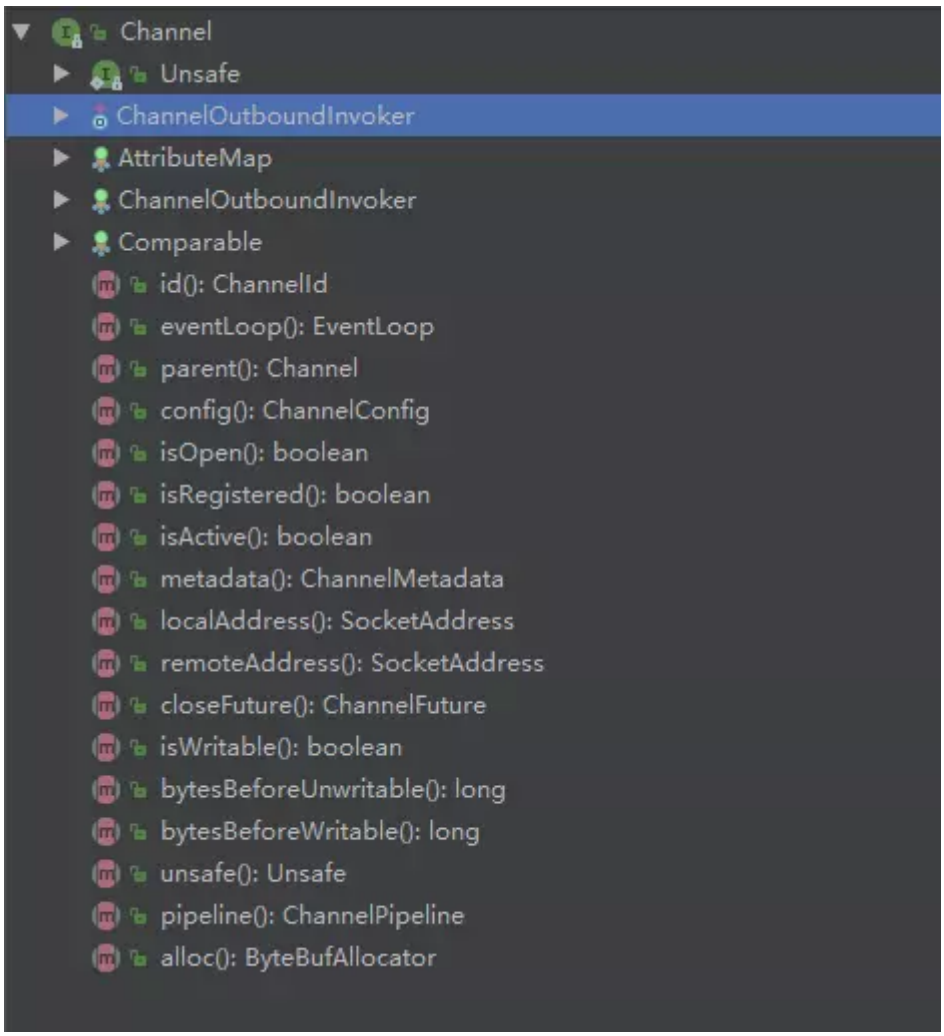
Netty笔记之ChannelHandler

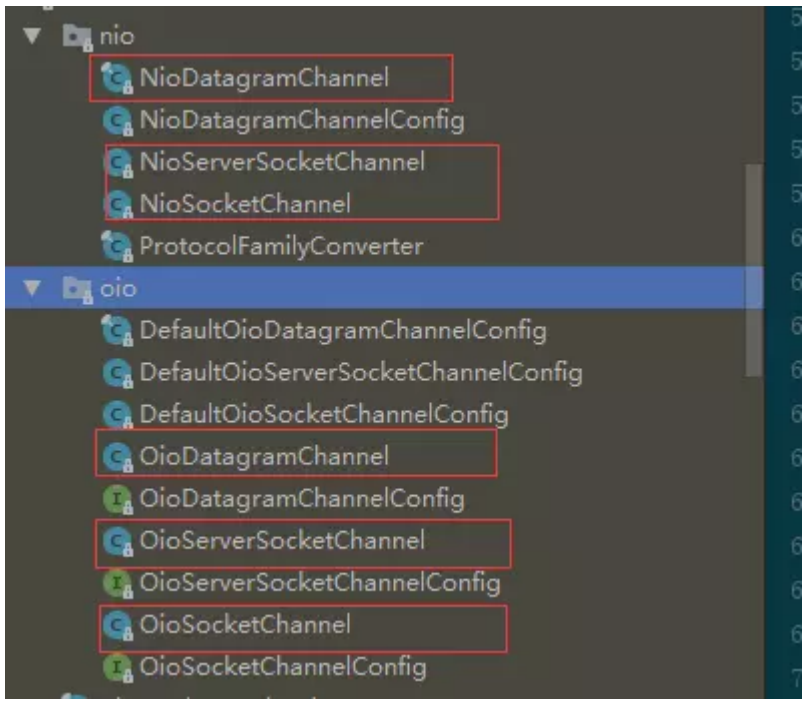
ChannelHandler是netty中的核心处理部分，我们使用netty的绝大部分代码都写在这部分，所以了解它的一些机制和特性是很有必要的

Channel

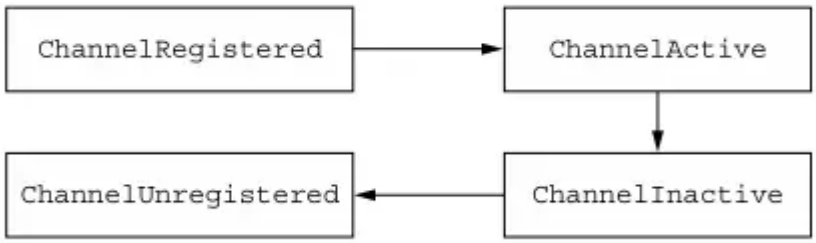
Channel接口抽象了底层socket的一些状态属性以及调用方法



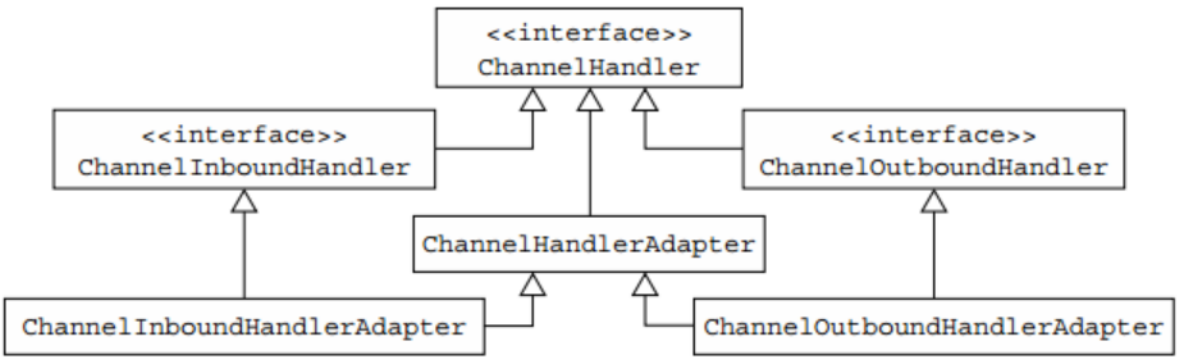
针对不同类型的socket提供不同的子类实现。



Channel生命周期

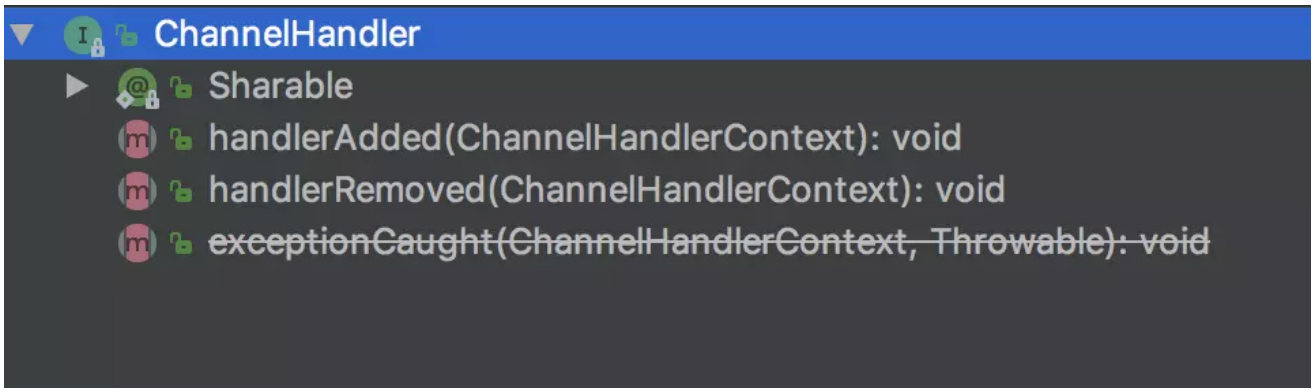


ChannelHandler



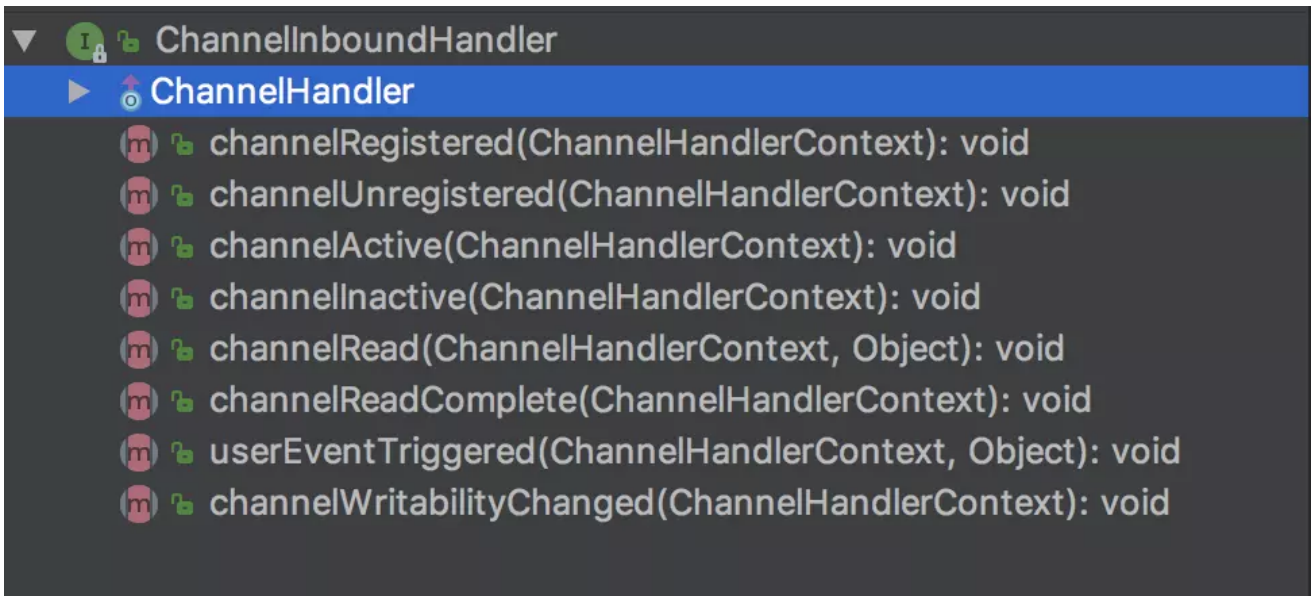
ChannelHandler用于处理Channel对应的事件 ChannelHandler接口里面只定义了三个生命周期方法，我们主要实现它的子接口ChannelInboundHandler和ChannelOutboundHandler，为了便利，框架提供了ChannelInboundHandlerAdapter，ChannelOutboundHandlerAdapter和ChannelDuplexHandler这三个适配类，在使用的时候只需要实现你关注的方法即可

ChannelHandler生命周期方法



ChannelHandler里面定义三个生命周期方法，分别会在当前ChannelHander加入ChannelHandlerContext中，从ChannelHandlerContext中移除，以及ChannelHandler回调方法出现异常时被回调

ChannelInboundHandler

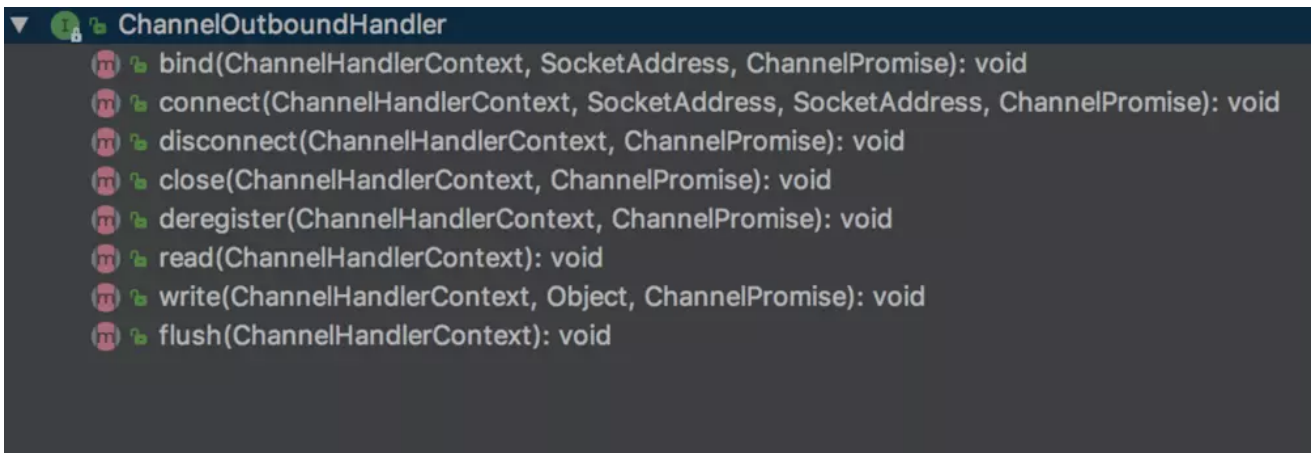


介绍一下这些回调方法被触发的时机

回调方法	触发时机	client	server
channelRegistered	当前channel注册到EventLoop	true	true
channelUnregistered	当前channel从EventLoop取消注册	true	true
channelActive	当前channel激活的时候	true	true
channelInactive	当前channel不活跃的时候，也就是当前channel到了它生命周期末	true	true
channelRead	当前channel从远端读取到数据	true	true
channelReadComplete	channel read消费完读取的数据的时候被触发	true	true
userEventTriggered	用户事件触发的时候		
channelWritabilityChanged	channel的写状态变化的时候触发		

可以注意到每个方法都带了ChannelHandlerContext作为参数，具体作用是，在每个回调事件里面，处理完成之后，使用ChannelHandlerContext的fireChannelXXX方法来传递给下个ChannelHandler，netty的codec模块和业务处理代码分离就用到了这个链路处理

ChannelOutboundHandler



回调方法	触发时机	client	server
bind	bind操作执行前触发	false	true
connect	connect 操作执行前触发	true	false
disconnect	disconnect 操作执行前触发	true	false
close	close操作执行前触发	false	true
deregister	deregister操作执行前触发		
read	read操作执行前触发	true	true
write	write操作执行前触发	true	true
flush	flush操作执行前触发	true	true

注意到一些回调方法有ChannelPromise这个参数，我们可以调用它的addListener注册监听，当回调方法所对应的操作完成后，会触发这个监听 下面这个代码，会在写操作完成后触发，完成操作包括成功和失败

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws
Exception {
    ctx.write(msg,promise);
    System.out.println("out write");
    promise.addListener(new GenericFutureListener<Future<? super Void>>() {
        @Override
        public void operationComplete(Future<? super Void> future) throws Exception {
            if(future.isSuccess()){
                System.out.println("OK");
            }
        }
    });
}
```

ChannelInboundHandler和ChannelOutboundHandler的区别

区别主要在于ChannelInboundHandler的channelRead和channelReadComplete回调和ChannelOutboundHandler的write和flush回调上，ChannelOutboundHandler的channelRead回调负责执行入栈数据的decode逻辑，ChannelOutboundHandler的write负责执行出站数据的encode工作。其他回调方法和具体触发逻辑有关，和in与out无关。

ChannelHandlerContext

每个ChannelHandler通过add方法加入到ChannelPipeline中去的时候，会创建一个对应的ChannelHandlerContext，并且绑定，ChannelPipeline实际维护的是ChannelHandlerContext 的关系 在DefaultChannelPipeline源码中可以看到会保存第一个ChannelHandlerContext以及最后一个ChannelHandlerContext的引用

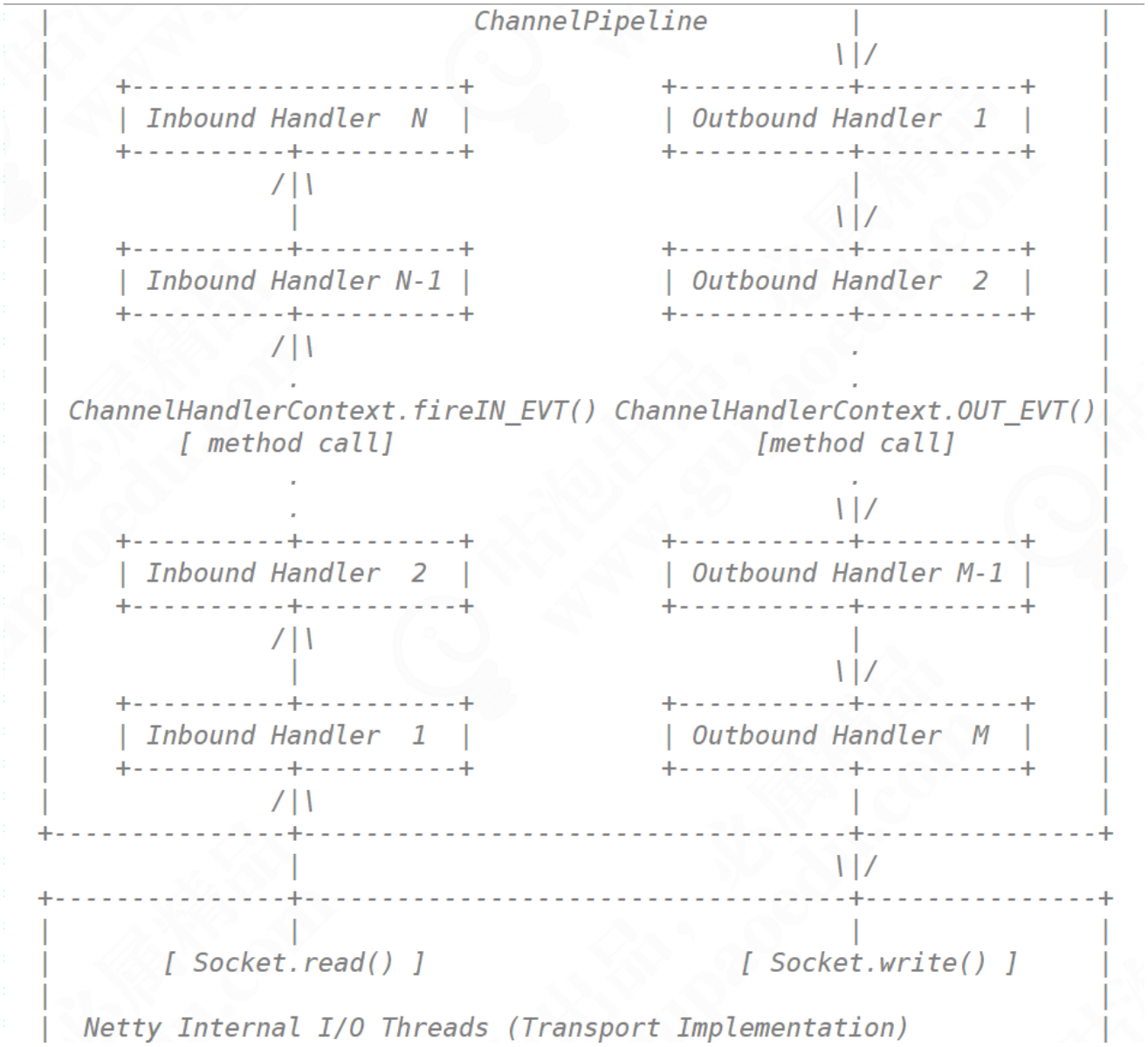
```
final AbstractChannelHandlerContext head;
final AbstractChannelHandlerContext tail;
```

而在AbstractChannelHandlerContext源码中可以看到

```
volatile AbstractChannelHandlerContext next;
volatile AbstractChannelHandlerContext prev;
```

每个ChannelHandlerContext之间形成双向链表

ChannelPipeline



在Channel创建的时候，会同时创建ChannelPipeline

```
protected AbstractChannel(Channel parent) {
    this.parent = parent;
    id = newId();
    unsafe = newUnsafe();
    pipeline = newChannelPipeline();
}
```

在ChannelPipeline中也会持有Channel的引用

```
protected DefaultChannelPipeline newChannelPipeline() {
    return new DefaultChannelPipeline(this);
}
```

ChannelPipeline会维护一个ChannelHandlerContext的双向链表

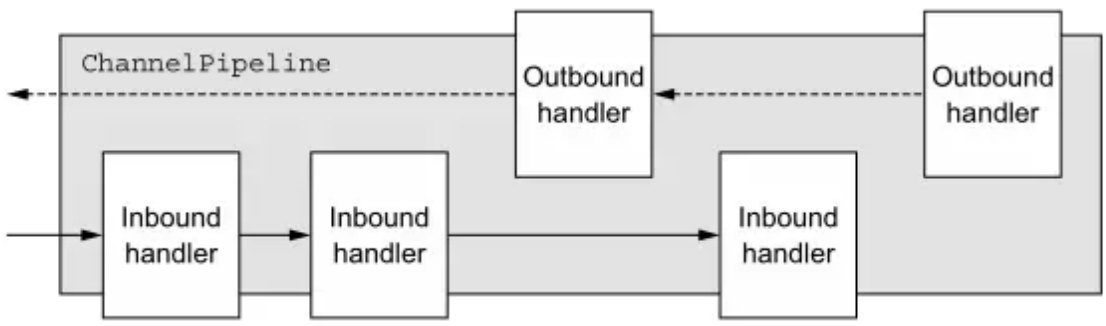
```
final AbstractChannelHandlerContext head;
final AbstractChannelHandlerContext tail;
```

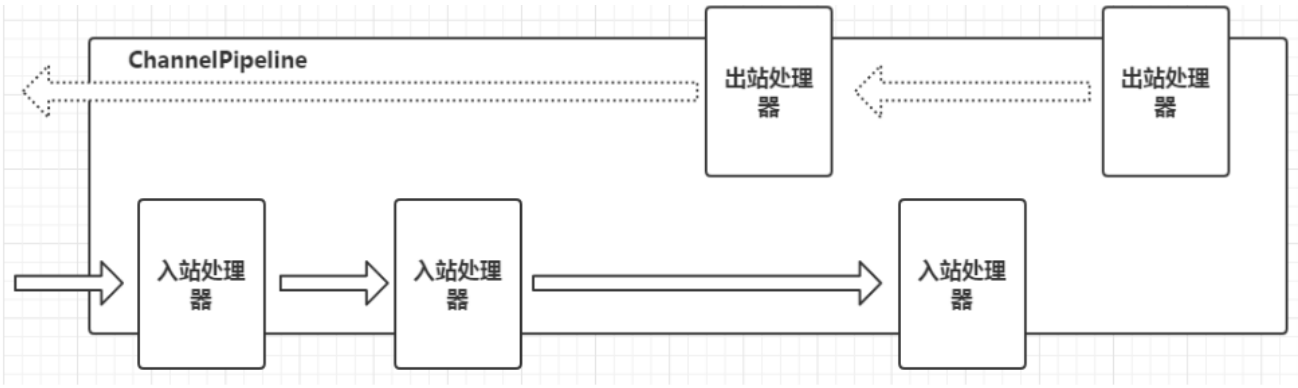
链表的头尾有默认实现

```
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;
}
```

我们添加的自定义ChannelHandler会插入到head和tail之间，如果是ChannelInboundHandler的回调，根据插入的顺序从左向右进行链式调用，ChannelOutboundHandler则相反

具体关系如下，但是下图没有把默认的head和tail画出来，这两个ChannelHandler做的工作相当重要





上面的整条链式的调用是通过Channel接口的方法直接触发的，如果使用ChannelContextHandler的接口方法间接触发，链路会从ChannelContextHandler对应的ChannelHandler开始，而不是从头或尾开始

HeadContext

HeadContext实现了ChannelOutboundHandler，ChannelInboundHandler这两个接口

```
class HeadContext extends AbstractChannelHandlerContext
            implements ChannelOutboundHandler, ChannelInboundHandler
```

因为在头部，所以说HeadContext中关于in和out的回调方法都会触发 关于ChannelInboundHandler，HeadContext的作用是进行一些前置操作，以及把事件传递到下一个ChannelHandlerContext的ChannelInboundHandler中去 看下其中channelRegistered的实现

```
public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
    invokeHandlerAddedIfNeeded();
    ctx.fireChannelRegistered();
}
```

从语义上可以看出来在把这个事件传递给下一个ChannelHandler之前会回调ChannelHandler的handlerAdded方法 而有关ChannelOutboundHandler接口的实现，会在链路的最后执行，看下write方法的实现

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws
Exception {
    unsafe.write(msg, promise);
}
```

这边的unsafe接口封装了底层Channel的调用，之所以取名为unsafe，是不需要用户手动去调用这些方法。这个和阻塞原语的unsafe不是同一个 也就是说，当我们通过Channel接口执行write之后，会执行ChannelOutboundHandler链式调用，在链尾的HeadContext ，在通过unsafe回到对应Channel做相关调用 从netty Channel接口的实现就能论证这个

```
public ChannelFuture write(Object msg) {
    return pipeline.write(msg);
}
```

TailContext

TailContext实现了ChannelInboundHandler接口，会在ChannelInboundHandler调用链最后执行，只要是对调用链完成处理的情况进行处理，看下channelRead实现

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    onUnhandledInboundMessage(msg);
}
```

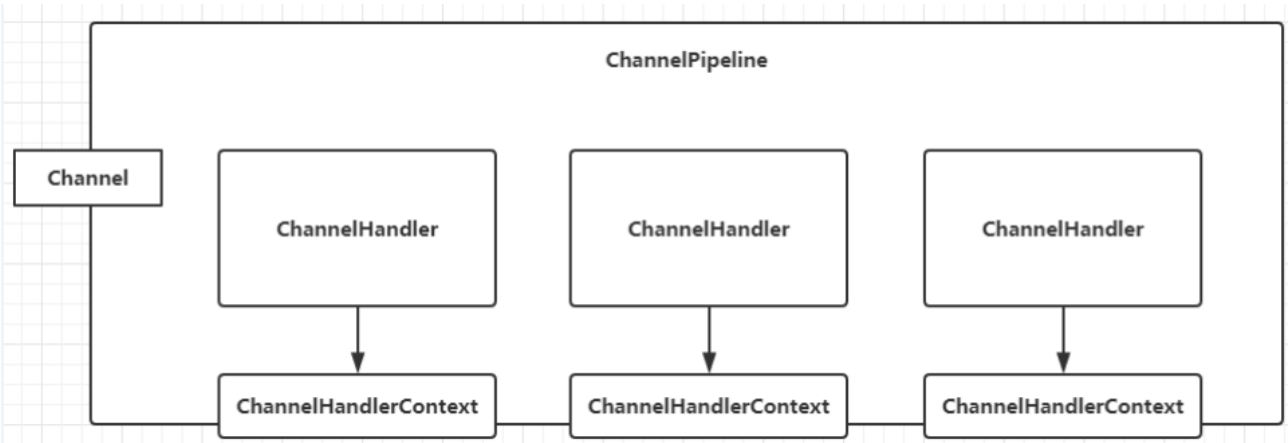
如果我们自定义的最后一个ChannelInboundHandler，也把处理操作交给下一个ChannelHandler，那么就会到TailContext，在TailContext会提供一些默认处理

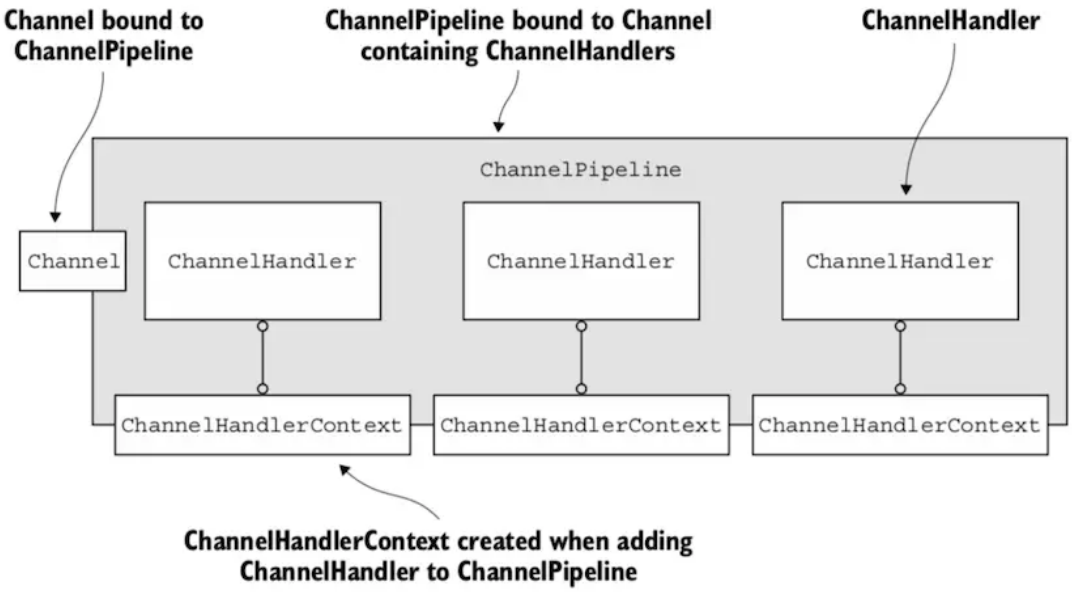
```
protected void onUnhandledInboundMessage(Object msg) {
    try {
        logger.debug(
                "Discarded inbound message {} that reached at the tail of the pipeline.
" +
                        "Please check your pipeline configuration.", msg);
    } finally {
        ReferenceCountUtil.release(msg);
    }
}
```

比如channelRead中的onUnhandledInboundMessage方法，会把msg资源回收，防止内存泄露

强调一点的是，如果要执行整个链路，必须通过调用Channel方法触发，ChannelHandlerContext引用了ChannelPipeline，所以也能间接操作channel的方法，但是会从当前ChannelHandlerContext绑定的ChannelHandler作为起点开始，而不是ChannelHandlerContext的头和尾 这个特性在不需要调用整个链路的情况下可以使用，可以增加一些效率

上述组件之间的关系





1. 每个Channel会绑定一个ChannelPipeline，ChannelPipeline中也会持有Channel的引用
2. ChannelPipeline持有ChannelHandlerContext链路，保留ChannelHandlerContext的头尾节点指针
3. 每个ChannelHandlerContext会对应一个ChannelHandler，也就相当于ChannelPipeline持有ChannelHandler链路
4. ChannelHandlerContext同时也会持有ChannelPipeline引用，也就相当于持有Channel引用
5. ChannelHandler链路会根据Handler的类型，分为InBound和OutBound两条链路