1：Java NIO 由以下几个核心部分组成：

Channels

Buffers

Selectors

虽然Java NIO 中除此之外还有很多类和组件，但在我看来，Channel，Buffer 和 Selector 构成了核心的API。

2：所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。

3：Channel和Buffer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**

DatagramChannel

**SocketChannel UDP 和 TCP 网络IO**

ServerSocketChannel

正如你所看到的，这些通道涵盖了UDP 和 TCP 网络IO，以及文件IO。

4：以下是Java NIO里关键的Buffer实现：

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：byte, short, int, long, float, double 和 char。

5：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：



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。

[Java NIO系列教程（二） Channel](https://www.cnblogs.com/game-life/p/3971578.html)

1：Java NIO的通道类似流，但又有些不同：

既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。

通道可以异步地读写。

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。

2：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。

3：基本的 Channel 示例

下面是一个使用FileChannel读取数据到Buffer中的示例：

RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");

2 FileChannel inChannel = aFile.getChannel();

3 ByteBuffer buf = ByteBuffer.allocate(48);

6 int bytesRead = inChannel.read(buf);

7 while (bytesRead != -1) {

9 System.out.println("Read " + bytesRead);

10 buf.flip();

12 while(buf.hasRemaining()){

13 System.out.print((char) buf.get());14

}

16 buf.clear();

17 bytesRead = inChannel.read(buf);

}

19 aFile.close();

3：基本的 Channel 示例

**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：

|  |  |
| --- | --- |
| 1 | RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw"); |

|  |  |
| --- | --- |
| 2 | FileChannel inChannel = aFile.getChannel(); |

**从FileChannel读取数据**

调用多个read()方法之一从FileChannel中读取数据。如：

|  |  |
| --- | --- |
| 1 | ByteBuffer buf = ByteBuffer.allocate(48); |

|  |  |
| --- | --- |
| 2 | int bytesRead = inChannel.read(buf); |

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**

**向FileChannel写数据**

使用FileChannel.write()方法向FileChannel写数据，该方法的参数是一个Buffer。如：

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);

}

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。

### 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。

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

这里有两个例子:

[view source](http://ifeve.com/file-channel/" \l "viewSource" \o "view source)[print](http://ifeve.com/file-channel/" \l "printSource" \o "print)[?](http://ifeve.com/file-channel/" \l "about" \o "?)

|  |  |
| --- | --- |
| 1 | **long** pos = channel.position(); |

|  |  |
| --- | --- |
| 2 | channel.position(pos +123); |

### FileChannel的truncate方法

可以使用FileChannel.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。如：

|  |  |
| --- | --- |
| 1 | channel.truncate(1024); |

这个例子截取文件的前1024个字节。

[Java NIO系列教程（三） Buffer](https://www.cnblogs.com/game-life/p/3985055.html)

1：使用Buffer读写数据一般遵循以下四个步骤：

写入数据到Buffer

调用flip()方法

从Buffer中读取数据

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

2：下面是一个使用Buffer的例子：

RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");

FileChannel inChannel = aFile.getChannel();

**//create buffer with capacity of 48 bytes**

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //向Buffer中写入数据

while (bytesRead != -1) {

//make buffer ready for read

buf.flip(); //读和写切换，切换成读模式

while(buf.hasRemaining()){

System.out.print((char) buf.get()); // read 1 byte at a time

}

buf.clear(); //make buffer ready for writing

bytesRead = inChannel.read(buf);

}

aFile.close();

3：Buffer的capacity,position和limit

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。

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。

这里有一个关于capacity，position和limit在读写模式中的说明，详细的解释在插图后面。



**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。 **写模式下，limit等于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）

4：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。

5：**从Buffer中读取数据**

从Buffer中读取数据有两种方式：

**从Buffer读取数据到Channel。**

**使用get()方法从Buffer中读取数据。**

从Buffer读取数据到Channel的例子：

int bytesWritten = inChannel.write(buf);

使用get()方法从Buffer中读取数据的例子

byte aByte = buf.get();

**向Buffer中写数据**

写数据到Buffer有两种方式：

**从Channel写到Buffer。**

**通过Buffer的put()方法写到Buffer里。**

从Channel写到Buffer的例子

|  |  |
| --- | --- |
| 1 | int bytesRead = inChannel.read(buf); //read into buffer. |

通过put方法写Buffer的例子：

|  |  |
| --- | --- |
| 1 | buf.put(127); |

6：mark()与reset()方法

通过调用Buffer.mark()方法，可以标记Buffer中的一个特定position。之后可以通过调用Buffer.reset()方法恢复到这个position。例如：

buffer.mark();

buffer.reset(); //set position back to mark.

Java NIO系列教程（六） Selector

1：为什么使用Selector?

仅用单个线程来处理多个Channels的好处是，只需要更少的线程来处理通道。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。

2：Selector的创建

通过调用Selector.open()方法创建一个Selector，如下：

Selector selector = Selector.open();

3：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。而套接字通道都可以

注意**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：

Connect Accept Read Write

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”。

这四种事件用SelectionKey的四个常量来表示：

SelectionKey.OP\_CONNECT

SelectionKey.OP\_ACCEPT

SelectionKey.OP\_READ

SelectionKey.OP\_WRITE

**SelectionKey的一些方法：**

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：

selectionKey.isAcceptable();

selectionKey.isConnectable();

selectionKey.isReadable();

selectionKey.isWritable();

Channel + Selector

从SelectionKey访问Channel和Selector很简单。如下：

[view source](http://ifeve.com/selectors/" \l "viewSource" \o "view source)[print](http://ifeve.com/selectors/" \l "printSource" \o "print)[?](http://ifeve.com/selectors/" \l "about" \o "?)

Channel channel = selectionKey.channel();

Selector selector = selectionKey.selector();

**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。

下面是select()方法：

int select()

int select(long timeout)

int selectNow()

select()阻塞到至少有一个通道在你注册的事件上就绪了。

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

selectNow()不会阻塞，不管什么通道就绪都立刻返回（译者注：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。）。

**select()方法返回的int值表示有多少通道已经就绪**。亦即，自上次调用select()方法后有多少通道变成就绪状态。如果调用select()方法，因为有一个通道变成就绪状态，返回了1，若再次调用select()方法，如果另一个通道就绪了，它会再次返回1。如果对第一个就绪的channel没有做任何操作，现在就有两个就绪的通道，但在每次select()方法调用之间，只有一个通道就绪了。

**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

Set selectedKeys = selector.selectedKeys();

Iterator keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

}

keyIterator.remove();

}

**完整的示例**

这里有一个完整的示例，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。

Selector selector = Selector.open();//打开selector

channel.configureBlocking(false);//非堵塞channel

//在selector上面注册channel

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

while(true) {

int readyChannels = selector.select();//返回有多少就绪的channel

if(readyChannels == 0) continue;//当前没有就绪的channel，继续查询

Set selectedKeys = selector.selectedKeys();//查看就绪channel中事件

Iterator keyIterator = selectedKeys.iterator();

//对事件集合，进行迭代，并处理

while(keyIterator.hasNext()) {

SelectionKey key = keyIterator.next();//得到一个事件key

if(key.isAcceptable()) {

// a connection was accepted by a ServerSocketChannel.

} else if (key.isConnectable()) {

// a connection was established with a remote server.

} else if (key.isReadable()) {

// a channel is ready for reading

} else if (key.isWritable()) {

// a channel is ready for writing

}

keyIterator.remove();//事件key处理完之后，删除

}

}

1：并发读取文件，按行读取

整体思路就是根据初始化线程数量来将文件进行分段读取，一个线程读取一个片段的内容，然后将读取到的行数据交由事先注册的处理接口来进行处理

文件读取工具类：文件读取的具体实现类，提供读取文件的入口   
/\*\*

\* 启动多线程读取文件

\*/

**public** **void** startRead(){

//使用FileChannel，可以从文件中读取数据和将数据写入文件

FileChannel infile = **null**;

**try** {

//通过RandomAccessFile打开FileChannel,并开通对filePath文件的读权限

RandomAccessFile raf = **new** RandomAccessFile(filePath,"r");

infile = raf.getChannel();

//文件总的大小

**long** size = infile.size();

//平均分给threadNum，每个线程需要读取多少内容

**long** subSize = size/threadNum;

**for**(**int** i = 0; i < threadNum; i++){

//记录每个线程从那个位置开始读，以及读取的内容大小

**long** startIndex = i\*subSize;

//最后一个线程，负责把剩余的除不尽的内容读完

**if**(size%threadNum > 0 && i == threadNum - 1){

subSize += size%threadNum;

}

//为每个线程分配一个channel，互不影响，并把内容读取到自己的channnel中去

RandomAccessFile accessFile = **new** RandomAccessFile(filePath,"r");

FileChannel inch = accessFile.getChannel();

//启动线程进行读取，从chaneel中startIndex的位置开始读取subSize大小的数据

threadPool.execute(**new** MultiThreadReader(inch,startIndex,subSize));

}

threadPool.shutdown();

} **catch** (FileNotFoundException e1) {

e1.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **finally**{

**try** {

**if**(infile != **null**){

infile.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}