# 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

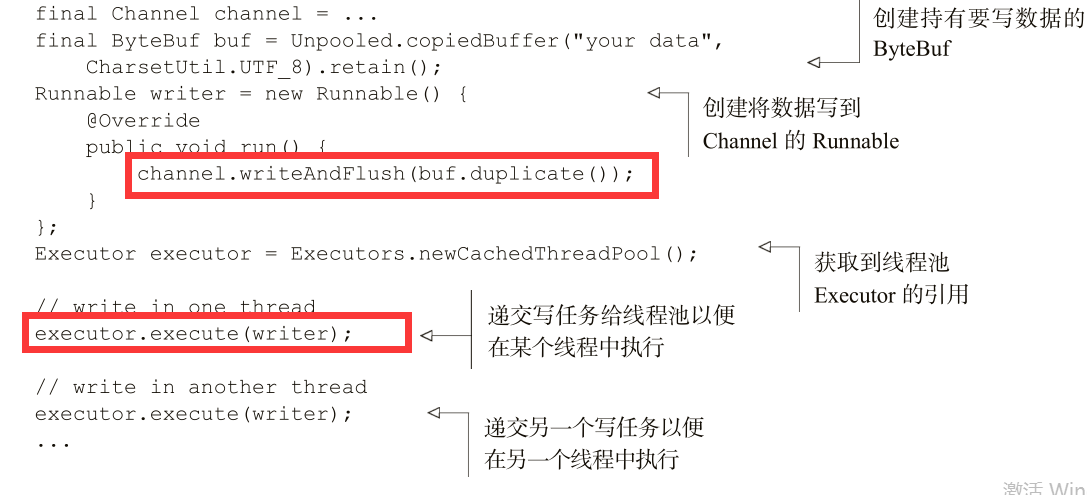
图1



### Netty的**Channel**是线程安全的

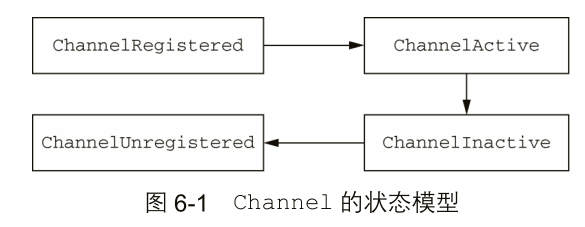
多个线程可以使用同一个Channel实例读写。

图1



### Channel的**生命**周期

图1



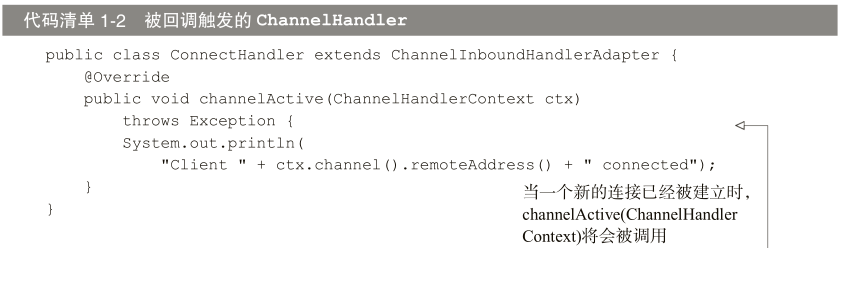
## ChannelHandler

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。

## 回调

回调就是ChannelHandler中的方法，相关的事件触发时，回调方法会被调用。如图1当新的连接建立时，ChannelHandler的channelActive()方法会被调用。

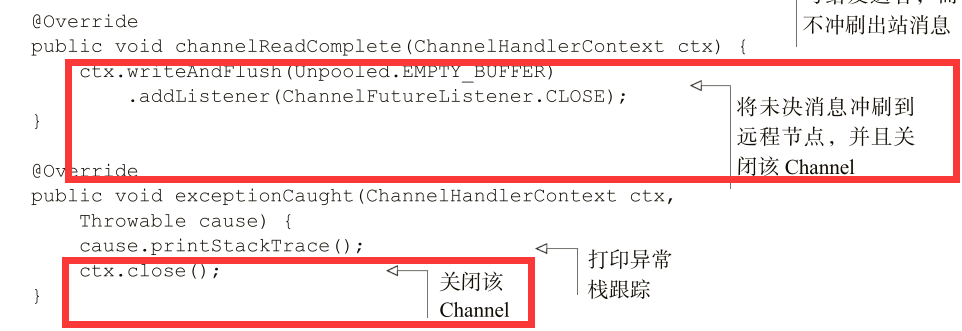
图1



## ChannelHanderContext

ChannelHanderContext代表了ChannelHandler和ChannelPipeLine之间的绑定。它可以获取底层的Channel，但它主要是用来写出站数据。

图1



### 数据直接写入Channel和用ChannelHanderContext写入的区别

数据直接写入Channel，事件会从ChannelPipeline的尾端开始流动；用ChannelHanderContext写入，事件会从ChannelPipeline的下一个ChannelHandler开始流动。

## **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

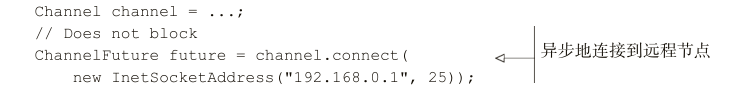
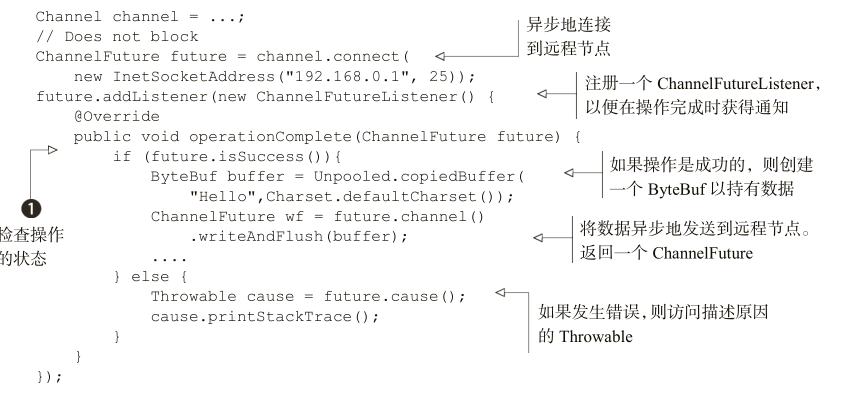


图2

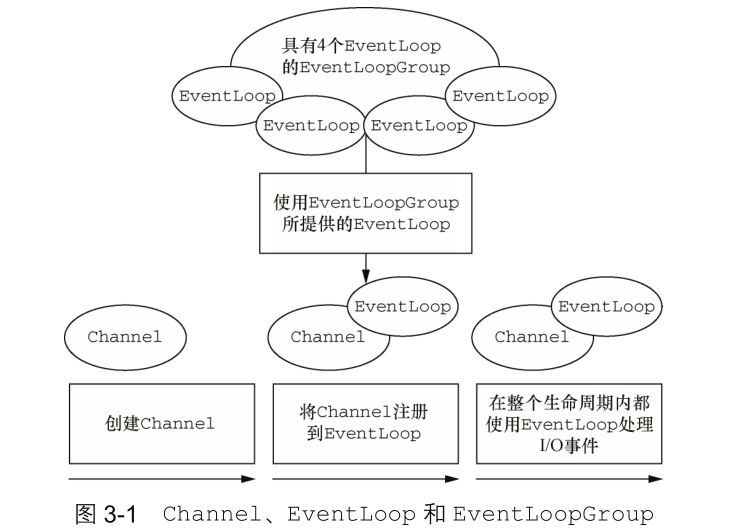


## **EventLoopGroup，**EventLoop，Channel的关系

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。

图1



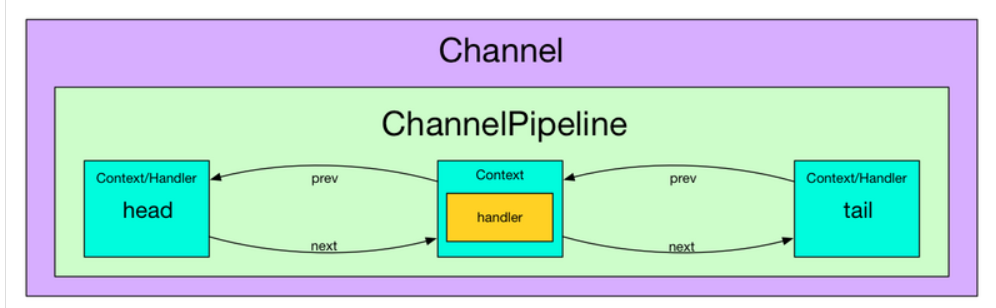
## ChannelPipeLine

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。

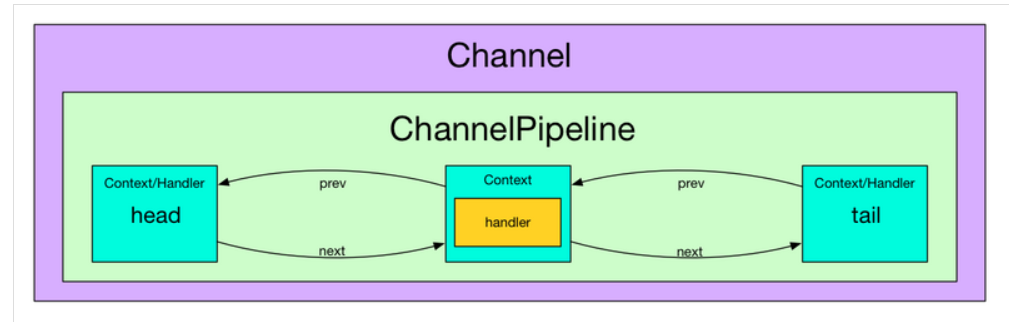
图1



### 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



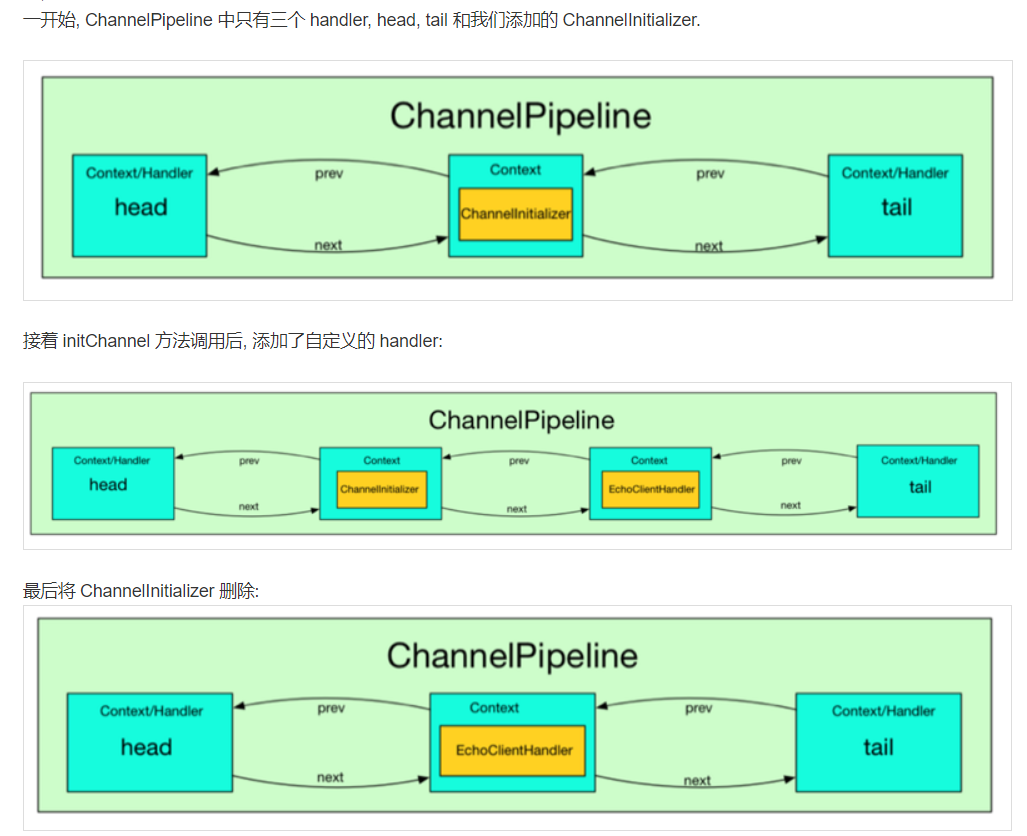
### 入站和出站的区别

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事件是某个事件发生了，需要通知，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）。

## ChannelInitializer

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



## Bootstrap

Bootstrap是引导程序，一种用于客户端（Bootstrap），另一种用于服务器（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。

## ByteBuf

Netty的ByteBuf是NIO的BtyeBuffer的替代品。

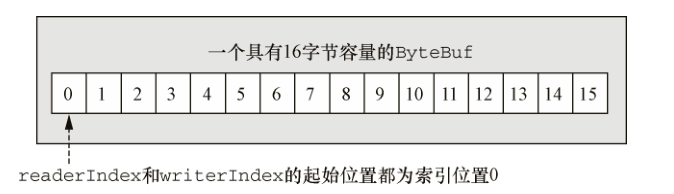
ByteBuf通过2个组件暴露：abstract class ByteBuf和interface ByteBufHolder。

### 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

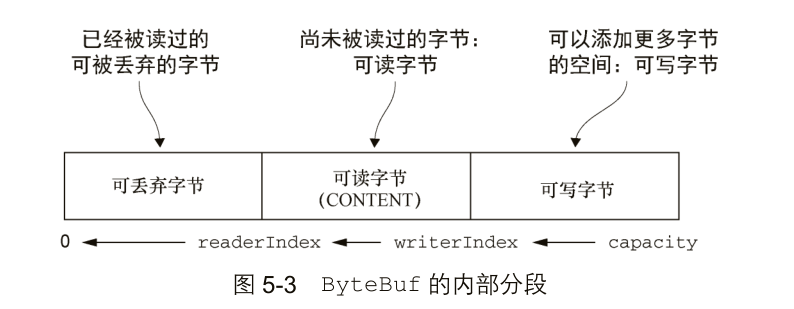


0~readerIndex：已读（可丢弃）

readerIndex~writerIndex：未读（可读）

writerIndex~capacity：可写

图2

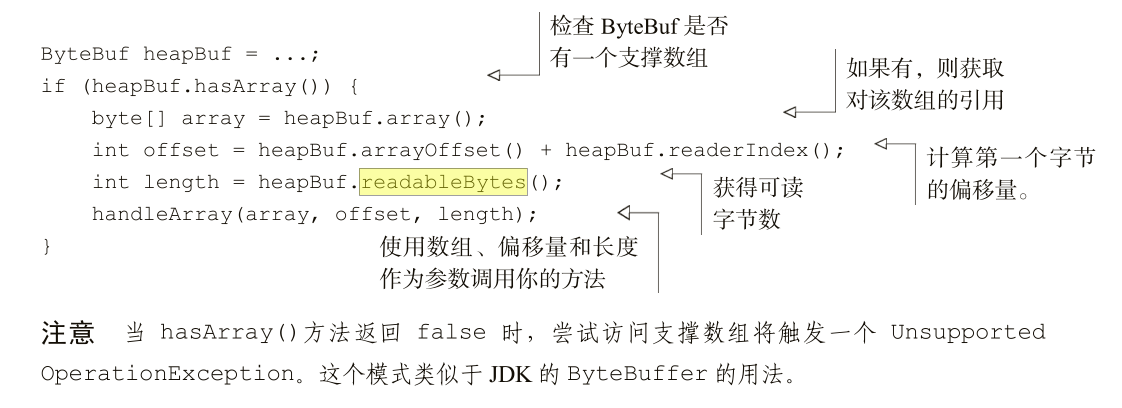


### 堆**缓冲区**（支持数组）

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**

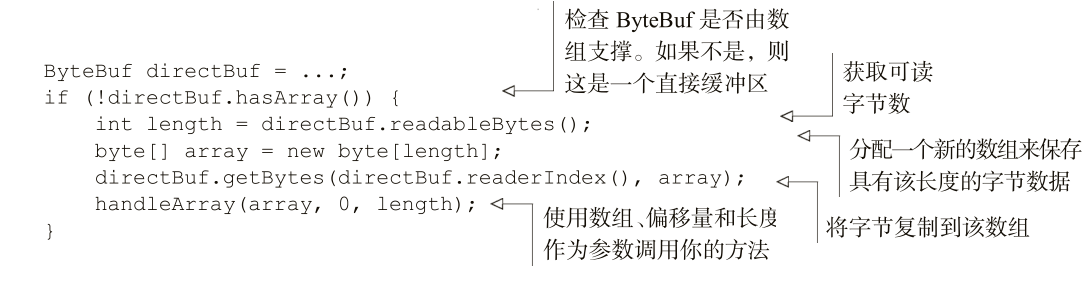
直接缓冲区模式：ByteBuf会保存在直接缓冲区中（其实就是NIO的内核空间）。

（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。

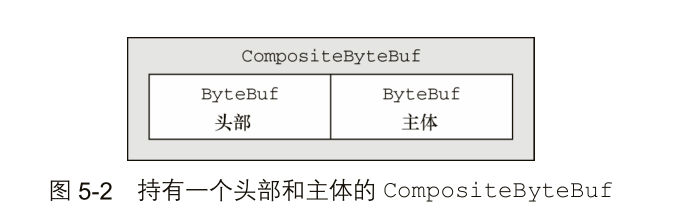
图1



### 复合**缓冲区**

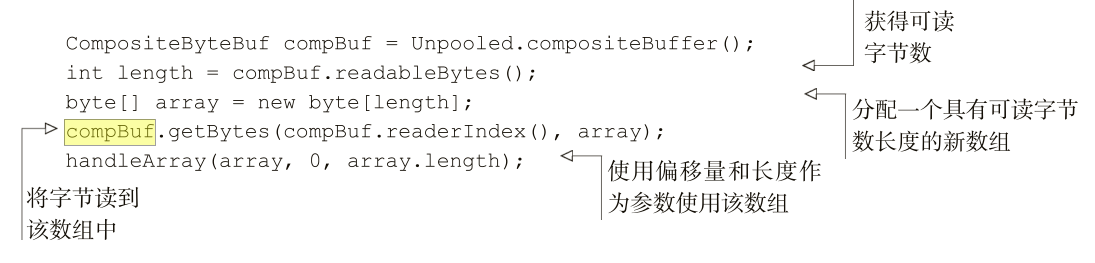
复合缓冲区由头部和主体2个ByteBuf组成。

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

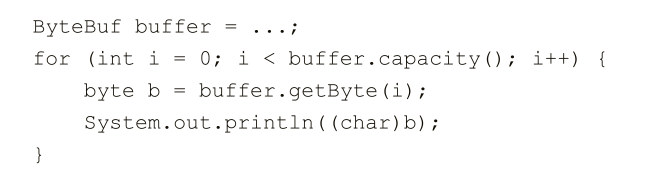


优点：

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

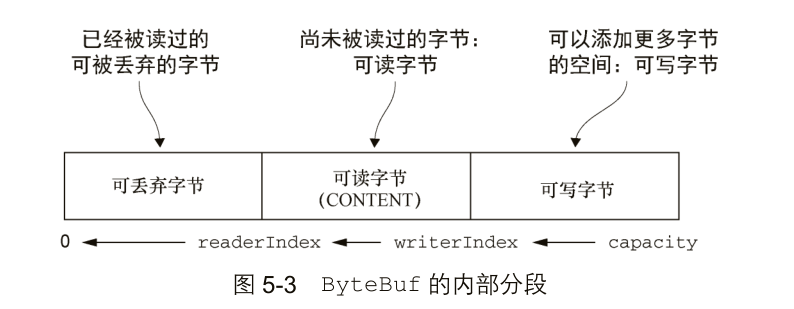
图1



### **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



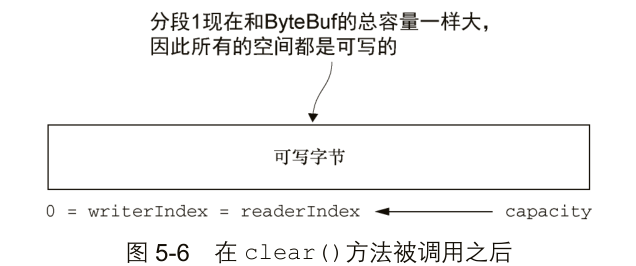
缺点：由于需要复制可读分段到缓冲区的最开始位置，效率会比NIO的ByteBuffer的close()简单的修改指针值要低。

### 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

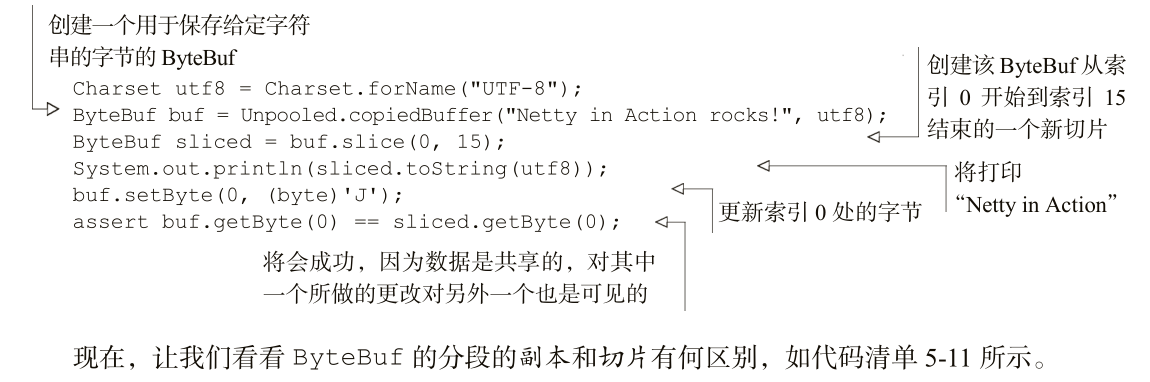


### 派生**缓冲区**

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）

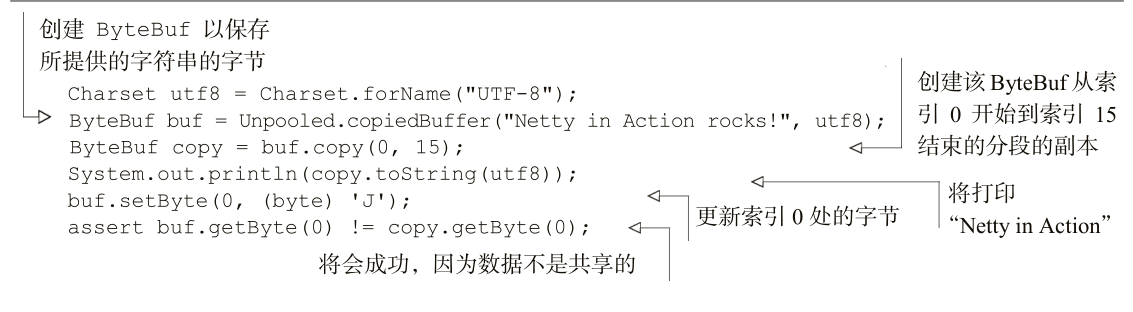
图1



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



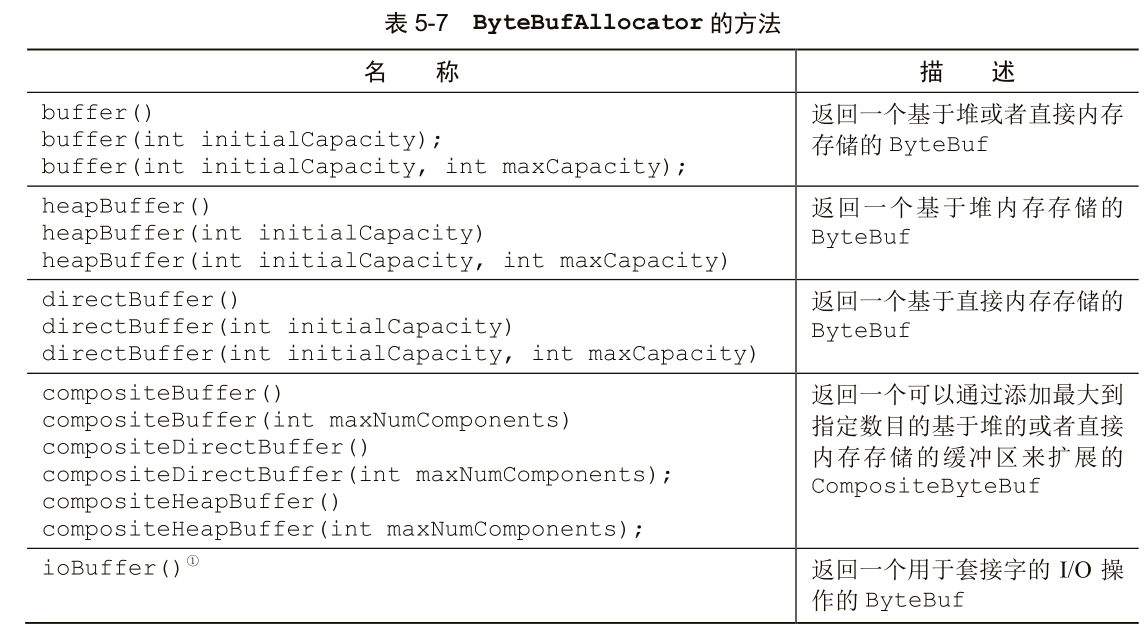
## ByteBuf 分配（实例化）

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

