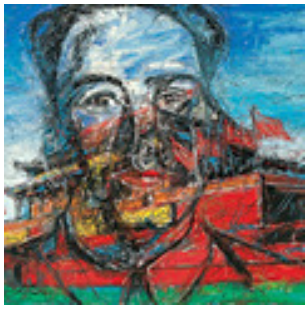


NIO学习总结

作者: zhangshixi http://zhangshixi.javaeye.com

NIO学习总结

目 录

1. Core Java

1.1 NIO学习系列：核心概念及基本读写

发表时间: 2010-05-31

1. 引言

I/O流或者输入/输出流指的是计算机与外部世界或者一个程序与计算机的其余部分的之间的接口。新的输入/输出(NIO)库是在JDK 1.4中引入的。NIO弥补了原来的I/O的不足，它在标准Java代码中提供了高速的、面向块的I/O。

原来的I/O库与NIO最重要的区别是数据打包和传输的方式的不同，原来的 I/O 以流 的方式处理数据，而 NIO 以块 的方式处理数据。

面向流的I/O系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。为流式数据创建过滤器非常容易。链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的。不利的一面是，面向流的I/O通常相当慢。

NIO与原来的I/O有同样的作用和目的，但是它使用块I/O的处理方式。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多。但是面向块的I/O缺少一些面向流的I/O所具有的优雅性和简单性。

2. 从一个例子开始

下面我们从一个简单的使用IO和NIO读取一个文件中的内容为例，来进入NIO的学习之旅。

使用IO来读取指定文件中的前1024字节并打印出来：

```
/**
 * 使用IO读取指定文件的前1024个字节的内容。
 * @param file 指定文件名称。
 * @throws java.io.IOException IO异常。
 */
public void ioRead(String file) throws IOException {
    FileInputStream in = new FileInputStream(file);
    byte[] b = new byte[1024];
    in.read(b);
    System.out.println(new String(b));
}

/**
 * 使用NIO读取指定文件的前1024个字节的内容。
 * @param file 指定文件名称。
```

```
 * @throws java.io.IOException IO异常。
 */
public void nioRead(String file) throws IOException {
    FileInputStream in = new FileInputStream(file);
    FileChannel channel = in.getChannel();

    ByteBuffer buffer = ByteBuffer.allocate(1024);
    channel.read(buffer);
    byte[] b = buffer.array();
    System.out.println(new String(b));
}
```

从上面的例子中可以看出，NIO以通道Channel和缓冲区Buffer为基础来实现面向块的IO数据处理。下面将讨论并学习NIO 库的核心概念以及从高级的特性到底层编程细节的几乎所有方面。

3. 核心概念：通道和缓冲区

1) 概述：

通道和缓冲区是NIO中的核心对象，几乎在每一个I/O操作中都要使用它们。

通道Channel是对原I/O包中的流的模拟。到任何目的地(或来自任何地方)的所有数据都必须通过一个Channel对象。

缓冲区Buffer实质上是一个容器对象。发送给一个通道的所有对象都必须首先放到缓冲区中；同样地，从通道中读取的任何数据都要读到缓冲区中。

2) 缓冲区：

Buffer是一个容器对象，它包含一些要写入或者刚读出的数据。在NIO中加入Buffer对象，体现了新库与原I/O的一个重要区别。在面向流的I/O中，您将数据直接写入或者将数据直接读到Stream对象中。

在NIO库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问NIO中的数据，您都是将它放到缓冲区中。

缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

最常用的缓冲区类型是ByteBuffer。 一个ByteBuffer可以在其底层字节数组上进行get/set操作(即字节的获取和设置)。

ByteBuffer不是NIO中唯一的缓冲区类型。事实上，对于每一种基本Java类型都有一种缓冲区类型：

ByteBuffer

CharBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

DoubleBuffer

每一个Buffer类都是Buffer接口的一个实例。 除了ByteBuffer， 每一个Buffer类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准I/O操作都使用ByteBuffer，所以它具有所有共享的缓冲区操作以及一些特有的操作。

下面的UseFloatBuffer列举了使用类型化的缓冲区FloatBuffer的一个应用例子：

```
/**
 * 使用 float 缓冲区。
 * @version 1.00 2010-5-19, 10:30:59
 * @since 1.5
 * @author ZhangShixi
 */
public class UseFloatBuffer {

    public static void main(String[] args) {
        // 分配一个容量为10的新的 float 缓冲区
        FloatBuffer buffer = FloatBuffer.allocate(10);
        for (int i = 0; i < buffer.capacity(); i++) {
            float f = (float) Math.sin((((float) i) / 10) * (2 * Math.PI));
            buffer.put(f);
        }
        // 反转此缓冲区
        buffer.flip();

        // 告知在当前位置和限制之间是否有元素
        while (buffer.hasRemaining()) {
            float f = buffer.get();
            System.out.println(f);
        }
    }
}
```

3) 通道：

Channel是对原I/O包中的流的模拟，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。

正如前面提到的，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。

通道与流的不同之处在于通道是双向的。而流只是在一个方向上移动(一个流必须是InputStream或者OutputStream的子类)， 而通道可以用于读、写或者同时用于读写。

因为它们是双向的，所以通道可以比流更好地反映底层操作系统的真实情况。特别是在UNIX模型中，底层操作系统通道是双向的。

4. 从理论到实践：NIO中的读和写

1) 概述：

读和写是I/O的基本过程。从一个通道中读取很简单：只需创建一个缓冲区，然后让通道将数据读到这个缓冲区中。写入也相当简单：创建一个缓冲区，用数据填充它，然后让通 道用这些数据来执行写入操作。

2) 从文件中读取：

如果使用原来的I/O，那么我们只需创建一个FileInputStream并从它那里读取。而在NIO中，情况稍有不同：我们首先从FileInputStream获取一个FileChannel对象，然后使用这个通道来读取数据。

在NIO系统中，任何时候执行一个读操作，您都是从通道中读取，但是您不是直接从通道读取。因为所有数据最终都驻留在缓冲区中，所以您是从通道读到缓冲区中。

因此读取文件涉及三个步骤：

(1) 从FileInputStream获取Channel。

(2) 创建Buffer。

(3) 将数据从Channel读到Buffer 中。

现在，让我们看一下这个过程。

```
// 第一步是获取通道。我们从 FileInputStream 获取通道：
FileInputStream fin = new FileInputStream( "readandshow.txt" );
FileChannel fc = fin.getChannel();
// 下一步是创建缓冲区：
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
// 最后，需要将数据从通道读到缓冲区中：
fc.read( buffer );
```

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

统计机制的内容。

3) 写入文件：

在 NIO 中写入文件类似于从文件中读取。

```
// 首先从 FileOutputStream 获取一个通道：
FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();
// 下一步是创建一个缓冲区并在其中放入一些数据，这里，用message来表示一个持有数据的数组。
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
for (int i=0; i<message.length; ++i) {
     buffer.put( message[i] );
}
buffer.flip();
// 最后一步是写入缓冲区中：
fc.write( buffer );
```

注意在这里同样不需要告诉通道要写入多数据。缓冲区的内部统计机制会跟踪它包含多少数据以及还有多少数据要写入。

4) 读写结合：

下面的示例将展示使用读写结合，将一个文件的所有内容拷贝到另一个文件中。

```
/**
 * 将一个文件的所有内容拷贝到另一个文件中。
 *
 * CopyFile.java 执行三个基本操作：
 * 首先创建一个 Buffer，然后从源文件中将数据读到这个缓冲区中，然后将缓冲区写入目标文件。
 * 程序不断重复 — 读、写、读、写 — 直到源文件结束。
 *
 * @version 1.00 2010-5-19, 10:49:46
 * @since 1.5
 * @author ZhangShixi
 */
public class CopyFile {

    public static void main(String[] args) throws Exception {
```

```
        String infile = "C:\\copy.sql";
        String outfile = "C:\\copy.txt";

        // 获取源文件和目标文件的输入输出流
        FileInputStream fin = new FileInputStream(infile);
        FileOutputStream fout = new FileOutputStream(outfile);

        // 获取输入输出通道
        FileChannel fcin = fin.getChannel();
        FileChannel fcout = fout.getChannel();

        // 创建缓冲区
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        while (true) {
            // clear方法重设缓冲区，使它可以接受读入的数据
            buffer.clear();

            // 从输入通道中将数据读到缓冲区
            int r = fcin.read(buffer);

            // read方法返回读取的字节数，可能为零，如果该通道已到达流的末尾，则返回-1
            if (r == -1) {
                break;
            }

            // flip方法让缓冲区可以将新读入的数据写入另一个通道
            buffer.flip();

            // 从输出通道中将数据写入缓冲区
            fcout.write(buffer);
        }
    }
}
```

后续： 在下一篇文章中，会具体介绍缓冲区Buffer的内部实现机制，以理解缓冲区如何能够内部地管理自己的资源。有兴趣的可以共同学习、讨论。

附件下载:

- nio.zip (1.7 KB)
- dl.javaeye.com/topics/download/c33b7130-c1a3-3b13-9f3e-20397b802452

1.2 NIO学习系列：缓冲区内部实现机制

发表时间: 2010-06-02

接上一篇NIO学习系列：核心概念及基本读写 ，本文继续探讨和学习缓冲区的内部实现机制。

5. 缓冲区内部实现

从上面对NIO的学习中，我们知道每一个缓冲区都有复杂的内部统计机制，它会跟踪已经读了多少数据以及还有多少空间可以容纳更多的数据，以便我们对缓冲区的操作。在本节我们就将学习NIO的两个重要的缓冲区组件：状态变量和访问方法。虽然NIO的内部统计机制初看起来可能很复杂，但是您很快就会看到大部分的实际工作都已经替您完成了。您只需像平时使用字节数组和索引变量一样进行操作即可。

1) 状态变量：

状态变量是前一节中提到的"内部统计机制"的关键。 每一个读/写操作都会改变缓冲区的状态。通过记录和跟踪这些变化，缓冲区就可能够内部地管理自己的资源。

每一种Java基本类型的缓冲区都是抽象类Buffer的子类，从Buffer的源代码中可以发现，它定义了三个私有属性：

```
private int position = 0;
private int limit;
private int capacity;
```

实际上，这三个属性值可以指定缓冲区在任意时刻的状态和它所包含的数据。

我们知道，每一个基本类型的缓冲区底层实际上就是一个该类型的数组。如在ByteBuffer中，有：

```
final byte[] hb;
```

在从通道读取时，所读取的数据将放被到底层的数组中；同理，向通道中写入时，将从底层数组中将数据写入通道。下面我们来具体介绍这三个变量的作用：

a) position

position变量跟踪了向缓冲区中写入了多少数据或者从缓冲区中读取了多少数据。

更确切的说，当您从通道中读取数据到缓冲区中时，它指示了下一个数据将放到数组的哪一个元素中。比如，如果您从通道中读三个字节到缓冲区中，那么缓冲区的position将会设置为3，指向数组中第4个元素。反之，当您从缓冲区中获取数据进行写通道时，它指示了下一个数据来自数组的哪一个元素。比如，当您从缓冲区写了5个字节到通道中，那么缓冲区的 position 将被设置为5，指向数组的第六个元素。

b) limit

limit变量表明还有多少数据需要取出(在从缓冲区写入通道时)，或者还有多少空间可以放入数据(在从通道读入缓冲区时)。

position总是小于或者等于limit。

c) capacity

capacity变量表明可以储存在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—或者至少是指定了准许我们使用的底层数组的容量。

limit总是小于或者等于capacity。

d) 举例说明：

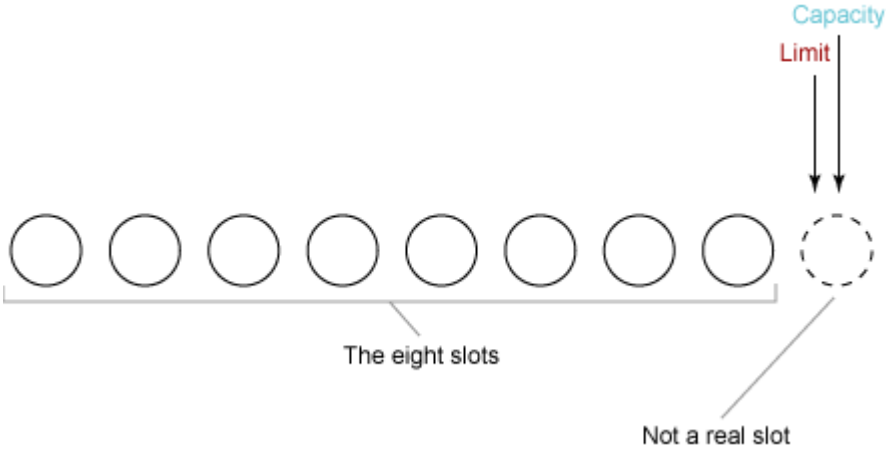
下面我们就以数据从一个输入通道拷贝到一个输出通道为例，来详细分析每一个变量，并说明它们是如何协同工作的：

初始变量：

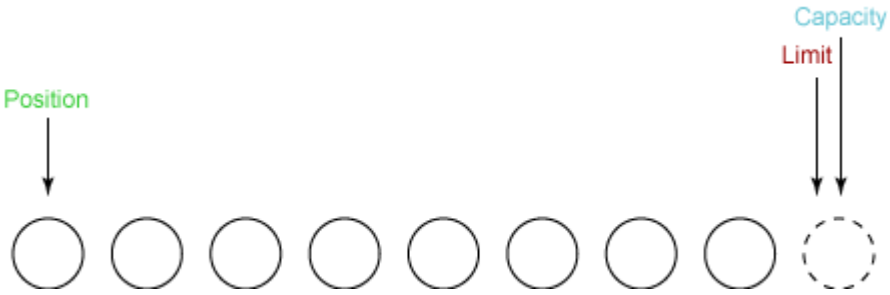
我们首先观察一个新创建的缓冲区，以ByteBuffer为例，假设缓冲区的大小为8个字节，ByteBuffer初始状态如下：

The eight slots

回想一下 ，limit决不能大于capacity，此例中这两个值都被设置为8。我们通过将它们指向数组的尾部之后(第8个槽位)来说明这点。



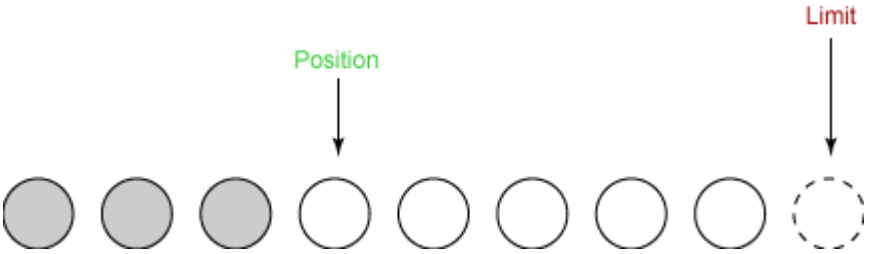
我们再将position设置为0。表示如果我们读一些数据到缓冲区中，那么下一个读取的数据就进入 slot 0。如果我们从缓冲区写一些数据，从缓冲区读取的下一个字节就来自slot 0。position设置如下所示：



由于缓冲区的最大数据容量capacity不会改变，所以我们在下面的讨论中可以忽略它。

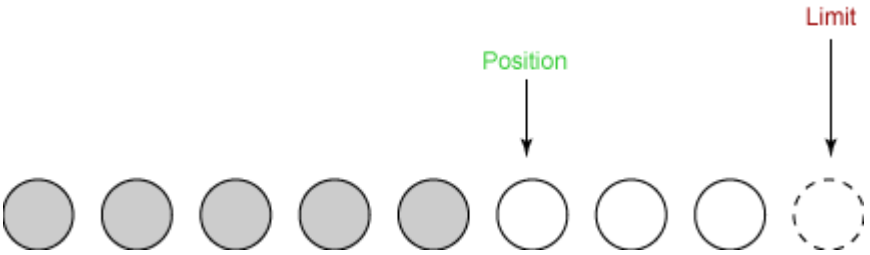
第一次读取：

现在我们可以开始在新创建的缓冲区上进行读/写操作了。首先从输入通道中读一些数据到缓冲区中。第一次读取得到三个字节。它们被放到数组中从position开始的位置，这时position被设置为0。读完之后，position就增加到了3，如下所示，limit没有改变。



第二次读取：

在第二次读取时，我们从输入通道读取另外两个字节到缓冲区中。这两个字节储存在由position所指定的位置上， position因而增加2，limit没有改变。



flip：

现在我们要将数据写到输出通道中。在这之前，我们必须调用flip()方法。 其源代码如下：

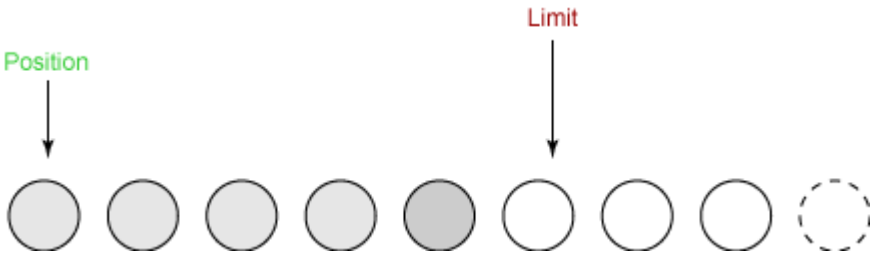
```
public final Buffer flip() {
    limit = position;
    position = 0;
    mark = -1;
    return this;
}
```

这个方法做两件非常重要的事：

i 它将limit设置为当前position。

ii 它将position设置为0。

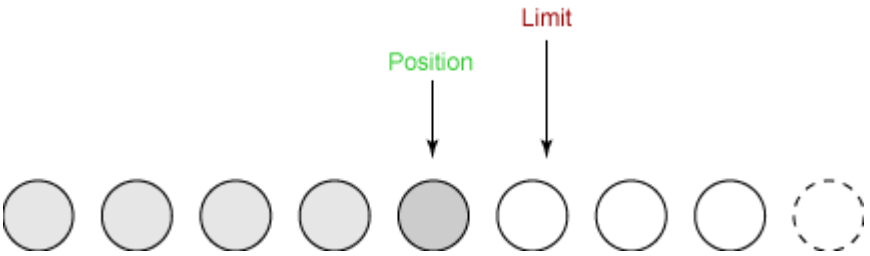
上一个图显示了在flip之前缓冲区的情况。下面是在flip之后的缓冲区：



我们现在可以将数据从缓冲区写入通道了。position被设置为0，这意味着我们得到的下一个字节是第一个字节。limit已被设置为原来的position，这意味着它包括以前读到的所有字节，并且一个字节也不多。

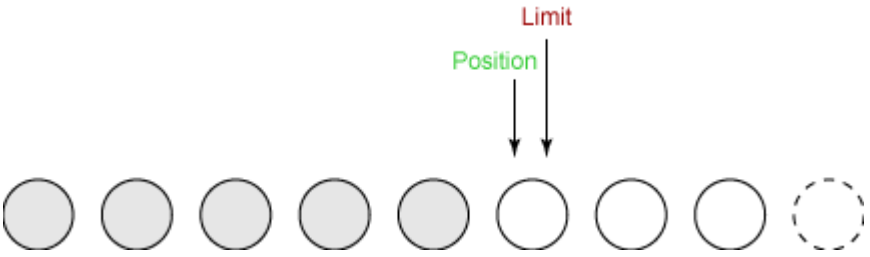
第一次写入：

在第一次写入时，我们从缓冲区中取四个字节并将它们 写入输出通道。这使得position增加到4，而limit不变，如下所示：



第二次写入：

我们只剩下一个字节可写了。limit在我们调用flip()时被设置为5，并且position不能超过limit。 所以最后一次写入操作从缓冲区取出一个字节并将它写入输出通道。这使得position增加到5，并保持limit不变，如下所示：



clear：

最后一步是调用缓冲区的clear()方法。这个方法重设缓冲区以便接收更多的字节。其源代码如下：

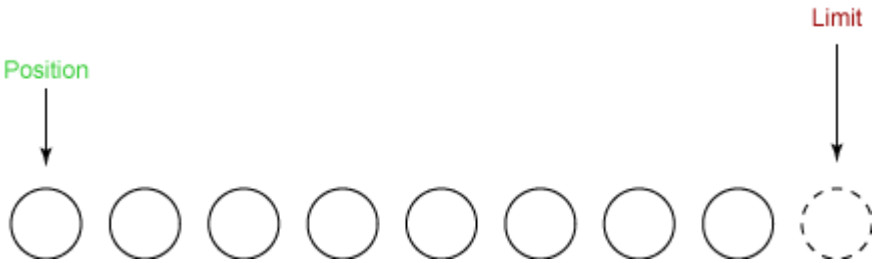
```
public final Buffer clear() {
    osition = 0;
    limit = capacity;
    mark = -1;
    return this;
}
```

clear做两种非常重要的事情：

i 它将limit设置为与capacity相同。

ii 它设置position为0。

下图显示了在调用clear()后缓冲区的状态， 此时缓冲区现在可以接收新的数据了。



2) 访问方法：

到目前为止，我们只是使用缓冲区将数据从一个通道转移到另一个通道。然而，程序经常需要直接处理数据。例如，您可能需要将用户数据保存到磁盘。在这种情况下，您必须将这些数据直接放入缓冲区，然后用通道将缓冲区写入磁盘。 或者，您可能想要从磁盘读取用户数据。在这种情况下，您要将数据从通道读到缓冲区中，然后检查缓冲区中的数据。

实际上，每一个基本类型的缓冲区都为我们提供了直接访问缓冲区中数据的方法，我们以ByteBuffer为例，分析如何使用其提供的get()和put()方法直接访问缓冲区中的数据。

a) get()

ByteBuffer类中有四个get()方法：

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

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

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

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

b) put()

ByteBuffer类中有五个put()方法：

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

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

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

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

c) 类型化的 get() 和 put() 方法

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

getByte()
getChar()
getShort()
getInt()
getLong()
getFloat()
getDouble()
putByte()
putChar()
putShort()

putInt()
putLong()
putFloat()
putDouble()

事实上，这其中的每个方法都有两种类型：一种是相对的，另一种是绝对的。它们对于读取格式化的二进制数据（如图像文件的头部）很有用。

3) 如何使用？

下面的内部循环概括了使用缓冲区将数据从输入通道拷贝到输出通道的过程。

```
while (true) {
     buffer.clear();
     int r = fcin.read( buffer );

     if (r==-1) {
       break;
     }

     buffer.flip();
     fcout.write( buffer );
}
```

read()和write()调用得到了极大的简化，因为许多工作细节都由缓冲区完成了。clear()和flip()方法用于让缓冲区在读和写之间切换。

后续： 在下一篇文章中，会具体介绍连网和非阻塞IO的原理及使用，这也是NIO重要的一部分。有兴趣的可以共同学习、讨论。

1.3 NIO学习系列：连网和异步IO

发表时间: 2010-06-04

接前两篇关于NIO系列的学习文章：核心概念及基本读写 及缓冲区内部实现机制 ，本文继续探讨和学习连网和非阻塞IO相关的内容。

6. 连网和异步IO

1) 概述：

连网是学习异步I/O的很好基础，而异步I/O对于在Java语言中执行任何输入/输出过程的人来说，无疑都是必须具备的知识。NIO中的连网与NIO中的其他任何操作没有什么不同，它依赖通道和缓冲区，而您通常使用InputStream和OutputStream来获得通道。

本节首先介绍异步I/O的基础：它是什么以及它不是什么，然后转向更实用的、程序性的例子。

2) 异步 I/O

异步I/O是一种“没有阻塞地读写数据”的方法。通常，在代码进行read()调用时，代码会阻塞直至有可供读取的数据。同样， write()调用将会阻塞直至数据能够写入。 但异步I/O调用不会阻塞。相反，您可以注册对特定I/O事件的兴趣：如可读的数据的到达、新的套接字连接等等，而在发生这样的事件时，系统将会告诉您。

异步I/O的一个优势在于，它允许您同时根据大量的输入和输出执行I/O。同步程序常常要求助于轮询，或者创建许许多多的线程以处理大量的连接。使用异步I/O，您可以监听任何数量的通道上的事件，不用轮询，也不用额外的线程。

我们来看一个基于非阻塞I/O的服务器端的处理流程，它接受网络连接并向它们回响它们可能发送的数据。在这里假设它能同时监听多个端口，并处理来自所有这些端口的连接。下面是其主方法：

```
private void execute () throws IOException {
    // 创建一个新的选择器
    Selector selector = Selector.open();

    // 打开在每个端口上的监听，并向给定的选择器注册此通道接受客户端连接的I/O事件。
    for (int i = 0; i < ports.length; i++) {
        // 打开服务器套接字通道
        ServerSocketChannel ssc = ServerSocketChannel.open();
        // 设置此通道为非阻塞模式
        ssc.configureBlocking(false);
        // 绑定到特定地址
        ServerSocket ss = ssc.socket();
```

```
        InetSocketAddress address = new InetSocketAddress(ports[i]);
        ss.bind(address);
        // 向给定的选择器注册此通道的接受连接事件
        ssc.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("Going to listen on " + ports[i]);
    }

    while (true) {
        // 这个方法会阻塞，直到至少有一个已注册的事件发生。
        // 当一个或者更多的事件发生时，此方法将返回所发生的事件的数量。
        int num = selector.select();

        // 迭代所有的选择键，以处理特定的I/O事件。
        Set<SelectionKey> selectionKeys = selector.selectedKeys();
        Iterator<SelectionKey> iter = selectionKeys.iterator();

        SocketChannel sc;
        while (iter.hasNext()) {
            SelectionKey key = iter.next();

            if ((key.readyOps() & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT) {
                // 接受服务器套接字撒很能够传入的新的连接，并处理接受连接事件。
                ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
                sc = ssc.accept();
                // 将新连接的套接字通道设置为非阻塞模式
                sc.configureBlocking(false);

                // 接受连接后，在此通道上从新注册读取事件，以便接收数据。
                sc.register(selector, SelectionKey.OP_READ);
                // 删除处理过的选择键
                iter.remove();

                System.out.println("Got connection from " + sc);
            } else if ((key.readyOps() & SelectionKey.OP_READ) == SelectionKey.OP_READ) {
                // 处理读取事件，读取套接字通道中发来的数据。
                sc = (SocketChannel) key.channel();
```

```
                // 读取数据
                int bytesEchoed = 0;
                while (true) {
                    echoBuffer.clear();
                    int r = sc.read(echoBuffer);

                    if (r == -1) {
                        break;
                    }

                    echoBuffer.flip();
                    sc.write(echoBuffer);

                    bytesEchoed += r;
                }
                System.out.println("Echoed " + bytesEchoed + " from " + sc);
                // 删除处理过的选择键
                iter.remove();
            }
        }
    }
}
```

下面我们就此例来一步一步的学习异步IO的相关知识。

3) Selectors

Selector是异步I/O中的核心对象。Selector就是您注册对各种I/O事件的兴趣的地方，而且当那些事件发生时，就是这个对象告诉您所发生的事件。所以，我们需要做的第一件事就是创建一个Selector：

```
Selector selector = Selector.open();
```

然后，我们将对不同的通道对象调用register()方法，以便注册我们对这些对象中发生的I/O事件的兴趣。register()的第一个参数就是这个Selector对象。

4) 打开一个ServerSocketChannel

在服务端为了接收连接，我们需要一个ServerSocketChannel。 事实上，我们要监听的每一个端口都需要有一个ServerSocketChannel。对于每一个端口，我们打开一个ServerSocketChannel， 如下所示：

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking( false );

ServerSocket ss = ssc.socket();
InetSocketAddress address = new InetSocketAddress( ports[i] );
ss.bind( address );
```

第一行创建一个新的ServerSocketChannel，最后三行将它绑定到给定的端口。第二行将ServerSocketChannel设置为非阻塞的。我们必须对每一个要使用的套接字通道调用这个方法，否则异步I/O就不能工作。

5) 选择键

下一步是将新打开的ServerSocketChannels注册到Selector上。为此我们使用ServerSocketChannel.register()方法，如下所示：

```
SelectionKey key = ssc.register( selector, SelectionKey.OP_ACCEPT );
```

register()方法的第一个参数总是这个Selector。第二个参数是OP_ACCEPT，这里它指定我们想要监听accept事件，也就是在新的连接建立时所发生的事件。这是适用于ServerSocketChannel的唯一事件类型。

请注意对register()的调用的返回值。SelectionKey代表这个通道在此Selector上的这个注册。当某个Selector通知您某个传入事件时，它是通过提供对应于该事件的SelectionKey来进行的。SelectionKey还可以用于取消通道的注册。

6) 内部循环

现在已经注册了我们对一些 I/O 事件的兴趣，下面将进入主循环。使用 Selectors 的几乎每个程序都像下面这样使用内部循环：

```
int num = selector.select();

Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();

while (it.hasNext()) {
     SelectionKey key = (SelectionKey)it.next();
     // ... 处理I/O事件...
}
```

首先，我们调用Selector的select()方法。这个方法会阻塞，直到至少有一个已注册的事件发生。当一个或者更多的事件发生时，select()方法将返回所发生的事件的数量。

接下来，我们调用Selector的selectedKeys()方法，它返回发生了事件的SelectionKey对象的一个集合。

我们通过迭代SelectionKeys并依次处理每个SelectionKey来处理事件。对于每一个SelectionKey，您必须确定发生的是什么I/O事件，以及这个事件影响哪些I/O对象。

7) 监听新连接

程序执行到这里，我们仅注册了ServerSocketChannel， 并且仅注册它们“接收”事件。为确认这一点，我们对SelectionKey调用readyOps()方法，并检查发生了什么类型的事件：

```
if ((key.readyOps() & SelectionKey.OP_ACCEPT)
     == SelectionKey.OP_ACCEPT) {
     // ...
}
```

可以肯定地说，readOps()方法告诉我们该事件是新的连接。

8) 接受新的连接

因为我们知道这个服务器套接字上有一个传入连接在等待，所以可以安全地接受它；也就是说，不用担心accept()操作会阻塞：

```
ServerSocketChannel ssc = (ServerSocketChannel)key.channel();
SocketChannel sc = ssc.accept();
```

下一步是将新连接的SocketChannel配置为非阻塞的。而且由于接受这个连接的目的是为了读取来自套接字的数据，所以我们还必须将SocketChannel注册到Selector上，如下所示：

```
sc.configureBlocking( false );
SelectionKey newKey = sc.register( selector, SelectionKey.OP_READ );
```

注意我们使用register()的OP_READ参数，将SocketChannel注册用于“读取”而不是“接受”新连接。

9) 删除处理过的SelectionKey

在处理SelectionKey之后，我们几乎可以返回主循环了。但是我们必须首先将处理过的SelectionKey从选定的键集合中删除。如果我们没有删除处理过的键，那么它仍然会在主集合中以一个激活的键出现，这会导致我们尝试再次处理它。我们调用迭代器的remove()方法来删除处理过的SelectionKey：

```
it.remove();
```

现在我们可以返回主循环并接受从一个套接字中传入的数据(或者一个传入的I/O事件)了。

10) 传入的I/O

当来自一个套接字的数据到达时，它会触发一个I/O事件。这会导致在主循环中调用Selector.select()，并返回一个或者多个I/O事件。这一次， SelectionKey将被标记为OP_READ事件，如下所示：

```
} else if ((key.readyOps() & SelectionKey.OP_READ)
     == SelectionKey.OP_READ) {
     // Read the data
     SocketChannel sc = (SocketChannel)key.channel();
     // ...
}
```

与以前一样，我们取得发生I/O事件的通道并处理它。在本例中，由于这是一个echo server，我们只希望从套接字中读取数据并马上将它发送回去。关于这个过程的细节，请参见附件中的源代码 (MultiPortEcho.java)。

11) 回到主循环

每次返回主循环，我们都要调用select的Selector()方法，并取得一组SelectionKey。每个键代表一个I/O事件。我们处理事件，从选定的键集中删除SelectionKey，然后返回主循环的顶部。

说明： 这个程序有点过于简单，因为它的目的只是展示异步I/O所涉及的技术。在现实的应用程序中，您需要通过将通道从Selector中删除来处理关闭的通道。而且您可能要使用多个线程。这个程序可以仅使用一个线程，因为它只是一个演示，但是在现实场景中，创建一个线程池来负责I/O事件处理中的耗时部分会更有意义。

后续： 到此，我们已学习了NIO的核心内容，在下一篇文章中，会介绍NIO提供的一些其他特性，如：缓冲区的分片、包装，分散和聚集、文件锁定、字符集等知识。有兴趣的可以共同学习、讨论。

附件下载:

- nio.zip (1.7 KB)
- dl.javaeye.com/topics/download/879b8e62-d664-333d-914f-827f3c6f0e9b

1.4 NIO学习系列：缓冲区更多特性及分散/聚集IO

发表时间: 2010-06-05

在前面三篇关于NIO系列的学习文章：核心概念及基本读写 、缓冲区内部实现机制 、连网和异步IO 中，我们已经介绍了NIO的核心知识，本文继续探讨和学习缓冲区更多特性及分散/聚集IO等相关内容。

7. 缓冲区更多内容

到目前为止，我们已经学习了使用缓冲区进行日常工作所需要掌握的大部分内容。我们所举的例子也没怎么超出标准的读/写过程种类，在原来的I/O中可以像在NIO中一样容易地实现这样的标准读写过程。

在本节将讨论使用缓冲区的一些更复杂的方面，比如缓冲区分配、包装和分片。我们还会讨论NIO带给Java平台的一些新功能。我们将学如何创建不同类型的缓冲区以达到不同的目的，如可保护数据不被修改的“只读缓冲区”，和直接映射到底层操作系统缓冲区的“直接缓冲区”，以及如何在 NIO 中创建内存映射文件。

1) 缓冲区分配和包装

在能够读和写之前，必须有一个缓冲区。要创建缓冲区，您必须“分配”它。我们使用静态方法allocate()来分配缓冲区：

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

allocate()方法分配一个具有指定大小的底层数组，并将它包装到一个缓冲区对象中，在本例中是一个ByteBuffer。

您还可以将一个现有的数组转换为缓冲区，如下所示：

```
byte array[] = new byte[1024];
ByteBuffer buffer = ByteBuffer.wrap( array );
```

本例使用了wrap()方法将一个数组包装为缓冲区。必须非常小心地进行这类操作。一旦完成包装，底层数据就可以通过缓冲区或者直接访问。

2) 缓冲区分片

slice()方法根据现有的缓冲区创建一个子缓冲区。也就是说，它创建一个新的缓冲区，新缓冲区与原来的缓冲区的一部分共享数据。

使用例子可以最好地说明这点。让我们首先创建一个长度为10的ByteBuffer：

```
ByteBuffer buffer = ByteBuffer.allocate( 10 );
```

然后使用数据来填充这个缓冲区，在第n个槽中放入数字n：

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

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

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

```
buffer.position( 3 );
buffer.limit( 7 );
ByteBuffer slice = buffer.slice();
```

该“片段”是缓冲区的子缓冲区。不过，“片段”和“缓冲区”共享同一个底层数据数组，我们在下一节将会看到这一点。

3) 缓冲区片份和数据共享

我们已经创建了原缓冲区的子缓冲区，并且已经知道缓冲区和子缓冲区共享同一个底层数据数组。让我们看看这意味着什么。

我们遍历子缓冲区，将每一个元素乘以11来改变它。例如，5会变成55。

```
for (int i=0; i<slice.capacity(); ++i) {
     byte b = slice.get( i );
     b *= 11;
     slice.put( i, b );
}
```

最后，再看一下原缓冲区中的内容：

```
buffer.position( 0 );
buffer.limit( buffer.capacity() );

while (buffer.remaining()>0) {
     System.out.println( buffer.get() );
}
```

结果表明只有在子缓冲区窗口中的元素被改变了：

0
1
2
33
44
55
66
7
8
9

缓冲区片对于促进抽象非常有帮助。可以编写自己的函数处理整个缓冲区，而且如果想要将这个过程应用于子缓冲区上，您只需取主缓冲区的一个片，并将它传递给您的函数。这比编写自己的函数来取额外的参数以指定要对缓冲区的哪一部分进行操作更容易。

4) 只读缓冲区

只读缓冲区的含义已经很直白了：您可以读取它们，但是不能向它们写入。可以通过调用缓冲区的asReadOnlyBuffer()方法，来将任何常规缓冲区转换为只读缓冲区，这个方法返回一个与原缓冲区完全相同的缓冲区(并与其共享数据)，只不过它是只读的。

只读缓冲区对于保护数据很有用。在将缓冲区传递给某个对象的方法时，您无法知道这个方法是否会修改缓冲区中的数据。创建一个只读的缓冲区可以保证该缓冲区不会被修改。不能将只读的缓冲区转换为可写的缓冲区。

5) 直接和间接缓冲区

另一种有用的ByteBuffer是直接缓冲区。“直接缓冲区”是为加快I/O速度，而以一种特殊的方式分配其内存的缓冲区。实际上，直接缓冲区的准确定义是与实现相关的。

Sun的文档是这样描述直接缓冲区的：给定一个直接字节缓冲区，Java虚拟机将尽最大努力直接对它执行本机I/O操作。也就是说，它会在每一次调用底层操作系统的本机I/O操作之前(或之后)，尝试避免将缓冲区的内容拷贝到一个中间缓冲区中(或者从一个中间缓冲区中拷贝数据)。

附件中，您可以在例子程序FastCopyFile.java中看到直接缓冲区的实际应用，这个程序是CopyFile.java的另一个版本，它使用了直接缓冲区以提高速度。还可以用内存映射文件创建直接缓冲区。

6) 内存映射文件I/O

内存映射文件I/O是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的I/O快得多。

内存映射文件I/O是通过使文件中的数据神奇般地出现为内存数组的内容来完成的。这其初听起来似乎不过就是将整个文件读到内存中，但是事实上并不是这样。一般来说，只有文件中实际读取或者写入的部分才会送入

（或者映射）到内存中。

内存映射并不真的神奇或者多么不寻常。现代操作系统一般根据需要将文件的部分映射为内存的部分，从而实现文件系统。Java内存映射机制不过是在底层操作系统中可以采用这种机制时，提供了对该机制的访问。

尽管创建内存映射文件相当简单，但是向它写入可能是危险的。仅只是改变数组的单个元素这样的简单操作，就可能会直接修改磁盘上的文件。修改数据与将 数据保存到磁盘是没有分开的。

7) 将文件映射到内存

了解内存映射的最好方法是使用例子。在下面的例子中，我们要将一个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 );
```

map()方法返回一个MappedByteBuffer，它是ByteBuffer的子类。因此，您可以像使用其他任何ByteBuffer一样使用新映射的缓冲区，操作系统会在需要时负责执行行映射。

8. 分散和聚集

1) 概述：

分散/聚集I/O是使用多个而不是单个缓冲区来保存数据的读写方法。

一个分散的读取就像一个常规通道读取，只不过它是将数据读到一个缓冲区数组中而不是读到单个缓冲区中。同样地，一个聚集写入是向缓冲区数组而不是向 单个缓冲区写入数据。

分散/聚集I/O对于将数据流划分为单独的部分很有用，这有助于实现复杂的数据格式。

2) 分散/聚集 I/O：

通道可以有选择地实现两个新的接口：ScatteringByteChannel和GatheringByteChannel。一个ScatteringByteChannel是一个具有两个附加读方法的通道：

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

这些long read()方法很像标准的read方法，只不过它们不是取单个缓冲区而是取一个缓冲区数组。

在“分散读取”中，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。

3) 分散/聚集的应用：

分散/聚集I/O对于将数据划分为几个部分很有用。例如，您可能在编写一个使用消息对象的网络应用程序，每一个消息被划分为固定长度的头部和固定长度的正文。您可以创建一个刚好可以容纳头部的缓冲区和另一个刚好可以容难正文的缓冲区。当您将它们放入一个数组中并使用分散读取来向它们读入消息时，头部和正文将整

齐地划分到这 两个缓冲区中。

我们从缓冲区所得到的方便性对于缓冲区数组同样有效。因为每一个缓冲区都跟踪自己还可以接受多少数据，所以分散读取会自动找到有空间接受数据的第一个缓冲区。在这个缓冲区填满后，它就会移动到下一个缓冲区。

4) 聚集写入：

聚集写入类似于分散读取，只不过是用来写入。它也有接受缓冲区数组的方法：

```
long write( ByteBuffer[] srcs );
long write( ByteBuffer[] srcs, int offset, int length );
```

聚集写对于把一组单独的缓冲区中组成单个数据流很有用。为了与上面的消息例子保持一致，您可以使用聚集写入来自动将网络消息的各个部分组装为单个数据流，以便跨越网络传输消息。

从附件的例子程序 UseScatterGather.java 中可以看到分散读取和聚集写入的实际应用。

后续： 在下一篇文章中，会介绍NIO相关的文件锁定、字符集等知识。有兴趣的可以共同学习、讨论。

附件下载:

- nio.zip (3.2 KB)
- dl.javaeye.com/topics/download/eb628781-3c98-316a-a50b-47fa9c6900cb

1.5 NIO学习系列：文件锁定和字符集

发表时间: 2010-06-06

在前面五名已经学习了四篇关于NIO系列的文章：

核心概念及基本读写 、缓冲区内部实现机制 、连网和异步IO 、缓冲区更多特性及分散/聚集IO ，

这里我们继续探讨和学习有关文件锁定和字符集相关的内容。

9. 文件锁定

1) 概述：

文件锁定初看起来可能让人迷惑。它似乎指的是防止程序或者用户访问特定文件。事实上，文件锁就像常规的Java对象锁，它们是“劝告式”的（advisory）锁。它们不阻止任何形式的数据访问，相反，它们通过锁的共享和获取赖允许系统的不同部分相互协调。

您可以锁定整个文件或者文件的一部分。如果您获取一个排它锁，那么其他人就不能获得同一个文件或者文件的一部分上的锁。如果您获得一个共享锁，那么其他人可以获得同一个文件或者文件一部分上的共享锁，但是不能获得排它锁。文件锁定并不总是出于保护数据的目的。例如，您可能临时锁定一个文件以保证特定 的写操作成为原子的，而不会有其他程序的干扰。

大多数操作系统提供了文件系统锁，但是它们并不都是采用同样的方式。有些实现提供了共享锁，而另一些仅提供了排它锁。事实上，有些实现使得文件的锁 定部分不可访问，尽管大多数实现不是这样的。

在这里，我们将学习如何在 NIO 中执行简单的文件锁过程，还将探讨一些保证被锁定的文件尽可能可移植的方法。

2) 锁定文件：

要获取文件的一部分上的锁，您要调用一个打开的FileChannel上的lock()方法。注意，如果要获取一个排它锁，您必须以写方式打开文件。

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

在拥有锁之后，您可以执行需要的任何敏感操作，然后再释放锁：

```
lock.release();
```

在释放锁后，尝试获得锁的其他任何程序都有机会获得它。

本附件的例子程序UseFileLocks.java必须与它自己并行运行。这个程序获取一个文件上的锁，持有三秒钟，

然后释放它。如果同时运行这个程序的多个实例，您会看到每个实例依次获得锁。

3) 文件锁定和可移植性：

文件锁定可能是一个复杂的操作，特别是考虑到 不同的操作系统是以不同的方式实现锁这一事实。下面的指导原则将帮助您尽可能保持代码的可移植性：

i 只使用排它锁。

ii将所有的锁视为劝告式的（advisory）。

10. 字符集

1) 概述：

根据Sun的文档，一个Charset是“十六位Unicode字符序列与字节序列之间的一个命名的映射”。实际上，一个Charset允许您以尽可能最具可移植性的方式读写字符序列。

Java语言被定义为基于Unicode。然而在实际上，许多人编写代码时都假设一个字符在磁盘上或者在网络流中用一个字节表示。这种假设在许多情况下成立，但是并不是在所有情况下都成立，而且随着计算机变得对Unicode越来越友好，这个假设就日益变得不能成立了。

在这里，我们将看一下如何使用Charsets以适合现代文本格式的方式处理文本数据。这里将使用的示例程序相当简单，不过，它触及了使用Charset的所有关键方面：为给定的字符编码创建Charset，以及使用该Charset解码和编码文本数据。

2) 编码/解码：

要读和写文本，我们要分别使用CharsetDecoder和CharsetEncoder。将它们称为“编码器”和“解码器”是有道理的。一个字符不再表示一个特定的位模式，而是表示字符系统中的一个实体。因此，由某个实际的位模式表示的字符必须以某种特定的编码来表示。

CharsetDecoder用于将逐位表示的一串字符转换为具体的char值。同样，一个CharsetEncoder用于将字符转换回位。

3) 处理文本的正确方式：

现在我们将分析这个例子程序UseCharsets.java。这个程序非常简单：它从一个文件中读取一些文本，并将该文本写入另一个文件。但是它把该数据当作文本数据，并使用CharBuffer来将该数句读入一个CharsetDecoder中。同样，它使用CharsetEncoder来写回该数据。

我们将假设字符以ISO-8859-1(Latin1)字符集（这是ASCII的标准扩展）的形式储存在磁盘上。尽管我们必须为使用Unicode做好准备，但是也必须认识到不同的文件是以不同的格式储存的，而ASCII无疑是非常普遍的一种格式。事实上，每种Java实现都要求对以下字符编码提供完全的支持：

US-ASCII

ISO-8859-1

UTF-8

UTF-16BE

UTF-16LE

UTF-16

4) 示例程序：

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

```
Charset latin1 = Charset.forName( "ISO-8859-1" );
```

然后，创建一个解码器（用于读取）和一个编码器（用于写入）：

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

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

```
CharBuffer cb = decoder.decode( inputData );
```

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

要写回数据，我们必须使用CharsetEncoder将它转换回字节：

```
ByteBuffer outputData = encoder.encode( cb );
```

在转换完成之后，我们就可以将数据写到文件中了。

11. 结束语和参考资料

正如您所看到的，NIO库有大量的特性。在一些新特性（例如文件锁定和字符集）提供新功能的同时，许多特性在优化方面也非常优秀。

在基础层次上，通道和缓冲区可以做的事情几乎都可以用原来的面向流的类来完成。但是通道和缓冲区允许以快得多的方式完成这些相同的旧操作，事实上接近系统所允许的最大速度。

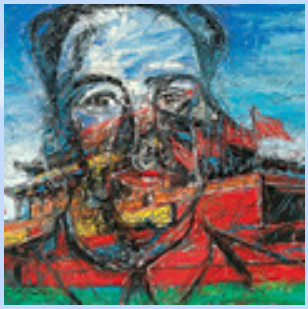
不过NIO最强大的长度之一在于，它提供了一种在Java语言中执行进行输入/输出的新的（也是迫切需要的）结构化方式。随诸如缓冲区、通道和异步I/O这些概念性（且可实现的）实体而来的，是我们重新思考Java程序中的I/O过程的机会。这样，NIO甚至为我们最熟悉的I/O过程也带来了新的活力，同时赋予我们通过和以前不同并且更好的方式执行它们的机会。

本系列教程的整理参考了IBM developerworks 中国社区关于NIO的一些学习资料。

附件下载:

- nio.zip (1.6 KB)
- dl.javaeye.com/topics/download/d6eaa2bf-d9b0-3021-8155-89b6b8dc2f13

NIO学习总结



NIO学习总结

作者: zhangshixi
http://zhangshixi.javaeye.com