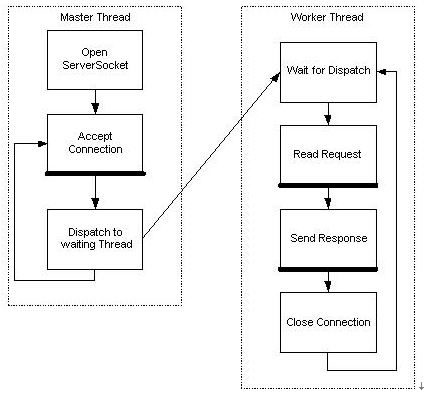
Java NIO教程

参考：http://ifeve.com/

**首先先分析下：为什么要nio套接字？**

nio是new io的简称。nio的主要作用就是用来解决速度差异的。举个例子：计算机处理的速度，和用户按键盘的速度。这两者的速度相差悬殊。如果按照经典的方法：一个用户设定一个线程，专门等待用户的输入，无形中就造成了严重的资源浪费：每一个线程都需要珍贵的cpu时间片，由于速度差异造成了在这个交互线程中的cpu都用来等待。下图为普通IO对于多个Socket的处理过程，也就是为每个用户设定一个线程。



**nio套接字是怎么做到的？**

其实，其中的思想很简单：轮询。一个线程轮询多个input；传统的方式是：有n个客户端就要有n个服务线程+一个监听线程，现在采取这种凡是，可以仅仅使用1个线程来代替n个服务线程以此来解决。

Java NIO 由以下几个核心部分组成：

* Channels
* Buffers
* Selectors

1． Buffers：它是包含数据且用于读写的线形表结构。其中还提供了一个特殊类用于内存映射文件的I/O操作。

2． Channels：包含socket，file和pipe三种管道，它实际上是双向交流的通道。

3． Selector：它将多元异步I/O操作集中到一个或多个线程中（它可以被看成是Unix中select（）函数或Win32中WaitForSingleEvent（）函数的面向对象版本）。

下图是 Channel 与 Buffer之间的联系



### Buffer的实现

以下是Java NIO里关键的Buffer实现：

* ByteBuffer
* CharBuffer
* DoubleBuffer
* FloatBuffer
* IntBuffer
* LongBuffer
* ShortBuffer

### Channel的实现

这些是Java NIO中最重要的通道的实现：

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

FileChannel 从文件中读写数据。

DatagramChannel 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。

ServerSocketChannel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。

## Selector

Selector允许单线程处理多个 Channel。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：



**具体Buffer 实例：**

使用Buffer读写数据一般遵循以下四个步骤：

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

当向buffer写入数据时，buffer会记录下写了多少数据。一旦要读取数据，需要通过flip()方法将Buffer从写模式切换到读模式。在读模式下，可以读取之前写入到buffer的所有数据。

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。

下面是一个例子：

/\*\*

\* @author NaihuiChen

\* @time 2016-11-1

\*/

public class FChannel {

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

RandomAccessFile aFile = new RandomAccessFile("E:\\code\\file\\input.txt", "rw"); // open file

FileChannel inChannel = aFile.getChannel(); // get channel from file

ByteBuffer buf = ByteBuffer.allocate(48); //allocate 48 byte for buffer

int bytesRead = inChannel.read(buf); // read into buffer.

while(bytesRead != -1){

buf.flip(); //make buffer ready to read

while(buf.hasRemaining()){

System.out.print((char) buf.get()); //read byte

}

buf.clear(); //clear buff for next write

bytesRead = inChannel.read(buf); //read data from channel

}

aFile.close(); // close file

}

}

**flip()方法**

flip方法将Buffer从写模式切换到读模式。调用flip()方法会将position设回0，并将limit设置成之前position的值。

换句话说，position现在用于标记读的位置，limit表示之前写进了多少个byte、char等 —— 现在能读取多少个byte、char等。

**clear()与compact()方法**

一旦读完Buffer中的数据，需要让Buffer准备好再次被写入。可以通过clear()或compact()方法来完成。

如果调用的是clear()方法，position将被设回0，limit被设置成 capacity的值。换句话说，Buffer 被清空了。Buffer中的数据并未清除，只是这些标记告诉我们可以从哪里开始往Buffer里写数据。

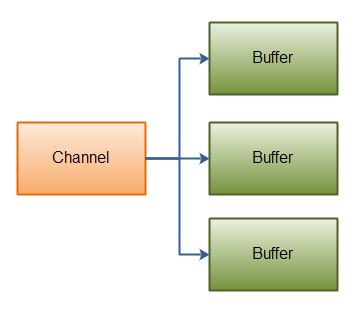
如果Buffer中有一些未读的数据，调用clear()方法，数据将“被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

如果Buffer中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用compact()方法。

compact()方法将所有未读的数据拷贝到Buffer起始处。然后将position设到最后一个未读元素正后面。limit属性依然像clear()方法一样，设置成capacity。现在Buffer准备好写数据了，但是不会覆盖未读的数据。

**Scattering Reads**

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



代码示例如下：

|  |  |
| --- | --- |
| 1 | ByteBuffer header = ByteBuffer.allocate(128); |

|  |  |
| --- | --- |
| 2 | ByteBuffer body   = ByteBuffer.allocate(1024); |

|  |  |
| --- | --- |
| 4 | ByteBuffer[] bufferArray = { header, body }; |

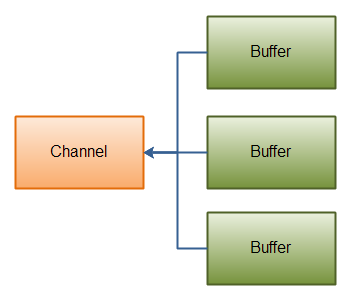
|  |  |
| --- | --- |
| 6 | channel.read(bufferArray); |

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**

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



代码示例如下：

|  |  |
| --- | --- |
| 1 | ByteBuffer header = ByteBuffer.allocate(128); |
| 2 | ByteBuffer body   = ByteBuffer.allocate(1024); | |

|  |  |
| --- | --- |
| 4 | //write data into buffers |

|  |  |
| --- | --- |
| 6 | ByteBuffer[] bufferArray = { header, body }; |

|  |  |
| --- | --- |
| 8 | channel.write(bufferArray); |

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。

SocketChannel（可以设置 block模式）

### 打开 SocketChannel

下面是SocketChannel的打开方式：

SocketChannel socketChannel = SocketChannel.open();

socketChannel.connect(new InetSocketAddress("http://jenkov.com", 80));

### 关闭 SocketChannel

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

socketChannel.close();

### 从 SocketChannel 读取数据

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = socketChannel.read(buf);

### 写入 SocketChannel

String newData = "New String to write to file..." + System.currentTimeMillis();

ByteBuffer buf = ByteBuffer.allocate(48);

buf.clear();

buf.put(newData.getBytes());

buf.flip();

while(buf.hasRemaining()) {

channel.write(buf);

}