咕泡学院 JavaVIP 高级课程教案

深入分析 Netty 源码

关于本文档

主题	咕泡学院 Java VIP 高级课程教案—深入分析 Netty 源码
主讲	Tom 老师
适用对象	咕泡学院 Java 高级 VIP 学员及 VIP 授课老师(源码版本 4.1.6.Final)

1. 揭开 BootStrap 神秘面纱

1.1. 客户端 BootStrap

Bootstrap 是 Netty 提供的一个便利的工厂类，我们可以通过它来完成 Netty 的客户端或服务器端的 Netty 初始化.
下面我先来看一个例子，从客户端和服务器端分别分析一下 Netty 的程序是如何启动的。
首先，让我们从客户端方面的代码开始

```
public class ChatClient {
    public ChatClient connect(int port,String host,final String nickName){
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.SO_KEEPALIVE, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    pipeline.addLast(new StringDecoder());
                    pipeline.addLast(new StringEncoder());
                    pipeline.addLast(new ChatClientHandler(nickName));
                }
            });
            //发起同步连接操作
            ChannelFuture channelFuture = bootstrap.connect(host, port).sync();
            channelFuture.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally{
            //关闭，释放线程资源
            group.shutdownGracefully();
        }
        return this;
    }

    public static void main(String[] args) {
        new ChatClient().connect(8080, "localhost","Tom 老师");
    }
}
```

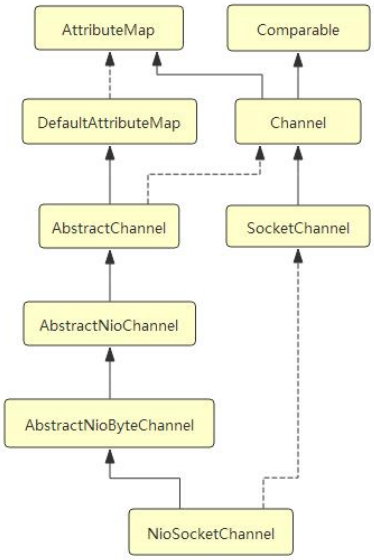
从上面的客户端代码虽然简单，但是却展示了 Netty 客户端初始化时所需的所有内容：

1、EventLoopGroup：不论是服务器端还是客户端，都必须指定 EventLoopGroup. 在这个例子中，指定了 NioEventLoopGroup，表示一个 NIO 的 EventLoopGroup.
2、ChannelType：指定 Channel 的类型. 因为是客户端，因此使用了 NioSocketChannel.
3、Handler：设置数据的处理器.

下面我们深入代码，看一下客户端通过 Bootstrap 启动后，都做了哪些工作.

1.1.1 NioSocketChannel 的初始化过程

在 Netty 中，Channel 是一个 Socket 的抽象，它为用户提供了关于 Socket 状态(是否是连接还是断开) 以及对 Socket 的读写等操作. 每当 Netty 建立了一个连接后，都会有一个对应的 Channel 实例.
NioSocketChannel 的类层次结构如下：



接下来我们着重分析一下 Channel 的初始化过程。

1.1.2 ChannelFactory 和 Channel 类型的确定

除了 TCP 协议以外，Netty 还支持很多其他的连接协议，并且每种协议还有 NIO(非阻塞 IO) 和 OIO(Old-IO，即传统的阻塞 IO) 版本的区别. 不同协议不同的阻塞类型的连接都有不同的 Channel 类型与之对应下面是一些常用的 Channel 类型：

NioSocketChannel，代表异步的客户端 TCP Socket 连接.
NioServerSocketChannel，异步的服务器端 TCP Socket 连接.
NioDatagramChannel，异步的 UDP 连接
NioSctpChannel，异步的客户端 Sctp 连接.
NioSctpServerChannel，异步的 Sctp 服务器端连接.
OioSocketChannel，同步的客户端 TCP Socket 连接.
OioServerSocketChannel，同步的服务器端 TCP Socket 连接.
OioDatagramChannel，同步的 UDP 连接
OioSctpChannel，同步的 Sctp 服务器端连接.
OioSctpServerChannel，同步的客户端 TCP Socket 连接.

那么我们是如何设置所需要的 Channel 的类型的呢？答案是 channel() 方法的调用.
回想一下我们在客户端连接代码的初始化 Bootstrap 中，会调用 channel() 方法，传入

NioSocketChannel.class，这个方法其实就是初始化了一个 ReflectiveChannelFactory:

```
public B channel(Class<? extends C> channelClass) {
        if (channelClass == null) {
            throw new NullPointerException("channelClass");
        }
        return channelFactory(new ReflectiveChannelFactory<C>(channelClass));
    }
```

而 ReflectiveChannelFactory 实现了 ChannelFactory 接口，它提供了唯一的方法，即 newChannel. ChannelFactory，顾名思义，就是产生 Channel 的工厂类.

进入到 ReflectiveChannelFactory.newChannel 中，我们看到其实现代码如下:

```
  @Override
    public T newChannel() {
         // 删除了 try...catch 块
         return clazz.newInstance();
    }
```

根据上面代码的提示，我们就可以确定:

1、Bootstrap 中的 ChannelFactory 的实现是 ReflectiveChannelFactory

2、生成的 Channel 的具体类型是 NioSocketChannel.

Channel 的实例化过程，其实就是调用的 ChannelFactory.newChannel 方法，而实例化的 Channel 的具体的类型又是和在初始化 Bootstrap 时传入的 channel() 方法的参数相关. 因此对于我们这个例子中的客户端的 Bootstrap 而言，生成的的 Channel 实例就是 NioSocketChannel.

1.1.3 Channel 的实例化

前面我们已经知道了如何确定一个 Channel 的类型，并且了解到 Channel 是通过工厂方法 ChannelFactory.newChannel() 来实例化的，那么 ChannelFactory.newChannel() 方法在哪里调用呢?继续跟踪，我们发现其调用链是:

Bootstrap.connect->Bootstrap.doResolveAndConnect->AbstractBootstrap.initAndRegister

在 AbstractBootstrap.initAndRegister 中调用了 channelFactory().newChannel()来获取一个新的 NioSocketChannel 实例，其源码如下:

```
final ChannelFuture initAndRegister() {
        // 去掉非关键代码
        Channel channel = channelFactory.newChannel();
        init(channel);

        ChannelFuture regFuture = config().group().register(channel);
       // 去掉非关键代码
        return regFuture;
    }
```

在 newChannel 中，通过类对象的 newInstance 来获取一个新 Channel 实例，因而会调用 NioSocketChannel 的默认构造器.NioSocketChannel 默认构造器代码如下:

```
    public NioSocketChannel() {
```

```
        this(DEFAULT_SELECTOR_PROVIDER);
    }
```

这里的代码比较关键，我们看到，在这个构造器中，会调用 newSocket 来打开一个新的 Java NIO SocketChannel：

```
    private static SocketChannel newSocket(SelectorProvider provider) {
            // 删除了try...catch块
            return provider.openSocketChannel();
    }
```

接着会调用父类，即 AbstractNioByteChannel 的构造器：

```
AbstractNioByteChannel(Channel parent, SelectableChannel ch)
```

并传入参数 parent 为 null，ch 为刚才使用 newSocket 创建的 Java NIO SocketChannel，因此生成的 NioSocketChannel 的 parent channel 是空的.

```
 protected AbstractNioByteChannel(Channel parent, SelectableChannel ch) {
        super(parent, ch, SelectionKey.OP_READ);
 }
```

接着会继续调用父类 AbstractNioChannel 的构造器，并传入了参数 readInterestOp = SelectionKey.OP_READ:

```
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp)
{
        super(parent);
        this.ch = ch;
        this.readInterestOp = readInterestOp;
        // 省略...catch块
        // 设置Java NIO SocketChannel为非阻塞的
        ch.configureBlocking(false);
}
```

然后继续调用父类 AbstractChannel 的构造器：

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

到这里，一个完整的 NioSocketChannel 就初始化完成了，我们可以稍微总结一下构造一个 NioSocketChannel 所需要做的工作：

1、调用 NioSocketChannel.newSocket(DEFAULT_SELECTOR_PROVIDER) 打开一个新的 Java NIO SocketChannel

2、AbstractChannel(Channel parent) 中初始化 AbstractChannel 的属性：

parent 属性置为 null

unsafe 通过 newUnsafe() 实例化一个 unsafe 对象，它的类型是 AbstractNioByteChannel.NioByteUnsafe 内部类

pipeline 是 new DefaultChannelPipeline(this) 新创建的实例. 这里体现了:Each channel has its own pipeline and it is created automatically when a new channel is created.

3、AbstractNioChannel 中的属性：

SelectableChannel ch 被设置为 Java SocketChannel，即 NioSocketChannel.newSocket 返回的 Java NIO SocketChannel.

readInterestOp 被设置为 SelectionKey.OP_READ

SelectableChannel ch 被配置为非阻塞的 ch.configureBlocking(false)

4、NioSocketChannel 中的属性：

SocketChannelConfig config = new NioSocketChannelConfig(this, socket.socket())

1.1.4 关于 unsafe 字段的初始化

我们简单地提到了，在实例化 NioSocketChannel 的过程中，会在父类 AbstractChannel 的构造器中，调用 newUnsafe() 来获取一个 unsafe 实例. 那么 unsafe 是怎么初始化的呢？它的作用是什么？

其实 unsafe 特别关键，它封装了对 Java 底层 Socket 的操作，因此实际上是沟通 Netty 上层和 Java 底层的重要的桥梁.

那么我们就来看一下 Unsafe 接口所提供的方法吧：

```
interface Unsafe {
        RecvByteBufAllocator.Handle recvBufAllocHandle();
        SocketAddress localAddress();
        SocketAddress remoteAddress();
        void register(EventLoop eventLoop, ChannelPromise promise);
        void bind(SocketAddress localAddress, ChannelPromise promise);
        void connect(SocketAddress remoteAddress, SocketAddress localAddress,
ChannelPromise promise);
        void disconnect(ChannelPromise promise);
        void close(ChannelPromise promise);
        void closeForcibly();
        void deregister(ChannelPromise promise);
        void beginRead();
        void write(Object msg, ChannelPromise promise);
        void flush();

        ChannelPromise voidPromise();
        ChannelOutboundBuffer outboundBuffer();
    }
}
```

一看便知，这些方法其实都会对应到相关的 Java 底层的 Socket 的操作.

回到 AbstractChannel 的构造方法中，在这里调用了 newUnsafe() 获取一个新的 unsafe 对象，而 newUnsafe 方法在 NioSocketChannel 中被重写了：

```
  protected AbstractNioUnsafe newUnsafe() {
        return new NioSocketChannelUnsafe();
  }
```

NioSocketChannel.newUnsafe 方法会返回一个 NioSocketChannelUnsafe 实例. 从这里我们就可以确定了，在实例化的 NioSocketChannel 中的 unsafe 字段，其实是一个

NioSocketChannelUnsafe 的实例.

1.1.5 关于 pipeline 的初始化

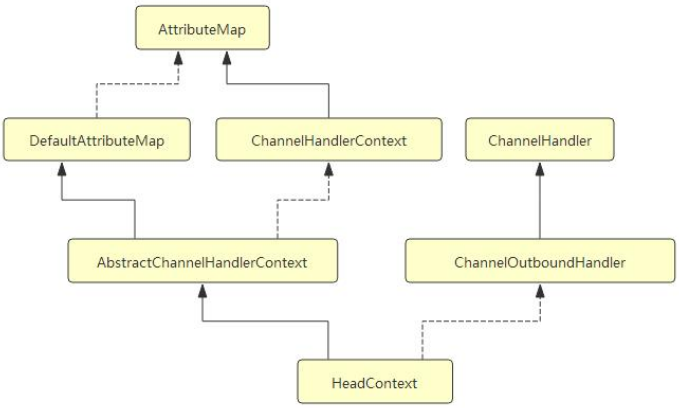
上面我们分析了一个 Channel（在这个例子中是 NioSocketChannel）的大体初始化过程，但是我们漏掉了一个关键的部分，即 ChannelPipeline 的初始化.

根据 Each channel has its own pipeline and it is created automatically when a new channel is created.，我们知道，在实例化一个 Channel 时，必然伴随着实例化一个 ChannelPipeline. 而我们确实在 AbstractChannel 的构造器看到了 pipeline 字段被初始化为 DefaultChannelPipeline 的实例. 那么我们就来看一下，DefaultChannelPipeline 构造器做了哪些工作吧:

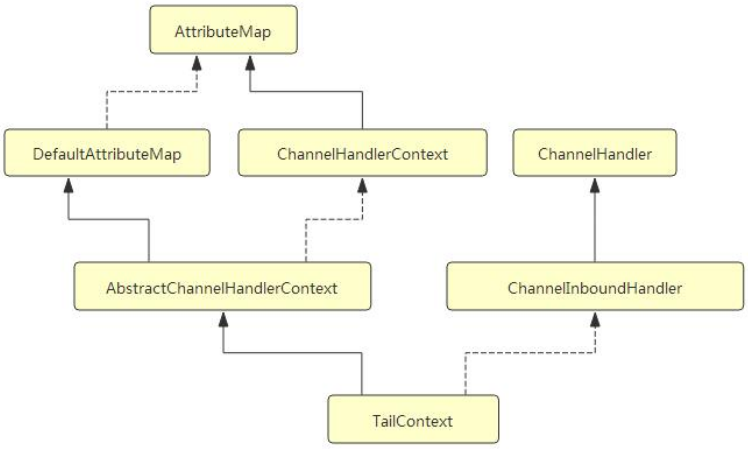
```
protected DefaultChannelPipeline(Channel channel) {
      this.channel = ObjectUtil.checkNotNull(channel, "channel");
      succeededFuture = new SucceededChannelFuture(channel, null);
      voidPromise =  new VoidChannelPromise(channel, true);

      tail = new TailContext(this);
      head = new HeadContext(this);

      head.next = tail;
      tail.prev = head;
  }
```

我们调用 DefaultChannelPipeline 的构造器，传入了一个 channel，而这个 channel 其实就是我们实例化的 NioSocketChannel，DefaultChannelPipeline 会将这个 NioSocketChannel 对象保存在 channel 字段中. DefaultChannelPipeline 中，还有两个特殊的字段，即 head 和 tail，而这两个字段是一个双向链表的头和尾. 其实在 DefaultChannelPipeline 中，维护了一个以 AbstractChannelHandlerContext 为节点的双向链表，这个链表是 Netty 实现 Pipeline 机制的关键. 关于 DefaultChannelPipeline 中的双向链表以及它所起的作用，这一节我们暂时不做详细分析。

先看看 HeadContext 的继承层次结构如下所示:



TailContext 的继承层次结构如下所示:



我们可以看到，链表中 head 是一个 ChannelOutboundHandler，而 tail 则是一个 ChannelInboundHandler.

接着看一下 HeadContext 的构造器：

```
    HeadContext(DefaultChannelPipeline pipeline) {
            super(pipeline, null, HEAD_NAME, false, true);
            unsafe = pipeline.channel().unsafe();
            setAddComplete();
        }
```

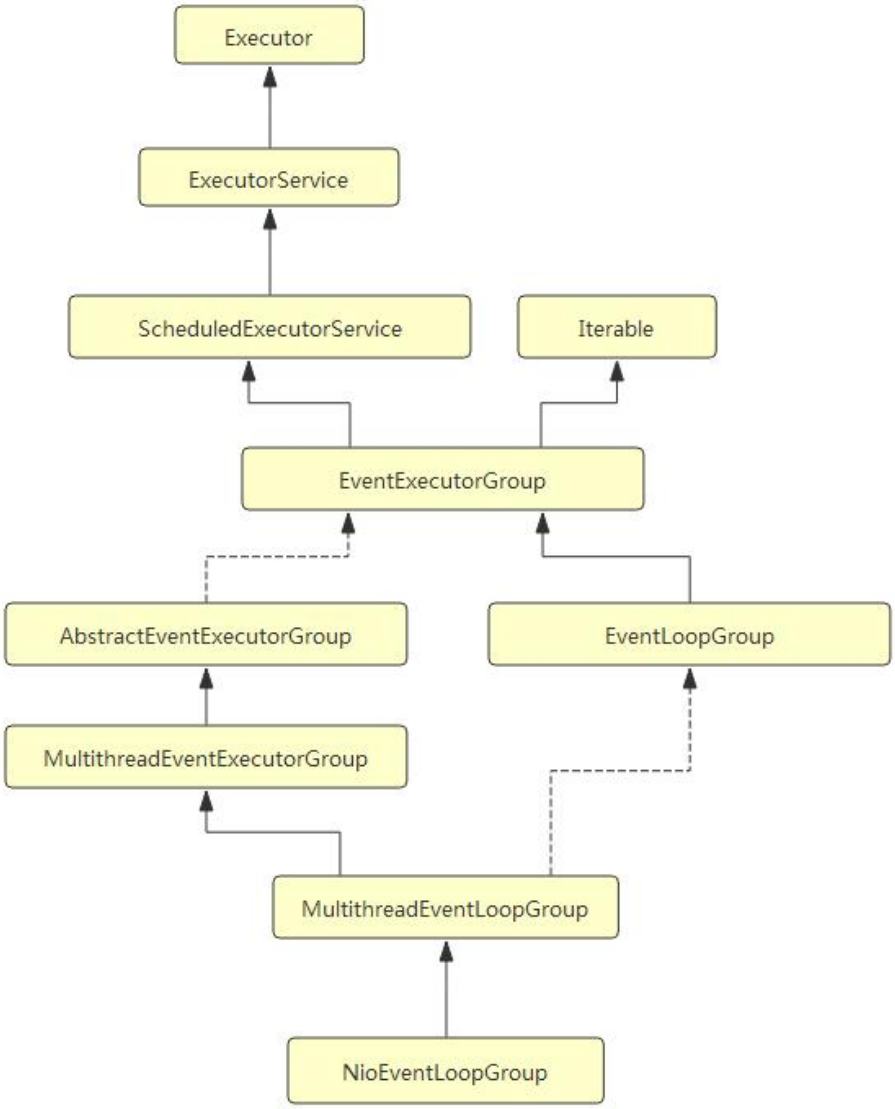
它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = false, outbound = true.

TailContext 的构造器与 HeadContext 的相反，它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = true, outbound = false. 即 header 是一个 outboundHandler，而 tail 是一个 inboundHandler，关于这一点，大家要特别注意，因为在分析到 Netty Pipeline 时，我们会反复用到 inbound 和 outbound 这两个属性.

1.1.6 关于 EventLoop 的初始化

回到最开始的 ChatClient.java 代码中，我们在一开始就实例化了一个 NioEventLoopGroup 对象，因此我们就从它的构造器中追踪一下 EventLoop 的初始化过程.

首先来看一下 NioEventLoopGroup 的类继承层次：



NioEventLoop 有几个重载的构造器，不过内容都没有什么大的区别，最终都是调用的父类 MultithreadEventLoopGroup 构造器：

```
 protected MultithreadEventLoopGroup(int nThreads, Executor executor, Object... args)
{
        super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads, executor, args);
    }
```

其中有一点有意思的地方是，如果我们传入的线程数 nThreads 是 0，那么 Netty 会为我们设置默认的线程数 DEFAULT_EVENT_LOOP_THREADS，而这个默认的线程数是怎么确定的呢？

其实很简单，在静态代码块中，会首先确定 DEFAULT_EVENT_LOOP_THREADS 的值：

```
static {
DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
Runtime.getRuntime().availableProcessors() * 2));}
```

Netty 会首先从系统属性中获取 "io.netty.eventLoopThreads" 的值，如果我们没有设置它的话，那么就返回默认值：处理器核心数 * 2.

回到 MultithreadEventLoopGroup 构造器中，这个构造器会继续调用父类

MultithreadEventExecutorGroup 的构造器:

```
protected MultithreadEventExecutorGroup(int nThreads, Executor executor,
                                        EventExecutorChooserFactory chooserFactory,
Object... args) {
      // 去掉了参数检查，异常处理等代码
       children = new EventExecutor[nThreads];

       for (int i = 0; i < nThreads; i ++) {
       // 去掉了 try...catch...finally 代码块
            children[i] = newChild(executor, args);
       }
       chooser = chooserFactory.newChooser(children);
       // 去掉了包装处理的代码
   }
```

根据代码，我们就很清楚 MultithreadEventExecutorGroup 中的处理逻辑了:

1、创建一个大小为 nThreads 的 SingleThreadEventExecutor 数组

2、根据 nThreads 的大小，创建不同的 Chooser，即如果 nThreads 是 2 的幂，则使用 PowerOfTwoEventExecutorChooser，反之使用 GenericEventExecutorChooser. 不论使用哪个 Chooser，它们的功能都是一样的，即从 children 数组中选出一个合适的 EventExecutor 实例.

3、调用 newChhild 方法初始化 children 数组.

根据上面的代码，我们知道，MultithreadEventExecutorGroup 内部维护了一个 EventExecutor 数组，Netty 的 EventLoopGroup 的实现机制其实就建立在 MultithreadEventExecutorGroup 之上. 每当 Netty 需要一个 EventLoop 时，会调用 next() 方法获取一个可用的 EventLoop.

上面代码的最后一部分是 newChild 方法，这个是一个抽象方法，它的任务是实例化 EventLoop 对象. 我们跟踪一下它的代码，可以发现，这个方法在 NioEventLoopGroup 类中实现了，其内容很简单:

```
   protected EventLoop newChild(Executor executor, Object... args) throws Exception {
       return new NioEventLoop(this, executor, (SelectorProvider) args[0],
           ((SelectStrategyFactory) args[1]).newSelectStrategy(),
(RejectedExecutionHandler) args[2]);
   }
```

其实就是实例化一个 NioEventLoop 对象，然后返回它.

最后总结一下整个 EventLoopGroup 的初始化过程吧:

1、EventLoopGroup(其实是 MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children 数组，其大小是 nThreads，这样就构成了一个线程池

2、如果我们在实例化 NioEventLoopGroup 时，如果指定线程池大小，则 nThreads 就是指定的值，反之是处理器核心数 * 2

3、MultithreadEventExecutorGroup 中会调用 newChild 抽象方法来初始化 children 数组

4、抽象方法 newChild 是在 NioEventLoopGroup 中实现的，它返回一个 NioEventLoop 实例.

5、NioEventLoop 属性:

SelectorProvider provider 属性：NioEventLoopGroup 构造器中通过 SelectorProvider.provider() 获取一个 SelectorProvider

Selector selector 属性：NioEventLoop 构造器中通过调用通过 selector = provider.openSelector() 获取一个 selector 对象.

1.1.7 Channel 的注册过程

在前面的分析中，我们提到，channel 会在 Bootstrap.initAndRegister 中进行初始化，但是这个方法还会将初始化好的 Channel 注册到 EventGroup 中. 接下来我们就来分析一下 Channel 注册的过程.

回顾一下 AbstractBootstrap.initAndRegister 方法:

```
final ChannelFuture initAndRegister() {
     // 删除了非关键代码
       Channelchannel = channelFactory.newChannel();
       init(channel);

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

       return regFuture;
   }
```

当 Channel 初始化后，会紧接着调用 group().register() 方法来注册 Channel，我们继续跟踪的话，会发现其调用链如下:

AbstractBootstrap.initAndRegister -> MultithreadEventLoopGroup.register -> SingleThreadEventLoop.register -> AbstractChannel$AbstractUnsafe.register

通过跟踪调用链，最终我们发现是调用到了 unsafe 的 register 方法，那么接下来我们就仔细看一下 AbstractChannel$AbstractUnsafe.register 方法中到底做了什么:

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
          // 省略了条件判断和错误处理的代码
          AbstractChannel.this.eventLoop = eventLoop;
          register0(promise);
}
```

首先，将 eventLoop 赋值给 Channel 的 eventLoop 属性，而我们知道这个 eventLoop 对象其实是 MultithreadEventLoopGroup.next() 方法获取的，根据我们前面的小节中，我们可以确定 next() 方法返回的 eventLoop 对象是 NioEventLoop 实例.

register 方法接着调用了 register0 方法:

```
private void register0(ChannelPromise promise) {
         // 省略了非关键代码
              boolean firstRegistration = neverRegistered;
              doRegister();
              neverRegistered = false;
              registered = true;

              pipeline.invokeHandlerAddedIfNeeded();

              safeSetSuccess(promise);
```

```
                pipeline.fireChannelRegistered();

                if (isActive()) {
                    if (firstRegistration) {
                        pipeline.fireChannelActive();
                    }
                }
 }
```

register0 又调用了 AbstractNioChannel.doRegister:

```
protected void doRegister() throws Exception {
    // 省略了错误处理的代码
    selectionKey = javaChannel().register(eventLoop().selector, 0, this);
}
```

javaChannel() 这个方法在前面我们已经知道了，它返回的是一个 Java NIO SocketChannel，这里我们将这个 SocketChannel 注册到与 eventLoop 关联的 selector 上了.

我们总结一下 Channel 的注册过程:

1、首先在 AbstractBootstrap.initAndRegister 中，通过 group().register(channel)，调用 MultithreadEventLoopGroup.register 方法

2、在 MultithreadEventLoopGroup.register 中，通过 next() 获取一个可用的 SingleThreadEventLoop，然后调用它的 register

3、在 SingleThreadEventLoop.register 中，通过 channel.unsafe().register(this, promise) 来获取 channel 的 unsafe() 底层操作对象，然后调用它的 register.

4、在 AbstractUnsafe.register 方法中，调用 register0 方法注册 Channel

5、在 AbstractUnsafe.register0 中，调用 AbstractNioChannel.doRegister 方法

6、AbstractNioChannel.doRegister 方法通过 javaChannel().register(eventLoop().selector, 0, this) 将 Channel 对应的 Java NIO SockerChannel 注册到一个 eventLoop 的 Selector 中，并且将当前 Channel 作为 attachment.

总的来说，Channel 注册过程所做的工作就是将 Channel 与对应的 EventLoop 关联，因此这也体现了，在 Netty 中，每个 Channel 都会关联一个特定的 EventLoop，并且这个 Channel 中的所有 IO 操作都是在这个 EventLoop 中执行的；当关联好 Channel 和 EventLoop 后，会继续调用底层的 Java NIO SocketChannel 的 register 方法，将底层的 Java NIO SocketChannel 注册到指定的 selector 中. 通过这两步，就完成了 Netty Channel 的注册过程.

1.1.8 Handler 的添加过程

Netty 的一个强大和灵活之处就是基于 Pipeline 的自定义 handler 机制. 基于此，我们可以像添加插件一样自由组合各种各样的 handler 来完成业务逻辑. 例如我们需要处理 HTTP 数据，那么就可以在 pipeline 前添加一个 Http 的编解码的 Handler，然后接着添加我们自己的业务逻辑的 handler，这样网络上的数据流就向通过一个管道一样，从不同的 handler 中流过并进行编解码，最终在到达我们自定义的 handler 中.

既然说到这里，有些同学肯定会好奇，既然这个 pipeline 机制是这么的强大，那么它是怎么实现的呢？在这里我还不打算详细讲解，在这一小节中，我们从简单的入手，展示一下我们自

定义的 handler 是如何以及何时添加到 ChannelPipeline 中的.
首先让我们看一下如下的代码片段:

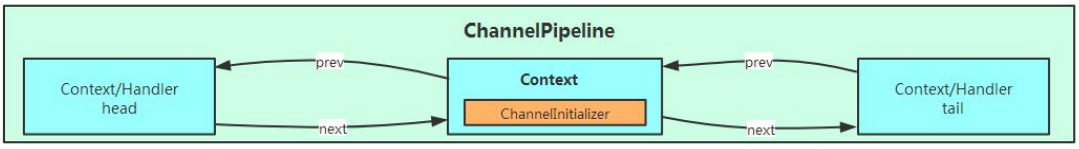
```
// 此处省略 N 句代码
.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    pipeline.addLast(new StringDecoder());
                    pipeline.addLast(new StringEncoder());
                    pipeline.addLast(new ChatClientHandler(nickName));
                }
            });
```

这个代码片段就是实现了 handler 的添加功能. 我们看到, Bootstrap.handler 方法接收一个 ChannelHandler, 而我们传递的是一个 派生于 ChannelInitializer 的匿名类, 它正好也实现了 ChannelHandler 接口. 我们来看一下, ChannelInitializer 类内到底有什么玄机:

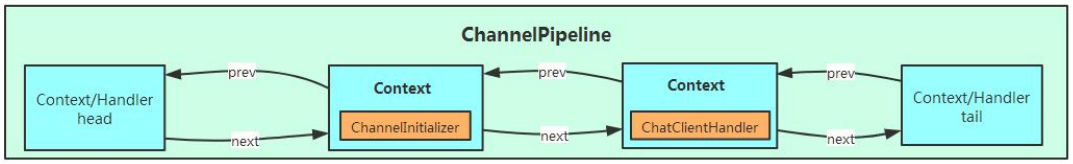
```
public abstract class ChannelInitializer<C extends Channel> extends
ChannelInboundHandlerAdapter {
    private static final InternalLogger logger =
InternalLoggerFactory.getInstance(ChannelInitializer.class);
    private final ConcurrentMap<ChannelHandlerContext, Boolean> initMap =
PlatformDependent.newConcurrentHashMap();

    protected abstract void initChannel(C ch) throws Exception;

    @Override
    @SuppressWarnings("unchecked")
    public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        if (initChannel(ctx)) {
            ctx.pipeline().fireChannelRegistered();
        } else {
            ctx.fireChannelRegistered();
        }
    }
    // 这个方法在 channelRegistered 中被调用
    private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
        initChannel((C) ctx.channel());
        remove(ctx);
        return false;
    }
    // 省略...
}
```

ChannelInitializer 是一个抽象类, 它有一个抽象的方法 initChannel, 我们正是实现了这个方法, 并在这个方法中添加的自定义的 handler 的. 那么 initChannel 是哪里被调用的呢? 答案是 ChannelInitializer.channelRegistered 方法中.

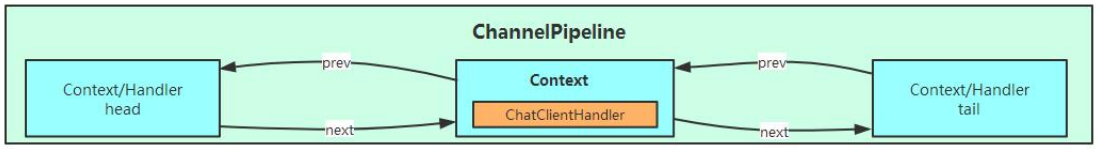
我们来关注一下 channelRegistered 方法. 从上面的源码中, 我们可以看到, 在 channelRegistered 方法中, 会调用 initChannel 方法, 将自定义的 handler 添加到 ChannelPipeline 中, 然后调用 ctx.pipeline().remove(this) 将自己从 ChannelPipeline 中删除. 上面的分析过程, 可以用如下图片展示:
一开始, ChannelPipeline 中只有三个 handler, head, tail 和我们添加的 ChannelInitializer.



接着 initChannel 方法调用后, 添加了自定义的 handler:



最后将 ChannelInitializer 删除:



分析到这里, 我们已经简单了解了自定义的 handler 是如何添加到 ChannelPipeline 中的, 在后面的我们再进行深入的探讨.

1.1.9 客户端连接分析

经过上面的各种分析后, 我们大致了解了 Netty 初始化时, 所做的工作, 那么接下来我们就直奔主题, 分析一下客户端是如何发起 TCP 连接的.

首先, 客户端通过调用 Bootstrap 的 connect 方法进行连接.
在 connect 中, 会进行一些参数检查后, 最终调用的是 doConnect 方法, 其实现如下:

```
private static void doConnect(
        final SocketAddress remoteAddress,
        final SocketAddress localAddress,
        final ChannelPromise connectPromise) {
        final Channel channel = connectPromise.channel();
        channel.eventLoop().execute(new Runnable() {
            @Override
            public void run() {
                if (localAddress == null) {
                    channel.connect(remoteAddress, connectPromise);
                } else {
                    channel.connect(remoteAddress, localAddress, connectPromise);
```

```
                }
                connectPromise.addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
            }
        });
    }
```

在 doConnect 中，会在 event loop 线程中调用 Channel 的 connect 方法，而这个 Channel 的具体类型是什么呢？我们在前面已经分析过了，这里 channel 的类型就是 NioSocketChannel.

进行跟踪到 channel.connect 中，我们发现它调用的是 DefaultChannelPipeline.connect，而，pipeline 的 connect 代码如下：

```
public final ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise)
{
        return tail.connect(remoteAddress, promise);
}
```

而 tail 字段，我们已经分析过了，是一个 TailContext 的实例，而 TailContext 又是 AbstractChannelHandlerContext 的子类，并且没有实现 connect 方法，因此这里调用的其实是 AbstractChannelHandlerContext.connect，我们看一下这个方法的实现：

```
public ChannelFuture connect(
            final SocketAddress remoteAddress,
            final SocketAddress localAddress, final ChannelPromise promise) {
        // 删除参数检查的代码
        final AbstractChannelHandlerContext next = findContextOutbound();
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            next.invokeConnect(remoteAddress, localAddress, promise);
        } else {
            safeExecute(executor, new Runnable() {
                @Override
                public void run() {
                    next.invokeConnect(remoteAddress, localAddress, promise);
                }
            }, promise, null);
        }
        return promise;
    }
```

上面的代码中有一个关键的地方，即 final AbstractChannelHandlerContext next = findContextOutbound()，这里调用 findContextOutbound 方法，从 DefaultChannelPipeline 内的双向链表的 tail 开始，不断向前寻找第一个 outbound 为 true 的 AbstractChannelHandlerContext，然后调用它的 invokeConnect 方法，其代码如下：

```
private void invokeConnect(SocketAddress remoteAddress, SocketAddress localAddress,
ChannelPromise promise) {
 // 忽略 try...catch 块
 ((ChannelOutboundHandler) handler()).connect(this, remoteAddress, localAddress,
promise);
```

```
}
```

前面我们提到，在 DefaultChannelPipeline 的构造器中，会实例化两个对象：head 和 tail，并形成了双向链表的头和尾。head 是 HeadContext 的实例，它实现了 ChannelOutboundHandler 接口，并且它的 outbound 字段为 true。因此在 findContextOutbound 中，找到的 AbstractChannelHandlerContext 对象其实就是 head。进而在 invokeConnect 方法中，我们向上转换为 ChannelOutboundHandler 就是没问题的了。

而又因为 HeadContext 重写了 connect 方法，因此实际上调用的是 HeadContext.connect。我们接着跟踪到 HeadContext.connect，其代码如下：

```
public void connect(
                ChannelHandlerContext ctx,
                SocketAddress remoteAddress, SocketAddress localAddress,
                ChannelPromise promise) throws Exception {
            unsafe.connect(remoteAddress, localAddress, promise);
        }
```

这个 connect 方法很简单，仅仅调用了 unsafe 的 connect 方法。而 unsafe 又是什么呢？回顾一下 HeadContext 的构造器，我们发现 unsafe 是 pipeline.channel().unsafe() 返回的是 Channel 的 unsafe 字段，在这这里，我们已经知道了，其实是 AbstractNioByteChannel.NioByteUnsafe 内部类。兜兜转转了一大圈，我们找到了创建 Socket 连接的关键代码。

进行跟踪 NioByteUnsafe -> AbstractNioUnsafe.connect：

```
public final void connect(
                final SocketAddress remoteAddress, final SocketAddress localAddress,
final ChannelPromise promise) {

                // 省去前面的判断
                boolean wasActive = isActive();
                if (doConnect(remoteAddress, localAddress)) {
                    fulfillConnectPromise(promise, wasActive);
                } else {
                    // 此处省略 N 行代码
                }
}
```

AbstractNioUnsafe.connect 的实现如上代码所示，在这个 connect 方法中，调用了 doConnect 方法，注意，这个方法并不是 AbstractNioUnsafe 的方法，而是 AbstractNioChannel 的抽象方法。doConnect 方法是在 NioSocketChannel 中实现的，因此进入到 NioSocketChannel.doConnect 中：

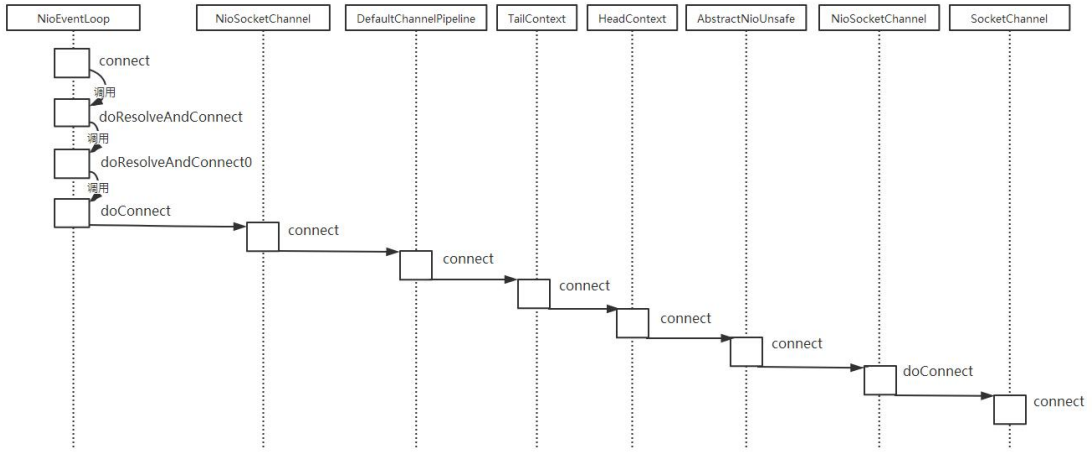
```
protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress)
throws Exception {
        if (localAddress != null) {
            doBind0(localAddress);
        }

        boolean success = false;
```

```
        try {
            boolean connected = javaChannel().connect(remoteAddress);
            if (!connected) {
                selectionKey().interestOps(SelectionKey.OP_CONNECT);
            }
            success = true;
            return connected;
        } finally {
            if (!success) {
                doClose();
            }
        }
    }
```

我们终于看到的最关键的部分了，庆祝一下！

上面的代码不用多说，首先是获取 Java NIO SocketChannel，从 NioSocketChannel.newSocket 返回的 SocketChannel 对象；然后是调用 SocketChannel.connect 方法完成 Java NIO 层面上的 Socket 的连接.

最后，上面的代码流程可以用如下时序图直观地展示：



1.2. 服务端 ServerBootStrap

在分析客户端的代码时，我们已经对 Bootstrap 启动 Netty 有了一个大致的认识，那么接下来分析服务器端时，就会相对简单一些了.

首先还是来看一下服务器端的启动代码：

```
public class ChatServer {
    public void start(int port) throws Exception{
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
```

```
            .channel(NioServerSocketChannel.class)
            .childHandler(new ChannelInitializer<SocketChannel>(){
                  @Override
                  public void initChannel(SocketChannel ch) throws Exception {
                      ChannelPipeline pipeline = ch.pipeline();

                      pipeline.addLast(new HttpServerCodec());
                      pipeline.addLast(new HttpObjectAggregator(64 * 1024));
                      pipeline.addLast(new ChunkedWriteHandler());
                      pipeline.addLast(new HttpHandler());

                      pipeline.addLast(new WebSocketServerProtocolHandler("/im"));
                      pipeline.addLast(new WebSocktHandler());

                      pipeline.addLast(new StringDecoder());
                      pipeline.addLast(new StringEncoder());
                      pipeline.addLast(new SocketHandler());

                  }
              })
            .option(ChannelOption.SO_BACKLOG, 128)
            .childOption(ChannelOption.SO_KEEPALIVE, true);

          System.out.println("聊天服务已启动,监听端口" + port + "");

          // 绑定端口，开始接收进来的连接
          ChannelFuture f = b.bind(port).sync();

          // 等待服务器  socket 关闭 。
          // 在这个例子中，这不会发生，但你可以优雅地关闭你的服务器。
          f.channel().closeFuture().sync();

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

          System.out.println("聊天服务已关闭");
      }
  }


  public static void main(String[] args) {
      try {
          new ChatServer().start(8080);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

和客户端的代码相比，没有很大的差别，基本上也是进行了如下几个部分的初始化：

1、EventLoopGroup：不论是服务器端还是客户端，都必须指定 EventLoopGroup. 在这个例子中，指定了 NioEventLoopGroup，表示一个 NIO 的 EventLoopGroup，不过服务器端需要指定两个 EventLoopGroup，一个是 bossGroup，用于处理客户端的连接请求；另一个是 workerGroup，用于处理与各个客户端连接的 IO 操作.

2、ChannelType：指定 Channel 的类型. 因为是服务器端，因此使用了 NioServerSocketChannel.

3、Handler：设置数据的处理器.

1.2.1 Channel 的初始化过程

我们在分析客户端的 Channel 初始化过程时，已经提到，Channel 是对 Java 底层 Socket 连接的抽象，并且知道了客户端的 Channel 的具体类型是 NioSocketChannel，那么自然的，服务器端的 Channel 类型就是 NioServerSocketChannel 了.

那么接下来我们按照分析客户端的流程对服务器端的代码也同样地分析一遍，这样也方便我们对比一下服务器端和客户端有哪些不一样的地方.

1.2.2 Channel 类型的确定

同样的分析套路，我们已经知道了，在客户端中，Channel 的类型其实是在初始化时，通过 Bootstrap.channel() 方法设置的，服务器端自然也不例外.

在服务器端，我们调用了 ServerBootstarap.channel(NioServerSocketChannel.class)，传递了一个 NioServerSocketChannel Class 对象. 这样的话，按照和分析客户端代码一样的流程，我们就可以确定，NioServerSocketChannel 的实例化是通过 ReflectiveChannelFactory 工厂类来完成的，而 ReflectiveChannelFactory 中的 clazz 字段被设置为了 NioServerSocketChannel.class，因此当调用 ReflectiveChannelFactory.newChannel() 时：

```
public T newChannel() {
      // 删除了 try 块
      return clazz.newInstance();
}
```

就获取到了一个 NioServerSocketChannel 的实例.

最后我们也来总结一下：

1、ServerBootstrap 中的 ChannelFactory 的实现是 ReflectiveChannelFactory

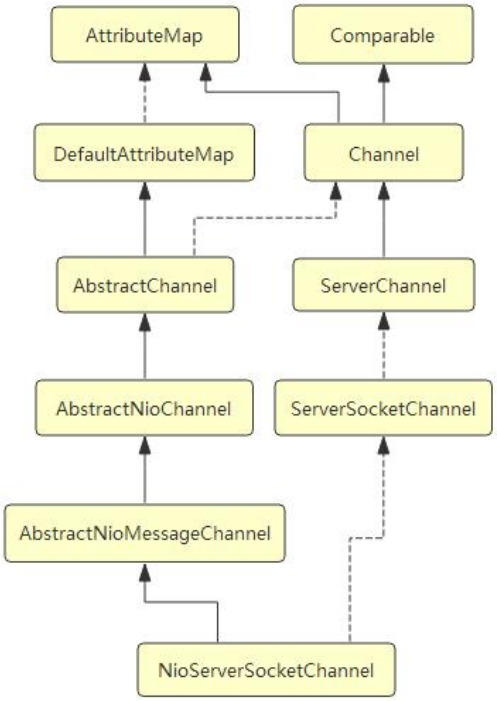
2、生成的 Channel 的具体类型是 NioServerSocketChannel.

Channel 的实例化过程，其实就是调用的 ChannelFactory.newChannel 方法，而实例化的 Channel 的具体的类型又是和在初始化 ServerBootstrap 时传入的 channel() 方法的参数相关. 因此对于我们这个例子中的服务器端的 ServerBootstrap 而言，生成的的 Channel 实例就是 NioServerSocketChannel.

1.2.3 NioServerSocketChannel 的实例化过程

首先还是来看一下 NioServerSocketChannel 的实例化过程.

下面是 NioServerSocketChannel 的类层次结构图：



首先，我们来看一下它的默认的构造器. 和 NioSocketChannel 类似，构造器都是调用了 newSocket 来打开一个 Java 的 NIO Socket，不过需要注意的是，客户端的 newSocket 调用的是 openSocketChannel，而服务器端的 newSocket 调用的是 openServerSocketChannel. 顾名思义，一个是客户端的 Java SocketChannel，一个是服务器端的 Java ServerSocketChannel.

```
private static ServerSocketChannel newSocket(SelectorProvider provider) {
      return provider.openServerSocketChannel();
}
public NioServerSocketChannel() {
      this(newSocket(DEFAULT_SELECTOR_PROVIDER));
}
```

接下来会调用重载的构造器：

```
public NioServerSocketChannel(ServerSocketChannel channel) {
        super(null, channel, SelectionKey.OP_ACCEPT);
        config = new NioServerSocketChannelConfig(this, javaChannel().socket());
}
```

这个构造其中，调用父类构造器时，传入的参数是 SelectionKey.OP_ACCEPT. 作为对比，我们回想一下，在客户端的 Channel 初始化时，传入的参数是 SelectionKey.OP_READ. 我前面讲过，Java NIO 是一种 Reactor 模式，我们通过 selector 来实现 I/O 的多路复用复用. 在一开始时，服务器端需要监听客户端的连接请求，因此在这里我们设置了 SelectionKey.OP_ACCEPT，即通知 selector 我们对客户端的连接请求感兴趣.

接着和客户端的分析一下，会逐级地调用父类的构造器 NioServerSocketChannel -> AbstractNioMessageChannel -> AbstractNioChannel -> AbstractChannel.

同样的，在 AbstractChannel 中会实例化一个 unsafe 和 pipeline：

```
protected AbstractChannel(Channel parent) {
        this.parent = parent;
```

```
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
}
```

不过，这里有一点需要注意的是，客户端的 unsafe 是一个 AbstractNioByteChannel#NioByteUnsafe 的实例，而在服务器端时，因为 AbstractNioMessageChannel 重写了 newUnsafe 方法：

```
protected AbstractNioUnsafe newUnsafe() {
        return new NioMessageUnsafe();
}
```

因此在服务器端，unsafe 字段其实是一个 AbstractNioMessageChannel.AbstractNioUnsafe 的实例.

我们来总结一下，在 NioServerSocketChannsl 实例化过程中，所需要做的工作：

1、调用 NioServerSocketChannel.newSocket(DEFAULT_SELECTOR_PROVIDER) 打开一个新的 Java NIO ServerSocketChannel

2、AbstractChannel(Channel parent) 中初始化 AbstractChannel 的属性：

parent 属性置为 null

unsafe 通过 newUnsafe() 实例化一个 unsafe 对象，它的类型是 AbstractNioMessageChannel#AbstractNioUnsafe 内部类

pipeline 是 new DefaultChannelPipeline(this) 新创建的实例.

3、AbstractNioChannel 中的属性：

SelectableChannel ch 被设置为 Java ServerSocketChannel，即 NioServerSocketChannel#newSocket 返回的 Java NIO ServerSocketChannel.

readInterestOp 被设置为 SelectionKey.OP_ACCEPT

SelectableChannel ch 被配置为非阻塞的 ch.configureBlocking(false)

4、NioServerSocketChannel 中的属性：

ServerSocketChannelConfig config = new NioServerSocketChannelConfig(this, javaChannel().socket())

1.2.4 ChannelPipeline 初始化

服务器端和客户端的 ChannelPipeline 的初始化一致，因此就不再单独分析了.

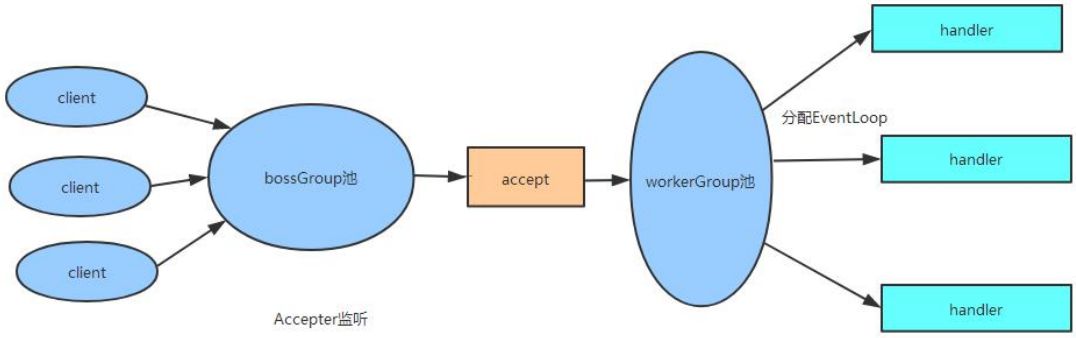
1.2.5 Channel 的注册

服务器端和客户端的 Channel 的注册过程一致，因此就不再单独分析了.

1.2.6 关于 bossGroup 与 workerGroup

在客户端的时候，我们只提供了一个 EventLoopGroup 对象，而在服务器端的初始化时，我们设置了两个 EventLoopGroup，一个是 bossGroup，另一个是 workerGroup. 那么这两个 EventLoopGroup 都是干什么用的呢？其实呢，bossGroup 是用于服务端 的 accept 的，即用于处理客户端的连接请求. 我们可以把 Netty 比作一个饭店，bossGroup 就像一个像一个前台接待，当客户来到饭店吃时，接待员就会引导顾客就坐，为顾客端茶送水等. 而 workerGroup，其实就是实际上干活的啦，它们负责客户端连接通道的 IO 操作：当接待员 招待好顾客后，就可以稍做休息，而此时后厨里的厨师们(workerGroup)就开始忙碌地准备饭菜了.

关于 bossGroup 与 workerGroup 的关系，我们可以用如下图来展示：



首先，服务器端 bossGroup 不断地监听是否有客户端的连接，当发现有一个新的客户端连接到来时，bossGroup 就会为此连接初始化各项资源，然后从 workerGroup 中选出一个 EventLoop 绑定到此客户端连接中. 那么接下来的服务器与客户端的交互过程就全部在此分配的 EventLoop 中了.

口说无凭，我们还是以源码说话吧.

首先在 ServerBootstrap 初始化时，调用了 b.group(bossGroup, workerGroup) 设置了两个 EventLoopGroup，我们跟踪进去看一下:

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
        super.group(parentGroup);
       // 此处省略 N 行代码
        this.childGroup = childGroup;
        return this;
}
```

显然，这个方法初始化了两个字段，一个是 group = parentGroup，它是在 super.group(parentGroup) 中初始化的，另一个是 childGroup = childGroup. 接着我们启动程序调用了 b.bind 方法来监听一个本地端口. bind 方法会触发如下的调用链:

AbstractBootstrap.bind -> AbstractBootstrap.doBind -> AbstractBootstrap.initAndRegister

源码看到到这里为止，AbstractBootstrap.initAndRegister 已经是我们的老朋友了，我们在分析客户端程序时，和它打过很多交到了，现在再来回顾一下这个方法吧:

```
final ChannelFuture initAndRegister() {
        // 省略异常判断
        Channel channel = channelFactory.newChannel();
        init(channel);
        // 省略非关键代码
        ChannelFuture regFuture = config().group().register(channel);

        return regFuture;
    }
```

这里 group() 方法返回的是上面我们提到的 bossGroup，而这里的 channel 我们也已经分析过了，它是一个 NioServerSocketChannsl 实例，因此我们可以知道，group().register(channel) 将 bossGroup 和 NioServerSocketChannsl 关联起来了.

那么 workerGroup 是在哪里与 NioServerSocketChannel 关联的呢?

我们继续看 init(channel) 方法:

```
void init(Channel channel) throws Exception {
```

```
        // 省略参数判断
        ChannelPipeline p = channel.pipeline();

        final EventLoopGroup currentChildGroup = childGroup;
        final ChannelHandler currentChildHandler = childHandler;
        final Entry<ChannelOption<?>, Object>[] currentChildOptions;
        final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

        // 省略非关键代码

        p.addLast(new ChannelInitializer<Channel>() {
            @Override
            public void initChannel(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(
                                currentChildGroup, currentChildHandler,
currentChildOptions, currentChildAttrs));
                    }
                });
            }
        });
    }
```

init 方法在 ServerBootstrap 中重写了，从上面的代码片段中我们看到，它为 pipeline 中添加了一个 ChannelInitializer，而这个 ChannelInitializer 中添加了一个关键的 ServerBootstrapAcceptor handler. 关于 handler 的添加与初始化的过程，我们留待下一小节中分析，我们现在关注一下 ServerBootstrapAcceptor 类.

ServerBootstrapAcceptor 中重写了 channelRead 方法，其主要代码如下:

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
            final Channel child = (Channel) msg;
            child.pipeline().addLast(childHandler);
            // 省略非关键代码
                childGroup.register(child).addListener(...);

}
```

ServerBootstrapAcceptor 中的 childGroup 是构造此对象是传入的 currentChildGroup，即我们的 workerGroup，而 Channel 是一个 NioSocketChannel 的实例，因此这里的 childGroup.register 就是将 workerGroup 中的某个 EventLoop 和 NioSocketChannel 关联了. 既然这样，那么现在的问题是，

ServerBootstrapAcceptor.channelRead 方法是怎么被调用的呢？其实当一个 client 连接到 server 时，Java 底层的 NIO ServerSocketChannel 会有一个 SelectionKey.OP_ACCEPT 就绪事件，接着就会调用到 NioServerSocketChannel.doReadMessages：

```
protected int doReadMessages(List<Object> buf) throws Exception {
        SocketChannel ch = javaChannel().accept();
        // 省略异常处理
        buf.add(new NioSocketChannel(this, ch));
        return 1;
        // 省略错误处理
}
```

在 doReadMessages 中，通过 javaChannel().accept() 获取到客户端新连接的 SocketChannel，接着就实例化一个 NioSocketChannel，并且传入 NioServerSocketChannel 对象(即 this)，由此可知，我们创建的这个 NioSocketChannel 的父 Channel 就是 NioServerSocketChannel 实例 .

接下来就经由 Netty 的 ChannelPipeline 机制，将读取事件逐级发送到各个 handler 中，于是就会触发前面我们提到的 ServerBootstrapAcceptor.channelRead 方法啦.

1.2.6 Handler 的添加过程

服务器端的 handler 的添加过程和客户端的有点区别，和 EventLoopGroup 一样，服务器端的 handler 也有两个，一个是通过 handler() 方法设置 handler 字段，另一个是通过 childHandler() 设置 childHandler 字段. 通过前面的 bossGroup 和 workerGroup 的分析，其实我们在这里可以大胆地猜测：handler 字段与 accept 过程有关，即这个 handler 负责处理客户端的连接请求；而 childHandler 就是负责和客户端的连接的 IO 交互.

那么实际上是不是这样的呢？来，我们继续通过代码证明.

在 关于 bossGroup 与 workerGroup 小节中，我们提到，ServerBootstrap 重写了 init 方法，在这个方法中添加了 handler：

```
void init(Channel channel) throws Exception {
       // 省去逻辑判断
        ChannelPipeline p = channel.pipeline();

        final EventLoopGroup currentChildGroup = childGroup;
        final ChannelHandler currentChildHandler = childHandler;
        final Entry<ChannelOption<?>, Object>[] currentChildOptions;
        final Entry<AttributeKey<?>, Object>[] currentChildAttrs;

        p.addLast(new ChannelInitializer<Channel>() {
            @Override
            public void initChannel(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(
                            currentChildGroup, currentChildHandler,
currentChildOptions, currentChildAttrs));
                }
            });
        }
    });
}
```

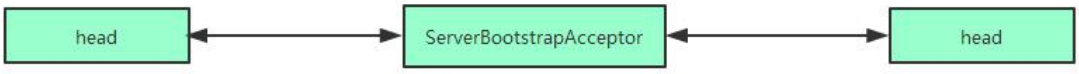
上面代码的 initChannel 方法中，首先通过 handler() 方法获取一个 handler，如果获取的 handler 不为空,则添加到 pipeline 中. 然后接着，添加了一个 ServerBootstrapAcceptor 实例. 那么这里 handler() 方法返回的是哪个对象呢? 其实它返回的是 handler 字段，而这个字段就是我们在服务器端的启动代码中设置的:

```
b.group(bossGroup, workerGroup)
```

那么这个时候，pipeline 中的 handler 情况如下:

ServerBootstrap.init方法中

根据我们原来分析客户端的经验，我们指定，当 channel 绑定到 eventLoop 后(在这里是 NioServerSocketChannel 绑定到 bossGroup) 中时，会在 pipeline 中发出 fireChannelRegistered 事件，接着就会触发 ChannelInitializer.initChannel 方法的调用. 因此在绑定完成后，此时的 pipeline 的内如下:



channelRegistered后的pipeline内容

前面我们在分析 bossGroup 和 workerGroup 时，已经知道了在 ServerBootstrapAcceptor.channelRead 中会为新建的 Channel 设置 handler 并注册到一个 eventLoop 中，即:

```
public void channelRead(ChannelHandlerContext ctx, Object msg) {
        final Channel child = (Channel) msg;
        child.pipeline().addLast(childHandler);
        // 省去非关键代码
        childGroup.register(child).addListener(...);

}
```

而这里的 childHandler 就是我们在服务器端启动代码中设置的 handler:

```
...
.childHandler(new ChannelInitializer<SocketChannel>(){
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
```

```
                    ...
                }
            })
```

后续的步骤就没有什么好说的了，当这个客户端连接 Channel 注册后，就会触发 ChannelInitializer.initChannel 方法的调用。

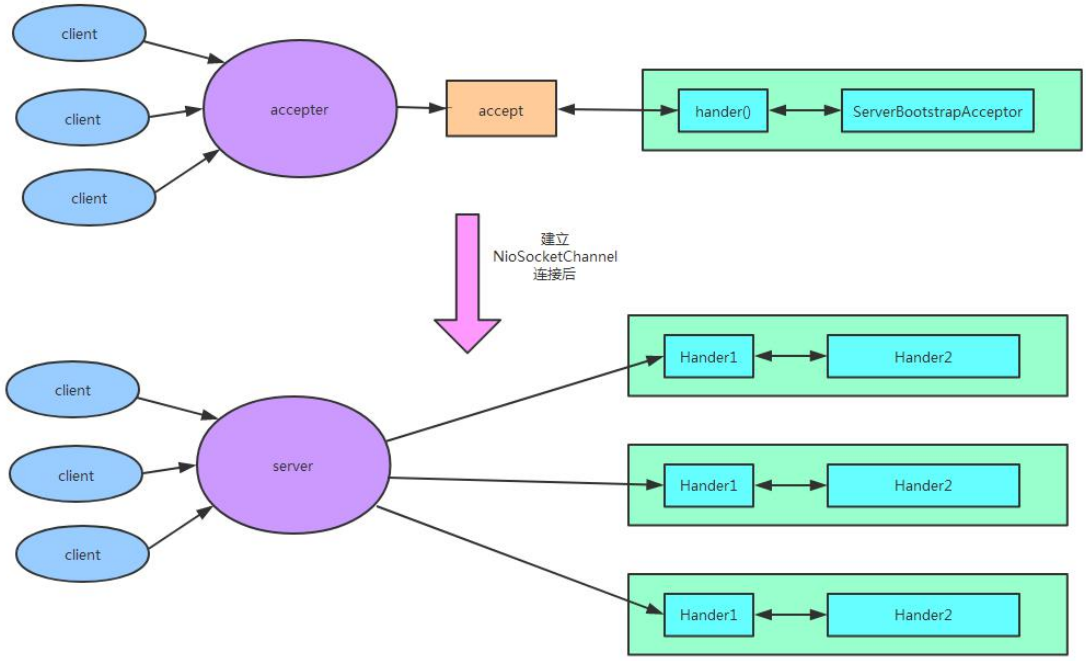
最后我们来总结一下服务器端的 handler 与 childHandler 的区别与联系：

1、在服务器 NioServerSocketChannel 的 pipeline 中添加的是 handler 与 ServerBootstrapAcceptor.

2、当有新的客户端连接请求时，ServerBootstrapAcceptor.channelRead 中负责新建此连接的 NioSocketChannel 并添加 childHandler 到 NioSocketChannel 对应的 pipeline 中，并将此 channel 绑定到 workerGroup 中的某个 eventLoop 中.

3、handler 是在 accept 阶段起作用，它处理客户端的连接请求.

4、childHandler 是在客户端连接建立以后起作用，它负责客户端连接的 IO 交互.

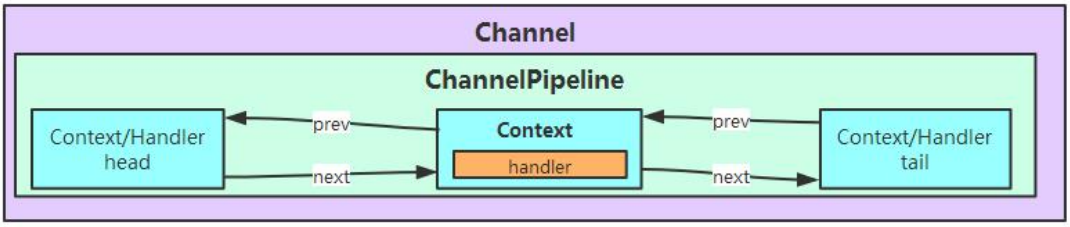


2. Netty 大动脉 ChannelPipeline

2.1. ChannelPipeline

2.1.1 Channel 与 ChannelPipeline

相信大家都知道了，在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应，它们的组成关系如下：



通过上图我们可以看到，一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 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 中又关联着一个 ChannelHandler.

上面的图示给了我们一个对 ChannelPipeline 的直观认识，但是实际上 Netty 实现的 Channel 是否真的是这样的呢？我们继续用源码说话.

在前我们已经知道了一个 Channel 的初始化的基本过程，下面我们再回顾一下.

下面的代码是 AbstractChannel 构造器：

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

AbstractChannel 有一个 pipeline 字段，在构造器中会初始化它为 DefaultChannelPipeline 的实例. 这里的代码就印证了一点：每个 Channel 都有一个 ChannelPipeline.

接着我们跟踪一下 DefaultChannelPipeline 的初始化过程.

首先进入到 DefaultChannelPipeline 构造器中：

```
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;
}
```

在 DefaultChannelPipeline 构造器中，首先将与之关联的 Channel 保存到字段 channel 中，然后实例化两个 ChannelHandlerContext，一个是 HeadContext 实例 head，另一个是 TailContext 实例 tail. 接着将 head 和 tail 互相指向，构成一个双向链表.

特别注意到，我们在开始的示意图中，head 和 tail 并没有包含 ChannelHandler，这是因为 HeadContext 和 TailContext 继承于 AbstractChannelHandlerContext 的同时也实现了 ChannelHandler 接口了，因此它们有 Context 和 Handler 的双重属性.

2.1.2 再探 ChannelPipeline 的初始化

前面 我们已经对 ChannelPipeline 的初始化有了一个大致的了解，不过当时重点毕竟不在 ChannelPipeline 这里，因此没有深入地分析它的初始化过程. 那么下面我们就来看一下具体的 ChannelPipeline 的初始化都做了哪些工作吧.

先回顾一下，在实例化一个 Channel 时，会伴随着一个 ChannelPipeline 的实例化，并且此 Channel 会与这个 ChannelPipeline 相互关联，这一点可以通过 NioSocketChannel 的父类 AbstractChannel 的构造器予以佐证:

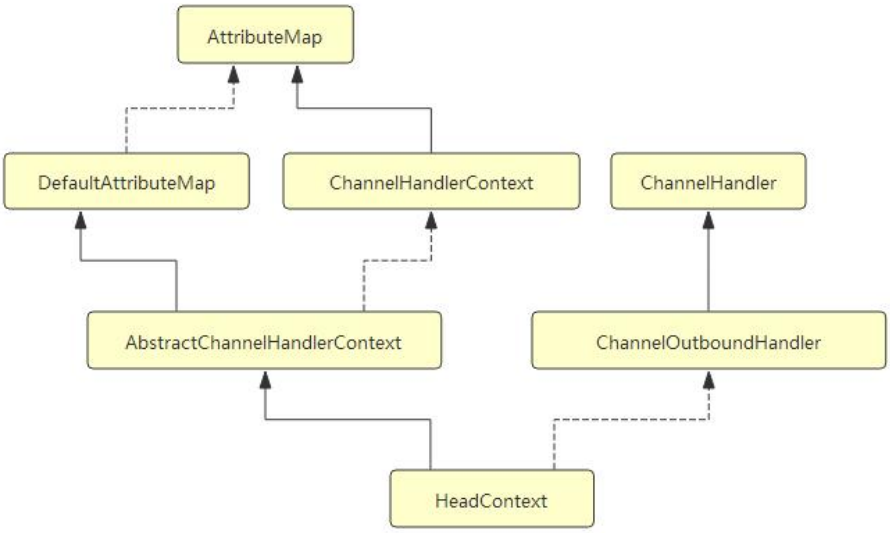
```
protected AbstractChannel(Channel parent) {
        this.parent = parent;
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
}
```

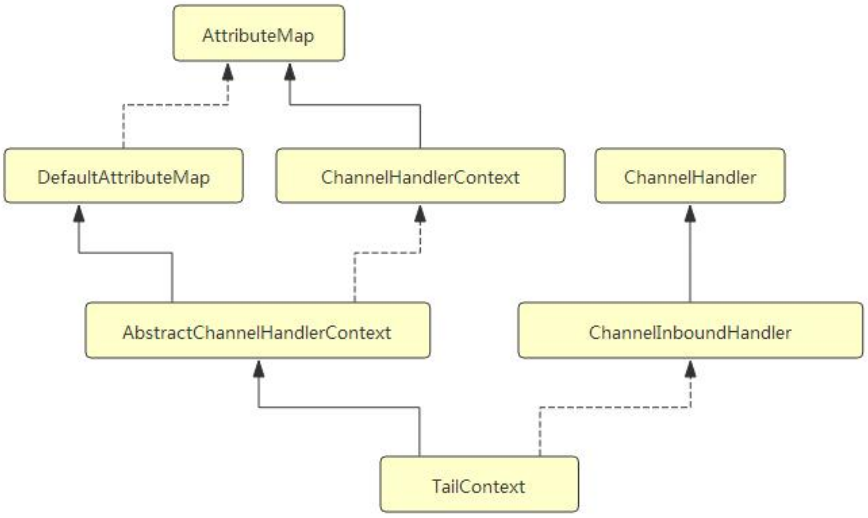
当实例化一个 Channel(这里以 EchoClient 为例，那么 Channel 就是 NioSocketChannel)，其 pipeline 字段就是我们新创建的 DefaultChannelPipeline 对象，那么我们就来看一下 DefaultChannelPipeline 的构造方法吧:

```
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;
}
```

可以看到，在 DefaultChannelPipeline 的构造方法中，将传入的 channel 赋值给字段 this.channel，接着又实例化了两个特殊的字段: tail 与 head. 这两个字段是一个双向链表的头和尾. 其实在 DefaultChannelPipeline 中，维护了一个以 AbstractChannelHandlerContext 为节点的双向链表，这个链表是 Netty 实现 Pipeline 机制的关键.

再回顾一下 head 和 tail 的类层次结构:





从类层次结构图中可以很清楚地看到，head 实现了 ChannelInboundHandler，而 tail 实现了 ChannelOutboundHandler 接口，并且它们都实现了 ChannelHandlerContext 接口，因此可以说 head 和 tail 即是一个 ChannelHandler，又是一个 ChannelHandlerContext.

接着看一下 HeadContext 的构造器:

```
HeadContext(DefaultChannelPipeline pipeline) {
            super(pipeline, null, HEAD_NAME, false, true);
            unsafe = pipeline.channel().unsafe();
            setAddComplete();
}
```

它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = false, outbound = true.

TailContext 的构造器与 HeadContext 的相反，它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = true, outbound = false. 即 header 是一个 outboundHandler，而 tail 是一个 inboundHandler，关于这一点，大家要特别注意，因为在后面的分析中，我们会反复用到 inbound 和 outbound 这两个属性.

2.1.3 ChannelInitializer 的添加

前面我们已经分析了 Channel 的组成，其中我们了解到，最开始的时候 ChannelPipeline 中含有两个 ChannelHandlerContext(同时也是 ChannelHandler)，但是这个 Pipeline 并不能实现什么特殊的功能，因为我们还没有给它添加自定义的 ChannelHandler.

通常来说，我们在初始化 Bootstrap，会添加我们自定义的 ChannelHandler，就以我们熟悉的 ChatClient 来举例吧：

```
Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(group)
            .channel(NioSocketChannel.class)
            .option(ChannelOption.SO_KEEPALIVE, true)
            .handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    pipeline.addLast(new ChatClientHandler(nickName));
                }
            });
```

上面代码的初始化过程，相信大家都不陌生. 在调用 handler 时，传入了 ChannelInitializer 对象，它提供了一个 initChannel 方法供我们初始化 ChannelHandler. 那么这个初始化过程是怎样的呢？下面我们就来揭开它的神秘面纱.

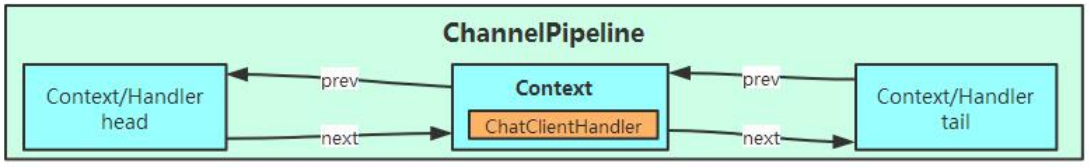
ChannelInitializer 实现了 ChannelHandler，那么它是在什么时候添加到 ChannelPipeline 中的呢？进行了一番搜索后，我们发现它是在 Bootstrap.init 方法中添加到 ChannelPipeline 中的.

其代码如下：

```
void init(Channel channel) throws Exception {
        ChannelPipeline p = channel.pipeline();
        p.addLast(config.handler());
       //略去 N 句代码
}
```

上面的代码将 handler() 返回的 ChannelHandler 添加到 Pipeline 中，而 handler() 返回的是 handler 其实就是我们在初始化 Bootstrap 调用 handler 设置的 ChannelInitializer 实例，因此这里就是将 ChannelInitializer 插入到了 Pipeline 的末端.

此时 Pipeline 的结构如下图所示：



有同学可能就有疑惑了，我明明插入的是一个 ChannelInitializer 实例，为什么在 ChannelPipeline 中的双向链表中的元素却是一个 ChannelHandlerContext? 为了解答这个问题，我们继续在代码中寻找答案吧.

我们刚才提到，在 Bootstrap.init 中会调用 p.addLast() 方法，将 ChannelInitializer 插入到链表末端：

```
public final ChannelPipeline addLast(EventExecutorGroup group, String name,
ChannelHandler handler) {
        final AbstractChannelHandlerContext newCtx;
        synchronized (this) {
            checkMultiplicity(handler);
            newCtx = newContext(group, filterName(name, handler), handler);
            addLast0(newCtx);
            // 略去 N 句代码
        return this;
    }
private AbstractChannelHandlerContext newContext(EventExecutorGroup group, String
name, ChannelHandler handler) {
        return new DefaultChannelHandlerContext(this, childExecutor(group), name,
handler);
        }
```

addLast 有很多重载的方法，我们关注这个比较重要的方法就可以了.

上面的 addLast 方法中，首先检查这个 ChannelHandler 的名字是否是重复的，如果不重复的话，则调用 newContext 方法为这个 Handler 创建一个对应的 DefaultChannelHandlerContext 实例，并与之关联起来(Context 中有一个 handler 属性保存着对应的 Handler 实例).

为了添加一个 handler 到 pipeline 中，必须把此 handler 包装成 ChannelHandlerContext. 因此在上面的代码中我们可以看到新实例化了一个 newCtx 对象，并将 handler 作为参数传递到构造方法中. 那么我们来看一下实例化的 DefaultChannelHandlerContext 到底有什么玄机吧.

首先看它的构造器：

```
DefaultChannelHandlerContext(
            DefaultChannelPipeline pipeline, EventExecutor executor, String name,
ChannelHandler handler) {
        super(pipeline, executor, name, isInbound(handler), isOutbound(handler));
        if (handler == null) {
            throw new NullPointerException("handler");
        }
        this.handler = handler;
    }
```

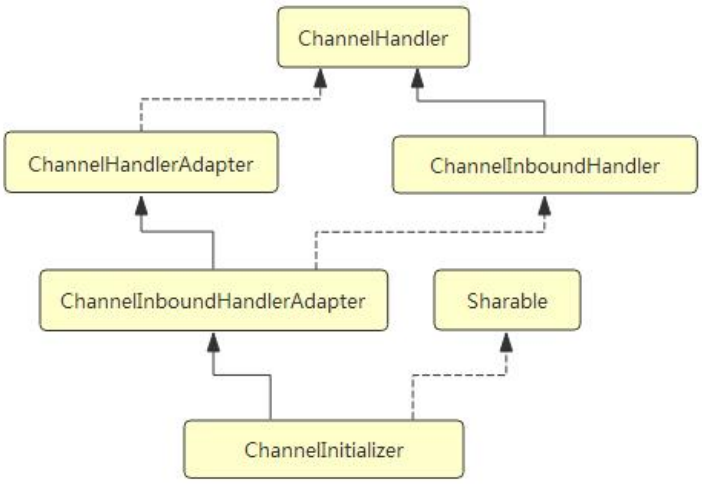
DefaultChannelHandlerContext 的构造器中，调用了两个很有意思的方法：isInbound 与 isOutbound，这两个方法是做什么的呢?

```
private static boolean isInbound(ChannelHandler handler) {
        return handler instanceof ChannelInboundHandler;
    }

    private static boolean isOutbound(ChannelHandler handler) {
        return handler instanceof ChannelOutboundHandler;
    }
```

从源码中可以看到，当一个 handler 实现了 ChannelInboundHandler 接口，则 isInbound 返回真；相似地，当一个 handler 实现了 ChannelOutboundHandler 接口，则 isOutbound 就返回真.

而这两个 boolean 变量会传递到父类 AbstractChannelHandlerContext 中，并初始化父类的两个字段：inbound 与 outbound.

那么这里的 ChannelInitializer 所对应的 DefaultChannelHandlerContext 的 inbound 与 inbound 字段分别是什么呢？那就看一下 ChannelInitializer 到底实现了哪个接口不就行了？如下是 ChannelInitializer 的类层次结构图：



可以清楚地看到，ChannelInitializer 仅仅实现了 ChannelInboundHandler 接口，因此这里实例化的 DefaultChannelHandlerContext 的 inbound = true，outbound = false.

不就是 inbound 和 outbound 两个字段嘛，为什么需要这么大费周章地分析一番？其实这两个字段关系到 pipeline 的事件的流向与分类，因此是十分关键的，不过我在这里先卖个关子，后面我们再来详细分析这两个字段所起的作用. 在这里，我暂且只需要记住，ChannelInitializer 所对应的 DefaultChannelHandlerContext 的 inbound = true，outbound = false 即可.

当创建好 Context 后，就将这个 Context 插入到 Pipeline 的双向链表中：

```
    private void addLast0(AbstractChannelHandlerContext newCtx) {
        AbstractChannelHandlerContext prev = tail.prev;
        newCtx.prev = prev;
        newCtx.next = tail;
        prev.next = newCtx;
        tail.prev = newCtx;
    }
```

2.1.4 自定义 ChannelHandler 的添加过程

前面我们已经分析了一个 ChannelInitializer 如何插入到 Pipeline 中的，接下来就来探讨一下 ChannelInitializer 在哪里被调用，ChannelInitializer 的作用，以及我们自定义的 ChannelHandler 是如何插入到 Pipeline 中的.

现在我们再简单地复习一下 Channel 的注册过程：

1、首先在 AbstractBootstrap.initAndRegister 中，通过 group().register(channel)，调用 MultithreadEventLoopGroup.register 方法

2、在 MultithreadEventLoopGroup.register 中，通过 next() 获取一个可用的 SingleThreadEventLoop，然后调用它的 register

3、在 SingleThreadEventLoop.register 中，通过 channel.unsafe().register(this, promise) 来获取 channel 的 unsafe() 底层操作对象，然后调用它的 register.

4、在 AbstractUnsafe.register 方法中，调用 register0 方法注册 Channel

5、在 AbstractUnsafe.register0 中，调用 AbstractNioChannel#doRegister 方法

6、AbstractNioChannel.doRegister 方法通过 javaChannel().register(eventLoop().selector, 0, this) 将 Channel 对应的 Java NIO SockerChannel 注册到一个 eventLoop 的 Selector 中，并且将当前 Channel 作为 attachment.

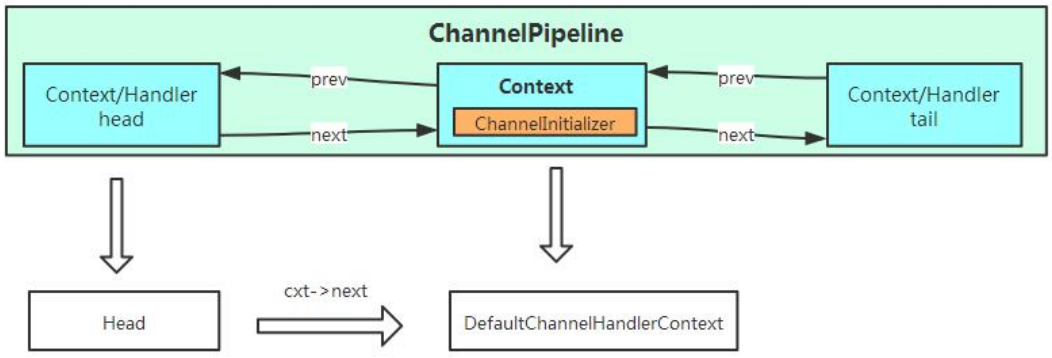
而我们自定义 ChannelHandler 的添加过程，发生在 AbstractUnsafe.register0 中，在这个方法中调用了 pipeline.fireChannelRegistered() 方法，其实现如下：

```
public final ChannelPipeline fireChannelRegistered() {
        AbstractChannelHandlerContext.invokeChannelRegistered(head);
        return this;
    }
```

再看 AbstractChannelHandlerContext.invokeChannelRegistered 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();
                }
            });
        }
    }
```

很显然，这个代码会从 head 开始遍历 Pipeline 的双向链表，然后找到第一个属性 inbound 为 true 的 ChannelHandlerContext 实例. 想起来了没？我们在前面分析 ChannelInitializer 时，花了大量的笔墨来分析了 inbound 和 outbound 属性，你看现在这里就用上了. 回想一下，ChannelInitializer 实现了 ChannelInboudHandler，因此它所对应的 ChannelHandlerContext 的 inbound 属性就是 true，因此这里返回就是 ChannelInitializer 实例所对应的 ChannelHandlerContext. 即：



当获取到 inbound 的 Context 后，就调用它的 invokeChannelRegistered 方法：

```
private void invokeChannelRegistered() {
        if (invokeHandler()) {
            try {
                ((ChannelInboundHandler) handler()).channelRegistered(this);
            } catch (Throwable t) {
                notifyHandlerException(t);
            }
        } else {
            fireChannelRegistered();
        }
    }
```

我们已经强调过了，每个 ChannelHandler 都与一个 ChannelHandlerContext 关联，我们可以通过 ChannelHandlerContext 获取到对应的 ChannelHandler. 因此很显然了，这里 handler() 返回的，其实就是一开始我们实例化的 ChannelInitializer 对象，并接着调用了 ChannelInitializer.channelRegistered 方法. 看到这里，是否会觉得有点眼熟呢？ChannelInitializer.channelRegistered 这个方法我们在一开始的时候已经大量地接触了，但是我们并没有深入地分析这个方法的调用过程，那么在这里读者朋友应该对它的调用有了更加深入的了解了吧.

那么这个方法中又有什么玄机呢？继续看代码：

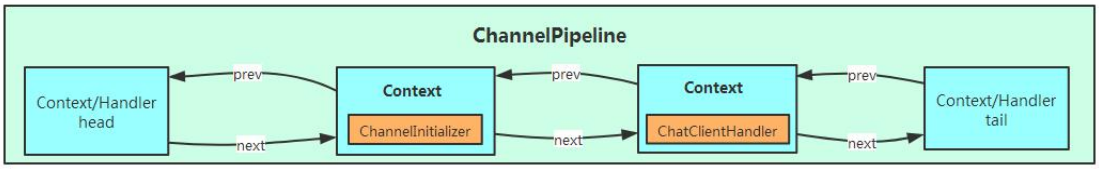
```
public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        if (initChannel(ctx)) {
            ctx.pipeline().fireChannelRegistered();
        } else {
            ctx.fireChannelRegistered();
        }
    }
private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
        if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) { // Guard against
            try {
                initChannel((C) ctx.channel());
            } catch (Throwable cause) {
                exceptionCaught(ctx, cause);
```

```
            } finally {
                remove(ctx);
            }
            return true;
        }
        return false;
        }
```

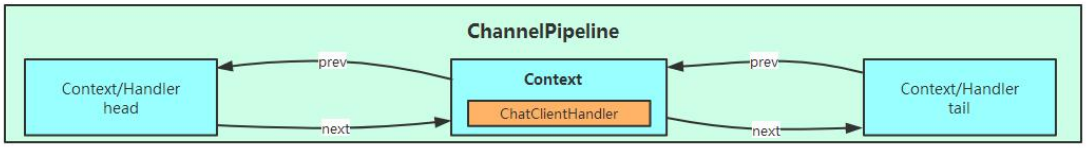
initChannel((C) ctx.channel()); 这个方法我们很熟悉了吧，它就是我们在初始化 Bootstrap 时，调用 handler 方法传入的匿名内部类所实现的方法：

```
.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    pipeline.addLast(new ChatClientHandler(nickName));
                }
});
```

因此当调用了这个方法后，我们自定义的 ChannelHandler 就插入到 Pipeline 了，此时的 Pipeline 如下图所示：



当添加了自定义的 ChannelHandler 后，会删除 ChannelInitializer 这个 ChannelHandler，即 "ctx.pipeline().remove(this)"，因此最后的 Pipeline 如下：



好了，到了这里，我们的 自定义 ChannelHandler 的添加过程 也分析的查不多了.

2.1.5 ChannelHandler 的名字

我们注意到，pipeline.addXXX 都有一个重载的方法，例如 addLast，它有一个重载的版本是：

```
ChannelPipeline addLast(String name, ChannelHandler handler);
```

第一个参数指定了所添加的 handler 的名字(更准确地说是 ChannelHandlerContext 的名字，不过我们通常是以 handler 作为叙述的对象，因此说成 handler 的名字便于理解). 那么 handler 的名字有什么用呢? 如果我们不设置 name，那么 handler 会有怎样的名字?

为了解答这些疑惑，老规矩，依然是从源码中找到答案.

我们还是以 addLast 方法为例：

```
public final ChannelPipeline addLast(String name, ChannelHandler handler) {
        return addLast(null, name, handler);
}
```

这个方法会调用重载的 addLast 方法：

```
public final ChannelPipeline addLast(EventExecutorGroup group, String name,
ChannelHandler handler) {
        final AbstractChannelHandlerContext newCtx;
        synchronized (this) {
            checkMultiplicity(handler);

            newCtx = newContext(group, filterName(name, handler), handler);

            addLast0(newCtx);
            // 略去 N 句代码
        }
        return this;
    }
```

第一个参数被设置为 null，我们不关心它. 第二参数就是这个 handler 的名字. 看代码可知，在添加一个 handler 之前，需要调用 ***checkMultiplicity*** 方法来确定此 handler 的名字是否和已添加的 handler 的名字重复.

2.1.6 自动生成 handler 的名字

如果我们调用的是如下的 addLast 方法

```
 ChannelPipeline addLast(ChannelHandler... handlers);
```

那么 Netty 会调用 generateName 为我们的 handler 自动生成一个名字:

```
private String filterName(String name, ChannelHandler handler) {
        if (name == null) {
            return generateName(handler);
        }
        checkDuplicateName(name);
        return name;
    }
private String generateName(ChannelHandler handler) {
        Map<Class<?>, String> cache = nameCaches.get();
        Class<?> handlerType = handler.getClass();
        String name = cache.get(handlerType);
        if (name == null) {
            name = generateName0(handlerType);
            cache.put(handlerType, name);
        }
        // 此处省略 N 行代码
        return name;
        }
```

而 generateName 会接着调用 generateName0 来实际产生一个 handler 的名字:

```
 private static String generateName0(Class<?> handlerType) {
        return StringUtil.simpleClassName(handlerType) + "#0";
 }
```

自动生成的名字的规则很简单，就是 handler 的简单类名加上 "#0"，因此我们的

ChatClientHandler 的名字就是 "ChatClientHandler#0"

2.1.7 关于 Pipeline 的事件传输机制

前面章节中，我们知道 AbstractChannelHandlerContext 中有 inbound 和 outbound 两个 boolean 变量，分别用于标识 Context 所对应的 handler 的类型，即：

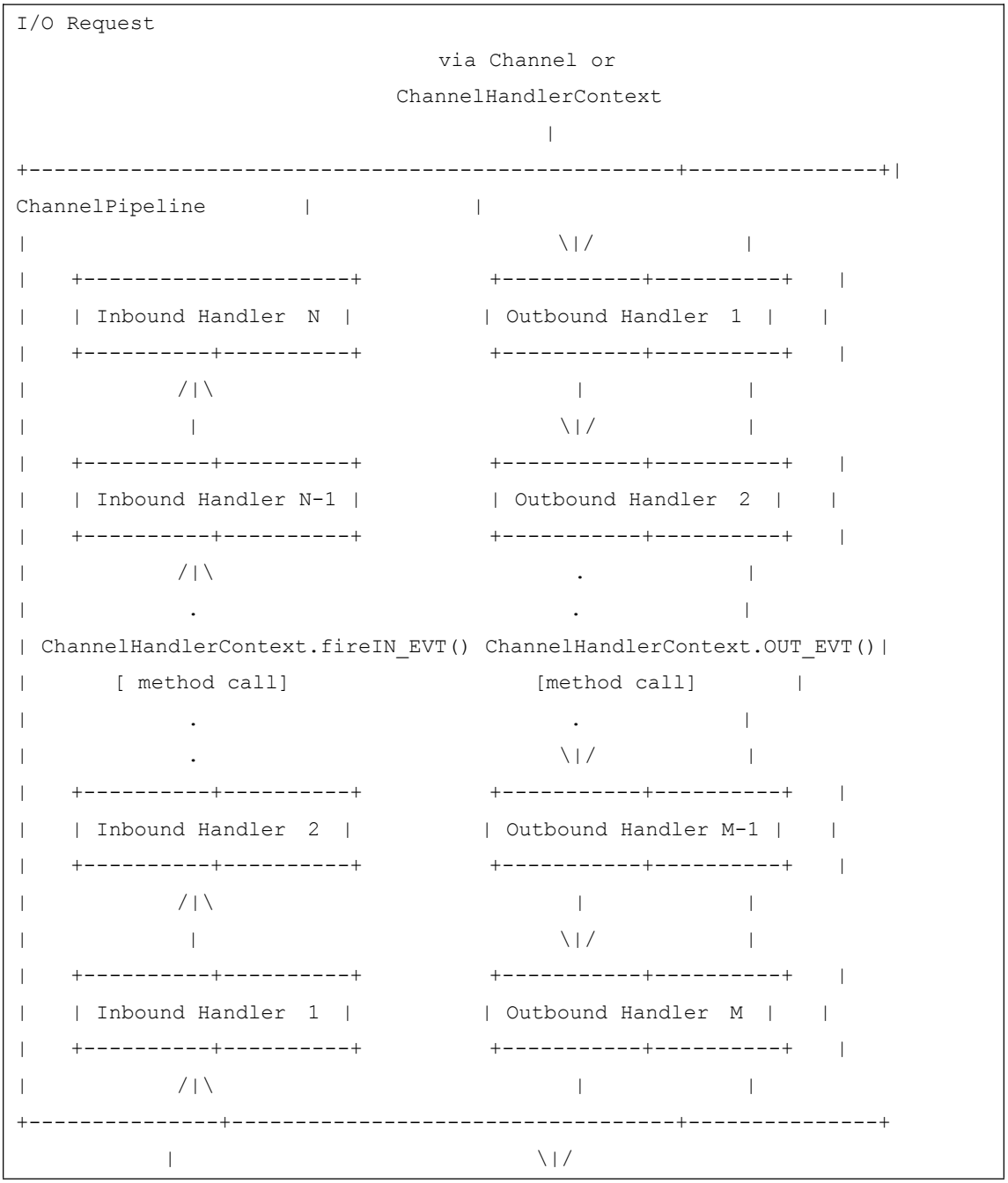
1、inbound 为真时，表示对应的 ChannelHandler 实现了 ChannelInboundHandler 方法.

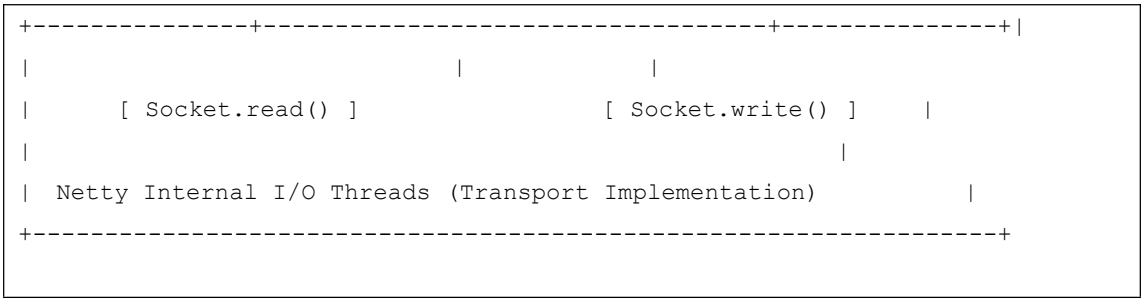
2、outbound 为真时，表示对应的 ChannelHandler 实现了 ChannelOutboundHandler 方法.

这里大家肯定很疑惑了吧：那究竟这两个字段有什么作用呢？其实这还要从 ChannelPipeline 的传输的事件类型说起.

Netty 的事件可以分为 Inbound 和 Outbound 事件.

如下是从 Netty 官网上拷贝的一个图示：





从上图可以看出，inbound 事件和 outbound 事件的流向是不一样的，inbound 事件的流行是从下至上，而 outbound 刚好相反，是从上到下. 并且 inbound 的传递方式是通过调用相应的 ChannelHandlerContext.fireIN_EVT() 方法，而 outbound 方法的的传递方式是通过调用 ChannelHandlerContext.OUT_EVT() 方法. 例如 ChannelHandlerContext.fireChannelRegistered() 调用会发送一个 ChannelRegistered 的 inbound 给下一个 ChannelHandlerContext，而 ChannelHandlerContext.bind 调用会发送一个 bind 的 outbound 事件给 下一个 ChannelHandlerContext.

Inbound 事件传播方法有：

```
public interface ChannelInboundHandler extends ChannelHandler {
    void channelRegistered(ChannelHandlerContext ctx) throws Exception;
    void channelUnregistered(ChannelHandlerContext ctx) throws Exception;
    void channelActive(ChannelHandlerContext ctx) throws Exception;
    void channelInactive(ChannelHandlerContext ctx) throws Exception;
    void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;
    void channelReadComplete(ChannelHandlerContext ctx) throws Exception;
    void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception;
    void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;
    void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
}
```

Outbound 事件传输方法有：

```
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 flush(ChannelHandlerContext ctx) throws Exception;
}
```

注意，如果我们捕获了一个事件，并且想让这个事件继续传递下去，那么需要调用 Context 相应的传播方法.

例如：

```
public class MyInboundHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        System.out.println("连接成功");
        ctx.fireChannelActive();
    }

}

public class MyOutboundHandler extends ChannelOutboundHandlerAdapter {

    @Override
    public void close(ChannelHandlerContext ctx, ChannelPromise promise) throws
Exception {
        System.out.println("客户端关闭");
        ctx.close(promise);
    }

}
```

上面的例子中，MyInboundHandler 收到了一个 channelActive 事件，它在处理后，如果希望将事件继续传播下去，那么需要接着调用 ctx.fireChannelActive().

2.1.8 Outbound 的操作(outbound operations of a channel)

Outbound 事件都是请求事件(request event)，即请求某件事情的发生，然后通过 Outbound 事件进行通知.
Outbound 事件的传播方向是 tail -> customContext -> head.

我们接下来以 connect 事件为例，分析一下 Outbound 事件的传播机制.
首先，当用户调用了 Bootstrap.connect 方法时，就会触发一个 **Connect 请求事件**，此调用会触发如下调用链:

Bootstrap.connect->Bootstrap.doResolveAndConnect->Bootstrap.doResolveAndConnect0->
->Bootstrap.doConnect->AbstractChannel.connect
继续跟踪的话，我们就发现，AbstractChannel.connect 其实由调用了 DefaultChannelPipeline.connect 方法:

```
public ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {
        return pipeline.connect(remoteAddress, promise);
    }
```

而 pipeline.connect 的实现如下

```
public final ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise)
{
```

```
        return tail.connect(remoteAddress, promise);
    }
```

可以看到，当 outbound 事件(这里是 connect 事件)传递到 Pipeline 后，它其实是以 tail 为起点开始传播的.

而 tail.connect 其实调用的是 AbstractChannelHandlerContext.connect 方法:

```
public ChannelFuture connect(
            final SocketAddress remoteAddress,
            final SocketAddress localAddress, final ChannelPromise promise) {
        //此处省略 N 句
        final AbstractChannelHandlerContext next = findContextOutbound();
        EventExecutor executor = next.executor();
            next.invokeConnect(remoteAddress, localAddress, promise);
        //此处省略 N 句
        return promise;
    }
```

findContextOutbound() 顾名思义，它的作用是以当前 Context 为起点，向 Pipeline 中的 Context 双向链表的前端寻找第一个 outbound 属性为真的 Context(即关联着 ChannelOutboundHandler 的 Context)，然后返回.

它的实现如下:

```
private AbstractChannelHandlerContext findContextOutbound() {
        AbstractChannelHandlerContext ctx = this;
        do {
            ctx = ctx.prev;
        } while (!ctx.outbound);
        return ctx;
}
```

当我们找到了一个 outbound 的 Context 后，就调用它的 invokeConnect 方法，这个方法中会调用 Context 所关联着的 ChannelHandler 的 connect 方法:

```
private void invokeConnect(SocketAddress remoteAddress, SocketAddress localAddress,
ChannelPromise promise) {
        if (invokeHandler()) {
            try {
                ((ChannelOutboundHandler) handler()).connect(this, remoteAddress,
localAddress, promise);
            } catch (Throwable t) {
                notifyOutboundHandlerException(t, promise);
            }
        } else {
            connect(remoteAddress, localAddress, promise);
        }
    }
```

如果用户没有重写 ChannelHandler 的 connect 方法，那么会调用 ChannelOutboundHandlerAdapter 所实现的方法:

```
    public void connect(ChannelHandlerContext ctx, SocketAddress remoteAddress,
```

```
            SocketAddress localAddress, ChannelPromise promise) throws Exception {
        ctx.connect(remoteAddress, localAddress, promise);
    }
```

我们看到，ChannelOutboundHandlerAdapter.connect 仅仅调用了 ctx.connect，而这个调用又回到了：

Context.connect -> Connect.findContextOutbound -> next.invokeConnect -> handler.connect -> Context.connect

这样的循环中，直到 connect 事件传递到 DefaultChannelPipeline 的双向链表的头节点，即 head 中. 为什么会传递到 head 中呢？回想一下，head 实现了 ChannelOutboundHandler，因此它的 outbound 属性是 true.

因为 head 本身既是一个 ChannelHandlerContext，又实现了 ChannelOutboundHandler 接口，因此当 connect 消息传递到 head 后，会将消息转递到对应的 ChannelHandler 中处理，而恰好，head 的 handler() 返回的就是 head 本身：

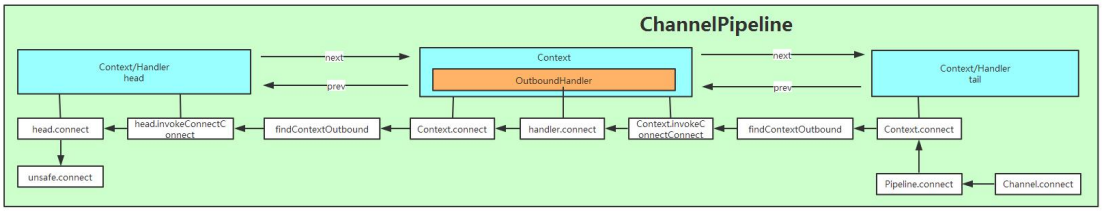
```
public ChannelHandler handler() {
            return this;
}
```

因此最终 connect 事件是在 head 中处理的. head 的 connect 事件处理方法如下：

```
public void connect(
                ChannelHandlerContext ctx,
                SocketAddress remoteAddress, SocketAddress localAddress,
                ChannelPromise promise) throws Exception {
            unsafe.connect(remoteAddress, localAddress, promise);
}
```

到这里，整个 Connect 请求事件就结束了.

下面以一幅图来描述一个整个 Connect 请求事件的处理过程：



我们仅仅以 Connect 请求事件为例，分析了 Outbound 事件的传播过程，但是其实所有的 outbound 的事件传播都遵循着一样的传播规律,同学们可以试着分析一下其他的 outbound 事件，体会一下它们的传播过程.

2.1.9 Inbound 事件

Inbound 事件和 Outbound 事件的处理过程有点像.

Inbound 事件是一个通知事件，即某件事已经发生了，然后通过 Inbound 事件进行通知. Inbound 通常发生在 Channel 的状态的改变或 IO 事件就绪.

Inbound 的特点是它传播方向是 head -> customContext -> tail.

既然上面我们分析了 Connect 这个 Outbound 事件，那么接着分析 Connect 事件后会发生什么 Inbound 事件，并最终找到 Outbound 和 Inbound 事件之间的联系.

当 Connect 这个 Outbound 传播到 unsafe 后，其实是在 AbstractNioUnsafe.connect 方法

中进行处理的:

```
public final void connect(
                final SocketAddress remoteAddress,
                final SocketAddress localAddress, final ChannelPromise promise) {

                if (doConnect(remoteAddress, localAddress)) {
                    fulfillConnectPromise(promise, wasActive);
                } else {
                    ...
                }
}
```

在 AbstractNioUnsafe.connect 中, 首先调用 doConnect 方法进行实际上的 Socket 连接, 当连接上后, 会调用 fulfillConnectPromise 方法:

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {
            if (!wasActive && active) {
                pipeline().fireChannelActive();
            }
}
```

我们看到, 在 fulfillConnectPromise 中, 会通过调用 pipeline().fireChannelActive() 将通道激活的消息(即 Socket 连接成功)发送出去.

而这里, 当调用 pipeline.fireXXX 后, 就是 Inbound 事件的起点.

因此当调用了 pipeline().fireChannelActive() 后, 就产生了一个 ChannelActive Inbound 事件, 我们就从这里开始看看这个 Inbound 事件是怎么传播的吧.

```
public final ChannelPipeline fireChannelActive() {
        AbstractChannelHandlerContext.invokeChannelActive(head);
        return this;
    }
```

哈哈, 果然, 在 fireChannelActive 方法中, 调用的是 head.invokeChannelActive, 因此可以证明了, Inbound 事件在 Pipeline 中传输的起点是 head.那么, 在 head.invokeChannelActive() 中又做了什么呢?

```
static void invokeChannelActive(final AbstractChannelHandlerContext next) {
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            next.invokeChannelActive();
        } else {
            executor.execute(new Runnable() {
                @Override
                public void run() {
                    next.invokeChannelActive();
                }
            });
        }
}
```

上面的代码应该很熟悉了吧. 回想一下在 Outbound 事件(例如 Connect 事件)的传输过程中时, 我们也有类似的操作:

1、首先调用 findContextInbound, 从 Pipeline 的双向链表中中找到第一个属性 inbound 为真的 Context, 然后返回

2、调用这个 Context 的 invokeChannelActive

invokeChannelActive 方法如下:

```
private void invokeChannelActive() {
        if (invokeHandler()) {
            try {
                ((ChannelInboundHandler) handler()).channelActive(this);
            } catch (Throwable t) {
                notifyHandlerException(t);
            }
        } else {
            fireChannelActive();
        }
    }
```

这个方法和 Outbound 的对应方法(例如 invokeConnect) 如出一辙. 同 Outbound 一样, 如果用户没有重写 channelActive 方法, 那么会调用 ChannelInboundHandlerAdapter 的 channelActive 方法:

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelActive();
}
```

同样地, 在 ChannelInboundHandlerAdapter.channelActive 中, 仅仅调用了 ctx.fireChannelActive 方法, 因此就会有如下循环:

Context.fireChannelActive -> Connect.findContextInbound -> nextContext.invokeChannelActive -> nextHandler.channelActive -> nextContext.fireChannelActive

这样的循环中. 同理, tail 本身 既实现了 ChannelInboundHandler 接口, 又实现了 ChannelHandlerContext 接口, 因此当 channelActive 消息传递到 tail 后, 会将消息转递到对应的 ChannelHandler 中处理, 而恰好, tail 的 handler() 返回的就是 tail 本身:

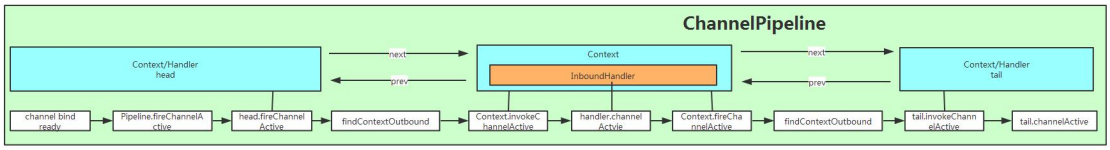
```
public ChannelHandler handler() {
            return this;
}
```

因此 channelActive Inbound 事件最终是在 tail 中处理的, 我们看一下它的处理方法:

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {}
```

TailContext.channelActive 方法是空的. 如果读者自行查看 TailContext 的 Inbound 处理方法时, 会发现, 它们的实现都是空的. 可见, 如果是 Inbound, 当用户没有实现自定义的处理器时, 那么默认是不处理的.

用一幅图来总结一下 Inbound 的传输过程吧:



2.1.10 总结

对于 Outbound 事件：

1、Outbound 事件是请求事件(由 Connect 发起一个请求，并最终由 unsafe 处理这个请求)

2、Outbound 事件的发起者是 Channel

3、Outbound 事件的处理者是 unsafe

4、Outbound 事件在 Pipeline 中的传输方向是 tail -> head.

5、在 ChannelHandler 中处理事件时，如果这个 Handler 不是最后一个 Hnalder，则需要调用 ctx.xxx（例如 ctx.connect）将此事件继续传播下去. 如果不这样做，那么此事件的传播会提前终止.

6、Outbound 事件流：Context.OUT_EVT -> Connect.findContextOutbound -> nextContext.invokeOUT_EVT -> nextHandler.OUT_EVT -> nextContext.OUT_EVT

对于 Inbound 事件：

1、Inbound 事件是通知事件，当某件事情已经就绪后，通知上层.

2、Inbound 事件发起者是 unsafe

3、Inbound 事件的处理者是 Channel，如果用户没有实现自定义的处理方法，那么 Inbound 事件默认的处理者是 TailContext，并且其处理方法是空实现.

4、Inbound 事件在 Pipeline 中传输方向是 head -> tail

5、在 ChannelHandler 中处理事件时，如果这个 Handler 不是最后一个 Hnalder，则需要调用 ctx.fireIN_EVT（例如 ctx.fireChannelActive）将此事件继续传播下去. 如果不这样做，那么此事件的传播会提前终止.

6、Outbound 事件流：Context.fireIN_EVT -> Connect.findContextInbound -> nextContext.invokeIN_EVT -> nextHandler.IN_EVT -> nextContext.fireIN_EVT

outbound 和 inbound 事件十分的像，并且 Context 与 Handler 直接的调用关系是否容易混淆，因此我们在阅读这里的源码时，需要特别的注意.

3. 大名鼎鼎的 EventLoop

前面的章节中我们已经知道了，一个 Netty 程序启动时，至少要指定一个 EventLoopGroup(如果使用到的是 NIO，那么通常是 NioEventLoopGroup)，那么这个 NioEventLoopGroup 在 Netty 中到底扮演着什么角色呢？我们知道，Netty 是 Reactor 模型的一个实现，那么首先从 Reactor 的线程模型开始吧.

3.1.1 关于 Reactor 的线程模型

首先我们来看一下 Reactor 的线程模型.

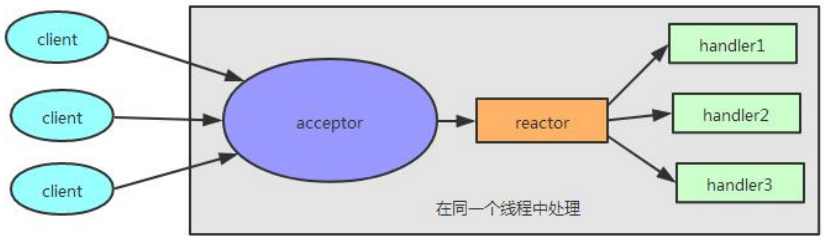
Reactor 的线程模型有三种：

1、单线程模型

2、多线程模型

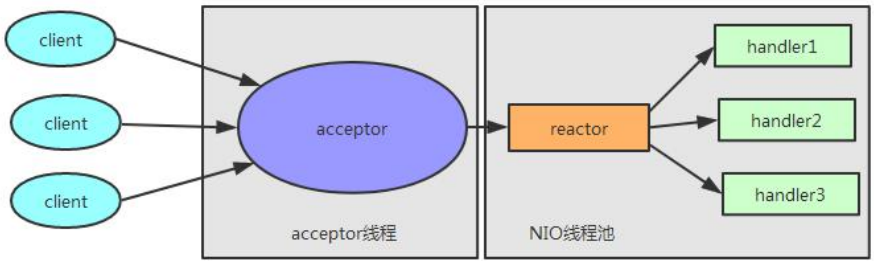
3、主从多线程模型

首先来看一下 单线程模型：



所谓单线程，即 acceptor 处理和 handler 处理都在一个线程中处理. 这个模型的坏处显而易见：当其中某个 handler 阻塞时，会导致其他所有的 client 的 handler 都得不到执行，并且更严重的是，handler 的阻塞也会导致整个服务不能接收新的 client 请求(因为 acceptor 也被阻塞了). 因为有这么多的缺陷，因此单线程 Reactor 模型用的比较少.

那么什么是 多线程模型 呢? Reactor 的多线程模型与单线程模型的区别就是 acceptor 是一个单独的线程处理，并且有一组特定的 NIO 线程来负责各个客户端连接的 IO 操作. Reactor 多线程模型如下：



Reactor 多线程模型 有如下特点：

1、有专门一个线程，即 Acceptor 线程用于监听客户端的 TCP 连接请求.

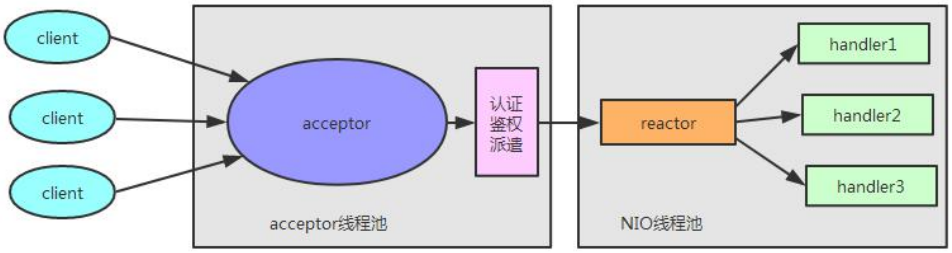
2、客户端连接的 IO 操作都是由一个特定的 NIO 线程池负责. 每个客户端连接都与一个特定的 NIO 线程绑定，因此在这个客户端连接中的所有 IO 操作都是在同一个线程中完成的.

3、客户端连接有很多，但是 NIO 线程数是比较少的，因此一个 NIO 线程可以同时绑定到多个客户端连接中.

接下来我们再来看一下 Reactor 的主从多线程模型.

一般情况下，Reactor 的多线程模式已经可以很好的工作了，但是我们考虑一下如下情况：如果我们的服务器需要同时处理大量的客户端连接请求或我们需要在客户端连接时，进行一些权限的检查，那么单线程的 Acceptor 很有可能就处理不过来，造成了大量的客户端不能连接到服务器.

Reactor 的主从多线程模型就是在这样的情况下提出来的，它的特点是：服务器端接收客户端的连接请求不再是一个线程，而是由一个独立的线程池组成. 它的线程模型如下：



可以看到，Reactor 的主从多线程模型和 Reactor 多线程模型很类似，只不过 Reactor 的主从多线程模型的 acceptor 使用了线程池来处理大量的客户端请求.

3.1.2 NioEventLoopGroup 与 Reactor 线程模型的对应

我们介绍了三种 Reactor 的线程模型，那么它们和 NioEventLoopGroup 又有什么关系呢？其实，不同的设置 NioEventLoopGroup 的方式就对应了不同的 Reactor 的线程模型.

单线程模型

来看一下下面的例子：

```
    EventLoopGroup bossGroup = new NioEventLoopGroup(1);
    ServerBootstrap server = new ServerBootstrap();
    server.group(bossGroup);
```

注意，我们实例化了一个 NioEventLoopGroup，然后接着我们调用 server.group(bossGroup) 设置了服务器端的 EventLoopGroup. 有人可能会有疑惑：我记得在启动服务器端的 Netty 程序时，是需要设置 bossGroup 和 workerGroup 的，为什么这里就只有一个 bossGroup?

其实很简单，ServerBootstrap 重写了 group 方法：

```
public ServerBootstrap group(EventLoopGroup group) {
       return group(group, group);
}
```

因此当传入一个 group 时，那么 bossGroup 和 workerGroup 就是同一个 NioEventLoopGroup 了.

这时候呢，因为 bossGroup 和 workerGroup 就是同一个 NioEventLoopGroup，并且这个 NioEventLoopGroup 只有一个线程，这样就会导致 Netty 中的 acceptor 和后续的所有客户端连接的 IO 操作都是在一个线程中处理的. 那么对应到 Reactor 的线程模型中，我们这样设置 NioEventLoopGroup 时，就相当于 Reactor 单线程模型.

多线程模型

同理，再来看一下下面的例子：

```
    EventLoopGroup bossGroup = new NioEventLoopGroup(128);
    ServerBootstrap server = new ServerBootstrap();
    server.group(bossGroup);
```

将 bossGroup 的参数就设置为大于 1 的数,其实就是 Reactor 多线程模型.

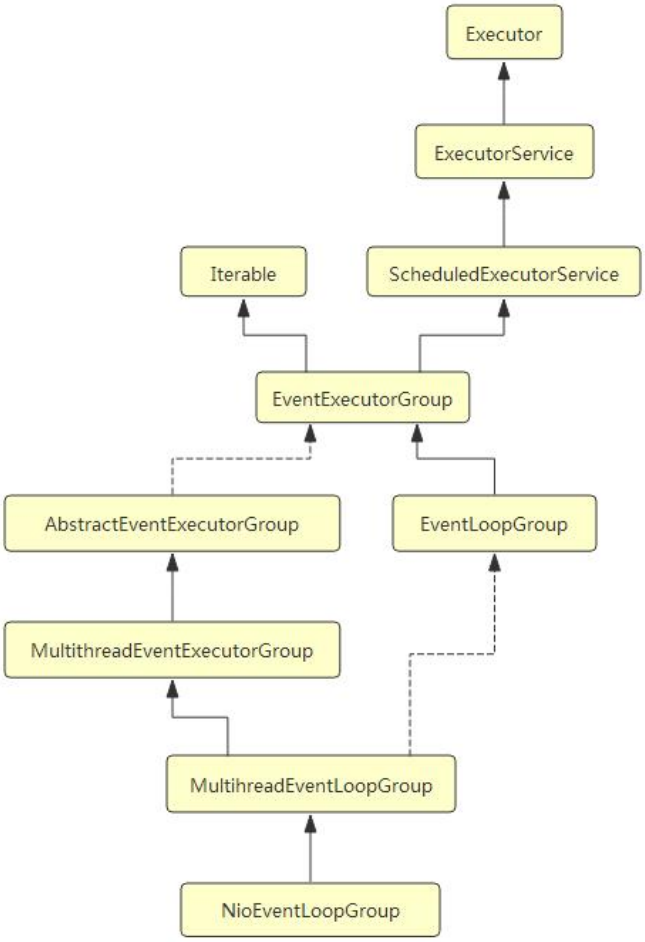
主从线程模型

相信同学们都想到了，实现主从线程模型的例子如下：

```
       EventLoopGroup bossGroup = new NioEventLoopGroup();
       EventLoopGroup workerGroup = new NioEventLoopGroup();
       ServerBootstrap b = new ServerBootstrap();
       b.group(bossGroup, workerGroup);
```

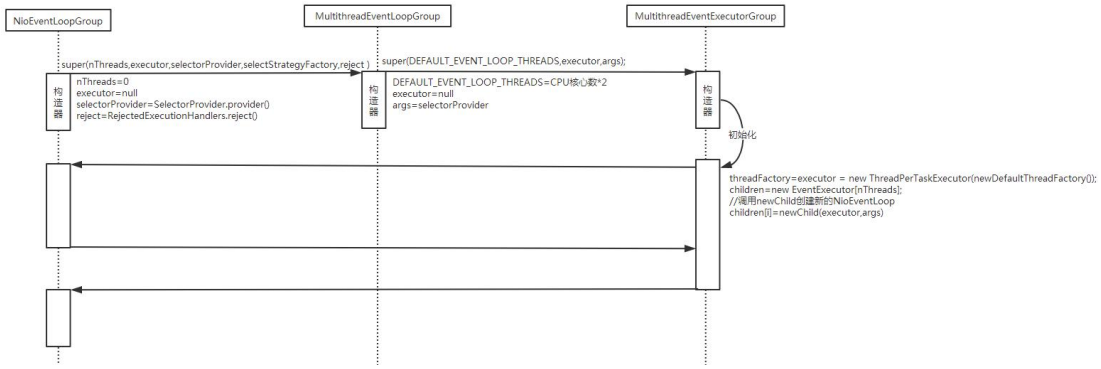
bossGroup 为主线程，而 workerGroup 中的线程是 CPU 核心数乘以 2，因此对应的到 Reactor 线程模型中，我们知道，这样设置的 NioEventLoopGroup 其实就是 Reactor 主从多线程模型.

3.1.3 NioEventLoopGroup 类层次结构



3.1.4 NioEventLoopGroup 实例化过程

在前面的章节中我们已经简单地介绍了一下 NioEventLoopGroup 的初始化过程，这里再回顾一下：



即：

1、EventLoopGroup(其实是 MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children 数组，其大小是 nThreads，这样就构成了一个线程池

2、如果我们在实例化 NioEventLoopGroup 时，如果指定线程池大小，则 nThreads 就是指定的值，反之是处理器核心数 * 2

3、MultithreadEventExecutorGroup 中会调用 newChild 抽象方法来初始化 children 数组

4、抽象方法 newChild 是在 NioEventLoopGroup 中实现的，它返回一个 NioEventLoop 实

例.

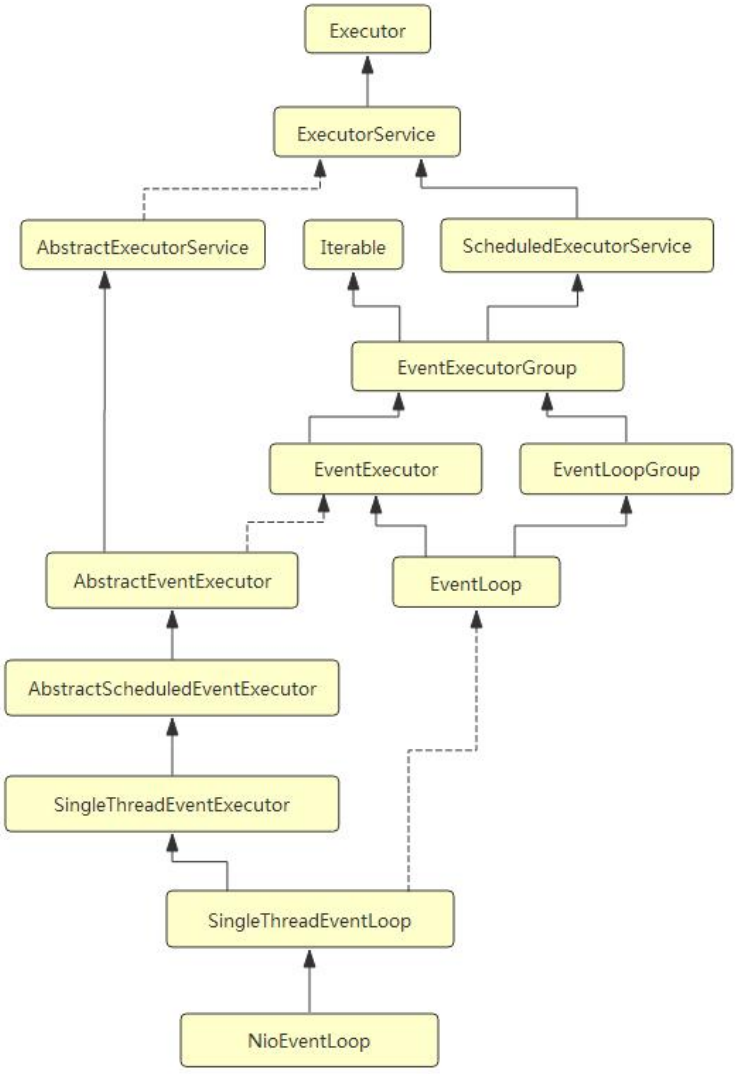
5、NioEventLoop 属性:

SelectorProvider provider 属性: NioEventLoopGroup 构造器中通过 SelectorProvider.provider() 获取一个 SelectorProvider

Selector selector 属性: NioEventLoop 构造器中通过调用通过 selector = provider.openSelector() 获取一个 selector 对象.

3.1.5 NioEventLoop 类层次结构

NioEventLoop 继承于 SingleThreadEventLoop, 而 SingleThreadEventLoop 又继承于 SingleThreadEventExecutor. SingleThreadEventExecutor 是 Netty 中对本地线程的抽象, 它内部有一个 Thread thread 属性, 存储了一个本地 Java 线程. 因此我们可以认为, 一个 NioEventLoop 其实和一个特定的线程绑定, 并且在其生命周期内, 绑定的线程都不会再改变.



NioEventLoop 的类层次结构图还是比较复杂的, 不过我们只需要关注几个重要的点即可. 首先 NioEventLoop 的继承链如下:

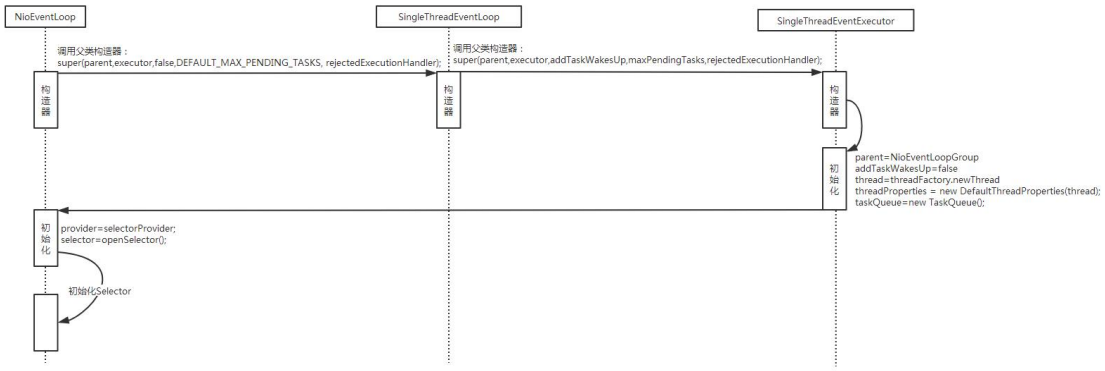
NioEventLoop -> SingleThreadEventLoop -> SingleThreadEventExecutor -> AbstractScheduledEventExecutor

在 AbstractScheduledEventExecutor 中, Netty 实现了 NioEventLoop 的 schedule 功能,

即我们可以通过调用一个 NioEventLoop 实例的 schedule 方法来运行一些定时任务. 而在 SingleThreadEventLoop 中，又实现了任务队列的功能，通过它，我们可以调用一个 NioEventLoop 实例的 execute 方法来向任务队列中添加一个 task，并由 NioEventLoop 进行调度执行.

通常来说，NioEventLoop 肩负着两种任务，第一个是作为 IO 线程，执行与 Channel 相关的 IO 操作，包括 调用 select 等待就绪的 IO 事件、读写数据与数据的处理等；而第二个任务是作为任务队列，执行 taskQueue 中的任务，例如用户调用 eventLoop.schedule 提交的定时任务也是这个线程执行的.

3.1.6 NioEventLoop 的实例化过程



从上图可以看到，SingleThreadEventExecutor 有一个名为 thread 的 Thread 类型字段，这个字段就代表了与 SingleThreadEventExecutor 关联的本地线程.

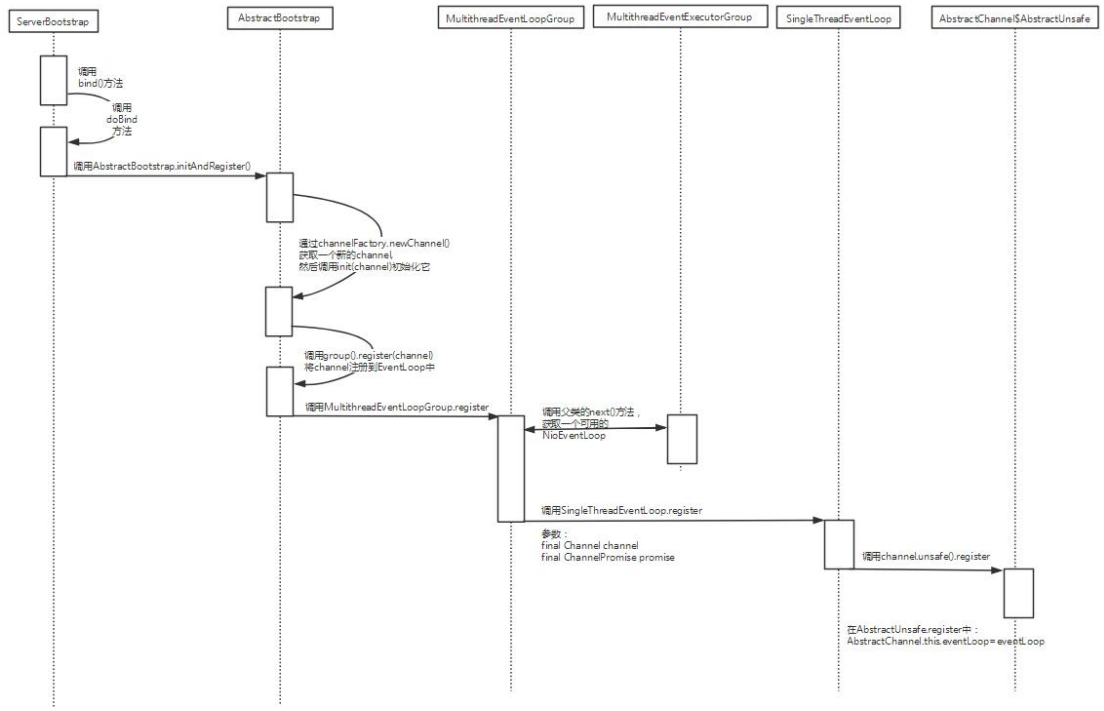
我们看看 thread 在哪里赋的值:

```
private void doStartThread() {
        assert thread == null;
        executor.execute(new Runnable() {
            @Override
            public void run() {
                thread = Thread.currentThread();
                boolean success = false;
                updateLastExecutionTime();
                try {
                    SingleThreadEventExecutor.this.run();
                    success = true;
                } catch (Throwable t) {
                    logger.warn("Unexpected exception from an event executor: ", t);
                } finally {
                    // 此处省略清理代码                }
            }
        });
    }
```

SingleThreadEventExecutor 启动时会调用 doStartThread 方法,然后执行 executor.execute 方法，将当前线程赋值给 thread. 在这个线程中所做的事情主要就是调用 SingleThreadEventExecutor.this.run() 方法，而因为 NioEventLoop 实现了这个方法，因此根据多态性，其实调用的是 NioEventLoop.run() 方法.

3.1.7 EventLoop 与 Channel 的关联

Netty 中，每个 Channel 都有且仅有一个 EventLoop 与之关联，它们的关联过程如下：



从上图中我们可以看到，当调用了 AbstractChannel$AbstractUnsafe.register 后，就完成了 Channel 和 EventLoop 的关联. register 实现如下：

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
          // 删除条件检查

          AbstractChannel.this.eventLoop = eventLoop;

          if (eventLoop.inEventLoop()) {
              register0(promise);
          } else {
              try {
                  eventLoop.execute(new Runnable() {
                      @Override
                      public void run() {
                          register0(promise);
                      }
                  });
              } catch (Throwable t) {
                  // 删除 catch 块内容
              }
          }
      }
```

在 AbstractChannel$AbstractUnsafe.register 中，会将一个 EventLoop 赋值给

AbstractChannel 内部的 eventLoop 字段，到这里就完成了 EventLoop 与 Channel 的关联过程.

3.1.7 EventLoop 的启动

在前面我们已经知道了，NioEventLoop 本身就是一个 SingleThreadEventExecutor，因此 NioEventLoop 的启动，其实就是 NioEventLoop 所绑定的本地 Java 线程的启动.
依照这个思想，我们只要找到在哪里调用了 SingleThreadEventExecutor 的 thread 字段的 start() 方法就可以知道是在哪里启动的这个线程了.
从代码中搜索，thread.start() 被封装到 SingleThreadEventExecutor.startThread() 方法中了:

```
 private void startThread() {
        if (STATE_UPDATER.get(this) == ST_NOT_STARTED) {
            if (STATE_UPDATER.compareAndSet(this, ST_NOT_STARTED, ST_STARTED)) {
                doStartThread();
            }
        }
    }
```

STATE_UPDATER 是 SingleThreadEventExecutor 内部维护的一个属性，它的作用是标识当前的 thread 的状态. 在初始的时候，STATE_UPDATER == ST_NOT_STARTED，因此第一次调用 startThread() 方法时，就会进入到 if 语句内，进而调用到 thread.start().
而这个关键的 startThread() 方法又是在哪里调用的呢? 经过方法调用关系搜索，我们发现，startThread 是在 SingleThreadEventExecutor.execute 方法中调用的:

```
public void execute(Runnable task) {
        if (task == null) {
            throw new NullPointerException("task");
        }

        boolean inEventLoop = inEventLoop();
        if (inEventLoop) {
            addTask(task);
        } else {
            startThread();  // 调用 startThread 方法、启动 EventLoop 线程
            addTask(task);
            if (isShutdown() && removeTask(task)) {
                reject();
            }
        }

        if (!addTaskWakesUp && wakesUpForTask(task)) {
            wakeup(inEventLoop);
        }
    }
```

既然如此，那现在我们的工作就变为了寻找 在哪里第一次调用了 SingleThreadEventExecutor.execute() 方法.

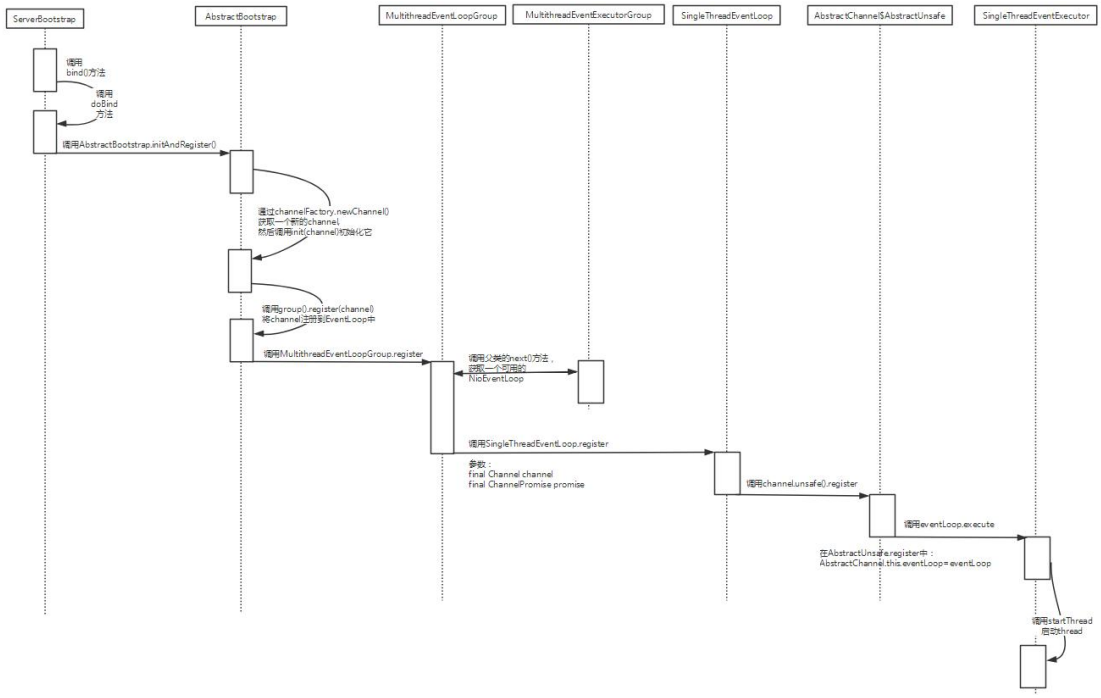
如果留心的同学可能已经注意到了，我们在前面 EventLoop 与 Channel 的关联 这一小节时，有提到到在注册 channel 的过程中，会在 AbstractChannel$AbstractUnsafe.register 中调用 eventLoop.execute 方法，在 EventLoop 中进行 Channel 注册代码的执行，AbstractChannel$AbstractUnsafe.register 部分代码如下：

```
public final void register(EventLoop eventLoop, final ChannelPromise promise) {
          // 删除判断
          AbstractChannel.this.eventLoop = eventLoop;

          if (eventLoop.inEventLoop()) {
              register0(promise);
          } else {
              try {
                  eventLoop.execute(new Runnable() {
                      @Override
                      public void run() {
                          register0(promise);
                      }
                  });
              } catch (Throwable t) {
                    // 删除异常处理代码
              }
          }
      }
```

很显然，一路从 Bootstrap.bind 方法跟踪到 AbstractChannel$AbstractUnsafe.register 方法，整个代码都是在主线程中运行的，因此上面的 eventLoop.inEventLoop() 就为 false，于是进入到 else 分支，在这个分支中调用了 eventLoop.execute. eventLoop 是一个 NioEventLoop 的实例，而 NioEventLoop 没有实现 execute 方法，因此调用的是 SingleThreadEventExecutor.execute:

```
public void execute(Runnable task) {
      // 条件判断
      boolean inEventLoop = inEventLoop();
      if (inEventLoop) {
          addTask(task);
      } else {
          startThread();
          addTask(task);
          if (isShutdown() && removeTask(task)) {
              reject();
          }
      }
      if (!addTaskWakesUp && wakesUpForTask(task)) {
          wakeup(inEventLoop);
      }
  }
```

我们已经分析过了，inEventLoop == false，因此执行到 else 分支，在这里就调用了 startThread() 方法来启动 SingleThreadEventExecutor 内部关联的 Java 本地线程了.
总结一句话，当 EventLoop.execute 第一次被调用时，就会触发 startThread() 的调用，进而导致了 EventLoop 所对应的 Java 线程的启动.
我们将 EventLoop 与 Channel 的关联 小节中的时序图补全后，就得到了 EventLoop 启动过程的时序图:



4. Promise 与 Future 双子星的秘密

java.util.concurrent.Future 是 Java 提供的接口，表示异步执行的状态，Future 的 get 方法会判断任务是否执行完成，如果完成就返回结果，否则阻塞线程，直到任务完成。
Netty 扩展了 Java 的 Future，最主要的改进就是增加了监听器 Listener 接口，通过监听器可以让异步执行更加有效率，不需要通过 get 来等待异步执行结束，而是通过监听器回调来精确地控制异步执行结束的时间点。

```
public interface Future<V> extends java.util.concurrent.Future<V> {
    boolean isSuccess();
    boolean isCancellable();
    Throwable cause();
    Future<V> addListener(GenericFutureListener<? extends Future<? super V>> listener);
    Future<V> addListeners(GenericFutureListener<? extends Future<? super V>>...
listeners);
    Future<V> removeListener(GenericFutureListener<? extends Future<? super V>>
listener);
    Future<V> removeListeners(GenericFutureListener<? extends Future<? super V>>...
listeners);
    Future<V> sync() throws InterruptedException;
```

```
    Future<V> syncUninterruptibly();
    Future<V> await() throws InterruptedException;
    Future<V> awaitUninterruptibly();
    boolean await(long timeout, TimeUnit unit) throws InterruptedException;
    boolean await(long timeoutMillis) throws InterruptedException;
    boolean awaitUninterruptibly(long timeout, TimeUnit unit);
    boolean awaitUninterruptibly(long timeoutMillis);
    V getNow();
    boolean cancel(boolean mayInterruptIfRunning);
}
```

ChannelFuture 接口扩展了 Netty 的 Future 接口，表示一种没有返回值的异步调用，同时关联了 Channel，跟一个 Channel 绑定

```
public interface ChannelFuture extends Future<Void> {
    Channel channel();
    ChannelFuture addListener(GenericFutureListener<? extends Future<? super Void>>
listener);
    ChannelFuture addListeners(GenericFutureListener<? extends Future<? super Void>>...
listeners);
    ChannelFuture removeListener(GenericFutureListener<? extends Future<? super Void>>
listener);
    ChannelFuture removeListeners(GenericFutureListener<? extends Future<? super
Void>>... listeners);
    ChannelFuture sync() throws InterruptedException;
    ChannelFuture syncUninterruptibly();
    ChannelFuture await() throws InterruptedException;
    ChannelFuture awaitUninterruptibly();
    boolean isVoid();
}
```

Promise 接口也扩展了 Future 接口，它表示一种可写的 Future，就是可以设置异步执行的结果

```
public interface Promise<V> extends Future<V> {
    Promise<V> setSuccess(V result);
    boolean trySuccess(V result);
    Promise<V> setFailure(Throwable cause);
    boolean tryFailure(Throwable cause);
    boolean setUncancellable();
    Promise<V> addListener(GenericFutureListener<? extends Future<? super V>>
listener);
    Promise<V> addListeners(GenericFutureListener<? extends Future<? super V>>...
listeners);
    Promise<V> removeListener(GenericFutureListener<? extends Future<? super V>>
listener);
    Promise<V> removeListeners(GenericFutureListener<? extends Future<? super V>>...
listeners);
```

```
    Promise<V> await() throws InterruptedException;
    Promise<V> awaitUninterruptibly();
    Promise<V> sync() throws InterruptedException;
    Promise<V> syncUninterruptibly();
}
```

ChannelPromise 接口扩展了 Promise 和 ChannelFuture，绑定了 Channel，又可写异步执行结构，又具备了监听者的功能，是 Netty 实际编程使用的表示异步执行的接口

```
public interface ChannelPromise extends ChannelFuture, Promise<Void> {
    Channel channel();
    ChannelPromise setSuccess(Void result);
    ChannelPromise setSuccess();
    boolean trySuccess();
    ChannelPromise setFailure(Throwable cause);
    ChannelPromise addListener(GenericFutureListener<? extends Future<? super Void>>
listener);
    ChannelPromise addListeners(GenericFutureListener<? extends Future<? super
Void>>... listeners);
    ChannelPromise removeListener(GenericFutureListener<? extends Future<? super Void>>
listener);
    ChannelPromise removeListeners(GenericFutureListener<? extends Future<? super
Void>>... listeners);
    ChannelPromise sync() throws InterruptedException;
    ChannelPromise syncUninterruptibly();
    ChannelPromise await() throws InterruptedException;
    ChannelPromise awaitUninterruptibly();
    ChannelPromise unvoid();
}
```

DefaultChannelPromise 是 ChannelPromise 的实现类，它是实际运行时的 Promoise 实例。

Netty 使用 addListener 的方式来回调异步执行的结果。

看一下 DefaultPromise 的 addListener 方法，它判断异步任务执行的状态，如果执行完成，就理解通知监听者，否则加入到监听者队列通知监听者就是找一个线程来执行调用监听的回调函数。

```
public Promise<V> addListener(GenericFutureListener<? extends Future<? super V>>
listener) {
        checkNotNull(listener, "listener");

        synchronized (this) {
            addListener0(listener);
        }

        if (isDone()) {
            notifyListeners();
        }
```

```
        return this;
}

private void addListener0(GenericFutureListener<? extends Future<? super V>> listener)
{
        if (listeners == null) {
            listeners = listener;
        } else if (listeners instanceof DefaultFutureListeners) {
            ((DefaultFutureListeners) listeners).add(listener);
        } else {
            listeners = new DefaultFutureListeners((GenericFutureListener<? extends
Future<V>>) listeners, listener);
        }
}

private void notifyListeners() {
        EventExecutor executor = executor();
        if (executor.inEventLoop()) {
            final InternalThreadLocalMap threadLocals = InternalThreadLocalMap.get();
            final int stackDepth = threadLocals.futureListenerStackDepth();
            if (stackDepth < MAX_LISTENER_STACK_DEPTH) {
                threadLocals.setFutureListenerStackDepth(stackDepth + 1);
                try {
                    notifyListenersNow();
                } finally {
                    threadLocals.setFutureListenerStackDepth(stackDepth);
                }
                return;
            }
        }

        safeExecute(executor, new Runnable() {
            @Override
            public void run() {
                notifyListenersNow();
            }
        });
}
```

再来看监听者的接口，就一个方法，即等异步任务执行完成后，拿到 Future 结果，执行回调的逻辑

```
public interface GenericFutureListener<F extends Future<?>> extends EventListener {
    void operationComplete(F future) throws Exception;
}
```

5. Handler 的各种姿势

5.1 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。ChannelHandlerContext 不会改变添加到其中的 ChannelHandler，因此它是安全的。

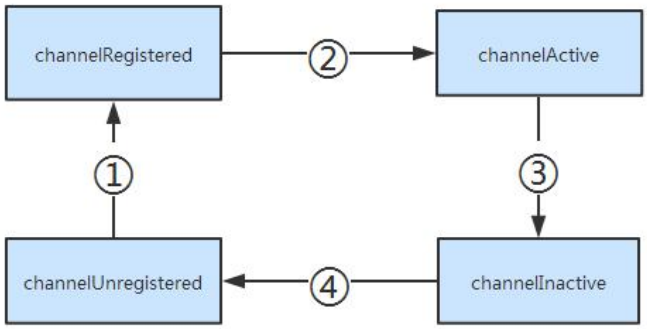
下图显示了 ChannelHandlerContext、ChannelHandler、ChannelPipeline 的关系：

5.2 Channel 的状态模型

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：

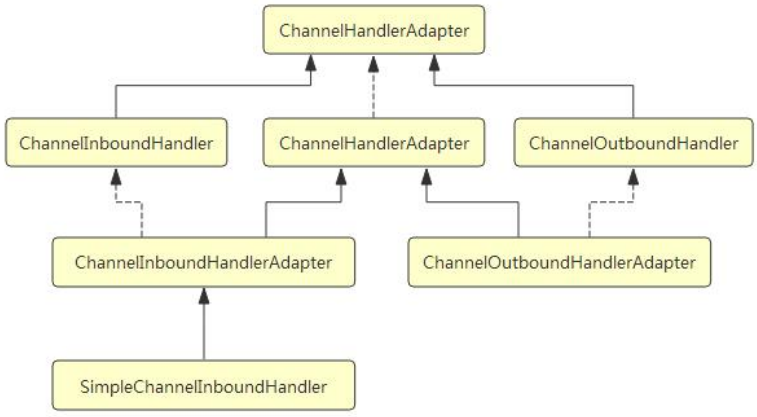
1. channelUnregistered
2. channelRegistered
3. channelActive
4. channelInactive

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：



5.2 ChannelHandler 和其子类

先看一张 Handler 的类继承图



5.3 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。

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

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。我们也可以直接继承于 ChannelHandlerAdapter，然后重写里面的方法。

5.4 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：

1. channelRegistered，ChannelHandlerContext 的 Channel 被注册到 EventLoop；
2. channelUnregistered，ChannelHandlerContext 的 Channel 从 EventLoop 中注销
3. channelActive，ChannelHandlerContext 的 Channel 已激活
4. channelInactive，ChannelHanderContxt 的 Channel 结束生命周期
5. channelRead，从当前 Channel 的对端读取消息
6. channelReadComplete，消息读取完成后执行
7. userEventTriggered，一个用户事件被触发
8. channelWritabilityChanged，改变通道的可写状态，可以使用 Channel.isWritable()检查
9. 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。
看下面的代码：

```
public class UnreleaseHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        //手动释放消息
        ReferenceCountUtil.release(msg);
    }
}
```

SimpleChannelInboundHandler 会自动释放消息

```
public class ReleaseHandler extends SimpleChannelInboundHandler<Object> {
```

```
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception
{
        //不需要手动释放
    }
}
```

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

6.数据翻译官编码和解码

6.1 TCP 黏包/拆包

TCP 是一个“流”协议，所谓流，就是没有界限的一长串二进制数据。TCP 作为传输层协议并不不了解上层业务数据的具体含义，它会根据 TCP 缓冲区的实际情况进行数据包的划分，所以在业务上认为是一个完整的包，可能会被 TCP 拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的 TCP 粘包和拆包问题。

6.2 粘包问题的解决策略

由于底层的 TCP 无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决。业界的主流协议的解决方案，可以归纳如下：

1. 消息定长，报文大小固定长度，例如每个报文的长度固定为 200 字节，如果不够空位补空格；
2. 包尾添加特殊分隔符，例如每条报文结束都添加回车换行符（例如 FTP 协议）或者指定特殊字符作为报文分隔符，接收方通过特殊分隔符切分报文区分；
3. 将消息分为消息头和消息体，消息头中包含表示信息的总长度（或者消息体长度）的字段；
4. 更复杂的自定义应用层协议。

6.3 编、解码技术

通常我们也习惯将编码（Encode）称为序列化（serialization），它将对象序列化为字节数组，用于网络传输、数据持久化或者其它用途。

反之，解码（Decode）/反序列化（deserialization）把从网络、磁盘等读取的字节数组还原成原始对象（通常是原始对象的拷贝），以方便后续的业务逻辑操作。

进行远程跨进程服务调用时（例如 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，以便完成远程调用。

6.4 Netty 为什么要提供编解码框架

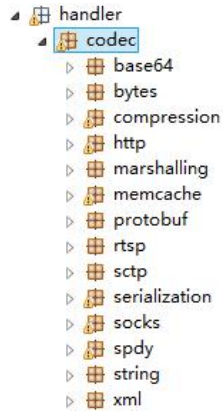
作为一个高性能的异步、NIO 通信框架，编解码框架是 Netty 的重要组成部分。尽管站在微内核的角度看，编解码框架并不是 Netty 微内核的组成部分，但是通过 ChannelHandler 定制扩展出的编解码框架却是不可或缺的。

然而，我们已经知道在 Netty 中，从网络读取的 Inbound 消息，需要经过解码，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 就是 ByteBuf）才能够发送到网络对端。编码和解码功能是 NIO 框架的有机组成部分，无论是由业务定制扩展实现，还是 NIO 框架内置编解码能力，该功能是必不可少的。

为了降低用户的开发难度，Netty 对常用的功能和 API 做了装饰，以屏蔽底层的实现细节。编解码功能的定制，对于熟悉 Netty 底层实现的开发者而言，直接基于 ChannelHandler 扩展开发，难度并不是很大。但是对于大多数初学者或者不愿意去了解底层实现细节的用户，需要提供给他们更简单的类库和 API，而不是 ChannelHandler。

Netty 在这方面做得非常出色，针对编解码功能，它既提供了通用的编解码框架供用户扩展，又提供了常用的编解码类库供用户直接使用。在保证定制扩展性的基础之上，尽量降低用户的开发工作量和开发门槛，提升开发效率。

Netty 预置的编解码功能列表如下：base64、Protobuf、JBoss Marshalling、spdy 等。



6.5 Netty 粘包和拆包解决方案

6.5.1 Netty 中常用的解码器

Netty 提供了多个解码器，可以进行分包的操作，分别是：

* LineBasedFrameDecoder
* DelimiterBasedFrameDecoder（添加特殊分隔符报文来分包）
* FixedLengthFrameDecoder（使用定长的报文来分包）
* LengthFieldBasedFrameDecoder

LineBasedFrameDecoder 解码器

LineBasedFrameDecoder 是回车换行解码器，如果用户发送的消息以回车换行符作为消息结束的标识，则可以直接使用 Netty 的 LineBasedFrameDecoder 对消息进行解码，只需要在初始化 Netty 服务端或者客户端时将 LineBasedFrameDecoder 正确的添加到 ChannelPipeline 中即可，不需要自己重新实现一套换行解码器。

LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断看是否有“\n”或者“\r\n”，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。防止由于数据报没有携带换行符导致接收到 ByteBuf 无限制积压，引起系统内存溢出。

它的使用效果如下：

```
解码之前：
+--------------------------------------------------------------+
                        接收到的数据报
“This is a netty example for using the nio framework.\r\n When you“
```

```
+------------------------------------------------------------------+
解码之后的 ChannelHandler 接收到的 Object 如下：
+------------------------------------------------------------------+
                          解码之后的文本消息
“This is a netty example for using the nio framework.“
+------------------------------------------------------------------+
```

通常情况下，LineBasedFrameDecoder 会和 StringDecoder 配合使用，组合成按行切换的文本解码器，对于文本类协议的解析，文本换行解码器非常实用，例如对 HTTP 消息头的解析、FTP 协议消息的解析等。

下面我们简单给出文本换行解码器的使用示例：

```
pipeline.addLast(new LineBasedFrameDecoder(1024));
pipeline.addLast(new StringDecoder());
```

初始化 Channel 的时候，首先将 LineBasedFrameDecoder 添加到 ChannelPipeline 中，然后再依次添加字符串解码器 StringDecoder，业务 Handler。

DelimiterBasedFrameDecoder 解码器

DelimiterBasedFrameDecoder 是分隔符解码器，用户可以指定消息结束的分隔符，它可以自动完成以分隔符作为码流结束标识的消息的解码。回车换行解码器实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。

分隔符解码器在实际工作中也有很广泛的应用，笔者所从事的电信行业，很多简单的文本私有协议，都是以特殊的分隔符作为消息结束的标识，特别是对于那些使用长连接的基于文本的私有协议。

分隔符的指定:与大家的习惯不同,分隔符并非以 char 或者 string 作为构造参数,而是 ByteBuf，下面我们就结合实际例子给出它的用法。

假如消息以“$_”作为分隔符，服务端或者客户端初始化 ChannelPipeline 的代码实例如下：

```
ByteBuf delimiter = Unpooled.copiedBuffer("$_".getBytes());
pipeline.addLast(new DelimiterBasedFrameDecoder(1024,delimiter));
pipeline.addLast(new StringDecoder());
```

首先将“$_”转换成 ByteBuf 对象，作为参数构造 DelimiterBasedFrameDecoder，将其添加到 ChannelPipeline 中，然后依次添加字符串解码器（通常用于文本解码）和用户 Handler，请注意解码器和 Handler 的添加顺序，如果顺序颠倒，会导致消息解码失败。

DelimiterBasedFrameDecoder 原理分析：解码时，判断当前已经读取的 ByteBuf 中是否包含分隔符 ByteBuf，如果包含，则截取对应的 ByteBuf 返回，源码如下：

```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
        if (lineBasedDecoder != null) {
            return lineBasedDecoder.decode(ctx, buffer);
        }
        // Try all delimiters and choose the delimiter which yields the shortest frame.
        int minFrameLength = Integer.MAX_VALUE;
        ByteBuf minDelim = null;
        for (ByteBuf delim: delimiters) {
            int frameLength = indexOf(buffer, delim);
            if (frameLength >= 0 && frameLength < minFrameLength) {
                minFrameLength = frameLength;
```

```
                minDelim = delim;
            }
        }

        // 此处省略 N 句
    }
```

详细分析下 indexOf(buffer, delim)方法的实现，代码如下：

```
private static int indexOf(ByteBuf haystack, ByteBuf needle) {
        for (int i = haystack.readerIndex(); i < haystack.writerIndex(); i ++) {
            int haystackIndex = i;
            int needleIndex;
            for (needleIndex = 0; needleIndex < needle.capacity(); needleIndex ++) {
                if (haystack.getByte(haystackIndex) != needle.getByte(needleIndex)) {
                    break;
                } else {
                    haystackIndex ++;
                    if (haystackIndex == haystack.writerIndex() &&
                        needleIndex != needle.capacity() - 1) {
                        return -1;
                    }
                }
            }

            if (needleIndex == needle.capacity()) {
                // Found the needle from the haystack!
                return i - haystack.readerIndex();
            }
        }
        return -1;
    }
```

该算法与 Java String 中的搜索算法类似，对于原字符串使用两个指针来进行搜索，如果搜索成功，则返回索引位置，否则返回-1。

FixedLengthFrameDecoder 解码器

FixedLengthFrameDecoder 是固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑 TCP 的粘包/拆包等问题，非常实用。

对于定长消息，如果消息实际长度小于定长，则往往会进行补位操作，它在一定程度上导致了空间和资源的浪费。但是它的优点也是非常明显的，编解码比较简单，因此在实际项目中仍然有一定的应用场景。

利用 FixedLengthFrameDecoder 解码器，无论一次接收到多少数据报，它都会按照构造函数中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder 会缓存半包消息并等待下个包到达后进行拼包，直到读取到一个完整的包。

假如单条消息的长度是 20 字节，使用 FixedLengthFrameDecoder 解码器的效果如下：

```
解码前：
+------------------------------------------------------------------+
```

```
                        接收到的数据报
“HELLO NETTY FOR USER DEVELOPER“
+-------------------------------------------------------------------+
解码后：
+-------------------------------------------------------------------+
                        解码后的数据报
“HELLO NETTY FOR USER“
+-------------------------------------------------------------------+
```

LengthFieldBasedFrameDecoder 解码器

了解 TCP 通信机制的该都知道 TCP 底层的粘包和拆包，当我们在接收消息的时候，显示不能认为读取到的报文就是个整包消息，特别是对于采用非阻塞 I/O 和长连接通信的程序。

如何区分一个整包消息，通常有如下 4 种做法：

1) 固定长度，例如每 120 个字节代表一个整包消息，不足的前面补位。解码器在处理这类定常消息的时候比较简单，每次读到指定长度的字节后再进行解码；

2) 通过回车换行符区分消息，例如 HTTP 协议。这类区分消息的方式多用于文本协议；

3) 通过特定的分隔符区分整包消息；

4) 通过在协议头/消息头中设置长度字段来标识整包消息。

前三种解码器之前的章节已经做了详细介绍，下面让我们来一起学习最后一种通用解码器-LengthFieldBasedFrameDecoder。

大多数的协议（私有或者公有），协议头中会携带长度字段，用于标识消息体或者整包消息的长度，例如 SMPP、HTTP 协议等。由于基于长度解码需求的通用性，以及为了降低用户的协议开发难度，Netty 提供了 LengthFieldBasedFrameDecoder，自动屏蔽 TCP 底层的拆包和粘包问题，只需要传入正确的参数，即可轻松解决“读半包“问题。

下面我们看看如何通过参数组合的不同来实现不同的“半包”读取策略。第一种常用的方式是消息的第一个字段是长度字段，后面是消息体，消息头中只包含一个长度字段。它的消息结构定义如图所示：

```
+--------+----------------+
| Length | Actual Content |
| 0x000C | "HELLO, WORLD" |
+--------+----------------+
```

使用以下参数组合进行解码：

1) lengthFieldOffset = 0;

2) lengthFieldLength = 2;

3) lengthAdjustment = 0;

4) initialBytesToStrip = 0。

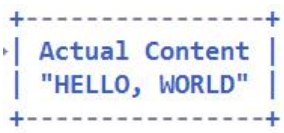
解码后的字节缓冲区内容如图所示：

```
+--------+----------------+
| Length | Actual Content |
| 0x000C | "HELLO, WORLD" |
+--------+----------------+
```

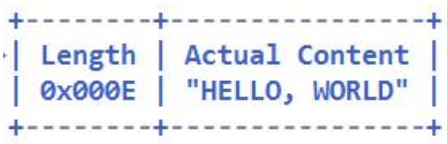
通过 ByteBuf.readableBytes()方法我们可以获取当前消息的长度，所以解码后的字节缓冲区可以不携带长度字段，由于长度字段在起始位置并且长度为 2，所以将 initialBytesToStrip 设置为 2，参数组合修改为：

1) lengthFieldOffset = 0;

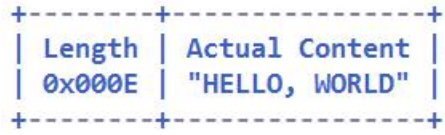
2) lengthFieldLength = 2;
3) lengthAdjustment = 0;
4) initialBytesToStrip = 2。
解码后的字节缓冲区内容如图所示：



解码后的字节缓冲区丢弃了长度字段，仅仅包含消息体，对于大多数的协议，解码之后消息长度没有用处，因此可以丢弃。
在大多数的应用场景中，长度字段仅用来标识消息体的长度，这类协议通常由消息长度字段+消息体组成，如上图所示的几个例子。但是，对于某些协议，长度字段还包含了消息头的长度。在这种应用场景中，往往需要使用 lengthAdjustment 进行修正。由于整个消息（包含消息头）的长度往往大于消息体的长度，所以，lengthAdjustment 为负数。图 2-6 展示了通过指定 lengthAdjustment 字段来包含消息头的长度：
1) lengthFieldOffset = 0;
2) lengthFieldLength = 2;
3) lengthAdjustment = -2;
4) initialBytesToStrip = 0。
解码之前的码流：



解码之后的码流：



由于协议种类繁多，并不是所有的协议都将长度字段放在消息头的首位，当标识消息长度的字段位于消息头的中间或者尾部时，需要使用 lengthFieldOffset 字段进行标识，下面的参数组合给出了如何解决消息长度字段不在首位的问题：
1) lengthFieldOffset = 2;
2) lengthFieldLength = 3;
3) lengthAdjustment = 0;
4) initialBytesToStrip = 0。
其中 lengthFieldOffset 表示长度字段在消息头中偏移的字节数,lengthFieldLength 表示长度字段自身的长度，解码效果如下：
解码之前：

解码之后：

```
+----------+----------+----------------+
| Header 1 |  Length  | Actual Content |
|  0xCAFE  | 0x00000C | "HELLO, WORLD" |
+----------+----------+----------------+
```

由于消息头 1 的长度为 2，所以长度字段的偏移量为 2；消息长度字段 Length 为 3，所以 lengthFieldLength 值为 3。由于长度字段仅仅标识消息体的长度，所以 lengthAdjustment 和 initialBytesToStrip 都为 0。

最后一种场景是长度字段夹在两个消息头之间或者长度字段位于消息头的中间,前后都有其它消息头字段，在这种场景下如果想忽略长度字段以及其前面的其它消息头字段，则可以通过 initialBytesToStrip 参数来跳过要忽略的字节长度，它的组合配置示意如下：

1) lengthFieldOffset = 1;
2) lengthFieldLength = 2;
3) lengthAdjustment = 1;
4) initialBytesToStrip = 3。

解码之前的码流（16 字节）：

```
+------+--------+------+----------------+
| HDR1 | Length | HDR2 | Actual Content |
| 0xCA | 0x000C | 0xFE | "HELLO, WORLD" |
+------+--------+------+----------------+
```

解码之后的码流（13 字节）：

```
+------+----------------+
| HDR2 | Actual Content |
| 0xFE | "HELLO, WORLD" |
+------+----------------+
```

由于 HDR1 的长度为 1，所以长度字段的偏移量 lengthFieldOffset 为 1；长度字段为 2 个字节，所以 lengthFieldLength 为 2。由于长度字段是消息体的长度,解码后如果携带消息头中的字段，则需要使用 lengthAdjustment 进行调整，此处它的值为 1，代表的是 HDR2 的长度，最后由于解码后的缓冲区要忽略长度字段和 HDR1 部分，所以 lengthAdjustment 为 3。解码后的结果为 13 个字节，HDR1 和 Length 字段被忽略。

事实上，通过 4 个参数的不同组合，可以达到不同的解码效果，用户在使用过程中可以根据业务的实际情况进行灵活调整。

由于 TCP 存在粘包和组包问题，所以通常情况下用户需要自己处理半包消息。利用 LengthFieldBasedFrameDecoder 解码器可以自动解决半包问题，它的习惯用法如下：

```
pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(65536,0,2));
```

在 pipeline 中增加 LengthFieldBasedFrameDecoder 解码器，指定正确的参数组合，它可以将 Netty 的 ByteBuf 解码成整包消息，后面的用户解码器拿到的就是个完整的数据报，按照逻辑正常进行解码即可，不再需要额外考虑“读半包”问题，降低了用户的开发难度。

6.5.2 Netty 常用的编码器

Netty 默认提供了丰富的编解码框架供用户集成使用，我们只对较常用的 Java 序列化编码器进行讲解。其它的编码器，实现方式大同小异。

ObjectEncoder 编码器

ObjectEncoder是Java序列化编码器,它负责将实现Serializable接口的对象序列化为byte[],然后写入到ByteBuf中用于消息的跨网络传输。

下面我们一起分析下它的实现:

首先,我们发现它继承自MessageToByteEncoder,它的作用就是将对象编码成ByteBuf:

```
public class ObjectEncoder extends MessageToByteEncoder<Serializable>
```

如果要使用Java序列化,对象必须实现Serializable接口,因此,它的泛型类型为Serializable。

MessageToByteEncoder的子类只需要实现encode(ChannelHandlerContext ctx, I msg, ByteBuf out)方法即可,下面我们重点关注encode方法的实现:

```
protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) throws
Exception {
        int startIdx = out.writerIndex();

        ByteBufOutputStream bout = new ByteBufOutputStream(out);
        bout.write(LENGTH_PLACEHOLDER);
        ObjectOutputStream oout = new CompactObjectOutputStream(bout);
        oout.writeObject(msg);
        oout.flush();
        oout.close();

        int endIdx = out.writerIndex();

        out.setInt(startIdx, endIdx - startIdx - 4);
    }
```

首先创建ByteBufOutputStream和ObjectOutputStream,用于将Object对象序列化到ByteBuf中,值得注意的是在writeObject之前需要先将长度字段(4个字节)预留,用于后续长度字段的更新。

依次写入长度占位符(4字节)、序列化之后的Object对象,之后根据ByteBuf的writeIndex计算序列化之后的码流长度,最后调用ByteBuf的setInt(int index, int value)更新长度占位符为实际的码流长度。

有个细节需要注意,更新码流长度字段使用了setInt方法而不是writeInt,原因就是setInt方法只更新内容,并不修改readerIndex和writerIndex。

6.5.3 自定义编、解码

尽管Netty预置了丰富的编解码类库功能,但是在实际的业务开发过程中,总是需要对编解码功能做一些定制。使用Netty的编解码框架,可以非常方便的进行协议定制。本章节将对常用的支持定制的编解码类库进行讲解,以期让读者能够尽快熟悉和掌握编解码框架。

ByteToMessageDecoder抽象解码器

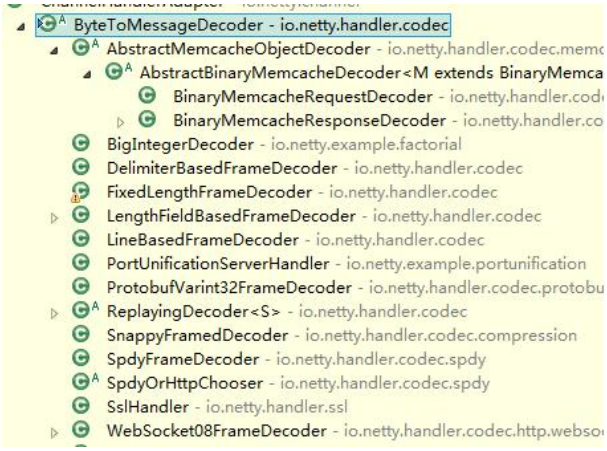
使用NIO进行网络编程时,往往需要将读取到的字节数组或者字节缓冲区解码为业务可以使用的POJO对象。为了方便业务将ByteBuf解码成业务POJO对象,Netty提供了ByteToMessageDecoder抽象工具解码类。

用户自定义解码器继承ByteToMessageDecoder,只需要实现void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)抽象方法即可完成ByteBuf到POJO对象的解码。由于ByteToMessageDecoder并没有考虑TCP粘包和拆包等场景,用户自定义解码器需要自己处

理“读半包”问题。正因为如此，大多数场景不会直接继承 ByteToMessageDecoder，而是继承另外一些更高级的解码器来屏蔽半包的处理。

实际项目中，通常将 LengthFieldBasedFrameDecoder 和 ByteToMessageDecoder 组合使用，前者负责将网络读取的数据报解码为整包消息，后者负责将整包消息解码为最终的业务对象。

除了和其它解码器组合形成新的解码器之外，ByteToMessageDecoder 也是很多基础解码器的父类，它的继承关系如下图所示：



MessageToMessageDecoder 抽象解码器

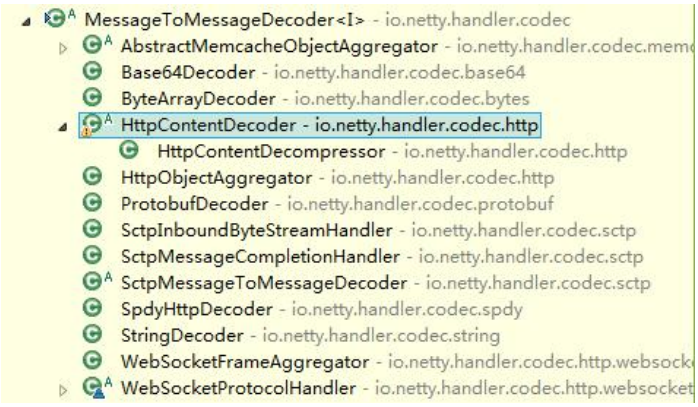
MessageToMessageDecoder 实际上是 Netty 的二次解码器，它的职责是将一个对象二次解码为其它对象。

为什么称它为二次解码器呢？我们知道，从 SocketChannel 读取到的 TCP 数据报是 ByteBuffer，实际就是字节数组。我们首先需要将 ByteBuffer 缓冲区中的数据报读取出来，并将其解码为 Java 对象；然后对 Java 对象根据某些规则做二次解码，将其解码为另一个 POJO 对象。因为 MessageToMessageDecoder 在 ByteToMessageDecoder 之后，所以称之为二次解码器。

二次解码器在实际的商业项目中非常有用，以 HTTP+XML 协议栈为例，第一次解码往往是将字节数组解码成 HttpRequest 对象，然后对 HttpRequest 消息中的消息体字符串进行二次解码，将 XML 格式的字符串解码为 POJO 对象，这就用到了二次解码器。类似这样的场景还有很多，不再一一枚举。

事实上，做一个超级复杂的解码器将多个解码器组合成一个大而全的 MessageToMessageDecoder 解码器似乎也能解决多次解码的问题，但是采用这种方式的代码可维护性会非常差。例如，如果我们打算在 HTTP+XML 协议栈中增加一个打印码流的功能，即首次解码获取 HttpRequest 对象之后打印 XML 格式的码流。如果采用多个解码器组合，在中间插入一个打印消息体的 Handler 即可，不需要修改原有的代码；如果做一个大而全的解码器，就需要在解码的方法中增加打印码流的代码，可扩展性和可维护性都会变差。

用户的解码器只需要实现 void decode(ChannelHandlerContext ctx, I msg, List<Object> out) 抽象方法即可，由于它是将一个 POJO 解码为另一个 POJO，所以一般不会涉及到半包的处理，相对于 ByteToMessageDecoder 更加简单些。它的继承关系图如下所示：



MessageToByteEncoder 抽象编码器

MessageToByteEncoder 负责将 POJO 对象编码成 ByteBuf，用户的编码器继承 MessageToByteEncoder，实现 void encode(ChannelHandlerContext ctx, I msg, ByteBuf out)接口接口，示例代码如下：

```
public class IntegerEncoder extends MessageToByteEncoder<Integer> {
    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg,ByteBuf out)
       throws Exception {
           out.writeInt(msg);
    }
}
```

它的实现原理如下：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，如果不支持则直接透传；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：

```
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);
            }
        } catch (EncoderException e) {
            throw e;
        } catch (Throwable e) {
            throw new EncoderException(e);
        } finally {
            if (buf != null) {
                buf.release();
            }
        }
    }
```

编码使用的缓冲区分配完成之后，调用 encode 抽象方法进行编码，方法定义如下：它由子类负责具体实现。

```
protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf out) throws Exception;
```

编码完成之后,调用 ReferenceCountUtil 的 release 方法释放编码对象 msg。对编码后的 ByteBuf 进行以下判断：

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。

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，它的一种实现方式是：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 解码器，实现 void encode(Channel HandlerContext ctx, I msg, List<Object> out)方法即可。注意，它与 MessageToByteEncoder 的区别是输出是对象列表而不是 ByteBuf，示例代码如下：

```
public class IntegerToStringEncoder extends MessageToMessageEncoder <Integer> {
        @Override
        public void encode(ChannelHandlerContext ctx, Integer message,
           List<Object> out)
                throws Exception
        {
            out.add(message.toString());
        }
}
```

MessageToMessageEncoder 编码器的实现原理与之前分析的 MessageToByteEncoder 相似，唯一的差别是它编码后的输出是个中间对象，并非最终可传输的 ByteBuf。

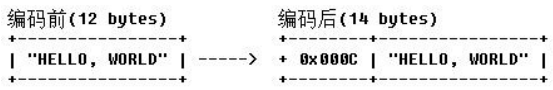
简单看下它的源码实现：创建 RecyclableArrayList 对象，判断当前需要编码的对象是否是编码器可处理的类型，如果不是，则忽略，执行下一个 ChannelHandler 的 write 方法。

具体的编码方法实现由用户子类编码器负责完成，如果编码后的 RecyclableArrayList 为空，说明编码没有成功，释放 RecyclableArrayList 引用。
如果编码成功，则通过遍历 RecyclableArrayList，循环发送编码后的 POJO 对象，代码如下所示：

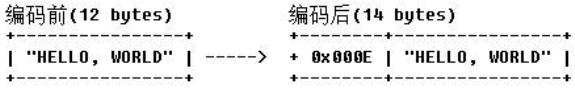
```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws
Exception {
        CodecOutputList out = null;
        try {
            if (acceptOutboundMessage(msg)) {
                out = CodecOutputList.newInstance();
                @SuppressWarnings("unchecked")
                I cast = (I) msg;
                try {
                    encode(ctx, cast, out);
                } finally {
                    ReferenceCountUtil.release(cast);
                }

                if (out.isEmpty()) {
                    out.recycle();
                    out = null;

                    throw new EncoderException(
                            StringUtil.simpleClassName(this) + " must produce at least one
message.");
                }
            } else {
                ctx.write(msg, promise);
            }
        }
        // 省略异常处理代码
    }
```

LengthFieldPrepender 编码器

如果协议中的第一个字段为长度字段，Netty 提供了 LengthFieldPrepender 编码器，它可以计算当前待发送消息的二进制字节长度，将该长度添加到 ByteBuf 的缓冲区头中，如图所示：



通过 LengthFieldPrepender 可以将待发送消息的长度写入到 ByteBuf 的前 2 个字节，编码后的消息组成为长度字段+原消息的方式。
通过设置 LengthFieldPrepender 为 true，消息长度将包含长度本身占用的字节数，打开 LengthFieldPrepender 后，图 3-3 示例中的编码结果如下图所示：



LengthFieldPrepender 工作原理分析如下：首先对长度字段进行设置，如果需要包含消息长度自身，则在原来长度的基础之上再加上 lengthFieldLength 的长度。
如果调整后的消息长度小于 0，则抛出参数非法异常。对消息长度自身所占的字节数进行判断，以便采用正确的方法将长度字段写入到 ByteBuf 中，共有以下 6 种可能：
1）长度字段所占字节为 1：如果使用 1 个 Byte 字节代表消息长度，则最大长度需要小于 256 个字节。对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeByte 将长度值写入到 ByteBuf 中；
2）长度字段所占字节为 2：如果使用 2 个 Byte 字节代表消息长度，则最大长度需要小于 65536 个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeShort 将长度值写入到 ByteBuf 中；
3）长度字段所占字节为 3：如果使用 3 个 Byte 字节代表消息长度，则最大长度需要小于 16777216 个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeMedium 将长度值写入到 ByteBuf 中；
4）长度字段所占字节为 4：创建新的 ByteBuf，并通过 writeInt 将长度值写入到 ByteBuf 中；
5）长度字段所占字节为 8：创建新的 ByteBuf，并通过 writeLong 将长度值写入到 ByteBuf 中；
6）其它长度值：直接抛出 Error。
相关代码如下：

```
protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws
Exception {
        int length = msg.readableBytes() + lengthAdjustment;
        if (lengthIncludesLengthFieldLength) {
            length += lengthFieldLength;
        }

        if (length < 0) {
            throw new IllegalArgumentException(
                    "Adjusted frame length (" + length + ") is less than zero");
        }

        switch (lengthFieldLength) {
        case 1:
            if (length >= 256) {
                throw new IllegalArgumentException(
                        "length does not fit into a byte: " + length);
            }
            out.add(ctx.alloc().buffer(1).order(byteOrder).writeByte((byte) length));
            break;
        case 2:
            if (length >= 65536) {
                throw new IllegalArgumentException(
```

```
                        "length does not fit into a short integer: " + length);
            }
            out.add(ctx.alloc().buffer(2).order(byteOrder).writeShort((short)
length));
            break;
        case 3:
            if (length >= 16777216) {
                throw new IllegalArgumentException(
                        "length does not fit into a medium integer: " + length);
            }
            out.add(ctx.alloc().buffer(3).order(byteOrder).writeMedium(length));
            break;
        case 4:
            out.add(ctx.alloc().buffer(4).order(byteOrder).writeInt(length));
            break;
        case 8:
            out.add(ctx.alloc().buffer(8).order(byteOrder).writeLong(length));
            break;
        default:
            throw new Error("should not reach here");
        }
        out.add(msg.retain());
    }
```