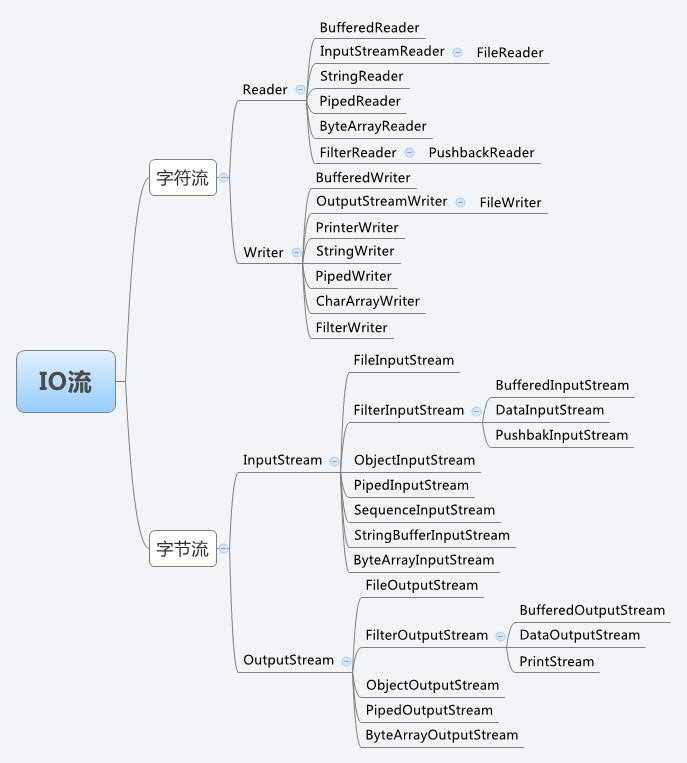
(1)stream 字节流

(2)reader writer 字符流

(3)Java输入输出的框架图(层次结构)：

(4)通道的使用方式：

第一步是获取通道。我们从 FileInputStream 获取通道：

|  |
| --- |
| FileInputStream fin = new FileInputStream( "readandshow.txt" );  FileChannel fc = fin.getChannel(); |

下一步是创建缓冲区：

|  |  |
| --- | --- |
| 1 | ByteBuffer buffer = ByteBuffer.allocate( 1024 ); |

最后，需要将数据从通道读到缓冲区中，如下所示：

|  |
| --- |
| fc.read( buffer ); |

您会注意到，我们不需要告诉通道要读 多少数据 到缓冲区中。每一个缓冲区都有复杂的内部统计机制，它会跟踪已经读了多少数据以及还有多少空间可以容纳更多的数据

写入文件

在 NIO 中写入文件类似于从文件中读取。首先从 FileOutputStream 获取一个通道：

|  |  |
| --- | --- |
| 1  2 | FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );  FileChannel fc = fout.getChannel(); |

下一步是创建一个缓冲区并在其中放入一些数据 - 在这里，数据将从一个名为 message 的数组中取出，这个数组包含字符串 "Some bytes" 的 ASCII 字节(本教程后面将会解释 buffer.flip() 和 buffer.put() 调用)。

|  |
| --- |
| ByteBuffer buffer = ByteBuffer.allocate( 1024 );  for (int i=0; i<message.length; ++i) {       buffer.put( message[i] );  }  buffer.flip(); |

最后一步是写入缓冲区中

|  |
| --- |
| fc.write( buffer ); |

ByteBuffer 类的 get() 和 put() 方法直接访问缓冲区中的数据

get() 方法

ByteBuffer 类中有四个 get() 方法：

byte get();

ByteBuffer get( byte dst[] );

ByteBuffer get( byte dst[], int offset, int length );

byte get( int index );

第一个方法获取单个字节。第二和第三个方法将一组字节读到一个数组中。第四个方法从缓冲区中的特定位置获取字节。那些返回ByteBuffer 的方法只是返回调用它们的缓冲区的 this 值。

此外，我们认为前三个 get() 方法是相对的，而最后一个方法是绝对的。 相对 意味着 get() 操作服从 limit 和 position 值 ― 更明确地说，字节是从当前 position 读取的，而 position 在 get 之后会增加。另一方面，一个 绝对 方法会忽略 limit 和 position 值，也不会影响它们。事实上，它完全绕过了缓冲区的统计方法。

上面列出的方法对应于 ByteBuffer 类。其他类有等价的 get() 方法，这些方法除了不是处理字节外，其它方面是是完全一样的，它们处理的是与该缓冲区类相适应的类型。

put()方法

ByteBuffer 类中有五个 put() 方法：

ByteBuffer put( byte b );

ByteBuffer put( byte src[] );

ByteBuffer put( byte src[], int offset, int length );

ByteBuffer put( ByteBuffer src );

ByteBuffer put( int index, byte b );

第一个方法 写入（put） 单个字节。第二和第三个方法写入来自一个数组的一组字节。第四个方法将数据从一个给定的源ByteBuffer 写入这个 ByteBuffer。第五个方法将字节写入缓冲区中特定的 位置 。那些返回 ByteBuffer 的方法只是返回调用它们的缓冲区的 this 值。

与 get() 方法一样，我们将把 put() 方法划分为 相对 或者 绝对 的。前四个方法是相对的，而第五个方法是绝对的。

上面显示的方法对应于 ByteBuffer 类。其他类有等价的 put() 方法，这些方法除了不是处理字节之外，其它方面是完全一样的。它们处理的是与该缓冲区类相适应的类型。

类型化的 get() 和 put() 方法

除了前些小节中描述的 get() 和 put() 方法， ByteBuffer 还有用于读写不同类型的值的其他方法，如下所示：

getByte()

getChar()

getShort()

getInt()

getLong()

getFloat()

getDouble()

putByte()

putChar()

putShort()

putInt()

putLong()

putFloat()

putDouble()

chanel复制文件的实际操作：

下面的内部循环概括了使用缓冲区将数据从输入通道拷贝到输出通道的过程。

|  |
| --- |
| while (true) {       buffer.clear();       int r = fcin.read( buffer );       if (r==-1) {         break;       }       buffer.flip();       fcout.write( buffer );  } |

子缓冲区的创建及作用：

|  |
| --- |
| ByteBuffer buffer = ByteBuffer.allocate( 10 ) |

然后使用数据来填充这个缓冲区，在第 n 个槽中放入数字 n：

|  |
| --- |
| for (int i=0; i<buffer.capacity(); ++i) {       buffer.put( (byte)i );  } |

现在我们对这个缓冲区 分片 ，以创建一个包含槽 3 到槽 6 的子缓冲区。在某种意义上，子缓冲区就像原来的缓冲区中的一个 窗口 。

窗口的起始和结束位置通过设置 position 和 limit 值来指定，然后调用 Buffer 的 slice() 方法：

|  |
| --- |
| buffer.position( 3 );  buffer.limit( 7 );  ByteBuffer slice = buffer.slice(); //创建子缓冲区 |

对子缓冲区的修改会同时作用域夫缓冲区(数据是共享的)

创建一个只读缓冲区(是原缓冲区的一个副本)

Buffer. asReadOnlyBuffer() 返回一个缓冲区对象，与夫缓冲区共享数据。

直接缓冲区：与具体的实现有关(Java平台的直接诶分配内存有关)--🡪作用：减少中间缓冲区的复制等耗时操作

内存映射文件I/O (重点就是速度快)

风险：数据的修改和保存时不分开的

创建方式：

是channel中的方法：

MappedByteBuffer mbb = fc.map( FileChannel.MapMode.READ\_WRITE, 0, 1024 );

分散聚集I/O ( 感觉很实用)

作用：使用多个不同的缓冲区，当一个缓冲区满时，会写入另一个缓冲区

应用：在实现一个网络消息系统的时候，可以设计一个缓冲区刚刚可以容纳消息头，而另一个缓冲区刚刚好容纳消息的正文。

( 分散读取，聚集写入)

例子：UseScatterGather.java

文件锁：

|  |
| --- |
| RandomAccessFile raf = new RandomAccessFile( "usefilelocks.txt", "rw" );  FileChannel fc = raf.getChannel();  FileLock lock = fc.lock( start, end, false ); |

在拥有锁之后，您可以执行需要的任何敏感操作，然后再释放锁：

|  |
| --- |
| lock.release(); |

连网和异步 I/O

参考同文件夹下的NIO实现

字符集Charset ：



Sun的定义：一个 Charset 是“十六位 Unicode 字符序列与字节序列之间的一个命名的映射

Charset charsets=StandardCharsets.US\_ASCII; 获得相应的字符集

Wrap()包装方法：可以将一个str包装成buffer对象

在打开相应的文件、将输入数据读入名为 inputData 的 ByteBuffer 之后，我们的程序必须创建 ISO-8859-1 (Latin1) 字符集的一个实例：

|  |
| --- |
| （1）Charset latin1 = Charset.forName( "ISO-8859-1" );  （2）Charset charsets = StandardCharsets.US\_ASCII; |

然后，创建一个解码器（用于读取）和一个编码器 （用于写入）：

|  |
| --- |
| CharsetDecoder decoder = latin1.newDecoder();  CharsetEncoder encoder = latin1.newEncoder(); |

　为了将字节数据解码为一组字符，我们把 ByteBuffer 传递给 CharsetDecoder，结果得到一个 CharBuffer

|  |
| --- |
| CharBuffer cb = decoder.decode( inputData ); |

　如果想要处理字符，我们可以在程序的此处进行。但是我们只想无改变地将它写回，所以没有什么要做的。

要写回数据，我们必须使用 CharsetEncoder 将它转换回字节：

|  |
| --- |
| ByteBuffer outputData = encoder.encode( cb ); |

使用Charset进行字符的过滤（原理：当遇到无法解析的字符是，设置为忽略）：

Encoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

Character也可以进行字符底层的操作

在NIO中存在大段小段的问题