NIO new IO Non Blocking IO 非阻塞IO

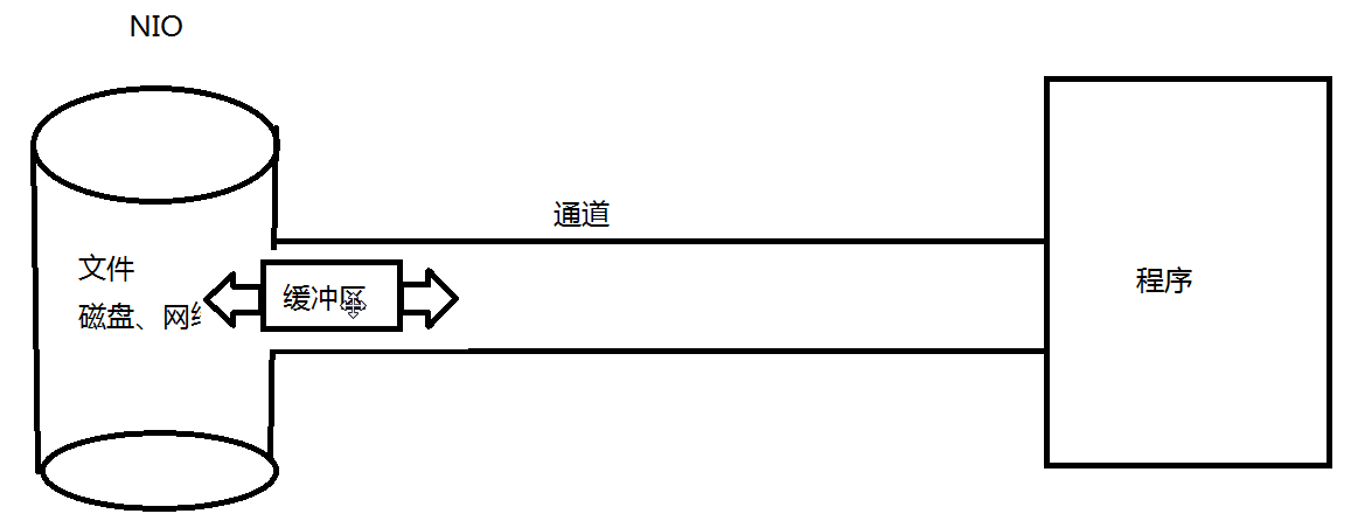
NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同。

NIO支持面向缓冲区的、基于通道的IO操作，NIO将以更加高效的方式进行问卷的读写操作。



通道(Channel)只负责建立连接，不存储数据。而缓冲区(Buffer)负责数据的存取。数据被存放在缓冲区中，利用管道来完成数据的传输。

管道相当于铁路线，而缓冲区相当于跑在铁路上的火车。



Buffer缓冲区底层用数组实现。

**java.nio.**

**java.nio.channels.**

**java.nio.charset.**

**Buffer缓冲区的核心属性**

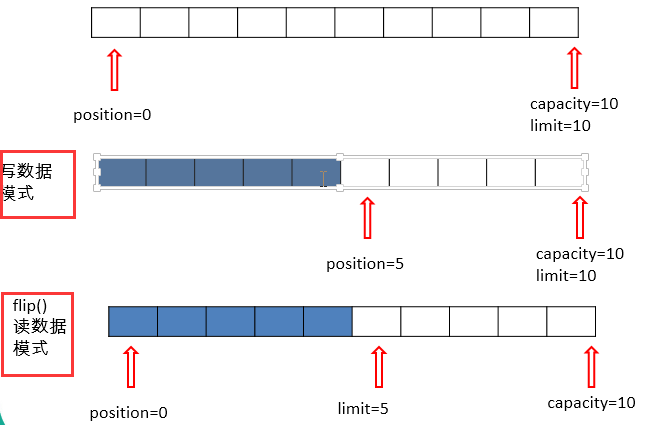
1. mark：标记，**记录**当前的position的位置，之后可以通过reset()将position恢复到mark的位置。
2. position：位置，表示缓冲区中**正在操作**的数据位置。类似读写指针，表示当前读(写)到什么位置。
3. limit：界限，表示缓冲区**可以操作**的数据的大小。(limit之后的数据不能读写)
4. capacity：容量，表示缓冲区中**最大存储**数据的容量，一旦声明，不可改变。在读写模式下都是固定的。

大小： 0<=mark<=position<=limit<=capacity

buffer中的flip方法涉及到bufer中的Capacity,Position和Limit三个概念。其中Capacity在读写模式下都是固定的，就是我们分配的缓冲大小,Position类似于读写指针，表示当前读(写)到什么位置,Limit在写模式下表示最多能写入多少数据，此时和Capacity相同，在读模式下表示最多能读多少数据，此时和缓存中的实际数据大小相同。在写模式下调用flip方法，那么limit就设置为了position当前的值(即当前写了多少数据),postion会被置为0，以表示读操作从缓存的头开始读。也就是说调用flip之后，读写指针指到缓存头部，并且设置了最多只能读出之前写入的数据长度(而不是整个缓存的容量大小)。

clear():清除此缓冲区。将position位置设置为 0，将限制设置为容量，并丢弃标记。不会删除缓冲区中的数据

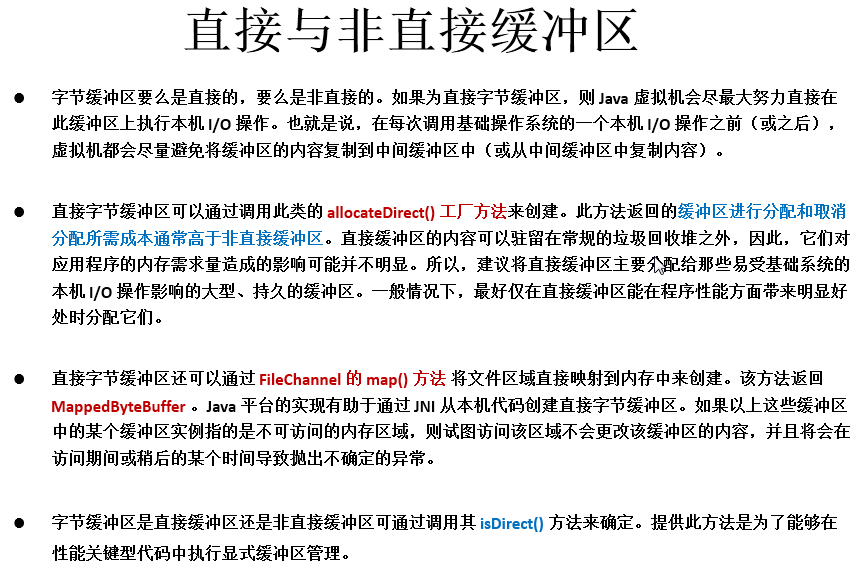
|  |  |
| --- | --- |
| [Buffer](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/Buffer.html) | [**reset**](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/Buffer.html#reset())()            将此缓冲区的位置重置为以前mark标记的位置。 |
| [Buffer](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/Buffer.html) | [flip](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/Buffer.html#flip())()            **在写模式下调用flip方法，那么limit就设置为了position当前的值(即当前写了多少数据),postion会被置为0，以表示读操作从缓存的头开始读** |

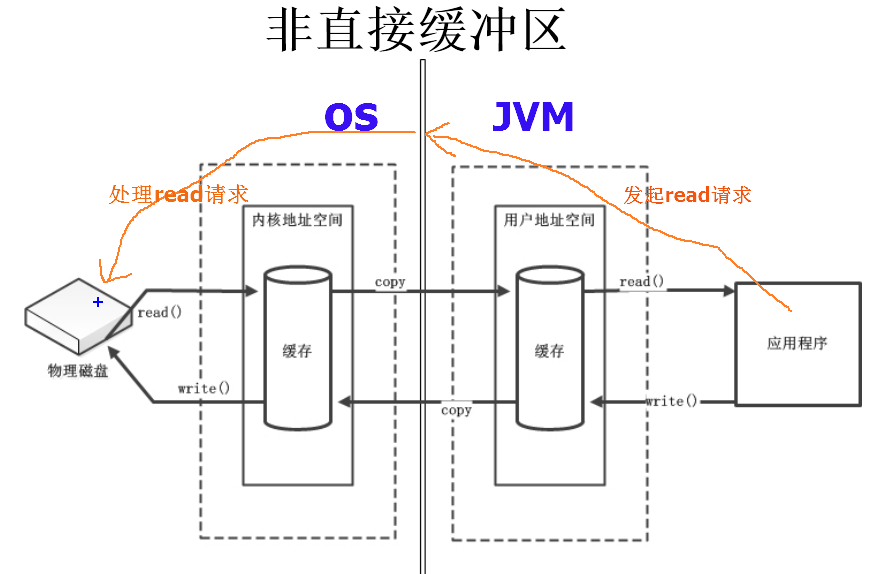


**直接缓冲区**与**非直接缓冲区**

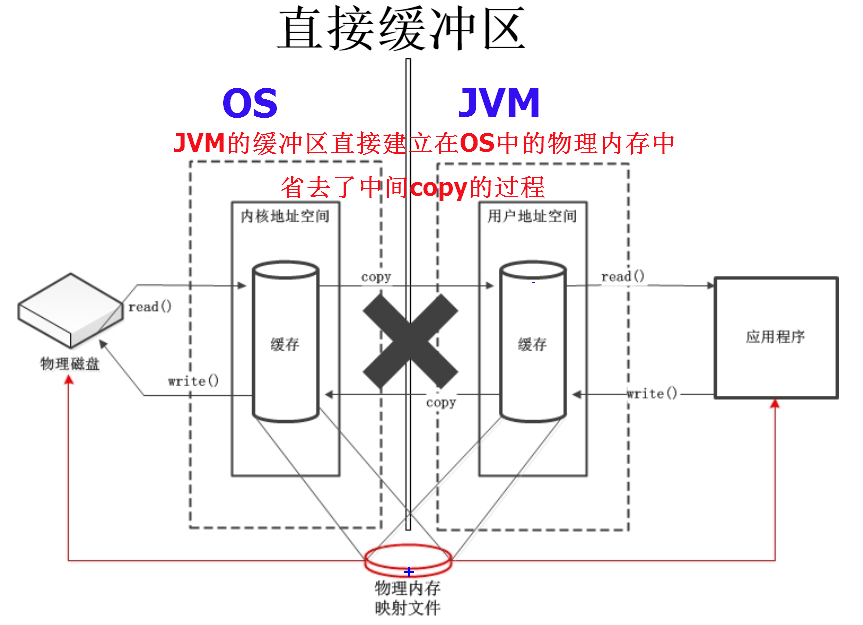
非直接缓冲区：通过allocate()方法分配缓冲区，将缓冲区建立在JVM 的内存中。

直接缓冲区：通过allocateDirect()方法分配缓冲区，将缓冲区建立在OS的物理内存中。





非直接缓冲区即JVM中的缓冲区，操作时需要经过OS缓冲区的再次拷贝，会增加时间消耗。 普通IO流即为非直接缓冲区读取



对于直接缓冲区，JVM直接将缓冲区建立在物理内存中。虽然文件的读写快速，但是JVM为此付出的代价是较高的开辟成本和清理成本，并且失去了对缓冲区的操作能力，转而完全交由OS管理。JVM的GC因此也很难对内存及时清理

**因此适合数据量很大同时在内存中存放时间长的数据。**

读写大文件时，直接缓冲区的读写速度比非直接缓冲区快十几甚至几十倍

**通道(Channel):**

通道用于源节点和目标节点的连接。在Java NIO中负责缓冲区中数据的传输。Channel中本身不存储数据，因此需要配合缓冲区(Buffer)进行传输。

主要实现类：

Java.nio.channels.Channel接口：

1. FileChannel 文件操作
2. SocketChannel 网络客户端TCP
3. ServerSocketChannel 网络服务端TCP
4. DatagramChannel 网络UDP

获取通道：

1. java针对支持通道的类提供了getChannel()方法
   1. 本地IO：FileInputStream、FileOutputStream、RandomAccessFile
   2. 网络IO：Socket、ServerSocket、DatagramSocket
2. JDK 1.7中的NIO.2中针对各个通道提供了静态方法open()
3. JDK 1.7中的NIO.2中的Files工具类的newByteChannel()

**通道间的数据传输：**

**使用该方法可以更加方便的实现文件的复制**

|  |  |
| --- | --- |
| abstract  long | [**transferFrom**](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/FileChannel.html#transferFrom(java.nio.channels.ReadableByteChannel, long, long))([ReadableByteChannel](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/ReadableByteChannel.html) src, long position, long count)            将字节从给定的可读取字节通道传输到此通道的文件中。 |
| abstract  long | [**transferTo**](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/FileChannel.html#transferTo(long, long, java.nio.channels.WritableByteChannel))(long position, long count, [WritableByteChannel](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/WritableByteChannel.html) target)            将字节从此通道的文件传输到给定的可写入字节通道。 |

**MappedByteBuffer:** **直接字节缓冲区**

其内容是文件的OS中的**内存映射区域**

|  |  |
| --- | --- |
| abstract  [MappedByteBuffer](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/MappedByteBuffer.html) | [**map**](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/FileChannel.html#map(java.nio.channels.FileChannel.MapMode, long, long))([FileChannel.MapMode](mk:@MSITStore:C:\Users\hasee\Desktop\JDK_API_1_6_zh_CN.CHM::/java/nio/channels/FileChannel.MapMode.html) mode, long position, long size)            将此通道的文件区域直接映射到内存中。 |

**分散(Scatter )**与**聚集(Gatter )**

由操作单一的缓冲区(Buffer)，变成操作缓冲区数组(Buffer[])

分散读取(Scattering Reads):将通道中的数据分散到多个缓冲区中。

聚集写入(Gathering Writes):将多个缓冲区中的数据聚集到通道中。



**将等待不同客户端线程阻塞的时间剥离到一个单独线程中去等待，用极少数的线程来 监控大量的Socket用户线程。从而提高运行的效率。NIO会在将数据准备好之后，再交由应用程序进行处理，数据的读取过程依然在应用线程中完成。**

阻塞与非阻塞只针对网络通信而言的 。

Java.nio.channels.Channel接口：

|--**SelectableChannel**

|--ServerSocketChannel

|--DatagramChannel

|--Pipe.SinkChannel

|--Pipe.SourceChannel

FileChannel没有继承SelectableChannel抽象类，因此不能实现非阻塞模式

传统的 IO 流都是阻塞式的。也就是说，**当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取或写入，该线程在此期间不能执行其他任务**。因此，在完成网络通信进行 IO 操作时，由于线程会阻塞，所以服务器端必须为每个客户端都提供一个独立的线程进行处理，当服务器端需要处理大量客户端时，性能急剧下降。

Java NIO 是非阻塞模式的。当线程从某通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。因此， NIO 可以让服务器端使用一个或有限几个线程来同时处理连接到服务器端的所有客户端。

**选择器Selector**

选择器是SelectableChannel的多路复用器。用于监控SelectableChannel的IO状况

Selector准备好数据以后，返回SelectionKey。SelectionKey表示一对Selector和Channel的关系，从SelectionKey中可以得到Channel数据(数据已准备好)并读取数据。在这个过程中服务端不需要再为某个客户端的延迟而等待。