第二章 Buffers

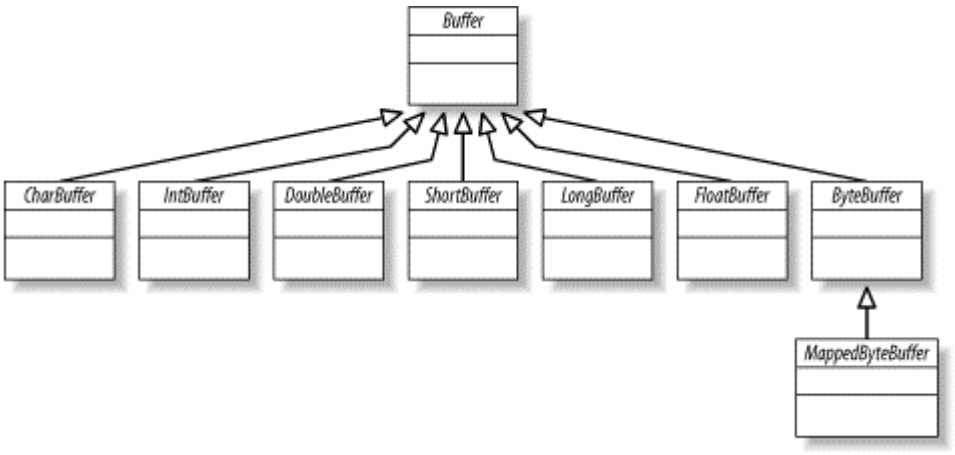
我们这次 java.nio 观光游的第一站就是 Buffers。Buffers 类是构建 java.nio 的基础。在这一章我们会详细分析 buffers，认识各种类型的 buffers 并学习怎样使用它们。我们还会弄清 java.nio 的 buffers 和 channel 之间的关联。

Buffer 对象就是可以持有固定数量数据的容器。它像是一个收集器或中转区，来存取数据。Buffer 可被填充[fill]或排出[drain]。每一种非布尔性的基本数据类型都一种对应的 buffer。虽然每种基本类型都有对应的 buffer，但 buffers 主要还是使用 byteBuffer。非字节的 buffer[non-byte Buffer]可以与 byteBuffer 相互转换，在 2.4.1 节 会讨论这种转化中的 byte-ordering 问题。在本章后续章节中我们就会讨论在 buffer 中存储数据的意义。

Buffer 和 Channel 联系密切。Channel 是 I/O 传输的门户，buffer 就是 channel 传输的目的或来源。对于输出来说，你会将要发送出去的数据放在 buffer 里，传给 Channel；对于输入来说，Channel 将数据接收到你提供的 buffer 中。这种 buffer 与其协作对象(通常是一个或多个 Channel)之间传递是高效处理数据的关键。Channel 将会在第三章详细讨论。

图 2-1 是 buffer 的层次关系图。顶上是公共的 Buffer 类，Buffer 类定义了一些公共的行为。我们的讨论就从这里开始。

Figure 2-1. The Buffer family tree



2.1 Buffer Basics

从概念上说，一个buffer就是包装了一个基本类型数组的对象。与简单数组比较，buffer的优势在于它将数据内容和数据内容的信息包装成一个对象。Buffer类及其具体的子类定义了处理数据缓冲的API。

2.1.1 属性

所有buffer 都有4个提供关于其持有的数据元素信息的属性。它们分别是：

Capacity 容量

元素的最大数量。容量是buffer创建时设置的并且不会被更改。

Limit 边界

是第一个不能被读取或写入的元素的索引。

Position 位置

是下一个要读取或写入的元素的索引。“位置”会被 相对的 get()或put()方法自动更新。

Mark 标记

是被记下的 “位置”。调用mark( ) 设置 mark = position；调用reset( ) 设置 position = mark。如果mark没有被设置那就是undefined。

标记、位置、边界和容量值遵守以下不变式：

0 <= 标记 <= 位置 <= 边界 <= 容量

0 <= mark <= position <= limit <= capacity

让我们看看这些属性变化在实际工作中的例子。图Figure 2-2显示了一个新建的，容量为10的Buffer的逻辑视图。

Figure 2-2. A newly created ByteBuffer

position 被设置为0,capacity 和limit 设置为 10，刚好超过缓冲区可容纳的最后一个字节的。

最初的mark时未定义的。 capacity 是固定的，其他的三个属性会在buffer的使用中变化。

2.1.2 Buffer API

现在就让我们看看如何使用它。下面就是Buffer 类的方法签名。

```
package java.nio;
public abstract class Buffer {
    public final int capacity( );
    public final int position( );
    public final Buffer position (int newPosition);
    public final int limit( );
    public final Buffer limit (int newLimit);
    public final Buffer mark( );
    public final Buffer reset( );
    public final Buffer clear( );
    public final Buffer flip( );
    public final Buffer rewind( );
    public final int remaining( );
    public final boolean hasRemaining( );
    public abstract boolean isReadOnly( );
}
```

对于这些API需要注意：这些方法看起来应该返回void，比如clear( )，但实际上却返回的是Buffer的自身引用。调用这些方法都会返回被调用对象的this引用 是一种“方法调用链”的class设计技术。链式调用允许你将这样的代码：

```
buffer.mark( );
buffer.position(5);
 buffer.reset( );
```

改写成这样的：

```
buffer.mark().position(5).reset( );
```

java.nio在设计的时候就考虑到了使用“链式调用”。你也许在之前使用*StringBuffer*时就见识过“链式调用”了。

> 当我们使用得当时，链式调用可以产生简洁、优雅、易读的代码；使用不当就会产生混乱。只有能改善可读性和使代码意图更明确的情况下才使用调用链；如果调用链威胁到了代码的明确意图，就别用。要保持代码的可读性嘛。

另一件需要注意的事就是isReadOnly( )方法。所有的buffer都是可读的(readable)，但不一定都是“可写的”(writable)。每个具体的buffer类都实现了该方法(isReadOnly( ))，来表明是否可以修改buffer的内容。有一些buffer的子类可能不会将数据保存在内部的数组中的，如MappedByteBuffer，它的内容就有可能对应一个只读文件。你当然也可以显式地创建只读的视图缓冲器，来保护数据不会被意外修改。尝试修改一个read-only buffer会抛出ReadOnlyBufferException 异常。但我们之前可以通过isReadOnly()知道能不能修改。

2.1.3 Accessing

我们从头开始讨论。buffer 管理了固定数量的数据元素，但在任意时刻，我们只需关注 buffer 中的部分元素。就是说，在我们排出[drain]buffer 之前先要填充[fill] 一些元素。我们需要一种记录 加入到 buffer 中的元素个数的方法，以及记录下一个插入元素的位置等。Position 属性，就是做这个的。当我们调用 put()方法时它指明了下一个元素的插入位置；当我们调用 get()方法时它指明了下一个读出元素的位置。聪明的读者会注意到，我的 Buffer API 并没有包含 get( ) or put( )方法。其实每一个 buffer 都有这俩方法，但这俩方法的参数类型和返回值类型只能在子类确定，所以这俩方法无法在顶层的 Buffer 类中声明。这里的讨论 我们先假设使用的是 ByteBuffer 类的这些方法(我们在 2.1.10 章节中讨论其他的 get()和 put())

```
public abstract class ByteBuffer
extends Buffer implements Comparable
{
    // This is a partial API listing
    public abstract byte get( );
    public abstract byte get (int index);
    public abstract ByteBuffer put (byte b);
    public abstract ByteBuffer put (int index, byte b);

}
```

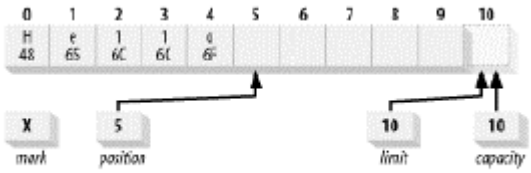
Gets 和 puts 可以是“相对的”也可以是“绝对的”。“相对的”版本就是在前面的 API 列表中，那些没有 index 参数的方法。当调用这些“相对的”方法时，position 就会前进 1 步。在相对的操作中如果 Position 超过了 limit，就会抛出异常。如 put()方法，会抛出 BufferOverflowException 异常，而 get()方法会抛出 BufferUnderflowException 异常。“绝对”访问不会对 Buffer 的 position 有影响，但如果 index 在允许的范围之外(负数或者是大于 limit)就会抛出 java.lang.IndexOutOfBoundsException 异常。

2.1.4 Filling

来个例子先。我们将“Hello”的 ASCII 字符序列转成 byte 值放入命名为 buffer 的 Buffer 对象中。下面的这些代码执行在来自 Figure 2-2 创建的那个新 buffer 上，Figure 2-3 显示了执行的结果即 buffer 的状态：

```
buffer.put((byte)'H').put((byte)'e').put((byte)'l').put((byte)'l').put((byte)'o');
```

Figure 2-3. Buffer after five put( )s



注意：本例中 的每一个 字符 都要先转化成 byte。我们无法不经转换地这样写：
buffer.put('H');

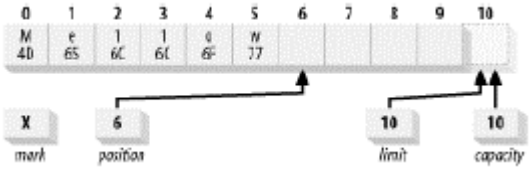
因为我们这里放入的是bytes 不是characters。记住，在Java中characters是以Unicode字符表示的，每个Unicode字符占16bits。在本节的例子中使用的 字节 就等于 ASCII 字符的 数值。通过强制转化char到byte，我们丢弃了高位的8个bits，从而产生了 8bits的byte值。这种转化方式在Latin characters中适用，但在其他的Unicode characters中就不一定了。为了让事情简单一些，这里我们有意地忽略了字符集映射问题。字符编码 会在 第六章详细讨论。

现在我们已经在buffer中放了一些数据，如果我们想变更这些数据而不变更Position该怎么做？“绝对”版本的put() 方法可以做到。假设我们想把Buffer中 的“Hello”变更为“Mellow”，代码应该这样写：

```
buffer.put(0, (byte)'M').put((byte)'w');
```

这里“直接”在buffer的位置0上用一个16进制的值0x4D[即'M'] 替换了 原来的 byte[即'H'的ASCII value]；并且在当前的position 上put了0x77['w']，使当前的position 增长了 1。[当前的position 并不会受到 “绝对的”put()影响]。结果就如图Figure 2-4：

Figure 2-4. Buffer after modification



2.1.5 Flipping 反转

我们填充[fill]完buffer后，现在就必须准备排出[draining]数据了。我们想把buffer传给一个Channel，去把buffer的内容“写出”去。但如果Channel调用的是Buffer.get()方法，那它就就会取到一些undefined data，因为当前位置[position]之后我们还没有写入任何东西。如果我们设置了position =0，通道就可以从正确的开始位置取数据了；但到哪里结束呢？---这就是limit [边界] 属性起作用的地方了。limit 就指明了buffer 可用内容的 结束位置。所以我们需要将limit设置为当前位置[position]，然后重置position到0。代码如下：

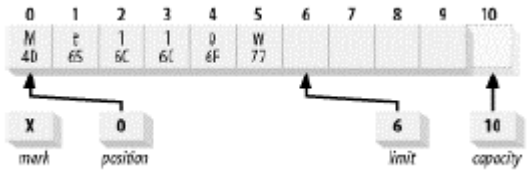
```
buffer.limit(buffer.position()).position(0);
```

Buffer在“填充”[fill]状态和“排出”[drain]状态之间的“反转”[flipping]，已经被API的设计者预见到了，他们为我们提供了一个简便的方法：

buffer.flip( );

flip( )方法将buffer从添加数据的“填充状态”[fill state]，反转[flip]成准备好“读出[read out]”数据的“排出状态”[drain state]。在反转之后图Figure 2-4的buffer，会变成Figure 2-5的样子：

Figure 2-5. Buffer after being flipped



rewind( )方法和flip( )相似，它也会设置position =0，但不会影响到limit。你可以使用rewind( )重新读取那些已经被 flip过一遍的 buffer 。

如果“反转”[flip]buffer两次会怎样呢？结果就是大小被搞成为零。在Figure 2-5的buffer上再搞一次flip：将limit设置为position ，将position设置为0；结果就使 limit和position都成了 0 。在这样的buffer上调用get会抛出BufferUnderflowException异常，调用put()会抛出BufferOverflowException异常。

2.1.6 Draining

如果我们将一个如图2-5的buffer传给Channel，Channel会排出我们之前放入的数据：从当前position 开始 一直到limit结束。

同样，你收到一个在别处被填充的buffer后，你也需要先flip再取数据。例如，如果Channel的read()操作已经完成，你需要看一下Channel读出了哪些数据，你就会在调用buffer.get()之前 先flip buffer。Channel会调用buffer 的put()方法填充数据；对buffer的puts和reads就可以交替进行。

布尔型返回值的hasRemaining()方法 能告诉你在draining时是否到达了buffer的limit。下面的代码将buffer的元素排出到数组中(在Section 2.1.10节，我们将会学到更有效率的批量[bulk]传输方式)。

```
for (int i = 0; buffer.hasRemaining( ), i++) {
      myByteArray [i] = buffer.get( );
      }
```

remaining( )方法会告诉你buffer中剩下的元素个数，即从position 到limit的元素个数：你可以使用如下的循环 排空 图 Figure 2-5中的buffer:

```
int count = buffer.remaining( );
for (int i = 0; i < count, i++) {
   myByteArray [i] = buffer.get( );
}
```

如果你的程序独自占有一个buffer，上面的代码要更有效率一些，因为不必在循环的每次迭代中检查 limit（hasRemain()，需要调用buffer的方法）。而在前一个例子中，就允许多线程并发的排出buffer的元素。

Buffer 并不是线程安全的。如果你想多线程的并发访问一个buffer，你就必须自己实现一个

同步策略。(例如，获取buffer对象的锁)。

Buffer在填充和排出 之后还可以再次 被重用[reused]。clear( )就可以重设buffer到清空状态。它不会改变buffer中的任何数据，而只是简单的将limit设置为capacity ，将position设置为0，如图2-2。这就让buffer重新准备好“填充”了。见 Example 2-1：

```
package com.ronsoft.books.nio.buffers;

import java.nio.CharBuffer;

/**
 * Buffer fill/drain example. This code uses the simplest means of filling and
 * draining a buffer: one element at a time.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class BufferFillDrain
{
    public static void main(String[] argv) throws Exception
    {
        CharBuffer buffer = CharBuffer.allocate(100);
        while (fillBuffer(buffer))
        {
            buffer.flip();
            drainBuffer(buffer);
            buffer.clear();
        }
    }

    private static void drainBuffer(CharBuffer buffer)
    {
        while (buffer.hasRemaining())
        {
            System.out.print(buffer.get());
        }
        System.out.println("");
    }

    private static boolean fillBuffer(CharBuffer buffer)
    {
        if (index >= strings.length) { return (false); }
        String string = strings[index++];
```

```
        for (int i = 0; i < string.length(); i++)
        {
            buffer.put(string.charAt(i));
        }
        return (true);
    }

    private static int          index    = 0;

    private static String[] strings =
                { "A random string value",
              "The product of an infinite number of monkeys",
              "Hey hey we're the Monkees",
              "Opening act for the Monkees: Jimi Hendrix",
              "'Scuse me while I kiss this fly", // Sorry Jimi ;-)
              "Help Me! Help Me!",     };
}
```

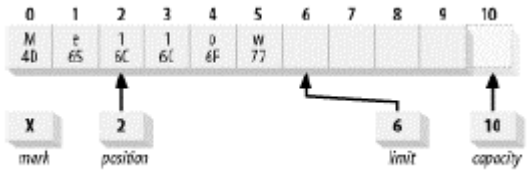
2.1.7 Compacting

压缩

```
public abstract class ByteBuffer
extends Buffer implements Comparable
{
    // This is a partial API listing
    public abstract ByteBuffer compact( );
}
```

有时候，我们会只排出一部分而不是全部，然后继续填充buffer。要做到这一点 就必须让那些还没有 被读到的向下移动， 第一个未读到的元素 需要移动到 index 0 上。反复的这样移动效率低下，这就需要API提供一种compact()方法，代你完成这样的移动。buffer的实现中能够高效的copy这些数据，这要比你使用get() 和put() 方法来调整高效得多。这种情况一定要使用 compact( )。Figure 2-6 显示了一个我们排出了一部分元素，并准备 被压缩 的buffer:

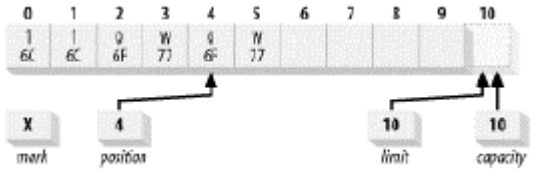
Figure 2-6. A partially drained buffer



Doing this:
buffer.compact( )

results in the buffer state shown in Figure 2-7.

Figure 2-7. Buffer after compaction



这里发生了几件事：2-5个元素被拷贝到第0-3的位置；虽然没有影响到位置4、5的元素，但它们已经在“当前位置[position]”之后，随后它们会被接下来的put( )进来的新元素覆盖掉，因此它们已经“死了[dead]”。还有，要注意的是：position的值是被拷贝的元素个数。就是说，position已经到了最后一个“可用的”元素后面。最后，limit也被设置成 capacity ，这样buffer就可以再次被 填满 了。调用compact( ) 的效果就是 **丢弃了那些已经被排出的元素，保留剩下的，并让buffer准备好被 重新填充**。

你可以以这种方式将buffer作为先进先出的队列[FIFO queue]使用。先进先出队列的高效算法当然还有(用buffer肯定不是)，但 压缩 也许是从socket流中读取数据的一个方便的方式。[译注：应用层的协议，常不知道包大小的。定义一个固定大小buffer，接收数据，通过不停的填充、排出、压缩，buffer就可以接收很大的socket数据]。

不管你随后是否填充了新元素，在压缩之后的排出，都需要反转一下buffer。

2.1.8 Marking

buffer的4个属性我们已经讲了三个了。第四个,mark 属性，允许buffer记住当前的position ，

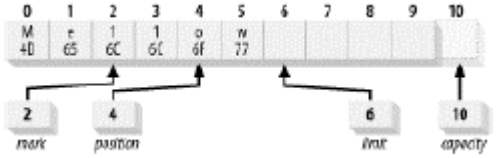
以便稍后再回来。mark属性在没调用buffer的mark()方法之前一直是undefined的，mark()方法就是将mark属性设置为当前的position。而reset()方法又将position设置为mark。如果mark还是undefinied状态调用reset的结果就会引起InvalidMarkException 异常。buffer中有一些方法还会 丢弃 mark属性，比如：rewind( ), clear( ), and flip( )。调用带index参数的 limit( ) 或 position( )方法时，如果index小于mark 的话，mark也会被丢弃。

注意，不要把 reset( ) and clear( )搞混了。clear( )方法清空buffer 而reset( )方法将position设置回之前设置的mark值。

Let's see how this works. Executing the following code on the buffer of Figure 2-5 results in the buffer state shown in Figure 2-8:

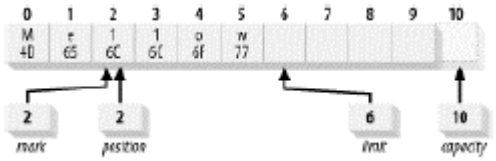
buffer.position(2).mark( ).position(4);

Figure 2-8. A buffer with a mark set



如果将这个buffer传给Channel后，发送了两个bytes ("ow")，position也到了6。如果我们调用reset()，position会被重新设置为mark(Figure 2-9)。将reset后的buffer中心传给Channel后它将会发送四个 bytes: ("llow")。

Figure 2-9. A buffer position reset to its mark



这个例子没有任何意义(Channel 写入“owllow”)，但还算不难理解。

2.1.9 Comparing

有时也需要比较两个buffer里面的数据。所有的buffer都提供了equals( ) 方法测试相等性；也提供了compareTo( )方法进行比较：

public abstract class ByteBuffer

extends Buffer implements Comparable

```
{
// This is a partial API listing
public boolean equals (Object ob)
public int compareTo (Object ob)
}
```

可以这样测试两个buffer的相等性：

```
if (buffer1.equals (buffer2)) {
doSomething( );
}
```

如果两个buffer 剩余的[remaining] 元素 是一样的，equals( ) 方法就会返回 true；否则返回false。因为“相等性”满足“交换律”，所以上面的代码中，交换两个buffer的位置结果相同 。

如果两个buffer被认为是相等的，只有：

- 两个对象都有相同的类型。如果Buffer包含的数据 类型不同，它们怎么都不会equal，Buffer也不会等于一个non-Buffer的对象。
- 两个buffer有相同数量的 “剩余元素”[remaining elements]。容量[capacity]和剩余元素的索引不必相同，但剩余元素的数量(from position to limit)必须相同。
- **依次** 被get()出来的剩余元素，必须相等。

以上任意一条不满足 equal都会返回false。

Figure 2-10 显示了两个buffer有不同的属性 但还是相等的；

Figure 2-11 显示了两个相似的buffer，它们甚至是同一个底层buffer的视图，但它们还是不相等的

Figure 2-10. Two buffers considered to be equal

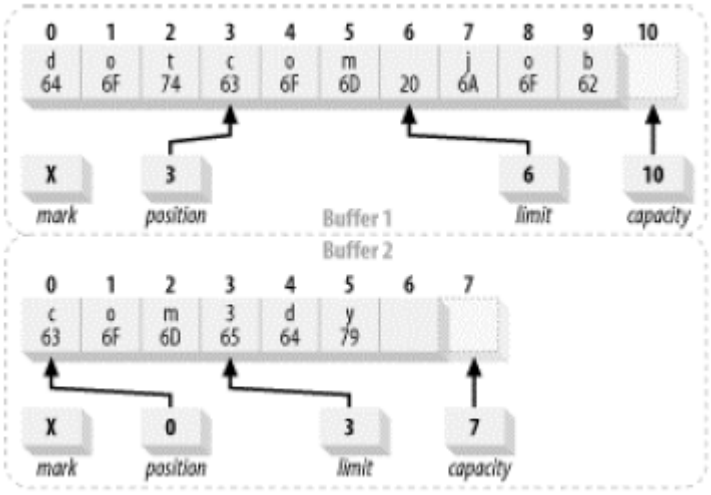
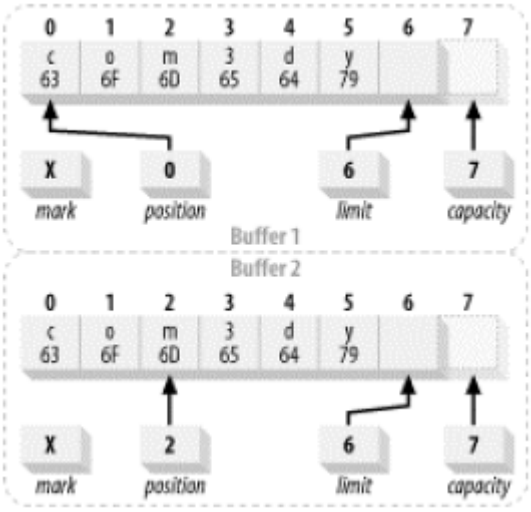


Figure 2-11. Two buffers considered to be unequal



buffer 还支持使用compareTo( )方法的 词典式比较。该方法返回负数、0、正数来分别反映小于、等于和大于。这正是java.lang.Comparable接口的语义，并且所有的buffer 都实现了该接口。这就意味着可以通过java.util.Arrays.sort( )按buffer的内容[比较]来给buffers数组排序了。

就像equals() 方法，compareTo( )也不能够比较两个不同类型的对象，但compareTo( )更严格一些：如果比较的是两个不同类型的对象，它会抛出 ClassCastException 异常，而equals不过返回false罢了。

在两个buffer上执行“compareTo( )”和equals( )一样 ，都是比较buffer的剩余元素。都会依次比较直到有元素不等或者任意一个buffer到达“限制”[limit]。如果其中一个buffer已经到达limit而前面的元素还是相等的，那么这个“短的”buffer 就被认为“小于” “长的”buffer。和equals( )不同compareTo( )不满足交换律。显而易见嘛。

```
if (buffer1.compareTo (buffer2) < 0) {
doSomething( );
}
```

上面的代码 作用在Figure 2-10上那 compareTo( )返回0。作用在Figure 2-11上 会返回一个正数(表示buffer2 大于buffer1)。

2.1.10 Bulk Moves

[批量操作]

buffer的设计初衷就是要更有效率的传输数据。但像Example 2-1那样循环移动单个元素，效率不是很高。下面就列出了，Buffer API 提供的一些 批量[bulk] 转移数据的方法：

```
public abstract class CharBuffer
extends Buffer implements CharSequence, Comparable
{
    // This is a partial API listing
    public CharBuffer get (char [] dst);
    public CharBuffer get (char [] dst, int offset, int length);
    public final CharBuffer put (char[] src);
    public CharBuffer put (char [] src, int offset, int length);
    public CharBuffer put (CharBuffer src);
    public final CharBuffer put (String src);
    public CharBuffer put (String src, int start, int end);
}
```

有两种形式的get()方法，将buffer的数据Copy到数组中。第一种只有一个数组参数，会将buffer的数据排出[drain]到指定的数组中；第二种带有的offset 和length参数能够指定排出到目标数组的 一个区域中。批量移动的和循环处理的结果一样，但更有效率。因为批量移动可以由本地代码实现或者进行其他的优化。

Bulk moves 通常都需要指定一个长度，就是说，通常都是操作固定数量的元素。这当然从方法签名上看不出来，但这种方式的get()：

```
buffer.get (myArray);
```

就等于

```
buffer.get (myArray, 0, myArray.length);
```

批量传输 通常都是固定长度的。忽略length 就等于填充了整个 数组。

如果指定传输的元素个数大于buffer剩余的元素个数，那一点数据都不会传输、buffer的状态

[译注：如position]也不会改变，并且还会抛出BufferUnderflowException 异常。所以，当你放入一个array却没有指定length的时候，就意味着length是整个数组的长度，如果剩下的元素不够，就抛出异常了。也就是当你将一个“小”buffer传输给“大”数组时，必须明确地指定length。将buffer排出到一个 大数组中 应该这么做：

```
char[] bigArray = new char[1000];
// Get count of chars remaining in the buffer
int length = buffer.remaining();
// Buffer is known to contain < 1,000 chars
buffer.get(bigArrray, 0, length);
// Do something useful with the data
processData(bigArray, length);
```

注意 在调用buffer的get()之前有必要查一下buffer的剩余元素个数。(调用processData( )也得传入放入bigArray的chars个数)。调用get()会让buffer的position前进。所以remaining( )后来会返回 0 。批量的get()方法也是返回buffer自己的引用而不是传输的元素数量。

另一方面，如果buffer的数据比array多，你就得反复地成块取出。示例代码：

```
char[] smallArray = new char[10];
while (buffer.hasRemaining())
{
    int length = Math.min(buffer.remaining(), smallArray.length);
    buffer.get(smallArray, 0, length);
    processData(smallArray, length);
}
```

批量的put()于此相似，但数据移动的方向相反：由array到buffer。对于传输数据的大小也有着相似的语义：

```
buffer.put (myArray);
```

就等于：

```
buffer.put (myArray, 0, myArray.length);
```

如果buffer有足够的空间接收array (buffer.remaining( ) >= myArray.length)，数据就会从buffer的当前position 给copy进来，position也会前进到加进来的元素末尾。如果buffer没有足够的空间，就不会发生数据传输并且还会抛出BufferOverflowException 异常。

还有使用put(Buffer)可能批量的将数据从一个buffer移动到另一个中：

```
dstBuffer.put (srcBuffer);
```

这就等于(假设dstBuffer 有足够的空间)：

```
while (srcBuffer.hasRemaining())
 {
      dstBuffer.put(srcBuffer.get());
 }
```

这俩buffer的position都会前进，并且值等于传输元素的个数。这里的边界检查和传递数组一样，特别是 如果 srcBuffer.remaining( )比 dstBuffer.remaining( )大，数据也不会传输，并且也会抛出 BufferOverflowException异常。 这里也许你会想：如果参数是他自己会怎样呢？这会奖励你一个又肥又大的java.lang.IllegalArgumentException异常。

本节中咱们已经使用CharBuffer做了好几个 examples 了，其实目前为止的讨论也适合其它类型的buffer 如FloatBuffer, LongBuffer等等。但下面列出的两个bulk方法 却只是CharBuffer才有：

```
public abstract class CharBuffer
extends Buffer implements CharSequence, Comparable
{
    // This is a partial API listing
    public final CharBuffer put (String src);
    public CharBuffer put (String src, int start, int end);
}
```

这些用String做参数的方法和那些char arrays的bulk move方法类似。Java程序员都知道，Strings和字符串数组不一样。但是Strings包含字符序列，我们又曾试图把字符数组和String在概念上统一(尤其是那些C/C++程序员)。为此，CharBuffer类提供了拷贝String到CharBuffer的简便方法。

传输String 和传输char array类似，不同点在于传输String的范围是通过start和end+1索引来确定的(和String.subString()类似)，而不是start索引和length，所以：

```
buffer.put (myString);
```

就等于：

```
buffer.put (myString, 0, myString.length( ));
```

如何copy myString中的第5-8，共4个字符 ，到 buffer:

```
buffer.put (myString, 5, 9);
```

String的bulk move等价于：

```
for (int i = start; i < end; i++)
 {
     buffer.put (myString.charAt (i));
}
```

String 的范围检查 和char array一样。如果字符数大于buffer.remaining()，将会抛出BufferOverflowException异常。

2.2 Creating Buffers

就像我们在 Figure 2-1 看到的那样，有七种基本类型的 Buffer，每种 buffer 都对应着一种 Java 语言中的非布尔基本类型。(第 8 种 Buffer 是 MappedByteBuffer 它是一种特殊的 ByteBuffer 用在 内存映射文件上。我们会在 第三章 讨论。) 这些 Buffer 都不可以直接实例化，它们都是 抽象类 ，但每个类都有一个 可以创建对应实例的 静态工厂方法 。

对于这里的讨论，我们拿 CharBuffer 作例子，其实拿其他的六个：IntBuffer, DoubleBuffer, ShortBuffer, LongBuffer, FloatBuffer, 以及 ByteBuffer 也一样。这种创建 buffer 的方法 适用于 所有的 Buffer 类。

```
public abstract class CharBuffer
        extends Buffer implements CharSequence, Comparable
        {
            // This is a partial API listing
            public static CharBuffer allocate (int capacity)
            public static CharBuffer wrap (char [] array)
            public static CharBuffer wrap (char [] array, int offset,
            int length)
            public final boolean hasArray( )
            public final char []array( )
            public final int arrayOffset( )
```

}

新的 Buffer 可以通过 分配 [allocation] 或 包装[wrapping]创建。通过 分配 创建的 Buffer 对象，会有一个 private space 来持有 分配的容量数据。包装 创建的 Buffer 则没有分配任何空间持有数据。它使用你提供的 数组 作为存储器持有 Buffer 的元素。

分配一个持有 100 个 char 的 CharBuffer：

CharBuffer charBuffer = CharBuffer.allocate (100);

如果你想使用自己定义的数组作为存储，可以调用 wrap( )方法：

char [] myArray = new char [100];

CharBuffer charbuffer = CharBuffer.wrap (myArray);

这里虽然创建了一个新的 Buffer 对象，而 Buffer 的元素却还在 数组中。也就是通过调用 buffer 的 put()方法对 buffer 的更新会影响到 数组 myArray，同样 对数组的更新对 buffer 也是可见的。使用带 offset 和 length 参数的 wrap()会创建一个设置了 position 和 limit 的 buffer。position 和 limit 与 offset 和 length 有关，这样做：

CharBuffer charbuffer = CharBuffer.wrap (myArray, 12, 42);

会创建一个属性如下：

position=12;limit=54,capacity=myArray.length.

的 Buffer。

这个方法不会向你期望的那样，用部分 array 的创建 buffer。实际上 buffer 可以访问整个 array；而参数 offset 和 length 只是用来设置 buffer 的初始状态。调用 clear()方法后 一直填充到 limit 就能够覆写数组的所有元素。slice()方法(会在 Section 2.3 讨论)会产生一个占用部分 array 的 buffer。

通过allocate()和wrap()创建的buffer都不是 **直接缓冲区**[direct buffer] (直接缓冲区在 2.4.2 节讨论)。就像我们之前讨论的 **非直接缓冲区**[Nondirect buffer] 拥有底层实现数组， 并且你可以通过剩下的那几个 API（hasArray( )\ array( )\ arrayOffset( )）访问它。布尔方法 hasArray( )告诉 Buffer 是否有底层实现数组。array()方法返回一个该 Buffer 使用的数组引用。

如果 hasArray()返回 false，就不要调用 array()或者 arrayOffset()，否则你会收到一个 UnsupportedOperationException 异常。如果一个缓冲区是只读的，它的底层实现数组就是一个禁区。即使是通过 wrap()方法传入的，调用 array()或者 arrayOffset()也会抛出一个 ReadOnlyBufferException 异常，以防止获得访问和修改 read-only buffer 的数据。但如果你能通过其他途径访问到底层实现数组，那对数组的修改也能在 read-only buffer 反映出来。read-only buffer 将在 2.3 详细讨论。

这最后一个方法，arrayOffset()，返回在缓冲区数据元素存储区在数组上的偏移量。如果你用三参数版本的 wrap()，arrayOffset()始终返回 0。当然，如果你 slice 一个做为 buffer 的底层实现数组，就可能产生一个非 0 的 Offset。数组偏移[array offset]和缓冲区的容量能够告诉你 Buffer 究竟使用了 array 的那些元素。buffer 的 slice 将在 Section 2.3 讨论。

目前为止，本节的讨论内容对其他类型的 buffer 也适用。CharBuffer 就是我们一直作为例子讲的 Buffer，还提供了一对别的 buffer 没有但非常有用的方法：

```
public abstract class CharBuffer extends Buffer implements CharSequence,
Comparable
    {
        // This is a partial API listing
        public static CharBuffer wrap (CharSequence csq)
        public static CharBuffer wrap (CharSequence csq, int start, int end)

    }
```

这俩版本的 wrap()创建的 buffer 以字符序列对象[CharSequence](或者是子序列)作为底层实现(字符序列对象在第五章详细描述)。字符序列对象描述了一个可读字符序列。在 JDK1.4，三种类实现字符序列：String,StringBuffer,和 CharBuffer。这个版本的 wrap()把给定的字符数据"buffer 化"，也就可以通过缓冲区 API 来访问字符。这对字符集编码(第六章)和正则表达式处理(第五章)是很有用的。

```
CharBuffer charBuffer = CharBuffer.wrap ("Hello World");
```

这三参数的 wrap()用 start 和 end 索引位置，来描述给定字符序列的子序列。这也可以通过 CharSequence.subsequence( )实现，但这样更简便。start 参数是序列中第一个要用的字符；end 是最后一个字符的 index 加 1。

2.3 Duplicating Buffers

就像我们之前讨论的，buffer 对象可以由外部数组创建。但 buffer 不仅可以管理外部数组的数据，它还能管理外部 buffers 的数据。能够管理其他 buffer 数据的 buffer 称作 view buffer。大部分的 view buffer 都是 ByteBuffers 的(见 Section 2.4.3)view。在讨论 byte buffer 之前，我们先集中精力学习对所有 buffer types 都适用的 view buffer。

总是需要调用已有 buffer 的实例方法来创建 View buffer。在已有的 buffer 上调用其工厂方法创建 view buffer，意味着这个 view buffer 能够了解 原 buffer 的实现细节。它能够直接访问原 buffer 的元素，不管它们基于数组还是基于其他方式实现，和通过原 buffer 的 get( )/put( ) API 方法访问一样。如果 原 buffer 是 直接分配的 buffer[direct buffer]，该 buffer 的 views 也会有相同的效率。这对于 mapped buffer 也是一样的(第三章 会讲 mapped buffer)。

本节中，我们继续使用 CharBuffer。本节所讲的 这些 buffer operations 同样也适用于 其他的 主类型 buffer (见图 2-1)。

```
public abstract class CharBuffer extends Buffer
        implements CharSequence, Comparable {
    // This is a partial API listing
    public abstract CharBuffer duplicate( );
    public abstract CharBuffer asReadOnlyBuffer( );
    public abstract CharBuffer slice( );
}
```

duplicate( )方法会创建一个和原 buffer 一样的缓冲区。这俩 buffer 共享一样的数据元素并有相同的容量。但各自拥有自己的 position，limit，和 mark。在其中一个 buffer 修改的数据，会在另一个中体现出来。duplicate buffer 和原 buffer 对 数据 拥有一致的视图。

如果原 buffer 是 read-only 或 direct 的，那新的 buffer 也会是同样的。直接缓冲区将在 2.4.2 讨论。

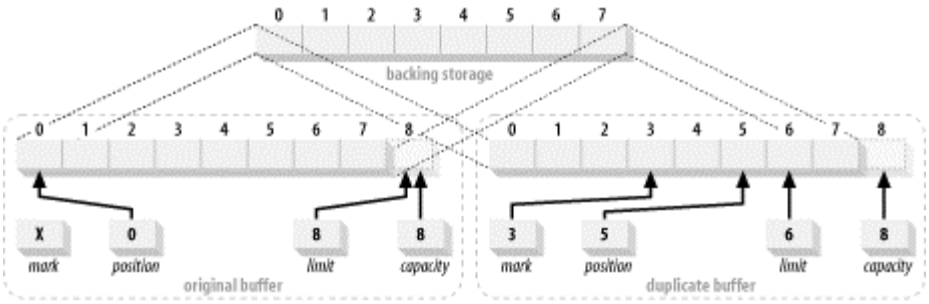
复制缓冲区创建一个新的缓冲区对象但是没有拷贝数据。original buffer 和 它的 copy 都操作同一份数据元素。

以下代码产生的 buffer 和它的 duplicate，以及它们之间的关系见 Figure 2-12。

```
CharBuffer buffer = CharBuffer.allocate (8);
buffer.position (3).limit (6).mark( ).position (5);
CharBuffer dupeBuffer = buffer.duplicate( );

buffer.clear( );
```

Figure 2-12. Duplicating a buffer



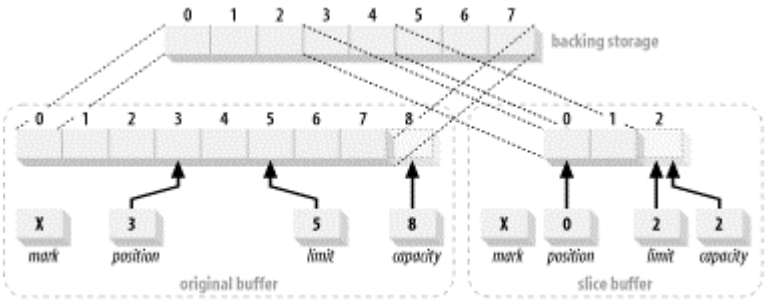
你可以通过 asReadOnlyBuffer( )方法创建一个 buffer 的只读视图[read-only view]。这和 duplicate( )一样,除了新的缓冲区不许做 put()，而且它的 isReadOnly( )方法返回 true。试图在只读缓冲区上调用 put()会抛出一个 ReadOnlyBufferException 异常。

如果 read-only buffer 和 writable buffer 共享一份数据，或者基于 包装的数组。那么通过 writable buffer 或 直接对数组 作出的修改 都会体现在相关 buffer 上，包括 read-only buffer。

分割[Slicing]buffer 和 duplicating 相似，但 slice( )是从原 buffer 的 current position 开始的，并且它的 capacity 是原 buffer 的剩余元素个数((limit - position)。新的分割 buffer [slice buffer]共享原 buffer 元素的子集。 分割 buffer 也会继承原 buffer 的 read-only 和 direct 性质。Figure 2-13 说明了如下代码创建的一个 slice buffer。

```
CharBuffer buffer = CharBuffer.allocate (8);
buffer.position (3).limit (5);
CharBuffer sliceBuffer = buffer.slice( );
```

Figure 2-13. Creating a slice buffer



创建一个映射了数组中从 12-20(9 个元素)的 buffer。其代码如下：

```
char [] myBuffer = new char [100];
CharBuffer cb = CharBuffer.wrap (myBuffer);
cb.position(12).limit(21);
CharBuffer sliced = cb.slice( );
```

更加复杂的 view buffer 会在 Section 2.4.3 节讨论。

2.4 Byte Buffers

我们将在本节详细讨论 ByteBuffer。除了 boolean 类型外 其他基本类型都有对应的 Buffer 类，但 ByteBuffer 有着和其他 buffer 不同的特性。Bytes 是操作系统和 I/O 设备使用的基本数据单元。在 JVM 和操作系统间 传输数据时 其他的数据类型必须 分解成字节。就如我们接下来所看到的：我们从 buffers 的设计和与操作系统级别的 I/O 的交互上就能感受到它们之间的通信是面向字节的。

作为参考，我们列出了全部的 ByteBuffer API。有一些方法我们之前就讨论了，其他一些新的方法接下来就会涉及。

```
package java.nio;
public abstract class ByteBuffer extends Buffer
implements Comparable
{
```

```
public static ByteBuffer allocate (int capacity);
public static ByteBuffer allocateDirect (int capacity);
public abstract boolean isDirect( );
public static ByteBuffer wrap (byte[] array, int offset,
int length)
public static ByteBuffer wrap (byte[] array);
public abstract ByteBuffer duplicate( );
public abstract ByteBuffer asReadOnlyBuffer( );
public abstract ByteBuffer slice( );
public final boolean hasArray( );
public final byte [] array( );
public final int arrayOffset( );
public abstract byte get( );
public abstract byte get (int index);
public ByteBuffer get (byte[] dst, int offset, int length);
public ByteBuffer get (byte[] dst, int offset, int length);
public abstract ByteBuffer put (byte b);
public abstract ByteBuffer put (int index, byte b);
public ByteBuffer put (ByteBuffer src);
public ByteBuffer put (byte[] src, int offset, int length);
public final ByteBuffer put (byte[] src);
public final ByteOrder order( );
public final ByteBuffer order (ByteOrder bo);
public abstract CharBuffer asCharBuffer( );
public abstract ShortBuffer asShortBuffer( );
public abstract IntBuffer asIntBuffer( );
public abstract LongBuffer asLongBuffer( );
public abstract FloatBuffer asFloatBuffer( );
public abstract DoubleBuffer asDoubleBuffer( );
public abstract char getChar( );
public abstract char getChar (int index);
public abstract ByteBuffer putChar (char value);
public abstract ByteBuffer putChar (int index, char value);
public abstract short getShort( );
public abstract short getShort (int index);
public abstract ByteBuffer putShort (short value);
public abstract ByteBuffer putShort (int index, short value);
public abstract int getInt( );
public abstract int getInt (int index);
public abstract ByteBuffer putInt (int value);
public abstract ByteBuffer putInt (int index, int value);
public abstract long getLong( );
public abstract long getLong (int index);
public abstract ByteBuffer putLong (long value);
```

```
    public abstract ByteBuffer putLong (int index, long value);
    public abstract float getFloat( );
    public abstract float getFloat (int index);
    public abstract ByteBuffer putFloat (float value);
    public abstract ByteBuffer putFloat (int index, float value);
    public abstract double getDouble( );
    public abstract double getDouble (int index);
    public abstract ByteBuffer putDouble (double value);
    public abstract ByteBuffer putDouble (int index, double value);
    public abstract ByteBuffer compact( );
    public boolean equals (Object ob) ;
    public int compareTo (Object ob) ;
    public String toString( );
    public int hashCode( );

}
```

Bytes Are Always Eight Bits, Right?

当下几乎公认 byte 有 8bits。但过去不是这样子的，那时候 byte 的范围在 2-12bits 之间[甚至更长]，大部分都是 6-9bits。8bit 的字节是在实用性和市场的双重力量下产生的。它的实用性在于 8bit 正好能够代表一个可用字符[译注：ASCII 码](英文字符)。8 也是 2 的指数(这让硬件的设计足够简单)。8bits 刚好能够持有 2 个 16 进制数，通过组合它也能存储其他类型的数值。而来自市场的力量就是 IBM 了，IBM 在 60s 年代推出的 360 主机首次引入 8bit 的 byte。它很好的解决了那些问题。想了解更多的背景，就去请教那个家伙把：IBM 的鲍勃.贝莫[译注：这位大师 已经挂了；360 人月神话] http://www.bobbemer.com/BYTE.HTM。

2.4.1 Byte Ordering

除了 boolean 外其他主要的数据类型都是由 byte 组合的 **2**。它们的大小如 Table 2-1 所示：

Table 2-1. Primitive data types and sizes

Data type	Size(in bytes)
Byte	1
Char	2

Short	2
Int	4
Long	8
Float	4
Double	8

2 Booleans represent one of two values: true or false. A byte can take on 256 unique values, so a boolean cannot be unambiguously mapped to one or several bytes. Bytes are the building blocks from which all buffers are constructed. The NIO architects determined that implementation of boolean buffers would be problematic, and the need for such a buffer type was debatable anyway.
Booleans 的值是true/false的二者之一。而一个byte可以表现256个值，所以一个boolean无法明确的用一个或多个byte映射。字节是构造buffer的积木。NIO的架构者认为实现boolean buffer不可取，并且这种boolean buffer的需求本身都是有问题的。

每个基本类型数据在内存中都是一块连续的字节序列。如 32 位的整数 0x037FB4C7 (即 10 进制 58,700,999)，在内存中也许是这样分布的，图 Figure 2-14(从左到右的内存地址)。请注意是“也许”，尽管字节的大小是统一了，但字节的顺序[byte order]还没有形成一致。一个整数的表示在内存中还可以组织成 Figure 2-15 那样：

Figure 2-14. Big-endian byte order

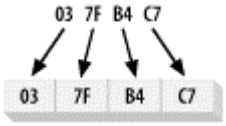
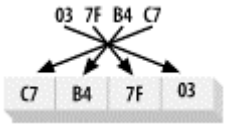


Figure 2-15. Little-endian byte order



多字节数值在内存中的存放方式通常称作：字节次序。如果数值上的“大数”在低位地址上，这样的系统为 big-endian(Figure 2-14)；如果“小数”在低位地址上那这样的系统就是 little-endian(Figure 2-15)。

字节顺序很少是软件设计者选择的结果；它通常是硬件设计者则规定的。这两种字节顺序有时还会被称作“字节性别”，这两种字节顺序今天都是被广泛应用的。它们各自都有好的来

由。Intel 处理器是 little-endian 设计，Motorola CPU、Sun Sparc, and PowerPC CPU 都是 big-endian 的。

字节顺序问题已经超越了 CPU 的设计。Internet 架构师在设计用于连接不同类型的计算机协议，IP 协议 时就遇到了不同系统间 内在的 字节顺序问题。因此，IP 协议就定义了一个“网络字节顺序”的概念，也就是统一使用 big-endian。IP 协议包内部的所有的多字节数值必须在 本地主机的字节顺序和通用的 网络直接顺序间 相互转换。

译注：big-endian/little-endian。 《格利佛游记》中有个王国。该国内部主要矛盾就是 鸡蛋应该 先从大头敲，还是先从小头敲。他们认为这个问题意义重大，两边的支持者为此打了多次内战。今天看来，这个讽刺还是那么新鲜☺。

在 java.nio 中，字节顺序被包装在 ByteOrder 类中：

```
package java.nio;
public final class ByteOrder
{
    public static final ByteOrder BIG_ENDIAN;
    public static final ByteOrder LITTLE_ENDIAN;
    public static ByteOrder nativeOrder();
    public String toString();

}
```

ByteOrder 类定义了两个用来从 buffer 中存储或接收多字节数值时 确定使用哪种字节顺序的 静态常量。ByteOrder 要作为 类新安全的枚举类型使用。

ByteOrder 类定义了两个预先初始化的自己的 instances 放在 public fields 中。这样，在一个 JVM 中就只能有这两个 instances[译注：ByteOrder 的构造器 是 private 的]，所以它们可以使用==操作符。如果你想获得 JVM 运行的平台上的字节顺序，调用 nativeOrder 静态类方法。它将返回这两个值中的一个。调用 toString()方法可以返回“BIG_ENDIAN”或者“LITTLE_ENDIAN”的字符串。

每个缓冲区类都有一个当前字节顺序的设置，可用 order()方法查看：

```
public abstract class CharBuffer extends Buffer
    implements Comparable, CharSequence
{
    // This is a partial API listing
    public final ByteOrder order();

}
```

这个方法返回 ByteOrder 常数中的一个。除了 ByteBuffer，其他 buffer 的 byte order 都是 只读 属性，它们只依赖于 buffer 的创建方式。除了 ByteBuffer，其他以 allocation 或 wrapping 一个数组创建的 buffer，order()所返回值都和 ByteOrder.nativeOrder( )一致。这是因为在 JVM 内部，buffer 的元素是作为基本数据类型的数据 被访问的。

ByteBuffer 是不一样的，它默认的 byte order 总是 ByteOrder.BIG_ENDIAN，且与 本地系统地 byte order 无关。Java 默认的 byte order 就是 big-endian 的，这就类文件和序列化对象能够运行在任意的 JVM 上。不过如果本地系统 是 little-endian 的 ，那多少会有点 性能问题，毕竟还要转化嘛。使用本地的 byte order 访问 ByteBuffer 的其他数据类型，可以更有效率一些。

为什么 ByteBuffer 还允许设置 byte order？字节就是字节，不是吗？当然是，但你会很快发现 Section 2.4.4 中还有许多 get/put 其他基本数据类型的方法。这些方法方法对 bytes 的 encode 和 decode 依赖于 ByteBuffer 的 byte order 设置。

ByteBuffer 的 byte-order 可以在任意时刻通过调用 order (ByteOrder bo)设置，参数是 ByteOrder.BIG_ENDIAN 或 ByteOrder.LITTLE_ENDIAN。

```
public abstract class ByteBuffer extends Buffer
implements Comparable
{
    // This is a partial API listing
    public final ByteOrder order( );
    public final ByteBuffer order (ByteOrder bo);

}
```

如果一个 buffer 是作为一个 ByteBuffer 对象的视图创建的(见 Section 2.4.3)，那么 order()方法返回的值是原 ByteBuffer 的在视图缓冲区创建时候的 byte-order 设置。这个

视图的 byte-order 在创建后不会被修改，并且如果原 ByteBuffer 的字节顺序改变了，也不会影响到该 view buffer。

2.4.2 Direct Buffers

ByteBuffer 与其他 buffer 最重要的区别在于，它可以作为 Channel I/O 的 sources 或 targets。如果你跳到 Chapter 3，你就会发现 Channel 只接受 ByteBuffer 类型的参数。

就像第一章中所描述的那样，操作系统是在内存上作 I/O。操作系统所使用的这些内存也就是一些连续的字节序列。所以，只有 ByteBuffer 可以参与实际的 I/O 操作，也就没什么好奇怪的了。还有，操作系统能够直接访问进程（这里指 JVM 进程）的地址空间，来传输数据。也就是，I/O 操作的 目标内存区域 也需要是连续的字节序列。但在 JVM 内部，Byte 数组在内存中却可以不是连续的，或者 Garbage Collector 在任意时刻移动了它们。数组，是 JAVA 对象，它的存储方式在不同的 JVM 上差异很大。

所以引入了 direct buffer 的概念。直接缓冲区试图让 Channel 直接和本地 I/O 程序交互。将 Channel 使用的字节元素存储在内存中，让本地代码可以直接访问即操作系统可以直接排出或填充 这块内存区域。

Direct byte buffer 常常是进行 I/O 操作的最佳选择。按照设计，他们给 JVM 提供了最有效率的 I/O 机制。Nondirect byte buffer 也可以传给 channels，但这样做降低一些性能。本地 I/O 通常无法操作 nondirect buffer。如果在 Channel 写入时传递了一个 nondirect ByteBuffer 对象，那 Channel 会在每次调用时隐式做了以下操作：

1. 创建一个临时 direct ByteBuffer 对象。
2. 将非 nondirect buffer 的内容 copy 到临时缓冲区
3. 用临时缓冲区执行底层的 I/O 操作。
4. 临时缓冲区超出作用域而最终被垃圾收集。

会导致在每次 I/O 操作都会引起多余的 buffer copy 和产生多余的对象，而这些都是我们应该避免的事情。当然，这也依赖于实现，事情也许没有那么糟糕，运行时 或许会 cache 并

重用 direct buffer，或者执行其他一些优化来提高 吞吐量。如果创建只使用一次 buffer，那么这些区别不会太明显。但如果是在一个高性能的环境中反复使用这些 buffer，那你最好是分配 direct buffers 并且重用他们。

Direct buffer 是用来优化 I/O 的，但创建它们通常要比创建 nondirect buffers 成本高。分配直接缓冲区，调用了本地操作系统的特定的代码分配了内存，且绕过了 JVM heap。直接缓冲区 的装载和卸载通常都要比 常驻 heap 的 buffer 成本高，这当然也依赖于操作系统和 JVM 的实现。直接缓冲区的内存区域是不会被垃圾收集的，因为它们在 JVM heap 外面。

在使用直接缓冲区还是使用 nondirect buffer 之间的性能权衡可能受到不同的 JVM、操作系统和代码设计而不同。分配 heap 外的内存，可能会让你的应用程序受制于 JVM 之外的控制。当你将这些额外的部件带进来时，要确保它们能够达到预期的效果。我还是要推荐古老的软件格言：要先让他们跑起来，再想如何跑得快。不要太执着于 预先优化，先关注软件的正确性。jvm 实现时 或许已经作了 buffer caching 或其他一些优化了；这些也许就能够满足性能需要而不需要多余努力。(不要去计较效率上一些小的优化，在 97%的情况下，过早的优化是一切问题的根源— Donald Knuth)。

Direct ByteBuffer 可以通过调用 ByteBuffer.allocateDirect(int capacity)创建，就像我们前面讨论的 allocate( )方法一样，需要传入 capacity。注意：那些由 wrap( )方法创建的 wrapped buffers，总是 non-direct 的。

所有缓冲区都提供一个 isDirect( )的 boolean 方法，来测试是否是直接缓冲区。只有 ByteBuffer 才能是 directBuffe。也只有它们基于的缓冲区是一个直接字节缓冲区，对应的非字节视图缓冲区的 isDirect( ) 才可能是 true。这带我们到 ---- view buffer。

2.4.3 View Buffers

就像之前讨论的，I/O 基本上就是来回搬运成堆的字节。当使用 ByteBuffer 做高吞吐的 I/O，来源是从文件中读取、从网络连接中接收等等。但当你通过 ByteBuffer 接收到这些数据后，你就必须考虑如何处理/维护它们了。幸好 ByteBuffer 提供了不少创建 view buffer 的 API 。

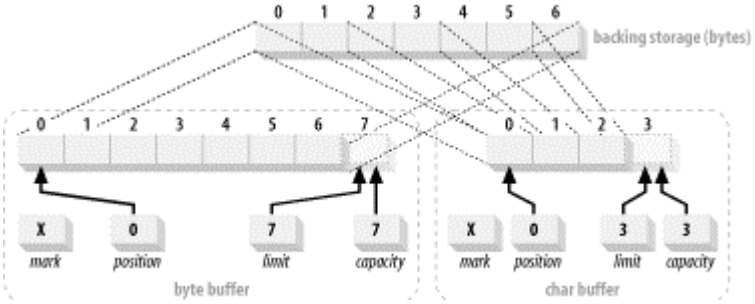
视图缓冲区是通过在已有的 Buffer 上调用其工厂方法创建的。该视图对象维护着自己的一套 capacity， position， limit，和 mark 属性，但和原 buffer 共享着相同的数据元素。我们已经在 Section 2.3 节简单的看了 buffer 的 duplicated 和 sliced。而 ByteBuffer 却能够创建将 bytes 映射成 其他主要类型的 buffer 视图。例如：asLongBuffer( )就创建了一个能够以 long 类型[8byte -long]的访问 ByteBuffer 的 view buffer。

下面列出的每个工厂方法都护创建一个原*ByteBuffer*的view Buffer。每个view Buffer都对应一种数据类型，且从当前position 到limit之间分割[slice] 了底层的*ByteBuffer*。view Buffer的capacity 等于原buffer剩余[remaining]的元素个数 除以 基本类型的字节数(见 Table 2-1)。分割之后(除不尽的 余数)，尾部剩余的字节[多余的字节]，在新的view Buffer中是不可见的。view Buffer的第一个元素 是从原buffer的当前位置开始的。对映射View Buffer数据元素 的byte分组，是可以在ByteBuffer的实现中优化的。

```
public abstract class ByteBuffer
extends Buffer implements Comparable
{
    // This is a partial API listing
    public abstract CharBuffer asCharBuffer( );
    public abstract ShortBuffer asShortBuffer( );
    public abstract IntBuffer asIntBuffer( );
    public abstract LongBuffer asLongBuffer( );
    public abstract FloatBuffer asFloatBuffer( );
    public abstract DoubleBuffer asDoubleBuffer( );
}
```

下面的代码创建了一个*ByteBuffer*的 *CharBuffer* view。如图所示(Example 2-2也使用了这个片段)。

Figure 2-16. A CharBuffer view of a ByteBuffer



Example 2-2. Creating a char view of a ByteBuffer

```
package com.ronsoft.books.nio.buffers;

import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.ByteOrder;

/**
 * Test asCharBuffer view.
 *
 * Created May 2002
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class BufferCharView
{
    public static void main(String[] argv) throws Exception
    {
        ByteBuffer byteBuffer = ByteBuffer.allocate(7).order(
                ByteOrder.BIG_ENDIAN);
        CharBuffer charBuffer = byteBuffer.asCharBuffer();
        // Load the ByteBuffer with some bytes
        byteBuffer.put(0, (byte) 0);
        byteBuffer.put(1, (byte) 'H');
        byteBuffer.put(2, (byte) 0);
        byteBuffer.put(3, (byte) 'i');
        byteBuffer.put(4, (byte) 0);
        byteBuffer.put(5, (byte) '!');
        byteBuffer.put(6, (byte) 0);
        println(byteBuffer);
        println(charBuffer);
    }

    // Print info about a buffer
    private static void println(Buffer buffer)
    {
        System.out.println("pos=" + buffer.position() + ", limit="
                + buffer.limit() + ", capacity=" + buffer.capacity() + ": '"
                + buffer.toString() + "'");
    }
}
```

```
pos=0, limit=7, capacity=7: 'java.nio.HeapByteBuffer[pos=0 lim=7 cap=7]'
```

pos=0, limit=3, capacity=3: 'Hi!'
Here's the output from executing BufferCharView:
pos=0, limit=7, capacity=7: 'java.nio.HeapByteBuffer[pos=0 lim=7 cap=7]'
pos=0, limit=3, capacity=3: 'Hi!
Once you've obtained the view buffer, you can create further subviews with *duplicate( )*, *slice( )*, and *asReadOnlyBuffer( )*, as discussed in Section 2.3.

当view buffer访问原*ByteBuffer*的字节元素时，这些 字节 便根据view buffer的byte-order设置打包成一个 基本类型的数据元素。创建view buffer 时，它会继承原ByteBuffer的byte-order。而view buffer的byte-order，之后也不可变更。在Figure 2-16中，你可以看到*ByteBuffer*中两个字节 映射为 view *CharBuffer*的一个 字符。而byte-order就决定了这俩字节组成 字符的前后顺序。Section 2.4.1节有更多的讨论。

当基于direct byte buffer时使用View buffer可能会更有效率。当View buffer的byte order与底层平台/硬件的byte order一致时，访问数据就省掉了 字节到基本类型的 打包 和 解包。

2.4.4 Data Element Views

ByteBuffer 类提供一种轻量级的机制，来访问作为基本类型的 bytes 组。ByteBuffer 包含了每种基本类型的访问器和修改器。

```
public abstract class ByteBuffer
extends Buffer implements Comparable
{
    public abstract char getChar( );
    public abstract char getChar (int index);
    public abstract short getShort( );
    public abstract short getShort (int index);
    public abstract int getInt( );
    public abstract int getInt (int index);
    public abstract long getLong( );
    public abstract long getLong (int index);
    public abstract float getFloat( );
    public abstract float getFloat (int index);
    public abstract double getDouble( );
    public abstract double getDouble (int index);
    public abstract ByteBuffer putChar (char value);
    public abstract ByteBuffer putChar (int index, char value);
    public abstract ByteBuffer putShort (short value);
```

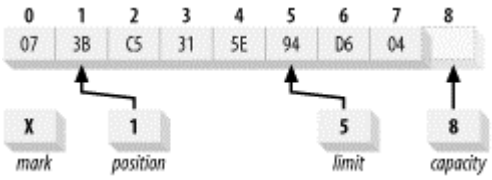
```
    public abstract ByteBuffer putShort (int index, short value);
    public abstract ByteBuffer putInt (int value);
    public abstract ByteBuffer putInt (int index, int value);
    public abstract ByteBuffer putLong (long value);
    public abstract ByteBuffer putLong (int index, long value);
    public abstract ByteBuffer putFloat (float value);
    public abstract ByteBuffer putFloat (int index, float value);
    public abstract ByteBuffer putDouble (double value);
    public abstract ByteBuffer putDouble (int index, double value);

}
```

这些方法能够像访问基本类的buffer一样从当前位置访问ByteBuffer的字节组。这些字节组会被根据byte-order编成需要的 基本数据类型 输入输出。例如调用getInt( )，从前位置以后的4个bytes会被打包成一个int，并返回这个int值。见Section 2.4.1节。

假设有一个命名为 buffer 的字节缓冲区，其状态如 Figure 2-17.

Figure 2-17. A ByteBuffer containing some data



这行代码:

```
int value = buffer.getInt( );
```

将返回一个由 buffer 中 1-4 索引的字节的值组成的整数值。这个实际的整数值还依赖于当前缓冲区的字节顺序。更准确地说：

```
int value = buffer.order (ByteOrder.BIG_ENDIAN).getInt( );
```

返回数值 0x3BC5315E, 而:

```
int value = buffer.order (ByteOrder.LITTLE_ENDIAN).getInt( );
```

返回值是 0x5E31C53B。

如果你试图获取的基本数据类型需要的字节比缓冲区中剩余的字节多，那将抛出一个 BufferUnderflowException 异常。对图 2-17 中的缓冲区来说下面代码将抛出异常，因为一个 long 是 8 字节长，而缓冲区中只有 5 个字节：

long value = buffer.getLong( );

[译注：word 字，JVM 内部的 word 是 32bits，一个 int，无论在 32/64 的系统上。]

[译注：内存字对齐，内存中或 cpu 运算时的一个单元通常称一个 word，如 32 位的系统上一个字常是一个 int。cpu 运算时会把 boolean/byte/char 等作为一个字，而 long[64]作为两个字[32]来运算。]

被这些方法返回的元素不需要任何取模进行对齐(取模就是一个数被另一个除的时候的余数。模界就是一个数字系列上的一些点，这些地方余数是 0.例如，任何数字被 4 除尽的是模 4：4，8，12，16 等。很多 CPU 设计需要多字节数值在内存模对齐)。数据值被从缓冲区当前位置获取并装配，无需 字 [word]对齐。这样效率可能差一些，但它允许在一个字节流中的数据随意放置。这对于从二进制文件数据中抽取数值，或者可以把数据打包进平台特定的格式输出到外部系统 应该非常有用。

put 方法和 get 执行相反的操作。基本数据值将根据缓冲区的字节顺序被打碎成字节并存储。如果没有足够空间存储，将抛出一个 BufferOverflowException 异常。

每个方法都有相对和绝对的形式。相对的形式增加 position 受影响的字节数。(见 Table 2-1).绝对的版本不改变 position。

2.4.5 Accessing Unsigned Data

Java 语言没有提供直接支持无符号数值(char 除外），但在很多场合下，你需要从数据流或文件中提取无符号信息，或者打包数据区创建文件头或者其他带有无符号字段的结构性信息。字节缓冲区 API 没有提供对此的直接支持，但做这个不困难。你只需要当心精度。当你必须处理缓冲区中的无符号数据时，Example 2-3 中的工具类还是可用的。

Example 2-3. Utility routines for getting/putting unsigned values

```
package com.ronsoft.books.nio.buffers;

import java.nio.ByteBuffer;

/**
 * Utility class to get and put unsigned values to a ByteBuffer object. All
 * methods here are static and take a ByteBuffer argument. Since java does not
 * provide unsigned primitive types, each unsigned value read from the buffer is
 * promoted up to the next bigger primitive data type. getUnsignedByte() returns
 * a short, getUnsignedShort( ) returns an int and getUnsignedInt() returns a
 * long. There is no getUnsignedLong( ) since there is no primitive type to hold
 * the value returned. If needed, methods returning BigInteger could be
```

```
 * implemented. Likewise, the put methods take a value larger than the type they
 * will be assigning. putUnsignedByte takes a short argument, etc.
 *
 * @author Ron Hitchens (ron@ronsoft.com)
 */
public class Unsigned
{
    public static short getUnsignedByte(ByteBuffer bb)
    {
        return ((short) (bb.get() & 0xff));
    }

    public static void putUnsignedByte(ByteBuffer bb, int value)
    {
        bb.put((byte) (value & 0xff));
    }

    public static short getUnsignedByte(ByteBuffer bb, int position)
    {
        return ((short) (bb.get(position) & (short) 0xff));
    }

    public static void putUnsignedByte(ByteBuffer bb, int position, int value)
    {
        bb.put(position, (byte) (value & 0xff));
    }

    // ----------------------------------------------------------
    public static int getUnsignedShort(ByteBuffer bb)
    {
        return (bb.getShort() & 0xffff);
    }

    public static void putUnsignedShort(ByteBuffer bb, int value)
    {
        bb.putShort((short) (value & 0xffff));
    }

    public static int getUnsignedShort(ByteBuffer bb, int position)
    {
        return (bb.getShort(position) & 0xffff);
    }

    public static void putUnsignedShort(ByteBuffer bb, int position, int value)
```

```
    {
        bb.putShort(position, (short) (value & 0xffff));
    }

    // ------------------------------------------------------------
    public static long getUnsignedInt(ByteBuffer bb)
    {
        return ((long) bb.getInt() & 0xffffffffL);
    }

    public static void putUnsignedInt(ByteBuffer bb, long value)
    {
        bb.putInt((int) (value & 0xffffffffL));
    }

    public static long getUnsignedInt(ByteBuffer bb, int position)
    {
        return ((long) bb.getInt(position) & 0xffffffffL);
    }

    public static void putUnsignedInt(ByteBuffer bb, int position, long value)
    {
        bb.putInt(position, (int) (value & 0xffffffffL));
    }
}
```

2.4.6 Memory-Mapped Buffers

Mapped buffer 是一个数据元素存储在文件中，且通过 memory mapping 访问的 byte buffer。**Mapped buffer 都是 直接缓冲区** ，且只能通过 FileChannel 创建。MappedByteBuffer 的用法类似 direct buffer，但也有一些文件访问的特性。所以，我们会推迟到 Section 3.4 再讨论 mapped buffer，在那里还同时讨论了文件锁。

2.5 Summary

本章讨论了 java.nio 包中的 buffers。Buffer 能让那些在剩下的章节讨论到的高级的 I/O 运行。下面就列出了 Buffer 包含的一些要点：

Buffer attributes

所有缓冲区都拥有这些在 2.1.1 节中讨论的属性。这些属性描述了一个缓冲区的当前状态并影响它的行为。在本节，我们还学习了如何操作缓冲区的状态和如何添加或移

除数据元素。

Buffer creation

Section 2.2，我们学习了如何创建缓冲区；Section 2.3，如何复制缓冲区。还有很多缓冲区类型。缓冲区创建的方式决定了缓冲区应当被放在哪里和如何使用。

Byte buffers

除了boolean，我们可以创建其他基本数据类型的buffers。但byte buffer有着和其他类型不同的特性。只有byte buffer可以和Channel一起使用(第三章讨论)，并且byte buffer还提供将它的内容映射为其他类型 的view 缓冲区。我们还了解了关于byte ordering的问题。在Section 2.4我们讨论了*ByteBuffer*s。

到这里我们就结束了 java.nio 中的 buffer 之旅。旅行的下一站是 java.nio.channels，毫无意外，我们会遭遇到 channels。Channels 和 byte buffer 结合，共同打开高性能 I/O 之门。回到车上，即将抵达下一站。