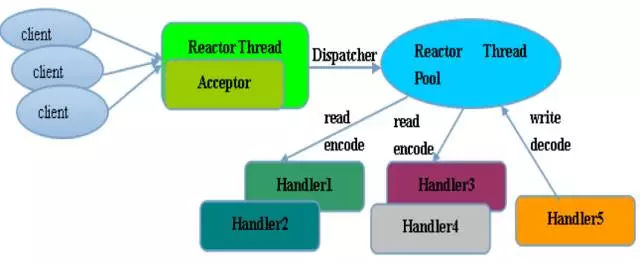
Netty Reactor模型

目前主流的Java网络程序基本上是在Java NIO的基础上实现Reactor模型，隐藏NIO底层的复杂细节，大大简化了NIO编程，其原理如下图所示：



1. Acceptor(Main Reactor)，负责接收客户端Socket发起的新建连接请求，并把该Socket绑定到Sub Reacor线程成
2. 客户端Socket随后的读写事件都有Reactor线程来处理
3. Reactor线程读取数据，对数据进行预处理(编解码成Java POJO)后交给程序的Handler来实现特定的业务逻辑处理。为了不影响Reactor线程，通常使用单独的线程池来异步执行Handler接口方法。

Netty是Reactor模型的一种实现，同时支持单线程、多线程和主从多线程模型，可以在启动参数中配置线程池个数来切换不同的模型，Netty Reactor的模型图如下所示：



在Netty中将Main /Sub Reactor的类比概念为BossPool及WorkerPool，其中Boss Reactor只有一个线程，而Worker的线程数由Worker线程池的大小决定的，但是默认不会超过CPU核数\*2。

http://www.blogjava.net/jb2011/archive/2018/11/05/433468.html

# Netty Server入口

下面是Netty Server的启动示例：

*public void run() throws Exception {*

*EventLoopGroup bossGroup = new NioEventLoopGroup(1);*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*try {*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup)*

*.channel(NioServerSocketChannel.class) //接收客户端请求*

*.childHandler(new ChannelInitializer<SocketChannel>() {*

*public void initChannel(SocketChannel ch) throws Exception {*

*ch.pipeline().addLast(new DiscardHandler());*

*}*

*})*

*.option(ChannelOption.SO\_BACKLOG,128)*

*.childOption(ChannelOption.SO\_KEEPALIVE, true);*

*ChannelFuture f = b.bind(port).sync();*

*f.channel().closeFuture().sync();*

*} ....*

*}*

Netty Server主要是由ServerBootStrap来设置线程池、Channel及流水线Pipe，核心是创建NioEventLoop，其是Netty的Reactor线程，对于Main/Sub Reactor来说是相同的线程执行类。设置完这些参数后则调用bind开始监听流程，最终会调用doBind方法，执行如下：

*private ChannelFuture doBind(final SocketAddress localAddress) {*

*final ChannelFuture regFuture = initAndRegister();*

*final Channel channel = regFuture.channel();*

*......*

*if (regFuture.isDone()) {*

*// At this point we know that the registration was complete and successful.*

*ChannelPromise promise = channel.newPromise();*

*doBind0(regFuture, channel, localAddress, promise);*

*return promise;*

*} ......*

*}*

在initAndRegister中初始化并注册，此方法中有createChannel和init(Channel)

*final ChannelFuture initAndRegister() {*

*Channel channel = null;*

*try { //通过ServerBootStrap.channel来指定，<=.channel(NioServerSocketChannel.class)*

*channel = channelFactory.newChannel();*

*init(channel);*

*}.....*

*}*

核心是init(channel)操作，其设置options及流水线pipe，如下所示：

*void init(Channel channel) throws Exception {*

*//设置options，此处省略*

*final Map<ChannelOption<?>, Object> options = options0();*

*...*

*//设置NioServerSocketChannel的pipeline*

*ChannelPipeline p = channel.pipeline();*

*final EventLoopGroup currentChildGroup = childGroup; // 设置Sub Reactor*

*final ChannelHandler currentChildHandler = childHandler;*

*//设置Sub Reactor的Handler，在Netty中Handler是List 串*

*p.addLast(new ChannelInitializer<Channel>() {*

*@Override*

*public void initChannel(final Channel ch) throws Exception {*

*final ChannelPipeline pipeline = ch.pipeline();*

*ChannelHandler handler = config.handler();*

*if (handler != null) {*

*pipeline.addLast(handler);*

*}*

*ch.eventLoop().execute(new Runnable() {*

*@Override*

*public void run() { //核心是将ServerBootstrapAcceptor添加到Server Socket中*

*pipeline.addLast(new ServerBootstrapAcceptor(*

*ch, currentChildGroup,*

*currentChildHandler,*

*currentChildOptions,*

*currentChildAttrs));*

*}}); });}*

# NioEventLoopGroup

NioEventLoop是Netty的Reactor线程，其在Netty线程模型中的主要职责如下：

* Boss NioEventLoop作为服务端的Acceptor线程，负责处理客户端的请求接入
* Worker NioEventLoop作为客户端的Connector的线程，负责注册监听连接操作符，用于判断异步连接结果
* 作为IO线程，监听网络读操作为，负责从SocketChannel中读取报文
* 作为IO线程，负责向SocketChannel写入报文并发送给对方，如果发生写半包，会自动注册写事件，用于后续继续发送半包数据，直到数据全部发送完成

在上面的示例中，ServerBootStrap的设置NioEventLoop如下：

*EventLoopGroup bossGroup = new NioEventLoopGroup(1);*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup)*

其分别赋值给group及childGroup成员变量：

*public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {*

*super.group(parentGroup); //Main Reactor*

*this.childGroup = childGroup; //Sub Reactor*

*return this;*

*}*

## Boss NioEventLoop

在ServerBootStrap中设置为parentGroup，在初始化时将NioServerSocketChannel注册到parentGroup中

*ServerBootStrap#initAndRegister中*

*channel = channelFactory.newChannel();*

*ChannelFuture regFuture = config().group().register(channel);*

register操作，判断是否与Event Loop线程相同，第一次是boss Group，肯定不相同，将该线程作为main线程：

*AbstractChannel.this.eventLoop = eventLoop;*

*if (eventLoop.inEventLoop()) {*

*register0(promise);*

*} else {*

*try {*

*eventLoop.execute(new Runnable() {*

*@Override*

*public void run() {*

*register0(promise); }});}*

在ServerBootStream#init(Channel)中设置了Pipeline，核心是将ServerBootstrapAcceptor添加到Channel中，其核心源码如下：

*ChannelPipeline p = channel.pipeline();*

*p.addLast(new ChannelInitializer<Channel>() {  
 @Override  
 public void initChannel(final Channel ch) throws Exception {  
 final ChannelPipeline pipeline = ch.pipeline();  
 ChannelHandler handler = config.handler();  
 if (handler != null) {  
 pipeline.addLast(handler);  
 }  
  
 ch.eventLoop().execute(new Runnable() {  
 @Override  
 public void run() {  
 pipeline.addLast(new ServerBootstrapAcceptor(  
 ch, currentChildGroup, currentChildHandler,*

*currentChildOptions, currentChildAttrs));  
 }  
 });  
 }  
});*

Boss NIOEventLoopGroup在示例中是主线程，处理的Channel为NioServerSocketChannel，其启动在NIOEventLoop中，执行如下：

*@Override*

*protected void run() {*

*for (;;) {*

*//设置执行策略，省略*

*cancelledKeys = 0;*

*needsToSelectAgain = false;*

*final int ioRatio = this.ioRatio;*

*if (ioRatio == 100) {*

*try {*

*processSelectedKeys();*

*} finally { runAllTasks(); }*

*} else {*

*final long ioStartTime = System.nanoTime();*

*try {*

*processSelectedKeys();*

*} ...*

*}*

*}*

其中的ioRatio的含义为控制控制IO运行比例，如果ioRatio默认是50，则表示IO操作和执行Task所占用的线程比例是1:1。

从代码中可以看到，这是一个死循环，用于轮询事件，如果有task存在则触发select，接下来处理SelectKey，默认进入processSelectedKeysOptimized方法，如下所示：

*private void processSelectedKeys() {*

*if (selectedKeys != null) {*

*processSelectedKeysOptimized();*

*} else {*

*processSelectedKeysPlain(selector.selectedKeys());*

*}*

*}*

在processSelectedKeysOptimized方法中有三个if判断，如下：

*private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {*

*final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();*

*if (!k.isValid()) {*

*final EventLoop eventLoop;*

*try {*

*eventLoop = ch.eventLoop();*

*......*

*try {*

*int readyOps = k.readyOps();*

*if ((readyOps & SelectionKey.OP\_CONNECT) != 0) {*

*int ops = k.interestOps();*

*ops &= ~SelectionKey.OP\_CONNECT;*

*k.interestOps(ops);*

*unsafe.finishConnect();*

*}*

*if ((readyOps & SelectionKey.OP\_WRITE) != 0) {*

*ch.unsafe().forceFlush();*

*}*

*if ((readyOps & (SelectionKey.OP\_READ | SelectionKey.OP\_ACCEPT)) != 0*

*|| readyOps == 0) {*

*unsafe.read();*

*}*

*}*

对于第一个分支，客户端程序进入表示tcp连接完成

*if ((readyOps & SelectionKey.OP\_CONNECT) != 0) {*

*int ops = k.interestOps();*

*ops &= ~SelectionKey.OP\_CONNECT;*

*k.interestOps(ops);*

*unsafe.finishConnect();*

*}*

对于Boss NioEventLoop来说，主要处理第二个if分支的OP\_ACCEPT事件，处理客户端接入请求Accept，

*if ((readyOps & (SelectionKey.OP\_READ | SelectionKey.OP\_ACCEPT)) != 0*

*|| readyOps == 0) {*

*unsafe.read(); <= final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();*

*}*

对于Boss NioEventLoop来说，unsafe类为NioServerSocketChannel，只监听OP\_ACCEPT事件，如下：

*public NioServerSocketChannel(ServerSocketChannel channel) {*

*super(null, channel, SelectionKey.OP\_ACCEPT);*

其read执行如下：

*@Override //实现类AbstractNioMesssageChannel#read*

*public void read() {*

*assert eventLoop().inEventLoop();*

*final ChannelConfig config = config();*

*final ChannelPipeline pipeline = pipeline();*

*final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();*

*allocHandle.reset(config);........*

*try {try {*

*do {*

*int localRead = doReadMessages(readBuf);*

*....*

*allocHandle.incMessagesRead(localRead);*

*} while (allocHandle.continueReading());*

*}*

*int size = readBuf.size();*

*for (int i = 0; i < size; i ++) {*

*readPending = false;*

*pipeline.fireChannelRead(readBuf.get(i));*

*}}}*

*config = new NioServerSocketChannelConfig(this, javaChannel().socket()); }*

最重要的方法就是doReadMessages

*@Override*

*protected int doReadMessages(List<Object> buf) throws Exception {*

*SocketChannel ch = SocketUtils.accept(javaChannel());*

*try {*

*if (ch != null) {*

*buf.add(new NioSocketChannel(this, ch));*

*return 1;*

*}}*

*return 0;*

*}*

这里的add操作，添加到List<Object>中，childEventLoopGroup().next()操作的执行过程中，会为NioSocketChannel挑选一个WorkGroup线程，这个线程服务于客户端和服务端，这里也是Netty Reactor的核心。具体的执行见read执行：

*for (int i = 0; i < size; i ++) {*

*readPending = false;*

*pipeline.fireChannelRead(readBuf.get(i));*

*}*

pipeline#fireChannelRead，即将NioSocketChannel发送给Handler处理，对于Boss WorkGroup来说，其会调用ServerBootstrapAcceptor#channelRead，执行如下：

*public void channelRead(ChannelHandlerContext ctx, Object msg) {*

*final Channel child = (Channel) msg; //msg为NioSocketChannel*

*child.pipeline().addLast(childHandler);*

*//将childHandler添加到Channel中，用于客户端请求的处理*

*setChannelOptions(child, childOptions, logger);*

*for (Entry<AttributeKey<?>, Object> e: childAttrs) { //添加参数*

*child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());*

*} //以上完成了客户端Socket Channel的初始化*

*try { //Netty Reactor的核心，childGroup.register*

*childGroup.register(child).addListener(new ChannelFutureListener() {*

*@Override*

*public void operationComplete(ChannelFuture future) throws Exception {*

*if (!future.isSuccess()) {*

*forceClose(child, future.cause());*

*}*

*}*

*});*

*}*

*}*

## **Worker NioEventLoop**

**1）Worker NioEventLoopGroup的初始化**

ChildGroup即Worker NioEventLoopGroup，其register的执行如下：

*@Override <= MultithreadEventLoopGroup  
public ChannelFuture register(Channel channel) {  
 return next().register(channel);  
}*

next为MultithreadEventLoopGroup#next，从初始化好的EventLoopGroup中随机挑选客户端Channel的处理线程

*public EventExecutor next() { <= MulithreadEventExecutorGroup  
 return chooser.next(); <= EventExecutorChooser  
}*

对于EventExecutorChooser处理的成员变量是：

*private final EventExecutor[] children*

在程序初始时赋值，见示例

*EventLoopGroup bossGroup = new NioEventLoopGroup(1);*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup)*

不指定线程数，则默认使用可用线程：

*DEFAULT\_EVENT\_LOOP\_THREADS = Math.max(1, SystemPropertyUtil.getInt(*

*"io.netty.eventLoopThreads", NettyRuntime.availableProcessors() \* 2)*

对于NioEventLoopGroup的初始化如下：

*protected MultithreadEventExecutorGroup(int nThreads, Executor executor,*

*EventExecutorChooserFactory chooserFactory, Object... args) {*

*children = new EventExecutor[nThreads];*

*for (int i = 0; i < nThreads; i ++) {*

*boolean success = false;*

*try {*

*children[i] = newChild(executor, args);*

*success = true;*

*} }*

*chooser = chooserFactory.newChooser(children);*

*final FutureListener<Object> terminationListener = new FutureListener<Object>() {*

*@Override*

*public void operationComplete(Future<Object> future) throws Exception {*

*if (terminatedChildren.incrementAndGet() == children.length) {*

*terminationFuture.setSuccess(null);*

*}*

*}*

*};*

*......*

*}*

1. 服务线程-从NioSocketChannel中读取数据

NioSocketChannel实例化后

*protected AbstractNioByteChannel(Channel parent, SelectableChannel ch) {*

*super(parent, ch, SelectionKey.OP\_READ);*

*}*

接收OP\_READ事件，根据上面的分析，其核心是read方法

*@Override*

*public final void read() {*

*final ChannelConfig config = config();*

*final ChannelPipeline pipeline = pipeline(); //获取channel piple*

*final ByteBufAllocator allocator = config.getAllocator(); //缓存分配器*

*final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();*

*allocHandle.reset(config);*

*ByteBuf byteBuf = null; //缓存，存放接收数据*

*boolean close = false;*

*try {*

*do {*

*byteBuf = allocHandle.allocate(allocator);*

*allocHandle.lastBytesRead(doReadBytes(byteBuf)); //读取数据*

*......*

*pipeline.fireChannelRead(byteBuf); //交给Pipeline处理*

*byteBuf = null;*

*} while (allocHandle.continueReading());*

*}*

读取数据doReadBytes的实现如下：

*protected int doReadBytes(ByteBuf byteBuf) throws Exception {*

*final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();*

*allocHandle.attemptedBytesRead(byteBuf.writableBytes());*

*return byteBuf.writeBytes(javaChannel(), allocHandle.attemptedBytesRead());*

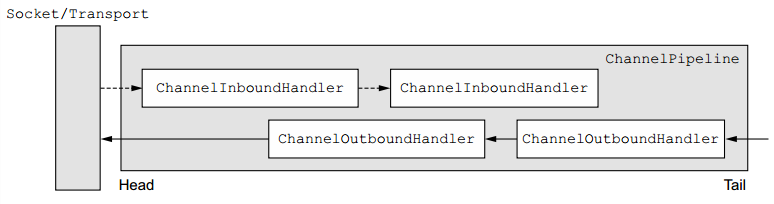
*}*

最终调用NioByteUnsafe读取数据

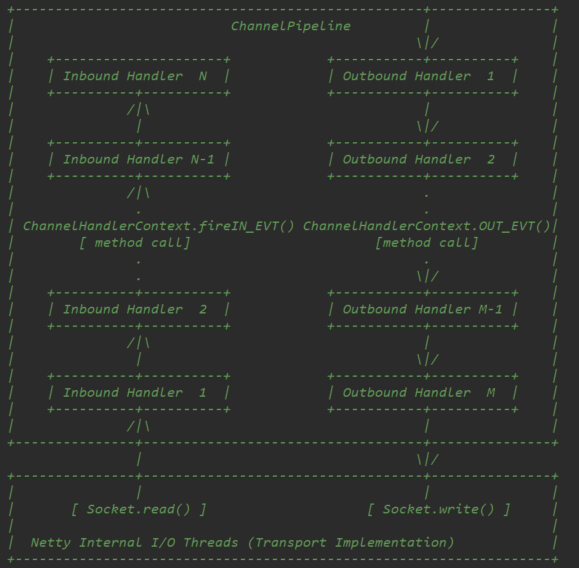
# ChannelInboundHandler

ChannelHandler负责I/O事件或者I/O操作进行拦截和处理，用户可以通过ChannelHandlerAdapter来选择性实现自己感兴趣的事件拦截和处理。由于Channel只负责实际的I/O操作，因此数据的编解码和实际处理都需要通过ChannelHandler进行处理。

ChannelPipeline相当于ChannelHandler的容器，Channel事件消息在ChannelPipeline中流动和传播，相应的事件能够被ChannelHandler拦截处理、传递、忽略或者终止，如下图所示：



Channel事件分为INBOUND和OUTBOUND事件: Inbound事件，当发生某个I/O操作时由IO线程流向用户业务处理线程的事件，如链路建立、链路关闭或者读完成等；OUTBOUND，由用户线程或者代码发起的IO操作事件。



在上面的示例中，Handler的初始化如下所示：

*.childHandler(new ChannelInitializer<SocketChannel>() {*

*public void initChannel(SocketChannel ch) throws Exception {*

*ch.pipeline().addLast(new DiscardHandler());*

*}*

*})*

其中DiscardHandler的实现如下：

*public class DiscardHandler extends ChannelInboundHandlerAdapter {  
 @Override  
 public void channelRead(ChannelHandlerContext ctx, Object msg) {  
 ByteBuf in = (ByteBuf) msg; //将信息进行处理  
 try {  
 while (in.isReadable()) {  
 System.out.println((char) in.readByte());  
 System.out.flush();  
 }  
 } finally {  
 ReferenceCountUtil.release(msg);  
 }  
 }}*

1. **ChannelHandler的Pipeline初始化**

在NioSocketChannel中Pipeline的初始化如下：

其会调用ServerBootstrapAcceptor#channelRead，执行如下：

*public void channelRead(ChannelHandlerContext ctx, Object msg) {*

*final Channel child = (Channel) msg; //msg为NioSocketChannel*

*child.pipeline().addLast(childHandler);*

*//将childHandler添加到Channel中，用于客户端请求的处理*

*setChannelOptions(child, childOptions, logger);*

*for (Entry<AttributeKey<?>, Object> e: childAttrs) { //添加参数*

*child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());*

*} //以上完成了客户端Socket Channel的初始化*

*try { //Netty Reactor的核心，childGroup.register*

*childGroup.register(child).addListener(new ChannelFutureListener() {*

*@Override*

*public void operationComplete(ChannelFuture future) throws Exception {*

*if (!future.isSuccess()) {*

*forceClose(child, future.cause());*

*}*

*}*

*});*

*} }*

1. **数据处理**

根据NioEventLoopGroup的分析，从Channel中读取数据后，交由Pipleline处理，

*@Override NioSocketChannel#*

*public final void read() {*

*....*

*final ChannelPipeline pipeline = pipeline(); //获取channel piple*

*try {*

*do {*

*byteBuf = allocHandle.allocate(allocator);*

*allocHandle.lastBytesRead(doReadBytes(byteBuf)); //读取数据*

*......*

*pipeline.fireChannelRead(byteBuf); //交给Pipeline处理*

*byteBuf = null;*

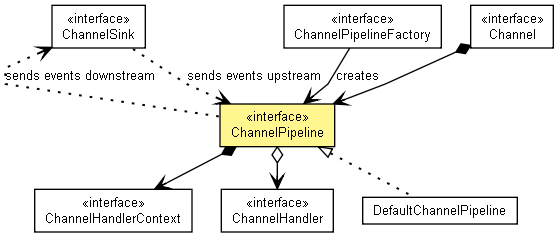
*} while (allocHandle.continueReading());*

*}*

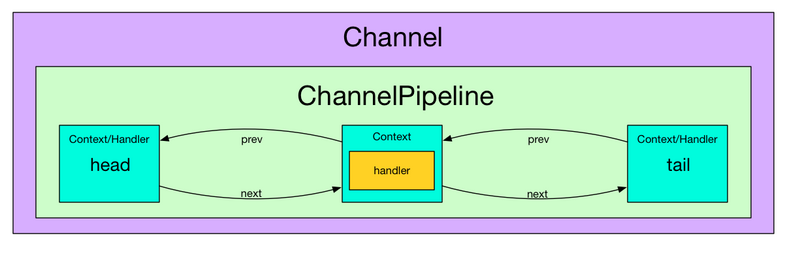
**3）ChannelPipeline**

hannelPipeline实际上应该叫做ChannelHandlerPipeline，可以看做是ChannelHandler的链表，当需要对Channel进行某种处理时，pipeline负责依次调用每一个Handler进行处理。ChannelPipeline为ChannelHandler链提供一个容器并定义了用于链接传播入站和出站事件流的API，当创建Channel时，会自动创建一个附属的ChanelPipeline，调用Channel.pipeline可以获得Channel的Pipeline，调用Pipeline#channel方法可以获得Pipe的Channel。

ChannelPipeline相关的接口及类图如下所示：



Netty中每个Channel都有且仅有一个ChannelPipeline与之对应，组成关系如下：



如上图，一个Channel包含一个ChannelPipeline，而ChannelPipeline中又维护由一个ChannelHandlerContext组成的双向链表，这个链表的头是HeadContext，链表的尾是TailContext，并且每个ChannelHandlerContext中又关联一个ChannelHandler。当channel完成register、active和read等操作时，会触发pipeline的相关方法：

**4）数据的处理**

*static void invokeChannelRead(final AbstractChannelHandlerContext next, Object msg) {*

*final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg, "msg"), next);*

*EventExecutor executor = next.executor();*

*if (executor.inEventLoop()) {*

*next.invokeChannelRead(m);*

*} else {*

*executor.execute(new Runnable() {*

*@Override*

*public void run() {*

*next.invokeChannelRead(m);*

*}*

*});*

*}*

对于invokeChannelRead具体对信息使用

*private void invokeChannelRead(Object msg) {*

*if (invokeHandler()) {*

*try {*

*((ChannelInboundHandler) handler()).channelRead(this, msg);*

*} catch (Throwable t) {*

*notifyHandlerException(t);*

*}*

*} else {*

*fireChannelRead(msg);*

*}*

*}*

对于InboundHandler的定义不再介绍。

# ChannelOutboundHandler

在Netty Server中响应信息的输出通过ChannelOutboundHandler来处理，其接口如下：

*public interface ChannelOutboundHandler extends ChannelHandler {*

*void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) throws Exception;*

*void connect(*

*ChannelHandlerContext ctx, SocketAddress remoteAddress,*

*SocketAddress localAddress, ChannelPromise promise) throws Exception;*

*void disconnect(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;*

*void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;*

*void deregister(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;*

*void read(ChannelHandlerContext ctx) throws Exception;*

*void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception;*

*void flush(ChannelHandlerContext ctx) throws Exception;*

*}*

其对服务器发往客户端的数据进行处理，一般用来编码和发送报文到客户端,其核心方法是write。其定义示例如下：

*public class OutboundHandler extends ChannelOutboundHandlerAdapter {*

*@Override*

*public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {*

*super.write(ctx, msg, promise);*

*}*

*}*

https://www.jianshu.com/p/467a9b41833e

https://www.jianshu.com/p/0d0eece6d467

https://blog.csdn.net/xxb249/article/details/78866583