

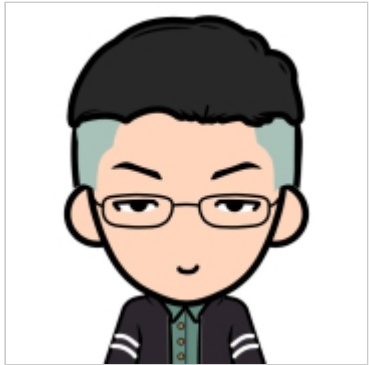


阿涵-_-的博客

<http://blog.sina.com.cn/jiangafu> [订阅] [手机订阅]

[首页](#) [博文目录](#) [图片](#) [关于我](#)

个人资料



阿涵-_-

微博

加好友

发纸条

写留言

加关注

正文

字体大小：大 中 小

Netty简介、架构、机制、特性

(2011-12-31 13:05:52)

转载 ▼

标签： 杂谈 分类： 资料

1. 简介

Netty 是一个异步的，事件驱动的网络编程框架和工具，使用Netty 可以快速开发出可维护的，高性能、高扩展能力的协议服务及其客户端应用。

也就是说，Netty 是一个基于NIO的客户，服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

“快速”和“简单”并不意味着会让你的最终应用产生维护性或性能上的问题。Netty 是一个吸收了多种协议的实现经验，这些协议包括FTP, SMPT, HTTP，各种二进制，文本协议，并经过相当精心设计的项目，最终，Netty 成功的找到了一种方式，在保证易于开发的同时还保证了其应用的性能，稳定性和伸缩性。

2. 总体架构



博客等级：**20**

博客积分：**1725**

博客访问：**1,235,374**

关注人气：**159**

获赠金笔：**70**

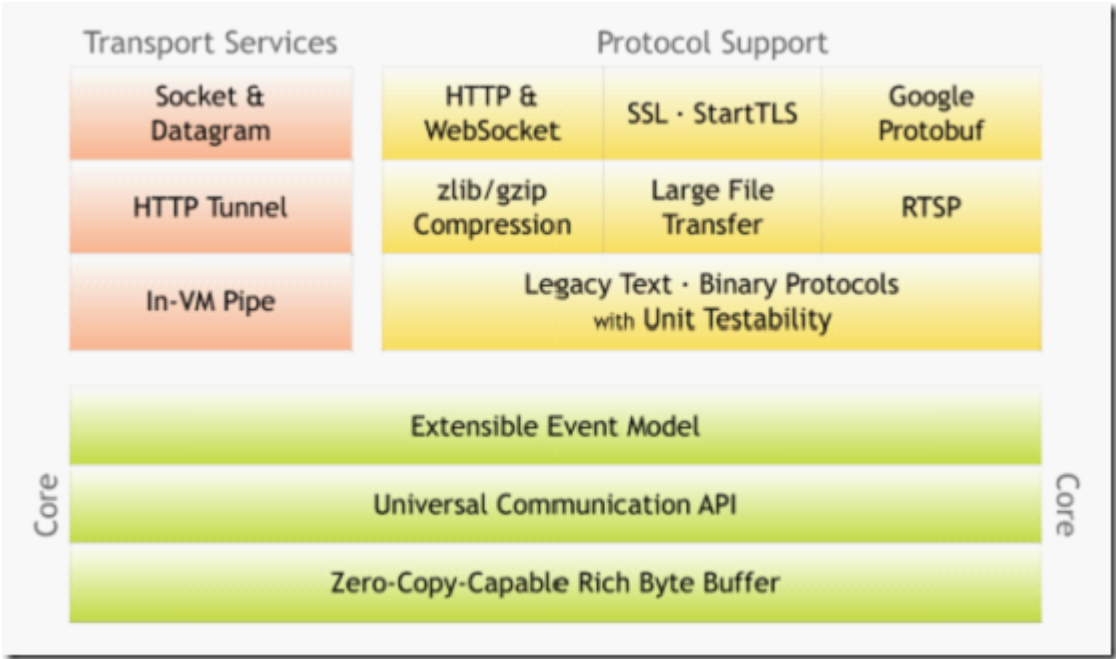
赠出金笔：**2**

荣誉徽章：

相关博文

女星透视装若隐若现朦胧美
娱娱玉玉

更多>>



2. 1. 丰富的缓冲数据结构

Netty使用自建的buffer API，而不是使用NIO的ByteBuffer来代表一个连续的字节序列。与ByteBuffer相比这种方式拥有明显的优势。Netty使用新的buffer类型ChannelBuffer，ChannelBuffer被设计为一个可从底层解决ByteBuffer问题，并可满足日常网络应用开发需要的缓冲类型。这些很酷的特性包括：

- 1 如果需要，允许自定义缓冲类型。
- 1 通过内置复合缓冲类型实现透明的零拷贝。
- 1 提供开箱即用的动态缓冲类型，其容量可以根据需要扩充，类似于StringBuffer。
- 1 不再需要调用的flip()方法。
- 1 一般来说比ByteBuffer更快。

关于Netty buffer更多的信息，参考：

http://docs.jboss.org/netty/3.2/api/org/jboss/netty/buffer/package-summary.html#package_description。

2. 2. 统一的异步 I/O API

传统的Java I/O API在应对不同的传输协议时需要使用不同的类型和方法。例如，java.net.Socket 和 java.net.DatagramSocket它们并不具有相同的超类型，因此，这就需要使用不同的调用方式执行socket操作。这种模式上的不匹配使得在更换一个网络应用的传输协议时变得繁杂和困难。由于（Java I/O API）缺乏协议间的移植性，当你试图在不修改网络传输层的前提下增加多种协议的支持，这时便会产生问题。并且理论上讲，多种应用层协议可运行在多种传输 层协议之上例如TCP/IP, UDP/IP, SCTP和串口通信。让这种情况变得更糟的是，Java新的I/O（NIO）API与原有的阻塞式的I/O（OIO）API并不兼容，NIO.2(AIO)也是如此。由于所有的API无论是在其设计上还是性能上的特性都与彼此不同，在进入开发阶段，你常常会被迫的选择一种你需要的API。

例如，在用户数较小的时候你可能会选择使用传统的OIO(Old I/O) API，毕竟与NIO相比使用OIO将更加容易一些。然而，当你的业务呈指数增长并且服务器需要同时处理成千上万的客户连接时你便会遇到问题。这种情况下你可能会尝试使用NIO，但是复杂的NIO Selector编程接口又会耗费你大量时间并最终会阻碍你的快速开发。Netty有一个叫做Channel的统一的异步I/O编程接口，这个编程接口抽象了所有点对点的通信操作。也就是说，如果你的应用是基于Netty的某一种传输实现，那么同样的，你的应用也可以运行在Netty的另一种传输实现上。Netty提供了几种拥有相同编程接口的基本传输实现：

- 1 NIO-based TCP/IP transport (See org.jboss.netty.channel.socket.nio),
- 1 OIO-based TCP/IP transport (See org.jboss.netty.channel.socket.oio),
- 1 OIO-based UDP/IP transport, and
- 1 Local transport (See org.jboss.netty.channel.local).

推荐博文

《常回家看看》歌手陈红用逝去母

穷人读书不一定有出息，但不读一

刘兴亮丨听说美团和饿了么联手干

8万元卖亲生女儿：被物化的生命

020下半场:团购已死，得数据

【解局】祁连山为何又上了环保头

苹果有什么权力让我们的手机变慢

程鹤麟：“家庭医生”不是“私人

哈佛大学被提起诉讼之后——亚裔



帕米尔高原上的
卖玉人



罕见的“火龙钢
花”



高冷范儿美女



冬天也光腿的
不怕冷美女



隐藏在深山中的
传奇修道院



建在第七大奇迹
原址上的。

[查看更多>>](#)

谁看过这篇博文

码农	1月16日
xiangyu85	7月20日
hzieedu	7月17日
colool	7月17日
jinjin	7月16日
Andyliu	7月16日
yangyunfeng	7月15日
北国之南	7月13日
小冰	7月13日
Hensen	7月13日
xiafish201	7月12日
你走了	7月12日

切换不同的传输实现通常只需对代码进行几行的修改调整，例如选择一个不同的ChannelFactory实现。此外，你甚至可以利用那些还没有写入的新的传输协议，只需替换一些构造器的调用方法即可，例如串口通信。而且由于核心API具有高度的可扩展性，你还可以完成自己的传输实现。

2.3. 基于拦截链模式的事件模型

一个定义良好并具有扩展能力的事件模型是事件驱动开发的必要条件。Netty具有定义良好的I/O事件模型。由于严格的层次结构区分了不同的事件类型，因此Netty也允许你在不破坏现有代码的情况下实现自己的事件类型。这是与其他框架相比另一个不同的地方。很多NIO框架没有或者仅有有限的事件模型概念；在你试图添加一个新的事件类型的时候常常需要修改已有的代码，或者根本就不允许你进行这种扩展。在一个ChannelPipeline内部一个ChannelEvent被一组ChannelHandler处理。这个管道是拦截过滤器模式的一种高级形式的实现，因此对于一个事件如何被处理以及管道内部处理器间的交互过程，你都将拥有绝对的控制力。例如，你可以定义一个从socket读取到数据后的操作：

Java代码

```
public class MyReadHandler implements SimpleChannelHandler {  
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent evt) {  
        Object message = evt.getMessage();  
        // Do something with the received message.  
        ...  
        // And forward the event to the next handler.  
        ctx.sendUpstream(evt);  
    }  
}
```

同时你也可以定义当处理器接收入到写请求时的动作：

Java代码

```
public class MyWriteHandler implements SimpleChannelHandler {  
    public void writeRequested(ChannelHandlerContext ctx, MessageEvent evt) {  
        Object message = evt.getMessage();  
        // Do something with the message to be written.  
        ...  
        // And forward the event to the next handler.  
        ctx.sendDownstream(evt);  
    }  
}
```

有关事件模型的更多信息，参考API文档中的ChannelEvent 和ChannelPipeline部分。

2.4. 适用快速开发的高级组件

上述所提及的核心组件已经足够实现各种类型的网络应用，除此之外，Netty也提供了一系列的高级组件来加速你的开发过程。

2.4.1. Codec框架

从业务逻辑代码中分离协议处理部分总是一个很不错的想法。然而如果一切从零开始便会遭遇 到实现上的复杂性。你不得不处理分段的消息。一些协议是多层的（例如构建在其他低层协议之上的协议）。一些协议过于复杂以致难以在一台主机（single state machine）上实现。

因此，一个好的网络应用框架应该提供一种可扩展，可重用，可单元测试并且是多层的codec框架，为用户提供易维护的codec代码。

Netty提供了一组构建在其核心模块之上的codec实现，这些简单的或者高级的codec实现帮你解决了大部分在你进行协议处理开发过程会遇到的问题，无论这些协议是简单的还是复杂的，二进制的或是简单文本的。

2.4.2. SSL / TLS 支持

不同于传统阻塞式的I/O实现，在NIO模式下支持SSL功能是一个艰难的工作。你不能只是简单的包装一下流数据并进行加密或解密工作，你不得不借助于 javax.net.ssl.SSLEngine，SSLEngine是一个有状态的实现，其复杂性不亚于SSL自身。你必须管理所有可能的状态，例如密码套件，密钥协商（或重新协商），证书交换以及认证等。此外，与通常期望情况相反的是SSLEngine甚至不是一个绝对的线程安全实现。

在Netty内部，SslHandler 封装了所有艰难的细节以及使用SSLEngine可能带来的陷阱。你所做的仅是配置并将该SslHandler插入到你的ChannelPipeline中。同样Netty也允许你实现像StartTLS 那样所拥有的高级特性，这很容易。

2.4.3. HTTP实现

HTTP无疑是互联网上最受欢迎的协议，并且已经有了一些例如Servlet容器这样的HTTP实现。因此，为什么Netty还要在其核心模块之上构建一套HTTP实现？

与现有的HTTP实现相比Netty的HTTP实现是相当与众不同的。在HTTP消息的低层交互过程中你将拥有绝对的控制力。这是因为Netty的 HTTP实现只是一些HTTP codec和HTTP消息类的简单组合，这里不存在任何限制——例如那种被迫选择的线程模型。你可以随心所欲的编写那种可以完全按照你期望的工作方式工作的客户端或服务端代码。这包括线程模型，连接生命期，快编码，以及所有HTTP协议允许你做的，所有的一切，你都将拥有绝对的控制力。

由于这种高度可定制化的特性，你可以开发一个非常高效的HTTP服务器，例如：

- 1 要求持久化链接以及服务器端推送技术的聊天服务（e.g. Comet ）
- 1 需要保持链接直至整个文件下载完成的媒体流服务（e.g. 2小时长的电影）
- 1 需要上传大文件并且没有内存压力的文件服务（e.g. 上传1GB文件的请求）
- 1 支持大规模mash-up应用以及数以万计连接的第三方web services异步处理平台

2.4.4. Google Protocol Buffer 整合

Google Protocol Buffers 是快速实现一个高效的二进制协议的理想方案。通过使用 ProtobufEncoder 和 ProtobufDecoder，你可以把Google Protocol Buffers 编译器（protoc）生成的消息类放入到Netty的codec实现中。请参考官方示例中的“LocalTime” 示例，这个例子也同时显示出开发一个由简单协议定义的客户及服务端是多么的容易。

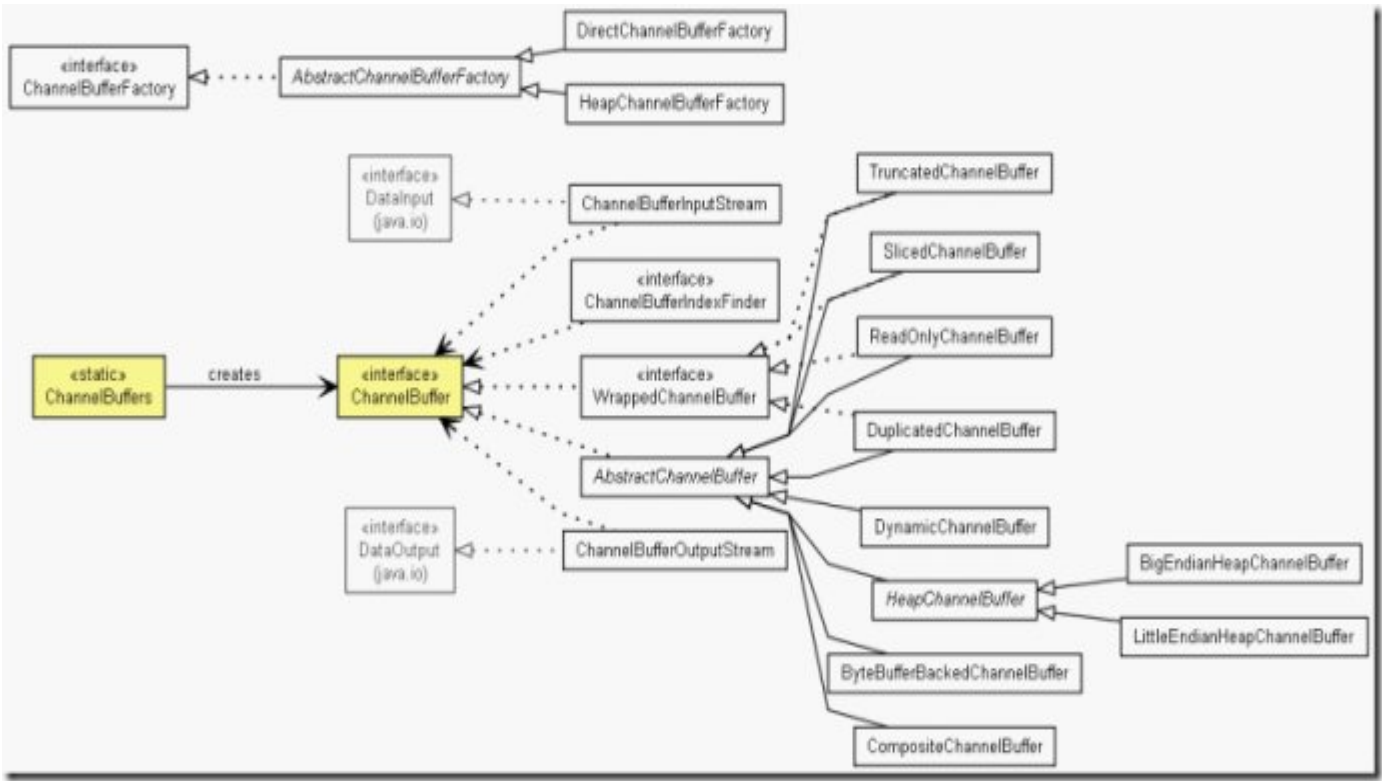
2.5. 总述

在这一章节，我们从功能特性的角度回顾了Netty的整体架构。Netty有一个简单却不失强大的架构。这个架构由三部分组成——缓冲（buffer）， 通道（channel），事件模型（event model）——所有的高级特性都构建在这三个核心组件之上。一旦你理解了它们之间的工作原理，你便不难理解在本章简要提及的更多高级特性。

3. 比较重要的几个特性

3.1. 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。根据网络字节序的不同，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。

1 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。

1 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。

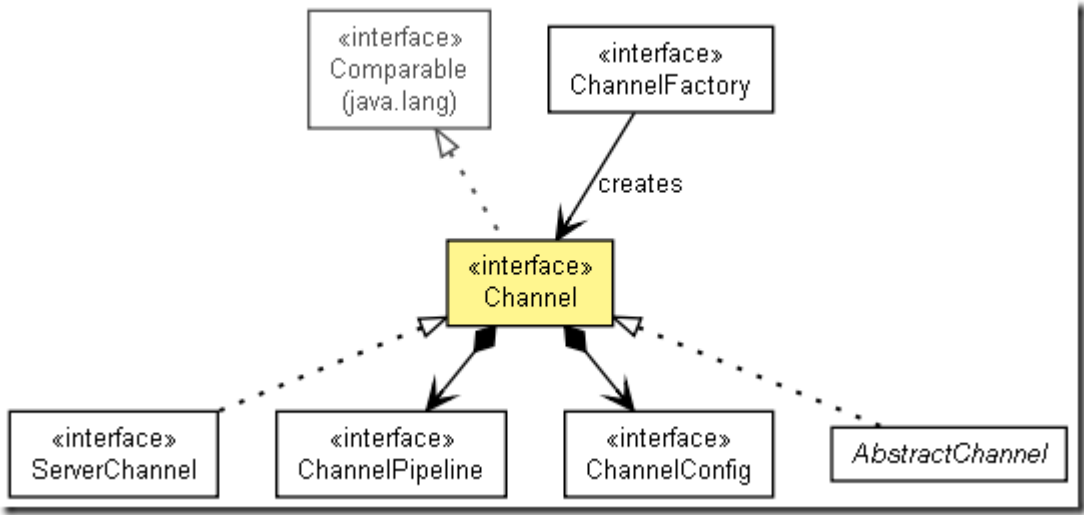
1 针对固定大小的buffer缺点，就引入动态buffer。

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。

3.2. Channel

和Channel相关的接口及类结构图如下：



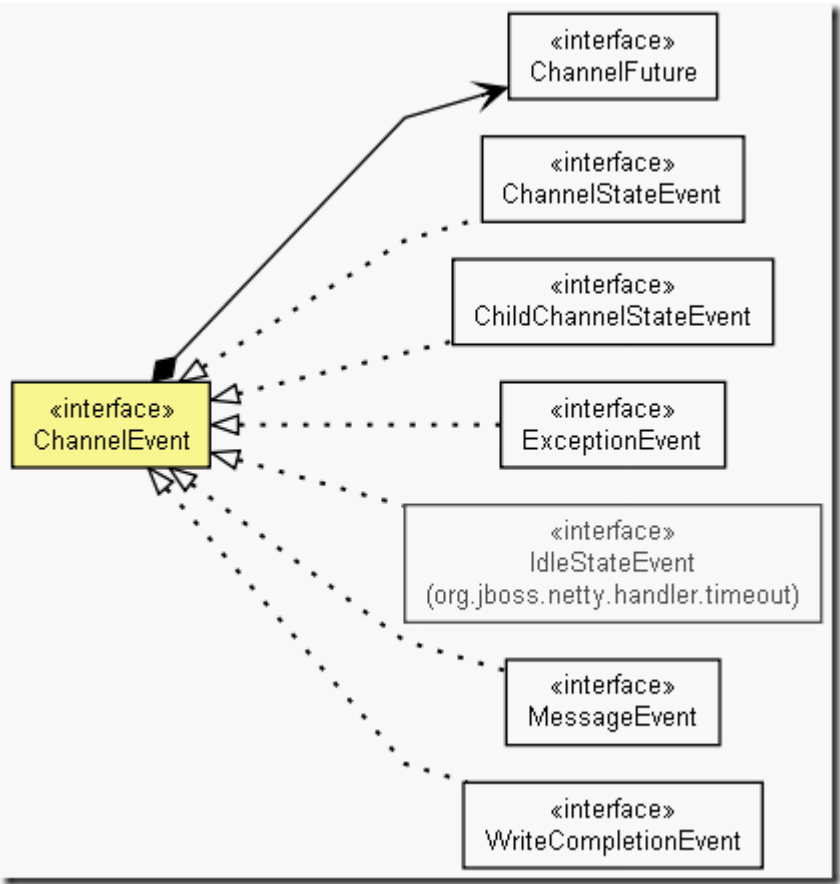
从该结构图也可以看到，Channel主要提供的功能如下：

- 1 当前Channel的状态信息，比如是打开还是关闭等。
- 1 通过ChannelConfig可以得到的Channel配置信息。
- 1 Channel所支持的如read、write、bind、connect等IO操作。
- 1 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。

3.3. 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将被ChannelPipeline的一系列ChannelHandler处理。下面是和 ChannelEvent相关的接口及类图：



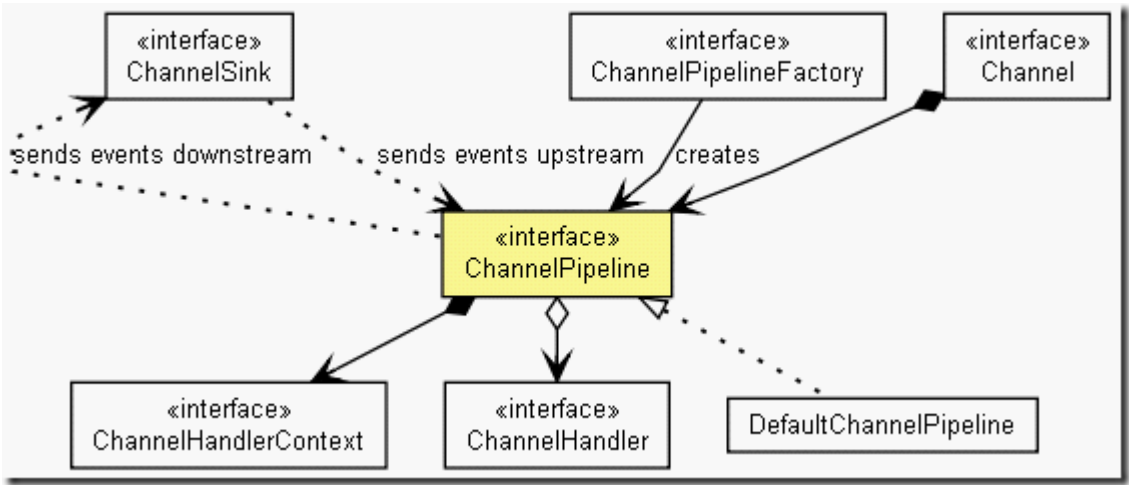
事件流有两种，upstream event和downstream event。当服务器端接收到来自客户端的消息时，这个事件连带着消息就是一个“upstream event”，当服务器端发送消息或响应给客户端时，这个事件连带着相关的写请求就是

一个“downstream event”。注意，“upstream event”被ChannelPipeline中的ChannelHandler处理的顺序是从第一个到最后一个，而“downstream event”正好相反。

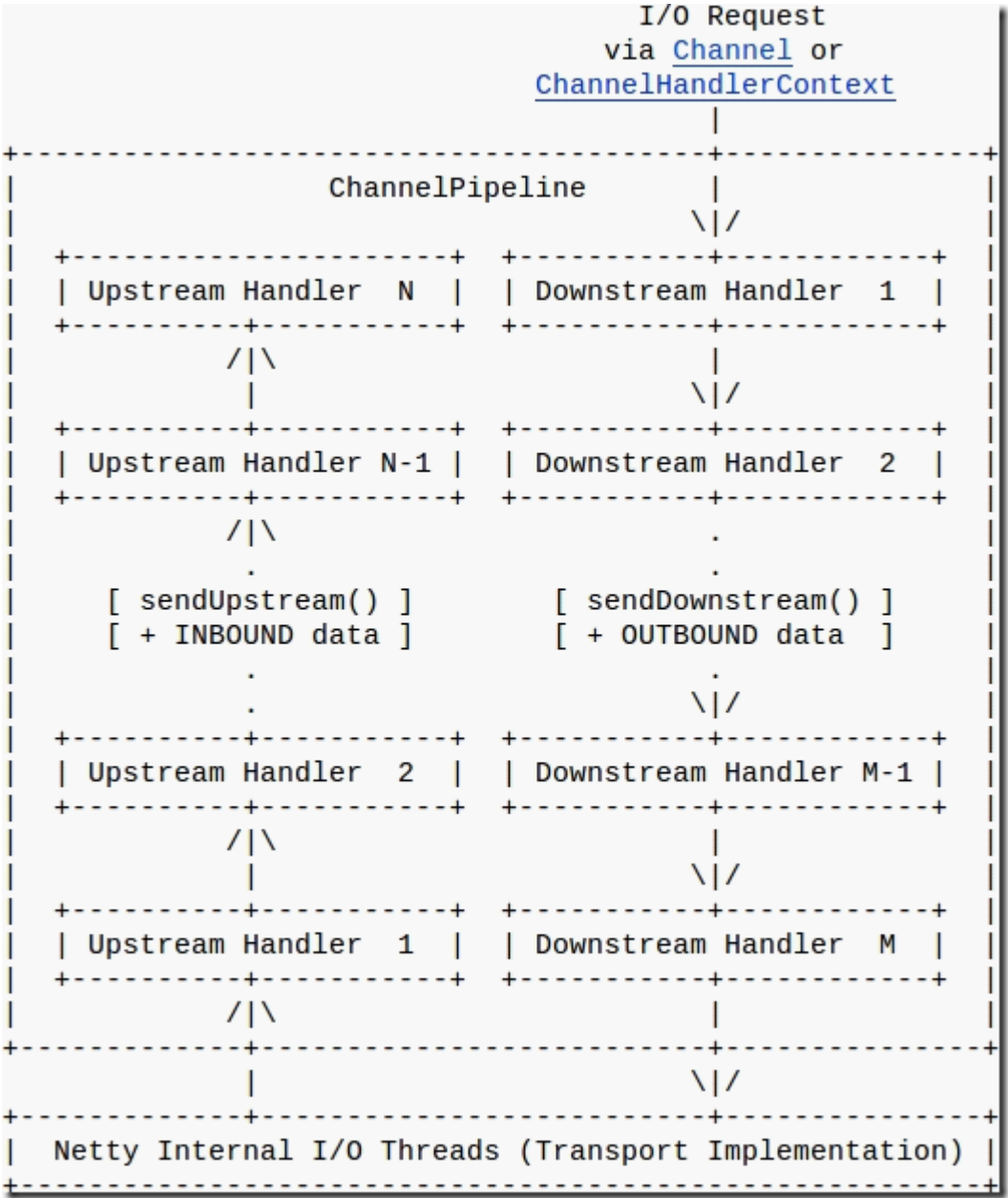
对于使用者来说，在ChannelHandler实现类中会使用继承于ChannelEvent的MessageEvent，调用其getMessage()方法来获得读到的ChannelBuffer或被转化的对象。

3.4. 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：



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。

举例来说，假设我们创建了以下的管道：

```
ChannelPipeline p = Channels.pipeline();
p.addLast("1", new UpstreamHandlerA());
p.addLast("2", new UpstreamHandlerB());
p.addLast("3", new DownstreamHandlerA());
p.addLast("4", new DownstreamHandlerB());
p.addLast("5", new UpstreamHandlerX());
```

其中类名以Upstream开头的是一个upstream handler，类名以Downstream开头的是一个downstream handler。当一个upstream event发生的时候，处理的顺序应该是1，2，3，4，5，而当一个downstream event发生的时候，处理的顺序应该是5，4，3，2，1，在这个原则之上，ChannelPipeline会跳过一些handler：

- 1 3和4号handler没有实现ChannelUpstreamHandler，因此实际上upstream event 被处理的顺序为：1，2，5。
- 1 1，2和5号handler没有实现ChannelDownstreamHandler，因此实现上downstream event 被处理的顺序为4，3。

1 如果5号handler继承了SimpleChannelHandler（SimpleChannelHandler实现了ChannelUpstreamHandler 和 ChannelDownstreamHandler接口），那么upstream event和downstream event被处理的顺序就应该是125和543。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。

3.5. 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。

1 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。

1 ObjectEncoder 和ObjectDecoder：编码解码序列化的Java对象。

1 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() < 4) {
return null;
}
buf.markReaderIndex();
int length = buf.readInt();
if (buf.readableBytes() < 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。

4. 源代码分析

4.1. org.jboss.netty.bootstrap

提供了一些辅助类用于实现典型的服务器和客户端初始化。

4.2. org.jboss.netty.buffer

提供Netty中的byte buffer的抽象和实现。

4.3. org.jboss.netty.channel

channel核心API，包括异步和事件驱动等各种传送接口。

4.4. org.jboss.netty.channel.group

channel group，帮助用户维护channel列表。

4.5. org.jboss.netty.local

一种虚拟传输方式，允许同一个虚拟机上的两个部分可以互相通信。

4.6. org.jboss.netty.socket

TCP、UDP接口，继承了核心的channel API。

4.7. org.jboss.netty.socket.nio

基于nio的socket channel实现

4.8. org.jboss.netty.socket.oio

基于老io的socket channel实现

4.9. org.jboss.netty.socket.http

基于http的客户端和相应的server端的实现，工作在有防火墙的情况

4.10. org.jboss.netty.container

各种容器的兼容

4.11. org.jboss.netty.container.microcontainer

JBoss Microcontainer集成接口

4.12. org.jboss.netty.container.osgi

OSGi framework集成接口

4.13. org.jboss.netty.container.spring

Spring framework集成接口

4.14. org.jboss.netty.handler

处理器

4.15. org.jboss.netty.handler.codec

编码解码器

4.16. org.jboss.netty.handler.execution

基于Executor的实现

4.17. org.jboss.netty.handler.queue

将event存入内部队列的处理

4.18. org.jboss.netty.handler.ssl

基于SSLEngine的SSL以及TLS实现

4.19. org.jboss.netty.handler.stream

异步写入大数据，不会产生OutOfMemory也不会花费很多内存

4.20. `org.jboss.netty.handler.timeout`

通过Timer来对读写超时或者闲置链接进行通知

4.21. `org.jboss.netty.handler.codec.base64`

Base64编码

4.22. `org.jboss.netty.handler.codec.compression`

压缩格式

4.23. `org.jboss.netty.handler.codec.embedder`

嵌入模式下编码和解码

4.24. `org.jboss.netty.handler.codec.frame`

评估流的数据的排列和内容

4.25. `org.jboss.netty.handler.codec.http.websocket`

websocket编码和解码

4.26. `org.jboss.netty.handler.codec.http`

http的编码解码以及类型信息

4.27. `org.jboss.netty.handler.codec.oneone`

对象到对象编码解码

4.28. `org.jboss.netty.handler.codec.protobuf`

Protocol Buffers的编码解码

4.29. `org.jboss.netty.handler.codec.replay`

在阻塞io中实现非阻塞解码

4.30. `org.jboss.netty.handler.codec.rtsp`

RTSP的编码解码

4.31. `org.jboss.netty.handler.codec.serialization`

序列化对象到bytebuffer实现

4.32. `org.jboss.netty.handler.codec.string`

字符串编码解码，继承oneone

4.33. `org.jboss.netty.logging`

根据不同的log framework实现的类

4.34. `org.jboss.netty.util`

Netty util类

4.35. `org.jboss.netty.internal`

Netty 内部util类，不被外部使用

5. 参考资料

Netty官网：<http://www.jboss.org/netty>

Netty文档及范例：<http://www.jboss.org/netty/documentation>

9

喜欢

2

赠金笔

分享：

阅读(11988) | 评论 (0) | 收藏(2) | 转载(2) | 喜欢▼ | 打印 | 举报

已投稿到： 排行榜

上一篇：Kafka Zookeeper ZkTimeoutException解决方法

后一篇：Linux部署安装zookeeper集群







评论





重要提示：警惕虚假中奖信息





[发评论]

做第一个评论者吧！ 抢沙发>>

发评论

更多>>





登录名： 密码： [找回密码](#) [注册](#) ☒ 记住登录状态

☐ 评论并转载此博文

发评论

以上网友发言只代表其个人观点，不代表新浪网的观点或立场。

< 上一篇

Kafka Zookeeper ZkTimeoutException解决方法

后一篇 >

Linux部署安装zookeeper集群

[新浪BLOG意见反馈留言板](#) [不良信息反馈](#) [电话：4006900000](#) [提示音后按1键（按当地市话标准计费）](#) [欢迎批评指正](#)

[新浪简介](#) | [About Sina](#) | [广告服务](#) | [联系我们](#) | [招聘信息](#) | [网站律师](#) | [SINA English](#) | [会员注册](#) | [产品答疑](#)

Copyright © 1996 - 2017 SINA Corporation, All Rights Reserved

新浪公司 版权所有