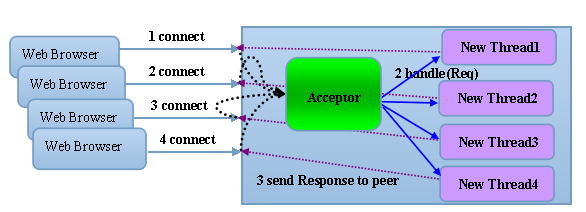
Netty实现原理浅析

Netty是一个高性能、异步事件驱动的NIO框架，提供对TCP、UDP和文件传输的支持，作为一个异步NIO框架，Netty所有的IO操作都是异步非阻塞的，通过FutureListener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。

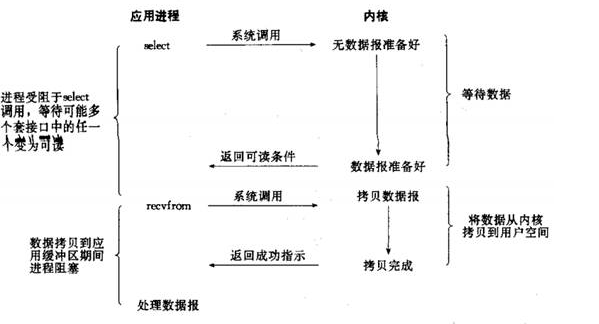
# 1.简述

传统的RPC框架或者基于RMI等方式的远程服务（过程）调用采用同步阻塞IO，当客户端的并发压力或者网络时延增大之后，同步阻塞IO会由于频繁的wait导致IO线程经常性的阻塞，由于线程无法高效的工作，IO处理能力自然下降。BIO通信模型如下图所示：

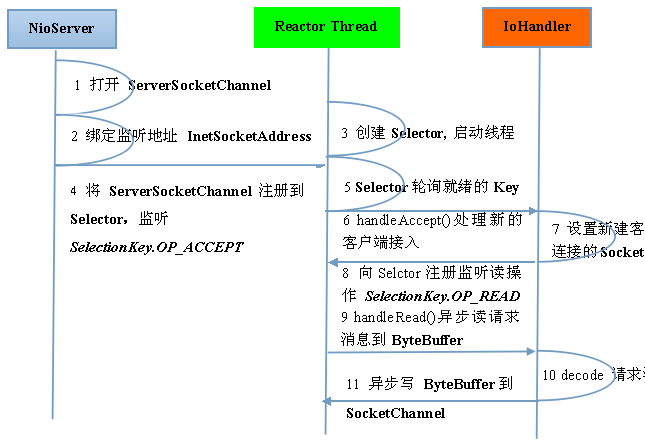


采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，接收到客户端连接后为客户端连接创建一个新的线程处理请求消息，处理完成之后，返回应答消息给客户端，然后销毁线程，这是典型请求-应答模型。最大的问题是不具备弹性伸缩能力，当并发访问量增加后，服务端的线程个数和并发访问数成线性正比，由于线程是JAVA虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能急剧下降，随着并发量的继续增加，可能会发生句柄溢出、线程堆栈溢出等问题，并导致服务器最终宕机。

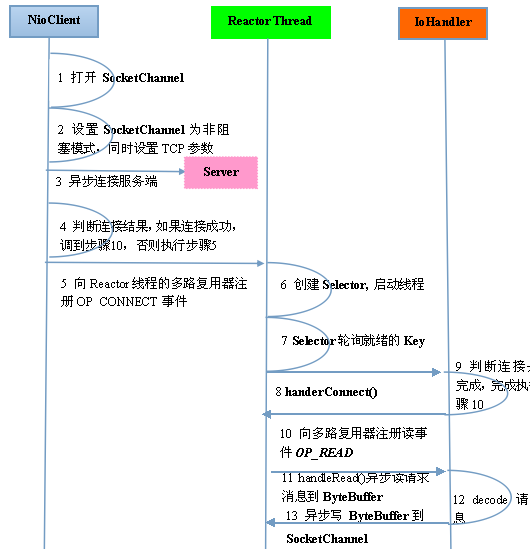
在IO编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者IO多路复用技术进行处理，IO多路复用技术通过把多个IO阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以处理多个客户端请求。JDK NIO的通信模型如下：



与同步阻塞IO的Socket类和ServerSocket类对应，NIO提供了SocketChannel和ServerSocktChannel两种不同的套接字通道实现。这两种新增的通道支持阻塞和非阻塞两种模式：阻塞模式使用简单，但是性能和可靠性不高，非阻塞模式正好相反。一般低负载、低并发的应用程序可以选择同步阻塞IO以降低编程负责的，但是对于高负载、高并发的网络应用，使用NIO的非阻塞模式进行开发。Netty架构按照Reactor模式设计和实现，服务端通信序列图如下：



客户端通信序列图：



Netty的IO线程NioEventLoop聚合多路复用器Selector，可以同时并发处理成百上千个客户端Channel，由于读写操作都是非阻塞的，这就可以充分提高IO线程的运行效率，避免由于频繁IO阻塞导致线程挂起。另外由于Netty采用异步通信模式，一个IO线程可以并发处理N个客户端连接和读写操作，从根本上解决同步阻塞IO—连接线程模型，架构的性能、弹性伸缩能力和可靠性都得到极大提升。

# 2.Reactor模式

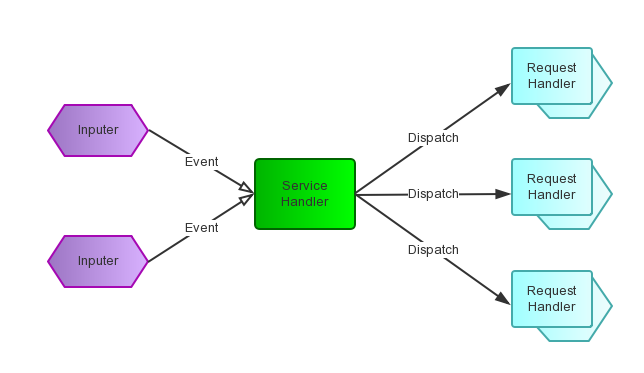
Reactor模式的定义如下：

*“The reactor design pattern is an event handling pattern for handling service requests delivered concurrently by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to associated request handlers“*

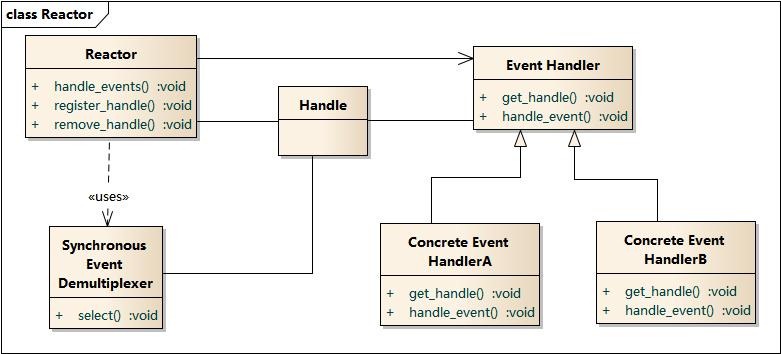
Reacotr设计模式用于处理多客户端并发输入的事件处理模式，其使用多路复用器将请求分发到相关的请求处理handler上。核心概念如下：

1. 事件驱动
2. 一个或者多个并发输入源
3. Service Handler，同步将输入请求（event）多路复用的分发给Request Handler
4. Request Handler

如下图所示:

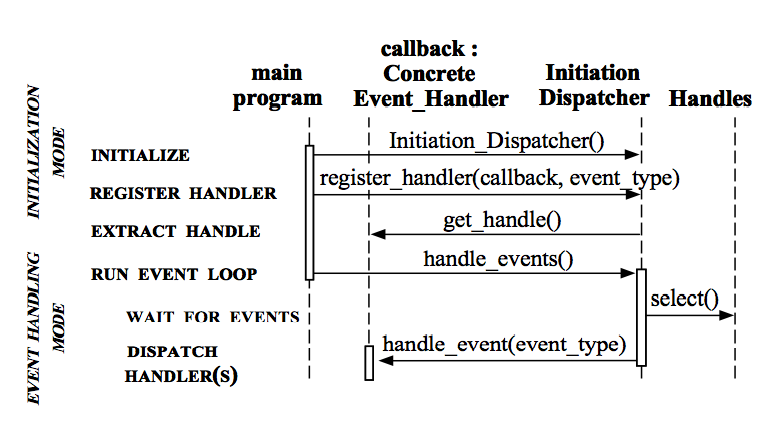


结构上，类似生产者消费者模式，即有一个或者多个生产者将事件放入一个Queue中，消费者主动从Queue中Poll事件来处理。在Reactor模式则没有Queue来做缓冲，每当Event输入到Service Handler之后，Service Handler主动的根据不同的Event类型将其分发给对应的Request Handler来处理。Reactor模式如下所示：



1. Handle，操作系统中的句柄，对资源在操作系统层面上的一种抽象，可以是打开的文件，Socket,Timer等，一般指的是Socket Handle(网络连接，在Java NIO中Channel)。这个Channle注册到Synchronous Event Demultiplexers中，以监听Handle中发生的事件包括CONNECT,READ,WRITE及CLOSE等。
2. Synchronous Event Demultiplexer：阻塞等待一系列的Handle中的事件，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型，这个模块一般使用操作系统的select来实现，在Java NIO中用Selector来封装，当Selector.select返回时，调用Selector的selectKeys方法获取SelectKey(一个有时间发生的Channel及该Channel上的事件类型)
3. Reactor（Intialtion Dispatcher），用于管理Event Handler，用于注册、移除EventHandler等，另外调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handler将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event方法
4. Event Handler，定义事件处理方法，handle\_event，以供Reactor回调使用
5. Concrete Event Handler，事件EventHandler皆苦，实现特定事件处理逻辑。

Reactor各模块之间的交互流程，序列图如下所示：

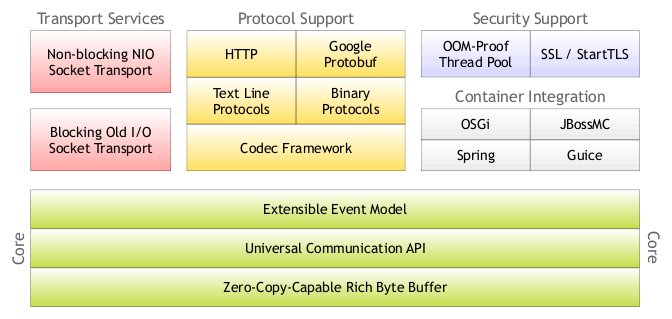


1. 初始化Initialization Dispatcher，并初始化一个Handle到EventHandler的Map
2. 注册EventHandler到Dispatcher中，每个EventHandler包含对应的Handle引用，从而建立Handler到EventHandler的映射
3. 调用Initialization Dispatcher的handle\_events方法以启动Event Loop，调用select方法阻塞等到event发生
4. 当某个Handle的Event发生后，select方法返回，根据返回的Handle找到注册的EventHandler，并回调给EventHandler的handle\_events方法
5. 在EventHandler的handle\_events方法中还可以向InitiationDispatcher中注册新的EventHandler。

# 3.Netty 架构

## 3.1 总体结构

Netty的总体结构图如下所示：



结构图中描述了Netty的核心功能，如下所示：

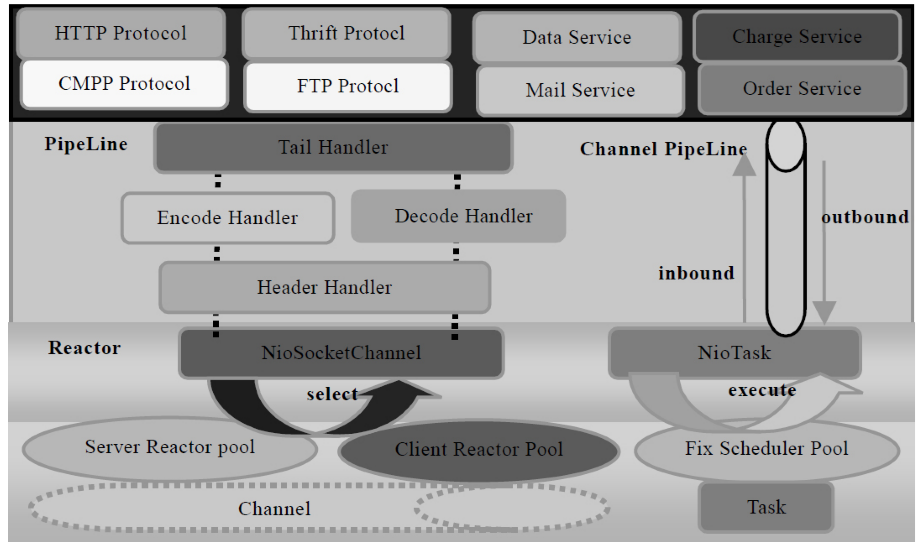
1. Zero-copy-capable Rich Byte Bufffer，Netty基于NIO，但是没有使用NIO的ByteBuffer，而是使用了自定义的ChannelBuffer，表示二进制或文本消息的最基本数据结构。

NIO原生的ByteBuffer处理时，需要对ByteBuffer内容进行拷贝产生新的ByteBuffer，但是在ChannelBuff中Netty提供Composite和Slice两种Buffer操作实现零拷贝。

1. Universal Communication API，Java的Old I/O和New I/O使用互不兼容的API，Socket和DatagramSocket不具有相同的超类型，因此需要使用不同的调用方式。Netty提供统一的API来封装这两种IO
2. Extensible Event Model，定义良好的I/O事件模型，由于严格的层次结构区分了不同的事件类型，Netty允许不破坏现有代码的情况下实现新的事件类型。
3. Security Support，提供安全保障（如SSL/TLS等），还为应用层提供解决OOM的方案。在Netty内部，SslHandler封装了SSLEngine的使用，可以将其插入到ChannelPipeline中。
4. Container Integration，应用于其他主流的开发框架集合，可以和Sprint,OSGI等快速继承
5. Protocol Support，提供多种协议的编解码，包括HTTP, Binary Protocol及Google Protobuf等
6. Transport Service，定义的高级组件来加快开发过程。

## 3.2 逻辑架构

逻辑架构图如下所示：



1. Reactor通信调度层，主要职责是监听网络的读写和连接操作，负责将网络层的数据读取到内存缓冲区中，然后触发各种网络事件，例如连接创建、连接激活、读事件、写事件等。将这些事件触发到PipeLine中，由PipeLine管理的职责链后续进行处理
2. PipeLine，PipeLine是职责链 ChannelPipeLine，负责事件在职责链中的有序传播，同时负责动态的编排职责链。职责链可以选择监听和处理自己关心的事件，也可以拦截处理和向后/向前传播时间。不同应用的Handler节点的功能也不同，通常条件下往往会开发编解码Handler用于消息的编解码，它可以将外部的协议消息转换成内部的POJO对象，这样上层业务则只需要关心处理业务逻辑即可，不需要感知底层的协议差异和线程模型差异，实现层面的分层隔离
3. 业务逻辑编排层，一类是纯粹的业务逻辑编排，还有一类是其他的应用层协议插件，用户特定协议相关的会话和链路管理。
4. 分层设计，架构的不同层面，需要关系和处理的对象不同，通常情况下，对于业务开发者，只需要关心职责链的拦截和业务Handler的编排，因为应用层协议栈往往是开发一次，到处运行，实际上对于业务开发者来说，只需要关心服务层的业务逻辑开发即可。各种应用协议以插件形式提供，只有协议人员需要关协议插件，对于其他业务开发人员来说，只需要关心业务逻辑定制即可。这种分层的架构设计理念实现了NIO架构设计理念实现NIO框架各层之间的解耦，便于上层业务协议栈的开发和业务逻辑的定制。

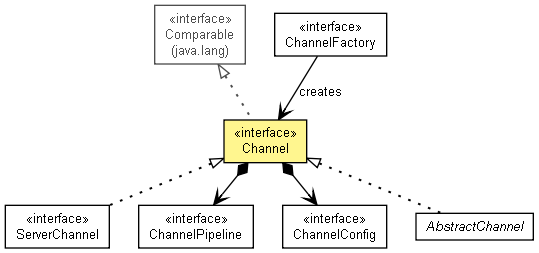
# 4 Netty核心概念及类库简介

Netty的主要组成模块包括Channels,Handlers,Events,Futures和Callbacks等，这些模块代表不同类型的概念：资源、逻辑和通知等，应用使用这些模块来获取网络及网络上的数据。

## 4.1.Channel

Channel是Netty的核心概念之一，是网络通信的主体，Java NIO的一个基本抽象，代表一个连接如硬件设备、文件、网络Socket等实体的开放连接，或者是能够完成一种或多种读写的不同I/O操作的程序，由它负责同时对端进行网络通信、注册和数据操作等功能。Channel可以看做是通道，通过其读取和写入数据，网络数据通过Channel读取和写入，通道与流的不同之处在于通道是双向的。而流只是一个方向上移动（一个流必须是InputStream或者OutputStream的子类），而通道可以用于读、写或者同时用于读写。因为Channel是全双工的，可以比流更好的映射底层操作系统API，特别是在UNIX网络编程模型中，底层操作系统的通道都是全双工，同时支持读写操作。

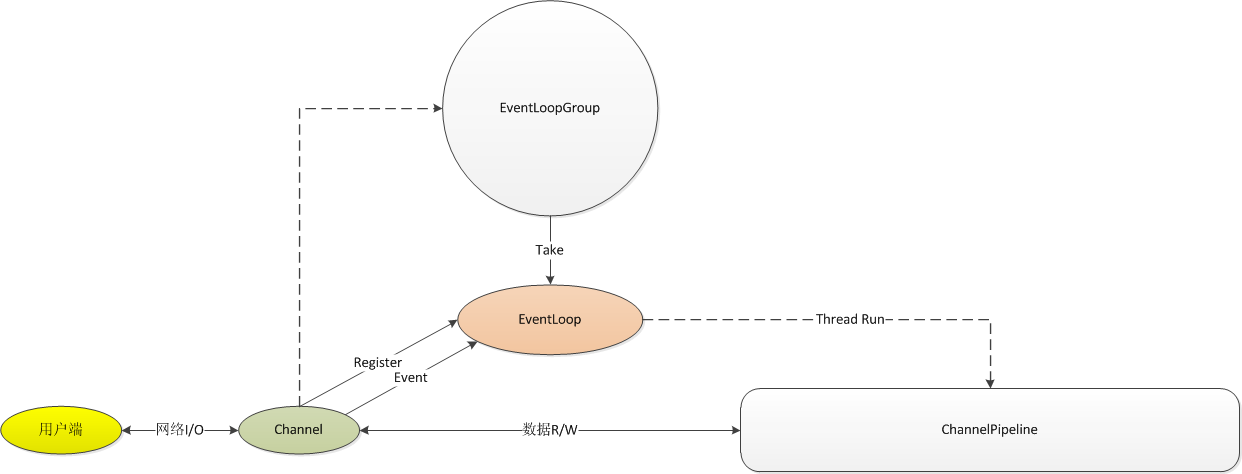
在Netty中，Channel的相关接口的类结构图如下：



Channel具有以下特征：

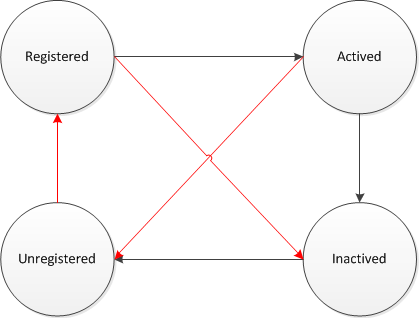
1. 提供状态信息，包括打开、连接和关闭
2. 通道的主要操作：读、写、连接和绑定（bind）
3. 所有的IO操作都是异步的，调用诸如read,write方法，并不保证IO操作完成，但会返回一个凭证，在IO操作成功，取消会失败会记录在该凭证中
4. Channel有父子关系，SocketChannel是通过ServerSocketChannel接受创建的，故SocketChannel的parent方法返回的就是ServerSocketChannel
5. 在Channel使用完毕后，调用close方法，释放通道占用的资源
6. 通过ChannelConfig可以得到Channel配置信息
7. 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。

Channel的工作原理如下：

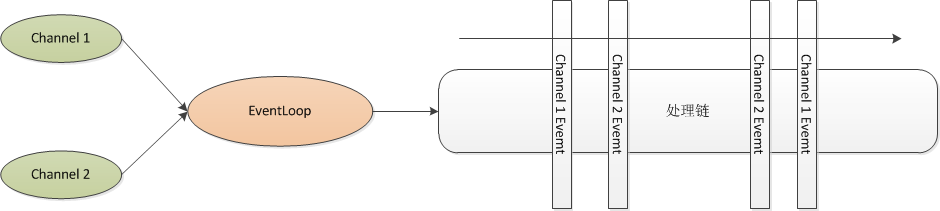


1. 一旦用户端连接成功，将新建一个Channel同该用户端进行绑定
2. Channel从EventLoopGroup获得一个EventLoop，并注册该EventLoop，channel生命周期内都和该EventLoop在一起，注册时获得selectionkey
3. Channel同用户端进行网络连接、关闭和读写，生成对应的event(改变selectionKey)，触发eventLoop调度线程进行执行
4. 如果是读事件，执行线程调度pipeline来处理用户业务逻辑

对于Channel，包含注册、活跃、非活跃和非注册状态，在一般情况下从注册->活跃->非活跃->非注册，单用户可以从eventLoop取消和重注册channel，因此在此情况下活跃->非注册->注册，状态转换如下所示：



多个channel可以注册到一个eventLoop中，所有的操作都是顺序执行的，EventLoop会依据channel的事件调用channel的方法进行相关操作，每个channel的操作和处理在eventLoop中都是顺序的，如下图所示：

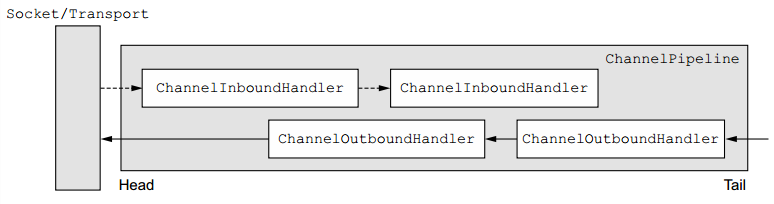


http://blog.csdn.net/suifeng3051/article/details/28861883

## 4.2 ChannelHandler

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。

1. ChannelHandler，应用程序的核心，可以当做一个通用的容器，处理进来的事件（包括数据），其类图如下所示：



可以处理任何类型的操作。Handler和Servlet有些类似，一个ChannelHandler处理完接收到的数据会传到下一个Handler,或者传递给下一个。

ChannelHandler有两个子类：ChannelInboundHandler和ChannelOutboundHandler，这两个类对应两个数据流向，如果数据流入应用程序看做是inbound，相反便是outbound。ChannelPipeline可以把两种handler(ChannelInboundHandler和ChannelOutboundHnalder)混合在一起，当一个数据流入ChannelPipeline时，会从ChannelPipeline头部开始传给第一个ChannelInboundHandler，当第一个处理完后再传给下一个，一直传递管道的尾部。与之相对应的是，当数据写出时，从管道的尾部开始，先经过管道尾部的最后一个ChannelOutboundHandler，当处理完后传递给前一个ChannelOutboundHandler。数据在各个Handler之间传递，需要调用方法中传递的ChannelHandlerCountext来操作，在netty的API中提供了两个基类：ChannelOutboundHandlerAdapter和ChannelInboundHandlerAdapter，它们仅仅实现了调用ChannelHandlerContext来把消息传递给下一个handler。

当一个ChannelHandler被加入到ChannelPipeline中时，便会获得一个ChannelHandlerContext的引用，而ChannelHandlerContext可以用来读写Netty中的数据流。因此可以有两种方式来发送数据，一种是数据直接写入Channel，一种是把数据写入ChannelHandlerContext，区别是写入Channel的话，数据流会从Channel的头开始传递，如果写入ChannelHandlerContext的话，数据流会写入管道中的下一个Handler。

Netty的强大和灵活之处在于基于Pipeline的自定义handler机制，可以基于添加插件的方式组合多种handler，例如处理Http数据，在pipeline中添加HTTP解编码的Handler，然后再添加业务逻辑handler，在主程序中handler的初始化过程如下：

*BootStrap.handler(new ChannelInitializer<SocketChannel>() {*

*@Override*

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

*ChannelPipeline p = ch.pipeline();*

*if (sslCtx != null) {*

*p.addLast(sslCtx.newHandler(ch.alloc(), HOST, PORT));*

*}*

*p.addLast(new EchoClientHandler());}});*

Bootstrap.handler方法接收一个ChannelHandler，传递一个ChannelInitializer匿名类，其如下所示：

*@Sharable*

*public abstract class ChannelInitializer<C extends Channel> extends ChannelInboundHandlerAdapter {*

*protected abstract void initChannel(C ch) throws Exception;*

*@Override*

*@SuppressWarnings("unchecked")*

*public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {*

*initChannel((C) ctx.channel());*

*ctx.pipeline().remove(this);*

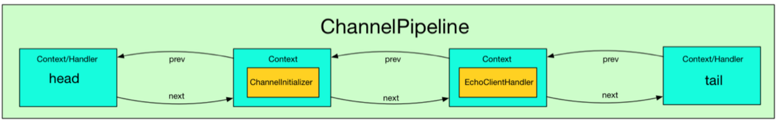
*ctx.fireChannelRegistered();*

*}*

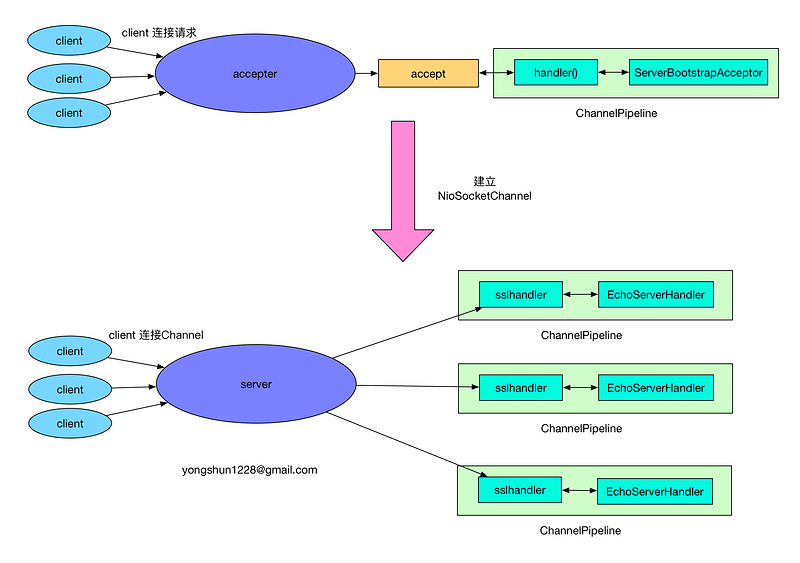
*...*

*}*

ChannelInitializer是一个抽象类，实现initChannel，然后在该方法中向ChannelPipeline添加handler。在ChannelPiple中添加Handler后，如下所示：



在ChannelHandlerContext中，有next和prev context可以形成一个Handler链表。添加流程如下：

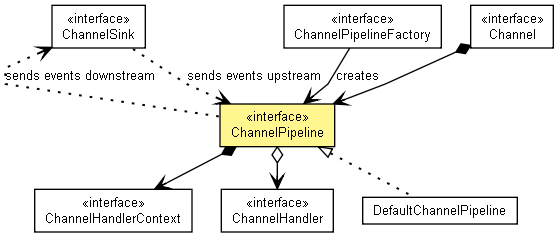


https://segmentfault.com/a/1190000007283053

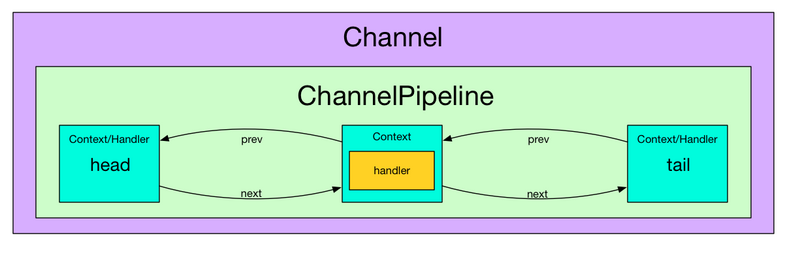
## 4.3 ChannelPipeline

ChannelPipeline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：

1. 当Channel注册到selector时，触发pipeline的fireChannelRegistered方法
2. 当Channel的socket绑定完成时，触发pipeline的fireChannelActive方法
3. 当有客户端请求时，触发pipeline的fireChannelRead方法
4. 当本地客户端请求，pipeline执行完fireChannelRead，触发pipeline的fireChannelReadComplte方法。

其中DefaultChannelHandlerContext保存了当前Handler的上下文，如channel、pipeline等信息，默认实现head和tail，定义如下：

*public class DefaultChannelPipeline implements ChannelPipeline {*

*final AbstractChannelHandlerContext head;*

*final AbstractChannelHandlerContext tail;*

*private final Channel channel;*

*private Map<EventExecutorGroup, EventExecutor> childExecutors;*

*......*

*protected DefaultChannelPipeline(Channel channel) {*

*this.channel = ObjectUtil.checkNotNull(channel, "channel");*

*succeededFuture = new SucceededChannelFuture(channel, null);*

*voidPromise = new VoidChannelPromise(channel, true);*

*tail = new TailContext(this);*

*head = new HeadContext(this);*

*head.next = tail;*

*tail.prev = head;*

*}*

*}*

TailContext实现了ChannelOutboundHandler接口，HeadContext实现了ChannelInboundHandler接口，head和tail形成一个链表。对于Inbound操作，当Channel注册到selector时，触发pipeline的fireChannelRegistered，从head开始遍历，找到实现了ChannelInboundHandler接口的handler，并执行fireChannleRegistered方法。

*static void invokeChannelRegistered(final AbstractChannelHandlerContext next) {*

*EventExecutor executor = next.executor();*

*if (executor.inEventLoop()) {*

*next.invokeChannelRegistered();*

*} else {*

*executor.execute(new Runnable() {*

*@Override*

*public void run() {*

*next.invokeChannelRegistered();*

*}*

*});*

*}*

*}*

*private void invokeChannelRegistered() {*

*if (invokeHandler()) {*

*try {*

*((ChannelInboundHandler) handler()).channelRegistered(this);*

*} catch (Throwable t) {*

*notifyHandlerException(t);*

*}*

*} else {*

*fireChannelRegistered();*

*}*

*}*

ChannelHandler可以实时修改ChannelPipeline的布局，通过添加、移除、替换其他ChannelHandler，方法包括：

|  |  |
| --- | --- |
| addFirst addBefore addAfter addLast | 添加ChannelHandler到ChannelPipeline |
| Remove | 从ChannelPipeline移除ChannelHandler |
| replace | 在ChannelPipeline替换另外一个ChannleHandler |

ChannelPipeline的修改示例如下：

*ChannelPipeline pipeline = null; // get reference to pipeline;*

*FirstHandler firstHandler = new FirstHandler(); //1*

*pipeline.addLast("handler1", firstHandler); //2*

*pipeline.addFirst("handler2", new SecondHandler()); //3*

*pipeline.addLast("handler3", new ThirdHandler()); //4*

*pipeline.remove("handler3"); //5*

*pipeline.remove(firstHandler); //6*

*pipeline.replace("handler2", "handler4", new ForthHandler()); //6*

1. 创建FirstHandler实例
2. 添加该实例作为handler1到ChannelPipeline
3. 添加SecondHandler实例作为handler2到ChannelPipeline的第一个槽，意味着替换之前已经存在的handler1
4. 添加ThirdHandler实例作为handler3到ChannelPipeline的最后一个槽
5. 移除handler3
6. 通过引用移除FirstHandler
7. 将作为handler2的SecondHandler实例替换为handler4的FourthHandler

通常每个ChannelHandler添加到ChannelPipeline将处理事件传递到EventLoop(I/O线程)，至关重要的是不要阻塞这个线程，将会影响整体处理I/O。如果一个定制的EventExecutorGrou p传入事件将由含在这个EventExecutorGroup中的EventExecutor之一进行处理，并且从Channel的EventLoop本身离开，一个默认实现是DefaultEventExecutorGroup。下面是ChannelPipeline中关于ChannelHandler的操作：

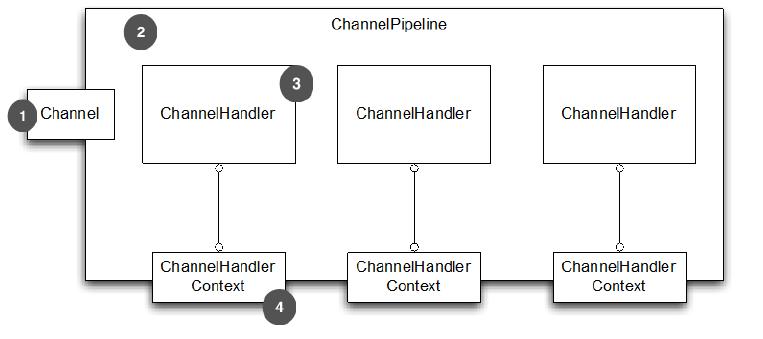
|  |  |
| --- | --- |
| get(…) | 返回一种类型或者name的ChannelHandler |
| context(…) | 返回绑定到ChannelHandler的ChannelHandlerContext |
| names().iterator | 返回ChannelPipeline中所有ChannelHandler的名称 |

https://www.ctolib.com/docs/sfile/essential-netty-in-action/CORE%20FUNCTIONS/ChannelHandler%20and%20ChannelPipeline.html

## 4.4 ChannelContext

接口ChannelHandlerContext代表ChannelHandler和ChannelPipeline之间的关联，并在ChannelHandler添加到ChannelPipeline时创建一个实例。ChannelHandlerContext的主要功能是管理通过同一个ChannelPipeline关联的ChannelHandler之间的交互。

ChannelHandlerContext的方法也会出现在Channel和ChannelPipeline本身，如果通过Channel或ChannelPipeline的实例来调用这些方法，就会在整个pipeline中传播，但是通过ChannelHandlerContext实例上调用，就只会从当前的ChannelHandler开始并传播到相关管道的下一个有处理能力的ChannelHandler。ChannelPipeline,Channel,ChannelHandler和ChannelHandlerContext的关系如下图所示：



1. Channel绑定到ChannelPipeline
2. ChannelPipeline绑定到包含ChannelHandler的Channel
3. ChannelHandler
4. 当添加ChannelHandler到ChannelPipeline时，ChannelHandlerContext被创建

下面展示了从ChannelHandlerContext获取到Channel的引用，通过调用Channel的write方法来触发一个写事件到通过管道的流中：

*ChannelHandlerContext ctx = context;*

*Channel channel = ctx.channel(); //1*

*channel.write(Unpooled.copiedBuffer("Netty in Action",*

*CharsetUtil.UTF\_8)); //2*

1. 得到与ChannelHandlerContext关联的Channel引用
2. 通过Channel写缓存

下面展示了从ChannelHandlerContext关联的ChannelPipeline的相同示例：

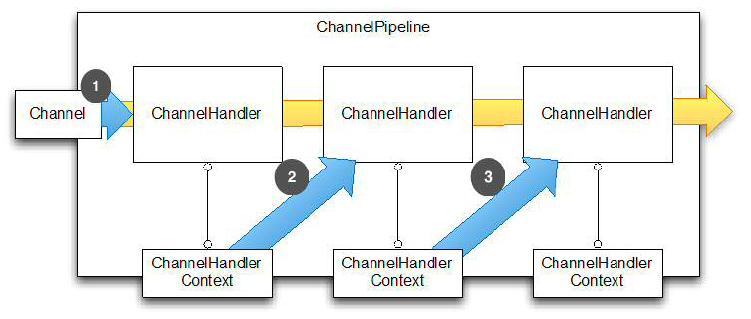
*ChannelHandlerContext ctx = context;*

*ChannelPipeline pipeline = ctx.pipeline(); //1*

*pipeline.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF\_8)); //2*

1. 得到与ChannelHandlerContext关联的ChannelPipeline引用
2. 通过ChannelPipeline的缓冲区

流在上述的两个例子中，虽然在Channel或者ChannePipeline上调用write都会把事件在整个管道传播，但是在ChannelHandler级别上从一个处理程序转到下一个却要通过在ChannelHandlerContext调用方法实现，如下图所示：



1. 事件传递给ChannelPipeline的第一个ChannelHandler
2. ChannelHandler通过关联的ChannelHandlerContext传递事件给ChannelPipeline中的下一个
3. 逐渐重复过程2

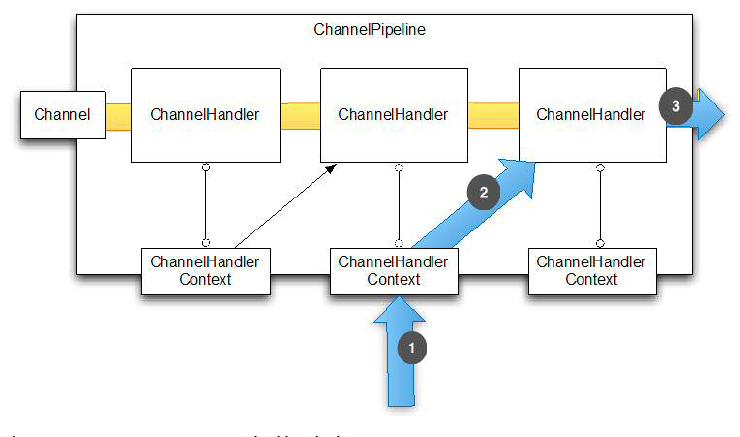
想实现从一个特定的ChannelHandler开始处理，必须引用与此ChannelHandler的前一个ChannelHandler关联的ChannelHandlerContext，这个ChannelHandlerContext将会调用与自身关联的ChannelHandler的下一个ChannelHandler，使用场景如下：

*ChannelHandlerContext ctx = context;*

*ctx.write(Unpooled.copiedBuffer("Netty in Action",CharsetUtil.UTF\_8));*

1. 获得ChannelHandlerContext的引用
2. write将会把缓冲区发送到下一个ChannelHandler

如下图，消息将会从下一个ChannelHandler开始流过ChannelPipeline绕过所有它之前的ChannelHandler：



1. ChannelHandlerContext方法调用
2. 事件发送到下一个ChannelHandler
3. 经过最后一个ChannelHandler后，事件从ChannelPipeline移除

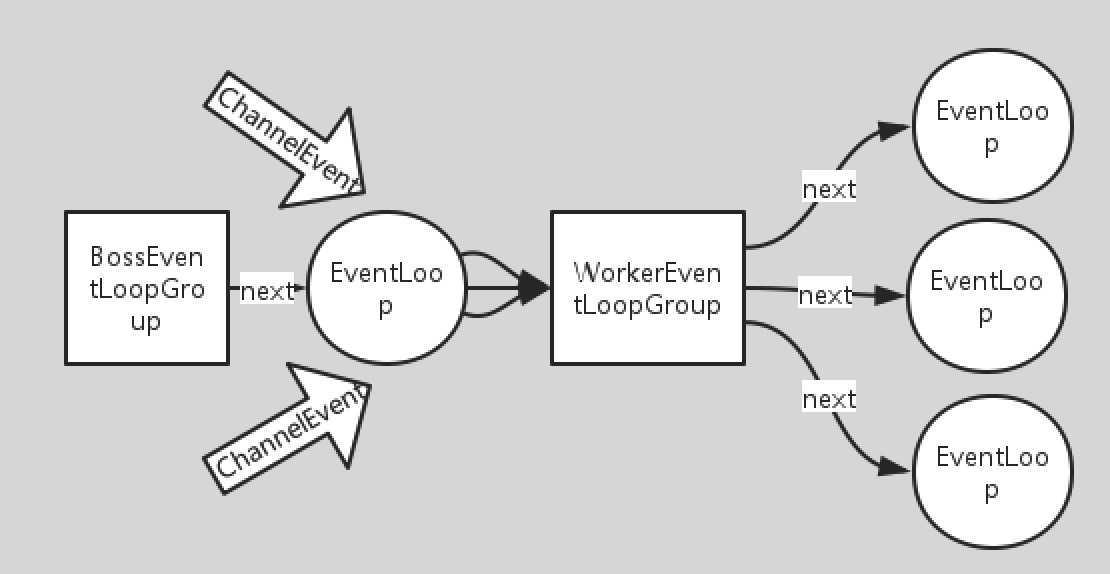
## 4.5 EventLoopGroup/EventLoop

1）基本组件，Netty线程模型是Reactor模式的一种，基于JAVA NIO提供的API实现，在JAVA NIO方面Selector提供基础，线程模型的主要组件：

1. Selector
2. EventLoopGroup/EventLoop
3. ChannelPipeline

其中，Selector是JAVA NIO提供的SelectableChannel多路复用器，内部维护这三个SelectionKey集合，负责配置select操作将就绪的IO事件分离处理。

EventLoopGroup是EventLoop的抽象，EventLoop维护着Selector实例，通常一个服务端口即ServerSocketChannel对应一个Selector和EeventLoop线程，其中EventLoop分成两个角色：BossEventGoup负责接收客户端连接，将SocketChannel交给WorkerEventLoopGroup进行IO处理，示意图如下所示：



BossEventLoop维护一个注册了ServerSocketChannel的Selector实例，不断轮询Selector将连接事件分离处理，通常是OP\_ACCEPT事件，然后将accept得到的SocketChannel交给WorkerEventLoopGroup，然后由next选择其中一个EventLoopGroup来将SocketChannel注册到其维护的Selector并对后续IO事件进行处理，EventLoop的实现涵盖IO事件的分离和分发。

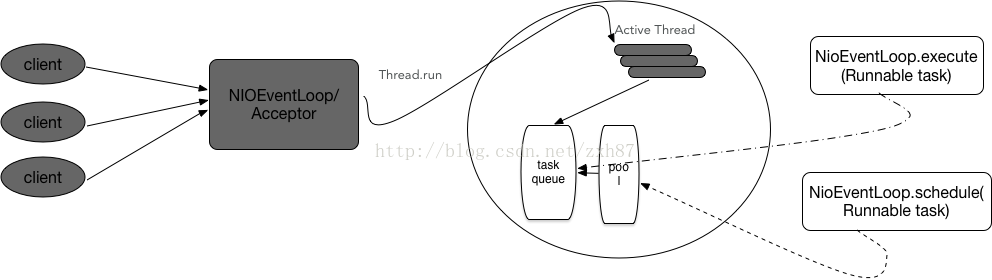
2）类关系

Netty的EventLoop具体的继承关系如下图所示：



3）线程模型

在这个模型中，EventLoop是一个循环执行的线程驱动，任务（Runnable或Callable）可以直接提交给EventLoop，以便立即或者有计划的执行。其线程模型如下：



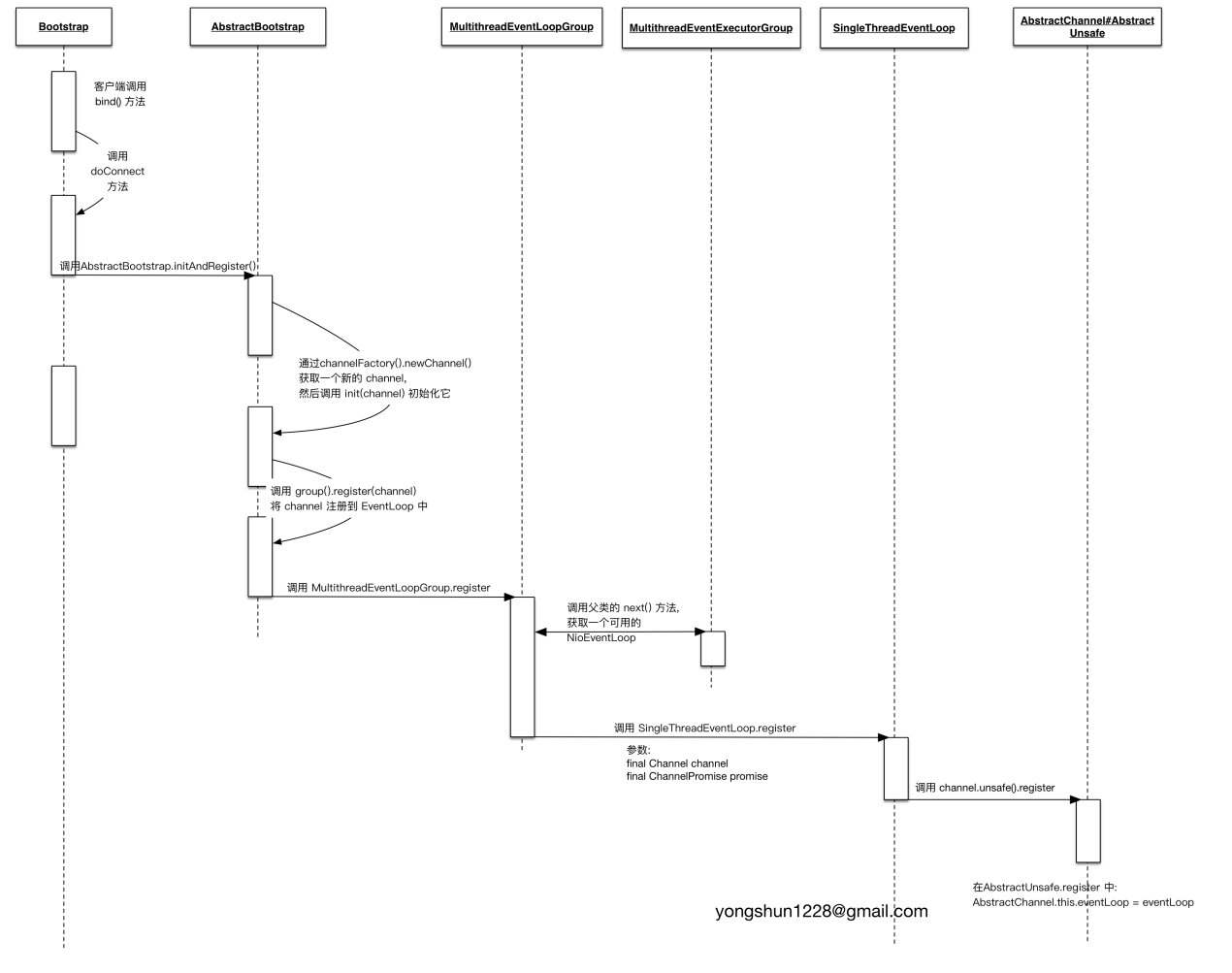
启动两个NioEventLoopGroup，一个用于监听客户端的连接：

1. 接收客户端tcp请求，初始化Channel参数
2. 将链路状态变更时间通知给ChannelPipeline

用于处理IO操作的Reactor线程池职责如下：

1. 异步读取通信端数据，发送读事件给ChannelPipeline
2. 异步发送消息给通信对端，调动ChannelPipeline的消息发送接口
3. 执行系统调用Task
4. 执行定时任务Task，例如链路空闲状态监测定时任务

4）EventLoop与Channel的关联，顺序图如下：



当调用AbstractChannel.unsafe().register后，就完成Channel和EventLoop的关联，实现如下：

*public final void register(EventLoop eventLoop, final ChannelPromise promise) {*

*.....*

*AbstractChannel.this.eventLoop = eventLoop;*

*if (eventLoop.inEventLoop()) {*

*register0(promise);*

*} else {*

*try {*

*eventLoop.execute(new Runnable() {*

*@Override*

*public void run() {*

*register0(promise);*

*}*

*});*

*} catch (Throwable t) {*

*......*

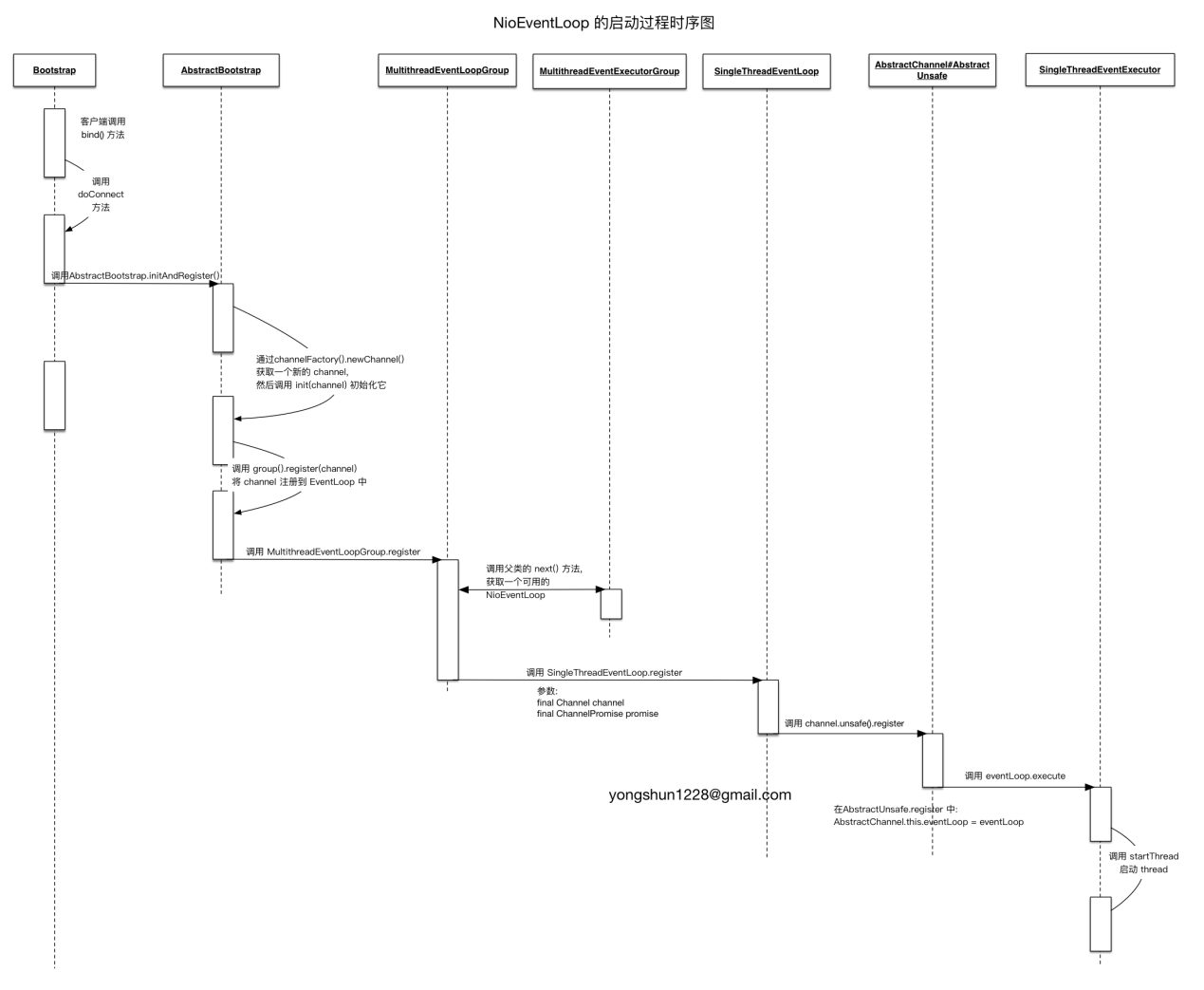
*}*

*}*

*}*

在register操作时，将一个EventLoop赋值给AbstractChannel的eventLoop字段，完成关联。

1. EventLoop的启动



1. Netty的IO循环处理

在Netty中，EventLoop需要负责两个工作：IO线程，负责相应的IO操作；任务线程，执行taskQueue任务。在NioEventLoop中有Selector线程，不断调用select方法，查询当前是否有就绪的IO事件，Selector使用流程如下：

1. 通过Selector.open打开一个Selector
2. 将Channel注册到Selector中，并设置需要监听的事件，

*public void register(final SelectableChannel ch, final int interestOps, final NioTask<?> task) {*

*try {*

*ch.register(selector, interestOps, task);*

*} catch (Exception e) {*

*throw new EventLoopException("failed to register a channel", e);*

*}*

*}*

其中interestOps为需要指定感兴趣的事件：

* Connnect，连接事件，对应于SelectionKey.OP\_CONNECT
* Accept，确认事件，对应于SelectionKey.OP\_ACCEPT
* Read，读事件，对应于SelectionKey.OP\_READ
* Write，即写事件，，对应于SelectionKey.OP\_WRITE

1. Selector不停的遍历，获取READY的Channel

https://segmentfault.com/a/1190000007403873

## 4.6 Bootstrap

在Netty中，客户端和服务端都是从由Bootstrap类启动，可配置的参数：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段 | 类型 | 说明 | Server模式 | Client模式 |
| options | Map | Channel的配置项 | ServerChannel |  |
| childOptions | Map | Channel的配置项 | Channel |  |
| attr | Map | 自定义channel属性 | ServerChannel | Channel |
| childAttrs | Map | 自定义channel属性 | Channel |  |
| handler | ChannelHandler | 连接处理器 | ServerChannel | Channel |
| childHandler | ChannelHandler | 连接处理器 | Channel | Channel |
| group | EventLoopGroup | 注册并处理连接 | ServerChannel | Channel |
| childGroup | EventLoopGroup | 注册并处理连接 | Channel |  |
| channelFactory | ChannelFactory | 生成连接对象的工厂类 | ServerChannel | Channel |

ChannelFactory所有字段都分成xxx和childxxx两个相对应的字段，从名称上看可以区分字段的作用范围。如设置SO\_REUSEADDR参数，该参数作用于ServerSocket，则设置时调用：

*option(ChannelOption.SO\_REUSEADDR, true)*

对于Server端来说，比较常见的几个设置：

*SO\_KEEPALIVE、SO\_REUSEADDR、TCP\_NODELAY、SO\_BACKLOG*

1. Bootstrap类结构图



Bootstrap的结构比较简单，涉及的类和接口很少，BootStrap则是客户端程序用户端引导类，ServerBootstrap是服务端程序用的引导类。

1. AbstractBootstrap

类描述如下：

*volatile EventLoopGroup group;*

*@SuppressWarnings("deprecation")*

*private volatile ChannelFactory<? extends C> channelFactory;*

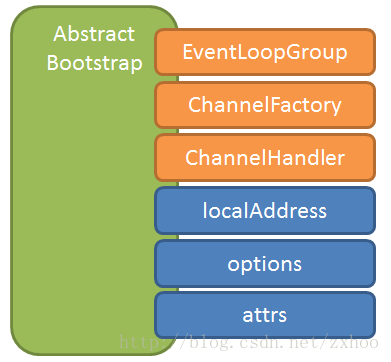
*private volatile SocketAddress localAddress;*

*private final Map<ChannelOption<?>, Object> options = new LinkedHashMap<ChannelOption<?>, Object>();*

*private final Map<AttributeKey<?>, Object> attrs = new LinkedHashMap<AttributeKey<?>, Object>();*

*private volatile ChannelHandler handler;*

AbstractBootstrap这个抽象Builder共需要6个part，如下图所示：



AbstractBootstrap通过ChannelFactory创建Channel实例，channel(channelClass)方法看起来好像是没设置一个Channel，但实际上只是设置了默认的ChannelFactory实现：

*public T newChannel() {*

*try {*

*return clazz.getConstructor().newInstance();*

*} catch (Throwable t) {*

*throw new ChannelException("Unable to create Channel from class " + clazz, t);*

*}*

*}*

1. ServerbootStrap分析

Netty采用了reactor设计模式，其中mainReactor负责连接的建立，连接建立后交由subReactor处理，SubReactor负责处理读写等具体的事件，这两个reactor一般称为bossGroup和workerGroup，下面是HttopSnoopServer类中main的主要代码：

*EventLoopGroup bossGroup = new NioEventLoopGroup(1);*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*try {*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup)*

*.channel(NioServerSocketChannel.class)*

*.handler(new LoggingHandler(LogLevel.INFO))*

*.childHandler(new HttpSnoopServerInitializer(sslCtx));*

*Channel ch = b.bind(PORT).sync().channel();*

*ch.closeFuture().sync();*

*} finally {*

*bossGroup.shutdownGracefully();*

*workerGroup.shutdownGracefully();*

*}*

这里BoosGroup只启用了一个线程，一个端口只能创建一个ServerChannel，该ServerChannel的整个生命周期都在bossGroup中。在上述示例中，handle设置为LogginHandler，表示在ServerChannel的处理链中加入了日志记录。childHandler设置为HttpSnoopServerInitializer，即用户连接使用HttpSnoopServerInitializer进行处理。

1. 初始化完成开始调用bind(port)方法，bind会对各个参数进行验证，比如channelFactory是否设置group,childGroup是否设置，端口是否设置，验证通过后，最终调用doBind方法

*private ChannelFuture doBind(final SocketAddress localAddress) {*

*final ChannelFuture regFuture = initAndRegister(); //初始化并注册channel*

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

*if (regFuture.cause() != null) {*

*return regFuture;*

*}*

*//注册完成后调用doBind0,否则添加一个注册事件的监听器，该监听器在监听注册完成后也会触发doBind0操作*

*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;*

*} else {*

*// Registration future is almost always fulfilled already, but just in case it's not.*

*final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);*

*regFuture.addListener(new ChannelFutureListener() {*

*@Override*

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

*Throwable cause = future.cause();*

*if (cause != null) {*

*promise.setFailure(cause);*

*} else {*

*promise.registered();*

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

*}*

*}*

*});*

*return promise;*

*}*

*}*

1. doBind首先会调用initAndRegister方法，代码如下所示：

*final ChannelFuture initAndRegister() {*

*Channel channel = null;*

*try {*

*channel = channelFactory.newChannel();*

*init(channel);*

*} ...*

*ChannelFuture regFuture = config().group().register(channel); //注册到group中*

*if (regFuture.cause() != null) {*

*if (channel.isRegistered()) {*

*channel.close();*

*} else {*

*channel.unsafe().closeForcibly();*

*}*

*}*

*return regFuture;*

*}*

channelFactory.newChannel方法创建一个NioServerSocketChannel实例，初始化时调用openServerSocketChannel打开一个ServerSocketChannel，同时配置为configureBlocking(false)将其IO模式设置为非阻塞。

ServerSocketChannel的初始化分别为：设置option，设置attr,如果设置handler，将加入到处理链中，这里为LoggingHnalder。最后会加入一个ChannelInitializer，主要功能是获取客户端连接后对连接进行初始化。

初始化完成后立即将ServerChannel注册到bossGroup中，调用代码为AbstractChannel.A

bstractUnsafe.register，如下所示：

*public final void register(EventLoop eventLoop, final ChannelPromise promise) {*

*.....*

*AbstractChannel.this.eventLoop = eventLoop;*

*if (eventLoop.inEventLoop()) {*

*register0(promise);*

*} else {*

*try {*

*eventLoop.execute(new Runnable() {*

*@Override*

*public void run() {*

*register0(promise);*

*}*

*});*

*} catch (Throwable t) {*

*}*

*}*

上代码最重要的部分是EventLoop的封装，然后调用register0

*private void register0(ChannelPromise promise) {*

*try {*

*if (!promise.setUncancellable() || !ensureOpen(promise)) {*

*return;*

*}*

*boolean firstRegistration = neverRegistered;*

*doRegister(); //真正注册方法*

*neverRegistered = false;*

*registered = true;*

*pipeline.invokeHandlerAddedIfNeeded(); //开始接收任务*

*safeSetSuccess(promise);*

*pipeline.fireChannelRegistered();*

*if (isActive()) {*

*if (firstRegistration) {*

*pipeline.fireChannelActive();*

*} else if (config().isAutoRead()) {*

*beginRead();*

*}*

*}*

*} catch (Throwable t) {}*

*}*

1. doRegister方法，将ServerSocketChannel注册到Selector中，如下所示：

*protected void doRegister() throws Exception {*

*boolean selected = false;*

*for (;;) {*

*try {*

*selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);*

*return;*

*} catch (CancelledKeyException e) {}*

*}*

*}*

1. ServerBootstrapAcceptor类主要作用是接收客户端连接后，使用childOpts和childAttrs对连接初始化，然后将连接注册到childGroup中，channelRead方法如下：

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

*final Channel child = (Channel) msg;*

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

*setChannelOptions(child, childOptions, logger);*

*for (Entry<AttributeKey<?>, Object> e: childAttrs) {*

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

*}*

*try {*

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

*@Override*

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

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

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

*}*

*}*

*});*

*} catch (Throwable t) {*

*forceClose(child, t);*

*}*

*}*

1. 回到主流程，如果第一次启动启动触发channelActive方法，本例中触发LoggerHandler.channleActive，调动完成后AbstractBootstrap.doBind0方法

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

*@Override*

*public void run() {*

*if (regFuture.isSuccess()) {*

*channel.bind(localAddress, promise).addListener(ChannelFutureListener.CLOSE\_ON\_FAILURE);*

*} else {*

*promise.setFailure(regFuture.cause());*

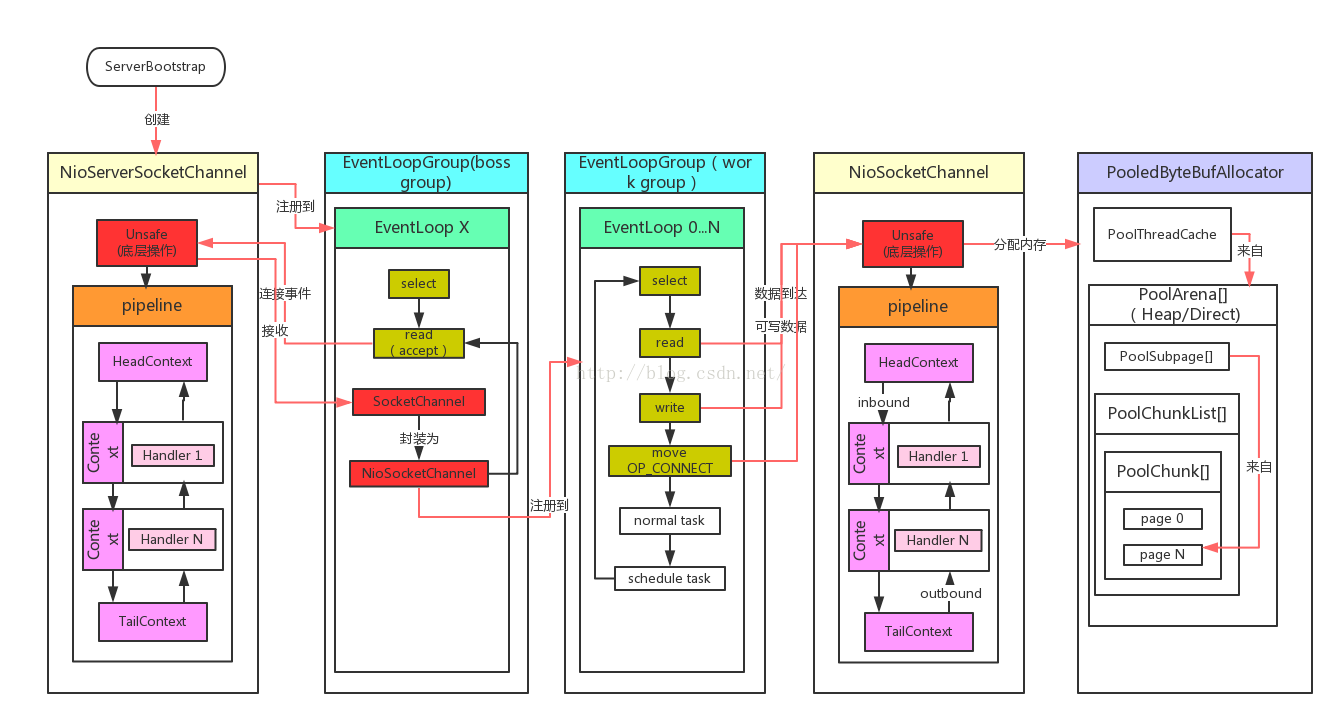
*}*

*}*

*});*

doBind0最终调用channel.bind方法对执行端口进行监听，为了保证线程安全，channel所有方法都需要到EventLoop中执行。

总体流程如下图所示：



整个启动过程：

1. 应用设置启动所需的各个参数
2. 应用调用bind(port)启动监听,bind过程如下
3. 验证启动参数设置是否正确，调用doBind
4. boBind创建的NioServerSocketChannel，并对其进行初始化，包括创建一个实际的ServerChannel，设置为非阻塞模式，创建底层处理实例NioMessageUnsafe，创建pipeline
5. pipeline中加入ChannelInitializer，该ChannelInitializer往pipeline中加入ServerBootStrap

Acceptor用于接受客户连接后设置其初始化参数，然后注册到childGroup

1. 将NioServerSocketChannel注册到bossGroup，此时bossGroup被激活开始接收任务及IO事件
2. 往EventLoop中添加一个任务，该任务的内容为将之前创建的ServerSocket绑定到指定端口
3. 绑定端口后增加一个任务，该任务内容为注册NioServerSocketChannel关注事件AP\_ACCEPT到SelectKey中，到此服务端可以接收到来自客户端的请求。
4. BootStrap分析

客户端的初始化如下：

*EventLoopGroup group = new NioEventLoopGroup();*

*try {*

*Bootstrap b = new Bootstrap();*

*b.group(group)*

*.channel(NioSocketChannel.class)*

*.handler(new HttpSnoopClientInitializer(sslCtx));*

*Channel ch = b.connect(host, port).sync().channel();*

*HttpRequest request = new DefaultFullHttpRequest(*

*HttpVersion.HTTP\_1\_1, HttpMethod.GET, uri.getRawPath());*

*request.headers().set(HttpHeaderNames.HOST, host);*

*request.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.CLOSE);*

*request.headers().set(HttpHeaderNames.ACCEPT\_ENCODING, HttpHeaderValues.GZIP);*

*request.headers().set(*

*HttpHeaderNames.COOKIE,*

*ClientCookieEncoder.STRICT.encode(*

*new DefaultCookie("my-cookie", "foo"),*

*new DefaultCookie("another-cookie", "bar")));*

*ch.writeAndFlush(request);*

*ch.closeFuture().sync();*

*} finally {*

*group.shutdownGracefully();*

*}*

调用BootStrap.connect后，首先调用init方法将handler添加到Pipeline的末尾：

*void init(Channel channel) throws Exception {*

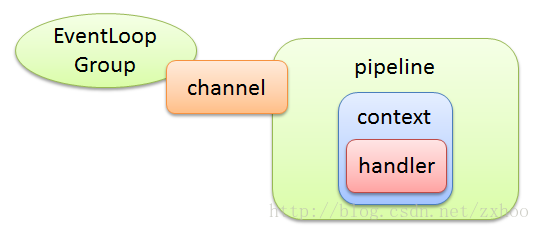
*ChannelPipeline p = channel.pipeline();*

*p.addLast(config.handler());*

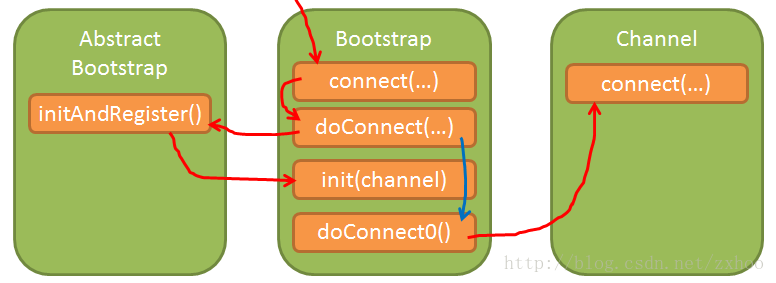
*......*

*}*

到这里Channel准备就绪：



然后Bootstrap.doConnect在initAndRegister之后，接着调用doConnect0，然后调用Channel.connect，这样就Channel就接通服务器进行收发消息，如下：



http://blog.csdn.net/youaremoon/article/details/50371221

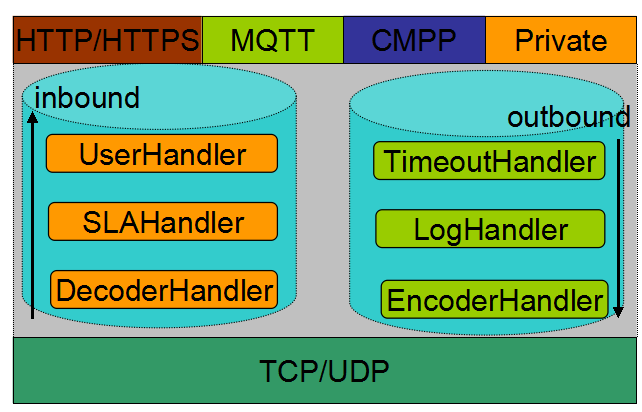
http://blog.csdn.net/zxhoo/article/details/17419229

http://blog.csdn.net/joeyon1985/article/details/46789701

## 4.7 Encoders和Decoders

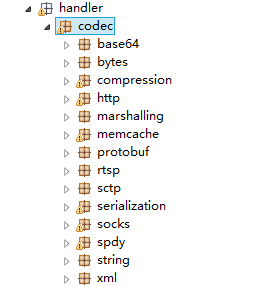
远程跨进程服务调用（如RPC调用），需要使用特定的编解码技术，对需要进行网络传输的对象做编码或者解码，一般完成远程调用。常用的编解码框架包括：Java序列化、Google的Protobuf，Apche Thrift等。

作为一个高性能的异步、NIO通信框架，解编码框架是Netty的重要组成部分，解编码通过ChannelHandler定制扩展，Netty的逻辑架构图如下：



从网络读取的Inbound消息，需要经过编码，将二进制的数据报转换成应用层协议消息或者业务消息，才能够被上层的应用逻辑识别和逻辑处理。同理，用户发送到网络的outbound业务消息，需要经过编码转换成二进制字节数组（对于Netty为ByteBuf）才能够发送到网络断。解码和编码功能是NIO框架的有机组成部分，无论是业务定制扩展实现，还是NIO框架内置的编解码能力，该能够是必不可少的。

在Netty底层实现，既提供了通用的编解码框架供用户扩展，又提供常用的编解码类库供用户使用，Netty内置的解编码的功能列表如下：



类图如下所示：



1. MessageToByteEncoder抽象编码器

负责将POJO对象编码成ByteBuf，用户的编码器MessageToByteEncoder，实现encode，ObjectEncoder实例代码如下：

*@Override*

*protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) throws Exception {*

*int startIdx = out.writerIndex();*

*ByteBufOutputStream bout = new ByteBufOutputStream(out);*

*ObjectOutputStream oout = null;*

*try {*

*bout.write(LENGTH\_PLACEHOLDER);*

*oout = new CompactObjectOutputStream(bout);*

*oout.writeObject(msg);*

*oout.flush();*

*} finally {*

*if (oout != null) {*

*oout.close();*

*} else {*

*bout.close();*

*}*

*}*

*int endIdx = out.writerIndex();*

*out.setInt(startIdx, endIdx - startIdx - 4);*

*}*

实现原理如下：调用write操作时，首先判断当前编码器是否支持需要发送的消息，不支持则直接传输，如果支持则判断缓冲区的类型，对于直接内存分配ioBuffer(堆外内存)，对于堆内存通过heapBuffer方法分配，源码如下：

*@Override*

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

*ByteBuf buf = null;*

*try {*

*if (acceptOutboundMessage(msg)) {*

*@SuppressWarnings("unchecked")*

*I cast = (I) msg;*

*buf = allocateBuffer(ctx, cast, preferDirect);*

*try {*

*encode(ctx, cast, buf);*

*} finally {*

*ReferenceCountUtil.release(cast);*

*}*

*if (buf.isReadable()) {*

*ctx.write(buf, promise);*

*} else {*

*buf.release();*

*ctx.write(Unpooled.EMPTY\_BUFFER, promise);*

*}*

*buf = null;*

*} else {*

*ctx.write(msg, promise);*

*}*

*}*

*}*

编码完成后，调用ReferenceCountUtil的release方法释放编码对象msg，对编码后的ByteBuf进行判断：

1. 如果缓冲区包含可发送的字节，则调用ChannelHandlerContext的write方法发送ByteBuf；
2. 如果缓冲区没有包含可写的字节，则需要释放编码后的ByteBuf，写入一个空的ByteBuf到ChannelHandlerContext中。

发送操作完成之后，在方法退出之前释放编码缓冲区ByteBuf对象。

1. MessageToMessageEncoder，抽象编码器

将一个POJO对象编码成另一个对象，以HTTP+XML协议为例，实现的方式是先将POJO对象编码成XML字符串，再将字符串编码为HTTP请求或者应答消息。对于复杂协议，往往需要经历多次编码，为了便于功能扩展，可以通过多个编码器组合来实现相关功能。

用户的解码器继承MessageToMessageEncoder，实现encode方法即可，与MessageToByteEncoder的区别是输出是对象序列表而不是ByteBuf，ProtobufEncoder示例如下：

*@Override*

*protected void encode(ChannelHandlerContext ctx, MessageLiteOrBuilder msg, List<Object> out)*

*throws Exception {*

*if (msg instanceof MessageLite) {*

*out.add(wrappedBuffer(((MessageLite) msg).toByteArray()));*

*return;*

*}*

*if (msg instanceof MessageLite.Builder) {*

*out.add(wrappedBuffer(((MessageLite.Builder) msg).build().toByteArray()));*

*}*

*}*

MessageToMessageEncoder编码器的实现原理与MessageToByteEncoder相似，唯一的差别是编码后输出的是中间对象，并非最终可传输的ByteBuf。

1. ByteToMessageDecoder，抽象解码器

使用NIO进行网络编程时，往往需要将读取到的字节数组或者字节缓冲区解码为业务所使用的POJO对象。为了方便业务将ByteBuf编码成POJO对象，Netty提供ByteToMessageDecoder抽象工具解码类。

用户自定义解码器继承ByteToMessageDecoder，实现decoder抽象方法完成ByteBuf到POJO对象的解码。

4）MessageToMessageDecoder，抽象解码器

将一个对象二次解码成其他对象，从SocketChannel读取到的TCP数据是ByteBuffer，实际上是字节数组，首先需要将ByteBuffer缓冲区中的数据读取出来，并将其解码成Java对象，然后对Java对象根据某些规则做二次解码，将其解码成另一个POJO对象。因为MessageToMessageDecoder在ByteToMessageDecoder之后，所以称之为二次解码器。

http://www.infoq.com/cn/articles/netty-codec-framework-analyse/

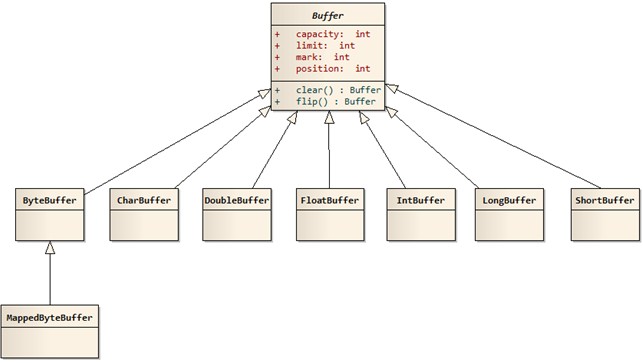
http://blog.csdn.net/suifeng3051/article/details/28861883

## 4.8 缓冲区Buffer

Buffer是一个对象，包含一些要写入或者要读出的数据。在NIO类库中加入Buffer对象，体现了新库与原I/O的一个重要的区别。在面向流的I/O中，将数据直接写入或者将数据直接读到Stream对象中。

在所有的NIO类库中，所有的数据都是用缓冲区进行处理，在读取数据时，它是直接读到缓冲区中；在写入数据时，也是写到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行读写操作。

最常用的缓冲区是ByteBuffer，一个ByteBuffer提供一组功能用于操作byte数组。每一种Java基本类型都对应有一个缓冲区，如下所示：



每一个Buffer类都是Buffer接口的一个子实例，除了Buffer类都有完全一样的操作，只是他们处理的数据类型不一致。因为大多数标准I/O操作都使用ByteBuffer，所以它除了具有一般缓冲区的操作之外还提供一些特有的操作，方便网络读写。

ChannelBuffer https://yq.aliyun.com/articles/88541

Netty源码分析：https://segmentfault.com/a/1190000007282789#articleHeader8

Netty Examples: http://netty.io/4.1/xref/io/netty/example/uptime/package-summary.html

Netty Future和ChannelFuture: http://blog.csdn.net/zhuang\_wk/article/details/52807818

Netty的概念：http://dirlt.com/html/netty.html

Netty逻辑架构：https://www.cnblogs.com/TomSnail/p/6122433.html

Netty总体框架：http://www.cnblogs.com/wxpjimmy/p/3712819.html

Netty:实现原理解析：http://www.importnew.com/15656.html

https://www.cnblogs.com/me115/p/4452801.html

https://yq.aliyun.com/articles/88541

Reactor: http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html

http://www.infoq.com/cn/articles/netty-high-performance

http://ifeve.com/netty-2-3-1/