# Netty中的ByteBuf

**Byte字节**

java中byte类型被定义为二进制字节，一说到二进制，那么联想到二进制文件视频、图片等等、也就是多用于对File进行是I/O操作的时候

**Buffer缓冲区**

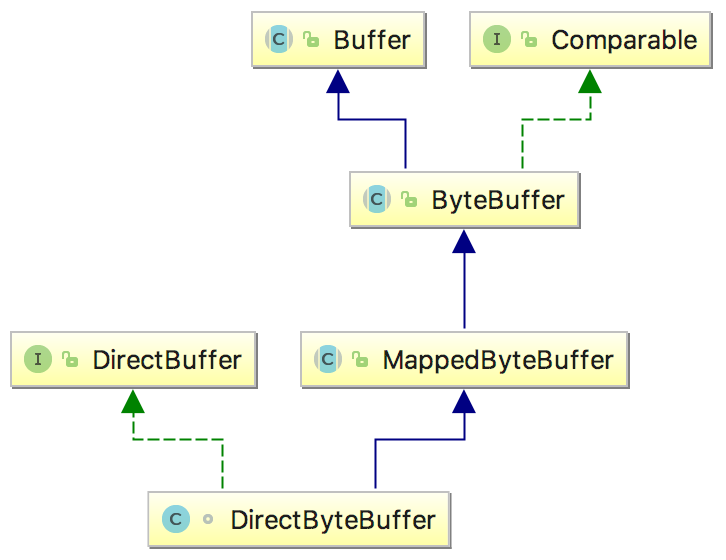
**缓冲区(Buffer)就是在内存中预留指定大小的存储空间用来对输入/输出(I/O)的数据作临时存储，这部分预留的内存空间就叫做缓冲区：**

这个其实是很好理解的日常查看视频、经常出现请等待缓存，视频就是一堆图片组成一帧一张图片，大部分视频都通过编码压缩，那么加载的时候不可能一帧加载完了就给你展示.涉及**视频编码知识**;视频一帧一帧加载成了一个片段才给你播出.这个就是缓冲区;

java中为什么要缓冲区？很简单，你一个new出来的对象能复用就多复用；尽量减少声明新变量；不仅仅是代码美观干净，也是为了减少内存损耗； 同样的缓冲区也是如此；即**重用**、**减少内存损耗。**

**为了避免 Java 垃圾收集器不可预测的行为以及额外的性能开销，这些产品（Netty、Flink等）一般倾向于使用 JVM 之外的内存来存储和管理数据。这样的数据，就是我们常说的堆外数据（off-heap data）。**

**使用堆外存储最常用的办法，就是使用 ByteBuffer 这个类来分配直接存储空间（direct buffer）。JVM 虚拟机会尽最大努力直接在直接存储空间上执行 IO 操作，避免数据在本地和 JVM 之间的拷贝。**



**Nio的ByteBuffer**

基本方法使用这个demo过一遍就ok、jdk 1.4以上 即可

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 | import io.netty.buffer.ByteBuf; import io.netty.buffer.CompositeByteBuf; import io.netty.buffer.Unpooled; import org.junit.Test;  import java.nio.ByteBuffer;  public class NettyDemo {  @Test  public void TestByteBuf()  {   ByteBuffer byteBuffer = ByteBuffer.allocate(88);  // 一种新的动态缓冲区被创建。在内部，实际缓冲区是被“懒”创建，从而避免潜在的浪费内存空间。  ByteBuf b = Unpooled.buffer(4);  System.out.println(b.capacity());  // 当第一个执行写尝试，内部指定初始容量 4 的缓冲区被创建  b.writeByte('1');   b.writeByte('2');  b.writeByte('3');  b.writeByte('4');  System.out.println(b.capacity());  // 当写入的字节数超过初始容量 4 时，  //内部缓冲区自动分配具有较大的容量  b.writeByte('5');  System.out.println(b.capacity());  } } |

**缺陷**

第一点

ByteBuffer的长度**是固定的**，一旦分配完成就不能进行扩容和收缩，当需要操作的对象大于Buffer的容量时，会发生异常；

第二点

在进行读写状态切换时，需要调用flip()或rewind()方法改变指针position的位置，稍有不慎，就不能完成任务；

**Netty的ByteBuf**

netty为了修复上面两个蛋痛的问题，重写了ByteBuffer、即ByteBuf；

**自动扩容**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | @Test public void TestByteBuf() {   ByteBuffer byteBuffer = ByteBuffer.allocate(88);  // 一种新的动态缓冲区被创建。在内部，实际缓冲区是被“懒”创建，从而避免潜在的浪费内存空间。  ByteBuf b = Unpooled.buffer(4);  System.out.println(b.capacity());  // 当第一个执行写尝试，内部指定初始容量 4 的缓冲区被创建  b.writeByte('1');   b.writeByte('2');  b.writeByte('3');  b.writeByte('4');  System.out.println(b.capacity());  // 当写入的字节数超过初始容量 4 时，  //内部缓冲区自动分配具有较大的容量  b.writeByte('5');  System.out.println(b.capacity()); } |

注意：

1. 当申请的新空间大于阀值时，采用每次步进4MB的方式进行扩张内存，而不是倍增，因为这会造成内存膨胀和浪费
2. 而但申请的新空间小于阀值时，则以64为基数进行倍增而不是步进，因为当内存比较小的时候，倍增是可以接受的（64 -> 128 和 10Mb -> 20Mb相比链接！

**位置指针**

ByteBuf通过两个位置指针来协助缓冲区的读写操作，读操作使用readerIndex，写操作使用writerIndex。  
readerIndex和writerIndex的取值一开始都是0，随着数据的写入writerIndex会增加，读取数据会使readerIndex增加，但是它不会超过writerIndex。在读取之后，0～readerIndex的就被视为discard的，调用discardReadBytes方法，可以释放这部分空间，它的作用类似ByteBuffer的compact方法。readerIndex和writerIndex之间的数据是可读取的，等价于ByteBuffer position和limit之间的数据。writerIndex和capacity之间的空间是可写的，等价于ByteBuffer limit和capacity之间的可用空间。  
**由于写操作不修改readerIndex指针，读操作不修改writerIndex指针，因此读写之间不再需要调整位置指针**，这极大地简化了缓冲区的读写操作，避免了由于遗漏或者不熟悉flip()操作导致的功能异常。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | @Test public void TestByteBufReadWriteIndex() {  ByteBuf b = Unpooled.buffer();  System.out.println("读:"+b.readerIndex());  System.out.println("写:"+b.writerIndex());  b.writeByte('1');  b.writeByte('2');  b.writeByte('3');  b.writeByte('4');  b.writeByte('5');  System.out.println("读:"+b.readerIndex());  System.out.println("写:"+b.writerIndex());   b.readBytes(5);  System.out.println("读:"+b.readerIndex());  System.out.println("写:"+b.writerIndex()); } |

**复合缓冲**

最好使用我的这个netty版本

|  |  |
| --- | --- |
| 1 2 3 4 5 | <dependency>  <groupId>io.netty</groupId>  <artifactId>netty-all</artifactId>  <version>4.1.30.Final</version> </dependency> |

测试版本过低这段代码报错

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | @Test  public void TestByteBuffhhc()  {  //组合缓冲区  CompositeByteBuf compBuf = Unpooled.compositeBuffer(); //堆缓冲区  ByteBuf heapBuf = Unpooled.buffer(8); //直接缓冲区  ByteBuf directBuf = Unpooled.directBuffer(16); //添加ByteBuf到CompositeByteBuf  compBuf.addComponents(heapBuf, directBuf); //删除第一个ByteBuf  compBuf.removeComponent(0);  Iterator<ByteBuf> iter = compBuf.iterator();  while(iter.hasNext()){  System.out.println(iter.next().toString());  }  //使用数组访问数据  if(!compBuf.hasArray()){  int len = compBuf.readableBytes();  byte[] arr = new byte[len];  compBuf.getBytes(0, arr);  }  } |