Java NIO

前置知识：

1. 同步： 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）。

2. 异步： 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）。

3. 阻塞： ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）。

4. 非阻塞： 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）。

Java BIO（Blocking IO）： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。JDK1.4以前唯一的版本。

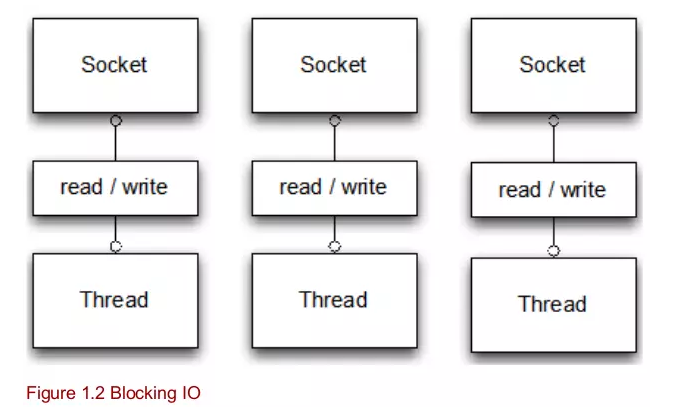
Java NIO（Non Blocking IO）： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。JDK1.4开始支持。

Java AIO(NIO.2)： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。JDK1.7开始支持。

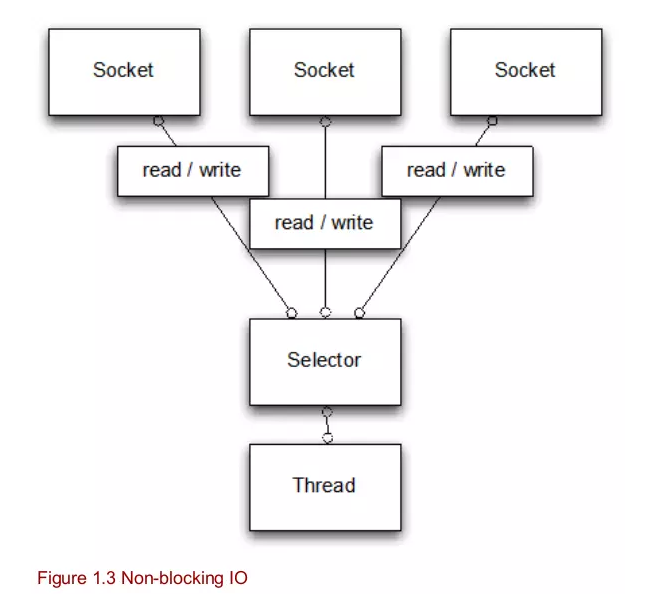
[Blocking VS non-blocking IO](https://www.jianshu.com/p/be2724f24b41)

[Java BIO、NIO、AIO 学习](https://blog.51cto.com/stevex/1284437)

BIO中，一次连接创建一个线程处理这个连接的IO事件



NIO中，使用单个selector可处理多个连接



Java NIO 由以下几个核心部分组成：

\* Channels

\* Buffers

\* Selectors

Channel

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点像流，但又有些不同。

1. 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中，但流的读写通常是单向的。

2. 通道可以异步地读写。

3. 通道中的数据总是要先读到一个Buffer，或者总是要从一个Buffer中写入。

正如上面所说，从通道读取数据到缓冲区，从缓冲区写入数据到通道。

如下图



Channel有好几种类型。下面是JAVA NIO中的一些主要Channel的实现：

FileChannel // 文件IO，从文件中读写数据

DatagramChannel // UDP网络IO，能通过UDP读写网络中的数据

SocketChannel // 能通过TCP读写网络中的数据

// 可以监听新进来的TCP连接，像Web服务器那样。对每一个新进来的连接都会创建一个SocketChannel

ServerSocketChannel

例子，使用FileChannel读取数据到Buffer中



注意 buf.flip() 的调用，首先读取数据到Buffer，然后反转Buffer,接着再从Buffer中读取数据。

[Java NIO系列教程（二） Channel](http://ifeve.com/channels/)

Buffer

缓冲区本质上**是一块**可以写入数据，然后可以从中读取数据的**内存**。这块内存被包装成NIO Buffer对象，并提供了一组方法，用来方便的访问该块内存。

以下是Java NIO里关键的Buffer实现（八大数据类型中除了布尔值Boolean）：

ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer、ShortBuffer

这些Buffer覆盖了你能通过IO发送的基本数据类型（除了布尔值）：

byte、short、int、long、float、double 、char。

Java NIO 还有个 MappedByteBuffer，用于表示内存映射文件。

Buffer基本用法包括四个步骤：

1. 写入数据到Buffer

2. 调用flip()方法

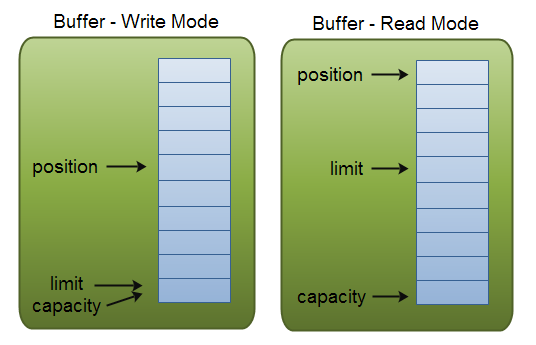
3. 从Buffer中读取数据

4. 调用clear()方法或者compact()方法

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用clear()或compact()方法。clear()方法会**清空整个缓冲区**。compact()方法只会**清除已经读过的数据**。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

为了理解Buffer的工作原理，需要熟悉它的三个属性：capacity、position、limit

position和limit的含义取决于Buffer处在读模式还是写模式。不管Buffer处在什么模式，capacity的含义总是一样的。



capacity，作为一个内存块，Buffer有一个固定的大小值，也叫“capacity”.你只能往里写capacity个byte、long，char等类型。一旦Buffer满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

position，当你写数据到Buffer中时，position表示当前的位置。初始的position值为0.当一个byte、long等数据写到Buffer后，position会向前移动到下一个可插入数据的Buffer单元。position最大可为capacity – 1.

当读取数据时，也是从某个特定位置读。当将Buffer从写模式切换到读模式，position会被重置为0. 当从Buffer的position处读取数据时，position向前移动到下一个可读的位置。

limit，在写模式下，Buffer的limit表示你最多能往Buffer里写多少数据，等效于Buffer的capacity。

当切换Buffer到读模式时， limit表示你最多能读到多少数据。因此，当切换Buffer到读模式时，limit会被设置成写模式下的position值。换句话说，你能读到之前写入的所有数据（limit被设置成已写数据的数量，这个值在写模式下就是position）

[Java NIO系列教程（三） Buffer](http://ifeve.com/buffers/)

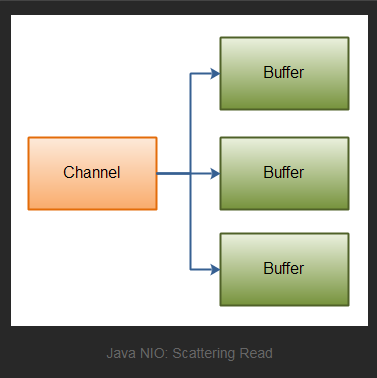
Scatter/Gather

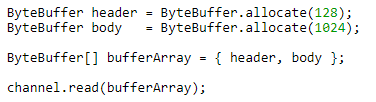
分散（scatter）从Channel中读取是指在读操作时将读取的数据写入多个buffer中。因此，Channel将从Channel中读取的数据“分散（scatter）”到多个Buffer中。

聚集（gather）写入Channel是指在写操作时将多个buffer的数据写入同一个Channel，因此，Channel 将多个Buffer中的数据“聚集（gather）”后发送到Channel。

scatter / gather经常用于需要将传输的数据分开处理的场合，例如传输一个由消息头和消息体组成的消息，你可能会将消息体和消息头分散到不同的buffer中，这样你可以方便的处理消息头和消息体。

Scattering Reads是指数据从一个channel读取到多个buffer中。如下图描述：

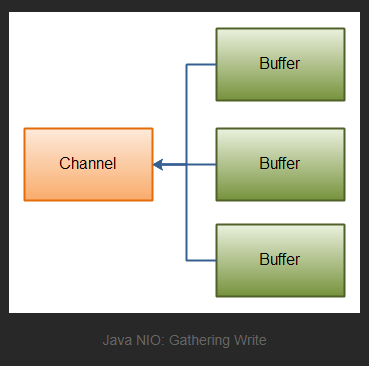


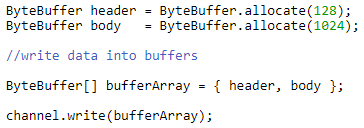


注意buffer首先被插入到数组，然后再将数组作为channel.read() 的输入参数。read()方法按照buffer在数组中的顺序将从channel中读取的数据写入到buffer，当一个buffer被写满后，channel紧接着向另一个buffer中写。

Scattering Reads在移动下一个buffer前，必须填满当前的buffer，这也意味着它不适用于动态消息(译者注：消息大小不固定)。换句话说，如果存在消息头和消息体，消息头必须完成填充（例如 128byte），Scattering Reads才能正常工作。

Gathering Writes是指数据从多个buffer写入到同一个channel。如下图描述：





buffers数组是write()方法的入参，write()方法会按照buffer在数组中的顺序，将数据写入到channel，注意只有position和limit之间的数据才会被写入。因此，如果一个buffer的容量为128byte，但是仅仅包含58byte的数据，那么这58byte的数据将被写入到channel中。因此与Scattering Reads相反，Gathering Writes能较好的处理动态消息。

[Java NIO系列教程（四） Scatter/Gather](http://ifeve.com/java-nio-scattergather/)

Selector

Selector是Java NIO中能够检测一到多个NIO通道，并能够知晓通道是否为诸如读写事件做好准备的组件。允许单线程管理多个 Channel。事实上，可以只用一个线程处理所有的通道。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系统的一些资源（如内存）。因此，使用的线程越少越好。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

这是在一个单线程中使用一个Selector处理3个Channel的图示：



要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

1. Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

Selector selector = Selector.open();

2. 向Selector注册通道

为了将Channel和Selector配合使用，必须将channel注册到selector上。通过SelectableChannel.register()方法来实现，如下：

channel.configureBlocking(false);

SelectionKey key = channel.register(selector,Selectionkey.OP\_READ);

与Selector一起使用时，**Channel必须处于非阻塞模式下**。这意味着不能将FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式。

register()方法的第二个参数。这是一个“interest集合”，意思是在通过Selector监听Channel时对什么事件感兴趣。可以监听四种不同类型的事件以及对应的常量：

1. Connect（SelectionKey.OP\_CONNECT） // 连接事件

2. Accept（SelectionKey.OP\_ACCEPT） // 接收事件

3. Read（SelectionKey.OP\_READ） // 读事件

4. Write（SelectionKey.OP\_WRITE） // 写事件

通道触发了一个事件意思是该事件已经就绪。所以，某个channel成功连接到另一个服务器称为“连接就绪”。一个server socket channel准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

如果你对不止一种事件感兴趣，那么可以用“位或”操作符将常量连接起来，如下：

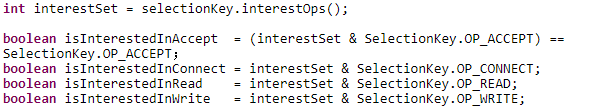
int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

3. SelectionKey

当向Selector注册Channel时，register()方法会返回一个SelectionKey对象。这个对象包含了一些你感兴趣的属性：

3.1 interest集合

interest集合是你所选择的感兴趣的事件集合。可以通过SelectionKey读写interest集合。



可以看到，用“位与”操作interest 集合和给定的SelectionKey常量，可以确定某个确定的事件是否在interest 集合中。

3.2. ready集合

ready 集合是通道**已经准备就绪的操作的集合**。可以这样访问ready集合：

int readySet = selectionKey.readyOps();

可以用像检测interest集合那样的方法，来检测channel中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：



3.3 Channel+Selector

从SelectionKey访问Channel和Selector很简单。如下：



3.4 附加的对象（可选）

可以将一个对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。例如，可以附加与通道一起使用的Buffer，或是包含聚集数据的某个对象。使用方法如下：



还可以在用register()方法向Selector注册Channel的时候附加对象。如：



4. 通过Selector选择通道

一旦向Selector注册了一或多个通道，就可以调用几个重载的select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道。换句话说，如果你对“读就绪”的通道感兴趣，select()方法会返回**读事件已经就绪的那些通道**。

int select() // 阻塞到至少有一个通道在你注册的事件上就绪了

int select(long timeout) // 和select()一样，除了最长会阻塞timeout毫秒(参数)

int selectNow() // 不会阻塞，不管什么通道就绪都立刻返回（此方法执行非阻塞的选择操作。如果自从前一次选择操作后，没有通道变成可选择的，则此方法直接返回零。）

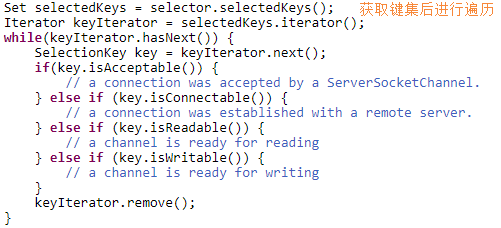
4.1 selectedKeys()

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。如下所示：

Set selectedKeys = selector.selectedKeys();

当像Selector注册Channel时，Channel.register()方法会返回一个SelectionKey 对象。这个对象代表了注册到该Selector的通道。可以通过SelectionKey的selectedKeySet()方法访问这些对象。

可以遍历这个已选择的键集合来访问就绪的通道。如下：



这个循环遍历已选择键集中的每个键，并检测各个键所对应的通道的就绪事件。

**注意**每次迭代末尾的keyIterator.remove()调用。Selector不会自己从已选择键集中移除SelectionKey实例。必须在处理完通道时自己移除。**下次该通道变成就绪时，Selector会再次将其放入已选择键集中**。

SelectionKey.channel()方法返回的通道需要转型成你要处理的类型，如ServerSocketChannel或SocketChannel等。

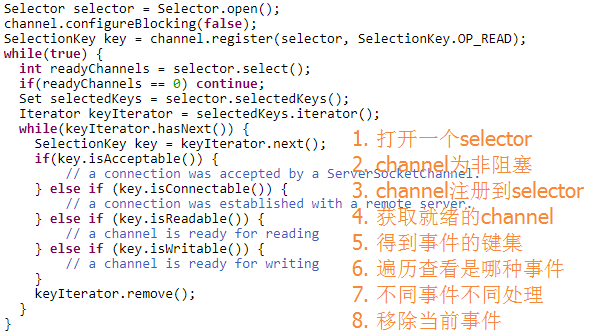
4.2 wakeUp()

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，也有办法让其从select()方法返回。只要让其它线程在第一个线程调用select()方法的那个对象上调用Selector.wakeup()方法即可。阻塞在select()方法上的线程会立马返回。如果有其它线程调用了wakeup()方法，但当前没有线程阻塞在select()方法上，下个调用select()方法的线程会立即“醒来（wake up）”。

4.3 close()

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey实例无效。通道本身并不会关闭

完整示例，打开一个Selector，注册一个通道注册到这个Selector上(通道的初始化过程略去),然后持续监控这个Selector的四种事件（接受，连接，读，写）是否就绪。



[Java NIO系列教程（六） Selector](http://ifeve.com/selectors/)

FileChannel

FileChannel是一个连接到文件的通道。可以通过文件通道读写文件。无法设置为非阻塞模式，它总是运行在阻塞模式下。

1. 打开FileChannel

在使用FileChannel之前，必须先打开它。但是，我们**无法直接打开**一个FileChannel，需要通过使用一个InputStream、OutputStream或RandomAccessFile来获取一个FileChannel实例。下面是通过RandomAccessFile打开FileChannel的示例：



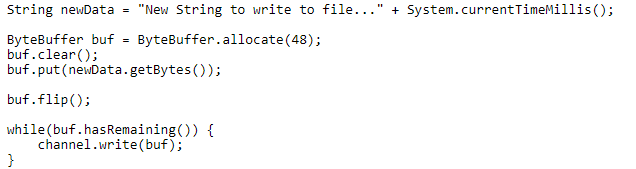
2. 从FileChannel读取数据

首先，分配一个Buffer。从FileChannel中读取的数据将被读到Buffer中。然后，调用FileChannel.read()方法。该方法将数据从FileChannel读取到Buffer中。read()方法返回的int值表示了有多少字节被读到了Buffer中。如果返回-1，表示到了文件末尾。



3. 向FileChannel写数据

注意FileChannel.write()是在while循环中调用的。因为无法保证write()方法一次能向FileChannel写入多少字节，因此需要重复调用write()方法，直到Buffer中已经没有尚未写入通道的字节。



4. 关闭FileChannel

用完FileChannel后必须将其关闭。如：



FileChannel的position方法

有时可能需要在FileChannel的某个特定位置进行数据的读/写操作。

可以通过调用position()方法获取FileChannel的当前位置(long pos = channel.position();)。

也可以通过调用position(long pos)方法设置FileChannel的当前位置。

如果将位置设置在文件结束符之后，然后试图从文件通道中读取数据，读方法将返回-1（文件结束标志）。

如果将位置设置在文件结束符之后，然后向通道中写数据，文件将撑大到当前位置并写入数据。这可能导致“文件空洞”，磁盘上物理文件中写入的数据间有空隙。

// TODO

FileChannel的size方法

将返回该实例所关联文件的大小。



FileChannel的truncate方法

截取一个文件。截取文件时，文件将中指定长度后面的部分将被删除。



FileChannel的force方法

将通道里尚未写入磁盘的数据强制写到磁盘上。出于性能方面的考虑，操作系统会将数据缓存在内存中，所以无法保证写入到FileChannel里的数据一定会即时写到磁盘上。要保证这一点，需要调用force()方法。

force()方法有一个boolean类型的参数，指明是否同时将文件元数据（权限信息等）写到磁盘上。

下面的例子同时将文件数据和元数据强制写到磁盘上：



[Java NIO系列教程（七） FileChannel](http://ifeve.com/file-channel/)

SocketChannel

Java NIO中的SocketChannel是一个连接到TCP网络套接字的通道。可以通过以下2种方式创建SocketChannel：

1. 打开一个SocketChannel并连接到互联网上的某台服务器。

2. 一个新连接到达ServerSocketChannel时，会创建一个SocketChannel。

1. 打开 SocketChannel



2. 关闭 SocketChannel

当用完SocketChannel之后调用SocketChannel.close()关闭SocketChannel：



3. 从 SocketChannel 读取数据

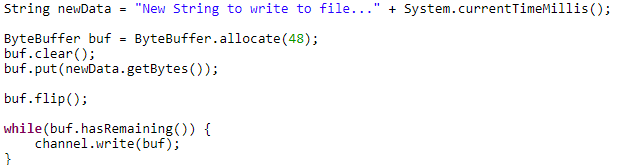
首先，分配一个Buffer。从SocketChannel读取到的数据将会放到这个Buffer中。

然后，调用SocketChannel.read()。该方法将数据从SocketChannel 读到Buffer中。read()方法返回的int值表示读了多少字节进Buffer里。如果返回的是-1，表示已经读到了流的末尾（连接关闭了）



4. 写入 SocketChannel

写数据到SocketChannel用的是SocketChannel.write()方法，该方法以一个Buffer作为参数。注意SocketChannel.write()方法的调用是在一个while循环中的。Write()方法无法保证能写多少字节到SocketChannel。所以，我们重复调用write()直到Buffer没有要写的字节为止。

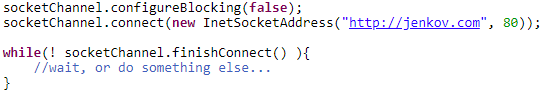


非阻塞模式

可以设置 SocketChannel 为非阻塞模式（non-blocking mode）.设置之后，就可以在异步模式下调用connect(), read() 和write()了。

connect()

如果SocketChannel在非阻塞模式下，此时调用connect()，该方法可能在连接建立之前就返回了。为了确定连接是否建立，可以调用finishConnect()的方法。像这样：



write()

非阻塞模式下，write()方法在尚未写出任何内容时可能就返回了。所以需要在循环中调用write()。前面已经有例子了，这里就不赘述了。

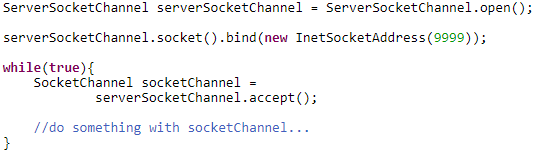
read()

非阻塞模式下,read()方法在尚未读取到任何数据时可能就返回了。所以需要关注它的int返回值，它会告诉你读取了多少字节。

[Java NIO系列教程（八） SocketChannel](http://ifeve.com/socket-channel/)

ServerSocketChannel

Java NIO中的 ServerSocketChannel 是一个可以监听新进来的TCP连接的通道, 就像标准IO中的ServerSocket一样。ServerSocketChannel类在 java.nio.channels包中。



1. 打开 ServerSocketChannel

通过调用 ServerSocketChannel.open() 方法来打开ServerSocketChannel.如：



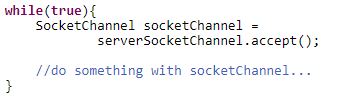
2. 关闭 ServerSocketChannel

通过调用ServerSocketChannel.close() 方法来关闭ServerSocketChannel. 如：



3. 监听新进来的连接

通过 ServerSocketChannel.accept() 方法监听新进来的连接。当 accept()方法返回的时候,它返回一个包含新进来的连接的 SocketChannel。因此, accept()方法会一直阻塞到有新连接到达。

通常不会仅仅只监听一个连接,在while循环中调用 accept()方法. 当然,也可以在while循环中使用除了true以外的其它退出准则。如下面的例子：

非阻塞模式

ServerSocketChannel可以设置成非阻塞模式。在非阻塞模式下，accept() 方法会立刻返回，如果还没有新进来的连接,返回的将是null。因此，需要检查返回的SocketChannel是否是null。如：



[Java NIO系列教程（九） ServerSocketChannel](http://ifeve.com/server-socket-channel/)

DatagramChannel

Java NIO中的DatagramChannel是一个能收发UDP包的通道。因为UDP是无连接的网络协议，所以不能像其它通道那样读取和写入。它发送和接收的是数据包。

1. 打开 DatagramChannel

打开的 DatagramChannel可以在UDP端口9999上接收数据包。

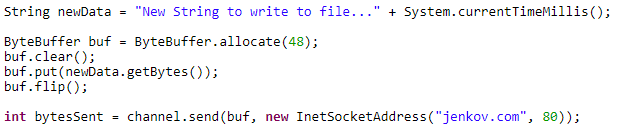


2. 接收数据

通过receive()方法从DatagramChannel接收数据，接收到的数据包内容复制到指定的Buffer，如果Buffer容不下收到的数据，多出的数据将被丢弃。



3. 发送数据



这个例子发送一串字符到”jenkov.com”服务器的UDP端口80。因为服务端并没有监控这个端口，所以什么也不会发生。也不会通知你发出的数据包是否已收到，因为UDP在数据传送方面没有任何保证。

4. 连接到特定的地址

可以将DatagramChannel“连接”到网络中的特定地址的。由于UDP是无连接的，连接到特定地址并不会像TCP通道那样创建一个真正的连接。而是锁住DatagramChannel，让其只能从特定地址收发数据。



当连接后，也可以使用read()和write()方法，就像在用传统的通道一样。只是在数据传送方面没有任何保证。



[Java NIO系列教程（十） Java NIO DatagramChannel](http://ifeve.com/datagram-channel/)

[Java NIO系列教程（十二） Java NIO与IO](http://ifeve.com/java-nio-vs-io/)

[Java NIO系列教程（十二） Java NIO与IO带原文图片](http://tutorials.jenkov.com/java-nio/nio-vs-io.html)