## 开始之前

### IO的概念

IO 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。它对于任何计算机系统都非常关键，因而所有 I/O 的主体实际上是内置在操作系统中的。例如：在实际过程中，我们一般是先将文件读入内存，再从内存写出到别的地方。

### 关于流

在 Java 编程中，直到NIO出现之前，一直使用 流 的方式完成 IO。所有 IO 都被视为单个的字节的移动，通过一个称为 Stream 的对象一次移动一个字节。流 IO 用于与外部世界接触。它也在内部使用，用于将对象转换为字节，然后再转换回对象。

### 为什么引入NIO

NIO库是在 JDK 1.4 中引入的。它是以块的形式处理这些数据，NIO 不用使用本机代码就可以利用低级优化，这是原来的 I/O 包所无法做到的。NIO 的创建目的是为了让 Java 程序员可以实现高速 I/O 而无需编写自定义的本机代码。NIO 将最耗时的 I/O 操作(即填充和提取缓冲区)转移回操作系统，因而可以极大地提高速度。

当程序从硬盘往内存读取数据的时候，操作系统使用了 2 个“小伎俩”来提高性能，那就是预读，如果我读取了第一扇区的第三磁道的内容，那么你很有可能也会使用第二磁道和第四磁道的内容，所以操作系统会把附近磁道的内容提前读取出来，放在内存中，即缓存。通过上面可以看到，操作系统是按块 Block从硬盘拿数据，就如同一个大脸盆，一下子就放入了一盆水。但是，当 Java 使用的时候，旧的 IO 是基于流 Stream的，也就是虽然操作系统给我了一脸盆水，但是我得用吸管慢慢喝。于是，NIO 横空出世。

### 阻塞与非阻塞IO

就速度来说 CPU > 内存 > 硬盘，在以前的 Java IO 中，都是阻塞式 IO，NIO 引入了非阻塞式 IO。

* **阻塞式 IO：**我从硬盘读取数据，然后程序一直等，数据读完后，继续操作。这种方式是最简单的，叫阻塞IO。
* **非阻塞式 IO：**我从硬盘读取数据，然后程序继续向下执行，等数据读取完后，通知当前程序（对硬件来说叫中断，对程序来说叫回调），然后此程序可以立即处理数据。

Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

### 同步和异步IO

有两种类型的文件IO同步：同步文件IO和异步文件IO。异步文件IO也就是重叠IO。在同步文件IO中，线程启动一个IO操作然后就立即进入等待状态，直到IO操作完成后才醒来继续执行。而异步文件IO方式中，线程发送一个IO请求到内核，然后继续处理其他的事情，内核完成IO请求后，将会通知线程IO操作完成了。

同步IO：当一个IO操作进行时，系统停下来等待这个IO的操作完成后才会继续进行下面的计算。

异步IO：当一个IO操作进行时，通过多线程等方法，当CPU去处理那些不需要依赖IO执行结果的计算，充分的利用CPU的运算能力。当IO操作执行完毕时，继续执行后续的操作。

**java nio的io模型是同步非阻塞**

### 流与块的优缺点

原来的 I/O 库(在 java.io.\*中) 与 NIO 最重要的区别是数据打包和传输的方式。正如前面提到的，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。然而，一个 面向块 的 I/O 系统以块的形式处理数据。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。

### 面向流与面向缓冲

Java IO 面向流意味着每次从流中读一个字节（使用字符流的时候会读取多个字节，它是按字符读取），直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。

Java NIO 的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

使用缓冲区有这么两个好处：

1、减少实际的物理读写次数

2、缓冲区在创建时就被分配内存，这块内存区域一直被重用，可以减少动态分配和回收内存的次数

### 关于IO中缓冲区和NIO中的缓冲区

有些人可能会说“~~IO 是面向流的，NIO 是面向缓冲区的~~”，其实这句话前半句是不对的，应该这样说“IO中除了BufferedInputStream、BufferedOutputStream、BufferedReader和BufferedWriter外都是面向流的，NIO是面向缓冲区的”。

有些学者可能会疑惑，在就的IO中也有用到缓冲技术，比如：BufferedInputStream、BufferedOutputStream、BufferedReader和BufferedWriter。那么它们与NIO中的缓冲区有什么不同呢？我们知道在实际程序处理过程中，有时为了提高效率，我们会先将外部数据缓存到内存中，然后在从内存去取数据做其他操作，比如：写到其他外部文件中。那么“将外部数据缓存到内存中”这个操作我们是交由操作系统来实现，在旧的IO中是使用流的方式进行缓存，而NIO是通过块的形式缓存，而在我们进行取数据的时候，也是通过流或块的形式操作数据。所以这里的面向流和面向块可以做类比，而缓冲则是另外一个概念。

### NIO内部机制

在 JDK 1. 4 中 新 加入 了 NIO( New Input/ Output) 类, 引入了一种基于通道和缓冲区的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆的 DirectByteBuffer 对象作为这块内存的引用进行操作，避免了在 Java 堆和 Native 堆中来回复制数据。

### IO和NIO的集成

在 JDK 1.4 中原来的 I/O 包和 NIO 已经很好地集成了。 java.io.\* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如， java.io.\* 包中的一些类包含以块的形式读写数据的方法，这使得即使在更面向流的系统中，处理速度也会更快。也可以用 NIO 库实现标准 I/O 功能。例如，可以容易地使用块 I/O 一次一个字节地移动数据。但是正如您会看到的，NIO 还提供了原 I/O 包中所没有的许多好处。

## 通道和缓冲区

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。一个 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。

### 缓冲区

Buffer 是一个对象， 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。

### 缓冲区的好处

缓冲区(Buffer)就是在内存中预留指定大小的存储空间用来对输入/输出(I/O)的数据作临时存储，这部分预留的内存空间就叫做缓冲区。

使用缓冲区有这么两个好处：

1、减少实际的物理读写次数

2、缓冲区在创建时就被分配内存，这块内存区域一直被重用，可以减少动态分配和回收内存的次数

举个简单的例子，比如A地有1w块砖要搬到B地。由于没有工具（缓冲区），我们一次只能搬一本，那么就要搬1w次（实际读写次数）。如果A，B两地距离很远的话（IO性能消耗），那么性能消耗将会很大，但是要是此时我们有辆大卡车（缓冲区），一次可运5000本，那么2次就够了，相比之前，性能肯定是大大提高了。

而且一般在实际过程中，我们一般是先将文件读入内存，再从内存写出到别的地方，这样在输入输出过程中我们都可以用缓存来提升IO性能。

所以，buffer在IO中很重要。在旧I/O类库中中的BufferedInputStream、BufferedOutputStream、BufferedReader和BufferedWriter在其实现中都运用了缓冲区。java.nio包公开了Buffer API，使得Java程序可以直接控制和运用缓冲区。

### 缓冲区的类型

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，如：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。

*// 测试：使用类型化的Buffer类*@org.junit.Test  
**public void** testUseFloatBuffer() {  
 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);  
 }  
  
 buffer.flip();  
  
 **while** (buffer.hasRemaining()) {  
 **float** f = buffer.get();  
 System.***out***.println(f);  
 }  
}

### Buffer 常见方法

* flip(): 重设此缓冲区，将limit设置为当前position，然后将当前position设置为0 。读写缓冲区都会改变position和limit的位置。
* clear(): 置为初始化状态 (position 变成 0 ， limit 变成 capacity) 。
* asReadOnlyBuffer():返回一个只读的缓冲区，和原始缓冲区共用同一块内存，只是只读。修改会抛异常。不能将只读的缓冲区转换为可写的缓冲区。
* get():返回当前位置的数据，同时将 position+=1。
* put():加入元素，position+=增加的元素个数。
* remaining():返回剩余的数据的数量, (即limit - position)。
* hasRemaining():是否还有剩余数据，循环的时候用。
* rewind(): 将 position 重置为 0 ，一般用于重复读。
* limit(int newLimit),position(int newPosition):修改这两个值。
* compact(): 将未读取的数据拷贝到 buffer 的头部位。
* mark()、reset(): mark 可以标记一个位置， reset 可以重置到该位置。

### 通道

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

### 通道与流的区别

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。

## 使用NIO读和写文件示例

### 从文件中读数据

@org.junit.Test  
**public void** testReadFile() **throws** IOException {  
  
 String inputPath = **"D:"** + File.***separator*** + **"temp1.txt"**;  
 FileChannel inChannel = **new** FileInputStream(inputPath).getChannel();  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
  
 String str = **""**;  
 **while** (**true**) {  
 buffer.clear();  
 *// 将数据从通道读到缓冲区中* **int** r = inChannel.read(buffer);  
 **if** (r == -1) **break**;  
 buffer.flip();  
 str += getStringFromBufferByGBK(buffer);  
 }  
 System.***out***.println(str);  
}

**public** String getStringFromBufferByGBK(ByteBuffer buffer) {  
 Charset charset = **null**;  
 CharsetDecoder decoder = **null**;  
 CharBuffer charBuffer = **null**;  
 **try** {  
 charset = Charset.*forName*(**"GBK"**);  
 decoder = charset.newDecoder();  
 charBuffer = decoder.decode(buffer.asReadOnlyBuffer());  
 **return** charBuffer.toString();  
 } **catch** (Exception ex) {  
 ex.printStackTrace();  
 **return ""**;  
 }  
}

### 写数据到文件中

@org.junit.Test  
**public void** testWriteFile() **throws** IOException {  
 String outputPath = **"D:"** + File.***separator*** + **"temp1.txt"**;  
 FileChannel outChannel = **new** FileOutputStream(outputPath).getChannel();  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
  
 **byte**[] message = **"仅仅只是一段测试"**.getBytes();  
 buffer.put(message);  
 buffer.flip();  
 outChannel.write(buffer);  
}

### 文件复制

@org.junit.Test  
**public void** testCopyFile() **throws** IOException {  
 String inputPath = **"D:"** + File.***separator*** + **"temp1.txt"**;  
 String outputPath = **"D:"** + File.***separator*** + **"temp2.txt"**;  
  
 FileChannel inChannel = **new** FileInputStream(inputPath).getChannel();  
 FileChannel outChannel = **new** FileOutputStream(outputPath).getChannel();  
  
 ByteBuffer buffer = ByteBuffer.*allocate*(1024);  
 **while** (**true**) {  
 buffer.clear();  
 **int** r = inChannel.read(buffer);  
 **if** (r == -1) **break**;  
 buffer.flip();*// flip()方法的作用是让缓冲区可以将新读入的数据写入另一个通道中* outChannel.write(buffer);  
 }  
}

## 缓冲区内部细节

状态变量是缓冲区"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。

例如，在从通道读取数据时，数据被放入到缓冲区。在有些情况下，可以将这个缓冲区直接写入另一个通道，但是在一般情况下，您还需要查看数据。这是使用 访问方法 get() 来完成的。同样，如果要将原始数据放入缓冲区中，就要使用访问方法 put()。

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。您可能习惯于通过手工编码进行簿记 ― 即使用字节数组和索引变量，现在它已在 NIO 中内部地处理了。

### 状态变量

可以用三个值指定缓冲区在任意时刻的状态：position、limit和capacity。这三个变量一起可以跟踪缓冲区的状态和它所包含的数据。

#### Position

缓冲区实际上就是美化了的数组。position的作用是记录每次从缓冲区读入数据（往数组添加数据）或写出数据（从数组取数据）时的开始位置。

例如，在从通道读取时，您将所读取的数据放到底层的数组中。 position 变量跟踪已经放入了多少数据。更准确地说，它指定了下一个字节将放到数组的哪一个元素中。因此，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，指向数组中第四个元素。

同样，在写入通道时，您是从缓冲区中获取数据。 position 值跟踪从缓冲区中获取了多少数据。更准确地说，它指定下一次拿数据的时候是从数组的哪一个元素开始拿。因此如果从缓冲区写了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。

#### **Limit**

limit 变量表明还有多少空间可以放入数据，或者还有多少数据需要取出。

在从通道读入数据到缓冲区时，即往数组插入数据时，limit表示数组还有多少空间可以放入数据；

在从通道写入数据到缓冲区时，即往数组读取数据时，limit表示数组还有多少数据没有被读取。

#### **Capacity**

缓冲区的 capacity 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。

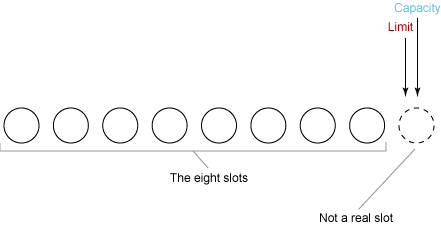
**注意：position 总是小于或者等于 limit，而limit 决不能大于 capacity。**

### 分析状态变量的内部机制

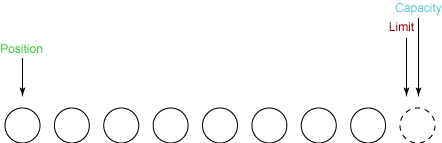
我们首先观察一个新创建的缓冲区。出于本例子的需要，我们假设这个缓冲区的 总容量 为8个字节。 Buffer 的状态如下所示：

IMG_256

我们知道，limit 决不能大于 capacity的，所以，在此例中这两个值的初始值都被设置为 8。我们通过将它们指向数组的尾部之后来说明这点。(如果有第8个槽，则是第8个槽所在的位置)



position 设置为0。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：

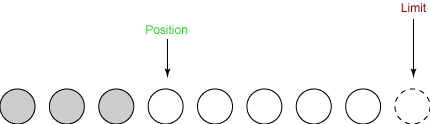


由于 capacity 不会改变，所以我们在下面的讨论中可以忽略它。

#### **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。

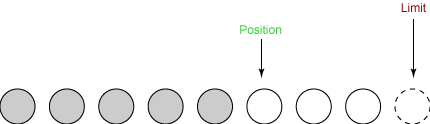
首先从输入通道中读一些数据到缓冲区中。第一次读取三个字节，它们被放到数组中从 position 开始的位置，这时 position 被设置为 0。读完之后，position 就增加到 3，如下所示：



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：



注意，两个读数据到缓冲区时，limit都没有改变。

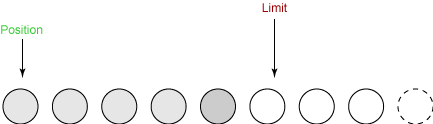
#### **flip操作**

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。这个方法做两件非常重要的事：

1、它将 limit 设置为当前 position。

2、它将 position 设置为 0。

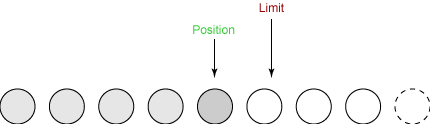
上面两次读取的图显示的都是在 flip 之前缓冲区的情况，下面是在 flip 之后的缓冲区：



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。

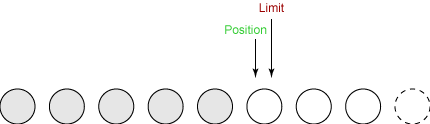
#### **第一次写入**

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：



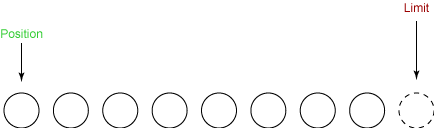
#### **clear操作**

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。 Clear 做两种非常重要的事情：

1、它将 limit 设置为与 capacity 相同。

2、它设置 position 为 0。

下图显示了在调用 clear() 后缓冲区的状态，它又回到了最原始的状态：



缓冲区现在可以接收新的数据了。

### 缓冲区的get/put方法

我们常常需要将数据保存到磁盘，这时，我们必须将这些数据直接放入缓冲区，然后用通道将缓冲区写到磁盘；或者，需要从磁盘读取数据，在这种情况下，我们需要将数据从通道读入到缓冲区中，然后从缓冲区中读取数据。

简单的说，我们利用通道写数据到磁盘，或利用通道从磁盘读取数据，我们都需要先将数据放到缓冲区去，然后通道在从缓冲区取数据，至于数据流向，取决于通道的读或写操作。

接下来，我们将分析如何使用 ByteBuffer 类的 get() 和 put() 方法直接访问缓冲区中的数据。

#### get()方法

ByteBuffer 类中有四个 get() 方法：

1. byte get();
2. ByteBuffer get( byte dst[] );
3. ByteBuffer get( byte dst[], int offset, int length );
4. 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。

此外，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() 操作是从当前 position 读取的，而 position 在 get 之后会增加。而绝对方法不会影响 limit 和 position 值。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。

#### **put()方法**

ByteBuffer 类中有五个 put() 方法：

1. ByteBuffer put( byte b );
2. ByteBuffer put( byte src[] );
3. ByteBuffer put( byte src[], int offset, int length );
4. ByteBuffer put( ByteBuffer src );
5. ByteBuffer put( int index, byte b );

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；

第五个方法将单个字节写入缓冲区中特定的位置 。

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。

#### **类型化的get/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。例如：

@org.junit.Test  
**public void** testTypesInByteBuffer() {  
 ByteBuffer buffer = ByteBuffer.*allocate*( 64 );  
  
 buffer.putInt( 30 );  
 buffer.putLong( 7000000000000L );  
 buffer.putDouble( Math.***PI*** );  
  
 buffer.flip();  
  
 System.***out***.println( buffer.getInt() );  
 System.***out***.println( buffer.getLong() );  
 System.***out***.println( buffer.getDouble() );  
}

#### **缓冲区的使用：一个内部循环**

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 = inChannel.read(buffer);  
 **if** (r == -1) **break**;  
 buffer.flip();*// flip()方法的作用是让缓冲区可以将新读入的数据写入另一个通道中* outChannel.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。

## 关于缓冲区的更多内容

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。您将学到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。我们将在本节的最后介绍如何在 NIO 中创建内存映射文件。

### 使用分配和包装创建缓冲区

**1、用分配的方式创建缓冲区：**

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，该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：

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

**2、用包装的方式创建缓冲区：**

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。

### 缓冲区分片和数据共享

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。

接下来，我们使用slice方法对缓存区进行分片，通过如下测试代码，我们发现片段（子缓冲区） 和 原缓冲区是共享同一个底层数据数组的。

@org.junit.Test  
**public void** testSliceBuffer() {  
 ByteBuffer buffer = ByteBuffer.*allocate*( 10 );  
 **for** (**int** i=0; i<buffer.capacity(); ++i) {  
 buffer.put( (**byte**)i );  
 }  
  
 *// 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,子缓冲区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：* buffer.position( 3 );  
 buffer.limit( 7 );  
 ByteBuffer slice = buffer.slice();  
  
 *// 更新子缓冲区数据* **for** (**int** i=0; i<slice.capacity(); ++i) {  
 **byte** b = slice.get( i );  
 b \*= 11;  
 slice.put( i, b );  
 }  
  
 *// 输出原缓冲区中的内容* buffer.position( 0 );  
 buffer.limit( buffer.capacity() );  
 **while** (buffer.remaining()>0) {  
 System.***out***.println( buffer.get() );  
 }  
}

结果表明原缓冲区和子缓冲区共享的数据被改变了：

0

1

2

33

44

55

66

7

8

9

### 只读缓冲区

只读缓冲区非常简单，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，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。

### 直接和间接缓冲区

直接缓冲区是为加快I/O速度，使用一种特殊方式为其分配内存的缓冲区，JDK文档中的描述为：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。

要分配直接缓冲区，需要调用allocateDirect()方法，而不是allocate()方法，使用方式与普通缓冲区并无区别。还可以用内存映射文件创建直接缓冲区。

以下我们使用直接缓冲区的方法进行文件复制：

*// 测试：使用直接缓冲区进行文件复制（比testCopyFile效率高）*@org.junit.Test  
**public void** testFastCopyFile() **throws** IOException {  
  
 String inputPath = **"D:"** + File.***separator*** + **"temp1.txt"**;  
 String outputPath = **"D:"** + File.***separator*** + **"temp2.txt"**;  
 FileChannel inChannel = **new** FileInputStream(inputPath).getChannel();  
 FileChannel outChannel = **new** FileOutputStream(outputPath).getChannel();  
  
 ByteBuffer buffer = ByteBuffer.*allocateDirect*(1024);  
 **while** (**true**) {  
 buffer.clear();  
 **int** r = inChannel.read(buffer);  
 **if** (r == -1) **break**;  
 buffer.flip();  
 outChannel.write(buffer);  
 }  
}

此外，我们还可以用内存映射文件创建直接缓冲区。

### 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。

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，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。

*// 测试：内存映射文件 I/O*@org.junit.Test  
**public void** testUseMappedFile() **throws** IOException {  
 **final int** start = 0;  
 **final int** size = 1024;  
  
 String filePath = **"D:"** + File.***separator*** + **"usemappedfile.txt"**;  
 RandomAccessFile raf = **new** RandomAccessFile( filePath, **"rw"** );  
 FileChannel fc = raf.getChannel();  
  
 MappedByteBuffer mbb = fc.map( FileChannel.MapMode.***READ\_WRITE***, start, size );  
  
 mbb.put( 0, (**byte**)97 );*// 向文件写入“a”* mbb.put( 1023, (**byte**)122 );*// 向文件写入“z”* raf.close();  
}

## 分散和聚集

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。

ScatteringByteChannel、GatheringByteChannel 是两个interface，而 FileChannel 和 SocketChannel 等实现了这两个接口

### 分散读取：ScatteringByteChannel 接口

 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。

### 聚集写入：GatheringByteChannel接口

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。

### 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。

从如下程序中可以看到分散读取和聚集写入的实际应用：

*// 测试：文件的分散读取和聚集写入，FileChannel 实现了 ScatteringByteChannel 和 GatheringByteChannel 接口*@org.junit.Test  
**public void** testUseScatterGather1() **throws** IOException {  
 String inputPath = **"D:"** + File.***separator*** + **"temp1.txt"**;  
 String outputPath = **"D:"** + File.***separator*** + **"temp2.txt"**;  
  
 RandomAccessFile randomAccessFile1 = **new** RandomAccessFile(inputPath, **"rw"**);  
 ScatteringByteChannel scatteringByteChannel = randomAccessFile1.getChannel();  
  
 ByteBuffer buf1 = ByteBuffer.*allocate*(3);  
 ByteBuffer buf2 = ByteBuffer.*allocate*(1024);  
  
 *// 分散读取* ByteBuffer[] bufs = {buf1, buf2};  
 scatteringByteChannel.read(bufs);  
  
 **for** (ByteBuffer byteBuffer : bufs) {  
 byteBuffer.flip();  
 }  
 System.***out***.println(**new** String(bufs[0].array(), 0, bufs[0].limit()));  
 System.***out***.println(**"----------------------------"**);  
 System.***out***.println(**new** String(bufs[1].array(), 0, bufs[1].limit()));  
  
 *// 聚集写入* RandomAccessFile randomAccessFile2 = **new** RandomAccessFile(outputPath, **"rw"**);  
 GatheringByteChannel gatheringByteChannel = randomAccessFile2.getChannel();  
 gatheringByteChannel.write(bufs);  
  
}

## 文件锁定

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来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。

### 示例

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。

*// 测试：文件锁定*@org.junit.Test  
**public void** testUseFileLocks() **throws** IOException {  
 **int** start = 10;  
 **int** end = 20;  
   
 String filePath = **"D:"** + File.***separator*** + **"usefilelocks.txt"**;  
 RandomAccessFile raf = **new** RandomAccessFile( filePath, **"rw"** );  
 FileChannel fc = raf.getChannel();  
  
 *// 获取文件锁* System.***out***.println( **"trying to get lock"** );  
 FileLock lock = fc.lock( start, end, **false** );  
 System.***out***.println( **"got lock!"** );  
  
 *// 将文件锁定3秒，模拟操作文件，比如向文件写入数据* System.***out***.println( **"pausing"** );  
 **try** { Thread.*sleep*( 3000 ); } **catch**( InterruptedException ie ) {}  
  
 *// 释放锁* System.***out***.println( **"going to release lock"** );  
 lock.release();  
 System.***out***.println( **"released lock"** );  
  
 raf.close();  
}

### 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：

* 只使用排它锁。
* 将所有的锁视为劝告式的（advisory）。

## 选择器（Selectors）

Java NIO 的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来 “选择” 通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

## 连网和异步 I/O（待完善。。。）

### 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。

### 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。

另一方面，异步 I/O 调用不会阻塞。相反，您将注册对特定 I/O 事件的兴趣 ― 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。

我们将通过研究一个名为 MultiPortEcho.java 的例子程序来查看异步 I/O 的实际应用。这个程序就像传统的 echo server，它接受网络连接并向它们回响它们可能发送的数据。不过它有一个附加的特性，就是它能同时监听多个端口，并处理来自所有这些端口的连接。并且它只在单个线程中完成所有这些工作。

### Selectors

本节的阐述对应于 MultiPortEcho 的源代码中的 go() 方法的实现，因此应该看一下源代码，以便对所发生的事情有个更全面的了解。

异步 I/O 中的核心对象名为 Selector。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() 的第一个参数总是这个 Selector。

### 打开一个 ServerSocketChannel

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。

### 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() 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。

### 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();

// ... deal with I/O event ...

}

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。

### 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) {

// Accept the new connection

// ...

}

可以肯定地说， readOps() 方法告诉我们该事件是新的连接。

### 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。

### 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。

### 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 [参考资料](https://www.ibm.com/developerworks/cn/education/java/j-nio/section12.html) 中的源代码 (MultiPortEcho.java)。

### 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。

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。

## 字符集

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。

在本节中，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。

### 编码/解码

Java.nio.charset 提供了编码解码一套解决方案。

字符编码解码 : 字节本身只是一些数字，放到正确的上下文中被正确的解析。向 ByteBuffer 中存放数据时需要考虑字符集的编码方式，读取展示 ByteBuffer 数据时涉及对字符集解码。

Charset **charset** = Charset.*forName*(**"GBK"**);*// 创建GBK字符集*ByteBuffer **bf** = **charset**.encode(**"hello world我是中国人"**);  
CharBuffer **cf** = **charset**.decode(**bf**); *// 每调用一次都会创建一个CharsetDecoder 对象*String **name** = **charset**.displayName();  
String **ret** = **cf**.toString();  
  
CharsetDecoder **decoder** = **charset**.newDecoder();  
CharsetEncoder **encoder** = **charset**.newEncoder();

### 示例程序

@org.junit.Test  
**public void** testUseCharsets() **throws** IOException {  
 String inputFile = **"D:"** + File.***separator*** + **"samplein.txt"**;  
 String outputFile = **"D:"** + File.***separator*** + **"sampleout.txt"**;  
  
 RandomAccessFile inf = **new** RandomAccessFile(inputFile, **"r"**);  
 RandomAccessFile outf = **new** RandomAccessFile(outputFile, **"rw"**);  
  
 FileChannel inc = inf.getChannel();  
 FileChannel outc = outf.getChannel();  
  
 **long** inputLength = **new** File(inputFile).length();  
 MappedByteBuffer inputData = inc.map(FileChannel.MapMode.***READ\_ONLY***, 0, inputLength);  
  
 *// 创建 ISO-8859-1 (Latin1) 字符集的一个实例* Charset latin1 = Charset.*forName*(**"ISO-8859-1"**);  
 CharsetDecoder decoder = latin1.newDecoder();*// 编码器* CharsetEncoder encoder = latin1.newEncoder();*// 解码器  
  
 // 使用编码器：将 字节 转为 字符* CharBuffer cb = decoder.decode(inputData);  
 *// 使用解码器：将 字符 转为 字节* ByteBuffer outputData = encoder.encode(cb);  
  
 *// 要写回数据，我们必须使用 CharsetEncoder 将它转换回字节，在转换完成之后，我们就可以将数据写到文件中了* outc.write(outputData);  
 inf.close();  
 outf.close();  
}