009.NIO

同步、异步、阻塞和非阻塞

1）同步(Synchronization)和异步(Asynchronous)的方式：

同步和异步都是基于应用程序所在操作系统处理IO事件所采用的方式，

比如同步：是应用程序要直接参与IO读写的操作。

异步：所有的IO读写交给操作系统去处理，应用程序只需要等待通知。

举个通俗的例子：你打电话问书店老板有没有《分布式系统》这本书，

如果是同步通信机制，书店老板会说，你稍等，”我查一下″，然后开始查啊查，

等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。

而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。

然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

2）阻塞(Block)和非租塞(NonBlock)：

阻塞和非阻塞是进程在访问数据的时候，数据是否准备就绪的一种处理方式，

当数据没有准备的时候阻塞：往往需要等待缓冲区中的数据准备好过后才处理其他的事情，否则一直等待在那里。

非阻塞：当我们的进程访问我们的数据缓冲区的时候，如果数据没有准备好则直接返回，不会等待。

如果数据已经准备好，也直接返回。

还是上面的例子，你打电话问书店老板有没有《分布式系统》这本书，

你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，

如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了，

当然你也要偶尔过几分钟check一下老板有没有返回结果。

在这里阻塞与非阻塞与是否同步异步无关。跟老板通过什么方式回答你结果无关。

BIO、NIO、AIO的概述

首先，传统的 java.io包，它基于流模型实现，提供了我们最熟知的一些 IO 功能，比如 File 抽象、输入输出流等。

交互方式是同步、阻塞的方式，也就是说，在读取输入流或者写入输出流时，在读、写动作完成之前，

线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。

java.io包的好处是代码比较简单、直观，缺点则是 IO 效率和扩展性存在局限性，容易成为应用性能的瓶颈。
尤其是在网络编程中，瓶颈体现的非常明显！
很多时候，人们也把 java.net下面提供的部分网络 API，
比如 Socket、ServerSocket、HttpURLConnection 也归类到同步阻塞 IO 类库，因为网络通信同样是 IO 行为。

第二，在 Java 1.4 中引入了 NIO 框架（java.nio 包），提供了 Channel、Selector、Buffer 等新的抽象，
可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层的高性能数据操作方式。

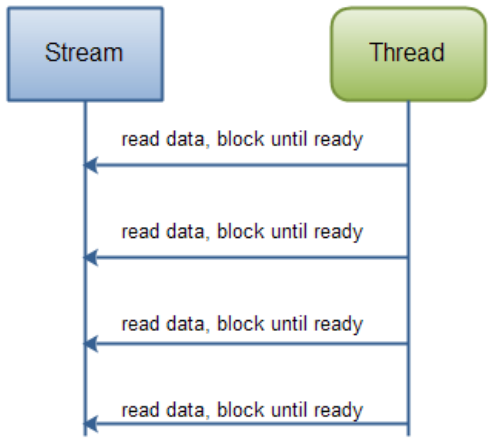
第三，在 Java 7 中，NIO 有了进一步的改进，也就是 NIO 2，引入了异步非阻塞 IO 方式，
也有很多人叫它 AIO（Asynchronous IO）。异步 IO 操作基于事件和回调机制，可以简单理解为，
应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

一、IO流（同步、阻塞），有Block-IO，有的资料上你可以看到BIO其实就是我们前面学的很爽的IO模型

二、NIO（同步、非阻塞）

BIO和NIO的工作模式对比：

BIO工作模式：



看程序：

分析：一旦reader.readLine()方法返回，你就可以确定整行已经被读取，readLine()阻塞直到一整行都被读取

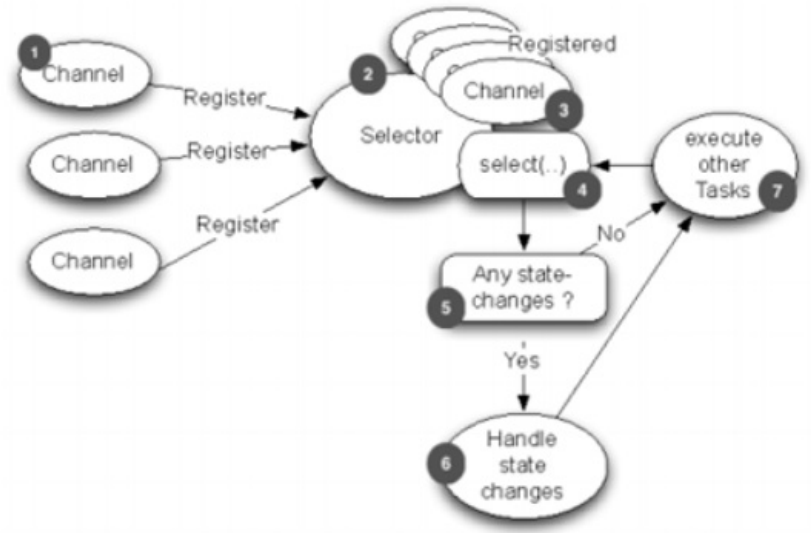
NIO工作模式：

NIO包三个核心组件

1)Channel

2)Buffer

3)Selector



1.由一个专门的线程(Selector)来处理所有的IO事件，并负责分发。

2.事件驱动机制：事件到的时候触发，而不是同步的去监视事件。

3.线程通讯：线程之间通过 wait,notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的线程切换。

核心组件的细节：

通道Channel

通道是一个对象，通过它可以读取和写入数据，当然了所有数据都通过Buffer对象来处理。我们永远不会将字节直接写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区，然后把缓冲区里的数据写入通道。

同样不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

在NIO中，提供了多种通道对象，而所有的通道对象都实现了Channel接口。

JAVA NIO中的一些主要Channel的实现：

`FileChannel` 连接到文件的通道

`DatagramChannel` 连接到UDP包的通道 （下门课网络编程与NIO进阶）

`SocketChannel` 连接到TCP网络套接字的通道（下门课网络编程与NIO进阶）

`ServerSocketChannel` 监听新进来的TCP连接的通道（下门课网络编程与NIO进阶）

FileChannel

Java NIO中的`FileChannel`是一个连接到文件的通道。可以通过文件通道读写文件。

FileChannel无法设置为非阻塞模式，它总是运行在阻塞模式下。

打开FileChannel

在使用FileChannel之前，必须先打开它。但是，我们无法直接打开一个FileChannel，需要通过使用一个InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。

下面是通过RandomAccessFile打开FileChannel的示例：

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
```

从FileChannel读取数据

调用多个read()方法之一从FileChannel中读取数据。如：

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
```

首先，分配一个Buffer。从FileChannel中读取的数据将被读到Buffer中。

然后，调用FileChannel.read()方法。该方法将数据从FileChannel读取到Buffer中。read()方法返回的int值表示了有多少字节被读到了Buffer中。如果返回-1，表示到了文件末尾。

```
  aFile = new RandomAccessFile("D:\\SerialVersion.txt", "rw");
    // 获取通道
        FileChannel inChannel = aFile.getChannel();
        // 创建缓冲区
        ByteBuffer byteBuffer = ByteBuffer.allocate(48);
        // 读取数据到缓冲区 ， 读取的字节数，可能为零，如果通道已达到流末端， 则为-1
        int byteReader = inChannel.read(byteBuffer);
        while (byteReader != -1) {
            System.out.println("Read:" + byteReader);
            byteBuffer.flip();
            while (byteBuffer.hasRemaining()) { //hasRemaining()判断有没有剩余的可用数据
                System.out.println((char)byteBuffer.get());
            }
            byteBuffer.clear();
            byteReader = inChannel.read(byteBuffer);
        }
```

向FileChannel写数据

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer

```
public class FileOutputProgram {
    static private final byte message[] = { 83, 111, 109, 101, 32,   98, 121, 116, 101,
115, 46 };
    static public void main( String args[] ) throws Exception {
        FileOutputStream fout = new FileOutputStream( "e:\\test.txt" );
        FileChannel fc = fout.getChannel();
        ByteBuffer buffer = ByteBuffer.allocate( 1024 );
        for (int i=0; i<message.length; ++i) {
            buffer.put( message[i] );
        }
        buffer.flip();
     while(buffer.hasRemaining()) {
```

```
            fc.write( buffer );
        }
        fout.close();
    }
}
```

注意FileChannel.write()是在while循环中调用的。因为无法保证write()方法一次能向FileChannel写入多少字节，

因此需要重复调用write()方法，直到Buffer中已经没有尚未写入通道的字节。

关闭FileChannel

用完FileChannel后必须将其关闭。如：

```
channel.close();
```

FileChannel的position方法

有时可能需要在FileChannel的某个特定位置进行数据的读/写操作。

可以通过调用position()方法获取FileChannel的当前位置。

也可以通过调用position(long pos)方法设置FileChannel的当前位置。

这里有两个例子:

```
long pos = channel.position();
channel.position(pos +123);
```

如果将位置设置在文件结束符之后，然后试图从文件通道中读取数据，读方法将返回-1

如果将位置设置在文件结束符之后，然后向通道中写数据，文件将撑大到当前位置并写入数据。

这可能导致“文件空洞”，磁盘上物理文件中写入的数据间有空隙。

FileChannel的size方法

FileChannel实例的size()方法将返回该实例所关联文件的大小。如:

```
long fileSize = channel.size();
```

FileChannel的truncate方法

可以使用FileChannel.truncate()方法截取一个文件。截取文件时，文件中指定长度后面的部分将被删除。如:

```
channel.truncate(1024);    //这个例子截取文件的前1024个字节。
```

FileChannel的force方法

FileChannel.force()方法将通道里尚未写入磁盘的数据强制写到磁盘上。

出于性能方面的考虑，操作系统会将数据缓存在内存中，所以无法保证写入到FileChannel里的数据一定会即时写到磁盘上。

要保证这一点，需要调用force()方法。

force()方法有一个boolean类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

下面的例子同时将文件数据和元数据强制写到磁盘上：

```
channel.force(true);
```

缓冲区Buffer

缓冲区实际上是一个容器对象，更直接的说，其实就是一个数组，在NIO中，所有数据都是用缓冲区处理的。

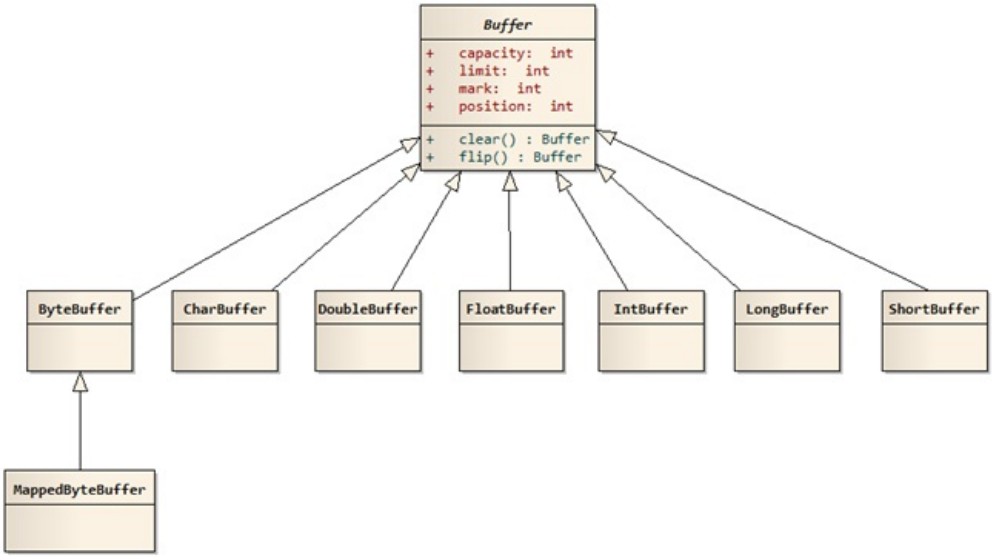
在读取数据时，它是直接读到缓冲区中的； 在写入数据时，它也是写入到缓冲区中的；

任何时候访问 NIO 中的数据，都是将它放到缓冲区中。

而在面向流I/O系统中，所有数据都是直接写入或者直接将数据读取到Stream对象中。

在NIO中，所有的缓冲区类型都继承于抽象类Buffer，最常用的就是ByteBuffer，

对于Java中的基本类型，基本都有一个具体Buffer类型与之相对应，它们之间的继承关系如下图所示：



1、缓冲区基础

理论上，Buffer是经过包装后的数组，而Buffer提供了一组通用api对这个数组进行操作。

1.1、属性：

容量（Capacity）

缓冲区能够容纳的数据元素的最大数量。这一容量在缓冲区创建时被设定，并且永远不能被改变。

上界（Limit）

缓冲区的第一个不能被读或写的元素。或者说，缓冲区中现存元素的计数。

位置（Position）

下一个要被读或写的元素的索引。位置会自动由相应的get( )和put( )函数更新。

标记（Mark）

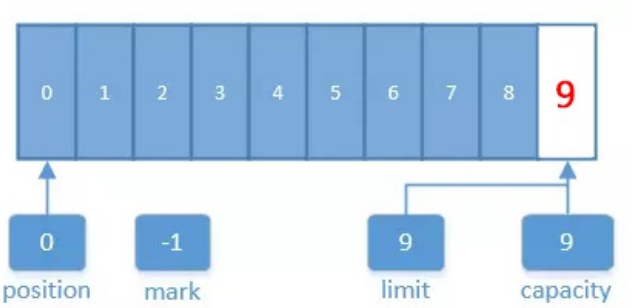
一个备忘位置。调用mark( )来设定mark = postion。

调用reset( )设定position = mark。

标记在设定前是默认值是-1。

这四个属性之间总是遵循以下关系： mark <= position <= limit <= capacity 让我们来看看这些属性在实际应用中的一些例子。

新创建的容量为9的ByteBuffer逻辑视图：



位置被设置为0，而且容量和上界被设置为9，超过缓冲区能容纳的最后一个字节，

标记初始未定义（-1），当缓冲区创建完毕，则容量就不变，而其他属性随读写等操作改变。

1.2、缓存区API

public final int capacity( ) ：返回缓冲区的容量

public final int position( ) ：返回缓冲区的位置

public final Buffer position (int newPosition)：设置缓冲区的位置

public final int limit( ) ：返回缓冲区的上界

public final Buffer limit (int newLimit) ：设置缓冲区的上界

public final Buffer mark( ) ：设置缓冲区的标记

public final Buffer reset( ) ：将缓冲区的位置重置为标记的位置

public final Buffer clear( ) ：清除缓冲区

public final Buffer flip( ) ：反转缓冲区

public final Buffer rewind( ) ：重绕缓冲区

public final int remaining( ) ：返回当为位置与上界之间的元素数

public final boolean hasRemaining( ) ：缓存的位置与上界之间是否还有元素

public abstract boolean isReadOnly( )：缓冲区是否为只读缓冲区

1.3、存取

Buffer提供了一组获取缓冲区的api：

public abstract byte get( )：获取缓冲区当前位置上的字节，然后增加位置；

public abstract byte get (int index)：获取指定索引中的字节；

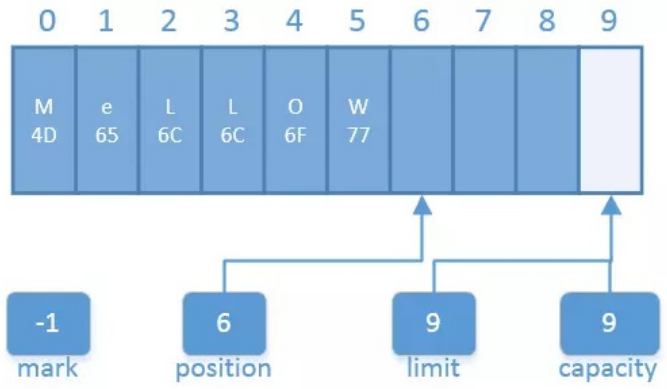
public abstract ByteBuffer put (byte b)：将字节写入当前位置的缓冲区中，并增加位置；

public abstract ByteBuffer put (int index, byte b)：将自己写入给定位置的缓冲区中；

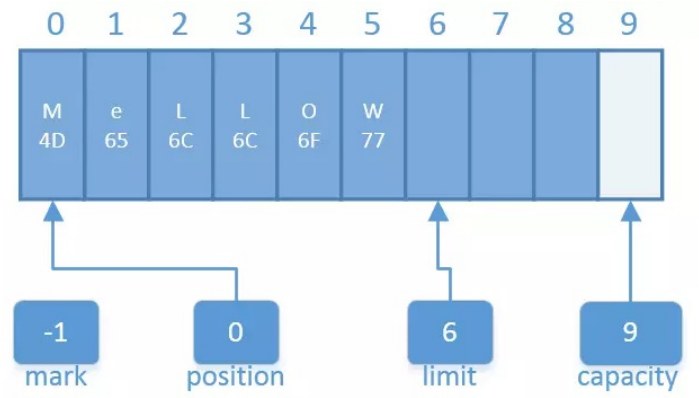
1.4、翻转，flip( )函数

void flip( )：将一个能够继续添加数据元素的填充状态的缓冲区翻转成一个准备读出元素的释放状态。

翻转前：



翻转后：



flip()源码如下：

```
public final Buffer flip() {
    limit = position;
    position = 0;
    mark = -1;
    return this;
}
```

Rewind()函数与flip()相似，但不影响上界属性。

Rewind()源码如下：

```
public final Buffer rewind() {
    position = 0;
    mark = -1;
    return this;
}
```

1.5、释放，get()、clear()、hasRemaining()和remaining()函数

如果接收到一个在别处被填满的缓冲区，检索内容之前需要将其翻转。

例如，如果一个通道的read()操作完成，要查看被通道放入缓冲区内的数据，那么您需要在调用get()之前翻转缓冲区。

布尔函数hasRemaining()会在释放缓冲区时判断是否已经达到缓冲区的上界。

以下是一种将数据元素从缓冲区释放到一个数组的方法。

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

作为选择，remaining()函数将告知您从当前位置到上界还剩余的元素数目。

缓冲区并不是多线程安全的。

如果您想以多线程同时存取特定的缓冲区，您需要在存取缓冲区之前进行同步。

一旦缓冲区对象完成填充并释放，它就可以被重新使用了。

clear()函数将缓冲区重置为空状态。

它并不改变缓冲区中的任何数据元素，而是仅仅将上界设为容量的值，并把位置设回0。

1.6 、压缩，compact()函数，应用的位置，在clear() 的地方，需要的时候，替换clear()方法来用的。

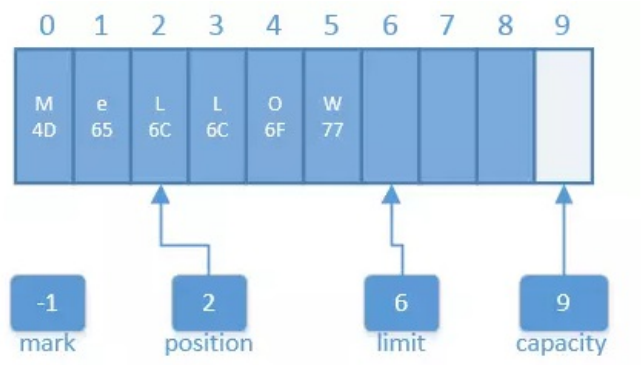
有时，只想从缓冲区中释放一部分数据，而不是全部，然后重新填充。

为了实现这一点，未读的数据元素需要下移至第一个元素位置，即索引为0。

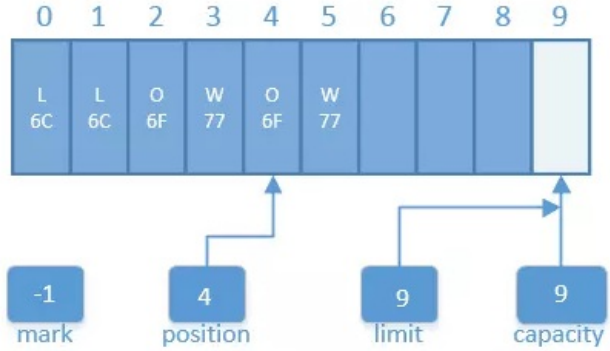
尽管重复这样做会效率低下，但这有时非常必要，而API对此提供了一个compact()函数。

这一缓冲区工具在复制数据时要比使用get()和put()函数高效得多。所以当需要时，可以使用compact()。

compact()之前的缓冲区：



compact()之后的缓冲区：



1.7、标记，mark( )和reset( )函数

标记，使缓冲区能够记住一个位置并在之后将其返回。

缓冲区的标记在mark( )函数被调用之前是未定义的，调用时标记被设为当前位置的值。

reset( )函数将位置设为当前的标记值。

如果标记值未定义，调用reset( )将导致InvalidMarkException异常。

一些缓冲区函数会抛弃已经设定的标记（rewind( )，clear( )，以及flip( )总是抛弃标记）。

如果新设定的值比当前的标记小，调用limit(索引参数)或position(索引参数)会抛弃标记。

1.8、比较，equals()和compareTo( )函数

有时候比较两个缓冲区所包含的数据是很有必要的。

所有的缓冲区都提供了一个常规的equals( )函数用以测试两个缓冲区的是否相等，

以及一个compareTo( )函数用以比较缓冲区哪个大，哪个小。

```
public boolean equals (Object ob)
public int compareTo (Object ob)
```

两个缓冲区被认为相等的充要条件是：

1.两个对象类型相同。包含不同数据类型的buffer永远不会相等，而且buffer绝不会等于非buffer对象。

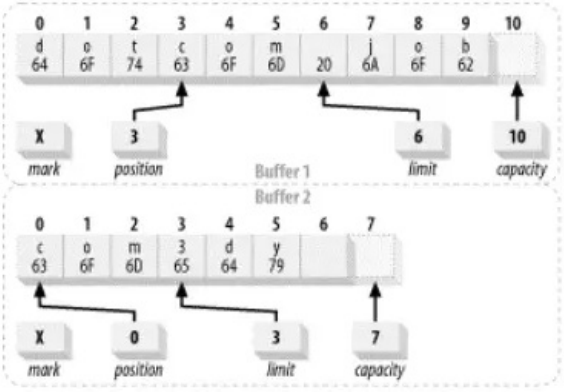
2.两个对象都剩余同样数量的元素。Buffer的容量不需要相同，而且缓冲区中剩余数据的索引也不必相同。

但每个缓冲区中剩余元素的数目（从位置到上界）必须相同。

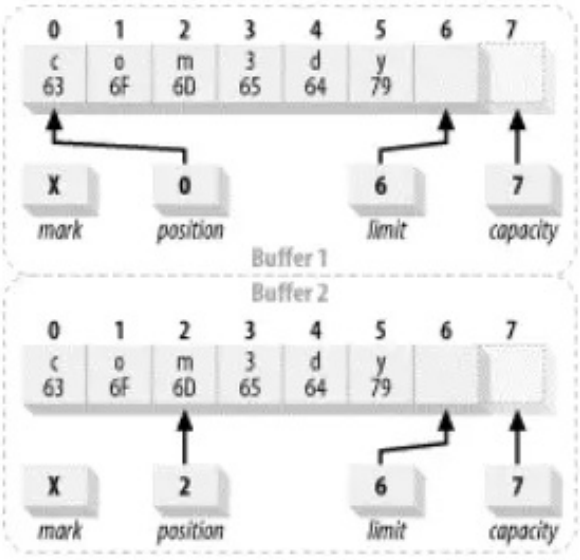
3.在每个缓冲区中应被get()函数返回的剩余数据元素序列必须一致。

如果不满足以上任意条件，就会返回false。

相等的缓冲区：



不等的缓冲区：



缓冲区也支持用compareTo( )函数以词典顺序进行比较。

A.compareTo(B)，A小于B，A等于B，A大于B分别返回一个负整数，0和正整数。

与equals( )相似，compareTo( )不允许不同对象间进行比较。

但compareTo( )更为严格：如果您传递一个类型错误的对象，它会抛出ClassCastException异常，但equals( )只会返回false。

比较是针对每个缓冲区内剩余数据进行的，与它们在equals( )中的方式相同，直到不相等的元素被发现或者到达缓冲区的上界。

如果一个缓冲区在不相等元素发现前已经被耗尽，较短的缓冲区被认为是小于较长的缓冲区。

1.9、批量移动，get(带数组参数）和put(带数组参数/字符串）

buffer API提供了向缓冲区内外批量移动数据元素的函数：

```
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()可供从缓冲区到数组进行的数据复制使用。

第一种形式只将一个数组作为参数，将一个缓冲区释放到给定的数组。

第二种形式使用offset和length参数来指定目标数组的子区间。

批量的用get方法释放缓冲区里的数据可以能会发生异常：

如果所要求的数量的数据不能被传送，那么不会有数据被传递，缓冲区的状态保持不变，同时抛出BufferUnderflowException异常。

换个说法：如果缓冲区中的数据不够完全填满数组，你会得到一个BufferUnderflowException异常。

这意味着如果你想将一个小型缓冲区传入一个大型数组，你需要明确地指定缓冲区中剩余的数据长度。

2、缓冲区的创建

public static CharBuffer allocate (int capacity)：创建容量为capacity的缓冲区

public static CharBuffer wrap (char [] array) ：将array数组包装成缓冲区

public static CharBuffer wrap (char [] array, int offset, int length) ：将array包装为缓冲区，position=offset，limit=lenth

public final boolean hasArray( ) ：判断缓冲区底层实现是否为数组

public final char [] array( ) ：返回缓冲中的数组

public final int arrayOffset( ) ：返回缓冲区数据在数组中存储的开始位置的偏移量；

新的缓冲区是由分配或包装两种操作方式创建的。

分配操作创建一个缓冲区对象并分配一个容量大小的私有的空间来储存数据元素。

包装操作创建一个缓冲区对象，但是不分配任何空间来储存数据元素。

它使用所提供的数组作为存储空间来储存缓冲区中的数据元素。

要分配一个容量为100个char变量的Charbuffer:

```
CharBuffer charBuffer = CharBuffer.allocate (100);
```

这段代码隐含地从堆空间中分配了一个char型数组作为备份存储器来储存100个char变量。

如果您想提供您自己的数组用做缓冲区的备份存储器，请调用wrap()函数：

```
char [] myArray = new char [100];
CharBuffer charbuffer = CharBuffer.wrap (myArray);
```

这段代码构造了一个新的缓冲区对象，但数据元素会存在于数组中。

这意味着通过调用put()函数造成的对缓冲区的改动会直接影响这个数组，而且对这个数组的任何改动也会对这个缓冲区对象可见。

带有offset和length作为参数的wrap()函数版本则会构造一个按照提供的offset和length参数值初始化位置和上界的缓冲区。

通过allocate()或者wrap()函数创建的缓冲区通常都是**间接的**。
间接的缓冲区使用备份数组，像我们之前讨论的，可以通过上面列出的API函数获得对这些数组的存取权。

Boolean型函数hasArray()告诉您这个缓冲区是否有一个可存取的备份数组。
如果这个函数的返回true，array()函数会返回这个缓冲区对象所使用的数组存储空间的引用。
如果hasArray()函数返回false，不要调用array()函数或者arrayOffset()函数。
如果您这样做了您会得到一个UnsupportedOperationException异常。

如果一个缓冲区是只读的，它的备份数组将会是超出上界的，即使一个数组对象被提供给wrap()函数。
调用array()函数或者arrayOffset()会抛出一个ReadOnlyBufferException异常，来阻止您得到存取权来修改只读缓冲区的内容。
如果您通过其它的方式获得了对备份数组的存取权限，对这个数组的修改也会直接影响到这个只读缓冲区。

最后一个函数，arrayOffset()，返回缓冲区数据在数组中存储的开始位置的偏移量。
如果您使用了带有三个参数的版本的wrap()函数来创建一个缓冲区，就是截取原本数组中的一部分来做缓冲区的存储空间，
这个数组偏移量和缓冲区容量值会告诉您数组中哪些元素是被缓冲区使用的。

3、视图缓冲区

缓冲区不限于能管理数组中的数据元素，它们也能管理其他缓冲区中的数据元素。
当一个管理其他缓冲区所包含的数据元素的缓冲区被创建时，这个缓冲区被称为视图缓冲区。

Duplicate()：

Duplicate()函数创建了一个与原始缓冲区相似的新缓冲区，相当于复制。
两个缓冲区共享数据元素，拥有同样的容量，但每个缓冲区拥有各自的位置，上界和标记属性。
对一个缓冲区内的数据元素所做的改变会反映在另外一个缓冲区上。
这一副本缓冲区具有与原始缓冲区同样的数据视图。
如果原始的缓冲区为只读，或者为直接缓冲区，新的缓冲区将继承这些属性。

asReadOnlyBuffer()：

asReadOnlyBuffer() 函数来生成一个**只读**的视图缓冲区。
这与duplicate() 相同，除了这个新的缓冲区不允许使用 put()，并且其 isReadOnly()

函数将会返回true。
如果一个只读的缓冲区与一个可写的缓冲区共享数据，或者有包装好的备份数组，
那么对这个可写的缓冲区或直接对这个数组的改变将反映在所有关联的缓冲区上，包括只读缓冲区。

slice()：

分割缓冲区与复制相似，但 slice() 创建一个从原始缓冲区的当前 position 开始的新缓冲区，
并且其容量是原始缓冲区的剩余元素数量（limit - position）。
这个新缓冲区与原始缓冲区共享一段数据元素子序列。分割出来的缓冲区也会继承只读和直接属性。

4、字节缓冲区独特之处

ByteBuffer 只是 Buffer 的一个子类，但字节缓冲区有字节的独特之处。
字节缓冲区跟其他缓冲区类型最明显的不同在于，它可以成为通道所执行的 I/O 的源头或目标，
我们可以看看FileChannel的源码，就会发现文件通道只接收 ByteBuffer 作为参数。

字节是操作系统及其 I/O 设备使用的基本数据类型。
当在 JVM 和操作系统间传递数据时，都会将其他数据类型拆分成构成它们的字节形式进行传递，
因为，系统层次的 I/O 都是面向字节的。

同时，操作系统是在内存区域中进行 I/O 操作。
这些内存区域，就操作系统方面而言，是相连的字节序列。于是，毫无疑问，只有字节缓冲区有资格直接参与 I/O 操作。

在一个32位的CPU中“字长”为32个bit，也就是4个byte。
在这样的CPU中，总是以4字节对齐的方式来读取或写入内存，
那么像4个字节的多字节数值是以什么顺序保存在内存中的呢？
多字节数值被存储在内存中的方式一般被称为 endian-ness（字节顺序）。

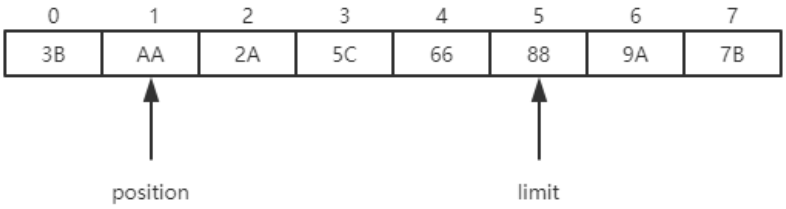
如果多字节数值的最高字节 — big end（大端），
位于低位地址(即 big end 先写入内存，先写入的内存的地址是低位的，后写入内存的地址是高位的)，
那么系统就是大端字节顺序来存储数据的。如果最低字节最先保存在内存中，那么系统就是小端字节顺序。

对于我们常用的CPU架构，如Intel，AMD的CPU使用的都是小端字节顺序。

ByteBuffer类有所不同：默认字节顺序总是ByteOrder.BIG_ENDIAN。

在java.nio中，字节顺序由ByteOrder类封装。
我们可以通过它改变ByteBuffer类的字节顺序。
如果想知道JVM运行的硬件平台的固有字节顺序，请调用静态类函数nativeOrder()来查询。

为什么要讲这个东西呢？



这里有一个字节序列，不同的字节顺序，在从这个字节序列中批量取出字节的结果会一样不！

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

会返回一个由缓冲区中位置1-4的byte数据值组成的int型变量的值。
更具体的写法是：

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

对比：

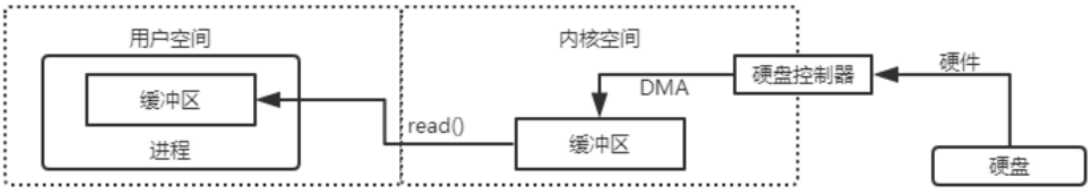
```
int value = buffer.order (ByteOrder.LITTLE_ENDIAN).getInt( );
```

5、直接缓冲区

在电脑开机之前，内存就是一块原始的物理内存。什么也没有。开机加电，系统启动后，就对物理内存进行了划分。
物理内存条上本身没有划分好的地址和空间范围。
这些划分都是操作系统在逻辑上的划分。不同版本的操作系统划分的结果都是不一样的。

一般情况下，会把内存分为用户空间和系统空间(也可以叫内核空间)。
用户空间就是用户进程所在的内存区域，相对的，内核空间就是操作系统占据的内存区域。

内核空间（含运行起来的操作系统程序）能一方面与设备控制器（硬件）链接，
另一方面控制着用户空间中的进程（如 JVM）的运行状态。



在Java NIO中，缓冲区对象，在上面的模式下，就有了直接缓冲区与非直接缓冲区的划

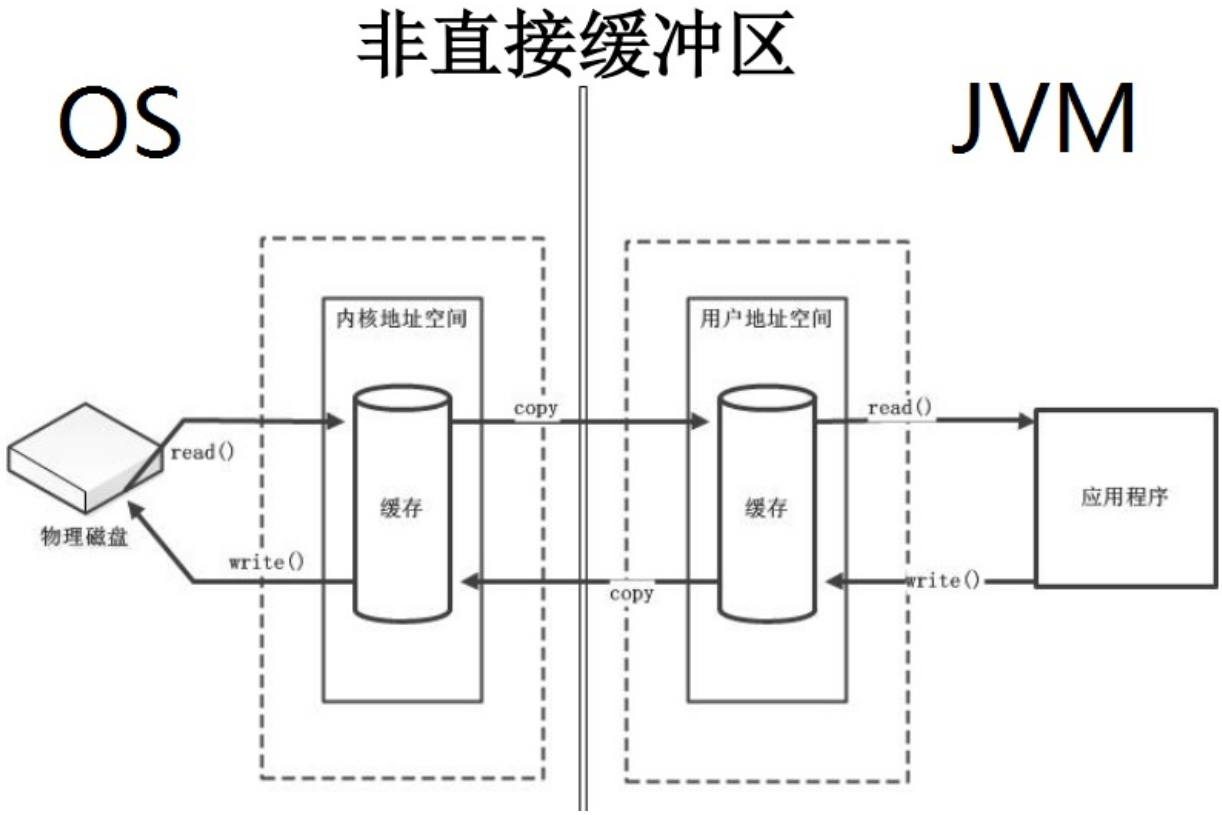
分！

非直接缓冲区：通过 allocate() 方法分配缓冲区，将缓冲区建立在 JVM 的内存中

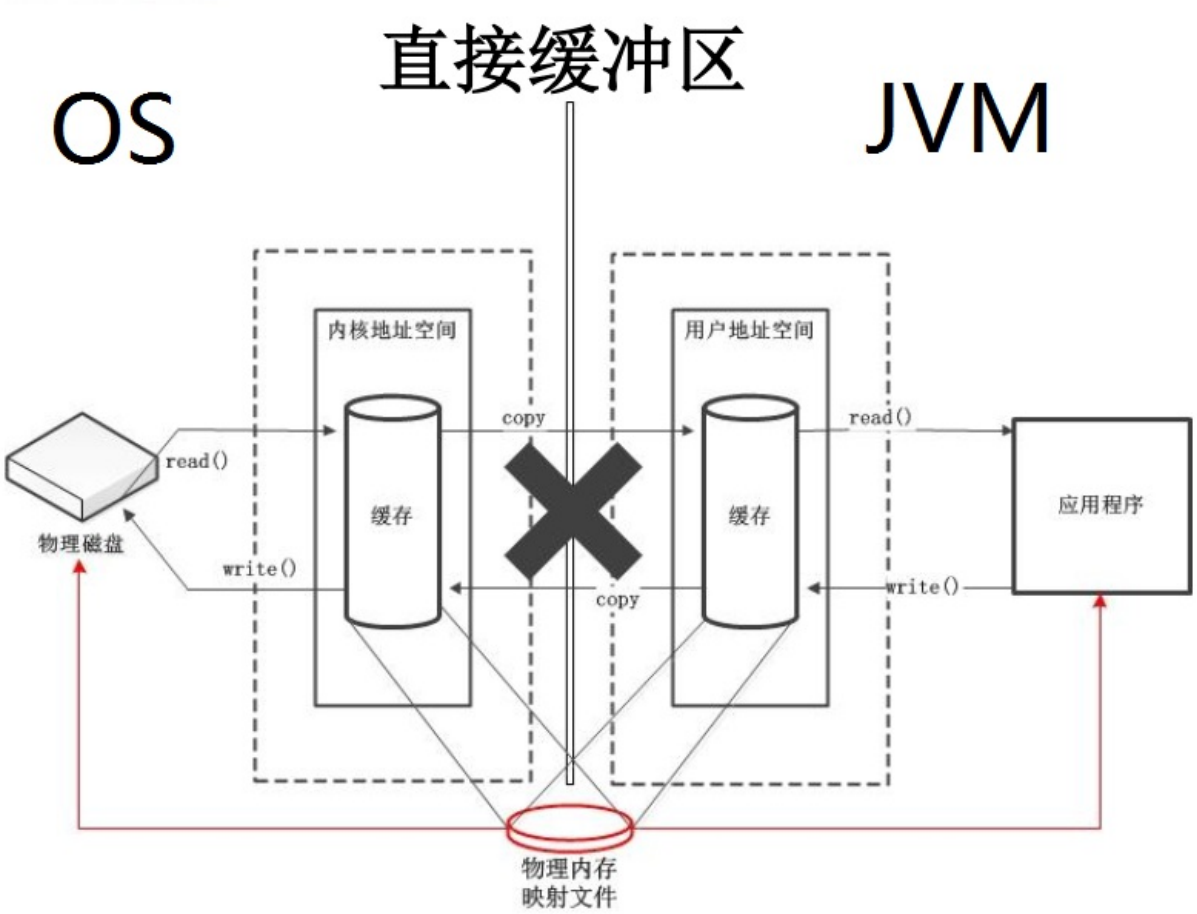
直接缓冲区：通过 allocateDirect() 方法分配的缓冲区，将缓冲区建立在物理内存中。可能提升效率！

非直接缓冲区写入步骤：

1. 创建一个临时的直接ByteBuffer对象。
2. 将非直接缓冲区的内容复制到临时缓冲中。
3. 使用临时缓冲区执行低层次I/O操作。
4. 临时缓冲区对象离开作用域，并最终成为被回收的无用数据。



直接缓冲区：在JVM内存外开辟内存，在每次调用操作系统的一个本机I/O之前或者之后，虚拟机都会避免将缓冲区的内容复制到中间缓冲区（或者从中间缓冲区复制内容），缓冲区的内容驻留在物理内存内，会少一次复制过程，如果需要循环使用缓冲区，用直接缓冲区可以很大地提高性能。虽然直接缓冲区使JVM可以进行高效的I/O操作，但它使用的内存是操作系统分配的，绕过了JVM堆栈，建立和销毁比堆栈上的缓冲区要更大的开销。



注意： allocateDirect()方法创建的直接缓冲区的大小，直接内存DirectMemory的大小默认为 -Xmx 的JVM堆的最大值，但是并不受其限制，而是由JVM参数MaxDirectMemorySize单独控制。

实验验证：

```
public static void main(String[] args) {
    ByteBuffer.allocateDirect(1024*1024*100); // 100MB
}
```

我们设置了JVM堆 64M限制，然后在 直接内存上分配了 100MB空间，程序执行后直接报错：

Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory。

接着我设置 -Xmx200M，程序正常结束。

然后我修改配置： -Xmx64M -XX:MaxDirectMemorySize=200M，程序正常结束。

接下来我们来证明直接内存不是分配在JVM堆中。我们先执行以下程序，并设置 JVM参数 -XX:+PrintGC，

```
public static void main(String[] args) {
   for(int i=0;i<20000;i++) {
        ByteBuffer.allocateDirect(1024*100);  //100K
    }
```

```
}
```

我们看到这里执行 GC的次数较少，但是触发了 Full GC，原因在于直接内存不受 GC(新生代的Minor GC)影响，

只有当执行老年代的 Full GC时候才会顺便回收直接内存！

而直接内存是通过存储在JVM堆中的DirectByteBuffer对象来引用的，

所以当众多的DirectByteBuffer对象从新生代被送入老年代后才触发了 full gc。

把allocateDirect换成allocate后：

可以看到，由于直接在 JVM堆上分配内存，所以触发了多次GC，且不会触及 Full GC，因为对象根本没机会进入老年代。

试验对比：

```
File file = new File("F:\\aa");
FileInputStream in = new FileInputStream(file);
FileChannel channel = in.getChannel();
ByteBuffer buff = ByteBuffer.allocate(1024);

long begin = System.currentTimeMillis();
while (channel.read(buff) != -1) {
    buff.flip();
    buff.clear();
}
long end = System.currentTimeMillis();
System.out.println("time is:" + (end - begin));
```

对比换成allocateDirect后：

```
File file = new File("F:\\aa");
FileInputStream in = new FileInputStream(file);
FileChannel channel = in.getChannel();
ByteBuffer buff = ByteBuffer.allocateDirect(1024);    //(int)file.length()

long begin = System.currentTimeMillis();
while (channel.read(buff) != -1) {
    buff.flip();
    buff.clear();
}
long end = System.currentTimeMillis();
System.out.println("time is:" + (end - begin));
```

普通的NIO通道操作时间：51M文件：369ms，1.7G文件：13643ms

直接缓冲区操作时间：51M文件：117ms，1.7G文件：3511ms

直接缓冲区比普通的NIO通道操作时间要快一点！

通过修改ByteBuffer buff = ByteBuffer.allocateDirect(1024);
为 ByteBuffer buff = ByteBuffer.allocateDirect((int)file.length()),
即一次性分配整个文件长度大小的堆外内存，最终输出为546毫秒，

由此可以得出结论：堆外内存的分配耗时比较大.
最后一点为 DirectMemory的内存只有在 JVM执行 full gc 的时候才会被回收，
那么如果在其上分配过大的内存空间，那么也将出现 OutofMemoryError，即便 JVM 堆中的很多内存处于空闲状态。

直接字节缓冲区还可以过 通过FileChannel 的 map() 方法将文件区域直接映射到内存中来创建 。
该方法返回MappedByteBuffer。

MappedByteBuffer

java io操作中通常采用BufferedReader，BufferedInputStream等带缓冲的IO类处理大文件，
不过java nio中引入了一种基于MappedByteBuffer操作大文件的方式，其读写性能极高！

内存管理

在深入MappedByteBuffer之前，先看看计算机内存管理的几个术语：

- MMU：Memory Management Unit的缩写，中文名是内存管理单元，有时称作分页内存管理单元，CPU的内存管理单元。
- **物理内存**：即内存条的内存空间。
- **虚拟内存**：计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，待应用程序需要访问那这部分数据时，再现从外部磁盘上调度进入物理内存。
- **页面文件**：操作系统反映构建并使用虚拟内存的硬盘空间大小而创建的文件，在windows下，即pagefile.sys文件，其存在意味着物理内存被占满后，将暂时不用的数据移动到硬盘上。
- **缺页中断**：就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。

为什么会有虚拟内存？

如果正在运行的一个进程，它所需的内存是有可能大于内存条的总容量，如内存条是256M，程序却要创建一个2G的数据区，这个时候就必须将一部分数据暂时存储在外部磁盘存储器上，虚拟内存技术就派上用场了。

什么是虚拟内存地址和物理内存地址？

虚拟内存地址范围的大小由CPU的位数决定，例如一个32位的CPU，它的地址范围是0~0xFFFFFFFF （4G)，这个范围就是我们的程序能够产生的地址范围，我们把这个地址范围称为虚拟地址空间，该空间中的某一个地址我们称之为虚拟地址。

物理内存地址：与虚拟地址空间和虚拟地址相对应的则是物理地址空间和物理地址，大多数时候我们的系统所具备的物理地址空间只是虚拟地址空间的一个子集。这里举一个最简单的例子直观地说明这两者，对于一台内存为256M的32bit x86主机来说，它的虚拟地址空间范围是0~0xFFFFFFFF（4G），而物理地址空间范围是0x00000000 ~ 0x0FFFFFFF（256M）。

内存分页机制

大多数使用虚拟内存的系统都使用一种称为分页（paging）机制。虚拟地址空间划分成称为页（page）的单位，而相应的物理地址空间也被进行划分，单位是页帧(frame)。页和页帧的大小必须相同。

在这个例子中我们有一台可以生成32位地址的机器，它的虚拟地址范围从0~0xFFFFFFFF（4G），而这台机器只有256M的物理地址，因此他可以运行4G的程序，但该程序不能一次性调入内存运行。这台机器必须有一个达到可以存放4G程序的外部存储器（例如磁盘），以保证程序片段在需要时可以被调用。在这个例子中，页的大小为4K，页帧大小与页相同——这点是必须保证的，因为内存和外围存储器之间的传输总是以页为单位的。对应4G的虚拟地址和256M的物理存储器，他们分别包含了1M个页和64K个页帧。

虚拟内存与物理内存之间的联系是通过分页表来建立

页表是一种特殊的数据结构，放在系统空间的页表区，存放逻辑页与物理页帧的对应关系。每一个进程都拥有一个自己的页表，PCB(进程管理块)表中有指针指向页表。

虚拟内存页的个数 > 物理内存页帧的个数，岂不是有些虚拟内存页的地址永远没有对应的物理内存地址空间？

不是的，操作系统是这样处理的。操作系统有个页面失效（page fault）功能。

操作系统找到一个使用的最少的页帧，使之失效，并把它写入磁盘，随后把需要访问的页放到这个页帧中来，

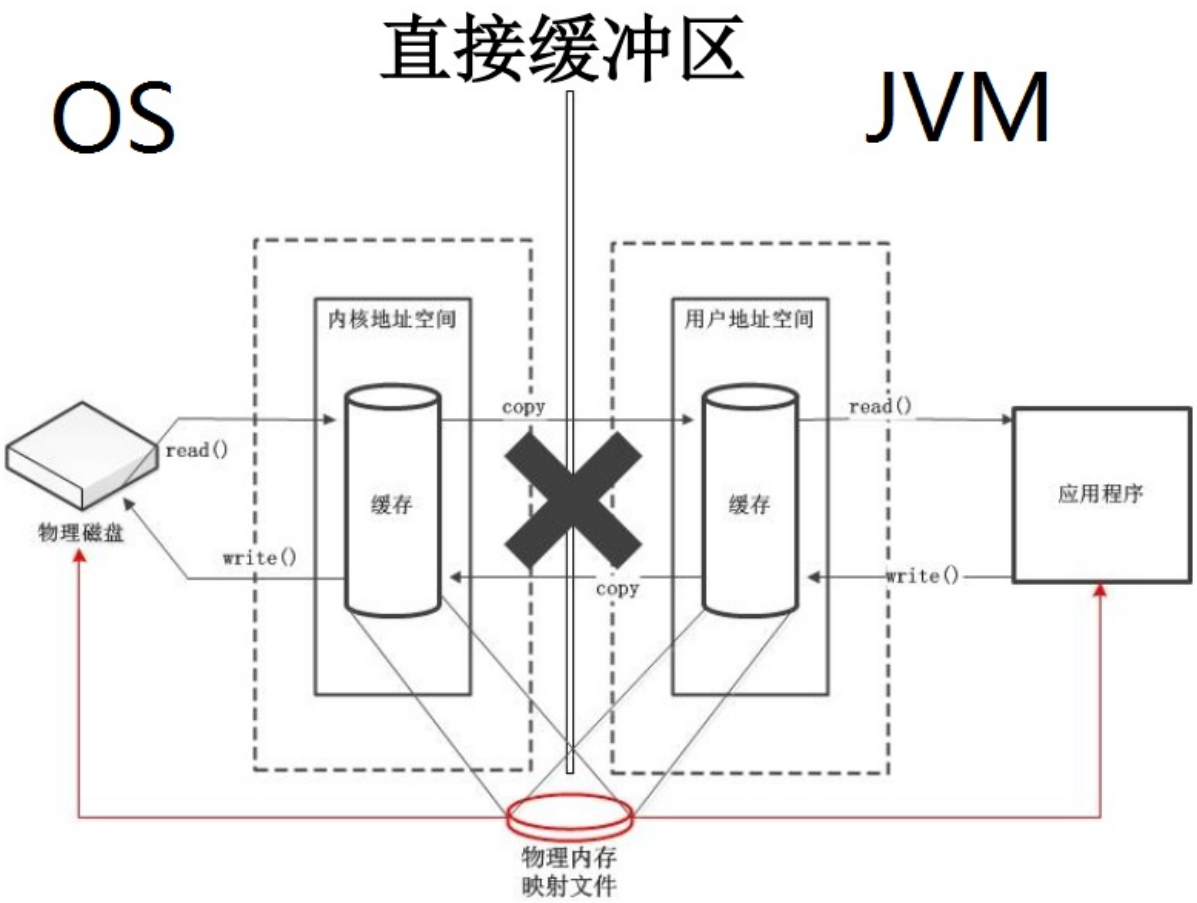
并修改**页表中的映射**，保证了所有的页都会有对应的物流内存地址空间。

虚拟内存地址组成：由页号（与页表中的页号关联）和偏移量（页的小大，即这个页能存了多少数据）组成。

举个例子，有一个虚拟地址它的页号是4，偏移量是20，那么他的寻址过程是这样的：首先

到页表中找到页号4对应的页帧号（比如为8），如果这个页号没有与之对应的页帧号，则用失效机制调入页，接着把页帧号和偏移量传给MMU组成一个物理上真正存在的地址，最后就是访问物理内存的数据了。

直接缓冲区内部真正的实现的逻辑：



虚拟内存技术可以让硬盘上文件的位置与进程逻辑地址空间（虚拟内存地址空间）中一块大小相同的区域之间的一一对应，
这种对应关系纯属是逻辑上的概念，物理上是不存在的，原因是进程的逻辑地址空间本身就是不存在的。

在内存映射的过程中，并没有实际的数据拷贝，文件没有被载入内存，只是逻辑上被放入了内存，
具体到代码：

```
static final int SPLITERATOR_CHARACTERISTICS =
    Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.ORDERED;

// Invariants: mark <= position <= limit <= capacity
private int mark = -1;
private int position = 0;
private int limit;
private int capacity;

// Used only by direct buffers
// NOTE: hoisted here for speed in JNI GetDirectBufferAddress
long address;
```

```
    }

    var7 = 0L;
    var33 = new FileDescriptor();
    if (this.writable && var6 != 0) {
        var13 = Util.newMappedByteBuffer(0, 0L, var33, (Runnable)null);
        return var13;
    }
```

map函数的作用是把文件映射到内存中，获得内存地址var33，然后通过这个var33构造MappedByteBuffer类

```
private static void initDBBConstructor() {
    AccessController.doPrivileged(run() → {
            try {
                Class var1 = Class.forName("java.nio.DirectByteBuffer");
                Constructor var2 = var1.getDeclaredConstructor(Integer.TYPE, Long.
                var2.setAccessible(true);
                Util.directByteBufferConstructor = var2;
                return null;
            } catch (NoSuchMethodException | IllegalArgumentException | ClassCastE
                throw new InternalError(var3);
            }
    });
}
```

实质上是通过构造DirectByteBuffer。

DirectByteBuffer继承了MappedByteBuffer，主要是实现了byte获得函数get等

```
public long address() { return address; }

private long ix(int i) { return address + ((long)i << 0); }

public byte get() { return ((unsafe.getByte(ix(nextGetIndex())))); }

public byte get(int i) { return ((unsafe.getByte(ix(checkIndex(i))))); }
```

由于已经通过map0()函数返回内存文件映射的address，这样就无需调用read或write方法对文件进行读写，通过address就能够操作文件。底层采用unsafe.getByte方法，通过（address + 偏移量）获取指定内存的数据。

既然建立内存映射没有进行实际的数据拷贝，

那么进程又怎么能通过（address + 偏移量）最终直接通过内存操作访问到硬盘上的

文件呢？

（address + 偏移量）所指向的是一个虚拟地址，要操作其中的数据，必须通过MMU将虚拟地址转换成物理地址

建立内存映射并没有实际拷贝数据，这时，MMU在地址映射表中是无法找到与address + 偏移量相对应的物理地址的，也就是MMU失败，将产生一个缺页中断，缺页中断的中断响应函数会在物流内存中寻找相对应的页帧，如果找不到（也就是该文件从来没有被读入内存的情况），则会通过map()建立的映射关系，直接从硬盘上将文件读取到物流内存中。

逻辑清楚了，接下来具体来看FileChannel 的 map（FileChannel.MapMode mode，long position，long size）方法使用：

1）mode内存映像文件访问的方式，共三种：

a）MapMode.READ_ONLY：只读，试图修改得到的缓冲区将导致抛出异常。

b）MapMode.READ_WRITE：读/写，对得到的缓冲区的更改最终将写入文件；

c）MapMode.PRIVATE：私用，可读可写,但是修改的内容不会写入文件，只是buffer自身的改变。

2）position：文件中的位置，映射区域从此位置开始；必须为非负数。

3）size：要映射的区域大小；必须为非负数且不大于Integer.MAX_VALUE。

```
File file = new File("e:\\aaa");
FileInputStream in = new FileInputStream(file);
FileChannel channel = in.getChannel();
MappedByteBuffer buff = channel.map(FileChannel.MapMode.READ_ONLY, 0,channel.size());

int len = (int) file.length();
byte[] ds = new byte[(int) len];

long begin = System.currentTimeMillis();
while(buffer.hasRemaining()){
    buffer.get();
}
channel.close();
long end = System.currentTimeMillis();
System.out.println("time is:" + (end - begin));
```

第1个坑，MappedByteBuffer没有unmap，这个文件会一直被程序的资源句柄占用着。

```
Cleaner var1 = ((DirectBuffer)buffer).cleaner();
if (var1 != null) {
    var1.clean();
}

//文件的资源句柄有没有被释放，可以通过修改文件名来测试
file.renameTo(new File("e:\\eee");
```

名字被改了，证明资源句柄释放了，没改表名没放

第2个坑，MappedByteBuffer处理大文件，一次最多只能读2G内容到内存中，为了读取

大文件，需要循环读取处理：

```
File file = new File("e:\\ccc");
FileInputStream fin = new FileInputStream(file);
FileChannel channel = fin.getChannel();
long len = file.length();
long cur = 0;
//512M
long length = 1L << 29;
MappedByteBuffer buffer = null;

long start = System.currentTimeMillis();
for (;cur < len;cur+=length){
    if(len-cur < length){
        buffer = channel.map(FileChannel.MapMode.READ_ONLY, cur, len-cur);
    }else{
        buffer = channel.map(FileChannel.MapMode.READ_ONLY,cur,length);
    }
    while (buffer.hasRemaining()){
        buffer.get();
    }
}
channel.close();
long end = System.currentTimeMillis();
System.out.println("执行时间：" + (end-start));
```

MappedByteBuffer是从FileChannel中map()出来的，为 什么它又不提供unmap()呢？SUN自己也没有讲清楚为什么。

在sun网站也有相应的BUG报告：bug id:4724038链接为http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4724038，但是sun自己不认为是BUG，而只是一个RFE(Request For Enhancement)，有待改进。

字节缓冲区是直接缓冲区还是非直接缓冲区可通过调用其 isDirect() 方法来确定。

6、数据元素视图

ByteBuffer类提供了一个不太重要的机制来以多字节数据类型的形式存取byte数据组。

ByteBuffer类为每一种原始数据类型提供了存取的和转化的方法：

getXXX()，如：将ByteBuffer转int，getInt()；

putXXX()；如：将int转ByteBuffer，putInt()；

这些函数从当前位置开始存取ByteBuffer的字节数据，就好像一个数据元素被存储在那里一样。

根据这个缓冲区的当前的有效的字节顺序，这些字节数据会被排列或打乱成需要的原始数据类型。

原始数据的值会根据字节顺序被分拆成一个个byte数据。

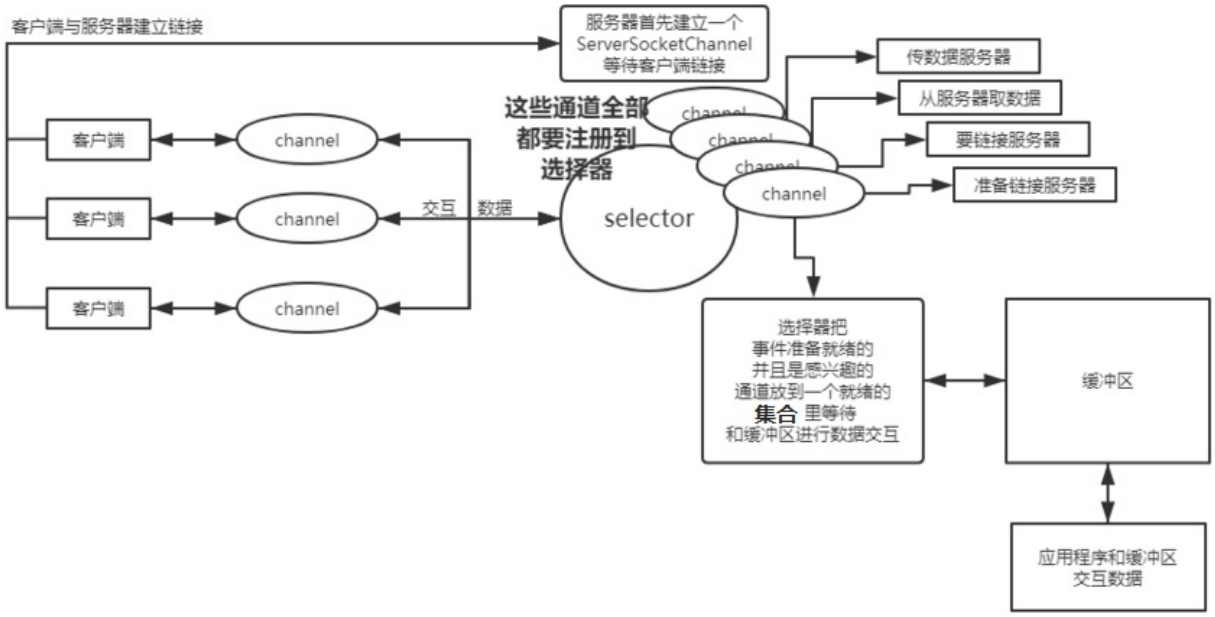
如果存储这些字节数据的空间不够，会抛出BufferOverflowException。

每一个函数都有有参数和无参数的形式。有参数对位置属性加上特定的需求。然后无参数的

形式则不改变位置属性。

选择器

放进NIO体系进行网络编程的工作流程：



Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

```
Selector selector = Selector.open();
```

向Selector注册通道

通过Channel.register()方法来实现，

注意：Channel和Selector一起使用时，Channel必须处于非阻塞模式下。

```
channel.configureBlocking(false);  //设置通道为非阻塞模式
SelectionKey key = channel.register(selector,Selectionkey.OP_READ);
```

register()方法的第二个参数：是一个“*兴趣(interest)集合*”，意思是在通过Selector监听Channel时对什么事件感兴趣。

可以监听四种不同类型的事件：

1.Connect　链接就绪，某个channel成功连接到另一个服务器称为“连接就绪”。

2.Accept　接收就绪，一个server socket channel准备好接收新进入的连接称为“接收就绪”。

3.Read　读就绪，一个有数据可读的通道可以说是“读就绪”。

4.Write　写就绪，等待写数据的通道可以说是“写就绪”。

这四种事件用SelectionKey的四个常量来表示：

1.SelectionKey.OP_CONNECT

2.SelectionKey.OP_ACCEPT

3.SelectionKey.OP_READ

4.SelectionKey.OP_WRITE

如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

SelectionKey说明

当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。

这个对象包含了一些有用的属性：

1.interest集合

2.ready集合

3.Channel

4.Selector

5.附加的对象（可选）

interest集合

interest集合是你所选择的感兴趣的事件集合。

可以通过SelectionKey读写interest集合，像这样：

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept  = (interestSet & SelectionKey.OP_ACCEPT) ==
SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE;
```

可以看到，用“和”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的事件是否在interest 集合中。

ready集合

ready 集合是通道已经准备就绪的操作的集合。在一次选择(Selection)之后，你会首先访问这个ready set。

可以这样访问ready集合：

```
int readySet = selectionKey.readyOps();
```

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。

但是，也可以使用以下四个方法，它们都会返回一个布尔类型：

```
selectionKey.isAcceptable();
```

```
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

Channel + Selector

从SelectionKey访问Channel和Selector很简单。如下：

```
Channel  channel  = selectionKey.channel();
Selector selector = selectionKey.selector();
```

附加的对象

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。

例如，可以附加 与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：

```
selectionKey.attach(theObject);
Object attachedObj = selectionKey.attachment();
```

还可以在用register()方法向Selector注册Channel的时候附加对象。如：

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ,
theObject);
```

选择器的select()

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。

这些方法返回你所感兴趣的事件（如连接、接受、读或写）同时这些事件已经准备就绪的那些通道。

换句话说，如果你对“读就绪”的通道感兴趣，select()方法会返回读事件已经就绪的那些通道。

下面是select()方法：

1.select()阻塞到至少有一个通道在你注册的事件上就绪了。

2.select(long timeout)和select()一样，除了最长会阻塞timeout毫秒(参数)。

3.selectNow()不会阻塞，不管什么通道就绪都立刻返回，此方法执行非阻塞的选择操作。

如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。

select()方法有返回值，返回的int值表示有多少通道已经就绪。亦即，自上次调用select()方法后有多少通道变成就绪状态。

如果调用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调

用select()方法，

如果另一个通道就绪了，它会再次返回1。

如果对第一个就绪的channel没有做任何操作，现在就有两个就绪的通道，

但在每次select()方法调用之间，只有一个通道就绪了。

选择器的selectedKeys()

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，

然后可以通过调用selector的selectedKeys()方法获取到：

```
Set selectedKeys = selector.selectedKeys();
```

当像Selector注册Channel时，Channel.register()方法会返回一个SelectionKey对象。

这个对象代表了注册到该Selector的通道。

可以通过SelectionKey的selectedKeySet()方法访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。如下：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // 一个连接被ServerSocketChannel接受
    } else if (key.isConnectable()) {
        // 与远程服务器建立了连接
    } else if (key.isReadable()) {
        // 一个channel做好了读准备
    } else if (key.isWritable()) {
        // 一个channel做好了写准备
    }
    keyIterator.remove();
}
```

这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。

注意每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。

必须在处理完通道时自己移除。下次该通道变成就绪时，Selector会再次将其放入已选择键集中。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，

如ServerSocketChannel或SocketChannel等。

选择器的wakeUp()

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select()方法返回。

只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。

阻塞在select()方法上的线程会立马返回。

如果有其它线程调用了wakeup()方法，但当前没有线程阻塞在select()方法上，上一个调用select()方法的阻塞线程会立即醒来（wake up）。

选择器的close()

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。

通道本身并不会关闭。

客户端与服务端简单交互实例

下面的程序涉及到一些网络编程的知识：

1） 服务器必须先建立ServerSocket或者ServerSocketChannel 来等待客户端的连接

2）客户端必须建立相对应的Socket或者SocketChannel来与服务器建立连接

3）服务器接受到客户端的连接受，再生成一个Socket或者SocketChannel与此客户端通信

服务器：

```
public static void main(String[] args) throws IOException {
    try {
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.socket().bind(new InetSocketAddress("127.0.0.1", 8000));
        ssc.configureBlocking(false);

        Selector selector = Selector.open();
        // 注册 channel，并且指定感兴趣的事件是 Accept
        ssc.register(selector, SelectionKey.OP_ACCEPT);

        ByteBuffer readBuff = ByteBuffer.allocate(1024);
        ByteBuffer writeBuff = ByteBuffer.allocate(128);
        writeBuff.put("server".getBytes());
        writeBuff.flip();

        while (true) {
            int nReady = selector.select(1000);
            if(nReady == 0) continue;
            Set<SelectionKey> keys = selector.selectedKeys();
```

```
            Iterator<SelectionKey> it = keys.iterator();
            while (it.hasNext()) {
                SelectionKey key = it.next();
         it.remove();
                if (key.isAcceptable()) {
                    // 创建新的连接，并且把连接注册到selector上，而且，
                    // 声明这个channel只对读操作感兴趣。
                    SocketChannel socketChannel = ssc.accept();
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector, SelectionKey.OP_READ);
                } else if (key.isReadable()) {
                    SocketChannel socketChannel = (SocketChannel)
key.channel();
                    readBuff.clear();
                    socketChannel.read(readBuff);

                    readBuff.flip();
                    System.out.println("服务器received : " + new
String(readBuff.array()));
                    key.interestOps(SelectionKey.OP_WRITE);
                } else if (key.isWritable()) {
                    writeBuff.rewind();
                    SocketChannel socketChannel = (SocketChannel)
key.channel();
                    socketChannel.write(writeBuff);
                    key.interestOps(SelectionKey.OP_READ);
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

客户器：

```
public static void main(String[] args) {
    try {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.connect(new InetSocketAddress("127.0.0.1", 8000));

        ByteBuffer writeBuffer = ByteBuffer.allocate(32);
        ByteBuffer readBuffer = ByteBuffer.allocate(32);

        writeBuffer.put("hello".getBytes());
```

```
            writeBuffer.flip();

            while (true) {
                writeBuffer.rewind();
                socketChannel.write(writeBuffer);
                readBuffer.clear();
                socketChannel.read(readBuffer);
                readBuffer.flip();
                System.out.println("客户端received : " + new
String(readBuffer.array()));
            }
        } catch (IOException e) {
        }
}
```

Scatter/Gather

Java NIO开始支持scatter/gather，scatter/gather用于描述从Channel中读取或者写入到Channel的操作。

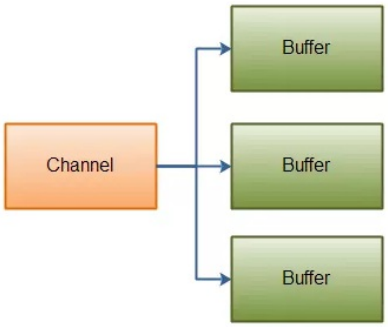
从Channel中分散（scatter）读取，是指在读操作时将读取的数据写入多个buffer中。因此，从Channel中读取的数据将“分散（scatter）”到多个Buffer中。

聚集（gather）写入一个Channel，是指在写操作时将多个buffer的数据写入同一个Channel，
因此，多个Buffer中的数据将“聚集（gather）”后写入到一个Channel。

scatter/gather经常用于需要将传输的数据分开处理的场合，例如传输一个由消息头和消息体组成的消息，
你可能会将消息体和消息头分散到不同的buffer中，这样你可以方便的处理消息头和消息体。

Scattering Reads

Scattering Reads是指数据从一个channel读取到多个buffer中。如下图描述：



```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { header, body };
channel.read(bufferArray);
```

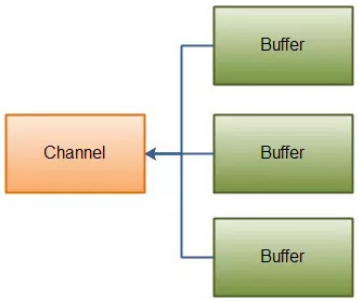
read()方法按照buffer在数组中的顺序将从channel中读取的数据写入到buffer，当一个buffer被写满后，channel紧接着向另一个buffer中写

Scattering Reads在移动下一个buffer前，必须填满当前的buffer，**这也意味着它不适用于*大小不固定*消息。**

换句话说，如果存在消息头和消息体，消息头必须完成填充（这里是128byte），Scattering Reads才能正常工作。

Gathering Writes

Gathering Writes是指数据从多个buffer写入到同一个channel。如下图描述：




```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);
//此处写数据到buffer中
ByteBuffer[] bufferArray = { header, body };
channel.write(bufferArray);
```

buffers数组是write()方法的输入参数，write()方法会按照buffer在数组中的顺序，将数据写入到channel，

注意只有position和limit之间的数据才会被写入。因此，如果一个buffer的容量为128byte，

但是仅仅包含58byte的数据，那么这58byte的数据将被写入到channel中。

因此与Scattering Reads相反，**Gathering Writes能较好的处理动态消息。**

通道之间的数据传输

在Java NIO中，如果两个通道中有一个是FileChannel，那你可以直接将数据从一个channel传输到另外一个channel。

transferFrom()

FileChannel的transferFrom()方法可以将数据从源通道传输到FileChannel中，下面是一个简单的例子：

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel  fromChannel = fromFile.getChannel();
RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel  toChannel = toFile.getChannel();
long position = 0;
long count = fromChannel.size();
toChannel.transferFrom(fromChannel, position, count);
```

方法的输入参数position表示从position处开始向目标文件写入数据，
count表示最多传输的字节数。

transferTo()

transferTo()方法将数据从FileChannel传输到其他的channel中。下面是一个简单的例子：

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
FileChannel  fromChannel = fromFile.getChannel();
RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
FileChannel  toChannel = toFile.getChannel();
long position = 0;
long count = fromChannel.size();
fromChannel.transferTo(position, count, toChannel);
```

是不是发现这个例子和前面那个例子特别相似？除了调用方法的FileChannel对象不一样外，其他的都一样。

字符集和Charset

1.基础知识

计算机中储存的信息都是用二进制数表示的；
而我们在屏幕上看到的英文、汉字等字符是二进制数转换之后的结果。
通俗的说，按照何种规则将字符存储在计算机中，如'a'用什么表示，称为"编码"；
反之，将存储在计算机中的二进制数解析显示出来，称为"解码"，如同密码学中的加密和解密。
在解码过程中，如果使用了错误的解码规则，则导致'a'解析成'b'或者乱码。

字符集（Charset）：多个字符的集合，字符集种类较多，每个字符集包含的字符个数不同，字符和二进制数字的对应规则不同

常见字符集名称：ASCII字符集、GB2312字符集、BIG5字符集、GBK字符集、UTF-8字符集等。

字符编码（Character Encoding）：是一套法则，使用该法则能够对自然语言的字符的一个集合（如字母表或音节表），

与其他东西的一个集合（如号码或电脉冲）进行配对。即在符号集合与数字系统之间建立对应关系，

它是信息处理的一项基本技术。通常人们用符号集合（一般情况下就是文字）来表达信息。

而以计算机为基础的信息处理系统则是利用元件（硬件）不同状态的组合来存储和处理信息的。

元件不同状态的组合能代表数字系统的数字，因此字符编码就是将符号转换为计算机可以接受的数字系统的数，

称为数字代码。

2.常用字符集和字符编码

2.1. ASCII字符集&编码

ASCII（American Standard Code for Information Interchange，美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统。

它主要用于显示现代英语。

2.2. GBXXXX字符集&编码

计算机发明之处及后面很长一段时间，只用应用于美国及西方一些发达国家，ASCII能够很好满足用户的需求。

但是当天朝也有了计算机之后，为了显示中文，必须设计一套编码规则用于将汉字转换为计算机可以接受的数字系统的数。

规定：一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字，

前面的一个字节（他称之为高字节）从0xA1用到 0xF7，后面一个字节（低字节）从0xA1到0xFE，

这样我们就可以组合出大约7000多个简体汉字了。

上面的编码规则形成的字符集就是GB2312。

GB2312或GB2312-80是中国国家标准简体中文字符集，全称《信息交换用汉字编码字符集·基本集》

由于GB 2312-80只收录6763个汉字，有不少汉字并未有收录在内，如部分在GB 2312-80推出以后才简化的汉字（如"啰"），

部分人名用字（如中国前总理朱镕基的"镕"字），台湾及香港使用的繁体字，日语及朝鲜语汉字等。

因此1995年发布了GBK编码，是在GB2312-80标准基础上的进行了扩展，共收录了21003个汉字

2.3. BIG5字符集&编码

Big5，又称为大五码或五大码，是使用繁体中文（正体中文）社区中最常用的电脑汉字字符集标准。

3.Unicode

像天朝一样，当计算机传到世界各个国家时，为了适合当地语言和字符，
设计和实现类似GB2312/GBK/BIG5的编码方案。这样各搞一套，在本地使用没有问题，
一旦出现在网络中，由于不兼容，互相访问就出现了乱码现象。

为了解决这个问题，产生了——Unicode。Unicode编码系统为表达任意语言的任意字符而设计。
它使用4字节的数字来表达每个字母、符号，或者表意文字(ideograph)。
每个数字代表唯一的至少在某种语言中使用的符号。
在计算机科学领域中，Unicode（统一码、万国码、单一码、标准万国码）是业界的一种标准，
它可以使电脑得以体现世界上数十种文字的系统。

Unicode是标准，是规则，
UTF-32/ UTF-16/ UTF-8是三种字符编码方案就是Unicode的具体实现。
UTF-8：电子邮件、网页及其他存储或传送文字的应用中，优先采用的编码方案.

Java默认使用Unicode字符集，但很多系统并不使用Unicode字符集，那么当从系统中读取数据到Java程序时，
就可能出现乱码的问题。

JDK1.4提供了Charset来处理字节序列和字符序列（字符串）之间的转换关系，
该类包含了用于创建解码器和编码器的方法，还提供了获取Charset所支持字符集的方法，Charset类是不可变的。
Charset类提供了一个静态方法availableCharset()来获取当前JDK所支持的所有字符集。
返回的是一个SortedMap<String,Charset>

```
// 获取Java支持的全部字符集
SortedMap<String,Charset>  map = Charset.availableCharsets();
for (String alias : map.keySet())
{
 // 输出字符集的别名
 System.out.println(alias);
```

```
}
```

一旦知道字符集的别名后，就可以用Charset的forName()方法来创建对应的Charset对象，forName方法的参数就是字符集的别名。
拿到Charset对象，就可以调用它的newEncoder和newDecoder方法，创建对应的编码器和解码器对象。

解码器对象CharsetDecoder里的decode()方法就可以将ByteBuffer转为CharBuffer，CharsetEncoder里的encode()方法作用相反，将CharBuffer转为ByteBuffer。

接下来使用编码器将CharBuffer中的字符序列转换为字节序列ByteBuffer。

CharBuffer和ByteBuffer是java NIO中的IO操作类。

```
// 创建简体中文对应的Charset
Charset charset = Charset.forName("GBK");
// 获取charset对象对应的编码器
CharsetEncoder charsetEncoder = charset.newEncoder();
// 创建一个CharBuffer对象
CharBuffer charBuffer = CharBuffer.allocate(20);
charBuffer.put("技术社区");
charBuffer.flip();
// 将CharBuffer中的字符序列转换成字节序列
ByteBuffer byteBuffer = charsetEncoder.encode(charBuffer);
// 循环访问ByteBuffer中的每个字节
for (int i = 0; i < byteBuffer.limit(); i++)
 {
     System.out.print(byteBuffer.get(i) + " ");
}
```

然后使用解码器将ByteBuffer中的字节序列转换为字符序列CharBuffer（字符串）。

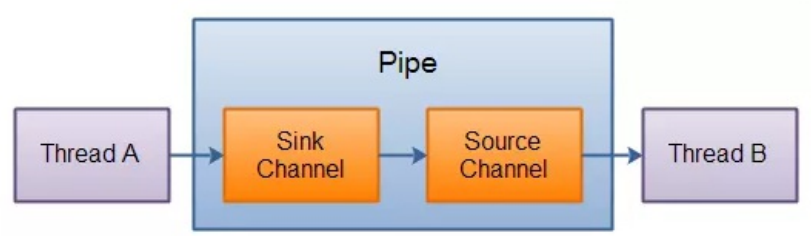
```
// 创建简体中文对应的Charset
Charset charset = Charset.forName("GBK");
// 获取charset对象对应的编码器
CharsetDecoder charsetDecoder = charset.newDecoder();
// 创建一个ByteBuffer对象
ByteBuffer byteBuffer = ByteBuffer.allocate(50);
byteBuffer.put("技术社区".getBytes("GBK"));
byteBuffer.flip();
// 将ByteBuffer的数据解码成字符序列
System.out.println(charsetDecoder.decode(byteBuffer));
```

Pipe

Java NIO `Pipe`是2个线程之间的单向数据连接。

Pipe有一个`source`通道和一个`sink`通道。

数据会被写到`sink`通道，从`source`通道读取。



创建管道

通过`Pipe.open()`方法打开管道。例如：

```
Pipe pipe = Pipe.open();
```

向管道写数据

要向管道写数据，需要访问sink通道。像这样：

```
Pipe.SinkChannel sinkChannel = pipe.sink();
```

通过调用`SinkChannel`的`write()`方法，将数据写入`SinkChannel`,像这样：

```
String newData = "New String to write to file..." +
System.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
while(buf.hasRemaining()) {
    sinkChannel.write(buf);
}
```

从管道读取数据

从读取管道的数据，需要访问`source`通道，像这样：

```
Pipe.SourceChannel sourceChannel = pipe.source();
```

调用`source`通道的`read()`方法来读取数据，像这样：

```
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
```

read()方法返回的int值会告诉我们多少字节被读进了缓冲区。