Netty In Action中文版 - 第六章：ChannelHandler

目录 [-]

本章介绍

- ChannelPipeline
- ChannelHandlerContext
- ChannelHandler
- Inbound vs outbound(入站和出站)

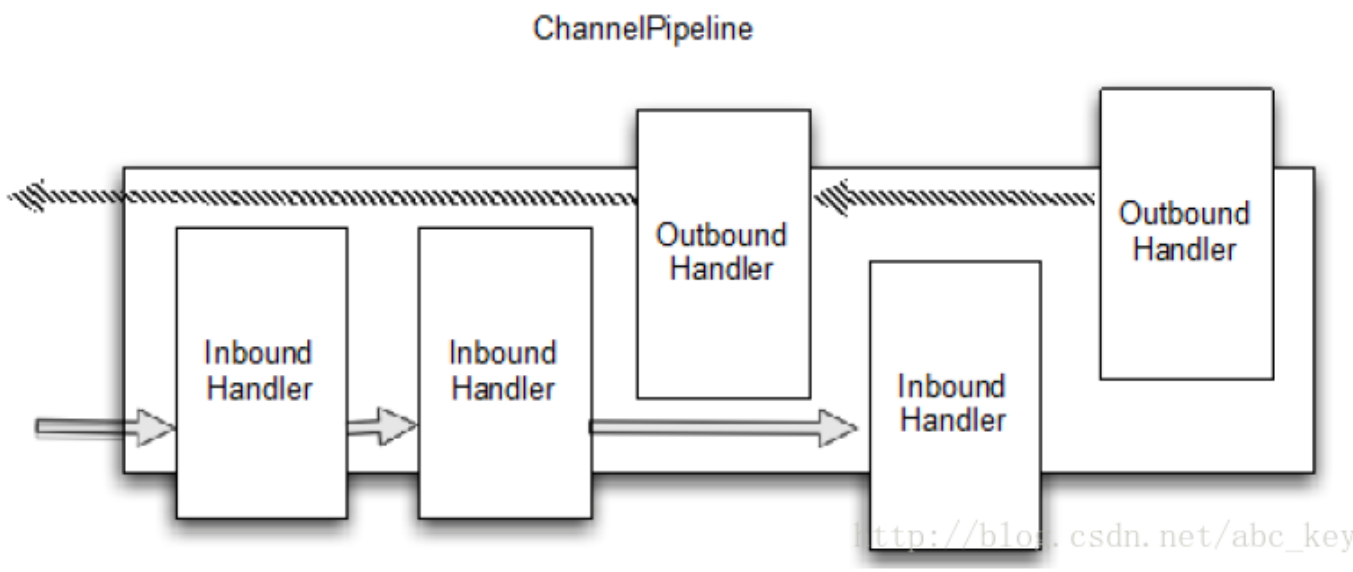
接受连接或创建他们只是你的应用程序的一部分，虽然这些任何很重要，但是一个网络应用程序旺旺是更复杂的，需要更多的代码编写，如处理传入和传出的数据。Netty提供了一个强大的处理这些事情的功能，允许用户自定义ChannelHandler的实现来处理数据。使得ChannelHandler更强大的是可以连接每个ChannelHandler来实现任务，这有助于代码的整洁和重用。但是处理数据只是ChannelHandler所做的事情之一，也可以压制I/O操作，例如写请求。所有这些都可以动态实现。

6.1 ChannelPipeline

ChannelPipeline是ChannelHandler实例的列表，用于处理或截获通道的接收和发送数据。ChannelPipeline提供了一种高级的截取过滤器模式，让用户可以在ChannelPipeline中完全控制一个事件及如何处理ChannelHandler与ChannelPipeline的交互。

对于每个新的通道，会创建一个新的ChannelPipeline并附加至通道。一旦连接，Channel和ChannelPipeline之间的耦合是永久性的。Channel不能附加其他的ChannelPipeline或从ChannelPipeline分离。

下图描述了ChannelHandler在ChannelPipeline中的I/O处理，一个I/O操作可以由一个ChannelInboundHandler或ChannelOutboundHandler进行处理，并通过调用ChannelInboundHandler处理入站IO或通过ChannelOutboundHandler处理出站IO。



如上图所示，ChannelPipeline是ChannelHandler的一个列表；如果一个入站I/O事件被触发，这个事件会从第一个开始依次通过ChannelPipeline中的ChannelHandler；若是一个入站I/O事件，则会从最后一个开始依次通过ChannelPipeline中的ChannelHandler。ChannelHandler可以处理事件并检查类型，如果某个ChannelHandler不能处理则会跳过，并将事件传递到下一个ChannelHandler。ChannelPipeline可以动态添加、删除、替换其中的ChannelHandler，这样的机制可以提高灵活性。

修改ChannelPipeline的方法：

- addFirst(...)，添加ChannelHandler在ChannelPipeline的第一个位置
- addBefore(...)，在ChannelPipeline中指定的ChannelHandler名称之前添加ChannelHandler
- addAfter(...)，在ChannelPipeline中指定的ChannelHandler名称之后添加ChannelHandler
- addLast(ChannelHandler...)，在ChannelPipeline的末尾添加ChannelHandler
- remove(...)，删除ChannelPipeline中指定的ChannelHandler
- replace(...)，替换ChannelPipeline中指定的ChannelHandler

```java
ChannelPipeline pipeline = ch.pipeline();
FirstHandler firstHandler = new FirstHandler();
pipeline.addLast("handler1", firstHandler);
pipeline.addFirst("handler2", new SecondHandler());
pipeline.addLast("handler3", new ThirdHandler());
pipeline.remove("handler3");
pipeline.remove(firstHandler);
pipeline.replace("handler2", "handler4", new FourthHandler());
```

被添加到ChannelPipeline的ChannelHandler将通过IO-Thread处理事件，这意味了必须不能有其他的IO-Thread阻塞来影响IO的整体处理；有时候可能需要阻塞，例如JDBC。因此，Netty允许通过一个EventExecutorGroup到每一个ChannelPipeline.add*方法，自定义的事件会被包含在EventExecutorGroup中的EventExecutor来处理，默认的实现是DefaultEventExecutorGroup。

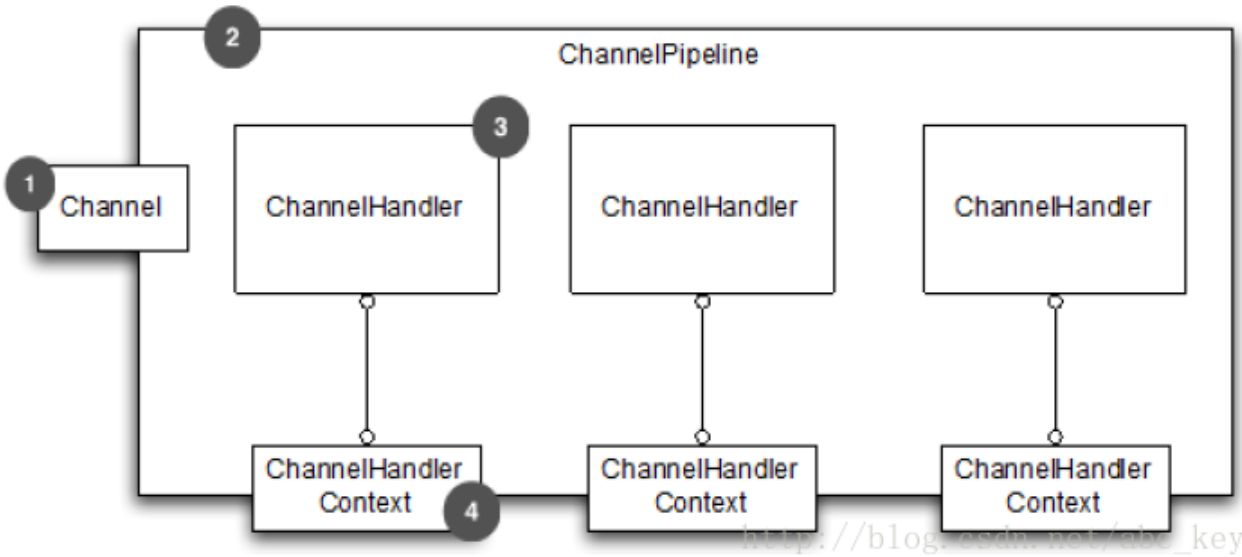
ChannelPipeline除了一些修改的方法，还有很多其他的方法，具体是方法及使用可以看API文档或源码。

6.2 ChannelHandlerContext

每个ChannelHandler被添加到ChannelPipeline后，都会创建一个ChannelHandlerContext并与之创建的ChannelHandler关联绑定。ChannelHandlerContext允许ChannelHandler与其他的ChannelHandler实现进行交互，这是相同ChannelPipeline的一部分。ChannelHandlerContext不会改变添加到其中的ChannelHandler，因此它是安全的。

6.2.1 通知下一个ChannelHandler

在相同的ChannelPipeline中通过调用ChannelInboundHandler和ChannelOutboundHandler中各个方法中的一个方法来通知最近的handler，通知开始的地方取决你如何设置。下图显示了ChannelHandlerContext、ChannelHandler、ChannelPipeline的关系：



如果你想有一些事件流全部通过ChannelPipeline，有两个不同的方法可以做到：

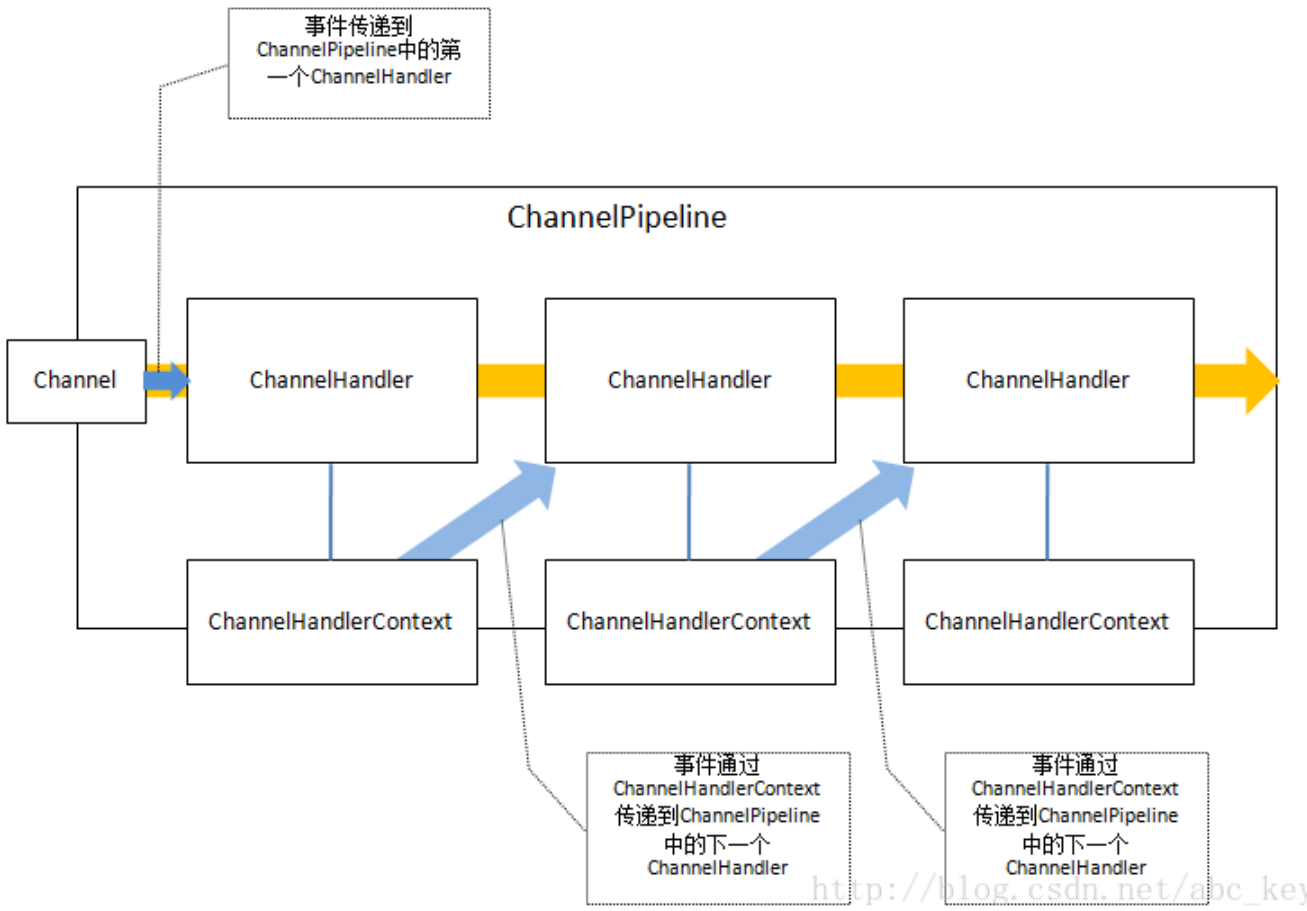
- 调用Channel的方法
- 调用ChannelPipeline的方法

这两个方法都可以让事件流全部通过ChannelPipeline。无论从头部还是尾部开始，因为它主要依赖于事件的性质。如果是一个“入站”事件，它开始于头部；若是一个“出站”事件，则开始于尾部。

下面的代码显示了一个写事件如何通过ChannelPipeline从尾部开始：

```java
@Override
protected void initChannel(SocketChannel ch) throws Exception {
    ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            //Event via Channel
            Channel channel = ctx.channel();
            channel.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
            //Event via ChannelPipeline
            ChannelPipeline pipeline = ctx.pipeline();
            pipeline.write(Unpooled.copiedBuffer("netty in action", CharsetUtil.UTF_8));
        }
    });
}
```

下图表示通过Channel或ChannelPipeline的通知：



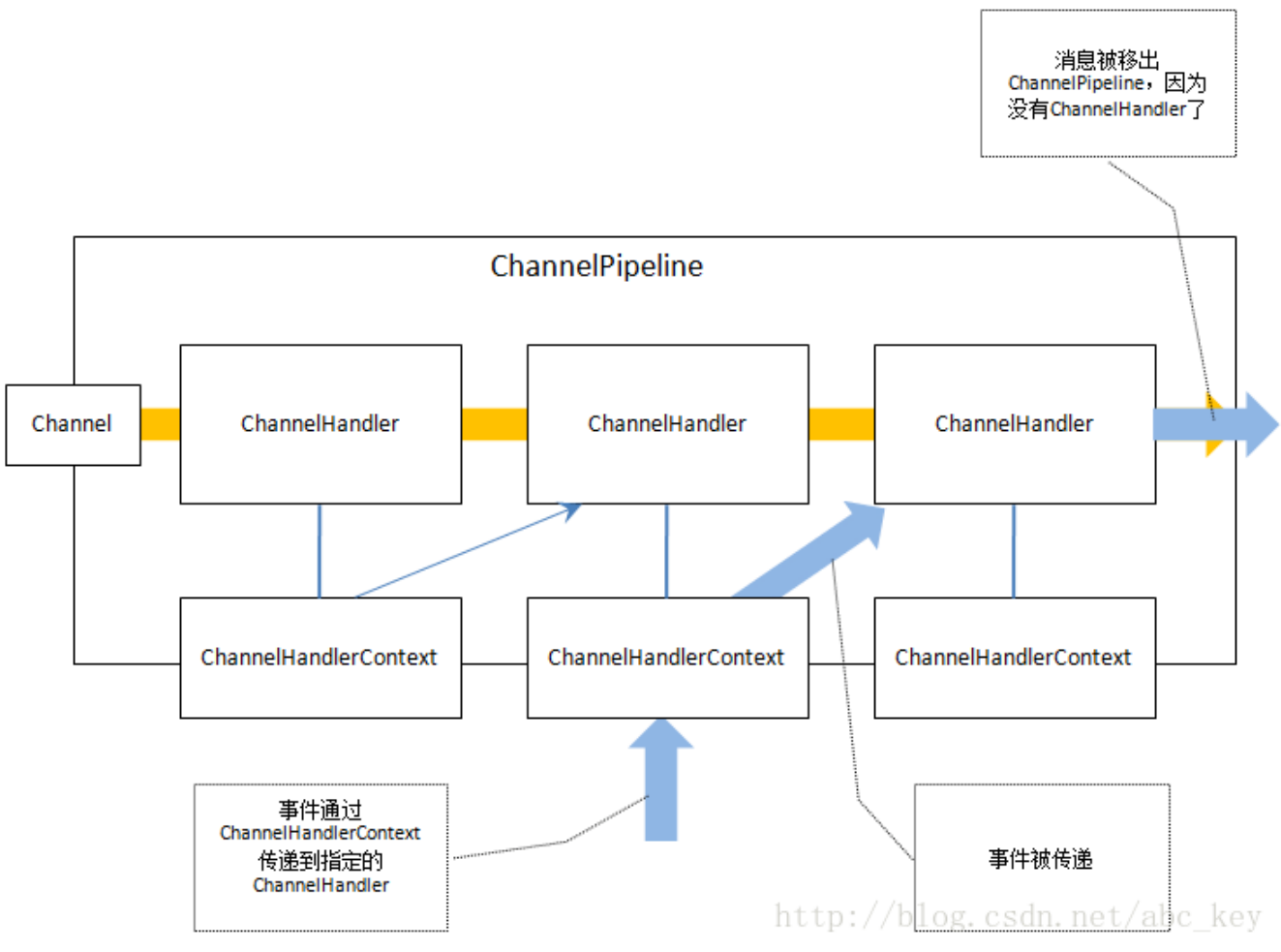
可能你想从ChannelPipeline的指定位置开始，不想流经整个ChannelPipeline，如下情况：

- 为了节省开销，不感兴趣的ChannelHandler不让通过
- 排除一些ChannelHandler

在这种情况下，你可以使用ChannelHandlerContext的ChannelHandler通知起点。它使用ChannelHandlerContext执行下一个ChannelHandler。下面代码显示了直接使用ChannelHandlerContext操作：

```java
// Get reference of ChannelHandlerContext
ChannelHandlerContext ctx = ..;
// Write buffer via ChannelHandlerContext
ctx.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));
```

该消息流经ChannelPipeline到下一个ChannelHandler，在这种情况下使用ChannelHandlerContext开始下一个ChannelHandler。下图显示了事件流：



如上图显示的，从指定的ChannelHandlerContext开始，跳过前面所有的ChannelHandler，使用ChannelHandlerContext操作是常见的模式，最常用的是从ChannelHanlder调用操作，也可以在外部使用ChannelHandlerContext，因为这是线程安全的。

6.2.2 修改ChannelPipeline

调用ChannelHandlerContext的pipeline()方法能访问ChannelPipeline，能在运行时动态的增加、删除、替换ChannelPipeline中的ChannelHandler。可以保持ChannelHandlerContext供以后使用，如外部Handler方法触发一个事件，甚至从一个不同的线程。

下面代码显示了保存ChannelHandlerContext供之后使用或其他线程使用：

```java
public class WriteHandler extends ChannelHandlerAdapter {
    private ChannelHandlerContext ctx;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) throws Exception {
        this.ctx = ctx;
    }

    public void send(String msg){
        ctx.write(msg);
    }
}
```

请注意，ChannelHandler实例如果带有@Sharable注解则可以被添加到多个ChannelPipeline。也就是说单个ChannelHandler实例可以有多个ChannelHandlerContext，因此可以调用不同ChannelHandlerContext获取同一个ChannelHandler。如果添加不带@Sharable注解的ChannelHandler实例到多个ChannelPipeline则会抛出异常；使用@Sharable注解后的ChannelHandler必须在不同的线程和不同的通道上安全使用。怎么是不安全的使用？看下面代码：

```java
@Sharable
public class NotSharableHandler extends ChannelInboundHandlerAdapter {

    private int count;

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        count++;
        System.out.println("channelRead(...) called the " + count + " time");
        ctx.fireChannelRead(msg);
    }

}
```

上面是一个带@Sharable注解的Handler，它被多个线程使用时，里面count是不安全的，会导致count值错误。

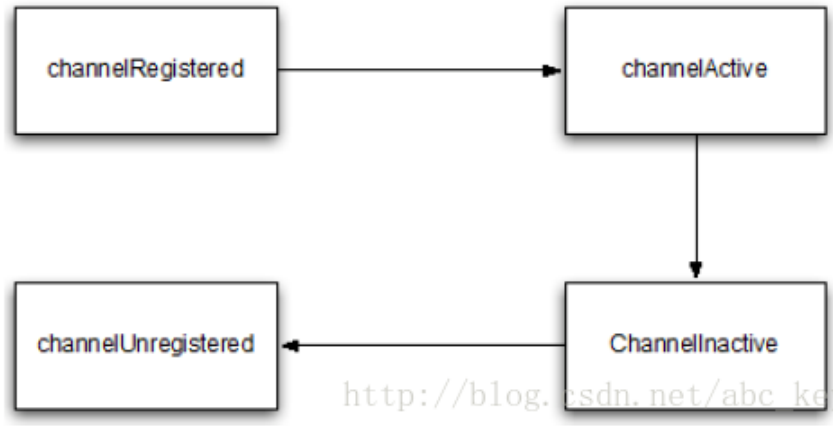
为什么要共享ChannelHandler？使用@Sharable注解共享一个ChannelHandler在一些需求中还是有很好的作用的，如使用一个ChannelHandler来统计连接数或来处理一些全局数据等等。

6.3 状态模型

Netty有一个简单但强大的状态模型，并完美映射到ChannelInboundHandler的各个方法。下面是Channel生命周期四个不同的状态：

- channelUnregistered
- channelRegistered
- channelActive
- channelInactive

Channel的状态在其生命周期中变化，因为状态变化需要触发，下图显示了Channel状态变化：

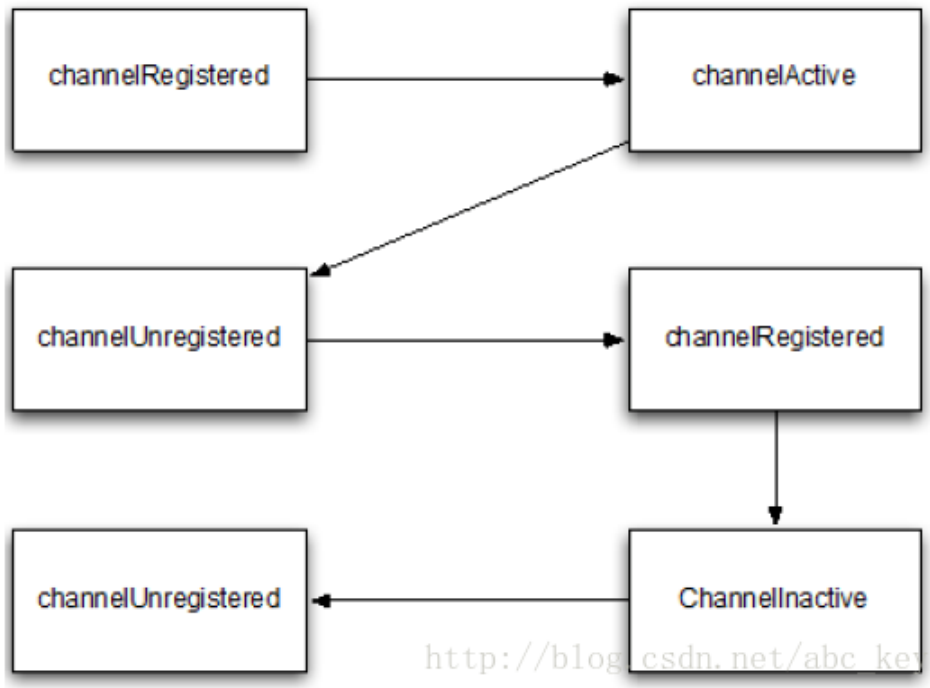


http://blog.csdn.net/abc_key

还可以看到额外的状态变化，因为用户允许从EventLoop中注销Channel暂停事件执行，然后再重新注册。在这种情况下，你会看到

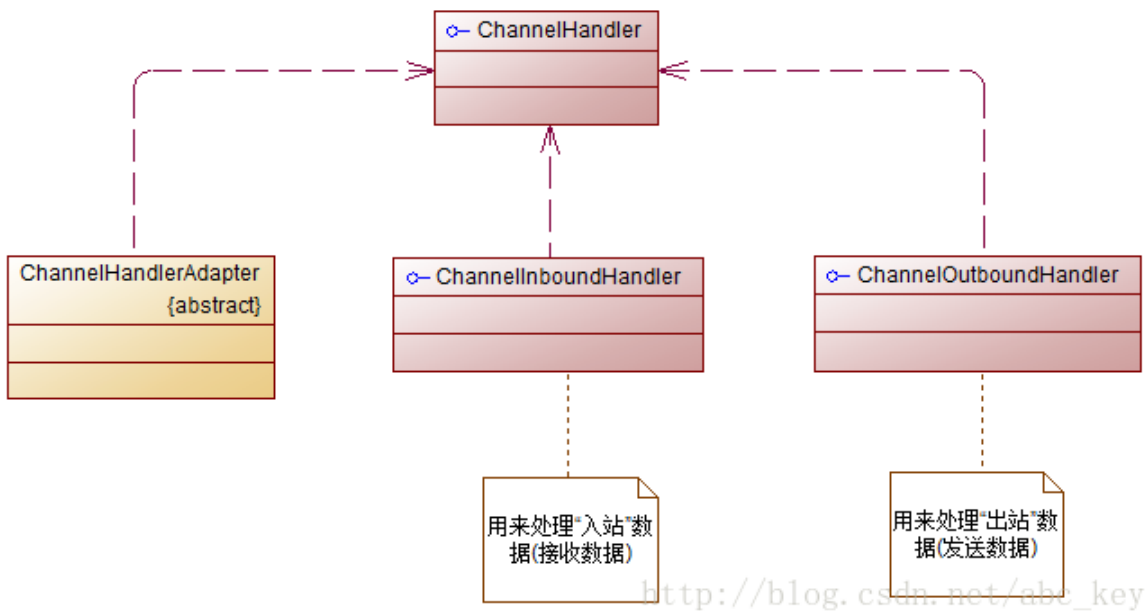
多个channelRegistered和channelUnregistered状态的变化，而永远只有一个channelActive和channelInactive的状态，因为一个通道在其生命周期内只能连接一次，之后就会被回收；重新连接，则是创建一个新的通道。

下图显示了从EventLoop中注销Channel后再重新注册的状态变化：



6.4 ChannelHandler和其子类

Netty中有3个实现了ChannelHandler接口的类，其中2个是接口，一个是抽象类。如下图：



6.4.1 ChannelHandler中的方法

Netty定义了良好的类型层次结构来表示不同的处理程序类型，所有的类型的父类是ChannelHandler。ChannelHandler提供了在其生命周期内添加或从ChannelPipeline中删除的方法。

- handlerAdded，ChannelHandler添加到实际上下文中准备处理事件
- handlerRemoved，将ChannelHandler从实际上下文中删除，不再处理事件
- exceptionCaught，处理抛出的异常

上面三个方法都需要传递ChannelHandlerContext参数，每个ChannelHandler被添加到ChannelPipeline时会自动创建ChannelHandlerContext。ChannelHandlerContext允许在本地通道安全的存储和检索值。Netty还提供了一个实现了ChannelHandler的抽象类：ChannelHandlerAdapter。ChannelHandlerAdapter实现了父类的所有方法，基本上就是传递事件到ChannelPipeline中的下一个ChannelHandler直到结束。

6.4.2 ChannelInboundHandler

ChannelInboundHandler提供了一些方法再接收数据或Channel状态改变时被调用。下面是ChannelInboundHandler的一些方法：

- channelRegistered，ChannelHandlerContext的Channel被注册到EventLoop；
- channelUnregistered，ChannelHandlerContext的Channel从EventLoop中注销
- channelActive，ChannelHandlerContext的Channel已激活
- channelInactive，ChannelHanderContxt的Channel结束生命周期
- channelRead，从当前Channel的对端读取消息
- channelReadComplete，消息读取完成后执行
- userEventTriggered，一个用户事件被处罚

- channelWritabilityChanged，改变通道的可写状态，可以使用Channel.isWritable()检查
- exceptionCaught，重写父类ChannelHandler的方法，处理异常

Netty提供了一个实现了ChannelInboundHandler接口并继承ChannelHandlerAdapter的类：ChannelInboundHandlerAdapter。ChannelInboundHandlerAdapter实现了ChannelInboundHandler的所有方法，作用就是处理消息并将消息转发到ChannelPipeline中的下一个ChannelHandler。ChannelInboundHandlerAdapter的channelRead方法处理完消息后不会自动释放消息，若想自动释放收到的消息，可以使用SimpleChannelInboundHandler<I>。

看下面代码：

```java
/**
 * 实现ChannelInboundHandlerAdapter的Handler，不会自动释放接收的消息对象
 * @author c.k
 *
 */
public class DiscardHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        //手动释放消息
        ReferenceCountUtil.release(msg);
    }
}
```

```java
/**
 * 继承SimpleChannelInboundHandler，会自动释放消息对象
 * @author c.k
 *
 */
public class SimpleDiscardHandler extends SimpleChannelInboundHandler<Object> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        //不需要手动释放
    }
}
```

如果需要其他状态改变的通知，可以重写Handler的其他方法。通常自定义消息类型来解码字节，可以实现ChannelInboundHandler或ChannelInboundHandlerAdapter。有一个更好的解决方法，使用编解码器的框架可以很容的实现。使用ChannelInboundHandler、ChannelInboundHandlerAdapter、SimpleChannelInboundhandler这三个中的一个来处理接收消息，使用哪一个取决于需求；大多数时候使用SimpleChannelInboundHandler处理消息，使用ChannelInboundHandlerAdapter处理其他的“入站”事件或状态改变。

ChannelInitializer用来初始化ChannelHandler，将自定义的各种ChannelHandler添加到ChannelPipeline中。

6.4.3 ChannelOutboundHandler

ChannelOutboundHandler用来处理“出站”的数据消息。ChannelOutboundHandler提供了下面一些方法：

- bind，Channel绑定本地地址
- connect，Channel连接操作
- disconnect，Channel断开连接
- close，关闭Channel
- deregister，注销Channel
- read，读取消息，实际是截获ChannelHandlerContext.read()
- write，写操作，实际是通过ChannelPipeline写消息，Channel.flush()属性到实际通道
- flush，刷新消息到通道

ChannelOutboundHandler是ChannelHandler的子类，实现了ChannelHandler的所有方法。所有最重要的方法采取ChannelPromise，因此一旦请求停止从ChannelPipeline转发参数则必须得到通知。Netty提供了ChannelOutboundHandler的实现：ChannelOutboundHandlerAdapter。ChannelOutboundHandlerAdapter实现了父类的所有方法，并且可以根据需要重写感兴趣的方法。所有这些方法的实现，在默认情况下，都是通过调用ChannelHandlerContext的方法将事件转发到ChannelPipeline中下一个ChannelHandler。

看下面的代码：

```java
public class DiscardOutboundHandler extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
```

```
04.             ReferenceCountUtil.release(msg);
05.             promise.setSuccess();
06.         }
07.     }
```

重要的是要记得释放致远并直通ChannelPromise，若ChannelPromise没有被通知可能会导致其中一个ChannelFutureListener不被通知去处理一个消息。

如果消息被消费并且没有被传递到ChannelPipeline中的下一个ChannelOutboundHandler，那么就需要调用ReferenceCountUtil.release(message)来释放消息资源。一旦消息被传递到实际的通道，它会自动写入消息或在通道关闭是释放。