Netty 代码分析

Posted by bucketli on 2010-09-25 Leave a comment (1) Go to comments

Netty 提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序[官方定义]，整体来看其包含了以下内容：1.提供了丰富的协议编解码支持，2.实现自有的 buffer 系统，减少复制所带来的消耗，3.整套 channel 的实现，4.基于事件的过程流转以及完整的网络事件响应与扩展，5.丰富的 example。本文并不对 Netty 实际使用中可能出现的问题做分析，只是从代码角度分析它的架构以及实现上的一些关键细节。

首先来看下最如何使用 Netty(其自带 example 很好展示了使用)，Netty 普通使用一般是通过 BootStrap 来启动，BootStrap 主要分为两类：1.面向连接(TCP)的 BootStrap(ClientBootStrap 和 ServerBootstrap),2.非面向连接(UDP) 的(ConnectionlessBootstrap)。

Netty 整体架构很清晰的分成 2 个部分，ChannelFactory 和 ChannelPipelineFactory,前者主要生产网络通信相关的 Channel 实例和 ChannelSink 实例，Netty 提供的 ChannelFactory 实现基本能够满足绝大部分用户的需求，当然你也可以定制自己的 ChannelFactory,后者主要关注于具体传输数据的处理，同时也包括其他方面的内容，比如异常处理等等，只要是你希望的,你都可以往里添加相应的 handler,一般 ChannelPipelineFactory 由用户自己实现，因为传输数据的处理及其他操作和业务关联比较紧密，需要自定义处理的 handler。

现在，使用 Netty 的步骤实际上已经非常明确了，比如面向连接的 Netty 服务端客户端使用，第一步：实例化一个 BootStrap,并且通过构造方法指定一个 ChannelFactory 实现，第二步：向 bootstrap 实例注册一个自己实现的 ChannelPipelineFactory,第三步：如果是服务器端，bootstrap.bind(new InetSocketAddress(port))，然后等待客户端来连接,如果是客户端，bootstrap.connect(new InetSocketAddress(host,port))取得一个 future,这个时候 Netty 会去连接远程主机，在连接完成后，会发起类型为 CONNECTED 的 ChannelStateEvent，并且开始在你自定义的 Pipeline 里面流转，如果你注册的 handler 有这个事件的响应方法的话那么就会调用到这个方法。在此之后就是数据的传输了。下面是一个简单客户端的代码解读。

```
// 实例化一个客户端 Bootstrap 实例，其中 NioClientSocketChannelFactory 实例由 Netty 提供
        ClientBootstrap bootstrap = new ClientBootstrap(
                new NioClientSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));
```

```
        // 设置 PipelineFactory,由客户端自己实现
        bootstrap.setPipelineFactory(new
FactorialClientPipelineFactory(count));

        //向目标地址发起一个连接
        ChannelFuture connectFuture =
            bootstrap.connect(new InetSocketAddress(host, port));

        // 等待链接成功，成功后发起的 connected 事件将会使 handler 开始
发送信息并且等待 messageRecive,当然这只是示例。
        Channel channel =
connectFuture.awaitUninterruptibly().getChannel();

        // 得到用户自定义的 handler
        FactorialClientHandler handler =
            (FactorialClientHandler) channel.getPipeline().getLast();

        // 从 handler 里面取数据并且打印，这里需要注意的是，
handler.getFactorial 使用了从结果队列 result take 数据的阻塞方法，而结
果队列会在 messageRecieve 事件发生时被填充接收回来的数据
        System.err.format(
                "Factorial of %,d is: %,d", count,
handler.getFactorial());
```

Netty 提供了 NIO 与 BIO(OIO)两种模式处理这些逻辑，其中 NIO 主要通过一个 BOSS 线程处理等待链接的接入，若干个 WORKER 线程(从 worker 线程池中挑选一个赋给 Channel 实例，因为 Channel 实例持有真正的 java 网络对象)接过 BOSS 线程递交过来的 CHANNEL 进行数据读写并且触发相应事件传递给 pipeline 进行数据处理，而 BIO(OIO)方式服务器端虽然还是通过一个 BOSS 线程来处理等待链接的接入，但是客户端是由主线程直接 connect,另外写数据 C/S 两端都是直接主线程写，而数据读操作是通过一个 WORKER 线程 BLOCK 方式读取(一直等待，直到读到数据，除非 channel 关闭)。

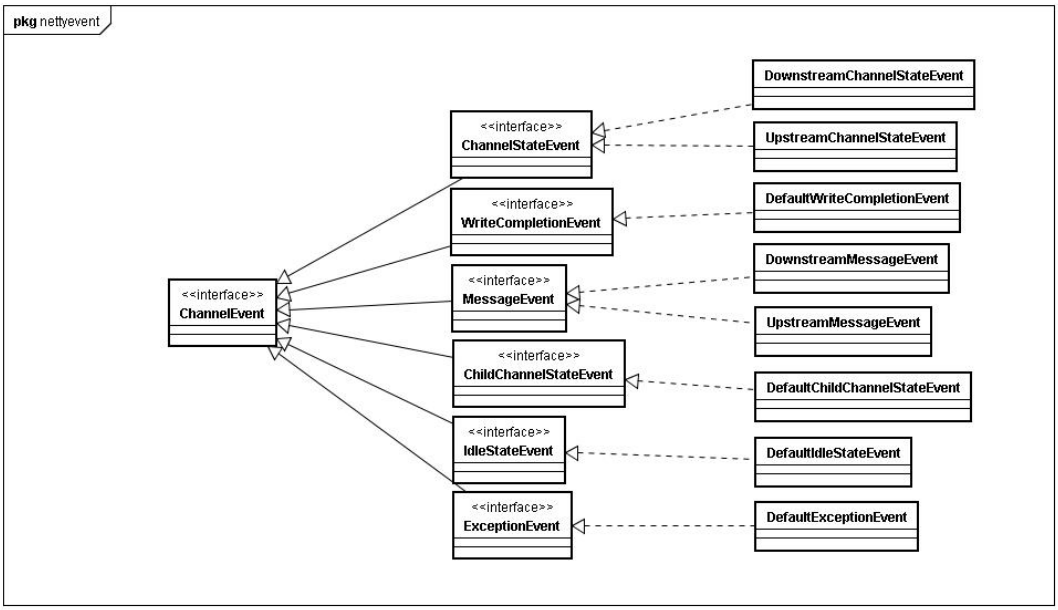
网络动作归结到最简单就是服务器端 bind->accept->read->write,客户端 connect->read->write,一般 bind 或者 connect 后会有多次 read、write。这种特性导致，bind,accept 与 read,write 的线程分离，connect 与 read、write 线程分离，这样做的好处就是无论是服务器端还是客户端吞吐量将有效增大，以便充分利用机器的处理能力，而不是卡在网络连接上，不过一旦机器处理能力充

分利用后，这种方式反而可能会因为过于频繁的线程切换导致性能损失而得不偿失，并且这种处理模型复杂度比较高。

采用什么样的网络事件响应处理机制对于网络吞吐量是非常重要的，Netty 采用的是标准的 SEDA（Staged Event-Driven Architecture）架构[http://en.wikipedia.org/wiki/ Staged_event-driven_architecture]，其所设计的事件类型，代表了网络交互的各个阶段，并且在每个阶段发生时，触发相应事件交给初始化时生成的 pipeline 实例进行处理。事件处理都是通过 Channels 类的静态方法调用开始的，将事件、channel 传递给 channel 持有的 Pipeline 进行处理，Channels 类几乎所有方法都为静态，提供一种 Proxy 的效果(整个工程里无论何时何地都可以调用其静态方法触发固定的事件流转,但其本身并不关注具体的处理流程)。

Channels 部分事件流转静态方法
1. fireChannelOpen 2. fireChannelBound 3. fireChannelConnected
4.fireMessageReceived 5.fireWriteComplete 6.fireChannelInterestChanged
7. fireChannelDisconnected 8. fireChannelUnbound 9. fireChannelClosed
10.fireExceptionCaught 11.fireChildChannelStateChanged

Netty 提供了全面而又丰富的网络事件类型，其将 java 中的网络事件分为了两种类型 Upstream 和 Downstream。一般来说，Upstream 类型的事件主要是由网络底层反馈给 Netty 的，比如 messageReceived,channelConnected 等事件，而 Downstream 类型的事件是由框架自己发起的，比如 bind,write,connect,close 等事件。



Netty 的 Upstream 和 Downstream 网络事件类型特性也使一个 Handler 分为了 3 种类型，专门处理 Upstream,专门处理 Downstream,同时处理 Upstream,Downstream。实现方式是某个具体 Handler 通过继承 ChannelUpstreamHandler 和 ChannelDownstreamHandler 类来进行区分。
PipeLine 在 Downstream 或者 Upstream 类型的网络事件发生时，会调用匹配事

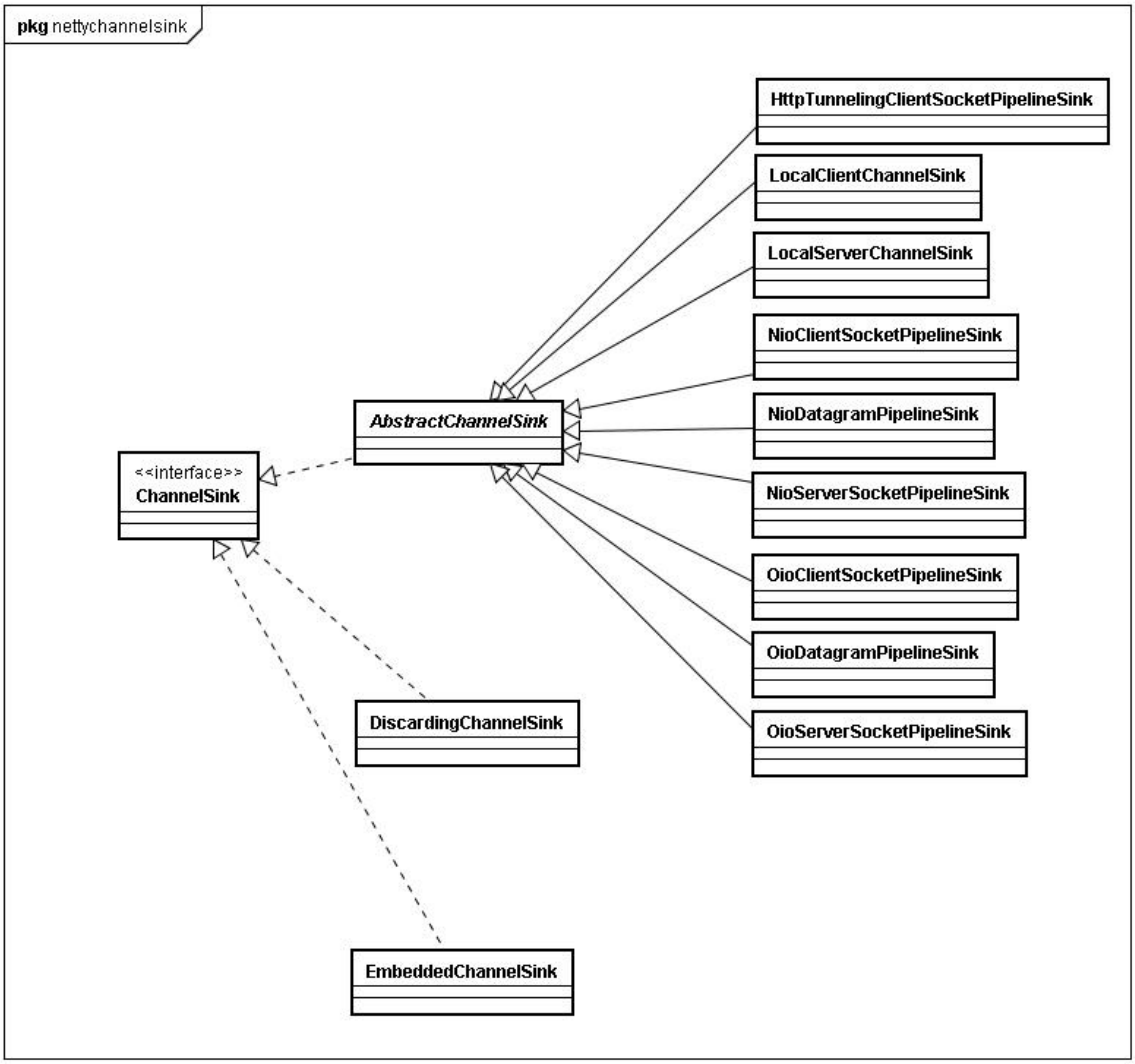
件类型的 Handler 响应这种调用。ChannelPipeline 维持有所有 handler 有序链表，并且由 handler 自身控制是否继续流转到下一个 handler(ctx.sendDownstream(e),这样设计有个好处就是随时终止流转，业务目的达到无需继续流转到下一个 handler)。下面的代码是取得下一个处理 Downstream 事件的处理器。

```
DefaultChannelHandlerContext realCtx = ctx;
while (!realCtx.canHandleUpstream()) {
    realCtx = realCtx.next;
    if (realCtx == null) {
        return null;
    }
}

return realCtx;
```

如果是一个网络会话最末端的事件，比如 messageRecieve，那么可能在某个 handler 里面就直接结束整个会话，并把数据交给上层应用，但是如果是网络会话的中途事件，比如 connect 事件，那么当触发 connect 事件时，经过 pipeline 流转，最终会到达挂载 pipeline 最底下的 ChannelSink 实例中，这类实例主要

作用就是发送请求和接收请求，以及数据的读写操作。



NIO 方式 ChannelSink 一般会有 1 个 BOSS 实例(implements Runnable)，以及若干个 worker 实例（不设置默认为 cpu cores*2 个 worker），这在前面已经提起过，BOSS 线程在客户端类型的 ChannelSink 和服务器端类型的 ChannelSink 触发条件不一样，客户端类型的 BOSS 线程是在发生 connect 事件时启动，主要监听 connect 是否成功，如果成功，将启动一个 worker 线程,将 connected 的 channel 交给这个线程继续下面的工作，而服务器端的 BOSS 线程是发生在 bind 事件时启动，它的工作也相对比较简单，对于 channel.socket().accept()进来的请求向 Nioworker 进行工作分配即可。这里需要提到的是，Server 端 ChannelSink 实现比较特别，无论是 NioServerSocketPipelineSink 还是 OioServerSocketPipelineSink 的 eventSunk 方法实现都将 channel 分为 ServerSocketChannel 和 SocketChannel 分开处理。这主要原因是 Boss 线程 accept()一个新的连接生成一个 SocketChannel 交给 Worker 进行数据接收。

```
public void eventSunk(
        ChannelPipeline pipeline, ChannelEvent e) throws Exception {
```

```
        Channel channel = e.getChannel();
        if (channel instanceof NioServerSocketChannel) {
            handleServerSocket(e);
        } else if (channel instanceof NioSocketChannel) {
            handleAcceptedSocket(e);
        }
    }
NioWorker worker = nextWorker();
                worker.register(new NioAcceptedSocketChannel(
                        channel.getFactory(), pipeline, channel,
                        NioServerSocketPipelineSink.this,
acceptedSocket,
                        worker, currentThread), null);
```

另外两者实例化时都会走一遍如下流程：

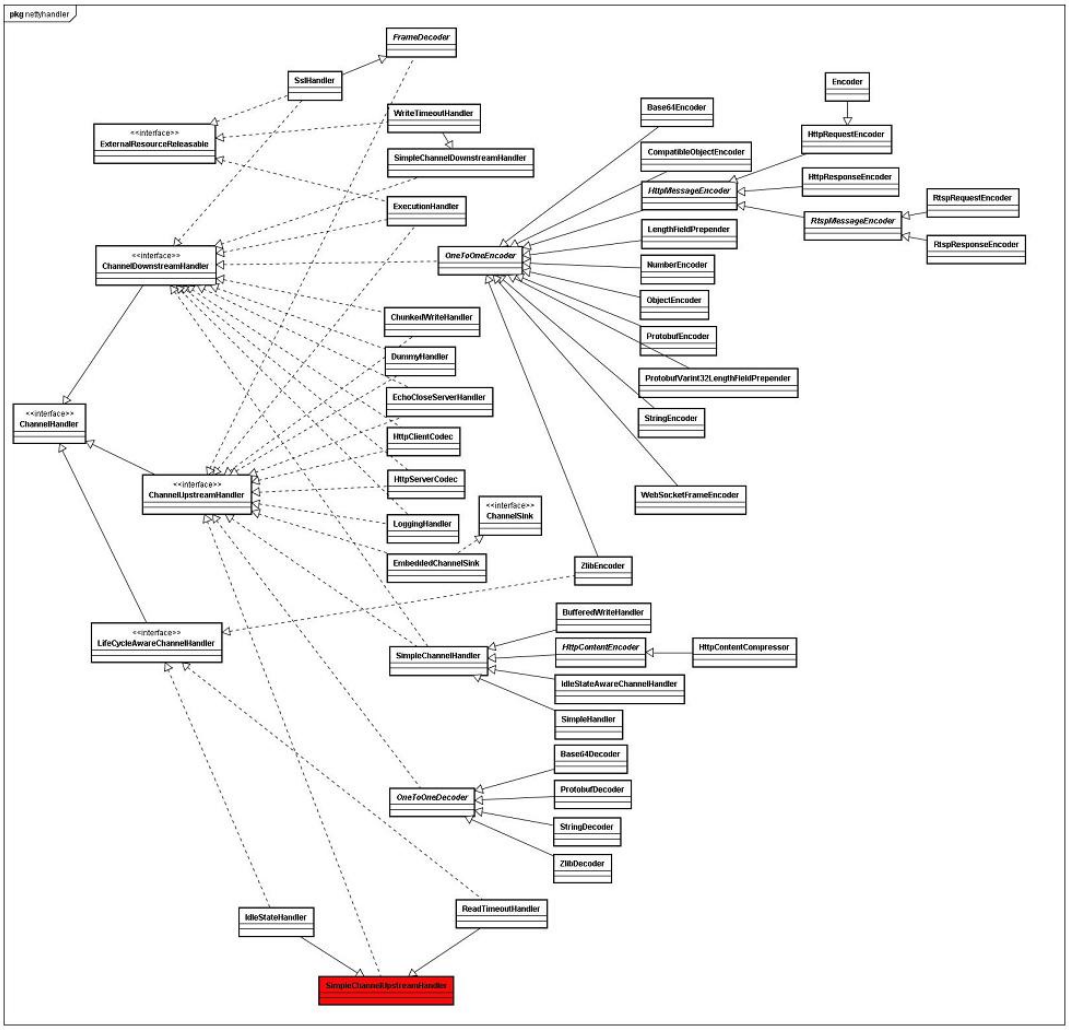
```
setConnected();
      fireChannelOpen(this);
      fireChannelBound(this, getLocalAddress());
      fireChannelConnected(this, getRemoteAddress());
```

而对应的 ChannelSink 里面的处理代码就不同于 ServerSocketChannel 了，因为走的是 handleAcceptedSocket（e）这一块代码，从默认实现代码来说,实例化调用 fireChannelOpen（this）;fireChannelBound（this,getLocalAddress()）;fireChannelConnected（this,getRemoteAddress（））没有什么意义，但是对于自己实现的 ChannelSink 有着特殊意义。具体的用途我没去了解，但是可以让用户插手 Server accept 连接到准备读写数据这一个过程的处理。

```
switch (state) {
          case OPEN:
              if (Boolean.FALSE.equals(value)) {
                  channel.worker.close(channel, future);
              }
              break;
          case BOUND:
          case CONNECTED:
              if (value == null) {
```

```
                channel.worker.close(channel, future);
            }
            break;
        case INTEREST_OPS:
            channel.worker.setInterestOps(channel, future,
((Integer) value).intValue());
            break;
        }
```

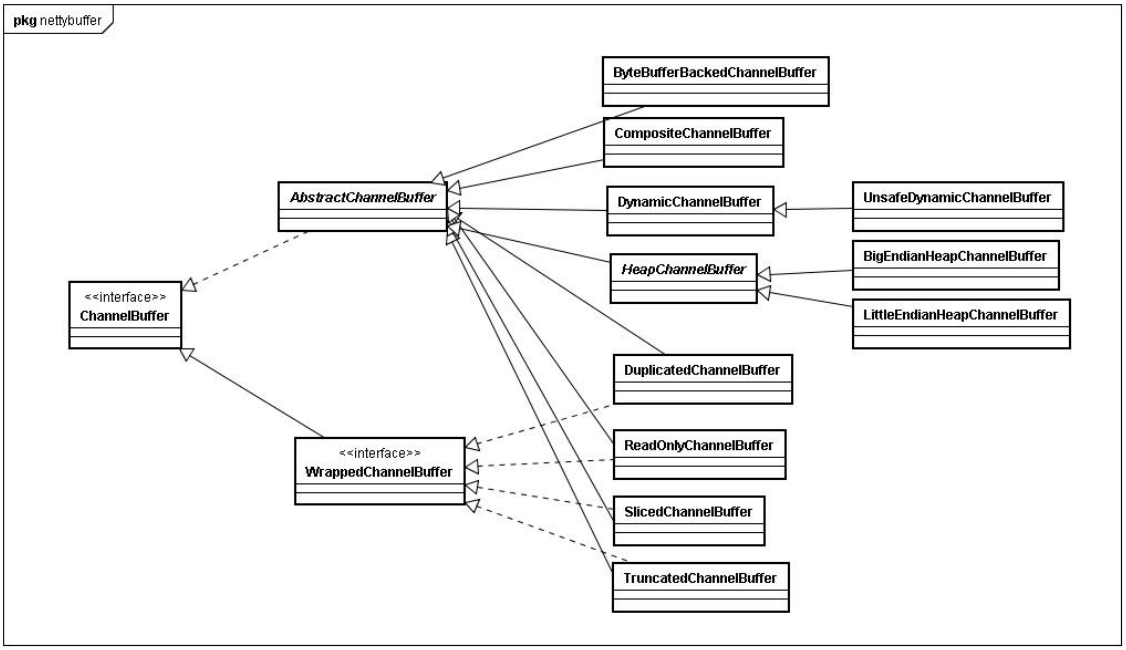
Netty 提供了大量的 handler 来处理网络数据，但是大部分是 CODEC 相关的，以便支持多种协议，下面一个图绘制了现阶段 Netty 提供的 Handlers(红色部分不完全)



Netty 实现封装实现了自己的一套 ByteBuffer 系统，这个 ByteBuffer 系统对外统一的接口就是 ChannelBuffer,这个接口从整体上来说定义了两类方法，一种是类似 getXXX(int index…)，setXXX(int index…)需要指定开始操作 buffer 的起始位置，简单点来说就是直接操作底层 buffer,并不用到 Netty 特有的高可重用性 buffer 特性，所以 Netty 内部对于这类方法调用非常少，另外一种是类

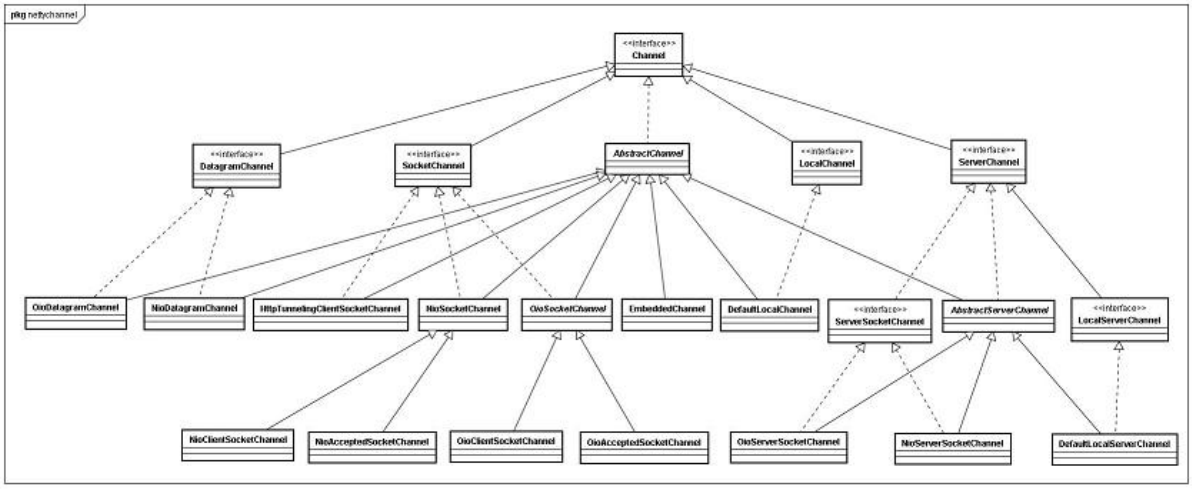
似 readXXX(),writeXXX()不需要指定位置的 buffer 操作，这类方法实现放在了 AbstractChannelBuffer，其主要的特性就是维持 buffer 的位置信息，包括 readerIndex,writerIndex,以及回溯作用的 markedReaderIndex 和 markedWriterIndex，当用户调用 readXXX()或者 writeXXX()方法时，AbstractChannelBuffer 会根据维护的 readerIndex,writerIndex 计算出读取位置，然后调用继承自己的 ChannelBuffer 的 getXXX(int index…)或者 setXXX(int index…)方法返回结果，这类方法在 Netty 内部被大量调用，因为这个特性最大的好处就是很方便地重用 buffer 而不必去费心费力维护 index 或者新建大量的 ByteBuffer。

另外 WrappedChannelBuffer 接口提供的是对 ChannelBuffer 的代理，他的用途说白了就是重用底层 buffer，但是会转换一些 buffer 的角色，比如原本是读写皆可 ，wrap 成 ReadOnlyChannelBuffer,那么整个 buffer 只能使用 readXXX()或者 getXXX()方法，也就是只读，然后底层的 buffer 还是原来那个,再如一个已经进行过读写的 ChannelBuffer 被 wrap 成 TruncatedChannelBuffer，那么新的 buffer 将会忽略掉被 wrap 的 buffer 内数据,并且可以指定新的 writeIndex,相当于 slice 功能。



Netty 实现了自己的一套完整 Channel 系统,这个 channel 说实在也是对 java 网络做了一层封装，加上了 SEDA 特性(基于事件响应，异步，多线程等)。其最终的网络通信还是依靠底下的 java 网络 api。提到异步，不得不提到 Netty 的 Future 系统，从 channel 的定义来说，
write,bind,connect,disconnect,unbind,close,甚至包括 setInterestOps 等方法都会返回一个 channelFuture，这这些方法调用都会触发相关网络事件，并且在 pipeline 中流转。Channel 很多方法调用基本上不会马上就执行到最底层，而是触发事件,在 pipeline 中走一圈,最后才在 channelsink 中执行相关操作，如果涉及网络操作，那么最终调用会回到 Channel 中，也就是 serversocketchannel,socketchannel,serversocket,socket 等 java 原生网络

api 的调用，而这些实例就是 jboss 实现的 channel 所持有的(部分 channel)。



Netty 新版本出现了一个特性 zero-copy,这个机制可以使文件内容直接传输到相应 channel 上而不需要通过 cpu 参与，也就少了一次内存复制。Netty 内部 ChunkedFile 和 FileRegion 构成了 non zero-copy 和 zero-copy 两种形式的文件内容传输机制，前者需要 CPU 参与，后者根据操作系统是否支持 zero-copy 将文件数据传输到特定 channel,如果操作系统支持，不需要 cpu 参与，从而少了一次内存复制。ChunkedFile 主要使用 file 的 read,readFully 等 API，而 FileRegion 使用 FileChannel 的 transferTo API,2 者实现并不复杂。Zero-copy 的特性还是得看操作系统的，本身代码没有很大的特别之处。

最后总结下，Netty 的架构思想和细节可以说让人眼前一亮，对于 java 网络 IO 的各个注意点，可以说 Netty 已经解决得比较完全了，同时 Netty 的作者也是另外一个 NIO 框架 MINA 的作者，在实际使用中积累了丰富的经验，但是本文也只是一个新手对于 Netty 的初步理解,还没有足够的能力指出某一细节的所发挥的作用。