参考：<http://www.importnew.com/15656.html>

Netty实现原理浅析

2015/04/03 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [2 条评论](http://www.importnew.com/15656.html#comments) | 标签： [NETTY](http://www.importnew.com/tag/netty)

分享到：**69**

原文出处： [kafka0102的博客](http://www.kafka0102.com/2010/06/167.html)

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](http://www.kafka0102.com/2010/06/netty%E4%BD%BF%E7%94%A8%E5%88%9D%E6%AD%A5/)。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](http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf)”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：

|  |  |
| --- | --- |
| 1  2  3  4 | 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：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)，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。