**缓冲区分配和包装**

在能够读和写之前，必须有一个缓冲区，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。

还可以将一个现有的数组转换为缓冲区：

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。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个 子缓冲区。也就是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。

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。

**缓冲区份片和数据共享**

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);

}

原缓冲区中的内容：

buffer.position( 0 );

buffer.limit(buffer.capacity());

while (buffer.hasRemaining()) {

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，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 — 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。

**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，要将一个 FileChannel (它的全部或者部分)映射到内存中。为此将使用 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，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。

import java.io.RandomAccessFile;

import java.nio.MappedByteBuffer;

import java.nio.channels.FileChannel;

public class UseMappedFile {

private static final int start = 0;

private static final int size = 1024;

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

RandomAccessFile raf = new RandomAccessFile("C:\\usemappedfile.txt", "rw");

FileChannel fc = raf.getChannel();

MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ\_WRITE, start, size);

mbb.put(0, (byte)97);

mbb.put(1023, (byte)122);

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。

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 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。

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**

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。

**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。**

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。

**创建一个 Selector**

Selector selector = Selector.open();

然后，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，需要一个 ServerSocketChannel。事实上，要监听的每一个端口都需要有一个 ServerSocketChannel 。对于每一个端口，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 。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，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，调用 Selector 的 selectedKeys() 方法，它返回发生了事件的 SelectionKey 对象的一个 集合 。

通过迭代 SelectionKeys 并依次处理每个 SelectionKey 来处理事件。对于每一个 SelectionKey，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，仅注册了 ServerSocketChannel，并且仅注册它们“接收”事件。为确认这一点，对 SelectionKey 调用 readyOps() 方法，并检查发生了什么类型的事件：

if(key.isAcceptable()) {

// 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.isReadable()) {

// 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。

回到主循环

每次返回主循环，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。