第 4 章 非阻塞通信

对于用 ServerSocket 以及 Socket 编写的服务器程序和客户程序，它们在运行过程中常常会阻塞。例如当一个线程执行 ServerSocket 的 accept()方法时，假如没有客户连接，该线程就会一直等到有了客户连接才从 accept()方法返回。再例如当线程执行 Socket 的输入流的 read()方法时，如果输入流中没有数据，该线程就会一直等到读入了足够的数据才从 read()方法返回。

假如服务器程序需要同时与多个客户通信，就必须分配多个工作线程，让它们分别负责与某个客户通信，当然每个工作线程都有可能经常处于长时间的阻塞状态。

从 JDK1.4 版本开始，引入了非阻塞的通信机制。服务器程序接收客户连接、客户程序请求建立与服务器的连接，以及服务器程序和客户程序收发数据的操作都可以按非阻塞的方式进行。服务器程序只需要创建一个线程，就能完成同时与多个客户通信的任务。

非阻塞的通信机制主要由 java.nio 包（新 I/O 包）中的类实现，主要的类包括 ServerSocketChannel、SocketChannel、Selector、SelectionKey 和 ByteBuffer 等。

本章介绍如何用 java.nio 包中的类来创建服务器程序和客户程序，并且分别采用阻塞模式和非阻塞模式来实现它们。通过比较不同的实现方式，可以帮助读者理解它们的区别和适用范围。

4.1 线程阻塞的概念

在生活中，最常见的阻塞现象是公路上汽车的堵塞。汽车在公路上快速运行，如果前方交通受阻，就只好停下来等待，等到公路顺畅，才能恢复运行。

线程在运行中也会因为某些原因而阻塞。所有处于阻塞状态的线程的共同特征是：放弃 CPU，暂停运行，只有等到导致阻塞的原因消除，才能恢复运行；或者被其他线程中断，该线程会退出阻塞状态，并且抛出 InterruptedException。

4.1.1 线程阻塞的原因

导致线程阻塞的原因主要有以下方面：

- 线程执行了 Thread.sleep(int n)方法，线程放弃 CPU，睡眠 n 毫秒，然后恢复运行。
- 线程要执行一段同步代码，由于无法获得相关的同步锁，只好进入阻塞状态，等到获得了同步锁，才能恢复运行。
- 线程执行了一个对象的 wait()方法，进入阻塞状态，只有等到其他线程执行了该对象的 notify()或 notifyAll()方法，才可能将其唤醒。
- 线程执行 I/O 操作或进行远程通信时，会因为等待相关的资源而进入阻塞状态。例如当线程执行 System.in.read()方法时，如果用户没有向控制台输入数据，则该线程会一直等读到了用户的输入数据才从 read()方法返回。

进行远程通信时，在客户程序中，线程在以下情况可能进入阻塞状态：

- 请求与服务器建立连接时，即当线程执行 Socket 的带参数的构造方法，或执行 Socket 的 connect()方法时，会进入阻塞状态，直到连接成功，此线程才从 Socket 的构造方法或 connect()方法返回。
- 线程从 Socket 的输入流读入数据时，如果没有足够的数据，就会进入阻塞状态，

直到读到了足够的数据，或者到达输入流的末尾，或者出现了异常，才从输入流的read()方法返回或异常中断。输入流中有多少数据才算足够呢？这要看线程执行的read()方法的类型：

（1）int read()：只要输入流中有一个字节，就算足够。

（2）int read(byte[] buff)：只要输入流中的字节数目与参数 buff 数组的长度相同，就算足够。

（3）String readLine()：只要输入流中有一行字符串，就算足够。值得注意的是，InputStream 类并没有 readLine()方法，在过滤流 BufferedReader 类中才有此方法。

- 线程向 Socket 的输出流写一批数据时，可能会进入阻塞状态，等到输出了所有的数据，或者出现异常，才从输出流的 write()方法返回或异常中断。
- 当调用 Socket 的 setSoLinger()方法设置了关闭 Socket 的延迟时间，那么当线程执行 Socket 的 close()方法时，会进入阻塞状态，直到底层 Socket 发送完所有剩余数据，或者超过了 setSoLinger()方法设置的延迟时间，才从 close()方法返回。

在服务器程序中，线程在以下情况可能会进入阻塞状态：

- 线程执行 ServerSocket 的 accept()方法，等待客户的连接，直到接收到了客户连接，才从 accept()方法返回。
- 线程从 Socket 的输入流读入数据时，如果输入流没有足够的数据，就会进入阻塞状态。
- 线程向 Socket 的输出流写一批数据时，可能会进入阻塞状态，等到输出了所有的数据，或者出现异常，才从输出流的 write()方法返回或异常中断。

由此可见，无论是在服务器程序还是客户程序中，当通过 Socket 的输入流和输出流来读写数据时，都可能进入阻塞状态。这种可能出现阻塞的输入和输出操作被称为阻塞 I/O。与此对照，如果执行输入和输出操作时，不会发生阻塞，则称为非阻塞 I/O。

4.1.2 服务器程序用多线程处理阻塞通信的局限

本书第 3 章的 3.6 节（创建多线程的服务器）已经介绍了服务器程序用多线程来同时处理多个客户连接的方式。服务器程序的处理流程如图 4-1 所示。主线程负责接收客户的连接。在线程池中有若干工作线程，它们负责处理具体的客户连接。每当主线程接收到一个客户连接，主线程就会把与这个客户交互的任务交一个空闲的工作线程去完成，主线程继续负责接收下一个客户连接。

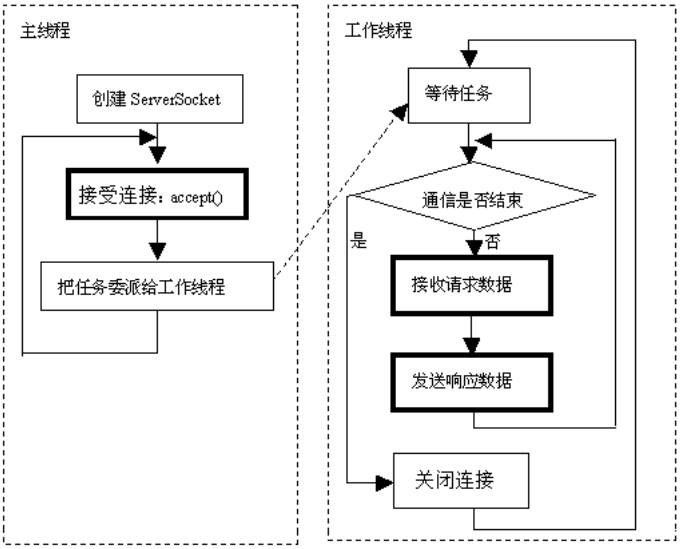


图 4-1　服务器程序用多线程处理阻塞通信

在图 4-1 中，用粗体框标识的步骤为可能引起阻塞的步骤。从图中可以看出，当主线程接收客户连接，以及工作线程执行 I/O 操作时，都有可能进入阻塞状态。

服务器程序用多线程来处理阻塞 I/O，尽管能满足同时响应多个客户请求的需求，但是有以下局限：

（1）Java 虚拟机会为每个线程分配独立的堆栈空间，工作线程数目越多，系统开销就越大，而且增加了 Java 虚拟机调度线程的负担，增加了线程之间同步的复杂性，提高了线程死锁的可能性。

（2）工作线程的许多时间都浪费在阻塞 I/O 操作上，Java 虚拟机需要频繁地转让 CPU 的使用权，使进入阻塞状态的线程放弃 CPU，再把 CPU 分配给处于可运行状态的线程。

由此可见，工作线程并不是越多越好。如图 4-2 所示，保持适量的工作线程，会提高服务器的并发性能，但是当工作线程的数目到达某个极限，超出了系统的负荷时，反而会降低并发性能，使得多数客户无法快速得到服务器的响应。

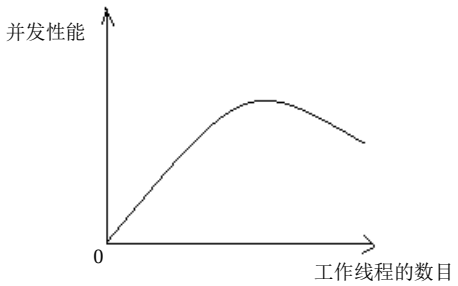


图 4-2　线程数目与并发性能的关系

4.1.3　非阻塞通信的基本思想

假如同时要做两件事：烧开水和烧粥。烧开水的步骤如下：

```
锅子里放水，打开煤气炉；
等待水烧开；//阻塞
关闭煤气炉，把开水灌到水壶里；
```

烧粥的步骤如下：

```
锅子里放水和米，打开煤气炉；
```

```
等待粥烧开；//阻塞
调整煤气炉，改为小火；
等待粥烧熟；//阻塞
关闭煤气炉；
```

为了同时完成两件事，一种方案是同时请两个人分别做其中的一件事，这相当于采用多线程来同时完成多个任务。还有一种方案是让一个人同时完成两件事，这个人应该善于利用一件事的空闲时间去做另一件事，这个人一刻也不应该闲着：

```
锅子里放水，打开煤气炉；//开始烧开水
锅子里放水和米，打开煤气炉；//开始烧粥
while(一直等待，直到有水烧开、粥烧开或粥烧熟事件发生){  //阻塞
  if(水烧开)
    关闭煤气炉，把开水灌到水壶里；
  if(粥烧开)
    调整煤气炉，改为小火；
  if(粥烧熟)
    关闭煤气炉；
}
```

这个人不断监控烧水以及烧粥的状态，如果发生了“水烧开”、“粥烧开”或“粥烧熟”事件，就去处理这些事件，处理完一件事后继续监控烧水以及烧粥的状态，直到所有的任务都完成。

以上工作方式也可以运用到服务器程序中，服务器程序只需要一个线程就能同时负责接收客户的连接、接收各个客户发送的数据，以及向各个客户发送响应数据。服务器程序的处理流程如下：

```
while(一直等待，直到有接收连接就绪事件、读就绪事件或写就绪事件发生){ //阻塞
  if(有客户连接)
    接收客户的连接；  //非阻塞
  if(某个 Socket 的输入流中有可读数据)
    从输入流中读数据；  //非阻塞
  if(某个 Socket 的输出流可以写数据)
    向输出流写数据；  //非阻塞
}
```

以上处理流程采用了轮询的工作方式，当某一种操作就绪，就执行该操作，否则就察看是否还有其他就绪的操作可以执行。线程不会因为某一个操作还没有就绪，就进入阻塞状态，一直傻傻地在那里等待这个操作就绪。

为了使轮询的工作方式顺利进行，接收客户的连接、从输入流读数据、以及向输出流写数据的操作都应该以非阻塞的方式运行。所谓非阻塞，就是指当线程执行这些方法时，如果操作还没有就绪，就立即返回，而不会一直等到操作就绪。例如当线程接收客户连接时，如果没有客户连接，就立即返回；再例如当线程从输入流中读数据时，如果输入流中还没有数据，就立即返回，或者如果输入流还没有足够的数据，那么就读取现有的数据，然后返回。值得注意的是，以上 while 循环条件中的操作还是按照阻塞方式进行的，如果未发生任何事件，就会进入阻塞状态，直到接收连接就绪事件、读就绪事件或写就绪事件中至少有一个事件发生，此时就会执行 while 循环体中的操作。

4.2　java.nio 包中的主要类

java.nio 包提供了支持非阻塞通信的类，主要包括：

- ServerSocketChannel：ServerSocket 的替代类，支持阻塞通信与非阻塞通信。
- SocketChannel：Socket 的替代类，支持阻塞通信与非阻塞通信。
- Selector：为 ServerSocketChannel 监控接收连接就绪事件，为 SocketChannel 监控连接就绪、读就绪和写就绪事件。
- SelectionKey：代表 ServerSocketChannel 以及 SocketChannel 向 Selector 注册事件的句柄。当一个 SelectionKey 对象位于 Selector 对象的 selected-keys 集合中，就表示与这个 SelectionKey 对象相关的事件发生了。

ServerSocketChannel 以及 SocketChannel 都是 SelectableChannel 的子类，参见图 4-3。SelectableChannel 类以及其子类都能委托 Selector 来监控它们可能发生的一些事件，这种委托过程也称为注册事件过程。

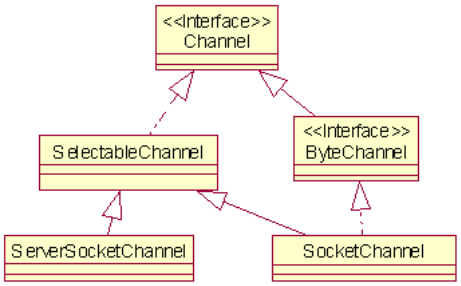


图 4-3　SelectableChannel 类及其子类的类框图

ServerSocketChannel 向 Selector 注册接收连接就绪事件的代码如下：

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

SelectionKey 类的一些静态常量表示事件类型，ServerSocketChannel 只可能发生一种事件：

- SelectionKey.OP_ACCEPT：接收连接就绪事件，表示至少有了一个客户连接，服务器可以接收这个连接。

SocketChannel 可能发生以下三种事件：

- SelectionKey.OP_CONNECT：连接就绪事件，表示客户与服务器的连接已经建立成功。
- SelectionKey.OP_READ：读就绪事件，表示输入流中已经有了可读数据，可以执行读操作了。
- SelectionKey.OP_WRITE：写就绪事件，表示已经可以向输出流写数据了。

SocketChannel 提供了接收和发送数据的方法：

- read(ByteBuffer buffer)：接收数据，把它们存放到参数指定的 ByteBuffer 中。
- write(ByteBuffer buffer)：把参数指定的 ByteBuffer 中的数据发送出去。

ByteBuffer 表示字节缓冲区，SocketChannel 的 read()和 write()方法都会操纵 ByteBuffer。ByteBuffer 类继承于 Buffer 类。ByteBuffer 中存放的是字节，为了把它们转换为字符串，还需要用到 Charset 类，Charset 类代表字符编码，它提供了把字节流转换为字符串（解码过程）和把字符串转换为字节流（编码过程）的实用方法。

下面几小节分别介绍 Buffer、Charset、SelectableChannel、ServerSocketChannel、SocketChannel、Selector 和 SelectionKey 的用法。如果读者觉得单独看这些类的用法太枯燥，可以先阅读本章的 4.3 节，它介绍如何用这些类来创建 EchoServer 服务器程序。

4.2.1 缓冲区 Buffer

数据输入和输出往往是比较耗时的操作。缓冲区从两个方面提高 I/O 操作的效率：

- 减少实际的物理读写次数。
- 缓冲区在创建时被分配内存，这块内存区域一直被重用，这可以减少动态分配和回收内存区域的次数。

旧 I/O 类库（对应 java.io 包）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。图 4-4 显示了 Buffer 类的层次结构。

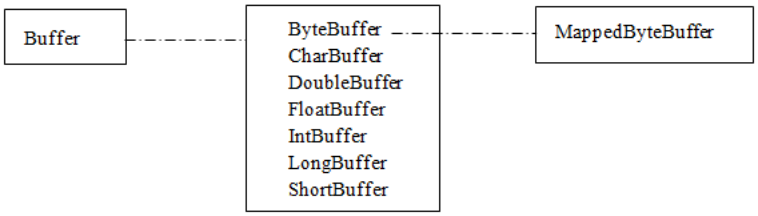


图 4-4 Buffer 类的层次结构

所有的缓冲区都有以下属性：

- 容量（capacity）：表示该缓冲区可以保存多少数据。
- 极限（limit）：表示缓冲区的当前终点，不能对缓冲区中超过极限的区域进行读写操作。极限是可以修改的，这有利于缓冲区的重用。例如，假定容量为 100 的缓冲区已经填满了数据，接着程序在重用缓冲区时，仅仅将 10 个新的数据写入缓冲区中从位置 0 到 10 的区域，这时可以将极限设为 10，这样就不能读取位置从 11 到 99 的原先的数据了。极限是一个非负整数，不应该大于容量。
- 位置（position）：表示缓冲区中下一个读写单元的位置，每次读写缓冲区的数据时，都会改变该值，为下一次读写数据作准备。位置是一个非负整数，不应该大于极限。

如图 4-5 所示，以上三个属性的关系为：容量>=极限>=位置>=0

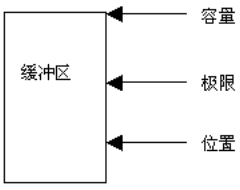


图 4-5 缓冲区的三个属性

缓冲区提供了用于改变以上三个属性的方法：

- clear()：把极限设为容量，再把位置设为 0。
- flip()：把极限设为位置，再把位置设为 0。
- rewind()：不改变极限，把位置设为 0。

Buffer 类的 remaining()方法返回缓冲区的剩余容量，取值等于极限-位置。Buffer 类的 compact()方法删除缓冲区内从 0 到当前位置 position 的内容，然后把从当前位置 position 到

极限 limit 的内容拷贝到 0 到 limit-position 的区域内，当前位置 position 和极限 limit 的取值也做相应的变化，参见图 4-6。

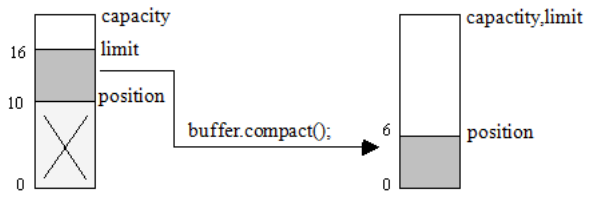


图 4-6　Buffer 类的 compact()的作用

java.nio.Buffer 类是一个抽象类，不能被实例化。共有 8 个具体的缓冲区类，其中最基本的缓冲区是 ByteBuffer，它存放的数据单元是字节。ByteBuffer 类并没有提供公开的构造方法，但是提供了两个获得 ByteBuffer 实例的静态工厂方法：

- allocate(int capacity)：返回一个 ByteBuffer 对象，参数 capacity 指定缓冲区的容量。
- directAllocate(int capacity)：返回一个 ByteBuffer 对象，参数 capacity 指定缓冲区的容量。该方法返回的缓冲区称为直接缓冲区，它与当前操作系统能够更好地耦合，因此能进一步提高 I/O 操作的速度。但是分配直接缓冲区的系统开销很大，因此只有在缓冲区较大并且长期存在，或者需要经常重用时，才使用这种缓冲区。

除 boolean 类型以外，每种基本类型都有对应的缓冲区类，包括 CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer 和 ShortBuffer。这几个缓冲区类都有一个能够返回自身实例的静态工厂方法 allocate(int capacity)。在 CharBuffer 中存放的数据单元为字符，在 DoubleBuffer 中存放的数据单元为 double 数据，以此类推。还有一种缓冲区是 MappedByteBuffer，它是 ByteBuffer 的子类。MappedByteBuffer 能够把缓冲区和文件的某个区域直接映射。

所有具体缓冲区类都提供了读写缓冲区的方法：

- get()：相对读。从缓冲区的当前位置读取一个单元的数据，读完后把位置加 1。
- get(int index)：绝对读。从参数 index 指定的位置读取一个单元的数据。
- put(单元数据类型 data)：相对写。向缓冲区的当前位置写入一个单元的数据，写完后把位置加 1。
- put(int index，单元数据类型 data)：绝对写。向参数 index 指定的位置写入一个单元的数据。

ByteBuffer 类不仅可以读取和写入一个单元的字节，还可以读取和写入 int、char、float 和 double 等基本类型的数据，例如：

- getInt()
- getInt(int index)
- putInt(int value)
- putInt(int index, int value)
- getChar()
- getChar(int index)
- putChar(char value)
- putChar(int index, char value)

以上不带 index 参数的方法会在当前位置读取或写入数据，称为相对读写。带 index 参数的方法会在 index 参数指定的位置读取或写入数据，称为绝对读写。

ByteBuffer 类还提供了用于获得缓冲区视图的方法，例如：

- ShortBuffer asShortBuffer()
- CharBuffer asCharBuffer()
- IntBuffer asIntBuffer()
- FloatBuffer asFloatBuffer()

例如以下程序代码获取 ByteBuffer 的 CharBuffer 缓冲区视图：

```
CharBuffer charBuffer=byteBuffer.asCharBuffer();
```

以上 CharBuffer 视图和底层 ByteBuffer 共享同样的数据，修改 CharBuffer 视图的数据，会反映到底层 ByteBuffer。不过，CharBuffer 视图和底层 ByteBuffer 有各自独立的位置 position、极限 limit 和容量 capacity 属性。

4.2.2 字符编码 Charset

java.nio.Charset 类的每个实例代表特定的字符编码类型。如图 4-7 所示，把字节序列转换为字符串的过程称为解码；把字符串转换为字节序列的过程称为编码。

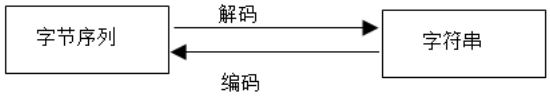


图 4-7 编码与解码

Charset 类提供了编码与解码的方法：

- ByteBuffer encode(String str)：对参数 str 指定的字符串进行编码，把得到的字节序列存放在一个 ByteBuffer 对象中，并将其返回。
- ByteBuffer encode(CharBuffer cb)：对参数 cb 指定的字符缓冲区中的字符进行编码，把得到的字节序列存放在一个 ByteBuffer 对象中，并将其返回。
- CharBuffer decode(ByteBuffer bb)：对参数 bb 指定的 ByteBuffer 中的字节序列进行解码，把得到的字符序列存放在一个 CharBuffer 对象中，并将其返回。

Charset 类的静态 forName(String encode)方法返回一个 Charset 对象，它代表参数 encode 指定的编码类型。例如以下代码创建了一个代表“GBK”编码的 Charset 对象：

```
Charset charset=Charset.forName("GBK");
```

Charset 类还有一个静态方法 defaultCharset()，它返回代表本地平台的默认字符编码的 Charset 对象。

4.2.3 通道 Channel

通道 Channel 用来连接缓冲区与数据源或数据汇（即数据目的地）。如图 4-8 所示，数据源的数据经过通道到达缓冲区，缓冲区的数据经过通道到达数据汇。

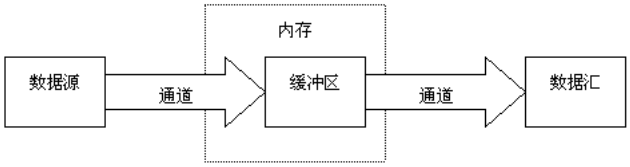


图 4-8 通道的作用

图 4-9 显示了 Channel 的主要层次结构。

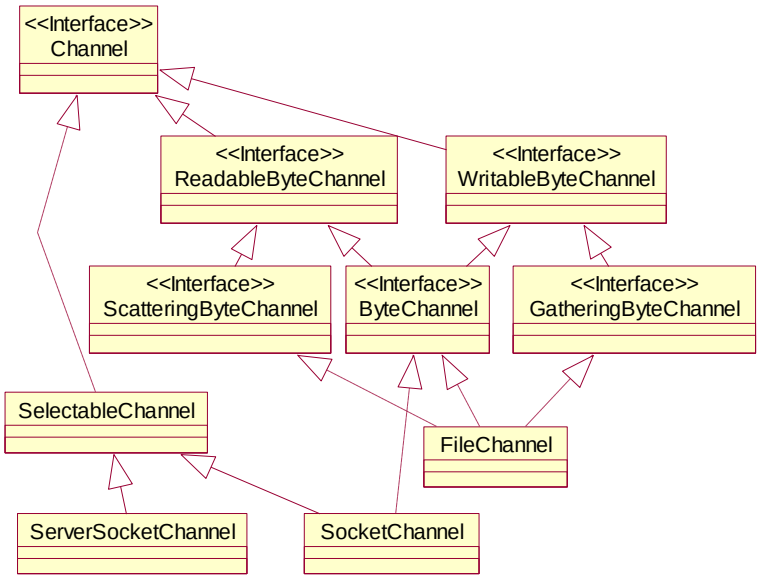


图 4-9　Channel 的主要层次结构

java.nio.channels.Channel 接口只声明了两个方法：

- close()：关闭通道。
- isOpen()：判断通道是否打开。

通道在创建时被打开，一旦关闭通道，就不能重新打开它。

Channel 接口的两个最重要的子接口是 ReadableByteChannel 和 WritableByteChannel。ReadableByteChannel 接口声明了 read(ByteBuffer dst)方法，该方法把数据源的数据读入到参数指定的 ByteBuffer 缓冲区中。WritableByteChannel 接口声明了 write(ByteBuffer src)方法，该方法把参数指定的 ByteBuffer 缓冲区中的数据写到数据汇中。图 4-10 显示了 Channel 与 Buffer 的关系。ByteChannel 接口是一个便利接口，它扩展了 ReadableByteChannel 和 WritableByteChannel 接口，因而同时支持读写操作。

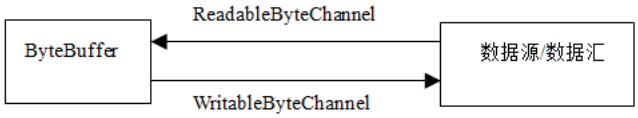


图 4-10　Channel 与 Buffer 的关系

ScatteringByteChannel 接口扩展了 ReadableByteChannel 接口，允许分散地读取数据。分散读取数据是指单个读取操作能填充多个缓冲区。ScatteringByteChannel 接口声明了 read(ByteBuffer[] dsts)方法，该方法把从数据源读取的数据依次填充到参数指定的 ByteBuffer 数组的各个 ByteBuffer 中。GatheringByteChannel 接口扩展了 WritableByteChannel 接口，允许集中地写入数据。集中写入数据是指单个写操作能把多个缓冲区的数据写到数据汇。GatheringByteChannel 接口声明了 write(ByteBuffer[] srcs)方法，该方法依次把参数指定的 ByteBuffer 数组的每个 ByteBuffer 中的数据写到数据汇。分散读取和集中写数据能够进一步提高输入和输出操作的速度。

FileChannel 类是 Channel 接口的实现类，代表一个与文件相连的通道。该类实现了

ByteChannel、ScatteringByteChannel 和 GatheringByteChannel 接口，支持读操作、写操作、分散读操作和集中写操作。FileChannel 类没有提供公开的构造方法，因此客户程序不能用 new 语句来构造它的实例。不过，在 FileInputStream、FileOutputStream 和 RandomAccessFile 类中提供了 getChannel()方法，该方法返回相应的 FileChannel 对象。

SelectableChannel 也是一种通道，它不仅支持阻塞的 I/O 操作，还支持非阻塞的 I/O 操作。SelectableChannel 有两个子类：ServerSocketChannel 和 SocketChannel。SocketChannel 还实现了 ByteChannel 接口，具有 read(ByteBuffer dst)和 write(ByteBuffer src)方法。

4.2.4 SelectableChannel 类

SelectableChannel 是一种支持阻塞 I/O 和非阻塞 I/O 的通道。在非阻塞模式下，读写数据不会阻塞，并且 SelectableChannel 可以向 Selector 注册读就绪和写就绪等事件。Selector 负责监控这些事件，等到事件发生时，比如发生了读就绪事件，SelectableChannel 就可以执行读操作了。

SelectableChannel 的主要方法如下。

- public SelectableChannel configureBlocking(boolean block) throws IOException

当参数 block 为 true，表示把 SelectableChannel 设为阻塞模式，如果参数为 false，表示把 SelectableChannel 设为非阻塞模式。默认情况下，SelectableChannel 采用阻塞模式。该方法返回 SelectableChannel 对象本身的引用，相当于“return this”。

SelectableChannel 的 isBlocking()方法判断 SelectableChannel 是否处于阻塞模式，如果返回 true，表示处于阻塞模式，否则表示处于非阻塞模式。

- public SelectionKey register(Selector sel,int ops)throws ClosedChannelException
- public SelectionKey register(Selector sel,int ops,Object attachment) throws ClosedChannelException

以上两个方法都向 Selector 注册事件，例如以下 socketChannel（SelectableChannel 的一个子类）向 Selector 注册读就绪和写就绪事件：

```
SelectionKey key=socketChannel.register(selector,
            SelectionKey.OP_READ |SelectionKey.OP_WRITE);
```

register()方法返回一个 SelectionKey 对象，SelectionKey 用来跟踪被注册的事件。第二个 register()方法还有一个 Object 类型的参数 attachment，它用于为 SelectionKey 关联一个附件，当被注册事件发生后，需要处理该事件时，可以从 SelectionKey 中获得这个附件，该附件可用来包含与处理这个事件相关的信息。以下这两段代码是等价的：

```
MyHandler handler=new MyHandler();  //负责处理事件的对象
SelectionKey key=socketChannel.register(selector,
    SelectionKey.OP_READ| SelectionKey.OP_WRITE,handler);

等价于：

SelectionKey key=socketChannel.register(selector,
    SelectionKey.OP_READ| SelectionKey.OP_WRITE);
MyHandler handler=new MyHandler();
key.attach(handler);  //为 SelectionKey 关联一个附件
```

4.2.5 ServerSocketChannel 类

ServerSocketChannel 从 SelectableChannel 中继承了 configureBlocking()和 register()方法。ServerSocketChannel 是 ServerSocket 的替代类，也具有负责接收客户连接的 accept()方法。

ServerSocketChannel 并没有 public 类型的构造方法，必须通过它的静态方法 open()来创建 ServerSocketChannel 对象。每个 ServerSocketChannel 对象都与一个 ServerSocket 对象关联。ServerSocketChannel 的 socket()方法返回与它关联的 ServerSocket 对象。可通过以下方式把服务器进程绑定到一个本地端口：

```
serverSocketChannel.socket().bind(port);
```

ServerSocketChannel 的主要方法如下。

- public static ServerSocketChannel open()throws IOException

这是 ServerSocketChannel 类的静态工厂方法，它返回一个 ServerSocketChannel 对象。这个对象没有与任何本地端口绑定，并且处于阻塞模式。

- public SocketChannel accept()throws IOException

类似于 ServerSocket 的 accept()方法，用于接收客户的连接。如果 ServerSocketChannel 处于非阻塞模式，当没有客户连接时，该方法立即返回 null。如果 ServerSocketChannel 处于非阻塞模式，当没有客户连接时，它会一直阻塞下去，直到有客户连接就绪，或者出现了 IOException。

值得注意的是，该方法返回的 SocketChannel 对象处于阻塞模式，如果希望把它改为非阻塞模式，必须执行以下代码：

```
socketChannel.configureBlocking(false);
```

- public final int validOps()

返回 ServerSocketChannel 所能产生的事件，这个方法总是返回 SelectionKey.OP_ACCEPT。

- public ServerSocket socket()

返回与 ServerSocketChannel 关联的 ServerSocket 对象。每个 ServerSocketChannel 对象都与一个 ServerSocket 对象关联。

4.2.6 SocketChannel 类

SocketChannel 可看作是 Socket 的替代类，但它比 Socket 具有更多的功能。SocketChannel 不仅从 SelectableChannel 父类中继承了 configureBlocking()和 register()方法，而且实现了 ByteChannel 接口，因此具有用于读写数据的 read(ByteBuffer dst)和 write(ByteBuffer src)方法。SocketChannel 没有 public 类型的构造方法，必须通过它的静态方法 open()来创建 SocketChannel 对象。

SocketChannel 的主要方法如下。

- public static SocketChannel open() throws IOException
- public static SocketChannel open(SocketAddress remote) throws IOException

SocketChannel 的静态工厂方法 open()负责创建 SocketChannel 对象，第二个带参数的构造方法还会建立与远程服务器的连接。在阻塞模式以及非阻塞模式下，第二个 open()方法有不同的行为，这与 SocketChannel 类的 connect()方法类似，参见本节 connect()方法的介绍。

以下两段代码是等价的：

```
SocketChannle socketChannel= SocketChannel.open();
channel.connect(remote);  //remote 为 SocketAddress 类型

等价于：
```

```
//remote 为 SocketAddress 类型
SocketChannle socketChannel = SocketChannel.open(remote);
```

值得注意的是，open()方法返回的 SocketChannel 对象处于阻塞模式，如果希望把它改为非阻塞模式，必须执行以下代码：

```
socketChannel.configureBlocking(false);
```

- public final int validOps()

返回 SocketChannel 所能产生的事件，这个方法总是返回以下值：

```
SelectionKey.OP_CONNECT | SelectionKey.OP_READ | SelectionKey.OP_WRITE
```

- public Socket socket()

返回与这个 SocketChannel 关联的 Socket 对象。每个 SocketChannel 对象都与一个 Socket 对象关联。

- public boolean isConnected()

判断底层 Socket 是否已经建立了远程连接。

- public boolean isConnectionPending()

判断是否正在进行远程连接。当远程连接操作已经开始，但还没有完成，则返回 true，否则返回 false。也就是说，当底层 Socket 还没有开始连接，或者已经连接成功，该方法都会返回 false。

- public boolean connect(SocketAddress remote)throws IOException

使底层 Socket 建立远程连接。当 SocketChannel 处于非阻塞模式，如果立即连接成功，该方法返回 true，如果不能立即连接成功，该方法返回 false，程序过会儿必须通过调用 finishConnect()方法来完成连接。当 SocketChannel 处于阻塞模式，如果立即连接成功，该方法返回 true，如果不能立即连接成功，将进入阻塞状态，直到连接成功，或者出现 I/O 异常。

- public boolean finishConnect()throws IOException

试图完成连接远程服务器的操作。在非阻塞模式下，建立连接从调用 SocketChannel 的 connect()方法开始，到调用 finishConnect()方法结束。如果 finishConnect()方法顺利完成连接，或者在调用此方法之前连接已经建立，则 finishConnect()方法立即返回 true。如果连接操作还没有完成，则立即返回 false。如果连接操作中遇到异常而失败，则抛出相应的 I/O 异常。

在阻塞模式下，如果连接操作还没有完成，则会进入阻塞状态，直到连接完成，或者出现 I/O 异常。

- public int read(ByteBuffer dst)throws IOException

从 Channel 中读入若干字节，把它们存放到参数指定的 ByteBuffer 中。假定执行 read()方法前，ByteBuffer 的位置为 p，剩余容量为 r，r 等于 dst.remaining()方法的返回值。假定 read()方法实际上读入了 n 个字节，那么 0<=n<=r。read()方法返回后，参数 dst 引用的 ByteBuffer 的位置变为 p+n，极限保持不变，参见图 4-11。

图 4-11　read()方法读入 n 个字节

在阻塞模式下，read()方法会争取读到 r 个字节，如果输入流中不足 r 个字节，就进入阻塞状态，直到读入了 r 个字节，或者读到了输入流末尾，或者出现了 I/O 异常。

在非阻塞模式下，read()方法奉行能读到多少数据就读多少数据的原则。read()方法读取当前通道中的可读数据，有可能不足 r 个字节，或者为 0 个字节，read()方法总是立即返回，而不会等到读取了 r 个字节再返回。

read()方法返回实际上读入的字节数，有可能为 0。如果返回-1，就表示读到了输入流的末尾。

- public int write(ByteBuffer src)throws IOException

把参数 src 指定的 ByteBuffer 中的字节写到 Channel 中。假定执行 write()方法前，ByteBuffer 的位置为 p，剩余容量为 r，r 等于 src.remaining()方法的返回值。假定 write()方法实际上向通道中写了 n 个字节，那么 0<=n<=r。write()方法返回后，参数 src 引用的 ByteBuffer 的位置变为 p+n，极限保持不变，参见图 4-12。

图 4-12　write()方法输出 n 个字节

在阻塞模式下，write()方法会争取输出 r 个字节，如果底层网络的输出缓冲区不能容纳 r 个字节，就进入阻塞状态，直到输出了 r 个字节，或者出现了 I/O 异常。

在非阻塞模式下，write()方法奉行能输出多少数据就输出多少数据的原则，有可能不足 r 个字节，或者为 0 个字节，write()方法总是立即返回，而不会等到输出 r 个字节再返回。

write()方法返回实际上输出的字节数，有可能为 0。

4.2.7　Selector 类

只要 ServerSocketChannel 以及 SocketChannel 向 Selector 注册了特定的事件，Selector 就会监控这些事件是否发生。SelectableChannel 的 register()方法负责注册事件，该方法返回一个 SelectionKey 对象，该对象是用于跟踪这些被注册事件的句柄。一个 Selector 对象中会包含三种类型的 SelectionKey 的集合：

- all-keys 集合：当前所有向 Selector 注册的 SelectionKey 的集合，Selector 的 keys()方法返回该集合。
- selected-keys 集合：相关事件已经被 Selector 捕获的 SelectionKey 的集合。Selector 的 selectedKeys()方法返回该集合。
- cancelled-keys 集合：已经被取消的 SelectionKey 的集合。Selector 没有提供访问这种集合的方法。

以上第二种和第三种集合都是第一种集合的子集。对于一个新建的 Selector 对象，它的上述集合都为空。

当执行 SelectableChannel 的 register()方法，该方法新建一个 SelectionKey，并把它加入到 Selector 的 all-keys 集合中。

如果关闭了与 SelectionKey 对象关联的 Channel 对象，或者调用了 SelectionKey 对象的 cancel()方法，这个 SelectionKey 对象就会被加入到 cancelled-keys 集合中，表示这个 SelectionKey 对象已经被取消，在程序下一次执行 Selector 的 select()方法时，被取消的 SelectionKey 对象将从所有的集合（包括 all-keys 集合、selected-keys 集合和 cancelled-keys

集合）中删除。

在执行 Selector 的 select()方法时，如果与 SelectionKey 相关的事件发生了，这个 SelectionKey 就被加入到 selected-keys 集合中。程序直接调用 selected-keys 集合的 remove()方法，或者调用它的 Iterator 的 remove()方法，都可以从 selected-keys 集合中删除一个 SelectionKey 对象。

程序不允许直接通过集合接口的 remove()方法删除 all-keys 集合中的 SelectionKey 对象。如果程序试图这么做，那么会导致 UnsupportedOperationException。

Selector 类的主要方法如下。

- public static Selector open()throws IOException

这是 Selector 的静态工厂方法，创建一个 Selector 对象。

- public boolean isOpen()

判断 Selector 是否处于打开状态。Selector 对象创建后就处于打开状态，当调用了 Selector 对象的 close()方法，它就进入关闭状态。

- public Set<SelectionKey> keys()

返回 Selector 的 all-keys 集合，它包含了所有与 Selector 关联的 SelectionKey 对象。

- public int selectNow()throws IOException

返回相关事件已经发生的 SelectionKey 对象的数目。该方法采用非阻塞的工作方式，返回当前相关事件已经发生的 SelectionKey 对象的数目，如果没有，就立即返回 0。

- public int select()throws IOException
- public int select(long timeout)throws IOException

返回相关事件已经发生的 SelectionKey 对象的数目。该方法采用阻塞的工作方式，返回相关事件已经发生的 SelectionKey 对象的数目，如果一个也没有，就进入阻塞状态，直到出现以下情况之一，就会从 select()方法中返回：

（1）至少有一个 SelectionKey 的相关事件已经发生。

（2）其他线程调用了 Selector 的 wakeup()方法，导致执行 select()方法的线程立即从 select()方法中返回。

（3）当前执行 select()方法的线程被其他线程中断。

（4）超出了等待时间。该时间由 select(long timeout)方法的参数 timeout 设定，单位为毫秒。如果等待超时，就会正常返回，但不会抛出超时异常。如果程序调用的是不带参数的 select()方法，那么永远不会超时，这意味着执行 select()方法的线程进入阻塞状态后，永远不会因为超时而中断。

- public Selector wakeup()

唤醒执行 Selector 的 select()方法（也同样适用于 select(long timeout)方法）的线程。当线程 A 执行 Selector 对象的 wakeup()方法时，如果线程 B 正在执行同一个 Selector 对象的 select()方法，或者线程 B 过一会儿会执行这个 Selector 对象的 select()方法，那么线程 B 在执行 select()方法时，会立即从 select()方法中返回，而不会被阻塞。假如线程 B 已经在 select()方法中阻塞了，也会立刻被唤醒，从 select()方法中返回。

wakeup()方法只能唤醒执行 select()方法的线程 B 一次。如果线程 B 在执行 select()方法时被唤醒后，以后再执行 select()方法，则仍旧按照阻塞方式工作，除非线程 A 再次调用 Selector 对象的 wakeup()方法。

- public void close()throws IOException

关闭 Selector。如果有其他线程正执行这个 Selector 的 select()方法并且处于阻塞状态，

那么这个线程会立即返回。close()方法使得 Selector 占用的所有资源都被释放，所有与 Selector 关联的 SelectionKey 都被取消。

4.2.8 SelectionKey 类

ServerSocketChannel 或 SocketChannel 通过 register()方法向 Selector 注册事件时，register()方法会创建一个 SelectionKey 对象，这个 SelectionKey 对象是用来跟踪注册事件的句柄。在 SelectionKey 对象的有效期间，Selector 会一直监控与 SelectionKey 对象相关的事件，如果事件发生，就会把 SelectionKey 对象加入到 selected-keys 集合中。在以下情况，SelectionKey 对象会失效，这意味着 Selector 再也不会监控与它相关的事件：

（1）程序调用 SelectionKey 的 cancel()方法。

（2）关闭与 SelectionKey 关联的 Channel。

（3）与 SelectionKey 关联的 Selector 被关闭。

在 SelectionKey 中定义了四种事件，分别用 4 个 int 类型的常量来表示：

- SelectionKey.OP_ACCEPT：接收连接就绪事件，表示服务器监听到了客户连接，服务器可以接收这个连接了。常量值为 16
- SelectionKey.OP_CONNECT：连接就绪事件，表示客户与服务器的连接已经建立成功。常量值为 8。
- SelectionKey.OP_READ：读就绪事件，表示通道中已经有了可读数据，可以执行读操作了。常量值为 1。
- SelectionKey.OP_WRITE：写就绪事件，表示已经可以向通道写数据了。常量值为 4。

以上常量分别占居不同的二进制位，因此可以通过二进制的或运算“|”，来将它们进行任意组合。一个 SelectionKey 对象中包含两种类型的事件：

- 所有感兴趣的事件：SelectionKey 的不带参数的 interestOps()方法返回所有感兴趣的事件，例如假定返回值为 SelectionKey.OP_WRITE | SelectionKey.OP_READ，就表示这个 SelectionKey 对读就绪和写就绪事件感兴趣。与之关联的 Selector 对象会负责监控这些事件。当通过 SelectableChannel 的 register()方法注册事件时，可以在参数中指定 SelectionKey 感兴趣的事件，例如以下代码表明新建的 SelectionKey 对连接就绪和读就绪事件感兴趣：

```
SelectionKey key=socketChannel.register(selector,
          SelectionKey.OP_CONNECT | SelectionKey.OP_READ);
```

SelectionKey 的带参数的 interestOps(int ops)方法用于为 SelectionKey 对象增加一个感兴趣的事件。例如以下代码使得 SelectionKey 增加了一个感兴趣的事件：

```
key. interestOps(SelectionKey.OP_WRITE);
```

- 所有已经发生的事件：SelectionKey 的 readyOps()方法返回所有已经发生的事件，例如假定返回值为 SelectionKey.OP_WRITE | SelectionKey.OP_READ，表示读就绪和写就绪事件已经发生了，这意味着与之关联的 SocketChannel 对象可以进行读操作和写操作了。

当程序调用一个 SelectableChannel（包括 ServerSocketChannel 和 SocketChannel）的 register(selector,XXX)方法，该方法会建立 SelectableChannel 对象、Selector 对象，以及 register()方法所创建的 SelectionKey 对象之间的关联关系，参见图 4-13。SelectionKey 的 channel()方法返回与它关联的 SelectableChannel 对象，selector()方法返回与它关联的 Selector 对象。

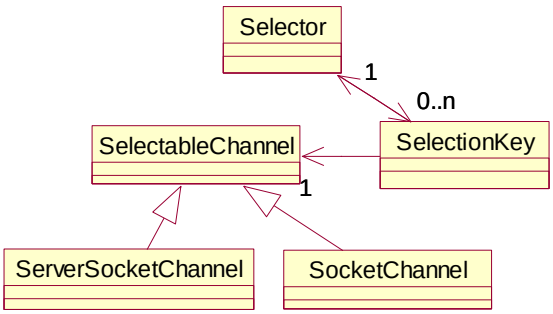


图 4-13　SelectionKey、Selector 与 SelectableChannel 之间的关联关系

SelectionKey 的主要方法如下。

- public SelectableChannel channel()

返回与这个 SelectionKey 对象关联的 SelectableChannel 对象。

- public Selector selector()

返回与这个 SelectionKey 对象关联的 Selector 对象。

- public boolean isValid()

判断这个 SelectionKey 是否有效。当 SelectionKey 对象创建后，它就一直处于有效状态。如果调用了它的 cancel()方法，或者关闭了与它关联的 SelectableChannel 或 Selector 对象，它就失效。

- public void cancel()

使 SelectionKey 对象失效。该方法把 SelectionKey 对象加入到与它关联的 Selector 对象的 cancelled-keys 集合中。当程序下一次执行 Selector 的 select()方法时，该方法会把 SelectionKey 对象从 Selector 对象的 all-keys、selected-keys 和 cancelled-keys 这三个集合中删除。

- public int interestOps()

返回这个 SelectionKey 感兴趣的事件。

- public SelectionKey interestOps(int ops)

为 SelectionKey 增加感兴趣的事件。该方法返回当前 SelectionKey 对象本身的引用，相当于“return this”。

- public int readyOps()

返回已经就绪的事件。

- public final boolean isReadable()

判断与之关联的 SocketChannel 的读就绪事件是否已经发生。该方法等价于：

```
key.readyOps() & SelectionKey.OP_READ != 0
```

- public final boolean isWritable()

判断与之关联的 SocketChannel 的写就绪事件是否已经发生。该方法等价于：

```
key.readyOps() & SelectionKey.OP_WRITE != 0
```

- public final boolean isConnectable()

判断与之关联的 SocketChannel 的连接就绪事件是否已经发生。该方法等价于：

```
k.readyOps() & SelectionKey.OP_CONNECT != 0
```

- public final boolean isAcceptable()

判断与之关联的 ServerSocketChannel 的接收连接就绪事件是否已经发生。该方法等价于：

```
k.readyOps() & SelectionKey.OP_ACCEPT != 0
```

- public final Object attach(Object ob)

使 SelectionKey 关联一个附件。一个 SelectionKey 对象只能关联一个 Object 类型的附件。如果多次调用该方法，则只有最后一个附件与 SelectionKey 对象关联。调用 SelectionKey 对象的 attachment()方法可获得这个附件。

- public final Object attachment()

返回与 SelectionKey 对象关联的附件。

4.2.9 Channels 类

Channels 类是一个简单的工具类，提供了通道与传统的基于 I/O 的流、Reader 和 Writer 之间进行转换的静态方法：

- ReadableByteChannel newChannel(InputStream in)：输入流转换成读通道。
- WritableByteChannel newChannel(OutputStream out)：输出流转换成写通道。
- InputStream newInputStream(AsynchronousByteChannel ch)：异步通道转换成输入流。
- InputStream newInputStream(ReadableByteChannel ch)：读通道转换成输入流。
- OutputStream newOutputStream(AsynchronousByteChannel ch)：异步通道转换成输出流。
- OutputStream newOutputStream(WritableByteChannel ch) ：写通道转换成输出流。
- Reader newReader(ReadableByteChannel ch, String csName)：读通道转换成 Reader。参数 csName 指定字符编码。
- Reader newReader(ReadableByteChannel ch, Charset charset)：读通道转换成 Reader。参数 charset 指定字符编码。
- Reader newReader(ReadableByteChannel ch, CharsetDecoder dec, int minBufferCap)：读通道转换成 Reader。参数 dec 指定字符解码器。参数 minBufferCap 指定内部字节缓冲区的最小容量。
- Writer newWriter(WritableByteChannel ch, String csName)：写通道转换成 Writer。参数 csName 指定字符编码。
- Writer newWriter(WritableByteChannel ch, Charset charset)：写通道转换成 Writer。参数 charset 指定字符编码。
- Writer newWriter(WritableByteChannel ch, CharsetEncoder enc, int minBufferCap)：写通道转换成 Writer。参数 enc 指定字符编码器。参数 minBufferCap 指定内部字节缓冲区的最小容量。

以上方法的参数包括 ReadableByteChannel、WritableByteChannel 和 AsynchronousByteChannel 类型。SocketChannel 实现了 ReadableByteChannel 和 WritableByteChannel 接口，AsynchronousSocketChannel 实现了 AsynchronousByteChannel 接口。本章 4.5 节会介绍 AsynchronousSocketChannel 的用法。

以下程序代码把 SocketChannel 转换成输入流，接下来就能按照输入流的方式来读取数据：

```
SocketChannel socketChannel=serverSocketChannel.accept();
InputStream in=Channels.newInputStream(socketChannel);
```

4.2.10 Socket 选项

从 JDK7 开始，SocketChannel、ServerSocketChannel、AsynchronousSocketChannel、AsynchronousServerSocketChannel 和 DatagramChannel 都实现了新的 NetworkChannel 接口。NetworkChannel 接口的主要作用是设置和读取各种 Socket 选项，例如本书第 2 章和第 3 章介绍的 TCP_NODELAY、SO_LINGER、SO_SNDBUF 和 SO_RCVBUF 等。

NetworkChannel 接口提供了用于设置和读取这些选项的方法：

- <T> T getOption(SocketOption<T> name)：获取特定的 Socket 选项值。
- <T> NetworkChannel setOption(SocketOption<T> name, T value)：设置特定的 Socket 选项。
- Set<SocketOption<?>> supportedOptions()：获取所有支持的 Socket 选项。

SocketOptionl 类是一个泛型类，SocketOption<T>中的“T”代表特定选项的取值类型，可选值包括：Integer、Boolean 和 NetworkInterface。StandardSocketOptions 类提供了以下表示特定选项的常量：

- SocketOption<NetworkInterface> StandardSocketOptions.IP_MULTICAST_IF
- SocketOption<Boolean> StandardSocketOptions. IP_MULTICAST_LOOP
- SocketOption<Integer> StandardSocketOptions.IP_MULTICAST_TTL
- SocketOption<Integer> StandardSocketOptions.IP_TOS
- SocketOption<Boolean> StandardSocketOptions.SO_BROADCAST
- SocketOption<Boolean> StandardSocketOptions.SO_KEEPALIVE
- SocketOption<Integer> StandardSocketOptions.SO_LINGER
- SocketOption<Integer> StandardSocketOptions.SO_RCVBUF
- SocketOption<Boolean> StandardSocketOptions.SO_REUSEADDR
- SocketOption<Boolean> StandardSocketOptions.SO_REUSEPORT
- SocketOption<Integer> StandardSocketOptions.SO_SNDBUF
- SocketOption<Boolean> StandardSocketOptions. TCP_NODELAY

以下程序代码把 SocketChannel 的 SO_LINGER 选项设为 240 秒：

```
SocketChannel socketChannel=SocketChannel.open();
socketChannel.setOption(StandardSocketOptions.SO_LINGER,240);
```

以下例程 4-1 的 OptionPrinter 类会打印出每一种类型的通道支持的所有选项。

例程 4-1 OptionPrinter.java

```
import java.io.*;
import java.net.*;
import java.nio.channels.*;

public class OptionPrinter {

  public static void main(String[] args) throws IOException {
    printOptions(SocketChannel.open());
    printOptions(ServerSocketChannel.open());
    printOptions(AsynchronousSocketChannel.open());
    printOptions(AsynchronousServerSocketChannel.open());
    printOptions(DatagramChannel.open());
```

```
    }

    private static void printOptions(NetworkChannel channel)
                        throws IOException {
      System.out.println(channel.getClass().getSimpleName()
           + " supports:");
      for (SocketOption<?> option : channel.supportedOptions()) {
        try{
          System.out.println(option.name() + ": "
                    + channel.getOption(option));
        }catch(AssertionError e){e.printStackTrace();}
      }
      System.out.println();
      channel.close();
    }
  }
```

运行以上程序，打印结果会显示每种类型通道支持的所有选项以及其默认值：

```
SocketChannelImpl supports:
SO_KEEPALIVE: false
TCP_NODELAY: false
SO_OOBINLINE: false
SO_SNDBUF: 65536
SO_LINGER: -1
SO_RCVBUF: 65536
IP_TOS: 0
SO_REUSEADDR: false

ServerSocketChannelImpl supports:
SO_RCVBUF: 65536
SO_REUSEADDR: false
……

WindowsAsynchronousSocketChannelImpl supports:
SO_KEEPALIVE: false
TCP_NODELAY: false
SO_SNDBUF: 65536
SO_RCVBUF: 65536
SO_REUSEADDR: false

WindowsAsynchronousServerSocketChannelImpl supports:
SO_RCVBUF: 65536
SO_REUSEADDR: false

DatagramChannelImpl supports:
SO_SNDBUF: 65536
IP_MULTICAST_IF: null
SO_RCVBUF: 65536
IP_MULTICAST_LOOP: true
IP_MULTICAST_TTL: 1
IP_TOS: 0
SO_REUSEADDR: false
SO_BROADCAST: false
```

4.3 服务器编程范例

本节介绍如何用 java.nio 包中的类来创建服务器 EchoServer，本节提供了三种实现方式：

- 4.3.1 节的例程 4-2：采用阻塞模式，用线程池中的工作线程处理每个客户连接。
- 4.3.2 节的例程 4-3：采用非阻塞模式，单个线程同时负责接收多个客户连接，以及与多个客户交换数据的任务。
- 4.3.3 节的例程 4-4：由一个线程负责接收多个客户连接，采用阻塞模式；由另一个线程负责与多个客户交换数据，采用非阻塞模式。

4.3.1 创建阻塞的 EchoServer

当 ServerSocketChannel 与 SocketChannel 采用默认的阻塞模式时，为了同时处理多个客户的连接，必须使用多个线程。在例程 4-2 的 block.EchoServer 类中，利用 java.util.concurrent 包中提供的线程池 ExecutorService 来处理与客户的连接。

例程 4-2　EchoServer.java（阻塞模式）

```
package block;
import java.io.*;
……
public class EchoServer {
  private int port=8000;
  private ServerSocketChannel serverSocketChannel = null;
  private ExecutorService executorService;  //线程池
  private static final int POOL_MULTIPLE = 4;  //线程池中工作线程的数目

  public EchoServer() throws IOException {
    //创建一个线程池
    executorService= Executors.newFixedThreadPool(
        Runtime.getRuntime().availableProcessors() * POOL_MULTIPLE);
    //创建一个 ServerSocketChannel 对象
    serverSocketChannel= ServerSocketChannel.open();
    //使得在同一个主机上关闭了服务器程序，紧接着再启动该服务器程序时，
    //可以顺利绑定相同的端口
    serverSocketChannel.socket().setReuseAddress(true);
    //把服务器进程与一个本地端口绑定
    serverSocketChannel.socket().bind(new InetSocketAddress(port));
    System.out.println("服务器启动");
  }

  public void service() {
    while (true) {
      SocketChannel socketChannel=null;
      try {
        socketChannel = serverSocketChannel.accept();
         //处理客户连接
        executorService.execute(new Handler(socketChannel));
      }catch (IOException e) {
         e.printStackTrace();
      }
    }
  }
```

```
  public static void main(String args[])throws IOException {
    new EchoServer().service();
  }
}

class Handler implements Runnable{  //处理客户连接
  private SocketChannel socketChannel;
  public Handler(SocketChannel socketChannel){
    this.socketChannel=socketChannel;
  }
  public void run(){
    handle(socketChannel);
  }

  public void handle(SocketChannel socketChannel){
    try {
       //获得与 socketChannel 关联的 Socket 对象
       Socket socket=socketChannel.socket();
       System.out.println("接收到客户连接，来自: " +
       socket.getInetAddress() + ":" +socket.getPort());

       BufferedReader br =getReader(socket);
       PrintWriter pw = getWriter(socket);

       String msg = null;
       while ((msg = br.readLine()) != null) {
         System.out.println(msg);
         pw.println(echo(msg));
         if (msg.equals("bye"))
           break;
       }
     }catch (IOException e) {
        e.printStackTrace();
     }finally {
        try{
          if(socketChannel!=null)socketChannel.close();
        }catch (IOException e) {e.printStackTrace();}
     }
  }
  private PrintWriter getWriter(Socket socket)throws IOException{
    OutputStream socketOut = socket.getOutputStream();
    return new PrintWriter(socketOut,true);
  }
  private BufferedReader getReader(Socket socket)throws IOException{
    InputStream socketIn = socket.getInputStream();
    return new BufferedReader(new InputStreamReader(socketIn));
  }

  public String echo(String msg) {
    return "echo:" + msg;
  }
}
```

EchoServer 类的构造方法负责创建线程池，启动服务器，把它绑定到一个本地端口。EchoServer 类的 service()方法负责接收客户的连接。每接收到一个客户连接，就把它交给线程池来处理，线程池取出一个空闲的线程，来执行 Handler 对象的 run()方法。Handler 类的 handle()方法负责与客户通信。该方法先获得与 SocketChannel 关联的 Socket 对象，然后从 Socket 对象中得到输入流与输出流，再接收和发送数据。

SocketChannel 实际上也提供了 read(ByteBuffer buffer)，但是通过它来读取一行字符串比较麻烦。以下 readLine()方法就通过 SocketChannel 的 read(ByteBuffer buffer)方法来读取一行字符串，它的作用与 BufferedReader 的 readLine()方法是等价的：

```
public String readLine(SocketChannel socketChannel)throws IOException{
  //存放所有读到的数据，假定一行字符串对应的字节序列的长度小于 1024
  ByteBuffer buffer=ByteBuffer.allocate(1024);
  //存放一次读到的数据，一次只读一个字节
  ByteBuffer tempBuffer=ByteBuffer.allocate(1);
  boolean isLine=false;  //表示是否读到了一行字符串
  boolean isEnd=false;  //表示是否到达了输入流的末尾
  String data=null;
  while(!isLine && !isEnd){
    tempBuffer.clear();  //清空缓冲区
    //在阻塞模式下，只有等读到了 1 个字节或者读到输入流末尾才返回
    //在非阻塞模式下，有可能返回零
    int n=socketChannel.read(tempBuffer);
    if(n==-1){
      isEnd = true;  //到达输入流的末尾
      break;
    }
    if(n==0)
      continue;
    tempBuffer.flip();  //把极限设为位置，把位置设为 0
    buffer.put(tempBuffer); //把 tempBuffer 中的数据拷贝到 buffer 中
    buffer.flip();
    Charset charset=Charset.forName("GBK");
    CharBuffer charBuffer=charset.decode(buffer);  //解码
    data=charBuffer.toString();
    if(data.indexOf("\r\n")!=-1){
      isLine = true; //读到了一行字符串
      data=data.substring(0,data.indexOf("\r\n"));
      break;
    }
    buffer.position(buffer.limit());  //把位置设为极限，为下次读数据作准备
    buffer.limit(buffer.capacity());  //把极限设为容量，为下次读数据作准备
  }//#while
  //如果读入了一行字符串，就返回这行字符串，不包括行结束符“\r\n”
  //如果到达输入流的末尾，就返回 null
  return data;
}
```

从以上程序代码可以看出，用 SocketChannel 来读取一行一行的字符串很麻烦，需要操纵 ByteBuffer 缓冲区，而且需要处理字节流与字符串之间的转换。这比使用输入流和输出流来处理一行一行的字符串麻烦多了。

本书配套源代码包的 sourcecode/chapter04/src/block/EchoServer1.java 演示用上述 readLine()方法来读取一行字符串，本书不再列出完整代码。

4.3.2 创建非阻塞的 EchoServer

在非阻塞模式下，EchoServer 只需要启动一个主线程，就能同时处理三件事：

（1）接收客户的连接。

（2）接收客户发送的数据。

（3）向客户发回响应数据。

EchoServer 委托 Selector 来负责监控接收连接就绪事件、读就绪事件和写就绪事件，如果有特定事件发生，就处理该事件。

EchoServer 类的构造方法负责启动服务器，把它绑定到一个本地端口，代码如下：

```
//创建一个 Selector 对象
selector = Selector.open();
//创建一个 ServerSocketChannel 对象
serverSocketChannel= ServerSocketChannel.open();
//使得在同一个主机上关闭了服务器程序，紧接着再启动该服务器程序时，
//可以顺利绑定到相同的端口
serverSocketChannel.socket().setReuseAddress(true);
//使 ServerSocketChannel 工作于非阻塞模式
serverSocketChannel.configureBlocking(false);
//把服务器进程与一个本地端口绑定
serverSocketChannel.socket().bind(new InetSocketAddress(port));
```

EchoServer 类的 service()方法负责处理本节开头所说的三件事，体现其主要流程的代码如下：

```
public void service() throws IOException{
  serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT );
  while (selector.select() > 0 ){  //第一层 while 循环
     //获得 Selector 的 selected-keys 集合
    Set readyKeys = selector.selectedKeys();
    Iterator it = readyKeys.iterator();
    while (it.hasNext()){  //第二层 while 循环
      SelectionKey key=null;
      try{  //处理 SelectionKey
        key = (SelectionKey) it.next();  //取出一个 SelectionKey
        //把 SelectionKey 从 Selector 的 selected-key 集合中删除
        it.remove();
        if (key.isAcceptable()) {处理接收连接就绪事件; }
        if (key.isReadable()) {处理读就绪事件; }
        if (key.isWritable()) {处理写就绪事件; }
      }catch(IOException e){
        e.printStackTrace();
        try{
          if(key!=null){
            //使这个 SelectionKey 失效,
            //使得 Selector 不再监控这个 SelectionKey 感兴趣的事件
            key.cancel();
            //关闭与这个 SelectionKey 关联的 SocketChannel
            key.channel().close();
          }
        }catch(Exception ex){e.printStackTrace();}
      }
    }//#while
```

```
        }//#while
    }
```

在 service()方法中，首先由 ServerSocketChannel 向 Selector 注册接收连接就绪事件。如果 Selector 监控到该事件发生，就会把相应的 SelectionKey 对象加入到 selected-keys 集合中。service()方法接下来在第一层 while 循环中不断询问 Selector 已经发生的事件，然后依次处理每个事件。

Selector 的 select()方法返回当前相关事件已经发生的 SelectionKey 的个数。如果当前没有任何事件发生，select()方法就会阻塞下去，直到至少有一个事件发生。Selector 的 selectedKeys()方法返回 selected-keys 集合，它存放了相关事件已经发生的 SelectionKey 对象。

service()方法在第二层 while 循环中，从 selected-keys 集合中依次取出每个 SelectionKey 对象，把它从 selected-keys 集合中删除，然后调用 isAcceptable()、isReadable()和 isWritable()方法判断到底是哪种事件发生了，从而作出相应的处理。处理每个 SelectionKey 的代码放在一个 try 语句中，如果出现异常，就会在 catch 语句中使这个 SelectionKey 失效，并且关闭与之关联的 Channel。

1．处理接收连接就绪事件

service()方法中处理接收连接就绪事件的代码如下：

```
    if (key.isAcceptable()) {
      //获得与 SelectionKey 关联的 ServerSocketChannel
      ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
      //获得与客户连接的 SocketChannel
      SocketChannel socketChannel = (SocketChannel) ssc.accept();
      System.out.println("接收到客户连接，来自:" +
                         socketChannel.socket().getInetAddress() +
                         ":" + socketChannel.socket().getPort());
      //把 SocketChannel 设置为非阻塞模式
      socketChannel.configureBlocking(false);
      //创建一个用于存放用户发送来的数据的缓冲区
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      //SocketChannel 向 Selector 注册读就绪事件和写就绪事件
      socketChannel.register(selector,
             SelectionKey.OP_READ |
             SelectionKey.OP_WRITE, buffer);  //关联了一个 buffer 附件
    }
```

如果 SelectionKey 的 isAcceptable()方法返回 true，就意味着这个 SelectionKey 所感兴趣的接收连接就绪事件已经发生了。service()方法首先通过 SelectionKey 的 channel()方法获得与之关联的 ServerSocketChannel 对象，然后调用 ServerSocketChannel 的 accept()方法获得与客户连接的 SocketChannel 对象。这个 SocketChannel 对象默认情况下处于阻塞模式。如果希望它执行非阻塞的 I/O 操作，需要调用它的 configureBlocking(false)方法。SocketChannel 调用 Selector 的 register()方法来注册读就绪事件和写就绪事件，还向 register()方法传递了一个 ByteBuffer 类型的参数，这个 ByteBuffer 将作为附件与新建的 SelectionKey 对象关联。本节稍后会介绍这个 ByteBuffer 的作用。

2．处理读就绪事件

如果 SelectionKey 的 isReadable()方法返回 true，就意味着这个 SelectionKey 所感兴趣的读就绪事件已经发生了。EchoServer 类的 receive()方法负责处理这一事件：

```
    public void receive(SelectionKey key)throws IOException{
```

```
        //获得与 SelectionKey 关联的附件
        ByteBuffer buffer=(ByteBuffer)key.attachment();
        //获得与 SelectionKey 关联的 SocketChannel
        SocketChannel socketChannel=(SocketChannel)key.channel();
        //创建一个 ByteBuffer，用于存放读到的数据
        ByteBuffer readBuff= ByteBuffer.allocate(32);
        socketChannel.read(readBuff);
        readBuff.flip();

        //把 buffer 的极限设为容量
        buffer.limit(buffer.capacity());
        //把 readBuff 中的内容拷贝到 buffer 中，
        //假定 buffer 的容量足够大，不会出现缓冲区溢出异常
        buffer.put(readBuff);
      }
```

在 receive()方法中，先获得与这个 SelectionKey 关联的 ByteBuffer 和 SocketChannel。SocketChannel 每次读到的数据都被添加到这个 ByteBuffer，在程序中，由 buffer 变量引用这个 ByteBuffer 对象。在非阻塞模式下，socketChannel.read(readBuff)方法读到多少数据是不确定的，假定读到的字节为 n 个，那么 0<=n<readBuff 的容量。EchoServer 要求每接收到客户的一行字符串 XXX（也就是字符串以“\r\n”结尾），就返回字符串 echo:XXX。由于无法保证 socketChannel.read(readBuff)方法一次读入一行字符串，因此只好把它每次读入的数据都放到 buffer 中，当这个 buffer 中凑足了一行字符串，再把它发送给客户。

receive()方法的许多代码多涉及对 ByteBuffer 的三个属性（position、limit 和 capacity）的操作，以下图 4-14 演示了以上 readBuff 和 buffer 变量的三个属性的变化过程。假定 SocketChannel 的 read()方法读入了 6 个字节，把它存放在 readBuff 中，并假定 buffer 中原来有 10 个字节，buffer.put(readBuff)方法把 readBuff 中的 6 个字节拷贝到 buffer 中，buffer 中最后有 16 个字节。

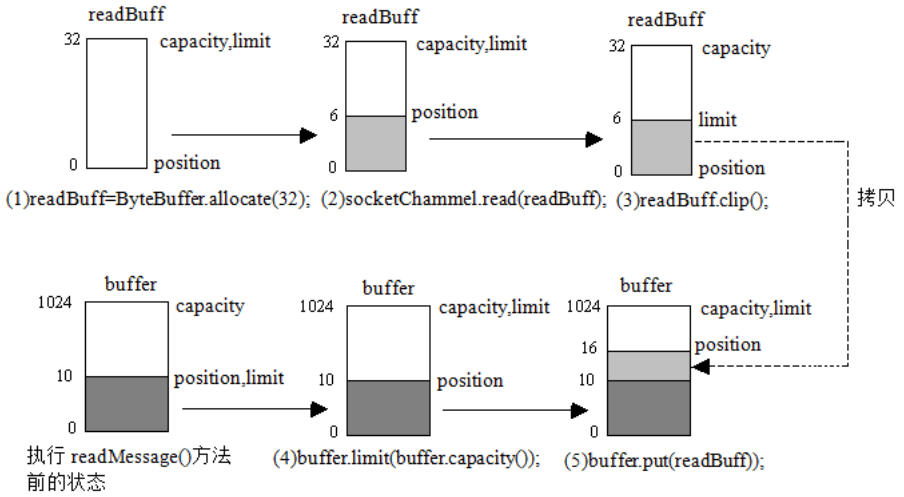


图 4-14　receive()方法操纵 readBuff 和 buffer 的过程

3．处理写就绪事件

如果 SelectionKey 的 isWritable()方法返回 true，就意味着这个 SelectionKey 所感兴趣的写就绪事件已经发生了。EchoServer 类的 send()方法负责处理这一事件：

```
    public void send(SelectionKey key)throws IOException{
      //获得与 SelectionKey 关联的 ByteBuffer
```

```
        ByteBuffer buffer=(ByteBuffer)key.attachment();
        //获得与 SelectionKey 关联的 SocketChannel
        SocketChannel socketChannel=(SocketChannel)key.channel();
        buffer.flip();  //把极限设为位置，把位置设为 0
        //按照 GBK 编码，把 buffer 中的字节转换为字符串
        String data=decode(buffer);
        //如果还没有读到一行数据，就返回
        if(data.indexOf("\r\n")==-1)return;
        //截取一行数据
        String outputData=data.substring(0,data.indexOf("\n")+1);
        System.out.print(outputData);
        //把输出的字符串按照 GBK 编码，转换为字节，把它放在 outputBuffer 中
        ByteBuffer outputBuffer=encode("echo:"+outputData);
        //输出 outputBuffer 中的所有字节
        while(outputBuffer.hasRemaining())
          socketChannel.write(outputBuffer);

        //把 outputData 字符串按照 GBK 编码，转换为字节，把它放在 ByteBuffer 中
        ByteBuffer temp=encode(outputData);
        //把 buffer 的位置设为 temp 的极限
        buffer.position(temp.limit());
        //删除 buffer 中已经处理的数据
        buffer.compact();
        //如果已经输出了字符串“bye\r\n”，就使 SelectionKey 失效，并关闭 SocketChannel
        if(outputData.equals("bye\r\n")){
          key.cancel();
          socketChannel.close();
          System.out.println("关闭与客户的连接");
        }
      }
```

EchoServer 的 receive()方法把读入的数据都放到一个 ByteBuffer 中，send()方法就从这个 ByteBuffer 中取出数据。如果 ByteBuffer 中还没有一行字符串，就什么也不做，直接退出 send()方法；否则，就从 ByteBuffer 中取出一行字符串 XXX，然后向客户发送 echo:XXX。接着，send()方法把 ByteBuffer 中的字符串 XXX 删除。如果 send()方法处理的字符串为“bye\r\n”，就使 SelectionKey 失效，并关闭 SocketChannel，从而断开与客户的连接。

EchoServer 的 receive()方法和 send()方法都操纵同一个 ByteBuffer，receive()方法向 ByteBuffer 中添加数据，而 send()方法从 ByteBuffer 中取出数据。ByteBuffer 的容量为 1024。本程序假设这个 ByteBuffer 的容量足够大，不会发生 receive()方法向 ByteBuffer 中添加数据时，缓冲区溢出的异常。如果要使程序代码更加健壮，就要考虑万一缓冲区中已经填充满了数据，无法再添加更多数据的情况，本书第 5 章的 5.2.2 节的例程 5-3 的 ChannalIO 类提供了解决方案，它实现了容量可增长的缓冲区。

4．编码与解码

在 ByteBuffer 中存放的是字节，它表示字符串的编码。而程序需要把字节转换为字符串，才能进行字符串操作，比如判断里面是否包含“\r\n”，以及截取子字符串。EchoServer 类的实用方法 decode()负责解码，也就是把字节序列转换为字符串：

```
    public String decode(ByteBuffer buffer){  //解码
      CharBuffer charBuffer= charset.decode(buffer);
```

```
    return charBuffer.toString();
  }
```

decode()方法中的 charset 变量是 EchoServer 类的成员变量，它表示 GBK 中文编码，它的定义如下：

```
  private Charset charset=Charset.forName("GBK");
```

在 send()方法中，当通过 SocketChannel 的 write(ByteBuffer buffer)方法发送数据时，write(ByteBuffer buffer)方法不能直接发送字符串，而只能发送 ByteBuffer 中的字节。因此程序需要对字符串进行编码，把它们转换为字节序列，放在 ByteBuffer 中，然后再发送：

```
  ByteBuffer outputBuffer=encode("echo:"+outputData);
  while(outputBuffer.hasRemaining())
    socketChannel.write(outputBuffer);
```

EchoServer 类的实用方法 encode()负责编码，也就是把字符串转换为字节序列：

```
  public ByteBuffer encode(String str){  //编码
    return charset.encode(str);
  }
```

5．在非阻塞模式下确保发送一行数据

在 send()方法的 outputBuffer 中存放了字符串 echo:XXX 的编码。在非阻塞模式下，SocketChannel.write(outputBuffer)方法并不保证一次就把 outputBuffer 中的所有字节发送完，而是奉行能发送多少就发送多少的原则。如果希望把 outputBuffer 中的所有字节发送完，需要采用以下循环：

```
  //hasRemaining()方法判断是否还有未处理的字节
  while(outputBuffer.hasRemaining())
    socketChannel.write(outputBuffer);
```

6．删除 ByteBuffer 中的已处理数据

与 SelectionKey 关联的 ByteBuffer 附件中存放了读操作与写操作的共享数据。receive()方法把读到的数据放入 ByteBuffer，而 send()方法从 ByteBuffer 中一行行地取出数据。当 send()方法从 ByteBuffer 中取出一行字符串 XXX，就要把字符串从 ByteBuffer 中删除。在 send()方法中，outputData 变量就表示取出的一行字符串 XXX，程序先把它编码为字节序列，放在一个名为 temp 的 ByteBuffer 中。接着把 buffer 的位置设为 temp 的极限，然后调用 buffer 的 compact()方法删除代表字符串 XXX 的数据。

```
  ByteBuffer temp=encode(outputData);
  buffer.position(temp.limit());
  buffer.compact();
```

图 4-15 演示了以上代码操纵 buffer 的过程。图中假定 temp 中有 10 个字节，buffer 中本来有 16 个字节，buffer.compact()方法删除缓冲区开头的 10 个字节，最后剩下 6 个字节。

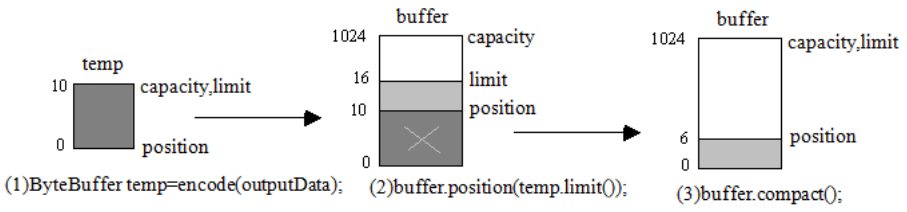


图 4-15　从 buffer 中删除已经处理过的一行字符串 XXX

以下例程 4-3 是 EchoServer 的源程序。

例程 4-3　EchoServer.java(非阻塞模式)

```
package nonblock;
import java.io.*;
……
public class EchoServer{
  private Selector selector = null;
  private ServerSocketChannel serverSocketChannel = null;
  private int port = 8000;
  private Charset charset=Charset.forName("GBK");

  public EchoServer()throws IOException{
    selector = Selector.open();
    serverSocketChannel= ServerSocketChannel.open();
    serverSocketChannel.socket().setReuseAddress(true);
    serverSocketChannel.configureBlocking(false);
    serverSocketChannel.socket().bind(new InetSocketAddress(port));
    System.out.println("服务器启动");
  }
  public void service() throws IOException{
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT );
    while (selector.select() > 0 ){
      Set readyKeys = selector.selectedKeys();
      Iterator it = readyKeys.iterator();
      while (it.hasNext()){
        SelectionKey key=null;
        try{
          key = (SelectionKey) it.next();
          it.remove();

          if (key.isAcceptable()) {
            ServerSocketChannel ssc =
                 (ServerSocketChannel) key.channel();
            SocketChannel socketChannel =(SocketChannel)ssc.accept();
            System.out.println("接收到客户连接，来自:" +
                          socketChannel.socket().getInetAddress() +
                          ":" + socketChannel.socket().getPort());
            socketChannel.configureBlocking(false);
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            socketChannel.register(selector,
                                SelectionKey.OP_READ |
                                SelectionKey.OP_WRITE, buffer);
          }
          if (key.isReadable()) {
            receive(key);
          }
          if (key.isWritable()) {
            send(key);
          }
        }catch(IOException e){
          e.printStackTrace();
          try{
```

```
                if(key!=null){
                    key.cancel();
                    key.channel().close();
                }
            }catch(Exception ex){e.printStackTrace();}
        }
      }//#while
    }//#while
  }

  public void send(SelectionKey key)throws IOException{
    ByteBuffer buffer=(ByteBuffer)key.attachment();
    SocketChannel socketChannel=(SocketChannel)key.channel();
    buffer.flip();  //把极限设为位置，把位置设为0
    String data=decode(buffer);
    if(data.indexOf("\r\n")==-1)return;
    String outputData=data.substring(0,data.indexOf("\n")+1);
    System.out.print(outputData);
    ByteBuffer outputBuffer=encode("echo:"+outputData);
    while(outputBuffer.hasRemaining())  //发送一行字符串
      socketChannel.write(outputBuffer);

    ByteBuffer temp=encode(outputData);
    buffer.position(temp.limit());
    buffer.compact(); //删除已经处理的字符串

    if(outputData.equals("bye\r\n")){
      key.cancel();
      socketChannel.close();
      System.out.println("关闭与客户的连接");
    }
  }

  public void receive(SelectionKey key)throws IOException{
    ByteBuffer buffer=(ByteBuffer)key.attachment();

    SocketChannel socketChannel=(SocketChannel)key.channel();
    ByteBuffer readBuff= ByteBuffer.allocate(32);
    socketChannel.read(readBuff);
    readBuff.flip();

    buffer.limit(buffer.capacity());
    buffer.put(readBuff);  //把读到的数据放到buffer中
  }

  public String decode(ByteBuffer buffer){  //解码
    CharBuffer charBuffer= charset.decode(buffer);
    return charBuffer.toString();
  }
  public ByteBuffer encode(String str){  //编码
    return charset.encode(str);
  }
```

```
  public static void main(String args[])throws Exception{
    EchoServer server = new EchoServer();
    server.service();
  }
}
```

4.3.3 在 EchoServer 中混合用阻塞模式与非阻塞模式

在本章 4.3.2 节的例程 4-3 中，EchoServer 的 ServerSocketChannel 以及 SocketChannel 都被设置为非阻塞模式，这使得接收连接、接收数据和发送数据的操作都采用非阻塞模式，EchoServer 采用一个线程同时完成这些操作。假如有许多客户请求连接，可以把接收客户连接的操作单独由一个线程完成，把接收数据和发送数据的操作由另一个线程完成，这可以提高服务器的并发性能。

负责接收客户连接的线程按照阻塞模式工作，如果收到客户连接，就向 Selector 注册读就绪和写就绪事件，否则进入阻塞状态，直到接收到了客户的连接。负责接收数据和发送数据的线程按照非阻塞模式工作，只有在读就绪或写就绪事件发生时，才执行相应的接收数据和发送数据操作。

例程 4-4 是 EchoServer 类的源程序。其中 receive()、send()、decode()和 encode()方法的代码与 4.3.2 节的例程 4-3 的 EchoServer 类相同，为了节省篇幅，不再重复显示。

例程 4-4　EchoServer.java（混合使用阻塞模式与非阻塞模式）

```
package thread2;
import java.io.*;
……
public class EchoServer{
  private Selector selector = null;
  private ServerSocketChannel serverSocketChannel = null;
  private int port = 8000;
  private Charset charset=Charset.forName("GBK");

  public EchoServer()throws IOException{
    selector = Selector.open();
    serverSocketChannel= ServerSocketChannel.open();
    serverSocketChannel.socket().setReuseAddress(true);
    serverSocketChannel.socket().bind(new InetSocketAddress(port));
    System.out.println("服务器启动");
  }

  public void accept(){
     for(;;){
       try{
         SocketChannel socketChannel =serverSocketChannel.accept();
         System.out.println("接收到客户连接，来自:" +
                            socketChannel.socket().getInetAddress() +
                            ":" + socketChannel.socket().getPort());
         socketChannel.configureBlocking(false);

         ByteBuffer buffer = ByteBuffer.allocate(1024);
         synchronized(gate){
              selector.wakeup();
              socketChannel.register(selector,
                                     SelectionKey.OP_READ |
```

```
                                    SelectionKey.OP_WRITE, buffer);
          }
        }catch(IOException e){e.printStackTrace();}
      }
    }
    private Object gate=new Object();
    public void service() throws IOException{
      for(;;){
        synchronized(gate){}
        int n = selector.select();

        if(n==0)continue;
        Set readyKeys = selector.selectedKeys();
        Iterator it = readyKeys.iterator();
        while (it.hasNext()){
          SelectionKey key=null;
          try{
              key = (SelectionKey) it.next();
              it.remove();
              if (key.isReadable()) {
                  receive(key);
              }
              if (key.isWritable()) {
                  send(key);
              }
          }catch(IOException e){
             e.printStackTrace();
             try{
               if(key!=null){
                 key.cancel();
                 key.channel().close();
               }
             }catch(Exception ex){e.printStackTrace();}
          }
        }//#while
      }//#while
    }

    public void send(SelectionKey key)throws IOException{…}
    public void receive(SelectionKey key)throws IOException{… }

    public String decode(ByteBuffer buffer){…}
    public ByteBuffer encode(String str){… }

    public static void main(String args[])throws Exception{
      final EchoServer server = new EchoServer();
      Thread accept=new Thread(){
         public void run(){
             server.accept();
         }
      };
      accept.start();
      server.service();
```

```
        }
    }
```

以上 EchoServer 类的构造方法与 4.3.2 节的例程 4-3 的 EchoServer 类的构造方法基本相同，唯一的区别是，在本例中， ServerSocketChannel 采用默认的阻塞模式，即没有调用以下方法：

```
serverSocketChannel.configureBlocking(false);
```

EchoServer 类的 accept()方法负责接收客户连接，ServerSocketChannel 的 accept()方法工作于阻塞模式，如果没有客户连接，就会进入阻塞状态，直到接收到了客户连接。接下来调用 socketChannel.configureBlocking(false)方法把 SocketChannel 设为非阻塞模式，然后向 Selector 注册读就绪和写就绪事件。

EchoServer 类的 service()方法负责接收和发送数据，它在一个无限 for 循环中，不断调用 Selector 的 select()方法查寻已经发生的事件，然后作出相应的处理。

在 EchoServer 类的 main()方法中，定义了一个匿名线程（暂且称它为 Accept 线程），它负责执行 EchoServer 的 accept()方法。执行 main()方法的主线程启动了 Accept 线程后，主线程就开始执行 EchoServer 的 service()方法。因此当 EchoServer 启动后，共有两个线程在工作，Accept 线程负责接收客户连接，主线程负责接收和发送数据：

```
public static void main(String args[])throws Exception{
  final EchoServer server = new EchoServer();
  Thread accept=new Thread(){  //定义 Accept 线程
     public void run(){
       server.accept();
     }
  };
  accept.start();  //启动 Accept 线程
  server.service();  //主线程执行 service()方法
}
```

当 Accept 线程开始执行以下方法时：

```
socketChannel.register(selector,
    SelectionKey.OP_READ|SelectionKey.OP_WRITE,buffer);
```

如果主线程正好在执行 selector.select()方法，而且处于阻塞状态，那么 Accept 线程也会进入阻塞状态。两个线程都处于阻塞状态，很有可能导致死锁。导致死锁的具体情形为：Selector 中尚没有任何注册的事件，即 all-keys 集合为空，主线程执行 selector.select()方法时将进入阻塞状态，只有 Accept 线程向 Selector 注册了事件，并且该事件发生后，主线程才会从 selector.select()方法中返回。假如 Selector 中尚没有任何注册的事件，此时 Accept 线程调用 socketChannel.register()方法向 Selector 注册事件，由于主线程正在 selector.select()方法中阻塞，这使得 Accept 线程也在 socketChannel.register()方法中阻塞。Accept 线程无法向 Selector 注册事件，而主线程没有任何事件可以监控，所以这两个线程都将永远阻塞下去。

SelectableChannel 的 register(Selector selector,…)和 Selector 的 select()方法都会操纵 Selector 对象的共享资源 all-keys 集合。SelectableChannel 以及 Selector 的实现对操纵共享资源的代码块进行了同步，从而避免对共享资源的竞争。同步机制使得一个线程执行 SelectableChannel 的 register(Selector selector,…)时，不允许另一个线程同时执行 Selector 的 select()方法，反之亦然。

为了避免死锁，程序必须保证当 Accept 线程正在通过 socketChannel.register()方法向 Selector 注册事件时，不允许主线程正在 selector.select()方法中阻塞。

为了协调 Accept 线程和主线程，EchoServer 类在以下代码前加了同步标记。当 Accept 线程开始执行这段代码时，必须先获得 gate 对象的同步锁，然后进入同步代码块，先执行 Selector 对象的 wakeup()方法，假如此时主线程正好在执行 selector.select()方法，而且处于阻塞状态，那么主线程就会被唤醒，立即退出 selector.select()方法。

```
synchronized(gate){  //Accept 线程执行这个同步代码块
  selector.wakeup();
  socketChannel.register(selector,
                    SelectionKey.OP_READ |
                    SelectionKey.OP_WRITE, buffer);
}
```

主线程被唤醒后，在下一次循环中又会执行 selector.select()方法，为了保证让 Accept 线程先执行完 socketChannel.register()方法，再让主线程执行 selector.select()方法，主线程必须先获得 gate 对象的同步锁：

```
for(;;){
  //一个空的同步代码块，其作用是为了让主线程等待 Accept 线程执行完同步代码块
  synchronized(gate){}  //主线程执行这个同步代码块
  int n = selector.select();
  ……
}
```

假如 Accept 线程还没有执行完同步代码块，就不会释放 gate 对象的同步锁，这使得主线程必须等待片刻，等到 Accept 线程执行完同步代码块，释放了 gate 对象的同步锁，主线程才能恢复运行，再次执行 selector.select()方法。

4.4 客户端编程范例

本节介绍如何用 java.nio 包中的类来创建客户程序 EchoClient，本节提供了两种实现方式：

- 4.4.1 节的例程 4-5：采用阻塞模式，单线程。
- 4.4.2 节的例程 4-6：采用非阻塞模式，单线程。

4.4.1 创建阻塞的 EchoClient

客户程序一般不需要同时建立与服务器的多个连接，因此用一个线程，按照阻塞模式运行就能满足需求。本书第 1 章的 1.5.2 节的例程 1-3 的 EchoClient 类是通过创建 Socket 来建立与服务器的连接的，以下例程 4-5 的 EchoClient 类通过创建 SocketChannel 来与服务器连接，这个 SocketChannel 采用默认的阻塞模式。

例程 4-5　EchoClient.java（阻塞模式）

```
package block;
import java.net.*;
……
public class EchoClient{
  private SocketChannel socketChannel = null;

  public EchoClient()throws IOException{
```

```
            socketChannel = SocketChannel.open();
            InetAddress ia = InetAddress.getLocalHost();
            InetSocketAddress isa = new InetSocketAddress(ia,8000);
            socketChannel.connect(isa);  //连接服务器
            System.out.println("与服务器的连接建立成功");
        }
        public static void main(String args[])throws IOException{
           new EchoClient().talk();
        }
        private PrintWriter getWriter(Socket socket)throws IOException{
           OutputStream socketOut = socket.getOutputStream();
           return new PrintWriter(socketOut,true);
        }
        private BufferedReader getReader(Socket socket)throws IOException{
           InputStream socketIn = socket.getInputStream();
           return new BufferedReader(new InputStreamReader(socketIn));
        }
        public void talk()throws IOException {
           try{
             BufferedReader br=getReader(socketChannel.socket());
             PrintWriter pw=getWriter(socketChannel.socket());
             BufferedReader localReader=
                  new BufferedReader(new InputStreamReader(System.in));
             String msg=null;
             while((msg=localReader.readLine())!=null){
               pw.println(msg);
               System.out.println(br.readLine());

               if(msg.equals("bye"))
                 break;
             }
           }catch(IOException e){
              e.printStackTrace();
           }finally{
              try{socketChannel.close();
             }catch(IOException e){e.printStackTrace();}
           }
         }
       }
```

在 EchoClient 类中，调用 socketChannel.connect(isa)方法连接远程服务器，该方法在阻塞模式下运行时，将等到与远程服务器的连接建立成功才返回。在 EchoClient 类的 talk()方法中，通过 socketChannel.socket()方法获得与 SocketChannel 关联的 Socket 对象，然后从这个 Socket 中获得输出流与输入流，再一行行地发送和接受数据，这种处理方式与第 1 章的 1.5.2 节的例程 1-3 的 EchoClient 类相同。

4.4.2 创建非阻塞的 EchoClient

对于客户与服务器之间的通信，按照它们收发数据的协调程度来区分，可分为同步通信和异步通信。同步通信是指甲方向乙方发送了一批数据后，必须等接收到了乙方的响应数据后，再发送下一批数据。异步通信是指发送数据和接收数据的操作互不干扰，各自独立进行。值得注意的是，通信的两端并不要求都采用同样的通信方式，一方采用同步通信方式时，另一方可以采用异步通信方式。

同步通信要求一个 I/O 操作完成之后，才能完成下一个 I/O 操作，用阻塞模式更容易实现它。异步通信允许发送数据和接收数据的操作各自独立进行，用非阻塞模式更容易实现它。本书第 1 章和第 3 章介绍的 EchoServer 都采用同步通信，每当接收到客户的一行数据，就发回一行响应数据，然后再接收下一行数据。本章 4.3.2 节的例程 4-3，以及 4.3.3 节的例程 4-4 介绍的 EchoServer 都采用异步通信，每次接收数据时，能读到多少数据，就读多少数据。

第 1 章的 1.5.2 节的例程 1-3 和本章 4.4.1 节的例程 4-5 的 EchoClient 类都采用同步通信方式，每次向 EchoServer 发送了一行数据后，必须等接收到了 EchoServer 发回的响应数据，然后再发送下一行数据。以下例程 4-6 的 EchoClient 类利用非阻塞模式来实现异步通信。在 EchoClient 类中，定义了两个 ByteBuffer：sendBuffer 和 receiveBuffer。EchoClient 把用户向控制台输入的数据存放到 sendBuffer 中，并且把 sendBuffer 中的数据发送给远程服务器；EchoClient 把从远程服务器接收到的数据存放在 receiveBuffer 中，并且把 receiveBuffer 中的数据打印到控制台。图 4-16 显示了这两个 Buffer 的作用。

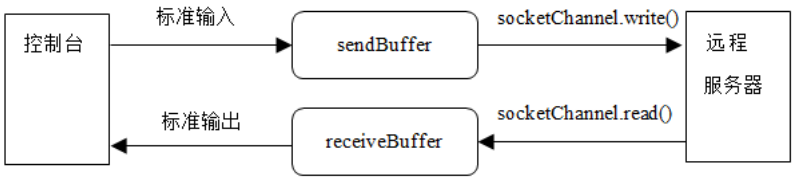


图 4-16　sendBuffer 和 receiveBuffer 的作用

例程 4-6　EchoClient.java（非阻塞模式）

```
package nonblock;
import java.net.*;
……
public class EchoClient{
  private SocketChannel socketChannel = null;
  private ByteBuffer sendBuffer=ByteBuffer.allocate(1024);
  private ByteBuffer receiveBuffer=ByteBuffer.allocate(1024);
  private Charset charset=Charset.forName("GBK");
  private Selector selector;

  public EchoClient()throws IOException{
    socketChannel = SocketChannel.open();
    InetAddress ia = InetAddress.getLocalHost();
    InetSocketAddress isa = new InetSocketAddress(ia,8000);
    socketChannel.connect(isa);  //采用阻塞模式连接服务器
    socketChannel.configureBlocking(false);  //设置为非阻塞模式
    System.out.println("与服务器的连接建立成功");
    selector=Selector.open();
  }
  public static void main(String args[])throws IOException{
    final EchoClient client=new EchoClient();
    Thread receiver=new Thread(){  //创建 Receiver 线程
      public void run(){
        client.receiveFromUser();  //接收用户向控制台输入的数据
      }
    };

    receiver.start();  //启动 Receiver 线程
    client.talk();
```

```
}
/** 接收用户从控制台输入的数据，把它放到 sendBuffer 中 */
public void receiveFromUser(){
  try{
    BufferedReader localReader=new BufferedReader(
        new InputStreamReader(System.in));
    String msg=null;
    while((msg=localReader.readLine())!=null){
      synchronized(sendBuffer){
         sendBuffer.put(encode(msg + "\r\n"));
       }
      if(msg.equals("bye"))
        break;
    }
  }catch(IOException e){
     e.printStackTrace();
  }
}

public void talk()throws IOException { //接收和发送数据
   socketChannel.register(selector,
                    SelectionKey.OP_READ |
                    SelectionKey.OP_WRITE);
   while (selector.select() > 0 ){
     Set readyKeys = selector.selectedKeys();
     Iterator it = readyKeys.iterator();
     while (it.hasNext()){
       SelectionKey key=null;
       try{
          key = (SelectionKey) it.next();
          it.remove();

          if (key.isReadable()) {
             receive(key);
          }
          if (key.isWritable()) {
             send(key);
          }
       }catch(IOException e){
         e.printStackTrace();
         try{
            if(key!=null){
               key.cancel();
               key.channel().close();
            }
         }catch(Exception ex){e.printStackTrace();}
       }
    }//#while
  }//#while
}

public void send(SelectionKey key)throws IOException{
  //发送 sendBuffer 中的数据
```

```
            SocketChannel socketChannel=(SocketChannel)key.channel();
            synchronized(sendBuffer){
                sendBuffer.flip(); //把极限设为位置，把位置设为零
                socketChannel.write(sendBuffer);  //发送数据
                sendBuffer.compact();  //删除已经发送的数据
             }
          }

          public void receive(SelectionKey key)throws IOException{
            //接收 EchoServer 发送的数据，把它放到 receiveBuffer 中
            //如果 receiveBuffer 中有一行数据，就打印这行数据，
            //然后把它从 receiveBuffer 中删除
            SocketChannel socketChannel=(SocketChannel)key.channel();
            socketChannel.read(receiveBuffer);
            receiveBuffer.flip();
            String receiveData=decode(receiveBuffer);

            if(receiveData.indexOf("\n")==-1)return;

            String outputData=receiveData.substring(0,
                      receiveData.indexOf("\n")+1);
            System.out.print(outputData);
            if(outputData.equals("echo:bye\r\n")){
                key.cancel();
                socketChannel.close();
                System.out.println("关闭与服务器的连接");
                selector.close();
                System.exit(0);  //结束程序
            }

            ByteBuffer temp=encode(outputData);
            receiveBuffer.position(temp.limit());
            receiveBuffer.compact();  //删除已经打印的数据
          }

          public String decode(ByteBuffer buffer){  //解码
            CharBuffer charBuffer= charset.decode(buffer);
            return charBuffer.toString();
          }
          public ByteBuffer encode(String str){  //编码
            return charset.encode(str);
          }
        }
```

在 EchoClient 类的构造方法中，创建了 SocketChannel 对象后，该 SocketChannel 对象采用默认的阻塞模式，随后调用 socketChannel.connect(isa)方法，该方法将按照阻塞模式来与远程服务器 EchoServer 连接，只有当连接建立成功，该 connect()方法才会返回。接下来程序再调用 socketChannel.configureBlocking(false)方法把 SocketChannel 设为非阻塞模式，这使得接下来通过 SocketChannel 来接收和发送数据都会采用非阻塞模式。

```
        socketChannel = SocketChannel.open();
        ……
        socketChannel.connect(isa);
```

```
socketChannel.configureBlocking(false);
```

EchoClient 类共使用了两个线程：主线程和 Receiver 线程。主线程主要负责接收和发送数据，这些操作由 talk()方法实现。Receiver 线程负责读取用户向控制台输入的数据，该操作由 receiveFromUser()方法实现。

```
public static void main(String args[])throws IOException{
  final EchoClient client=new EchoClient();
  Thread receiver=new Thread(){  //创建 receiver 线程
    public void run(){
      client.receiveFromUser();  //读取用户向控制台输入的数据
    }
  };

  receiver.start();
  client.talk(); //接收和发送数据
}
```

receiveFromUser()方法读取用户输入的字符串，把它存放到 sendBuffer 中。如果用户输入字符串“bye”，就退出 receiveFromUser()方法，这使得执行该方法的 Receiver 线程结束运行。由于主线程在执行 send()方法时，也会操纵 sendBuffer，为了避免两个线程对共享资源 sendBuffer 的竞争，receiveFromUser()方法对操纵 sendBuffer 的代码进行了同步。

```
BufferedReader localReader=new BufferedReader(
                 new InputStreamReader(System.in));
String msg=null;
while((msg=localReader.readLine())!=null){
  synchronized(sendBuffer){
     sendBuffer.put(encode(msg + "\r\n"));
  }
  if(msg.equals("bye"))
    break;
}
```

talk()方法向 Selector 注册读就绪和写就绪事件，然后轮询已经发生的事件，并做出相应的处理。如果发生读就绪事件，就执行 receive()方法，如果发生写就绪事件，就执行 send()方法。

receive()方法接收 EchoServer 发回的响应数据，把它们存放在 receiveBuffer 中。如果 receiveBuffer 中已经满一行数据，就向控制台打印这一行数据，并且把这行数据从 receiveBuffer 中删除。如果打印的字符串为“echo:bye\r\n”，就关闭 SocketChannel，并且结束程序。

send()方法把 sendBuffer 中的数据发送给 EchoServer，然后删除已经发送的数据。由于 Receiver 线程以及执行 send()方法的主线程都会操纵共享资源 sendBuffer，为了避免对共享资源的竞争，对 send()方法中操纵 sendBuffer 的代码进行了同步。

4.5 异步通道和异步运算结果

从 JDK7 开始，引入了表示异步通道的 AsynchronousSocketChannel 类和 AsynchronousServerSocketChannel 类，这两个类的作用与 SocketChannel 类和 ServerSocketChannel 相似，区别在于异步通道的一些方法总是采用非阻塞工作模式，并且它们的非阻塞方法会立即返回一个 Future 对象，用来存放方法的异步运算结果。

AsynchronousSocketChannel 类有以下非阻塞方法：

- Future<Void> connect(SocketAddress remote)：连接远程主机。
- Future<Integer> read(ByteBuffer dst)：从通道中读入数据，存放到 ByteBuffer 中。Future 对象中包含了实际从通道中读到的字节数。
- Future<Integer> write(ByteBuffer src)：把 ByteBuffer 中的数据写入到通道中。Future 对象中包含了实际写入通道的字节数。

AsynchronousServerSocketChannel 类有以下非阻塞方法：

- Future<AsynchronousSocketChannel> accept()：接受客户连接请求。Future 对象中包含了连接建立成功后创建的 AsynchronousSocketChannel 对象。

使用异步通道，可以使程序并行执行多个异步操作，例如：

```
SocketAddress socketAddress=……;
AsynchronousSocketChannel client= AsynchronousSocketChannel.open();
//请求建立连接
Future<Void > connected=client.connect(socketAddress);
ByteBuffer byteBuffer=ByteBuffer.allocate(128);

//执行其他操作
//……

//等待连接完成
connected.get();
//读取数据
Future<Integer> future=client.read(byteBuffer);

//执行其他操作
//……

//等待从通道读取数据完成
future.get();

byteBuffer.flip();
WritableByteChannel out=Channels.newChannel(System.out);
out.write(byteBuffer);
```

以下例程 4-7 的 PingClient 类演示了异步通道的用法。它不断接收用户输入的域名（即网络上主机的名字），然后与这个主机上的 80 端口建立连接，最后打印建立连接所花费的时间。如果程序无法连接到指定的主机，就打印相关错误信息。如果用户输入“bye”，就结束程序。以下是运行 PingClient 类时用户输入的信息以及程序输出的信息。其中采用黑色字体的行表示用户向控制台输入的信息，采用浅色字体的行表示程序的输出结果：

```
C:\chapter04\classes>java nonblock.PingClient
www.abc888.com
www.javathinker.net
ping www.abc888.com的结果 : 连接失败
ping www.javathinker.net的结果 : 20ms
bye
```

从以上打印结果可以看出，PingClient 连接远程主机 www.javathinker.net 用了 20ms，而连接 www.abc888.com 主机失败。从打印结果还可以看出，PingClient 采用异步通信方式，

当用户输入一个主机名后，不必等到程序输出对这个主机名的处理结果，就可以继续输入下一个主机名。对每个主机名的处理结果要等到连接已经成功或者失败后才打印出来。

例程 4-7　PingClient.java

```
package nonblock;
import java.net.*;
……
class PingResult {  //表示连接一个主机的结果
  InetSocketAddress address;
  long connectStart;  //开始连接时的时间
  long connectFinish = 0;  //连接成功时的时间
  String failure;
  Future<Void> connectResult;  //连接操作的异步运算结果
  AsynchronousSocketChannel socketChannel;
  String host;
  final String ERROR="连接失败";

  PingResult(String host) {
      try {
          this.host=host;
          address =
              new InetSocketAddress(InetAddress.getByName(host),80);
      } catch (IOException x) {
          failure = ERROR;
      }
  }

  public void print() {  //打印连接一个主机的执行结果
      String result;
      if (connectFinish != 0)
          result = Long.toString(connectFinish - connectStart) + "ms";
      else if (failure != null)
          result = failure;
      else
          result = "Timed out";
      System.out.println("ping "+ host+"的结果" + " : " + result);
  }
}

public class PingClient{
  //存放所有 PingResult 结果的队列
  private LinkedList<PingResult> pingResults=
             new LinkedList<PingResult>();
  boolean shutdown=false;
  ExecutorService executorService;

  public PingClient()throws IOException{
    executorService= Executors.newFixedThreadPool(4);
    executorService.execute(new Printer());
    receivePingAddress();
  }

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

```
    new PingClient();
  }

  /** 接收用户输入的主机地址，由线程池执行 PingHandler 任务 */
  public void receivePingAddress(){
    try{
      BufferedReader localReader=new BufferedReader(
                  new InputStreamReader(System.in));
      String msg=null;
      //接收用户输入的主机地址
      while((msg=localReader.readLine())!=null){
        if(msg.equals("bye")){
          shutdown=true;
          executorService.shutdown();
          break;
        }
        executorService.execute(new PingHandler(msg));
      }
    }catch(IOException e){ }
  }

  /** 尝试连接特定主机，并且把运算结果加入到 PingResults 结果队列中 */
  public void addPingResult(PingResult pingResult) {
     AsynchronousSocketChannel socketChannel = null;
     try {
       socketChannel = AsynchronousSocketChannel.open();

       pingResult.socketChannel=socketChannel;
       pingResult.connectStart = System.currentTimeMillis();

       synchronized (pingResults) {
         //向 pingResults 队列中加入一个 PingResult 对象
         pingResults.add(pingResult);
         pingResults.notify();
       }

       Future<Void> connectResult=
           socketChannel.connect(pingResult.address);
       pingResult.connectResult = connectResult;
    }catch (Exception x) {
      if (socketChannel != null) {
        try {socketChannel.close();} catch (IOException e) {}
      }
      pingResult.failure = pingResult.ERROR;
    }
  }

  /** 打印 PingResults 结果队列中已经执行完毕的任务的结果 */
  public void printPingResults() {
    PingResult pingResult = null;
    while(!shutdown ){
      synchronized (pingResults) {
        while (!shutdown && pingResults.size() == 0 ){
```

```
            try{
              pingResults.wait(100);
            }catch(InterruptedException e){e.printStackTrace();}
          }

          if(shutdown  && pingResults.size() == 0 )break;
          pingResult=pingResults.getFirst();

          try{
            if(pingResult.connectResult!=null)
              pingResult.connectResult.get(500,TimeUnit.MILLISECONDS);
          }catch(Exception e){
              pingResult.failure= pingResult.ERROR;
          }

          if(pingResult.connectResult!=null
             && pingResult.connectResult.isDone()){

            pingResult.connectFinish = System.currentTimeMillis();
          }

          if(pingResult.connectResult!=null
             && pingResult.connectResult.isDone()
             || pingResult.failure!=null){

             pingResult.print();
             pingResults.removeFirst();
             try {
               pingResult.socketChannel.close();
             } catch (IOException e) { }
           }
        }
      }
    }

    /** 尝试连接特定主机，生成一个 PingResult 对象，
       把它加入到 PingResults 结果队列中 */
    public class PingHandler implements Runnable{
      String msg;
      public PingHandler(String msg){
          this.msg=msg;
      }
      public void run(){
          if(!msg.equals("bye")){
            PingResult pingResult=new PingResult(msg);
            addPingResult(pingResult);
          }
      }
    }

    /** 打印 PingResults 结果队列中已经执行完毕的任务的结果 */
    public class Printer implements Runnable{
      public void run(){
```

```
            printPingResults();
          }
        }
      }
```

以上 PingResult 类表示连接一个主机的执行结果。PingClient 类的 PingResults 队列存放所有的 PingResult 对象。

PingClient 类还定义了两个表示特定任务的内部类：

- PingHandler 任务类：负责通过异步通道去尝试连接客户端输入的主机地址，并且创建一个 PingResult 对象，它包含了连接操作的异步运算结果。再把 PingResult 对象加入到 PingResults 结果队列中。
- Printer 任务类：负责打印 PingResults 结果队列中已经执行完毕的任务结果。打印完毕的 PingResult 对象会从 PingResults 队列中删除。

PingClient 类的 main 主线程完成以下操作：

- 创建线程池。
- 向线程池提交 Printer 任务。
- 不断读取客户端输入的主机地址，向线程池提交 PingHandler 任务。如果客户端输入“bye”，就结束程序。

PingClient 类的线程池完成以下操作：

- 执行 Printer 任务。
- 执行 PingHander 任务。

4.6 在 GUI 中用 SwingWorker 实现异步交互

对于 4.4.1 节创建的 EchoClient 类，也可以改成用图形用户界面（简称 GUI，Graphical User Interface）来接受用户输入的字符串。以下例程 4-8 的 gui.EchoClient 类提供了图形用户界面，用户在文本框 JTextField 中输入字符串，gui.EchoClient 类会在一个 JTextPane 文本面板中显示服务器端返回的响应结果。

例程 4-8　gui.EchoClient 类

```
package gui;
import javax.swing.text.*;
……
public class EchoClient extends JFrame implements ActionListener{
  private JLabel clientLabel=new JLabel("客户端输入内容：");
  private JTextField clientTextField=new JTextField();
  private JLabel serverLabel=new JLabel("服务器返回的响应结果");
  private JTextPane serverTextPane=new JTextPane();

  private SocketChannel socketChannel = null;

  public EchoClient(String title){
    super(title);

    clientTextField.addActionListener(this);
    serverTextPane.setEditable(false);
```

```
  JScrollPane serverScrollPane=new JScrollPane(serverTextPane);

  JPanel clientPanel=new JPanel();
  clientPanel.setLayout(new BorderLayout());
  clientPanel.add(clientLabel,BorderLayout.NORTH);
  clientPanel.add(clientTextField,BorderLayout.SOUTH);

  JPanel serverPanel=new JPanel();
  serverPanel.setLayout(new BorderLayout());
  serverPanel.add(serverLabel,BorderLayout.NORTH);
  serverPanel.add(serverScrollPane,BorderLayout.CENTER);

  Container container=getContentPane();
  container.add(clientPanel,BorderLayout.NORTH);
  container.add(serverPanel,BorderLayout.CENTER);

  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(500,300);
  setVisible(true);

  connect(); //连接服务器
}

public void connect(){
  try{  //连接服务器
    socketChannel = SocketChannel.open();
    InetAddress ia = InetAddress.getLocalHost();
    InetSocketAddress isa = new InetSocketAddress(ia,8000);
    socketChannel.connect(isa);
    setServerTextPane("与服务器的连接建立成功");
  }catch(Exception e){
     setServerTextPane("与服务器连接失败");
  }
}

public void setServerTextPane(String text){
  serverTextPane.setText(serverTextPane.getText()+"\r\n"+text);
}

public static void main(String[] args){
  //向 EDT 线程提交创建 EchoClient 任务
  EventQueue.invokeLater(new Runnable() {
    public void run() {
      try {
         EchoClient echoClient=new EchoClient("EchoClient");
       } catch (Exception e) {
         e.printStackTrace();
      }
    }
  });
}

private PrintWriter getWriter(Socket socket)throws IOException{……}
```

```
    private BufferedReader getReader(Socket socket)
            throws IOException{……}

    public String talk()throws IOException {
      BufferedReader br=getReader(socketChannel.socket());
      PrintWriter pw=getWriter(socketChannel.socket());
     //获得文本框输入的消息
      String msg=clientTextField.getText();
      pw.println(msg);  //向服务器端发送消息
      return br.readLine();  //获得服务器端返回的响应结果
    }

    /** 处理用户在文本框中回车的事件 */
    public void actionPerformed(ActionEvent evt){
      try{
        setServerTextPane(talk());
      }catch(Exception e){setServerTextPane(e.getMessage());}
    }
  }
```

以上程序创建的用户界面如图 4-1 所示。当用户在 JTextField 文本框中输入一些字符，然后回车，就会触发 ActionEvent 事件。EchoClient 对该事件的处理过程如下：

（1）获取用户在文本框输入的消息。

（2）向服务器端发送消息。

（3）接收服务器端返回的响应结果。

（4）在文本面板上显示响应结果。

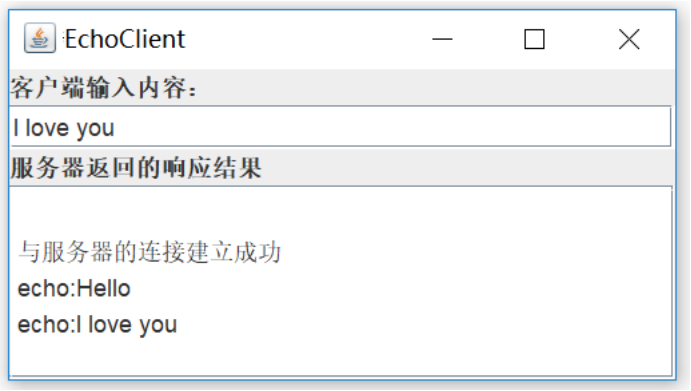


图 4-1　gui.EchoClient 类创建的用户界面

当 EchoClient 类在处理 ActionEvent 事件时，如果在接收服务器端的返回响应结果时，长时间进入阻塞状态，会出现什么情况呢？下面通过实验来演示实际产生的运行效果。步骤如下。

（1）修改 4.3.1 节的 EchoServer，使它在向客户端发送响应结果之前，故意睡眠若干秒，使得客户端在接收响应结果时进入阻塞状态。修改后的服务类为 gui.EchoServer。主要的修改代码如下：

```
    while ((msg = br.readLine()) != null) {
      System.out.println(msg);

      //服务器端在发送响应数据之前故意睡眠一段时间，
      //使得客户端在接收数据时进入阻塞
```

```
        try{
          Thread.sleep(5000);
        }catch(Exception e){}

        pw.println(echo(msg));
        if (msg.equals("bye"))
          break;
      }
```

（2）先运行 gui.EchoServer 类，再运行 gui.EchoClient 类。在 EchoClient 类的用户界面的文本框中输入一些字符再回车，不断快速重复这一操作。会发现用户界面常常被“卡”住，即来不及响应用户的操作。

为什么用户界面会被“卡”住呢，下面来分析原因。当 gui.EchoClient 类运行时，共有两个线程在工作：

- main 主线程：负责执行 main()方法，把创建 EchoClient 对象的任务提交給 EDT 线程。
- EDT 线程（Event Dispatch Thread，事件分派线程）：这是 Java 虚拟机为图形用户界面自动提供的线程。该线程的工作包括：（1）负责绘制和刷新图形用户界面；（2）监听和处理由图形界面触发的各种事件；（3）执行其他线程提交的任务，例如在本例中，main 主线程向 EDT 线程提交了创建 EchoClient 对象的任务。

gui.EchoClient 类开始运行时，main 主线程执行完 main()方法后就结束生命周期，接下来只有 EDT 线程以单线程的方式运行。它要全盘兼顾对图形用界面的绘制和刷新，监听和处理由图形界面触发的各种事件，势必应接不暇，顾此失彼。当 EDT 线程在处理本范例中的 ActionEvent 事件时，它因为等待服务器端返回响应结果而进入阻塞状态，就没办法立即响应用户在图形用户界面上继续输入字符并回车的操作，这样就产生界面很“卡”的情况。

如何解决这一问题呢？这就需要想办法由其他线程来分担 EDT 线程的一些工作，让 EDT 线程主要负责对图形用界面的绘制和刷新，监听由图形界面触发的各种事件，而在处事件时，并不用亲力亲为，只要把具体的事件处理任务交給其他线程去执行就行了。EDT 线程转交完事件处理任务后，就能立刻继续负责对用户界面的维护和事件监听。

接下来的一个问题是，EDT 线程把具体的事件处理任务交給哪个线程呢？一种办法是交給开发人员自定义的线程。但是，图形用户界面中的组件都是非线程安全的。所谓非线程安全，是指假如有几个线程（包括 EDT 线程和开发人员自定义的线程）并发对界面中的组件进行外观或组件中文本的修改，可能会导致界面无法正常显示。所以，还有一种更可靠安全的办法是采用 Java Swing API 提供的 SwingWorker 类，它和 EDT 线程都是由 JDK 本身提供，它们内部的配合更加默契。

4.6.1 SwingWorker 类的用法

SwingWorker 类的定义如下：

```
public abstract class SwingWorker<T,V> extends Object
                                    implements RunnableFuture<T>
```

SwingWorker 类实现了 RunnableFuture 接口，而 RunnableFuture 接口又继承了 Future 接口和 Runnable 接口。所以 SwingWorker 类支持异步运算。SwingWorker<T,V>有两个参数：

- “T”：表示异步运算的最终结果。SwingWorker 类的 doInBackground()和 get()方法返回最终结果。

- “V”表示异步运算的中间运算数据。SwingWorker 类的 publish()方法产生中间运算数据，process 方法()处理中间运算数据。

SwingWorker 类主要包含以下方法：

- protected abstract T doInBackground()

该方法中包含了主要的后台处理任务，由 SwingWorker 工作线程来执行。执行完毕，会返回最终运算结果。如果执行中遇到异常，可以抛出该异常。

- protected void publish(V chunks)

参数 chunks 表示运算中的中间运算数据，该方法通常由 SwingWorker 工作线程来执行。在 doInBackground()方法中可以调用此方法来发送一些中间运算数据。publish()方法会通知 EDT 线程执行 process()方法。

- protected void process(List<V> chunks)

该方法由 EDT 线程执行。当 SwingWorker 工作线程执行一次 publish()方法，就会导致 EDT 线程执行一次 process()方法，process()方法会处理由 publish()方法发送的中间运算数据。process()方法的 chunks 参数就表示中间运算数据。

- protected void done()

该方法由 EDT 线程执行。当 SwingWorker 工作线程执行完 doInBackground()方法后，EDT 线程会自动执行 done()方法。由此可见，SwingWorker 类与 EDT 线程内部配合非常默契。这种配合默契程度是开发人员自定义的线程很难做到的。

- public void execute()

当其他线程（例如 main 主线程或 EDT 线程）调用了一个 SwingWorker 对象的 execute()方法，该方法就会把 SwingWorker 任务提交給 SwingWorker 工作线程池。SwingWorker 工作线程池会委派一个处于空闲状态的 SwingWorker 工作线程来执行 SwingWorker 任务，确切地说，就是执行 SwingWorker 对象的 doInBackground()方法。

- public T get()

由其他线程来调用，获得 SwingWorker 任务的最终运算结果。get()方法会等待 doInBackground()方法计算完成，返回 doInBackground()方法产生的最终运算结果。

- public boolean isDone()

由其他线程来调用，判断是否完成了整个 SwingWorker 任务，如果任务完成，就返回 true 。

- public boolean cancel(boolean mayInterruptIfRunning)

由其他线程来调用，取消 SwingWorker 任务。如果取消成功，就返回 true 。假如任务还没开始执行，那么 cancel()方法使得该任务永远不会执行。如果任务正在执行中，并且参数 mayInterruptIfRunning 为 true，就会取消这一执行中的任务。

4.6.2 用 SwingWorker 类来展示进度条

在 Swing API 中，JProgressBar 表示进度条，它能直观地告诉用户某个任务执行的进度。JProgressBar 的 setValue(int value)方法依据参数 value 来显示进度条的长度。当程序不断调用 setValue(int value)方法，并且每次提供不同的 value 参数值，就会使得进度条不断动态更新。

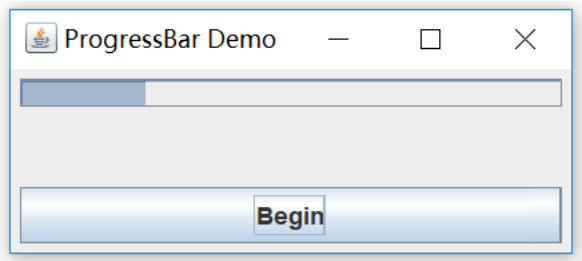
在以下例程 4-9 的 gui.BarDemo 类中，利用 SwingWorker 与 EDT 线程的默契合作，就能不断更新进度条，展示执行一个写文件任务的进度。

例程 4-9　BarDemo.java

```
package gui;
import java.awt.BorderLayout;
……
public class BarDemo extends JFrame {
  private JPanel contentPane;
  private JProgressBar progressBar;

  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          BarDemo bar = new BarDemo("ProgressBar Demo");
        } catch (Exception e) { e.printStackTrace();}
      }
    });
  }

  public BarDemo(String title) {
    super(title);
    ……
    progressBar = new JProgressBar(0, 100);
    contentPane.add(progressBar, BorderLayout.NORTH);

    JButton btnBegin = new JButton("Begin");
    btnBegin.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
           new ProgressBarHandler().execute();
       }
    });
    contentPane.add(btnBegin, BorderLayout.SOUTH);
    setVisible(true);
  }

  class ProgressBarHandler extends SwingWorker<Void, Integer> {

    /** SwingWorker 工作线程执行,后台任务,每隔一秒，发送一个中间运算数据 */
    protected Void doInBackground() throws Exception {
       FileOutputStream out=new FileOutputStream("data.txt");
       PrintWriter pw=new PrintWriter(out,true);
       // 模拟一个很耗时的任务，每次睡 1 秒，再向文件中写入一行文本。
       for (int i = 0; i < 100; i++) {
          Thread.sleep(1000);
          pw.println("data"+i);
          publish(i);//将当前进度信息加入 chunks 中
       }
       pw.close();
       return null;
    }

    /** EDT 线程执行，依据当前的中间运算数据，更新进度条的信息 */
    protected void process(List<Integer> chunks) {
      progressBar.setValue(chunks.get(chunks.size() - 1));
```

```
        }

        /** EDT 线程执行，显示任务完成的消息框 */
        protected void done() {
          JOptionPane.showMessageDialog(null, "任务完成！");
        }
      }
    }
```

BarDemo 类创建的图形用户界面如图 4-1 所示。



BarDemo 类的图形用户界面

当用户在界面上按下“Begin”按钮，所触发的 ActionEvent 事件由 BarDemo 构造方法中定义的匿名 ActionListener 来处理：

```
    btnBegin.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new ProgressBarHandler().execute();
      }
    });
```

以上 actionPerformed()方法由 EDT 线程来执行。该方法创建一个 ProgressBarHandler 对象，再调用它的 execute()方法。execute()方法向 SwingWorker 工作线程池提交了一个 ProgressBarHandler 任务。SwingWorker 工作线程池会委派特定的 SwingWorker 工作线程来执行 ProgressBarHandler 对象的 doInBackground()方法。

ProgressBarHandler 类是 BarDemo 类的内部类。ProgressBarHandler 类继承了 SwingWorker 类。ProgressBarHandler 类在 doInBackground()方法中执行一个耗时的操作，向文件中不断写入数据，并且定期通过 publish()方法发送中间运算数据。

ProgressBarHandler 类的 process()方法由 EDT 线程执行，根据当前的中间运算数据来绘制进度条。

当 SwingWorker 工作线程执行完 doInBackground()方法，EDT 线程会调用 done()方法，该方法向用户显示一个“任务完成”的消息框。

提示：如果用户不断在 BarDemo 的用户界面上按下“Begin”按钮，就会导致若干 SwingWorker 工作线程并发执行 ProgressBarHandler 任务，并发修改进度条，这会导致进度条无法正常显示。解决这一问题的办法是当用户按下“Begin”按钮，就调用按钮的 setEnabled(false)方法，使得该按钮暂时失效，直到 ProgressBarHandler 任务完成，才使按钮重新有效。并且提供一个“Cancel”按钮方法，允许用户中途取消 ProgressBarHandler 任务。在程序中，调用 SwingWorker 类的 cancel()方法可以取消任务。。

4.6.3 用 SwingWorker 类实现异步的 AsynEchoClient

以下例程 4-10 的 gui.AsynEchoClient 类创建的用户界面和 4.6 节的例程 4-1 的 EchoClient 类相同。区别在于，在 AsynEchoClient 类中，定义了继承 SwingWorker 类的 ActionEventHandler 类，它在 doInBackGround()方法中负责接收服务器端的响应结果。

例程 4-10 AsynEchoClient.java

```
package gui;
import javax.swing.text.*;
……
public class AsynEchoClient extends JFrame implements ActionListener{
  ……
  /** 处理用户在 clientTextField 文本框中回车的事件 */
  public void actionPerformed(ActionEvent evt){
     new ActionEventHandler().execute();
  }

  class ActionEventHandler extends SwingWorker<String, Void> {

    /**  后台任务（SwingWorker 工作线程执行）：接受服务器端的响应结果 */
    protected String doInBackground() throws Exception {
      return talk();
    }

    /** 前台任务（EDT 线程执行）：显示服务器端的响应结果 */
    protected void done(){
       try{
         String result=get();
         setServerTextPane(result);
       }catch(Exception e){setServerTextPane(e.getMessage());}
    }
  }
}
```

当 SwingWorker 工作线程执行完 ActionEventHandler 对象的 doInBackground()方法，EDT 线程就会执行 ActionEventHandler 对象的 done()方法，该方法会在文本面板中显示服务器端发送的响应结果。

先运行 gui.EchoServer 类，再运行 gui.AsynEchoClient 类，当用户在 AsynEchoClient 类的用户界面的文本框中输入一些字符再回车，接着不断快速重复这一操作，不会出现界面很“卡”的情况。这是因为每次处理 ActionEvent 事件的具体任务都是由 SwingWorker 工作线程池中的工作线程去执行，EDT 线程有更多的时间来响应用户与图形界面之间的交互操作。

4.7 小结

本章介绍了用 ServerSocketChannel 与 SocketChannel 来创建服务器和客户程序的方法。ServerSocketChannel 与 SocketChannel 既可以工作于阻塞模式，也可以工作于非阻塞模式，默认情况下，它们都工作于阻塞模式，可以调用 configureBlocking()方法来重新设置模式。

AsynchronousSocketChannel 类和 AsynchronousServerSocketChannel 类表示异步通道。AsynchronousSocketChannel 类的请求建立连接方法和读写方法，以及 AsynchronousServerSocketChannel 类的接受连接方法总是采用非阻塞工作模式，并且它们的

非阻塞方法会立即返回一个 Future 对象，用来存放方法的执行结果。

第 1 章、第 3 章以及本章对 EchoServer 共提供了六种实现方式，参见表 4-1。

表 4-1　EchoServer 的六种实现方式

编号	例程	实现方式	特点
方案一	第 1 章的 1.5.1 节的例程 1-2	用 ServerSocket 与 Socket 实现，阻塞模式，单线程，同步通信	编程简单，不能同时处理多个连接。
方案二	第 3 章的 3.6.1 节的例程 3-5	用 ServerSocket 与 Socket 实现，阻塞模式，对于每个客户连接分配一个线程，同步通信。	编程简单，能同时处理多个连接。但是当连接数目非常多时，导致线程数目也非常多，系统开销较大。因此系统所能同时处理的最大连接数必须受到限制。
方案三	第 3 章的 3.6.2 节的例程 3-8	用 ServerSocket 与 Socket 实现，阻塞模式，用线程池来处理每个客户的连接，同步通信。	与方案二相比，可以减少线程的数目。不过线程池本身的实现必须非常健壮。
方案四	第 4 章的 4.3.1 节的例程 4-2	用 ServerSocketChannel 与 SocketChannel 实现，阻塞模式，用线程池来处理每个客户的连接，同步通信。	与方案三的优缺点相同。区别在于用 ServerSocketChannel 取代 ServerSocket，用 SocketChannel 取代 Socket。
方案五	第 4 章的 4.3.2 节的例程 4-3	用 ServerSocketChannel 与 SocketChannel 实现，非阻塞模式，单个线程，异步通信。	能同时处理多个连接，与方案二和方案三相比，使用的线程较少，系统开销小，因此具有更好的并发性能，可以同时处理更多的并发连接。缺点是编程难度高，要求熟练掌握缓冲区 ByteBuffer 以及字符编码转换的用法。
方案六	第 4 章的 4.3.3 节的例程 4-4	用 ServerSocketChannel 与 SocketChannel 实现。ServerSocketChannel 采用阻塞模式，SocketChannel 采用 非阻塞模式。两个线程，异步通信。一个线程负责接收客户连接，一个线程负责接收和发送数据。	与方案五相比，又增加了一个线程，这可以进一步提高并发性能，提高响应客户的速度。缺点是线程数目越多，编程难度也会增大，必须处理好线程之间的同步与协调，避免死锁。

总的说来，尽管阻塞模式与非阻塞模式都可以同时处理多个客户连接，但阻塞模式需要使用较多的线程，而非阻塞模式只需使用较少的线程，非阻塞模式能更有效地利用 CPU，系统开销小，因此有更高的并发性能。

阻塞模式编程相对简单，但是当线程数目很多时，必须处理好线程之间的同步，如果自己编写线程池，要实现健壮的线程池难度较高。阻塞模式比较适用于同步通信，并且通信双方稳定地发送小批量的数据，双方都不需要花很长时间等待对方的回应。假如通信过程中，由于一方迟迟没有回应，导致另一方长时间地阻塞，为了避免线程无限期地阻塞下去，应该设置超时时间，及时中断长时间阻塞的线程。

非阻塞模式编程相对难一些，对 ByteBuffer 缓冲区的处理比较麻烦。非阻塞模式比较适用于异步通信，并且通信双方发送大批量的数据，尽管一方接收到另一方的数据可能要花一段时间，但在这段时间内，接收方不必傻傻地等待，可以处理其他事情。

4.8 练习题

1．在服务器程序中，线程在哪些情况可能会进入阻塞状态？（多选）

a) 线程执行 Socket 的 getInputStream()方法获得输入流。

b) 线程执行 Socket 的 getOutputStream()方法获得输出流。

c) 线程执行 ServerSocket 的 accept()方法。

d) 线程从 Socket 的输入流读入数据。

e) 线程向 Socket 的输出流写一批数据。

2．ServerSocketChannel 可能发生哪个事件？（单选）

a) SelectionKey.OP_ACCEPT：接收连接就绪事件

b) SelectionKey.OP_CONNECT：连接就绪事件

c) SelectionKey.OP_READ：读就绪事件

d) SelectionKey.OP_WRITE：写就绪事件

3．SocketChannel 可能发生哪些事件？（多选）

a) SelectionKey.OP_ACCEPT：接收连接就绪事件

b) SelectionKey.OP_CONNECT：连接就绪事件

c) SelectionKey.OP_READ：读就绪事件

d) SelectionKey.OP_WRITE：写就绪事件

4. 对于以下代码：

```
int n=socketChannel.read(byteBuffer);  //假定 n>=0
byteBuffer.flip();
```

假定执行 socketChannel.read(byteBuffer)方法前，byteBuffer 的容量、极限和位置分别为 c、l 和 p，执行完以上代码后，byteBuffer 的容量、极限和位置分别是多少？（单选）

a) 容量为 c，极限为 l，位置为 p+n

b) 容量为 c，极限为 p+n，位置为 0

c) 容量为 l，极限为 p+n，位置为 0

d) 容量为 c，极限为 p+n，位置为 p+n

5．在哪些情况，SelectionKey 对象会失效？（多选）

a) 程序调用 SelectionKey 的 cancel()方法

b) 程序调用 SelectionKey 的 close()方法

c) 关闭与 SelectionKey 关联的 Channel

d) 关闭与 SelectionKey 关联的 Selector

6．线程执行 Selector 对象的 select(long timeout)方法时进入阻塞状态，在哪些情况，线程会从 select()方法中返回？（多选）

a) 至少有一个 SelectionKey 的相关事件已经发生

b) 其他线程调用了 Selector 对象的 wakeup()方法

c) 与 Selector 对象关联的一个 SocketChannel 对象被关闭

d) 当前执行 select()方法的线程被其他线程中断

e) 超出了等待时间

7．默认情况下，SocketChannel 对象处于什么模式？（单选）

a) 阻塞模式　b) 非阻塞模式

8. SwingWorker 类的哪个方法表示由 SwingWorker 工作线程执行的后台任务？（单选）

a) done()　　b) publish()　　c) doInBackground()　　d) process()

9．运行 4.6.3 节的 AsynEchoClient 类，当用户快速在用户界面的文本框中不断进行回车操作，会有哪些线程在并发运行？（单选）

a) 一个 main 主线程。

b) 一个 main 主线程和一个 EDT 线程。

b)一个 EDT 线程和一个 SwingWorker 工作线程。

d) 一个 EDT 线程和多个来自于 SwingWorker 工作线程池的 SwingWorker 工作线程。

10．以下哪些选项中的两个线程属于异步执行各自的操作？（多选）

a)线程 A 不断向一个 ByteBuffer 缓冲区中写入数据，线程 B 不断从这个 ByteBuffer 缓冲区中读取数据。

b) 线程 A 不断向一个 ByteBuffer 缓冲区中写入数据，线程 B 一直等待，直到这个缓冲区已经写满后，才开始从缓冲区中读取数据。当线程 B 从缓冲区读取数据时，线程 A 开始等待，直到线程 B 把缓冲区清空。

c)一个 SwingWorker 工作线程在执行一个 SwingWorker 对象的 doInBackground()方法，而 EDT 线程在监听图形用户界面触发的事件。

d) 一个 SwingWorker 工作线程在执行一个 SwingWorker 对象的 doInBackground()方法。而 EDT 线程在执行这个 SwingWorker 对象的 get()方法。

11. 编写一个用于诊断网络传输数据准确性的程序。服务器端（exercise.IntgenServer 类）不断发送 int 类型的整数，从 0 开始递增。客户端（exercise.IntgenClient 类）不断接受来自服务器端发送的数据，判断实际收到的数据与预期应该收到的数据是否一致。如果不一致，就说明网络在发送这一 int 数据的字节序列时存在错误。

服务器和客户端都采用通道来通信。服务器端采用非阻塞模式，而客户端采用阻塞模式。

答案：(1)c,d,e　(2)a　(3)b,c,d　(4)b　(5) a,c,d　(6)a,b,d,e　(7)a　(8)c　(9)d　(10)a,c

(11)参见配套源代码包的 sourcecode/chapter04/src/exercise 目录下的 IntgenServer 和 IntgenClient 类。

编程提示：服务器端以递增的方式发送 int 类型的数据，当达到 int 类型的最大值时，Integer.MAX_VALUE+1 的取值为负数（等价于 Integer.MIN_VALUE），所以，接下去会继续以递增的方式发送负数。客户端的 SocketChannel 采用阻塞模式读取数据，读到一个 int 数据后，就会判断它与预期的数据是否一致，如果不一致，就会打印出错信息，然后再接受来自服务器的下一个 int 数据。因此客户端采用的是同步通信。而服务器端则不必等到客户端已经收到当前 int 数据，就能继续发送下一个 int 数据，所以服务器端采用的是异步通信。