Netty-Buffer类都集中在io.netty.buffer的package中，主要功能是在数据传输时保存传输的数据，同时对通信数据进行功能封装，便于对数据空间进行管理。注：在其他的package中有一些继承自buffer的类，不在这里说明。

package如下：

**工作原理简介：**

**一、从存储方式上来分，ByteBuf分为：HeapByteBuf和DirectByteBuf。**

HeapByteBuf是用byte数组格式来存储数据，DirectByteBuf是使用java.nio.ByteBuffer来存储数据。

DirectByteBuf利用java.nio.ByteBuffer是借助于JVM调用操作系统的底层通信函数，直接操作直接缓冲区可以减少中间缓冲区的复制操作，进而提供程序性能。

HeapByteBuf是在JVM内部开辟缓冲区，在数据操作前先把数据复制到byte数组再进行处理。因为中间增加了一层数据复制操作，所以会影响性能。

**二、从空间初始化方式上来分，ByteBuf分为：缓存方式分配和非缓存方式分配。**

空间分配工具类：PooledByteBufAllocator和UnpooledByteBufAllocator，其中UnpooledByteBufAllocator在外又封装了一个类Unpooled。

在缓存方式的分配类中，预申请了8个16K空间(分别是byte数组和java.nio.ByteBuffer)放在缓存中，通过类似工厂的创建函数创建ByteBuf时，先检查缓存中的空间是否符合要求，如果符合先从缓存中分配空间，否则申请新的空间。

空间申请时判断为：大于chunkSize（16K），大于PageSize（8192）小于chunkSize（16K），小于PageSize大于512，小于512几个档。

注：这几个值会根据操作系统的不同有所调整，以上值是在win7x64系统上的值。

**三、空间分配和扩展**

**1、HeapByteBuf**

    ①缓存方式

       a、分配

        根据构建是传入参数的初始化空间，从对应的缓存空间中取得byte[]，分配给到ByteBuf中memory变量。

       b、扩展

       在需要扩展的空间时，从缓存中取出一个符合要求的新的空间，即byte数组，把新申请的byte数组替换到ByteBuf中，执行System.arraycopy(src, srcOffset, dst, dstOffset, length);即可

    ②非缓存方式

      a、分配

       new一个byte[]赋值到ByteBuf的memory中，长度是传入的初始化长度参数。

      b、扩展

      new一个新长度的byte[]，替换ByteBuf中旧的memory，然后执行System.arraycopy(src, srcOffset, dst, dstOffset, length);即可

**2、DirectByteBuf**

    ①缓存方式

      a、分配

      根据入口的初始化大小参数从缓存中取得对应的空间区域进行分配。

      b、扩展

      根据申请空间的要求，从缓存中取得（缓存中无法满足时new一个java.nio.ByteBuffer）满足要求的java.nio.ByteBuffer，替换掉原有的java.nio.ByteBuffer，如果系统判断为unSafe时，直接复制旧的数据到新的java.nio.ByteBuffer；如果系统判断不为unSafe时，说明旧的数据还可能有其他Buffer访问，所以新旧java.nio.ByteBuffer同时进行自身复制一个新的java.nio.ByteBuffer，然后进行数据复制。

    ②非缓存方式

      a、分配

      new一个java.nio.ByteBuffer复制给ByteBuf的memory变量，长度为入口的初始化大小参数。

      b、扩展

      new一个新的java.nio.ByteBuffer复制给ByteBuf的memory变量，然后把数据复制到新的java.nio.ByteBuffer中。

**四、其他类型ByteBuf：**

**1、CompositeByteBuf**

      是一个虚拟的ByteBuf，在内部用List保存多个ByteBuf，然后虚拟为一个ByteBuf供使用。应用场景未知。

     同类型：FixedCompositeByteBuf，List长度为固定长度

**2、ReadOnly类型**

      只读ByteBuf。包括：ReadOnlyByteBuf、ReadOnlyByteBufferBuf、ReadOnlyUnsafeDirectByteBuf等

**3、EmptyByteBuf**

      定义一个空的ByteBuf

**4、DuplicatedByteBuf**

      不建议直接构造此类ByteBuf。执行ByteBuf内部duplicate()生成一个本身的副本。

**5、WrappedByteBuf**

    未知

     SimpleLeakAwareByteBuf

**6、SwappedByteBuf**

    反转字序的ByteBuf

**7、SlicedByteBuf**

    分片的ByteBuf，通过ByteBuf的slice()方法返回，和调用者共享从position指针到limit指针的这部分内容。

**五、通信时的处理**

       在服务端和客户端通信时，先把ByteBuf的capacity初始化为1024，然后在接收数据的过程中，再根据每次接收到的数据长度，猜测（动态计算）下次传输可能的数据长度，然后记录这个长度，当下一次数据开始传输时把这个长度当做初始长度生成ByteBuf。然后依次循环使用，直到连接关闭为止。

**重点类介绍：**

**类继承关系图：**

**1、UnpooledByteBufAllocator**

    非缓存方式ByteBuf生成工具类，是ByteBufAllocator的一种简单实现，生成的方式是每次调用都new一个新的ByteBuf。

    提供了各种ByteBuf的实现方法。

**2、PooledByteBufAllocator**

   缓存方式的ByteBuf生成工具类。预生成了一个高性能的buffer池，分配策略则是结合了buddy allocation和slab allocation的jemalloc变种。

   这是官方推荐的工具类，由于在先前的测试中可能存在内存泄露的现象，所以在连接初始化时使用的UnpooledByteBufAllocator分配ByteBuf。但是在官方的最新公告中表示已经解决了此问题，以后还是推荐使用PooledByteBufAllocator来生成ByteBuf来提供整体性能。

**3、PoolArena**

    ByteBuf空间分配核心类。提供了allocate()方法来初始化ByteBuf空间，以及reallocate()方法来动态扩容空间。

    同时在继承类HeapArena和DirectArena类中提供了memoryCopy()函数的空间扩容的具体实现。

**与其他模块接口：**

**一、ByteBuf的生成**

     ByteBuf的生成不建议直接使用new一个具体类的方式来实现，netty提供了生成工具类PooledByteBufAllocator和UnpooledByteBufAllocator供用户使用。

**二、ByteBuf主要使用的方法说明**

**1、capacity()方法和capacity(newCapacity)**

      无参数时是返回buf现有的容量；

      有参数时是重新设置buf的容量，一般用于写入时容量不足的情况下进行扩容操作

**2、order()方法和order(ByteOrder endianness)方法**

      无参数时返回buf的字节序

      有参数时设置buf的字节序，同时依照新的字节序进行转换

**3、readableBytes()和readBytes(obj);**

     readableBytes()返回当前可读的字节数：从readIndex到结尾的字节数。一般用于读取数据前进行数据完整性的判断。

    readBytes()读取数据到指定的obj变量中（还有其他参数）。

**4、writableBytes()和maxWritableBytes()**

     writableBytes()是返回当前capacity可写入的字节数是多少；

    maxWritableBytes()是返回ByteBuf创建时的maxCapacity可写入的字节数是多少

**5、markReaderIndex()和resetReaderIndex()**

    markReaderIndex()把当前的readerIndex赋值到markReaderIndex中。

    resetReaderIndex()重设readerIndex，把markIndex赋值到readerIndex。

    这两个方法和readableBytes()、readBytes()结合使用可以完成数据的读取操作。

**6、markWriterIndex()和resetWriterIndex()**

    markWriterIndex()方法是把当前writeIndex赋值到markWriteIndex中。

    resetWriterIndex()是把writeIndex设置为markWriteIndex的值。

**7、writeBytes()**

    写数据函数，把数据吸入到ByteBuf中。

    一般在写数据时和markWriterIndex()和resetWriterIndex()结合使用

**8、 clear()**

    清空ByteBuf，同时把readerIndex、markReaderIndex、writeIndex、markWriteIndex等设置为0

**三、netty→ByteBuf和java.nio.ByteBuffer的不同**

   在java.nio.ByteBuffer有capacity、position、limit、mark四种概念，除了容量不会改变以外，position、limit和mark在读写时都会发生改变，并且在读操作前要调用flip()方法重设position和limit才可以正确读出写入的数据。

    而在netty的ByteBuf中，封装时重新定义了readerIndex和writeIndex，在读写时只是操作对应的标志位，开发者在使用当中读写时不用关心position、limit、mark，也不用执行flip()方法就可以很方便的读写操作。

    java.nio.Bytebuffer读写实例：

    ByteBuffer buf = ByteBuffer.allocate(10);

    buf.put(xxx);//写完毕转读取时

    buf.flip();//必须调用，否则读取的数据不正确

    buf.get(xxx);

    而使用netty的ByteBuf时不需要关心这些指针，如：

    ByteBuf buf = xxx；//生成新的ByteBuf

    byte writeData[] = {...};

    buf.writeBytes(writeData);//写入数据

    byte readData[] = new byte[11];

    buf.readBytes(readData);//不用其他操作，可直接读取

    通过上面的示例可以看出，netty的ByteBuf使用起来更方便

**四、HeapByteBuf和DirectByteBuf的选择（个人理解）**

   HeapByteBuf由Heap管理，Heap是Java堆的意思，内部实现直接采用byte数组；DirectByteBuf使用是堆外内存，Direct应是采用Direct I/O之意，内部实现使用java.nio.DirectByteBuffer。

   注：HeapByteBuf通过

   在使用上的选择：

   场景1：网络间的通信：

     A计算机的netty进程A1和B计算机上的netty进程A2进行通信，这个时候涉及到跨网络通信的数据传输，前面说过DirectByteBuf是JVM直接操作操作系统的直接缓冲区能够提供性能，那么此时使用DirectByteBuf有利于提高程序性能，减少资源的使用。如果此时使用HeapByteBuf，那么进程现在JVM内部分配缓冲区，写完数据后再赋值到操作系统的直接缓冲区，然后进行网络传输，这样缓冲区的复制会影响性能，同时使用完毕后操作系统和JVM都要回收缓冲区资源，实际上增加了资源的使用，也降低了程序的性能。

   场景2：进程内部的通信：

    同一台计算机上的netty进程内进行通信，这个时候只涉及到了进程内的通信、涉及不到网络间传输，这个时候使用HeapByteBuf会有更好的优势。

**测试结果对比：**

    测试环境描述：在同一个进程内启动server和client进行通信，先用一个线程启动server，然后用另一个线程延迟3秒启动client。

    通信时间是发送1000次消息，计算客户端发送消息一直到收到服务端返回此消息时的时间平均数，重复10次得到的最小值。

   1、使用DirectByteBuf时，最短时间为519120（纳秒），JVM情况如图

   2、使用HeapByteBuf时，最短通信时间为：576218（纳秒），JVM情况如图：

因为每次发送的消息为一个long型数值，所以从图上CPU看不到什么大的差别，但是在实际执行过程中经常看到0.x%的CPU跳动。

最明显的差别就是内存的占用，使用HeapByteBuf时是在JVM内部分配缓冲区，所以内存占用是使用DirectByteBuf时的好几倍，再加上在通信比较频繁的时候，JVM内存的占用会导致频繁的GC，CPU的使用率也会成倍的增加。所以建议在涉及到网络通信时使用DirectByteBuf。