**Java NIO API详解**

*在JDK 1.4以前，Java的IO操作集中在java.io这个包中，是基于流的同步（blocking）API。对于大多数应用来说，这样的API使用很方便，然而，一些对性能要求较高的应用，尤其是服务端应用，往往需要一个更为有效的方式来处理IO。从JDK 1.4起，NIO API作为一个基于缓冲区，并能提供异步(non-blocking)IO操作的API被引入。本文对其进行深入的介绍。*

NIO API主要集中在java.nio和它的subpackages中：

java.nio

定义了Buffer及其数据类型相关的子类。其中被java.nio.channels中的类用来进行IO操作的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

定义了可用来实现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

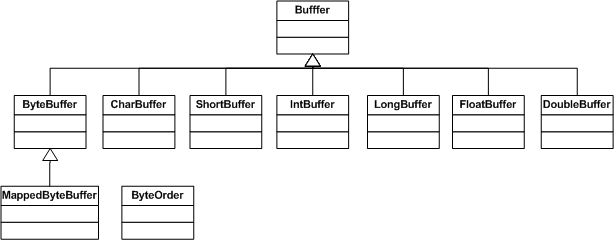
       定义了可用来实现charset API的抽象类。

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。

Package 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。

下面这张UML类图描述了java.nio中的类的关系：



**Buffer**

定义了一个可以线性存放primitive type数据的容器接口。Buffer主要包含了与类型（byte, char…）无关的功能。值得注意的是Buffer及其子类都不是线程安全的。

每个Buffer都有以下的属性：

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。

limit

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。

mark

一个临时存放的位置下标。调用mark()会将mark设为当前的position的值，以后调用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。

这些属性总是满足以下条件：

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

limit和position的值除了通过limit()和position()函数来设置，也可以通过下面这些函数来改变：

**Buffer clear()**

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

**Buffer flip()**

把limit设为当前position，把position设为0，一般在从Buffer读出数据前调用。

**Buffer rewind()**

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

Buffer对象有可能是只读的，这时，任何对该对象的写操作都会触发一个ReadOnlyBufferException。isReadOnly()方法可以用来判断一个Buffer是否只读。

**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：

**ByteBuffer allocate(int capacity)**

创建一个指定capacity的ByteBuffer。

**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。

**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 get(byte [] dst)

byte get(int index)

ByteBuffer put(byte b)

ByteBuffer put(byte [] src)

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()

       从ByteBuffer中读出一个int值。

ByteBuffer putInt(int value)

       写入一个int值到ByteBuffer中。

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（int,long…）。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。

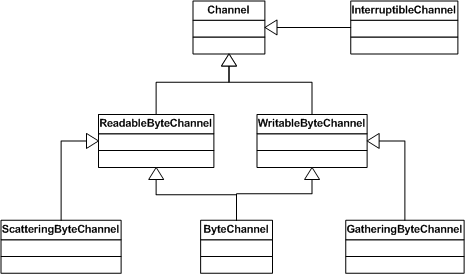
接下来看看一个使用ByteBuffer的例子，这个例子从标准输入不停地读入字符，当读满一行后，将收集的字符写到标准输出：

|  |
| --- |
| **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();             }         }      } |

Package java.nio.channels

这个包定义了Channel的概念，Channel表现了一个可以进行IO操作的通道（比如，通过FileChannel，我们可以对文件进行读写操作）。java.nio.channels包含了文件系统和网络通讯相关的channel类。这个包通过Selector和SelectableChannel这两个类，还定义了一个进行异步（non-blocking）IO操作的API，这对需要高性能IO的应用非常重要。

下面这张UML类图描述了java.nio.channels中interface的关系：



**Channel**

Channel表现了一个可以进行IO操作的通道，该interface定义了以下方法：

boolean isOpen()

       该Channel是否是打开的。

void close()

       关闭这个Channel，相关的资源会被释放。

**ReadableByteChannel**

定义了一个可从中读取byte数据的channel interface。

int read(ByteBuffer dst)

从channel中读取byte数据并写到ByteBuffer中。返回读取的byte数。

**WritableByteChannel**

定义了一个可向其写byte数据的channel interface。

int write(ByteBuffer src)

       从ByteBuffer中读取byte数据并写到channel中。返回写出的byte数。

**ByteChannel**

ByteChannel并没有定义新的方法，它的作用只是把ReadableByteChannel和WritableByteChannel合并在一起。

**ScatteringByteChannel**

继承了ReadableByteChannel并提供了同时往几个ByteBuffer中写数据的能力。

**GatheringByteChannel**

继承了WritableByteChannel并提供了同时从几个ByteBuffer中读数据的能力。

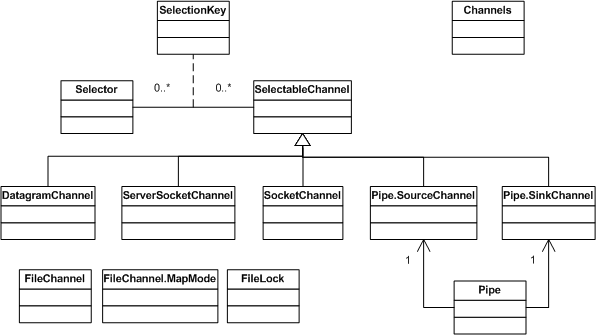
**InterruptibleChannel**

用来表现一个可以被异步关闭的Channel。这表现在两方面：

1．    当一个InterruptibleChannel的close()方法被调用时，其它block在这个InterruptibleChannel的IO操作上的线程会接收到一个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。

接下来的这张UML类图描述了java.nio.channels中类的关系：



**异步IO**

异步IO的支持可以算是NIO API中最重要的功能，异步IO允许应用程序同时监控多个channel以提高性能，这一功能是通过Selector，SelectableChannel和SelectionKey这3个类来实现的。

SelectableChannel代表了可以支持异步IO操作的channel，可以将其注册在Selector上，这种注册的关系由SelectionKey这个类来表现（见UML图）。Selector这个类通过select()函数，给应用程序提供了一个可以同时监控多个IO channel的方法：

应用程序通过调用select()函数，让Selector监控注册在其上的多个SelectableChannel，当有channel的IO操作可以进行时，select()方法就会返回以让应用程序检查channel的状态，并作相应的处理。

下面是JDK 1.4中异步IO的一个例子，这段code使用了异步IO实现了一个time server：

|  |
| --- |
| **private** **static** **void** acceptConnections(**int** port) **throws** Exception {         // 打开一个Selector         Selector acceptSelector =             SelectorProvider.provider().openSelector();           // 创建一个ServerSocketChannel，这是一个SelectableChannel的子类         ServerSocketChannel ssc = ServerSocketChannel.open();         // 将其设为non-blocking状态，这样才能进行异步IO操作         ssc.configureBlocking(**false**);           // 给ServerSocketChannel对应的socket绑定IP和端口         InetAddress lh = InetAddress.getLocalHost();         InetSocketAddress isa = **new** InetSocketAddress(lh, port);         ssc.socket().bind(isa);           // 将ServerSocketChannel注册到Selector上，返回对应的SelectionKey         SelectionKey acceptKey =             ssc.register(acceptSelector, SelectionKey.OP\_ACCEPT);    **int** keysAdded = 0;           // 用select()函数来监控注册在Selector上的SelectableChannel         // 返回值代表了有多少channel可以进行IO操作 (ready for IO)  **while** ((keysAdded = acceptSelector.select()) > 0) {             // selectedKeys()返回一个SelectionKey的集合，             // 其中每个SelectionKey代表了一个可以进行IO操作的channel。             // 一个ServerSocketChannel可以进行IO操作意味着有新的TCP连接连入了             Set readyKeys = acceptSelector.selectedKeys();             Iterator i = readyKeys.iterator();    **while** (i.hasNext()) {                SelectionKey sk = (SelectionKey) i.next();                // 需要将处理过的key从selectedKeys这个集合中删除                i.remove();                // 从SelectionKey得到对应的channel                ServerSocketChannel nextReady =                    (ServerSocketChannel) sk.channel();                // 接受新的TCP连接                Socket s = nextReady.accept().socket();                // 把当前的时间写到这个新的TCP连接中                PrintWriter out =  **new** PrintWriter(s.getOutputStream(), **true**);                Date now = **new** Date();                out.println(now);                // 关闭连接                out.close();             }         }      } |

这是个纯粹用于演示的例子，因为只有一个ServerSocketChannel需要监控，所以其实并不真的需要使用到异步IO。不过正因为它的简单，可以很容易地看清楚异步IO是如何工作的。

**SelectableChannel**

这个抽象类是所有支持异步IO操作的channel（如DatagramChannel、SocketChannel）的父类。SelectableChannel可以注册到一个或多个Selector上以进行异步IO操作。

SelectableChannel可以是blocking和non-blocking模式（所有channel创建的时候都是blocking模式），只有non-blocking的SelectableChannel才可以参与异步IO操作。

SelectableChannel configureBlocking(boolean block)

       设置blocking模式。

boolean isBlocking()

       返回blocking模式。

通过register()方法，SelectableChannel可以注册到Selector上。

int validOps()

返回一个bit mask，表示这个channel上支持的IO操作。当前在SelectionKey中，用静态常量定义了4种IO操作的bit值：OP\_ACCEPT，OP\_CONNECT，OP\_READ和OP\_WRITE。

SelectionKey register(Selector sel, int ops)

将当前channel注册到一个Selector上并返回对应的SelectionKey。在这以后，通过调用Selector的select()函数就可以监控这个channel。ops这个参数是一个bit mask，代表了需要监控的IO操作。

SelectionKey register(Selector sel, int ops, Object att)

这个函数和上一个的意义一样，多出来的att参数会作为attachment被存放在返回的SelectionKey中，这在需要存放一些session state的时候非常有用。

boolean isRegistered()

       该channel是否已注册在一个或多个Selector上。

SelectableChannel还提供了得到对应SelectionKey的方法：

SelectionKey keyFor(Selector sel)

返回该channe在Selector上的注册关系所对应的SelectionKey。若无注册关系，返回null。

**Selector**

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

Selector open()

       Selector的一个静态方法，用于创建实例。

在一个Selector中，有3个SelectionKey的集合：

1． key set代表了所有注册在这个Selector上的channel，这个集合可以通过keys()方法拿到。

2． Selected-key set代表了所有通过select()方法监测到可以进行IO操作的channel，这个集合可以通过selectedKeys()拿到。

3． Cancelled-key set代表了已经cancel了注册关系的channel，在下一个select()操作中，这些channel对应的SelectionKey会从key set和cancelled-key set中移走。这个集合无法直接访问。

以下是select()相关方法的说明：

int select()

监控所有注册的channel，当其中有注册的IO操作可以进行时，该函数返回，并将对应的SelectionKey加入selected-key set。

int select(long timeout)

       可以设置超时的select()操作。

int selectNow()

       进行一个立即返回的select()操作。

Selector wakeup()

       使一个还未返回的select()操作立刻返回。

**SelectionKey**

代表了Selector和SelectableChannel的注册关系。

Selector定义了4个静态常量来表示4种IO操作，这些常量可以进行位操作组合成一个bit mask。

int OP\_ACCEPT

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

int OP\_CONNECT

       代表连接已经建立（或出错），SocketChannel支持这一异步IO。

int OP\_READ

int OP\_WRITE

       代表了读、写操作。

以下是其主要方法：

Object attachment()

返回SelectionKey的attachment，attachment可以在注册channel的时候指定。

Object attach(Object ob)

       设置SelectionKey的attachment。

SelectableChannel channel()

       返回该SelectionKey对应的channel。

Selector selector()

       返回该SelectionKey对应的Selector。

void cancel()

       cancel这个SelectionKey所对应的注册关系。

int interestOps()

       返回代表需要Selector监控的IO操作的bit mask。

SelectionKey interestOps(int ops)

       设置interestOps。

int readyOps()

       返回一个bit mask，代表在相应channel上可以进行的IO操作。

**ServerSocketChannel**

支持异步操作，对应于java.net.ServerSocket这个类，提供了TCP协议IO接口，支持OP\_ACCEPT操作。

ServerSocket socket()

       返回对应的ServerSocket对象。

SocketChannel 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这个类，提供了TCP协议IO接口，支持OP\_CONNECT，OP\_READ和OP\_WRITE操作。这个类还实现了ByteChannel，ScatteringByteChannel和GatheringByteChannel接口。

DatagramChannel和这个类比较相似，其对应于java.net.DatagramSocket，提供了UDP协议IO接口。

Socket socket()

       返回对应的Socket对象。

boolean connect(SocketAddress remote)

boolean finishConnect()

connect()进行一个连接操作。如果当前SocketChannel是blocking模式，这个函数会等到连接操作完成或错误发生才返回。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，否则函数返回false，应用程序需要在以后用finishConnect()方法来完成连接操作。

**Pipe**

包含了一个读和一个写的channel(Pipe.SourceChannel和Pipe.SinkChannel)，这对channel可以用于进程中的通讯。

**FileChannel**

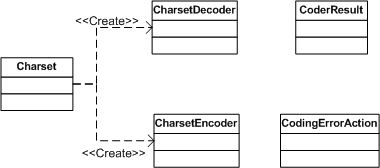
用于对文件的读、写、映射、锁定等操作。和映射操作相关的类有FileChannel.MapMode，和锁定操作相关的类有FileLock。值得注意的是FileChannel并不支持异步操作。

**Channels**

这个类提供了一系列static方法来支持stream类和channel类之间的互操作。这些方法可以将channel类包装为stream类，比如，将ReadableByteChannel包装为InputStream或Reader；也可以将stream类包装为channel类，比如，将OutputStream包装为WritableByteChannel。

Package java.nio.charset

这个包定义了Charset及相应的encoder和decoder。下面这张UML类图描述了这个包中类的关系，可以将其中Charset，CharsetDecoder和CharsetEncoder理解成一个Abstract Factory模式的实现：



**Charset**

代表了一个字符集，同时提供了factory method来构建相应的CharsetDecoder和CharsetEncoder。

Charset提供了以下static的方法：

SortedMap availableCharsets()

       返回当前系统支持的所有Charset对象，用charset的名字作为set的key。

boolean isSupported(String charsetName)

       判断该名字对应的字符集是否被当前系统支持。

Charset forName(String charsetName)

       返回该名字对应的Charset对象。

Charset中比较重要的方法有：

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。

**CharsetDecoder**

将按某种字符集编码的字节流解码为unicode字符数据的引擎。

CharsetDecoder的输入是ByteBuffer，输出是CharBuffer。进行decode操作时一般按如下步骤进行：

1． 调用CharsetDecoder的reset()方法。（第一次使用时可不调用）

2． 调用decode()方法0到n次，将endOfInput参数设为false，告诉decoder有可能还有新的数据送入。

3． 调用decode()方法最后一次，将endOfInput参数设为true，告诉decoder所有数据都已经送入。

4． 调用decoder的flush()方法。让decoder有机会把一些内部状态写到输出的CharBuffer中。

CharsetDecoder reset()

       重置decoder，并清除decoder中的一些内部状态。

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

从ByteBuffer类型的输入中decode尽可能多的字节，并将结果写到CharBuffer类型的输出中。根据decode的结果，可能返回3种CoderResult：CoderResult.UNDERFLOW表示已经没有输入可以decode；CoderResult.OVERFLOW表示输出已满；其它的CoderResult表示decode过程中有错误发生。根据返回的结果，应用程序可以采取相应的措施，比如，增加输入，清除输出等等，然后再次调用decode()方法。

CoderResult flush(CharBuffer out)

有些decoder会在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。如果输出的空间不够，该函数返回CoderResult.OVERFLOW，这时应用程序应该扩大输出CharBuffer的空间，然后再次调用该方法。

CharBuffer decode(ByteBuffer in)

一个便捷的方法把ByteBuffer中的内容decode到一个新创建的CharBuffer中。在这个方法中包括了前面提到的4个步骤，所以不能和前3个函数一起使用。

decode过程中的错误有两种：malformed-input CoderResult表示输入中数据有误；unmappable-character CoderResult表示输入中有数据无法被解码成unicode的字符。如何处理decode过程中的错误取决于decoder的设置。对于这两种错误，decoder可以通过CodingErrorAction设置成：

1． 忽略错误

2． 报告错误。（这会导致错误发生时，decode()方法返回一个表示该错误的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()

       返回malformed-input的出错处理。

CharsetDecoder onMalformedInput(CodingErrorAction newAction)

       设置malformed-input的出错处理。

CodingErrorAction unmappableCharacterAction()

       返回unmappable-character的出错处理。

CharsetDecoder onUnmappableCharacter(CodingErrorAction newAction)

       设置unmappable-character的出错处理。

String replacement()

       返回decoder的替换字串。

CharsetDecoder replaceWith(String newReplacement)

       设置decoder的替换字串。

**CharsetEncoder**

将unicode字符数据编码为特定字符集的字节流的引擎。其接口和CharsetDecoder相类似。

**CoderResult**

描述encode/decode操作结果的类。

CodeResult包含两个static成员：

CoderResult OVERFLOW

       表示输出已满

CoderResult UNDERFLOW

       表示输入已无数据可用。

其主要的成员函数有：

boolean isError()

boolean isMalformed()

boolean isUnmappable()

boolean isOverflow()

boolean isUnderflow()

       用于判断该CoderResult描述的错误。

int length()

       返回错误的长度，比如，无法被转换成unicode的字节长度。

void throwException()

       抛出一个和这个CoderResult相对应的exception。

**CodingErrorAction**

表示encoder/decoder中错误处理方法的类。可将其看成一个enum类型。有以下static属性：

CodingErrorAction IGNORE

       忽略错误。

CodingErrorAction REPLACE

       用替换字串替换有错误的部分。

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。

参考文献

David Flanagan – Java in a Nutshell

作者：DaiJiaLin

<mailto:woodydai@gmail.com>

<http://blog.csdn.net/DaiJiaLin>