用 Java 实现非阻塞通信

孙卫琴

摘　要　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，从而理解它们的区别和适用范围。

关键词　阻塞通信，非阻塞通信，同步通信，异步通信，缓冲区

用 ServerSocket 和 Socket 来编写服务器程序和客户程序，是 Java 网络编程的最基本的方式。这些服务器程序或客户程序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 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，让它们分别负责与一个客户通信，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。

一、线程阻塞

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。

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。

2. 服务器程序用多线程处理阻塞通信的局限

图 1 显示了服务器程序用多线程来同时处理多个客户连接的工作流程。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。

在图 1 中，用粗体框标识的步骤为可能引起阻塞的步骤。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。

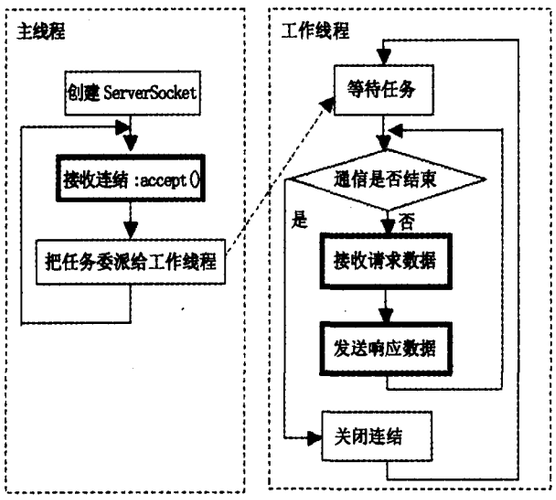

图 1 服务器程序用多线程处理阻塞通信

由此可见，工作线程并不是越多越好。如图 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。

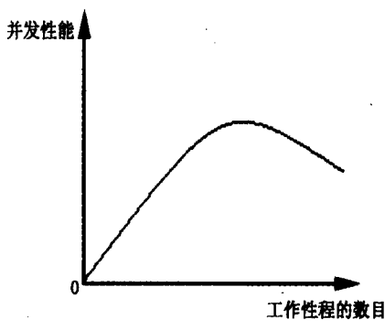

图 2 线程数目与并发性能的关系

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：

```
锅里放水，打开煤气炉; //开始烧开水
锅里放水和米，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，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。

二、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，如图 3 所示。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。

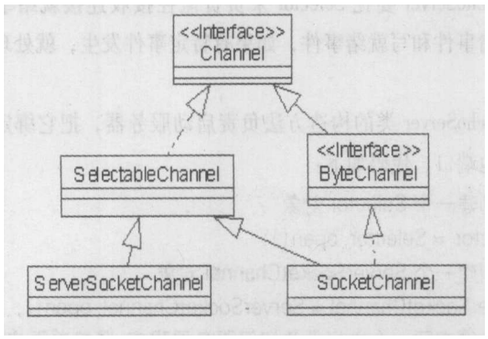

图 3　SelectableChannel 类及其子类的类框图

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。

三、非阻塞编程实例

1. 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：

- 接收客户的连接。
- 接收客户发送的数据。
- 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
  Set readyKeys = selector.selectedKeys();
 //获得 Selector 的 selected -keys 集合
  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
    it.remove();
   //把 SelectionKey 从 Selector 的 selected -key 集合中删除
    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();
      key.channel().close();
     //关闭与这个 SelectionKey 关联的 SocketChannel
     }
    }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 configureBlock-

ing(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。

（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()方法的许多代码都涉及对 ByteBuffer 的三个属性（position、limit 和 capacity）的操作，图 4 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。

（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：

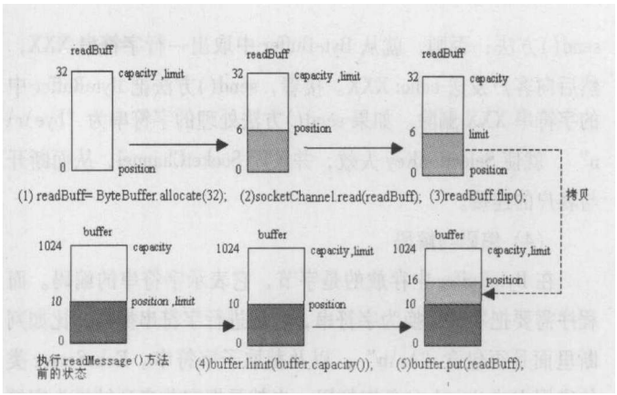

图 4　receive()方法操纵 readBuff 和 buffer 的过程

```
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。

（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：

```
while(outputBuffer.hasRemaining())
 //hasRemaining()方法判断是否还有未处理的字节
 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 5 演示了以上代码操纵 buffer 的过程。图 5 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。

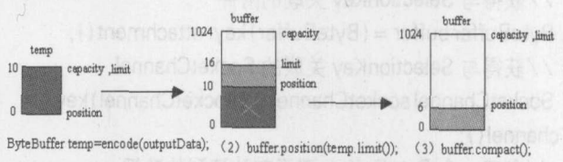

图 5 从 buffer 中删除已经处理过的一行字符串 XXX

例程 1 是 EchoServer 的源程序。

```
//例程 1 EchoServer.java(非阻塞模式)
package nonblock;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.net.*;
import java.util.*;
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 InetSocketAd-
dress(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();
}
}
```

2. 在 EchoServer 中混合用阻塞模式与非阻塞模式

在例程 1 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 2 是 EchoServer 类的源程序。其中 receive()、send()、decode()和 encode()方法的代码与例程 1 的 EchoServer 类相同，为了节省篇幅，不再重复显示。

```
//例程 2 EchoServer. java(混合使用阻塞模式与非阻塞模式)
package thread2;
import java. io. *;
import java. nio. *;
import java. nio. channels. *;
import java. nio. charset. *;
import java. net. *;
import java. util. *;
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 类的构造方法与例程 1 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。

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。

3. 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。例程 1 和例程 2 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，并不要求必须读到一行数据后，才能执行发送数据的操作。

例程 3 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 6 显示了这两个 Buffer 的作用。

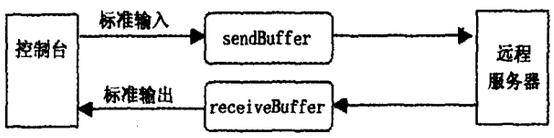

图 6　sendBuffer 和 receiveBuffer 的作用

```
//例程 3 EchoClient.java(非阻塞模式)
package nonblock;
import java.net.*;
import java.nio.channels.*;
import java.nio.*;
import java.io.*;
import java.nio.charset.*;
import java.util.*;
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();
}
public void receiveFromUser(){
//接收用户从控制台输入的数据,把它放到 sendBuffer 中
try{
BufferedReader localReader = new BufferedReader(new In-
putStreamReader(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。

四、结语

本文介绍了用 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。

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，导致另一方长时间的阻塞，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。

参考文献

1 孙卫琴．Java 面向对象编程．电子工业出版社．2006.
2 孙卫琴．Java 网络编程精解．电子工业出版社．2007.

（收稿日期：2008 年 3 月 19 日）