面向对象源码阅读：Netty Btyebuf 模块——功能分析和建模

宗吉祥 2019K8009918011

一、 需求分析

在网络通信与传输过程中，数据 IO 的吞吐处理速度常常成为整个节点系统的瓶颈（正如本地计算机中 IO 系统的速度远低于 CPU 的处理速度）。因此，为了提高处理速度，降低延迟，必须设立缓存机制（正如本地计算机系统中连接 CPU 寄存器与内存的 CPU CACHE）。

JDK 本身提供了 ByteBuffer 作为它的字节容器，但是这个字节容器存在以下缺点：

1. 类过于复杂，不够抽象
2. 读写不分离，灵活性差
3. 交互性能差，使用繁琐

为了克服传统的 ByteBuffer 的这些缺陷，提供更加适合开发者的 API,Netty 的 ByteBuf 作为替代品应运而生。

二、 需求建模

ByteBuf 有几个基本的需求

1. 实现灵活的读写：读写头分离，读写区分离，读写长度自由
2. 实现灵活的扩展：包括用户自定义的缓冲区类型、按需求增长的缓冲区容量
3. 实现灵活的操作：支持引用计数、池化、链式调用方法等

根据这样的需求，我们可以给出一个较为抽象的 ByteBuf 建模：

类	ByteBuf
值	读写头
	读写标志
	容量显示
方法	读出数据、写入数据
	清理无用空间
	标记关键数据位置
	合并或删除区域

	.....

三、 需求建模的实现：

1.相关变量

为了实现上述的需求和功能，ByteBuf 的类定义由抽象基类 AbstractByteBuf 实现。

AbstractByteBuf 定义了以下五个变量：

```
int readerIndex; //读索引
int writerIndex; //写索引
private int markedReaderIndex;//标记读索引
private int markedWriterIndex;//标记写索引
private int maxCapacity;//缓冲区的最大容量
```

ByteBuf 类中实现的复杂方法以及基本读写操作都会单独维护读写索引，以及读写索引的标记，实现读写的灵活分离。

源码中用下面的代码图形象地展示了空间如何被读写头分割：

```
+-------------------+------------------+------------------+
| discardable bytes |  readable bytes  |  writable bytes  |
|                   |     (CONTENT)    |                  |
+-------------------+------------------+------------------+
|                   |                  |                  |
0      <=      readerIndex   <=   writerIndex    <=    capacity
```

当前的可读数据区是[readIndex,writeIndex)；可写区是[writeIndex,capacity)；而[0,readIndex)区间的字节是可废弃数据(Discardable)。

2.方法

为了实现灵活的扩展和 API 操作，ByteBuf 实现了丰富的方法，这里给出方法表：

基本方法	

capacity	缓冲区能容纳的字节数
readerIndex	下一个读位置
writeIndex	下一个写位置
capacity(int newCapacity)	扩展当前缓冲区至新容量，返回一个容量等于 newCapacity 的 ByteBuf 对象(不保证和旧是同一个对象)，如果 newCapacity 小于之前的容量，那么数据可能被截断
maxCapacity	此缓冲区能扩充的最大容量，也即上面 newCapacity 的最大值
ByteBufAllocator alloc()	分配此 ByteBuf 的分配器
unwrap()	如果此对象是包装另一个 ByteBuf 对象生成的，返回后者，否则返回 null
isDirect()	是否该 ByteBuf 是一个直接内存缓冲区
isReadOnly()	是否只读
asReadOnly()	返回该 ByteBuf 的一个只读版本
readerIndex(int readerIndex)	设置 readerIndex，不能<0，或>writerIndex
writerIndex(int writerIndex)	设置 writerIndex，不能<readerIndex，或>capacity
setIndex(int readerIndex, int writerIndex)	同时设置 readerIndex, writerIndex
readableBytes()	=writerIndex-readerIndex
isReadable()	=(writerIndex-readerIndex>0)
isReadable(size)	=(writerIndex-readerIndex>size)
writableBytes()	=capacity-writerIndex
maxWritableBytes()	=maxCapacity-writerIndex
maxFastWritableBytes()	在不需要执行内存分配、数据 copy 的前提下，能达到的最大可写字节数，默认等于 writableBytes，实际算法取决于 Bytebuf 的实现细节
isWritable	=(capacity - writerIndex)>0
isWritable(int size)	=(capacity - writerIndex)>size
clear()	恢复到初始状态，相当于 setIndex(0,0)
markReaderIndex&resetReaderIndex	markReaderIndex 记住当前 readIndex，resetReaderIndex 恢复
markWriterIndex&resetWriterIndex	同上
discardReadBytes	抛弃已读字节，将[readerIndex, writeIndex)字节迁移到[0, readableBytes)，修改 readerIndex=0, writeIndex=readableBytes
discardSomeReadBytes	抛弃部分已读字节以节省内存，具体数量由实现来决定，以达到最高效
ensureWritable(int minWritableBytes)	按需扩展容量，如果 writeIndex+minWritableBytes> maxCapacity，抛出 IndexOutOfBoundsException
ensureWritable(int minWritableBytes, boolean	功能同上，但不会抛出异常，参数 force 影响

force)	（writeIndex+minWritableBytes> maxCapacity）条件下的行为：force=true，扩充容量至maxCapacity；force=false，不做扩充。返回值只一个反映操作行为的状态码：=0，容量本来就满足，所以未扩充；=1，容量不足，但未扩充；=2，容量已扩充，满足需求；=3，容量以扩充至maxCapacity，但仍未满足需求。
get 方式读数据方法	注意：所有的 get 方法类似 byte 数组操作，不影响 readIndex
getBoolean(int index)	get bool 值，其他 getByte，getShort，getUnsignedByte，getUnsignedShort，getInt，getLong,getChar,getFloat,getDouble 功能类似
getShortLE(int index)	以 Litter Endian 的格式获取数据，getUnsignedShortLE，getIntLE，getLongLE,getFloatLE,getDoubleLE 类似
getMedium(int index)	24bit 方式读取 int，getMediumLE，getUnsignedMedium，getUnsignedMediumLE
set 方式写数据方法	注意：所有的 set 方法类似 byte 数组操作，不影响 writeIndex
setBoolean(int index, boolean value)	其他 setByte，setShort，setUnsignedByte，setUnsignedShort，setInt，setLong,setChar,setFloat,setDouble 功能类似
setShortLE(int index)	以 Litter Endian 的格式获取数据，setUnsignedShortLE，setIntLE，setLongLE,setFloatLE,setDoubleLE 类似
setMedium(int index)	24bit 方式读取 int，setMediumLE，setUnsignedMedium，setUnsignedMediumLE
getBytes 操作	所有 getBytes 操作不影响本 buf 的 readIndex，writeIndex
getBytes(int index, ByteBuf dst)	将[index,writeIndex)区间字节写入 dst，该操作增加 dst 的 writeIndex
getBytes(int index, ByteBuf dst, int length)	同上，指定写入字节数，而不是默认全部
getBytes(int index, ByteBuf dst, int dstIndex, int length)	将[index,index+length)写入目标 dst 的[dstIndex, dstIndex +length)，注意，该操作也不影响 dst 的 readIndex 和 writeIndex
getBytes(int index, byte[] dst)	上面操作的数组版本，其他几个变体都有
getBytes(int index, OutputStream out, int length)	目标是 stream
getBytes(int index, GatheringByteChannel out, int length)	目标是 NIO Channel
getBytes(int index, FileChannel out, long position, int length)	目标是 NIO FileChannel
getCharSequence(int index, int length, Charset charset)	读取为字符串
setBytes 操作	所有 getBytes 操作不影响本 buf 的 readIndex，

	writeIndex
setBytes(int index, ByteBuf src)	将 src 可读 byte 全部写入本 buf 的 index 开始的位置，它会增加 sr 的 readIndex。
setBytes 其他变体	参考 getBytes 变体
read 基本类型	read 操作增加 buf 的 readInex
readBoolean	从 readIndex 处读一个 bool 值，readIndex++
readXXX	readByte，readShort，readMedium，readInt 等，readIndex 增加响应的字节数
write 基本类型	write 操作增加 buf 的 writeInex
write Boolean	从 writeIndex 写入一个 bool 值，writeIndex++
writeXXX	writeByte, writeShort, writeMedium, writeInt 等，writeIndex 增加响应的字节数
readBytes 操作	read 操作导致 buf 的 readInex 增加读取的字节数
ByteBuf readBytes(int length)	读取 length 字节数, 并返回一个新创建的 Buf 对象，新 buf 的 readIndex=0,writeIndex=length，源 buf 的 readIndex+=length
ByteBuf readBytes(ByteBuf dst)	将 buf 的可读字节写入 dst，字节数=min(src.readableBytes，dst.writableBytes)，src.readIndex 增加，src.writeIndex 增加
readBytes(ByteBuf dst, int length)	同上，指定长度
readBytes(ByteBuf dst, int dstIndex, int length)	同上，但不修改 dst.writeIndex
readBytes(byte[] dst)	数组版本，也有其他变体
readBytes(ByteBuffer dst)	nio buffer 版本
readBytes(OutputStream out, int length)	ostream 版本
readBytes(GatheringByteChannel out, int length)	nio channel 版本
readCharSequence(int length, Charset charset)	字符串版本
readBytes(FileChannel out, long position, int length)	文件 Channel 版本
skipBytes(int length)	跳过一些可读字节，readIndex+=length
writeBytes 操作	write 操作导致 buf 的 writeInex 增加读取的字节数，支持的操作和 readBytes 几乎一一对应
字节遍历操作	
bytesBefore(int index, int length, byte value)	查询 value 在 buf 出现的位置，仅限[index,index+length)区间内查找
forEachByte(ByteProcessor processor)	遍历可读 bytes，其他还有几个变体
buf 复制	
copy()	复制 buf 的可读字节区, 新 buf 的 readIndex=0，新 buf 和源 buf 互相独立
copy(int index, int length)	新 buf 的 readIndex=0, writeIndex=capacity=length
slice()	返回源 buf 可读字节的一个切片，新 buf 和源 buf 共享 byte 内存，但 readIndex,writeIndex 互

	相独立
retainedSlice()	相当于 slice().retain()
slice(int index, int length)	
retainedSlice(int index, int length)	slice 变体
duplicate()	
retainedDuplicate()	制作 buf 的一个副本，底层共享 byte 内存，但 readIndex,writeIndex 互相独立
nio buffer 相关操作	ByteBuf 是否支持下面某个操作，与具体实现有关
nioBufferCount()	buf 底层包含的 ByteBuffer 数量，返回-1，如果该 buf 不是由 ByteBuffer 构成的
nioBuffer()	返回一个包含可读字节区的 ByteBuffer，该 ByteBuffer 与源 ByteBuf 是否共享内存与具体实现有关，但 readIndex&writeIndex 是独立的
nioBuffer(int index, int length)	同上，指定字节区，也不是用可读字节区
internalNioBuffer(int index, int length)	特定实现支持的接口
nioBuffers()	
nioBuffers(int index, int length)	特定实现支持的接口
内存操作	
hasArray()	buf 内部是否被一个 byte array 支撑
array()	返回支撑的 byte array，如果不支持，抛出 UnsupportedOperationException
arrayOffset()	返回 buf 的 0 位置，在 byte 数组中的位置
hasMemoryAddress()	buf 是否拥有一个指向底层内存的地址
memoryAddress()	返回底层内存地址，如果不支持，抛出 UnsupportedOperationException
isContiguous()	buf 时候由一整块内存支持

此外，ByteBuf 提供 3 种缓冲区模式：HeapBuffer、DirectBuffer、CompositeBuffer。其中 CompositeBuffer 为相较于 ByteBuffer 提出的新的缓冲区类型。

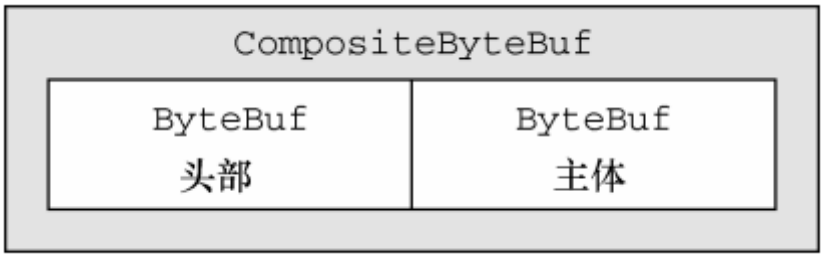
HeapBuffer 就是将数据存在 JVM 堆空间中，在没有被池化的情况可以快速分配和释放。由于数据是存储在 JVM 堆中，因此可以快速的创建与快速的释放，并且它提供了直接访问内部字节数组的方法。但是每次读写数据时，都需要先将数据复制到直接缓冲区中再进行网路传输。

DirectBuffer 在堆外直接分配内存空间，DirectBuffer 并不会占用堆的容量空间，因为它是由操作系统在本地内存进行的数据分配。在使用 Socket 进行数据传递时，性能非常好，因

为数据直接位于操作系统的本地内存中，所以不需要从 JVM 将数据复制到直接缓冲区中 。但是 Direct Buffer 是直接在操作系统内存中的，内存空间的分配与释放要比堆空间更加复杂，速度要慢一些。

CompositeBuffer 拥有以上两种的缓冲区，可以为使用者提供多个 ByteBuf 的综合视图，可以虚拟地把多个缓冲区合并为单个缓冲区，并且使用者可以根据需求在该缓冲区中增加或者删除具体的 ByteBuf 实例。

具体地讲一个合并缓冲区的用途：
例如服务器需要处理一条用户传递的消息。根据网络通信协议，一条消息在主体内容之上，会被封装上一层一层的通讯协议头，通过对这些协议头的拆包解包，实现消息传递的顺利握手。那么在 Netty 中，就可以使用 CompositeByteBuf 对消息的主体和头部进行拆解或者组合，同时在传输过程中可以合并为一整个虚拟的区，简化了传输过程。



四、 具体源码实现分析

由于本章主要关注抽象的建模和功能，因此我们不深入地对每个 ByteBuf 实现的方法进行分析（实际上这也是难以做到的），而是选择挑选最具代表性的三个方法：数据读取、数据写入、清理缓冲区，进行分析。

0. ByteBuf 的创建：

（1）通过 ByteBufAllocator 这个接口来创建 ByteBuf，这个接口可以创建上面的三种 Buffer，一般都是通过 channel 的 alloc()接口获取。

（2）通过 Unpooled 类里面的静态方法，创建 Buffer

```
CompositeByteBuf compBuf = Unpooled.compositeBuffer();
ByteBuf heapBuf = Unpooled.buffer(8);
ByteBuf directBuf = Unpooled.directBuffer(16);
```

1. Readbytes

代码如下：

```
public ByteBuf readBytes(ByteBuf dst, int dstIndex, int length) {
    checkReadableBytes(length);
    getBytes(readerIndex, dst, dstIndex, length);
    readerIndex += length;
    return this;
}
```

首先 checkReadableBytes 函数会检查读取数据的行为是否合法，即读头是否在写头之后，以及需要读取的内容长度是否非负。如果检查不通过，则会报出异常

在检查通过无误后，会调用 getBytes 函数读取从当前读头开始的对应长度的数据，如果读取成功，则 readerIndex 应该增长对应的长度。

注意，checkReadableBytes 和 getBytes 都有具体的另在别处的函数实现，但是我们这里更关心流程，因此将它们作为原子的、抽象的操作即可。

2. WriteBytes

代码如下：

```
public ByteBuf writeBytes(byte[] src, int srcIndex, int length) {
    ensureWritable(length);
    setBytes(writerIndex, src, srcIndex, length);
    writerIndex += length;
    return this;
}
```

写入数据的流程可以概括为：

（1）检查写入的长度是否为非负，否则报错。检查写入的长度是否超过可用区的大小，否则自动扩展区的大小。（使用 ensureWritable 函数）。扩展大小的计算以及扩展区的空间分配会涉及到 calculateNewCapacity 函数和 capacity 函数。

（2）使用 setBytes 函数进行写入数据

（3） 根据写入数据的长度增长 WriterIndex

3. discardReadBytes

代码如下：

```
public ByteBuf discardReadBytes() {
    ensureAccessible();
    if (readerIndex == 0) {
        return this;
    }

    if (readerIndex != writerIndex) {
        setBytes(0, this, readerIndex, writerIndex - readerIndex);
        writerIndex -= readerIndex;
        adjustMarkers(readerIndex);
        readerIndex = 0;
    } else {
        adjustMarkers(readerIndex);
        writerIndex = readerIndex = 0;
    }
    return this;
}
```

流程可以概括如下：

（1） 对 readerIndex 进行判断：如果 readerIndex 等于 0，说明没有可以用来复用的空间，此时函数流程直接结束。如果 readerIndex 大于 0 且不等于 writerIndex 的话，说明有进行数据读取被丢弃的缓冲区，也有还没有被读取的缓冲区。

（2） 进行字节数组的复制，将没被读取的数据移动到缓冲区的起始位置，重新去设置 readerIndex 和 writerIndex，readerIndex

（3） 使用 adjustMarkers 函数，重新设置 mark 的位置。

参考：
ByteBuf 的具体方法表来自：
https://blog.csdn.net/baiye_xing/article/details/76551937
ByteBuf 的 CompositeBuff 缓冲区案例来自：
https://www.cnblogs.com/duanxz/p/3724448.html