Java NIO API 介绍

# 介绍

java1.4引入了非阻塞的通信机制，可用于：

服务器程序接收客户连接

# 线程阻塞的原因

1.

当时的撒

## package

java.nio 定义了Buffer及其数据类型相关的子类。

有Buffer；DirectBuffer；ByteOrder；Exception；其中ByteBuffer的作用非常重要。

java.nio.channels 定义了一系列处理IO的Channel接口以及这些接口在文件系统和网络通讯上的实现。通过Selector这个类，还提供了进行异步IO操作的办法。这个包可以说是NIO API的核心。

java.nio.channels.spi 实现channel和selector API的抽象类。

java.nio.charset 定义了处理字符编码和解码的类。

java.nio.charset.spi 定义了可用来实现charset API的抽象类。

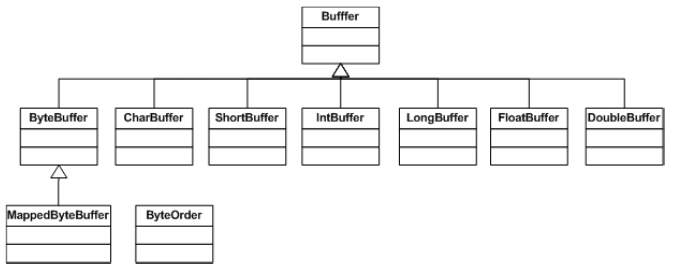
java.nio.channels.spi和java.nio.charset.spi这两个包主要被用来对现有NIO API进行扩展，在实际的使用中，我们一般只和另外的3个包打交道。下面将对这3个包一一介绍。

## java.nio

这个包主要定义了Buffer及其子类。Buffer定义了一个线性存放primitive type数据的容器接口。对于除boolean以外的其他primitive type，都有一个相应的Buffer子类，ByteBuffer是其中最重要的一个子类。

## nio class UML

下面这张UML类图描述了java.nio中的类的关系：



## Buffer

primitive type数据的容器接口。Buffer及其子类都不是线程安全的。

**属性：**

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

### capacity()

这个Buffer最多能放多少数据。capacity一般在buffer被创建的时候指定。

缓冲区中的最大数据容量，实际上，它指定了底层数组的大小。一旦确定不再改变

### limit()

写时：还有多少空间可以放入数据

读时：还有多少数据需要取出

在Buffer上进行的读写操作都不能越过这个下标。当写数据到buffer中时，limit一般和capacity相等，当读数据时，limit代表buffer中有效数据的长度。

position ：读/写操作的当前下标。当使用buffer的相对位置进行读/写操作时，读/写会从这个下标进行，并在操作完成后，buffer会更新下标的值。

### position()

写时：指定了下一个字节存放的位置：如果您写三个字节到缓冲区，那么缓冲区的 position 将会设置为3，指向数组中第四个元素。

读时：指定下一个字节来自数组的哪一个元素。因此您从缓冲区读了5个字节，那么缓冲区的 position 将被设置为5，指向数组的第六个元素。

### mark()

一个临时下标。调用mark()会将mark设为当前的position的值，调用reset()会将position属性设置为mark的值。mark的值总是小于等于position的值，如果将position的值设的比mark小，当前的mark值会被抛弃掉。

### clear()

设置limit=capacity，position=0，丢弃mark，一般在把数据写入Buffer前调用。

### flip()

设置limit=position，position=0，丢弃mark，一般在从Buffer读出数据前调用。

### rewind()

设置position=0，limit不变，丢弃mark，一般在把数据重写入Buffer前调用。

### isReadOnly()

判断一个Buffer是否只读，对只读buffer写会抛出ReadOnlyBufferException

### limit()

设置limit

### get(..?)

1.byte get(); //获取指定位置的单个字节

2.ByteBuffer get( byte dst[] ); //获取当前位置单个字节，并position++

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

4.byte get( int index );

### put(..?)

1.ByteBuffer put( byte b );

2.ByteBuffer put( byte src[] );

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

4.ByteBuffer put( ByteBuffer src );

5.ByteBuffer put( int index, byte b );

### getXXX()

getByte()

getChar()

getShort()

getInt()

getLong()

getFloat()

getDouble()

putByte()

putChar()

putShort()

putInt()

putLong()

putFloat()

putDouble()

## ByteBuffer

在Buffer的子类中，ByteBuffer是一个地位较为特殊的类，因为在java.io.channels中定义的各种channel的IO操作基本上都是围绕ByteBuffer展开的。

ByteBuffer定义了4个static方法来做创建工作：

### allocate(…)

ByteBuffer allocate(int capacity) ：创建一个指定capacity的ByteBuffer。

### allocateDirect(…)

ByteBuffer allocateDirect(int capacity) ：创建一个direct的ByteBuffer，这样的ByteBuffer在参与IO操作时性能会更好（很有可能是在底层的实现使用了DMA技术），相应的，创建和回收direct的ByteBuffer的代价也会高一些。isDirect()方法可以检查一个buffer是否是direct的。 只有bytebuffer才能创建直接缓冲区

ByteBuffer wrap(byte [] array)

ByteBuffer wrap(byte [] array, int offset, int length) 把一个byte数组或byte数组的一部分包装成ByteBuffer。

ByteBuffer定义了一系列get和put操作来从中读写byte数据，如下面几个：

byte get() ByteBuffer put(byte b)

ByteBuffer get(byte [] dst) ByteBuffer put(byte [] src)

byte get(int index) ByteBuffer put(int index, byte b)

这些操作可分为绝对定位和相对定为两种，相对定位的读写操作依靠position来定位Buffer中的位置，并在操作完成后会更新position的值。

在其它类型的buffer中，也定义了相同的函数来读写数据，唯一不同的就是一些参数和返回值的类型。

除了读写byte类型数据的函数，ByteBuffer的一个特别之处是它还定义了读写其它primitive数据的方法，如：

int getInt() ；putInt(int value)

读写其它类型的数据牵涉到字节序问题，ByteBuffer会按其字节序（大字节序或小字节序）写入或读出一个其它类型的数据。字节序可以用order方法来取得和设置：

ByteOrder order() ：返回ByteBuffer的字节序。

ByteBuffer order(ByteOrder bo) ：设置ByteBuffer的字节序。

ByteBuffer另一个特别的地方是可以在它的基础上得到其它类型的buffer。如：

CharBuffer asCharBuffer()

为当前的ByteBuffer创建一个CharBuffer的视图。在该视图buffer中的读写操作会按照ByteBuffer的字节序作用到ByteBuffer中的数据上。 用这类方法创建出来的buffer会从ByteBuffer的position位置开始到limit位置结束，可以看作是这段数据的视图。视图buffer的readOnly属性和direct属性与ByteBuffer的一致，而且也只有通过这种方法，才可以得到其他数据类型的direct buffer。

## ByteOrder：

用来表示ByteBuffer字节序的类，可将其看成java中的enum类型。主要定义了下面几个static方法和属性：

ByteOrder.*BIG\_ENDIAN*;// 代表大字节序的ByteOrder

ByteOrder.*LITTLE\_ENDIAN*;// 代表小字节序的ByteOrder

ByteOrder.*nativeOrder*();//返回当前硬件平台的字节序

MappedByteBuffer :

ByteBuffer的子类，是文件内容在内存中的映射。这个类的实例需要通过FileChannel的map()方法来创建。

## ByteBuffer例子

接下来看看一个使用ByteBuffer的例子，这个例子从标准输入不停地读入字符，当读满一行后，将收集的字符写到标准输出：

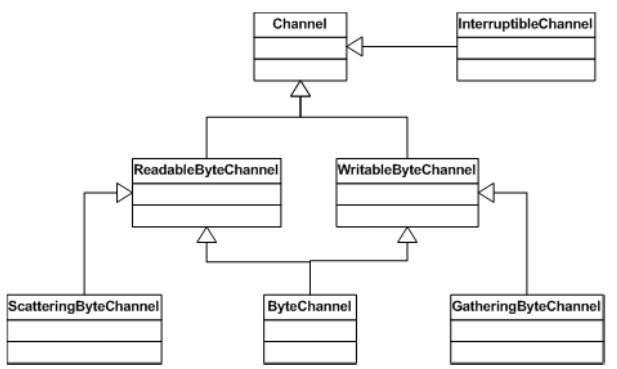
|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  // 创建一个capacity为256的ByteBuffer  ByteBuffer buf = ByteBuffer.*allocate*(256);  **while** (**true**) {  // 从标准输入流读入一个字符  **int** c = System.*in*.read();  // 当读到输入流结束时，退出循环  **if** (c == -1)  **break**;  // 把读入的字符写入ByteBuffer中  buf.put((**byte**) c);  // 当读完一行时，输出收集的字符  **if** (c == '\n') {  // 调用flip()使limit变为当前的position的值,position变为0,  // 为接下来从ByteBuffer读取做准备  buf.flip();  // 构建一个byte数组  **byte**[] content = **new** **byte**[buf.limit()];  // 从ByteBuffer中读取数据到byte数组中  buf.get(content);  // 把byte数组的内容写到标准输出  System.*out*.print(**new** String(content));  // 调用clear()使position变为0,limit变为capacity的值，  // 为接下来写入数据到ByteBuffer中做准备  buf.clear();  }  }  } |

## java.nio.channels

Channel表现一个可以进行IO操作的通道；包含文件系统和网络通讯相关的Channel类；

通过selector， SelectableChannel这两个类定义了一个进行异步(non-blo)IO的操作API

## channels interface UML



## Channel

**channel接口方法：**

void close()

boolean isOpen()

**ReadableByteChannel接口方法**

int read(ByteBuffer) //从channel中读取byte数据写到buffer中

**WritableByteChannel接口方法**

int write(ByteBuffer) //从buffer中读取byte数据写到channel中

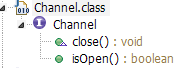
InterruptibleChannel

void close()

1当close方法被调用时，其他block在该对象的IO上的线程会接受到AsynchronousCloseException.

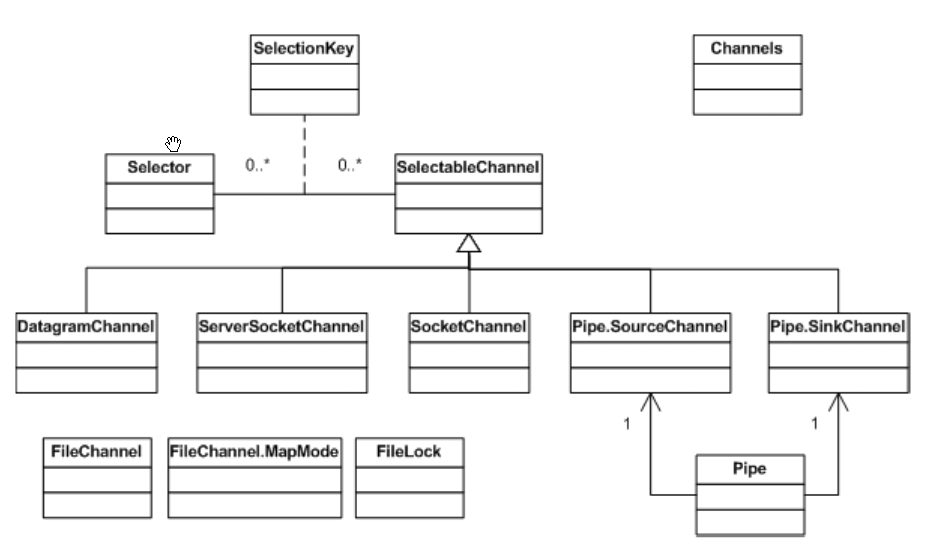
2当一个线程block在InterruptibleChannel的IO操作上时，另一个线程调用该线程的interrupt()方法会导致channel被关闭，该线程收到一个ClosedByInterruptException，同时线程的interrupt状态会被设置。

----------------------------------------------------------------------------------------------------------



## Channels class UML



# NIO

异步IO允许应用程序同时监控多个channel以提高性能，这一功能通过一下3个类来实现：

Selector，SelectableChannel，SelectionKey

SelectableChannel：

Selector.select()给应用程序提供了一个同时监控多个IOchannel的途径。

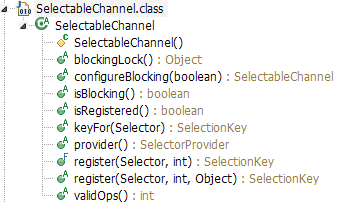
应用程序通过Selector.select()监控注册在Selector上的多个SelectableChannel，当由Channel的IO操作可以进行的时候，select ()方法就会返回让应用程序检查channel的状态，并做相应处理。

|  |
| --- |
| **public** **static** **void** acceptConnections(**int** port) **throws** IOException{  Selector acceptSelector = SelectorProvider.*provider*().openSelector();    ServerSocketChannel ssc = ServerSocketChannel.*open*();  ssc.configureBlocking(**false**);    InetAddress ia = InetAddress.*getLocalHost*();  InetSocketAddress isa = **new** InetSocketAddress(ia,port);  ssc.socket().bind(isa);    ssc.register(acceptSelector,SelectionKey.*OP\_ACCEPT*);    **while**(acceptSelector.select() > 0){  Set<SelectionKey> readyKeys = acceptSelector.selectedKeys();  Iterator<SelectionKey> i = readyKeys.iterator();  **while**(i.hasNext()){  SelectionKey sk = i.next();  i.remove();  ServerSocketChannel nextReay = (ServerSocketChannel)sk.channel();  Socket socket = nextReay.accept().socket();  PrintWriter out = **new** PrintWriter(socket.getOutputStream(),**true**);  out.print(**new** Date());  out.close();  }  }    } |

## SelectableChannel

异步IO操作的顶层抽象类,可以注册到一个或多个Selector上以进行异步IO操作。

方法



isBlocking()

configureBlocking(boolean block)

所有channel创建的时候都是blocking模式，只有non-blocking模式才能参与异步IO

register(Selector sel，int ops，attachment)

将当前channel注册到一个Selector上，并返回对应的SelectionKey，ops代表了需要监控的IO操作。

validOps()

返回bit mask的4种操作，定义在SelectionKey中

SelectKey keyFor(Selector seletor)

返回该cannel在selector上的注册关系对应的selectKey，无注册关系，返回null；

## Selector

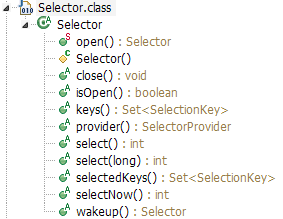
同时监控多个SelectableChannel的IO状况，是异步IO的核心。

Selector有3个SelectionKey的集合：

key Set ：所有注册的channel，通过keys()方法返回

Selected key set ：所有可IO操作的Channel，通过selectedKeys()返回

Cancelled key set ：所有已经cancel了注册关系的Channel，无法直接访问。



Selector open()

创建selector实例，内部通过SelectorProvider.provider().openSelector()实现。

### int select()

阻塞，监控所有注册的channel，当有可IO操作的channel时返回，并将对应的SelectionKey加入selected-key set中。

### int select(long timeout)

设置超时的select()操作；timeout毫秒数，超过该时间返回。

### int selectNow()

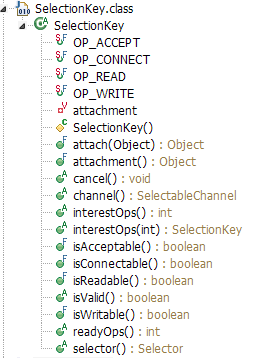
立即返回的select()操作；

Selector wakeup()

使一个还未返回的select()操作立即返回。

## SelectionKey

代表了Selector和SelectableChannel的注册关系。



定义了以下4中IO操作，且可以位操作组合成一个bit mask：

**public** **static** **final** **int** OP\_ACCEPT = 1 << 4;

**public** **static** **final** **int** OP\_CONNECT = 1 << 3;

**public** **static** **final** **int** OP\_WRITE = 1 << 2;

**public** **static** **final** **int** OP\_READ = 1 << 0;

OP\_ACCEPT:有新的网络连接可以accept，ServerSocketChannel支持这一异步IO。

OP\_CONNECT：代表连接已经建立或出错，SocketChannel支持这一异步IO。

OP\_WRITE，OP\_READ：代表了读写。

Object attachment()

//返回SelectionKey的attachment，可以在注册的时候制定

Object attach(Object obj)

//设置SelectionKey的attachment

SelectableChannel channel()

//返回SelectionKey对应的Channel

Selector selector()

//返回SelectionKey对应的Selector

void cannel()

//注销这个SelectionKey所对应的注册关系

int interestOps()

//返回需要Selector监控的IO操作的bit mask

SelectionKey interestOps(int opt)

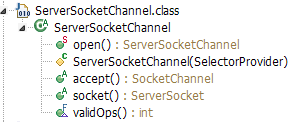
//设置interestOps

int readyOps()

//返回一个bit mask，代表在相应channe上可以进行的IO操作。

## ServerSocketChannel

支持异步操作，对应于java.net.ServerSocket类，支持OP\_ACCEPT操作。



socket()

返回对应的ServerSocket

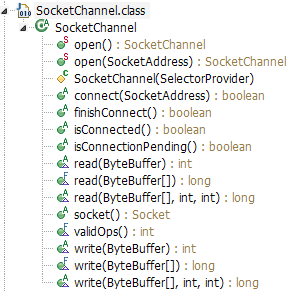
accept()

接受一个连接，返回代表这个连接的SocketChannel对象。

## SocketChannel

支持异步操作，对应于java.net.Socket类，支持OP\_CONNECT,OP\_READ,OP\_WRITE操作。

这个类还实现了ByteChannel，ScantteringByteChannel，GatheringByteChannel接口



Socket socket ()

//返回对应的socket

boolean connect(SocketAddress remote)

boolean finishConnect()

connect进行一个连接操作，

如果当前SocketChannel是blocking模式，需要等到连接完成或错误才返回，

如果当前SocketChannel是non-blocking模式，函数在连接能立刻被建立时返回true，否则返回false，应用程序需要在以后用finashConnect()方法来完成连接操作。

## DatagramChannel

和SocketChannel类似，对应于java.net.DatagramSockte,提供UDP协议的IO接口。

## Pipe

包含了一个读一个写的channel(Pipe.SourceChannel和Pipe.SinkChannel),这对channel可用于进程中的通讯。

## FileChannel

用于对文件的读，写，映射，锁定等操作。**不支持异步操作**。

和映射操作相关的类FileCannel.MapMode

和锁定操作相关的类FileLock

## FileLock

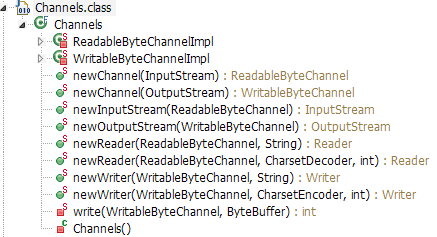
|  |
| --- |
| FileChannel fc = output.getChannel();  // FileLock lock = fc.lock( start,end,false ); //共享锁，不阻塞共享锁的获取  FileLock lock = fc.tryLock();//排它锁，阻塞所有锁的获取  if (lock != null) {  //...  lock.release();  }  fout.close(); |

## MappedByteBuffer

mappedByteBuffer = fileChannel .map(FileChannel.MapMode.READ\_ONLY, begin, length);

## Channels

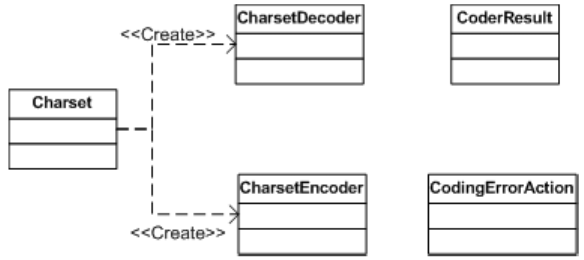
工具类，支持stream类和channel类之间的互操作。



## CharSet

### java.nio.charset

这个包定义了Charset及相应的encoder，decoder



代表了一个字符集，提供factory method来构建相应的CharsetDecoder可CharsetEncoder

SortedMap<String,Charset> availableCharsets()

返回当前系统支持的所有Carset对象，key是charset名字

boolean *isSupported*(charsetName);

判断该名字的字符集是否被当前系统支持。

Charset forNames(String carsetname)

返回该名字对应的Charset对象。

String name()

返回该字符集的规范名

Set aliases()

返回该字符集的所有别名

CharsetDecoder newDecoder()

创建一个对应于这个Charset的decoder

CharsetEncoder newEncoder()

创建一个对应于这个Charset的encoder

### 使用示例

|  |
| --- |
| Charset latin1 = Charset.forName("GBK");  CharsetEncoder encoder = latin1.newEncoder();  CharsetDecoder decoder = latin1.newDecoder();  CharBuffer cb = CharBuffer.wrap("你好。");  ByteBuffer buf = encoder.encode(cb);  System.out.println(decoder.decode(buf)); |

### CharsetDecoder

将某种字符集编码的字节流解码为unicode。

输入是ByteBuffer，输出CharBuffer

步骤：

1 调用CharsetDecoder的reset()方法，第一次可以跳过；

2 调用decode()中间过程，设置endOfInput参数为false，告诉decoder有可能还有新的数据输入。

3 调用decode()方法最后一次，设置endOfInput参数为true，告诉decoder所有数据已经送入。

4 调用decoder的flush()，让decoder有机会把一些内部状态写到CharBuffer中。

方法：

CharsetDecoder **reset**(){…; return this}

重置decoder，并清除decoder中的一些内部状态。

CoderResult **decode**(ByteBuffer in, CharBuffer out, boolean endOfInput)

从ButeBuffer类型的输入中decode尽可能多的字节到CharBuffer，根据decode的结果可能返回3中CoderResult

CoderResult.UNDERFLOW表示已经没有输入可以decode；

CoderResult.OVERFLOW表示输出已满；

CoderResult.XXX其他结果表示decode过程中有错误发生，根据返回结果可以采取相应的措施，比如增加输入，清楚输出等然后再次调用decode()方法。

CoderResult **flush**(CharBuffer out)

有些decoder在decode的过程中保留一些内部状态，调用这个方法让这些decoder有机会将这些内部状态写到输出的CharBuffer中。

调用成功返回CoderResult.UNDERFLOW，若返回CoderResult.OVERFLOW表示输出已满，应扩大输出CharBuffer的空间再次调用该方法。

CharBuffer **decode**(ButeBuffer in)

快捷的方法，把ByteBuffer中的内容decode到一个新创建的CharBuffer中，包含了以上4的步骤，所以不能和前面3个方法一期使用。

Decode过程的错误有两种：

malformed-input CoderResult表示输入数据有误：

unmappable-character CoderResult表示无法被解码成unicode字符

如何处理这些错误，取决于decoder的设置。可以通过CodingErrorAction设置成：

1. 忽略错误
2. 报告错误，decode()返回表示错误的CoderResult
3. 替换错误，用decoder中的替换字串替换掉有错误的部分。

CodingErrorAction **malformedInputAction**()

CharsetDecoder **onMalformedInput**(CodingErrorAction newAction)

返回/设置malformed-input的出错处理。

CodingErrorAction **unmappableCharacterAction**()

CharsetDecoder **onUnmappableCharacter**(CodingErrorAction newAction)

返回/设置unmappable-character的出错处理。

String **replacement**()

CharsetDecoder **replaceWith**(String newReplacement)

返回/设置的替换字串

### CharsetEncoder

将unicode字符数据编码为特定字符集的字节流，接口和CharsetDecoder相似。

## CoderResult

描述decode/encode操作结果的类，有两个static成员

CoderResult OVERFLOW //输出已满

CoderResult UNDERFLOW //输入已无数据可用

方法：用于判断CoderResult描述的错误

boolean isError()

boolean ismalformed()

boolean isUnmappable()

boolean isOverflow()

boolean isunderflow()

int length() //返回错误的长度，比如无法转换成unicode的字节长度。

void throwException() //抛出一个和该CoderResult相对应的异常。

## CodingErrorAction

表示decoder/encoder中错误处理方法类，可看成enum类型，

Static属性

CodingErrorAction IGNORE 忽略错误

CodingErrorAction REPLACE 替换错误

CodingErrorAction REPORT 报告错误，对于不同的函数，有可能是返回一个和错误有关的CoderResult，也有可能是抛出一个CharacterCodingException

# 通道和缓冲区

Channel, Buffer

Channel是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流；

通道是双向的，流是单向的；

所有数据都通过 Buffer 对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您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。

## Buffer

数据的容器接口。Buffer及其子类都不是线程安全的。

**属性：**

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

### capacity

这个Buffer最多能放多少数据。capacity一般在buffer被创建的时候指定。

缓冲区中的最大数据容量，实际上，它指定了底层数组的大小。

### limit

写时：还有多少空间可以放入数据

读时：还有多少数据需要取出

在Buffer上进行的读写操作都不能越过这个下标。当写数据到buffer中时，limit一般和capacity相等，当读数据时，limit代表buffer中有效数据的长度。

position ：读/写操作的当前下标。当使用buffer的相对位置进行读/写操作时，读/写会从这个下标进行，并在操作完成后，buffer会更新下标的值。

### position

写时：指定了下一个字节存放的位置：如果您写三个字节到缓冲区，那么缓冲区的 position 将会设置为3，指向数组中第四个元素。

读时：指定下一个字节来自数组的哪一个元素。因此您从缓冲区读了5个字节，那么缓冲区的 position 将被设置为5，指向数组的第六个元素。

### mark

一个临时下标。调用mark()会将mark设为当前的position的值，调用reset()会将position属性设置为mark的值。mark的值总是小于等于position的值，如果将position的值设的比mark小，当前的mark值会被抛弃掉。

### clear()

设置limit=capacity，position=0，一般在把数据写入Buffer前调用。

### flip()

设置limit=position，position=0，一般在从Buffer读出数据前调用。

### rewind()

设置position=0，limit不变，一般在把数据重写入Buffer前调用。

### isReadOnly()

判断一个Buffer是否只读，对只读buffer写会抛出ReadOnlyBufferException

### ByteBuffer动态图例

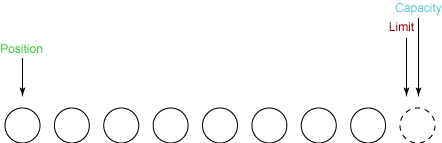
#### 1申请buffer

申请一个容量=8字节的ByteBuffer

ByteBuffer buffer = ByteBuffer.allocate(8);

[pos=0 lim=8 cap=8]

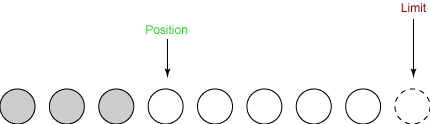
limit和capacity指向数组的尾部之后(如果有第8个槽，则是第8个槽所在的位置)



#### 2 写入3个字节

buffer.put(new byte[]{1,2,3});

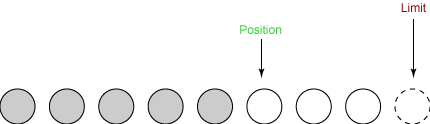
[pos=3 lim=8 cap=8]



#### 3 再写入2个字节

buffer.put(new byte[]{4,5});

[pos=5 lim=8 cap=8]

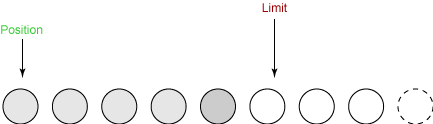


#### 4 flip()

使用之前要做flip()

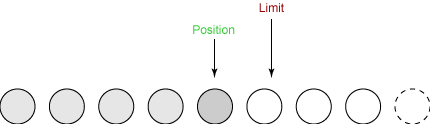
buffer.flip();

[pos=0 lim=5 cap=8]



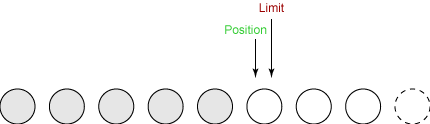
#### 5 读取4个字节

还剩下一个字节



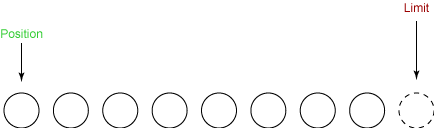
#### 6 读取剩下全部字节

读取一个字节



#### 6 clear()

准备再次写入数据



## 缓冲区分片

buffer.position(3)

buffer.limit(7)

ButeBuffer slice = buffer.slice();

分片缓冲区的作用：如果要对谋片区域修改，可以写一个队整个缓冲区作用的函数去操作片

## 只读缓冲区

asReadOnlyBuffer()

直接缓冲区

间接缓冲区

## 内存映射文件

将一个 FileChannel (它的全部或者部分)映射到内存中。为此我们将使用 FileChannel.map() 方法。下面代码行将文件的前 1024 个字节映射到内存中：

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

## 分散和聚集

ScatteringByteChannel

GatheringByteChannel

long read( ByteBuffer[] dsts );

long read( ByteBuffer[] dsts, int offset, int length );

通道依次填充每个缓冲区。填满一个缓冲区后，它就开始填充下一个。在某种意义上，缓冲区数组就像一个大缓冲区。

# 示例1-文件拷贝

|  |
| --- |
| **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.FileChannel;  /\*\*  \* 拷贝文件  \* **@author** cheng.huang  \*  \*/  **public** **class** Demo02CopyFile {  **public** **static** **void** main(String[] args) **throws** Exception {  String fileIn = "c:\\channel.txt";  String fileOut = "c:\\target.txt";  **int** totalIn = 0;  **int** totalOut = 0;  **int** count = 0;  **int** index = 0;    //1 获取通道  FileInputStream fis = **new** FileInputStream(fileIn);  FileChannel inChannel = fis.getChannel();    FileOutputStream fos = **new** FileOutputStream(fileOut);  FileChannel outChannel = fos.getChannel();    //2 创建缓冲区  ByteBuffer buffer = ByteBuffer.*allocate*(32);  **while**(**true**){  System.*out*.println("====>index="+(index++));  buffer.clear();  count = inChannel.read(buffer);  System.*out*.println("i count="+(count));  **if**(count == -1){  **break**;  }  totalIn+= count;    buffer.flip();  count = outChannel.write(buffer);  System.*out*.println("o count="+(count));  totalOut+=count;  }    System.*out*.println("totalIn="+(totalIn));  System.*out*.println("totalOut="+(totalOut));  }  } |

# AIO

AsynchronousServerSocketChannel server

= AsynchronousServerSocketChannel.open().bind(new InetSocketAddress(PORT));

# 技巧

## 写：

1 执行写之前请取消写注册，否则cpu肯定100%；

2 nio是极好的处理率短连接的架构

3

|  |
| --- |
| while (bb.hasRemaining()) {  int len = socketChannel.write(bb);  if (len < 0){  throw new EOFException();  }  if (len == 0) {  selectionKey.interestOps(  selectionKey.interestOps() | SelectionKey.OP\_WRITE);  mainSelector.wakeup(); //  break;  }  }  代码参见<http://qingfengjushi1.iteye.com/blog/1185070> |

## 读：

|  |
| --- |
| **int** count = socketchannel.read(buffer);  **if**(count > 0){  }**else** **if**(count < 0){  key.cancel();  sc.close();  } |

# 其他

## 影响nio性能

为了处理很多socket连接和优化吞吐量，会导致了大量的线程切换。

有很多次的selector的中断和调用，唤醒seletor是很费资源的操作。

1

read或者write操作时候，但没有读满或者写完的情况下，并不是立即返回并再次注册channel到selector，而是再尝试若干次（3次），如mina 处理代码：

for (int i = WRITE\_SPIN\_COUNT; i > 0; i --){

localWrittenBytes = ch.write(buf.buf());

if (localWrittenBytes != 0 || !buf.hasRemaining()){

break;

}

}

优化：

<http://my.oschina.net/javagg/blog/3361>

# 参考

参见<http://developer.51cto.com/art/201112/307463_1.htm>