零拷贝机制

Netty自己的ByteBuf

ByteBuf是为解决ByteBuffer的问题和满足网络应用程序开发人员的日常需求而设计的。

JDK ByteBuffer的缺点：

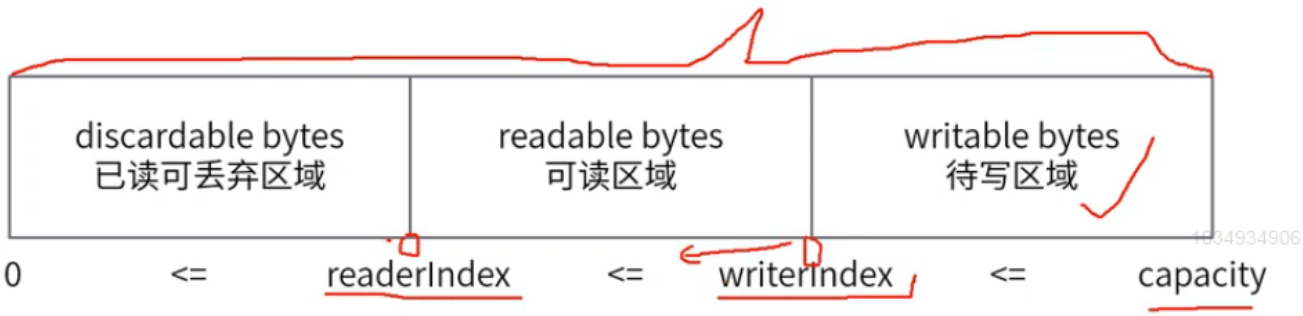
1. 无法动态扩容，不能动态扩展和收缩，当数据大于ByteBuffer容量时，会发生索引越界异常。
2. API使用复杂，读写的时候需要手工调用filp()和rewind()方法，使用时需要非常谨慎的使用这些api，否则很容易出错。

ByteBuf做了哪些增强

1. API操作便捷性。
2. 动态扩容。
3. 多种ByteBuf实现。
4. 高效的零拷贝机制。

ByteBuf操作

下图显示了一个缓冲区是如何被两个指针分割成三个区域的：




```
public class ByteBufDemo {
    @Test
    public void apiTest() {
        //  +-------------------+------------------+------------------+
        //  | discardable bytes |  readable bytes  |  writable bytes  |
        //  |                   |     (CONTENT)    |                  |
        //  +-------------------+------------------+------------------+
        //  |                   |                  |                  |
        //  0      <=       readerIndex   <=   writerIndex    <=    capacity

        // 1.创建一个非池化的ByteBuf，大小为10个字节
        ByteBuf buf = Unpooled.buffer(10);
        System.out.println("原始ByteBuf为====================>" + buf.toString());
        System.out.println("1.ByteBuf中的内容为===============>" +
Arrays.toString(buf.array()) + "\n");

        // 2.写入一段内容
```

```
        byte[] bytes = {1, 2, 3, 4, 5};
        buf.writeBytes(bytes);
        System.out.println("写入的bytes为==================>" + Arrays.toString(bytes));
        System.out.println("写入一段内容后ByteBuf为==========>" + buf.toString());
        System.out.println("2.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");

        // 3.读取一段内容
        byte b1 = buf.readByte();
        byte b2 = buf.readByte();
        System.out.println("读取的bytes为==================>" + Arrays.toString(new byte[]
{b1, b2}));
        System.out.println("读取一段内容后ByteBuf为==========>" + buf.toString());
        System.out.println("3.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");

        // 4.将读取的内容丢弃
        buf.discardReadBytes();
        System.out.println("将读取的内容丢弃后ByteBuf为========>" + buf.toString());
        System.out.println("4.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");

        // 5.清空读写指针
        buf.clear();
        System.out.println("将读写指针清空后ByteBuf为==========>" + buf.toString());
        System.out.println("5.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");

        // 6.再次写入一段内容，比第一段内容少
        byte[] bytes2 = {1, 2, 3};
        buf.writeBytes(bytes2);
        System.out.println("写入的bytes为==================>" + Arrays.toString(bytes2));
        System.out.println("写入一段内容后ByteBuf为==========>" + buf.toString());
        System.out.println("6.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");

        // 7.将ByteBuf清零
        buf.setZero(0, buf.capacity());
        System.out.println("将内容清零后ByteBuf为==============>" + buf.toString());
        System.out.println("7.ByteBuf中的内容为================>" +
Arrays.toString(buf.array()) + "\n");

        // 8.再次写入一段超过容量的内容
        byte[] bytes3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
        buf.writeBytes(bytes3);
        System.out.println("写入的bytes为==================>" + Arrays.toString(bytes3));
        System.out.println("写入一段内容后ByteBuf为==========>" + buf.toString());
        System.out.println("8.ByteBuf中的内容为==============>" +
Arrays.toString(buf.array()) + "\n");
        //  随机访问索引 getByte
        //  顺序读 read*
        //  顺序写 write*
        //  清除已读内容 discardReadBytes
```

```
        //  清除缓冲区 clear
        //  搜索操作
        //  标记和重置
        //  完整代码示例：参考
        // 搜索操作 读取指定位置 buf.getByte(1);
        //
    }
}
原始ByteBuf为
==================>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx:
0, widx: 0, cap: 10)
1.ByteBuf中的内容为===============>[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

写入的bytes为===================>[1, 2, 3, 4, 5]
写入一段内容后ByteBuf为
===========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0, widx:
5, cap: 10)
2.ByteBuf中的内容为===============>[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]

读取的bytes为====================>[1, 2]
读取一段内容后ByteBuf为
===========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 2, widx:
5, cap: 10)
3.ByteBuf中的内容为===============>[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]

将读取的内容丢弃后ByteBuf为
========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0, widx: 3,
cap: 10)
4.ByteBuf中的内容为===============>[3, 4, 5, 4, 5, 0, 0, 0, 0, 0]

将读写指针清空后ByteBuf为
==========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0, widx: 0,
cap: 10)
5.ByteBuf中的内容为===============>[3, 4, 5, 4, 5, 0, 0, 0, 0, 0]

写入的bytes为====================>[1, 2, 3]
写入一段内容后ByteBuf为
===========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0, widx:
3, cap: 10)
6.ByteBuf中的内容为===============>[1, 2, 3, 4, 5, 0, 0, 0, 0, 0]

将内容清零后ByteBuf为
==============>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0,
widx: 3, cap: 10)
7.ByteBuf中的内容为================>[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

写入的bytes为====================>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
写入一段内容后ByteBuf为
===========>UnpooledByteBufAllocator$InstrumentedUnpooledUnsafeHeapByteBuf(ridx: 0, widx:
14, cap: 64)
8.ByteBuf中的内容为===============>[0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

ByteBuf动态扩容

1. capacity默认值：256字节、最大值：Integer.MAX_VALUE(2GB)。
2. write*方法调用时，通过AbstractByteBuf.ensureWritable0进行检查。容量计算方法：AbstractByteBufAllocator.calculateNewCapacity(新capacity的最小要求，capacity最大值)。

   根据新capacity的最小值要求，对应有两套计算方法：
3. 没超过4兆：从64字节开始，每次增加一倍，直至计算出来的newCapacity满足新容量的最小要求。比如：

   当前大小256，已写250，继续写10字节数据，需要的容量最小要求261，则新容量512。
4. 超过4兆：新容量=新容量最小要求/4兆 * 4兆 + 4兆，比如：当前大小3兆，已写3兆，继续写2兆数据，需要的容量最小是5兆，则新容量是9兆(不能超过最大值)。

选择合适的ByteBuf实现

堆内/堆外	是否池化	访问方式	具体实现类	备注
heap堆内	unpool	safe	UnpooledHeapByteBuf	数组实现
		unsafe	UnpooledUnsafeHeapByteBuf	Unsafe类直接操作内存
	pool	safe	PooledHeapByteBuf	
		unsafe	PooledUnsafeHeapByteBuf	~
direct堆外	unpool	safe	UnpooledDirectByteBuf	NIO DirectByteBuffer
		unsafe	UnpooledUnsafeDirectByteBuf	~
	pool	safe	PooledDirectByteBuf	~
		unsafe	PooledUnsafeDirectByteBuf	~

在使用中，都是通过ByteBufAllocator分配器进行申请，同时分配器具备有内存管理的功能

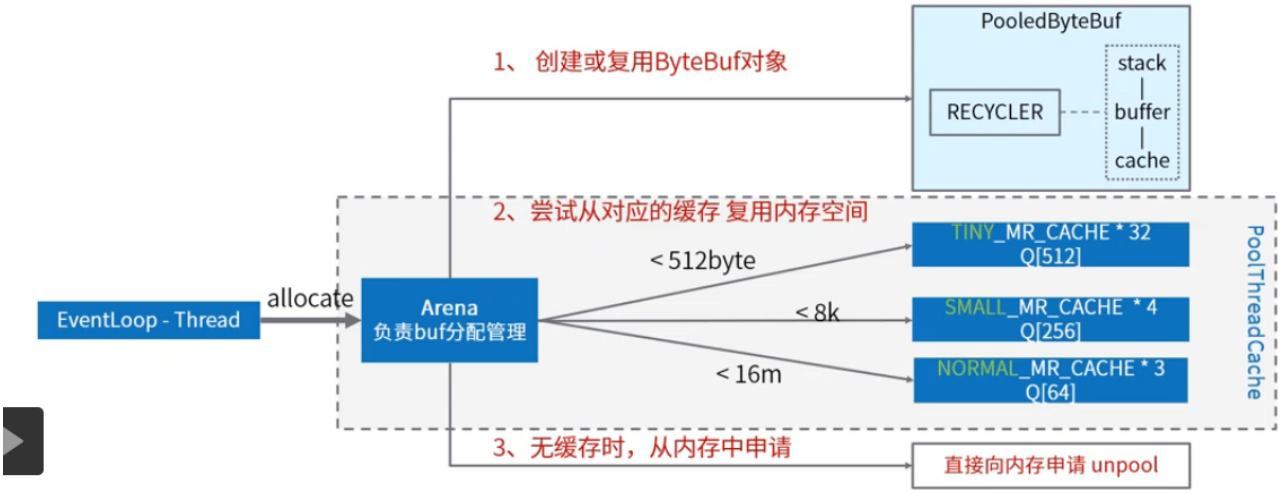
Unsafe的实现

unsafe意味着不安全的操作。但是更底层的操作会带来性能提升和特殊功能，Netty会尽力使用unsafe。Java语言很重要的特性是"一次编写到处运行"，所以它针对底层的内存或者其他操作，做了很多封装。而Unsafe提供了一系列我们操作底层的方法，可能会导致不兼容或者不可知的异常。

PooledByteBuf对象、内存复用

PoolThreadCache：PooledByteBufAllocator实例维护的一个线程变量。等多种分类的MemoryRegionCache数组用作内存缓存，MemoryRegionCache内部是链表，队列里面存Chunk。PoolChunk里面维护了内存引用，内存复用的做法就是把buf的memory指向chunk的memory。

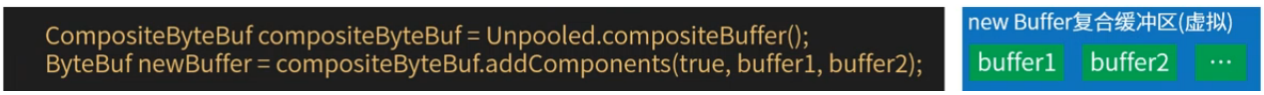
PooledByteBufAllocator.ioBuffer运作过程梳理：



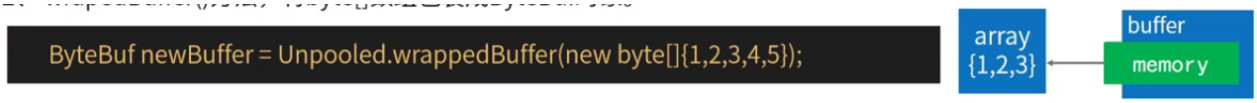
零拷贝机制

Netty的零拷贝机制，是一种应用层实现。和底层JVM、操作系统内存机制并无过多关联。

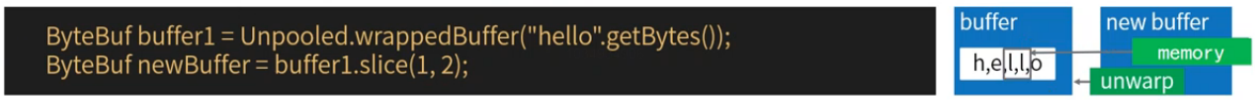
1. CompositeByteBuf，将多个ByteBuf合并为一个逻辑上的ByteBuf，避免各个ByteBuf之间的拷贝。



2. wrapedBuffer()方法，将byte[]数组包装成ByteBuf对象。



3. slice()方法。将一个ByteBuf对象切分成多个ByteBuf对象。




```
public class ZeroCopyTest {
    @org.junit.Test
    public void wrapTest() {
        byte[] arr = {1, 2, 3, 4, 5};
        ByteBuf byteBuf = Unpooled.wrappedBuffer(arr);
        System.out.println(byteBuf.getByte(4));
        arr[4] = 6;
        System.out.println(byteBuf.getByte(4));
    }

    @org.junit.Test
    public void sliceTest() {
        ByteBuf buffer1 = Unpooled.wrappedBuffer("hello".getBytes());
        ByteBuf newBuffer = buffer1.slice(1, 2);
        newBuffer.unwrap();
```

```
        System.out.println(newBuffer.toString());
    }

    @org.junit.Test
    public void compositeTest() {
        ByteBuf buffer1 = Unpooled.buffer(3);
        buffer1.writeByte(1);
        ByteBuf buffer2 = Unpooled.buffer(3);
        buffer2.writeByte(4);
        CompositeByteBuf compositeByteBuf = Unpooled.compositeBuffer();
        CompositeByteBuf newBuffer = compositeByteBuf.addComponents(true, buffer1,
buffer2);
        System.out.println(newBuffer);
    }
}

5
6
UnpooledSlicedByteBuf(ridx: 0, widx: 2, cap: 2/2, unwrapped: UnpooledHeapByteBuf(ridx:
0, widx: 5, cap: 5/5))
CompositeByteBuf(ridx: 0, widx: 2, cap: 2, components=2)
```