# 【第05章-ByteBuf】

网络数据的基本单位总是字节。Java NIO 提供了ByteBuffer 作为它的字节容器，但是这个类使用起来过于复杂，而且也有些繁琐。Netty 的ByteBuffer 替代品是ByteBuf，一个强大的实现，既解决了JDK API 的局限性，又为网络应用程序的开发者提供了更好的API。

## 5.1 ByteBuf 的API

Netty 的数据处理API 通过两个组件暴露——abstract class ByteBuf 和interface ByteBufHolder。

下面是一些ByteBuf API 的优点：

* 它可以被用户自定义的缓冲区类型扩展；
* 通过内置的复合缓冲区类型实现了透明的零拷贝；
* 容量可以按需增长（类似于JDK 的StringBuilder）；
* 在读和写这两种模式之间切换不需要调用ByteBuffer 的flip()方法；
* 读和写使用了不同的索引；
* 支持方法的链式调用；
* 支持引用计数；
* 支持池化。

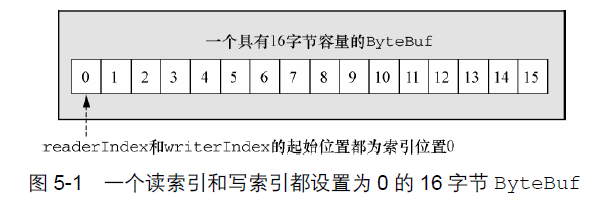
其他类可用于管理ByteBuf 实例的分配，以及执行各种针对于数据容器本身和它所持有的数据的操作。我们将在仔细研究ByteBuf 和ByteBufHolder 时探讨这些特性。

## 5.2 ByteBuf 类——Netty 的数据容器

因为所有的网络通信都涉及字节序列的移动，所以高效易用的数据结构明显是必不可少的。Netty 的ByteBuf 实现满足并超越了这些需求。让我们首先来看看它是如何通过使用不同的索引来简化对它所包含的数据的访问的吧。

### 5.2.1 它是如何工作的

ByteBuf 维护了两个不同的索引：一个用于读取，一个用于写入。当你从ByteBuf 读取时，它的readerIndex 将会被递增已经被读取的字节数。同样地，当你写入ByteBuf 时，它的writerIndex 也会被递增。图5-1 展示了一个空ByteBuf 的布局结构和状态。



要了解这些索引两两之间的关系，请考虑一下，如果打算读取字节直到readerIndex 达到和writerIndex 同样的值时会发生什么。在那时，你将会到达“可以读取的”数据的末尾。就如同试图读取超出数组末尾的数据一样，试图读取超出该点的数据将会触发一个IndexOutOfBoundsException。

名称以read 或者write 开头的ByteBuf 方法，将会推进其对应的索引，而名称以set 或者get 开头的操作则不会。后面的这些方法将在作为一个参数传入的一个相对索引上执行操作。

可以指定ByteBuf 的最大容量。试图移动写索引（即writerIndex）超过这个值将会触发一个异常，也就是说用户直接或者间接使capacity(int)或者ensureWritable(int)方法来增加超过该最大容量时抛出异常。（默认的限制是Integer.MAX\_VALUE。）

### 5.2.2 ByteBuf 的使用模式

在使用Netty 时，你将遇到几种常见的围绕ByteBuf 而构建的使用模式。在研究它们时，我们心里想着图5-1 会有所裨益— 一个由不同的索引分别控制读访问和写访问的字节数组。

**1．堆缓冲区**

最常用的ByteBuf 模式是将数据存储在JVM 的堆空间中。这种模式被称为支撑数组（backing array），它能在没有使用池化的情况下提供快速的分配和释放。这种方式，如代码清单5-1 所示，非常适合于有遗留的数据需要处理的情况。

|  |
| --- |
| **// 代码清单5-1 支撑数组**  **public static void** heapBuffer() {  ByteBuf heapBuf = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *//get reference form somewhere  // 检查ByteBuf 是否有一个支撑数组* **if** (heapBuf.hasArray()) {  *// 如果有，则获取对该数组的引用* **byte**[] array = heapBuf.array();  *// 计算第一个字节的偏移量。* **int** offset = heapBuf.arrayOffset() + heapBuf.readerIndex();  *// 获得可读字节数* **int** length = heapBuf.readableBytes();  *// 使用数组、偏移量和长度作为参数调用你的方法  handleArray*(array, offset, length);  } } |

注意 当hasArray()方法返回false 时，尝试访问支撑数组将触发一个UnsupportedOperationException。这个模式类似于JDK 的ByteBuffer 的用法。

**2．直接缓冲区**

直接缓冲区是另外一种ByteBuf 模式。我们期望用于对象创建的内存分配永远都来自于堆中，但这并不是必须的——NIO 在JDK 1.4 中引入的ByteBuffer 类允许JVM 实现通过本地调用来分配内存。这主要是为了避免在每次调用本地I/O 操作之前（或者之后）将缓冲区的内容复制到一个中间缓冲区（或者从中间缓冲区把内容复制到缓冲区）。

ByteBuffer的Javadoc明确指出：“直接缓冲区的内容将驻留在常规的会被垃圾回收的堆之外。”这也就解释了为何直接缓冲区对于网络数据传输是理想的选择。如果你的数据包含在一个在堆上分配的缓冲区中，那么事实上，在通过套接字发送它之前，JVM将会在内部把你的缓冲区复制到一个直接缓冲区中。

直接缓冲区的主要缺点是，相对于基于堆的缓冲区，它们的分配和释放都较为昂贵。如果你正在处理遗留代码，你也可能会遇到另外一个缺点：因为数据不是在堆上，所以你不得不进行一次复制，如代码清单5-2 所示。

显然，与使用支撑数组相比，这涉及的工作更多。因此，如果事先知道容器中的数据将会被作为数组来访问，你可能更愿意使用堆内存。

|  |
| --- |
| **// 代码清单5-2 访问直接缓冲区的数据**  **public static void** directBuffer() {  ByteBuf directBuf = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *//get reference form somewhere  // 检查ByteBuf 是否由数组支撑。如果不是，则这是一个直接缓冲区* **if** (!directBuf.hasArray()) {  *// 获取可读字节数* **int** length = directBuf.readableBytes();  *// 分配一个新的数组来保存具有该长度的字节数据* **byte**[] array = **new byte**[length];  *// 将字节复制到该数组* directBuf.getBytes(directBuf.readerIndex(), array);  *// 使用数组、偏移量和长度作为参数调用你的方法  handleArray*(array, **0**, length);  } } |

**3．复合缓冲区**

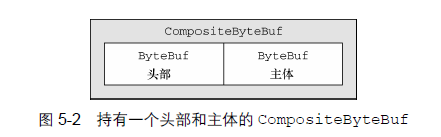
第三种也是最后一种模式使用的是复合缓冲区，它为多个ByteBuf 提供一个聚合视图。在这里你可以根据需要添加或者删除ByteBuf 实例，这是一个JDK 的ByteBuffer 实现完全缺失的特性。

Netty 通过一个ByteBuf 子类——CompositeByteBuf——实现了这个模式，它提供了一个将多个缓冲区表示为单个合并缓冲区的虚拟表示。

**警告** CompositeByteBuf 中的ByteBuf 实例可能同时包含直接内存分配和非直接内存分配。如果其中只有一个实例，那么对CompositeByteBuf 上的hasArray()方法的调用将返回该组件上的hasArray()方法的值；否则它将返回false。

为了举例说明，让我们考虑一下一个由两部分——头部和主体——组成的将通过HTTP 协议传输的消息。这两部分由应用程序的不同模块产生，将会在消息被发送的时候组装。该应用程序可以选择为多个消息重用相同的消息主体。当这种情况发生时，对于每个消息都将会创建一个新的头部。

因为我们不想为每个消息都重新分配这两个缓冲区，所以使用CompositeByteBuf 是一个完美的选择。它在消除了没必要的复制的同时，暴露了通用的ByteBuf API。图5-2 展示了生成的消息布局。



代码清单5-3 展示了如何通过使用JDK 的ByteBuffer 来实现这一需求。创建了一个包含两个ByteBuffer 的数组用来保存这些消息组件，同时创建了第三个ByteBuffer 用来保存所有这些数据的副本。

|  |
| --- |
| **// 代码清单5-3 使用ByteBuffer 的复合缓冲区模式**  **public static void** byteBufferComposite(ByteBuffer header, ByteBuffer body) {  *// Use an array to hold the message parts* ByteBuffer[] message = **new** ByteBuffer[]{header, body};   *// Create a new ByteBuffer and use copy to merge the header and body* ByteBuffer message2 = ByteBuffer.*allocate*(header.remaining() + body.remaining());  message2.put(header);  message2.put(body);  message2.flip(); } |

分配和复制操作，以及伴随着对数组管理的需要，使得这个版本的实现效率低下而且笨拙。代码清单5-4 展示了一个使用了CompositeByteBuf 的版本。

|  |
| --- |
| **// 代码清单5-4 使用CompositeByteBuf 的复合缓冲区模式**  **public static void** byteBufComposite() {  CompositeByteBuf messageBuf = Unpooled.*compositeBuffer*();  ByteBuf headerBuf = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *// can be backing or direct* ByteBuf bodyBuf = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *// can be backing or direct  // 将ByteBuf 实例追加到CompositeByteBuf* messageBuf.addComponents(headerBuf, bodyBuf);  *//...  // 删除位于索引位置为 0（第一个组件）的ByteBuf* messageBuf.removeComponent(**0**); *// remove the header  // 循环遍历所有的ByteBuf 实例* **for** (ByteBuf buf : messageBuf) {  System.***out***.println(buf.toString());  } } |

CompositeByteBuf 可能不支持访问其支撑数组，因此访问CompositeByteBuf 中的数据类似于（访问）直接缓冲区的模式，如代码清单5-5 所示。

|  |
| --- |
| **// 代码清单5-5 访问CompositeByteBuf 中的数据**  **public static void** byteBufCompositeArray() {  CompositeByteBuf compBuf = Unpooled.*compositeBuffer*();  *// 获得可读字节数* **int** length = compBuf.readableBytes();  *// 分配一个具有可读字节数长度的新数组* **byte**[] array = **new byte**[length];  *// 将字节读到该数组中* compBuf.getBytes(compBuf.readerIndex(), array);  *// 使用数组、偏移量和长度作为参数调用你的方法  handleArray*(array, **0**, array.**length**); } |

需要注意的是，Netty使用了CompositeByteBuf来优化套接字的I/O操作，尽可能地消除了由JDK的缓冲区实现所导致的性能以及内存使用率的惩罚。这种优化发生在Netty的核心代码中，因此不会被暴露出来，但是你应该知道它所带来的影响。

CompositeByteBuf API 除了从ByteBuf 继承的方法，CompositeByteBuf 提供了大量的附加功能。请参考Netty 的Javadoc 以获得该API 的完整列表。

### 5.3 字节级操作

ByteBuf 提供了许多超出基本读、写操作的方法用于修改它的数据。在接下来的章节中，我们将会讨论这些中最重要的部分。

### 5.3.1 随机访问索引

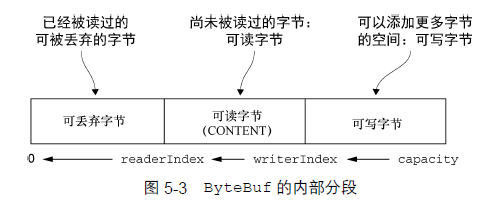
如同在普通的Java 字节数组中一样，ByteBuf 的索引是从零开始的：第一个字节的索引是0，最后一个字节的索引总是capacity() - 1。代码清单5-6 表明，对存储机制的封装使得遍历ByteBuf 的内容非常简单。

|  |
| --- |
| **//代码清单5-6 访问数据**  **public static void** byteBufRelativeAccess() {  ByteBuf buffer = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *//get reference form somewhere* **for** (**int** i = **0**; i < buffer.capacity(); i++) {  **byte** b = buffer.getByte(i);  System.***out***.println((**char**) b);  } } |

需要注意的是，使用那些需要一个索引值参数的方法（的其中）之一来访问数据既不会改变readerIndex 也不会改变writerIndex。如果有需要，也可以通过调用readerIndex(index)或者writerIndex(index)来手动移动这两者。

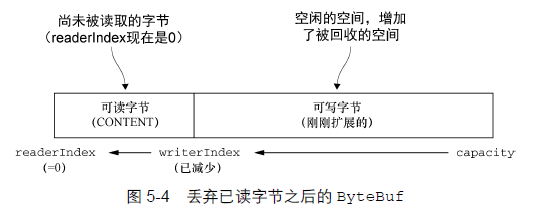
### 5.3.2 顺序访问索引

虽然ByteBuf 同时具有读索引和写索引，但是JDK 的ByteBuffer 却只有一个索引，这也就是为什么必须调用flip()方法来在读模式和写模式之间进行切换的原因。图5-3 展示了ByteBuf 是如何被它的两个索引划分成3 个区域的。



### 5.3.3 可丢弃字节

在图5-3 中标记为可丢弃字节的分段包含了已经被读过的字节。通过调用discardReadBytes()方法，可以丢弃它们并回收空间。这个分段的初始大小为0，存储在readerIndex 中，会随着read 操作的执行而增加（get\*操作不会移动readerIndex）。图5-4 展示了图5-3 中所展示的缓冲区上调用discardReadBytes()方法后的结果。可以看到，可丢弃字节分段中的空间已经变为可写的了。注意，在调用discardReadBytes()之后，对可写分段的内容并没有任何的保证，因为只是移动了可以读取的字节以及writerIndex，而没有对所有可写入的字节进行擦除写。



虽然你可能会倾向于频繁地调用discardReadBytes()方法以确保可写分段的最大化，但是请注意，这将极有可能会导致内存复制，因为可读字节（图中标记为CONTENT 的部分）必须被移动到缓冲区的开始位置。我们建议只在有真正需要的时候才这样做，例如，当内存非常宝贵的时候。

### 5.3.4 可读字节

ByteBuf 的可读字节分段存储了实际数据。新分配的、包装的或者复制的缓冲区的默认的readerIndex 值为0。任何名称以read 或者skip 开头的操作都将检索或者跳过位于当前readerIndex 的数据，并且将它增加已读字节数。

如果被调用的方法需要一个ByteBuf 参数作为写入的目标，并且没有指定目标索引参数，那么该目标缓冲区的writerIndex 也将被增加，例如：readByt es(ByteBuf dest);如果尝试在缓冲区的可读字节数已经耗尽时从中读取数据，那么将会引发一个IndexOutOfBoundsException。

|  |
| --- |
| **// 代码清单5-7 读取所有数据**  **public static void** readAllData() {  ByteBuf buffer = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *//get reference form somewhere* **while** (buffer.isReadable()) {  System.***out***.println(buffer.readByte());  } } |

### 5.3.5 可写字节

可写字节分段是指一个拥有未定义内容的、写入就绪的内存区域。新分配的缓冲区的writerIndex 的默认值为0。任何名称以write 开头的操作都将从当前的writerIndex 处开始写数据，并将它增加已经写入的字节数。如果写操作的目标也是ByteBuf，并且没有指定源索引的值，则源缓冲区的readerIndex 也同样会被增加相同的大小。这个调用如下所示：

writeBy tes(ByteBuf dest);

如果尝试往目标写入超过目标容量的数据，将会引发一个IndexOutOfBoundException。在往ByteBuf 中写入数据时，其将首先确保目标ByteBuf 具有足够的可写入空间来容纳当前要写入的数据，如果没有，则将检查当前的写索引以及最大容量是否可以在扩展后容纳该数据，可以则会分配

并调整容量，否则就会抛出该异常。代码清单5-8 是一个用随机整数值填充缓冲区，直到它空间不足为止的例子。writeableBytes()方法在这里被用来确定该缓冲区中是否还有足够的空间。

|  |
| --- |
| **// 代码清单5-8 写数据**  **public static void** byteProcessor() {  ByteBuf buffer = ***BYTE\_BUF\_FROM\_SOMEWHERE***; *//get reference form somewhere* **int** index = buffer.forEachByte(ByteProcessor.***FIND\_CR***); } |

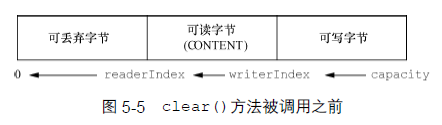
### 5.3.6 索引管理

JDK 的InputStream 定义了mark(int readlimit)和reset()方法，这些方法分别被用来将流中的当前位置标记为指定的值，以及将流重置到该位置。

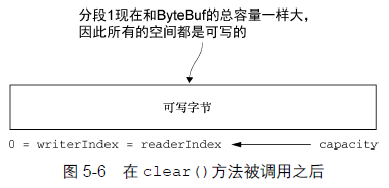
同样，可以通过调用markReaderIndex()、markWriterIndex()、resetWriterIndex()和resetReaderIndex()来标记和重置ByteBuf 的readerIndex 和writerIndex。这些和InputStream 上的调用类似，只是没有readlimit 参数来指定标记什么时候失效。

也可以通过调用readerIndex(int)或者writerIndex(int)来将索引移动到指定位置。试图将任何一个索引设置到一个无效的位置都将导致一个IndexOutOfBoundsException。

可以通过调用clear()方法来将readerIndex 和writerIndex 都设置为0。注意，这并不会清除内存中的内容。图5-5（重复上面的图5-3）展示了它是如何工作的。



和之前一样，ByteBuf 包含3 个分段。图5-6 展示了在clear()方法被调用之后ByteBuf的状态。



调用clear()比调用discardReadBytes()轻量得多，因为它将只是重置索引而不会复制任何的内存。

### 5.3.7 查找操作

在ByteBuf中有多种可以用来确定指定值的索引的方法。最简单的是使用indexOf()方法。较复杂的查找可以通过那些需要一个ByteBufProcessor作为参数的方法达成，在Netty 4.1.x 中，该类已经废弃，请使用io.netty.util.ByteProcessor。这个接口只定义了一个方法：

boolean process(byte value)

它将检查输入值是否是正在查找的值。ByteBufProcessor针对一些常见的值定义了许多便利的方法。假设你的应用程序需要和所谓的包含有以NULL结尾的内容的Flash套接字集成。调用

forEach Byte(ByteBufProcessor.FIND\_NUL)

将简单高效地消费该Flash 数据，因为在处理期间只会执行较少的边界检查。代码清单5-9 展示了一个查找回车符（\r）的例子。

### 5.3.8 派生缓冲区

**派生缓冲区为ByteBuf 提供了以专门的方式来呈现其内容的视图。**这类视图是通过以下方法被创建的：

* duplicate()；
* slice()；
* slice(int, int)；
* Unpooled.unmodifiableBuffer(…)；
* order(ByteOrder)；
* readSlice(int)。

每个这些方法都将返回一个新的ByteBuf 实例，它具有自己的读索引、写索引和标记索引。其内部存储和JDK 的ByteBuffer 一样也是共享的。这使得派生缓冲区的创建成本是很低廉的，但是这也意味着，如果你修改了它的内容，也同时修改了其对应的源实例，所以要小心。

**ByteBuf 复制 如果需要一个现有缓冲区的真实副本，请使用copy()或者copy(int, int)方法。不同于派生缓冲区，由这个调用所返回的ByteBuf 拥有独立的数据副本。**

代码清单5-10 展示了如何使用slice(int, int)方法来操作ByteBuf 的一个分段。

|  |
| --- |
| **// 代码清单5-10 对ByteBuf 进行切片**  **public static void** byteBufSlice() {  Charset utf8 = Charset.*forName*(**"UTF-8"**);  *// 创建一个用于保存给定字符串的字节的ByteBuf* ByteBuf buf = Unpooled.*copiedBuffer*(**"Netty in Action rocks!"**, utf8);  *// 创建该ByteBuf 从索引0 开始到索引15结束的一个新切片* ByteBuf sliced = buf.slice(**0**, **15**);  *// 将打印“Netty in Action”* System.***out***.println(sliced.toString(utf8));  *// 更新索引0 处的字节* buf.setByte(**0**, (**byte**) **'J'**);  *// 将会成功，因为数据是共享的，对其中一个所做的更改对另外一个也是可见的* **assert** buf.getByte(**0**) == sliced.getByte(**0**); } |

现在，让我们看看ByteBuf 的分段的副本和切片有何区别，如代码清单5-11 所示。

|  |
| --- |
| **// 代码清单5-11 复制一个ByteBuf**  **public static void** byteBufCopy() {  Charset utf8 = Charset.*forName*(**"UTF-8"**);  *// 创建一个用于保存给定字符串的字节的ByteBuf* ByteBuf buf = Unpooled.*copiedBuffer*(**"Netty in Action rocks!"**, utf8);  *// 创建该ByteBuf 从索引0 开始到索引15结束的分段的副本* ByteBuf copy = buf.copy(**0**, **15**);  *// 将打印“Netty in Action”* System.***out***.println(copy.toString(utf8));  *// 更新索引0 处的字节* buf.setByte(**0**, (**byte**) **'J'**);  *// 将会成功，因为数据不是共享的* **assert** buf.getByte(**0**) != copy.getByte(**0**); } |

除了修改原始ByteBuf 的切片或者副本的效果以外，这两种场景是相同的。只要有可能，使用slice()方法来避免复制内存的开销。

5.3.9 读/写操作

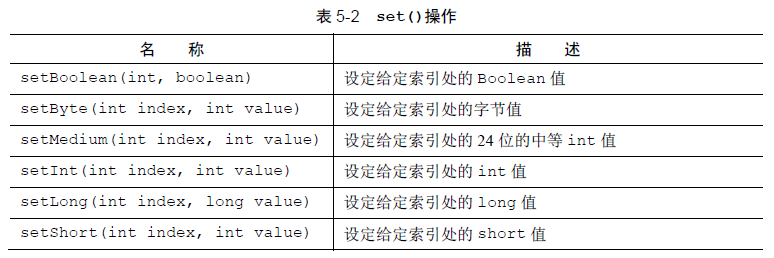
正如我们所提到过的，有两种类别的读/写操作：

* get()和set()操作，从给定的索引开始，并且保持索引不变；
* read()和write()操作，从给定的索引开始，并且会根据已经访问过的字节数对索引进行调整。

表5-1 列举了最常用的get()方法。完整列表请参考对应的API 文档。



大多数的这些操作都有一个对应的set()方法。这些方法在表5-2 中列出。



代码清单5-12 说明了get()和set()方法的用法，表明了它们不会改变读索引和写索引。

|  |
| --- |
| **// 代码清单5-12 get()和set()方法的用法**  **public static void** byteBufSetGet() {  Charset utf8 = Charset.*forName*(**"UTF-8"**);  *// 创建一个用于保存给定字符串的字节的ByteBuf* ByteBuf buf = Unpooled.*copiedBuffer*(**"Netty in Action rocks!"**, utf8);  System.***out***.println((**char**) buf.getByte(**0**));  *// 存储当前的readerIndex 和writerIndex* **int** readerIndex = buf.readerIndex();  **int** writerIndex = buf.writerIndex();  *// 将索引0 处的字节更新为字符'B'* buf.setByte(**0**, (**byte**) **'B'**);  *// 打印第一个字符，现在是'B'* System.***out***.println((**char**) buf.getByte(**0**));  *// 将会成功，因为这些操作并不会修改相应的索引* **assert** readerIndex == buf.readerIndex();  **assert** writerIndex == buf.writerIndex(); } |

现在，让我们研究一下read()操作，其作用于当前的readerIndex 或writerIndex。这些方法将用于从ByteBuf 中读取数据，如同它是一个流。表5-3 展示了最常用的方法。

