Netty-ByteBuf 学习

本章主要内容

- ByteBuf
- ByteBufHolder
- ByteBufAllocator
- 使用上述接口

传输数据时一般都会使用一个缓冲区包装数据。Java的NIO有自己的Buffer类，，它们实现的功能有限并且没有优化过。使用JDK的ByteBuffer往往是比较麻烦也比较复杂的。缓冲区是网络应用非常重要的一个组件，有必要提供给开发者，并且应该是API的一部分。

幸运的是，Netty提供了功能强大的缓冲区实现，它用来表示一个字节序列，帮助开发者操作原始字节数据或者自定义的Java对象。Netty中的缓冲区叫ByteBuf，实际上等价于JDK的ByteBuffer。ByteBuf的作用是通过Netty的管道传输数据。它从根本上解决了JDK缓冲区的问题，并且Netty应用开发者经常会使用到它，这些因素都让它有很强的生产力。在与JDK的缓冲期相比有很大优势。

因为缓冲区传送数据都是通过Netty的ChannelPipeline和ChannelHandler，所以Netty应用都会使用到缓冲区API。

一、Buffer API

Netty的Buffer API主要有两个接口：

- ByteBuf
- ByteBufHolder

Netty缓冲API有以下几个优点：

- 如果有需要你可以自定义缓冲类型
- 内置复合缓冲类型实现零拷贝
- 容量是按需扩展的，类似StringBuffer
- 不需要通过调用flip()方法切换读写模式
- 读索引和写索引是分开的
- 方法调用是链式结构的
- 引用技术功能自动释放资源
- 池优化技术

  上面这些我们后面都会介绍，包括池优化技术。现在我们先从最常用的字节容器开始。

二、字节容器ByteBuf

当你需要通过网络与对端如数据库交换数据时，都是通过字节来沟通的 ByteBuf是一个可以让你有效的读写字节数据的数据容器。为了易于操作，它使用了两个索引：一个用来读一个用来写。这样你只需要调整读索引就可以很方便的重复读取容器内的数据了。

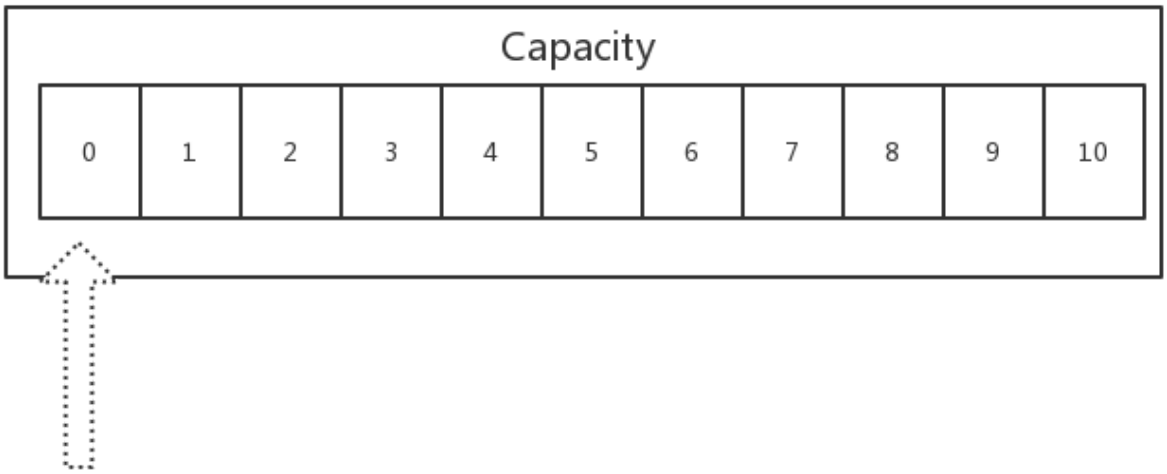
2.1、工作方式

当向ByteBuf写数据时，写索引会根据根据写入的字符数增加同样数值。当你开始读数据时，读索引也会增加，只有读写索引处于同一位置。然后ByteBuf就不可读了，如果继续读的话就会抛出IndexOutOfBoundsException异常，类似读数组越界一样。

当调用任何以read或write开头的方法时，读写索引都会自动增长。也有一些相对操作方法如get或set，这些方法不会去移动索引，但会去操作给定的相对索引。

ByteBuf是有容量上限的，如果将写索引移到超过容量上限的位置就会出现异常。默认的容量上限是Integer.MAX_VALUE。

下图展示了ByteBuf的基本结构图。



从上图可以看出，ByteBuf很类似字节数组，最明显的区别就是ByteBuf有分开了读写索引进行访问数据。

2.2、不同类型的ByteBuf

常用的ByteBuf一共有三种，其他的也有不少，不过一般开发者很少会用到，都是Netty内部使用。可能你还会实现自己的缓冲区，不过这个不在本章内容之内。首先来看看你可能最感兴趣的缓冲。

堆缓冲

最常用的ByteBuf是存储数据在JVM堆内存上的缓冲区。这种缓冲区内部实现就是一个数组。下面展示一下它的基本用法。

```
ByteBuf heapBuf = ...;
//检测ByteBuf是否有数组
if (heapBuf.hasArray()) {
    //获取数组引用
    byte[] array = heapBuf.array();
    //计算第一个字节所在位置
    int offset = heapBuf.arrayOffset() + heapBuf.position();
    //获取可读字节总数
    int length = heapBuf.readableBytes();
    //使用自己的业务逻辑处理数据
    YourImpl.method(array, offset, length);
}
```

如果访问非堆内存ByteBuf的数组，就会抛出UnsupportedOperationException。因此访问ByteBuf数组之前应该使用hasArray()方法检测此缓冲区是否支持数组。

直接缓冲区

另一个ByteBuf的实现是直接缓冲区。直接的意思就是这个缓冲区分配的内存是在JVM堆内存外部。使用直接内存时你不会在JVM堆空间中看见它的使用量。所以你计算你的应用内存使用量时，直接内存区域也要算上，然后限制它的最大使用量，防止系统内存不够用出现错误。通过网络传输数据时使用直接缓冲性能是很高的。

下面的一部分代码展示了如何通过数组的方式访问直接缓冲区的数据。

```
ByteBuf directBuf = ...;
//检查缓冲区是否有数组，返回flase就是直接缓冲
if (!directBuf.hasArray()) {
   //获取可读字节数量
    int length = directBuf.readableBytes();
    //初始化相同长度的直接数组
    byte[] array = new byte[length];
    //读取数据到数组中
    directBuf.getBytes(array);
    //用户业务逻辑操作字节数组
    YourImpl.method(array, 0, array.length);
}
```

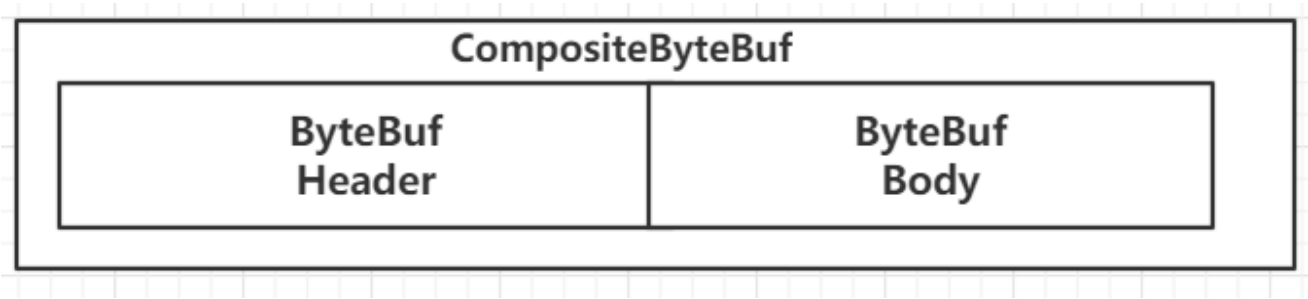
可以看出，直接缓冲区需要将数据复制到数组，如果你想直接通过字节数组访问数据，用堆缓冲区更好一些。

组合缓冲区

现在来介绍一个组合缓冲区，它的名字叫CompositeByteBuf。人如其名，它允许你将不同类型的ByteBuf组合在一起并且提供访问他们的方式。CompositeByteBuf也像是各种类型ByteBuf的视图，它的hasArray()方法返回的false，因为它里面可能包含多个堆缓冲或直接缓冲。

例如，一条消息可能由两个部分组成：消息头和消息体。模块化应用中，这两部分可能是由不同模块的产生并组合在一起然后再发送。很多时候，你发送的消息往往只修改一部分，比如消息体一样，改变消息头。这里就不用每次都重新分配缓存了。

上面这种场景就很适合CompositeByteBuf，不需要内存拷贝，而且使用的是和非组合缓冲相同的API。下图展示了CompositeByteBuf组合消息头和消息体的结构。



下面的伪代码展示这两个方式。

```
        //第一种使用数组组合
        ByteBuffer[] message = new ByteBuffer[] { header, body };
        //第二种将头尾都复制到新的ByteBuffer
        ByteBuffer message2 = ByteBuffer.allocate(
        header.remaining()+ body.remaining();
        message2.put(header);
        message2.put(body);
        message2.flip();
```

上面的方式都有很明显的缺点：使用数组处理，代码量及API都比较复杂。使用数据复制代价有很大，浪费时间和资源。下面我们看看Netty的CompositeByteBuf怎么处理这种情况。

```
        CompositeByteBuf compBuf = ...;
        ByteBuf heapBuf = ...;
        ByteBuf directBuf = ...;
        //将数据都放入到CompositeByteBuf
        compBuf.addComponent(heapBuf, directBuf)
        .....
        //移除索引是0的数据，就像List
        compBuf.removeComponent(0);
        //遍历组合缓冲中的缓冲实例
        for (ByteBuf buf: compBuf) {
            System.out.println(buf.toString());
        }
```

CompositeBytebuf也是不能通过数组的方式直接访问其里面的数据。下面展示CompositeBytebuf怎么访问数据，

```
        CompositeBuf compBuf = ...;
        //检测是否有数组，CompositeBuf会返回false
        if (!compBuf.hasArray()) {
            //获取可读字节总数
            int length = compBuf.readableBytes();
            //初始化数组长度与可读总数相同
            byte[] array = new byte[length];
            //将数据读到字节数组
            compBuf.getBytes(array);
            //处理数据的业务逻辑
            YourImpl.method(array, 0, array.length);
        }
```

CompositeByteBuf是ByteBuf的子类型，所以在它上面可以操作一些ByteBuf的方法，当然，也有一些它自己的方法。

三、ByteBuf的字节操作

ByteBuf提供了很多读写内容的方法。

3.1、随机访问

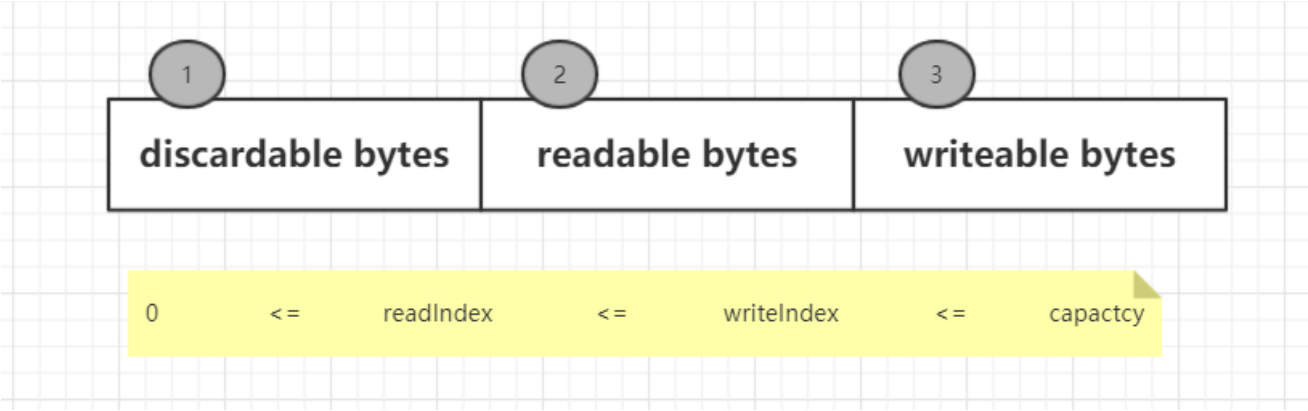
和原始字节数组一样，ByteBuf的索引也是从0开始的。也就是说它的第一个字节的位置是0，最后一个字节的位置是容量-1。例如下面的代码，可以直接遍历ByteBuf的所有字节，而不用去管它内部如何实现的。

```
ByteBuf buffer = ...;
for (int i = 0; i < buffer.capacity(); i ++) {
byte b = buffer.getByte(i);
System.out.println((char) b);
}
```

前面说过，使用set或get开头的方法不会增长它的读写索引，所以如果你使用上面的方式读取数据，可以写代码更新读写索引。

3.2、顺序访问

ByteBuf提供了两个索引来支持顺序访问的需求，一个是readerIndex读索引支持读操作的，一个是writerIndex写索引支持写操作的，各自之间分工很明确。而JDK的ByteBuffer只有一个索引，所以每次切换读写模式的时候你都需要调用一下flip()方法。下图展示了ByteBuf被两个索引分成三个区域的结构。



3.3、可读字节内容(实际内容)

ByteBuf中的实际内容存储在可读字节区域。以read或skip开头的方法，你读了多少数量的字节，相应的读索引就会增加多少

下面的代码展示了如何读取ByteBuf中的所有可读内容。

```
ByteBuf buffer = ...;
while (buffer.readable()) {
    System.out.println(buffer.readByte());
}
```

3.4、可写字节

这块区域是需要填充的未定义区域。以write开头的方法会从当前的writerIndex位置写入数据，并且写了多少数据这个writerIndex也会增加相应的数量。如果写操作方法的参数也是一个ByteBuf并且没有指定源索引，则这个参数的读索引也会增长相应的数量。

如果没有足够的可写区域了还继续使用写操作就会抛出IndexOutOfBoundException异常。新分配的ByteBuf的写索引默认值也是0。

下面的代码展示了用Int类型数据填满可写区域。

```
ByteBuf buffer = ...;
while (buffer.writableBytes() >= 4) {
    buffer.writeInt(random.nextInt());
}
```

3.5、派生缓冲区

派生缓冲区，也就是创建一个已经存在的缓冲区的视图，可以调用duplicate(),slice(),slice(int, int),

readOnly()和order(ByteOrder)等方法实现。派生缓冲区会产生自己的读写索引和其他标记索引，但是他们共享内部的数据。因为它们共享内部的数据，所以创建派生缓冲是没有什么性能损耗的，而且是比较好的方式，例如你想拥有一个缓冲区的切片。

如果需要复制一个缓冲区，可以使用copy()或copy(int, int)方法。下面的代码展示了怎么获取一个ByteBuf的切片。

```
        Charset utf8 = Charset.forName("UTF-8");
        //根据给定的字符串内容创建一个ByteBuf
        ByteBuf buf = Unpooled.copiedBuffer("xxxx", utf8);
        //创建ByteBuf的切片，起始位置是0，截止位置是14
        ByteBuf sliced = buf.slice(0, 14);
        //打印切片内容，正常应该是"xxxx"
        System.out.println(sliced.toString(utf8);
        //修改索引0内容
        buf.setByte(0, (byte) 'J');
        //断言会成功，因为他们共享内部的数据，其实修改的就是他们内部指向的共同数据块
        assert buf.get(0) == sliced.get(0);
```

可以发现，切片和原始ByteBuf里面的内容其实是同一块内存区域。现在我们来看看怎么复制ByteBuf，以及它和切片ByteBuf有什么不同，请看下面的代码。

```
        Charset utf8 = Charset.forName("UTF-8");
        //同样根据给定字符串创建ByteBuf
        ByteBuf buf = Unpooled.copiedBuffer("xxx!", utf8);
        //复制ByteBuf 0-14位置的内容
        ByteBuf copy = buf.copy(0, 14);
        //输出复制的ByteBuf内容，正常应该是"xxx"
        System.out.println(copy.toString(utf8);
        //同样修改内容
        buf.setByte(0, (byte) 'J');
        //断言会成功，因为复制缓冲和原始缓冲并没有共享数据
        assert buf.get(0) != copy.get(0);
```

API基本上都是一样的，不过不同的方式派生的ByteBuf，内部也会有细微差别

一般情况下建议使用切片，除非必须不共享数据才使用复制方式。因为复制ByteBuf需要进行内存复制操作，不仅消耗更多资源，执行方法也会更耗时。

3.6、读写操作

ByteBuf的读写操作共有两大类：

- 基于索引的，指定索引读写相应位置的数据
- 从当前索引读写数据，然后增长读写索引

  先来看一下这些方法，下面列出来的都是比较常用的方法，如果想看其他方法，请参考API DOC。先看一下读操作的。

名称	描述
getBoolean(int)	获取指定位置的Boolean类型数据
getByte(int)	获取指定位置的字节内容
getUnsignedByte(int)	获取指定位置的无符号字节内容
getMedium(int)	获取指定位置的24位整数内容
getUnsignedMedium(int)	获取指定位置的无符号24位整数内容
getInt(int)	获取指定位置的Int类型内容
getUnsignedInt(int)	获取指定位置的无符号Int类型内容
getLong(int)	获取指定位置的Long类型内容
getUnsignedLong(int)	获取指定位置的无符号Long类型内容
getShort(int)	获取指定位置的Short类型内容
getUnsignedShort(int)	获取指定位置的无符号Short类型内容
getBytes(int, ...)	从给定位置开始将数据转换到目标容器中

它的set操作和get操作都很类似，请看下面的表格。

名称	描述
setBoolean(int, boolean)	设置指定位置的Boolean类似数据
setByte(int, int)	设置指定位置字节数据
setMedium(int, int)	设置指定位置24位整数数据
setInt(int, int)	设置指定位置Int类型数据
setLong(int, long)	设置指定位置Long类型数据
setShort(int, int)	设置指定位置Short类型数据
setBytes(int,...)	从指定位置将源数据容器中转换过来

你可能会发现set方法没有无符号类型的。这是因为在设置数据的时候没有无符号的概念。上面已经介绍了相关的方法，现在我们来看看怎么在实际代码中使用它们。

```
        Charset utf8 = Charset.forName("UTF-8");
        //根据字符串创建ByteBuf
        ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);
        //打印位置0的字符，正常输出'N'
        System.out.println((char) buf.getByte(0));
        //查询现在的读写索引
```

```
        int readerIndex = buf.readerIndex();
        int writerIndex = buf.writerIndex();
        //修改位置0的数据
        buf.setByte(0, (byte)"B");
        //再次打印位置0的数据，这次应该是'B'
        System.out.println((char) buf.getByte(0));
        //set或get方法不会改变读写索引
        assert readerIndex = buf.readerIndex();
        assert writerIndex = buf.writerIndex();
```

接下来介绍到的读写操作就是read和write开头的方法，前面说过，这些方法也会影响到缓冲的读写索引。这里面用到最多的方法就是把ByteBuf当成一个流来读取数据。同样，write方法相当于在ByteBuf上追加数据。常用的read方法如下表格。

名称	描述
readBoolean()	读取当前读索引Boolean数据，同时读索引会加1
readByte()	读取当前读索引Byte数据，同时读索引会加1
readUnsignedByte()	读取当前读索引无符号Byte数据，同时读索引会加1
readMedium()	读取当前读索引24位整数数据，同时读索引会加1
readUnsignedMedium()	读取当前读索引无符号24位整数数据，同时读索引会加1
readInt()	读取当前读索引Int类型数据，同时读索引会加1
readUnsignedInt()	读取当前读索引无符号Int类型数据，同时读索引会加1
readLong()	读取当前读索引Long类型数据，同时读索引会加1
readUnsignedLong()	读取当前读索引无符号Long类型数据，同时读索引会加1
readShort()	读取当前读索引Short类型数据，同时读索引会加1
readUnsignedShort()	读取当前读索引无符号Short类型数据，同时读索引会加1
readBytes(int)	从当前读索引读取指定长度的字节内容，读索引也会增加指定长度

它的write方法基本上和read方法也是一一对应的。

名称	描述
writeBoolean(boolean)	在当前写索引位置写入Boolean类型数据，同时写索引加1
writeByte(int)	当前写索引写入Byte类型数据，同时写索引加1
writeMedium(int)	当前写索引写入24位整数数据，同时写索引会加3，24位=3字节
writeInt(int)	在当前写索引写入32位整数数据，同时写索引加4，32位=4字节
writeLong(long)	当前写索引写入64位整数数据，同时写索引加8，64位=8字节
writeShort(int)	当前写索引写入Short类型数据，同时写索引加2
writeBytes(ByteBuf src, int length)	从源数据中转换指定长度数据到当前写索引，同时写索引增加相应数量

现在我们来看看上面这些read和write方法在实际项目中的应用。

```
        Charset utf8 = Charset.forName("UTF-8");
        //根据字符串创建ByteBuf
        ByteBuf buf = Unpooled.copiedBuffer("xxx", utf8);
        //打印首位置字符，正常是'x'
        System.out.println((char) buf.readByte());
        //获取当前的读写索引
        int readerIndex = buf.readerIndex();
        int writerIndex = buf.writerIndex();
        //修改写索引0位置的数据
        buf.writeByte( (byte) '?');
        //写数据读索引不会变化，写索引会加1
        assert readerIndex = buf.readerIndex();
        assert writerIndex != buf.writerIndex();
```

一直忘记了说一件事，获取当前读写索引的方法是不会修改读写索引。上面已经介绍了很多读写数据的方法，当然还有一些其他很有用的方法，下面会介绍这些方法。

3.11、其他常用方法

还有一些常用的方法我们没有介绍到，下表列出来这些常用方法，并简要介绍它们的用法。

名称	描述
isReadable()	有一个字节可读就会返回true
isWritable()	有一个可写字节空间就会返回true
readableBytes()	返回可读字节内容总数
writablesBytes()	返回可写内容区域总数
capacity()	返回缓冲区目前容量，如果超过这个数值，缓冲区会扩容， 只到达到maxCapacity()的限制
maxCapacity()	返回缓冲区最大容量
hasArray()	如果缓冲区内部以数组存储返回true
array()	返回内部存储的数组，如果没有数组则 抛出UnsupportedOperationException

前面介绍的都是Java基础类型，但很多时候我们网络应用需要处理普通Java对象，也就是Java Bean。需要存储和检查，而且在容器中按照顺序存储也很重要。因此，Netty提供了另一个数据容器

MessageBuf。

四、ByteBufHolder

你有一个对象，里面很多属性，但是都是通过字节去传输的。例如，HTTP协议中的相应数据就很符合这种情况。HTTP响应数据有很多属性，如果状态码，Cookie等等，而且它实际的内容都是以字节方式传输的。

因为这种情况很普遍，所以Netty抽象了一个接口ByteBufHolder。这样做的好处就是Netty可以优化分配ByteBuf，例如使用池技术，也能够自动释放这部分资源。

ByteBufHolder实际上比较有用的方法就是访问它里面存储的数据以及使用引用计数功能。下表展示了它常用的方法。

名称	描述
content()	返回它存储的内容的ByteBuf
copy()	返回一个深拷贝的ByteBufHolder，也就是他们的数据不共享

如果你传输的数据存储的是ByteBuf，那么使用ByteBufHolder是一个不错的选择。

4.1、Netty缓冲区帮助类

使用JDK的NIO API的一个困难是完成一个简单的任务有很多重复性的代码。虽然Netty提供的缓冲区API已经很容易使用了，但是Netty还是提供了很多工具类方便开发者创建或使用缓冲区。

4.2、ByteBufAllocator

前面提到过，Netty提供了池技术优化了各种ByteBuf。为了实现这个功能Netty抽象了一个ByteBufAllocator接口。它主要是用来分配ByteBuf实例的。

我们先来看一下ByteBufAllocator提供了哪些操作。

名称	描述
buffer()	创建ByteBuf，类似可能是堆或直接的，依赖具体的实现
buffer(int)	
buffer(int, int)	
heapBuffer()	创建堆类型ByteBuf
heapBuffer(int)	
heapBuffer(int, int)	
directBuffer()	创建直接类型ByteBuf
directBuffer(int)	
directBuffer(int, int)	
compositeBuffer()	创建复合类型缓冲区
compositeBuffer(int)	
heapCompositeBuffer()	内部为堆类型ByteBuf的复合缓冲区
heapCompositeBuffer(int)	
directCompositeBuffer()	内部为直接类型ByteBuf的复合缓冲区
directCompositeBuffer(int)	
ioBuffer()	创建用于IO操作的ByteBuf

有些方法后面带了一个或两个Int类型参数，这两个参数就是指定ByteBuf的初始容量和最大容量的。你应该记得ByteBuf是可以扩容的。ByteBuf可以不断扩容只到达到了最大容量。

获取ByteBufAllocator的引用也是很容易的事情。通过Channel或者你实现的ChannelHandler的方法中的ChannelHandlerContext。更多关于ChannelHandlerContext和ChannelHandler的知识将会在第六章介绍。

下面的代码展示了获取ByteBufAllocator的方式。

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc();
....
ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc();
```

4.3、Unpooled

有时候你没办法获取ByteBufAllocator的实例，也就是没办法使用上一节介绍的方式创建ByteBuf实例。为了应对这种情况，Netty提供了一个工具类Unpooled。这个工具类提供了很多静态方法帮助我们创建ByteBuf实例，不过没有使用池技术。 先来看看它提供了哪些常用方法给我们使用。

名称	描述
buffer()	创建未使用池技术的堆缓冲ByteBuf
buffer(int)	
buffer(int, int)	
directBuffer()	创建未使用池技术的直接缓冲ByteBuf
directBuffer(int)	
directBuffer(int, int)	
wrappedBuffer()	创建包含给定数据的ByteBuf
copiedBuffer()	复制给定数据然后创建ByteBuf

你也可以在非Netty项目中使用

Unpooled，因为它的API很简单，性能也很高，项目中如果需要这种高性能缓冲区的地方可以尝试一下。

4.4、ByteBufUtil

另一个比较有用的类是ByteBufUtil。这个类提供了很多静态方法直接操作ByteBuf。相比较上面的Unpooled，这个类提供的方法是比较通用的并且不用关心你的ByteBuf是使用了池技术的还是没有使用。 例如它最常用的方法hexdump()，这个方法返回ByteBuf内容的十六进制格式。这个方法可以使用在很多场景，例如日志场景。十六进制格式的内容也很容易转成字节表示。你也许奇怪为什么不直接显示字节内容。主要是因为字节内容人类难以阅读，而16进制就比较友好了。 这个类提供了很多ByteBuf相关的静态方法，例如复制、编码、解码等等，这个大家在以后开发中会慢慢用到，甚至当你需要实现自己的ByteBuf的时候也会用到它们。

五、总结

我们还学习了Netty的数据容器有哪些类型的实现，已经各自的使用场景和性能高低，还有分配或释放需要的资源代价。