Thinking in Netty

By 谢乐

1. 写在前面

Netty的词根为 `net`，那么我们就已经猜想到它与网络有关。官方对Netty的解释为:

> Netty是一种异步的基于事件驱动的Java网络应用框架，可用于构建高性能的协议服务器与客户端。

像我司以及其他互联网公司，例如Alibaba，Facebook等。有很多内部的中间件，分布式框架。而这些产品或应用的底层很多都用到了Netty或基于Netty的再开发产品。姑且不用再过多描述Netty是什么吧，单是出于好奇的原因，我便想一探究竟，本文不会涉及到很多Netty的使用，比如编解码，协议栈开发，本文是对Netty的底层原理，设计思想的探索。如果需要的话，请参考《Netty in Action>》，《Netty权威指南》等书籍。

2. Netty快速开始

从一个经典的Echo服务器Demo开始.

```
package cn.netty;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.FixedLengthFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;

/**
 * @author Xie le
 * @date 2016/4/13
 */
public class EchoServer {

    public static void main(String[] args) throws Exception {

        int port = 8080;
        new EchoServer().bind(port);
    }

    public void bind(int port) throws Exception {

        //配置线程组
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap bootstrap = new ServerBootstrap
();
            bootstrap.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 1024)
                    .handler(new LoggingHandler(LogLevel.DEBU
G))
                    .childHandler(new ChannelInitializer<Sock
etChannel>() {
                        @Override
                        protected void initChannel(SocketChan
```

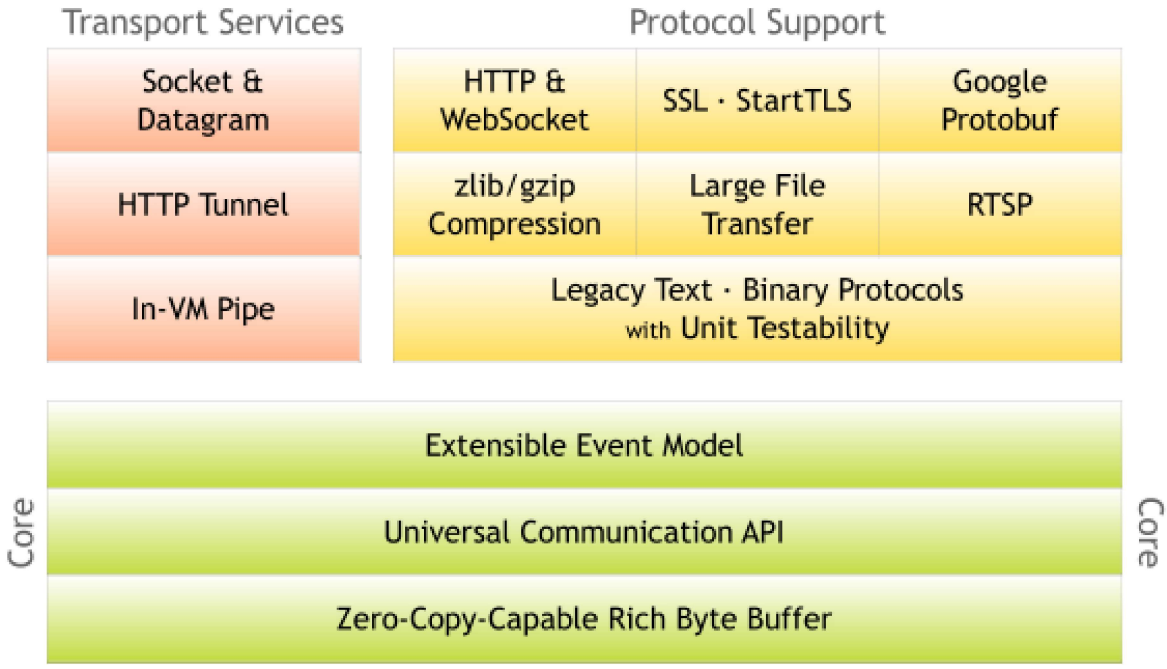
```
nel ch) throws Exception {

                            ch.pipeline().addLast(new LineBas
edFrameDecoder(128)) // DelimiterBasedFrameDecoder
                                    .addLast(new StringDecode
r())
                                    .addLast(new EchoServerHa
ndler());

                        }
                    });

            //绑定端口，同步等待
            ChannelFuture future = bootstrap.bind(port).sync
();

            future.channel().closeFuture().sync();

        } finally {

            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }


    private class EchoServerHandler extends ChannelHandlerAda
pter {
        @Override
        public void channelRead(ChannelHandlerContext ctx, Ob
ject msg) throws Exception {
            System.out.println("Receive Client : [" + msg +
"]");
            ctx.writeAndFlush(msg);
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ct
x, Throwable cause) throws Exception {
            cause.printStackTrace();
            ctx.close();
        }
    }
}
```

如上代码演示了Netty作为服务端的示例。启动EchoServer，然后通过Telnet 输入命令：

```
telnet 127.0.0.1 8080
```

然后观察Echo行为，虽然代码很短，但是五脏俱全。当然此处只是简单一窥而已，更多内容还在后面。

3. Netty的架构



这是http://netty.io 官网上的Netty架构图。可以看到，Netty包含几个部分：第一是传输服务，第二是协议支持，第三是底层核心部分：具有灵活的事件模型，提供通用通信API，还具有零拷贝的字节缓冲能力。

传输服务则包含有Socket和UDP通信服务，HTTP信道，虚拟机管道服务。协议支持则包含HTTP和WebSocket协议，还有压缩，大文件传输，Google Protobuf协议等。

4. Netty的线程模型

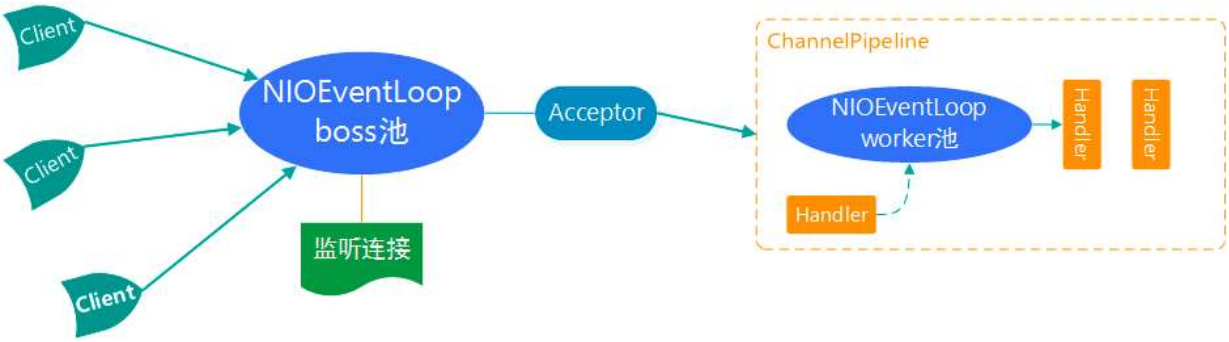
Netty的线程模型综合了Reactor线程模型，说到Reactor模型可以参考我的上一篇文章「Java IO模型&NIO」。
Netty的线程模型与文中的三种Reactor线程模型相似，下面章节我们通过Netty服务端和客户端的线程处理流程图来介绍Netty的线程模型。需要说明的是，使用的版本是Netty 5，部分代码与Netty 4 有出入。

4.1 服务端线程模型

Netty中通用的做法是将服务端监听线程和IO线程分离，类似于Reactor多线程模型，它的工作原理如图：

注：上图来源于李林峰的文章，本节的学习思路源于该文，感谢他的分享。

用户一般通过Web服务器或Main程序来启动Netty服务(类似于上述示例中的EchoServer代码)。Netty指定两个线程池组EventLoopGroup 作为主从线程池， 一个EventLoopGroup 即为EventLoop线程组，负责管理EventLoop的创建和回收。
EventLoopGroup管理的线程数可以通过构造函数设置，如果没有设置，默认取-Dio.netty.eventLoopThreads，如果该系统参数也没有指定，则为可用的CPU内核数 × 2。
bossGroup线程组实际就是Acceptor线程池，负责处理客户端的TCP连接请求，如果系统只有一个服务端端口需要监听，则建议bossGroup线程组线程数设置为1。
workerGroup是真正负责I/O读写操作的线程组，通过ServerBootstrap的group方法进行设置，用于后续的Channel绑定。
但Netty5引入了PipeLine模式，在原有的模型上作了一些改动。下图是我的理解：



Netty从主NIOEventLoop线程池分配好一个线程来作为Acceptor线程，用以接收连接并监听网络事件。但ServerBootstrap在初始化时，会为创建好的Channel上赋予一个管道ChannelPipeline，并把worker线程池绑定到该Pipeline上。Acceptor线程感知到连接事件后，会先创建SocketChannel，然后通过Pipeline在workerGroup中选取一个EventLoop线程作为IO处理线程，负责网络消息的读写，并把新建的 `SocketChannel` 注册到Selector上，以监听其他事件。

说到这里，需要先说明一下NIO的Server编程步骤，我们遵循源码就是最好的说明原则，直接show代码。

```
    /**
     * 获得一个ServerSocket通道，并对该通道做一些初始化的工作
     * @param port
     * @throws IOException
     */
    public NIOServer initServer(int port) throws IOException{

        //获得一个ServerSocket通道
        final ServerSocketChannel serverChannel = ServerSocke
tChannel.open();

        //设置通道为非阻塞
        serverChannel.configureBlocking(false);

        //将该通道对应的ServerSocket绑定到port端口
        serverChannel.socket().bind(new InetSocketAddress(por
t));

        //获得一个通道管理器
        this.selector =  Selector.open();

        /**
         * 将通道管理器和该通道绑定，并为该通道注册服务端接收客户端连接
事件（SelectionKey.OP_ACCEPT）
         * 注册该事件后，当该事件到达时，selector.select()会返回，如
果没有到达，selector.select（）
         * 会一直阻塞
         *
         */
        serverChannel.register(this.selector, SelectionKey.OP
_ACCEPT);

        serverChannel.validOps();

        return  this;
    }
```

NIO的Server编程一般会先初始化一个ServerSocketChannel，表示服务端的SocketChannel，这个类似于BIO中的ServerSocket，其负责在服务端监听特定端口并作请求接收。然后再打开一个Selector，表示多路复用选择器，然后把ServerSocketChannel注册到Selector上，用以在ServerSocketChannel上选择各种网络事件。

Netty作为NIO框架，虽赋予了很多灵活性和便捷性，但其底层实现还是依赖于NIO，我们从Netty的代码去剖析。

第一、创建ServerSocketChannel与注册

ServerBootstrap启动时，在 `channel()` 方法中指定Channel类型为 `NioServerSocketChannel.class` 。

然后在 `bind` 方法中会实例化该Channel,代码引用栈如下：

```
    bootstrap.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
    ...

    public B channel(Class<? extends C> channelClass) {
        if (channelClass == null) {
            throw new NullPointerException("channelClass");
        }
        return channelFactory(new ReflectiveChannelFactory<C>
(channelClass));
    }
   AbstractBootstrap#initAndRegister
   final ChannelFuture initAndRegister() {
        final Channel channel = channelFactory().newChannel
();
        try {
            init(channel);
        }
        ChannelFuture regFuture = group().register(channel);
        ....
    }
```

第二、ServerBootstrap通过group()和register方法把Channel注册给一个bossGroup中的线程，并以该线程作为Acceptor线程，然后监听服务端。ServerBootstrap选择一个Acceptor线程的主要逻辑如下：

```
MultithreadEventLoopGroup.java
public ChannelFuture register(Channel channel) {
        return next().register(channel);
}
MultithreadEventExecutorGroup.java
private final class PowerOfTwoEventExecutorChooser implements
EventExecutorChooser {
        @Override
        public EventExecutor next() {
            return children[childIndex.getAndIncrement() & ch
ildren.length - 1];
        }
}
```

第三、然后在 io.netty.channel.SingleThreadEventLoop#register(io.netty.channel.Channel)方法中把Channel注册到Group中的Acceptor线程，由该Acceptor线程来把Channel注册到Selector上，并作监听。代码调用栈的主要逻辑如下：

```
io.netty.channel.SingleThreadEventLoop#register(io.netty.chan
nel.Channel)
channel.unsafe().register(this, promise);
protected void doRegister() throws Exception {
        boolean selected = false;
        for (;;) {
            try {
                selectionKey = javaChannel().register(((NioEv
entLoop) eventLoop().unwrap()).selector, 0, this);
                return;
          }
....
```

第四、最后的注册实际上也是调用的 `java.nio.channels.spi.AbstractSelectableChannel#register` 方法，意为由NIO本身的API来完成注册。

第五、Selector开始在Acceptor线程中在ServerChannel上监听网络事件。由于之前我们指定的线程为NioEventLoop，因此这个职责很有可能在该类中，我们之间看代码：

```
io.netty.channel.nio.NioEventLoop#run
 try {
            if (hasTasks()) {
                selectNow();
            } else {
                select(oldWakenUp);
           ....
  }
private static void processSelectedKey(SelectionKey k, Abstra
ctNioChannel ch) {
           ....
        try {
            int readyOps = k.readyOps();
            //读事件，接收连接
            if ((readyOps & (SelectionKey.OP_READ | Selection
Key.OP_ACCEPT)) != 0 || readyOps == 0) {
                unsafe.read();
                if (!ch.isOpen()) {
                    // Connection already closed - no need to
handle write.
                    return;
                }
            }
            .....
}
```

第六、处理读或接收连接事件，调用了 `unsafe.read()` 方法。由于服务端我们指定定时 `NioServerSocketChannel`，而NioServerSocketChannel继承了 `AbstractNioMessageChannel`，所以unsafe的read方法在AbstractNioMessageChannel这个类中。

```
private final class NioMessageUnsafe extends AbstractNioUnsaf
e {
     public void read() {
          ....
          for (;;) {
               int localRead = doReadMessages(readBuf);
          }
     }
}
```

再看看doReadMessages方法就知道，NioServerSocketChannel创建了SocketChannel，以建立与客户端的通道连接。

```
protected int doReadMessages(List<Object> buf) throws Excepti
on {
         SocketChannel ch = javaChannel().accept();
         try {
             if (ch != null) {
                 buf.add(new NioSocketChannel(this, ch));
                 return 1;
             }
 }
```

第七、把新建的 `SocketChannel` 注册到workerGroup线程池中，以开始消息交互。触发的入口在AbstractNioMessageChannel类的read方法中：

```
for (int i = 0; i < size; i ++) {
                     pipeline.fireChannelRead(readBuf.get(i));
 }
```

需要说明的是，在ServerBootstrap的 `init` 方法中把workerGroup线程池绑定到了Pipeline中.

```
ch.pipeline().addLast(new ServerBootstrapAcceptor(
                          currentChildGroup, currentChildHandle
r, currentChildOptions, currentChildAttrs));
```

而 `ServerBootstrapAcceptor` 的 `channelRead` 方法则把新建的SocketChannel赋予到了childGroup中，
其中childGroup就是之前的workerGroup, 从中选择一个I/O线程负责网络消息的读写。

```
...
childGroup.register(child)
...
```

`register` 会把新建的SocketChannel注册到Selector上，然后开始正式的通信。如果有读事件来临，则又会触发 `processSelectedKey` 方法，从而进入数据处理与交互过程。

4.1 客户端线程模型

客户端线程模型相对要简洁一些，如通过阅读源码掌握了服务端的线程模型，那么客户端的线程模型则会很容易理解。其工作原理大致如下:

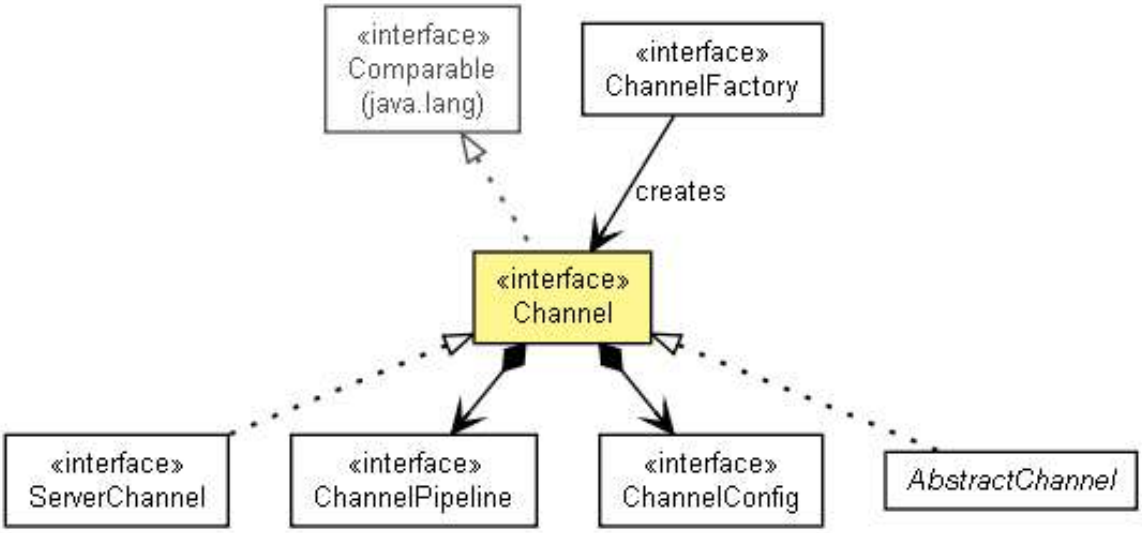
源码分析暂就不屠版了。

5. Netty的API设计

其实API需要多用多练才能更好的掌握与熟悉。Netty作为NIO框架，它的API几乎都是围绕NIO的API来封装或优化的，旨在提供便利，高效率的功用。

我们先看看Netty对NIO的Channel的封装

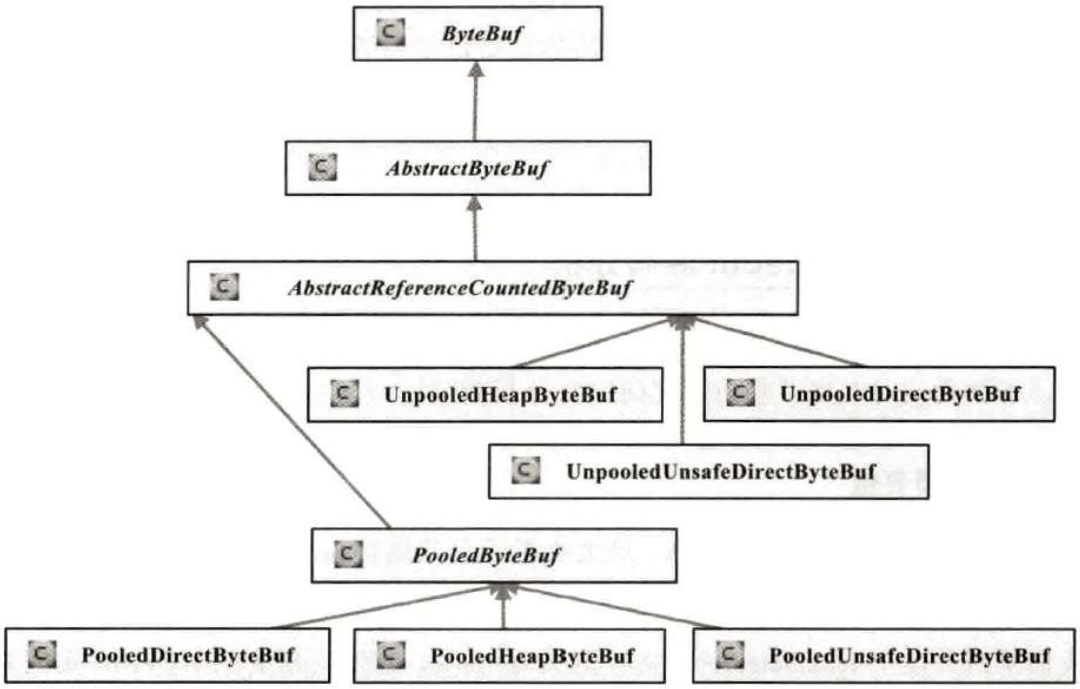
和Channel相关的接口及类结构图如下：



1）当前Channel的状态信息，比如是打开还是关闭等。
2）通过ChannelConfig可以得到的Channel配置信息。
3）Channel所支持的如read、write、bind、connect等IO操作。
4）得到处理该Channel的ChannelPipeline，既而可以调用其做和请求相关的IO操作。

在Channel实现方面，以通常使用的nio socket来说，Netty中的NioServerSocketChannel和NioSocketChannel分别封装了java.nio中包含的ServerSocketChannel和SocketChannel的功能。
注:此处内容参考自ImportNew上的文章, 觉得该作者总结的很好

其次是ByteBuf
ByteBuf是对NIO中的ByteBuffer的封装，新增了很多对字节的便利的方法，还提供了很多类型的ByteBuf, 比如是直接分配到堆上的Buf, 还有分配到对象池中的Buf，以节省空间， 还有原生内存Buf, 分配和回收效率更高。实际使用时，根据需要选择合适的ByteBuf就行。我们借用《Netty权威指南》中的一张图来加深理解。



6. Netty的通信过程

Debug是最好的跟踪方式，也是最好的源码阅读引导。以上文中的EchoServer为例，跟踪一下Netty服务端的数据处理以及整个通信过程。启动EchoServer后，使用 `telnet` 发送一段报文 `hello, netty`.

按照上文分析的逻辑，客户端对应的SocketChannel由IO线程来处理，Selector感知到读事件后，

在 `io.netty.channel.nio.NioEventLoop#processSelectedKey()` 方法中调用NioUnsafe来读取数据。

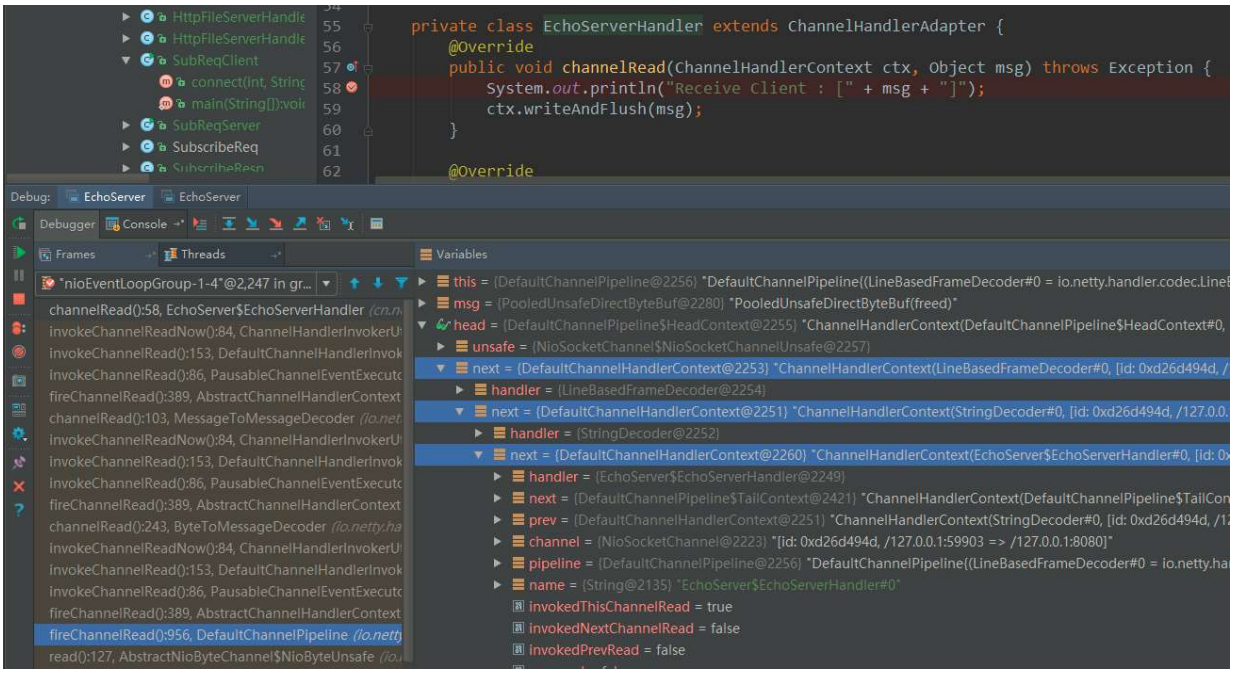
Netty按照默认的缓存区来读取字节数据，并读入到Bytebuf中，然后开始进入管道，由管道中的Handler层层处理。首先说明，在EchoServer中我们为客户端的Channel指定了三个Handler，说明如下：

`LineBasedFrameDecoder` 换行解码，

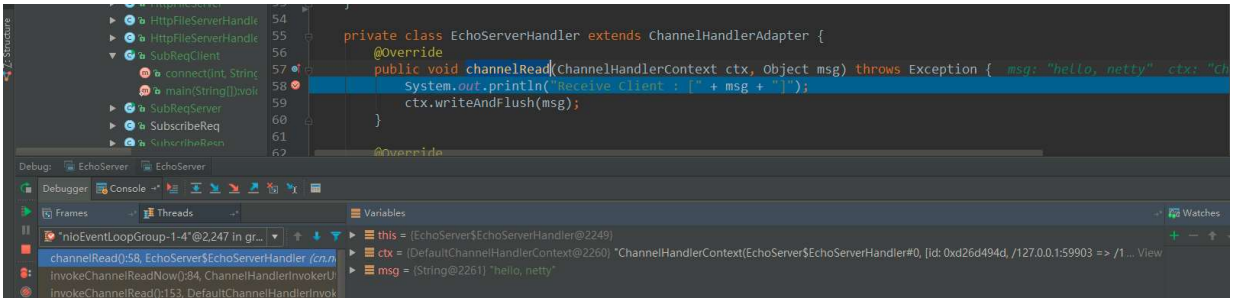
`StringDecoder` 字符串解码，

`EchoServerHandler` 业务处理。

运行中的Debug断点截图如下：



可以看到，客户端请求的数据经过Pipeline中的Handler层层编码，最后把实际报文传递到业务Handler中。即在 `EchoServerHandler` 的channelRead方法中拿到的数据已经为 `hello, netty`.



在示例中，业务处理很简单，只做了echo输出，一般正式的开发中，会自定义很多业务Handler来完成复杂的协议栈开发。我们跟踪一下echo回写的过程。

```
ctx.writeAndFlush -> ChannelHandlerInvokerUtil.invokeFlushNow
-> NioSocketChannel.doWrite -> SocketChannel.write
```

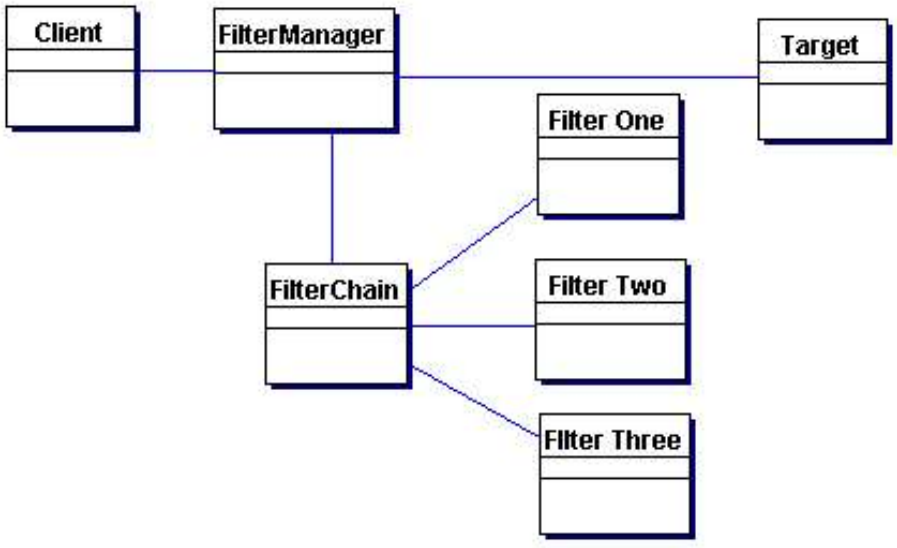
可以看到，最后仍是调用了 `SocketChannel` 的写方法，是为又回到了NIO的世界中。

7. Netty中的设计模式

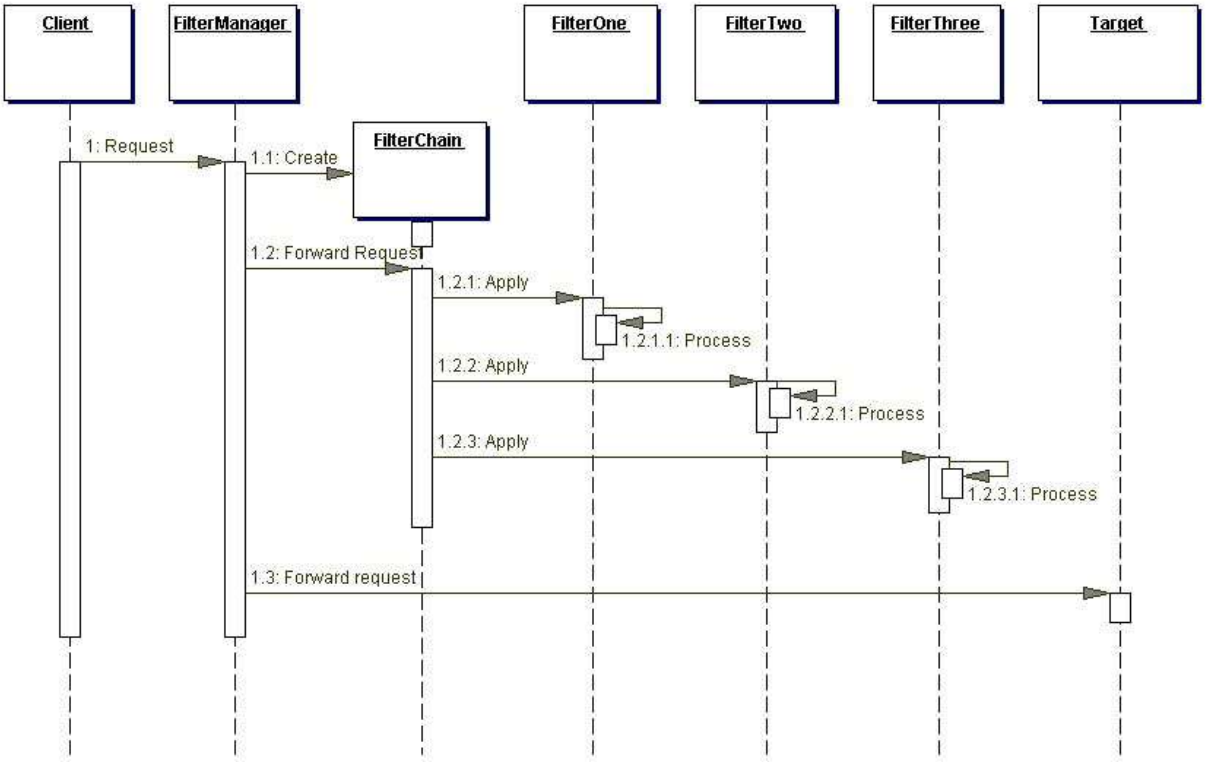
Netty作为一个优秀的框架用到了很多设计模式。我主要分析一下Netty中的ChannelPipeline，每个创建好的ChannelNetty会为该Channel创建一个Pipeline，用以处理或拦截入站的事件和出站的操作。ChannelPipeline是一种拦截过滤器模式的实现形式，可以让用户更好的控制事件以及让Pipeline中各个ChannelHandler的交互更便利。

拦截过滤器模式是 J2EE 的核心设计模式之一。它可以创建一种可插拔的过滤器，并用一种标准的方式来处理通用的服务，而不用修改核心的请求处理代码。过滤器拦截入站的请求和出站的响应，并可以作前置处理和后置处理。

过滤拦截器模式的类图如下：



引入参与者和划分职责后，时序图如下：



FilterManager

`FilterManager` 管理过滤处理。它用相关的过滤器来创建 `FilterChain` ，同时规定顺序，然后开始处理请求。

FilterChain

`FilterChain` 是一组有序的互相独立的filters.

FilterOne,FilterTwo,FilterThree

这些是映射到target的独立的filter， `FilterChain` 会协调他们之间的处理。

***Target**

Target是客户端的请求资源。

Netty中的ChannelPipeline是这种模式的实现。在ChannelPipeline中主要关联ChannelHandler，可以通过 `addFirst`，`addLast`，`addBefore` ， `remove` 方法来添加或移除ChannelHandler。 `DefaultChannelPipeline` 是ChannelPipeline的默认实现，采用自定义的链表结构来组合ChannelHandler。
这个部分分析的是过滤拦截器模式，而ChannelPipeline的更多实现细节可参看官方API和源码。

8. 最后

Netty很繁杂，层层抽象，层层封装。阅读时需要抽丝剥茧，捋一条主线，比如通信过程，或者线程模型这些主线。我阅读时是花了很多时日，如有错误，欢迎指出。