这一章讨论了在JDK1.4中引入的新I/O的特性.这些新的特性包含在新的java.nio包和它的子包中,和对java.io、java.net包的一小部分的补充修订.

我们将沿着总线路讨论这些特性,主要因为它们影响网络I/O.随后的章节将深入细节,分别讨论可扩展的TCP和可扩展的UDP.

4.1 介绍

"新I/O"在JDK 1.4中被引入,提供了新的特性和提高了缓冲区管理的性能,可扩展的网络和文件I/O,字符集支持,和正则表达式匹配.

该规范是在JCP中作为JSR 51发展的.JSR 51的组成主要是高级的网络编程,规定如下:

(a) 可扩展的I/O,以"非阻塞模式"和"多路复用"形式.

(b) 快速缓冲二进制和字符I/O.

非阻塞模式和多路复用使写出"生产质量web和应用程序服务器可扩展可以很好地处理数千的打开的连接,可以容易地利用多处理器(*)"成为可能.这主要是因为在TCP和UDP服务器中使用非阻塞I/O和多路复用,不需要为每个打开的连接请求一个独立的Java线程,不像我们在第3章看到的TCP服务器.

快速地缓冲二进制和字符I/O使得写出"高性能,I/O密集型(intensive)程序操纵流或者二进制数据的文件"成为可能.

---

* 在这章所有的引用(quotations)来自于JDK 1.4文档和JSR-51规范,除非另外指明.

4.1.1 与I/O流比较

传统Java IO-提供了java.io包作为"流"类的层级关系,共享和继承了一个公共的API.一个Java流是从InputStream或者OutputStream衍生出的对象,代表了一个打开的连接,或者数据路径到一个外部的数据源或者接收器(sink).流是单向的,也就是说,可以用来输入或者输出,但不是两个都是.流可以链接在一起,这样可以形成I/O过滤链:比如,一个DataInputStream经常被链接到一个BufferedInputStream,通常依次链接到一个从Socket或者FileInputStream获得的输入流.

```
Socket socket = new Socket("localhost",7);
InputStream in = socket.getInputStream();
// Add buffering to input
in = new BufferedInputStream(in);
// Add data-conversion functions to input
DataInputStream dataIn = new DataInputStream(in);
```

如这个例子所示的,各种类型的流一起组合成了输入-输出的功能,缓冲和数据转换.

传统的流提供了缓冲作为一个可选的特性,而不是一个内置的I/O机制的一部分;这个设计使得运行多种缓冲区成为可能,不管是有意的还是不注意的(FIXME: this design made it possible to run multiple buffers, whether deliberately or inadvertently).这个技术用来将流链接到一起,可以在每个流的交叉点(junction)上简化数据的复制操作(FIXME: 书中是imply a data..., 不知道是不是搞错了, 感觉应该是simply a data...),特别是,当做数据转换操作.所有的这些因素可能导致Java程序的低效的I/O.

在Java"新I/O"中,这三个功能已经被独立出来了.输入-输出通过一个新的称为"管道"的抽象概念来提供;缓冲和数据转换通过一个新的抽象称为"缓冲"来提供.有一个channel类的层级结构和一个独立的buffer类的层级结构.

新I/O背后的基本原理(rationable)是清晰的.将I/O功能分离为channel操作和buffer操作允许Java

设计者这么做：
(a) 请求一个channel总是和一个buffer关联.
(b) 指定channel和buffer可以适应在一起的方式.
(c) FXIME: Do so via the type system so you can't even compile an incorrect program
(d) 为异步关闭和中断I/O提供可靠的和可移植的语义.
(e) 在不同的方向上扩展channel和buffer类而不是被Java类型系统所禁锢.(FIXME: Extend the channel and buffer classes in different directions without being 'imprisoned' by the Java type system)
(f) 扩展channel和buffer类去提供高级的功能,比如在channel中轮询和非阻塞I/O,直接缓冲区和字符集转换.

4.1.2 新I/O概览

新I/O通过channel和buffer执行.一个channel连接到一个外部的数据源或者接收器(比如,文件或者网络套接字),拥有读和/或者写的操作.一个buffer在程序中保存数据,提供了"get"和"put"操作,从一个channel中读取从数据源读取的数据,然后将它放入buffer;相反地,写入到一个channel,从buffer中获得数据然后写入到接收器.

存在更多强大的channel类支持多路复用以及读和写.存在不同基本类型的不同buffer类.

4.2 管道(Channel)

一个Java管道代表了一个打开的文件路径,可能是双向的,到一个外部的数据源或者接收器,比如一个文件或者一个socket.

和流对比,管道可以是双向的,也就是说,输入和输出都可以执行,如果外部的数据对象(数据源或者接收器)许可.管道不能像流一样被链接到一起,相反,管道被限制(confine)去执行实际的I/O操作.缓冲区和数据转换的特性在java.nio中提供,在4.3节讨论.

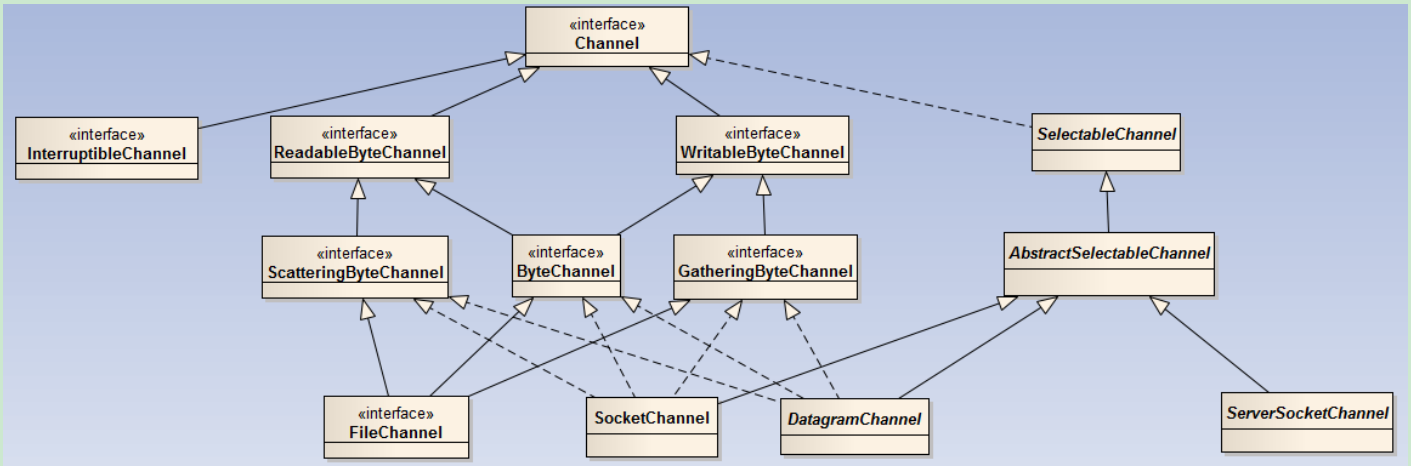
4.2.1 Channel 继承结构

管道接口和类的完整的树在表4.1中展示(书中有误,不存在DatagramSocketChannel,应该是DagagramChannel).

TABLE 4.1 Hierarchy of channels

Inheritance tree	Interfaces implemented
Channel	
⌊ *InterruptibleChannel*	*Channel*
⌊ *ReadableByteChannel*	*Channel*
⌊ ⌊ *ScatteringByteChannel*	*ReadableByteChannel*
⌊ *WritableByteChannel*	*Channel*
⌊ ⌊ *GatheringByteChannel*	*WritableByteChannel*
⌊ *ByteChannel*	*ReadableByteChannel*, *WritableByteChannel*
AbstractInterruptibleChannel[a]	*Channel*, *InterruptibleChannel*
⌊ FileChannel	*ByteChannel*, *ScatteringByteChannel*, *GatheringByteChannel*
⌊ SelectableChannel	*Channel*
⌊ AbstractSelectableChannel[a]	
⌊ DatagramSocketChannel	*ByteChannel*, *ScatteringByteChannel*, *GatheringByteChannel*
⌊ Pipe.SinkChannel	*WritableByteChannel*, *GatheringByteChannel*
⌊ Pipe.SourceChannel	*ReadableByteChannel*, *ScatteringByteChannel*
⌊ ServerSocketChannel	*Channel*
⌊ SocketChannel	*ByteChannel*, *ScatteringByteChannel*, *GatheringByteChannel*

a. in java.nio.channels.spi: an implementation detail.



每种具体的通道的实现准确的联系在一起,只有I/O操作的能力(FIXME: Each concrete channel implementation is associated with exactly and only the I/O operations it is capable of).比如,ServerSocketChanel实现了SelectableChannel,将在4.5节遇到,但是唯一的I/O操作就是支持关闭.作为对比,SockeChannel实现了ByteChannel:因此,间接地(indirectly),它也实现了ReadableByteChannel和WriteableByteChannel,所以它支持简单的读和写;它也实现了

ScatteringByteChannel和GatheringWriteChannel,所以它支持聚合写和分散读(两者将在后面描述)(这个地方,书中写的是gathering reads and scattering writes,应该是弄反了).

4.2.2 Channel接口和方法

java.nio.Channel是channel接口的继承关系的根.Channel接口只有一个close操作和一个isOpen查询操作:

```
interface Channel {
    boolean isOpen();
    void    close() throws IOException;
}
```

ReadableByteChannel只有1个简单的读操作:

```
interface ReadableByteChannel extends Channel {
    // returns the number of bytes read
    int  read(ByteBuffer destination) throws IOException;
}
```

WriteableByteChannel只有1个简单的写操作:

```
interface WritableByteChannel extends Channel {
    // returns the number of bytes written
    int  write(ByteBuffer source) throws IOException;
}
```

ByteChannel统一了readable和writeable接口:

```
interface ByteChannel extends ReadableByteChannel, WritableByteChannel {
}
```

ScatteringByteChannel接口展示了分散读方法,使用底层操作读取数据然后分散到多个缓冲区(目标):

```
interface ScatteringByteChannel extends ReadableByteChannel {
    // Both methods return the number of bytes read
    long read(ByteBuffer[] targets) throws IOException;
    long read(ByteBuffer[] targets, int offset, int length)throws IOException;
}
```

GatheringByteChannel接口展示了聚合写方法, 要写入的数据从多个缓冲区(数据源)聚合然后使用一个底层的操作写入.

```
interface GatheringByteChannel extends WritableByteChannel {
    // Both methods return the number of bytes written
    long write(ByteBuffer[] sources) throws IOException;
    long write(ByteBuffer[] sources, int offset, int length) throws IOException;
```

```
}
```

在这两种情况下,offset和length都是ByteBuffer[]自身的引用.而不是任何独立的ByteBuffer.

"分散读"和"聚合写"方法与Berkeley Sockets的readv()和writev()API一致.注意这些接口的方法返回的是long而不是int.这是在java.nio规范中后来的变化:需要long是因为在ByteBuffer[]数组中可达到过2^{31} - 1个元素,可以包含超达到2^{31} - 1个字节;因此传输的总共的大小可以达到2^{64} - 1个字节,可以比int代表的更多.

InterruptibleChannel标识了可异步关闭和中断的管道:

```
interface InterruptibleChannel extends Channel {
    void close() throws IOException;
}
```

一个可中断的管道可以被异步地关闭,也就是说,一个线程调用InterruptibleChannel.close()方法当另一个线程在管道上执行一个阻塞的I/O操作.这个将会导致阻塞的线程引发一个AsynchronousCloseException.

一个可中断的管道可以被异步地中断,比如,一个线程调用Thread.interrupt(),另一个线程在管道上执行阻塞的I/O操作.这将会关闭管道,然后导致阻塞的线程引发一个ClosedByInterruptException(*).

---

* 这个可能使人感到奇怪.我们可能期望通道保持打开和阻塞的线程获得一个InterruptedIOException.close语义通过需求指定跨越各种支持的平台支持相同的语义(FIXME: The close semantics are dictated by the requirement to provide the same semantics across the various supported platforms),特别是,Linux的奇怪行为:当一个线程在一个套接字上阻塞被异步的中断.(这段不是很明白什么意思)

4.2.3 获得一个管道

一个管道可以从FileInputStream,FileOutputStream或者一个RandomAccessFile中获得.

```
FileChannel FileInputStream.getChannel();
FileChannel FileOutputStream.getChannel();
FileChannel RandomAccessFile.getChannel();
```

一个管道也可以从Socket,ServerSocket或者DatagramSocket中获得.但是如果一个socket从一个管道中被创建,这样操作是间接的.

通常是从管道中获得socket,而不是从socket中获得channel.

```
SocketChannel        channel  = SocketChannel.open();
Socket socket  = channel.socket();

ServerSocketChannel  channel  = ServerSocketChannel.open();
ServerSocket        serverSocket  = channel.socket();
```

```
DatagramChannel      channel  = DatagramChannel.open();
DatagramSocket       datagramSocket= channel.socket();
```

一个MulticastSocketChannel类据说(reportedly)计划在JDK 1.5中引入但没有出现.

最后,一个管道也可以从一个java.nio.Piple中获得:

```
Pipe pipe = Pipe.open();
Pipe.SinkChannel     sinkChannel    = pipe.sink();
Pipe.SourceChannel   sourceChannel   = pipe.source();
```

管道与唯一的一个个文件输入或者输出流,随机访问文件,socket或者pipe关联.重复调用getChannel,Pipe.sink或者Pipe.Source总是返回相同的对象.

4.2.4 Channel 转换-Channels类

Channels类提供了静态的方法在流和管道间转换.

```
class Channels {
    // convert streams to channels
    static ReadableByteChannel  newChannel(InputStream is);
    static WritableByteChannel  newChannel(OutputStream os);
    // convert channels to streams
    static InputStream newInputStream(ReadableByteChannel ch);
    static OutputStream    newOutputStream(WritableByteChannel ch);
}
```

它也提供了方法转换channels为Reader和Writer,在这本书的后面没有讨论.通过这些方法传递的stream, reader和writer有大量的特殊的属性:

(a) 它们不是缓冲的.
(b) 它们不支持标记/重置机制.
(c) 它们是线程安全的.
(d) 它们只能用在阻塞模式:如果用在非阻塞模式(后面将会讨论),它们的读和写方法将会抛出IllegalBlockingModeException.
(e) 关闭它们将会引起底层的管道被关闭.

属性(d)是重要的.这意味着在一个阻塞的管道上你只能通过DataInput和DataOutput的方法读和写Java的数据类型.或者通过ObjectInput和ObjectOutput的方法序列化对象.

下面的代码片段将在指明的地方抛出IllegalBlockingModeException.

```
SocketChannel    channel = SocketChannel.open();
channel.connect(new InetSocketAddress("localhost",7));
channel.configureBlocking(false);
InputStream     in = Channels.newInputStream(channel);
ObjectInputStream  objIn = new ObjectInputStream(in);

// The next line throws an IllegalBlockingModeException
Object object = objIn.readObject();
```

```
OutputStream    out = Channels.newOutputStream(channel);
ObjectOutputStream   objOut = new ObjectOutputStream(out);

// The next line throws an IllegalBlockingModeException
objOut.writeObject(object);
```

4.2 缓冲区

一个管道和一个缓冲区结合(in conjunction with)只能执行输入-输出.

一个java.nio"缓冲区是"是单个基本类型的数据的容器.每种基本类型都提供了一个缓冲区类.byte,char,short,int,long,float和double(但没有boolean).缓冲区有有限的容量.

除了数据,缓冲区也包含了四种互相依赖的状态属性: 容量,极限,位置和标记:

(a) 容量是一个缓冲区包含的元素的数目.依赖于缓冲区支持的数据类型的大小(FIXME: What this represents in bytes depends on the size of the datatype supported by the buffer).容量是不可变的:当缓冲创建的时候就固定了.

(b) 缓冲区的极限是第一个不能被读或者写的元素的索引,是可变的.

(c) 缓冲的位置是下一个应该被读或者写的元素的索引.是可变的.

(d) 缓冲区的标记是它将要恢复的位置的索引,如果它的reset方法被调用.是可变的: 它总是未定义的,但是它可以被定义,随后可以被程序操作修改.当一个缓冲区第一次被创建的时候它是未定义的.标记将被废弃(也就是说变成未定义的)如果位置或者极限调整到小于当前的标记.

容量,极限,位置和标记总是满足下面的不变式:

$$0 \le mark \le limit \le capacity \quad (EQ\ 4.1)$$

在缓冲区上通过所有的操作这将都被保存(FIXME: which is preserved by all operations on the buffer).

在一个缓冲区中剩余的数据或者空间的数量:

$$limit - position \quad (EQ\ 4.2)$$

缓冲区的位置,极限和容量在图4.1说明.

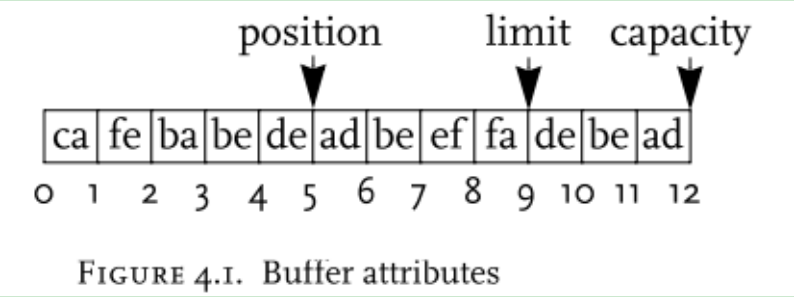


FIGURE 4.1. Buffer attributes

提供的新缓冲区的分配和从已经存在的缓冲区创建缓冲区(duplicate, slice, wrap)的操作,如我们将在4.3.2节中所看到的.

缓冲区提供了指定类型的put和get操作,分别表示将数据加加入或者移除: 这些是缓冲区的基本功能.put和get操作分别通过管道的read和write操作被调用.理解这个是非常重要的,以正确的方式理解它(FIXME: to get it the right way around).当写数据,一个管道需要从一个缓冲区中得到它;当读取数据,一个管道需要将它放入到缓冲区中.

缓冲区也提供了简单的操作,直接修改它们的内部状态(标记,位置,极限和容量),将在4.3.4节看到,和复合(compound)操作,在同一时间修改它们中的多个项目,将在4.3.5节看到.

4.3.1 Buffer继承关系

buffer类的完整的树展示在了表4.2的树图中.

TABLE 4.2 Hierarchy of buffers

- Buffer
  - ByteBuffer
    - MappedByteBuffer
  - CharBuffer
  - DoubleBuffer
  - FloatBuffer
  - IntBuffer
  - LongBuffer
  - ShortBuffer

为了简化讨论,我们引入了元符号(meta-notation)按照JDK 1.5 泛型语言特性(*).使用T去表明任何的一个缓冲区实现提供的基本类型.比如:

```
class Buffer<T> extends Buffer {
    // this is a buffer for objects of primitive type
}
```

---

* 要明确,实际上缓冲区在JDK 1.5中并没有以这种方式定义(因为相关的类型是基本类型).我只是使用这个符号来避免描述7种或者8数据结构相同的类.

4.3.2 获得一个Buffer

缓冲区没有构造但是可以获取(*).每种具体的基本类型T的缓冲区实现有创建缓冲区的方法.

```
class Buffer<T> extends Buffer {
    static Buffer<T> allocate(int capacity);
    Buffer<T> asReadOnlyBuffer();
    boolean isReadOnly();
    Buffer<T> duplicate();
    Buffer<T> slice();
    static Buffer<T> wrap(T[] array);
    static Buffer<T> wrap(T[] array, int offset, int length);
}
```

---

* 这是因为它们被规定为抽象类,通过隐藏类实现返回一个对象工厂,为了支持NIO SIP(服务提供接口).

静态的allocate方法实际上是一个工厂方法,返回指定容量和相关类型T的一个新的空的缓冲区.指定的容量作为类型T的期望的项目的数目.而不是作为字节的数目(如果不同的话).

静态的wrap方法返回一个新的缓冲区包装了一个数据的数组.

asReadOnlyBuffer,duplicate和slice实例方法返回当前缓冲区的内部数据镜像的缓冲区,但是内部状态以各种方式设置而不同.

asReadOnlyBuffer方法返回一个共享当前的缓冲区数据的只读的缓冲区,但是不包括标记,位置或者极限,虽然这些在最开始的时候是相同(identical).改变当前缓冲区的数据将反映在只读缓冲区(但是反之不行:一个只读缓冲区是不可变的).isReadOnly方法返回一个缓冲区的只读状态.

duplicate方法返回一个共享当前缓冲区数据的缓冲区,(但标记,位置或者极限不共享),尽管这些在开始的时候是相同的.改变当前的缓冲区的数据将反映在新的缓冲区中,反之亦然.如果当前的缓冲区是只读的,那么新的缓冲区也是只读的;(FIXME: ditto 'direct').

slice方法返回一个共享子序列内容的缓冲区-一个'slice'-源缓冲区的内容,从源缓冲区的位置到极限继承而来.改变当前的缓冲区的数据将反映到新的缓冲区,反之亦然.新的缓冲区的标记,位置和极限是独立的.如果当前的缓冲区是只读的,那么新的缓冲区也是只读的;(FIXME: ditto 'direct').

上面的每个方法返回的的缓冲区的初始设置总结在表格4.3中.

TABLE 4.3 Initial NIO buffer settings

Method	capacity	position	limit	mark
allocate(capacity)	= capacity	zero	= capacity	undefined
wrap(array)	= array.length	zero	= capacity	undefined
wrap(array, offset, length)	= length	= offset	= offset + length	undefined
asReadOnlyBuffer	initially as source, not mirrored			
duplicate	initially as source, not mirrored			
slice	= source capacity	zero	= source limit – source position	initially as source

4.3.3 缓冲区数据操作: 'get'和'put'

每种具体的基本类型T的缓冲区实现有从缓冲区获得类型T项目的方法,它也有将类型T项目放到缓冲区的方法.

```
class Buffer<T> extends Buffer {
    T get(); // Get one  T , relative
    Buffer<T> get( T [] buffer); // Get bulk  T , relative
    Buffer<T> get( T [] buffer, int offset, int length);
    Buffer<T> get(int index); // Get one  T , absolute
    Buffer<T> put( T data); // Put one  T , relative
    Buffer<T> put( T [] buffer); // Put bulk  T , relative
    Buffer<T> put( T [] buffer, int offset, int length);
    Buffer<T> put(int index,  T data); // Put one  T , absolute
    // Put remaining  T  from source into 'this'
    Buffer<T> put(Buffer<T> source);
}
```

带有数组参数的get和put方法提供了批量操作.带有index参数的get和put方法提供了绝对位置的操作,不

依赖于或者干扰缓冲区的位置.

所有的其它操作是相对的操作,在位置上操作,然后增长传输的项目的数量(不是传输的字节数).

绝对的get操作在指定的索引从缓冲区中获得单个数据项目不会干扰缓冲区的位置,在检查不变式之后:

```
0 ≤ index ≤ position (EQ 4.3)
```

如果不持有数据,将抛出IndexOutOfBoundException.(FIXME: holds, throwing an IndexOutOfBoundException if not).

相对的get操作首先通过Equation 4.2给出的等式检查剩余数据的数量是否足够满足需求.如果不满足将抛出BufferUnderflowException,如果操作指定了一个数组,offset和长度,它会检查下面的不变式:

```
0 ≤ length + offset ≤ data.length    (EQ 4.4)
```

如果不满足将会抛出IndexOutOfBoundsException;另外它从当前的位置开始传输数据,最后位置将增长传输的数据项的数目.

put方法是一个可选的操作.一个只读缓冲区不允许这样的操作.如果调用的话将抛出ReadOnlyBufferException.这是一个Java的RuntimeException.也就是说,是一个未检查的异常.

一个缓冲区可用于只读或者读-写的.任何的缓冲区都可以转变为相同类型的只读缓冲区,使用asReadOnly方法镜像相同的数据,将在4.3.2节描述.

绝对的put操作将单个数据项放到指定索引的缓冲区中,不会干扰缓冲区的位置,在检查等式4.3之后,如果不满足将抛出IndexOutOfBoundsException.

相对的put操作首先检查由等式4.2给出的剩余空间的数量是否足够去满足请求,如果不满足将抛出BufferOverflowException.如果操作指定了一个数组,offset和长度,它会检查等式4.4.如果不满足将会抛出IndexOutOfBoundsException.另外它从当前的位置开始传输数据,最后位置将增长传输的数据项的数目.

相对的get和put操作一个项目,缓冲区的属性的影响如图4.2.

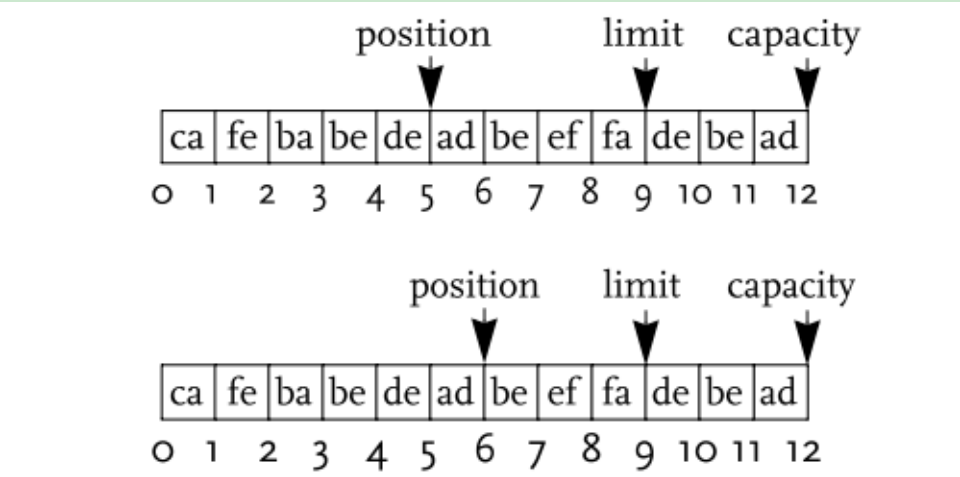


FIGURE 4.2. Buffer before and after one-item relative get or put

4.3.4 简单的缓冲区的状态操作

抽象的Buffer类展示了可以改变它的状态的方法,由具体的指定类型的缓冲区类继承这些方法.

```
class Buffer {
    int     position(); // get position
    int     limit(); // get limit
    int     capacity(); // get capacity
    Buffer  position(int position);   // set position
    Buffer  limit(int limit);         // set limit
    boolean hasRemaining();
    int     remaining();
    Buffer  mark();
    Buffer  reset();
}
```

hasReamaining方法返回true如果缓冲区在当前的位置和极限之间有数据元素.remaining方法返回这样的数据元素的数目(不是字节的数目).mark方法设置缓冲区的标记为当前的位置.reset方法将位置重置为当前的标记,另外,它会抛出InvalidMarkException.这个操作一般用在重新读或者覆盖写的时候.

4.3.5 复合的状态操作

下面方法在一个高级别的层次上操作标记,位置和极限.

```
class Buffer {
    Buffer  clear();
    Buffer  flip();
    Buffer  rewind();
}
```

这些操作将在随后的章节详细讨论.

4.3.6 Clear操作

clear操作逻辑上清理缓冲区(不会影响数据内容).这个通常在put操纵前执行.

```
buffer.clear();
buffer.put(…);
```

或者一个管道的读操作,相当于从缓冲区的角度来看(FIXME: which is equivalent from the buffer's point of view:)

```
buffer.clear();
channel.read(buffer);
```

clear操作设置位置为0,极限为容量,如图4.3所示.

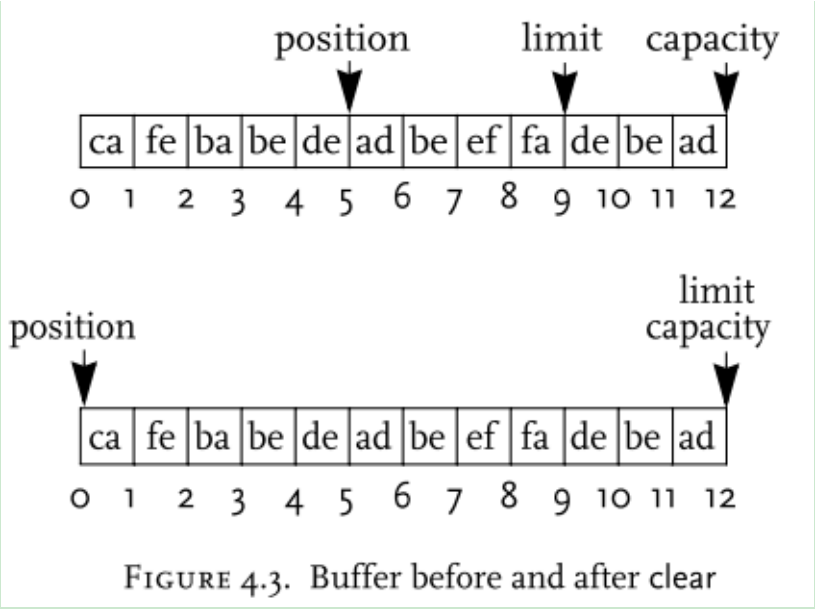


FIGURE 4.3. Buffer before and after clear

4.3.7 Flip操作

令人感兴趣的flip操作使'put'到缓冲区的数据为'getting'变得可用(FIXME: The curiously named flip operation makes data which has just been ‘put’ into the buffer available for ‘getting’).它通常在放入数据到缓冲区后操作,然后将它再次取出来.

```
buffer.put(…);
buffer.flip();
buffer.get(…);
```

因为read等同于put,write等同于get,flip操作可以用在双向的I/O.考虑一个通道的写操作:

```
buffer.put(array);
buffer.flip();
channel.write(buffer);
```

或者一个管道的读操作:

```
channel.read(buffer);
buffer.flip();
buffer.get(array);
```

flip操作可以被认为是从put/read模式到get/write模式的反向操作(FIXME: The flip operation can be thought of as flipping the buffer from put/read mode to get/write mode).flip操作设置极限为位置,位置为0,如图4.4所示.

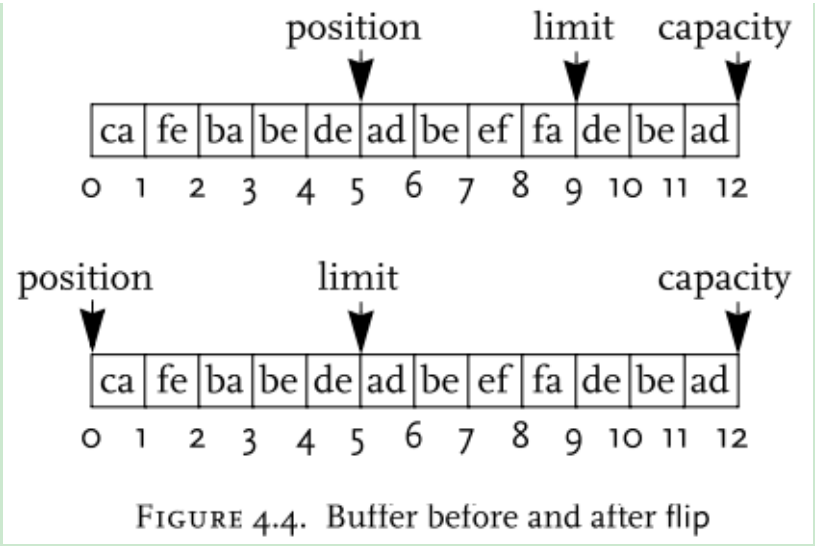


FIGURE 4.4. Buffer before and after flip

4.3.8 Rewind操作

rewind操作倒转缓冲区,假设limit已经被恰当的设置了.它应该被执行当获取或者写入相同的数据不止一次(more than once):

```
channel.write(buffer); // write data
buffer.rewind();
buffer.get(array); // get what was written
```

或者:

```
buffer.get(array); // get what's to be written
buffer.rewind();
channel.write(buffer); // write it
```

或者

```
channel1.write(buffer); // write data to channel 1
buffer.rewind();
channel2.write(buffer); // write same to channel 2
```

rewind操作设置位置为0,不会改变极限,如图4.5所示:

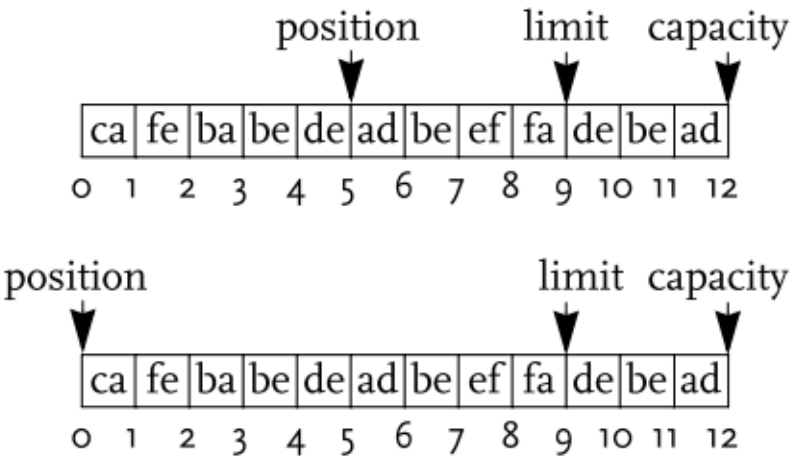


FIGURE 4.5. Buffer before and after *rewind*

4.3.9 'Compact'操作

每种具体的基本类型T的缓冲区实现展示了一个压缩缓冲区的方法：

```
class Buffer<T> extends Buffer {
    Buffer<T> compact();
}
```

compact操作在一个get操作之后压缩缓冲区(比如,一个管道的写),这样任何"未获得(un-got)"的数据现在在缓冲区的开始处,随后的put将会在这个数据之后用新的数据替换,将为put准备最大的可用空间.

compcat操作通常与flip操作结合执行当读取或者写入的时候,下面展示了相当整洁(rather neat)的复制操作：

```
buffer.clear();
// Loop while there is input or pending output
while (in.read(buffer) >= 0 || buffer.position() > 0) {
    buffer.flip();
    out.write(buffer);
    buffer.compact();
}
```

这个例子说明了几个关键的概念：

(a) 读操作的执行;如果返回一个负数,说明已经达到了流的末尾.

(b) 这时候,位置(就是位置所处的索引的值)是剩余的数据项的数量,将从位置0开始被写入;如果位置是0,没有剩余的可被写入.

(c) flip操作为写操作准备好缓冲区.

(d) write操作写入尽可能被写入的数据.(非阻塞模式下,write操作不一定能够将此次的数据都全部写完,也就是说在position并未达到limit,在position~limit之间还有剩余的数据).

(e) compact操作为下一次读操作准备好缓冲区.将新的数据追加到缓冲区的末尾.

(f) 循环迭代当新的数据被读取(read() > 0)或者仍然有未被读取的数据要被写(buffer.position() > 0).

注意,一旦达到了流的末尾,迭代操作只包含写操作(FIXME: iterations consist of write operations only);换句话或,下面的顺序：

```
while (buffer.position() > 0) {
    buffer.flip();
    out.write(buffer);
    buffer.compact();
}
```

等同于下面的顺序：

```
while (buffer.hasRemaining())
    out.write(buffer);
```

尽管后面的更加有效,(FIXME: it doesn't move the data around).如果需要最大效率,因此上面的复制例子可以这么写：

```
buffer.clear();
// Loop while there is input
```

```
while (in.read(buffer) >= 0) {
    buffer.flip();
    out.write(buffer);
    buffer.compact();
}
// final flush of pending output
while (buffer.hasRemaining())
    out.write(buffer);
```

这些复制的例子在阻塞模式或者非阻塞模式(在4.4节讨论)都工作的很好,尽管更优先使用Selector去探测未处理的输入和可用的输出空间(FIXME:
although it is preferable to use a Selector to detect pending input and available output space),在4.5节讨论.

像put操作一样,compact操作是可选的,它不被只读缓冲区支持;如果在一个不支持它的缓冲区上调用,将抛出ReadOnlyBufferException.

compact操作处理如下:

(a) 它废弃已经"got"的任何数据.也就是,在位置之前的任何数据.
(b) 它移动任何剩余的数据(在位置和极限之间)到缓冲区的开始处.
(c) 它设置位置为极限-位置(也就是,最后移动的数据项后的索引),然后设置极限为容量.

这个在图4.6说明.

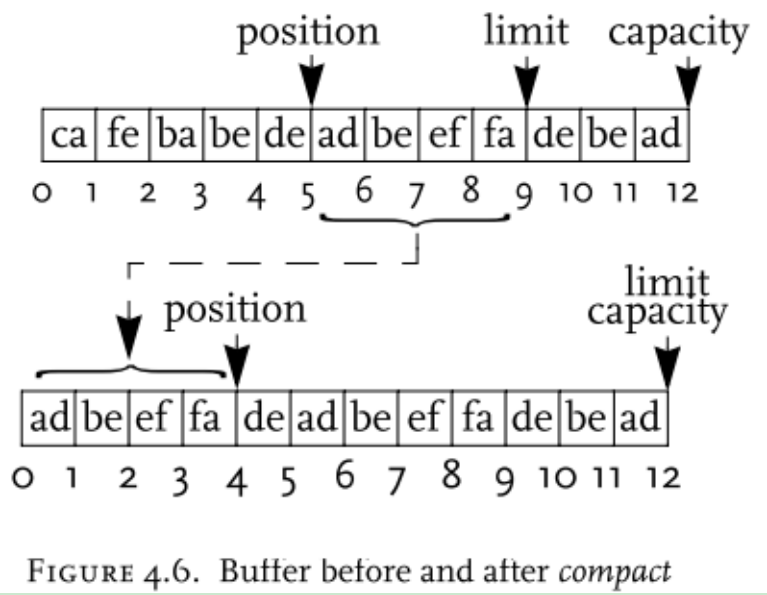


FIGURE 4.6. Buffer before and after *compact*

4.3.10 Byte buffers(FIXME)

4.3.11 视图缓冲区(FIXME)

4.3.12 缓冲区和线程

缓冲区被多个线程使用不是线程安全的,除非线程执行适当的同步,比如,通过在缓冲区自身上同步.

4.4 非阻塞的I/O

一般的Java I/O模式是阻塞的：输入输出阻塞调用线程进一步的执行(FIXME：input-output blocks the calling thread from further execution)直到有非空的数据传输.满足非空的规则可能包含在一个外部数据源或者接收器上等待分别产生或者消费数据.

新I/O管道提供了非阻塞的输入输出,没有非空的规则,另外永远不需要在一个外部的数据源或者接收器等待.

4.4.1 目的

非阻塞I/O最主要的目的是避免阻塞调用的线程,这样它可以做一些有用的事情而不是只是等待数据达到或者离开.这意味着一个端点(比如,服务器或者客户端线程)永远不会在另一个端点的动作上阻塞等待(FIXME：blocks awaiting an action by the other endpoint)(比如,客户端或者服务器端线程).比如,它允许一个网络客户端以消息驱动模型写出而不是一个RPC(远程过程调用)模型.

另外的优势就是单个线程可以处理多个任务(比如,多个非阻塞的连接)而不是一个网络连接有一个专门的线程:这将依次节省线程(FIMXE：this in turn economizes on threads)和内存;相反,它允许一个给定数量的线程去处理更多数量的网络连接.

4.4.2 阻塞I/O的特性

管道最开始是在非阻塞模式中创建的.流-即使与管道在一起-也只能在阻塞模式中被操作.

阻塞的read操作阻塞直到至少有一些数据可用,尽管不必要请求的所有的数据.它可能传输少于请求的数据的数量,也就是说,可能会返回一个更小的值.一个网络流或者管道中的read阻塞当在套接字的接收缓冲区中没有数据.一旦任意数量的数据在套接字接收缓冲区中变得可用,它就传输然后重新开始执行.如果没有数据可用,一个阻塞的读可能因此必须去等待另一个端点发送一些数据.

阻塞的write操作阻塞直到至少有一些数据可以被传输,尽管不必要请求的所有的数据.在一个管道上的写操作可能传输少于请求的数据的数目,也就是说,可能返回一个更小的值,尽管它总是传输一些什么,也就是说,它永远不会返回0(*).在一个网络管道上的写操作阻塞直到至少有一些套接字的发送缓冲区空间可用.一旦空间变得可用,数据被传输然后重新开始执行.由于发送缓冲区空间依赖于另一个端点的接收缓冲区,一个阻塞的写可能因此必须等待另一个端点通过将数据读取出然后在它的接收缓冲区创建空间.

阻塞的TCP管道的connect操作在5.2.1节描述.UDP管道的连接操作从来不阻塞,如10.1.3节描述的.

一个处于阻塞模式的管道不能与Selector一起使用,如4.5.9节描述的(**).

---

* 然而,流上的写操作阻塞直到传输完成或者出现异常.这个通过内部的循环实现,OutputStream.write方法总是返回void：一个未完成的传输不能通知调用的应用程序.

** 另外将抛出IllegalBlockingModeException.这个可能会使用Berkeley Sockets程序员感到奇怪,他们可以在阻塞和非阻塞的套接字上都使用select().

4.4.3 非阻塞I/O的特性

非阻塞I/O只能与JDK 1.4的管道执行.

非阻塞的read操作-不像阻塞的读-可返回0,表明没有数据可用.非阻塞只是读取网络通道传输的数据.如果有的话,那就是在这次调用的时候的已经存在于socket接收到的缓冲区(FIXME：that was already in the socket receive-buffer at the time of the call).如果数量为0,读传输没有数据,返回0.

非阻塞的write操作,不像阻塞的写-可返回0,表明没有空间可用.在一个网络通道上的非阻塞写操作尽可能的传输当前追加到socket发送缓冲区的数据.如果数量为0,写没有传输数据返回0.

非阻塞的TCP管道的connect操作在5.2.2节描述,UDP的管道,阻塞和非阻塞都在10.1.3节.处于非阻塞模式的管道可以与Selector使用,如4.5节描述的.

4.4.4 方法

非阻塞I/O可被任何从SelectableChannel衍生出来的channel执行.这些包括: DatagramChannel, Pipe.SinkChannel, Pipe.SourceChannel, ServerSocketChannel和SocketChannel.如表格4.1所示.一个管道的阻塞模式通过以下的方法设置和测试:

```
class SelectableChannel implements Channel {
    // Test and set blocking mode
    boolean           isBlocking();
    SelectableChannel configureBlocking(boolean block) throws IOException;
}
```

非阻塞I/O可被4.2.2节定义的read和write方法所执行.与阻塞模式唯一的区别就是非阻塞模式可返回0,表明没有数据传输,如果管道还没有准备好操作,也就是说,没有进来的数据已经为读操作缓冲好,或者没有流出的缓冲空间为写操作可用.

这不是这些方法返回0的唯一原因:如果没有请求实际的传输,它们也可以这样.比如,当读的时候在缓冲区没有空间,或者没有什么可写,在这两种情况下,因为position=limit.这个可以发生在阻塞模式也可以发生在非阻塞模式.

如果一个管道工作于非阻塞模式一个I/O流操作在它上面尝试,将抛出IllegalBlockingModeExceotin.如4.2.4节所示.

4.5 多路复用

多路复用的I/O允许一个线程通过事件提醒的方法在同一时刻管理多个I/O管道: 线程使用一个'selector'对象注册感兴趣的通道,然后调用当一个或者多个通道准备好I/O操作的selector方法,或者在一个指定的超时周期后.

多路复用有点像从网络控制者中通过硬件中断而直接被驱动(FIXME: Multiplexing is somewhat like being directly driven by hardware interrupts from the network controller).

在阻塞模式中管理单个通道和管理(多路复用)多个通道的区别在你图4.7说明.

因为一些原因,Sun没有怎么明确,多路复用必须结合非阻塞I/O使用(*).

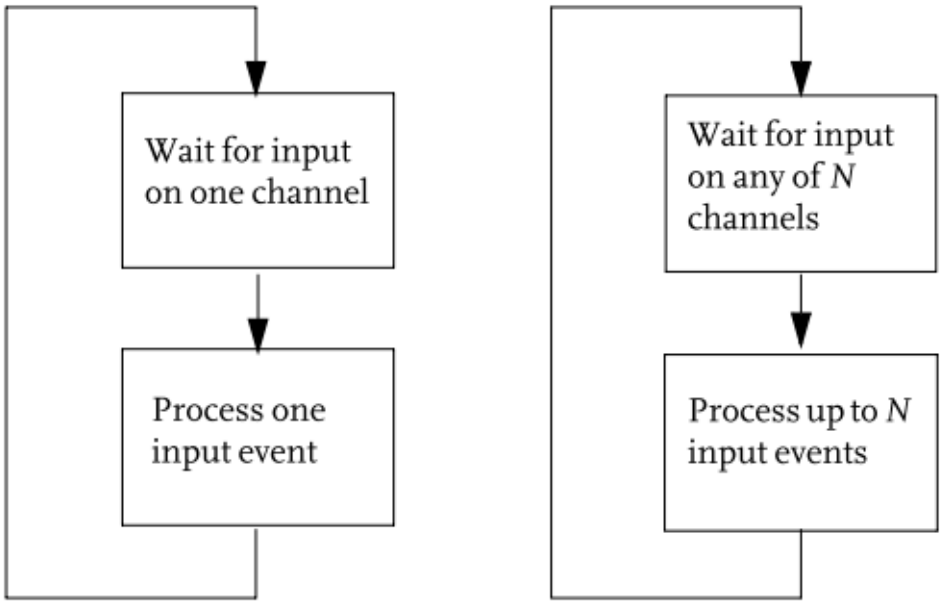


FIGURE 4.7. Blocking I/O *vs.* multiplexing

多路复用作为seelect()API而被Berkeley Sockets和winsock编程者所熟悉.和作为pool()API而被Unix系统V 'streams'编程者所熟悉.

---

* 知道最后的JDK 1.5.这个语义约束似乎是不必要的.我熟悉的(acquaint)所有的Java平台可通过Berkeley Sockets select() API支持阻塞模式的多路复用.winsock WSAAsyncSelect()和WSAEventSelect()APIs需要非阻塞模式,它们中其中有一个似乎被JDK 1.4 Win32实现所使用,如果为true,将解释语义约束和Bug库(FIXME: bug parade)中的BugId 4628289.修复这个bug也似乎使得它可能会去移除这个语义约束.

4.5.1 多路复用和可伸缩性

在Java新I/O的子系统中多路复用提供了可伸缩性.一个应用程序可以在单个线程中使用非阻塞I/O和多路复用去管理多个管道,而不是使用阻塞的I/O,需要一个管道一个线程.这种方式可伸缩性更好当同时有N个数目的线程打开管道时是large的(FIXME: This approach scales much better when the number N of of simultaneously open channels is large),在成百上千的时候,因为它减少了需要的线程的数目从N到N/M,M是每个线程管理的通道的数目.M可以在应用程序中按照需求调整去满足混合资源的约束和它的响应能力的需求.(FIMXE: M can be adjusted within the application as required to suit its mix of resource constraints and its requirements for responsiveness).

我们将在第12章看到利用多路复用的服务器和客户端架构.

4.5.2 多路复用的类

在Java中,I/O多路复用使用Selector类,SelectionKey类,和从SelectableChannel衍生出来的channel类.

如图4.1所示,我们已经在4.4节已经看到过,从SelectableChannel衍生出来的类包括DatagramChannel,Pipe.SinkChannel,Pipe.SourceChannel,ServerSocketChannel和SocketChannel.

4.5.3 Selector

I/O多路复用的核心是Selector类,有打开和关闭一个selector的方法:

```
class Selector {
    static Selector open()  throws IOException;
    boolean         isOpen();
    void close() throws IOException;
}
```

一旦获得一个selector,任何可选择的通道可以使用它注册:

```
class SelectableChannel extends Channel {
    boolean      isRegistered();
    SelectionKey  register (Selector selector, int  operations) throws
ClosedChannelException;
    SelectionKey  register(Selector selector, int operations, Object attachment) throws
ClosedChannelException;
}
```

操作的参数指定了感兴趣的集合:感兴趣的I/O操作的掩码位,这些值在SelectionKey类中定义.

```
class SelectionKey {
    static final int OP_ACCEPT;
    static final int OP_CONNECT;
    static final int OP_READ;
    static final int OP_WRITE;
}
```

比如,注册OP_READ和OP_WRITE可以这样做:

```
Selector  sel = Selector.open();
channel.register(sel, SelectionKey.OP_READ|SelectionKey.OP_WRITE);
```

操作参数的有效值可以从管道通过validOps方法获得:

```
class SelectableChannel extends Channel {
    int  validOps();
}
```

然而,你不应该像下面这样编码:

```
Selector  sel = Selector.open();
channel.register(sel, channel.validOps());
```

特别是在服务器中,一个服务器不会对OP_CONNECT感兴趣,不应该注册它.SocketChannel的validOps总是包含OP_CONNECT,即使对于一个从ServerSocketChannel.accept产生的一个SocketChannel.这样的一个SocketChannel已经准备好连接了,因此OP_CONNECT总是准备好的.这将会引起Selector.select立

即返回,然后硬循环(一直循环)(FIXME: probably hardloop).进一步,无论在客户端或者服务器端,在5.3节讨论,你永远不应该在同一时刻注册OP_CONNECT和OP_WRITE,或者同一时刻注册OP_ACCEPT和OP_READ.

可伸缩的I/O操作将进一步在4.5.4讨论.Selection Keys在4.5.5节讨论.可选的attachment参数在4.5.6节描述.

一个通道可以用一个空的感兴趣的集合注册,也就是说,一个为0的操作参数.这样的话一个通道永远不会被选择.通道注册的感兴趣的集合的key可以在任何时间通过调用SelectionKey.interestOps方法改变.这将会影响下一次的选择操作,将在下个章节讨论.在一些平台上这个方法需要同步防止Selector.select的并发执行,(FIMXE: friends).已经发现最好只从相同的线程中调用Selector.select.(SelectionKey.interestOps和Selector.select之间存在着并发的问题).

4.5.4 可伸缩的I/O操作

注册一个管道意味着想去被通知,当管道准备好一种或者多种类型的I/O操作.一个管道准备好了I/O操作如果在阻塞模式中执,操作不应该被阻塞,或者,在非阻塞模式中,它不会返回0,或者一个null的结果.它可能返回一个结果表明成功了,或者一些错误条件比如流的末尾;或者它抛出一个异常.

明显地,"不会阻塞","不会返回0"等,可能只是意味着"不会阻塞返回0,如果它在同一时刻被执行和select方法返回"(FIXME: if it had been executed at the time the select method returned).计算机不能预测未来.如果一个线程在调用select后立即在socket上操作.从select产生的转变状态可能不再是有效的.健壮的网络程序必须正确处理0,错误或者从I/O操作产生的null结果,即使select方法选择它们作为准备.(FIXME: 这一段有点不明确)

可伸缩的I/O操作和当它们准备好的意义在表格4.4中指明.

表格 4.4 可选择I/O操作

名称	当准好时的意义
OP_ACCEPT	ServerSocketChannel.accept不应该返回null
OP_CONNECT	ServerChannel.finishConnect不应该返回false
OP_READ	在通道上的read操作不应该返回0
OP_WRITE	在通道上的write操作不应该返回0

对于TCP,关于给出的操作的更进一步的信息,在表格5.1,UPD的在表格10.1.

4.5.5 Selection Keys

SelectableChannel.register返回的结果是SelectionKey,简单的代表注册的对象.这个设计允许一个通道被注册到多个slector:每次注册产生一个唯一的注册key,可被用来取消注册而不会影响其它相同通道的注册.

SelectionKey类展示了常量清单如4.5.4节展示的及展示了以下的方法:

```
class SelectionKey {
    // return the channel whose registration produced this key
    SelectableChannel channel();
    // return the selector which produced this key
    Selector selector();
    // return the operations-of-interest set
    int interestOps();
    // return the set of operations which are now ready
    int readyOps();
    boolean isAcceptable(); // OP_ACCEPT ready
```

```
    boolean isConnectable(); // OP_CONNECT ready
    boolean isReadable(); // OP_READ ready
    boolean isWritable(); // OP_WRITE ready
    // Alter the operations-of-interest set
    SelectionKey interestOps(int ops);
    // cancel the registration
    void cancel();
    boolean isValid(); // => not closed/cancelled
    // get/set the attachment
    Object attachment();
    Object attach(Object object);
}
```

4.5.6 SelectionKey附件

一个SelectionKey附件本质上提供了一个为这个key和channel定义的应用程序的上下文对象.它是一个任意的(arbitrary)对象维持从注册的结果产生的selection的关联性,可以被设置,不设置和在随后被检索.

```
class SelectionKey {
    Object attachment(); // get the attachment
    Object attachment(Object object); // set the attachment
}
```

attachment设置方法和上面描述SelectableChannel.register的第三个参数以相同的方式将一个对象附加到key上.一个对象可以被分离(detached)通过附加另一个对象或者null.如果有的话这个方法返回预先附加的附件.其它情况返回null;

4.5.7 Select操作

多路复用的主要的操作是Selector类的select方法,有以下三种形式:

```
class Selector {
    int select() throws IOException;
    int select(long timeout) throws IOException;
    int selectNow()  throws IOException;
}
```

这些方法选择准备好的在它们的感兴趣的集合中至少有1个I/O操作(在4.5.3节定义)的通道.比如:一个注册了{OP_READ}的通道被选择了当读操作不返回0.一个注册了OP_WRITE的感兴趣集合的通道准备好了当写操作不返回0;一个注册了{OP_READ,OP_WRITE}的通道准备好了当一个读操作或者写操作不返回0.

selectNow方法不会阻塞,没有参数的select方法阻塞直到至少有1个通道被选择了或者selector被Selector.wakeup方法异步地唤醒了.select(long timeout)方法阻塞直到至少有1个通道被选择了,或者selector被Selector.wakeup方法异步地唤醒了,或者超时周期到期了,timeout是一个正的毫秒数或者0,无参数的select方法表明无限期的等待.

select操作将准备好的通道的key加入到selected-key集合,然后返回已经准备好的通道的数量,也就是说,被选择的keys的数目可能为0.

注册的keys的集合和当前选择的keys的集合通过下面的方法返回:

```
class Selector {
    Set keys(); // currently registered keys
    Set selectedKeys(); // currently selected keys
}
```

选择的通道通过selected-keys集合获得,可以被独立的处理如下面的例子所展示的:

```
Selector selector = Selector.open();
channel.register(selector, channel.validOps());
int selectedCount = selector.select();
Iterator it = selector.selectedKeys().iterator();
while (it.hasNext()) {
SelectionKey key = (SelectionKey)it.next();
    it.remove();
    SelectableChannel selCh = key.channel();
    if (key.isAcceptable())
    handleAcceptable(key);  // not shown …
    if (key.isConnectable())
    handleConnectable(key); // not shown …
    if (key.isReadable())
    handleReadable(key); // not shown …
    if (key.isWritable())
    handleWritable(key); // not shown …
}
```

注意应用程序必须移除从selected-key集合中的每个SelectionKey,否则它将在下一次在Selector.select之后再次出现或者Selector.selectNow立即返回,因为selector永远不会清理selected-keys集合.(FIXME: As long as this practice is followed),selected-key集合的大小和select操作返回的值一样.

```
selector.select(...) == selector.selectedKeys().size()    (EQ 4.5)
selector.selectNow() == selector.selectedKeys().size()    (EQ 4.6)
```

selected-key集合不完全是不可变的.尽管keys如所展示的可以从selected-key中移除,但是keys不能明确的增加.只有Selector.select和Selector.selectNow可以增加keys到selected-key集合中.如果应用程序尝试这样做,将抛出UnsupportedOperationException.

4.5.8 Selection和超时

如果select操作返回0结果,说明出现了以下的一种或多种情况:

(a) 指定的超时时间过期了.
(b) selector被异步唤醒(见4.6.4节).
(c) 注册的key被异步取消(见4.6.5节).

当selection的结果为0,selected-keys的集合为空,所以可以持有下面的不变式:

```
selector.selectedKeys().size() == 0;
```

和没有任何感兴趣的事情发生在任何注册的通道上.如果只是如果我们假设没有异步操作,通道都超时

(FIXME: the channels have all timed out).在这种情况下,超时的通道的集合就是注册的通道的整个集合,可用的可通过注册的keys的集合的keys方法返回,可用如下面所示的独立的处理超时.

```
int  selectedCount = selector.select(TIMEOUT);
if (selectedCount > 0) {  // process ready channels-处理准备好的通道
Iterator  readyKeys = selector.selectedKeys().iterator();
    while (readyKeys.hasNext()) {
        SelectionKey readyKey = (SelectionKey)it.next():
        it.remove();
        SelectableChannel  readyChannel = readyKey.channel();
        // 'readyChannel' is ready …
    }
} else { // timeout-超时
// Precondition: no asynchronous operations-前提条件：没有异步操作
// process idle channels-处理空闲的通道
    Iterator  it = selector.keys().iterator();
    while (it.hasNext()) {
        SelectionKey       key = (SelectionKey)it.next();
        SelectableChannel  idleChannel = key.channel();
        // 'idleChannel' has timed out - process it … - '空闲通道'已经超时-处理它
    }
}
```

这种技术简单但是有缺陷(flawed).首先,前提条件可能不成立：没有异步操作的假设可能是无效的.其次,selection操作可能不会频繁返回0足以及时去处理空闲的通道(FIXME:the selection operation may not return zero frequently enough for timely processing of idle channels).如果一个通道在每次超时期间准备就绪,超时块永远不会被执行.在剩下的任何超时的通道永远不会被处理(FIXME: If even one channel become ready in each timeout period, the timeout block is never executed, and any timeouts on the remaining channels are never processed.).

我们需要一种技术,工作于任何Selector.select的返回值,允许异步的唤醒.通常,我们不能假设selection操作在整个超时期间内一直阻塞.我们必须明确的追踪消逝的时间:

```
    long timeout   = 180*1000L;
    long startTime = System.currentTimeMillis();
    int  selectCount= selector.select(timeout);
    long endTime   = System.currentTimeMillis();
```

正如Selector.select可能不返回0,我们不能假设之前的一样,所有注册的通道是空闲的.空闲的通道从代数集合(set-algebraic)中获得不同于注册的keys的集合和selected keys的集合,如下:

```
    // copy registered keys - 复制注册的keys
    Set idleKeys = new HashSet(selector.keys());
    // subtract ready keys - 移除准备好的keys
    idleKeys.removeAll(selector.selectedKeys());
    // now iterate over idle keys, - 线程迭代空闲的keys
    // calling SelectionKey.channel() on each …    - 调用SelectionKey.channel()
```

很明显,我们需要从空闲的keys集合中复制注册的key集合而不是直接修改集合.事实上,Selector.keys返回的注册的key集合是不可变的:将一个key从集合中移除的唯一的一种方式就是使用SelectionKey.cancel取消它的注册.

注意注册的key集合不是线程安全的,(FIXME: and therefore neither is the clone;);克隆的也这样;selected-key集合也这也;一个HashSet也这样.如果有必要我们可以使它从任何的集合中变为线程安全的集合:

```
idleKeys = Collections.synchronizedSet(idleKeys);
```

如果因为某些原因我们只是想使用超时时间去探测通道是否已经空闲,而不是对超时周期严格执行感兴趣(FIXME: without being interested in strict enforcement of timeout periods).也就是说,

```
long threshold = 60*1000L;
long timeout   = threshold*3;
```

处理阀值周期内保持空闲的通道:

```
long elapsedTime = endTime - startTime;
if (elapsedTime > threshold) {
    // …
}
```

使用这个超时方法的完整的处理看起来像这样:

```
static final long  THRESHOLD = 60*1000L;// 60 seconds
static final long  TIMEOUT = THRESHOLD*3;

Selector selector = Selector.open();
// …
long startTime = System.currentTimeMillis();
int  readyCount = selector.select(TIMEOUT);
long endTime   = System.currentTimeMillis();
long elapsed   = endTime - startTime;

// process ready channels - 处理准备好的通道
Iterator readyKeys = selector.selectedKeys().iterator();
while (readyKeys.hasNext()) {
    SelectionKey readyKey = (SelectionKey)it.next():
    it.remove();
    SelectableChannelreadyCh = readyKey.channel();
    // 'readyCh' is ready.
    // …
}
if (elapsed >= THRESHOLD) {
    // Process idle channels - 处理空闲的通道
    Set idleKeys = (Set)selector.keys().clone();
    idleKeys.removeAll(selectedKeys);
```

```
    Iterator  it = idleKeys.iterator();
    while (it.hasNext()) {
        SelectableChannel  idleCh =    (SelectableChannel)it.next();
        // 'idleCh' has been idle for >= THRESHOLD ms … -- 'idleCh'已经超时 >= THRESHOLD
ms
    }
}
```

4.6 并发的通道操作

"新I/O"的设计允许多线程去同时操作管道.一个管道可能同时被多个线程读取;可能同时被多个线程写入;也可能同时读取和写入.一个Selector可同时被多个线程操作.I/O操作在某些通道上可能被中断.Selection操作可能被中断或者唤醒.

4.6.1 并发的读和写

多线程可以读取或者写入相同的管道.

如果一个并发读操作正在一个管道上进行,一个新的读操作在管道上将阻塞直到第一个操作完成.类似的,如果一个并发写操作在一个管道上进行,一个新的写操作在管道上阻塞直到第一个操作完成.在两种情况下,无论通道是什么阻塞模式,这都将会发生,因为阻塞是通过Java对象同步强制执行的而不是底层的I/O系统(*).

一个通道可以或者不可以支持并发的读和写: 如果这样的话,一个读操作和一个写操作可能或不可能并发处理(没有阻塞),依赖于通道的类型.

然而,如我们在4.3.13节所说,缓冲区不是线程安全的,在一个通道上的并发I/O操作只能使用相同的缓冲区,如果执行适当的同步,比如,通过缓冲区自身同步.

---

* Socket通道支持并发的读和写没有阻塞.文件通道阻塞支持,部分是平台依赖的.

4.6.2 中断

如4.2.2节所描述的,一个线程在一个通道上在读或者写操作中被阻塞,实现了InterruptibleChannel接口可以通过调用Thread.interrupt方法被中断.

一个线程在一个Selector上的select操作中被阻塞可通过调用Thread.interrupt方法中断.

InterruptibleChannels和Selector的中断的语义是不同的: 见表格4.5.

4.6.3 异步关闭

一个可中断的通道可被异步的关闭,任何在一个读或者写操作上阻塞的线程在关闭的通道上会抛出AsynchronousCloseException.如何任何的selection操作包括关闭的通道在处理中,见4.6.5节.

类似的,如果一个selector被异步关闭,任何在Selector的select操作中阻塞的线程的行为和woken up一样(FIXME: any thread blocked in a select operation on a Selector behaves as though woken up) 如4.6.4节描述的.关闭的异常可通道Selecor.isOpen的结果false探测.

InterruptibleChannels和Selector的异步关闭语义是不同的,见表格4.5.

4.6.4 唤醒

一个selection操作可以通过Selector.wakeup方法被异步"唤醒".如果任何线程在那个selector上并发阻塞在selection操作中,第一个被阻塞的线程将立即返回.另外,wakeup引起下一次的selection操作立即返回,无论超时的时间, 包括selector.selectNow(不会阻塞)

wakeup的作用是很清晰的,通过任何的selection操作,包括Selector.selectNow.为了在Java中表达,下面的代码立即返回:

```
selector.wakeup();
selector.select(timeout); // no block here--不会阻塞
```

但是下面的将会阻塞:

```
selector.wakeup();
selector.selectNow();
selector.select(timeout); // blocks here --这里不会阻塞
```

表格 4.5 异步操作的语义

阻塞操作	异步操作	语义
InterruptibleChannel上的read或者write	Thread.interrupt	通道被关闭. Channel.isOpen == false. 阻塞的线程抛出ClosedByInterruptException. Thread.isInterrupted == true.
	在InterruptilbeChannel上的关闭	通道被关闭. Channel.isOpen == false. 阻塞的线程抛出AsynchronousCloseException. Thread.isInterrupted == false
Selector.select	Thread.interrupt	Selector.wakeup被调用. selector没有关闭. Selector.isOpen == true. 阻塞的线程立即返回. Thread.isInterrupted == true.
	Selector.close	Selector.wakeup被调用. selector没有关闭. Selector.isOpen == false. 阻塞的线程立即返回. Thread.isInterrupted == false.
	Selector.wakeup	selector没有关闭. Selector.isOpen == true. 第一个任何阻塞的线程立即返回, 且Thread.isInterrupted == false.

		如果没有阻塞的线程了.下一次的select操作立即返回.

4.6.6 Win 32 和 JDK 1.4.0

4.7 新I/O中的异常

异常可以在通道和缓冲区操作期间出现(arise),它们的含义,在表格4.6中列出.

名称	意义	受检异常(C)/未检查异常(U)
AlreadyConnectedException	由SocketChannel.connect抛出如果通道已经连接	U
AsynchronousCloseException	由任何在通道上阻塞的操作抛出,当它被另一线程关闭	C
BufferOverflowException	由任何相对的Buffer.put操作抛出,如果已经达到缓冲区的极限	U
BufferUnderflowException	由任何相对的Buffer.get操作抛出,如果已经达到缓冲区的极限	U
CancelledKeyException	由任何尝试去使用一个取消的SelectionKey抛出	U
CharacterCodingException	当一个字符编码或者解码错误发生时抛出	C
ClosedByInterruptException	由一个在通道上阻塞的操作抛出,如果正在调用的线程被另一个线程用Thread.interrupt中断.通道现在是关闭的,接收到异常的线程的Thread.isInterrupted为true. FileChannel.lock不抛出:见FileLockInterruptionException	U
ClosedChannelException	由任意通道的操作抛出,如果通道已经关闭,或者由SocketChannel.write抛出,如果套接字已经停止输出, 或者SocketChannel.read抛出,如果套接字已经停止输入.	U
ClosedSelectorException	由任何Selector的方法抛出,如果selector已经被关闭	U
ConnectionPendingException	由SocketChannel.connect抛出,如果通道已经有一个等待的连接	
FileLockInterruptionException	由FileChannel.lock抛出,如果正在调用的线程被另一线程(使用Thread.interrupt)中断.接收到异常的	C

	线程的Thread.isInterrupted为true.	
IllegalBlockingModeException	由任意的在通道上的指定为阻塞模式的操作抛出,如果通道不处于正确的模式, 也就是说,一个从通道上获得的流上的read和write,如果通道处于非阻塞模式中	U
IllegalCharsetNameException	当使用非法格式的字符集名称,如JDK文档中定义的.	U
InvalidMarkException	由一个Buffer上的任意的reset操作抛出,如果标记未定义	U
MalformedInputException	当输入的字节顺序对于给定的字符集不是合法的时候抛出, 或者输入字符顺序不是合法的16位Unicode顺序	U
NoConnectionPendingException	由SocketChannel.finishConnect抛出,如果没有连接在等待,也就是说, SocketChannel.connect没有被成功调用	U
NonReadableChannelException	由channel的任意的read方法抛出,如果channel没有打开用来读取.	U
NonWritableChannelException	由channel的任意的write方法抛出,如果channel没有打开用来写.	U
NotYetBoundException	由在还没有绑定到一个本地端口的ServerSocketChannel上的任意的I/O操作抛出	U
NotYetConnectedException	由在还没有连接的SocketChannel上的任意的I/O操作抛出	U
OverlappingFileLockException	由FileChannel.lock或者FileChannel.tryLock抛出,如果指定的范围与一个范围重叠(这个范围可以由同一个JVM锁定或者在同一JVM中的另一线程已经在等待锁)	C
ReadOnlyBufferException	由任意的read,put或者compat操作抛出,也就是说,	U
UnmappableCharacterException		C
UnresolvedAddressException		U
UnsupportedAddressTypeException		U
UnsupportedCharsetException		U