Netty 3.x源码解析

引言

Netty是Java世界知名的网络应用框架。本系列文章是Netty的源码导读。

为什么要读Netty源码？

我认为，一般研究Netty源码出于两个原因：

1. 日常工作中使用到Netty，想要进一步了解；
2. 对Java网络编程感兴趣，想知道如何构建一个高性能网络应用。

另外，Netty的代码组织比较优秀，从中可以学到代码结构组织的方法。

这些文章讲什么？

本系列文章的介绍点包括：Netty的设计思想，网络编程的领域知识，以及Netty代码结构的骨干，可能也会包括一些具体场景的应用以及一些特性的分析。

因为写作时间跨度较大，可能存在上下章节不连贯的情况，敬请谅解。

建议和反馈

这系列文章是在github上更新的。作者水平有限，如果有错误欢迎纠正，能给我发pull request是最好了！

github地址：https://github.com/code4craft/netty-learning

一、概述

起：Netty是什么

大概用Netty的，无论新手还是老手，都知道它是一个“网络通讯框架”。所谓框架，基本上都是一个作用：基于底层API，提供更便捷的编程模型。那么”通讯框架”到底做了什么事情呢？回答这个问题并不太容易，我们不妨反过来看看，不使用Netty，直接基于NIO编写网络程序，你需要做什么(以Server端TCP连接为例，这里我们使用Reactor模型)：

1. 监听端口，建立Socket连接
2. 建立线程，处理内容
    1. 读取Socket内容，并对协议进行解析
    2. 进行逻辑处理
    3. 回写响应内容
    4. 如果是多次交互的应用(SMTP、FTP)，则需要保持连接多进行几次交互
3. 关闭连接

建立线程是一个比较耗时的操作，同时维护线程本身也有一些开销，所以我们会需要多线程机制，幸好JDK已经有很方便的多线程框架了，这里我们不需要花很多心思。

此外，因为TCP连接的特性，我们还要使用连接池来进行管理：

1. 建立TCP连接是比较耗时的操作，对于频繁的通讯，保持连接效果更好
2. 对于并发请求，可能需要建立多个连接
3. 维护多个连接后，每次通讯，需要选择某一可用连接
4. 连接超时和关闭机制

想想就觉得很复杂了！实际上，基于NIO直接实现这部分东西，即使是老手也容易出现错误，而使用Netty之后，你只需要关注逻辑处理部分就可以了。

承：体验Netty

这里我们引用Netty的example包里的一个例子，一个简单的EchoServer，它接受客户端输入，并将输入原样返回。其主要代码如下：

```
    public void run() {
        // Configure the server.
        ServerBootstrap bootstrap = new ServerBootstrap(
                new NioServerSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        // Set up the pipeline factory.
```

```
            bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
                public ChannelPipeline getPipeline() throws Exception {
                    return Channels.pipeline(new EchoServerHandler());
                }
            });

            // Bind and start to accept incoming connections.
            bootstrap.bind(new InetSocketAddress(port));
        }
```

这里`EchoServerHandler`是其业务逻辑的实现者，大致代码如下：

```
    public class EchoServerHandler extends SimpleChannelUpstreamHandler {

        @Override
        public void messageReceived(
                ChannelHandlerContext ctx, MessageEvent e) {
            // Send back the received message to the remote peer.
            e.getChannel().write(e.getMessage());
        }
    }
```

还是挺简单的，不是吗？

转：Netty背后的事件驱动机制

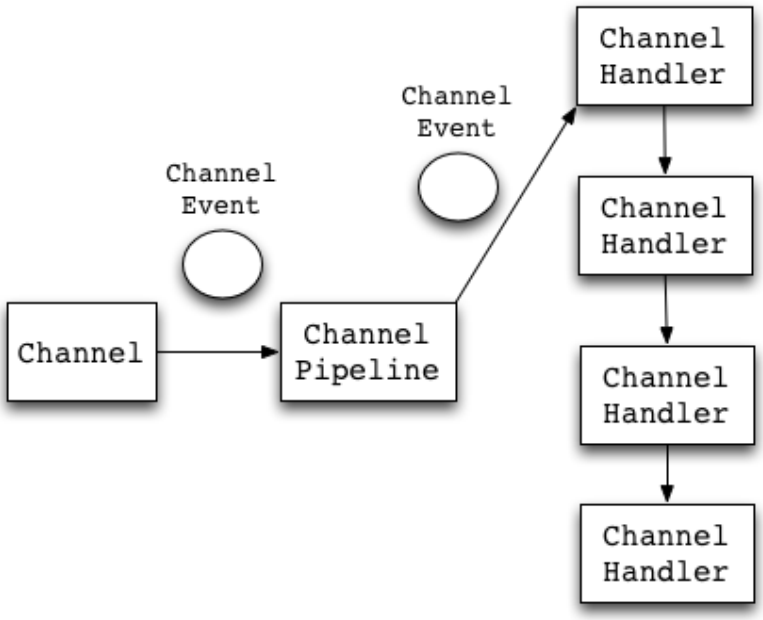
完成了以上一段代码，我们算是与Netty进行了第一次亲密接触。如果想深入学习呢？

阅读源码是了解一个开源工具非常好的手段，但是Java世界的框架大多追求大而全，功能完备，如果逐个阅读，难免迷失方向，Netty也并不例外。相反，抓住几个重点对象，理解其领域概念及设计思想，从而理清其脉络，相当于打通了任督二脉，以后的阅读就不再困难了。

理解Netty的关键点在哪呢？我觉得，除了NIO的相关知识，另一个就是事件驱动的设计思想。什么叫事件驱动？我们回头看看`EchoServerHandler`的代码，其中的参数：`public void messageReceived(ChannelHandlerContext ctx, MessageEvent e)`，MessageEvent就是一个事件。这个事件携带了一些信息，例如这里`e.getMessage()`就是消息的内容，而`EchoServerHandler`则描述了处理这种事件的方式。一旦某个事件触发，相应的Handler则会被调用，并进行处理。这种事件机制在UI编程里广泛应用，而Netty则将其应用到了网络编程领域。

在Netty里，所有事件都来自`ChannelEvent`接口，这些事件涵盖监听端口、建立连接、读写数据等网络通讯的各个阶段。而事件的处理者就是`ChannelHandler`，这样，不但是业务逻辑，连网络通讯流程中底层的处理，都可以通过实现`ChannelHandler`来完成了。事实上，Netty内部的连接处理、协议编解码、超时等机制，都是通过handler完成的。当博主弄明白其中的奥妙时，不得不佩服这种设计！

下图描述了Netty进行事件处理的流程。`Channel`是连接的通道，是ChannelEvent的产生者，而`ChannelPipeline`可以理解为ChannelHandler的集合。

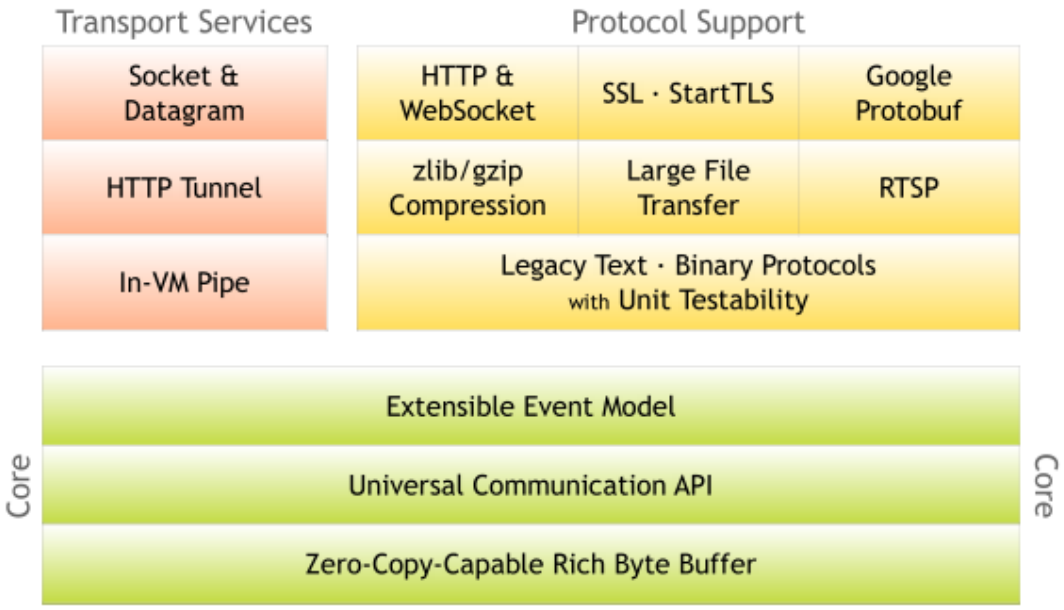


合：开启Netty源码之门

理解了Netty的事件驱动机制，我们现在可以来研究Netty的各个模块了。Netty的包结构如下：

```
org
└── jboss
    └── netty
        ├── bootstrap 配置并启动服务的类
        ├── buffer 缓冲相关类，对NIO Buffer做了一些封装
        ├── channel 核心部分，处理连接
        ├── container 连接其他容器的代码
        ├── example 使用示例
        ├── handler 基于handler的扩展部分，实现协议编解码等附加功能
        ├── logging 日志
        └── util 工具类
```

在这里面，`channel`和`handler`两部分比较复杂。我们不妨与Netty官方的结构图对照一下，来了解其功能。



具体的解释可以看这里：http://netty.io/3.7/guide/#architecture。图中可以看到，除了之前说到的事件驱动机制之外，Netty的核心功能还包括两部分：

- Zero-Copy-Capable Rich Byte Buffer

  零拷贝的Buffer。为什么叫零拷贝？因为在数据传输时，最终处理的数据会需要对单个传输层的报文，进行组合或者拆分。NIO原生的ByteBuffer无法做到这件事，而Netty通过提供Composite(组合)和Slice(切分)两种Buffer来实现零拷贝。这部分代码在 `org.jboss.netty.buffer` 包中。
  这里需要额外注意，不要和操作系统级别的Zero-Copy混淆了，操作系统中的零拷贝主要是用户空间和内核空间之间的数据拷贝，NIO中通过DirectBuffer做了实现.

- Universal Communication API

  统一的通讯API。这个是针对Java的Old I/O和New I/O，使用了不同的API而言。Netty则提供了统一的API(`org.jboss.netty.channel.Channel`)来封装这两种I/O模型。这部分代码在 `org.jboss.netty.channel` 包中。

此外，Protocol Support功能通过handler机制实现。

接下来的文章，我们会根据模块，详细的对Netty源码进行分析。

参考资料：

- Netty 3.7 User Guide http://netty.io/3.7/guide/
- What is Netty? http://ayedo.github.io/netty/2013/06/19/what-is-netty.html

二、Netty中的buffer

上一篇文章我们概要介绍了Netty的原理及结构，下面几篇文章我们开始对Netty的各个模块进行比较详细的分析。Netty的结构最底层是buffer机制，这部分也相对独立，我们就先从buffer讲起。

What：buffer二三事

buffer中文名又叫缓冲区，按照维基百科的解释，是”在数据传输时，在内存里开辟的一块临时保存数据的区域”。它其实是一种化同步为异步的机制，可以解决数据传输的速率不对等以及不稳定的问题。

根据这个定义，我们可以知道涉及I/O(特别是I/O写)的地方，基本会有Buffer了。就Java来说，我们非常熟悉的Old I/O– `InputStream` & `OutputStream` 系列API，基本都是在内部使用到了buffer。Java课程老师就教过，必须调用 `OutputStream.flush()` ，才能保证数据写入生效！

而NIO中则直接将buffer这个概念封装成了对象，其中最常用的大概是ByteBuffer了。于是使用方式变为了：将数据写入Buffer，flip()一下，然后将数据读出来。于是，buffer的概念更加深入人心了！

Netty中的buffer也不例外。不同的是，Netty的buffer专为网络通讯而生，所以它又叫ChannelBuffer(好吧其实没有什么因果关系…)。我们下面就来讲讲Netty中得buffer。当然，关于Netty，我们必须讲讲它的所谓”Zero-Copy-Capable”机制。

When & Where：TCP/IP协议与buffer

TCP/IP协议是目前的主流网络协议。它是一个多层协议，最下层是物理层，最上层是应用层(HTTP协议等)，而做Java应用开发，一般只接触TCP以上，即传输层和应用层的内容。这也是Netty的主要应用场景。

TCP报文有个比较大的特点，就是它传输的时候，会先把应用层的数据项拆开成字节，然后按照自己的传输需要，选择合适数量的字节进行传输。什么叫”自己的传输需要”？首先TCP包有最大长度限制，那么太大的数据项肯定是要拆开的。其次因为TCP以及下层协议会附加一些协议头信息，如果数据项太小，那么可能报文大部分都是没有价值的头信息，这样传输是很不划算的。因此有了收集一定数量的小数据，并打包传输的Nagle算法(这个东东在HTTP协议里会很讨厌，Netty里可以用setOption(“tcpNoDelay”, true)关掉它)。

这么说可能太学院派了一点，我们举个例子吧：

发送时，我们这样分3次写入(‘|’表示两个buffer的分隔):

```
    +-----+-----+-----+
```

```
| ABC | DEF | GHI |
+-----+-----+-----+
```

接收时，可能变成了这样:

```
+----+-------+---+---+
| AB | CDEFG | H | I |
+----+-------+---+---+
```

很好懂吧？可是，说了这么多，跟buffer有个什么关系呢？别急，我们来看下面一部分。

Why：Buffer中的分层思想

我们先回到之前的`messageReceived`方法：

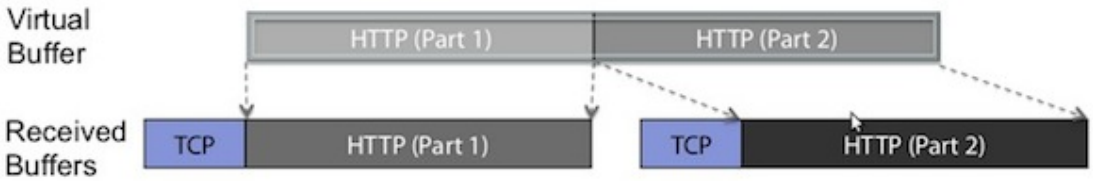
```
    public void messageReceived(
            ChannelHandlerContext ctx, MessageEvent e) {
        // Send back the received message to the remote peer.
        transferredBytes.addAndGet(((ChannelBuffer) e.getMessage()).readable
Bytes());
        e.getChannel().write(e.getMessage());
    }
```

这里`MessageEvent.getMessage()`默认的返回值是一个`ChannelBuffer`。我们知道，业务中需要的”Message”，其实是一条应用层级别的完整消息，而一般的buffer工作在传输层，与”Message”是不能对应上的。那么这个ChannelBuffer是什么呢？

来一个官方给的图，我想这个答案就很明显了：

```
requestPart1 = buffer1.slice(OFFSET_PAYLOAD,
        buffer1.readableBytes() - OFFSET_PAYLOAD);
requestPart2 = buffer2.slice(OFFSET_PAYLOAD,
        buffer2.readableBytes() - OFFSET_PAYLOAD);

request = ChannelBuffers.wrappedBuffer(requestPart1, requestPart2);
```



这里可以看到，TCP层HTTP报文被分成了两个ChannelBuffer，这两个Buffer对我们上层的逻辑(HTTP处理)是没有意义的。但是两个ChannelBuffer被组合起来，就成为了一个有意义的HTTP报文，这个报文对应的ChannelBuffer，才是能称之为”Message”的东西。这里用到了一个词”Virtual Buffer”，也就是所谓的”Zero-Copy-Capable Byte Buffer”了。顿时觉得豁然开朗了有没有！

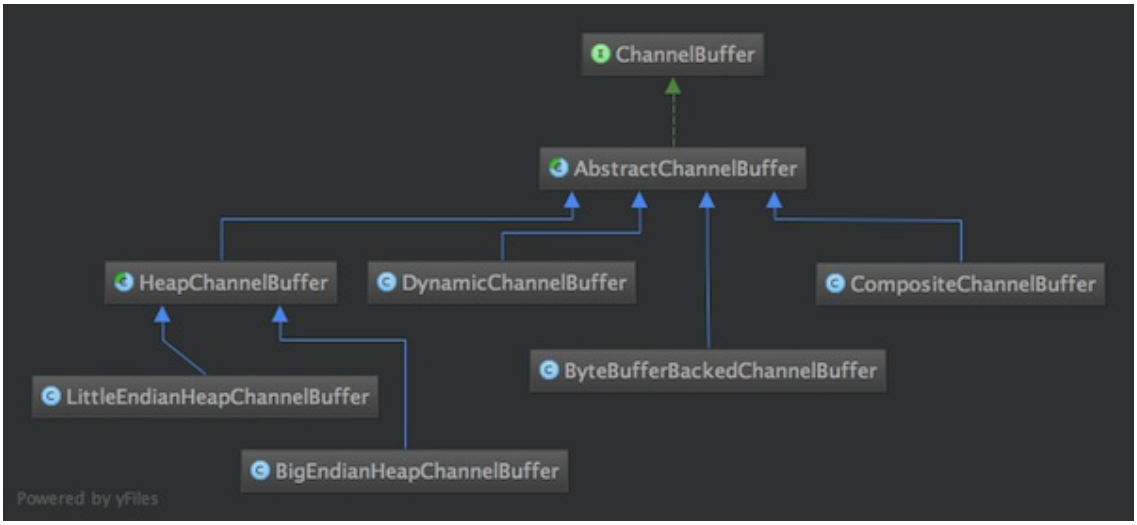
我这里总结一下，**如果说NIO的Buffer和Netty的ChannelBuffer最大的区别的话，就是前者仅仅是传输上的Buffer，而后者其实是传输Buffer和抽象后的逻辑Buffer的结合。**延伸开来说，NIO仅仅是一个网络传输框架，而Netty是一个网络应用框架，包括网络以及应用的分层结构。

当然，在Netty里，默认使用`ChannelBuffer`表示"Message"，不失为一个比较实用的方法，但是`MessageEvent.getMessage()`是可以存放一个POJO的，这样子抽象程度又高了一些，这个我们在以后讲到`ChannelPipeline`的时候会说到。

How：Netty中的ChannelBuffer及实现

好了，终于来到了代码实现部分。之所以嗦了这么多，因为我觉得，关于"Zero-Copy-Capable Rich Byte Buffer"，理解为什么需要它，比理解它是怎么实现的，可能要更重要一点。

我想可能很多朋友跟我一样，喜欢"顺藤摸瓜"式读代码–找到一个入口，然后顺着查看它的调用，直到理解清楚。很幸运，`ChannelBuffers`(注意有s!)就是这样一根"藤"，它是所有ChannelBuffer实现类的入口，它提供了很多静态的工具方法来创建不同的Buffer，靠"顺藤摸瓜"式读代码方式，大致能把各种ChannelBuffer的实现类摸个遍。先列一下ChannelBuffer相关类图。



此外还有`WrappedChannelBuffer`系列也是继承自`AbstractChannelBuffer`，图放到了后面。

ChannelBuffer中的readerIndex和writerIndex

开始以为Netty的ChannelBuffer是对NIO ByteBuffer的一个封装，其实不是的，**它是把ByteBuffer重新实现了一遍。**

以最常用的`HeapChannelBuffer`为例，其底层也是一个byte[]，与ByteBuffer不同的是，它是可以同时进行读和写的，而不需要使用flip()进行读写切换。ChannelBuffer读写的核心代码在`AbstactChannelBuffer`里，这里通过readerIndex和writerIndex两个整数，分别指向当前读的位置和当前写的位置，并且，readerIndex总是小于writerIndex的。贴两段代码，让大家能看的更明白一点：

```
    public void writeByte(int value) {
        setByte(writerIndex ++, value);
    }

    public byte readByte() {
        if (readerIndex == writerIndex) {
            throw new IndexOutOfBoundsException("Readable byte limit exceede
d: "
                    + readerIndex);
        }
        return getByte(readerIndex ++);
    }

    public int writableBytes() {
        return capacity() - writerIndex;
    }

    public int readableBytes() {
        return writerIndex - readerIndex;
    }
```

我倒是觉得这样的方式非常自然，比单指针与flip()要更加好理解一些。
AbstactChannelBuffer还有两个相应的mark指
针`markedReaderIndex`和`markedWriterIndex`，跟NIO的原理是一样的，这里不再赘述了。

字节序Endianness与HeapChannelBuffer

在创建Buffer时，我们注意到了这样一个方
法：`public static ChannelBuffer buffer(ByteOrder endianness, int capacity);`，其中`ByteOrder`是什么意思呢？

这里有个很基础的概念：字节序(ByteOrder/Endianness)。它规定了多余一个字节的数字(int啊long什么的)，如何在内存中表示。BIG_ENDIAN(大端序)表示高位在前，整型数`12`会被存储为`0 0 0 12`四字节，而LITTLE_ENDIAN则正好相反。可能搞C/C++的程序员对这个会比较熟悉，而Javaer则比较陌生一点，因为Java已经把内存给管理好了。但是在网络编程方面，根据协议的不同，不同的字节序也可能会被用到。目前大部分协议还是采用大端序，可参考RFC1700。

了解了这些知识，我们也很容易就知道为什么会
有`BigEndianHeapChannelBuffer`和`LittleEndianHeapChannelBuffer`了！

DynamicChannelBuffer

DynamicChannelBuffer是一个很方便的Buffer，之所以叫Dynamic是因为它的长度会根据内容的长度来扩充，你可以像使用ArrayList一样，无须关心其容量。实现自动扩容的核心在于`ensureWritableBytes`方法，算法很简单：在写入前做容量检查，容量不够时，新建一个容量x2的buffer，跟ArrayList的扩容是相同的。贴一段代码吧(为了代码易懂，这里我删掉了一些边界检查，只保留主逻辑)：

```
    public void writeByte(int value) {
        ensureWritableBytes(1);
        super.writeByte(value);
    }

    public void ensureWritableBytes(int minWritableBytes) {
        if (minWritableBytes <= writableBytes()) {
            return;
        }

        int newCapacity = capacity();
        int minNewCapacity = writerIndex() + minWritableBytes;
        while (newCapacity < minNewCapacity) {
            newCapacity <<= 1;
        }

        ChannelBuffer newBuffer = factory().getBuffer(order(), newCapacity);
        newBuffer.writeBytes(buffer, 0, writerIndex());
        buffer = newBuffer;
    }
```

CompositeChannelBuffer

`CompositeChannelBuffer`是由多个ChannelBuffer组合而成的，可以看做一个整体进行读写。这里有一个技巧：CompositeChannelBuffer并不会开辟新的内存并直接复制所有ChannelBuffer内容，而是直接保存了所有ChannelBuffer的引用，并在子ChannelBuffer里进行读写，从而实现了”Zero-Copy-Capable”了。来段简略版的代码吧：

```
    public class CompositeChannelBuffer{

        //components保存所有内部ChannelBuffer
        private ChannelBuffer[] components;
        //indices记录在整个CompositeChannelBuffer中，每个components的起始位置
        private int[] indices;
        //缓存上一次读写的componentId
        private int lastAccessedComponentId;

        public byte getByte(int index) {
            //通过indices中记录的位置索引到对应第几个子Buffer
            int componentId = componentId(index);
            return components[componentId].getByte(index - indices[component
Id]);
        }

        public void setByte(int index, int value) {
            int componentId = componentId(index);
            components[componentId].setByte(index - indices[componentId], va
lue);
        }

    }
```

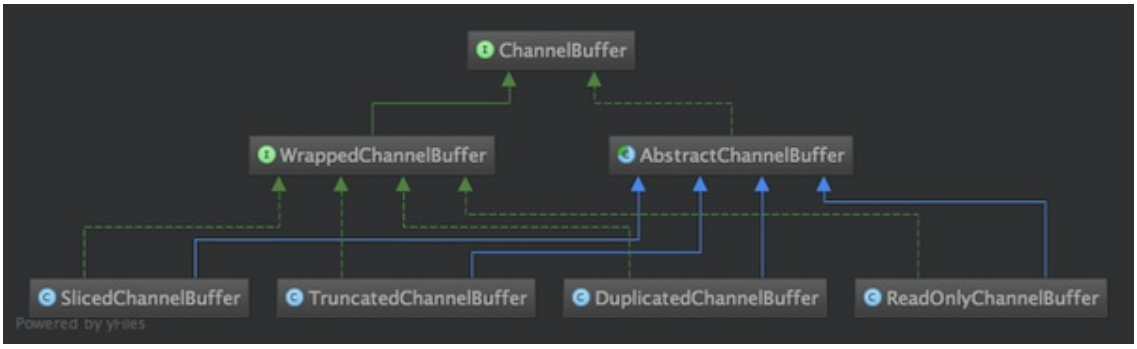
查找componentId的算法再次不作介绍了，大家自己实现起来也不会太难。值得一提的是，基

于ChannelBuffer连续读写的特性，使用了顺序查找(而不是二分查找)，并且用 `lastAccessedComponentId` 来进行缓存。

ByteBufferBackedChannelBuffer

前面说ChannelBuffer是自己的实现的，其实只说对了一半。 `ByteBufferBackedChannelBuffer` 就是封装了NIO ByteBuffer的类，用于实现堆外内存的Buffer(使用NIO的 `DirectByteBuffer`)。当然，其实它也可以放其他的ByteBuffer的实现类。代码实现就不说了，也没啥可说的。

WrappedChannelBuffer



`WrappedChannelBuffer` 都是几个对已有ChannelBuffer进行包装，完成特定功能的类。代码不贴了，实现都比较简单，列一下功能吧。

类名	入口	功能
SlicedChannelBuffer	ChannelBuffer.slice() ChannelBuffer.slice(int,int)	某个ChannelBuffer的一部分
TruncatedChannelBuffer	ChannelBuffer.slice() ChannelBuffer.slice(int,int)	某个ChannelBuffer的一部分， 可以理解为其实位置为0的SlicedChannelBuffer
DuplicatedChannelBuffer	ChannelBuffer.duplicate()	与某个ChannelBuffer使用同样的存储， 区别是有自己的index
ReadOnlyChannelBuffer	ChannelBuffers .unmodifiableBuffer(ChannelBuffer)	只读，你懂的

可以看到，关于实现方面，Netty 3.7的buffer相关内容还是比较简单的，也没有太多费脑细胞的地方。

而Netty 4.0之后就不同了。4.0，ChannelBuffer改名ByteBuf，成了单独项目buffer，并且为了性能优化，加入了BufferPool之类的机制，已经变得比较复杂了(本质倒没怎么变)。性能优化是个很复杂的事情，研究源码时，建议先避开这些东西，除非你对算法情有独钟。举个例

子，Netty4.0里为了优化，将Map换成了Java 8里6000行的ConcurrentHashMapV8，你们感受一下...

参考资料：

- TCP/IP协议 http://zh.wikipedia.org/zh-cn/TCP/IP%E5%8D%8F%E8%AE%AE
- Data_buffer http://en.wikipedia.org/wiki/Data_buffer
- Endianness http://en.wikipedia.org/wiki/Endianness

三、Channel中的Pipeline

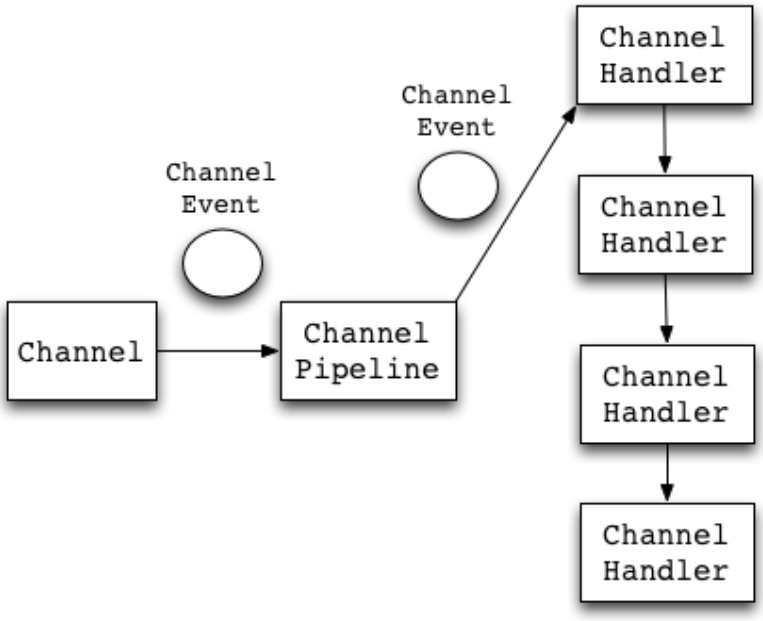
Channel是理解和使用Netty的核心。Channel的涉及内容较多，这里我使用由浅入深的介绍方法。在这篇文章中，我们主要介绍Channel部分中Pipeline实现机制。为了避免枯燥，借用一下《盗梦空间》的“梦境”概念，希望大家喜欢。

一层梦境：Channel实现概览

在Netty里，`Channel`是通讯的载体，而`ChannelHandler`负责Channel中的逻辑处理。

那么`ChannelPipeline`是什么呢？我觉得可以理解为ChannelHandler的容器：一个Channel包含一个ChannelPipeline，所有ChannelHandler都会注册到ChannelPipeline中，并按顺序组织起来。

在Netty中，`ChannelEvent`是数据或者状态的载体，例如传输的数据对应`MessageEvent`，状态的改变对应`ChannelStateEvent`。当对Channel进行操作时，会产生一个ChannelEvent，并发送到`ChannelPipeline`。ChannelPipeline会选择一个ChannelHandler进行处理。这个ChannelHandler处理之后，可能会产生新的ChannelEvent，并流转到下一个ChannelHandler。



例如，一个数据最开始是一个`MessageEvent`，它附带了一个未解码的原始二进制消息`ChannelBuffer`，然后某个Handler将其解码成了一个数据对象，并生成了一个新的`MessageEvent`，并传递给下一步进行处理。

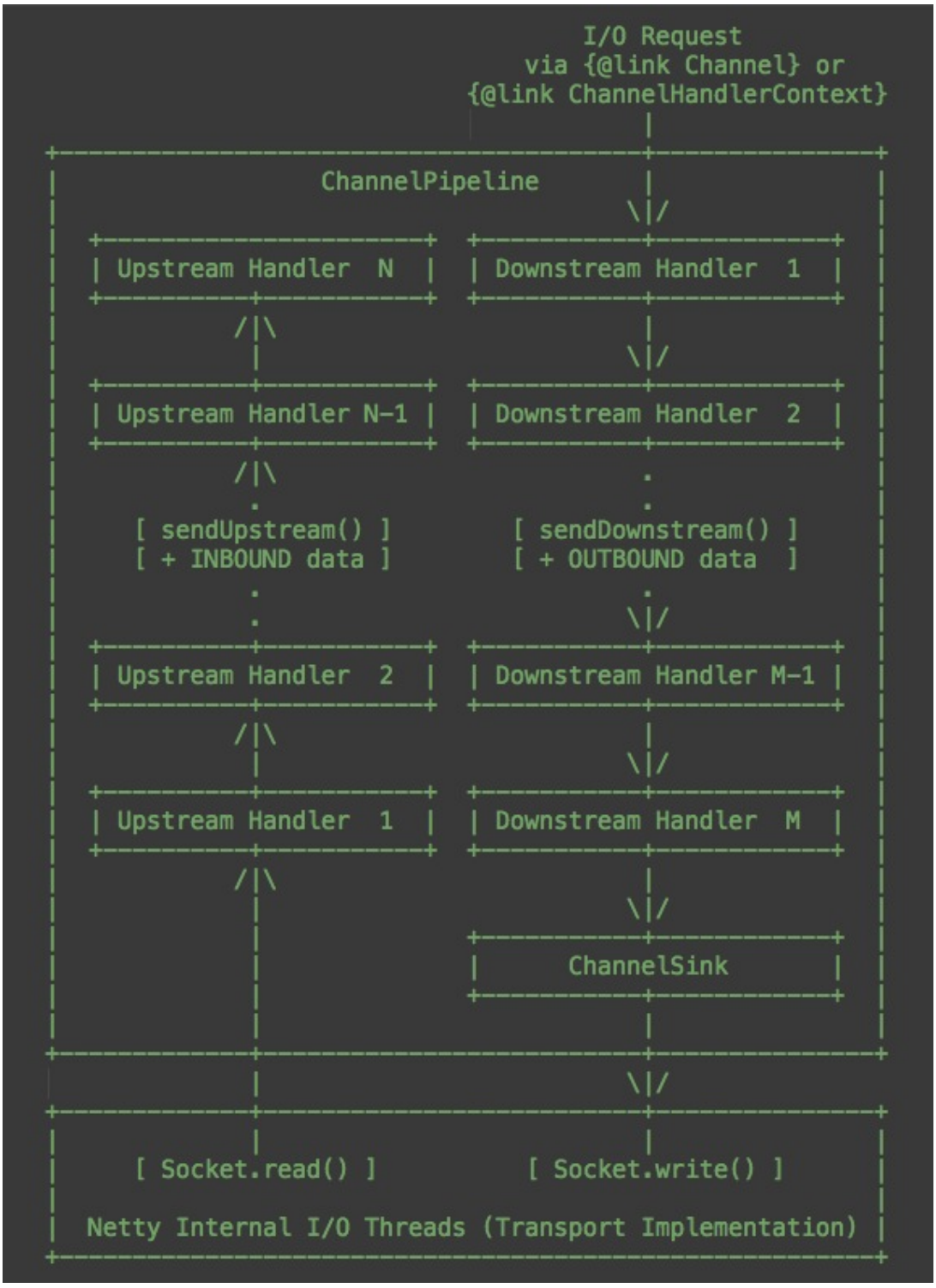
到了这里，可以看到，其实Channel的核心流程位于`ChannelPipeline`中。于是我们进入ChannelPipeline的深层梦境里，来看看它具体的实现。

二层梦境：ChannelPipeline的主流程

Netty的ChannelPipeline包含两条线路：Upstream和Downstream。Upstream对应上行，接收到的消息、被动的状态改变，都属于Upstream。Downstream则对应下行，发送的消息、主动的状态改变，都属于Downstream。`ChannelPipeline`接口包含了两个重要的方法:`sendUpstream(ChannelEvent e)`和`sendDownstream(ChannelEvent e)`，就分别对应了Upstream和Downstream。

对应的，ChannelPipeline里包含的ChannelHandler也包含两类：`ChannelUpstreamHandler`和`ChannelDownstreamHandler`。每条线路的Handler是互相独立的。它们都很简单的只包含一个方法：`ChannelUpstreamHandler.handleUpstream`和`ChannelDownstreamHandler.handleDownstream`。

Netty官方的javadoc里有一张图(`ChannelPipeline`接口里)，非常形象的说明了这个机制(我对原图进行了一点修改，加上了`ChannelSink`，因为我觉得这部分对理解代码流程会有些帮助)：



什么叫`ChannelSink`呢？ChannelSink包含一个重要方法`ChannelSink.eventSunk`，可以接受任意ChannelEvent。”sink”的意思是”下沉”，那么”ChannelSink”好像可以理解为”Channel下沉的地方”？实际上，它的作用确实是这样，也可以换个说法：”处于末尾的万能Handler”。最初读到这里，也有些困惑，这么理解之后，就感觉简单许多。**只有Downstream包含`ChannelSink`**，这里会做一些建立连接、绑定端口等重要操作。为什么UploadStream没有ChannelSink呢？我只能认为，一方面，不符合”sink”的意义，另一方面，也没有什么处理好做的吧！

这里有个值得注意的地方：在一条“流”里，一个`ChannelEvent`并不会主动的”流”经所有的Handler，而是由**上一个Handler显式的调用`ChannelPipeline.sendUp(Down)stream`产生，并**

交给下一个Handler处理。也就是说，每个Handler接收到一个ChannelEvent，并处理结束后，如果需要继续处理，那么它需要调用`sendUp(Down)stream`新发起一个事件。如果它不再发起事件，那么处理就到此结束，即使它后面仍然有Handler没有执行。这个机制可以保证最大的灵活性，当然对Handler的先后顺序也有了更严格的要求。

顺便说一句，在Netty 3.x里，这个机制会导致大量的ChannelEvent对象创建，因此Netty 4.x版本对此进行了改进。twitter的finagle框架实践中，就提到从Netty 3.x升级到Netty 4.x，可以大大降低GC开销。有兴趣的可以看看这篇文章：https://blog.twitter.com/2013/netty-4-at-twitter-reduced-gc-overhead

下面我们从代码层面来对这里面发生的事情进行深入分析，这部分涉及到一些细节，需要打开项目源码，对照来看，会比较有收获。

三层梦境：深入ChannelPipeline内部

DefaultChannelPipeline的内部结构

`ChannelPipeline`的主要的实现代码在`DefaultChannelPipeline`类里。列一下DefaultChannelPipeline的主要字段：

```
    public class DefaultChannelPipeline implements ChannelPipeline {

        private volatile Channel channel;
        private volatile ChannelSink sink;
        private volatile DefaultChannelHandlerContext head;
        private volatile DefaultChannelHandlerContext tail;
        private final Map<String, DefaultChannelHandlerContext> name2ctx =
            new HashMap<String, DefaultChannelHandlerContext>(4);
    }
```

这里需要介绍一下`ChannelHandlerContext`这个接口。顾名思义，ChannelHandlerContext保存了Netty与Handler相关的的上下文信息。而咱们这里的`DefaultChannelHandlerContext`，则是对`ChannelHandler`的一个包装。一个`DefaultChannelHandlerContext`内部，除了包含一个`ChannelHandler`，还保存了”next”和”prev”两个指针，从而形成一个双向链表。

因此，在`DefaultChannelPipeline`中，我们看到的是对`DefaultChannelHandlerContext`的引用，而不是对`ChannelHandler`的直接引用。这里包含”head”和”tail”两个引用，分别指向链表的头和尾。而name2ctx则是一个按名字索引DefaultChannelHandlerContext用户的一个map，主要在按照名称删除或者添加ChannelHandler时使用。

sendUpstream和sendDownstream

前面提到了，`ChannelPipeline`接口的两个重要的方法：`sendUpstream(ChannelEvent e)`和`sendDownstream(ChannelEvent e)`。**所有事件**的发起都是基于这两个方法进行的。`Channels`类有一系列`fireChannelBound`之类的`fireXXXX`方法，其实都是对这两个方法的facade包装。

下面来看一下这两个方法的实现。先看sendUpstream(对代码做了一些简化，保留主逻辑)：

```
    public void sendUpstream(ChannelEvent e) {
        DefaultChannelHandlerContext head = getActualUpstreamContext(this.he
ad);
        head.getHandler().handleUpstream(head, e);
    }

    private DefaultChannelHandlerContext getActualUpstreamContext(DefaultCha
nnelHandlerContext ctx) {
        DefaultChannelHandlerContext realCtx = ctx;
        while (!realCtx.canHandleUpstream()) {
            realCtx = realCtx.next;
            if (realCtx == null) {
                return null;
            }
        }
        return realCtx;
    }
```

这里最终调用了`ChannelUpstreamHandler.handleUpstream`来处理这个ChannelEvent。有意思的是，这里我们看不到任何”将Handler向后移一位”的操作，但是我们总不能每次都用同一个Handler来进行处理啊？实际上，我们更为常用的
是`ChannelHandlerContext.handleUpstream`方法(实现
是`DefaultChannelHandlerContext.sendUpstream`方法)：

```
    public void sendUpstream(ChannelEvent e) {
        DefaultChannelHandlerContext next = getActualUpstreamContext(this.ne
xt);
        DefaultChannelPipeline.this.sendUpstream(next, e);
    }
```

可以看到，这里最终仍然调用了`ChannelPipeline.sendUpstream`方法，但是**它会将Handler指针后移。**

我们接下来看看`DefaultChannelHandlerContext.sendDownstream`：

```
    public void sendDownstream(ChannelEvent e) {
        DefaultChannelHandlerContext prev = getActualDownstreamContext(this.
prev);
        if (prev == null) {
            try {
                getSink().eventSunk(DefaultChannelPipeline.this, e);
            } catch (Throwable t) {
                notifyHandlerException(e, t);
            }
        } else {
            DefaultChannelPipeline.this.sendDownstream(prev, e);
        }
    }
```

与sendUpstream好像不大相同哦？这里有两点：一是到达末尾时，就如梦境二所说，会调用

ChannelSink进行处理；二是这里指针是**往前移**的，所以我们知道了：

UpstreamHandler是从前往后执行的，DownstreamHandler是从后往前执行的。在ChannelPipeline里添加时需要注意顺序了！

DefaultChannelPipeline里还有些机制，像添加/删除/替换Handler，以及`ChannelPipelineFactory`等，比较好理解，就不细说了。

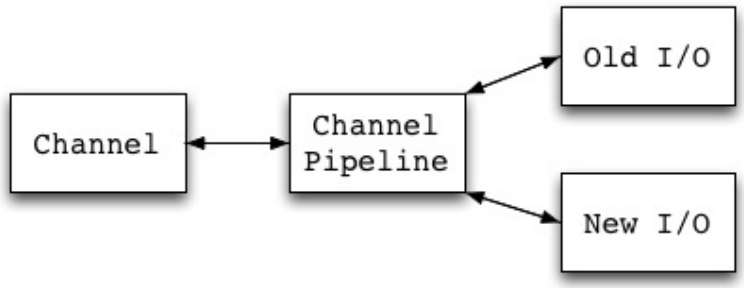
回到现实：Pipeline解决的问题

好了，深入分析完代码，有点头晕了，我们回到最开始的地方，来想一想，Netty的Pipeline机制解决了什么问题？

我认为至少有两点：

一是提供了ChannelHandler的编程模型，基于ChannelHandler开发业务逻辑，基本不需要关心网络通讯方面的事情，专注于编码/解码/逻辑处理就可以了。Handler也是比较方便的开发模式，在很多框架中都有用到。

二是实现了所谓的"Universal Asynchronous API"。这也是Netty官方标榜的一个功能。用过OIO和NIO的都知道，这两套API风格相差极大，要从一个迁移到另一个成本是很大的。即使是NIO，异步和同步编程差距也很大。而Netty屏蔽了OIO和NIO的API差异，通过Channel提供对外接口，并通过ChannelPipeline将其连接起来，因此替换起来非常简单。



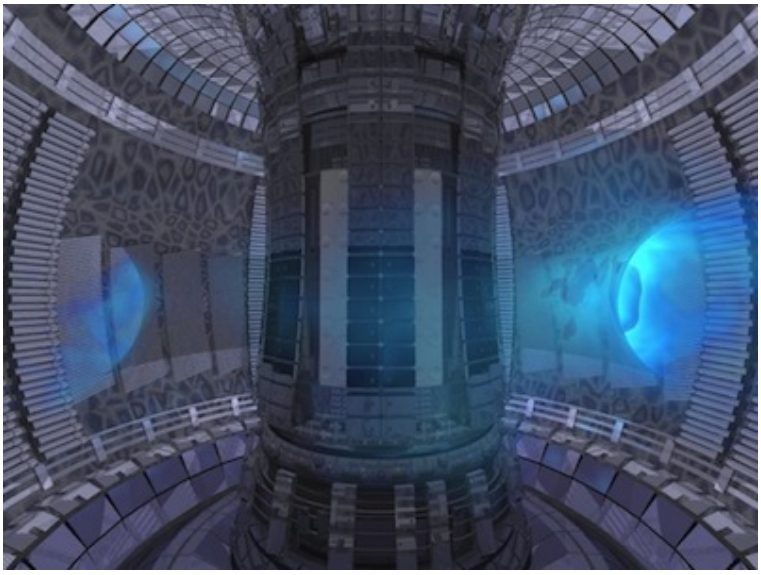
理清了ChannelPipeline的主流程，我们对Channel部分的大致结构算是弄清楚了。可是到了这里，我们依然对一个连接具体怎么处理没有什么概念，下篇文章，我们会分析一下，在Netty中，捷径如何处理连接的建立、数据的传输这些事情。

PS: Pipeline这部分拖了两个月，终于写完了。中间写的实在缓慢，写个高质量(至少是自认为吧！)的文章不容易，但是仍不忍心这部分就此烂尾。中间参考了一些优秀的文章，还自己使用netty开发了一些应用。以后这类文章，还是要集中时间来写完好了。

参考资料：

- Sink http://en.wikipedia.org/wiki/Sink_(computing)

四、Netty与Reactor模式



一：Netty、NIO、多线程？

时隔很久终于又更新了！之前一直迟迟未动也是因为积累不够，后面比较难下手。过年期间@李林锋hw发布了一个Netty5.0架构剖析和源码解读 http://vdisk.weibo.com/s/C9LV9iVqH13rW/1391437855，看完也是收获不少。前面的文章我们分析了Netty的结构，这次咱们来分析最错综复杂的一部分-Netty中的多线程以及NIO的应用。

理清NIO与Netty的关系之前，我们必须先要来看看Reactor模式。Netty是一个典型的多线程的Reactor模式的使用，理解了这部分，在宏观上理解Netty的NIO及多线程部分就不会有什么困难了。

本篇文章依然针对Netty 3.7，不过因为也看过一点Netty 5的源码，所以会有一点介绍。

二：Reactor，反应堆还是核电站？

1、Reactor的由来

Reactor是一种广泛应用在服务器端开发的设计模式。Reactor中文大多译为“反应堆”，我当初接触这个概念的时候，就感觉很厉害，是不是它的原理就跟“核反应”差不多？后来才知道其实没有什么关系，从Reactor的兄弟“Proactor”（多译为前摄器）就能看得出来，这两个词的中文

翻译其实都不是太好，不够形象。实际上，Reactor模式又有别名“Dispatcher”或者“Notifier”，我觉得这两个都更加能表明它的本质。

那么，Reactor模式究竟是个什么东西呢？这要从事件驱动的开发方式说起。我们知道，对于应用服务器，一个主要规律就是，CPU的处理速度是要远远快于IO速度的，如果CPU为了IO操作（例如从Socket读取一段数据）而阻塞显然是不划算的。好一点的方法是分为多进程或者线程去进行处理，但是这样会带来一些进程切换的开销，试想一个进程一个数据读了500ms，期间进程切换到它3次，但是CPU却什么都不能干，就这么切换走了，是不是也不划算？

这时先驱们找到了事件驱动，或者叫回调的方式，来完成这件事情。这种方式就是，应用业务向一个中间人注册一个回调（event handler），当IO就绪后，就这个中间人产生一个事件，并通知此handler进行处理。*这种回调的方式，也体现了“好莱坞原则”（Hollywood principle）- “Don’t call us, we’ll call you”，在我们熟悉的IoC中也有用到。看来软件开发真是互通的！*

好了，我们现在来看Reactor模式。在前面事件驱动的例子里有个问题：我们如何知道IO就绪这个事件，谁来充当这个中间人？Reactor模式的答案是：由一个不断等待和循环的单独进程（线程）来做这件事，它接受所有handler的注册，并负责先操作系统查询IO是否就绪，在就绪后就调用指定handler进行处理，这个角色的名字就叫做Reactor。

2、Reactor与NIO

Java中的NIO可以很好的和Reactor模式结合。关于NIO中的Reactor模式，我想没有什么资料能比Doug Lea大神（不知道Doug Lea？看看JDK集合包和并发包的作者吧）在《Scalable IO in Java》解释的更简洁和全面了。NIO中Reactor的核心是 `Selector`，我写了一个简单的Reactor示例，这里我贴一个核心的Reactor的循环（这种循环结构又叫做 `EventLoop`），剩余代码在learning-src目录下。

```
    public void run() {
        try {
            while (!Thread.interrupted()) {
                selector.select();
                Set selected = selector.selectedKeys();
                Iterator it = selected.iterator();
                while (it.hasNext())
                    dispatch((SelectionKey) (it.next()));
                selected.clear();
            }
        } catch (IOException ex) { /* ... */
        }
    }
```

3、与Reactor相关的其他概念

前面提到了Proactor模式，这又是什么呢？简单来说，Reactor模式里，操作系统只负责通知IO就绪，具体的IO操作（例如读写）仍然是要在业务进程里阻塞的去做的，而Proactor模式则更进一步，由操作系统将IO操作执行好（例如读取，会将数据直接读到内存buffer中），而handler只负责处理自己的逻辑，真正做到了IO与程序处理异步执行。所以我们一般又说Reactor是同步IO，Proactor是异步IO。

关于阻塞和非阻塞、异步和非异步，以及UNIX底层的机制，大家可以看看这篇文章IO - 同步，

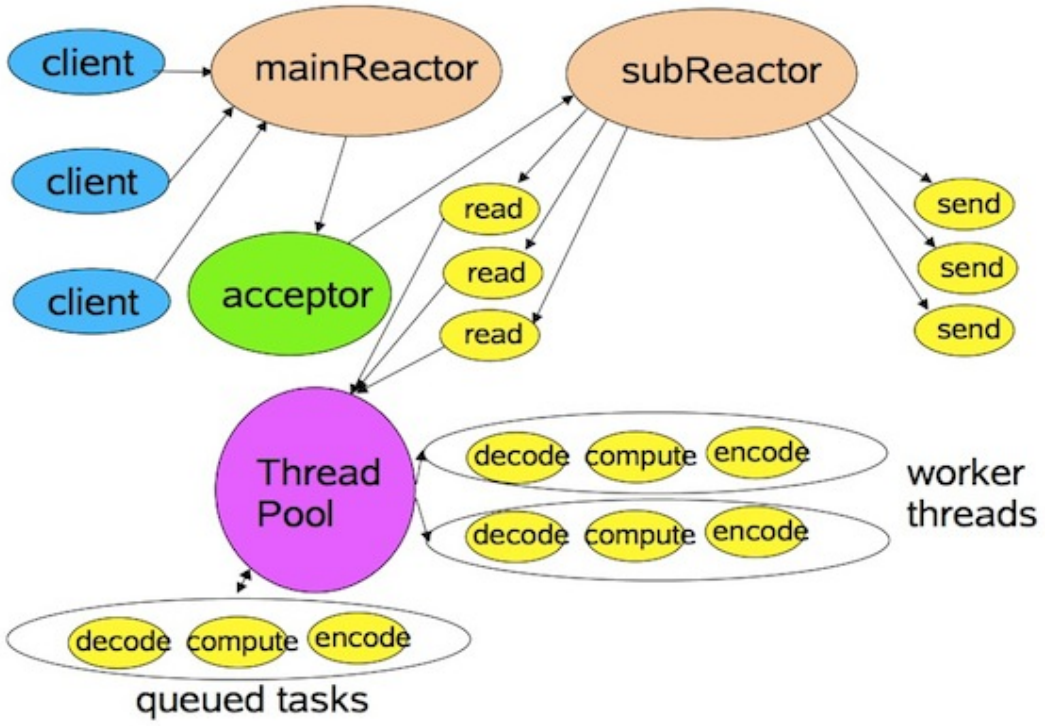
异步，阻塞，非阻塞（亡羊补牢篇），以及陶辉（《深入理解nginx》的作者）《高性能网络编程》的系列。

三：由Reactor出发来理解Netty

1、多线程下的Reactor

讲了一堆Reactor，我们回到Netty。在《Scalable IO in Java》中讲到了一种多线程下的Reactor模式。在这个模式里，mainReactor只有一个，负责响应client的连接请求，并建立连接，它使用一个NIO Selector；subReactor可以有一个或者多个，每个subReactor都会在一个独立线程中执行，并且维护一个独立的NIO Selector。

这样的好处很明显，因为subReactor也会执行一些比较耗时的IO操作，例如消息的读写，使用多个线程去执行，则更加有利于发挥CPU的运算能力，减少IO等待时间。



2、Netty中的Reactor与NIO

好了，了解了多线程下的Reactor模式，我们来看看Netty吧（以下部分主要针对NIO，OIO部分更加简单一点，不重复介绍了）。Netty里对应mainReactor的角色叫做“Boss”，而对应subReactor的角色叫做”Worker”。Boss负责分配请求，Worker负责执行，好像也很贴切！以TCP的Server端为例，这两个对应的实现类分别为 `NioServerBoss` 和 `NioWorker`（Server和Client的Worker没有区别，因为建立连接之后，双方就是对等的进行传输了）。

Netty 3.7中Reactor的EventLoop在 `AbstractNioSelector.run()` 中，它实现了 `Runnable` 接口。这个类是Netty NIO部分的核心。它的逻辑非常复杂，其中还包括一些对JDK Bug的处理（例如 `rebuildSelector` ），刚开始读的时候不需要深入那么细节。我精简了大部分代码，保留

主干如下：

```
abstract class AbstractNioSelector implements NioSelector {


    //NIO Selector
    protected volatile Selector selector;

    //内部任务队列
    private final Queue<Runnable> taskQueue = new ConcurrentLinkedQueue<Runn
able>();

    //selector循环
    public void run() {
        for (;;) {
            try {
                //处理内部任务队列
                processTaskQueue();
                //处理selector事件对应逻辑
                process(selector);
            } catch (Throwable t) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // Ignore.
                }
            }
        }
    }

    private void processTaskQueue() {
        for (;;) {
            final Runnable task = taskQueue.poll();
            if (task == null) {
                break;
            }
            task.run();
        }
    }

    protected abstract void process(Selector selector) throws IOException;

}
```

其中process是主要的处理事件的逻辑，例如在`AbstractNioWorker`中，处理逻辑如下：

```
    protected void process(Selector selector) throws IOException {
        Set<SelectionKey> selectedKeys = selector.selectedKeys();
        if (selectedKeys.isEmpty()) {
            return;
        }
        for (Iterator<SelectionKey> i = selectedKeys.iterator(); i.hasNext()
;) {
            SelectionKey k = i.next();
            i.remove();
```

```
            try {
                int readyOps = k.readyOps();
                if ((readyOps & SelectionKey.OP_READ) != 0 || readyOps == 0)
{
                    if (!read(k)) {
                        // Connection already closed - no need to handle wri
te.
                        continue;
                    }
                }
                if ((readyOps & SelectionKey.OP_WRITE) != 0) {
                    writeFromSelectorLoop(k);
                }
            } catch (CancelledKeyException e) {
                close(k);
            }

            if (cleanUpCancelledKeys()) {
                break; // break the loop to avoid ConcurrentModificationExce
ption
            }
        }
    }
```

这不就是第二部分提到的selector经典用法了么？

在Netty 4.0之后，作者觉得`NioSelector`这个叫法，以及区分`NioBoss`和`NioWorker`的做法稍微繁琐了点，干脆就将这些合并成了`NioEventLoop`，从此这两个角色就不做区分了。我倒是觉得新版本的会更优雅一点。

3、Netty中的多线程

下面我们来看Netty的多线程部分。一旦对应的Boss或者Worker启动，就会分配给它们一个线程去一直执行。对应的概念为`BossPool`和`WorkerPool`。对于每个`NioServerSocketChannel`，Boss的Reactor有一个线程，而Worker的线程数由Worker线程池大小决定，但是默认最大不会超过CPU核数*2，当然，这个参数可以通过`NioServerSocketChannelFactory`构造函数的参数来设置。

```
    public NioServerSocketChannelFactory(
            Executor bossExecutor, Executor workerExecutor,
            int workerCount) {
        this(bossExecutor, 1, workerExecutor, workerCount);
    }
```

最后我们比较关心一个问题，我们之前`ChannlePipeline`中的ChannleHandler是在哪个线程执行的呢？答案是在Worker线程里执行的，并且会阻塞Worker的EventLoop。例如，在`NioWorker`中，读取消息完毕之后，会触发`MessageReceived`事件，这会使得Pipeline中的handler都得到执行。

```
    protected boolean read(SelectionKey k) {
        ....
```

```
            if (readBytes > 0) {
                // Fire the event.
                fireMessageReceived(channel, buffer);
            }

            return true;
        }
```

可以看到，对于处理事件较长的业务，并不太适合直接放到ChannelHandler中执行。那么怎么处理呢？我们在Handler部分会进行介绍。

参考资料：

- Scalable IO in Java http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf
- Netty5.0架构剖析和源码解读 http://vdisk.weibo.com/s/C9LV9iVqH13rW/1391437855
- Reactor pattern http://en.wikipedia.org/wiki/Reactor_pattern
- Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf
- 高性能网络编程6–reactor反应堆与定时器管理 http://blog.csdn.net/russell_tao/article/details/17452997
- IO - 同步，异步，阻塞，非阻塞 （亡羊补牢篇） http://blog.csdn.net/historyasamirror/article/details/5778378

题图来自：http://www.worldindustrialreporter.com/france-gives-green-light-to-tokamak-fusion-reactor/