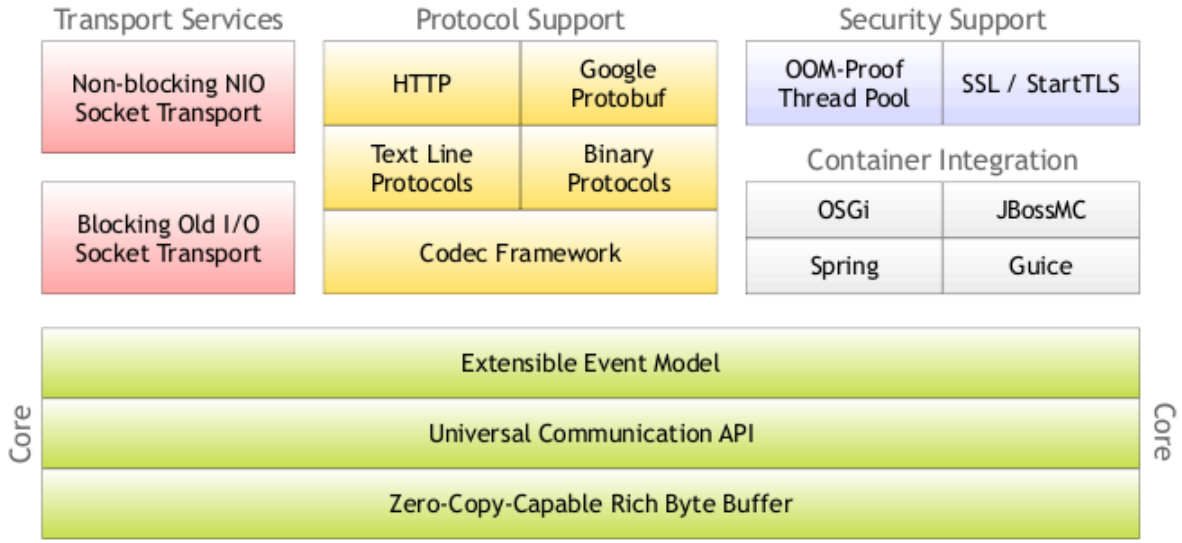
Netty 实现原理浅析

Netty 是 JBoss 出品的高效的 Java NIO 开发框架，关于其使用，可参考我的另一篇文章 netty 使用初步。本文将主要分析 Netty 实现方面的东西，由于精力有限，本人并没有对其源码做了极细致的研 究。如果下面的内容有错误或不严谨的地方，也请指正和谅解。对于 Netty 使用者来说，Netty 提供了几个典型的 example，并有详尽的 API doc 和 guide doc，本文的一些内容及图示也来自于 Netty 的文档，特此致谢。

1、总体结构

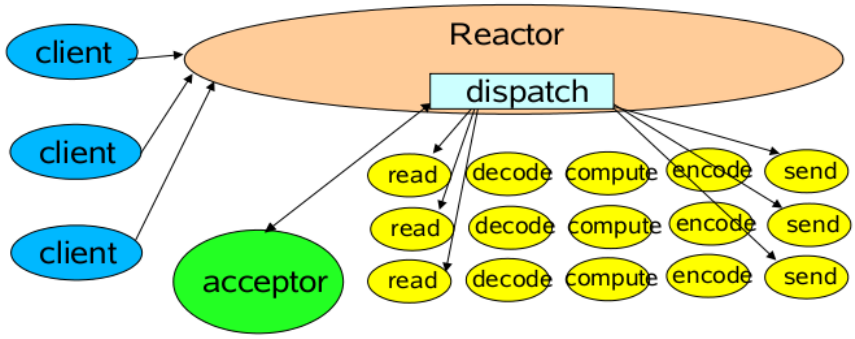


先放上一张漂亮的 Netty 总体结构图，下面的内容也主要围绕该图上的一些核心功能做分析，但对如 Container Integration 及 Security Support 等高级可选功能，本文不予分析。

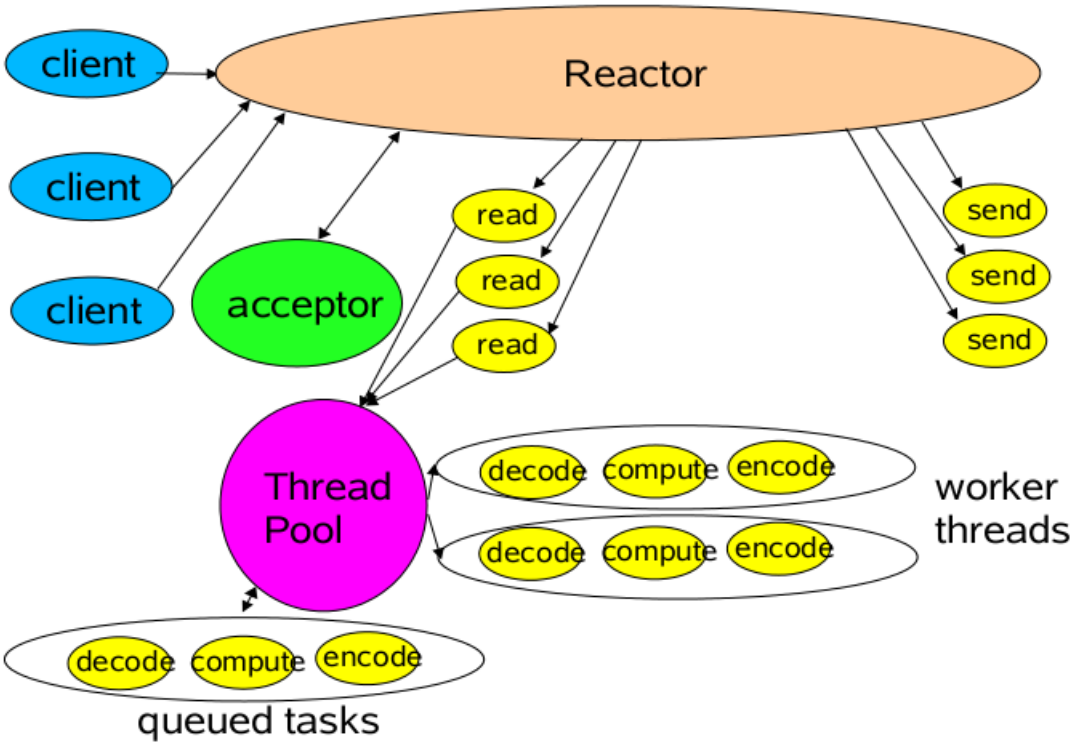
2、网络模型

Netty 是典型的 Reactor 模型结构，关于 Reactor 的详尽阐释，可参考 POSA2，这里不做概念性的解释。而应用 Java NIO 构建 Reactor 模式，Doug Lea（就是那位让人无限景仰的大爷）在“Scalable IO in Java”中给了很好的阐述。这里截取其 PPT 中经典的图例说明 Reactor 模式的典型实现：

1、这是最简单的单 Reactor 单线程模型。Reactor 线程是个多面手，负责多路分离套接字，Accept 新连接，并分派请求到处理器链中。该模型 适用于处理器链中业务处理组件能快速完成的场景。不过，这种单线程模型不能充分利用多核资源，所以实际使用的不多。

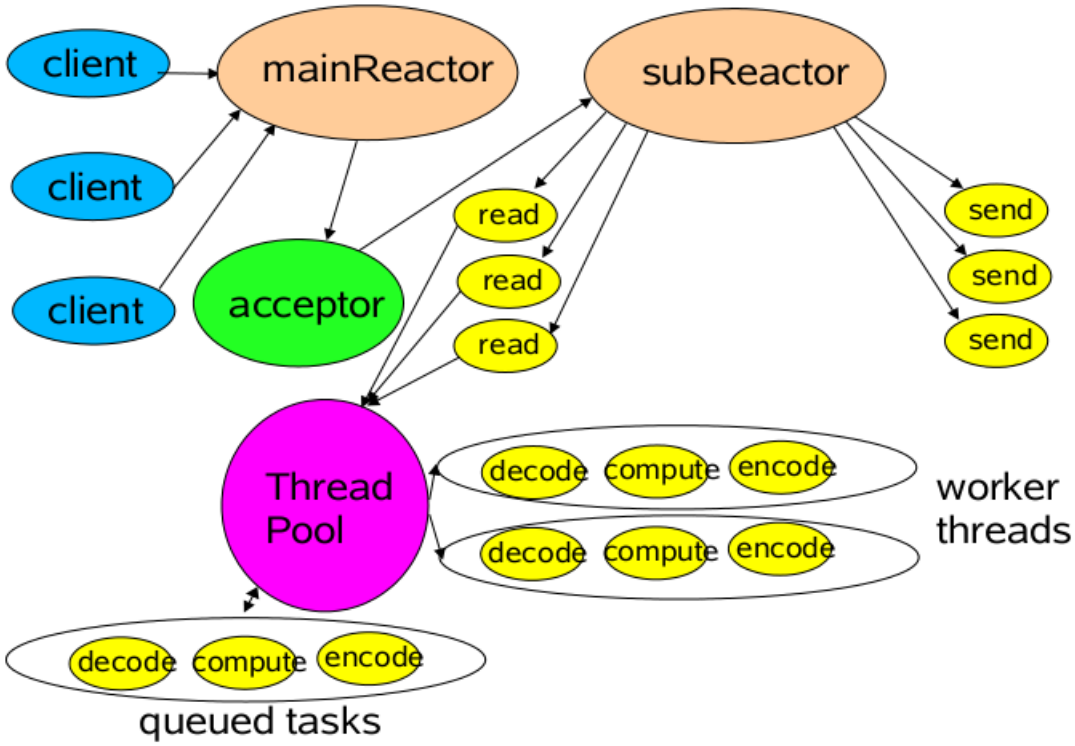


2、相比上一种模型，该模型在处理器链部分采用了多线程（线程池），也是后端程序常用的模型。



3、 第三种模型比起第二种模型，是将 Reactor 分成两部分，mainReactor 负责监听 server socket，accept 新连接，并将建立的 socket 分派给 subReactor。subReactor 负责多路分离已连接的 socket，读写网 络数据，对业务处理功能，

其扔给 worker 线程池完成。通常，subReactor 个数上可与 CPU 个数等同。



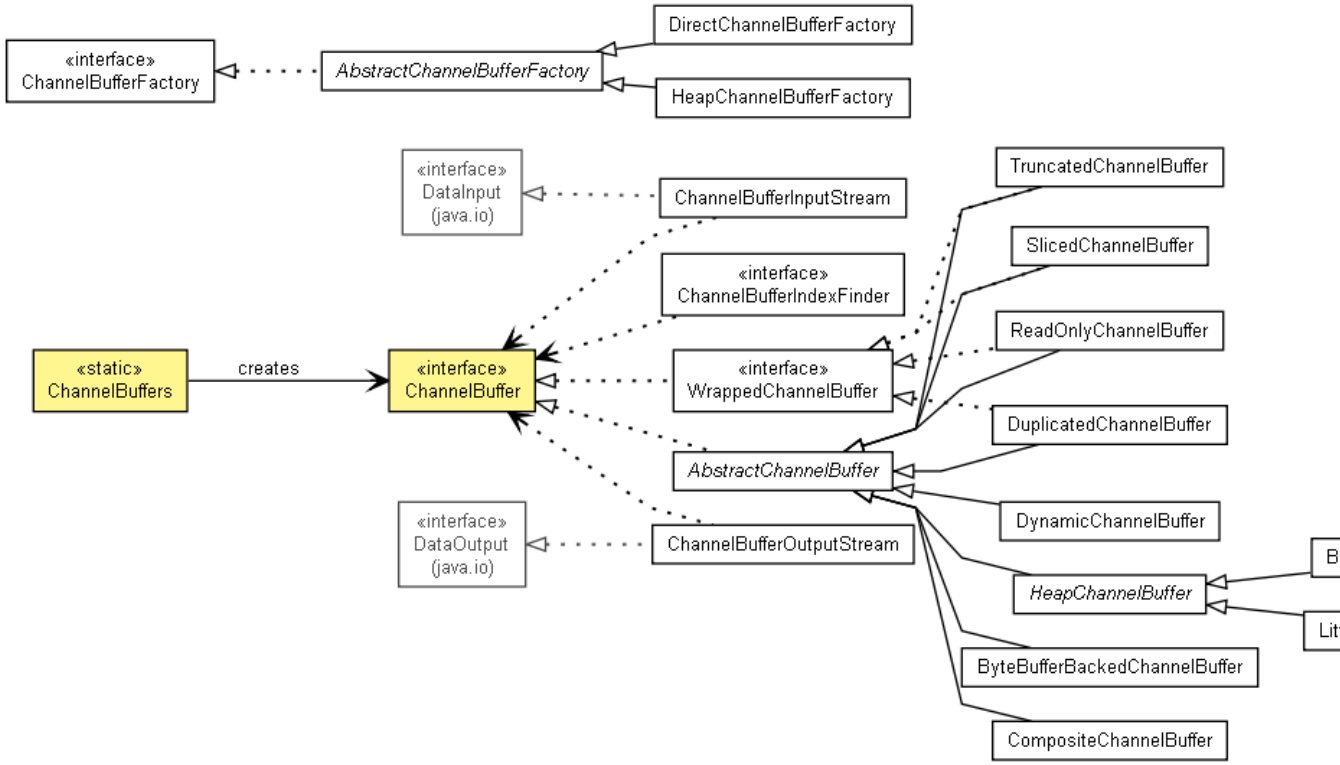
说完 Reacotr 模型的三种形式，那么 Netty 是哪种呢？其实，我还有一种 Reactor 模型的变种没说，那就是去掉线程池的第三种形式的变种，这也 是 Netty NIO 的默认模式。在实现上，Netty 中的 Boss 类充当 mainReactor，NioWorker 类充当 subReactor（默认 NioWorker 的个数是 Runtime.getRuntime().availableProcessors()）。在处理新来的请求 时，NioWorker 读完已收到的数据到 ChannelBuffer 中，之后触发 ChannelPipeline 中的 ChannelHandler 流。

Netty 是事件驱动的，可以通过 ChannelHandler 链来控制执行流向。因为 ChannelHandler 链的执行过程是在 subReactor 中同步的，所以如果业务处理 handler 耗时长，将严重影响可支持的并发数。这种模型适合于像 Memcache 这样的应用场景，但 对需要操作数据库或者和其他模块阻塞交互的系统就不是很合适。Netty 的可扩展性非常好，而像 ChannelHandler 线程池化的需要，可以通过在 ChannelPipeline 中添加 Netty 内置的 ChannelHandler 实现类 -ExecutionHandler 实现，对使用者来说只是 添加一行代码而已。对于 ExecutionHandler 需要的线程池模型，Netty 提供了两种可 选：1） MemoryAwareThreadPoolExecutor 可控制 Executor 中待处理任务的上限（超过上限时，后续进来的任务将被阻 塞），并可控制单个 Channel 待处理任务的上限；2） OrderedMemoryAwareThreadPoolExecutor 是 MemoryAwareThreadPoolExecutor 的子类，它还可以保证同一 Channel 中处理的事件流的顺序性，这主要是控制事件在异步处 理模式下可能出现的错误的事件顺序，但它并不保证同一 Channel 中的事件都在一个线程中执行（通常也没

必要）。一般来 说，OrderedMemoryAwareThreadPoolExecutor 是个很不错的选择，当然，如果有需要，也可以 DIY 一个。

3、 buffer

org.jboss.netty.buffer 包的接口及类的结构图如下：



该包核心的接口是 ChannelBuffer 和 ChannelBufferFactory,下面予以简要的介绍。

Netty 使用 ChannelBuffer 来存储并操作读写的网络数据。ChannelBuffer 除了提供和 ByteBuffer 类似的方法，还提供了 一些实用方法，具体可参考其 API 文档。ChannelBuffer 的实现类有多个，这里列举其中主要的几个：

1）HeapChannelBuffer：这是 Netty 读网络数据时默认使用的 ChannelBuffer，这里的 Heap 就是 Java 堆的意思，因为 读 SocketChannel 的数据是要经过 ByteBuffer 的，而 ByteBuffer 实际操作的就是个 byte 数组，所以 ChannelBuffer 的内部就包含了一个 byte 数组，使得 ByteBuffer 和 ChannelBuffer 之间的转换是零拷贝方式。根据网络字 节续的不同，HeapChannelBuffer 又分为 BigEndianHeapChannelBuffer 和 LittleEndianHeapChannelBuffer，默认使用的是 BigEndianHeapChannelBuffer。Netty 在读网络 数据时使用的就是 HeapChannelBuffer，HeapChannelBuffer 是个大小固定的 buffer，为了不至于分配的 Buffer 的 大小不太合适，Netty 在分配 Buffer 时会参考上次请求需要的大小。

2）DynamicChannelBuffer：相比于 HeapChannelBuffer，DynamicChannelBuffer 可动态自适应大 小。对于在 DecodeHandler 中的写数据操作，在数据大小未知的情况下，通常使用 DynamicChannelBuffer。

3）ByteBufferBackedChannelBuffer：这是 directBuffer，直接封装了 ByteBuffer 的 directBuffer。

对于读写网络数据的 buffer，分配策略有两种：1）通常出于简单考虑，直接分配固定大小的 buffer，缺点是，对一些应用来说这个大小限制有时是不 合理的，并且如果 buffer 的上限很大也会有内存上的浪费。2）针对固定大小的 buffer 缺点，就引入动态 buffer，动态 buffer 之于固定 buffer 相当于 List 之于 Array。

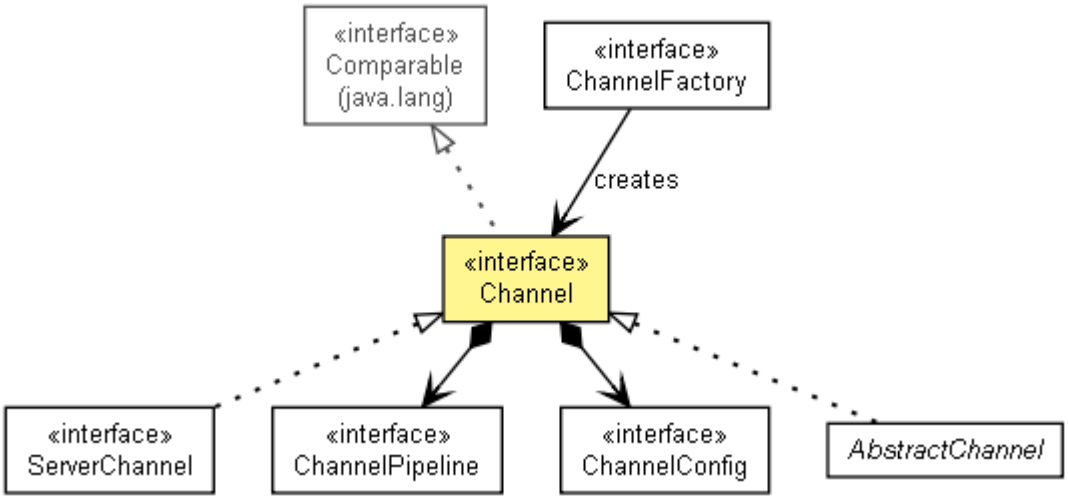
buffer 的寄存策略常见的也有两种（其实是我知道的就限于此）：1）在多线程（线程池） 模型下，每个线程维护自己的读写 buffer，每次处理新的请求前清空 buffer（或者在处理结束后清空），该请求的读写操作都需要在该线程中完成。 2）buffer 和 socket 绑定而与线程无关。两种方法的目的都是为了重用 buffer。

Netty 对 buffer 的处理策略是：读 请求数据时，Netty 首先读数据到新创建的固定大小的 HeapChannelBuffer 中，当 HeapChannelBuffer 满或者没有数据可读时，调用 handler 来处理数据，这通常首先触发的是用户自定义的 DecodeHandler，因为 handler 对象是和 ChannelSocket 绑定的，所以在 DecodeHandler 里可以设置 ChannelBuffer 成员，当解析数据包发现数据不完整时就终止此次处理流程，等下次读事件触 发时接着上次的数据继续解析。就这个过程来说，和 ChannelSocket 绑定的 DecodeHandler 中的 Buffer 通常是动态的可重用 Buffer（DynamicChannelBuffer），而在 NioWorker 中读 ChannelSocket 中的数据的 buffer 是临时分配的 固定大小的 HeapChannelBuffer，这个转换过程是有个字节拷贝行为的。

对 ChannelBuffer 的创建，Netty 内部使用的是 ChannelBufferFactory 接口，具体的实现有 DirectChannelBufferFactory 和 HeapChannelBufferFactory。对于开发者创建 ChannelBuffer，可使用实用类 ChannelBuffers 中的工厂方法。

4、Channel

和 Channel 相关的接口及类结构图如下：



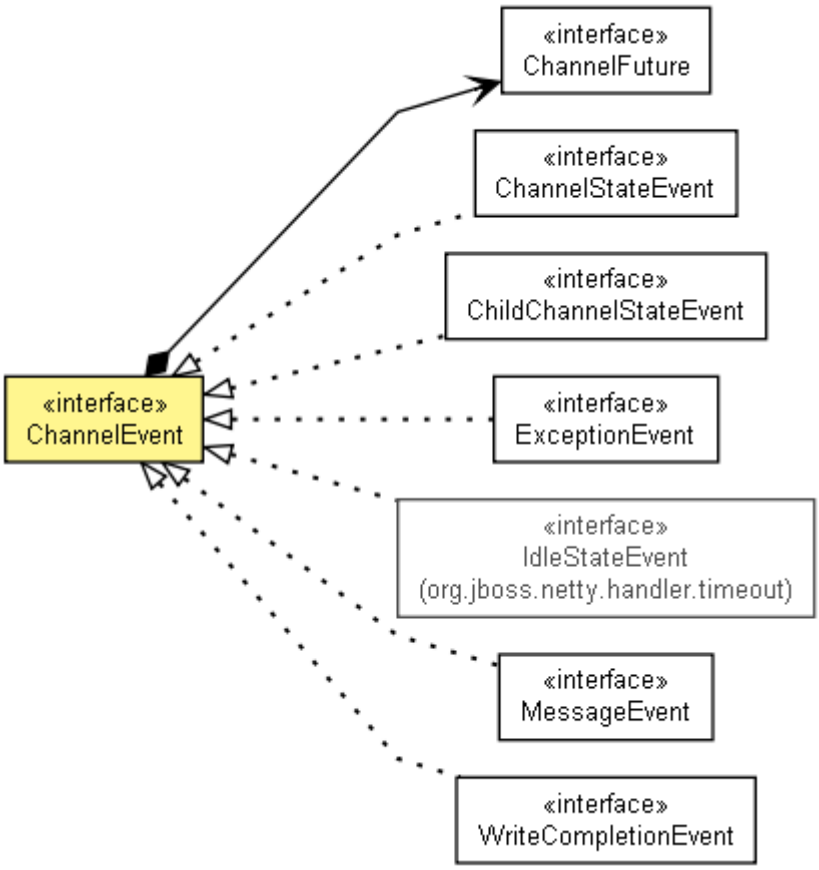
从该结构图也可以看到，Channel 主要提供的功能如下：

1）当前 Channel 的状态信息，比如是打开还是关闭等。
2）通过 ChannelConfig 可以得到的 Channel 配置信息。
3）Channel 所支持的如 read、write、bind、connect 等 IO 操作。
4）得到处理该 Channel 的 ChannelPipeline，既而可以调用其做和请求相关的 IO 操作。

在 Channel 实现方面，以通常使用的 nio socket 来说，Netty 中的 NioServerSocketChannel 和 NioSocketChannel 分别封装了 java.nio 中包含的 ServerSocketChannel 和 SocketChannel 的功能。

5、ChannelEvent

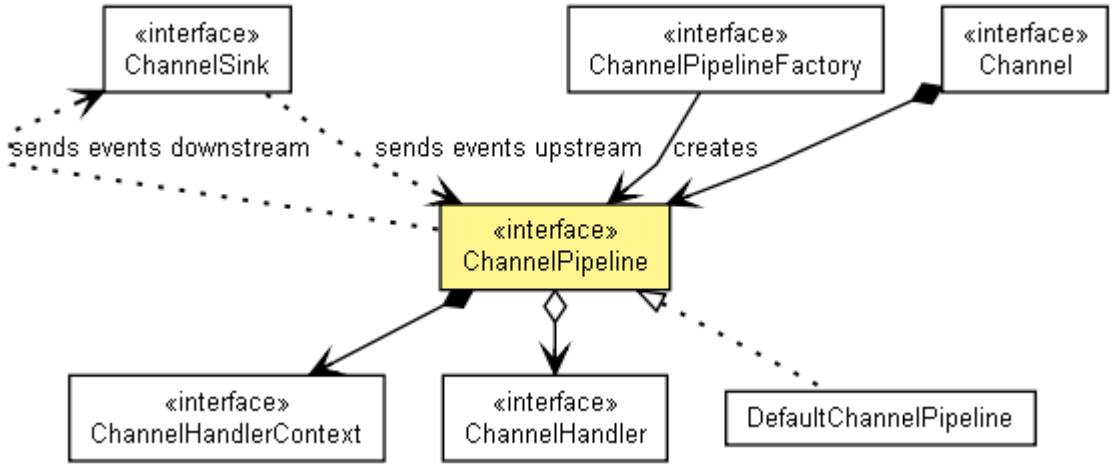
如前所述，Netty 是事件驱动的，其通过 ChannelEvent 来确定事件流的方向。一个 ChannelEvent 是依附于 Channel 的 ChannelPipeline 来处理，并由 ChannelPipeline 调用 ChannelHandler 来做具体的处理。下面是和 ChannelEvent 相关的接口及类图：



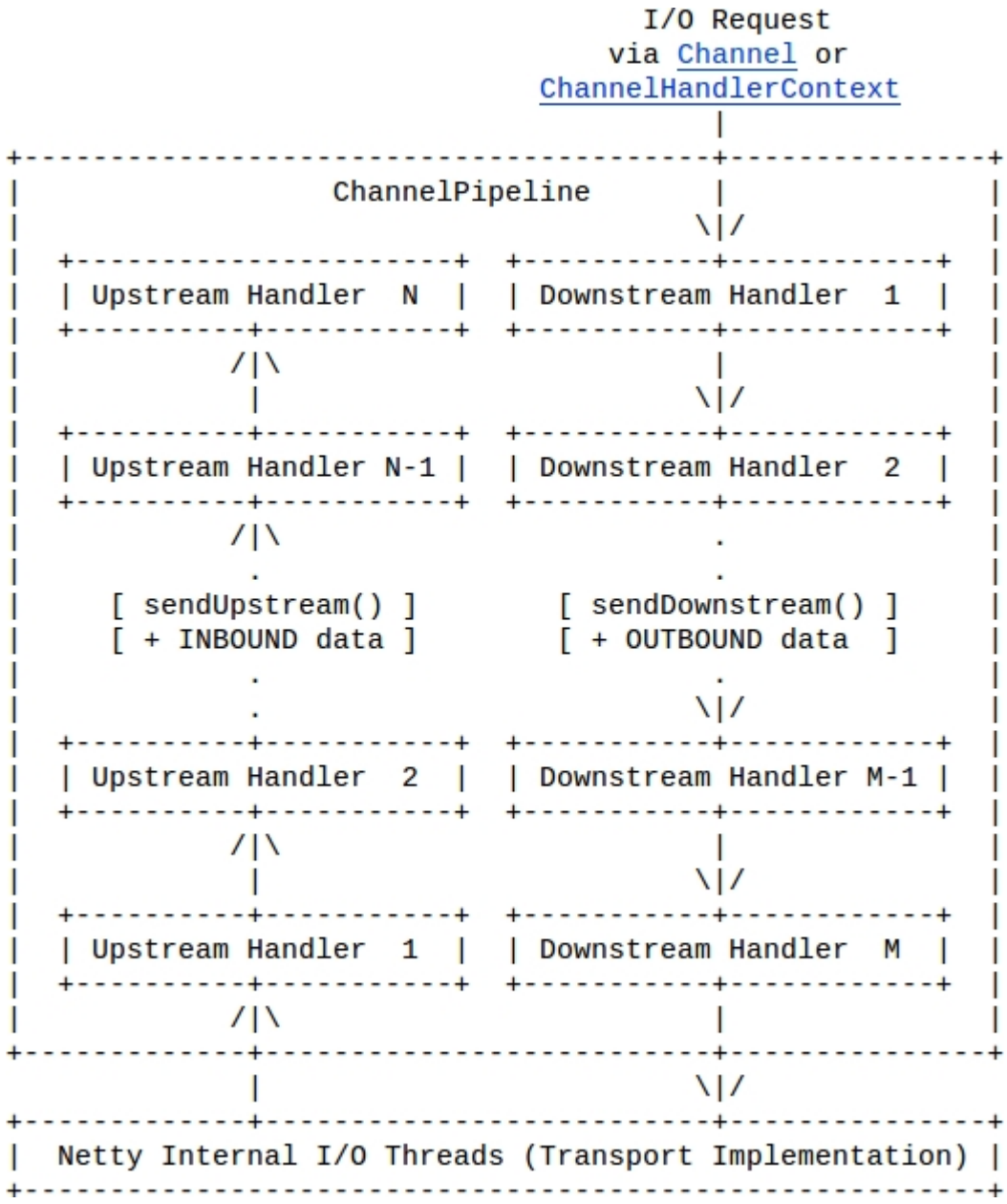
对于使用者来说，在 ChannelHandler 实现类中会使用继承于 ChannelEvent 的 MessageEvent，调用其 getMessage()方法来获得读到的 ChannelBuffer 或被转化的对象。

6、ChannelPipeline

Netty 在事件处理上，是通过 ChannelPipeline 来控制事件流，通过调用注册其上的一系列 ChannelHandler 来处理事件，这也是典型的拦截 器模式。下面是和 ChannelPipeline 相关的接口及类图：



事件流有两种，upstream 事件和 downstream 事件。在 ChannelPipeline 中，其可被注册的 ChannelHandler 既可以 是 ChannelUpstreamHandler 也可以是 ChannelDownstreamHandler ，但事件在 ChannelPipeline 传递过程中只会调用匹配流的 ChannelHandler。在事件流的过滤器链 中，ChannelUpstreamHandler 或 ChannelDownstreamHandler 既可以终止流程，也可以通过调用 ChannelHandlerContext.sendUpstream(ChannelEvent)或 ChannelHandlerContext.sendDownstream(ChannelEvent)将事件传递下去。下面是事件流处理的图示：



从上图可见，upstream event 是被 Upstream Handler 们自底向上逐个处理，downstream event 是被 Downstream Handler 们自顶向下逐个处理，这里的上下关系就是向 ChannelPipeline 里添加 Handler 的先后顺序关系。简单的理 解，upstream event 是处理来自外部的请求的过程，而 downstream event 是处理向外发送请求的过程。

服务端处 理请求的过程通常就是解码请求、业务逻辑处理、编码响应，构建的 ChannelPipeline 也就类似下面的代码片断：

```
ChannelPipeline pipeline = Channels.pipeline();
pipeline.addLast("decoder", new MyProtocolDecoder());
pipeline.addLast("encoder", new MyProtocolEncoder());
pipeline.addLast("handler", new MyBusinessLogicHandler());
```

其中,MyProtocolDecoder 是 ChannelUpstreamHandler 类型,MyProtocolEncoder 是 ChannelDownstreamHandler 类型，MyBusinessLogicHandler 既可以是 ChannelUpstreamHandler 类型，也可兼 ChannelDownstreamHandler 类型，视其是服务端程序还是客户端程序以及 应用需要而定。

补充一点，Netty 对抽象和实现做了很好的解耦。像
org.jboss.netty.channel.socket 包，定义了一些和 socket 处理相关的接口，而 org.jboss.netty.channel.socket.nio、
org.jboss.netty.channel.socket.oio 等包，则是和协议相关的实现。

7、codec framework

对于请求协议的编码解码，当然是可以按照协议格式自己操作 ChannelBuffer 中的字节数据。另一方面，Netty 也做了几个很实用的 codec helper，这里给出简单的介绍。

1）FrameDecoder：FrameDecoder 内部维护了一个 DynamicChannelBuffer 成员来存储接收到的数据，它就像个抽象模板，把整个解码过程模板写好了，其子类只需实现 decode 函数即可。 FrameDecoder 的直接实现类有两个：（1）DelimiterBasedFrameDecoder 是基于分割符 （比如\r\n）的解码器，可在构造函数中指定分割符。（2）LengthFieldBasedFrameDecoder 是基于长度字段的解码器。如果协 议 格式类似“内容长度”+内容、“固定头”+“内容长度”+动态内容这样的格式,就可以使用该解码器,其使用方法在 API DOC 上详尽的解释。
2）ReplayingDecoder： 它是 FrameDecoder 的一个变种子类，它相对于 FrameDecoder 是非阻塞解码。也就是说，使用 FrameDecoder 时需要考虑到读到的数据有可能是不完整的，而使用 ReplayingDecoder 就可以假定读到了全部的数据。
3）ObjectEncoder 和 ObjectDecoder：编码解码序列化的 Java 对象。
4）HttpRequestEncoder 和 HttpRequestDecoder：http 协议处理。

下面来看使用 FrameDecoder 和 ReplayingDecoder 的两个例子：

```
        public class IntegerHeaderFrameDecoder extends FrameDecoder {
                protected Object decode(ChannelHandlerContext ctx,
Channel channel,
                                ChannelBuffer buf) throws Exception {
                        if (buf.readableBytes() &lt; 4) {
                                return null;
                        }
                        buf.markReaderIndex();
                        int length = buf.readInt();
                        if (buf.readableBytes() &lt; length) {
                                buf.resetReaderIndex();
                                return null;
```

```
                }
                return buf.readBytes(length);
            }
        }
```

而使用 ReplayingDecoder 的解码片断类似下面的，相对来说会简化很多。

```
        public class IntegerHeaderFrameDecoder2 extends
ReplayingDecoder {
            protected Object decode(ChannelHandlerContext ctx,
Channel channel,
                    ChannelBuffer buf, VoidEnum state)
throws Exception {
                return buf.readBytes(buf.readInt());
            }
        }
```

就实现来说，当在 ReplayingDecoder 子类的 decode 函数中调用 ChannelBuffer 读数据时，如果读失败，那么 ReplayingDecoder 就会 catch 住其抛出的 Error，然后 ReplayingDecoder 接手控制权，等待下一次读到后续的数据后继 续 decode。

8、小结

尽管该文行至此处将止，但该文显然没有将 Netty 实现原理深入浅出的说全说透。当我打算写这篇文章时，也是一边看 Netty 的代码，一边总结些可写的东 西，但前后断断续续，到最后都没了多少兴致。我还是爱做一些源码分析的事情，但精力终究有限，并且倘不能把源码分析的结果有条理的托出来，不能产生有意义的心得，这分析也没什么价值和趣味。而就分析 Netty 代码的感受来说，Netty 的代码很漂亮，结构上层次上很清晰，不过这种面向接口及抽象层次对代码 跟踪很是个问题，因为跟踪代码经常遇到接口和抽象类，只能借助于工厂类和 API DOC，反复对照接口和实现类的对应关系。就像几乎任何优秀的 Java 开源项目都会用上一系列优秀的设计模式，也完全可以从模式这一点单独拿出一篇分析文章来，尽管我目前没有这样的想法。而在此文完成之后，我也没什么兴趣再看 Netty 的代码了。