NIO 入门

在开始之前

关于本教程

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。通过定义包含数据的类，以及通过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。

在本教程中，我们将讨论 NIO 库的几乎所有方面，从高级的概念性内容到底层的编程细节。除了学习诸如缓冲区和通道这样的关键 I/O 元素外，您还有机会看到在更新后的库中标准 I/O 是如何工作的。您还会了解只能通过 NIO 来完成的工作，如异步 I/O 和直接缓冲区。

在本教程中，我们将使用展示 NIO 库的不同方面的代码示例。几乎每一个代码示例都是一个大的 Java 程序的一部分，您可以在 参考资料 中找到这个 Java 程序。在做这些练习时，我们推荐您在自己的系统上下载、编译和运行这些程序。在您学习了本教程以后，这些代码将为您的 NIO 编程努力提供一个起点。

本教程是为希望学习更多关于 JDK 1.4 NIO 库的知识的所有程序员而写的。为了最大程度地从这里的讨论中获益，您应该理解基本的 Java 编程概念，如类、继承和使用包。多少熟悉一些原来的 I/O 库(来自 .. 包)也会有所帮助。

虽然本教程要求掌握 Java 语言的工作词汇和概念，但是不需要有很多实际编程经验。除了彻底介绍与本教程有关的所有概念外，我还保持代码示例尽可能短小和简单。目的是让即使没有多少 Java 编程经验的读者也能容易地开始学习 NIO。

如何运行代码

源代码归档文件(在 参考资料 中提供)包含了本教程中使用的所有程序。每一个程序都由一个 Java 文件构成。每一个文件都根据名称来识别，并且可以容易地与它所展示的编程概念相关联。

教程中的一些程序需要命令行参数才能运行。要从命令行运行一个程序，只需使用最方便的命令行提示符。在 Windows 中，命令行提供符是 “Command” 或者 “command.com” 程序。在 UNIX 中，可以使用任何 shell。

需要安装 JDK 1.4 并将它包括在路径中，才能完成本教程中的练习。如果需要安装和配置 JDK 1.4 的帮助，请参见 参考资料 。

关于作者

Greg Travis 是一位自由 Java 程序员和技术撰稿人，现居住在纽约市。Greg 于 1992 年开始其编程生涯，在高端 PC 游戏领域工作了三年。1995 年，他加入了 EarthWeb，在那里他开始用 Java 编程语言开发新技术。1997 年后，Greg 成为各个 Web 技术领域的顾问，致力于实时图像和声音。

他的兴趣包括算法优化、编程语言设计、信号处理(侧重于音乐)以及实时三维图像。 Greg 撰写的其他文章可从 他的个人 Web 页 上找到。他还是 Manning Publications 出版的 JDK 1.4 Tutorial 一书的作者。

关本教程的技术问题或建议，可通过 mito@panix.com 或单击任何小节顶部的"反馈"与 Greg Travis 联系。

输入/输出:概念性描述

I/O 简介

I/O 是 C 或者输入/输出 是 C 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。单独的程序一般是让系统为它们完成大部分的工作。

在 Java 编程中，直到最近一直使用 *流* 的方式完成 I/O。所有 I/O 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 I/O 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。

NIO 与原来的 I/O 有同样的作用和目的，但是它使用不同的方式C *块 I/O*。正如您将在本教程中学到的，块 I/O 的效率可以比流 I/O 高许多。

为什么要使用 NIO?

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。

流与块的比较

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。

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。

集成的 I/O

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

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。

在本节中，您会了解到 NIO 中通道和缓冲区是如何工作的。

什么是缓冲区？

`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：

- `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。

现在您可以花一点时间运行 UseFloatBuffer.java，它包含了类型化的缓冲区的一个应用例子。

什么是通道？

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。

通道类型

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。

从理论到实践:NIO 中的读和写

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。

在本节中，我们将学习有关在 Java 程序中读取和写入数据的一些知识。我们将回顾 NIO 的主要组件(缓冲区、通道和一些相关的方法)，看看它们是如何交互以进行读写的。在接下来的几节中，我们将更详细地分析这其中的每个组件以及其交互。

从文件中读取

在我们第一个练习中，我们将从一个文件中读取一些数据。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 FileInputStream 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。

三个容易的步骤

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，如下所示：

```
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。

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 - 在这里，数据将从一个名为 message 的数组中取出，这个数组包含字符串 "Some bytes" 的 ASCII 字节(本教程后面将会解释 buffer.flip() 和 buffer.put() 调用)。

```
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。

读写结合

下面我们将看一下在结合读和写时会有什么情况。我们以一个名为 CopyFile.java 的简单程序作为这个练习的基础，它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。这个程序不断重复 — 读、写、读、写 — 直到源文件结束。

CopyFile 程序让您看到我们如何检查操作的状态，以及如何使用 clear() 和 flip() 方法重设缓冲区，并准备缓冲区以便将新读取的数据写到另一个通道中。

检查状态

下一步是检查拷贝何时完成。当没有更多的数据时，拷贝就算完成，并且可以在 read() 方法返回 -1 是判断这一点，如下所示：

```
int r = fcin.read( buffer );

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

重设缓冲区

最后，在从输入通道读入缓冲区之前，我们调用 clear() 方法。同样，在将缓冲区写入输出通道之前，我们调用 flip() 方法，如下所示：

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

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

buffer.flip();
```

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

clear() 方法重设缓冲区，使它可以接受读入的数据。 flip() 方法让缓冲区可以将新读入的数据写入另一个通道。

缓冲区内部细节

概述

本节将介绍 NIO 中两个重要的缓冲区组件：状态变量和访问方法 (accessor)。

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。

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()。

在本节中，您将学习关于 NIO 中的状态变量和访问方法的内容。我们将描述每一个组件，并让您有机会看到它的实际应用。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
- 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。我们将在下面的小节中详细分析每一个变量，还要介绍它们如何适应典型的读/写(输入/输出)进程。在这个例子中，我们假定要将数据从一个输入通道拷贝到一个输出通道。

Position

您可以回想一下，缓冲区实际上就是美化了的数组。在从通道读取时，您将所读取的数据放到底层的数组中。 position 变量跟踪已经写了多少数据。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，它指定下一个字节来自数组的哪一个元素。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。

Capacity

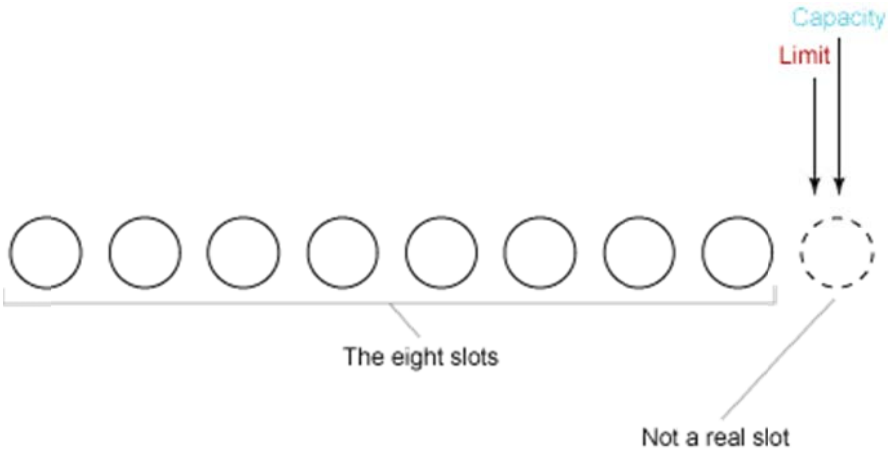
缓冲区的 capacity 表明可以储存在缓冲区中的最大数据容量。实际上，它指定了底层数组的大小 — 或者至少是指定了准许我们使用的底层数组的容量。

limit 决不能大于 capacity。

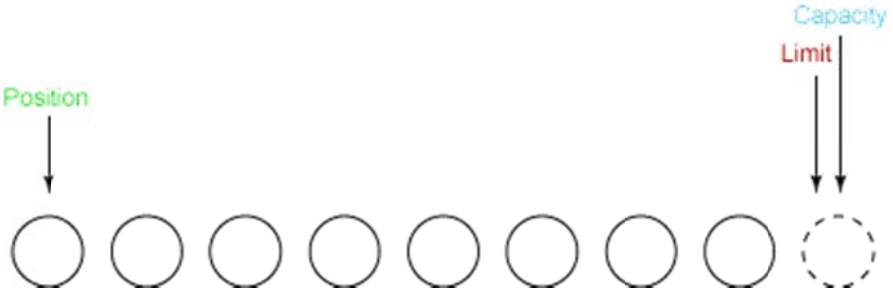
观察变量

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：

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(如果有第 8 个槽，则是第 8 个槽所在的位置)来说明这点。



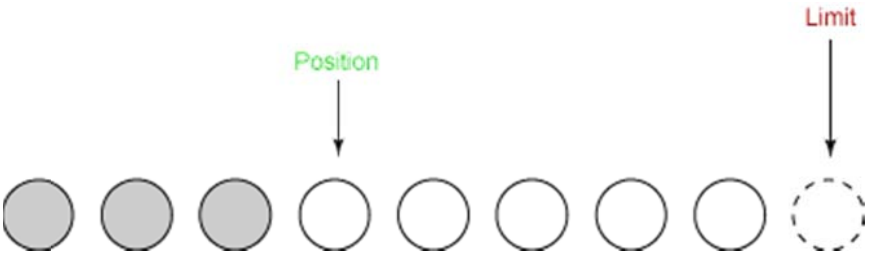
position 设置为 0。如果我们读一些数据到缓冲区中，那么下一个读取的数据就进入 slot 0 。如果我们从缓冲区写一些数据，从缓冲区读取的下一个字节就来自 slot 0 。 position 设置如下所示：



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

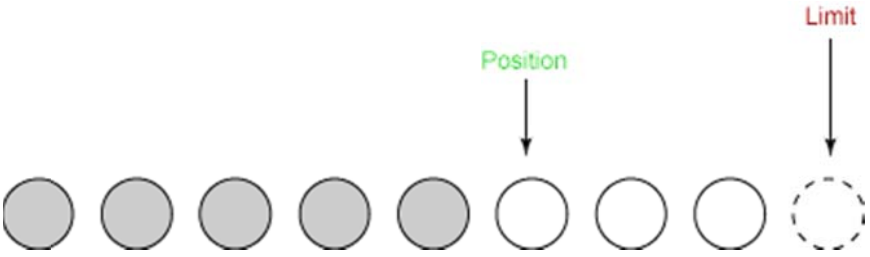
现在我们可以开始在新创建的缓冲区上进行读/写操作。首先从输入通道中读一些数据到缓冲区中。第一次读取得到三个字节。它们被放到数组中从 position 开始的位置，这时 position 被设置为 0。读完之后，position 就增加到 3，如下所示：



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：



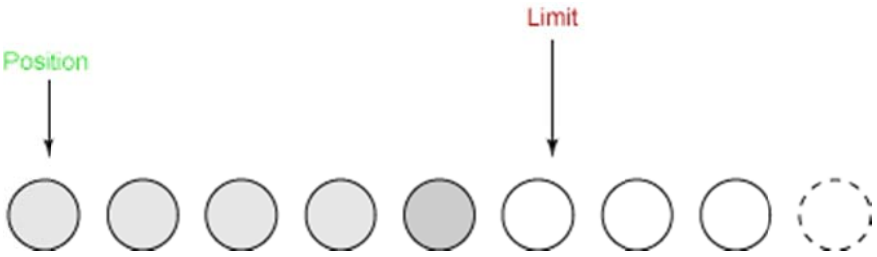
limit 没有改变。

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。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。

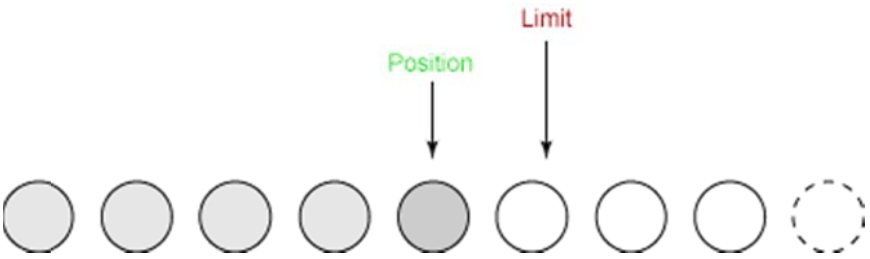
前一小节中的图显示了在 flip 之前缓冲区的情况。下面是在 flip 之后的缓冲区：



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。

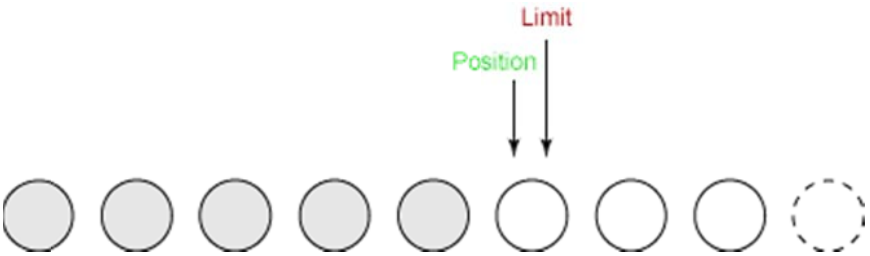
第一次写入

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：

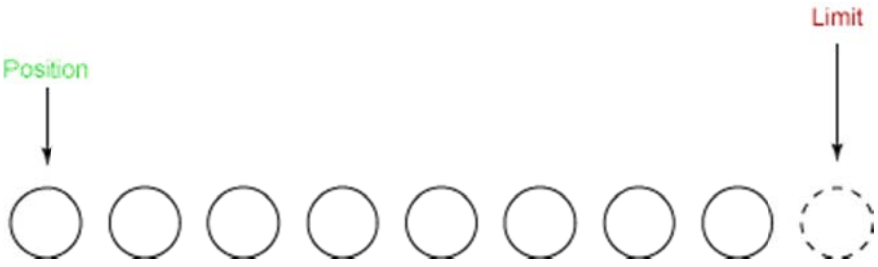


clear

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。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：



缓冲区现在可以接收新的数据了。

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。

在本节的最后，我们将详细分析如何使用 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。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。

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。

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。

您可以在例子程序 TypesInByteBuffer.java 中看到这些方法的实际应用。

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 = 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。

关于缓冲区的更多内容

概述

到目前为止，您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。我们的例子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。

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。

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。

缓冲区分片

`slice()` 方法根据现有的缓冲区创建一种 *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：

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

片 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。

缓冲区份片和数据共享

我们已经创建了原缓冲区的子缓冲区，并且我们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：

```
$ java SliceBuffer
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。

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

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)。

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。

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。

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。

分散和聚集

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。

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。

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。

聚集写入

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。

文件锁定

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。

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

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。

本小节的例子程序 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。

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）。

连网和异步 I/O

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 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

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

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。

在下一个小节中，我们将考察一个使用这些对象来读写数据的程序。

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

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 (Latin1) 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。

结束语和参考资料

结束语

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。

参考资料

- 下载 本教程中的例子的完整源代码。
- 关于安装和配置 JDK 1.4 的更多信息，请参见 SDK 文档 。
- Sun's guide to the new I/O APIs 提供了对 NIO 的全面介绍，包括一些本教程没有涵盖的细节内容。
- 在线 API 规范 描述了 NIO 的类和方法，该规范使用的是您了解并喜欢的 autodoc 格式。
- JSR 51 是 Java Community Process 文档，它最先规定了 NIO 的新特性。事实上，JDK 1.4 中实现的 NIO 是该文档描述的特性的一个子集。
- 想获得关于流 I/O(包括问题、解决方案和 NIO 的介绍)的全面介绍吗？再没有比 Merlin Hughes 的 "Turning streams inside out " (*developerWorks*，2002 年 7 月)更好的了。
- 当然，还可以学习教程"Introduction to Java I/O" (*developerWorks*，2000 年 4 月)，它讨论了 JDK 1.4 之前的 Java I/O 的所有基础。
- John Zukowski 在其 *Merlin 的魔力* 专栏中撰写了一些关于 NIO 的优秀文章：
  - "" The ins and outs of Merlin's new I/O buffers " (*developerWorks*，2003 年 3 月)是介绍缓冲区基本知识的另一篇文章。
  - "" Character sets " " (*developerWorks*，2002 年 10 月)专门讨论字符集(特别是转换和编码模式)。
- 通过 Kalagnanam 和 Balu G 的 "" Merlin brings nonblocking I/O to the Java platform " (*developerWorks*，2002 年 3 月)进一步了解 NIO。
- Greg Travis 在他的 " JDK 1.4 Tutorial” (Manning 出版社，2002 年 3 月)一书中仔细研究了 NIO。
- 您可以在 developerWorks Java 技术专区 找到数百篇关于 Java 编程的各个方面的文章。