

[首页](#) [资讯](#) [精华](#) [论坛](#) [问答](#) [博客](#) [专栏](#) [群组](#) [更多 ▾](#)
[您还未登录!](#) [登录](#) [注册](#)

滴水穿石

- [博客](#)
- [微博](#)
- [相册](#)
- [收藏](#)
- [留言](#)
- [关于我](#)



[java nio api详解《转》](#)

博客分类：

- [Java NIO](#)

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，还提供了进行非阻塞 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`

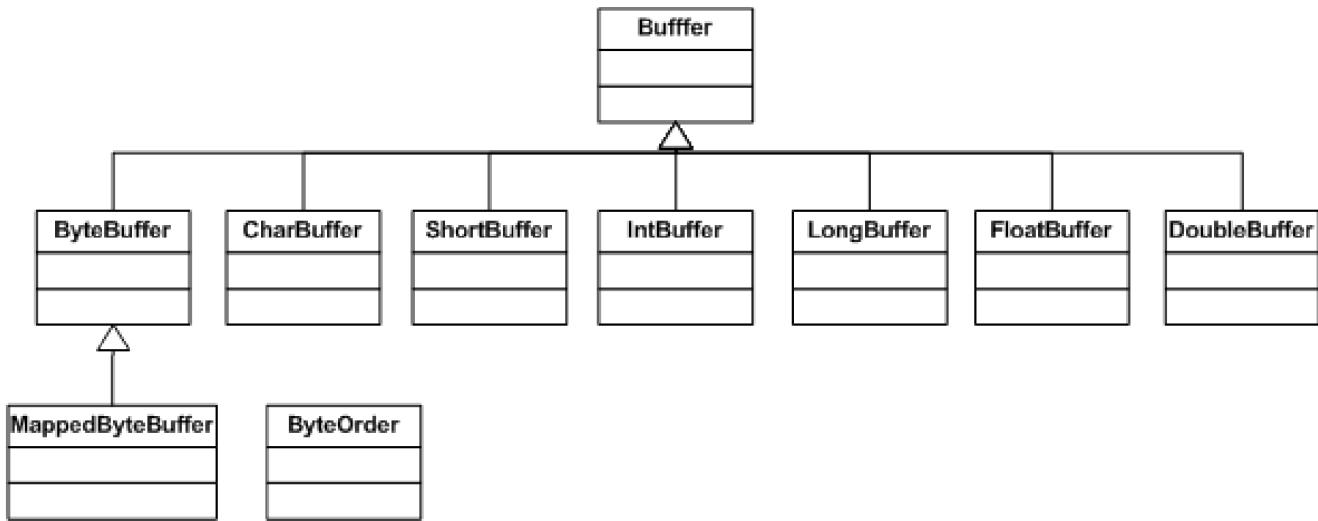
定义了可用来实现 `charset` API 的抽象类。

`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：

$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$

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位两种，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
{
```

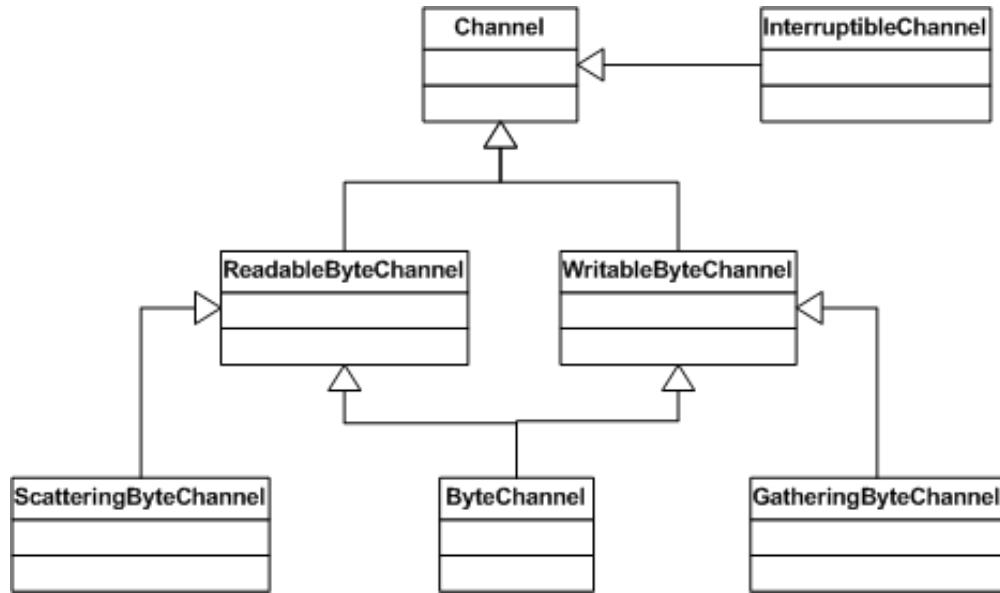
```
// 创建一个 capacity 为 256 的 ByteBuffer
ByteBuffer buf = ByteBuffer.allocate(256);
while ( true ) {
    // 从标准输入流读入一个字符
    int c = System.in.read();
    // 当读到输入流结束时，退出循环
    if (c == -1)
        break;

    // 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，还定义了一个进行非阻塞（ 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

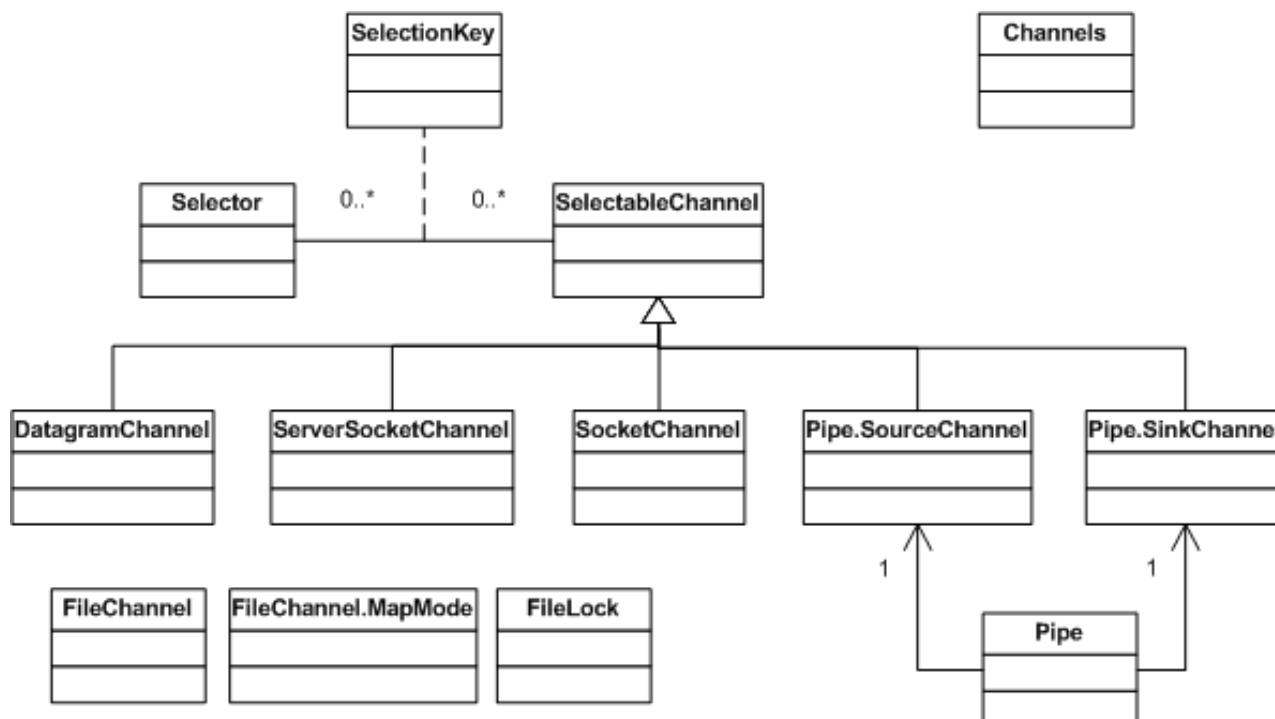
继承了 WritableByteChannel 并提供了同时从几个 ByteBuffer 中读数据的能力。

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:



非阻塞 IO

非阻塞 IO 的支持可以算是 NIO API 中最重要的功能, 非阻塞 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 代表了可以支持非阻塞 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 中非阻塞 IO 的一个例子 , 这段 code 使用了非阻塞 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 , 这样才能进行非阻塞 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非阻塞 IO。不过正因为它的简单，可以很容易地看清楚非阻塞 IO 是如何工作的。

SelectableChannel

这个抽象类是所有支持非阻塞 IO 操作的 channel（如 DatagramChannel、SocketChannel）的父类。SelectableChannel 可以注册到一个或多个 Selector 上以进行非阻塞 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 才可以参与非阻塞 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 channel 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，是非阻塞 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 支持这一非阻塞 IO。

`int OP_CONNECT`

 代表连接已经建立（或出错）， SocketChannel 支持这一非阻塞 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

支持非阻塞操作，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

支持非阻塞操作，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

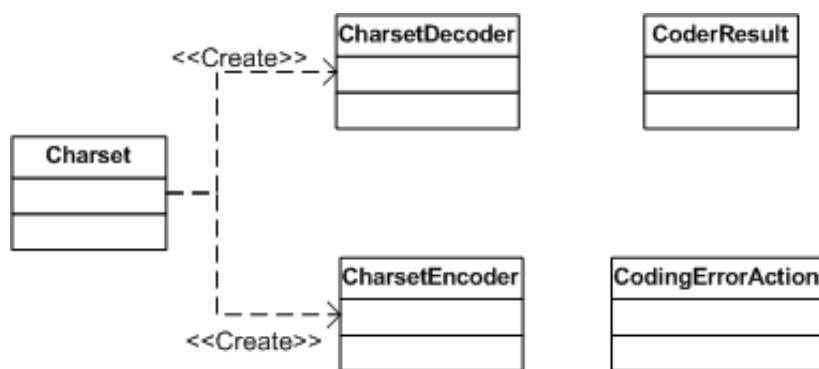
用于对文件的读、写、映射、锁定等操作。和映射操作相关的类有 FileChannel.MapMode，和锁定操作相关的类有 FileLock。值得注意的是 FileChannel 并不支持非阻塞操作。

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。

Packge 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 中的替换字符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 的替换字符串。

CharsetDecoder replaceWith(String newReplacement)

设置 decoder 的替换字符串。

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

。

<http://www.blogjava.net/19851985lili/articles/93524.html>

分享到：  

[Java NIO——Selector机制解析一《转》](#) | [Jprofiler使用一：查看对象的被引用情况和...](#)

- 2013-01-03 14:51
- 浏览 508
- [评论\(0\)](#)
- 分类[编程语言](#)
- [相关推荐](#)

评论

发表评论



[您还没有登录,请您登录后再发表评论](#)



goon

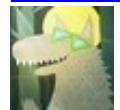
- 浏览: 50593 次
- 性别: ♂
- 来自: 上海
-  [我现在离线](#)

最近访客

[更多访客>>](#)



[nonamed](#)



[pcgreat](#)