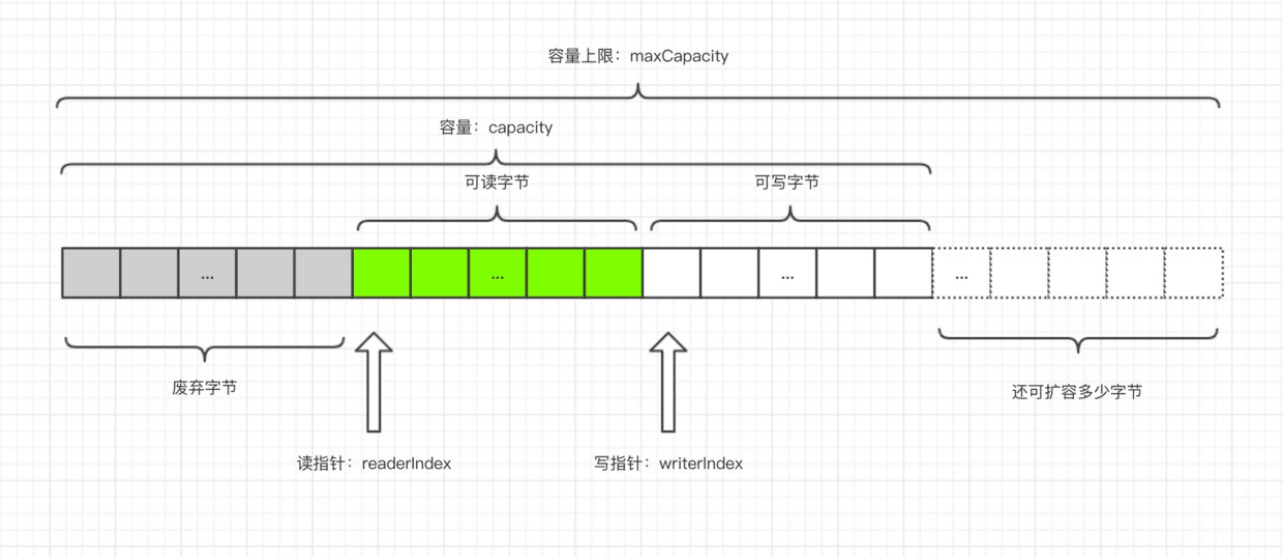
数据传输载体 ByteBuf 介绍

在前面一小节，我们已经了解到 Netty 里面数据读写是以 ByteBuf 为单位进行交互的，这一小节，我们就来详细剖析一下 ByteBuf

ByteBuf结构

首先，我们先来了解一下 ByteBuf 的结构



以上就是一个 ByteBuf 的结构图，从上面这幅图可以看到

1. ByteBuf 是一个字节容器，容器里面的的数据分为三个部分，第一个部分是已经丢弃的字节，这部分数据是无效的；第二部分是可读字节，这部分数据是 ByteBuf 的主体数据，从 ByteBuf 里面读取的数据都来自这一部分;最后一部分的数据是可写字节，所有写到 ByteBuf 的数据都会写到这一段。最后一部分虚线表示的是该 ByteBuf 最多还能扩容多少容量
2. 以上三段内容是被两个指针给划分出来的，从左到右，依次是读指针（readerIndex）、写指针（writerIndex），然后还有一个变量 capacity，表示 ByteBuf 底层内存的总容量
3. 从 ByteBuf 中每读取一个字节，readerIndex 自增1，

ByteBuf 里面总共有 writerIndex-readerIndex 个字节可读，由此可以推论出当 readerIndex 与 writerIndex 相等的时候，ByteBuf 不可读

4. 写数据是从 writerIndex 指向的部分开始写，每写一个字节，writerIndex 自增1，直到增到 capacity，这个时候，表示 ByteBuf 已经不可写了
5. ByteBuf 里面其实还有一个参数 maxCapacity，当向 ByteBuf 写数据的时候，如果容量不足，那么这个时候可以进行扩容，直到 capacity 扩容到 maxCapacity，超过 maxCapacity 就会报错

Netty 使用 ByteBuf 这个数据结构可以有效地区分可读数据和可写数据，读写之间相互没有冲突，当然，ByteBuf 只是对二进制数据的抽象，具体底层的实现我们在下面的小节会讲到，在这一小节，我们只需要知道 Netty 关于数据读写只认 ByteBuf，下面，我们就来学习一下 ByteBuf 常用的 API

容量 API

```
capacity()
```

表示 ByteBuf 底层占用了多少字节的内存（包括丢弃的字节、可读字节、可写字节），不同的底层实现机制有不同的计算方式，后面我们讲 ByteBuf 的分类的时候会讲到

```
maxCapacity()
```

表示 ByteBuf 底层最大能够占用多少字节的内存，当向 ByteBuf 中写数据的时候，如果发现容量不足，则进行扩容，直到扩容到 maxCapacity，超过这个数，就抛异常

> readableBytes() 与 isReadable()

readableBytes() 表示 ByteBuf 当前可读的字节数，它的值等于 writerIndex-readerIndex，如果两者相等，则不可读，isReadable() 方法返回 false

> writableBytes()、 isWritable() 与 maxWritableBytes()

writableBytes() 表示 ByteBuf 当前可写的字节数，它的值等于 capacity-writerIndex，如果两者相等，则表示不可写，isWritable() 返回 false，但是这个时候，并不代表不能往 ByteBuf 中写数据了，
如果发现往 ByteBuf 中写数据写不进去的话，Netty 会自动扩容 ByteBuf，直到扩容到底层的内存大小为 maxCapacity，而 maxWritableBytes() 就表示可写的最大字节数，它的值等于 maxCapacity-writerIndex

读写指针相关的 API

> readerIndex() 与 readerIndex(int)

前者表示返回当前的读指针 readerIndex, 后者表示设置读指针

> writeIndex() 与 writeIndex(int)

前者表示返回当前的写指针 writerIndex, 后者表示设置写指针

markReaderIndex() 与 resetReaderIndex()

前者表示把当前的读指针保存起来，后者表示把当前的读指针恢复到之前保存的值，下面两段代码是等价的

```
// 代码片段1
int readerIndex = buffer.readerIndex();
// .. 其他操作
buffer.readerIndex(readerIndex);


// 代码片段二
buffer.markReaderIndex();
// .. 其他操作
buffer.resetReaderIndex();
```

希望大家多多使用代码片段二这种方式，不需要自己定义变量，无论 buffer 当作参数传递到哪里，调用 resetReaderIndex() 都可以恢复到之前的状态，在解析自定义协议的数据包的时候非常常见，推荐大家使用这一对 API

markWriterIndex() 与 resetWriterIndex()

这一对 API 的作用与上述一对 API 类似，这里不再 赘述

读写 API

本质上，关于 ByteBuf 的读写都可以看作从指针开始的地方开始读写数据

writeBytes(byte[] src) 与 buffer.readBytes(byte[] dst)

writeBytes() 表示把字节数组 src 里面的数据全部写到 ByteBuf，而 readBytes() 指的是把 ByteBuf 里面的数据全部读取到 dst，这里 dst 字节数组的大小通常等于 readableBytes()，而 src 字节数组大小的长度通常小于等于 writableBytes()

writeByte(byte b) 与 buffer.readByte()

writeByte() 表示往 ByteBuf 中写一个字节，而 buffer.readByte() 表示从 ByteBuf 中读取一个字节，类似的 API 还有 writeBoolean()、writeChar()、writeShort()、writeInt()、writeLong()、writeFloat()、writeDouble()
与 readBoolean()、readChar()、readShort()、readInt()、readLong()、readFloat()、readDouble() 这里就不一一赘述了，相信读者应该很容易理解这些 API

与读写 API 类似的 API 还有 getBytes、getByte() 与 setBytes()、setByte() 系列，唯一的区别就是 get/set 不会改变读写指针，而 read/write 会改变读写指针，这点在解析数据的时候千万要注意

release() 与 retain()

由于 Netty 使用了堆外内存，而堆外内存是不被 jvm 直接管理的，也就是说申请到的内存无法被垃圾回收器直接回收，所以需要我们手动回收。有点类似于c语言里面，申请到的内存必须手工释放，否则会造成内存泄漏。

Netty 的 ByteBuf 是通过引用计数的方式管理的，如果一个 ByteBuf 没有地方被引用到，需要回收底层内存。默认情况下，当创建完一个 ByteBuf，它的引用为1，然后每次调用 retain() 方法，
它的引用就加一， release() 方法原理是将引用计数减一，减完之后如果发现引用计数为0，则直接回收 ByteBuf 底层的内存。

slice()、duplicate()、copy()

这三个方法通常情况会放到一起比较，这三者的返回值都是一个新的 ByteBuf 对象

1. slice() 方法从原始 ByteBuf 中截取一段，这段数据是从 readerIndex 到 writeIndex，同时，返回的新的 ByteBuf 的最大容量 maxCapacity 为原始 ByteBuf 的 readableBytes()
2. duplicate() 方法把整个 ByteBuf 都截取出来，包括所有的数据，指针信息
3. slice() 方法与 duplicate() 方法的相同点是：底层内存以及引用计数与原始的 ByteBuf 共享，也就是说经过 slice() 或者 duplicate() 返回的 ByteBuf 调用 write 系列方法都会影响到原始的 ByteBuf，但是它们都维持着与原始 ByteBuf 相同的内存引用计数和不同的读写指针
4. slice() 方法与 duplicate() 不同点就是：slice() 只截取从 readerIndex 到 writerIndex 之间的数据，它返回的 ByteBuf 的最大容量被限制到 原始 ByteBuf 的 readableBytes(), 而 duplicate() 是把整个 ByteBuf 都与原始的 ByteBuf 共享
5. slice() 方法与 duplicate() 方法不会拷贝数据，它们只是通过改变读写指针来改变读写的行为，而最后一个方法 copy() 会直接从原始的 ByteBuf 中拷贝所有的信息，包括读写指针以及底层对应的数据，因此，往 copy() 返回的 ByteBuf 中写数据不会影响到原始的 ByteBuf
6. slice() 和 duplicate() 不会改变 ByteBuf 的引用计数，所以原始的 ByteBuf 调用 release() 之后发现引用计数为零，就开始

释放内存，调用这两个方法返回的 ByteBuf 也会被释放，这个时候如果再对它们进行读写，就会报错。因此，我们可以通过调用一次 retain() 方法
来增加引用，表示它们对应的底层的内存多了一次引用，引用计数为2，在释放内存的时候，需要调用两次 release() 方法，将引用计数降到零，才会释放内存

7. 这三个方法均维护着自己的读写指针，与原始的 ByteBuf 的读写指针无关，相互之间不受影响

retainedSlice() 与 retainedDuplicate()

相信读者应该已经猜到这两个 API 的作用了，它们的作用是在截取内存片段的同时，增加内存的引用计数，分别与下面两段代码等价

```
// retainedSlice 等价于
slice().retain();

// retainedDuplicate() 等价于
duplicate().retain()
```

使用到 slice 和 duplicate 方法的时候，千万要理清内存共享，引用计数共享，读写指针不共享几个概念，下面举两个常见的易犯错的例子

1. 多次释放

```
Buffer buffer = xxx;
doWith(buffer);
// 一次释放
buffer.release();


public void doWith(Bytebuf buffer) {
// ...

// 没有增加引用计数
Buffer slice = buffer.slice();

foo(slice);

}


public void foo(ByteBuf buffer) {
    // read from buffer

    // 重复释放
    buffer.release();
}
```

这里的 doWith 有的时候是用户自定义的方法，有的时候是 Netty 的回调方法，比如 channelRead() 等等

2. 不释放造成内存泄漏

```
Buffer buffer = xxx;
doWith(buffer);
// 引用计数为2，调用 release 方法之后，引用计数为1，无法释放内存
buffer.release();


public void doWith(Bytebuf buffer) {
// ...

// 增加引用计数
Buffer slice = buffer.retainedSlice();

foo(slice);

// 没有调用 release

}


public void foo(ByteBuf buffer) {
    // read from buffer
}
```

想要避免以上两种情况发生，大家只需要记得一点，在一个函数体里面，只要增加了引用计数（包括 ByteBuf 的创建和手动调用 retain() 方法），就必须调用 release() 方法

实战

了解了以上 API 之后，最后我们使用上述 API 来 写一个简单的 demo

ByteBufTest.java

```
public class ByteBufTest {
    public static void main(String[] args) {
        ByteBuf buffer =
ByteBufAllocator.DEFAULT.buffer(9, 100);

        print("allocate ByteBuf(9, 100)",
buffer);

        // write 方法改变写指针，写完之后写指针未到
capacity 的时候，buffer 仍然可写
        buffer.writeBytes(new byte[]{1, 2, 3,
4});
        print("writeBytes(1,2,3,4)", buffer);

        // write 方法改变写指针，写完之后写指针未到
capacity 的时候，buffer 仍然可写，写完 int 类型之后，写
指针增加4
        buffer.writeInt(12);
        print("writeInt(12)", buffer);

        // write 方法改变写指针，写完之后写指针等于
capacity 的时候，buffer 不可写
        buffer.writeBytes(new byte[]{5});
        print("writeBytes(5)", buffer);

        // write 方法改变写指针，写的时候发现 buffer
不可写则开始扩容，扩容之后 capacity 随即改变
        buffer.writeBytes(new byte[]{6});
        print("writeBytes(6)", buffer);
```

```
        // get 方法不改变读写指针
        System.out.println("getByte(3) return: "
+ buffer.getByte(3));
        System.out.println("getShort(3) return: "
+ buffer.getShort(3));
        System.out.println("getInt(3) return: " +
buffer.getInt(3));
        print("getByte()", buffer);


        // set 方法不改变读写指针
        buffer.setByte(buffer.readableBytes() +
1, 0);
        print("setByte()", buffer);

        // read 方法改变读指针
        byte[] dst = new
byte[buffer.readableBytes()];
        buffer.readBytes(dst);
        print("readBytes(" + dst.length + ")",
buffer);

    }

    private static void print(String action,
ByteBuf buffer) {
        System.out.println("after ==========" +
action + "============");
        System.out.println("capacity(): " +
buffer.capacity());
        System.out.println("maxCapacity(): " +
buffer.maxCapacity());
```

```
        System.out.println("readerIndex(): " +
buffer.readerIndex());
        System.out.println("readableBytes(): " +
buffer.readableBytes());
        System.out.println("isReadable(): " +
buffer.isReadable());
        System.out.println("writerIndex(): " +
buffer.writerIndex());
        System.out.println("writableBytes(): " +
buffer.writableBytes());
        System.out.println("isWritable(): " +
buffer.isWritable());
        System.out.println("maxWritableBytes(): "
+ buffer.maxWritableBytes());
        System.out.println();
    }
}
```

最后，控制台输出

```
after ===========allocate ByteBuf(9,
100)============
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 0
isReadable(): false
writerIndex(): 0
writableBytes(): 9
isWritable(): true
maxWritableBytes(): 100

after ===========writeBytes(1,2,3,4)============
```

```
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 4
isReadable(): true
writerIndex(): 4
writableBytes(): 5
isWritable(): true
maxWritableBytes(): 96

after ===========writeInt(12)===========
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 8
isReadable(): true
writerIndex(): 8
writableBytes(): 1
isWritable(): true
maxWritableBytes(): 92

after ===========writeBytes(5)===========
capacity(): 9
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 9
isReadable(): true
writerIndex(): 9
writableBytes(): 0
isWritable(): false
maxWritableBytes(): 91

after ===========writeBytes(6)===========
```

```
capacity(): 64
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 10
isReadable(): true
writerIndex(): 10
writableBytes(): 54
isWritable(): true
maxWritableBytes(): 90

getByte(3) return: 4
getShort(3) return: 1024
getInt(3) return: 67108864
after ===========getByte()===========
capacity(): 64
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 10
isReadable(): true
writerIndex(): 10
writableBytes(): 54
isWritable(): true
maxWritableBytes(): 90

after ===========setByte()===========
capacity(): 64
maxCapacity(): 100
readerIndex(): 0
readableBytes(): 10
isReadable(): true
writerIndex(): 10
writableBytes(): 54
isWritable(): true
```

```
maxWritableBytes(): 90

after ==========readBytes(10)==========
capacity(): 64
maxCapacity(): 100
readerIndex(): 10
readableBytes(): 0
isReadable(): false
writerIndex(): 10
writableBytes(): 54
isWritable(): true
maxWritableBytes(): 90
```

> 完整代码已放置 github (https://github.com/lightningMan/flash-netty/tree/%E6%95%B0%E6%8D%AE%E4%BC%A0%E8%BE%9

相信大家在了解了 ByteBuf 的结构之后，不难理解控制台的输出

总结

1. 本小节，我们分析了 Netty 对二进制数据的抽象 ByteBuf 的结构，本质上它的原理就是，它引用了一段内存，这段内存可以是堆内也可以是堆外的，然后用引用计数来控制这段内存是否需要被释放，使用读写指针来控制对 ByteBuf 的读写，可以理解为是外观模式的一种使用
2. 基于读写指针和容量、最大可扩容容量，衍生出一系列的读写方法，要注意 read/write 与 get/set 的区别
3. 多个 ByteBuf 可以引用同一段内存，通过引用计数来控制内存的释放，遵循谁 retain() 谁 release() 的原则
4. 最后，我们通过一个具体的例子说明 ByteBuf 的实际使用

思考

slice 方法可能用在什么场景？欢迎留言讨论。