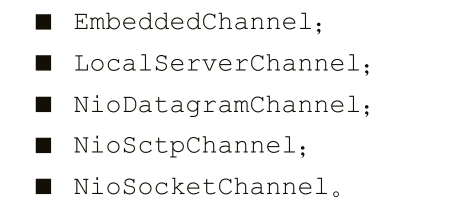
# 1 Netty的基础概念

## Channel

Channel表示一个到实体（文件，套接字等）的开放连接，在NIO中是通道，连接和通道都差不多意思。

### Channel**接口**的实现类

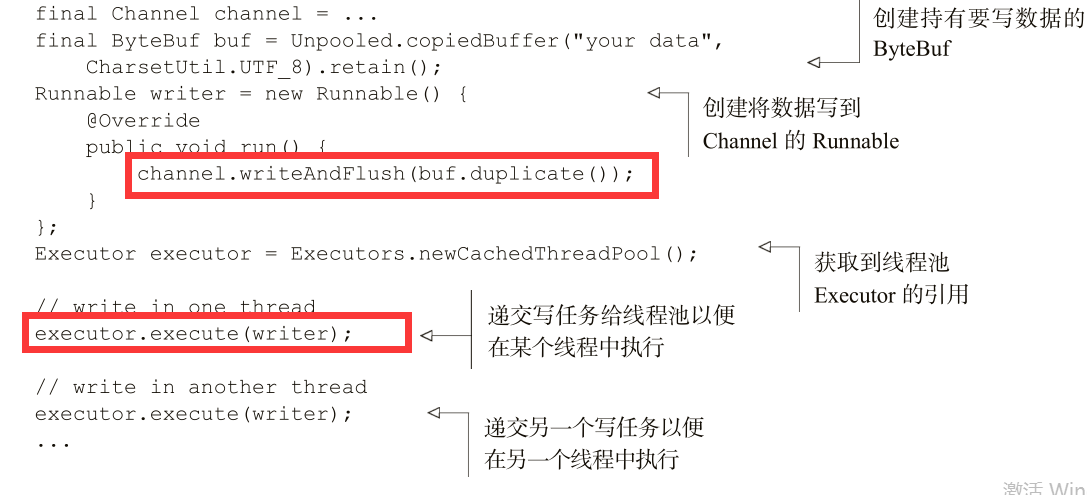
图1



### Netty的**Channel**是线程安全的

多个线程可以使用同一个Channel实例读写。

图1



### Channel的**生命**周期

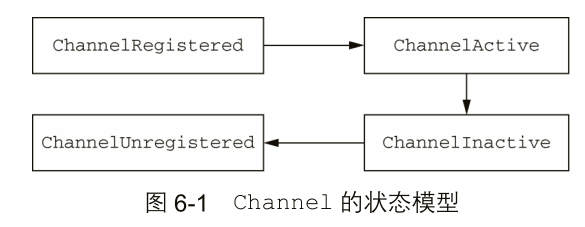
ChannelRegistered：Channel已经注册到了EventLoop。

ChannelActive：Channel已经连接到远程节点且处于就绪状态，可以接收和发送数据了。

ChannelInActive：Channel没有连接到远程节点。

ChannelUnregistered：Channel已经创建，但未注册到EventLoop。

图1



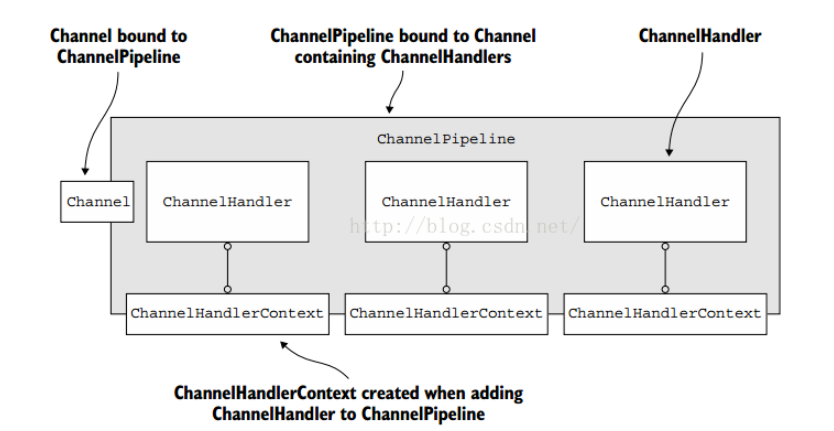
### Channel，ChannelHandler，ChannelHandlerConext，ChannelPipeline的区别

Channel是连接，连接到一个实体。

Channel连接到服务器后，会分配一个专属的ChannelPipeline，对Channel这个连接所有的业务处理都在ChannelPipeline中。

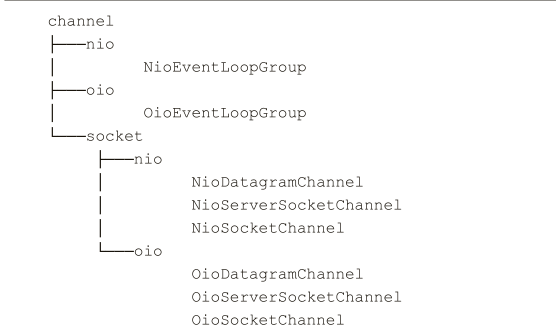
ChannelPipeline中安装了很多个ChannelHandler，每个ChannelHandler封装了一些业务逻辑处理。

ChannelHandlerConext是ChannelPipeline和ChannelHandler之间的桥梁。



### Channel和EventLoopGroup的兼容性

图1



### ChannelOption设置Channel的选项，帮Channel添加属性

当connect()或bind()方法调用时，ChannelOption和属性会被添加到已创建的Channel上。

图1



## ChannelHandler

Channel是连接，那么ChannelHandler就是对这个连接的业务处理的封装。所有对Channel连接的业务处理都封装在ChannelHandler中。

### ChannelHandler的继承图

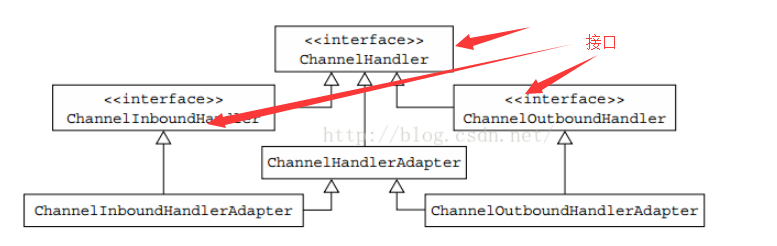
ChannelHandler是一个接口。

ChannelInboundHandler是一个继承ChannelHandler接口的接口。

ChannelOutboundHandler是一个继承ChannelHandler接口的接口。

ChannelHandlerAdapter是一个继承ChannelHandler接口的类。

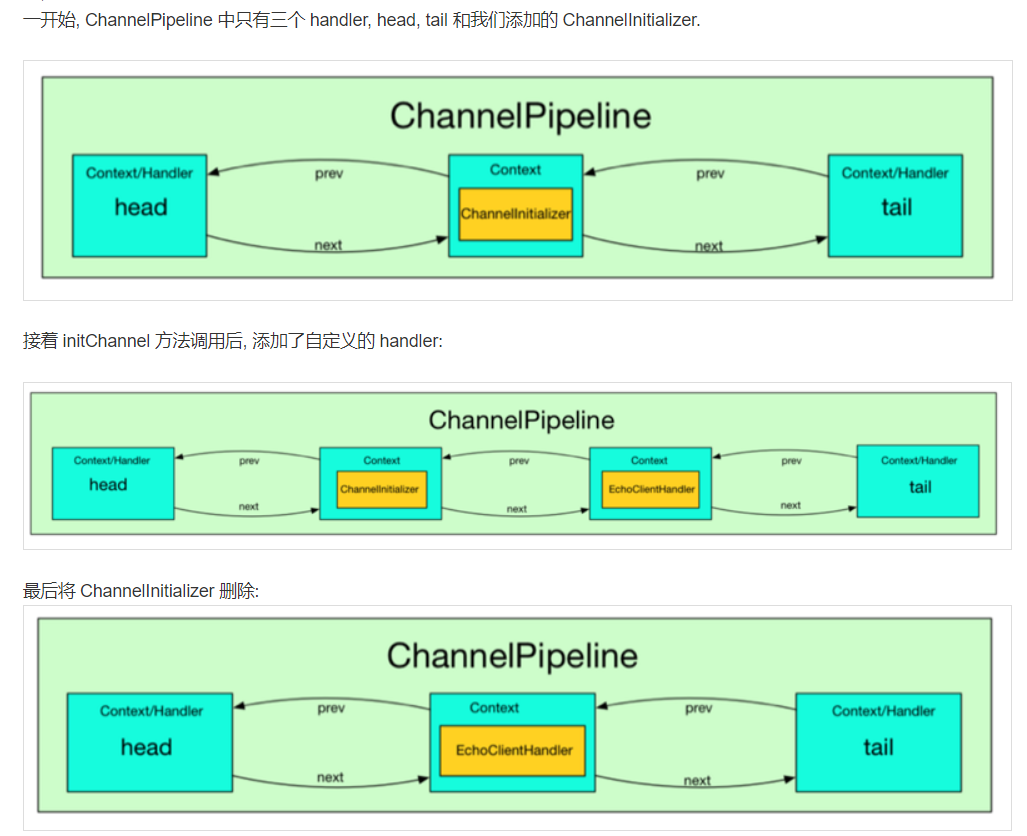
图1



### ChannelInitializer安装ChannelHandler被安装到ChannelPipleline的过程

ChannelInitializer是用来给ChannelPipeline安装一组自定义的ChannelHander的。ChannelInitializer会先注册到ServerBootstrap中，然后给ChannelPipeline安装一组自定义的ChannelHander，最后把自己从ChannelPipeline中移除。

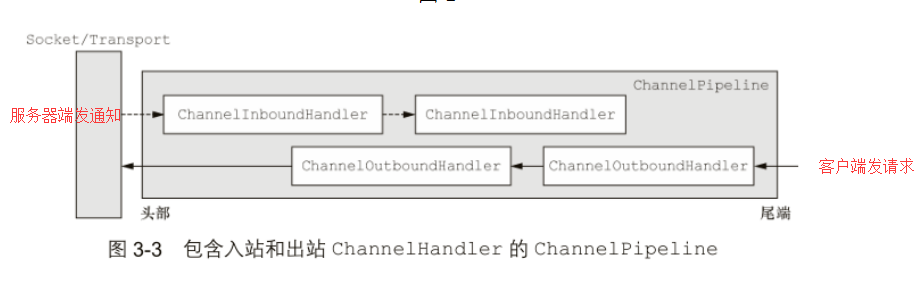
图1



### 入站和出站的区别

1. 从客户端的角度来说，如果事件的运动方向是从客户端到服务器，则是出站；如果是从服务器到客户端，则是入站。
2. Inbound事件是服务器端某个事件发生了，需要通知给客户端，通常是Channel状态发生改变或IO事件就绪，传播方向是head->customContext->tail（tail是最后一个inboundHandler）；Outbound事件是客户端请求事件，比如发起一个connect请求，传播方向是tail->customContext->head（head是最后一个outboundHandler）。

图1



### 编码器和解码器

入站用解码器，将字节转换成Java对象。

出战用编码器，将Java对象转换成字节。

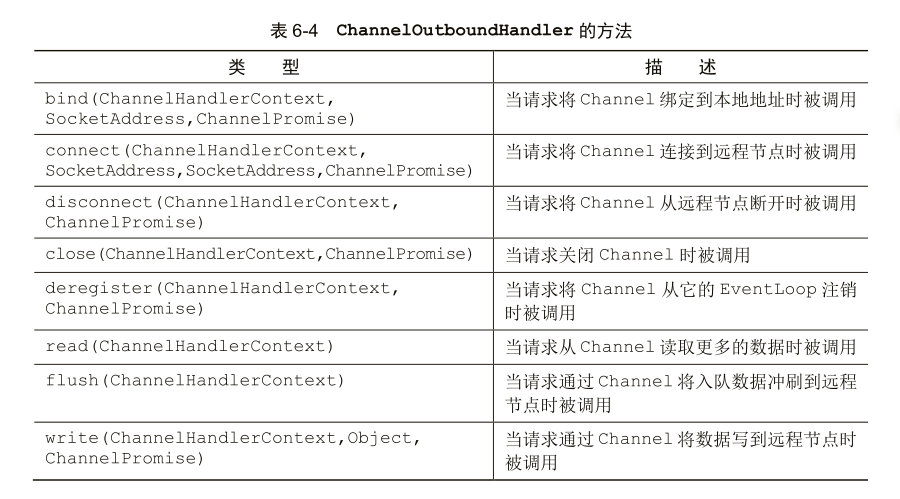
### ChannelInboundHandler的方法

图1



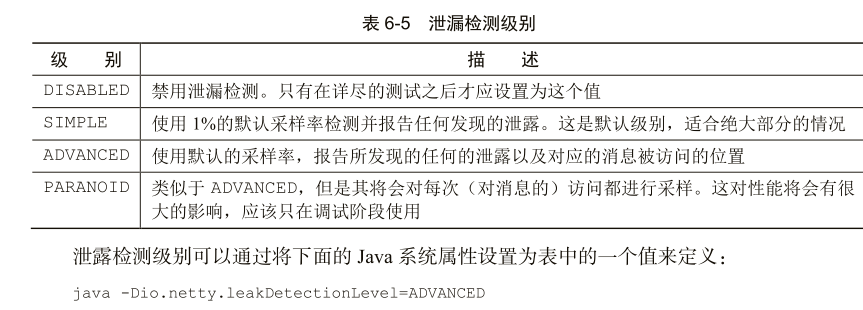
### 1.2.6 ChannelOutboundHandler的方法

图1



### Netty的内存泄漏检测级别

图1



### 释放消息，不传递给下一个ChannelHandler

图1

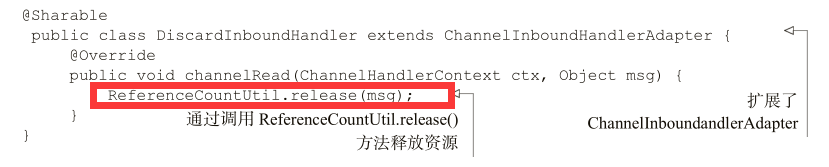
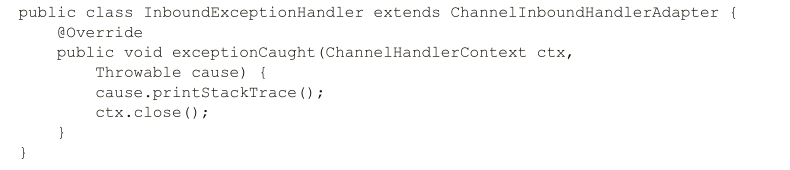


图2



### 处理入站异常

图1



### 处理出站异常

通过向ChannelFuture注册ChannelFutureListener后，因为几乎所有的ChannelOutboundHandler的方法中都有一个参数是ChannelPromise，是ChannelFuture的子类，通过设置该ChannelPromise的结果，可以将成功或失败结果通知到注册的ChannelFutureListener。

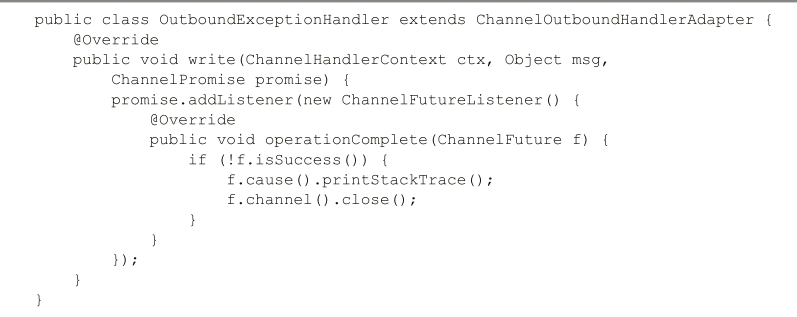
法1：每个出站操作，比如write都会返回一个ChannelFuture，在该ChannelFuture上注册ChannelFutureListener。

图1



法2：直接在ChannelPromise上注册ChannelFutureListener。

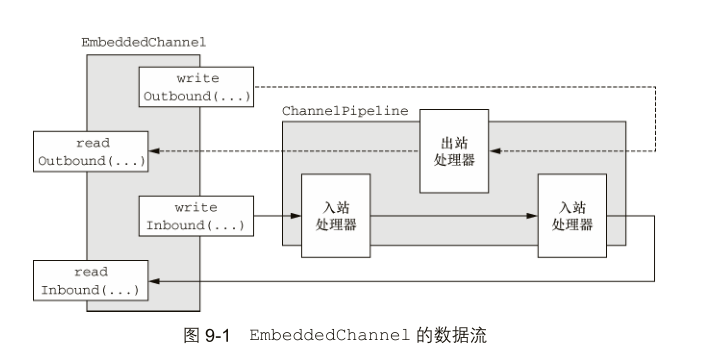
图2



### EmbeddedChannel 测试ChannelHandler

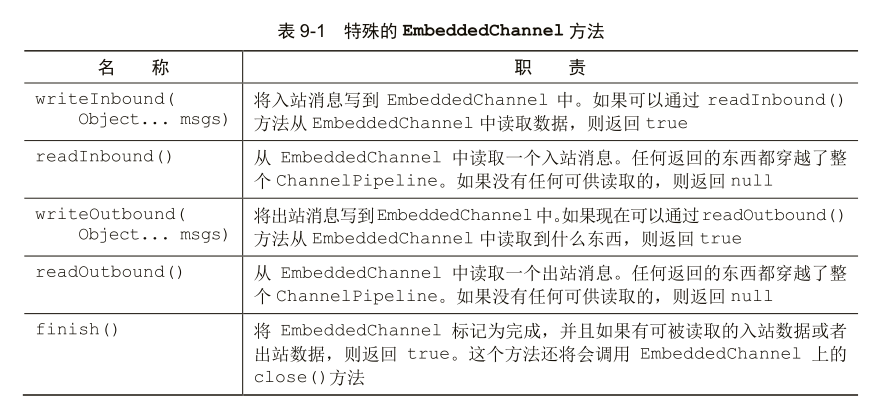
EmbeddedChannel 通过写数据穿过整个ChannelPipeline，然后读取穿越ChannelPipeline后的数据，来对ChannelHandler进行检测。

图1



EmbeddedChannel 的方法：

图2



## ChannelHanderContext

当一个ChannelHandler被添加到ChannelPipeline后，会创建一个ChannelHanderContext与该ChannelHandler关联。

ChannelHanderContext使得ChannelHandler可以和它的ChannelPipeline，和其他的ChannelHandler进行交互。

### 调用Channel，ChannelPipeline，用ChannelHanderContext的方法的区别

调用Channel或ChannelPipeline的方法，事件会沿着整个ChannelPipeline传播；调用ChannelHanderContext的方法，事件会从ChannelPipeline的下一个ChannelHandler开始传播。

图1（ChannelPipeline将事件传递给第一个ChannelHandler，第一个ChannelHandler关联的ChannelHandlerContext传递给了第二个ChannelHandler......）

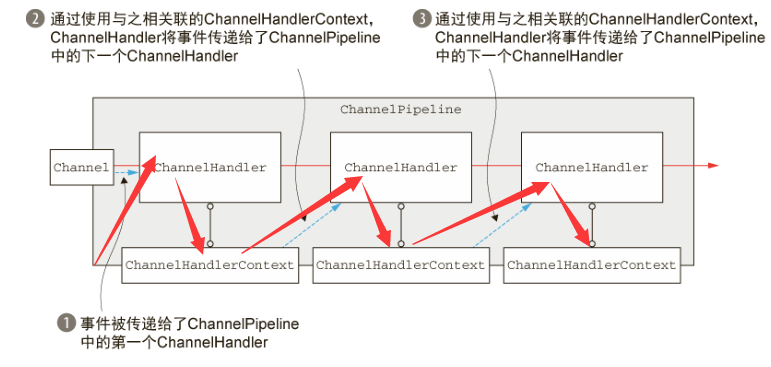
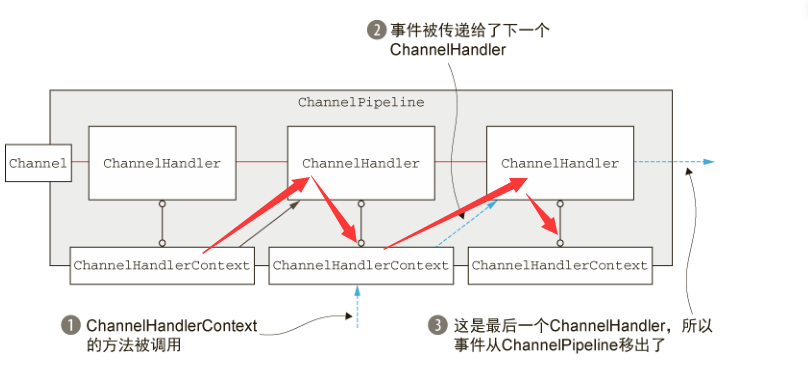
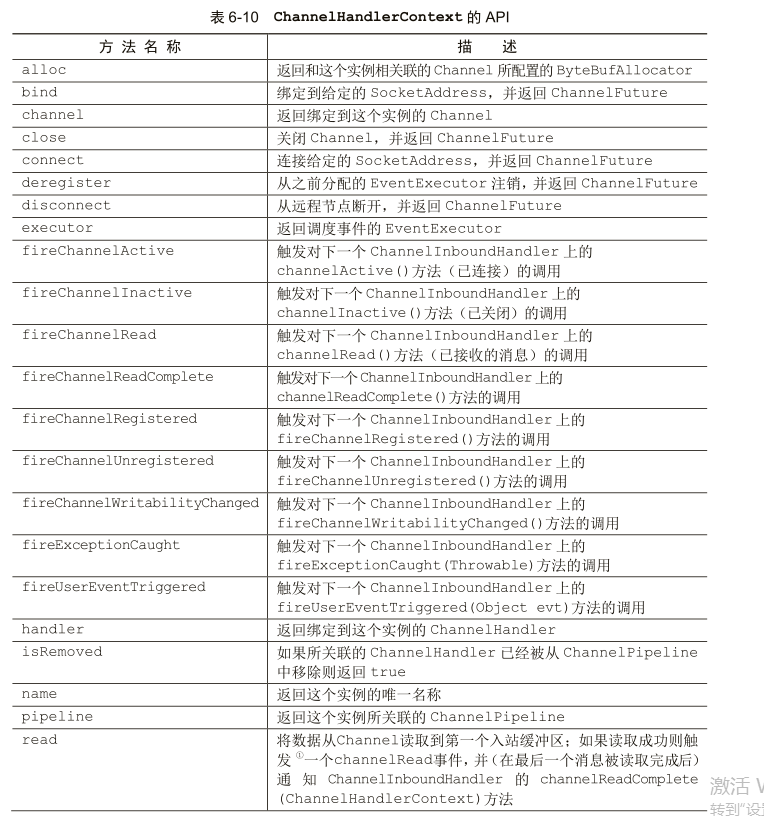


图2（ChannelHandlerContext将事件传递给了第二个ChannelHandler，跳过了第一个ChannelHandler）



### ChannelHanderContext的API

图1





## **ChannelFuture**

因为Netty中所有的I/O操作都是异步的，所以ChannelFuture用来在未来的某个时间点获取I/O的结果。

ChannelFuture可以注册多个ChannelFutureListener监听器实例，ChannelFutureListener监听器实例的operationComplete()方法会在某个操作完成后被调用。

图1

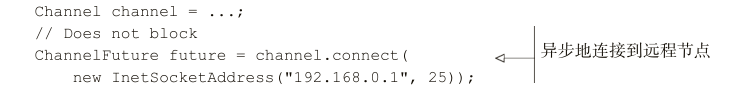
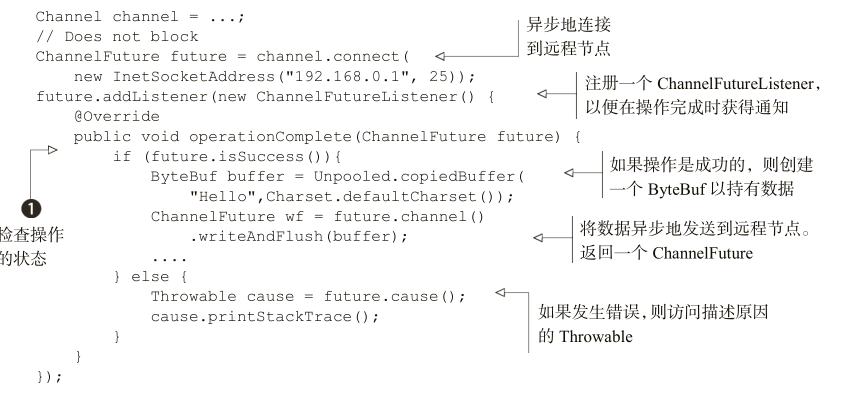


图2



## **EventLoop**

### 1.5.1 EventLoopGroup，EventLoop，Channel的关系

对于每一个客户端连接，会从EventLoopGroup（类似于线程池）获取一个EventLoop（类似于线程），EventLoop会绑定一个线程Thread。

一个Channel只注册到一个EventLoop中，但一个EventLoop可能会分配给多个Channel，所以同一个EventLoop下的多个Channel会共用同一个Thread。

（注意：对于同一个EventLoop下的多个Channel来说，ThreadLocal都是一样的）

### 1.5.2 EventLoop的类层次结构

图1

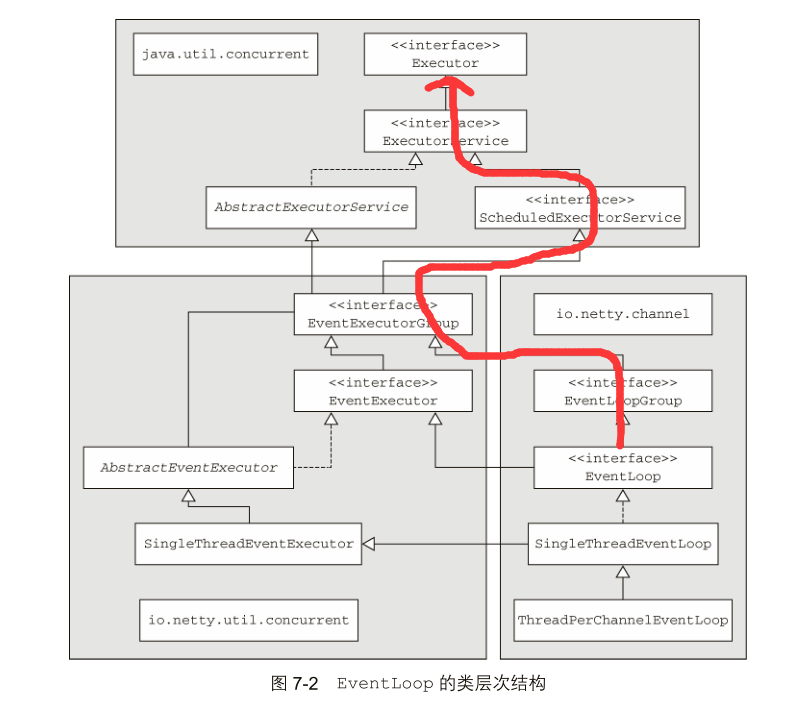
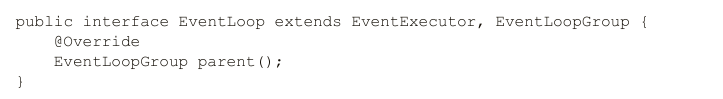


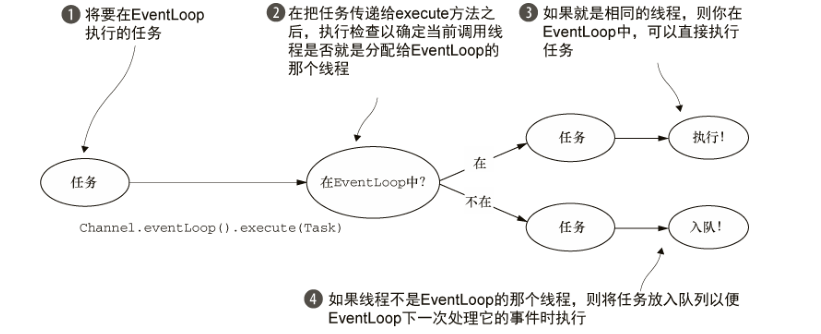
图2



### 1.5.3 任务和EventLoop的关系

当一个任务交给EventLoop后，EventLoop判断当前线程是否就是分配给EventLoop的那个线程，如果是则执行任务；如果不是则放入队列中。

图1

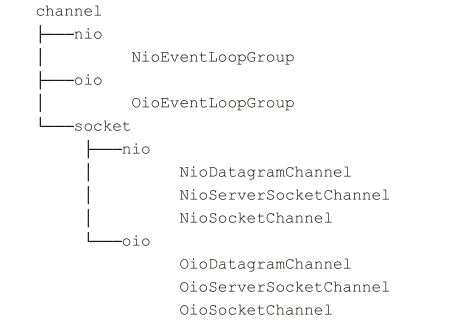


### 1.5.4 Channel和EventLoopGroup的兼容性

EventLoopGroup只有nio和oio两个版本。

Channel首先区分nio还是oio，然后是TCP还是UDP，TCP里面还分ServerSocket和Socket。

图1

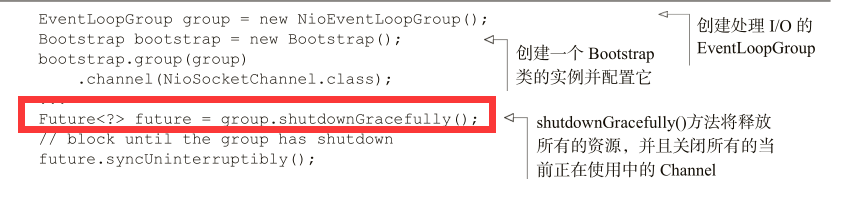


### 1.5.5 优雅关闭EventLoopGroup

因为关闭EventLoopGroup也是一个异步过程，首先立刻返回一个future。

也可以显示调用所有Channel的close()方法，但EventLoopGroup必须要关闭。

图1



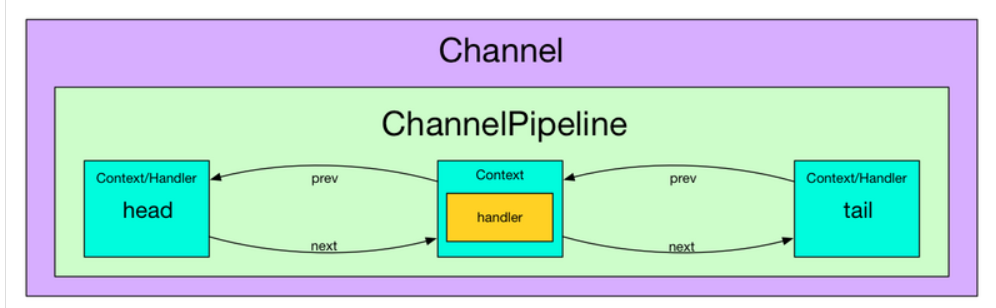
## ChannelPipeline

ChannelPipeline中维护了一个由ChannelHandlerContext组成的双向链接。链表头是HeadContext，链表尾是TailContext，每个ChannelHandlerContext中又关联着一个ChannelHandler。

（注意：HeadContext关联的是一个outboundHandler，TailContext关联的是一个inboundHandler，ChannelInitializer关联的是inboundHandler）

每一个ChannelHanlder会做一些业务处理，然后传递给ChannelPipeLine中的下一个ChannelHanlder。

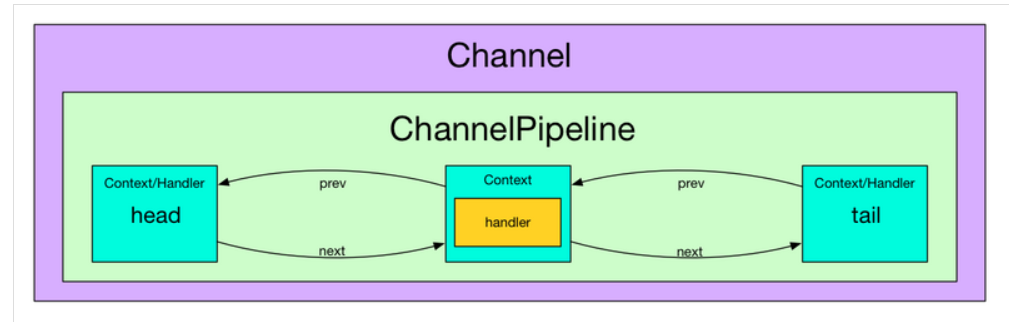
图1



### Channel和ChannelPipeline的关系

Channel是数据通道，Channel实例创建时会关联一个ChannelPipeLine实例，通道中的一系列处理都在ChannelPipeLine中进行。

图1



### ChannelPipeline的触发事件

ChannelPipeline的API可以触发下一个ChannelHandler的某个操作。

图1

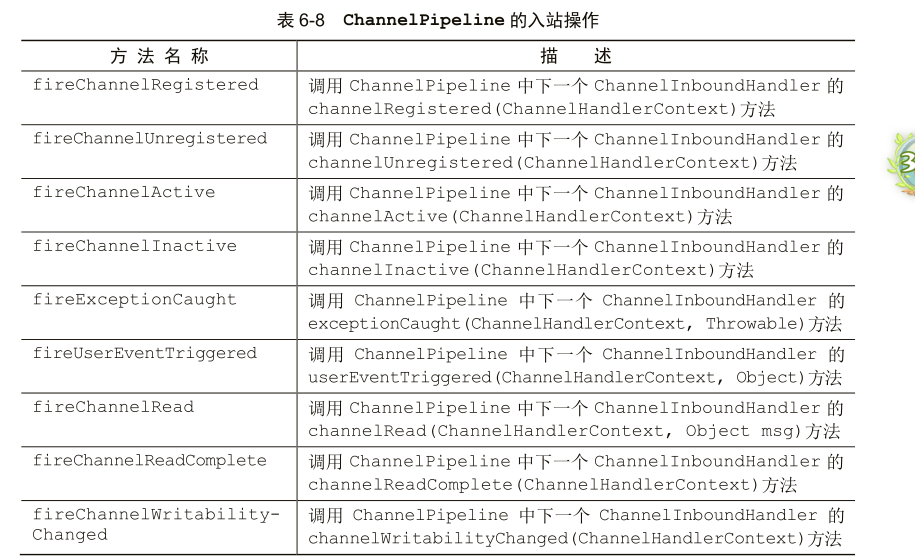


图2



## Bootstrap

Bootstrap是引导程序，一种用于引导客户端（Bootstrap），另一种用于引导服务器（ServerBootstrap）。

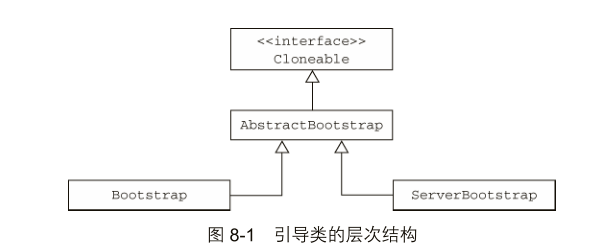
### Bootstrap和ServerBootstrap的区别

1.Bootstrap要绑定一个远程主机和端口；ServerBootstrap绑定到本地IP和端口，因为要监听连接。

2.Bootstrap只需要一个EventLoopGroup；ServerBootstrap需要2个EventLoopGroup（但可以是同一个EventLoopGroup实例），因为需要2组不同的Channel，第一组是ServerChannel，第二组是连接到服务器端的客户端Channel。

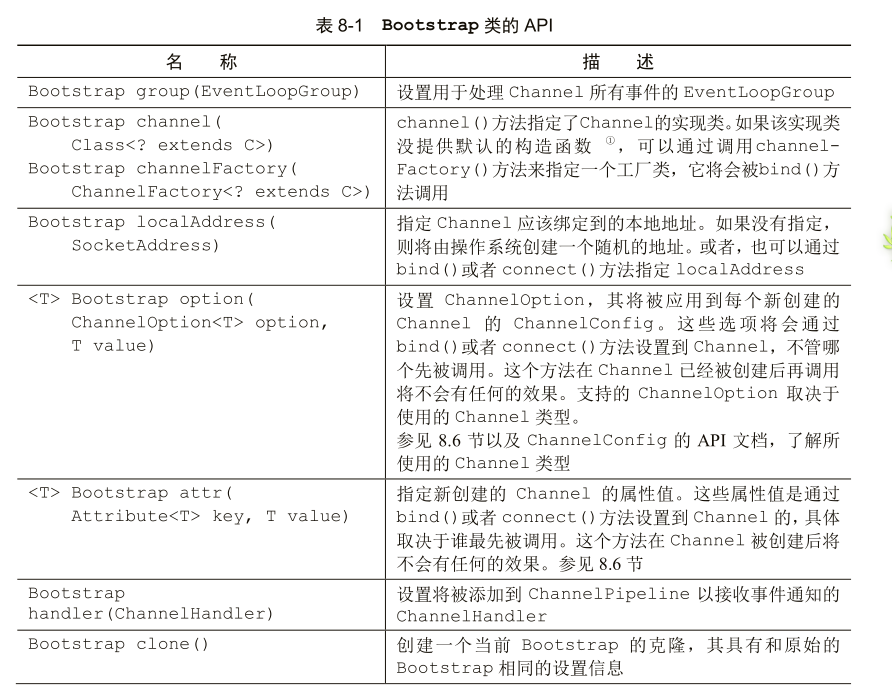
### 1.7.2 引导类的层次结构

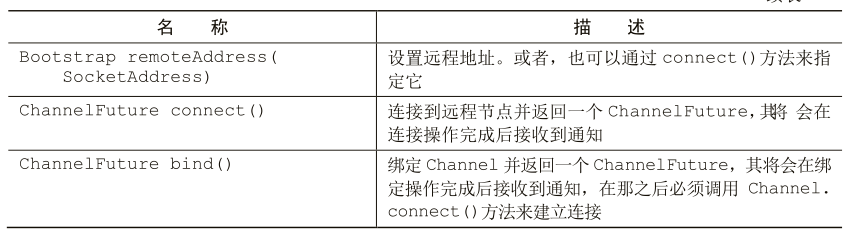
图1



### Bootstrap的API

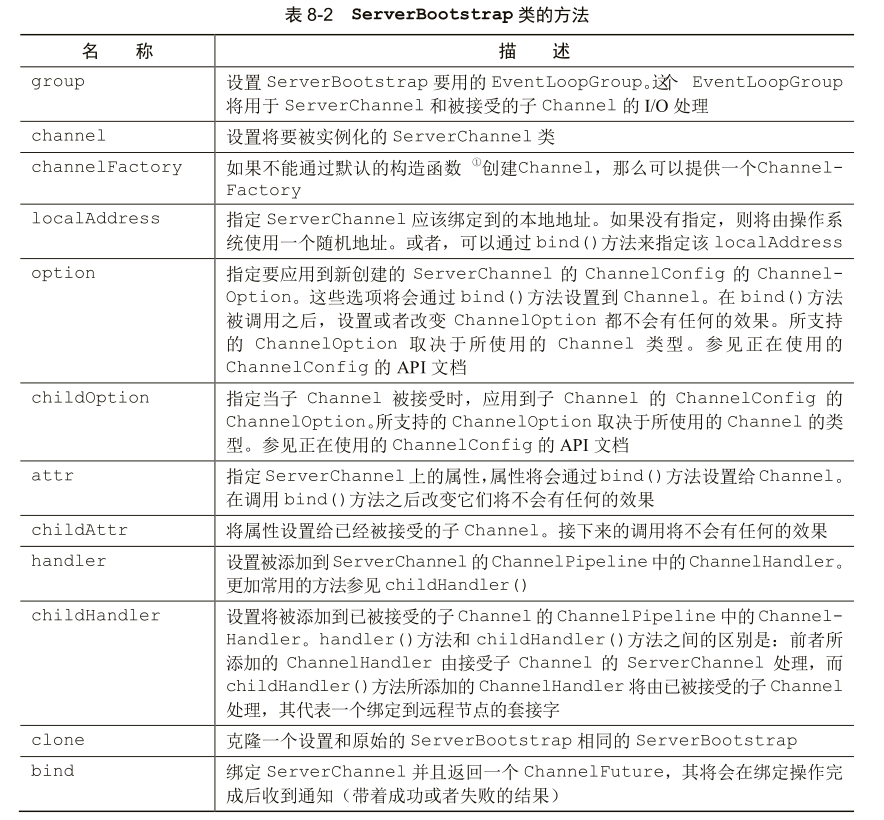
图1





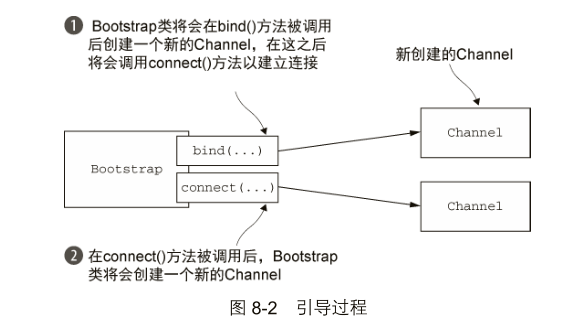
### ServerBootstrap的API

图1



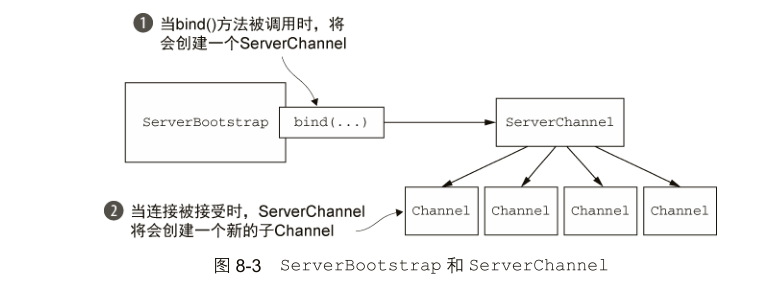
### Bootstrap创建Channel

图1



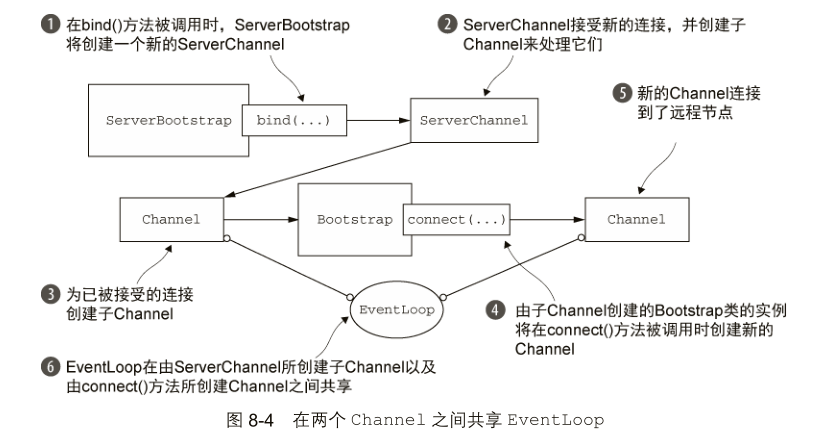
### ServerBootstrap创建ServerChannel和Channel

图1



### 从已被接受的Channel创建子Channel

图1



### 引导DatagramChannel

不需要connect()，只需要bind()，因为UDP是无连接的协议。

图1



## ByteBuf

Netty的ByteBuf是NIO的BtyeBuffer的替代品。

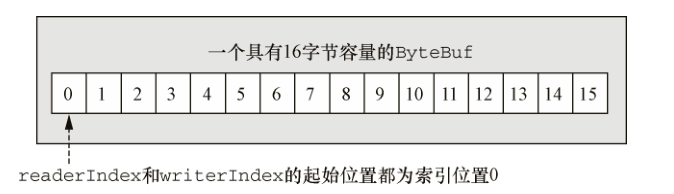
ByteBuf通过2个组件暴露：abstract class ByteBuf和interface ByteBufHolder。

### 两个索引

一个ByteBuf会有2个不同的索引：读索引和写索引。

（注意：read和write开头的方法会推进相应的索引，但set和get开头的方法不会）

图1

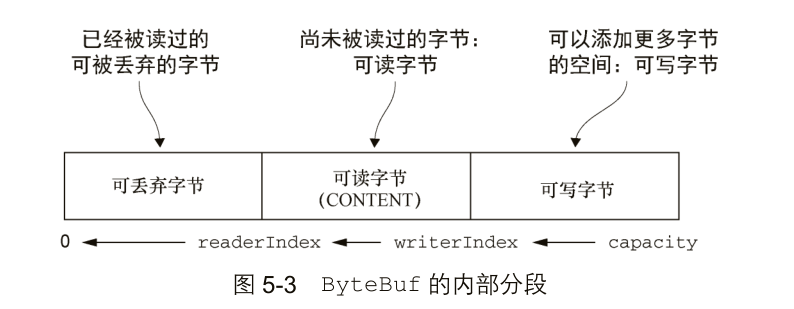


0~readerIndex：已读（可丢弃）

readerIndex~writerIndex：未读（可读）

writerIndex~capacity：可写

图2

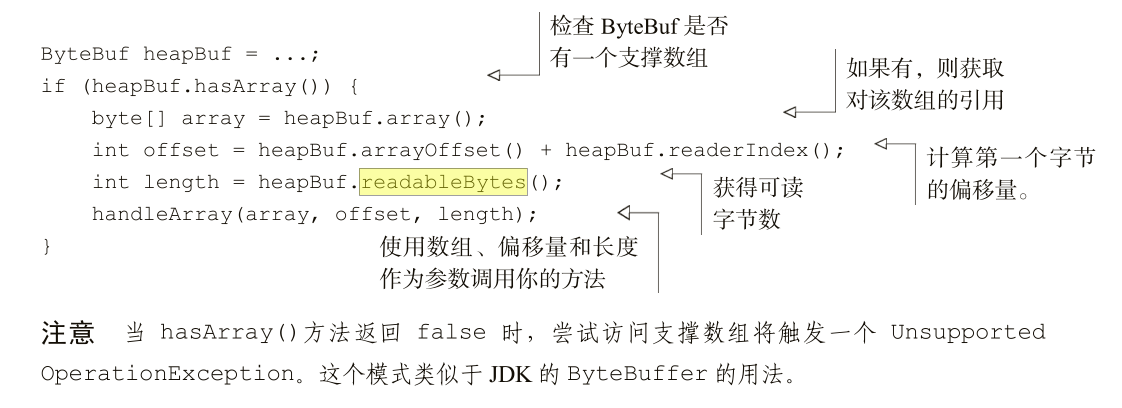


### 堆**缓冲区**（支持数组）

堆缓冲区模式：ByteBuf会保存在JVM的堆中（其实就是NIO说的用户空间）。

（注意：由于ByteBuf获取array后，这个array不一定和ByteBuf对齐，可能是因为array中逻辑分成了多个ByteBuf，所以偏移量offset需要加上arrayOffset()的值）

图1



缺点：

1. 读写数据需要先将堆中的ByteBuf数据复制到直接缓冲区（其实就是NIO的内核空间）。

### 直接**缓冲区**

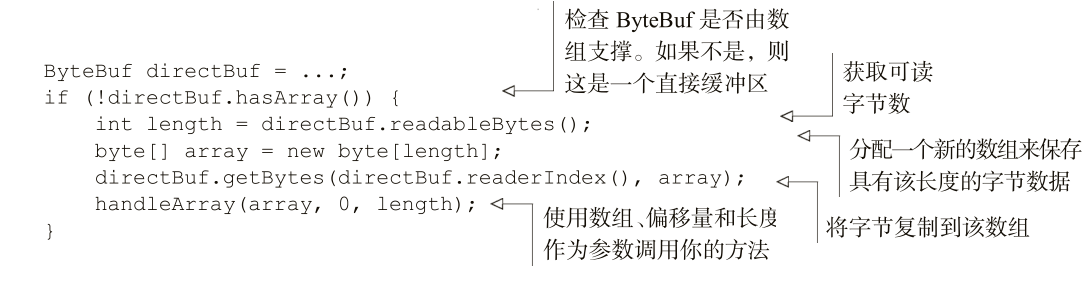
直接缓冲区模式：ByteBuf会保存在直接缓冲区中（其实就是NIO的内核空间）。

（注意：直接缓冲区因为不是在JVM的堆上的，所以没有array()方法直接变成一个数组）

缺点：

1. 分配和释放比较昂贵。
2. 因为数据不是在堆中，如果要获取需要复制。

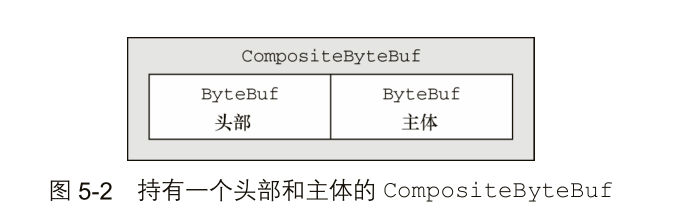
图1



### 复合**缓冲区**

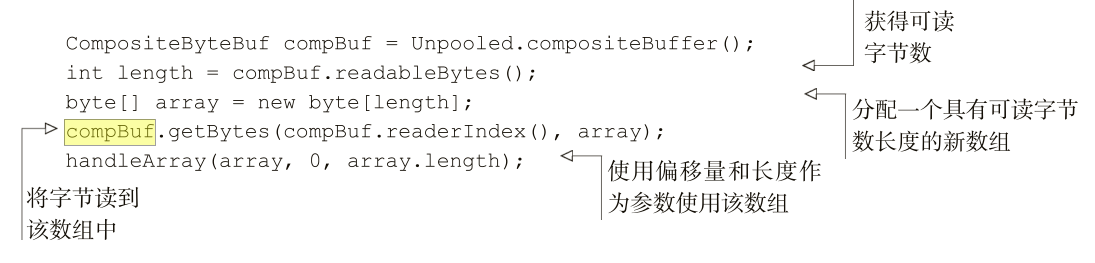
复合缓冲区由头部和主体2个ByteBuf组成。

图1



注意：由于CompositeByteBuf可能不支持访问其支撑数组，可以只能通过直接缓冲区的方式访问CompositeByteBuf。

图2

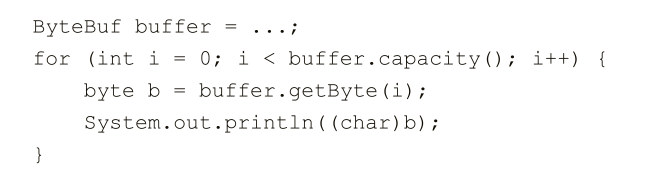


优点：

1. 适用于很多消息共用消息头的情况。
2. 由于Netty的优化，尽可能消除了使用直接缓冲区带来的惩罚。

### getByte(int i)随机**访问**索引

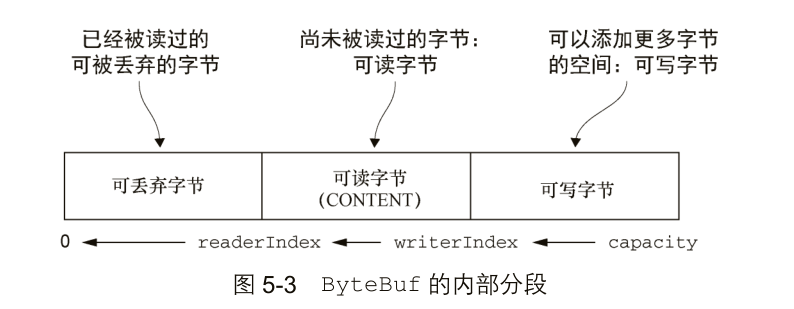
图1



### **discardReadBytes**()丢弃已读分段

原本可丢弃分段如图1，丢弃时需要将可读分段复制到缓冲区的最开始位置，并将readerIndex重置为0，并减少writerIndex的值。

图1



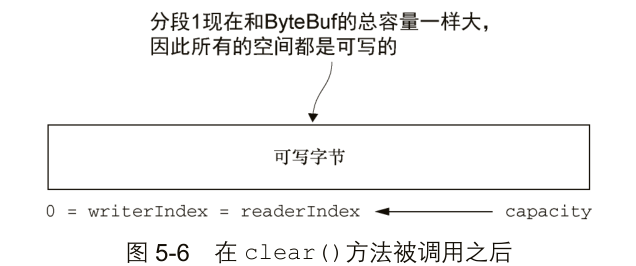
缺点：由于需要复制可读分段到缓冲区的最开始位置，效率会比NIO的ByteBuffer的close()简单的修改指针值要低。

### clear()重置2个索引值

clear()只是简单的将readerIndex和writerIndex都重置为0，并不会删除数据。

（注意：clear()之后，由于writerIndex为0，所以整个缓冲区都是可写的）

图1

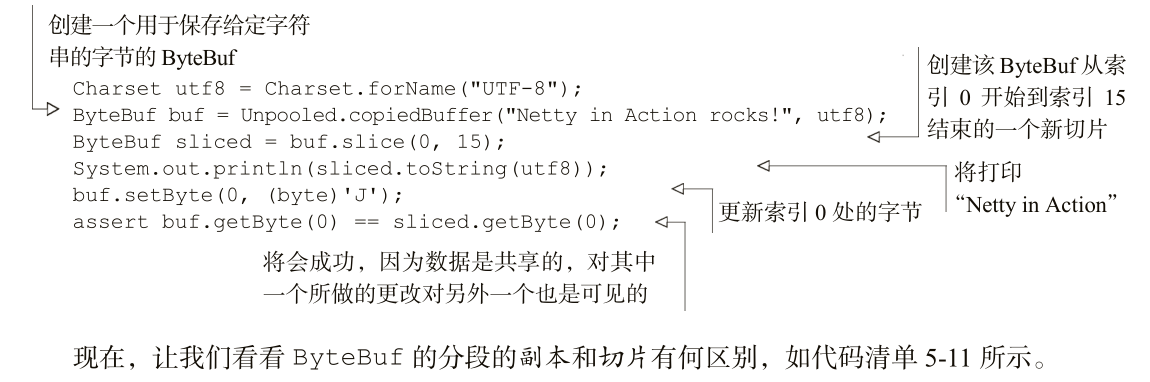


### 派生**缓冲区**

派生缓冲区由底层的缓冲区派生，可能是一个子集，可能是一个不可变的视图，这要看具体的派生方法。

（注意：派生缓冲区和底层的缓冲区是共享存储空间的，所以创建成本很低廉）

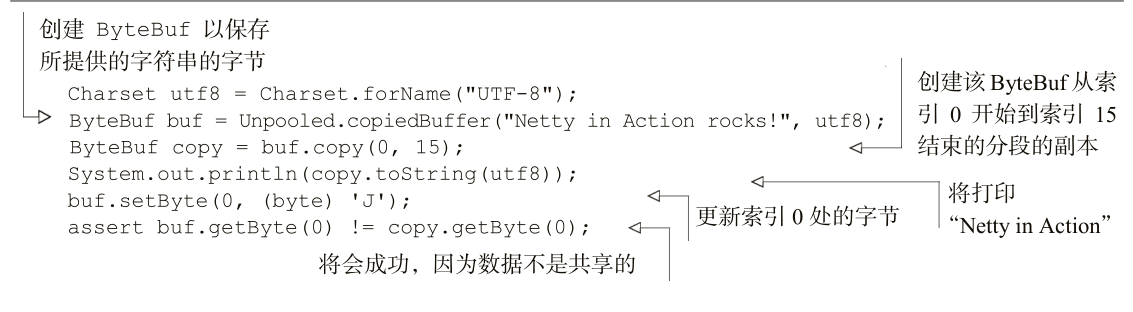
图1



派生缓冲区和副本缓冲区的区别：

1. 派生缓冲区和底层的缓冲区是共享存储空间的；而副本缓冲区是独立的存储空间。
2. 修改派生缓冲区同时会修改底层的缓冲区；修改副本缓冲区不会影响底层的缓冲区，如图2是副本缓冲区。

图2



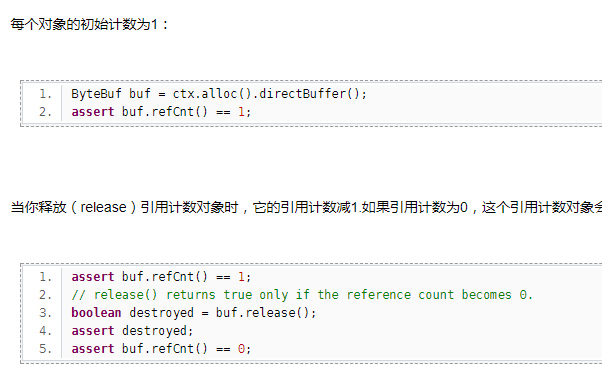
### 引用计数器

从Netty4开始，对象的生命周期由它们的引用计数器来管理。abstract class ByteBuf和interface ByteBufHolder都实现了interface ReferenceCounted。一个ReferenceCounted的实例的引用计数从1开始，当减少到0，该实例就会被最后引用这个实例的那个引用释放。

调用release()，引用计数会减一；调用retain()，引用计数会加一。

（注意：如果访问了引用计数是0的对象，会抛出IllegalReferenceCountException异常）

图1



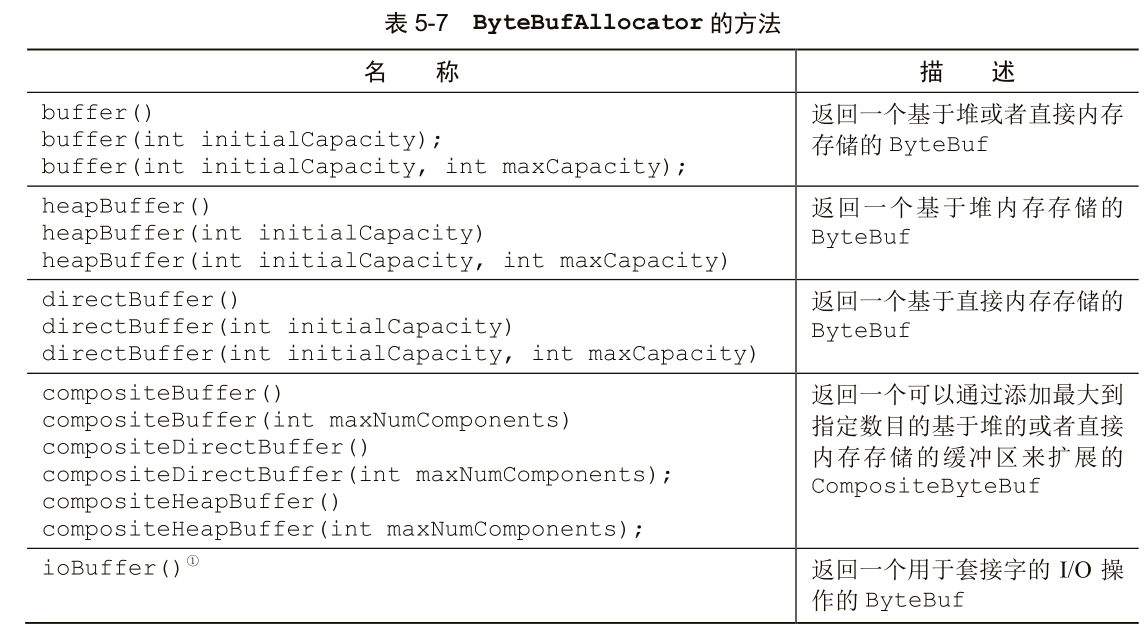
## ByteBuf 分配（实例化）

ByteBuf分配其实就是实例化一个ByteBuf对象。

### ByteBufAllocation接口按需分配

ByteBufAllocation接口有2个实现类：PooledByteBufAllocation和UnPooledByteBufAllocation。

图1



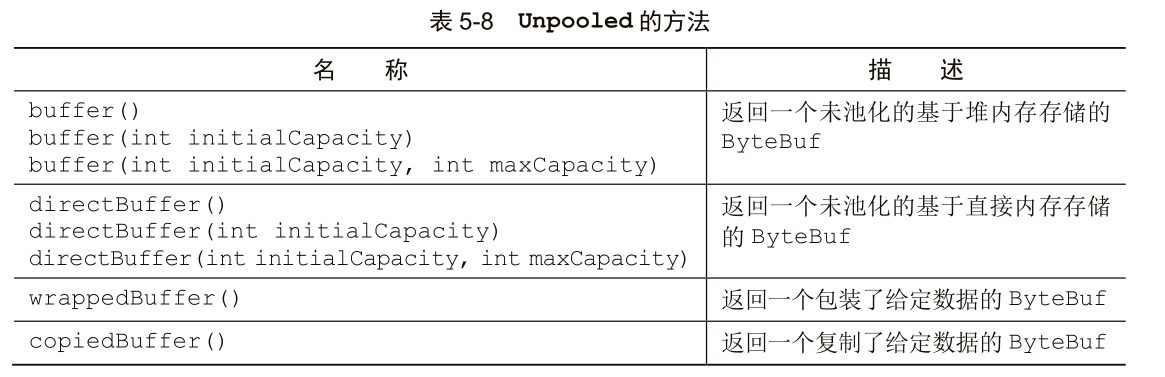
PooledByteBufAllocation和UnPooledByteBufAllocation的区别：

1. PooledByteBufAllocation池化ByteBuf实例；UnPooledByteBufAllocation不池化ByteBuf实例

### Unpooled**缓冲区**

有时候拿不到ByteBufAllocation的引用，可以使用Unpooled这个工具类来创建未池化的ByteBuf实例。

图1



### 1.9.3 ByteBufUtil静态方法操作ByteBuf

ByteBufUtil静态方法操作ByteBuf，与ByteBuf是否池化没关系。

# 2 编码器和解码器

## 2.1 编码器和解码器的区别

1.编码器将Java对象编码成二进制字节，解码器将二进制字节解码成Java对象。

2.编码器用在出站，解码器用在入站。

## 2.2 解码器

远程节点发过来的入站消息是二进制字节数组ByteBuf，解码器循环从ByteBuf中读取Java对象（每次读取出一个Java对象，直到ByteBuf没有可读数据）。

图1

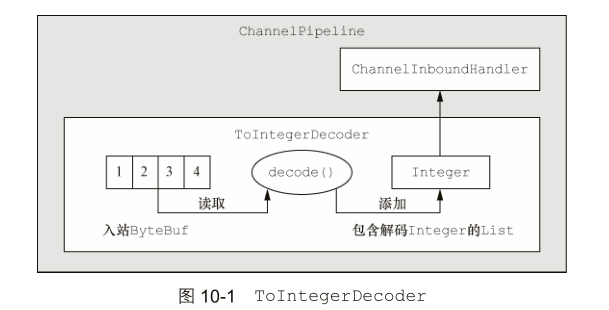


图2



### 2.2.1 ReplayingDecoder

ReplayingDecoder传入的ByetBuf是ReplayingByteBuf。ReplayingByteBuf不需要判断是否有字节可用，如果没有可用字节会抛出一个Error。

图1



ReplayingDecoder的缺点：

1. 不是所有的Bytebuf都支持ReplayingByteBuf，如果不支持会抛出一个 UnsupportedOpertionException。
2. ReplayingDecoder的速度稍慢于ByteToMessageDecoder。

### 2.2.2 MessageToMessageDecoder

MessageToMessageDecoder将一种POJO类型转换成另一种POJO类型。

图1

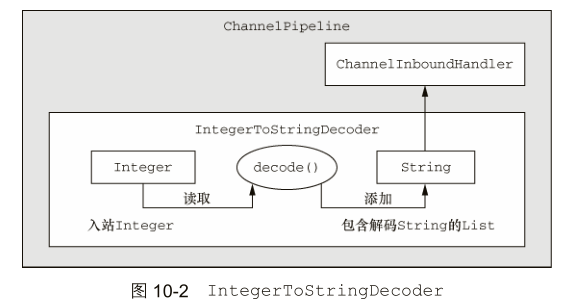


图2



### 2.2.3 TooLongFrameException 防止解码器缓冲大量数据以耗尽内存

设置一个阈值，当缓冲区大小超过整个阈值，抛出TooLongFrameException ，从而防止解码器缓冲大量数据以耗尽内存。

图1



## 2.3 编码器

客户端发送的出站消息是Java对象，必须编码成二进制字节数组才能在网络上传输。

图1

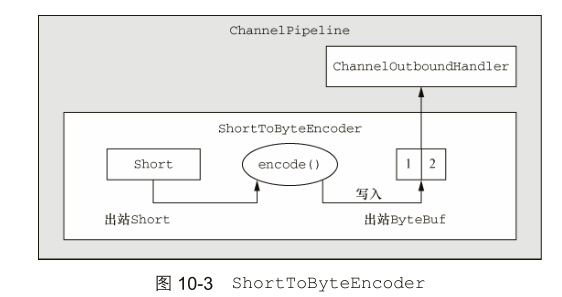


图2

