# 【第02章-NIO入门】【03-NIO编程】

## NIO编程

NIO 到底是什么的简称？有人称之为New IO ， 原因在于它相对于之前的IO类库是新增的。这是它的官方叫法。但是，由于之前老的IO类库是阻塞IO，New IO 类库的目标就是要让Java 支持非阻塞IO， 所以，更多的人喜欢称之为非阻塞IO( Non-block IO ）。由于非阻塞IO更能够体现NIO 的特点， 所以常用NIO 都指的是非阻塞IO。

与Socket类似 ServerSocket 类相对应，NIO也捉－供了SocketChannel 和ServerSocketChannel两种不同的套接字通道实现。这两种新增的通道都支持阻塞和非阻塞两种模式。阻塞模式使用非常简单， 但是性能和可靠性都不好，非阻寒模式则正好相反。开发人员可以根据自己的需要来选择合适的模式。一说，低负载、低并发的应用程序可以选择同步阻塞IO以降低编程复杂度：对于高负载、高并发的网络应用，需要使用NIO 的非阻塞模式进行开发。

## NIO类库简介

新的输入／输出（ N I O ）库是在JDK 1 .4 中引入的。N IO 弥补了原来同步阻塞IO不足，在标准Java 代码中提供了高速的、面向块的IO。通过定义包含数据的类，以及通过以块的形式处理这些数据， NIO 不用使用本机代码就可以利用低级优化，这是原来的IO包所无法做到的。下面我们对N IO 的一些概念和功能做下简单介绍，以便大家能够快速地了解NIO 类库和相关概念。

### 缓冲区Buffer

Buffer 是一个对象，它包含一些要写入或者要读出的数据。在NIO 类库中加入Buffer 对象，体现了新库与原IO的一个重要区别。在面向流的IO中，可以将数据直接写入或者将数据直接读到Stream 对象中。

在NIO 库看，所有的数据都是使用缓冲区处理的，在读取数据时，它是直接读取到缓冲区的：在写入数据时，写入到缓冲区中。任何时候访问NIO 中的数据， 都是通过缓冲区进行操作。

缓冲区实质上是一个数组。通常它是一个字节数组（ByteBuffer），也可以使用其他种类的数组。但是缓冲区不仅仅是一个数组，缓冲区提供了对数据的结构化访问以及维护读写位置（limit）等信息。

最常用的缓冲区是ByteBuffer， 一个ByteBuffer 提供了一组功能用于操作byte 数组。除了ByteBuffer. 还有其他的一些缓冲区， 事实上，每一种Java 基本类型（除了Boolean类型）都对应有一种缓冲区，缓冲区的类图继承关系如图1所示。



图1 Buffer继承关系图

每一个Buffer 类都是Buffer 接口的一个子类。除了Byte Buffer，每一个Buffer 类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准IO操作都使用ByteBuffer ，所以它在具有一般缓冲区的操作之外还提供了一些特有的操作，以方便网络读写。

### 通道Channel

Channel 是一个通道，它就像自来水管一样，网络数据通过Channel 读取和写入。通边与流的不同之处在于通道是双向的，流只是在一个方向上移动（一个流必须是InputStream 或者OutputStream 的子类），而通道可以用于读、写或者二者同时进行。

因为Channel 是全双工的，所以在可以比流更好地映射底层操作系统的API 。特别是在UNIX 网络编程模型中，底层操作系统的通道都是全双工的，同时支持读写操作。

Channel 的类图继承关系如图2所示。实际上Channel 可以分为两大类： 用于网络读写的SelectableChannel 和用于文件操作的FileChannel。

### 多路复用器Selector

多路复用器Selector ，它是Java NIO 编程的基础，熟练地掌握Selector 对于NIO 编程至关重要。多路复用器提供选择己经就绪的任务的能力。简单来讲，Selector会不断地轮询注册在其上的Channel ，如果某个Channel 上面发生读或者写事件，这个Channel 就处于就绪状态，会被Selector 轮询出来，然后通过SelectionKey 可以获取就绪Channel 的集合，进行后续的IO 操作。

一个多路复用器Selector 可以同时轮询多个Channel ，由于JDK 使用了epoll()代替传统的select 实现，所以它并没有最大连接句柄l024/2048 的限制。这也就意味着只需要一个线程负责Selector 的轮询，就可以接入成千上万的客户端。



图2 Channel 继承关系类图

## NIO 服务端序列图

NIO 服务端通信序列图如图3所示。



图3 NIO服务端通信序列图

NIO服务端的主要创建过程如下：

步骤一： 打开 ServerSocketChannel ， 用于监听客户端的连接，它是所有客户端连接的父管道。

|  |
| --- |
| ServerSocketChannel acceptorSvr = ServerSocketChannel.open(); |

步骤二： 绑定监昕端口，设置连接为非阻塞模式。

|  |
| --- |
| acceptorSvr.socket().bind(newInetSocketAddress(InetAddress.getByName(“IP”), port));  acceptorSvr.configureBlocking (false ); |

步骤三： 创建Reactor 线程，创建多路复用器并启动线程，

|  |
| --- |
| Selector selector= Selector.open();  New Thread(new ReactorTask()).start() ; |

步骤四： 将ServerSocketChannel 注册到Reactor 线程的多路复用器Selector 上，监听ACCEPT 事件。

|  |
| --- |
| SelectionKey key = acceptorSvr.register ( selector, SelectionKey.OP\_ACCEPT, ioHandler) ; |

步骤五： 多路复用器在线程run 方法的无限循环体内轮询准备就绪的Key

|  |
| --- |
| int num =selector.select();  Set selectedKeys = selector.selectedKeys();  Iterator it= selectedKeys.iterator();  while (it . hasNe x t ()) {  SelectionKey key= (SelectionKey) it.next() ;  // ... deal with IIO event  } |

步骤六： 多路复用器监听到有新的客户端接入， 处理新的接入请求， 完成TCP 三次握手，建立物理链路，

|  |
| --- |
| SocketChannel channel= svrChannel . accept( ); |

步骤七： 设置客户端链路为非阻塞模式。

|  |
| --- |
| channel.configureBlocking(false);  channel.socket().setReuseAddress(true); |

步骤八： 将新接入的客户端连接注册到Reactor 线程的多路复用器上，监听读操作，读取客户端发送的网络消息。

|  |
| --- |
| SelectionKey key= socketChannel.register (selector, SelectionKey.OP\_READ, ioHandler); |

步骤九： 异步读取客户端请求消息到缓冲区

|  |
| --- |
| int readNumber = channel.read(receivedBuffer); |

步骤十： 对ByteBuffer 进行编解码，如果有半包消息指针reset ， 继续读取后续的报文，将解码成功的消息封装成Task ，投递到业务线程池中，进行业务逻辑编排，

|  |
| --- |
| Object message = null ;  while(buffer.hasRemain()){  byteBuffer.mark();  Object message= decode (byteBuffer);  if (message == null){  byteBuffer . reset( );  break;  }  messageList.add(message) ;  if (!byteBuffer.hasRemain()){  byteBuffer.clear() ;  }else{  byteBuffer.compact();  }  if (messageList !=null && !messageList.isEmpty()){  for(Object messageE : messageList){  handlerTask (messageE);  }  } |

步骤十一： 将POJO 对象encode 成Byte Buffer ， 调用SocketChannel 的异步write 接口，将消息异步发送给客户端，

|  |
| --- |
| socketChannel.write(buffer); |

注意： 如果发送区TCP 缓冲区满，会导致写半包，此时，需要注册监听写操作位，循环写， 直到整包消息写入TCP 缓冲区。

## NIO 客户端序列图

NIO客户端创建序列图如图4所示。



图4 NIO 客户端创建序列图

步骤一： 打开SocketChannel ，绑定客户端本地地址（ 可边，默认系统会随机分配一个可用的本地地址）。

|  |
| --- |
| SocketChannel clientChannel = SocketChannel.open() ; |

步骤二： 设置SocketCbannel 为非阻塞模式，同时设置客户端连接的TCP 参数。

|  |
| --- |
| clientChannel.configureBlocking(false) ;  socket.setReuseAddress (true) ;  socket.setReceiveBufferSize(BlJFFER\_SIZE);  socket.setSendBufferSize(BUFFER\_SIZE) ; |

步骤三： 异步连接服务端

|  |
| --- |
| boolean connected=clientChannel.connect(new InetSocketAddress(“IP”, port)); |

步骤四： 判断是否连接成功，如果连接成功，则直接注册读状态位到多路复用器中，如果当前没有连接成功（异步连接，返回false ，说明客户端已经发送sync 包，服务端没有返回ack 包，物理链路还没有建立）。

|  |
| --- |
| if (connected){  clientChannel.register(selector, SelectionKey.OP\_READ, ioHandler);  }else{  clientChannel.register(selector, SelectionKey.OP\_CONNECT, ioHandler);  } |

步骤五： 向Reactor 线程的多路复用器注册OP\_CONNECT 状态位，监听服务端的TCP ACK 应答。

|  |
| --- |
| clientChannel.register(selector, SelectionKey.OP\_CONNECT, ioHandler); |

步骤六：创建Reactor 线程， 创建多路复用器并启动线程。

|  |
| --- |
| Selector selector= Selector.open();  New Thread(new ReactorTask()).start(); |

步骤七： 多路复用器在线程run 方法的无限循环体内轮询准备就绪的Key

|  |
| --- |
| int num = selector.select() ;  Set selectedKeys = selector.selectedKeys();  Iterator it = selectedKeys.iterator();  while (it. hasNext ()){  SelectionKey key = (SelectionKey)it.next () ;  // ... deal with IO event ...  } |

步骤八： 接收connect 事件进行处理

|  |
| --- |
| if (key.isConnectable())  // handlerConnect(); |

步骤九：判断连接结果，如果连接成功， 注册读事件到多路复用器，

|  |
| --- |
| if (channel.finishConnect()){  registerRead();  } |

步骤十： 注册读事件到多路复用器。

|  |
| --- |
| clientChannel.register(selector, SelectionKey.OP\_READ, ioHandler); |

步骤十一： 异步读客户端请求消息到缓冲区

|  |
| --- |
| int readNumber = channel. read (receivedBuffer) ; |

步骤十二： 对ByteBuffer 进行编解码，如果有半包消息接收缓冲区Reset，继续读取后续的报文， 将解码成功的消息封装成Task ，投递到业务线程池中，进行业务逻辑编排。

|  |
| --- |
| Object message = null ;  while (buffer.hasRemain()){  byteBuffer.mark();  Object message= decode (byteBufferJ;  if (message == null){  byteBuffer . reset() ;  break;  }  messageList .add(message );  }  if (!byteBuffer.hasRemain()){  byteBuffer. clear( );  }else{  byteBuffer.compact() ;  if (messageList !=null & !messageList.isEmpty()){  for (Object messageE : messageList){  handlerTask(messageE) ;  }  } |

步骤十三： 将POJO 对象encode 成ByteBuffe r，调用SocketChannel 的异步write 接口，将消息异步发送给客户端。

|  |
| --- |
| socketChannel.write(buffer); |

## 使用NIO编程的优点

（1）客户端发起的连接操作是异步的，可以通过在多路复用器注册OP\_CONNECT 等待后续结果， 不需要像之前的客户端那样被同步阻塞。

（2）SocketChannel的读写操作都是异步的，如果没有可读写的数据它不会同步等待，直接返回，这样IO通信线程就可以处理其他的链路，不需要同步等待这个链路可用。

（3）线程模型的优化：由于JDK 的Selector 在Linux 等主流操作系统上通过epoll 实现，它没有连接句柄数的限制（只受限于操作系统的最大句柄数或者对单个进程的句柄限制），这意味着一个Selector 线程可以同时处理成千上万个客户端连接，而且性能不会随着客户端的增加而线性下降。因此，它非常适合做高性能、高负载的网络服务器。

JDKl.7升级了NIO 类库，升级后的N IO 类库被称为NIO 2.0 。引人注目的是， Java正式提供了异步文件IO 操作，同时提供了与UNIX 网络编程事件驱动IO 对应的AIO 。