面向对象源码阅读：Netty Btyebuf模块——功能分析和建模

1. 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。

1. 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 |
| …… |

1. 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

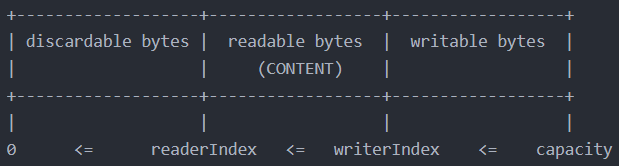
private int markedReaderIndex;//标记读索引

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：



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：

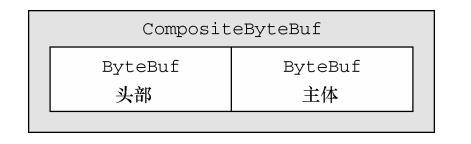
|  |  |
| --- | --- |
| 基本方法 |  |
| capacity | 缓冲区能容纳的字节数 |
| readerIndex | 下一个读位置 |
| writeIndex | 下一个写位置 |
| capacity(int newCapacity) | 扩展当前缓冲区至新容量，返回一个容量等于newCapacity的ByteBuf对象(不保证和旧是同一个对象)，如果newCapacity小于之前的容量，那么数据可能被截断 |
| maxCapacity | 此缓冲区能扩充的最大容量，也即上面newCapacity的最大值 |
| ByteBufAllocator alloc() | 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 force) | 功能同上，但不会抛出异常，参数force影响（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。

CompositeBuffer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。



1. 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。

1. 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。

1. 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

1. 根据写入数据的长度增长WriterIndex
2. 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