, 不过太过复杂, 还包括了 Http 请求的解析等, 对 [OneCoder](#)来说, 容易迷惑那些才是文件传输的关键部分。所以 [OneCoder](#)决定根据自己去写一个样例, 这个理解就是在最开始提到的, Netty 的传输是基于流的, 我们把文件流化应该就可以传递了。于是有了以下的代码:

```
?  
01  /**  
02   * 文件传输接收端, 没有处理文件发送结束关闭流的情景  
03   *  
04   * @author lihz  
05   * @alisa OneCoder  
06   * @blog http://www.coderli.com  
07   */  
08  public class FileServerHandler extends SimpleChannelHandler {  
09  
10      private File file = new File("F:/2.txt");  
11      private FileOutputStream fos;  
12  
13      public FileServerHandler() {  
14          try {  
15              if (!file.exists()) {  
16                  file.createNewFile();  
17              } else {  
18                  file.delete();  
19                  file.createNewFile();  
20              }  
21              fos = new FileOutputStream(file);  
22          } catch (IOException e) {  
23              e.printStackTrace();  
24          }  
25      }  
26  
27      @Override  
28      public void messageReceived(ChannelHandlerContext ctx, MessageEvent e)  
29          throws Exception {  
30          ChannelBuffer buffer = (ChannelBuffer) e.getMessage();  
31          int length = buffer.readableBytes();  
32          buffer.readBytes(fos, length);  
33          fos.flush();  
34          buffer.clear();  
35      }
```

36

37 }

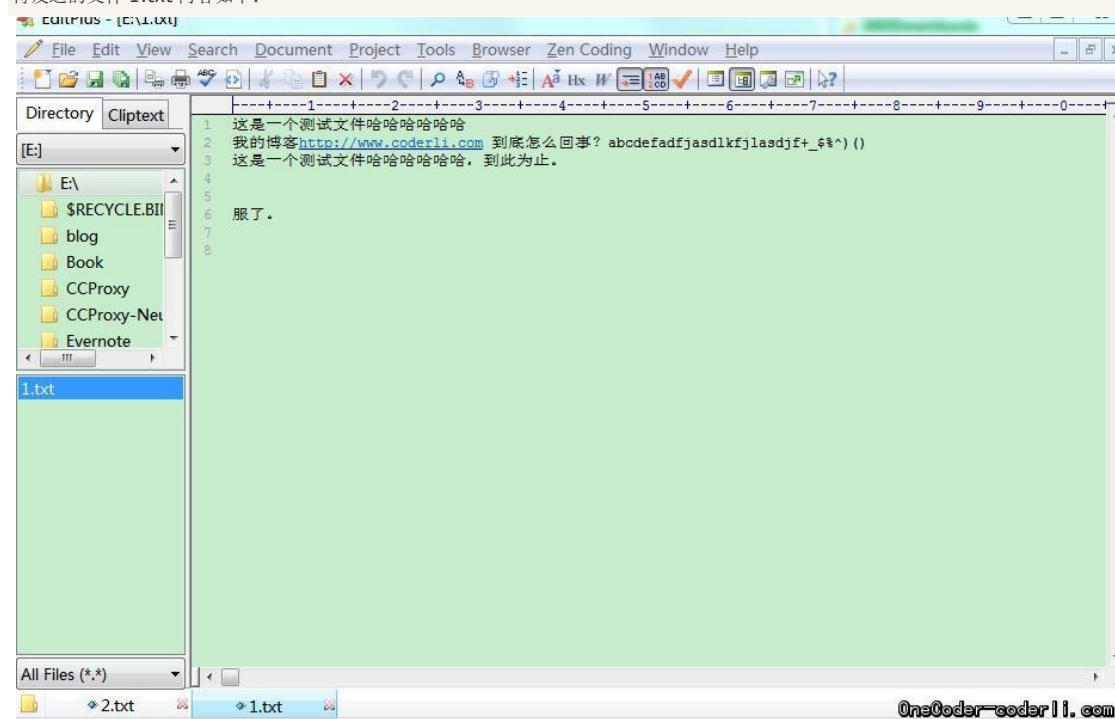


```
01  /**
02   * 文件发送客户端，通过字节流来发送文件，仅实现文件传输部分，<br>
03   * 没有对文件传输结束进行处理<br>
04   * 应该发送文件发送结束标识，供接受端关闭流。
05   *
06   * @author lihz
07   * @alia OneCoder
08   * @blog http://www.coderli.com
09   */
10  public class FileClientHandler extends SimpleChannelHandler {
11
12      // 每次处理的字节数
13      private int readLength = 8;
14
15      @Override
16      public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e)
17          throws Exception {
18          // 发送文件
19          sendFile(e.getChannel());
20      }
21
22      private void sendFile(Channel channel) throws IOException {
23          File file = new File("E:/1.txt");
24          FileInputStream fis = new FileInputStream(file);
25          int count = 0;
26          for (;;) {
27              BufferedInputStream bis = new BufferedInputStream(fis);
28              byte[] bytes = new byte[readLength];
29              int readNum = bis.read(bytes, 0, readLength);
30              if (readNum == -1) {
31                  return;
32              }
33              sendToServer(bytes, channel, readNum);
34              System.out.println("Send count: " + ++count);
35          }
36
37      }
38
39      private void sendToServer(byte[] bytes, Channel channel, int length)
40          throws IOException {
```

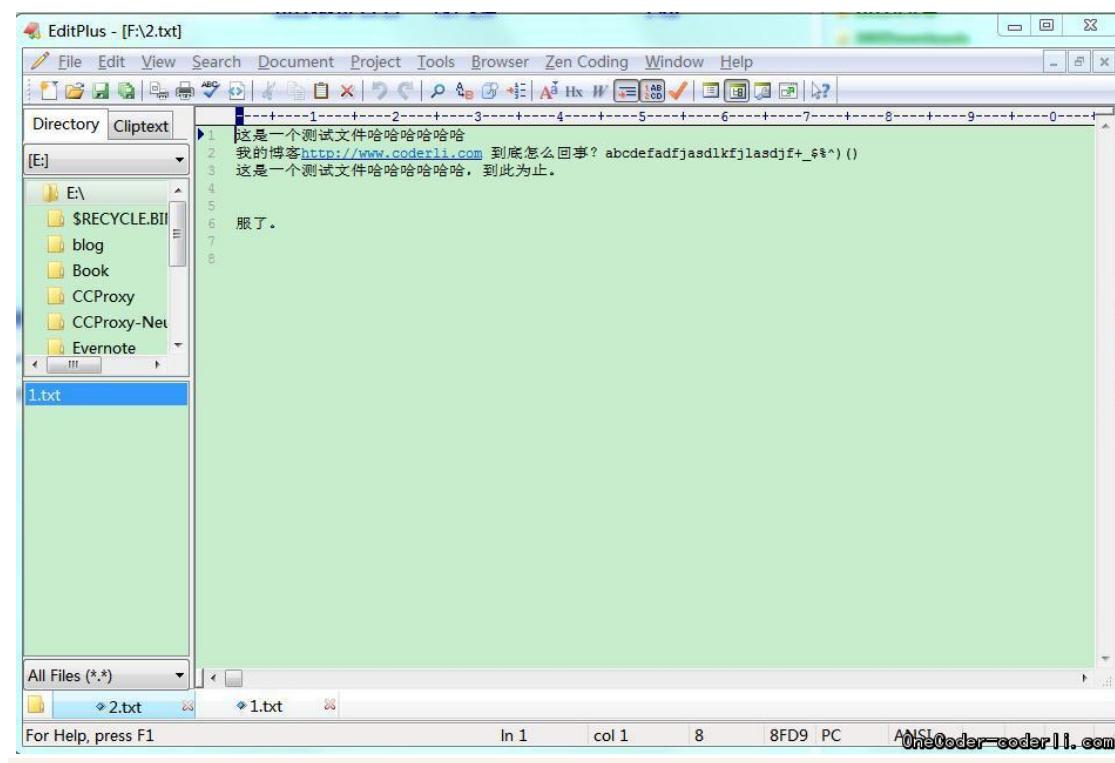
```

41         ChannelBuffer buffer = ChannelBuffers.copiedBuffer(bytes, 0, length);
42         channel.write(buffer);
43     }
44
45 }
```

待发送的文件 1.txt 内容如下：



运行上述代码，接收到的文件 2.txt 结果：



完全一模一样。成功！

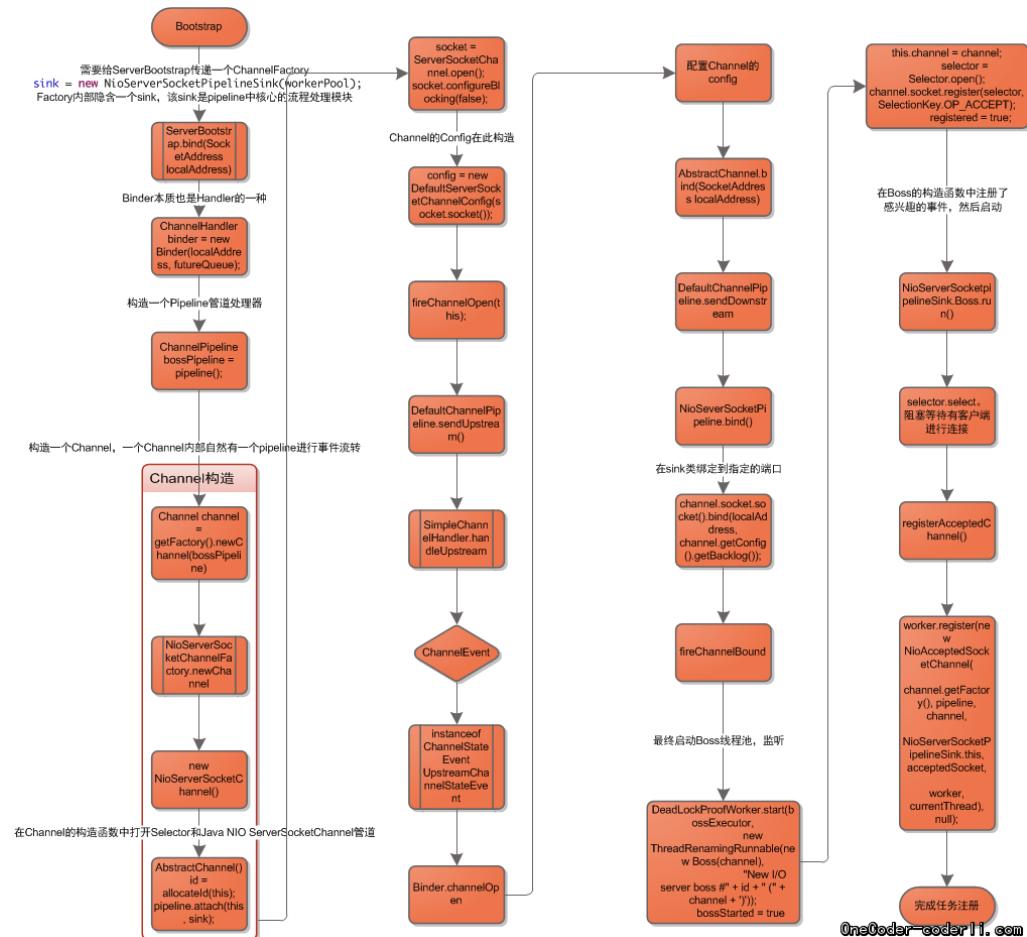
这只是一个简单的文件传输的例子，可以做为样例借鉴。对于大文件传输的情景，本样例并不支持，会出现内存溢出的情景，OneCoder准备另外单独介绍一下。

Java NIO 框架 Netty 教程(十六)-ServerBootStrap 启动流程源码分析

有一段事件没有更新文章了，各种原因都有吧。搬家的琐事，搬家后的安逸呵呵。不过，[OneCoder](#)明白，绝不能放松。对于 Netty 的学习，也该稍微深入一点了。

所以，这次[OneCoder](#)花了几段时间，仔细梳理了一下 Netty 的源码，总结了一下 ServerBootStrap 的启动和任务处理流程，基本涵盖了 Netty 的关键架构。

[OneCoder](#)总结了一张流程图：



该图是 [OneCoder](#) 通过阅读 Netty 源码，逐渐记录下来的。基本可以说明 Netty 服务的启动流程。这里在具体讲解一下。

首先说明，我们这次顺利的流程是基于 NioSocketServer 的。也就是基于 Java NIO Selector 的实现方式。在第六讲[《Java NIO 框架 Netty 教程\(六\)-Java NIO Selector 模式》](#)中，我们已经知道使用 Selector 的几个关键点，所以这里我们也重点关注一下，这些点在 Netty 中是如何使用的。

很多看过 Netty 源码的人都说 Netty 源码写的很漂亮。可漂亮在哪呢？Netty 通过一个 ChannelFactory 决定了你当前使用的协议类型 (Nio/nio 等)，比如，OneCoder 这里使用的是 NioServerSocket，那么需要声明的 Factory 即为 NioServerSocketChannelFactory，声明了这个 Factory，就决定了你使用的 Channel，pipeline 以及 pipeline 中，具体处理业务的 sink 的类型。这种使用方式十分简洁的，学习曲线很低，切换实现十分容易。

Netty 采用的是基于事件的管道式架构设计，事件(Event)在管道(Pipeline)中流转，交由(通过 pipelinesink)相应的处理器(Handler)。这些关键元素类型的匹配都是由开始声明的 ChannelFactory 决定的。

Netty 框架内部的业务也遵循这个流程，Server 端绑定端口的 binder 其实也是一个 Handler，在构造完 Binder 后，便要声明一个 Pipeline 管道，并赋给新建一个 Channel。Netty 在 newChannel 的过程中，相应调用了 Java 中的 ServerSocketChannel.open 方法，打开一个 channel。然后触发 fireChannelOpen 事件。这个事件的接受是可以复写的。Binder 自身接收了这个事件。在事件的处理中，继续向下完成具体的端口的绑定。对应 Selector 中的 socketChannel.socket().bind()。然后触发 fireChannelBound 事件。默认情况下，该事件无人接受，继续向下开始构造 Boss 线程池。我们知道在 Netty 中 Boss 线程池是用来接受和分发请求的核心线程池。所以在 channel 绑定后，必然要启动 Boss 线程池，随时准备接受 client 发来的请求。在 Boss 构造函数中，第一次注册了 selector 感兴趣的事件类型，SelectionKey.OP_ACCEPT。至此，在第六讲中介绍的使用 Selector 的几个关键步骤都体现在 Netty 中了。在 Boss 回启动一个死循环来查询是否有感兴趣的事件发生，对于第一次的客户端的注册事件，Boss 会将 Channel 注册给 worker 保存。

这里补充一下，也是图中忽略的部分，就是关于 worker 线程池的初始化时机问题。worker 池的构造，在最开始构造 ChannelFactory 的时候就已经准备好了。在 NioServerSocketChannelFactory 的构造函数里，会 new 一个 NioWorkerPool。在 NioWorkerPool 的基类 AbstractNioWorkerPool 的构造函数中，会调用 OpenSelector 方法，其中也打开了一个 selector，并且启动了 worker 线程池。

?

```

01  private void openSelector() {
02
03      try {
04          selector = Selector.open();
05      } catch (Throwable t) {
06          throw new ChannelException("Failed to create a selector.", t);
07      }
08
09      // Start the worker thread with the new Selector.
10      boolean success = false;
11
12      try {
13          DeadLockProofWorker.start(executor, new ThreadRenamingRunnable(this, "New I/O    worker #" + id));
14          success = true;
15      } finally {
16          if (!success) {
17              // Release the Selector if the execution fails.
18              try {
19                  selector.close();
20              } catch (IOException e) {
21                  logger.error("Failed to close the Selector.", e);
22              }
23          }
24      }
25  }

```

```

17             selector.close();
18         } catch (Throwable t) {
19             logger.warn("Failed to close a selector.", t);
20         }
21         selector = null;
22         // The method will return to the caller at this point.
23     }
24 }
25 assert selector != null && selector.isOpen();
26 }
```

至此，会分线程启动 `AbstractNioWorker` 中 `run` 逻辑。同样是循环处理任务队列。

2

```

1 processRegisterTaskQueue();
2 processEventQueue();
3 processWriteTaskQueue();
4 processSelectedKeys(selector.selectedKeys());
```

这样，设计使事件的接收和处理模块完全解耦。

由此可见，如果你想从 `nio` 切换到 `oio`，只需要构造不同的 `ChannelFacotry` 即可。果然简洁优雅

Netty 学习笔记（实验篇）二

7 人收藏此文章, [我要收藏](#) 发表于 5 个月前(2013-06-16 17:50), 已有 346 次阅读, 共 4 个评论

这次实验要用 netty 实现一个 EchoServer , Echo Protocol 的定义在这里 <http://tools.ietf.org/html/rfc862>

1.服务器端监听端口 7 (由于 Linux 下普通用户无法使用 1024 以下的端口，因此绑定 7777)

2.客户端连接服务器 后发送一条数据

3.服务器把接收到的数据直接返回给客户端

代码如下：

```

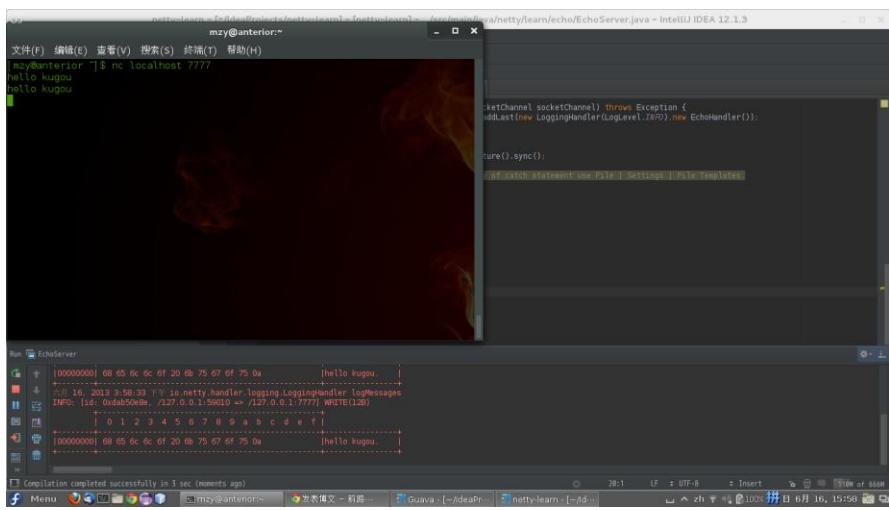
01 package netty.learn.echo;
02
03 import io.netty.bootstrap.ServerBootstrap;
04 import io.netty.channel.*;
05 import io.netty.channel.nio.NioEventLoopGroup;
06 import io.netty.channel.socket.SocketChannel;
07 import io.netty.channel.socket.nio.NioServerSocketChannel;
08 import io.netty.handler.logging.LogLevel;
09 import io.netty.handler.logging.LoggingHandler;
```

```
10
11 import java.util.logging.Level;
12 import java.util.logging.Logger;
13
14 /**
15  * This is a EchoServer implements the ECHO protocol
16  * User: mzy
17  * Date: 13-6-16
18  * Time: 下午 3:13
19  * Version:1.0.0
20 */
21 class EchoHandler extends ChannelInboundHandlerAdapter {
22     Logger log = Logger.getLogger(EchoHandler.class.getName());
23
24     @Override
25     public void messageReceived(ChannelHandlerContext ctx, MessageList<Object> msgs) throws Exception {
26         ctx.write(msgs);
27     }
28
29     @Override
30     public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
31         log.log(Level.WARNING, cause.getMessage());
32         ctx.close();
33     }
34 }
35 public class EchoServer {
36     private int port;
37
38     public EchoServer(int port) {
39         this.port = port;
40     }
41     public void run() {
42         NioEventLoopGroup boss = new NioEventLoopGroup();
43         NioEventLoopGroup worker = new NioEventLoopGroup();
44         ServerBootstrap b = new ServerBootstrap();
45         b.group(boss, worker).channel(NioServerSocketChannel.class)
```

```
46         .option(ChannelOption.SO_BACKLOG, 100)
47
48     .childHandler(new ChannelInitializer<SocketChannel>() {
49
50         @Override
51
52         protected void initChannel(SocketChannel socketChannel) throws Exception {
53
54             socketChannel.pipeline().addLast(new LoggingHandler(LogLevel.INFO), new EchoHan
55             dler());
56
57         }
58
59     });
60
61
62     try {
63
64         b.bind(port).sync().channel().closeFuture().sync();
65
66     } catch (InterruptedException e) {
67         e.printStackTrace(); //To change body of catch statement use File | Settings | File Templates.
68
69     } finally {
70
71         boss.shutdownGracefully();
72
73         worker.shutdownGracefully();
74
75     }
76
77 }
78
79
80 public static void main(String[] args) {
81
82     new EchoServer(7777).run();
83
84 }
85
86 }
```

这样服务器端就写好了，对比实验一中的 DiscardServer 不难发现基本上除了 Handler 之外没有什么变化，这样 TimeServer 中的代码其实可以做一个模板来使用。

下面我们用 nc 来测试一下 EchoServer



The screenshot shows the IntelliJ IDEA interface with the EchoServer.java file open. The code implements a simple echo server using Netty. In the terminal window, a connection is established via netcat to port 7777, and the message "Hello Kugou" is sent, followed by a response from the server.

```
mzy@anterior:~/Desktop/netty/learn$ nc localhost 7777
Hello Kugou
Hello Kugou

mzy@anterior:~/Desktop/netty/learn$
```

```
private void initChannel(SocketChannel socketChannel) throws Exception {
    socketChannel.pipeline().addLast(new LoggingHandler(LogLevel.INFO),new EchoHandler());
}

public void run() throws Exception {
    EventLoopGroup bossGroup = new NioEventLoopGroup();
    EventLoopGroup workerGroup = new NioEventLoopGroup();
    try {
        ServerBootstrap b = new ServerBootstrap();
        b.group(bossGroup, workerGroup)
        .channel(NioSocketChannel.class)
        .handler(new LoggingHandler(LogLevel.INFO))
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel socketChannel) throws Exception {
                socketChannel.pipeline().addLast(new LoggingHandler(LogLevel.INFO),new EchoHandler());
            }
        })
        .option(ChannelOption.SO_BACKLOG, 128)
        .option(ChannelOption.TCP_NODELAY, true);
        b.bind(7777).sync();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}
```

我们通过打印出的日志可以看到，服务器端先接收到了 Hello Kugou 然后又直接写回给了客户端。